Untitled Goose Framework

Framework per lo sviluppo di giochi da tavolo simili al Gioco dell'Oca

Samuele Burattini
samuele.burattini@studio.unibo.it

Andrea Cardiota
andrea.cardiota@studio.unibo.it

Luca Deluigi
luca.deluigi3@studio.unibo.it

Francesca Tonetti
francesca.tonetti@studio.unibo.it

Settembre 2020

Capitolo 1

Processo di sviluppo adottato

In questo capitolo si presenta la metodologia scelta nello sviluppo di questo progetto, mostrando gli aspetti chiave e gli strumenti che il gruppo ha utilizzato nella delineazione degli obiettivi e nella divisione del lavoro durante tutto l'arco di tempo impiegato per la realizzazione.

In particolare il processo di sviluppo adottato per questo progetto è stato scelto ispirandosi a **SCRUM**, come consigliato dalle specifiche, per poter conoscere e sperimentare l'approccio iterativo nella gestione di un lavoro in gruppo. I membri hanno quindi impersonato oltre che il team di sviluppo stesso anche i committenti del progetto e le figure tipiche del processo SCRUM: il product owner (Andrea Cardiota) e lo SCRUM master (Samuele Burattini). La valutazione del processo di sviluppo sarà approfondita nella sezione 7.1.

1.1 Definizione degli obiettivi

Nei primi incontri relativi al progetto si è scelto di fare un paio di riunioni intensive per delineare i requisiti di massima del sistema, gli obiettivi da raggiungere a livello funzionale oltre che un abbozzo del modello che potesse rappresentare il dominio.

In questa prima fase, che ha coperto circa la prima settimana di lavoro, il gruppo si è immedesimato nei committenti per poter scrivere una proposta di progetto sufficientemente dettagliata. É stato necessario raggiungere un compromesso tra delle specifiche chiare per poter iniziare da subito a lavorare ed una sufficiente genericità per includere le possibili espansioni emergenti durante lo sviluppo.

Oltre ai requisiti funzionali il gruppo ha cercato di individuare anche degli obiettivi da mantenere durante tutto lo sviluppo, primo fra tutti il rilascio di un prodotto sempre eseguibile a partire da una base prototipale e arricchito in modo incrementale di funzionalità al termine di ogni *sprint*.

1.2 Pianificazione e organizzazione del lavoro

In totale lo sviluppo è stato suddiviso in otto sprint di lavoro. In particolare i primi due sono stati della durata complessiva di due settimane ciascuno, mentre successivamente si è adottata una suddivisione in slot temporali più brevi, della durata di circa una settimana, per venire incontro alle

esigenze sia legate allo stato del progetto sia del team. Infatti, il gruppo ha valutato più produttivo avere obiettivi concreti e delineati con più precisione e con frequenza maggiore.

Lo strumento utilizzato per coordinare le attività è stato Trello [1], una bacheca online dove poter appuntare dei task, decorarli con etichette, utilizzando una codifica a colori, e identificare a colpo d'occhio diverse categorie, ordinare per priorità e assegnare determinati task ai membri del gruppo impostando scadenze se necessario.

Si è scelto inoltre di mantenere il backlog di ogni sprint salvato sulla bacheca per avere una istantanea del lavoro svolto durante ogni ciclo. Su Trello sono stati anche mantenuti gli artefatti di documentazione del processo SCRUM, nello specifico i goal, le review e le retrospective di ogni sprint svolto. Solitamente la modalità di riunione è stata per via telematica, in chiamata tramite Discord [1].

1.3 Strumenti di controllo della qualità

Per ottimizzare i tempi di sviluppo si è deciso di dare spazio ad una prototipazione agevole e di non applicare la filosofia stringente del Test Driven Developement (TDD) affrontata durante il corso. Tuttavia sono stati realizzati **unit tests** di quasi ogni componente autonomo. I test, sia automatizzati che manuali, sono il principale strumento di controllo della qualità e della funzionalità del codice prodotto.

A riprova dell'importanza attribuita ai test, si è scelto di utilizzare degli strumenti di **Continuous Integration** (CI) per garantire che ad ogni push sul progetto venissero eseguiti tutti test presenti nel repository e che un eventuale esito negativo venisse segnalato al team tramite email e sulla chat Discord del progetto.

Come strumento di CI si è scelto di utilizzare **Github Actions** in modo da poter visualizzare tutte le informazioni relative allo stato del progetto sulla pagina del repository, hostato su Github. L'esecuzione dei test è stata automatizzata con `sbt` in quanto, essendo un progetto Scala, si è scelto di gestire sia le dipendenze che i test tramite questo strumento. Per generare i file di configurazione delle Actions è stato usato un plugin per sbt, chiamato `sbt-github-actions` [2].

La piattaforma di testing utilizzata è FlatSpec per la sua eccellente integrazione con il linguaggio Scala e la leggibilità del codice prodotto.

Oltre al testing automatizzato i momenti già citati di review e retrospective sono stati particolarmente utili per valutare i risultati sia a livello qualitativo che a livello tecnico. Inoltre questi strumenti del processo SCRUM si sono rivelati utili per riflettere insieme sull'andamento del progetto come team e dare consigli ai diversi membri su come migliorare il codice già prodotto o su quali punti fare attenzione in vista dello sprint successivo.

[1] Discord - `https://discord.com/`

Capitolo 2

Analisi dei Requisiti

Nel capitolo corrente viene presentata l'analisi dei requisiti svolta dal gruppo nella prima fase del progetto. Questi ultimi descrivono in modo chiaro, consistente e completo, ciò che i committenti (e quindi i membri del gruppo stesso) si aspettano di ottenere al termine del tempo di sviluppo.

Il progetto consiste nella realizzazione di un framework che permetta di modellare qualsiasi variante fantasiosa di giochi da tavolo basati sui concetti di percorso e turni e quindi simili al classico Gioco dell'Oca. La volontà non è quella di ricalcare necessariamente versioni commerciali esistenti, bensì consentire al creatore del gioco di spaziare con la creatività introducendo nuove regole, nuove azioni attive per ciascun giocatore oltre che la modellazione di aspetti grafici del gioco. Per questo motivo il sistema dovrà adempiere a diversi casi d'uso in base all'attore che vi interagisce: giocatore o creatore del gioco.

2.1 Requisiti Funzionali

É possibile scorporare gli aspetti di interesse in due categorie: requisiti di sviluppo dei giochi e requisiti per l'esperienza di gioco.

Rispettivamente i primi riguardano la programmazione di nuovi giochi da tavolo, mentre i secondi riguardano le interazioni di un giocatore (non necessariamente un programmatore) con il sistema in esecuzione.

Di seguito viene raffigurato il diagramma dei casi d'uso del sistema preso in esame con annessa gerarchia tra gli attori [2.1].

2.1.1 Requisiti per la programmazione di giochi

L'obiettivo principale del sistema è quello di creare/definire dei giochi da tavolo in modo semplice, con il fine ultimo di poterli utilizzare su una qualsiasi macchina, in un ambiente auto-generato. Si richiede di fornire all'utente delle astrazioni di modello, un motore di esecuzione del gioco e un'interfaccia grafica auto-generata ma personalizzabile a seconda dei termini e vincoli del gioco che si vuole definire.

Un obiettivo fondamentale, quindi, è quello di arrivare a realizzare un intero gioco da tavola in breve tempo, scrivendo del codice facilmente leggibile, manutenibile e poco complesso. Il program-

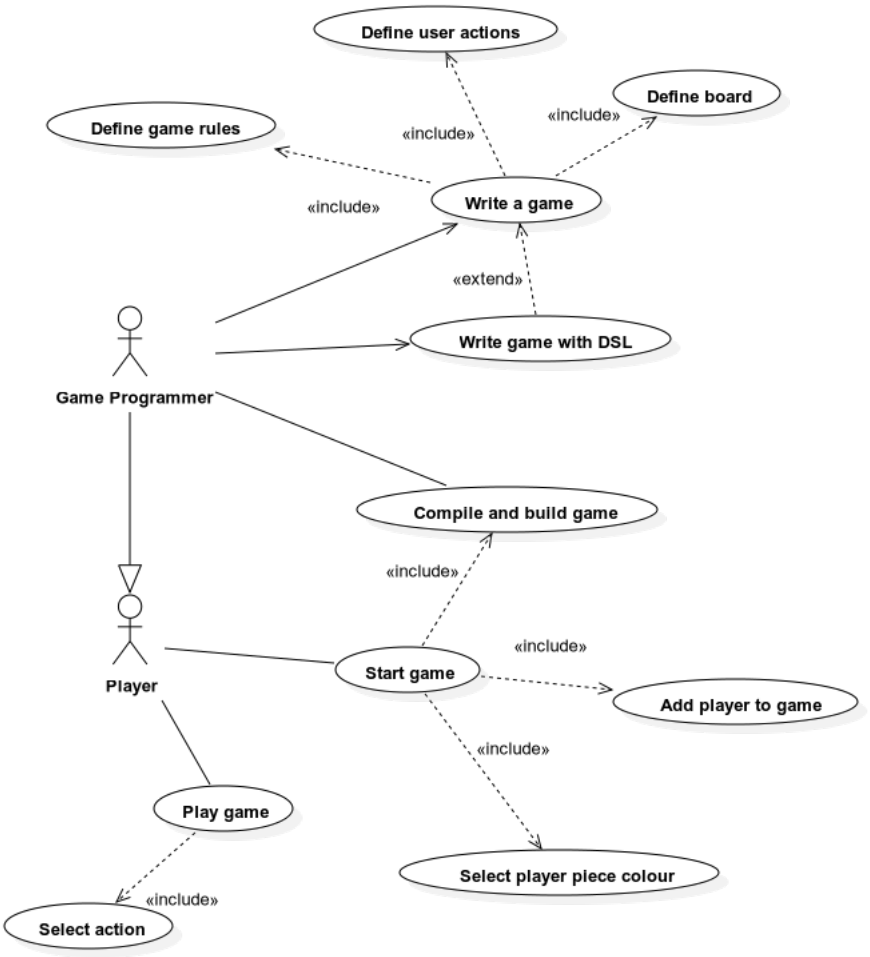

Figura 2.1: Diagramma dei casi d'uso - UML.

matore potrà concentrarsi sulla logica del gioco dato che gli aspetti di più basso livello sono gestiti dal framework stesso.

Per raggiungere questo obiettivo e rendere possibile l'utilizzo del framework anche ai programmatori meno esperti, nonché per la leggibilità del codice dei giochi, si è scelto di realizzare un *Domain-Specific Language* (DSL).

Nella fase iniziale di definizione sarà data la possibilità al creatore di impostare:

- la conformazione del tabellone di gioco;
- il comportamento del gioco, mediante regole specifiche attivate in relazione a determinati avvenimenti;
- le azioni che il giocatore può svolgere durante il proprio turno.

Per quanto riguarda la visualizzazione grafica del gioco, anche questa sarà configurabile in base a ciò che si vuole costruire, lasciando il sistema quanto più libero possibile. Sarà possibile specificare colori e immagini per le caselle.

2.1.2 Requisiti per i giocatori

Complessivamente l'obiettivo è quello di permettere a ciascun giocatore di comprendere velocemente l'interfaccia, in modo intuitivo e semplice, nonché di giocare al gioco e di godere un'esperienza piacevole. La volontà è quella di agevolare il giocatore nel corso della partita, calcolando automaticamente le azioni possibili in base allo stato e gestendo sempre in maniera automatica eventi e regole che ne descrivono le conseguenze.

