Relazione per progetto OOP
"Scramble 1981"

Lasse Werpers, Federico Capponi,
Francesco Teo Calzolari, Claudia Falcone

16 settembre 2024

Indice

1 Analisi **3**
- 1.1 Descrizione . . . 3
- 1.2 Requisiti . . . 4
  - 1.2.1 Casistica . . . 4
  - 1.2.2 Modello del dominio . . . 6

2 Design **8**
- 2.1 Architettura . . . 9
- 2.2 Design Dettagliato . . . 11
  - 2.2.1 Lasse Werpers: nemici e collisioni tramite hitBox . . . 11
  - 2.2.2 Federico Capponi: astronave e logica di gioco . . . 14
  - 2.2.3 Francesco Teo Calzolari: mappa e relativo controller . . . 17
  - 2.2.4 Claudia Falcone: input e proiettili . . . 20
- 2.3 View . . . 23

3 Sviluppo **25**
- 3.1 Testing automatizzato . . . 25
- 3.2 Note di sviluppo . . . 26
- 3.3 Note finali . . . 28
  - 3.3.1 Lasse . . . 28
  - 3.3.2 Federico . . . 28
  - 3.3.3 Francesco . . . 29
  - 3.3.4 Claudia . . . 29

4 Guida Utente **30**

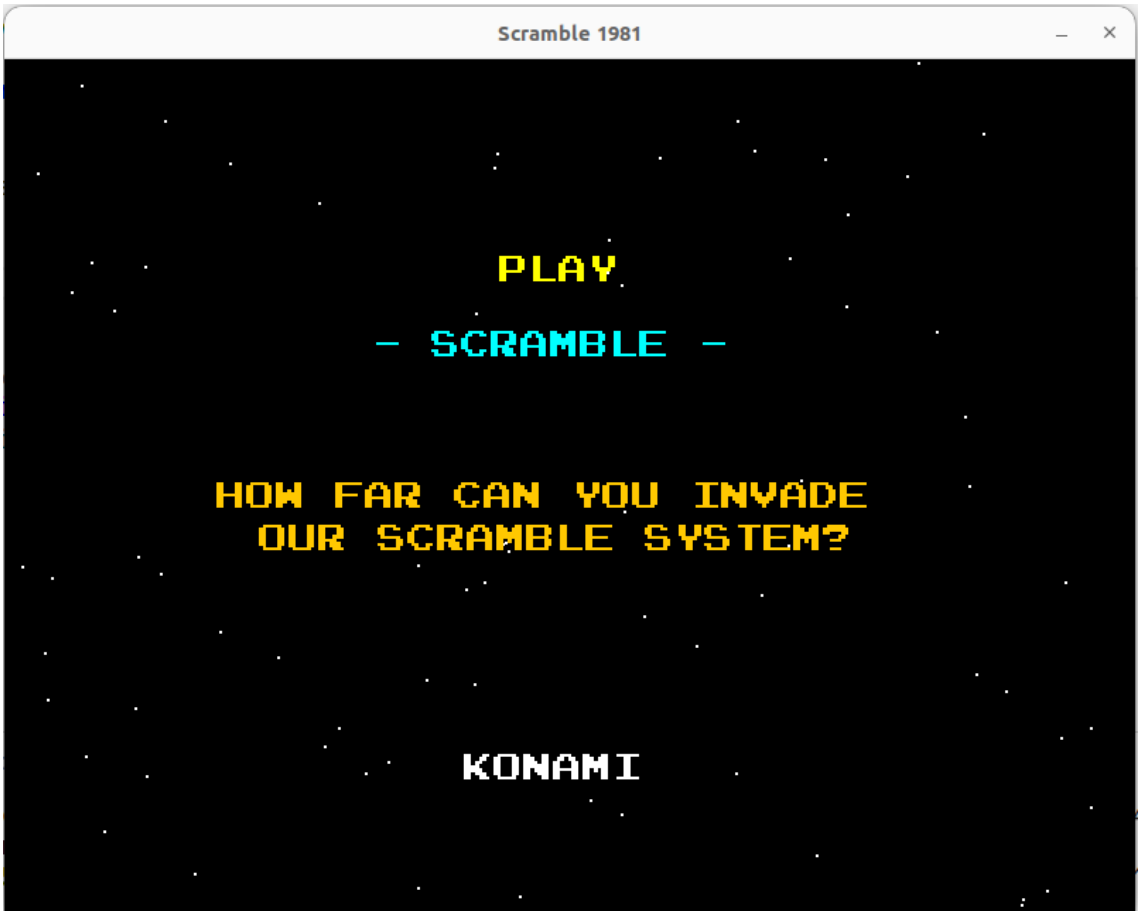

Figura 1: Starting Screen

Capitolo 1

Analisi

Il gioco preso in analisi è uno scrolling shooter arcade bidimensionale. Ovvero il giocatore prende posseso di una navicella nello scenario di gioco e deve raggiungere la fine del livello, senza venire colpita e mantenendo una ricarica di carburante necessaria per il viaggio.

Tra gli ostacoli rilevati durante il livello, anche i nemici (che si muovono nello scenario) possono infliggere danno al giocatore. Una volta colpiti, rilasciano un punteggio, che si accumula a fine livello.

Il terreno nella mappa è considerato una barriera e un impatto con esso causa perdita della vita di gioco. Una volta esaurite tutte le vite, il giocatore si ritroverà al menu principale per decidere di cominciare una nuova partita.

1.1 Descrizione

Il gioco si svolge all'interno di una mappa che raffigura la superficie di un pianeta. Nel mondo creato vi sono diversi elementi che interagiscono tra loro in vari modi. Il giocatore che muove l'astronave può muoversi fino a metà schermo e poi gli sembrerà di sbattere contro una parete invisibile.

