

Ingegneria e Scienze Informatiche LM
Cesena - Paradigmi di Programmazione Software

Progetto Missile Command

Studente: Daniele Di Lillo - Andrea Brighi - Matteo Lazzari

Professore: Mirko Viroli

Data: December 3, 2022

Contents

1	Introduzione	1
2	Processo di sviluppo	2
2.1	Meetings	2
2.1.1	Definizione dei task	2
2.1.2	Sprint planning	3
2.1.3	Sprint review	3
2.1.4	Gestione issues	3
2.1.5	Daily SCRUM	3
2.2	Divisione dei task	4
2.3	Aggregazione dei risultati	5
2.4	Versioni	6
2.5	Tool utilizzati	7
3	Requisiti	8
3.1	Requisiti di business	8
3.2	Requisiti utente	8
3.3	Requisiti funzionali	9
3.4	Requisiti non funzionali	10
3.5	Requisiti implementativi	10
4	Design architetturale	11
4.1	Architettura complessiva	11
4.2	Pattern architetturali utilizzati	11
4.2.1	Event-Drive Architecture	12
5	Design dettagliato	13
5.1	Cake Pattern	13
5.1.1	ModelModule	14
5.1.2	ViewModule	14
5.1.3	ControllerModule	15
5.2	Gestione del Controller	15
5.3	Gestione della View	17
5.3.1	WorldPane	17
5.3.2	GUI	18
5.4	Gestione Model	19
5.4.1	Behavior	20
5.4.2	Collisioni	21
5.4.3	Missili e esplosioni	23
5.4.4	Città e Torrette	25
5.4.5	Ground	27
5.4.6	Spawner	28
5.4.7	Altri tipi di nemici	30
5.5	Pattern utilizzati	30
6	Implementazione	33

6.1	Concetti relativi a programmazione funzionale	33
6.1.1	Trait e Mixins	33
6.1.2	Immutabilità	33
6.1.3	High order function	33
6.1.4	Self type	34
6.1.5	Alias	35
6.1.6	For Comprehension	35
6.1.7	Currying	36
6.1.8	Generici	36
6.1.9	Extensions	37
6.1.10	Pimp my Library	37
6.2	Concetti di programmazione asincrona e reattiva	37
6.2.1	GameLoop	38
6.2.2	View ed Event	38
6.3	Testing	40
6.3.1	Property-based testing	40
6.3.2	Behaviour-Driven Development (BDD)	41
6.3.3	Coverage	41
6.4	Presentazione grafica	43
6.5	Suddivisione del lavoro	43
7	Retrospettiva	50
7.1	Sprint e Backlog	50
7.2	Commenti finali	52
8	Conclusioni	53

1 Introduzione

Il progetto in questione consiste nel realizzare un clone del famoso gioco Missile Commander pubblicato da Atari nel 1980. In particolare, avviato il gioco, l'utente si ritroverà immerso in un mondo virtuale formato da 6 città da difendere e 3 torrette lancia missili da utilizzare per difendere le proprie strutture. Durante la partita, il giocatore verrà attaccato continuamente da missili nemici (anche di tipo diverso), rispettando le meccaniche del gioco originale e rivisitando l'aspetto grafico. Inoltre, durante la partita, la difficoltà aumenterà progressivamente, fornendo al giocatore una sfida sempre più crescente al passare del tempo.

2 Processo di sviluppo

Per il processo di sviluppo, il team di lavoro si è organizzato utilizzando un approccio di tipo SCRUM-inspired per la realizzazione del progetto.

In particolare, SCRUM è un framework di sviluppo software agile, iterativo e incrementale, che consente la gestione dello sviluppo di software.

In particolare, questo processo prevede di:

- nominare uno studente (ad esempio, chi ha l'idea del progetto) che fungerà da committente o esperto del dominio, e che cercherà di garantire l'usabilità e la qualità del risultato;
- designare uno studente (ad esempio, chi pensa di avere doti di coordinamento) che ricoprirà il ruolo di project manager e possa anche prendere decisioni sull'ambito dello sviluppo del progetto;
- utilizzare sprint corti, da 15-20 ore di lavoro, con l'obiettivo di ottenere ad ogni sprint dei risultati "tangibili", ossia con un valore per gli stakeholder;
- fare meeting frequenti ed a inizio/fine sprint realizzare uno sprint backlog per mantenere traccia dell'organizzazione dei lavori.

2.1 Meetings

I meeting sono stati svolti tramite incontri brevi e frequenti (quasi sempre quotidiani), in modo da mantenere aggiornati i diversi membri del gruppo sullo stato di avanzamento del progetto.

Inoltre si discutevano anche di possibili modifiche o si decideva insieme sul come risolvere problemi riscontrati durante lo sviluppo.

I meeting giornalieri permettevano al project manager di verificare se l'idea generale dei task da svolgere fosse compresa.

Per quanto riguarda gli sprint, sono stati svolti nell'arco di una settimana lavorativa e discussi in incontri a inizio e fine periodo (alcuni sprint sono stati più lunghi di altri a causa di festività e/o eventi).

In particolare, durante l'incontro di inizio sprint viene definito il backlog relativo alla settimana, mentre, durante quello di fine sprint, vengono discussi i risultati ottenuti e viene revisionato il lavoro svolto per verificare se le attività sono state svolte nella loro interezza e nel modo concordato.

I diversi meetings sono stati tenuti in presenza, spesso anche mostrando il codice svolto, al fine di ottenere pareri su come migliorare la qualità del codice e le funzionalità.

2.1.1 Definizione dei task

Durante i primi incontri, dopo aver analizzato i requisiti del sistema, si è cercato di estrarre i principali task da svolgere, raggruppandoli per tema. Abbiamo quindi creato una *roadmap* delle funzionalità da svolgere, definendo anche la sequenza in alcuni casi. Questo permette di definire in anticipo i task principali da svolgere e prevedere eventuali problematiche nel design proposto.

2.1.2 Sprint planning

Per *sprint planning* si intende un meeting in cui il team pianifica il lavoro che deve essere svolto e portato a termine durante lo sprint.

Questo viene fatto analizzando il lavoro fatto e, partendo dalla roadmap definita, si assegnano i task che il team dovrà svolgere durante lo sprint.

In alcuni casi, durante lo sprint planning, si aggiungono task non definiti inizialmente, ma ritenuti rilevanti per lo sprint della settimana corrente.

Si definisce quindi l'obiettivo principale dello sprint e lo stato da raggiungere a fine sprint tramite il backlog.

L'assegnazione dei task ad ogni membro del gruppo è stato pensato in modo tale da avere un'equa distribuzione del lavoro, sia per complessità che per tempo stimato di realizzazione.

Per l'assegnazione dei task si è anche considerata la continuità del lavoro già svolto.

2.1.3 Sprint review

Per *sprint review* si intende un meeting che ha luogo alla fine dello sprint, il cui obiettivo è di revisionare e valutare il lavoro svolto dal team di sviluppo.

In particolare al termine dello sprint vengono discussi tutti i problemi che si sono riscontrati durante lo sviluppo per capire quali sono state le parti più problematiche ed eventualmente fornire nuove soluzioni.

Inoltre si analizzano i task effettuati e si verifica che soddisfino l'obiettivo preposto; se si rilevano bug o mancanza di funzionalità questi dovranno essere risolti nello sprint successivo e vengono normalmente segnalati come issue.

Infine si valuta lo stato attuale del progetto grazie all'incremento dello sprint.

2.1.4 Gestione issues

Alla fine di ogni Sprint review, e in seguito ai primi test, si notano problemi e/o comportamenti inattesi del software.

Questi vengono segnalati e trattati come issues.

Le issues vengono riportate da uno o più componenti del gruppo ed assegnate a chi ha svolto la parte soggetta al problema o, più in generale, a tutti.

Le issues riscontrate rientrano nei task settimanali da svolgere per il componente del gruppo in questione, oppure suddivise se di tutti.

2.1.5 Daily SCRUM

Per *daily SCRUM* si intende il meeting giornaliero in cui il team si riunisce e ogni membro mette al corrente i collaboratori del proprio operato.

Il meeting è breve (circa 15-20 minuti) e durante questo tipo di incontro ogni membro del gruppo espone i seguenti punti:

- il lavoro svolto nella giornata passata;
- le eventuali problematiche riscontrate durante lo sviluppo, chiedendo, eventualmente, consigli agli altri membri del team su come poterle risolvere;

- la pianificazione dei successivi compiti che si intende svolgere, a fronte di quanto emerso nei precedenti punti.

Va specificato che questi meetings non sono stati svolti necessariamente tutti in presenza, ma anche via chat.

Inoltre, queste occasioni di discussione, in quanto giornaliere e di breve durata, non sono state documentate.

2.2 Divisione dei task

All'avvio di ciascuno sprint settimanale, mediante il backlog, sono stati assegnati a ciascun membro del team una serie di tasks da svolgere.

In particolare, ogni componente del team ha contribuito al progetto realizzando i tasks descritti di seguito.

Brighi Andrea

Macro obiettivo	Task
Elementi 2D base	Punti 2D
	Vettori 2D
Gestione interazioni	Hitbox
	Collisioni
	Calcolo punteggio
	Mondo di gioco
Controller	Generatore di eventi tempo
	Game loop

Di Lillo Daniele

Macro obiettivo	Task
Missili	Missili normali
	Missili zig zag
	Esplosioni
Spawner	Spawner generico
	Spawnable
	Spawner aggregato
GUI	Visualizzazione missili
	Eventi di click
	Interfaccia di gioco
	Interfaccia di fine gioco

Lazzari Matteo

Macro obiettivo	Task
Ground	Città
	Batterie missilistiche
	Ground
Nemici	Aerei
	Satelliti
	Spawner di nemici
GUI	Visualizzazione ground
	Immagini dei componenti di gioco
	Interfaccia di gioco

2.3 Aggregazione dei risultati

Il progetto è stato realizzato attraverso l'utilizzo di un repository GitHub. Partendo dal branch develop, in cui è presente l'attuale stato del progetto in sviluppo, ogni membro del gruppo ha creato un branch per eseguire uno o più task da svolgere, in cui lavorare in modo indipendente, seguendo la metodologia *Git Flow*.

Se necessario più componenti operano sullo stesso branch.

Una volta completato il task e verificato non vi siano problemi, il branch del task viene unito a develop tramite il comando merge (o pull request).

Se necessario, un branch viene unito a develop prima del suo completamento per perme-

ttare agli altri componenti di utilizzare le funzionalità implementate e far continuare lo sviluppo sullo stesso branch.

Quando si ritiene che il lavoro sia giunto al punto e che quindi sia possibile creare una versione dell'applicazione, si esegue un merge tra develop e main e si effettua una release, etichettando la versione e caricando il jar.

2.4 Versioni

Durante lo sviluppo del gioco sono state rilasciate 3 versioni:

Versione 1.0.0

Prima versione del gioco con funzionalità base, che comprendono:

- Presenza di missili nemici e alleati;
- Presenza di città e batterie missilistiche;
- Collisioni.

Presenta i seguenti problemi:

- Bug nell'esecuzione del jar;
- Performance non fluide.

Versione 1.0.1

Versione correttiva della precedente, in particolare risolve:

- Il bug nell'esecuzione del jar;
- Il gioco risulta più fluido;

Versione 1.1.0

Versione che aggiunge nuovi nemici e include punteggio e timer di sopravvivenza. In particolare:

- Aggiunti aerei che lanciano missili;
- Aggiunti satelliti che lanciano missili;
- Aggiunti missili a Zig Zag (direzione variabile);
- Aggiunto timer di sopravvivenza e punteggio;
- Miglioramento della grafica.

2.5 Tool utilizzati

Per la realizzazione del progetto sono stati utilizzati differenti tool a supporto del processo di sviluppo.

Tali strumenti hanno come obiettivo quello di agevolare gli sviluppatori durante tutta la realizzazione del progetto, cercando di automatizzarne i diversi aspetti.

Gli strumenti impiegati per il progetto sono riportati di seguito:

- SBT, per la build automation;
- ScalaTest, per la scrittura ed esecuzione di test automatizzati sul codice di produzione;
- GitHub, come servizio di hosting per il codice sorgente;
- GitHub Actions, per promuovere la *continuous integration*;
- Jira, come tool collaborativo, per la gestione degli sprint, tasks, report e gestione delle issues, nonché per tenere aggiornato il backlog.

3 Requisiti

In questa sezione verranno discussi i requisiti dell'applicazione che verrà realizzata, a partire da quelli di business fino a quelli funzionali e non funzionali.

I requisiti verranno numerati in modo tale che, nel seguito, potranno essere ricondotti alle rispettive sottosezioni.

3.1 Requisiti di business

Il requisito di business previsto dall'applicazione è quello di replicare il funzionamento del gioco arcade Missile Command per computer, ovvero simulare una battaglia tra il giocatore e nemici utilizzando come unico mezzo di danno i missili.

3.2 Requisiti utente

Il committente ha richiesto un sistema che implementi il gioco descritto, secondo l'estratto:

Sono un proprietario di una sala giochi e avrei bisogno di una replica del famosissimo gioco Missile Command eseguibile da un computer, con una grafica che ricordi quella originale e le medesime meccaniche.

Il giocatore prende il controllo di tre torrette antiaeree al fine di difendere sei città da continui attacchi missilistici nemici.

Attraverso l'utilizzo del cursore, il giocatore ha la possibilità di lanciare missili, che partono dalle torrette, verso il punto indicato dal puntatore.

I missili sono lanciati da una delle tre batterie a disposizione del giocatore, in particolare la torretta carica più vicina al punto selezionato.

Il missile così sparato, proseguirà nella sua traiettoria finché non raggiungerà il punto selezionato per poi esplodere.

Esplodendo, il missile crea un'area in cui qualsiasi oggetto nemico vi si inserisca viene distrutto.

Se invece vi entra un missile alleato, quest'ultimo potrà proseguire il suo percorso senza subire alcun danno.

L'esposizione dura una quantità di tempo prestabilito, questo permette di distuggere più nemici con un solo colpo.

Nel gioco ci sono diversi nemici:

- *missili*
- *aerei*
- *satelliti*

I missili nemici, a differenza di quelli dell'utente hanno due modi di muoversi, in linea retta o a zig-zag, quest'ultima è più difficile da intercettare.

Quando un missile raggiunge la base del campo espode ed, eventualmente, distrugge uno degli edifici presenti nel raggio d'azione (torretta e/o città).

Se un missile viene distrutto da un esplosione anche esso espoderà (si noti che se un

missile del giocatore entra nell'area dell'esposizione di un missile nemico espoderà, mentre uno nemico non verrà danneggiato).

Gli aerei e i satelliti sono nemici che hanno la possibilità di lanciare missili a loro volta, al fine di distruggere batterie missilistiche e/o città del giocatore.

La differenza tra satellite e aereo è che l'aereo si muove ed è quindi più difficile da colpire, mentre il satellite rimane stazionario.

Tutti i missili che vengono lanciati impiegano del tempo per raggiungere l'obiettivo, quindi è importante per l'utente anticipare il tiro.

Durante la partita, il giocatore guadagna punti distruggendo i nemici.

Inoltre, la difficoltà aumenta gradualmente, portando ad un aumento della generazione dei nemici.

La partita termina quando tutte le città del giocatore vengono distrutte. Questo porta ad una schermata che visualizza il punteggio finale e un bottone per ricominciare una nuova partita.

Possiamo quindi analizzare l'estratto del committente per meglio chiarire i vari aspetti riguardanti l'utente, in particolare nell'interagire con l'applicazione esso potrà visualizzare chiaramente le città (elementi da difendere), le torrette (elementi difensivi, atti a sparare missili) e tutte le entità nemiche (missili, aerei o satelliti) presenti nel cielo. L'interazione prevede l'utilizzo del mouse (o trackpad) con il quale l'utente, cliccando su una determinata zona del campo da gioco, sparerà un missile alleato direzionato verso quel punto.

- Il missile, una volta arrivato a destinazione, dovrà generare un'esplosione nell'esatto punto in cui l'utente ha cliccato;
- L'esplosione deve essere chiaramente visibile durante tutta la sua durata;
- Il gioco termina per l'utente quando tutte le città vengono distrutte;
- Una volta terminato il gioco l'utente deve essere adeguatamente avvisato dal gioco;

3.3 Requisiti funzionali

Sono stati identificati i seguenti requisiti funzionali:

1. L'utente deve avere la possibilità di sparare con le torrette per distruggere i nemici;
2. Un'esplosione continua a far danno durante tutta la sua durata, distruggendo i missili che entrano nel suo raggio d'azione (nel caso di esplosione alleata) e danneggiando città/torrette nel caso di esplosione nemica
3. L'utente deve avere la possibilità di giocare una nuova partita quando quella attuale termina.
4. La difficoltà deve crescere in base al trascorrere del tempo
5. Durante la simulazione, il giocatore deve sempre avere la possibilità di:

- (a) Visualizzare lo stato delle torrette (se sono in ricarica o pronte a sparare);
- (b) Visualizzare lo stato di salute delle città/torrette;
- (c) Visualizzare il punteggio attuale e il tempo trascorso in partita;
- (d) La GUI deve sempre essere aggiornata.

