Alma Mater Studiorum · Università di Bologna

Corso di
Paradigmi di Programmazione e Sviluppo
Docenti: Mirko Viroli, Roberto Casadei

Lost 'n Souls
Roguelike game

Autori:
Matteo Brocca · 1005681
Alan Mancini · 1005481
Federico Mazzini · 980559

Settembre 2021

Indice

Introduzione

Il progetto scelto mira a sviluppare un single player game di genere Roguelike, ispirato al videogioco The Binding of Isaac.

Scopo del gioco, per un giocatore, è controllare un personaggio all'interno di un dungeon composto in modo procedurale da un numero finito di stanze definite in maniera procedurale. All'interno delle stanze saranno presenti in maniera casuale elementi bloccanti, nemici e oggetti. Il giocatore dovrà cercare di sconfiggere i nemici per passare alla stanza successiva e raccogliere oggetti al fine di potenziarsi o cambiare le sue caratteristiche. Una partita termina con la sconfitta del boss, un nemico unico presente all'interno del dungeon in una specifica stanza, o con la morte perenne del personaggio, dove i progressi fatti durante il gioco vengono persi.

1 Processo di sviluppo adottato

Il processo di sviluppo adottato dal team è incrementale e iterativo. Si è cercato il più possibile di attenersi al framework **Scrum**, adattato alle esigenze lavorative e scolastiche dei membri del team. Il team ha effettuato **sprint** settimanali, in modo tale da massimizzare il numero di cicli iterativi di sviluppo. Inizialmente, gli sprint hanno avuto una parte di **planning**, mentre al loro termine, una parte di **review**. Di seguito si analizza nel dettaglio il metodo utilizzato.

1.1 Meeting

Ai meeting ha sempre partecipato il team al completo. Al bisogno, Matteo ha svolto il ruolo di esperto di dominio/committente. Ogni scelta all'interno del progetto è comunque sempre stata condivisa da tutto il team.

1.1.1 Sprint planning

Lo sprint planning è svolto all'inizio di ogni Sprint ed è di fondamentale importanta in quanto permette di definire nel dettaglio i task da eseguire all'interno dello sprint e i goal per esso. Ogni sprint planning si compone di:

- Raffinamento del product backlog e identificazione dei goal per lo sprint.
- Definizione dei **Task** come unità di lavoro pratica per soddisfare i **requisiti**;
- Assegnazione dei Task ai membri del team.

Lo Sprint Planning ha una durata massima di 2 ore.

1.1.2 Daily Scrum

Durante il Daily Scrum ogni sviluppatore espone al team i seguenti punti:

- Quale lavoro ha svolto la giornata precedente;
- Quale lavoro intende svolgere nella giornata corrente;
- Eventuali possibili impedimenti per il lavoro da svolgere, e come gli altri membri del team potrebbero aiutare ad affrontare il problema.

La durata di questo incontro è al massimo di 15 minuti.

1.1.3 Sprint review

La Sprint review analizza l'iterazione appena avvenuta e si concentra sul prodotto software in sè, in particolare si discute di:

- Ispezione dell'incremento ottenuto in termini di funzionalità e risultati tangibili per il cliente
- Adattamento del Product Backlog;
- Discussione su ciò che potrebbe essere fatto nel prossimo Sprint, utile come preparazione al prossimo Sprint Planning.

Durata massima: 1 ora.

1.1.4 Sprint retrospective

La Sprint retrospective analizza l'iterazione appena avvenuta concentrandosi sul processo di sviluppo e il team, in particolare si discute di:

- Come sono stati utilizzati i tool per il team e si analizza l'andamento dei meeting
- Idee per migliorare il processo di sviluppo, in particolare i punti critici individuati al punto sopra

Durata massima: 45 minuti.

1.2 Divisione dei Task

La suddivisione dei task è su base volontaria. Questo significa che tutti i membri del team si offrono volontariamente per svolgere un determinato task, nei limiti ovviamente della totalità dei task e dei goal per lo specifico sprint. Durante il daily scrum, può essere rivista qualche decisione, se non troppo radicale.

1.3 Definition of done

Il team ha definito, come e in che modo, un task può essere definito come done e di conseguenza concluso

1. Superamento di tutti gli Scalatest
2. Codice documentato con opportuna Scaladoc
3. Code review

1.4 Tool

Il team ha individuato i seguenti strumenti per favorire un processo agile, migliorare l'efficienza e favorire l'automazione durante il processo di sviluppo

- **SBT** come strumento di build automation
- **Scalatest** per la scrittura ed esecuzione dei test automatizzati
- **GitHub** come strumento di controllo versione e *continuous integration*
- **Jira** come strumento a supporto di scrum, gestione del product backlog e delle varie board di sviluppo

2 Requisiti

2.1 Requisiti di business

- Creazione di un gioco di genere Roguelike single player con la possibilità per un utente di comandare un personaggio all'interno di una mappa composta da stanze, affrontare nemici, raccogliere oggetti e sconfiggere un nemico finale.
- Possibilità di gioco su browser

2.2 Requisiti utente

Matteo Brocca in questo progetto ricopre il ruolo di esperto di dominio e committente. E' un appassionato di giochi Roguelike ma essendo troppo bravo li ha finiti tutti. Da qui l'idea di crearne uno nuovo per lui.

1. L'utente potrà
   - 1.1 avviare una nuova partita da un menu contestuale
   - 1.2 controllare con la tastiera il proprio personaggio all'interno della stanza per
     - 1.2.1 spostarsi e cambiare direzione
     - 1.2.2 sparare ai nemici che possono essere di diversa tipologia
     - 1.2.3 evitare elementi bloccanti o di disturbo se presenti
     - 1.2.4 raccogliere oggetti utili all'aumento delle sue caratteristiche
     - 1.2.5 spostarsi da una stanza all'altra attraverso delle porte
   - 1.3 visualizzare in modo continuo le sue statistiche e i suoi punti vita
   - 1.4 visualizzare in modo continuo una mappa del dungeon che indichi la sua posizione corrente
2. L'utente dovrà riuscire a
   - 2.1 distinguere visivamente nemici, oggetti e elementi di disturbo
   - 2.2 capire in che direzione si sta muovendo
   - 2.3 capire dove sta sparando
   - 2.4 capire di aver colpito un nemico
   - 2.5 capire di aver raccolto un oggetto
   - 2.6 capire qual'è la stanza del boss
   - 2.7 capire che sta fronteggiando un boss
   - 2.8 capire di aver vinto o perso una partita

2.3 Requisiti funzionali

1. Menù di gioco
   - 1.1 presenza di un tasto per avviare una nuova partita
2. Caricamento del gioco
   - 2.1 presenza di una schermata durante il caricamento degli elementi di gioco
3. Generazione del dungeon/mappa di gioco 2D in maniera casuale
   - 3.1 Durante la generazione deve essere visualizzata una schermata di attesa
   - 3.2 Un dungeon è formato da più stanze quadrate
   - 3.3 Ogni stanza è fisicamente adiacente ad almeno un'altra stanza
   - 3.4 Tra stanze adiacenti è presente una porta che le collega
   - 3.5 La disposione delle stanze avviene favorendo forme di mappa complesse simil labirinto
   - 3.6 Una stanza è di una tipologia fra
     - 3.6.1 Vuota
       - 3.6.1.1 Stanza completamente vuota
       - 3.6.1.2 La partita comincia con il personaggio in una stanza vuota
       - 3.6.1.3 In un dungeon ci sono circa un 10% di stanze vuote
     - 3.6.2 Oggetto
       - 3.6.2.1 Contiene al centro un singolo oggetto scelto randomicamente
       - 3.6.2.2 In un dungeon ci sono circa un 15% di stanze oggetto
     - 3.6.3 Combattimento
       - 3.6.3.1 Il contenuto è generato randomicamente
       - 3.6.3.2 Contiene alcuni nemici
       - 3.6.3.3 Contiene alcuni elementi bloccanti disposti in modo da non ostruire l'accesso alle porte da parte del personaggio
       - 3.6.3.4 Le porte si chiudono quando il giocatore entra nella stanza
       - 3.6.3.5 Le porte si aprono quando il giocatore ha eliminato tutti i nemici
       - 3.6.3.6 In un dungeon ci sono circa un 75% di stanze combattimento
     - 3.6.4 Boss
       - 3.6.4.1 Contiene solamente il nemico boss
       - 3.6.4.2 Una sola nel dungeon
       - 3.6.4.3 Adiacente ad una ed una sola altra stanza

4. Scena di gioco

    4.1 A video, dopo la generazione del dungeon, è visibile

        4.1.1 al centro dello schermo e rappresentata con vista dall'alto solo la stanza dove è correntemente presente il personaggio

        4.1.2 a sinistra della stanza l'elenco delle statistiche del personaggio

        4.1.3 a sinistra della stanza una mini mappa del dungeon

            4.1.3.1 le stanze vuote e combattimento sono di colore neutro (bianco)

            4.1.3.2 la stanza corrente, quelle oggetto e la stanza boss sono evidenziate con colori propri

    4.2 All'interno di una stanza nemici, oggetti, elementi bloccanti e personaggio, in linea generale

        4.2.1 sono vincolati a stare nei limiti del perimetro rappresentato dal pavimento

        4.2.2 non possono fisicamente attraversarsi tra loro sovrapponendosi graficamente, tuttavia la parte in alto di un'entità può sovrapporsi alla parte bassa di un'altra senza generare collisione ad emulare altezze diverse

    4.3 All'interno di una stanza un colpo sparato

        4.3.1 è rappresentato con un cerchio colorato

        4.3.2 è vincolato a stare nei limiti del perimetro rappresentato dal pavimento

        4.3.3 colpisce un'entità qualsiasi non appena il primo entra in contatto con il rettangolo che graficamente racchiude interamente la seconda

        4.3.4 quando colpisce qualcosa il cerchio viene sostituito da un esplosione animata

5. Personaggio controllabile con caratteristiche e punti vita

    5.1 Il personaggio è caratterizzato da

        5.1.1 Punti vita

        5.1.2 Tempo invulnerabilità dopo essere stato colpito

        5.1.3 Velocità di movimento

        5.1.4 Rate dello sparo

        5.1.5 Danno dei proiettili

        5.1.6 Velocità dei proiettili

        5.1.7 Range dei proiettili

    5.2 Il personaggio è controllato dall'utente con la tastiera

        5.2.1 L'utente può muovere il personaggio nelle quattro direzioni principali (sopra, sotto, destra, sinistra)

- 5.2.2 L'utente può sparare uno o più colpi verso una delle quattro direzioni principali (sopra, sotto, destra, sinistra)
- 5.2.3 Il personaggio può uscire dalla stanza attraverso le porte

5.3 Il personaggio è visivamente costituito da

- 5.3.1 Una testa orientata in modo da rappresentare la direzione dello sparo, se il personaggio non sta sparando allora è orientata nella direzione di movimento
- 5.3.2 Un corpo orientato ed animato in modo da rappresentare la direzione di movimento

5.4 Il personaggio infligge danno ai nemici quando un suo colpo colpisce un nemico

5.5 Il personaggio può raccogliere oggetti che modificano le sue caratteristiche

5.6 Il personaggio muore e la partita termina mostrando un messaggio "game over" quando la sua vita arriva a zero

6. Nemici con diverse caratteristiche e comportamenti

6.1 Un nemico può essere nello stato

- 6.1.1 Inattivo: non esegue azioni
- 6.1.2 Attacco: esegue azioni di attacco nei confronti del personaggio
- 6.1.3 Difesa: esegue azioni per difendersi dal personaggio
- 6.1.4 Nascosto: esegue azioni per nascondersi dal personaggio

6.2 Ogni nemico è caratterizzato da

- 6.2.1 Punti vita
- 6.2.2 Velocità di movimento
- 6.2.3 Danno da contatto
- 6.2.4 Se il nemico spara
  - 6.2.4.1 Rate dello sparo
  - 6.2.4.2 Danno dei proiettili
  - 6.2.4.3 Velocità dei proiettili
  - 6.2.4.4 Range dei proiettili

6.3 Un nemico infligge danno al personaggio quando vi entra in collisione

6.4 Un nemico che spara infligge danno al personaggio quando un suo colpo vi entra in collisione

6.5 Un nemico se si muove lo fa solo all'interno di una sola stanza

6.6 Un nemico muore quando la sua vita arriva a zero, al suo posto viene visualizzata un esplosione animata

6.7 Quattro tipologie di nemico

6.7.1 Nerve

6.7.1.1 Sta sempre nello stato di attacco

6.7.1.2 Sta immobile

6.7.1.3 Costituito da un corpo

6.7.2 Boney

6.7.2.1 Sta sempre nello stato di attacco

6.7.2.2 Si sposta continuamente in direzione del personaggio

6.7.2.3 Costituito da testa e corpo orientati nella direzione di movimento

6.7.3 Mask

6.7.3.1 Sta sempre nello stato di attacco

6.7.3.2 Si sposta mantenendo una distanza predefinita dal personaggio

6.7.3.3 Spara colpi in direzione del personaggio

6.7.3.4 Costituito da una testa orientata nella direzione di sparo

6.7.4 Parabite

6.7.4.1 Sta 1 secondo nello stato "inattivo", poi passa allo stato di "attacco", poi allo stato "nascosto" per 2 secondi poi ricomincia da capo

6.7.4.2 Quando passa da inattivo ad attacco calcola una linea retta in direzione del personaggio e la percorre velocemente

6.7.4.3 Quando passa allo stato "nascosto" può essere attraversato da personaggio o altri nemici e non può essere colpito

6.7.4.4 Costituito da un corpo orientato nella direzione di movimento

6.8 Boss

6.8.1 E' un nemico speciale che se viene sconfitto la partita termina visualizzando al giocatore il messaggio "hai vinto"

6.8.2 Il suo comportamento è determinato da 5 diverse azioni scelte casualmente con una certa probabilità in base al suo stato di vita e quello del personaggio.

6.8.2.1 Attacco con singolo proiettile in direzione del personaggio se questo si trova sullo stesso asse x o y all'interno della stanza

6.8.2.2 Attacco che spara 4 proiettili in contemporanea: sopra, sotto, destra e sinistra

6.8.2.3 Attacco che spara 4 proiettili in contemporanea in diagonale: sopra, sotto, destra e sinistra

6.8.2.4 Si sposta verso il personaggio per aggredirlo

6.8.2.5 Si teletrasporta in un punto della stanza specchiato rispetto a quello attuale. Se il punto è occupato da un elemento bloccante, trova il primo posto libero nel suo intorno

6.8.3 Costituito da un corpo

7. 10 Oggetti che il personaggio può raccogliere per incrementare le proprie caratteristiche

7.1 Arrow: incermenta il rate dello sparo

7.2 Drop: incermenta il danno dei proiettili

7.3 Eye: incrementa il danno dei proiettili ed il rate dello sparo

7.4 Fireball: incrementa il danno e la velocità dei proiettili

7.5 Glasses: incrementa il range dei proiettili

7.6 Heart: incrementa la vita massima di 1

7.7 Juice: incrementa la velocità di spostamento e danno dei proiettili

7.8 Mushroom: incrementa la vita massima di 2

7.9 Syringe: incrementa la velocità di spostamento, il rate dell sparo e la velocità dei proiettili

7.10 Tail: incrementa il danno ed il range dei proiettili

2.4 Requisiti non funzionali

1. Fluidità di gioco

1.1 Deve essere garantita una velocita di almeno 30 FPS su processore Apple M1 in situazioni di gioco "caotiche", dove per caotico si intende la presenza all'interno di una stanza del Character comandato dal giocatore, 5 nemici, 10 shot e 10 elementi bloccanti.