In ogni momento dovrà essere possibile visualizzare il proprio stato e quello degli avversari nella partita in corso. Ogni giocatore sarà riconoscibile attraverso pedine aventi colori differenti, scelti dai giocatori stessi, insieme al proprio nickname. Ciascuno dovrà avere la possibilità di conoscere sempre la propria posizione sul tabellone di gioco e poterlo navigare per vedere l'intero percorso.

2.2 Requisiti non funzionali

Per ciò che concerne i requisiti non funzionali, essi si concentrano principalmente sulle tecnologie da adottare nello sviluppo. Come richiesto dalle specifiche di progetto, un requisito è l'utilizzo di Scala come linguaggio di programmazione. Il team si è imposto l'utilizzo di ScalaFX [3] come libreria grafica. ScalaFX consiste infatti in un DSL che richiama metodi e classi di JavaFX, costituendone un *wrapper* che offre delle funzionalità aggiuntiva.

Capitolo 3

Analisi del Dominio

Il gioco dell'oca è un esempio di gioco da tavolo a turni con un tabellone a percorso univoco suddiviso in caselle. Il framework da realizzare permetterà di codificare e giocare qualsiasi gioco da tavolo di questo tipo, definendone regole e caratteristiche. L'analisi del dominio è il frutto di numerose iterazioni, a partire dalla progettazione effettuata nei primi sprint. Il modello costruito inizialmente è stato ripetutamente rivisto e migliorato fino ad arrivare all'attuale conformazione. Tale modello ha guidato il design dell'applicazione durante tutto lo sviluppo del progetto, in linea con la metodologia **Domain Driven Design** (*DDD*).

3.1 Il gioco da tavolo

Un gioco da tavolo di questa tipologia è descrivibile a livello statico, a priori, e a livello dinamico, ovvero descrivendo cosa ne compone lo stato in un dato momento, dal punto di vista del progettista del software, a runtime. Il componente che si occupa di eseguire la logica di gioco è detto **Engine**. Questo componente ottiene dalle regole una sequenza di operazioni da eseguire, sequenza che può crescere e decrescere ma che è destinata a svuotarsi per un gioco ben programmato. Tali operazioni sono mantenute all'interno di una pila, chiamata **Stack di gioco**, o **Game stack**.

3.1.1 Punto di vista statico: la definizione di un gioco

A priori, di un gioco, è nota la conformazione del tabellone (numero di tessere, o caselle, i relativi effetti, la disposizione), il regolamento (che include, ad esempio, la condizione di vittoria e le regole che descrivono le azioni possibili durante un turno) e altri aspetti accessori (ad esempio aspetti grafici, minimo numero di giocatori, età minima per giocare...). Le entità più importanti sono state formalizzate tramite le seguenti definizioni:

- **Board** - Una *board* definisce le caratteristiche del tabellone di gioco. Contiene un insieme di *tile*, un ordinamento sequenziale di quest'ultime e una *disposition*.
- **Tile** - Una *tile* è una tessera della *board*. Può avere un numero o un nome, e un insieme di gruppi di cui fa parte insieme ad altre *tile* simili.

- **Disposition** - Una *disposition* è il modo in cui le tessere si dispongono sul tabellone di gioco. Può essere rappresentata da una funzione che associa una *tile* ad una posizione espressa in coordinate nel piano cartesiano.
- **Game** - Un *game* è un template per istanziare partite. Contiene una *board* e le regole del gioco. Tra queste:
  - **PlayersRange** - Minimo e massimo di giocatori permessi.
  - **PlayerOrdering** - Politica di ordinamento dei giocatori. Può essere fissata dall'utente, casuale per il primo turno e fissata dal secondo o sempre casuale (ad ogni turno gioca un giocatore a caso).
  - **Insieme di ActionRule** - Sono le regole che riguardano le azioni disponibili all'utente di turno.
  - **Insieme di BehaviourRule** - Sono le regole del gioco che riguardano degli eventi e/o la modifica dello stato.
  - **Insieme di CleanupRule** - Sono le regole che vengono chiamate alla fine della risoluzione di una sequenza di eventi completa, solitamente si occupano di effettuare controlli o di modificare lo stato in un momento di "quiete", ad esempio alla fine del turno.
- **Operation** - Un'operazione da eseguire sullo stato. Tra un'operazione e la successiva vengono rivalutate tutte le regole per permettere la loro attivazione. L'esecuzione delle *operations* è demandata all'engine. Alcune operazioni sono trattate in modo specializzato dall'engine, come ad esempio la creazione di un *dialog* per comunicare con i giocatori, che mette in pausa la risoluzione dello stack fino ad una risposta.
- **Rule** - Una regola del gioco. In generale ogni regola appartiene esclusivamente ad una delle seguenti categorie:
  - **ActionRule** - Una regola che determina la disponibilità o la negazione di disponibilità di una o più azioni per l'utente di turno, in base allo stato del gioco attuale.
  - **BehaviourRule** - Una regola che, dopo aver valutato lo stato corrente del gioco, lo modifica direttamente e lancia una sequenza di *operations* che l'engine eseguirà successivamente.
  - **CleanupRule** - Una regola simile alla *BehaviourRule* che però viene valutata esclusivamente alla fine della risoluzione di uno stack di operazioni. Sono utili per gestire cambi di stato da effettuare esclusivamente in momenti di quiete, ovvero quando lo stack è vuoto, come ad esempio la fine di un turno.
- **Dice** - Un dado con un numero arbitrario di facce, ognuna rappresentante un oggetto arbitrario.

Le entità che fanno parte del gruppo principale statico sono chiamate **Definitions** delle rispettive entità a tempo di esecuzione.

3.1.2 Punto di vista dinamico: l'istanza di un gioco in esecuzione

Dato che una regola può potenzialmente riguardare un qualunque aspetto reale della partita è necessario creare un modello quanto più vicino alla realtà possibile. Le entità più importanti sono state formalizzate tramite versioni *stateful* delle definitions, aggiungendo le seguenti entità:

- **GameEvent** - Un evento di gioco. Lo scope di questo tipo di eventi è globale, ovvero non riguarda né un particolare giocatore né una particolare tessera di gioco.
- **PlayerEvent** - Un evento di gioco riguardante un singolo giocatore.
- **TileEvent** - Un evento di gioco riguardante un singolo tile.
- **GameState** - Una raccolta di dati che rappresentano parte dello stato della partita. Tra questi:
  - **CurrentTurn** - Il turno corrente.
  - **Player** - Un giocatore della partita. Lo stato mantiene salvato il giocatore di turno.
  - **Cycle** - L'indice del ciclo di risoluzione stack corrente.
  - **PlayerPieces** - Una mappa che associa ad ogni giocatore della partita la sua pedina, o *Piece*.
  - **Piece** - Una pedina del gioco, mantiene la propria posizione sul tabellone e il proprio colore.
  - **ConsumableBuffer** - Un buffer che contiene gli eventi scaturiti durante il ciclo corrente. Le regole controllano questo buffer per cambiare il proprio comportamento.
  - **GameHistory** - Lo storico degli eventi globali e persistenti.
  - **PlayerHistory** - Lo storico, relativo ad un singolo *player*, dei *PlayerEvent* accaduti per quel *player* particolare.
  - **TileHistory** - Lo storico, relativo ad un singolo *tile*, dei *TileEvent* accaduti per quella *tile* particolare.
- **DialogContent** - Dei dati che permettono la creazione e la visualizzazione di dialoghi pop-up.

Capitolo 4

Design Architetturale

Il framework è composto da due sottosistemi software: l'applicazione e il DSL [4.1]. L'applicazione ha valenza autonoma mentre il DSL rappresenta solamente una sintassi più leggibile per creare una definizione di gioco e, successivamente, un'istanza di gioco, se vengono superati dei controlli aggiuntivi e automatici sul modello. Questa separazione distingue i sottosistemi non solo per le diverse responsabilità ma anche a livello temporale. In particolare, l'esecuzione di un gioco definito tramite DSL è divisa in due fasi:

1. il parser o compilatore Scala e il model checking implementato per l'albero sintattico si occupano di interpretare la definizione del gioco e di costruire un modello, ovvero una definizione di gioco da tavolo valida;

2. viene generata l'applicazione che permette agli utenti di giocare istanziando una partita a partire dalla definizione di gioco data.

4.1 L'applicazione

Il design architetturale dell'applicazione segue il pattern Model-View-Controller (MVC) [4.2]. Il model descrive le entità del dominio, quali ad esempio le tessere, la tavola da gioco, i giocatori, le regole, le azioni possibili o gli eventi. La view è in grado di rappresentare il model in maniera grafica all'utente, mentre il controller si occupa di eseguire ed applicare la logica di gioco e di gestire l'asincronia e altri aspetti di interazione tra model e view. Sia la view che il controller hanno infatti un proprio flusso di controllo, indipendente.

Come osservabile nel diagramma di figura 4.2, il model è interamente indipendente dal resto dei componenti. Le entità del model vengono *usate* dall'engine per la logica delle regole, dello stack e delle operazioni speciali, mentre viene *visualizzato* dal controller della view, l'*ApplicationController*. L'engine utilizza l'interfaccia *ViewController* per inviare aggiornamenti sullo stato alla view, mentre implementa o usa un *EventSink* per ricevere in modo asincrono qualsiasi evento di gioco (*GameEvent*). L'ApplicationController implementa ViewController, si occupa della gestione degli aspetti grafici e dell'input dell'utente, infatti implementa anche l'interfaccia *InputManager*.

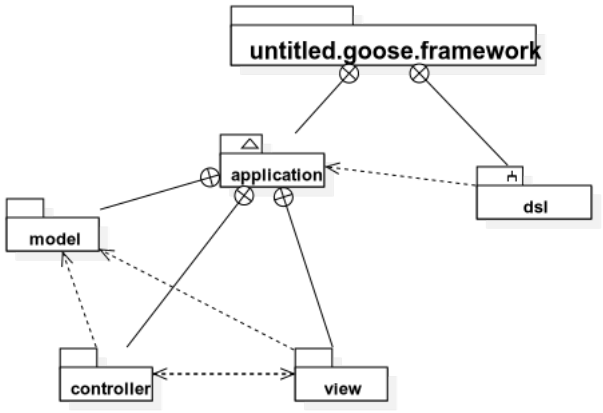

Figura 4.1: Diagramma di massima dell'architettura del framework - Diagramma dei package, UML.

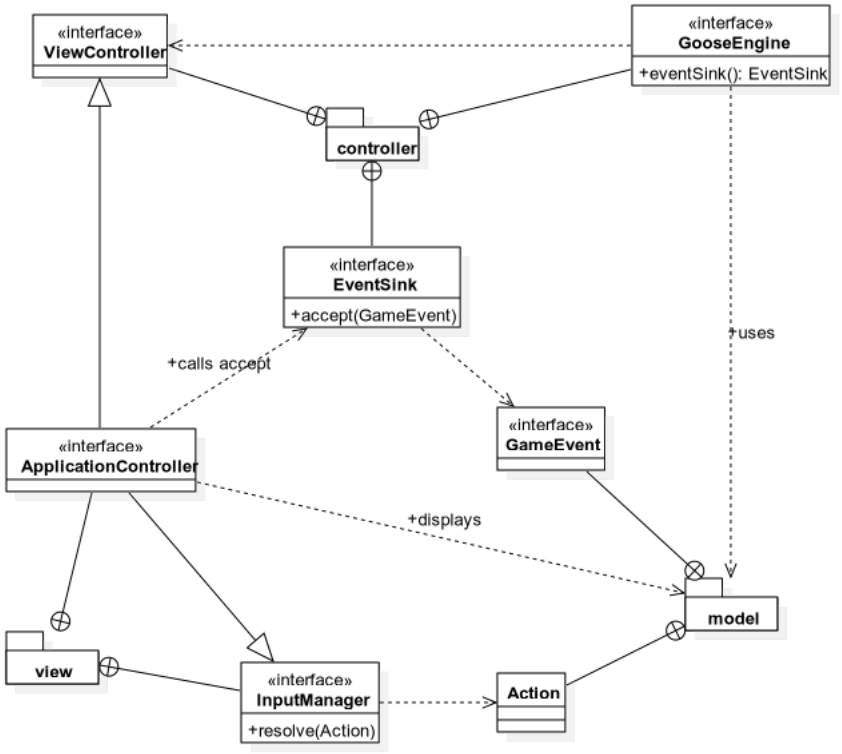

Figura 4.2: Diagramma architetturale dell'applicazione - UML.