3.4 Requisiti non funzionali

Sono stati identificati i seguenti requisiti non funzionali:

1. L'applicazione sviluppata deve poter essere eseguibile su qualsiasi sistema operativo in grado di supportare il Java Runtime Environment;
2. L'interfaccia deve rispondere prontamente alle azioni dell'utente;
3. La visualizzazione grafica deve essere indipendente da quella logica adottata nel model
4. I missili nemici devono essere chiaramente distinguibili da quelli alleati
5. Il gioco deve essere fluido e, per questo motivo, avere almeno 20fps (frame per secondo)

3.5 Requisiti implementativi

Qui sono riportati i requisiti relativi all'implementazione del sistema:

- Il sistema deve essere sviluppato utilizzando il linguaggio Scala 3
- Si farà utilizzo di Java Swing per la realizzazione dell'interfaccia grafica
- Il sistema verrà testato tramite l'utilizzo di ScalaTest

4 Design architetturale

Durante il primo meeting e l'avviamento del progetto, il team di sviluppo ha provveduto a definire, ad alto livello, il design architetturale dell'applicazione.

Il risultato è stato poi raffinato durante gli step successivi, in modo da meglio rappresentare i concetti presenti nei requisiti.

Il risultato finale viene descritto nelle sezioni seguenti.

4.1 Architettura complessiva

L'architettura si basa sull'utilizzo di un unico componente, per gestire i diversi elementi dell'applicazione.

Come si nota nella figura sottostante è diviso in tre elementi principali:

- Il Model, costituito dai diversi componenti dell'applicazione che si occupano di gestire la logica del gioco;
- La View, che si occupa di mostrare i diversi elementi nell'interfaccia grafica, gestire l'interazione con l'utente e inviare gli eventi risultanti da azioni di quest'ultimo;
- Il Controller che ricopre il ruolo di intermediario fra View e Model e si occupa di gestire lo scambio di informazioni fra questi due elementi. Contiene, inoltre, gli eventi che la View può generare.

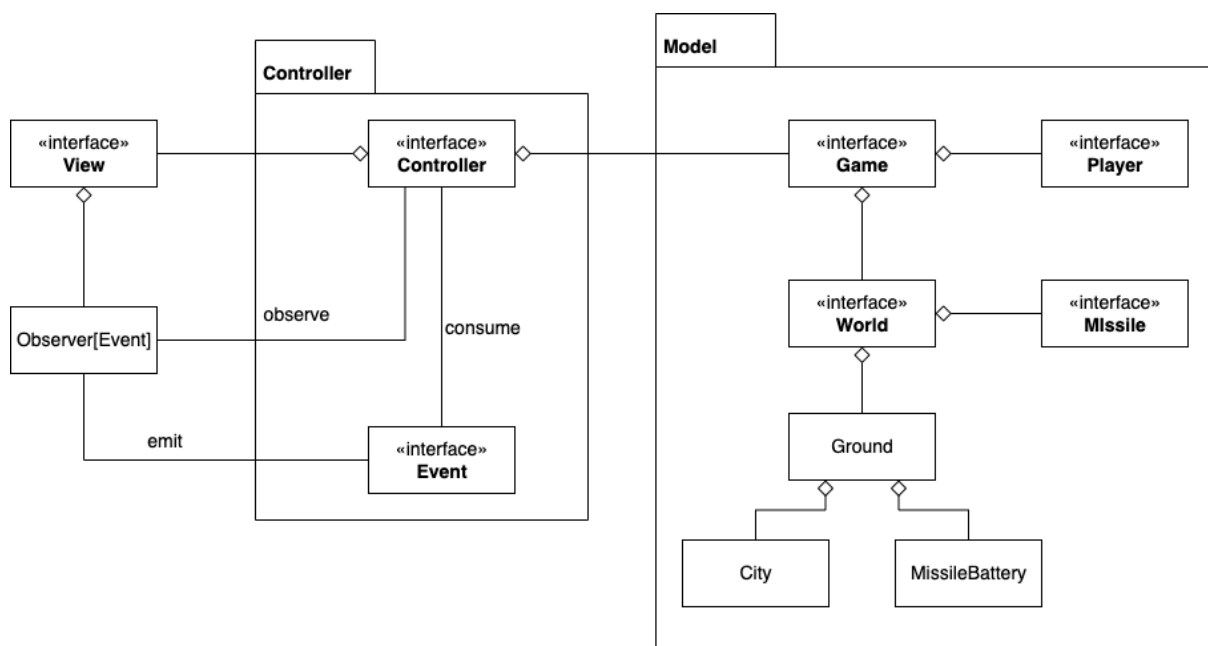


Figure 4.1: Architettura ad alto livello

4.2 Pattern architetturali utilizzati

Il progetto è stato realizzato seguendo un approccio MVC (Model, View, Controller) separando la parte di logica da quella grafica di presentazione, mantenendo

un'indipendenza tra logica di business e tecnologia grafica implementata.

Questo approccio ci ha permesso di lavorare sulla parte di logica e controllo nei primi sprint del progetto, per poi implementare un livello di presentazione in maniera totalmente indipendente dal modello dei dati e della logica di business.

I benefici sono:

- Manutenzione agevolata del controller/model senza coinvolgere la presentazione degli elementi di gioco
- Riutilizzo del codice qualora si decidesse di cambiare tecnologia grafica presentativa
- Loose coupling
- Maggior semplicità di testing

In riferimento allo schema precedentemente riportato (4.1) possiamo suddividere i vari elementi mostrati dell'architettura in un'ottica MVC:

- View: è catturata dall'interfaccia View, che ne definisce i metodi tramite i quali il controller interagisce con l'interfaccia grafica utilizzata
- Controller: è catturato dall'interfaccia Controller, il quale ne definisce il comportamento e l'interazione tra la parte di model e la parte di grafica.
Gestisce la serie di eventi generati dall'interfaccia grafica sfruttando il pattern Observable.
Gli eventi sono definiti dall'interfaccia Event.
- Model: racchiude una serie di interfacce e oggetti che definiscono gli elementi costitutivi del gioco, la loro logica di business e il loro modello dei dati interno

4.2.1 Event-Drive Architecture

Per gestire in maniera efficiente gli eventi generati dall'interfaccia grafica viene utilizzata un'architettura event-driven, realizzata tramite il pattern observer (che verrà analizzato in seguito).

Questo approccio fornisce uno stile architetturale secondo il quale ogni cambiamento di stato, nella grafica, genera un evento che verrà gestito dal controller.

In questa visione lo scorrere del tempo viene visto come un evento periodico.

Inoltre siccome le iterazioni devono essere successive, gli eventi generati vengono accodati e elaborati una alla volta.

L'implementazione di questo approccio è stata realizzata sfruttando la programmazione asincrona e gli Observable del framework Monix, come vedremo meglio nella sezione dedicata.

5 Design dettagliato

In questa capitolo mostriamo dettagliatamente l'architettura del gioco, analizzando più da vicino le sue componenti e relazioni. Il gioco è costituito da un'unica scena grafica, ovvero il campo da gioco, nel quale vengono mostrati tutti gli elementi presenti in campo e che gestirà le interazioni con l'utente, ovvero i click del mouse/trackpad all'interno del campo da gioco per sparare missili. Si è scelto di utilizzare il pattern MVC, come detto in precedenza, in accoppiata al cosiddetto Cake Pattern, per rendere l'architettura più orientata a componenti, facilitandone la manutenzione e testing e risolvendo in modo semplice le dipendenze dovute al pattern MVC, ovvero tra Model, View e Controller date dal seguente schema:

- $C \Rightarrow V$: il Controller comunica con la View per dire cosa visualizzare e quando aggiornare la visualizzazione grafica
- $V \Rightarrow C$: la View comunica, tramite eventi, il tipo di interazione che l'utente ha effettuato sull'interfaccia
- $C \Rightarrow M$: il Controller accede ai dati e alla logica di business incapsulati dentro agli elementi di Model

Questo è stato fatto in ottica di paradigma funzionale sfruttando i concetti di programmazione funzionale presenti in Scala.

5.1 Cake Pattern

Il Cake Pattern si basa sulla programmazione a componenti: un componente sfrutta una ben definita interfaccia verso l'esterno e definisce le dipendenze con altri componenti. L'idea è quella di creare per ogni componente:

- Un'interfaccia trait *T* che contiene le operazioni da mostrare all'esterno agli altri componenti
- Una trait *Provider* che mostra un unico oggetto val (singleton-like) di tipo *T* astratto (abstract val)
- Un interfaccia *Component* che implementa l'elemento astratto in una classe
- Un *Alias Type Requirements* che modella le dipendenze con altri componenti
- Una trait *Interface* che estende dal *Provider* usando il *Component* già implementato come mixin e specificando il *Self Type* pari al *Requirements*

In questo modo siamo in grado di creare un unico oggetto che estende tramite mixin le varie *Interface* definite. A questo punto tramite *Dependency Injection* delle implementazioni definite nei rispettivi *Component* possiamo poi istanziare i vari componenti.

5.1.1 ModelModule

Il ModelModule incapsula nella propria interfaccia Model un metodo per inizializzare il modello del gioco, ovvero il Game, esteso dall'*abstract val* presente nel Provider. Nel trait Component viene invece fornita l'implementazione dell'elemento astratto che inizializza e ritorna in output il modello di gioco istanziato.

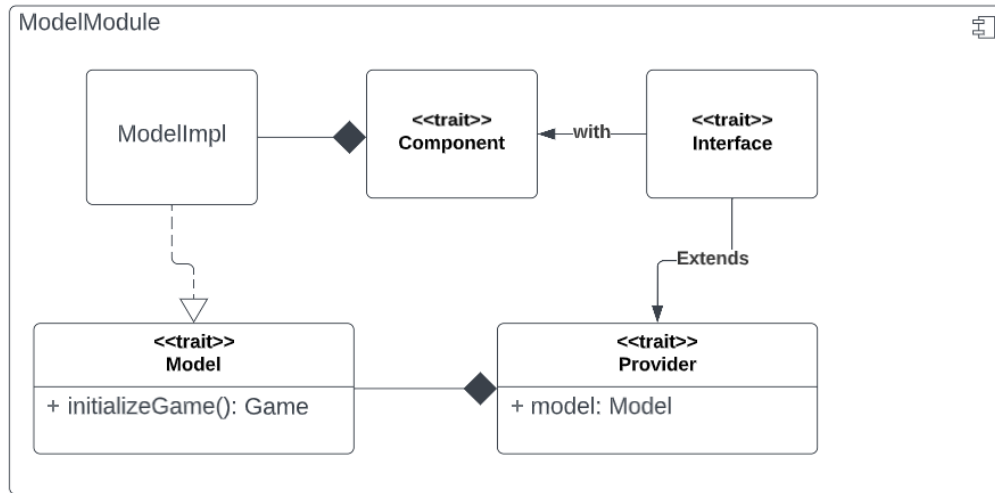


Figure 5.1: ModelModule con tutti i suoi componenti

5.1.2 ViewModule

Stessa cosa vale per il ViewModule: esso incapsula un Provider che fornisce un oggetto di tipo View, un *type member* Requirements che identifica le dipendenze necessarie (in questo caso solo il controller), un'interfaccia Component (con Requirements come *self type*) che incapsula l'implementazione della grafica e infine un interfaccia che estende da Provider e fornisce l'implementazione con il Component.

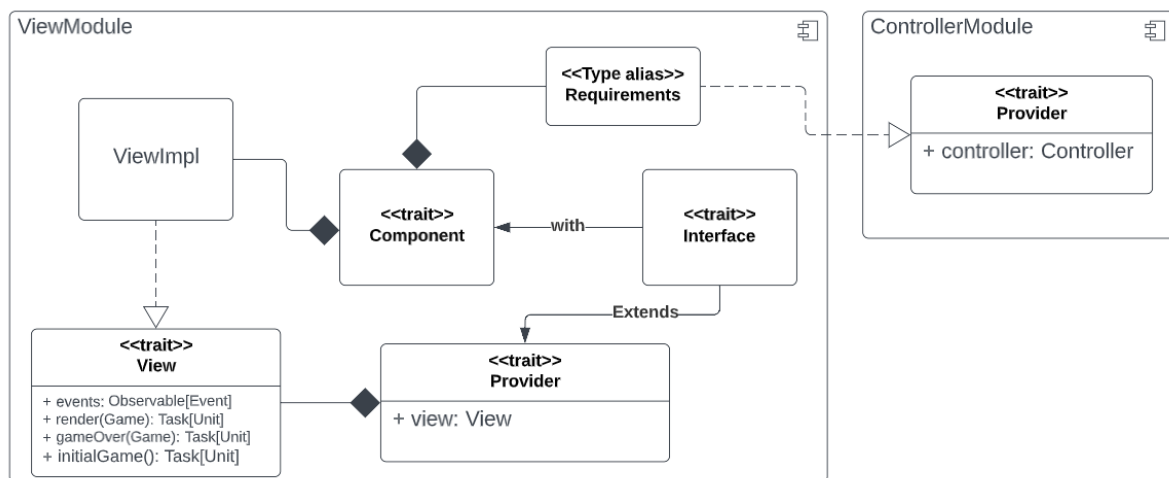


Figure 5.2: ViewModule con tutti i suoi componenti

5.1.3 ControllerModule

Anche il Controller è definito come il ViewModel: esso incapsula un Provider che fornisce un oggetto di tipo Controller, che altro non è che un'interfaccia con un unico metodo *startGame*. Anche qui troviamo un *type member* Requirements che identifica le dipendenze necessarie (in questo caso il provider del model e della view), un'interfaccia Component (con Requirements come *self type*) che incapsula l'implementazione del controller e infine un'interfaccia che estende da Provider e fornisce l'implementazione con il Component.

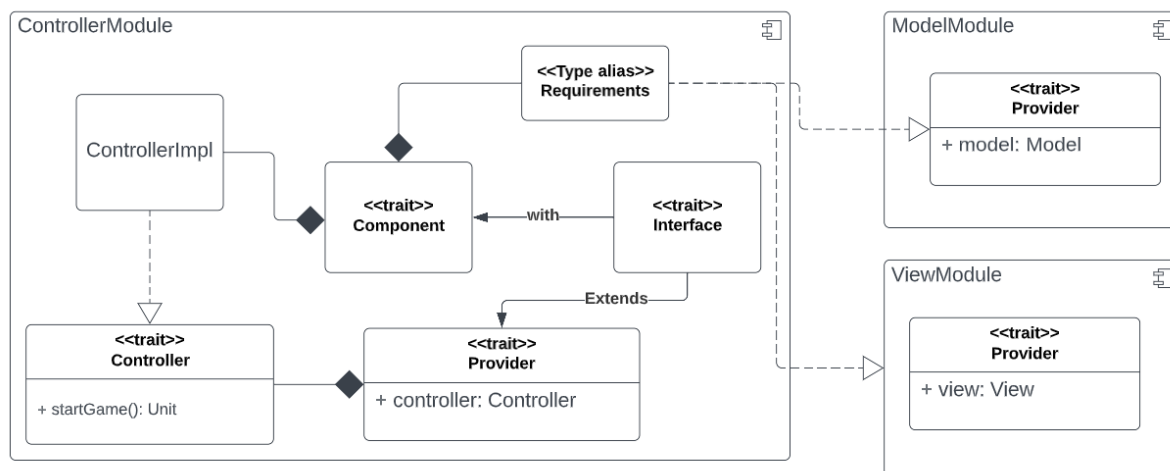


Figure 5.3: ControllerModule con tutti i suoi componenti

5.2 Gestione del Controller

Tutto ciò relativo al Controller e alla gestione degli eventi è racchiuso nel package *controller*, nel quale notiamo i seguenti file:

- Event: enum che comprende i vari eventi che possono essere generati:
 - StartGame: evento per inizio del gioco, generato quando si clicca sul bottone per iniziare o rigiocare;
 - TimePassed: evento che rappresenta lo scorrere del tempo (l'intervallo trascorso è preso come parametro);
 - LaunchMissileTo: evento generato quando l'utente clicca nel gioco. Richiede il lancio di un missile alleato ad una determina coordinata (presa come parametro);
- TimeFlow: rappresenta un Observable che emette un valore ogni delta time specificato trascorso. Permette di gestire il passaggio del tempo;
- Update e relative implementazioni: Update modella la gestione di un generico evento di aggiornamento del gioco, in particolare gestisce gli eventi del trascor-

rere di un intervallo di tempo e di un lancio di un missile.

Per comodità viene creato un Update diverso per ogni macro operazione ritenuta rilevante nel ciclo di gioco.