2.5 Requisiti di implementazione

1. Implementazione mediante Scala 3
2. Implementazione mediante Prolog della generazione del dungeon, delle stanze combattimento e del comportamento del Boss
3. Testing mediante ScalaTest
4. Sviluppo del software in modo da considerare possibili future estensioni di gioco come:

    4.1 Aggiunta di diversi personaggi con cui giocatore
    4.2 Aggiunta di nuovi nemici e boss
    4.3 Aggiunta di nuovi elementi bloccanti
    4.4 Aggiunta di nuovi oggetti con diverse caratteristiche
5. Rilascio del gioco su server tramite GitHub Actions

3 Design architetturale

Di seguito vengono analizzate l'architettura complessiva dell'applicazione e le decisioni critiche riguardanti pattern architetturali e tecnologie rilevanti. Prima di pensare a qualsiasi architettura per il gioco da realizzare, ciò che si è cercato di fare è stato individuare l'engine ideale per il progetto. La scelta è ricaduta su Indigo in quanto:

- E' un framework specifico per il game development
- E' scritto in Scala 3 e favorisce lo sviluppo tramite paradigma funzionale con un approccio unidirezionale ed immutabile al flusso di dati
- Permette il gioco su browser, come da requisito di business, mediante compilazione tramite Scala.js.

I concetti chiave di Indigo che ci guidano nello sviluppo del gioco sono

- Paradigma Model, View, ViewModel
- Game loop con modello ad eventi
- Scene

3.1 Paradigma Model-View-ViewModel

Il pattern architetturale **MVVM** (Model, View, ViewModel) è una variante del famoso MVC. Obiettivo di questo pattern è quello di separare il modello dei dati dalla sua rappresentazione, tenendo entrambi puri e funzionali al loro scopo, frapponendo tra loro un elemento "ibrido", il ViewModel. Di seguito l'interpretazione sulla quale si basa Indigo e che seguiamo per lo sviluppo del nostro gioco.

Model Rappresenta il modello di gioco puro, indipendentemente dalla sua rappresentazione visuale. Esso contiene lo stato del gioco e la sua logica.

ViewModel Si interpone tra Model e View. Ha lo scopo di mantenere alcuni dati utili per la rappresentazione che tuttavia non concernono la logica di gioco, devono perciò rimanere separati e non essere inclusi all'interno del Model.

View Si occupa di offrire una rappresentazione del Model e del ViewModel a schermo. Costituisce quindi la logica di visualizzazione di questi, senza contenere però alcuno stato o logica di gioco.

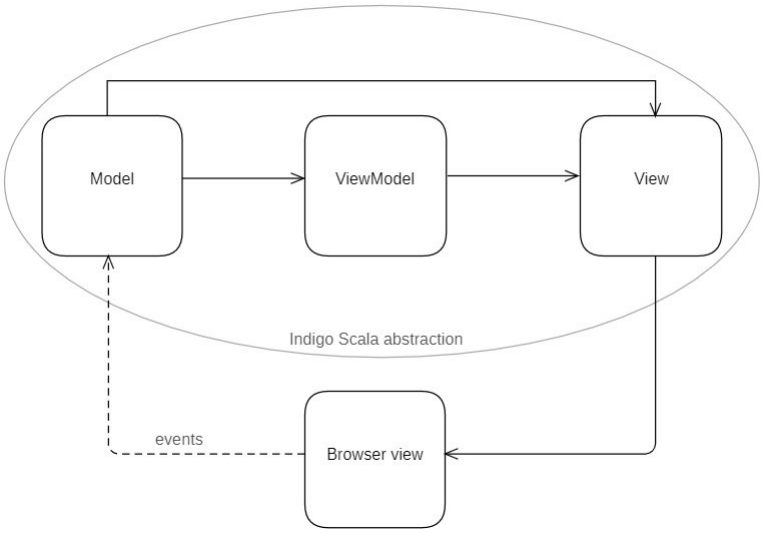

Figura 1: *Architettura Model, View, ViewModel*

3.2 Game loop con modello ad eventi

La logica di Indigo si basa su un game loop che processa, ad ogni iterazione, una coda di eventi. Ad ogni ciclo, il framework esegue le seguenti operazioni:

- Aggiunta dell'evento FrameTick alla coda eventi
- Per ogni evento, update del Model
- Per ogni evento, update del ViewModel
- Presentazione e rendering degli elementi a video
- Reset della coda di eventi

Il concetto di **immutabilità**, presente nel paradigma funzionale, qui si traduce non in un update di uno stato interno del modello, ma bensì nella generazione di un nuovo modello aggiornato.

Durante l'update del Model è possibile generare eventi custom che verranno processati all'iterazione successiva: a tale scopo Indigo fornisce la **monade Outcome** utile a wrappare dati ed eventi.

3.3 Scene

Le scene sono un modo di organizzare il codice secondo una logica di gioco ben definita. Sono un meccanismo di suddivisione che permette di individuare delle "fasi" di gioco da

sviluppare in modo separato le une dalle altre. In ogni istante all'interno del gioco la scena indicata come "corrente" è abilitata all'aggiornamento del Model/ViewModel, alla gestione degli eventi in coda ed infine alla presentazione a schermo: la scena corrente può leggere e aggiornare solo i suoi dati che sono un sottoinsieme di quelli globali.

Abbiamo identificato quattro scene principali su cui andare a definire Lost 'n Souls.

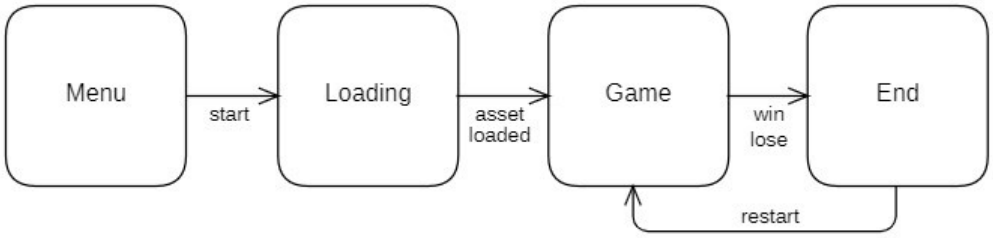

Figura 2: *Il flusso delle scene*

3.4 Architettutra di Model e ViewModel

Come conseguenza della suddivisione di cui sopra, il Model principale è costituito dai singoli Model relativi a ciascuna scena. Stessa cosa vale per il ViewModel.

Da notare che una scena, per la sua presentazione a video, può non necessitare di un Model o di un ViewModel, lo stesso concetto si può estendere per una View qualsiasi la quale potrebbe basarsi solo su Model.

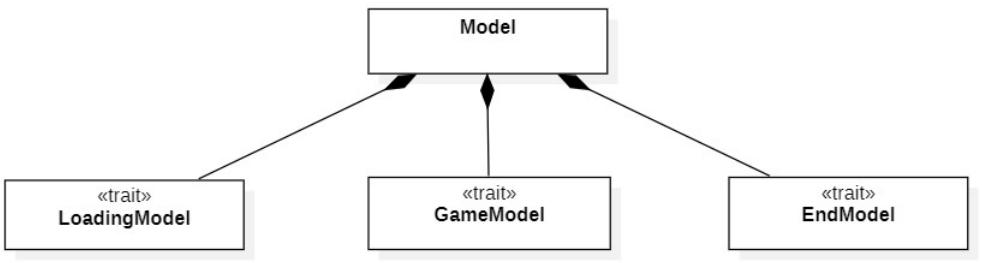

Figura 3: *Il Model*

3.5 Elementi principali del GameModel

All'interno della scena di gioco, abbiamo identificato i principali componenti:

- **Character** Personaggio controllato dal giocatore
- **Dungeon** Intera mappa di gioco nel suo complesso
- **Room** Stanza situata all'interno di una mappa di gioco
- **Anything** Una qualsiasi entità da visualizzare all'interno di una stanza: nemico, oggetto, elemento bloccante o anche lo stesso Character.

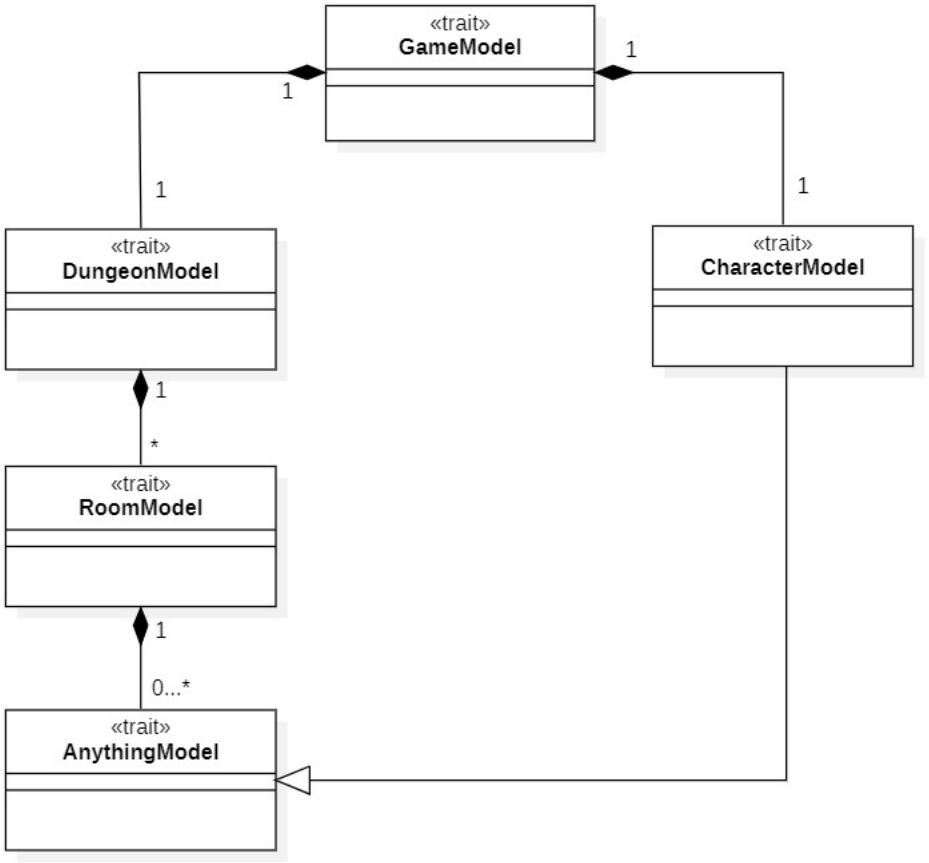

Figura 4: *Diagramma degli elementi principali del GameModel*

3.6 Anything OOP anziché pattern ECS

Per concludere la descrizione dell'architettura occorre specificare come organizzare gli oggetti presenti nel gioco, quelli che abbiamo chiamato **Anything**. Un oggetto è caratterizzato da proprietà e comportamenti comuni ad altri, ad esempio tutti gli oggetti sono posizionabili nella stanza di gioco e visualizzabili mentre alcuni di questi possono spostarsi, collidere, causare danno e così via.

Tradizionalmente molto usato nel game development è il pattern Entities-Components-Systems che prevede la sostituzione di una gerarchia di oggetti con il concetto di entità e componenti: le entità sono definite dai diversi componenti a loro associati. Alcune implementazioni di ECS definiscono i componenti come solo strutture dati mentre il comportamento è specificato nel sistema che li gestisce, quindi in stile funzionale. Questo pattern è nato per diversi motivi ma quello più banale è che, quando i linguaggi non supportano l'ereditarietà multipla, è impossibile specificare una precisa gerarchia di oggetti che soddisfi i requisiti di un gioco di una certa complessità, o comunque questa gerarchia sarebbe molto profonda e poco flessibile a future estensioni.

Quello che vogliamo fare in questo progetto è sfruttare appieno le potenzialità in ambito **OOP** che ci offre Scala, il quale ci permette tramite i **mix-ins** di appiattire la gerarchia ed ottenere la flessibilità richiesta dal problema. Quindi abbiamo appositamente scelto il nome Anything in sostituzione al classico Entity per specificare che non stiamo applicando il pattern ECS, un sottotipo di Anything rappresenta una nostra entità che può essere mixata con altri tratti per ottenere nuove proprietà e comportamenti.

Visto l'impiego del pattern MVVM, occorre inoltre tenere ben presente che un Anything deve essere in realtà codificato in tre parti ben distinte Model-ViewModel-View ciascuna sottotipo rispettivamente di **AnythingModel**, **AnythingViewModel** e **AnythingView** e ciascuna eventualmente mixata come descritto precedentemente. Inoltre è da specificare che ogni oggetto in gioco è costitutito sempre da un Model e una View ma quest'ultima potrebbe non necessitare di un ViewModel.

Il Model di un oggetto come comportamento base deve reagire alle richieste di aggiornamento che avvengono ad ogni iterazione del game loop: questa richiesta deve essere propagata a tutta la catena di ereditarietà e ai vari oggetti mixati ed infine restituire un nuovo Model aggiornato. Sebbene gran parte del comportamento e della logica di gioco è così incapsulata nei Model degli oggetti, dovremmo comunque elaborare dei sistemi che avranno una visione globale gestendo l'interazione tra diversi Model, come un sistema per controllare le collisioni tra oggetti.

Per concludere, la nostra soluzione OO non è la migliore soluzione possibile per lo sviluppo di un gioco. Il pattern ECS ha vantaggi prestazionali riconosciuti sia in termini di utilizzo CPU (meno calls per l'aggiornamento degli oggetti) che di ottimizzazione dell'utilizzo della memoria (data-oriented design), tuttavia a fini didattici vogliamo sperimentare elementi di Scala OOP avanzati.

3.7 Prolog engine e PrologService

Il motore Prolog da utilizzare per la generazione del dungeon, delle stanze combattimento e del comportamento del Boss deve potersi integrare ed interagire con Indigo. Poichè Indigo compila con Scala.js e per evitare possibili problemi con engine scritti in Java/-Scala, la scelta di quale motore Prolog usare ricade su **TauProlog**, un engine scritto in Javascript e quindi facilmente integrabile nel nostro contesto.

Tuttavia per comunicare con questo motore occorre utilizzare la **programmazione asincrona** in quanto è strutturato in modo da rispondere alle interrogazioni mediante una **callback** che deve essere fornita. Questa modalità di interazione non è supportata direttamente da Indigo, il quale con il suo game loop supporta come input eventi/messaggi che però possono essere generati solo internamente durante le fasi di update del Model: l'attivazione di una callback asyncrona fuori dal game loop non può ne modificare lo stato del Model che è immutabile ne generare un evento.

Per questo si rende necessario lo sviluppo di un componente separato che faccia da proxy per l'engine TauProlog e risponda alle richieste del gioco con opportuni eventi customizzati.

Indigo dispone dei **SubSystems**, servizi con un proprio Model ed esterni alla logica di gioco ma che vengono attivati nel game loop per intercettare eventi ed eseguire azioni: occorre quindi predispore un **PrologService** subsystem.

In figura sotto è mostrata l'interazione tra i sistemi: da notare che la callback chiamata da TauProlog necessita di scrivere dati nel Model del PrologService invalidando così il concetto di immutabilità, ma solo per quel servizio esterno al gioco ed il tutto senza introdurre alcun problema di concorrenza in quanto ogni esecuzione di procedura è atomica nel contesto dell'**event loop** di Javascript.