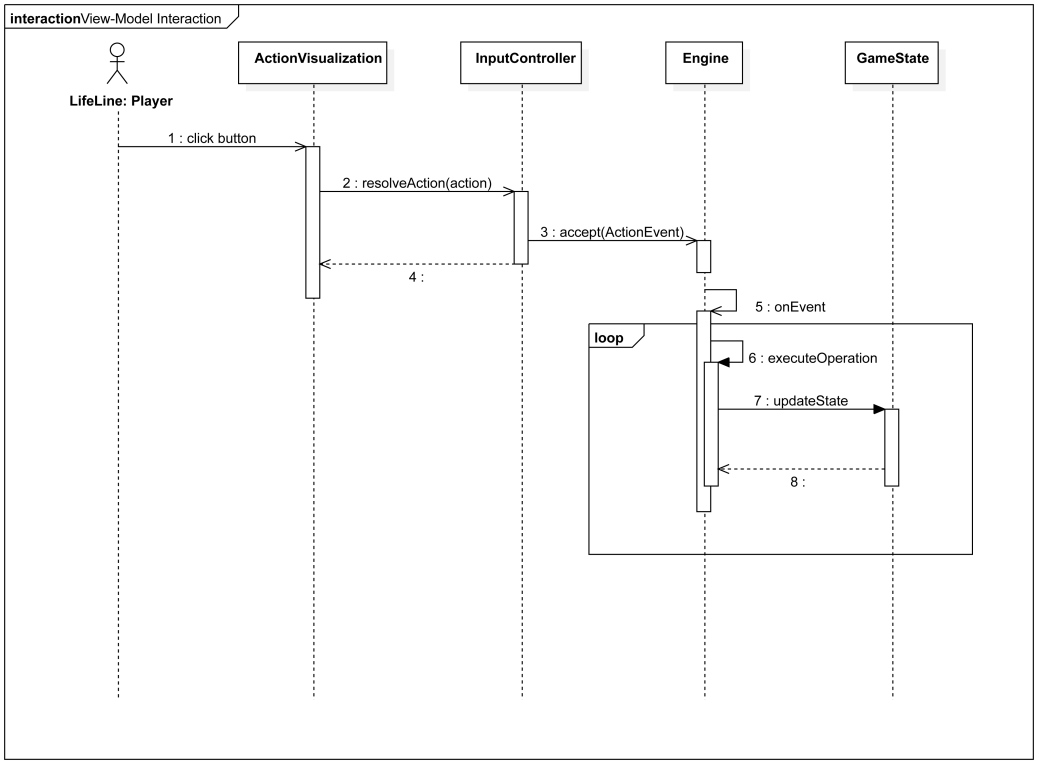

Figura 4.3: Diagramma di sequenza che mostra il flusso di informazione che parte da una interazione dell'utente fino ad una modifica dello stato di gioco nel modello.

Interazione view-model Il diagramma 4.4 mostra come sia possibile per l'utente interagire con il modello attraverso l'interfaccia grafica. Nell'interfaccia sono presenti dei bottoni *ActionVisualization* che al click da parte dell'utente fa una serie di chiamate che portano all'invio asincrono della azione desiderata all'Engine.

L'engine è poi in grado di interpretare le operazioni relative all'azione inviata e modificare di conseguenza lo stato del gioco.

Interazione model-view Il diagramma 4.4 mostra come il modello è in grado di interagire con l'interfaccia grafica per mostrare messaggi agli utenti e aggiornare lo stato del gioco.

L'Engine è in grado di interrogare l'istanza corrente del Game per ottenere le operazioni da svolgere in un determinato istante. Tra queste operazioni sono presenti alcune speciali DialogOperation che l'Engine è in grado di riconoscere ed interpretare per inviare alla view dei messaggi e sospendere la risoluzione dello stack.

Alla risoluzione di ogni operazione, inoltre, l'Engine invia lo stato aggiornato alla view per farlo visualizzare ai giocatori.

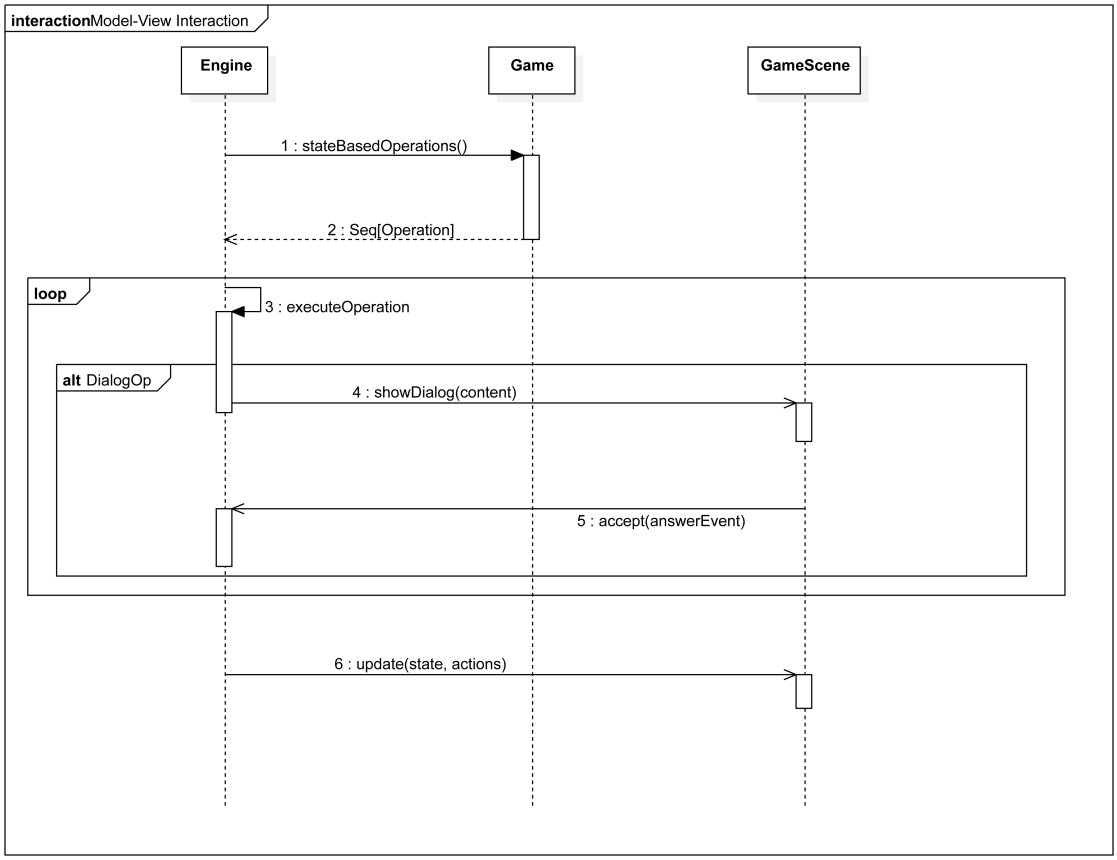

Figura 4.4: Diagramma di sequenza che mostra il flusso di informazione per comunicare dal model verso la View attraverso l'Engine

4.2 Il DSL

Il DSL consiste in un insieme di dichiarazioni e classi di utilità con lo scopo principale di permettere di definire un intero gioco utilizzando una sintassi quanto più vicina possibile al linguaggio naturale. Le tipologie principali di classi che formano il DSL sono *parole* e *nodi*.

Le **parole** assomigliano a dei pezzi di un puzzle in grado di comporre una frase. Ogni frase, a tempo di esecuzione, andrà ad aggiungere ad un albero astratto dei **nodi** contenenti le informazioni specifiche indicate dall'utente, in modo equivalente alla creazione di un *Abstract Syntax Tree*. Spesso la catena di parole per comporre una frase si occupa di raccogliere informazioni utili per la creazione del nodo e di trasferirle verso la parola finale, ultimo anello della catena, che si occupa di creare il nodo e di aggiungerlo all'albero.

In una seconda fase avviene il *Model Checking*, ovvero il controllo della correttezza dell'albero sintattico di nodi, che si conclude con l'eventuale generazione automatica di messaggi per l'utente che ha commesso qualche errore di programmazione. Questa funzionalità ha guidato molte decisioni di design strutturale del linguaggio in quanto è considerata fondamentale per la realizzazione di un buon DSL.

Se il Model Checking viene completato senza errori viene avviata la generazione della definizione del gioco e in seguito ne viene istanziata una versione eseguibile e immediatamente giocabile, analogamente a come avviene all'interno di un compilatore. L'aver scelto di creare un DSL invece di un intero compilatore ha risparmiato l'implementazione di un parser, delegata al compilatore di Scala.

Capitolo 5

Design di dettaglio

In questo capitolo vengono discusse le scelte principali che hanno guidato lo sviluppo dei componenti del framework e l'organizzazione generale del software.

5.1 Applicazione

Per quel che riguarda l'applicazione, come già identificato nel design architetturale 4.1 si descrivono nel dettaglio i tre componenti principali del sistema affrontando le sfide tecniche incontrate nella progettazione e le soluzioni proposte.

5.1.1 Modello

Il modello dell'applicazione è stato costruito sulla base dell'analisi del dominio, per poi essere iterativamente arricchito a seguito di nuove fasi di progettazione e numerosi *rework*. Il risultato attuale, aggiornato alla versione 0.9.0, è visibile in figura 5.1. La numerosità dei componenti, delle interfacce e delle classi è talmente elevata da rendere il diagramma difficile da leggere, si consiglia di usare la ScalaDoc. Ciononostante, nel diagramma in figura non sono rappresentate le classi, ad eccezione delle più importanti.