In particolare gli update presi in considerazione (descritti in ordine d'esecuzione) sono:

1. `UpdateTime`: attivato dall'evento `TimePassed` e aggiorna il tempo di tutti gli elementi nel gioco (non tutti gli elementi hanno comportamenti determinati dal tempo). Si occupa inoltre di eliminare gli elementi per cui il tempo è scaduto;
2. `UpdatePosition`: attivato dall'evento `TimePassed` e permette, una volta aggiornato il tempo degli elementi in gioco, di farli muovere (non tutti gli elementi si possono muovere).
Infatti la distanza percorsa dipende dal tempo passato, tra le altre cose;
3. `ActivateSpecialAbility`: attivato dall'evento `TimePassed` e permettere ad alcuni elementi, che hanno abilità speciali (come generare nemici o esplodere se raggiunta una determinata destinazione), di attivarle.
In questo update lo spawner genera nuovi elementi, se possibile;
4. `CollisionsDetection`: attivato dall'evento `TimePassed` e controlla le collisioni tra gli elementi.
In seguito aggiorna il punteggio del giocatore, se qualche nemico viene distrutto.
Inoltre elimina gli elementi che sono stati distrutti;
5. `LaunchNewMissile`: attivato dall'evento `LaunchMissileTo` e permette di creare un nuovo missile alleato nel mondo;

Gli Update specifici non sono nuove classi, ma vengono implementati tramite `strategy`, passata e generati tramite il metodo `on in Update`.

L'evento `StartGame` viene direttamente gestito dal controller, non essendo un elemento del ciclo di gioco;

- `GameController`: questa classe rappresenta il controller del gioco, ovvero la gestione degli eventi (unendo quelli generati dalla grafica al timer) e l'ordine in cui devono essere gestiti gli update da eseguire.
Il controller permette di fare da raccordo con la parte di view, infatti utilizza i metodi pubblici della grafica per mostrare gli stati del gioco in base agli eventi.
Gestisce infine la terminazione del gioco;

Per avere un gioco fluido (requisito non funzionale numero 5) viene definito come intervallo di tempo tra un evento temporale e l'altro il valore di 50 ms (questo permette di avere 20fps come richiesto).

Questo implica però che ogni iterazione del gioco richieda meno di 50ms, cosa che si verifica solo se il numero di elementi in gioco non è troppo elevato.

Il diagramma UML sottostante (5.4) rappresenta l'implementazione descritta in prece-

```
classDiagram
    class Event {
        <<enum>>
    }
    class Update {
        <<trait>>
        + andThen(control: Update): Update
    }
    class GameController {
        + start(): Task[Unit]
    }
    class TimeFlow {
        + tickEach(duration): Observable[Long]
    }
    class UpdateBox {
        + on[E < Event]((E, Game) => Task[Game]): Update
        + Combine(engines: Update*): Update
    }
    class TaskGame {
        (Event, Game) => Task[Game]
    }

    Event <|-- LaunchMissile
    Event <|-- TimePassed
    Update <|-- UpdateTime
    Update <|-- UpdatePosition
    Update <|-- ActivateSpecialAbility
    Update <|-- CheckCollision
    Update <|-- LaunchNewMissile

    UpdateBox <|-- Update
    UpdateBox ..> Update : 1..* {ordered}
    UpdateBox ..> TaskGame : (Event, Game) => Task[Game]
    UpdateBox ..> Event : <<parameter>>
    UpdateBox ..> Event : <<use>>

    GameController "1" *-- "1..*" Update
    GameController ..> UpdateBox : <<Bind>> <E->LaunchMissile>
    GameController ..> TimeFlow : <<use>>
    TimeFlow ..> Observable : Observable[Long]
```

5.3 Gestione della View

- Un metodo *events* che ritorna un oggetto Observable[Event], tramite il quale il controller può fare subscribe ed essere notificato per esempio nell'evento click
- Un metodo *initialGame* che ritorna un Task asincrono che mostrerà la scena di inizio del gioco
- Un metodo *render* che prende in input il model del Game (contenente tutto il necessario che deve essere renderizzato) e ritorna un Task da eseguire in modo asincrono per effettuare la presentazione grafica della nuova scena
- Un metodo *gameOver* che ritorna un Task asincrono che mostrerà la scena di game over una volta che il gioco è terminato

La classe `WorldPane` estende da `JPanel`: essa si occupa di creare un pannello nel quale mostrare tutti gli elementi presenti in gioco, mentre un secondo pannello, che viene mostrato al termine del gioco, permette di visualizzare informazioni come lo score raggiunto e il tempo totale di sopravvivenza.

WorldPane prende in input il nuovo model del Game. In particolare, sempre WorldPane, mostra tutte le città, torrette, missili ed esplosioni partendo da un set di elementi *Collisionable* e Ground (che vedremo nella sezione di Model) e convertendoli in elementi grafici sfruttando due moduli che fanno da intermediario tra il model e la view:

- Visualizer: dato un *Ground*, ovvero il model del campo di battaglia da difendere, lo converte in una lista di città e torrette grafiche, con immagine e posizione ad esempio
- CollisionableVisualizer: dato un set di Collisionable ritorna un set di CollisionableElement grafico, che incapsula l'immagine, la posizione e altre informazioni utili alla rappresentazione grafica

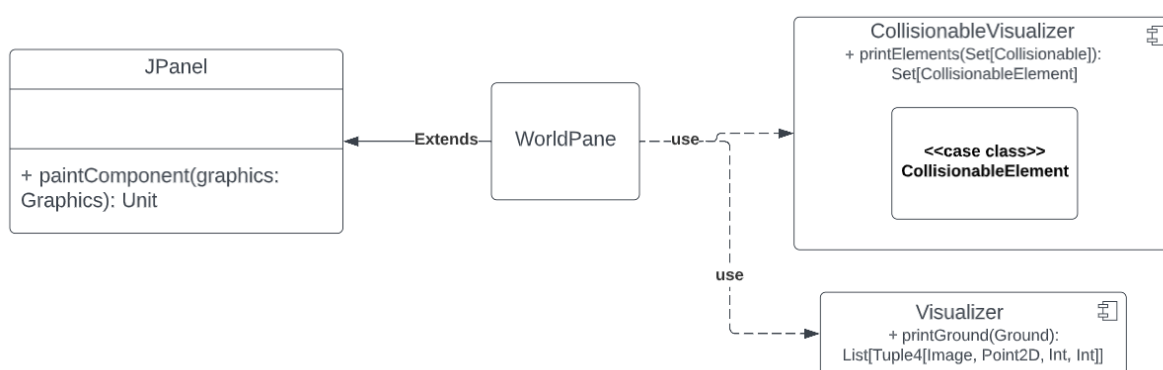


Figure 5.5: Pannelli di gioco e moduli di visualizzazione grafica

5.3.2 GUI

Questo modulo rappresenta l'implementazione della trait *View* che utilizza come tecnologia grafica il framework *JavaSwing*. In particolare all'interno di questo modulo troviamo una classe che estende da *View* al cui interno è presente un oggetto di tipo *JFrame* opportunamente settato (altezza e larghezza ecc...) e le implementazioni dei tre metodi sopra spiegati, sfruttando le *SwingUtilities* e il frame per mostrare la nuova scena (*WorldPane*) ad ogni evento di tipo render.

Il metodo *events* ritorna un *Observable[Event]* che ad ogni click del mouse sul pannello di visualizzazione genera un nuovo evento che mappa le coordinate di click in un evento di tipo *LaunchMissileTo* con le coordinate rilevate.

A questi eventi aggiunge anche quelli generati quando si clicca sul bottone per giocare (evento *GameStart*).

Questo *Observable* viene osservato dal Controller.

Il generatore di eventi click, sia sul campo che del bottone, è modellato dal modulo *MouseHandler*, che tramite *extension method* su *Component* e *JButton* di *Swing*, offre un nuovo metodo che crea un *Observable*: quest'ultimo al suo interno aggiunge un mouse listener al componente grafico in questione (nel nostro caso il *JFrame* e il *JButton*) e ad ogni click emette un nuovo evento con il metodo *onNext* catturando il punto di click con un *Point2D* (*JFrame*) o il semplice click (*JButton*).

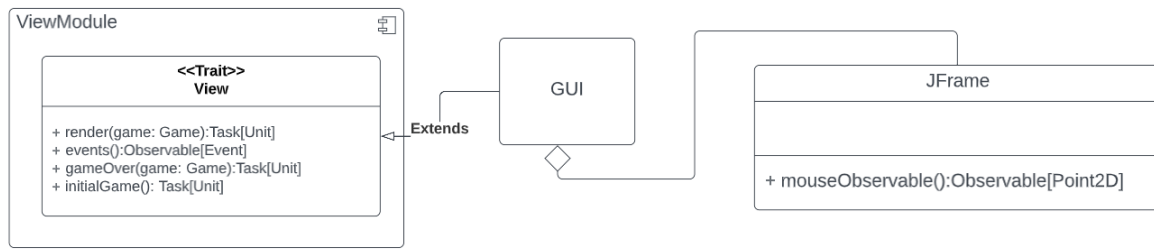


Figure 5.6: Classe GUI e relazioni

Il metodo `initialGame` mostra la schermata iniziale del gioco. Il metodo `render` semplicemente mostra un nuovo `WorldPane` usando il model del mondo aggiornato, mentre il metodo `gameOver` mostra il pannello per notificare all'utente che la partita è terminata.

5.4 Gestione Model

Il package *model* contiene tutte le definizioni che riguardano il modello delle entità presenti nel gioco, con i rispettivi dati e la logica di business. Nella root del package troviamo la modellazione delle entità principali del sistema, come ad esempio:

- **Player:** modella il giocatore, quindi il suo score e il timer di sopravvivenza;
- **World:** definisce il mondo di gioco, ovvero un `Ground` (insieme di città da difendere e le torrette) e un set di elementi attivi *Collisionable*, ovvero qualsiasi elemento che può collidere (come missili ad esempio);
- **Game:** rappresenta una singola partita, incapsulando un riferimento al `Player`, al `World` corrente e ad uno `spawner` aggregato di oggetti generici;
- **Timer:** definisce il timer virtuale globale, utilizzato per misurare il tempo di sopravvivenza del giocatore;
- **WorldActions:** definisce le operazioni che si svolgono nel mondo;
- **PlayerActions:** definisce le operazioni che si svolgono nel giocatore;
- **Scorable:** un elemento danneggiabile (*Damageable*) che se distrutto permette di ottenere un punteggio;

Il diagramma sottostante (5.7) mostra la gerarchia per la gestione del `World`, `Player` e `Game`.

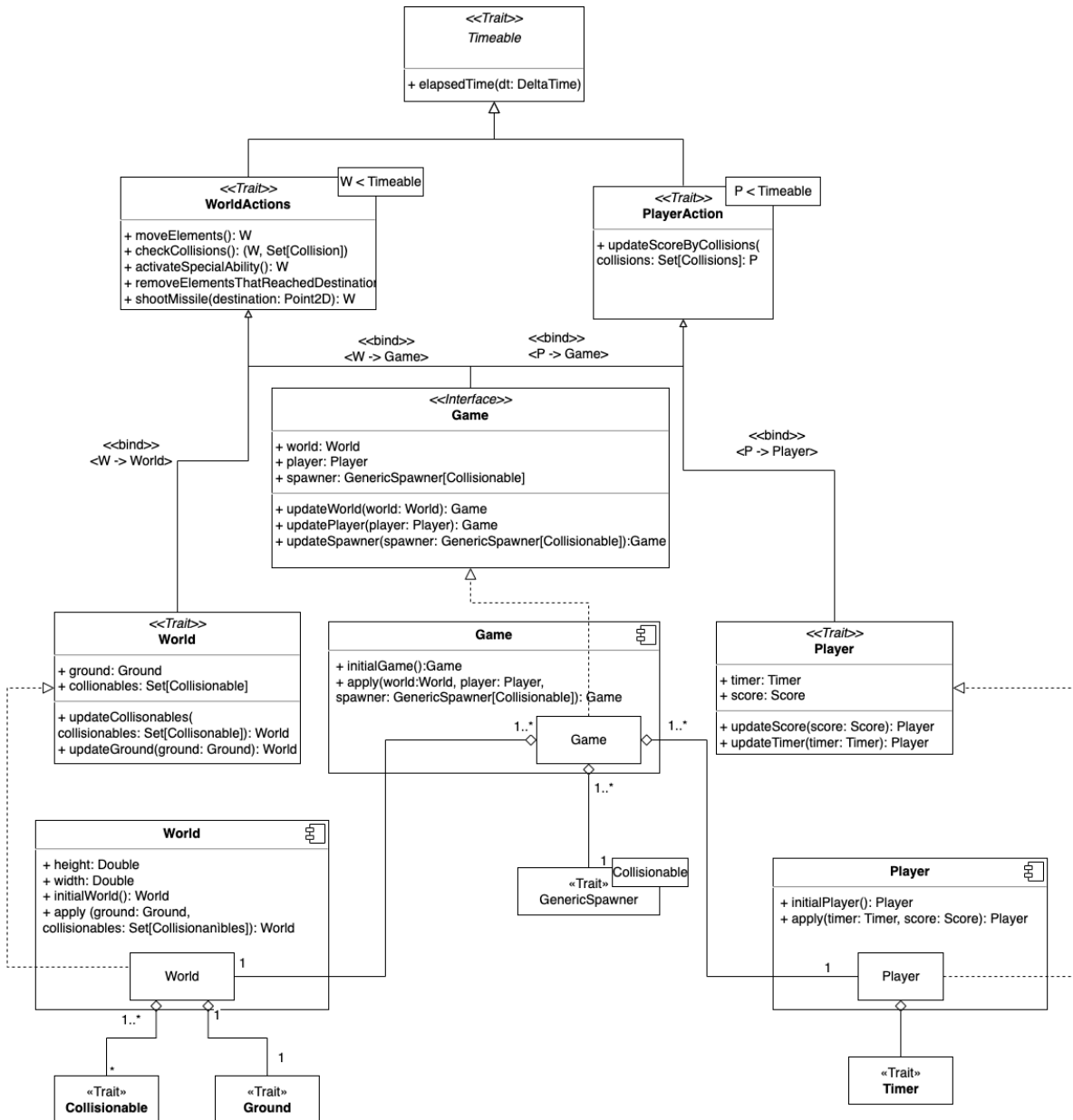


Figure 5.7: Trait e implementazioni di Game, World e Player

Per vedere Scorable consultare la sezione sulle collisioni e il diagramma 5.10. L'approccio per cui Game contiene World e Player, estendendo dalle stesse interfacce da cui quest'ultimi estendo, rispecchia il *Pattern Decorator*.

5.4.1 Behavior

Nel modulo *behavior* troviamo due trait molto importanti all'interno del sistema progettato:

- Timeable: identifica qualsiasi entità che necessita della rappresentazione del tempo che passa; ad esempio è implementata da Timer, ovvero l'entità che mantiene la rappresentazione globale del tempo che scorre da inizio partita fino al termine;

- Moveable: identifica qualsiasi entità che si muove nello spazio di gioco al passare del tempo;

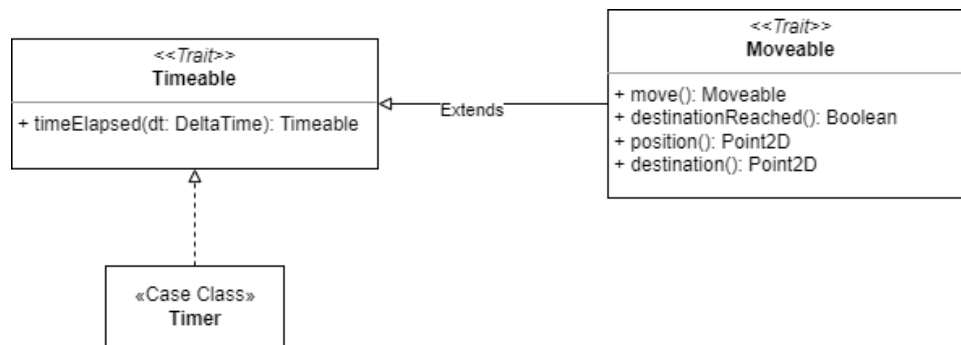


Figure 5.8: Classe GUI e relazioni

5.4.2 Collisioni

Per gestire le collisioni sono stati necessari i seguenti concetti:

5.4.2.1 HitBox

Una HitBox rappresenta lo spazio geometrico di una figura, in particolare sono un'area con un'insieme di punti.

Le sue implementazioni permettono di rappresentare molte figure geometriche:

- HitBoxEmpty: rappresenta un'area senza punti, è un object perché l'insieme vuoto è unico;
- HitBoxPoint: rappresenta un'area con un solo punto;
- HitBoxCircular: rappresenta un'area circolare. Necessita di un punto (il centro) e un raggio;
- HitBoxRectangular: rappresenta un'area rettangolare. Necessita di un punto (il centro), una base, un'altezza e un angolo di rotazione;
- HitBoxIntersection: rappresenta un'area generata dall'intersezione di più aree (particolarmente utile per calcolare le sovrapposizioni di due aree);
- HitBoxUnion: rappresenta l'unione di due o più aree, permette la creazione di aree di forma complessa;

La strategia descritta permette la creazione di aree più o meno complesse ed è particolarmente adattabile alle necessità.