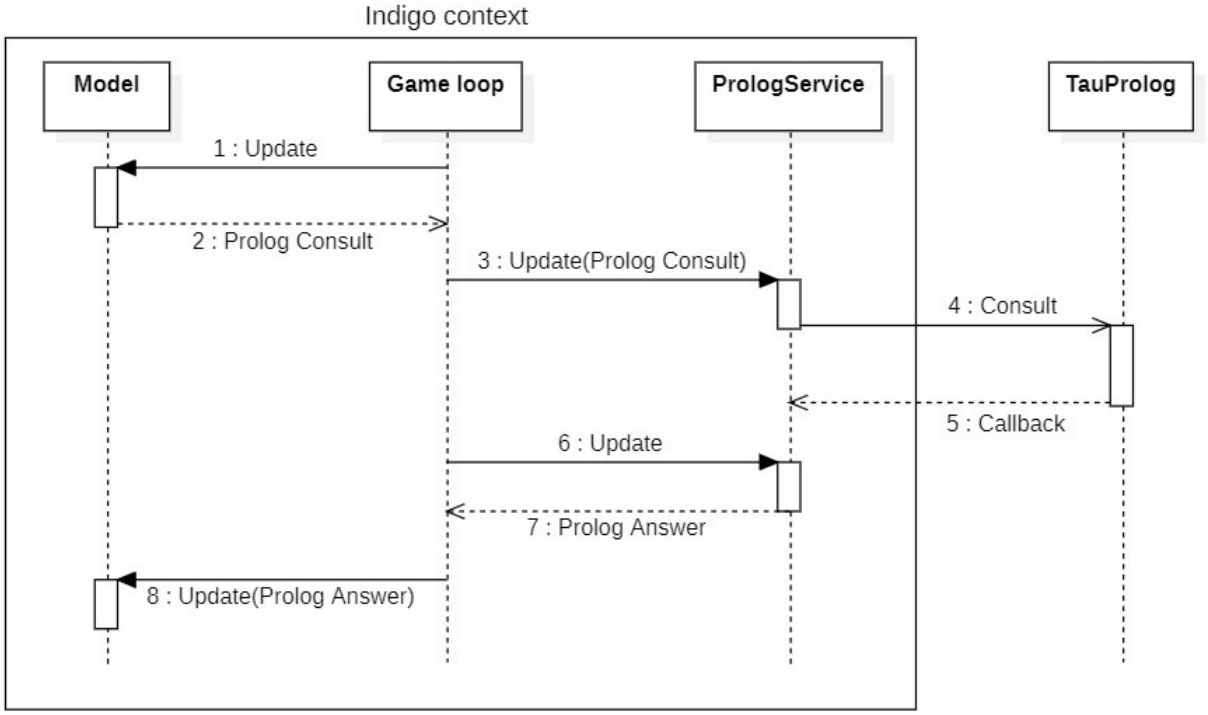

Figura 5: *Diagramma dell'interazione con TauProlog*

4 Design di dettaglio

Di seguito verranno analizzate le scelte di design rilevanti per il sistema, i pattern di progettazione utilizzati e l'organizzazione del codice.

4.1 GameModel

L'intero modello di gioco, dallo stato al comportamento, è incapsulato all'interno della struttura GameModel. E' possibile identificare due diverse accezioni del modello, a seconda del momento di gioco. Un momento in cui la partita non è ancora stata avviata e un momento successivo in cui invece la partita è avviata. Nel primo momento, a partita non avviata, è necessario generare il dungeon e le stanze che lo compongono. Nel secondo momento, una volta che è avvenuta la generazione, il modello è costituito da un Dungeon composto da diverse stanze e dal character controllato dal giocatore.

Il GameModel, nel caso di partita avviata, è dotato di metodi per aggiornare le sue varie componenti: il character, il dungeon, la stanza corrente e tutti gli Anything interni. Trattandosi di una struttura dati immutabile, aggiornare il modello significa creare un nuovo modello aggiornato. Per situazioni complesse, come il sistema di controllo collisioni e il sistema che verifica e gestisce il passaggio del character da una stanza all'altra, il GameModel sfrutta moduli separati, definiti *PassageUpdater e CollisionUpdater*

Una scelta rilevante ai fini dell'implementazione è il fatto che, a livello di modello, il character controllato dal giocatore è sempre fuori dalla stanza in cui si trova, in questo modo è possibile aggiornare in maniera indipendente le due componenti.

In figura 6 è mostrato il relativo diagramma delle classi.

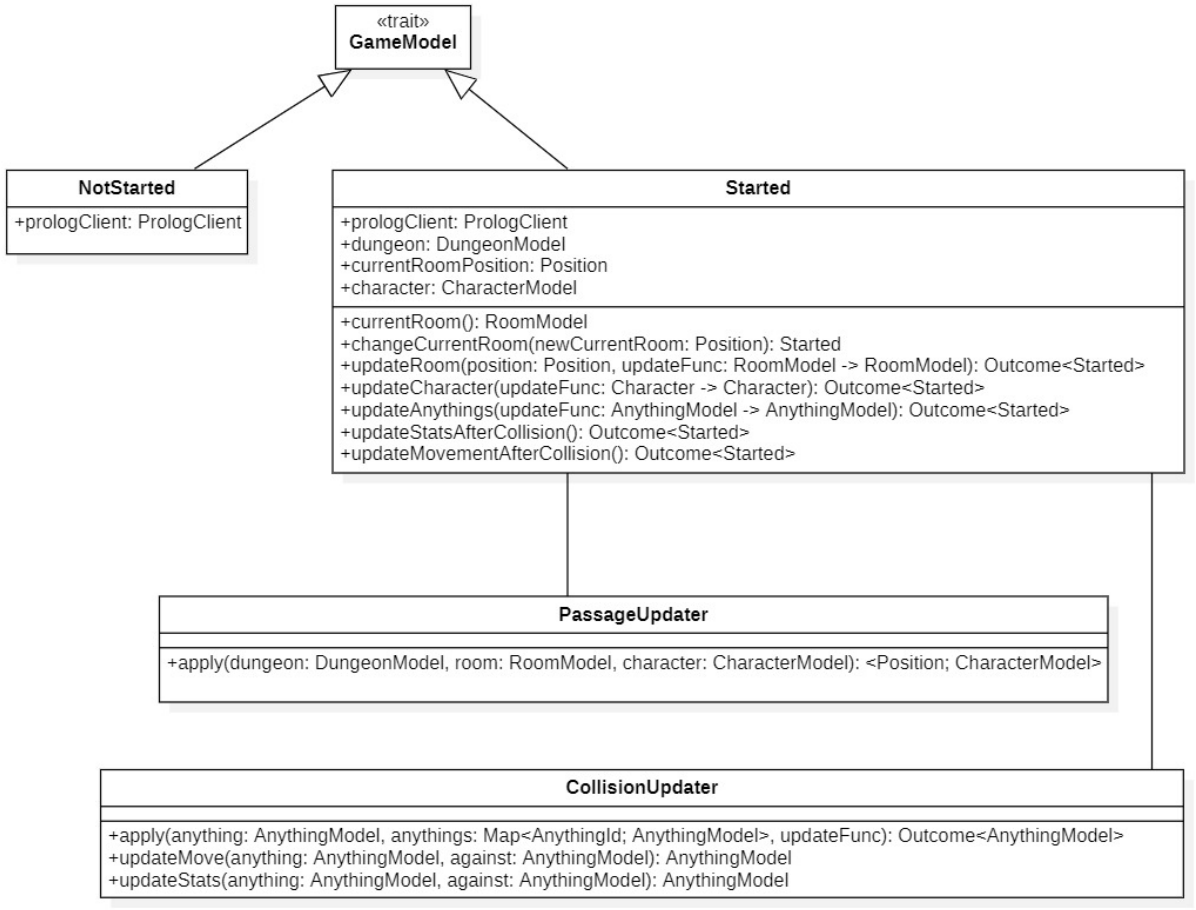

Figura 6: *Diagramma delle classi del Game Model e componenti rilevanti*

4.2 Dungeon

Il dungeon è composto da stanze disposte in maniera randomica ma collegate tra loro. La struttura Grid è ciò che descrive un Dungeon nella sua forma più basilare e geometrica e offre delle operazioni per esplorarlo.

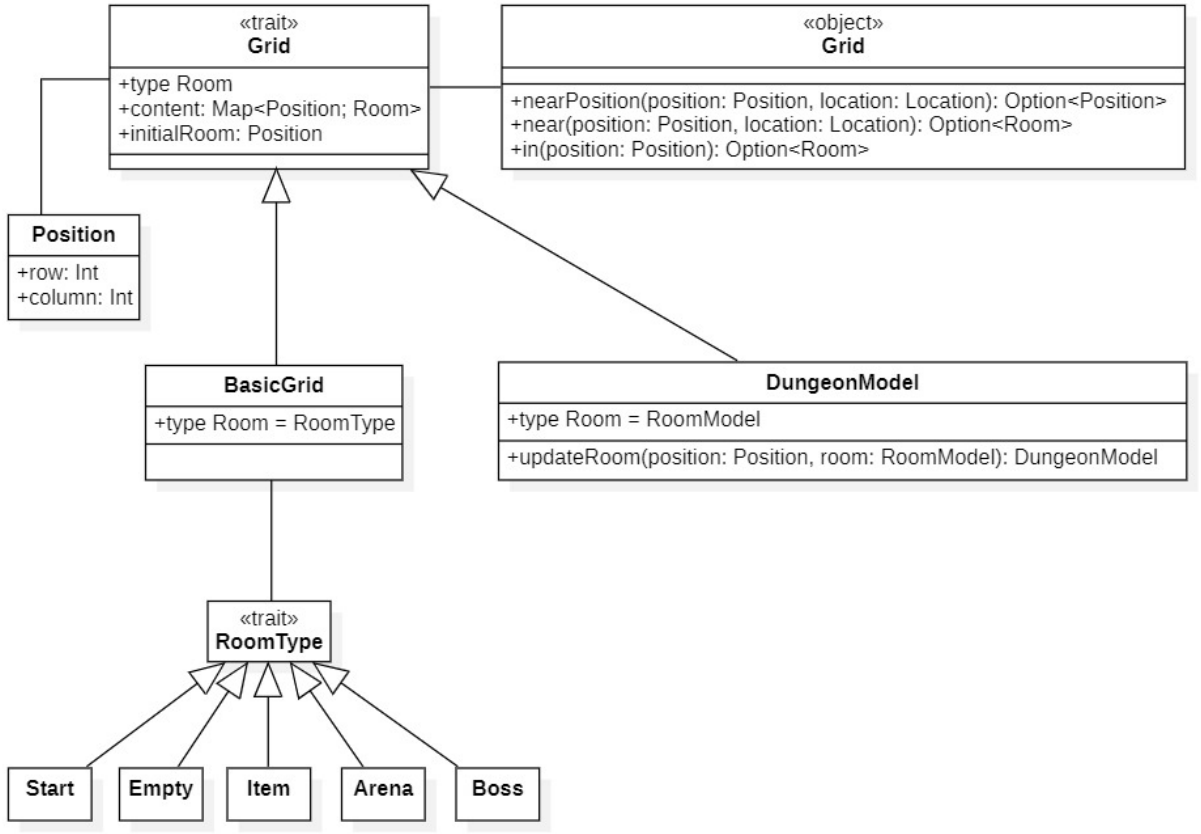

Figura 7: *Diagramma delle classi del Dungeon*

Grid definisce quindi una collezione di stanze, di tipo ancora non definito, in una specifica posizione all'interno di una griglia di dimensioni nXn. La definizione del tipo Room è lasciato alle specializzazioni di Grid, si mette quindi in pratica il pattern **Family Polimorfism**.

All'interno della griglia, un elemento potrebbe o non potrebbe essere presente, andando così a definire un dungeon con stanze in posizioni diverse. E' necessario perciò lavorare con strutture dati che consentano l'assenza di un dato.

Il Dungeon vero e proprio prende forma quando il tipo Room viene definito con un modello consono a descriverne lo stato e il comportamento.

4.3 Room

Come da requisiti, una stanza è la componente principale del Dungeon e può essere di diversi tipi: Empty, Arena, Item e Boss. Ogni stanza incapsula una collezione di elementi (Anything), i quali sono propri di essa e non possono muoversi all'interno del Dungeon, come ad esempio i nemici o gli elementi bloccanti.

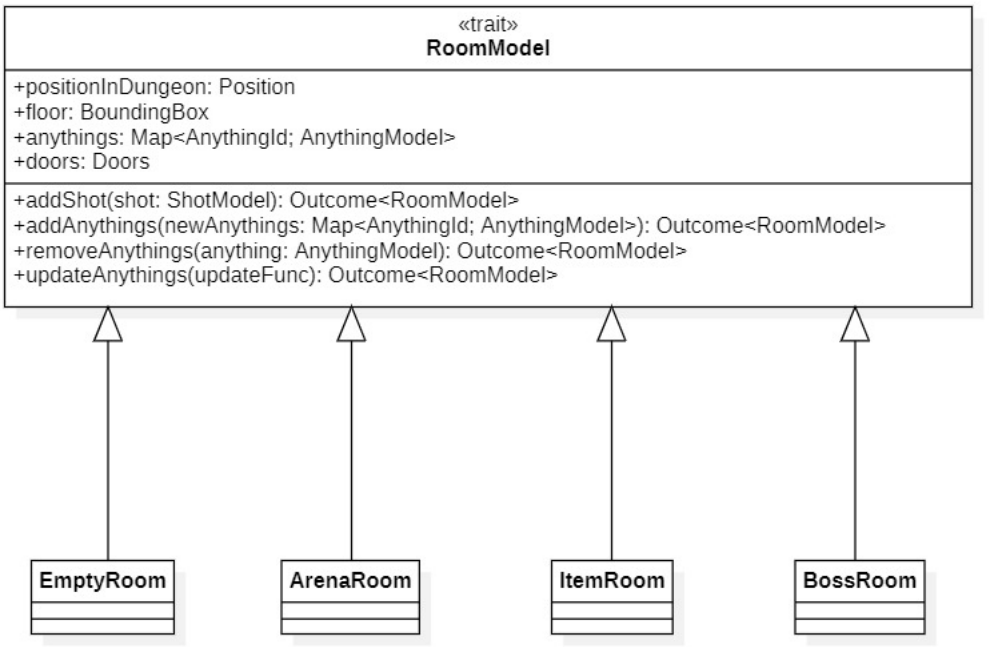

Figura 8: *Diagramma delle classi Room*

Una stanza è anche responsabile dell'aggiornamento di tutto ciò che incapsula. Anche in questo caso si cerca di lavorare con strutture dati immutabili, perciò l'aggiornamento di uno o più componenti consegue una nuova stanza aggiornata. L'aggiornamento viene fatto ad ogni FrameTick ed è comandato dal dungeon, solamente per la stanza correntemente visualizzata, le altre stanze non sono aggiornate o gestite.

4.3.1 Door

Un dettaglio importante di una stanza sono le porte. Una porta, per l'utente, rappresenta il collegamento tra una stanza e un altra, ma a livello di dominio sono semplicemente l'associazione di un lato della stanza con uno stato *open, closed, locked*.

Una porta è quindi un elemento statico di una stanza e non un link. E' il Dungeon a gestire il collegamento tra le varie stanze in base alla loro posizione occupata in griglia.

Il fatto che vi sia una porta, presuppone comunque che dalla parte opposta vi sia un'altra stanza, ma questo dettaglio è trasparente alla stanza e gestita dal Dungeon.

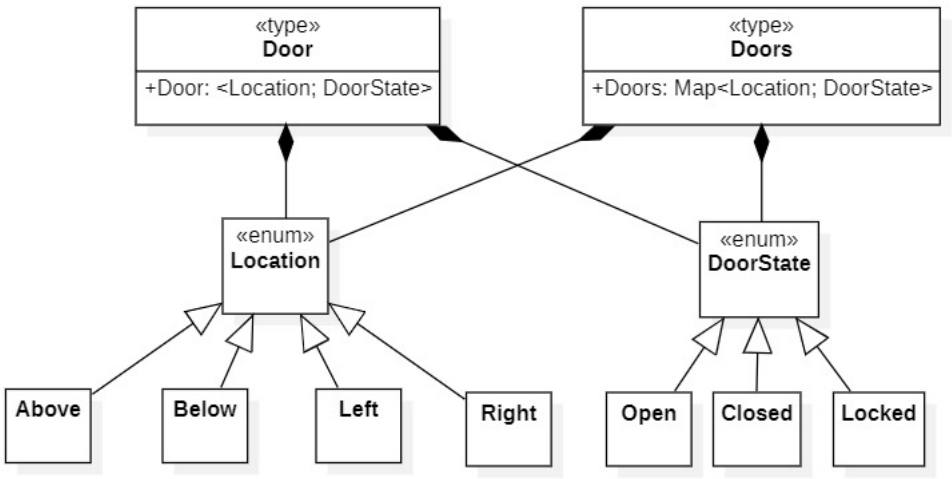

Figura 9: *Diagramma delle classi Door*

4.4 AnythingModel

In generale ogni oggetto in gioco contiene un identificativo univoco e una bounding box che ne determina posizione e dimensioni 2D, inoltre ad ogni istanza è associata la factory per la sua View in modo tale che, quando una collezione di AnythingModel viene renderizzata, è sufficiente creare la View e chiamare il metodo draw.