L'altra azione che il giocatore può effettuare è lanciare degli attacchi sotto forma di proiettili o bombe. I proiettili posso essere sparati in modo costante, mentre possono essere rilasciate in aria massimo 2 bombe per volta: nel momento in cui una bomba esplode a causa di una collisione, al giocatore sarà di nuovo permesso lanciare una nuova bomba.

Le bombe partono dall'astronave e viaggiono verso il terreno attirate dalla

gravità del mondo. Ci sono varie cose che la bomba può colpire:

1. il terreno →non succede nulla;
2. un serbatoio →si ricarica una parte del carburante;
3. un nemico →il nemico esplode e non ci darà più fastidio. Ogni nemico ha dei punti che, in caso venga distrutto dal giocatore, vengono sommati al punteggio finale.

I nemici possono interagire con il giocatore solo attraverso le collisioni. Infatti, se un nemico riesce a collidere con il giocatore, egli perderà una vita con conseguente teletrasporto all'ultimo checkpoint raggiunto, che corrisponde all'inizio dello stage attuale. Una volta esaurite tutte le vite, il giocatore si ritroverà al menu principale e potrà decidere di cominciare una nuova partita.

Anche le collisioni con il terreno o altri elementi di gioco comporteranno la perdita di una vita.

1.2 Requisiti

L'applicazione vorrebbe ricreare fedelmente il gioco originale, facendo in modo di essere portatile per poter eseguire questo gioco retro sulle moderne piattaforme.

Si partirà da uno scheletro base, con funzionalità limitate, in modo da carpire le meccaniche di gioco e riscriverle in un liguaggio ad oggetti trasformando un antico codice assembly in un linguaggio di programmazione ad alto livello.

Ad un attenta analisi su simulazioni e video di gioco, siamo riusciti a destrutturare il gameplay e il game engine come segue:

1.2.1 Casistica

Il software, che rappresenta una modernizzazione del videogioco Scramble 1981[1], mira a intrattenere e a sorprendere gli appassionati di retro gaming.

[1] `https://en.wikipedia.org/wiki/Scramble_(video_game)`

Il gioco originale rimane uno dei primi a fare uso di un background scorrevole, che non solo delimita la mappa di gioco, ma è anche parte integrante del gameplay. Infatti, la navicella (il giocatore) non deve venirne a contatto, pena la perdita di una vita e l'eventuale Game Over.