Questo approccio è in linea con il *Composite Pattern*. Inoltre, tramite questa implementazione è possibile verificare se due aree hanno dei punti in comune tramite intersezione. Se l'intersezione è vuota allora non si sovrappongono (questo risultato è particolarmente utile per verificare se due oggetti collidono).

La creazione dei trait HitBoxSymmetric e HitBoxAggregation permette, tramite l'uso di

template method di evitare la ripetizione di codice.

La figura seguente (5.9) mostra il diagramma UML della struttura delle hitbox.

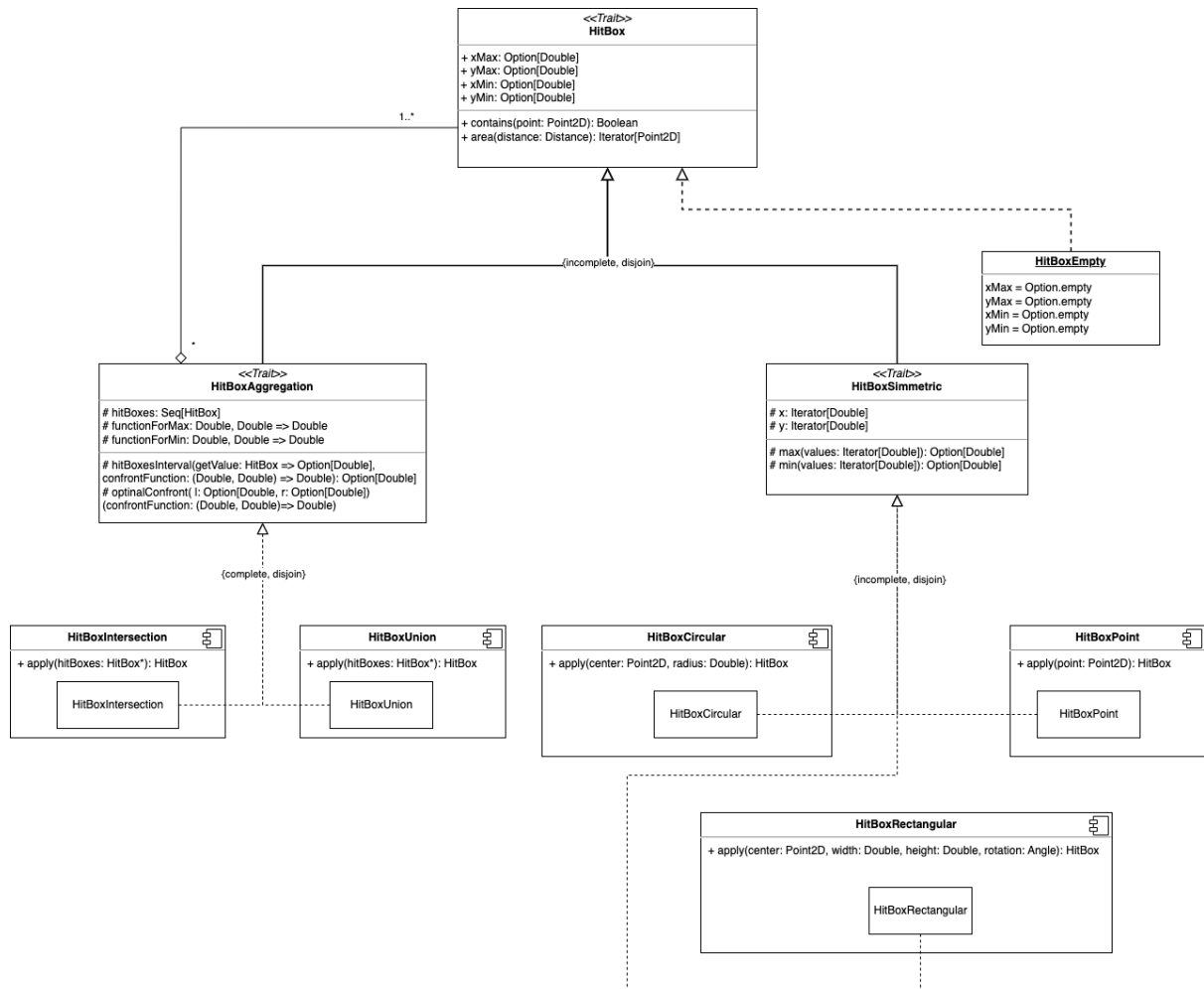


Figure 5.9: Gerarchia HitBox

5.4.2.2 Collisionable

Un Collisionable è un qualunque oggetto che ha una presenza nel mondo fisico. Fornisce anche l'affiliazione dell'elemento in modo da capire se è neutro, nemico o alleato. I collisionable si dividono in:

- **Damageable:** Un elemento dello spazio fisico che può essere danneggiato e ha quindi una vita;
- **Damager:** Un elemento dello spazio fisico che può applicare un danno ad un altro Collisionable;

Si noti che teoricamente è possibile che un oggetto sia entrambi.

Per raggruppare un insieme di Collisionable si usa un Set, questo perché non vi è un ordine.

Tramite extension methods al Set di Collisionable, si permette di calcolare le collisioni.

Una volta calcolate le collisioni, gli elementi in essa coinvolti vengono danneggiati, se Damageable, o infliggono danno, se Damager.

La figura seguente (5.10 mostra il diagramma UML della struttura dei Collisionable e Scorable.

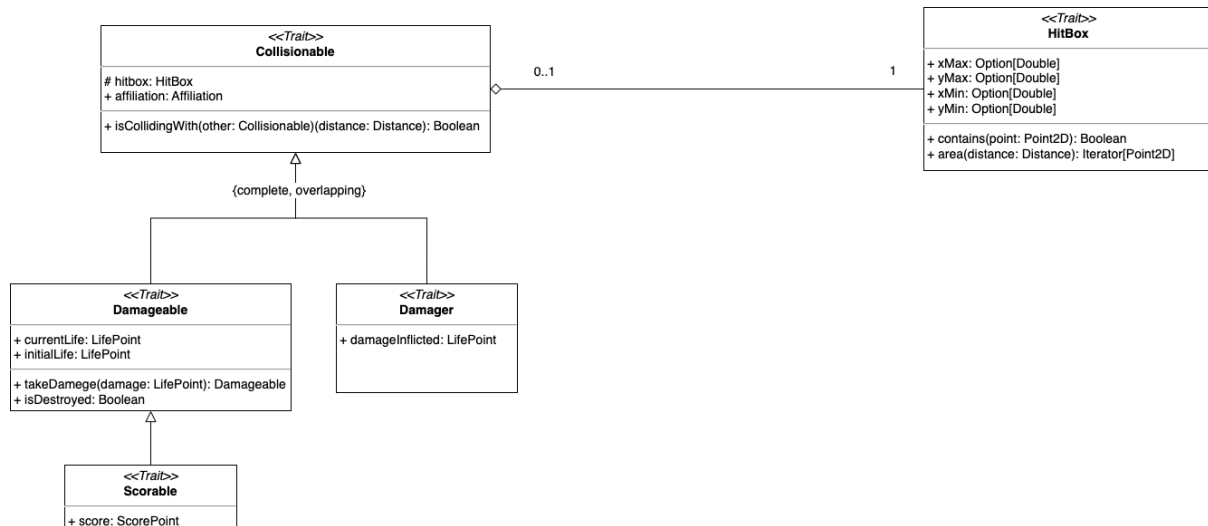


Figure 5.10: Gerarchia Collisionable

La divisione in concetti tra Collisionable e HitBox rispecchia il *pattern Bridge*, infatti questi due concetti sono ortogonali tra di loro.

Una collisione viene rappresentata come un Set di Collisionable (chiamato Collision) al quale sono aggiunti, tramite extension methods, le azioni necessarie.

Un'insieme di collisioni è rappresentata con un Set di Collision e viene utilizzato, tramite extension methods a ScorePoint (il punteggio), per calcolare il nuovo punteggio del giocatore.

5.4.3 Missili e esplosioni

I missili sono stati modellati partendo dalla definizione di qualcosa suscettibile a collisioni con altri elementi di gioco (Collisionable) e dalla definizione di qualcosa di danneggiabile (Damageable). Questo perché un missile è caratterizzato da una propria vita, che, diminuendo a seguito di danni esterni, può portare alla sua esplosione in caso di punti vita azzerati.

Inoltre un missile può muoversi da una posizione di partenza verso una posizione di destinazione, in moto lineare lungo il vettore che congiunge i due punti, perciò estende da Moveable.

Come possiamo vedere dall'immagine (5.11), un missile è caratterizzato fondamentalmente da una velocità, da una posizione, una destinazione, una vita, un danno da infliggere e una funzione di esplosione: quest'ultima viene utilizzata qualora il missile venga distrutto o raggiunga la destinazione per generare un'elemento *Explosion* nella sua esatta posizione. Un missile è anche associato ad una Affiliation, che ne indica la tipologia: *Enemy* o *Friendly*.

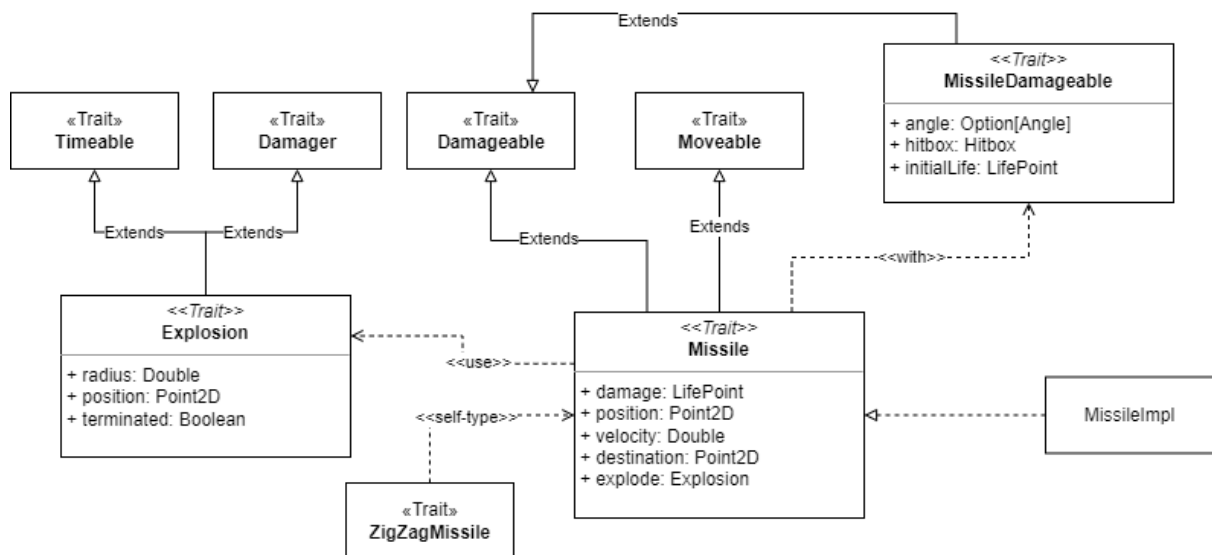


Figure 5.11: Relazioni all'interno del package missile

In questo modo un missile Enemy potrà infliggere danno solo ad elementi Damageable con affiliazione Friendly e viceversa, ma non a elementi di stessa affiliazione. Inoltre l'esplosione generata da un missile dovrà anch'esso "ereditare" la stessa affiliazione del missile parent che lo genera: questo perché si è deciso di sfruttare l'esplosione per infliggere danno e non il missile stesso, che incapsula soltanto la quantità di danno da infliggere mediante l'esplosione.

L'esplosione, al contrario del missile, è un Damager, in quanto infligge danno e non ne subisce ed è caratterizzato, oltre dal danno, anche da un raggio di azione e da una durata nel tempo: per questo motivo estende da Timeable.

Oltre al missile lineare nemico è stato sviluppato il missile con traiettoria variabile a *zig zag*, come descritto negli obiettivi opzionali.

Partendo dalla base del missile lineare si è deciso di fare override solo dei metodi interessanti per realizzare il comportamento voluto, con l'aggiunta di una sequenza di *sub destination* da raggiungere per il cambio rotta e un punto finale designato come destinazione nel quale esplodere.

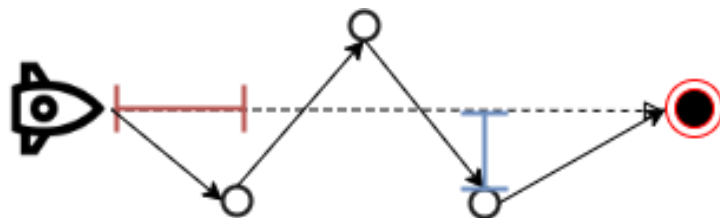


Figure 5.12: Esempio di generazione traiettoria a zig zag

La sequenza di punti, che ne definisce il percorso, può essere generato con cambio di rotta di 90° , alternando prima un cambio a destra e poi un cambio a sinistra oppure una delle due casualmente. Come possiamo vedere dalla figura 5.12 la distanza rossa rappresenta lo step per ogni cambio rotta lungo la traiettoria standard, calcolata in base al numero di cambi rotta che vogliamo effettuare. La distanza blu indica invece quanto

vogliamo traslare il punto dalla sua originale posizione (a destra, sinistra o casualmente). L'ultimo punto invece è rappresentato dal punto finale di destinazione.

5.4.4 Città e Torrette

Le città e le torrette sono state modellate partendo dalla necessità di avere due oggetti che interagiscano con le hitbox degli elementi di gioco e che siano oggetti danneggiabili da missili nemici (Damageable).

Entrambi hanno caratteristiche comuni quali:

- Vita: Entrambi hanno una determinata vita iniziale, che può essere modificata attraverso i file di configurazione. Essa diminuisce quando viene danneggiata da un missile nemico;
- Affiliation: Entrambe le strutture sono considerate alleate;
- Hitbox: Entrambe le strutture hanno le stesse hitbox di default. Possono essere specificate e modificate manualmente nei file di configurazione.

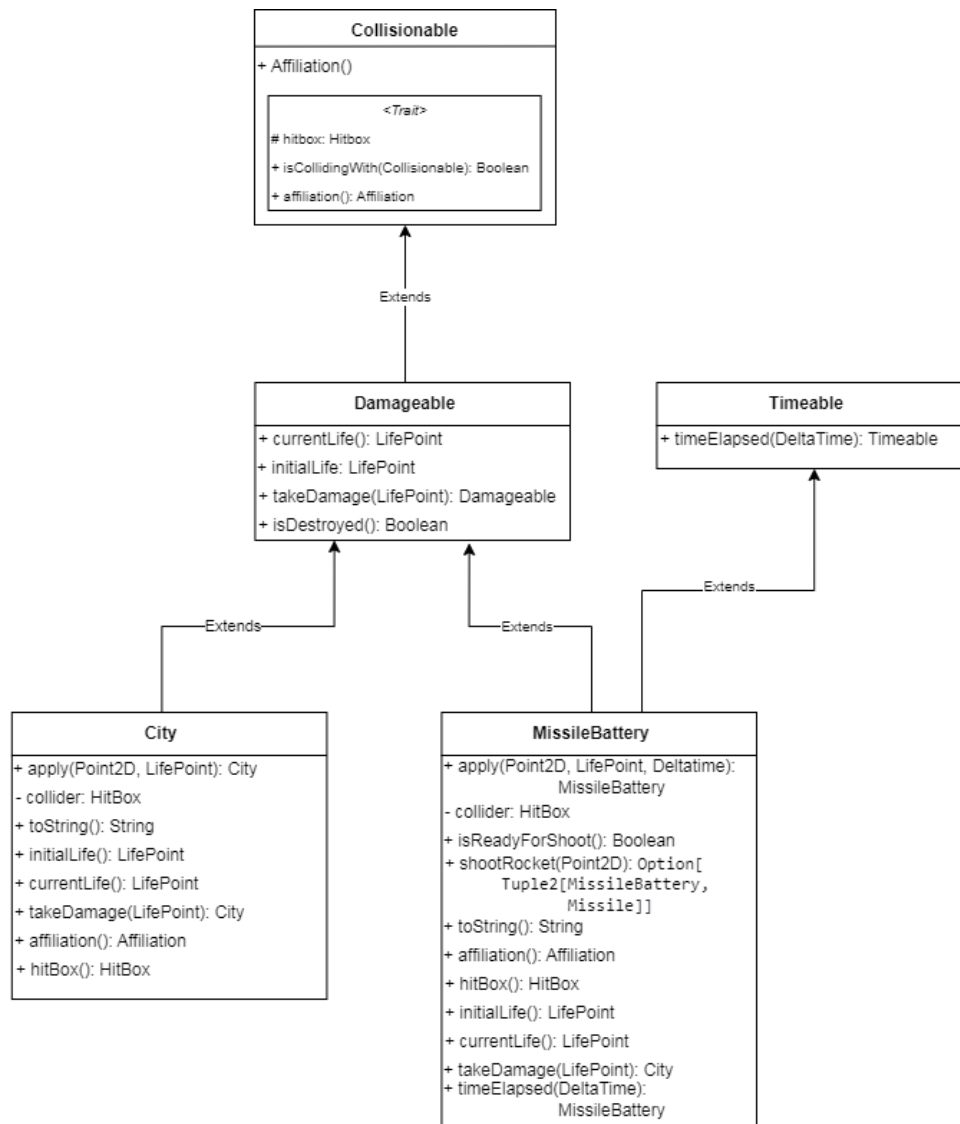


Figure 5.13: Design delle città e torrette.