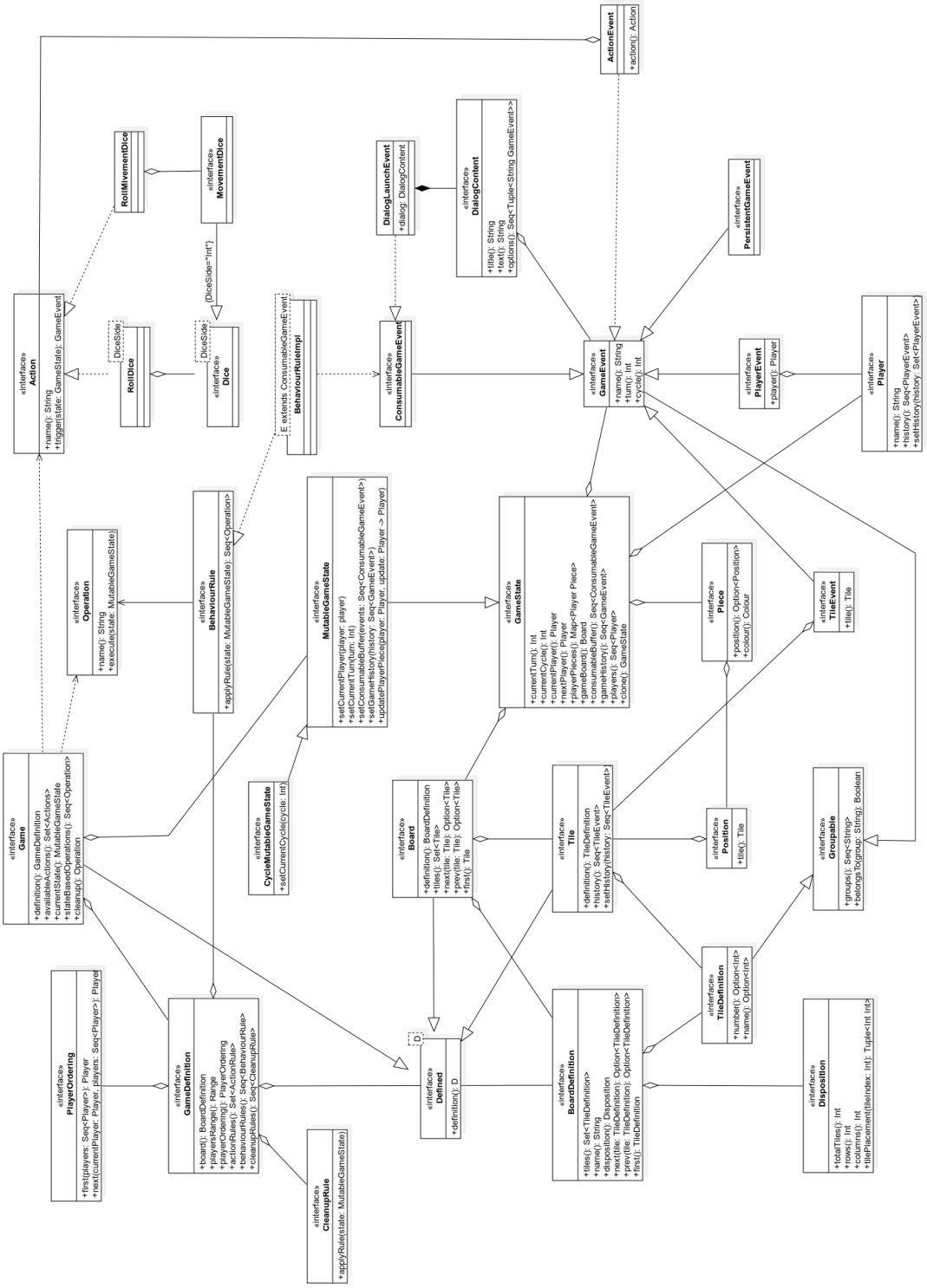

Figura 5.1: Diagramma delle classi più importanti del modello - UML.

Il model attuale permette di codificare praticamente qualsiasi gioco che rientri più o meno bene nello schema del gioco a percorso senza bivi. Le history dei giocatori, dei tile e la history globale permettono il salvataggio di eventi offerti dal model o creati dall'utente, costituendo di fatto un modo per arricchire lo stato del gioco di informazioni arbitrarie Ad esempio, in un gioco in cui ciascun giocatore possiede delle monete, è possibile codificare la somma posseduta salvando nella history di ogni player i guadagni e le spese, come eventi.

Entità statiche e dinamiche Nel modellare le entità del gioco è emersa ulteriormente la doppia natura, statica e dinamica, dei componenti del gioco, in particolare in riferimento alle tessere che compongono la Board, alle pedine di gioco e le relative posizioni e più in generale allo stato del gioco complessivo.

Le tessere della Board sono state rappresentate da TileDefinition statiche contenenti le informazioni relative al numero e/o al nome di ciascuna tessera. Nel momento in cui viene creata una nuova istanza del gioco ciascuna di queste viene sostanzialmente corrisposta da una Tile, che introduce la possibilità di mantenere eventi in uno storico. Tali eventi sono eventualmente valutabili dalle regole per determinare il comportamento.

Per quanto riguarda le pedine o Piece queste vengono create all'inizio della partita e mantenute accoppiate a ciascun Player all'interno del GameState. Tra le informazioni riguardanti ciascun Piece, oltre alla proprio colore, viene specificato come opzionale il valore della posizione. Se la posizione risulta *None* significa che la pedina si trova fuori dalla Board, ad esempio prima del primo lancio di dadi, oppure dopo un'eventuale eliminazione del giocatore.

I Player vengono istanziati dinamicamente all'interno del gioco e tra le informazioni vi è sempre uno storico di eventi, ovvero una sequenza di PlayerEvent che riguardano quel giocatore.

Stato del gioco Tra le entità dinamiche che permettono di specificare lo stato del gioco ritroviamo gli oggetti GameState e Game.

Il Game rappresenta l'istanza della partita corrente, e oltre a mantenere lo stato corrente del gioco si occupa di incapsulare il RuleSet, offrendo delle operazioni per interrogarlo. Lo stato del gioco mantiene la posizione aggiornata dei Player attraverso i Piece. Oltre a questo contiene i buffer degli eventi consumabili e la *history* permanente del gioco che possono essere interrogati per stabilire il comportamento.

Lo stato è per sua natura mutabile quindi è stato implementato come tale, ma, allo stesso tempo, viene esposta una sua versione immutabile per impedire la modifica da parte dei componenti che devono utilizzarlo in sola lettura.

Eventi Come individuato dall'analisi del dominio è semplice modellare gli avvenimenti durante una partita come eventi che possono scatenare delle risposte sulla base delle regole. Ciascun evento può servire sia per gestire le regole nel ciclo corrente, sia come avvenimento o informazione salvata in uno storico permanente. Le due funzioni non sono esclusive: questo ha portato alla progettazione di due tipologie di eventi derivanti da una stessa interfaccia GameEvent:

- ConsumableGameEvent sono eventi che nascono come consumabili e quindi in grado di attivare comportamenti; sono salvati nel buffer dei consumabili;
- PersistentGameEvent sono eventi che nascono per essere inseriti nella storia del gioco; saranno leggibili anche dopo la fine del ciclo corrente, e di fatto costituiscono una parte fondamentale dello stato dell'entità a cui fanno riferimento, come ad esempio una tessera o un giocatore.

Gli eventi consumabili possono quindi essere mantenuti anche dopo essere stati usati dalle regole, entrando nella storia permanente del gioco; per fare ciò è sufficiente aderire ad entrambe le interfacce.

Per semplificare la navigazione della storia di gioco gli eventi relativi a giocatori o tessere del tabellone vengono inserite anche nei rispettivi componenti Player e Tile. Per specificare che un evento riguarda queste entità sono stati introdotti dei *Mixins* che aggiungono i campi per referenziare l'oggetto di riferimento.

Azioni disponibili All'interno del model è possibile definire anche le azioni ovvero quelle componenti che permettono di specificare come ogni Player può agire durante il proprio turno. A ciascuna azione viene assegnato un nome che consente di identificarla e sarà lo stesso visualizzato dal Player di turno. Inoltre, la scelta di una specifica azione permette di lanciare un determinato evento basato sullo stato corrente del gioco.

Le regole delle azioni producono ActionAvailability in base allo stato. Queste ultime vengono valutate secondo un ordine differente in quanto ciascuna ActionAvailability si compone di una tripla contenente l'azione a cui fa riferimento, un permesso espresso come variabile booleana che specifica se potrà essere eseguita o viceversa, e il valore della priorità dell'azione stessa come valore intero. Per ogni azione quindi verrà scelta la availabilty con priorità maggiore tra tutte quelle prodotte dalle ActionRule.

Regole del comportamento di gioco All'interno dei giochi sviluppati tramite il framework, è possibile definire il comportamento attraverso tre specifici tipi di regole: CleanUpRule, BehaviourRule ed ActionRule.

In particolare, le regole di CleanUp si occupano di determinare cosa fare ogni volta che un Cycle sarà terminato. Ad esempio, *TurnEndConsumer* è una regola di tipo CleanUp, che si andrà ad occupare di controllare che un certo turno, per un certo Player, sia effettivamente terminato; nel caso fosse così, consumerà il suddetto evento e terminerà il turno, consegnando il testimone al giocatore successivo.

Per quanto riguarda le regole di Behaviour, ciascuna di queste agisce direttamente sul buffer degli eventi consumabili rimuovendo gli eventi che l'hanno attivata, allo scopo di non essere ripetuta infinitamente all'interno di un ciclo. Ogni Cycle infatti termina quando non sono più presenti operazioni da compiere e il regolamento viene interrogato ogni volta che viene eseguita una operazione per valutare se è necessario risolverne altre prima di proseguire con lo Stack corrente. In alcuni casi è possibile voler definire regole che non consumano gli eventi di attivazione, a tal fine è comunque necessario che ci sia almeno una regola successiva se ne occupi.

Il loro comportamento consiste nel controllare che alcune condizioni si verifichino e, solo in quel caso, consumare l'evento corrispondente dal buffer, per poi eseguire l'operazione effettiva che può a sua volta fare una tra queste due cose:

- lanciare uno o più eventi di gioco;
- alterare lo stato di gioco, di un giocatore o di una tessera.

Altro punto fondamentale riguarda l'ordinamento delle regole. Quest'ultime, infatti, sono ordinate dall'utente in base all'ordine della risoluzione per i motivi sopra spiegati.

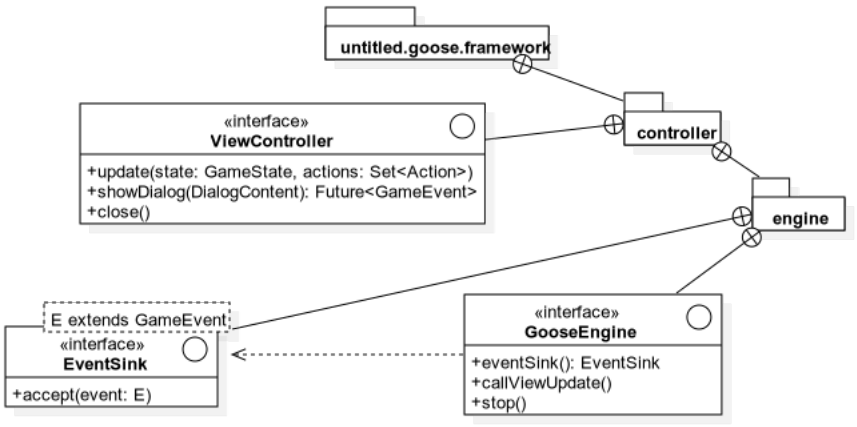

Figura 5.2: Diagramma delle interfacce e dei package del controller - UML.

Aspetto grafico del gioco Di fondamentale importanza è il *GraphicDescriptor*, un oggetto utilizzato per descrivere le proprietà grafiche che l'utente vuole che il gioco abbia. In particolare, sfruttando quest'oggetto, è possibile definire il colore e l'immagine di sfondo di ogni singola Tile all'interno del gioco in maniera semplice e veloce, riferendosi alle Tile con il loro nome, il numero che ne identifica la posizione o, in alternativa, un gruppo di appartenenza facendo in modo che le diverse proprietà grafiche si combinino secondo delle regole di precedenza, dalla più specifica alla più generale.

5.1.2 Controller

Il controller dell'applicazione è basato sul concetto di Event Loop.

In particolare il *GooseEngine* accetta, in maniera del tutto asincrona e non bloccante, qualsiasi tipo di evento considerabile un evento di gioco (*GameEvent*). In teoria sarebbe persino possibile creare un evento "Ha battuto le mani" inviato da un sensore di battiti di mani all'engine in tempo reale quando rileva un giocatore battere le mani. L'engine metterà in coda l'evento e appena sarà possibile valuterà l'arrivo di questo evento rispetto alle regole del gioco e se le regole sono definite correttamente per gestire i battiti di mani allora il gioco reagirà anche ad essi. Per la prima versione del framework tuttavia l'unico evento asincrono che viene inviato all'engine è un evento speciale, un *ActionEvent*, che determina l'azione scelta dall'utente fra quelle disponibili.