Requisiti funzionali

- Il player può:
  - Pilotare l'astronave in 4 direzioni
  - Sparare proiettili dall'astronave
- Nella mappa ci sono dei nemici che, se colpiscono l'astronave, comportano la perdita di una vita
- All'avvio del gioco il player ha a disposizione 3 vite
- Il gioco termina quando il numero di vite arriva a 0
- Una vita si perde quando:
  - L'astronave viene colpita da un nemico
  - L'astronave collide col terreno della mappa
- L'astronave è alimentata dal carburante, la cui riserva diminuisce al progredire del gioco
- Nella mappa sono presenti dei barili che, se colpiti, ricaricano una parte di carburante dell'astronave
- L'esaurimento del carburante comporta la perdita di una vita
- Ogni perdita di vita comporta il respawn all'ultimo checkpoint raggiunto

Requisiti non funzionali

- Si cercherà di mantenere la grafica e i colori del gioco originario ove possibile
- Il gioco si presenterà con un menu iniziale
- Il software:
  - deve essere eseguibile in modo relativamente fluido.

  - deve essere compatibile con una varia gamma di hardware e sistemi operativi (Linux, Windows e MacOS).

- L'applicativo dovrà essere più efficente possibile, eliminando anche prima del garbage collector tutti gli elementi di gioco non più presenti sullo schermo, quindi: quelli dei livelli già superati o anche quelli che escono dallo schermo o distrutti dal giocatore. Dovrà anche generare parsimoniosamente gli elementi di gioco che ancora devono apparire sullo schermo, non caricandoli dall'inizio ma creandoli dinamicamente.

- Il programma dovrà essere privo di comportamenti non previsti con un esecuzione fluida e una meccanica di gioco intuitiva

- Ogni elemento di gioco avrà una sua tipologia di movimento, abbinata ad una animazione relativa al tipo.

1.2.2 Modello del dominio

Alla luce dei requisiti individuati e precedentemente descritti, è stato ricavato il seguente schema UML che definisce il modello del dominio che si vuole rappresentare.