Per il calcolo delle collisioni, il meccanismo utilizzato si basa sul passare il punto a sinistra che si trova sulla base. In questo modo, per ottenere le hitbox, è sufficiente soltanto 1 punto e le relative dimensioni logiche della struttura (che sono calcolate e all'interno dei file di configurazione).

La differenza tra le due strutture è la capacità delle torrette di lanciare missili dopo un determinato lasso di tempo prefissato.

Infatti, il missile implementa un sistema di tempo virtuale (Timeable) che viene condiviso anche da altri componenti, che permette di gestire l'andamento del tempo nel gioco in un modo separato rispetto al tempo reale.

Quando arriverà una richiesta alla torretta di lanciare un missile, prima viene eseguito un check per vedere se la torretta ha ricaricato il missile.

Una volta che viene valutata la possibilità della torretta nello sparare, viene utilizzata la posizione finale (che corrisponderà alla posizione del click sulla GUI) di dove il

missile dovrà esplodere per generare il missile e la nuova torretta con tutte le informazioni aggiornate.

5.4.5 Ground

Il ground ha lo scopo di gestire tutte le funzionalità per quanto riguarda la creazione e gestione di tutte le strutture presenti del gioco.

All'interno della mappa è presente un ground, che all'inizio della partita genera tutte le città e le torrette in base alle distanze che sono state impostate all'interno del file di configurazione.

Queste distanze utilizzano valori predefiniti che vengono scalati in base alla grandezza del mondo grafico e logico.

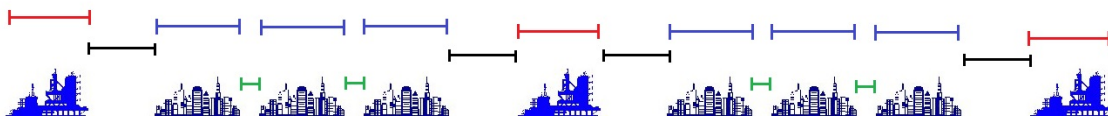


Figure 5.14: Visualizzazione degli spazi calcolati tra le strutture.

In particolare, dalla figura sopra (5.14, si possono distinguere 4 differenti misure:

- Rosso: Calcola la grandezza delle torrette;
- Blu: Calcola la grandezza delle città;
- Nero: Calcola la distanza tra la città e la torretta;
- Verde: Calcola la distanza tra le città.

Inoltre, il ground implementa metodi quali:

- Check dell'end game: Viene utilizzato per verificare se il gioco è terminato. Infatti, la partita termina quando tutte le città sono state distrutte (non importa la quantità di torrette vive/distrutte);
- Gestione lancio missile: La gestione del lancio del missile è affidata al ground. Il ground prima ottiene tutte le torrette che sono disponibili per sparare e poi ottiene la batteria missilistica più vicina. A questo punto, chiama il metodo per sparare il missile;
- Metodi di utility: Metodi usati per fornire utilità varie, quali:
 - Numero di città vive;

- Numero di batterie missilistiche vive;
- Numero di batterie missilistiche pronte per sparare;
- Metodi usati per infliggere danno ad una o più strutture specifiche.

Il ground lo si può vedere come un'entità contenitore, che serve per aggiungere un livello di astrazione tra le strutture e le città.

In questo modo, il controller non deve eseguire azioni direttamente sulle strutture, ma si interfaccia solamente con il ground, facendo da mediatore.

5.4.6 Spawner

Questo modulo è composto dai seguenti sotto-moduli:

- *Spawnable*: incapsula la definizione della strategia di spawning di un determinato elemento (*Spawnable*) e la creazione di Set di elementi dato un determinato *Spawnable*. Sostanzialmente una funzione che opera come supplier generico di oggetti
- *SpecificSpawners*: è una raccolta di *Spawnable* implementati, come ad esempio uno per i missili lineari, uno per i missili a zigzag e uno per gli aerei nemici
- *GenericSpawner*: identifica un generico spawner di elementi, che sfrutta uno *Spawnable* per generare una serie di elementi da spawnare nel tempo
- *SpawnerAggregator*: aggregatore di più *GenericSpawner* per generare elementi *Collisionable* differenti nello stesso istante

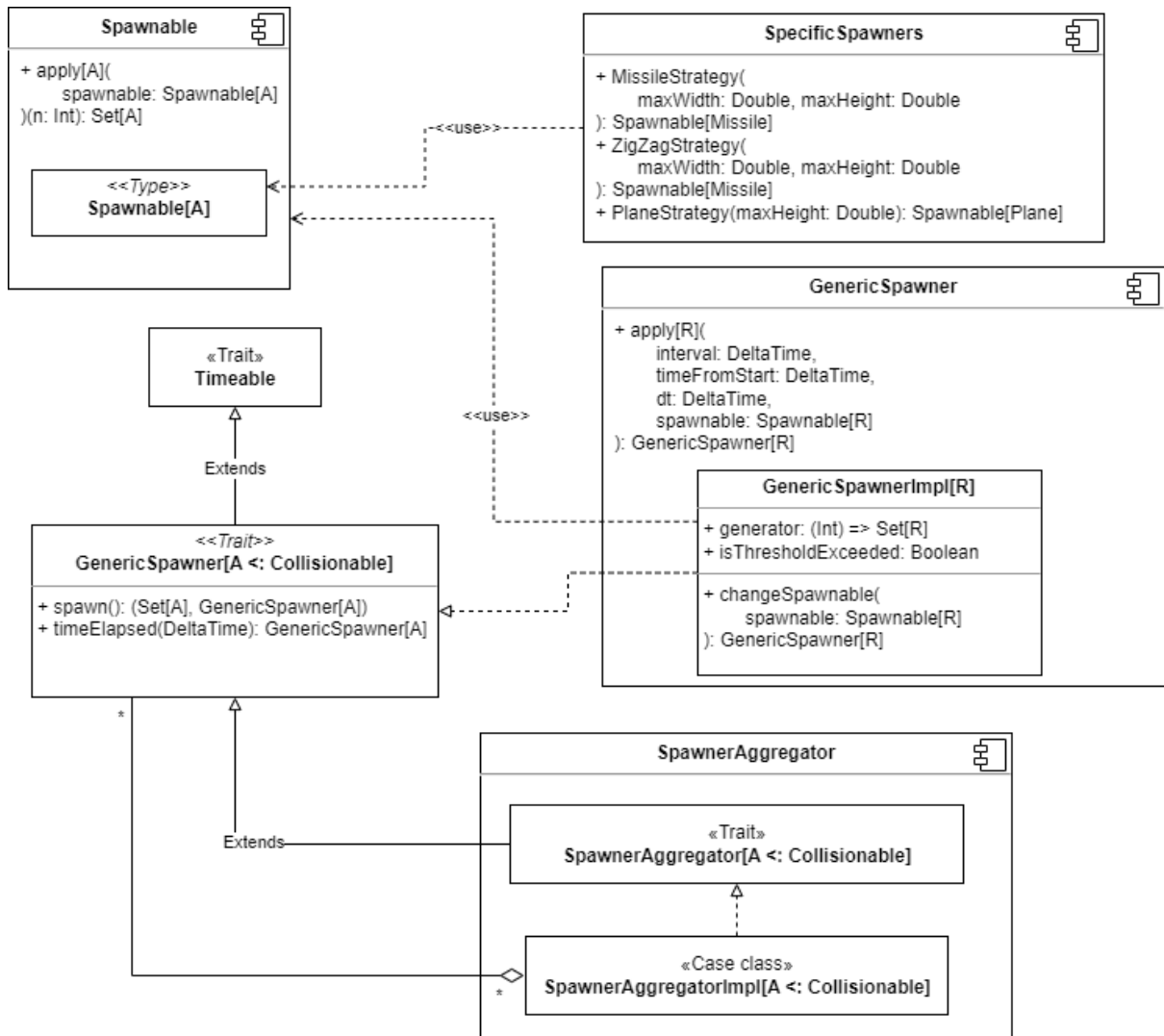


Figure 5.15: Schema UML dei moduli di spawning

Per generare oggetti dopo un certo intervallo di tempo si è scelto di realizzare un cosiddetto *GenericSpawner*, che data la tipologia dell'elemento ne genera una determinata quantità in base al tempo passato (uno ad ogni certo intervallo di tempo trascorso).

Lo Spawner estende da *Timeable* in quanto ha bisogno di sapere il delta time e il tempo virtuale per decidere se generare elementi o meno.

Estendendo da *Timeable* è in grado di avere una concezione del tempo che passa tramite il metodo *timeElapsed*.

Per gestire l'aumento di difficoltà, il *GenericSpawner* decrementa l'intervallo iniziale dopo un certo tempo trascorso, secondo una funzione sigmoideale basata sul valore dell'intervallo di partenza stesso.

Per diversificare le strategie di spawning (ovvero quale elemento generare e come) si è realizzata un'interfaccia *Spawnable*, anch'essa generica, con un unico metodo *generate*: quest'ultimo identifica uno specifico modo di generare l'oggetto di tipo specificato, implementato mediante delle High-Order Functions per ogni tipologia.

Un'altro aspetto importante è lo *SpawnerAggregator*: dalle meccaniche di gioco analizzate nella sezione dei Requisiti, si evince che possono essere diversi gli elementi che generano missili nemici, e diversi sono i nemici che possono essere spawnati.

Lo SpawnerAggregator è stato pensato in quest'ottica, ovvero come wrapper di una moltitudine di generici Spawner, pur sempre estendendo da GenericSpawner, in quanto è anche se stesso uno spawner generico.

Questo approccio è in linea con il *Composite Pattern*.

5.4.7 Altri tipi di nemici

Oltre ai missili, sono presenti anche altri 2 tipi di nemici: Aerei e Satelliti. Questi tipi di entità hanno la capacità di generare altri nemici, nel mentre si muovono lungo il percorso nella mappa.

Aerei

Gli aerei sono entità che possono essere danneggiate dai missili del giocatore (per questo sono una estensione dell'interfaccia Damageable) e inoltre possono muoversi all'interno della mappa (estensione di Moveable). Inoltre, essendo oggetti che dipendono dal tempo ed essendo un'estensione di uno spawner, deve dipendere da Timeable e GenericSpawner. Sono in grado quindi di generare missili.

I plane, quando vengono generati, possono seguire solamente una direzione lineare, ad una determinata altezza e con una determinata direzione. Inoltre, essendo che sono una estensione di GenericSpawner, quando vengono generati devono impostare una strategia di spawning dei missili.

Satelliti

I satelliti sono entità con le stesse identiche funzionalità degli aerei. L'unica differenza con essi è che non estendono l'interfaccia Moveable. Infatti, essendo degli oggetti statici nel mondo, non hanno bisogno di movimento.

5.5 Pattern utilizzati

Factory Method

È uno dei design pattern fondamentali per l'implementazione del concetto di factory. Come altri pattern creazionali, esso indirizza il problema della creazione di oggetti senza specificarne l'esatta classe. Questo pattern raggiunge il suo scopo fornendo un'interfaccia per creare un oggetto, ma lascia che le sottoclassi decidano quale oggetto istanziare.

Nel nostro caso, il pattern è stato largamente usato, ad esempio nella generazione di aerei, missili, strategie di spawning, torrette, etc.

Singleton

Il singleton è un design pattern creazionale che ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza, e di fornire un punto di accesso globale a tale istanza. Nel caso di Scala, questo è ottenuto grazie all'utilizzo di un Object, che implementa le

funzionalità del singleton.

Nel progetto, è possibile vedere questo utilizzo ad esempio nella classe `Visualizer`, che viene utilizzata per la visualizzazione degli oggetti grafici nella GUI.

La classe è stata pensata come singleton in quanto il suo utilizzo non è quello di visualizzare le entità, ma di preparare, mediante calcoli, le entità per la visualizzazione.

Per questo non si ha la necessità di avere un `visualizer` per ogni view.

Composite

Composite è un design pattern strutturale che consiste del creare un oggetto che implementa un'interfaccia e di comporre l'oggetto con altre implementazioni dell'interfaccia implementata.

Questo permette di creare alberi in cui gli elementi finali (le foglie) sono implementazioni concrete dell'interfaccia e non sono composte nuovamente da altri oggetti che implementano la stessa interfaccia.

Questo pattern torna molto utile nel caso in cui si voglia trattare un gruppo di oggetti come se fossero l'istanza di un singolo oggetto in egual modo.

Questo lo si può vedere bene nel caso degli `Spawner` generici: abbiamo una moltitudine di spawner che operano allo stesso modo; raggruppandoli in un aggregatore che estende dalla stessa interfaccia, siamo in grado di richiamare il metodo *spawn* solo sull'aggregatore, il quale si occuperà di effettuare la medesima chiamata su tutti gli spawner contenuti, aggregando i risultati e ritornandoli in output.

Un altro utilizzo da notare è in `HitBox`: infatti per creare una hit box complessa è possibile creare un albero che usa `HitBoxIntersection` e `HitBoxUnion` (`HitBoxAggregation`), mentre le foglie sono le singole forme geometriche.

Decorator

Decorator è un design pattern strutturale che consiste del creare un oggetto che implementa un'interfaccia e di comporre l'oggetto con altre implementazioni dell'interfaccia implementata.

Si differenzia dal pattern Composite perchè oltre a svolgere il ruolo di aggregatore dei risultati permette anche di svolgere operazioni extra.

Nel nostro caso `Game` è Decorator per `World` e `Player`, infatti implementa entrambe le interfacce e permette di effettuare le operazioni di entrambi e della gestione dello spawner.

Bridge

Bridge è un design pattern strutturale che consiste nel dividere la logica di un oggetto in due dimensioni ortogonali.

Sposta il problema da gerarchia a Composizione.

Questo pattern è utilizzato nella gestione `Collisionable-HitBox`, dividendo in due dimensioni, forma geometrica e ruolo all'interno del gioco.

Template method

Il template method è un pattern comportamentale basato su classi, che permette di definire la struttura di un algoritmo lasciando alle sottoclassi il compito di implementarne alcuni passi come preferiscono.

In questo modo si può ridefinire e personalizzare parte del comportamento nelle varie sottoclassi senza dover riscrivere più volte il codice in comune.

Questo risulta estremamente adatto quando si vuole implementare la parte invariante di un algoritmo una volta sola e lasciare che le sottoclassi implementino il comportamento che può variare.

Strategy

L'obiettivo di questa architettura è isolare un algoritmo all'interno di un oggetto, in maniera tale da risultare utile in quelle situazioni dove sia necessario modificare dinamicamente gli algoritmi utilizzati da un'applicazione.

In questo modo, invece di implementare direttamente un singolo algoritmo, il codice riceve direttamente, in fase di esecuzione, quale algoritmo utilizzare in una famiglia.

Questo lo si può vedere nel plane e nello spawner, dove viene utilizzata una determinata strategia per lo spawning dei missili.

Observer

Observer è un design pattern di comportamento che consiste nel far sotto-scrivere (subscribe) uno o più oggetti ad un oggetto in modo che quest'ultimo notifichi i suoi cambiamenti.

Questo pattern è usato per la gestione della relazione grafica-controller in modo che ad ogni cambiamento (tramite eventi) viene notificato il cambiamento al controller.

6 Implementazione

In questa sezione è mostrata una panoramica sugli aspetti di maggior interesse e rilievo relativi all'implementazione del gioco.

Per un maggior dettaglio è possibile visionare la documentazione (ScalaDoc) generata.

6.1 Concetti relativi a programmazione funzionale

Il progetto è stato realizzato seguendo in pieno il paradigma funzionale che, grazie ad un linguaggio come Scala, ci ha permesso di mantenere compatto il codice e riutilizzabile, preferendo un approccio di delegazione/sostituibilità al posto di gerarchie ed ereditarietà. Quest'ultimo è stato utilizzato ove necessario (implementazione di trait).

Si sono poi alternate fasi di *refactoring* del codice per raffinarlo ed eliminare eventuali concetti/costrutti più legati al paradigma OOP ove necessario.

6.1.1 Trait e Mixins

Si è fatto ampio uso di trait per modellare interfacce comuni e utilizzabili per realizzare costrutti di matching (match pattern).

Abbiamo usato diffusamente i trait in accoppiata ai cosiddetti Companion Object, i quali oltre a fungere da contenitori di metodi o dati, si occupano di modellare le implementazioni della relativa interfaccia mediante Factory Method, ovvero le *funzioni apply*.