La progettazione del controller, in linea con il resto del framework, è stata effettuata e rivista iterativamente, in alternanza a sessioni di sviluppo. Quello visibile in figura 5.2 è il risultato finale (versione 0.9.0 o superiore).

5.1.3 View

Come detto in precedenza, la View è stata realizzata sfruttando le potenzialità offerte dalla libreria ScalaFX. Quest'ultima ha di fatto permesso di modellare la view in maniera semplice e modulare, sfruttando le potenzialità offerte dal paradigma funzionale.

La classe principale View si occupa di ricevere il template del Gioco da istanziare generato dal DSL e si occupa di lanciare nell'ordine:

1. L'**IntroMenu**, che contiene e raccoglie le informazioni necessarie prima dell'avvio di una partita, ovvero quelle relative ai giocatori e alle pedine a loro associate, indicando i giocatori nell'ordine in cui devono susseguirsi.

2. La **GameScene**, che contiene la vista della board e dello stato della partita in corso, oltre a un menù da cui gli utenti possono selezionare le azioni da svolgere nel proprio turno tra quelle permesse. E' reso disponibile anche un logger testuale contenente informazioni dettagliate su ogni evento avvenuto nella partita in corso.

5.2 Domain Specific Language

5.2.1 Sintassi

Il punto di ingresso del sistema per supportare la sintassi è stato realizzato interamente con un trait contenente le definizione di valori e metodi con i quali costruire l'inizio delle frasi del DSL, da estendere come si estenderebbe `App`. Questo poi delega ad altri trait la gestione delle parti più specifiche sfruttando il meccanismo dell'ereditarietà, per organizzare il tutto in modo più semplice.

Ogni parola "iniziale" internamente definisce i metodi per le successive parole che completano la frase, spesso supportando sintassi diverse, talvolta riutilizzando parole chiave o introducendo sinonimi.

5.2.2 Model Checking

La struttura che rappresenta il modello di albero astratto adottato nel DSL è rappresentata nella Figura 5.3.

É chiaramente visibile come il RuleBook è un punto centralizzato che rappresenta la radice di un albero di nodi specializzati.

Sono presenti quindi diverse collezioni, e il nodo che funge da radice e le racchiude è in grado di interpretare il contenuto e valutarne la correttezza. In particolare i due macro-nodi figli del RuleBook sono:

- Il BoardBuilderNode, nodo che si occupa di costruire la definizione della Board

- Il RuleSetNode, nodo che contiene la definizione di tutte le regole esse siano riguardanti le azioni possibili o i comportamenti in risposta agli eventi.

Per supportare un meccanismo di messaggi automatici per l'utente si è scelto di far ritornare ad ogni nodo dell'albero una sequenza di stringhe che rappresentassero gli errori. In questo modo è il nodo stesso, o il suo genitore, a generare l'errore in quanto capace di auto-controllare la correttezza. Se la sequenza restituita da tutti i nodi è vuota allora il modello viene considerato corretto.

Nel descrivere il gioco è consigliato suddividere il file secondo un ordine logico per facilitarne la lettura e la comprensione, descrivendo per prima prima la Board, poi definendo gli eventi e solo successivamente azioni e comportamenti di gioco.

Tuttavia la struttura adottata per realizzare il model checking non vincola in nessun modo gli utenti del sistema. Infatti, i riferimenti ad oggetti definiti, per lo più gestiti tramite stringhe,

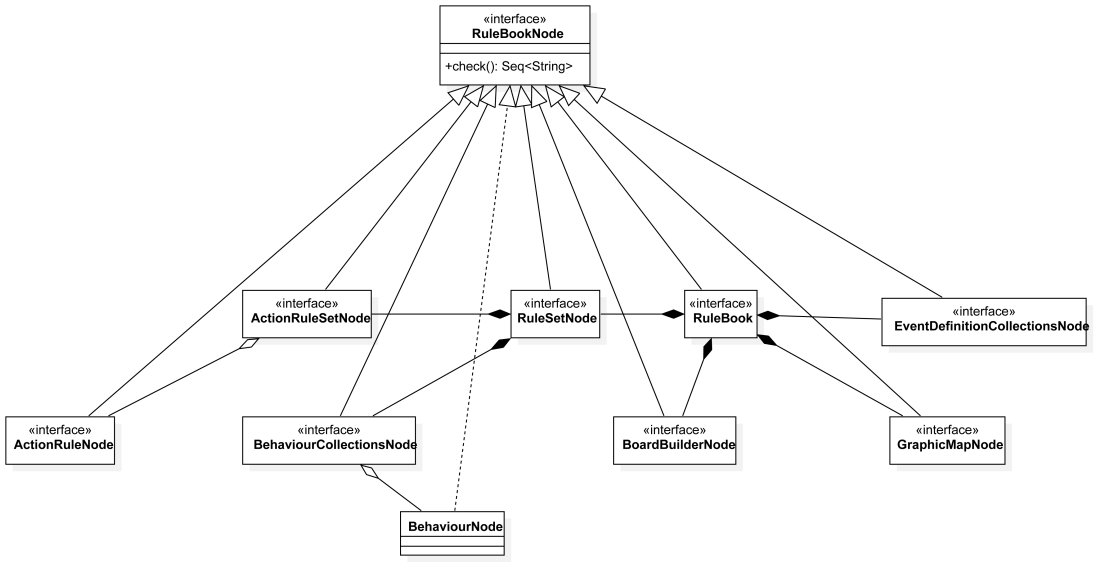

Figura 5.3: I principali elementi architetturali del DSL

vengono controllati in fase di check e risolti solamente in fase di generazione nell'ordine stabilito dal check stesso.

Per abilitare questa feature è stato necessario dichiarare delle interfacce, su alcuni nodi particolari, in grado di rispondere alle interrogazioni dall'esterno sulla presenza o meno di determinati identificativi. Un esempio è la EventDefinitionCollection.

Capitolo 6

Implementazione

Il capitolo corrente ha come obiettivo quello di introdurre tutte le decisioni e scelte implementative che caratterizzano il codice del framework.

Di seguito vengono riportate le parti più salienti e degne di essere documentate con maggior precisione, ricordando però che su tutto il codice è presente una documentazione autoesplicativa per ogni utilizzatore del framework.

6.1 Il framework

In questa sezione vengono presentate le tecnologie e le feature tecniche del linguaggio Scala impiegate dal team mostrando come sono state impiegate per raggiungere gli obiettivi e realizzare gli schemi progettati in fase di design.

6.1.1 Engine

L'engine è implementato usando la libreria Vert.x [4], con un singolo Verticle. Un Verticle è un entità in grado di ricevere dei messaggi in maniera asincrona che saranno poi gestiti tramite un apposito handler eseguito da un singolo thread. Il modello di un Verticle è esattamente quello di un event loop, che, grazie al fatto che esegue gli handler su un singolo thread, risolve numerosi problemi di concorrenza.

Come è possibile osservare dal diagramma di figura 6.1, l'engine è implementato dalla classe *VertxGooseEngine*, che aderisce alle interfacce *EventSink*, per la ricezione di eventi, e *GooseEngine*, che definisce i metodi invocabili dalla View. *GameMessageCodec* implementa la serializzazione e deserializzazione dei *GameEvent* salvati nella coda di eventi del GooseVerticle, mentre Vert.x si occupa di gestire gli aspetti di concorrenza a basso livello.

Il compito dell'engine è quello di gestire i nuovi eventi in arrivo, di interrogare lo stato corrente del gioco e di valutare tutte le regole ad ogni cambio di stato, nonché di mantenere la View aggiornata. In accordo con il design del Model, il Controller mantiene le operazioni da eseguire nel ciclo corrente all'interno di una pila, il *Game Stack*. L'event loop è mantenuto occupato durante tutta la risoluzione della pila, impedendo la *preemption* tra eventi scatenati da operazioni in pila ed eventi in arrivo dal mondo esterno, in maniera asincrona. Tali eventi potranno essere gestiti solamente al raggiungimento di uno stato di quiete, a fine ciclo.

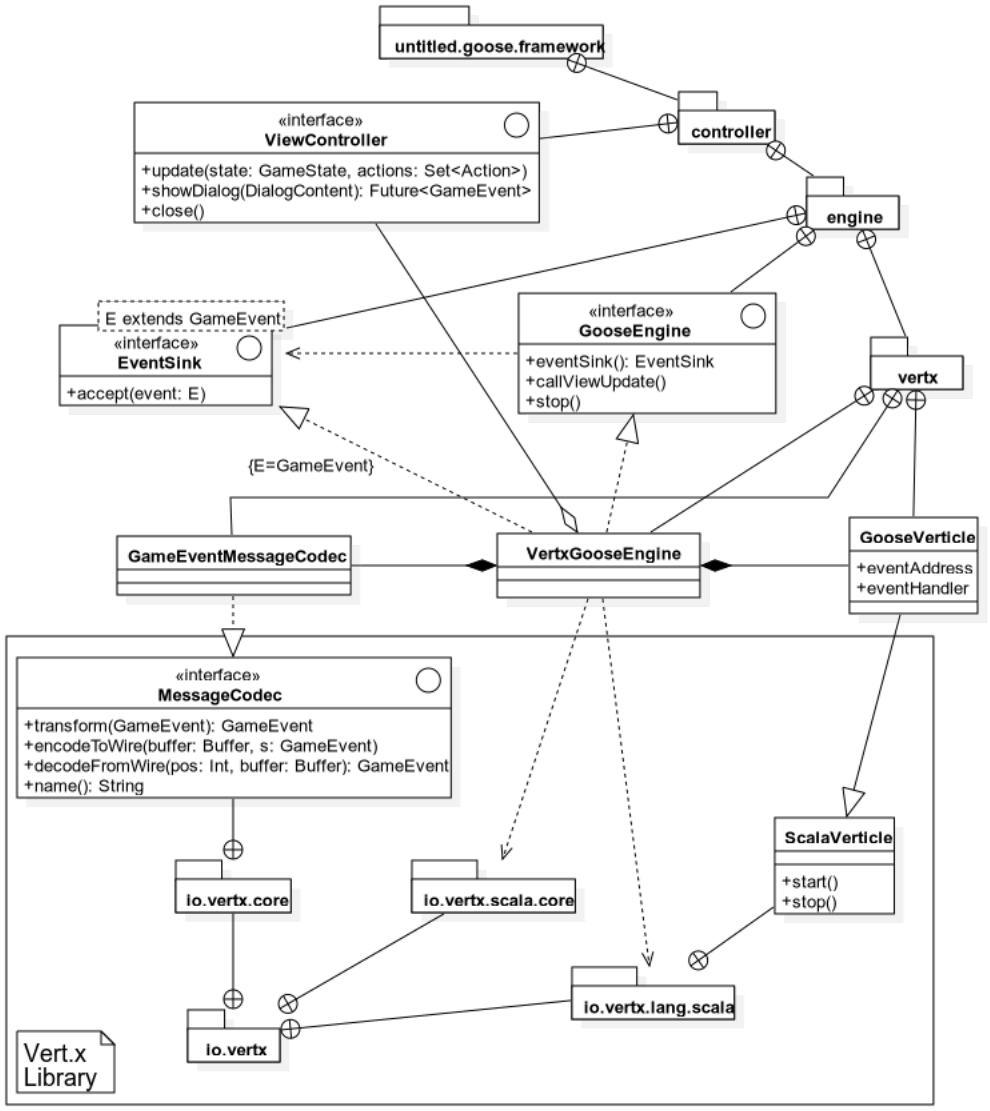

Figura 6.1: Diagramma delle classi del controller - UML.