Ogni oggetto inoltre reagisce alla richiesta di update che avviene ad ogni iterazione del game loop, ma il modo con cui lo fa dipende da specifici comportamenti definiti come sottotipi mixabili di AnythingModel.

I sottotipi principali sono:

- **DynamicModel** un oggetto in grado di spostarsi autonomamente durante un update, dove e come viene stabilito dalla classe che esegue il mixing attraverso l'implementazione di alcuni template-method predisposti
- **AliveModel** un oggetto vivo che può essere colpito perdendo vita. Non ha comportamento autonomo.
- **DamageModel** un oggetto che provoca danno di contatto. Non ha comportamento autonomo.
- **FireModel** un oggetto in grado di sparare autonomamente durante un update, dove e come viene stabilito dalla classe che esegue il mixing attraverso l'implementazione di alcuni template-method predisposti. Durante l'update può emettere uno ShotEvent contenente ShotModel
- **SolidModel** un oggetto solido che non può essere fisicamente attraversato da un altro. Non ha comportamento autonomo.

Molti oggetti all'interno del gioco hanno delle caratteristiche che ne influenzano il comportamento: punti vita, tempo invulnerabilità dopo essere stato colpito, velocità di movimento, rate dello sparo, danno dei proiettili, velocità dei proiettili, range dei proiettili. Di conseguenza tutti i principali sottotipi elencati, tranne il SolidModel, vengono mixati con **StatsModel** al quale è delegata la gestione di queste caratteristiche: le Stats sono assegnate durante l'istanzazione di un Model ma possono cambiare durante il gioco.

Di seguito in figura 10 mostriamo un estratto della gerarchia di AnythingModel con tutti i sottotipi che aggiungono proprietà e comportamenti al Model di base e che possono essere estesi o mixati da un Model finale: abbiamo riportato evidenziati in giallo alcuni utilizzatori concreti.

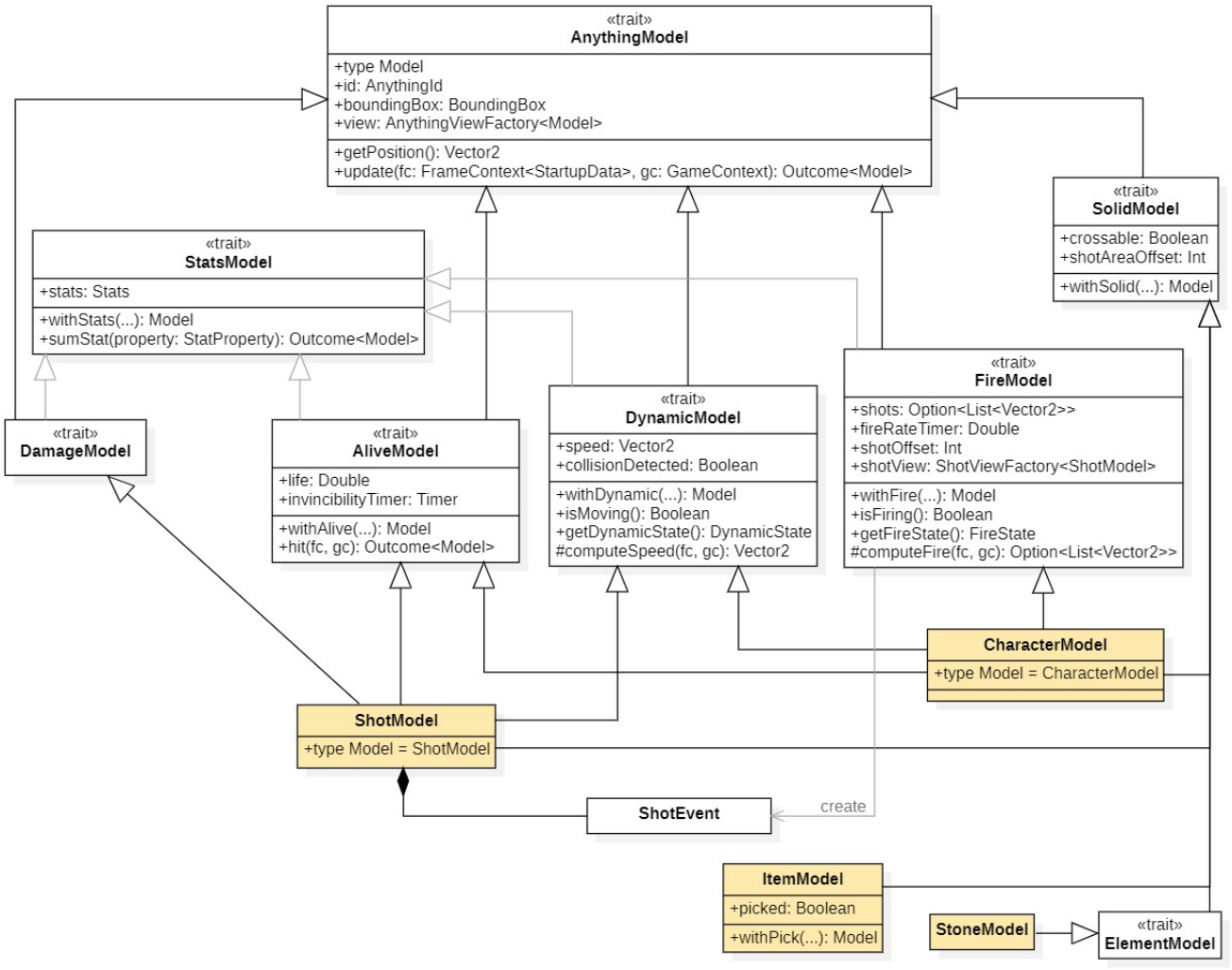

Figura 10: *Gerarchia AnythingModel - in giallo alcuni utilizzi concreti*

4.4.1 Update e immutabilità

Ogni metodo che applica modifiche ad un Model, a partire da *update* che viene richiamata ad ogni iterazione del game loop, deve ritornare una **copia aggiornata** dello stesso e del **tipo corrente**: adottiamo il pattern **F-Bounded Polymorphism con type-member**. Allo scopo predisponiamo un type member astratto *Model* e, per ogni trait sottotipo di AnythingModel, un template method come with*Comportamento*(...) che accetti come argomenti quelli da modificare e ritorni un nuovo oggetto aggiornato di tipo *Model*: entrambi da definire nella classe finale che mixa il trait. E' richiesto che il metodo update richiami super.update in modo da propagare l'aggiornamento a tutti

i trait mixati. *Si rimanda all'implementazione per il dettaglio di come il pattern viene applicato con Scala.*

Per concludere, il metodo update riceve in input, oltre al FrameContext di Indigo, il **GameContext** che contiene lo stato della stanza corrente e il CharacterModel in modo da permettere all'oggetto in gioco comportamenti che si adattano alla situazione.

Nota Sebbene una soluzione consigliata in sostituzione al polimorfismo F-Bounded è quella di impiegare polimorfismo ad-hoc con typeclass, abbiamo riscontrato che nel nostro caso volendo mantenere il mixing degli oggetti questo pattern non è facilmente applicabile e avrebbe generato più boilerplate code.

Nota Abbiamo deciso di inglobare il comportamento nei Model con il metodo update, ma un alternativa forse migliore e più semplice era delegarlo a sistemi esterni, i quali si sarebbero occupati ad esempio di spostare gli oggetti, farli sparare, etc: in questo progetto abbiamo cercato di emulare un modello ad agenti, oltre che garantirci la massima possibilità di personalizzazione del comportamento soprattutto per quanto riguarda i nemici.

4.4.2 Gestione degli spari

Riguardo a **FireModel** e agli spari **ShotModel** occorre specificare che:

- possono essere sparati più colpi ShotModel contemporaneamente in direzioni diverse
- tra un colpo e l'altro deve trascorrere del tempo in base alla Stats "rate dello sparo"
- occorre generare il colpo a certo un offset rispetto alla posizione del oggetto che sta separando
- occorre specificare una ShotView factory da passare allo ShotModel, infatti in questo caso sono disponibili più View da utilizzare

FireModel durante un update genera l'evento **ShotEvent** indicando le proprietà dello ShotModel da istanziare (tra cui velocità e danno) ed inserendolo nella monade Outcome di ritorno: verrà gestito nell'iterazione successiva del game loop e aggiunto al RoomModel.

Gli ShotModel sono oggetti che a loro volta mixano alcuni dei principali sottotipi Anything:

- DynamicModel per gestire il movimento ed il suo range massimo
- AliveModel in quanto ha una vita fittizia che viene azzerata dopo una collisione o se ha termitato il range

- SolidModel per gestire le collisioni
- DamageModel per arrecare danno contro cui avviene una eventuale collisione.

4.4.3 Mini framework per i nemici

Abbiamo pensato ad un mini framework che consenta la creazione rapida di nuovi nemici in modo da soddisfare i requisiti di possibili future espansioni. Al centro abbiamo **EnemyModel**, il quale dispone di una coda di stati temporizzati che vengono processati uno dopo l'altro: questo permette la realizzazione di nemici che eseguono una sequenza di azioni.

Sono stati definiti dei comportamenti mixabili elementari come:

- **Follower**: un nemico che insegue il character
- **FiresContinuosly**: un nemico che spara in continuazione nella direzione del character
- **KeepsAway**: un nemico che si mantiene a distanza dal giocatore
- **Traveller**: un nemico che segue un percorso indicato da una sequenza di punti

Abbiamo pensato di esprimerli come strategie da ri-utilizzare, quindi come mixin puri mediante il meccanismo dei **Self-types**, anzichè renderli degli EnemyModel ereditando da questo: il concetto è che se si vuole sviluppare un nemico si estende direttamente EnemyModel e non uno di questi comportamenti "plugin".

Di seguito in figura 11 mostriamo come sono stati definiti i nemici a livello di Model, abbiamo evidenziato i 4 comportamenti base in azzurro e le classi finali che rappresentano i nemici implementati in giallo.

4.4.4 Boss

Realizzare il comportamento del Boss tramite linguaggio Prolog ci permette di usufruire della sua espressività logica.

Per fare questo è necessario ampliare il framework dei nemici aggiungendo il **PrologEnemyModel**, un tratto in grado di gestire le operazioni necessarie ad interrogare il PrologService attraverso uno specifico *goal* e gestire le sue risposte per attivare un certo *comportamento*. In questo modo, in ottica di sviluppi futuri, si potrebbero creare rapidamente nuovi nemici, boss compresi, che implementino il comportamento tramite linguaggio Prolog, definendo unicamente la costruzione del goal e del comportamento.

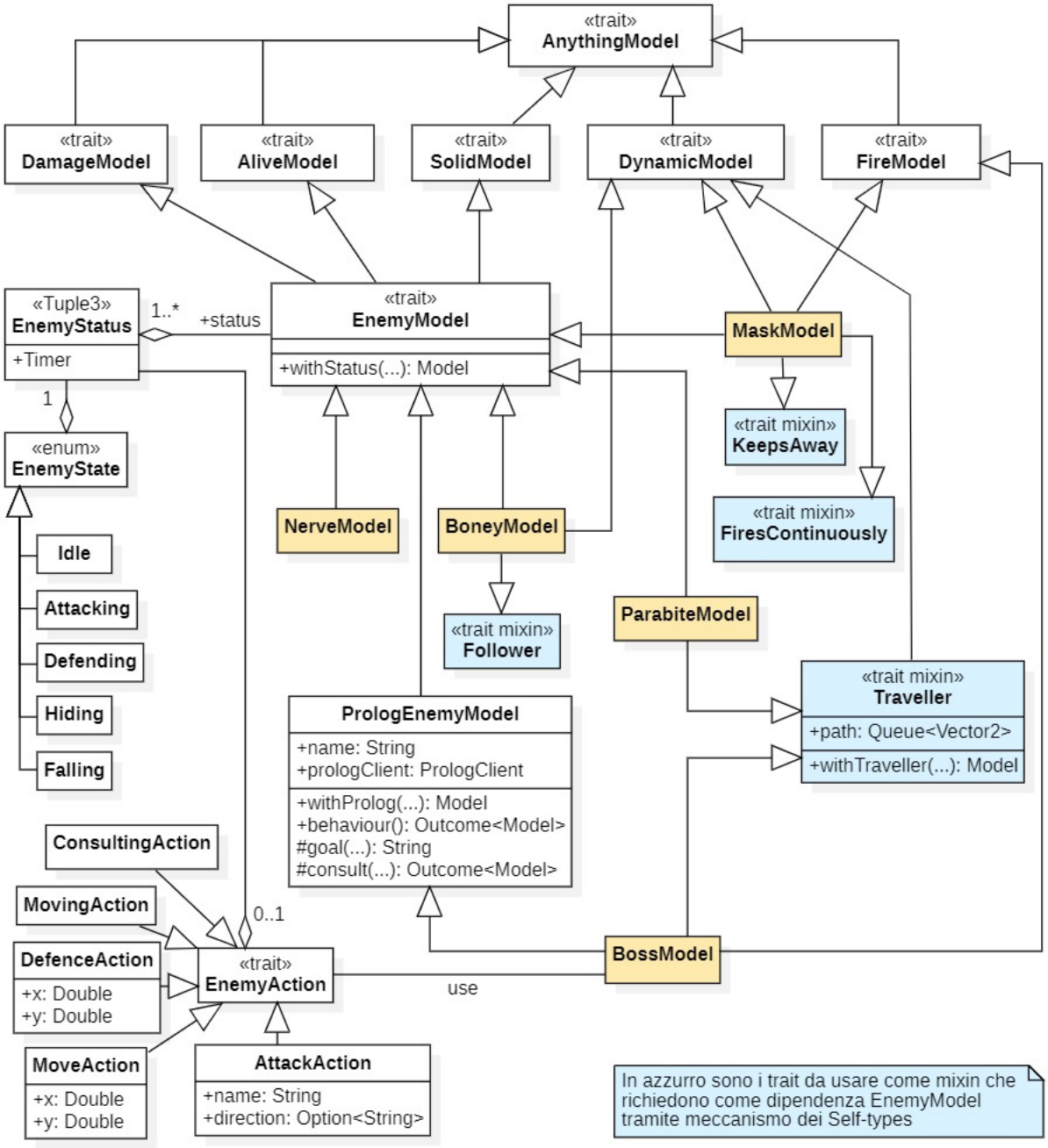

Figura 11: *Enemies framework - in giallo i nemici concreti*

4.5 AnythingView e AnythingViewModel

Come scelta di design stabiliamo che

- Una View e un ViewModel devono essere progettati per uno specifico Model.
- Un ViewModel potrebbe essere usato da diverse View
- Una View potrebbe non necessitare di un ViewModel: in generale quando non ha animazioni complesse
- Un ViewModel viene istanziato con lo stesso identificativo univoco usato per l'oggetto Model a cui si riferisce
- Un Model potrebbe disporre di diverse versioni di View ma la sua istanza ne utilizza una

Per la View abbiamo pensato di adottare il pattern **Family Polymorphism** definendo al suo interno i type member astratti **Model** e **ViewModel** che vengono concretizzati dai sottotipi della View. Nella versione finale tuttavia i due tipi interni vengono collegati a dei parametri generici in quanto abbiamo trovato che questa soluzione

- ci offre tutti i vantaggi di avere dei type member come la pulizia del codice o la possibilità di usare path-dependent types
- ci ha dato meno problemi di type inference
- ci consente di risolvere il problema di type test runtime su tipi astratti *(vedi approfondimento su implementazione)*
- a nostro avviso è più espressiva per indicare cosa richiede una View

Il pattern **Family Polymorphism** è stato applicato definendo il tipo astratto *View* che rappresenta il contenuto grafico da visualizzare attraverso il metodo **draw**, quest'ultimo richiama il template method **view** per ottenerlo.