Un altro concetto largamente utilizzato sono i mix-in, ovvero classi o interfacce nelle quali uno o più metodi (o proprietà) non sono implementati, richiedendo che un'altra classe o interfaccia ne provveda un'implementazione.

Un chiaro esempio è rappresentato dalle interfacce utilizzate nei diversi moduli del CakePattern, impilate una dopo l'altra e non ereditate:

```
trait Interface extends Provider with Component
```

6.1.2 Immutabilità

Si è scelto di seguire una strada puramente immutabile: l'intero codice è stato realizzato secondo questa filosofia, cercando di eliminare qualsiasi possibile side effect e ricreando ogni singolo oggetto in caso di modifica dello "stato interno".

Questo ci ha permesso di minimizzare la possibilità di incontrare bug dovuti a side effect non preventivati e non voluti.

6.1.3 High order function

Una High Order Function è una qualsiasi funzione in grado di prendere in input altre funzioni e/o ritornare come output una funzione, o funzioni che possono essere inviate come argomento: questo meccanismo risulta essenziale in un paradigma funzionale, tramite il quale è possibile ad esempio implementare il pattern Strategy, passando come input la strategia logica da utilizzare.

Un esempio di High Order Function lo si può vedere nel modulo Game:

```
def updatePlayer(update: Player => Player): Game =  
  updatePlayer(update(player))
```

La funzione *updatePlayer* prende in ingresso una funzione che dato il Player corrente ne ritorna un'altro aggiornato, e di conseguenza il nuovo Game con il nuovo Player.

Un altro esempio lo si può vedere nella gestione della strategia di spawning: in particolare è stato creato un alias (Members) per una funzione generica che, senza parametri, ritorna un elemento del tipo specificato.

Questo è utile per definire delle funzioni Supplier che generano un nuovo elemento secondo una certa strategia.

```
type Spawnable[+A] = () => A
```

6.1.4 Self type

I Self Type permettono di definire una *trait* in modo tale che essa possa essere istanziata mediante mix-in di una classe o un'altra trait del tipo specificato: sostanzialmente siamo in grado di realizzare una forma di ereditarietà sfruttando delegazione/decoratori e i mix-in.

Un esempio di tale meccanismo lo vediamo nell'implementazione del missile con traiettoria a zig zag: esso risulta un mix-in da aggiungere alla creazione di un missile, in quanto esegue override di alcuni metodi aggiungendone di nuovi.

```
trait ZigZagMissile(step: Int = 5, positions: LazyList[Point2D], to: Point2D):  
  missile: MissileImpl =>  
  ...  
  def subDestinationReached: Boolean = position ~= destination  
  override def destinationReached: Boolean = position ~= to  
  ...  
  
//creazione di un missile a zigzag nemico  
new MissileImpl(...) with Scorable(1) with ZigZagMissile(step, list, to)
```

Il Self Type è stato sfruttato anche nell'implementazione del Cake Pattern, per indicare le dipendenze necessarie ad un determinato modulo.

Nel caso di *ControllerModule* sono necessarie le dipendenze dal modulo della View e dal rispettivo del Model.

```
type Requirements = ViewModule.Provider with ModelModule.Provider  
...  
trait Interface extends Provider with Component:  
  self: Requirements =>
```

Per quanto riguarda invece l'istanziatura dei vari componenti sfruttiamo la keyword *with* nel seguente modo, con tutte le dipendenze implicite, già gestite:

```
object MVC
  extends ModelModule.Interface
  with ViewModule.Interface
  with ControllerModule.Interface:

  override val model: ModelModule.Model = new ModelImpl()
  override val view: ViewModule.View = new ViewImpl()
  override val controller: ControllerModule.Controller = new ControllerImpl()
```

6.1.5 Alias

Al fine di generalizzare il codice e permettere una più facile gestione in caso di modifiche, alcuni aspetti del gioco sono gestiti tramite alias. Si può vedere il loro utilizzo nel codice riportato qui sotto.

```
/**
 * Alias for life measure
 */
type LifePoint = Int

/**
 * Alias for the distance between points in the hit box area
 */
type Distance = Double
```

Questo aumenta la flessibilità del codice senza introdurre complicazioni, in quanto il type system andrà automaticamente a sostituire l'alias con il relativo tipo durante la fase di type checking

6.1.6 For Comprehension

Al fine di rendere il codice meno imperativo, si è fatto uso della for-comprehension: un costrutto funzionale per operare sulle collezioni e basato sulle monadi. Oltre a rendere il codice più funzionale, la scelta dell'utilizzo della for-comprehension è supportata dall'incremento della leggibilità del codice. Nell'esempio sotto è riportato un caso di utilizzo, dove vengono utilizzati per la generazione delle città e delle torrette.

```
val cities =
  for y <- List.range(0, 2)
    x <- List.range(0, 3)
    yield City(Point2D(missileBattery_BaseSize + 2*
      turretSpacer +
        (cityBaseSize + citySpacer) * x +
        (3 * cityBaseSize + 2 * citySpacer + 2 *
          turretSpacer + missileBattery_BaseSize) * y,
        World.height - cityHeightSize))
val turrets =
  for x <- List.range(0, 3)
```

```

yield MissileBattery(Point2D(turretSpacer +
    (missileBattery_BaseSize + 2 * turretSpacer + 3 *
    cityBaseSize + 2 * citySpacer) * x,
    World.height - missileBatteryHeightSize))
Ground(cities, turrets)

```

Nell'esempio sopra, si possono visualizzare 2 utilizzi delle for comprehension:

- Nel caso della generazione delle torrette, si itera su una lista contenente valori da 0 a 2 (compresi), usati poi per la generazione delle torrette;
- Nel caso della generazione delle città, vengono valutate tutte le possibili combinazioni di y/x per la generazione dei punti. Più nello specifico, vengono controllati nella sequenza (0,0),(0,1),(0,2),(1,0)...

6.1.7 Currying

Il currying è un meccanismo di Scala che permette la valutazione di una funzione, che assume più parametri, come una lista di funzioni che accettano un argomento. Un esempio di utilizzo è riscontrabile nel codice realizzato per creare una LazyList di punti casuali per una traiettoria dinamica:

```

def apply[A](start: A)(next: (A) => A)(cond: (A) => Boolean)(mapping: (A) =>
A): LazyList[A] =
    LazyList.iterate(start)(next) filter (cond) map(mapping)

```

La generazione di una LazyList sfruttando la precedente definizione è codificata come segue:

```

def randomList(from: Point2D, to: Point2D, step: Int)(using Random):
    LazyList[Point2D] =
    val d = (to <-> from) / step
    val v = ((to <--> from).normalize) * d
    DirectionList(from) {
        _ --> (-v)
    } (_ != from) {
        _ --> (-v -|- Rand(Random))
    }.take(step)

```

6.1.8 Generici

I tipi generici sono tornati molto utili nella realizzazione dell'entità spawner: esso altro non è che un'entità che genera una tipologia di oggetti, in quantità dipendente da uno specifico intervallo di tempo.

La tipologia di oggetti generabili è stata modellata con un tipo generico con lower bound il trait Collisionable: questo perché si è deciso che uno spawner debba essere in grado di generare solo elementi abilitati alle collisioni.

```

trait GenericSpawner[A <: Collisionable](using Random) extends Timeable:

  def spawn(): (Set[A], GenericSpawner[A])

  override def timeElapsed(dt: DeltaTime): GenericSpawner[A]

```

6.1.9 Extensions

Durante i vari sprint sono stati ampiamente utilizzati gli *Extension Methods*. Questi hanno la peculiarità di poter aggiungere, anche solo localmente, metodi add-on su implementazioni e trait già costruite, permettendoci da un lato di essere maggiormente indipendenti nello sviluppo, dall'altro dandoci uno strumento per aggiungere metodi e funzionalità esclusivamente dove veramente utile.

Un esempio è il loro utilizzo nella realizzazione dei missili a zigzag, dove sono stati utilizzati per generare un vettore ortogonale direzionato a destra o a sinistra (o uno dei due casualmente) dato un vettore di partenza.

```

extension(v: Vector2D)
def --(d: Direction, magnitude: Int = defaultMagnitude): Vector2D = d match
  case Right => Vector2D(magnitude, Angle.Degree(v.direction.get.degree + 90))
  case Left => Vector2D(magnitude, Angle.Degree(v.direction.get.degree - 90))
  case Rand(rand) =>
    val value = rand.nextDouble()
    Vector2D(magnitude, Angle.Degree(v.direction.get.degree + (90 * value.mapToSign)))
  case _ => v

```

6.1.10 Pimp my Library

Il pattern funzionale *Pimp my Library* è usato per fornire metodi non implementati ad una classe/oggetto.

Nel progetto il pattern è stato utilizzato tramite l'uso di extension methods.

È usato ad esempio per ampliare i metodi di Collisionable (per utilizzare metodi di classi figlie), e per aggiungere metodi ai set di Collisionable e di Collision.

Un altro utilizzo da citare è nel caso dei Vector2D, a cui è stato aggiunto un metodo che restituisce un vettore ortogonale a quello di partenza.

6.2 Concetti di programmazione asincrona e reattiva

Per lo sviluppo del progetto si è fatto uso sia della programmazione reattiva (di tipo event-based) che di quella asincrona, scegliendo di sfruttare i metodi forniti dalla libreria monix.io.

Meccanismi di programmazione asincrona, come Task, sono stati utilizzati per evitare di bloccare il thread grafico di Swing.

Per questo motivo, il controller gestisce gli eventi tramite una sequenza di task, eseguiti

in modo asincrono rispetto alla grafica.

6.2.1 GameLoop

In particolare l'intero ciclo di game (game loop) è una sequenza di Task che vengono usati come input per l'iterazione successiva tramite il metodo scanEval.

```
val game = Game.initialGame
val controls: Update = Update.combine(
    UpdateTime(),
    UpdatePosition(),
    ActivateSpecialAbility(),
    CollisionsDetection(),
    LaunchNewMissile()
)
val init = Task((game, controls))
val events =
    Observable(time, ui.events).merge
```

```
events
    .scanEval(init) { case ((game, controls), event) => controls(event,
        game) }
    .doOnNext { case (game, _) => ui.render(game) }
    .takeWhile { case (game, _) => game.world.ground.stillAlive }
    .last
    .doOnNext { case (game, _) => ui.gameOver(game) }
    .completedL
```

Il primo frammento di codice rappresenta la sequenza ordinata degli Update da eseguire, mentre la seconda è il cuore del ciclo di gioco.

Infatti lo stream, viene eseguito in modo lazy e solo all'arrivo di un evento, viene valutato a partire dallo stato iniziale del game fino alla fine.

Dopo avere eseguito tutti gli Update viene fatto il render del game.

Tutto questo viene fatto finchè il ground ha ancora vita.

Una volta terminato bisogna mostrare i valori finali di score e tempo di sopravvivenza (gameOver).

6.2.2 View ed Event

Per l'implementazione dell'interfaccia grafica si è utilizzata la libreria Java Swing (per via dell'esperienza nell'utilizzo di essa e grazie alla possibilità di Scala di permettere l'utilizzo di codice Java all'interno del programma).

Visto l'utilizzo del pattern MVC, si è eseguita una separazione tra la parte di grafica e la logica del mondo, per la gestione delle posizioni.

Questo ha portato allo sviluppo della classe Visualizer e CollisionableVisualizer che sono utilizzati per la conversione tra mondo logico a mondo virtuale.

L'interfaccia grafica viene gestita dal GameController, che tramite l'utilizzo di appositi metodi, esegue gli update della grafica.

In particolare, la GUI presenta tre metodi asincroni per la gestione della schermata d'inizio, del render e della schermata del game over e un metodo che viene utilizzato per generare un'Observable di eventi, utile, come detto, ed eseguire l'azione di lanciare un missile dopo un click e far partire il gioco.

In particolare:

- La visualizzazione del game initial visualizza a schermo la scritta "Missile command" e un bottone per giocare;
- Il render viene eseguito da un task asincrono. Quando il metodo viene usato, viene utilizzato WorldPane per aggiornare il panel di gioco con tutte le informazioni aggiornate presenti nel World.

Per fare ciò, ottiene tutte le informazioni delle entità del gioco (missili, città, aerei...) e grazie alle classi visualizer, ne scala le dimensioni rispetto alla grafica;

```
override def render(game: Game): Task[Unit] = Task {  
  SwingUtilities.invokeLater { () =>  
    clearPanel()  
    frame.getContentPane.add(WorldPane(game, width, height))  
    frame.getContentPane.repaint()  
  }  
}
```

- La visualizzazione del game over visualizza a schermo il tempo durante il quale il giocatore è sopravvissuto, lo score finale e un bottone per riniziare la partita;
- L'evento che modella il lancio del missile dalle basi incapsula la posizione a cui si vuole lanciare il missile.

```
private val gameEvent: Observable[Event] =  
  frame  
    .getContentPane  
    .mouseObservable()  
    .map(p =>  
      Event  
        .LaunchMissileTo(  
          Point2D(  
            (p.x * World.width.toDouble) /  
              ViewConstants.GUI_width.toDouble,  
            (p.y * World.height.toDouble) /  
              ViewConstants.GUI_height.toDouble  
          )  
        )  
    )  
  )
```

6.3 Testing

Per testare le funzionalità principali del programma, si è deciso di utilizzare ScalaTest.

Grazie all'utilizzo di ScalaTest, è possibile associare ad ogni test una breve descrizione, in modo tale da avere una maggior comprensione di cosa il test deve fare.

La scrittura dei test è avvenuta in simultanea con la scrittura del codice, in modo tale da verificare che qualsiasi funzione implementata abbia lo stesso funzionamento di quello desiderato e non presenti bug.

Successivamente alla fase di testing, come avviene per il paradigma di sviluppo TDD (Test driven development), si è eseguito un refactor del codice per migliorarne la qualità e leggibilità.

Questo poi seguito successivamente dall'esecuzione dei test per verificarne nuovamente il corretto funzionamento nelle varie casistiche di utilizzo.

Per lo sviluppo dell'applicazione si è deciso di adottare la pratica di Continuous Integration, testando solo su macchina Linux, principalmente perché la maggior parte del sistema testato è model e quindi platform independent.

Inoltre, grazie all'utilizzo di GitHub Pages, è stato possibile visualizzare la coverage dei test sul codice effettuato tramite un sito realizzato dal comando apposito in CI, per visualizzare al meglio quanto di un determinato elemento è stato testato.

Qui sotto sono riportati due esempi di test effettuati sui missili.

```
it("should calculate its own direction based on start position and final position") {
    val missile = TestMissile()
    assert(missile.direction == (finalPosition <-->
        startPosition).normalize)
}
it("should decrease its lifepoints when damaged") {
    val missile = TestMissile()
    val damagedMissile = missile.takeDamage(damage)
    assert(damagedMissile.currentLife == initialLife - damage)
}
```

Una volta rilasciata la prima versione ogni volta che viene rilevato un nuovo bug logico, data la divisione tra model, view e controller, viene ricreato il bug in un test, usando il mondo logico, in modo da risolverlo e da aver sicurezza che futuri cambiamenti non re-introducano il problema.

6.3.1 Property-based testing

Per verificare le funzioni matematiche basilari del progetto (angoli e vettori) si effettuano dei test di proprietà al fine di assicurarsi che le proprietà matematiche siano rispettate.

Esempio nel caso degli angoli:

```
private val degreeStraightAngle = 180.0
property("Angle in degree should be in interval [-180, 180] (Degree)") =
    forAll(angleDegreeGen) { angle =>
        angle.degree > -degreeStraightAngle && angle.degree <= degreeStraightAngle
```

```
}
```

6.3.2 Behaviour-Driven Development (BDD)

Per gestire i test della simulazione virtuale vengono effettuati dei test in modo che anche i committenti possano riscontrarvi i requisiti.

Per questo motivo viene utilizzata una classe che implementa AnyFeatureSpec e Given-WhenThen.

Esempio semplificato preso dalla simulazione del gioco:

```
Feature("Game going") {
  Scenario("In the game there are one friendly missile, one friendly
    explosion and one enemy missile that collides") {
    ...
    Given("An initial game with some missile")
    val game =
      Game
        .initialGame
        .updateWorld(
          _.addCollisionables(
            Set(
              enemyMissile,
              friendlyMissile,
              friendlyExplosion
            )
          )
        )

    When("The game executes collisions")
    val (updatedGame, collisions) =
      game
        .checkCollisions()
    val gameScore = (updatedGame, collisions).updateScore()

    Then("The collisionables should be 3 explosion")
    assert(updatedGame.world.collisionables.count(_.isInstanceOf[Explosion])
      == 3)
    ...
  }
}
```

6.3.3 Coverage

Come accennato in precedenza, tramite GitHub Action e GitHub Pages è stato possibile anche verificare la coverage che viene pubblicata ad ogni esecuzione della CI sul branch deployCoverage, e su un sito apposito.

La code coverage fa riferimento, sostanzialmente, alla quantità di istruzioni di codice che vengono eseguite durante l'esecuzione dei tests.