6.1.2 Model

Il modello che rappresenta le entità ed esprime il comportamento dei giochi è stato realizzato partendo da una design di stampo object-oriented cercando di integrare comunque i principi della programmazione funzionale che risultavano compatibili. Iterativamente nel corso di tutta la durata dello sviluppo l'implementazione è stata rivisitata più volte.

Immutabilità degli oggetti Molte delle entità all'interno del framework sono immutabili, tra queste: TileDefinition, Position, Piece, GameEvent e regole. Dove possibile, si è cercato di mantenere l'immutabilità dei componenti nel pieno stile funzionale. Dove invece ciò non è stato possibile, in particolare nel GameState, è stato scelto di inserire delle interfacce protettive, in modo da restituire l'oggetto impedendo modifiche non controllate.

Eventi Nel caso specifico degli eventi, sono stati utilizzati i *Mixin*, col fine di poter avere eventi di tipologie differenti, con categorie - eventualmente - sovrapponibili. Ogni evento del sistema, quindi, è un GameEvent; ognuno di essi è però dotato di una diversa specializzazione che lo rende consumabile/persistente. Questo è stato fondamentale per il meccanismo della submit degli eventi nel corretto buffer di appartenenza, permettendo quindi di mantenere le history aggiornate tramite caseMatch. Ciò è stato possibile grazie alla reificazione dei tipi generici di Scala.

Behaviour e Operazioni Dal punto di vista implementativo rappresentano la parte più dinamica appartenente al package del modello. La loro implementazione utilizza oggetti e interfacce funzionali passate tramite costruttore. La valutazione delle regole produce una sequenza di operazioni che agisce sullo stato o lancia nuovi eventi. Per la struttura dei Behaviour si è fatto uso di *sealed trait* ed implementazioni private in modo che non fossero estendibili, ma fossero istanziabili solamente attraverso l'uso dei factory method.

Questo permette di semplificare l'uso del DSL nella creazione di nuovi comportamenti in quanto non è necessario scrivere codice scala per estendere la classe base, assumendo che siano sufficientemente espressive per coprire tutti i casi d'uso possibili. Infine, usufruiscono dell'uso di ClassTag per riconoscere la tipologia di evento dal *Consumable Buffer* ignorando la type erasure di Java. Grazie a ciò è stata realizzata un'implementazione generica di una pimped class per le sequenze di eventi che potesse effettuare le operazioni indispensabili di casting per consentire l'accesso alle proprietà specifiche di un dato evento.

6.1.3 View

Dal punto di vista implementativo, gli aspetti appartenenti alla componente View utilizzano la libreria ScalaFX, un DSL scritto in Scala per definire interfacce grafiche, che utilizza JavaFX come supporto. E' stato preferito, a JavaFX, l'utilizzo di questa libreria a causa della volontà di mantenere lo sviluppo, ed il codice susseguente, quanto più omogeneo possibile, utilizzando quindi una libreria che non mischiasse codice Java/Scala. I Dialog sono implementati attraverso gli *Alerts*, finestre di dialogo personalizzabili a partire da modelli di base, che restituiscono la scelta dell'utente, convertita in risposta e sottomessa all'engine.

La classe IntroMenu è stata concepita per essere un primo interfacciamento con l'utente, prima di cominciare una partita; può essere quindi vista come una sorta di interfaccia di creazione di una *lobby*. Tra le funzionalità offerte vi è la possibilità di inserire tutti i giocatori nella partita,

ognuno correlato da un proprio segnaposto, il *Piece*, dotato di un colore a scelta tra varie opzioni, in modo che sia più agevole distinguere i giocatori durante le fasi della partita. E' inoltre possibile definire l'ordinamento dei giocatori mediante gli appositi controlli, indipendentemente dall'ordine con cui sono stati inseriti; ciò è di fondamentale importanza in quanto, questo dato, ptrebbe essere utilizzato nel *Ruleset* per stabilire l'ordine dei giocatori.

Prima di poter iniziare la partita viene controllato che il numero di giocatori inseriti rientri nel range speficiato. Una volta che tutti i vincoli sono rispettati, è possibile iniziare la partita tramite l'apposito pulsante start, il cui handler si occupa di creare un *Game* contenente le informazioni della partita e i giocatori appena inseriti. Viene inoltre istanziato un *ApplicationController* grafico - il cui compito è quello di gestire le scene (quella principale è una GameScene), ricevere aggiornamenti sullo stato del gioco, creare nuove Dialog, processare le scelte dell'utente e fermare l'engine.

La GameScene è stata costruita utilizzando diversi pannelli che potessero incastrarsi al meglio, per combinare una fluida e immersiva esperienza di gioco senza dover rinunciare alla presenza di tutti i controlli necessari per portare a termine la partita, avendo quindi informazioni riguardanti l'attuale giocatore di turno, le possibile azioni tra le quali dovrà scegliere, calcolate a seconda della tile su cui la propria pedina si trova. Il tutto è stato realizzato in modo tale da potersi adattare a qualsiasi tipologia di schermo e risoluzione, senza compromettere il render della board di gioco e la sua fruibilità.

Questo è reso possibile attraverso il binding delle proprietà; a ciascuna di esse viene infatti assegnato un valore che può essere osservabile. Queste features vengono sfruttare soprattutto in **BoardDisplay** e **TileVisualization**. Le proprietà possono quindi essere definite a partire da altre, sapendo che il cambiamento di una comporta un aggiornamento di quelle derivate, tra le altre cose, consente di ottenere facilmente interfacce resizable.

Graficamente ogni Piece, sulla board, è dotato di un proprio colore identificativo specificato nelle fasi preliminari di gioco. Nel caso in cui due giocatori dovessero trovarsi sulla stessa tessera, in un determinato momento della partita, i relativi Piece non si sovrapporranno ma si disporranno invece su porzioni diversi della suddetta tessera.

Per la realizzazione di una GUI semplice e gradevole abbiamo scelto di integrare anche fogli di stile in CSS. La volontà è quella di snellire il codice, separando gli aspetti puramente estetici e di formattazione grafica da quelli di definizione e comportamento dei componenti stessi, facilitandone, tra le altre cose, la personalizzazione da parte di utilizzatori esterni. Tutto ciò che fa parte della formattazione corretta dei componenti è stato definito in un file CSS statico e non modificabile importato con la libreria; mentre la scelta dei colori, del background e dello stile del testo possono essere modificati definendo dei file con i nomi `styleIntroMenu.css` e `styleGameScene.css`, nelle risorse, in modo che vengano riconosciuti dal framework e caricati all'avvio dell'applicazione.

6.1.4 Testing

Dal punto di vista del testing del sistema si è scelto di produrre Unit Tests in quanto la maggior parte dei componenti del framework hanno un comportamento semplice, poco articolato, per cui è sufficiente verificarne la correttezza in maniera isolata. Per mantenere uno stile omogeneo per gli Unit Tests abbiamo scelto una linea guida, che consiste nell'utilizzo del DSL **FlatSpec** e dei *should Matchers* , realizzando così dei test estremamente autodescrittivi.

Questo approccio si avvicina ad uno stile di sviluppo Behaviour Driven Developement (BDD) in cui i test sono combinati con un testo che ne specifica il comportamento o behaviour e comples-

sivamente può essere descritto come di seguito: "X should Y", dove per X si intende il nome del soggetto dei test mentre Y è il comportamento atteso.

Continuous Integration/Continuous Deployment Un plugin per sbt incluso nel progetto (`sbt-github-actions`) genera i file di configurazione per le **Github Actions** tramite un task sbt. Grazie a questi file i test sono stati resi automatici, vengono infatti eseguiti ad ogni push o pull request sul repository GitHub del framework. In aggiunta a ciò, quando viene effettuato un push sul branch `master`, viene creata automaticamente una nuova release del framework sotto forma di **jar**, resa disponibile all'interno della sezione dedicata alle release della pagina GitHub del progetto.

6.2 Il DSL

6.2.1 Sintassi

L'implementazione del design di progettazione del DSL è stata guidata dall'utilizzo delle feature principali del linguaggio Scala, vista anche la natura stessa del DSL che sfrutta alcune caratteristiche del linguaggio per permettere di agevolare la scrittura di codice.

Le "classi astratte" contenenti i punti di accesso alle nuove frasi sono state realizzate come trait in modo da poter utilizzare la potenza dei *Mixin* e creare strutture complesse e fortemente configurabili. Ogni macro-area del modello di gioco ha, quindi, un proprio Mixin che racchiude le parole che ne abilitano la definizione, in modo tale che una modifica o aggiunta di feature anche a livello del modello stesso sia facilmente manutenibile a livello di DSL.

Per quel che riguarda le parole si è scelto di utilizzare invece le case class per la loro facilità di dichiarazione dal momento che molte parole sono dotate di un comportamento minimale oltre che del passaggio dei parametri tramite costruttore.

Per fare sì che le parole potessero aggiungere i nodi all'albero si è fatto un largo uso di impliciti definendo il RuleBook come parametro implicito nel trait principale da cui estendere e passandolo alle parole responsabili della creazione dei nodi.

Oltre a questo alcune feature particolari del compilatore sono state sfruttate per realizzare la definizione di sintassi semplici e intuitive: in particolare la definizione di operatori con simboli e l'utilizzo di parametri anche fittizi composti da parole vuote per garantire fluidità nella lettura della frase e permettere al compilatore di inferire correttamente la catena di metodi e parametri.

6.2.2 Model Checking e generazione

A livello implementativo i nodi sono stati realizzati implementando un trait, *RuleBookNode*, in modo da offrire un'interfaccia comune per il controllo ricorsivo dell'albero.

Per la realizzazione di diverse categorie di nodi che rappresentassero un unico concetto nelle sue diverse sfaccettature (nello specifico azioni e behaviour, ma non solo) si è fatto uso di *sealed trait*. Sono state poi create implementazioni auto-contenute nel medesimo sorgente in modo da forzare *case match* esaustivi data questa scelta implementativa.

Per risolvere i riferimenti la scelta è ricaduta su delle semplici interfacce implementate da nodi specifici in grado di mantenere aggiornati i riferimenti.

6.3 Giochi

Per completare lo sviluppo del framework, abbiamo fornito tre esempi di giochi creati sfruttando il DSL sviluppato: *GooseGame*, *Snake and Ladders* e *Quiz Race*.

Goose Game Consiste nella versione più tradizionale del Gioco dell'Oca. I giocatori si alternano nel lancio di un dado che indica di quanti caselle potranno avanzare. A seconda dei risultati del dado, il giocatore si troverà a doversi spostare in alcune caselle che attivano dei Behaviour sulla base dello stato del gioco corrente. Alcuni esempi di tali comportamenti sono: saltare uno o più turni, avanzare/indietreggiare sul percorso di gioco, spostarsi istantaneamente da una tile all'altra, reagire al superamento da parte di un altro giocatore e così via.

Questa implementazione di gioco è stata utilizzata anche durante lo sviluppo del framework come test delle possibili interazioni di gioco. In questo modo GooseGame ha guidato alcune delle scelte implementative pur cercando di mantenere costantemente un occhio sulla generalità delle soluzioni adottate. Sulla base di questo gioco sono stati quindi prodotti la maggior parte degli eventi e dei behaviour già forniti dal sistema e quindi riutilizzabili da parte degli utenti finali per creare il proprio gioco originale.

Snake and Ladders Consiste in una versione del gioco simile al Goose Game, con la differenza che sul tabellone sono presenti delle scale e dei serpenti i quali hanno un comportamento prestabilito.