In figura 12 mostriamo l'utilizzo degli **Union Types** per definire che un ViewModel può non essere richiesto da una View. Per la gerarchia di ViewModel valgono gli stessi concetti già menzionati per il Model di immutabilità e polimorfismo F-Bounded. L'esempio concreto di utilizzo in figura riguarda il Character, ma è definito almeno un singleton object View per ciascun Model.

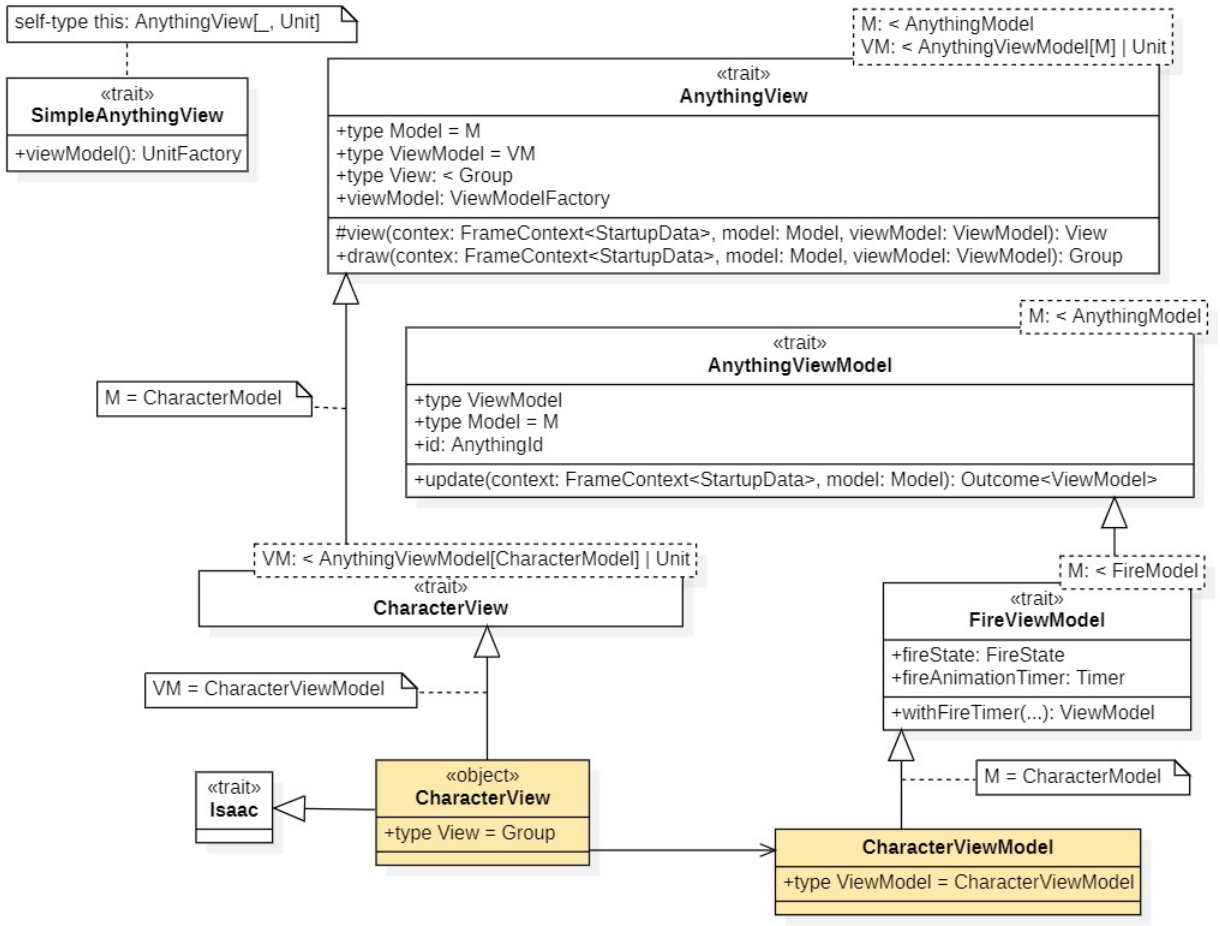

Figura 12: *Gerarchia AnythingView e AnythingViewModel*

4.5.1 Asset e animazioni

Per realizzare le View dei vari oggetti, è necessaria una buona organizzazione del codice per seguire i principi **DRY** e definire coerentemente come debba essere realizzata una visualizzazione, la quale dipende sempre da Asset grafici ed, in alcune situazioni, anche da animazioni.

L'obiettivo è quindi separare il *cosa* disegnare dal *come*, in modo tale da realizzare diversi trait per lo stesso oggetto senza mai replicare il codice in comune.

Sarà quindi necessaria una libreria **AnythingAssets** per codificare le informazioni di base dalla quale ogni singolo Asset (es IsaacAsset in figura 13) dovrà ereditare per definire le sue proprietà: nome del file, dimensioni, scala, ecc...

A questo punto si potrà creare un trait (es Isacc in figura 13) che definisca come viene

disegnato lo specifico asset, se questo è composto da una singola Sprite o da più di una, ad esempio il Character dovrà avere un corpo ed una testa con animazioni separate in base all'input dell'utente. La testa sarà animata soltanto durante lo sparo ed in direzione dei proiettili, mentre il corpo sarà animato durante il movimento e nella direzione di spostamento.

Una volta definito il trait dell'asset, questo può essere mixato con la corretta View del modello che rappresenta l'oggetto a schermo.

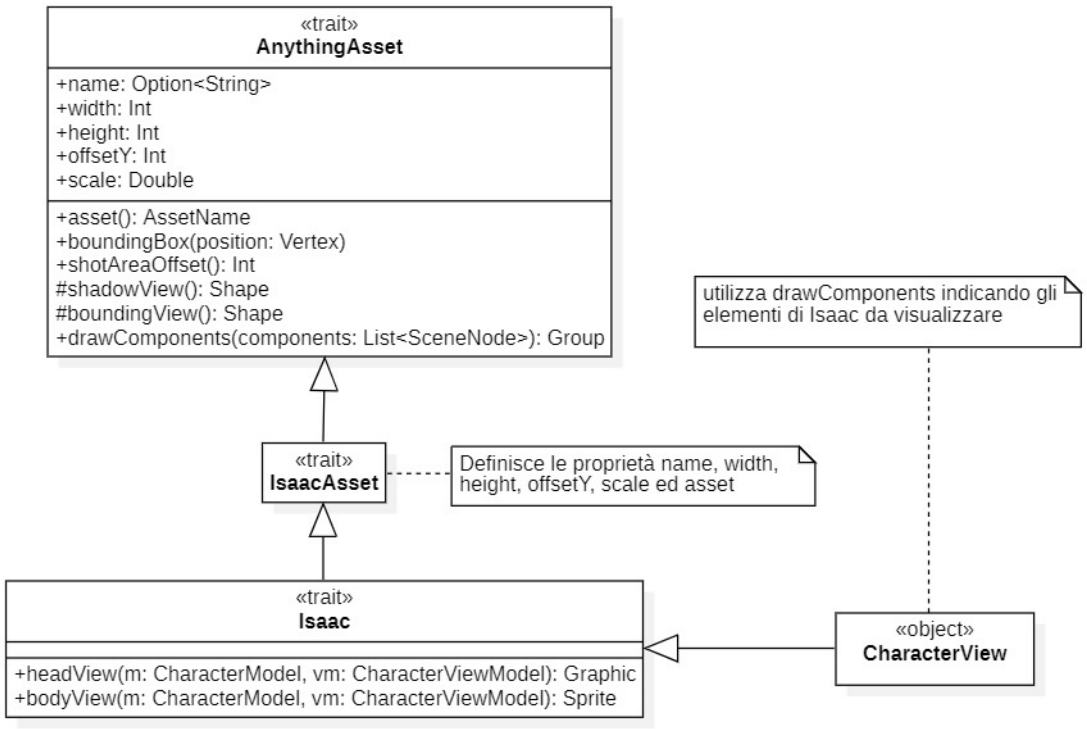

Figura 13: *AnythingAssets e utilizzo con Isaac*

4.6 Sistema di controllo delle collisioni

Il sistema di controllo delle collisioni ha l'obbiettivo di verificare se due elementi all'interno di una stanza collidono. Se questo accade, è necessario aggiornare la posizione degli elementi per evitare che gli elementi si intersechino e gestire un side effect che potrebbe o meno esserci durante la collisione (danno).

Il sistema agisce e controlla solamente gli Anything di tipo Solid all'interno della stanza correntemente visualizzata. Essendo il GameModel un'istantanea immutabile, il controllo avviene ad ogni Frame e su una struttura dati statica.

Il sistema è composto da una serie di funzioni racchiusi in un modulo e sfruttato dal GameModel stesso, il quale va a controllare gli Anything presenti nella Room corrente e ritorna un modello aggiornato, come si può vedere in figura 6.

Il controllo è eseguito in due momenti distinti:

1. In un primo momento viene verificata la collisione tra un elemento e tutti gli altri interni alla stanza. Per ogni collisione viene applicata una funzione all'elemento che corrisponde al side effect.

2. In un secondo momento vengono ricontrollate le collisioni e spostati gli elementi di conseguenza.

Il doppio controllo è dovuto al fatto che non è possibile spostare subito un elemento, perchè si potrebbero perdere collisioni con altri elementi. Questo aspetto, in termini di prestazioni, è largamente migliorabile, ma si rimanda a eventuali sviluppi futuri.

4.7 PrologService e PrologClient

Nel progettare il PrologService abbiamo considerato che:

- come servizio prolog deve accettare richieste dal gioco e restituire risposte sotto forma di eventi con payload

- deve comunicare con TauProlog in modo asincrono

- deve fornire al gioco un livello di astrazione da TauProlog: non vogliamo che il gioco riceva risposte con termini Tau

- non deve essere concepito specificamente per TauProlog ma per un qualsiasi altro engine asincrono.

Da queste considerazioni abbiamo definito un nostro insieme di Prolog **Term** ed abbiamo pensato PrologService in modo da interfacciarsi con un oggetto di tipo **AsyncSession** di cui **TauClient** è sottotipo concreto: quest'ultimo è il punto di contatto con TauProlog e si occupa di tradurre i TauTerm in Term.

Infine per facilitare l'utilizzo del Prolog da parte del gioco abbiamo predisposto **PrologClient** che emette comandi per il PrologService: PrologClient è predisposto per essere memorizzato nei Model e quindi ad ogni comando produce una copia di se stesso wrappata nella monade Outcome.

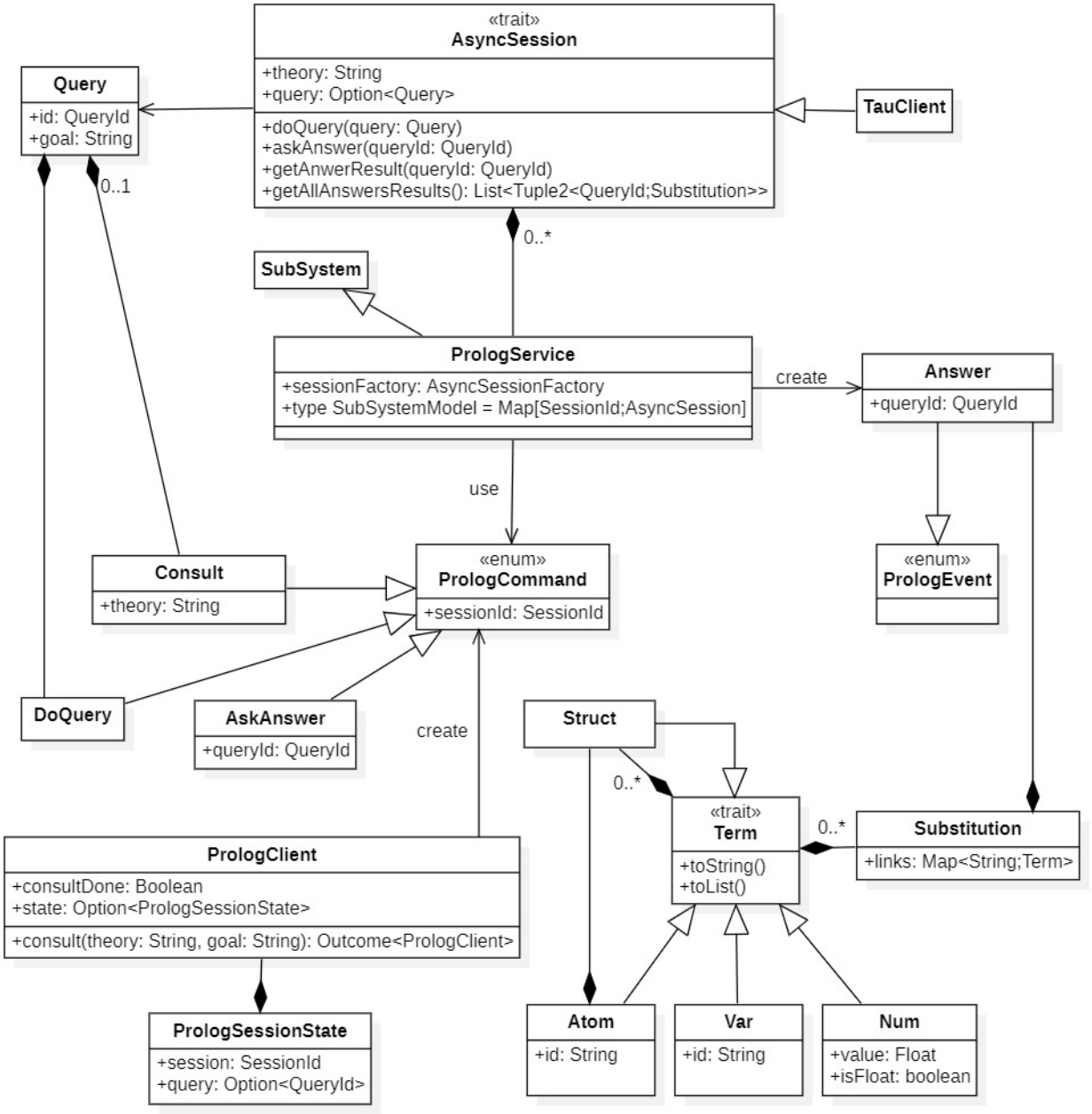

Figura 14: *PrologService, Gerarchia Term e PrologClient*

4.8 Organizzazione del codice

Il codice è stato organizzato in packages. Questi seguono la suddivisione in scene dell'applicativo. Al loro interno, si può ritrovare una suddivisione in Model, View, ViewModel e Componenti. Il package Core invece, contiene le parti basilari a cui il resto del sistema fa riferimento.