Tuttavia, ottenere una coverage del 100% non significa che il testing effettuato riesca

a ricoprire tutti gli scenari: infatti, l'obiettivo che ci si è dati non è stato quello di raggiungere il 100% della copertura ma di testare funzioni mirate. In particolare, per poter ottenere i risultati relativi alla coverage, si è fatto utilizzo del tool Scoverage.



Scoverage generated at Thu Dec 01 20:38:43 UTC 2022					
Lines of code:	4206	Files:	70	Classes:	114
Methods:	382	Lines per file:	60.09	Packages:	16
Classes per package:	7.12	Methods per class:	3.35	Total statements:	1705
Invoked statements:	1336	Total branches:	18	Invoked branches:	16
Ignored statements:	0				
Statement coverage:	78.36 %			Branch coverage:	88.89 %
					

Figure 6.1: Statistiche su coverage, realizzata con Scoverage

All packages	78.36%
elements2d	100.00%
model	100.00%
utilities	100.00%
zigzag	98.58%
hitbox	98.33%
collisions	96.84%
explosion	96.30%
spawner	93.14%
ground	83.69%
missile	81.91%
vehicle	73.76%
view	30.00%
components	0.00%
controller	0.00%
gui	0.00%
update	0.00%

Figure 6.2: Coverage del progetto nei vari package, realizzata con Scoverage

Come si può vedere dalle figure (6.2, 6.1) la coverage finale ottenuta è del 78% su un totale di 255 test effettuati.

Gli elementi per cui si ha una coverage più elevata sono quelli che fanno riferimento al Model dell'applicazione, come già detto, mentre quelli per cui si ha una coverage è

più bassa fanno riferimento agli elementi della View e Controller, che sono stati testati manualmente data la natura del risultato che si voleva ottenere (come avere un sistema fluido) per cui è difficile definire dei test automatici, essendo un concetto soggettivo.

6.4 Presentazione grafica

Il framework di rappresentazione grafica scelto per visualizzare l'andamento del gioco e gestire le interazioni con l'utente è *Java Swing*, il quale offre le funzionalità necessarie e un giusto compromesso di facilità d'uso e aspetto grafico.

Prima di tutto abbiamo definito un `JFrame` all'interno della classe `GUI`, la quale, estendendo da `View`, ha il compito di implementare la gestione degli eventi di click e il *render*, di cui è stata mostrata l'implementazione in 6.2.2.

```
private class WorldPane(val game: Game, width: Int, height: Int) extends
    JPanel:
    ...

    override def paintComponent(graphics: Graphics): Unit =
        ...
        Visualizer.printGround(game.world.ground).map(
            imageData => graphics.drawImage(imageData._1, imageData._2.x,
                imageData._2.y,
                imageData._3,
                imageData._4, this)
        )

        CollisionableVisualizer.printElements(game.world.collisionables.toSet)
        foreach { i =>
            g2d.translate(i.position.x, i.position.y)
            g2d.rotate(i.angle.radian - Angle.Degree(90).radian)

            graphics.drawImage(i.image, 0 - (i.baseWidth / 2), 0 -
                (i.baseHeight / 2), i.baseWidth, i.baseHeight, null)

            g2d.rotate(-1 * (i.angle.radian - Angle.Degree(90).radian))
            g2d.translate(-i.position.x, -i.position.y)
        }
```

Il `WorldPane` rappresenta il pannello della partita in corso, in cui viene mostrato il timer e il punteggio attuali e tutti gli elementi logici presenti in gioco.

In particolare, nella porzione di codice mostrata sopra viene fatto il render di tutte le città/torrette e tutti gli elementi `Collisionable` attualmente presenti.

Il modulo `Visualizer` e `CollisionableVisualizer` si occupano di creare gli elementi grafici partendo da quelli logici presenti nel model nel momento attuale.

6.5 Suddivisione del lavoro

La suddivisione del lavoro, come accennato, non è stato definito a priori nelle prime fasi del progetto, ma sono stati proposti ad ogni sprint una serie di task che ciascun mem-

bro poteva assegnarsi, anche in relazione a task precedentemente svolti, per similarità di lavoro o determinati legami di interazione ad esempio.

In questa sezione ciascun membro del team spiegherà in dettaglio il lavoro svolto singolarmente e/o in collaborazione e con chi.

Andrea Brighi

Prima del primo sprint mi sono occupato della creazione del repository, della creazione della base del progetto e ho impostato la CI tramite le GitHub Actions.

Una volta completate le operazioni iniziali ho creato le basi matematiche del progetto creando i file:

- **Angle**: rappresentazione degli angoli in gradi e radianti
- **Vector2D**: rappresentazione di un vettore a due dimensioni
- **Point2D**: rappresentazione di un punto a due dimensioni
- **MathUtilies**: contiene extension method per la libreria *scalactic* in modo da fornire i metodi `>==` e `<==` che permettono l'uso di una tolleranza nelle disuguaglianze e permette di mantenere la proprietà di convergenza ($a \leq b$ e $b \leq a$ allora $a = b$)
- **test e property test** per verificare la correttezza del lavoro svolto.

Nel primo sprint mi sono occupato della creazione delle gerarchie delle HitBox e dei Collisionable che sono già stati analizzati nel dettaglio in precedenza.

Il risultato di questo sviluppo sono quindi:

- Il package **hitbox**, che contiene le varie forme che le hitbox hanno nel gioco, più due che permettono di creare forme derivate;
- Il trait **Collisionable** da cui derivano tutti gli oggetti che hanno una presenza nel gioco, nonché le due trait derivate **Damageable** e **Damager**;
- La funzione per il calcolo delle collisioni, l'implementazione fatta nel primo sprint delle collisioni è diversa da quella finale.
Infatti quella realizzata a questo punto esegue sia la rilevazione che l'applicazione del danno.
Anche la struttura dati è diversa, infatti si usa una mappa con chiave un collisionable e come valore una lista di elementi con cui collide;

Anche in questo caso i test vengono realizzati contemporaneamente al codice.

Nel secondo sprint ho realizzato:

- **World**, che in questa fase il **World** è una versione semplificata di quello finale.
Rappresenta l'insieme degli elementi presenti nel gioco.
Racchiude, anche i concetti che, più avanti, diventeranno Game e Player;

- **Scorable** e la logica per modificare il punteggio.
La funzione per il calcolo del punteggio, implementata a questo punto è diversa da quella finale.

Nel terzo sprint ho realizzato il **GameController** (già analizzato in precedenza) e gli eventi da esso osservati.

Anche in questo caso la versione è più primitiva di quella finale, questo dovuto ad un'evoluzione successiva delle necessità.

In questo sprint sono anche stati creati gli eventi, i vari Update e l'observer che controlla il tempo.

Nel quarto sprint ho effettuato test e risolto bug rilevati dopo la pubblicazione della prima versione.

Inoltre è stato effettuato un refactory per le collisioni (**Collision** e **Collisions**), che viene creata come vera entità, (usando alias) e, di conseguenza anche della loro implementazione nella rilevazione è cambiata, così come il calcolo del punteggio.

Nel Quinto sprint ho raffinato il concetto di **World** e introdotto quelli di **Game** e **Player** (anche in questo caso visti in precedenza), nonché migliorato il controller con le funzioni per avviare e gestire la pagina iniziale.

Matteo Lazzari

Nel primo sprint mi sono occupato della realizzazione dei componenti principali del ground: torrette e città.

- **Città:** Implementata mediante l'utilizzo di una Case Class ed estende il trait **Damageable** sviluppata dagli altri membri del gruppo.
Questo ha permesso alle città la possibilità di essere danneggiate da missili nemici;
- **Torrette:** Sono state implementate come le città, mediante l'utilizzo di una Case Class.
Essendo che hanno bisogno di un riferimento temporale, è stato necessario estendere **Timeable** per rendere possibile l'utilizzo del tempo virtuale.
Inoltre, essendo un componente danneggiabile, è stata estesa anche la classe **Damageable**, per permettere alle torrette di essere danneggiate e distrutte da missili nemici.

Successivamente, grazie allo sviluppo dei componenti sopra elencati, durante il secondo sprint, ho sviluppato tutta la parte che riguarda il ground e le funzioni necessarie al controller per gestire correttamente tutti gli aspetti delle strutture.

In particolare, tra i metodi che sono stati sviluppati per le esigenze del controller, troviamo:

- **shootMissile:** Il metodo è chiamato quando il controller ha la necessità di sparare un missile verso un dato punto della mappa.
- **isGameEnded:** Il metodo è stato progettato per quando il GameLoop ha la necessità di verificare che il giocatore sia ancora vivo

- **dealDamage:** È un metodo di utility che è stato progettato per soddisfare le esigenze dei Collisionable nell'applicare i danni a specifiche strutture. Esistono 3 varianti di questo metodo:
 - Singolo danno a singola strutture;
 - Singolo danno a multiple strutture;
 - Multipli danni a multiple strutture.

Al terzo sprint mi sono occupato, insieme a Daniele Di Lillo, della parte di interfaccia del gioco, sviluppando la GUI e le classi che rendono possibile la visualizzazione delle entità all'interno del frame (CollisionableVisualizer e Visualizer).

Grazie a queste modifiche e al lavoro di Andrea Brighi sul GameController, siamo stati in grado di produrre una versione 1.0.0 all'inizio del 4° sprint.

Tutto il 4° sprint è stato dedicato al risolvere bug e sistemare problemi vari nelle iterazioni.

Una volta terminato lo sviluppo della versione 1.0.0, con la risoluzione dei bug identificati, sono passato allo svolgimento dei requisiti opzionali.

In particolare, allo sviluppo degli aerei e dei satelliti.

Questi hanno richiesto un affinamento nel codice dello spawner, per permettere di generare anche aerei/satelliti, oltre che a missili e la possibilità di inserire una strategia di spawn all'interno dello spawner.

Grazie a queste modifiche, è stato possibile procedere con lo sviluppo.

- **Aerei:** Nel caso degli aerei, la classe è stata gestita come una factory che nasconde l'implementazione: è presente una interfaccia Plane, una case class (che estende l'interfaccia Plane) e una factory, che viene utilizzata per la creazione degli oggetti. Per ottenere il comportamento desiderato, l'aereo deve estendere diverse classi, quali:
 - Damageable: L'aereo può essere distrutto dai missili del giocatore;
 - Moveable: L'aereo deve avere la possibilità di muoversi su mappa;
 - GenericSpawner: l'aereo deve essere uno spawner di missili.
- **Satellite:** Allo stesso modo dell'aereo, è avvenuta l'implementazione dei satelliti, con l'unica differenza che i satelliti non hanno l'estensione della classe Moveable, in quanto sono oggetti statici.

Infine, per quanto riguarda la parte di grafica, durante l'arco di tutti gli sprint dal 2° in poi, mi sono occupato con Daniele Di Lillo di rifinire la grafica e sviluppare insieme la parte della visualizzazione degli oggetti, grazie all'utilizzo delle classi CollisionableVisualizer e del Visualizer.

Insieme, queste due classi implementano metodi per le conversioni da mondo logico a mondo grafico, insieme ai metodi utilizzati dal WorldPane per ottenere le immagini (già pronte) da visualizzare sul mondo di gioco, ogni tick.

Daniele Di Lillo

Nel primo sprint mi sono occupato della realizzazione di due interfacce fondamentali per l'intero gioco:

- **Timeable:** interfaccia che rappresenta la concezione del tempo virtuale che trascorre per chi la estende.
Essa fornisce un metodo *timeElapsed*, il quale preso un valore di tipo *DeltaTime* restituisce l'oggetto stesso con il tempo interno virtuale aggiornato.
L'idea è quella di rendere il trascorrere del tempo come un aspetto puramente logico, senza il bisogno di sfruttare meccanismi di timer o wait
- **Moveable:** questa interfaccia modella una qualsiasi entità in grado di muoversi nello spazio del gioco.
Estende chiaramente da *Timeable*, in quanto un'entità si sposta man mano che il tempo trascorre

Sfruttando l'interfaccia *Timeable* ho poi realizzato un modulo *Timer*, ovvero un oggetto in grado di mantenere una rappresentazione del tempo trascorso da inizio partita.

Mi sono inoltre occupato della modellazione del missile lineare, il quale, oltre ad estendere da *Moveable* estende anche da *Damageable*, interfaccia realizzata dal collega Andrea Brighi.

Il missile lineare è stato realizzato sulle seguenti caratteristiche:

- Mantiene una rappresentazione del punto di partenza e uno di destinazione, una velocità, una quantità di danno, una quantità di vita, una affiliazione
- Un metodo per generare un'esplosione qualora venisse distrutto
- Una strategia di movimento

Inoltre sfrutta l'interfaccia *MissileDamageable* come mixin, in quanto offre una implementazione di *Damageable*.

Questa strategia è stata presa per non legare il model del missile alla logica delle hitbox, mantenendo i due concetti indipendenti: la hitbox, in questo caso rettangolare, viene implementata dal trait usato come mixin.

Successivamente ho realizzato il model dell'esplosione: esso, a differenza del missile, estende da *Damager*, realizzata da Andrea Brighi, in quanto preposto a infliggere danno e non subirlo.

Un'esplosione è caratterizzata da un tempo di persistenza, un raggio d'azione e l'affiliazione, data dal missile parent che la genera.

Un altro elemento fondamentale del gioco che ho realizzato è rappresentato dallo *Spawner* generico, realizzato nel secondo sprint

. Oltre ai moduli già mostrati in 5.4.6, troviamo all'interno del package *spawner*:

- *SpecificSpawners*: modulo che racchiude una serie di *factory methods* per diversi *Spawnable*, uno per ogni tipo necessario

- `PimpingByRandom` e `PimpingByDeltaTime`: questo modulo contiene un'extension method per il tipo `Random`, per permettere la generazione di punti casuali. Stessa cosa per il tipo `DeltaTime`, aggiungendo un metodo per diminuire un valore secondo una funzione iperbolica e un mapper condizionato

Sempre nel secondo sprint ho realizzato utilities grafiche e implementato la GUI in collaborazione con gli altri membri del team, includendo la rappresentazione del missile grafico. Nel terzo sprint ho esteso il `Generic Spawner` e l'esplosione con l'interfaccia `Timeable`, realizzato il modulo di gestione eventi della GUI e contribuito nella realizzazione del campo grafico.

Successivamente, negli ultimi sprint (4-5) ho contribuito a risolvere una serie di bug riscontrati e realizzato i missili con traiettoria alternata, o a "zig zag", come requisito opzionale.

Questi sono stati realizzati partendo dalla base del missile lineare sfruttando il *self type*, eseguendo override solo dei metodi di interesse e aggiungendone di nuovi.

Nel package *zigzag* troviamo:

- `DirectionList`: modulo che contiene factory methods per la generazione di un percorso alternato a "zig zag" per i missili, sfruttando le `LazyList`. Quest'ultima è stata scelta per evitare di valutare tutti i singoli punti in caso di un esplosione anticipata.
In particolare il metodo *randomList* permette di generare una lista di punti traslati in modo casuale a destra o sinistra rispetto al vettore direzione, mentre l'*apply* genera una lista di punti traslati in modo alternato a destra e sinistra
- `Pimping libraries`: extension methods raggruppate per tipologia (`Point2D`, `Double`, `LazyList`, `Vector2D`)
- `ZigZagMissile`: modulo che definisce le factory methods, l'interfaccia e l'implementazione di un missile a zig zag.
Esso sfrutta una `LazyList` di punti da usare come *sub-destination*, cambiando rotta ad ogni destinazione intermedia raggiunta

Ho inoltre aggiunto l'aumento di difficoltà all'interno del `Generic Spawner` (decrementando l'intervallo di spawning secondo una strategia sigmoideale).

Per quanto riguarda la View ho realizzato il gestore di eventi di click, definito nel modulo *MouseHandler* all'interno del package *view.gui*: esso rappresenta un'extension method di un `Component` di `JavaSwing`, con un nuovo metodo che ritorna un `Observable[Event]`. Quest'ultimo, in combinazione ad un `MouseListener`, genera un nuovo evento mappando le coordinate del click in un `Point2D`, e successivamente in un evento di tipo *LaunchMissileTo*.

Sempre riguardando la View ho implementato, in collaborazione con Matteo Lazzari, il modulo *CollisionableVisualizer*, il quale fornisce un metodo per convertire un elemento `Collisionable` in elementi grafici da mostrare graficamente e il pannello `WorldPane`, ovvero il campo da gioco grafico.

Infine mi sono occupato di gestire le dipendenze mediante `CakePattern`, realizzando i seguenti moduli:

- `ModelModule`: modulo che incapsula le dipendenze riguardo al model
- `ViewModule`: modulo che incapsula le dipendenze riguardo alla view
- `ControllerModule`: modulo che incapsula le dipendenze riguardo al controller
- `CakePattern`: modulo che istanzia i vari componenti dei singoli moduli

Per quanto riguarda i test ho realizzato i seguenti:

- `ExplosionTest.scala`
- `DirectionListTest.scala`
- `ZigZagMissileTest.scala`
- `GenericSpawnerTest.scala`
- `SpawnerAggregatorTest.scala`
- `SpecificSpawnersTest.scala`
- `MissileTest.scala`
- `CollisionableVisualizerTest.scala`

7 Retrospettiva

Il percorso di sviluppo, basato su SCRUM, ci ha portato a realizzare una serie di *product* al termine di ogni singolo sprint.