Nel momento in cui un giocatore si troverà alla base di una scala, potrà sfruttarla per prendere una scorciatoia ed avanzare verso la vittoria, salendo fino alla cima della stessa scala. Nel caso in cui dovesse invece trovarsi sulla testa di un serpente - di solito localizzata vicino alla tessera di vittoria - dovrà retrocedere lungo tutto il suo corpo per ritrovarsi sulla casella in cui inizia la coda.

Quiz Race Questa versione di gioco da tavola si basa sul celebre gioco "Trivial Pursuit" rivisitato e adattato per rientrare nei canoni di gioco a percorso con un inizio ed una fine definite.

Si è scelto di realizzarlo per dare una dimostrazione della versatilità del framework, oltre che dell'estendibilità possibile se messo nelle mani di un programmatore Scala. Le caratteristiche principali sono infatti l'esistenza di una diversa meccanica di movimento, priva di dadi e soprattutto la presenza di nuove entità: le domande.

Per permettere la realizzazione del gioco, impersonando un potenziale sviluppatore, è stato necessario progettare un'entità QuestionManager. Questa entità è implementata in questo caso tramite lettura di un file JSON di domande, mantenute divise per categoria e restituite in ordine casuale, per poi essere utilizzate e convertite dal meccanismo di display dei dialog presente nel framework. Si potrebbe dire che tutto il gioco viene generato in funzione dei file delle domande, ancora una volta grazie alla potenza dei costrutti di scala integrati con il DSL.

Questo utilizzo ibrido ha permesso di valutare attentamente le potenzialità espressive del framework, convincendoci di aver realizzato un prodotto effettivamente calzante con le specifiche delineate inizialmente.

Questo progetto è stato anche utilizzato come showcase delle feature relative agli eventi dimostrando l'importanza della definizione di eventi custom, utilizzati per uno scambio di messaggi specifici per l'applicazione, ma anche dell'estensione delle classi base per realizzare eventi specifici di natura più complessa.

6.4 Sviluppo individuale

In questa sezione ogni componente del team specifica quali parti, tra quelle elencate nella sezione precedente, sono state realizzate in prima persona o in collaborazione con gli altri, mettendo in luce eventuali ulteriori dettagli e presentando una visione più personale del processo di sviluppo.

6.4.1 Burattini

Il contributo personale apportato il progetto ha toccato quasi tutte le parti dal core dell'applicazione fino ad alcuni aspetti di interfaccia grafica per poi concludersi sulla progettazione e implementazione del DSL e dei giochi realizzati attraverso il DSL stesso.

Sviluppo collaborativo Come accennato in precedenza il lavoro ha toccato trasversalmente tutte le parti del progetto quindi spesso mi sono trovato a lavorare insieme ad altri membri del team per risolvere problemi o lavorare in pair programming.

La parte svolta in collaborazione più stretta è stata quella finale del progetto sul DSL con Deluigi in particolare personalmente mi sono occupato principalmente della struttura sintattica e della progettazione.

Esperienza come SCRUM master Avendo scelto a inizio progetto di assumere il ruolo di SCRUM master nel corso dello sviluppo gli altri membri del gruppo mi hanno poi preso come riferimento organizzativo per stabilire tempi, obiettivi a lungo e breve termine e mi sono occupato di stimolare il lavoro per rispettare le scadenze.

Il lavoro importante come SCRUM master è stato svolto durante i meeting programmati a inizio e fine di ogni sprint per cercare di raccogliere e risolvere problematiche sia di sviluppo che organizzative oltre alle difficoltà comunicative che purtroppo sono state presenti nel corso della realizzazione del progetto.

In generale l'attività del processo SCRUM è stata seguita nella prima parte di sviluppo meglio che in quella finale, ma posso ritenermi soddisfatto per l'attenzione dedicata all'iteratività nel processo ed in generale la pianificazione del lavoro oltre che essermi messo in gioco con una attività nuova.

Eventi e behaviour Per quanto riguarda il modello una parte seguita più in prima persona è stata quella dello sviluppo di Behaviour, Operazioni ed eventi sia dal punto di vista generale che poi nel particolare con le implementazioni fornite di default dal sistema. É stata una parte interessante in quanto cruciale per il meccanismo di estendibilità del framework e infatti ha subito un pesante refactor verso la metà del progetto per convertirla dalla prima modellazione più "object oriented" ad una seconda versione che facesse leva sulle caratteristiche del paradigma funzionale.

Il DSL Per quel che riguarda il DSL il mio contributo principale è stato sulla sintassi e quindi la creazione di parole e l'aggiunta dei nodi all'albero sintattico. É stato necessario uno sforzo significativo per trasferire la genericità del modello in un linguaggio che fosse il quanto più semplice da comprendere per utenti che non fossero necessariamente dei programmatori.

Sviluppo dei giochi attraverso il framework Al termine dello sviluppo del DSL mi sono concentrato sul valutare le funzionalità e la semplicità di integrazione creando dei nuovi progetti ed usando il framework importandolo come una libreria esterna. Questo ha permesso di raffinare alcune parti e correggere dei bug servendo quindi da vero e proprio test di tutte le funzionalità del progetto.

Ho personalmente realizzato Il Gioco dell'Oca classico e la nostra versione originale di gioco a domande Quiz-Race, curandone tutti gli aspetti a partire dalla progettazione, implementazione e aspetto grafico.

6.4.2 Cardiota

Il mio contributo ha riguardato principalmente il design e lo sviluppo delle entità di base del progetto, di tutti gli aspetti di interfacciamento grafico e parte del testing.

Sviluppo collaborativo Nelle fasi iniziali del progetto, abbiamo deciso di lavorare in maniera quanto più collaborativa possibile, a volte coinvolgendo l'intero team di sviluppo, altre in modalità *Pair Programming*, andando a toccare la quasi totalità del software; compresi le parti di Model ed Engine. Questo è stato utile a facilitare le prime fasi di sviluppo e capire al meglio come poter collaborare e suddividere il lavoro nel modo più efficiente possibile in vista delle fasi di sviluppo più avanzate, in modo da lavorare quanto più parallelamente possibile.

A tal merito, ho inizialmente lavorato con Samuele Burattini allo sviluppo delle entità di base del progetto, al flusso di gioco che l'utente deve seguire e agli eventi che potrebbero verificarsi durante l'intero svolgimento di una partita.

View Una volta terminata questa fase, mi sono dedicato allo sviluppo dell'interfaccia grafica del programma, alla quale hanno collaborato anche Samuele Burattini e Francesca Tonetti. Personalmente, alcune fasi di questa sotto-parte hanno evidenziato alcune difficoltà nell'utilizzo del paradigma funzionale, alcune dovute a mie lacune poi colmate, altre dovute alla documentazione a volte non sempre completa ed esaustiva. A tal merito, è stato fondamentale lo spirito di collaborazione per superare i vari problemi che abbiamo incontrato.

Testing Data l'importanza del testing studiata durante il corso, abbiamo cercato di avere una coverage del codice quanto più alta possibile, in modo da essere consapevoli di scenari all'interno, difficilmente scovabili in assenza di testing, che potrebbero portare ad avere problemi più o meno gravi. A tal merito, mi sono occupato di sviluppare test sulle *Entities, Regole* e *Actions* del Model, ma la maggior parte del tempo dedicato a questa fase l'ho speso nello scrivere i test dei *Behaviours*, che si sono rivelati molto complessi da portare a termine visto l'elevata pervasività di essi all'interno della codebase del software. Durante questa fase, abbiamo collettivamente deciso di utilizzare *AnyFlatSpec*, vista la sua natura adatta a sviluppare test **Behaviour-Driven**.

Snake and Ladders Come definito nei requisiti, una parte del progetto prevedeva anche lo sviluppo di giochi dedicati che sfruttassero il DSL per costruire i suddetti in maniera semplice e veloce. Personalmente, mi sono occupato dello sviluppo del gioco *Snake and Ladders*.

Product Ownership Durante le primissime fasi di analisi dei requisiti del progetto, ho deciso che mi sarei assunto anche la responsabilità di essere il *Product Owner* del progetto, in quanto mi era già capitato di studiare tali tematiche in altri corsi opzionali.

Visto che tutti i componenti del gruppo sono di fatto sviluppatori, l'importanza di questo ruolo è stata relativamente appiattita; il che vuol dire che la mia principale mansione nel ruolo di Product Owner è stata quella di collaborare con lo Scrum Master, oltre che a decidere quali fossero le priorità di sviluppo nel progetto e quali scelte fare in modo da rispettare le scadenze settimanali, e infine la deadline ultima del progetto stesso, coordinando quindi in parte il lavoro di gruppo.

Altro Mi sono occupato di scrivere parte della Scaladoc, in particolar modo la documentazione che riguarda le sezioni del framework alle quali ho lavorato maggiormente, in modo da poterla scrivere in maniera semplice ed intuitiva.

6.4.3 Deluigi

Il mio ruolo nel progetto è stato in un primo momento di progettista e designer di model e controller, in un secondo momento di sviluppatore di questi, e infine ho collaborato con Burattini al design e all'implementazione del DSL. Mi sono anche occupato della gestione del repository, CI/CD e della qualità del codice.

Sviluppo Collaborativo Durante i primi sprint ho partecipato attivamente al design dell'architettura e di dettaglio del core dell'applicazione (quello che poi è diventato model e controller). Successivamente ho sviluppato in pair programming o singolarmente la prima versione dei componenti principali per poi occuparmi in maniera isolata del loro mantenimento nei riguardi dei cambi di design iterativi.

Ho assunto numerose responsabilità in prima persona lungo lo sviluppo del progetto ma ho avuto piacere anche nel collaborare in pair programming come aiutante soprattutto di Burattini per il DSL.

Model Mi sono occupato della realizzazione e della manutenzione delle entità del modello, sia delle componenti statiche e si di quelle dinamiche, a livello architetturale, di design e di interfacce. Ho implementato in modo *naive* molte di queste ma ho realizzato numerose classi non banali, inclusi gli algoritmi per la disposizione, i builder, e ho guidato il refactoring delle implementazioni dei behaviour. Ogni volta che risultava opportuno non mi sono fatto problemi ad applicare modifiche anche pesanti al modello, senza mai incontrare troppe viscosità o impedimenti, segno di un buon lavoro di design.

Controller Mi sono occupato quasi interamente della logica di basso livello dell'engine, in particolare dell'uso di Vert.x e del logging, anche grazie alle conoscenze del corso di Programmazione Concorrente e Distribuita. Ho progettato con il team lo stack di gioco e l'ho implementato in pair programming. Successivamente ha subito varie modifiche effettuate da me singolarmente, ma il concetto iniziale è rimasto lo stesso.

DSL Ho progettato e implementato, sempre affiancato da Burattini, la logica di model checking del DSL, l'albero astratto, nonché quella di creazione del gioco, e qualche componente abilitante

per certe sintassi. Ho supervisionato Cardiota nello sviluppo del gioco Snakes and Ladders tramite DSL.

Gestione del repository, test, CI/CD Mi sono occupato di gestire il repository GitHub del progetto, abilitando funzionalità utili ad un possibile nuovo visitatore o collaboratore. Ho studiato la documentazione e scritto alcuni file di configurazione che hanno permesso di creare un sistema di Continuous Integration inizialmente limitato ad eseguire automaticamente i test presenti nel progetto. Mi sono occupato di scriverne alcuni all'inizio poi ho ceduto il compito di proseguire ai colleghi. Successivamente ho configurato anche un sistema di Continuous Deployment che crea nuove release del framework a partire dalla history Git del branch master. Questo è stato possibile grazie a *sbt-github-actions*, il plugin open source per sbt che genera i file per GitHub Actions, al quale ho anche contribuito personalmente durante l'utilizzo, dovendo aggiustare un bug.