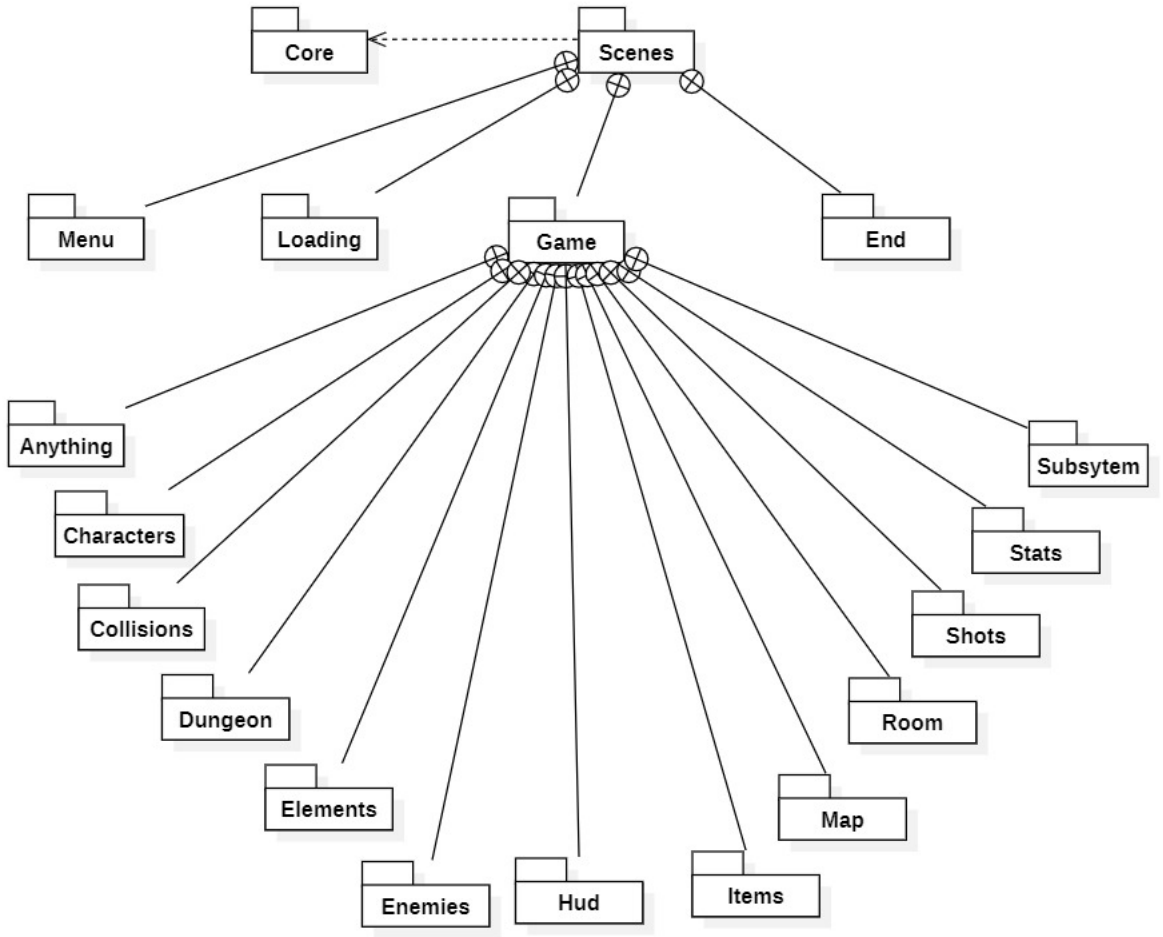

Figura 15: *Package Diagram del sistema*

5 Implementazione

Di seguito vengono analizzati per ogni membro del team gli aspetti implementativi più significativi del sistema.

5.1 Alan Mancini

Durante il corso del progetto mi sono occupato di studiare ed implementare:

- AnythingModel con alcuni suoi sottotipi come DynamicModel e AliveModel e sperimentazione del workflow per consentire mixin/estensione e l'update del Model immutabile
- Macro per ridurre il boilerplate code quando si estende un AnythingModel
- Adapter per semplificare l'update di un insieme di Anything
- AnythingView e AnythingViewModel di base e sperimentazione del workflow derivante
- Nemici per quanto concerne Model ed i loro comportamenti riutilizzabili
- PrologService come sottosistema di Indigo, integrazione di TauProlog e intefacciamento grazie a Scala.js, ed infine client per consultazioni da parte del gioco
- Generazione del dungeon randomica con Prolog
- Visualizzazione mini-mappa del dungeon

5.1.1 AnythingModel e immutabilità

Per quanto riguarda l'applicazione del pattern **F-Bounded Polymorphism** la soluzione che ho adottato con type-member anzichè l'utilizzo di argomenti generici garantisce un buon livello di type-safety: sembra difficile se non impossibile rompere i vincoli di tipo imposti. Impongo che il tipo di *this* sia sottotipo di Model e quest'ultimo sottotipo del tratto in cui è definito: tutto questo permette anche di implementare un metodo update di base che ritorni proprio *this* di tipo Model.

```
trait AnythingModel {
    type Model >: this.type <: AnythingModel

    val id: AnythingId
    val view: () => AnythingView[Model, _]

    ...
```

```
    def update(context: FrameContext[StartupData])(gameContext:
        GameContext): Outcome[Model] =
         Outcome(this)
    }
}

trait DynamicModel extends AnythingModel with StatsModel {
  type Model >: this.type <: DynamicModel

  val speed: Vector2
  val collisionDetected: Boolean

  def withDynamic(boundingBox: BoundingBox, speed: Vector2,
      collisionDetected: Boolean): Model

  ...

  override def update(context: FrameContext[StartupData])(gameContext:
      GameContext): Outcome[Model] =
    for {
      superObj <- super.update(context)(gameContext)
      (newSpeed, newPosition) = computeMove(context)(gameContext)
      boundLocation           = gameContext.room.boundPosition(
          newPosition)
      newObj = superObj
        .withDynamic(boundLocation, newSpeed, boundLocation.position !=
            newPosition.position)
        .asInstanceOf[Model]
    } yield newObj

}
```

Da notare la necessità di un cast a Model in quanto il compilatore Scala non riconosce che il type member di ritorno è lo stesso definito poco sopra: questo cast comunque è legittimo e sicuro.

Per concludere AnythingModel: la factory per la view è richiesta come funzione in linea con il **pattern strategy** applicato in modo funzionale con Scala.

Cambiando argomento, ho applicato il pattern **Pimp My Library** al fine di estendere le funzionalità di Double in modo da ottenere un Timer, utilizzato poi in diversi Model

```
type Timer = Double
extension (timer: Timer)
    def elapsed(time: Double): Timer = timer match {
        case 0                 => 0
        case x if x - time > 0 => x - time
```

```
        case _                  => 0
    }
```

Infine ho utilizzato il **pattern Adapter** per trasformare implicitamente Vector2 in Vertex, due modi di Indigo per gestire i vettori secondo me ridondanti e per questo abbiamo deciso di utilizzarne uno solo nei nostri Model

```
given Conversion[Vector2, Vertex] with
    def apply(v: Vector2): Vertex = Vertex(v.x, v.y)
```

5.1.2 Macro per ridurre boilerplate code

Quando si definisce il Model di un oggetto di gioco che mixa diversi comportamenti è necessario implementare altrettanti template method del tipo with*Comportamento*(...) in modo da permettere la sua copia aggiornata.

```
def withDynamic(x,y,z) = copy(x=x,y=y,z=z);
```

Per questo ho cercato un modo per generare in automatico questi metodi per una case class qualsiasi e l'unico sistema era scrivere una macro attivata da una annotation, ma Scala 3 al momento non supporta questa soluzione possibile su Scala 2.

Quindi ho implementato la **copyMacro** che consente di eseguire il metodo copy con come argomenti quelli dello scope dove viene attivata la macro, sfruttando le potenzialita della ancora non documentata Scala Reflection API.

```
def withDynamic(x,y,z) = copyMacro
```

5.1.3 Adapter per update di collection di Anything

Quando si esegue l'update di una collezione di oggetti AnythingModel o AnythingViewModel ad esempio Map[AnythingId, AnythingModel] si ottiene una Map[AnythingId, Outcome[AnythingModel]] ma quello che occorre è ottenere la Map originale unendo le Outcome e gli eventi contenuti da ciascuna.

```
anythings.map((id, any) => id -> any.update(context)(GameContext(this,
    character)))
```

Per comodità ho applicato il **pattern Adapter** fornendo un apposito convertitore implicito, di seguito quello per AnythingViewModel.

```
given Conversion[Map[AnythingId, Outcome[AnythingViewModel[_]]],
    Outcome[Map[AnythingId, AnythingViewModel[_]]]] with
```

```
  def apply(set: Map[AnythingId, Outcome[AnythingViewModel[_]]]):
     Outcome[Map[AnythingId, AnythingViewModel[_]]] =
    set.foldLeft(Outcome(Map[AnythingId, AnythingViewModel[_]]().empty)
       )((acc, el) =>
      acc.merge[AnythingViewModel[_], Map[AnythingId, AnythingViewModel
         [_]]](el._2)((set, el2) => set + (el._1 -> el2))
    )
```

5.1.4 AnythingView e AnythingViewModel

Nel codice che segue da notare è il context bound **Typeable** (alias di TypeTest[Any,T]) che importa ed abilita gli impliciti TypeTest[Any,M] e TypeTest[Any,VM] i quali ci permettono in modo molto veloce, pulito e sicuro di eseguire una draw dato un Model o ViewModel non correttamente tipati.

```
trait AnythingView[M <: AnythingModel: Typeable, VM <:
   AnythingViewModel[M] | Unit: Typeable] {
  type Model     = M
  type ViewModel = VM
  type View <: Group

  def viewModel: (id: AnythingId) => ViewModel

  protected def view(contex: FrameContext[StartupData], model: Model,
     viewModel: ViewModel): View

  ...

  def draw(contex: FrameContext[StartupData], model: Model, viewModel:
     ViewModel): Group =
    view(contex, model, viewModel)
        .moveTo(model.getPosition())
        .moveBy(Assets.Rooms.wallSize, Assets.Rooms.wallSize)
        .withDepth(depth(model))

  @targetName("anyDraw")
  def draw(contex: FrameContext[StartupData], model: AnythingModel,
     viewModel: AnythingViewModel[_] | Unit): Group =
    (model, viewModel) match {
      case (m: Model, vm: ViewModel) => draw(contex, m, vm)
      case _                         => Group()
    }
}
```

Il problema che avevamo nella visualizzazione degli Anything era che, data una loro collezione, non è possibile ottenere un suo singolo elemento tipato correttamente in modo

automatico, quindi l'unica soluzione trovata prevedeva di fare pericolosi type cast. Nel codice sotto ad esempio il Model genera la View sulla quale viene eseguita una draw: il Model passato alla draw non veniva riconosciuto dal compilatore del tipo richiesto e soprattutto il ViewModel cercato nell'altra collezione viene ovviamente tipato in modo generico. Ho quindi pensato di invertire il problema permettendo al metodo draw della View di accettare un qualsiasi Model o ViewModel in modo sicuro eseguendo un **type check del tipo a runtime** verificando che questo corrisponda ai type member specificati. Il TypeTest interviene implicitamente durante il match con i type Member della View.

```
def anythingView(context: FrameContext[StartupData], model: RoomModel,
    viewModel: RoomViewModel): Group =
     model.anythings.foldLeft(Group())((s1, s2) =>
       s1.addChild(
         s2._2
           .view()
           .draw(
             context,
             s2._2,
             viewModel.anythings
               .get(s2._2.id)
               .getOrElse[AnythingViewModel[_] | Unit](())
           )
       )
     )
```

L'utilizzo di **Typeable** è presente anche in **AnythingViewModel** dove il suo metodo update richiede in input un oggetto di tipo Model generalmente ottenuto da una collezione mista.

5.1.5 Nemici e comportamenti come mixin puri

Per quanto riguarda il mini framework per lo sviluppo di nemici ho applicato il pattern **Pimp My Library** al fine di creare un **mini DSL** per generare la coda di stati a partire da due di essi

```
extension (s1: EnemyStatus) def :+(s2: EnemyStatus): Queue[EnemyStatus]
    = Queue(s1, s2)
```

Circa i comportomenti dei nemici, mostro qui un esempio dell'impiego dei **Self-types** per evitare di estendere da EnemyModel

```
trait Follower { this: EnemyModel with DynamicModel =>
  def computeSpeed(context: FrameContext[StartupData])(gameContext:
      GameContext): Vector2 =
    status.head match {
```

```
      case (EnemyState.Attacking, _) =>
        (gameContext.character.getPosition() - getPosition()).normalise
            * MaxSpeed @@ stats
      case _ => Vector2.zero
    }
}
```

5.1.6 PrologService

Riguardo l'implementazione dei Term prolog ho proceduto secondo la modalita **Mixed OOP/FP** separando la definizione dei dati dal comportamento. Si è poi reso utile l'impiego del pattern **Adapter** per convertire implicitamente all'occorrenza i termini Tau in nostri Term.

```
given Conversion[TauTerm, Term] with
  def apply(t: TauTerm): Term = t.args.length match {
    case 0 => Atom(t.id)
    case _ =>
      Struct(
        Atom(t.id),
        t.args
          .map[Term](arg =>
            arg match {
              case a: TauNum  => a
              case a: TauVar  => a
              case a: TauTerm => a
            }
          )
          .toList: _*
      )
  }

given Conversion[TauSubstitution, Substitution] with
  def apply(t: TauSubstitution): Substitution = Substitution(
    t.links.foldLeft(HashMap[String, Term]())((hmap, kv) => hmap + (kv.
       _1 -> kv._2))
  )
```

5.1.7 Generazione dungeon con Prolog

L'algoritmo Prolog genera il dungeon partendo dalla stanza iniziale posizionata in 0,0 all'interno di una griglia virtualmente infinita. Le successive stanze vengono posizionate selezionando randomicamente una cella libera adiacente a quelle già aggiunte. Per fare questo viene mantenuta una lista di posizioni libere, questa inoltre viene troncata sempre per mantenere una lunghezza di 6 posti in modo che l'algoritmo proceda ad aggiungere

stanze con buona probabilità partendo dall'ultima stanza generando così una struttura più "tortuosa" come da requisito. La stanza del boss, aggiunta per ultima, viene posizionata in modo da essere adiacente ad una sola stanza.

5.2 Federico Mazzini

Durante il corso del progetto mi sono occupato di differenti parti all'interno di esso, tra cui, in ordine:

- Scena di loading e caricamento degli asset
- Il Dungeon, dal modello alla sua integrazione con le altre componenti (ad eccezione dell'algoritmo Prolog per la generazione e all' integrazione con il modello Scala)
- Le stanze di gioco, la loro tipologia, i diversi comportamenti e la gestione degli Anything interni
- Le porte e il passaggio da una stanza all'altra
- Definizione e visualizzazione di elementi bloccanti
- Disposizione, mediante Prolog, di nemici ed elementi bloccanti all'interno delle stanze Arena
- Sistema delle collisioni e di Bounding degli elementi interni alla stanza
- Scena finale e relativa verifica di avvenuta vittoria/sconfitta

5.2.1 Dungeon

Per quanto riguarda il Dungeon, ho cercato di fornire un'implementazione basilare del modello e delle sue funzioni principali. In particolare ho immaginato la mappa come una griglia, composta da una collezione di elementi in posizioni [X,Y]. Questi elementi sarebbero poi diventate le stanze all'interno del Dungeon effettivo, ma per la generazione, è stato utile ridurli a semplici "tipi", da definire successivamente, perciò ho messo in pratica il pattern **Family Polimorphism**.

```
type Position = (Int, Int)
trait Grid {
    type Room
    val content: Map[Position, Room]
    val initialRoom: Position
  }
```

Il type Room è stato ridefinito in due case class, la prima è stata utilizzata durante la generazione del dungeon, prima di generare gli elementi interni alla stanza. In questo caso Room è stato associato ad un semplice enum descrivente il tipo di room.

La seconda volta invece, è stato definito per il vero Dungeon, già generato e con elementi all'interno, in cui il type Room è stato effettivamente associato al vero e completo modello della stanza.