La durata di questi ultimi non è stata fissata a priori a inizio progetto, ma dimensionata prima dell'apertura di ogni sprint in base alla quantità di Task da eseguire.

Di seguito è possibile avere una chiara idea dei vari sprint effettuati e dei vari task svolti.

7.1 Sprint e Backlog

Pre Sprint - 28/09/2022

Gli obiettivi posti nello sprint 0 consistono nella realizzazione degli aspetti chiave, ovvero punti e vettori, parti essenziali del progetto che fanno da fondamento a tutti i calcoli logici e di posizionamento degli oggetti. I *product backlog* di questo sprint sono i seguenti:

- Point2D;
- Vector2D;
- Angle.

Questi due elementi forniscono la base per la realizzazione dei successivi prodotti.

Inoltre sono state definiti i requisiti e l'architettura del progetto, nonché la creazione del repository e la definizione delle operazioni dei CI.

Sprint 1 - 03/10/2022

Questo primo sprint ha come obiettivo quello di realizzare gli elementi fondamentali del gioco, con la finalità di realizzare il model del dominio.

Si sono utilizzati i risultati ottenuti dal *Pre Sprint* per implementare aspetti come le posizioni delle strutture e le direzioni dei missili. I *product backlog* di questo sprint sono:

- Missili ed esplosioni;
- Città e batterie anti-missilistiche;
- Collisioni.

Alla fine dello sprint è stato quindi possibile avere l'implementazione dei principali componenti del gioco.

Sprint 2 - 10/10/2022

Nel secondo sprint si è posto come obiettivo quello di realizzare la rappresentazione grafica degli elementi di model realizzati durante il precedente sprint.

In particolare i *product backlog* riscontrabili in questo sprint sono:

- Missile Spawner;
- Player Ground;
- Utility GUI;
- Missile grafico;
- World;
- Score;
- Batteria missilistica e città grafica.

Dopo questo sprint siamo arrivati ad un punto in cui abbiamo sia la rappresentazione logica sia quella grafica degli elementi di gioco, ponendo quindi la base per la realizzazione del GameController e la mappa grafica.

Sprint 3 - 18/10/2022

Durante questo sprint abbiamo implementato la parte relativa al GameController e alla Mappa grafica, gestendo il render degli elementi e il passaggio del tempo, compresi gli update e le interazioni tra i vari elementi.

- Esplosioni con interfaccia Timeable;
- Spawner con interfaccia Timeable;
- Batteria missilistica con interfaccia Timeable;
- Gestione eventi GUI;
- Mappa grafica;
- GameController.

Inoltre, lo sprint è stato esteso di 2 settimane per motivi interni al gruppo e per realizzare una versione iniziale funzionante.

Sprint 4 - 09/11/2022

In questo sprint, essendo arrivati alla versione 1.0.0, abbiamo analizzato il progetto e proposto una serie di bug da risolvere come *products backlog*. In particolare:

- Missili che si muovono in modo diverso da quello atteso;
- Spawn dei missili laterale e non centrale dalle torrette;
- Esplosioni alleate sembrano non avere effetto sui missili nemici;

- Risoluzione di bug minori;
- Risoluzione problemi legati alla generazione del jar.

Durante questa fase siamo riusciti a individuare una serie di soluzioni che hanno portato ad un codice perfettamente funzionante.

Sprint 5 - 15/11/2022

Il quinto sprint si è posto come obiettivo quello di migliorare le prestazioni del gioco e di aumentare le funzionalità del gioco.

Vengono infatti aggiunti come svolti i seguenti task:

- Implementazione del livello di difficoltà crescente nello spawner;
- Aereo e satellite nemico;
- Cake Pattern.
- Sistemati problemi relativi alle trasformazioni grafiche tra mondo logico e grafico;
- Miglioramento gestione delle collisioni;
- Miglioramento dell'asincronismo;
- Missili a zigzag.

7.2 Commenti finali

Come anticipato, la metodologia *SCRUM-inspired* ci ha permesso di seguire un processo di sviluppo software autonomo, consentendo a ogni membro di lavorare e ragionare sui propri task assegnati in maniera indipendente e rimanere sempre in contatto con il resto del gruppo, tramite riunioni e utilizzando strumenti quali Jira.

Tutto ciò realizzando una sorta di collaborazione continua all'interno del team, che ha favorito una grande crescita personale e professionale sulla gestione di un progetto software all'interno di un gruppo di persone, con attitudini ed esperienze diverse.

8 Conclusioni

In conclusione, ci riteniamo molto soddisfatti del lavoro svolto, perfettamente in linea con gli obiettivi posti.

Siamo riusciti a coordinarci molto bene nonostante una prima fase di incertezza sul processo di sviluppo da adottare: la metodologia SCRUM based, a cui abbiamo fatto riferimento durante l'intera realizzazione, ha facilitato l'organizzazione e suddivisione dei compiti, nonché l'interazione e coordinazione nei vari sprint.

Uno strumento risultato molto efficace da evidenziare è Jira, che in modo semplice ed intuitivo ci ha permesso di avere una chiara idea sullo stato di avanzamento in ogni fase del progetto, del carico di lavoro da suddividere e ci ha permesso di comunicare formalmente (anche con allegati) la presenza di eventuali bug riscontrati, ha inoltre fornito anche un supporto al versioning.

L'applicazione del Test Driven Development per lo sviluppo dei componenti fondamentali del progetto, ha favorito lo sviluppo di un maggior numero di test, che ha permesso una più rapida individuazione di problematiche e/o malfunzionamenti nel codice scritto.

Un altro aspetto fondamentale è risultata la Continuous Integration, grazie alla quale ad ogni aggiornamento avevamo una chiara idea sullo stato dei test.

Ci riteniamo molto soddisfatti inoltre per quanto riguarda l'utilizzo del paradigma funzionale e del linguaggio Scala, con una valutazione totalmente positiva.

Guida utente

Quando viene avviato il file jar dell'applicazione, verrà aperta una window iniziale dove sarà presente il tasto per avviare la partita (Gioca).

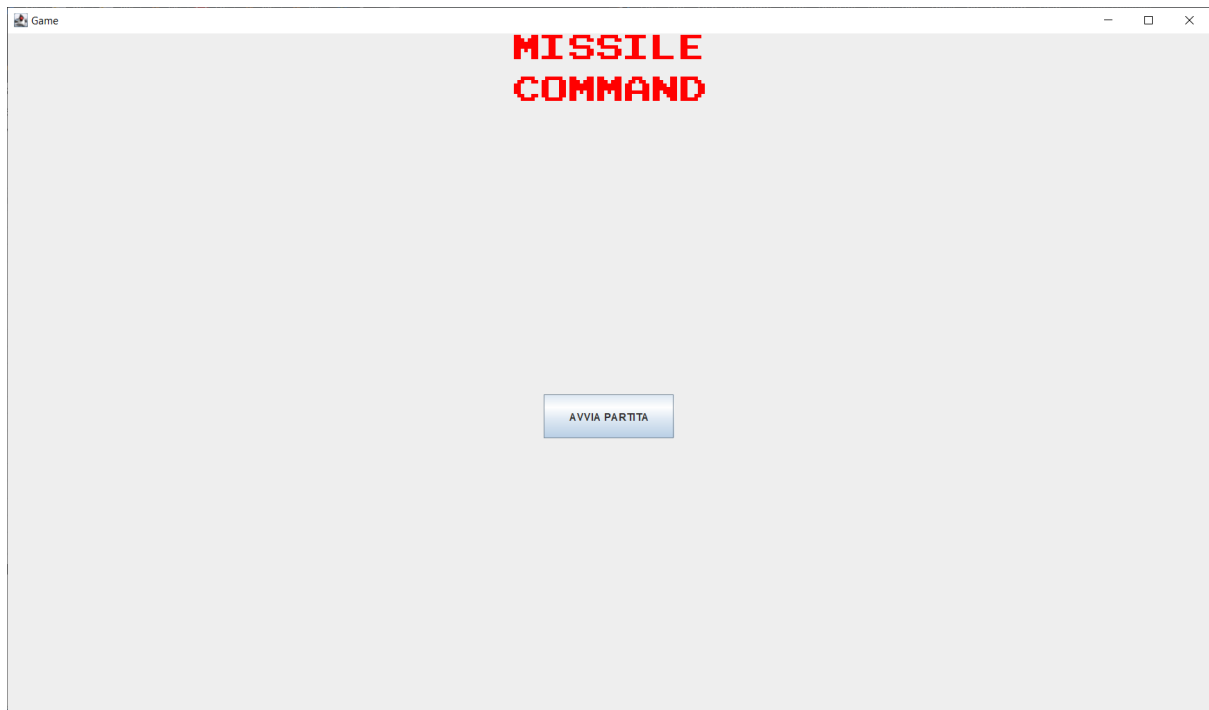


Figure 8.1: Mockup della pagina iniziale

Premendo il tasto Gioca, verrà aperta la window della partita.

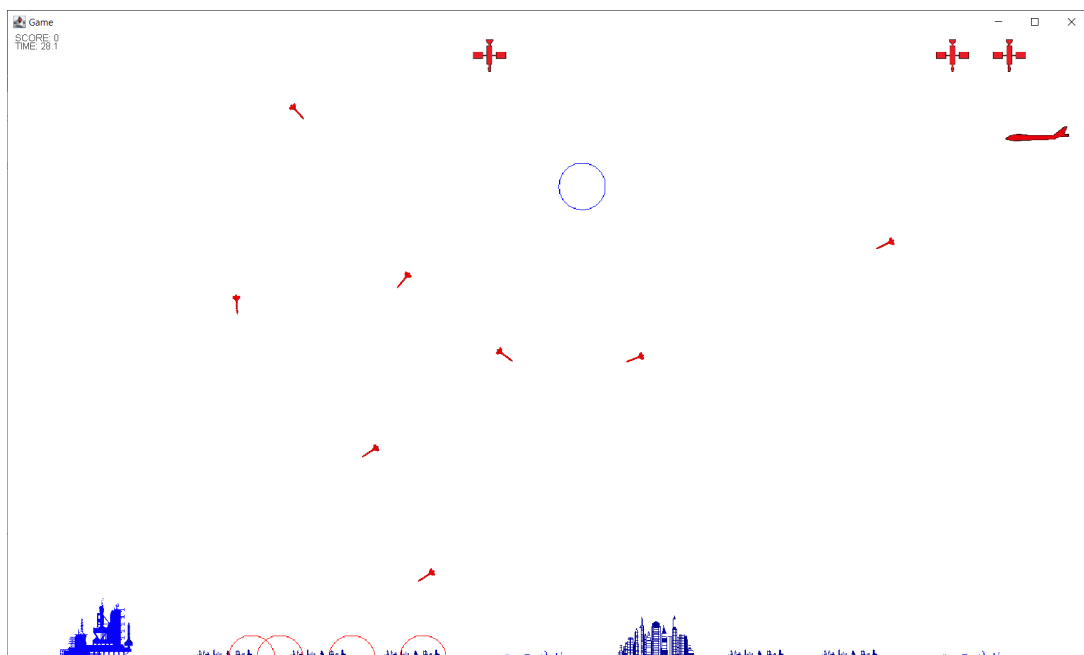


Figure 8.2: Screenshot di una partita in corso

Tramite questa view, si notino 2 cose importanti:

1. All'inizio della partita, tutte le torrette sono scariche. Impiegheranno 3 secondi (tempo di ricarica) per essere pronte a sparare;
2. Per visualizzare quali torrette sono cariche e quali no, basta controllare se sulla pedana di lancio sono presenti i missili.

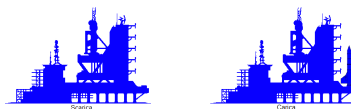


Figure 8.3: Differenza tra le basi.

Poco dopo l'inizio della partita, inizieranno a generarsi missili che piovono dal cielo (come nemico iniziale), seguito poi (dopo archi di tempo prestabiliti) aerei, satelliti e missili zig zag.

Durante la partita, in alto a sinistra saranno visualizzati score e tempo passato dall'inizio della partita.

Con lo scorrere del tempo, le batterie missilistiche si ricaricheranno (necessitano di 3 secondi) e cliccando con il tasto sinistro sullo schermo, la batteria missilistica più vicina al punto selezionato sparerà un missile. Il missile sarà sempre sparato dalla batteria missilistica carica più vicina. Se una torretta non è carica, non sparerà.

I missili alleati devono intercettare le entità nemiche. La cosa principale è che il missile non esplode al contatto con le entità nemiche, ma esploderà solo ed esclusivamente quando arriverà alla destinazione finale di dove si è cliccato col mouse.

Quindi, anche se il missile passerà sopra entità nemiche, non effettuerà nessuna detonazione.

Una volta che i missili arrivano a destinazione (sia alleati, che nemici), esploderanno, distruggendo tutte le entità dell'altra fazione che entrano in contatto con l'esplosione. Lo scopo del giocatore quindi, non è quello di colpire direttamente i nemici, ma deve anticiparne il movimento, in quanto sarà l'esplosione a distruggere l'entità.

Quando un nemico colpirà una esplosione alleata, esso esploderà. Questo vale per tutti i tipi di nemici, sia missili e che per tutti i veicoli.

Allo stesso modo, se un missile nemico colpirà il terreno, esso genererà un'area di esplosione che distruggerà tutte le strutture nella sua area.

Se una torretta viene colpita, verrà distrutta e diventerà inutilizzabile.

La partita terminerà quando tutte le città saranno state distrutte. Non importa quindi se tutte le torrette sono ancora disponibili oppure se sono state tutte distrutte e restano solo le città.

Al termine della partita, il gioco cambierà window in una schermata dove mostra lo score del giocatore e il tempo che è riuscito a sopravvivere. Inoltre, sarà presente anche un bottone "Rigioca", che permette al giocatore di riavviare la partita.

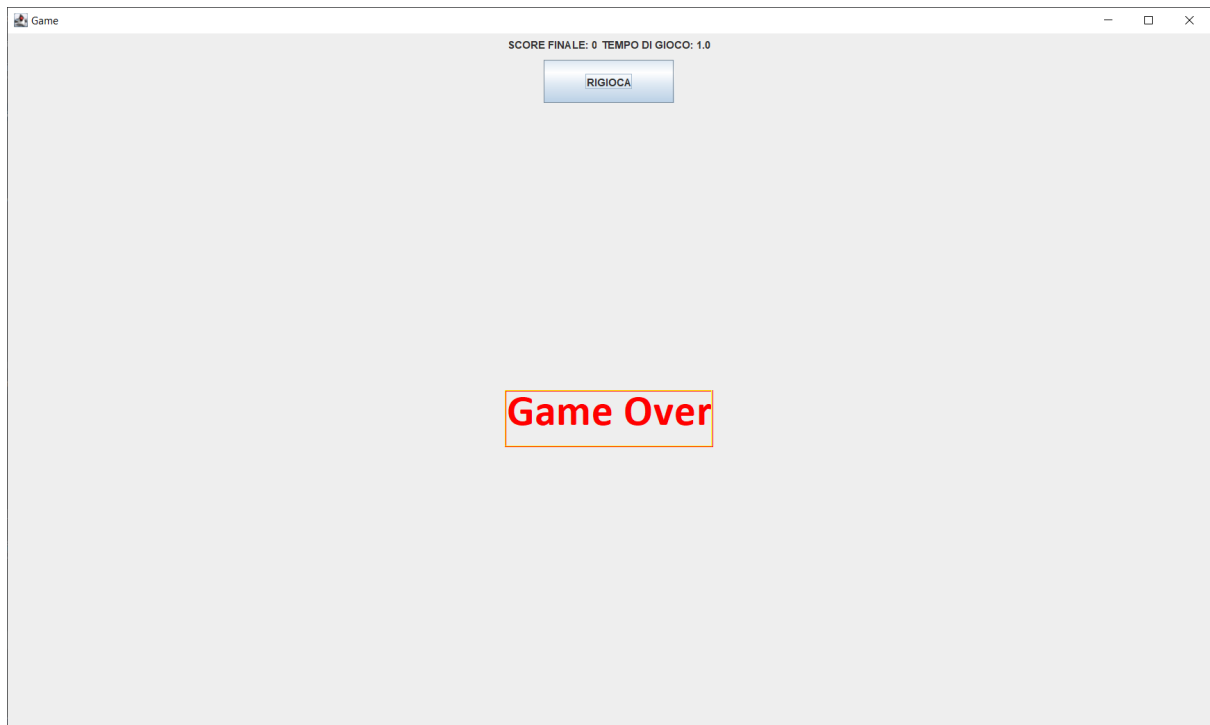


Figure 8.4: Pagina di game over con i risultati della partita