Altro Mi sono occupato dello zoom (attivabile con i tasti + e -) e della riusabilità della View, della scrittura della maggior parte della scaladoc dei componenti pubblici che la presentano e di pulizia del codice in generale.

6.4.4 Tonetti

Dal punto di vista implementativo ho avuto la possibilità di mettere mano alle varie componenti del framework, nello specifico mi sono cimentata sullo sviluppo di alcune classi del modello e della view, parallelamente alla realizzazione di numerose classi di test. Infine, ho avuto la possibilità di affiancare alcuni colleghi durante la realizzazione dei giochi sviluppati attraverso il DSL.

Sviluppo Collaborativo Come espresso precedentemente, durante lo sviluppo del framework ho avuto la possibilità di contribuire alla maggior parte dei componenti e questo mi ha permesso di avere una collaborazione sempre attiva con gli altri membri del team, lavorando parallelamente o aiutandosi nella risoluzione di eventuali problemi riscontrati.

Model Per quanto riguarda il modello, seguendo i primi sprint di lavoro, ho avuto la possibilità di cimentarmi nello sviluppo delle componenti principali di gioco come il player, la tile, la board o il piece, usufruendo inizialmente di un approccio Object Oriented in linguaggio Scala in modo da acquisire sempre più confidenza con il linguaggio stesso e successivamente con la programmazione funzionale applicata al progetto attraverso un importante refactor. Questo primo approccio mi è risultato estremamente utile per avvicinarmi alla struttura del gioco con maggior chiarezza. Altre componenti del modello di cui mi sono occupata riguardano gli eventi del gioco e le regole legate al comportamento e alle operazioni. Durante tutta questa prima fase di sviluppo ho avuto il piacere di collaborare con Luca Deluigi e Samuele Burattini.

Testing Parallelamente alla realizzazione delle classi di modello, mi sono occupata dello sviluppo di numerosi test relativi alle stesse classi e a componenti di altri package, seguendo un approccio omogeneo affinché ciasun test risultasse più autodescrittivo possibile. Questa parte mi ha permesso di cimentarmi nella comprensione di classi realizzate da altri componenti del team e acquisire maggior chiarezza su alcuni aspetti del framework.

View Per la realizzaizone dell'interfaccia grafica, la parte di cui mi sono occupata totalmente e con maggior dedizione riguarda l'inserimento dei fogli di stile css nel sistema. A partire da una semplice operazione cut-and-paste con l'obiettivo di snellire il codice delle classi di view, ho modificato la maggior parte degli elementi associando loro una classe in modo da referenziarli nei file css e modificarne la grafica in maniera precisa.

Altro Verso il termine dello sviluppo del framework ho avuto la possibilità di affiancare alcuni componenti del gruppo per la realizzazione dei giochi implementati attraverso il DSL. Inoltre, mi sono occupata di una parte della scrittura della scaladoc dei componenti del modello che la presentano.

Capitolo 7

Retrospettiva

In questo capitolo si cerca di analizzare e riassumere tutto il processo design, sviluppo e organizzazione del lavoro in relazione ai risultati finali ottenuti. Complessivamente il team si ritiene soddisfatto del risultato finale ottenuto in quanto si è fatta una proposta sicuramente ambiziosa che ha portato a delle sfide non banali.

7.1 Analisi del processo di sviluppo

Adottare un processo di sviluppo agile è stata sicuramente una delle più grandi novità nello svolgimento di questo progetto oltre a quella della scelta tecnologica.

Complessivamente il team si è impegnato ad affrontare la progettazione e lo sviluppo secondo questa nuova filosofia, all'inizio con più dedizione e poi nel corso del tempo con meno attenzione ai dettagli.

Indubbiamente l'esperienza ha messo in luce come una metodologia di questo tipo si possa applicare meglio a problemi dove è possibile affrontare il lavoro individuando dei moduli più isolati rispetto a quelli del progetto presentato che è stato caratterizzato da uno sviluppo più verticale e ostacolato dalle numerose interdipendenze che solo in una fase avanzata di progettazione si sono poi sciolte con la dovuta attenzione.

Inoltre il processo adottato sicuramente si adatta meglio a gruppi con un livello omogeneo di indipendenza nelle scelte di progettazione in quanto nel compiere un task ad ogni membro del team di sviluppo viene affidata anche una buona dose di libertà creativa. Questo è necessario perché è praticamente impossibile definire nel dettaglio gli obiettivi nella fase di planning preliminare dello sprint. Nel nostro caso questo ha portato a frequenti difficoltà e rallentamenti che sono stati risolti attraverso il valido strumento del pair-programming, molto usato anche per la situazione di forzata distanza che la situazione dei mesi di sviluppo ha imposto.

Come accennato in precedenza inoltre il team è riuscito a mantenere una pianificazione organizzata e precisa nella prima metà del progetto mentre verso la fine con molti filoni rimasti aperti è stato più complesso restare nei canoni rigidi della metodologia SCRUM.

In generale si sono comunque utilizzati tutti gli strumenti di tale metodologia a parte gli incontri giornalieri, difficili da organizzare data la dislocazione fisica dei membri del gruppo e il periodo estivo che per impegni personali e lavorativi ha reso complicato mantenere un orario di lavoro compatibile con le esigenze dei singoli.

7.2 Lavori futuri

Il progetto ha portato alla realizzazione di un prodotto completo, utilizzabile e manutenibile che il team è fiero di aver creato.

Dal momento che il prodotto finito potrebbe avere una usabilità il team si è interrogato su alcune features su cui poter lavorare in un eventuale futuro, anche in vista di una apertura al pubblico del repository per la quale il gruppo dovrà indubbiamente proporre una guida pratica per l'utilizzo del framework in un proprio progetto oltre che per l'utilizzo del DSL.

Alcune delle features - di minore importanza che sono state messe in considerazione ma non realizzate a casua del tempo di sviluppo al di fuori della nostra attuale portata sono: l'inclusione del progetto in un remote repository per gestire al meglio le dipendenze, l'animazione delle pedine dei giocatori che si spostano attraverso la tavola da gioco per rendere il gioco più immersivo, l'introduzione della possibilità di interrompere il movimento nel gioco senza dover cambiare la meccanica di movimento di base ed infine l'inclusione di una parte del software scritta in Prolog. In particolare quest'ultimo elemento aveva fatto proprio emergere l'idea di realizzare questo tipo di progetto ma poi lo sviluppo specialmente dell'Engine di risoluzione del gioco ha preso più tempo di quanto immaginato all'inizio ed è stata tralasciata.

In generale ci sono molte possibili espansioni, in primis un rework della grafica e del modello per poter supportare una customizzazione ancora più forte per quel che riguarda l'aspetto del gioco e la conformazione del tabellone ed in generale dell'esperienza di gioco.

In secondo luogo il DSL può essere ulteriormente raffinato specie supportando direttamente da modello più azioni di base, rendendo più semplice il compito degli sviluppatori per indicare solamente le meccaniche specifiche del gioco come composizione di meccanismi di base. Una ulteriore feature interessante sarebbe quella di dedurre in base alle regole adottate quali behaviour di sistema includere senza la necessità di esprimerli direttamente.

7.3 Conclusioni

Il progetto è stata una esperienza importante per toccare con mano gli aspetti di gestione, progettazione e sviluppo di un sistema software complesso adottando un approccio professionale curandone tutti gli aspetti compreso il delivery ed il controllo della qualità in itinere.

Oltre a questo è stata una più che valida esplorazione del linguaggio Scala e delle sue potenzialità in termini di sviluppo software. Nell'individuazione del progetto il gruppo è stato attento a scegliere un dominio applicativo in cui il paradigma funzionale potesse far valere i propri punti di forza.

Non è stato sicuramente immediato immergersi nella programmazione del lavoro con le metodologie studiate, né tantomeno nella progettazione del software con un paradigma diverso da quello a cui abituati in precedenza. Allo stesso tempo, tuttavia, con l'aumentare della dimestichezza nell'utilizzo del linguaggio è stato più semplice re-immaginare alcune componenti del modello per trasformarle ed integrarle meglio con l'approccio funzionale.

Infine, sviluppare un framework è stata una decisione interessante in quanto molto diversa dai progetti intrapresi fin'ora dai membri del gruppo. In particolare è stato interessante immaginare di creare uno strumento da mettere nelle mani di potenziali utilizzatori piuttosto che un prodotto finale auto-contenuto. Oltre a questo le diverse componenti del framework hanno permesso di spaziare molto con diversi domini, tecniche di programmazione e problemi da risolvere rendendo l'esperienza complessiva estremamente variegata e stimolante.

Riteniamo che il progetto sia stata quindi un'esperienza costruttiva e fondamentale per fare propri i concetti presentati nella durata di tutto il corso.

Backlog di progetto

Sprint 06/07 - 19/07

Al termine della prima settimana è stata realizzata un'implementazione basilare del modello del gioco da tavolo, con tutte le entità di base tra le quali la board, lo stato, il template delle actions e la tiledefinition. Inoltre è stata sviluppata una prima versione dell'interfaccia grafica organizzata in moduli che consentissero una semplice visualizzazione e interazione con la stessa.

Sprint 20/07 - 03/08

Al termine della seconda settimana è stato introdotto lo sviluppo dell'engine per la gestione degli eventi del gioco. É stato introdotta la meccanica per il movimento in seguito al lancio del dado e portata a termine la disposizione delle tile nella board.

Sprint 03/08 - 09/08

Al termine della terza settimana sono stati realizzati i primi test basilari relativi alle entità del modello. Inoltre, sono state aggiunte alcune funzioni di base come il teletrasporto o il movimento avanti e indietro nella board, oltre all'inserimento del logger e del dialog affinché si potessero gestire le risposte da parte dell'utente bloccando lo stack delle operazioni.

Sprint 10/08 - 19/08

Al termine delle quarta settimana sono stati realizzati la maggior parte dei test tralasciando quelli sui behaviour. Dal punto di vista dell'interfaccia grafica è stata realizzata la selezione dei player.

Sprint 17/08 - 24/08

Al termine della quinta settimana sono state apportate modifche alla view in modo tale da applicare il colore alle tile e l'inserimento di una lista di giocatori. Inoltre, è stato abbozzato un wrapper per tutti i componenti statici e rifattorizzato buona parte del modello.

Sprint 24/08 -31/08

Al termine della sesta settimana sono stati impostati i primi test sui behaviour, inserita la rimozione dei player nella selezione prima dell'inizio del gioco. Infine è stato implementato il DSL relativo alla board e alle tile, oltre che alle parti "statiche" del RuleSet come il numero di giocatori o l'ordine degli stessi.

Sprint 31/08 - 6/09

Al termine della settima settimana sono stati completati tutti i test sul modello con annessa risoluzione dei bug presenti. Inoltre, è stato completato lo sviluppo del DSL per la definizione di azioni ed eventi. Parallelamente per l'interfaccia grafica sono stati effettuati numerosi refactor per renderla indipendente dalla implementazione.

Sprint 07/09 - 16/09

Al termine della ottava settimana è stato portato a termine lo sviluppo del DSL. Inoltre, ci siamo occupati della gestione di alcuni bug dell'interfaccia grafica come ad esempio lo scroll della board.

Sprint 17/09 - 23/09

Al termine della nona settimana sono stati completati tutti e tre i giochi. Per la restante parte del tempo è stata scritta la relazione e completata la documentazione di progetto.

Bibliografia

[1] *Trello*. https://trello.com/en.

[2] *sbt-github-actions*. https://github.com/djspiewak/sbt-github-actions.

[3] *ScalaFx*. https://github.com/scalafx/scalafx.

[4] *Eclipse Vert.x*. https://vertx.io/.