5.2.2 Room

Come già descritto, le stanze sono di diverso tipo, Empty, Arena, Item e Boss. Per modellare questi concetti ho definito un trait base il quale contiene la maggior parte dei comportamenti ma lascia il resto alle classi specializzate. Una room qualsiasi è caratterizzata da un insieme di porte, una collezione di Anything e un area di gioco in cui gli elementi si muovono. Avendo lavorato con strutture dati immutabili, ho cercato di fornire più metodi possibili per la modifica di una Room, intesa come creazione di un nuovo modello con la modifica specificata. Per evitare ripetizioni di codice, la funzione generale sfrutta le **Higher Order Function** e mette in campo il pattern **strategy**.

```
    def updateAnythings
    (updateFunc: Map[AnythingId, AnythingModel] => Outcome[Map[
       AnythingId, AnythingModel]])
    : Outcome[RoomModel] =
    for (updatedAnythings <- updateFunc(anythings))
      yield this match {
        case room: EmptyRoom =>
          room.copy(anythings = updatedAnythings)
        case room: ItemRoom =>
          room.copy(anythings = updatedAnythings)
        case room: ArenaRoom =>
          room.copy(anythings = updatedAnythings)
        case room: BossRoom =>
          room.copy(anythings = updatedAnythings)
        case _ => this
      }
```

Ho sfruttato questa funzione in molte altre adibite alla modifica degli Anything interni di una room, come ad esempio

```
  def addShot(shot: ShotModel): Outcome[RoomModel] =
    updateAnythings(anythings =>
        Outcome(anythings + (shot.id -> shot)))
```

5.2.3 Door

Per quanto riguarda le porte, ho cercato di rimanere il più fedele possibile allo stile funzionale. Ho perciò definito un tipo Location e un tipo State, i quali insieme vanno a comporre una porta. Ho creato un object per permettermi di definire e modificare una Door, ma ciò che nella pratica ho utilizzato sono le **Extension** e le **Conversion** di scala 3 (impliciti precedentemente). In questo modo, è possibile definire una collezione di porte per una stanza nel seguente modo:

```
    (Left -> Open) :+ (Right -> Open) :+ (Above -> Lock)
```

5.2.4 Prolog

Ho utilizzato il prolog per la definizione di un "area di gioco" e un "area elementi" all'interno delle stanze. Ho immaginato come il pavimento di una stanza fosse una griglia. Ho generato, aiutandomi con una findall, la griglia e, in base alle porte della stanza, un'area di gioco di dimensione variabile che comprendesse le porte. Ciò che non rientrava nell'area di gioco è stato classificato come area elementi. Utilizzare il Prolog in questa situazione è stato molto utile per la natura "esplorativa" di esso, la quale mi ha permesso di calcolare facilmente le aree. Di contro, l'algoritmo da me sviluppato è relativamente lento considerando il resto del sistema.

5.2.5 Monadi

All'interno del progetto è stata utilizzata la struttura monadica Outcome propria di Indigo. Attraverso essa è stato possibile wrappare l'intero modello o sottoparti di esso, concatenando diverse modifiche al modello senza avere side effect. Per lavorare con questa monade, ho utilizzato spesso la **for comprehension** di Scala. Di seguito un esempio, riguardante l'intera catena di update del modello di gioco.

```
for {
      updatedCharacter <- model.updateCharacter(c => character.update(
         gameContext))
      updatedRoom      <- updatedCharacter.updateCurrentRoom(r => model
         .currentRoom.update(character))
      withPassage      <- updatedRoom.updateWithPassage
      withStats        <- withPassage.updateStatsAfterCollision(context
         )
      withMovements    <- withStats.updateMovementAfterCollision
    } yield withMovements
```

5.2.6 Collisioni

Il sistema di controllo delle collisioni deve gestire un gran numero di combinazioni possibili. Ad esempio, un oggetto in movimento che collide con una roccia verrà spostato il tanto che basta per non intersecarsi, ma non ha ripercussioni dopo il contatto. Se invece un oggetto in movimento collide con uno shot, questo oggetto perderà punti vita, a seconda del tipo di shot.

Per via del grande numero di combinazioni, durante lo sviluppo del sistema si è fatto largo uso di **strategy** per evitare la duplicazione del codice.

5.2.7 Verifica vittoria/sconfitta

- Un giocatore vince se sconfigge il boss

- Un giocatore perde quando il personaggio controllato finisce i punti vita

Il controllo viene effettuato tramite due eventi *Win* e *GameOver*. Uno dei due eventi viene lanciato quando, durante l'update del Character o del Boss, la life raggiunge lo zero. Gli eventi vengono intercettati dalla Scena di gioco la quale mostra un messaggio di vittoria/sconfitta e rimanda alla scena finale, dove è possibile eventualmente iniziare una nuova partita.

5.3 Matteo Brocca

Durante il corso del progetto mi sono occupato di studiare ed implementare:

- Rappresentazione grafica degli Anythings tramite Asset ed animazioni
- Definizione del FireModel e ShotModel per la gestione dei proiettili
- Gestione delle statistiche degli Anythings
- Character controllato dal giocatore
- Boss, PrologEnemyModel e definizione del suo comportamento tramite Prolog
- Item che il Character può raccogliere per modificare le proprie statistiche
- HUD per la visualizzazione delle statistiche a schermo

5.3.1 Rappresentazione grafica degli Anythings tramite Asset ed animazioni

È stata creata una libreria per gli **AnythingAsset**, la quale codifica le informazioni di base necessare a disegnare a schermo un certo Asset ed implementa metodi condivisi.

Questa organizzazione ci ha permesso di generare velocemente le specifiche View per Character, Enemies, Bosses, Items and Elements.

Facciamo un esempio: Il CharacterModel, viene rappresentato a schermo dalla sua relativa CharacterView e CharacterViewModel che estendono relativamente da AnythingView ed AnythingViewModel

La CharacterView definisce solo la funzione di "disegno" che sfrutta il metodo drawComponents di AnythingAsset e viene mixata con il tipo di carattere che si vuole disegnare a schermo, in questo caso Isaac. Il trait Isaac estende IsaacAsset per definire come disegnare ogni singolo componente a schermo. A sua volta il trait IsaacAsset è quello che estende il trait base AnythingAsset e definisce le dimensioni e nome della Sprite che viene caricata durante l'avvio del gioco.

Nel momento in cui si decida di modificare l'estetica del Character (il *"cosa"*), da "Isaac" a "Mario", senza dover modificare il *"come"* questo viene rappresentato (da una testa, un corpo, un ombra, ...), basterà create un nuovo trait Mario e relativo MarioAsset.

```
trait AnythingAsset {
  ...
  protected def drawComponents(components: List[SceneNode]): Group =
     ...
}

trait IsaacAsset extends AnythingAsset {
    ...
```

```
}

trait Isaac extends IsaacAsset{
  ...
  def headView(model: CharacterModel, viewModel: CharacterViewModel):
     Graphic[Material.Bitmap] = {...}
  def bodyView(model: CharacterModel): Sprite[Material.Bitmap] = =
     {...}
}

trait CharacterView[VM <: AnythingViewModel[CharacterModel] | Unit]
   extends AnythingView[CharacterModel, VM] {}

object CharacterView extends CharacterView[CharacterViewModel] with
   Isaac {
  ...
  def view(contex: FrameContext[StartupData], model: Model, viewModel:
     ViewModel): View =
    ...
    drawComponents(List(shadowView, bodyView(model), headView(model,
       viewModel)))
}
```

5.3.2 Definizione del FireModel e ShotModel per la gestione dei proiettili

Ogni proiettile generato dal FireModel è uno ShotModel.

La factory che ne permette la creazione durante la generazione dell'evento ShotEvent richiede anche quale sia la relativa View per la rappresentazione grafica del proiettile. Si può infatti notare che il FireModel richiede la definizione della funzione di creazione di questa View di tipo ShotView (pattern strategy: factory as a function), un trait che sfrutta il **Self-Type** per richiedere espressamente di essere mixato con il trait **ShotAsset** che definisce le informazioni di base che ogni proiettile deve avere ed implementa la funzione **drawShot**

```
trait FireModel extends AnythingModel with StatsModel {
    type Model >: this.type <: FireModel

  ...

    val shotView: () => ShotView[_]

    ...
}

class SingleShotView extends ShotView[Unit] with SimpleAnythingView {
  this: ShotAsset =>
```

```
  type View = Group

  def view(contex: FrameContext[StartupData], model: Model, viewModel:
      ViewModel): View =
    drawComponents(List(drawShot))

  ...
}
```

Un esempio dell'utilizzo del FireModel si può apprezzare all'interno del CharacterModel dove viene definita la shotView come un SingleShotView con il trait ShotBlue e viene implementato il computeFire con il mapping dei tasti premuti dall'utente.

```
case class CharacterModel(...) extends AliveModel
    with DynamicModel
    with FireModel
    with DamageModel
    with SolidModel {

  val shotView        = () => new SingleShotView() with ShotBlue

  ...
}
```

5.3.3 Gestione delle statistiche degli Anything

Si è realizzato il trait **StatsModel** da mixare con un qualsiasi Anything per gestire l'aggiornamento di queste caratteristiche nell'ambito dell'immutabilità.

Il metodo "sumStat" è l'unico utilizzato realmente all'interno del gioco e permette di aggiornare il valore di una determinata caratteristica sommando il valore proveniente da un Item raccolto verificando in automatico che il valore non diventi negativo.

Gli altri metodi "changeStats" e "changeStat" sono solo stati previsti per eventuali sviluppi futuri.

```
trait StatsModel {
  type Model >: this.type <: StatsModel

  val stats: Stats

  def withStats(stats: Stats): Model

  def changeStats(context: FrameContext[StartupData], newStats: Stats):
      Outcome[Model] = Outcome(withStats(newStats))

  def changeStat(context: FrameContext[StartupData], property:
      StatProperty): Outcome[Model] =
```

```
    Outcome(withStats(stats + property))

  def sumStat(context: FrameContext[StartupData], property:
      StatProperty): Outcome[Model] =
    Outcome(withStats(stats +++ property))
}
```

Questo trait usufruisce della libreria **Stats** la quale è implementata seguendo il pattern **Pimp My Library** per mettere a disposizione i propri metodi all'interno dello specifico dominio applicativo (**DSL**).

La libreria definisce che cosa sono le Stats, una mappa di proprietà da nome a valore. Sono stati implementate delle **Conversion** per utilizzare i valori definiti come Double anche in situazioni dove si accettano Int o String. Inoltre sono state realizzate delle **extension** per definire operatori specifici per manipolarle.

```
package lns.scenes.game.stats

enum PropertyName:
  case  MaxLife, Invincibility, MaxSpeed, Range, KeepAwayMin,
     KeepAwayMax, Damage,
    FireDamage, FireRange, FireRate,FireSpeed

type PropertyValue = Double
type StatProperty  = (PropertyName, PropertyValue)
type Stats         = Map[PropertyName, PropertyValue]

given Conversion[PropertyValue, Int] with
  def apply(v: PropertyValue): Int = v.toInt

given Conversion[PropertyValue, String] with
  def apply(v: PropertyValue): String = v.toString

extension (p: PropertyValue) {
  def |+|(v: PropertyValue): PropertyValue = p match {
  case p if (v + p) < 0 => 0
  case _                => BigDecimal(v + p).setScale(2, BigDecimal.
     RoundingMode.HALF_UP).toDouble
  }
}

extension (p: PropertyName) {
  def @@(s: Stats): PropertyValue = s.getOrElse(p, 0.0)
}

extension (stats: Stats) {
  def +++(p: StatProperty): Stats =
  stats + stats.get(p._1).map(x => p._1 -> (x |+| p._2)).getOrElse(p)
}
```

5.3.4 Boss, PrologEnemyModel e definizione del suo comportamento tramite Prolog

Per la creazione del boss, il cui comportamento è definito tramite linguaggio Prolog, è stato creato il trait **PrologEnemyModel**. Grazie a Scala 3, il trait accetta il paramentro "*name*" che definisce il nome del file prolog da interrogare e che sarà stato opportunamente caricato tra gli asset di gioco.

Se il suo EnemyStatus attuale (tipico degli EnemyModel) è di tipo *Idle* e non è definita nessuna *EnemyAction*, allora viene interrogato il codice Prolog. Il **goal** che viene passato è una stringa prodotta dal metodo del quale si richiede l'override. La risposta del prolog, catturata dal gameLoop, esegue il metodo **behaviour** del chiamante implementato dalla classe che lo estende.

```
trait PrologEnemyModel(name: String) extends EnemyModel {
  type Model >: this.type <: PrologModel

  val prologClient: PrologClient

  def withProlog(prologClient: PrologClient): Model

  protected def goal(context: FrameContext[StartupData])(gameContext:
     GameContext): String

  def behaviour(response: Substitution): Outcome[Model]

  protected def consult(context: FrameContext[StartupData])(gameContext
     : GameContext): Outcome[Model] = ...

  override def update(context: FrameContext[StartupData])(gameContext:
     GameContext): Outcome[Model] = ...
}

case class BossModel(...) )
  extends PrologEnemyModel("loki")
  with FireModel
  with Traveller {

  ...

  protected def goal(context: FrameContext[StartupData])(gameContext:
     GameContext): String = ...

  def behaviour(response: Substitution): Outcome[Model] = ...
}
```

Il boss di nome "Loki" che è stato implementato può eseguire 5 diverse azioni, 3 di attacco, 1 di movimento ed 1 di difesa, che hanno una probabilità in base allo stato di

vita del Boss e del Character. Viene definito *low* il livello di vita se questo è inferiore al 50% della vita massima, altrimenti *high*

```
% estratto del file prolog "assets/prolog/loki.pl"
...
% List of probability value for actions based on Boss and Character
    life level
% @high|low (Boss)
% @high|low (Character)
% -Probability List[Attack, Defense, Move]
getActionsProbability(high, high, [0.7, 0.0, 1.0]).
getActionsProbability(high, low,  [0.5, 0.0, 1.0]).
getActionsProbability(low,  high, [0.7, 1.0, 0.0]).
getActionsProbability(low,  low,  [0.5, 0.7, 1.0]).
...
```

Di seguito vengono descritte nel dettaglio le singole azioni:

- attack1(direction): viene generato un proiettile nella direzione del Character se questo si trova sullo stesso asse x o y.
- attack2: vengono generati 4 proiettili in contemporanea lungo i 2 assi x, y
- attack3: vengono generati 4 proiettili in contemporanea in diagonale lungo i 2 assi x, y
- move(x,y): il boss sposta rapidamente verso il punto x, y occupato dal Character
- defence(x,y): il boss si teletrasporta nel punto x,y se questo non è occupato da una roccia, altrimenti viene cercato il primo posto disponibile nel suo intorno

6 Retrospettiva

6.1 Sprint 1 27/09 - 03/10

Epic Visualizzare il personaggio a schermo

Il primo sprint è stato prevalentemente organizzativo ed è stato focalizzato sulla scelta e il setup degli strumenti, la definizione del processo di sviluppo e il design architetturale. In particolare si è scelto di lavorare con il framework Indigo e si è studiata la sua documentazione, al fine di comprendere come organizzare il codice e poter definire l'architettura generale del gioco. Era previsto anche il setup della CI/CD, abbiamo deciso però di spostarla nella seconda sprint, la quale sarà orientata a implementare la struttura base e produrre la prima schermata di gioco.

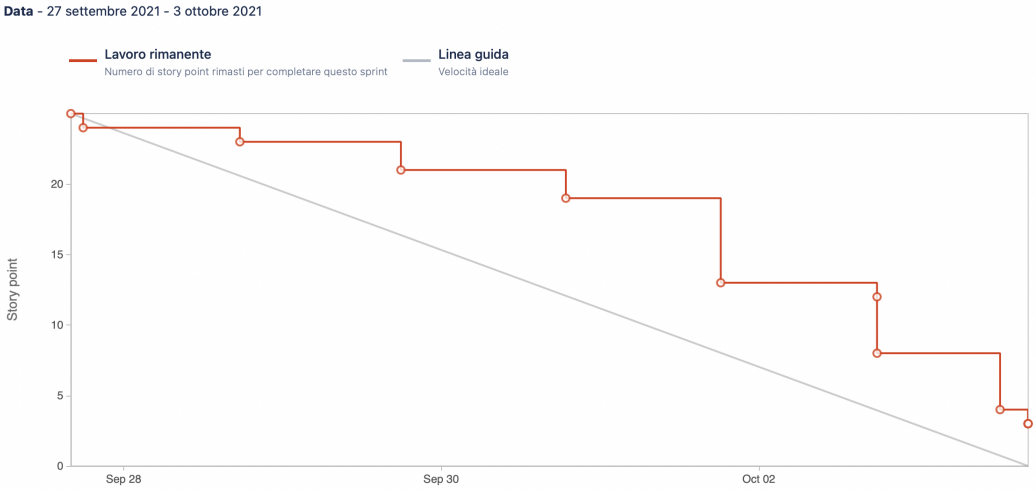

Figura 16: *Grafico Burn-down primo sprint*

6.2 Sprint 2 04/10 - 10/10

Epic Visualizzare il personaggio a schermo

Nel secondo sprint abbiamo effettuato il setup dell'architettura includendo Indigo, al fine di visualizzare il menu di avvio, il loading e permettere all'utente di visualizzare il proprio personaggio all'interno di una stanza vuota. Abbiamo inoltre abilitato la continuous integration. Siamo così giunti all'obbiettivo dell'epic. In questa sprint, abbiamo fatto i primi passi con l'engine Indigo, confrontandoci costantemente in modo da sperimentare insieme il suo utilizzo. Questo ci ha permesso di allineare la nostra conoscenza a riguardo e definire una modalità operativa per le future sprint. Non siamo riusciti a completare la visualizzazione del personaggio in quanto ci siamo resi conto di dover impostare meglio il modello generale al fine di realizzare una struttura a componenti che funzioni con Indigo.

Figura 17: *Grafico Burn-down secondo sprint*

6.3 Sprint 3 11/10 - 17/10

Il terzo sprint è stato dedicato al rifinimento del personaggio aggiungendo animazioni e vincolandolo all'interno dei confini della stanza. Questo ha richiesto di approfondire lo studio di Indigo riguardo le animazione e rifattorizzare la stanza, le sue proprietà e la sua logica. Oltre a questo è stata rivista la gerarchia di classi che supportano il personaggio e che sarà alla base di tutti gli elementi presenti dentro alla stanza (nemici, oggetti, elementi bloccanti). Infine, con lo studio di un workflow di test per Indigo, ci siamo predisposti per uno sviluppo TDD che avverrà dalle prossime sprint.

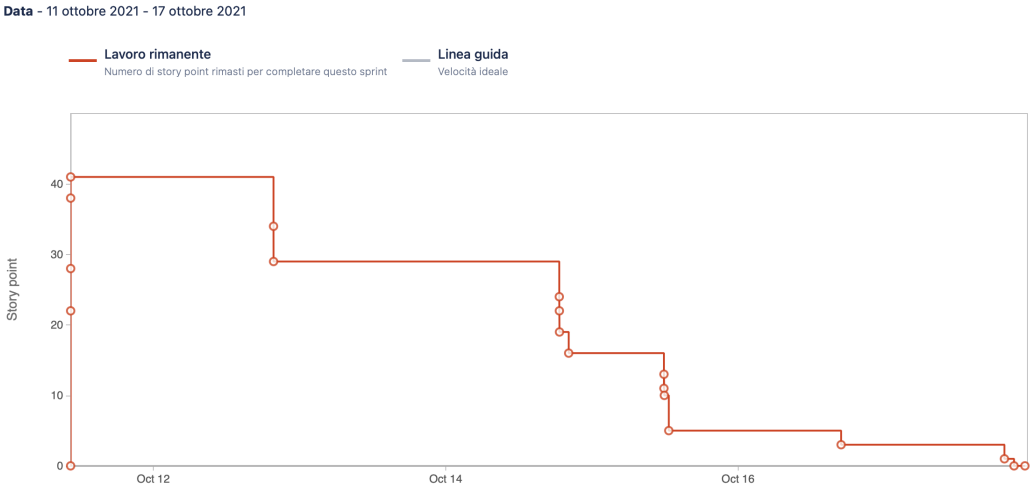

Figura 18: *Grafico Burn-down terzo sprint*

6.4 Sprint 4 18/10 - 24/10

Il quarto sprint è stato incentrato principalmente sul dungeon e sulla feature dello sparo per il personaggio controllato dall'utente. In particolare, per quanto riguarda il dungeon, è stata implementata la generazione tramite prolog e, inoltre, la possibilità di navigare tra le stanze attraverso l'uso di porte. Essendo Indigo basato su Scala.js abbiamo integrato un engine Prolog scritto in javascript e creato un'interfaccia in Scala per interagirvi.

Nella prossima sprint, sarà necessario un refactor e una forte integrazione tra le parti sviluppate individualmente dai diversi componenti del team.

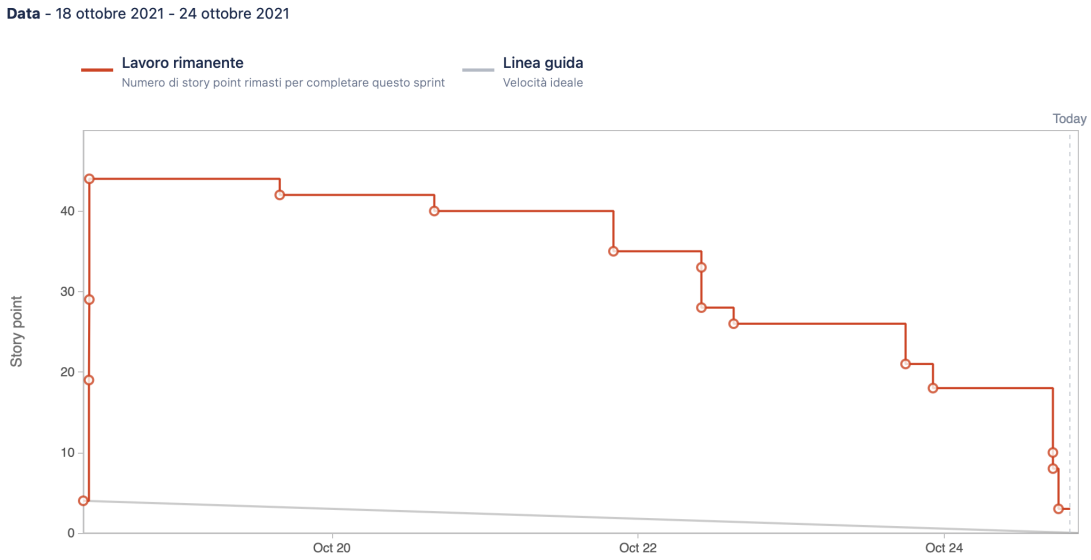

Figura 19: *Grafico Burn-down quarto sprint*

6.5 Sprint 5 25/10 - 31/10

Il quinto sprint è stato dedicato al refactor di diversi sezioni dell'applicativo, per permettere, nelle prossime sprint, di implementare in maniera più semplice le feature rimanenti. In particolare, Alan si è occupato dell'interazione Scala Prolog, in modo tale da poter utilizzare il Prolog anche per la generazione degli elementi delle stanze. Federico invece si è occupato delle Room, in modo da poter implementare le collisioni tra oggetti in maniera corretta nella prossima sprint. A livello di feature, all'interno di questa sprint si è dotato l'applicativo di una vera generazione casuale del dungeon e Matteo si è dedicato a dotare il personaggio di caratteristiche migliorabili. In questa sprint volevamo offrire più valore al committente, ma abbiamo riscontrato problemi con i test in quanto lavorando con Scala3, ScalaJS e Indigo siamo stati vincolati nella scelta dei framework di test da utilizzare. In particolare al momento non è possibile abilitare il Coverage.

Figura 20: *Grafico Burn-down quinto sprint*

6.6 Sprint 6 01/11 - 07/11

Il sesto sprint è stato incentrato nel creare le dinamiche di gioco, tra cui la rilevazione delle collisioni tra gli elementi, la creazione di diversi nemici con diversi comportamenti e statiche, la visualizzazione dello stato del personaggio. Durante lo sprint planning abbiamo sottostimato l'effort richiesto per l'inserimento degli elementi bloccanti, anche considerando l'utilizzo del Prolog per la loro disposizione, questo richiede una nuova sprint per essere portato a termine. Un altro imprevisto è stato il dover gestire il view model per i nemici, il tutto dovuto anche a dei limiti dei Indigo, il quale non è ancora maturo a livello di engine. Anche quest'ultima parte va rifattorizzata nel prossimo sprint.

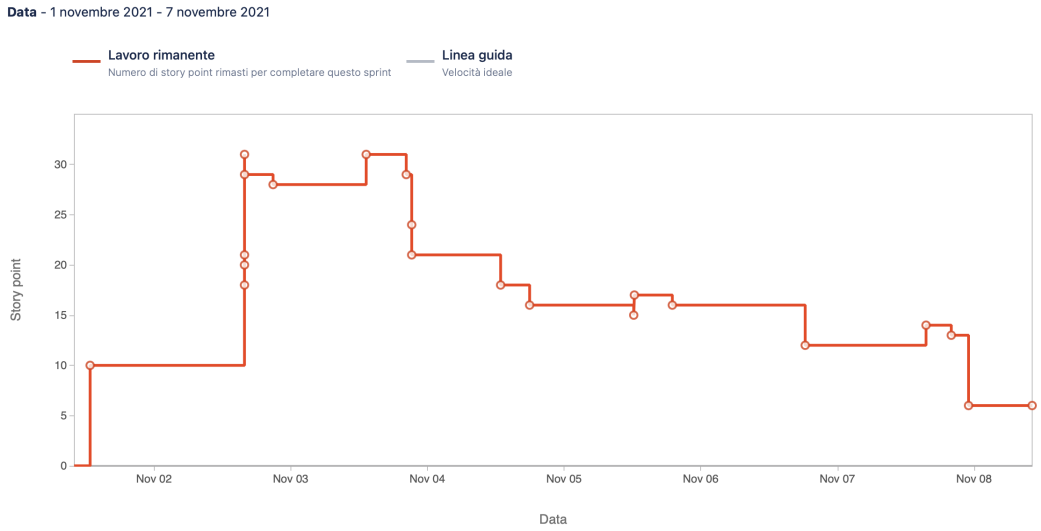

Figura 21: *Grafico Burn-down sesto sprint*

6.7 Sprint 7 08/11 - 14/11

Il settimo sprint è stato dedicato al refinement di alcuni punti degli sprint precedenti e all'introduzione di nuove feature, tra cui lo sviluppo ulteriore delle dinamiche di gioco e d'interazione tra il personaggio e i nemici. Si è quindi sviluppato un modello per le statistiche di qualsiasi elemento "Alive" e un modello di individuazione e gestione delle collisioni. Per quanto riguarda i refinement invece, si è ripensato all'interazione tra Model, View e ViewModel, rifattorizzando l'architettura. Durante questa sprint non si è ancora riusciti a concludere il posizionamento degli elementi bloccanti in Prolog per via dello sviluppo di altre feature più prioritarie.

Figura 22: *Grafico Burn-down settimo sprint*

6.8 Sprint 8 15/11 - 21/11

A livello di feature, sono stati aggiunti alcuni elementi di valore per il gioco. In particolare, si è dotata la schermata di gioco di una minimappa, in modo tale che l'utente possa visualizzare a video la sua posizione nel dungeon al momento. Si è sviluppato un algoritmo Prolog per generare un area di gioco e un area di elementi, in modo tale da poter disporre all'interno della stanza nemici ed elementi bloccanti in maniera consona con quanto riportato nei requisiti. Si è poi incrementato il modello delle stats in maniera tale da poter essere modificate in base ad avvenimenti accaduti durante il gioco. La disposizione degli elementi bloccanti necessiterà di sviluppo anche durante la prossima sprint, in particolare per quanto riguarda l'integrazione tra Prolog e Scala.

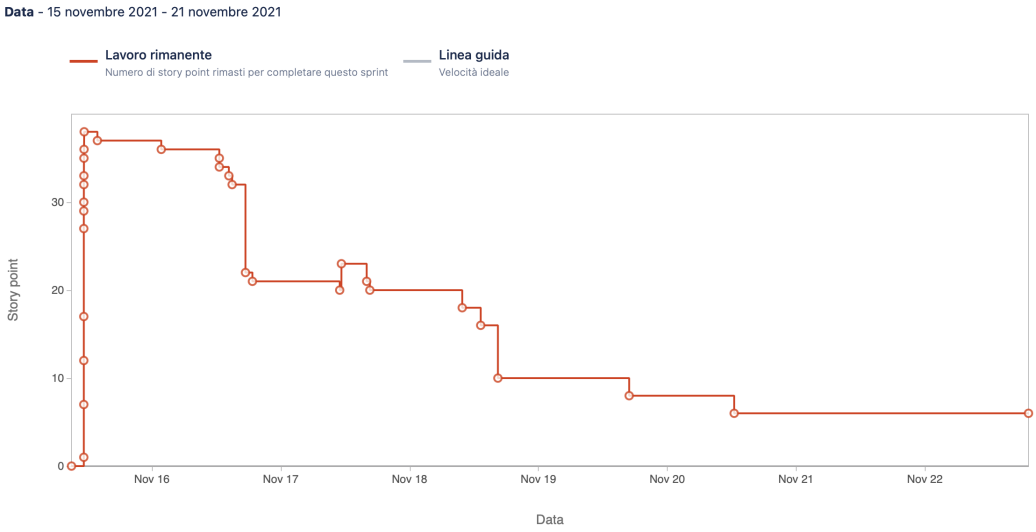

Figura 23: *Grafico Burn-down ottavo sprint*

6.9 Sprint 9 22/11 - 28/11

Il nono e ultimo sprint è stato dedicato alla chiusura di alcuni ticket ancora aperti e al refinement di diversi componenti. Si è conclusa la creazione casuale degli elementi bloccanti mediante Prolog, la gestione del fine partita, creando gli scenari di vittoria, sconfitta e rivincita.

L'elemento di maggior valore aggiunto in questa sprint è il boss finale. Questo è composto da logica Scala e logica Prolog ed è stato interamente sviluppato da Matteo.

Figura 24: *Grafico Burn-down nono sprint*

6.10 Considerazioni finali

Il progetto è stato realizzato seguendo la metodologia Scrum e in generale si è cercato il pià possibile di perseguire l'agilità. Questo ha permesso di concentrarci, sprint dopo sprint, alla creazione di valore per il committente e procedere iterativamente nello sviluppo del gioco.

Una nota sui test, ove possibile abbiamo cercato di applicare la metodologia TDD, sebbene sia stato complesso almeno inizialmente. Purtroppo segnaliamo che per una combinazioni di fattori (ScalaJS, Indigo e...?) non abbiamo potuto effettuare test coverage.

Complessivamente ci riteniamo soddisfatti di come è stato condotto il progetto, non sono stati riscontrati particolari attriti tra i membri del gruppo e la divisione del lavoro è stata equa e bilanciata. Inoltre tutti i requisiti posti inizialmente sono stati rispettati. La crescita dei membri del team nel corso di questi due mesi è stata esponenziale per quanto riguarda le competenza Scala, Prolog e di gestione del progetto, portandoci ad aggiungere valore al gioco molto più velocemente negli ultimi tempi rispetto ai primi.