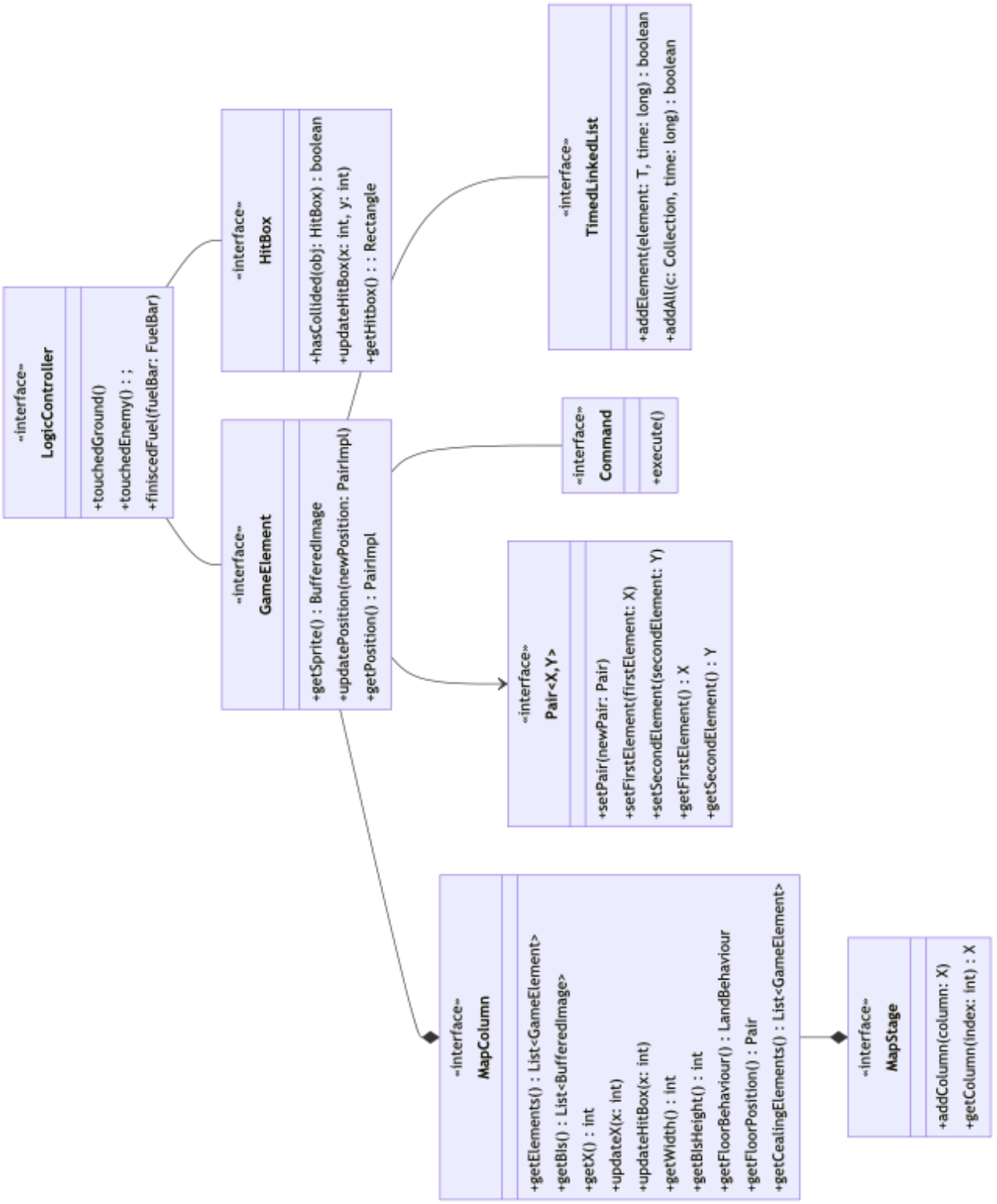

Figura 1.1: UML Generale

Capitolo 2

Design

L'architettura di Scramble 1981 segue il pattern architetturale MVC. Ognuna delle macroclassi del pattern MVC è stata suddivisa in più sottoclassi che descrivono gli aspetti del gioco da implementare.

- **Controller**: abbiamo diviso le classi controllori a seconda della loro funzione, abbiamo trovato necessario dividere la gestione dell'input, della mappa e della logica di gioco rispettivamente in input e command, map e mediator.

- **Model**: la caratterizzazione degli elementi che compongono l'architettura del gioco è stata descritta ramificando da un punto di partenza "common" le implentazioni dell'interfaccia GameElement. Elementi necessari per la gestione delle mappe e delle hitbox sono stati anch'essi inclusi nel model. Come risultato classi specifiche come quella del'astronave, ad esempio, estendono un wrapper di Rectangle e l'implentazione di GameElement.

- **View**: la grafica è strutturata partendo da estensioni di classi presenti nella libreria Swing. L'applicativo è stato realizzato partendo da un JFrame unico, con l'aggiunta di un singolo pannello centrale, composto a sua volta da tanti pannelli sovrapposti, ognuno con la sua profondità, che interagiscono tra loro. Ogniuno dei pannelli che gestiscono le diverse parti del model, estende la classe GamePanel.

2.1 Architettura

L'architettura del gioco originale, sviluppato in assembly, ci è sembrato suddividesse essenzialmente il gioco in elementi valicabili (lo spazio), elementi interagibili (nemici, carburante, mappa) ed elementi controllabili (astronave, proiettili).
Per questo, il pattern architetturale Model-View-Controller sembrava essere applicabile anche ad un gioco molto vecchio, in cui magari non era ancora apparso come stile di programmazione.

Alla base dell'architettura è stato definito un punto comune per la maggior parte delle sezioni di gioco. Gli elementi che condividono parte delle meccaniche sono stati raggruppati in modo tale che implementino un'interfaccia comune (o estendano la sua implementazione base).

L'entry point, quindi, del gioco è sicuramente l'interfaccia GameElement, implementata anche dalla classe MapElement, a ribadire che ogni elemento che compare sullo schermo rappresenti un elemento di gioco.
Questa interfaccia viene inoltre implementata da una classe astratta GameElementImpl che, seguendo il Template Method, estende anche la classe per i box di collisioni (Hitbox) e funge da base itinerante per ogni altro elemento di gioco.

Un altro esempio è dato dalle classi SpaceShip e Rocket che estendono la classe GameElementImpl e, seguendo il pattern Decorator, definiscono nuove funzionalità all'elemento base di gioco, mantendone le identità comuni.
Come illustrato nell'UML qui sotto, infatti, le due classi appena citate (Rocket e SpaceShip), nonostante estendano entrambe GameElementImpl, hanno un comportamento per il movimento totalmente differente: una viene controllata dall'utente, l'altra ha un movimento predefinito.

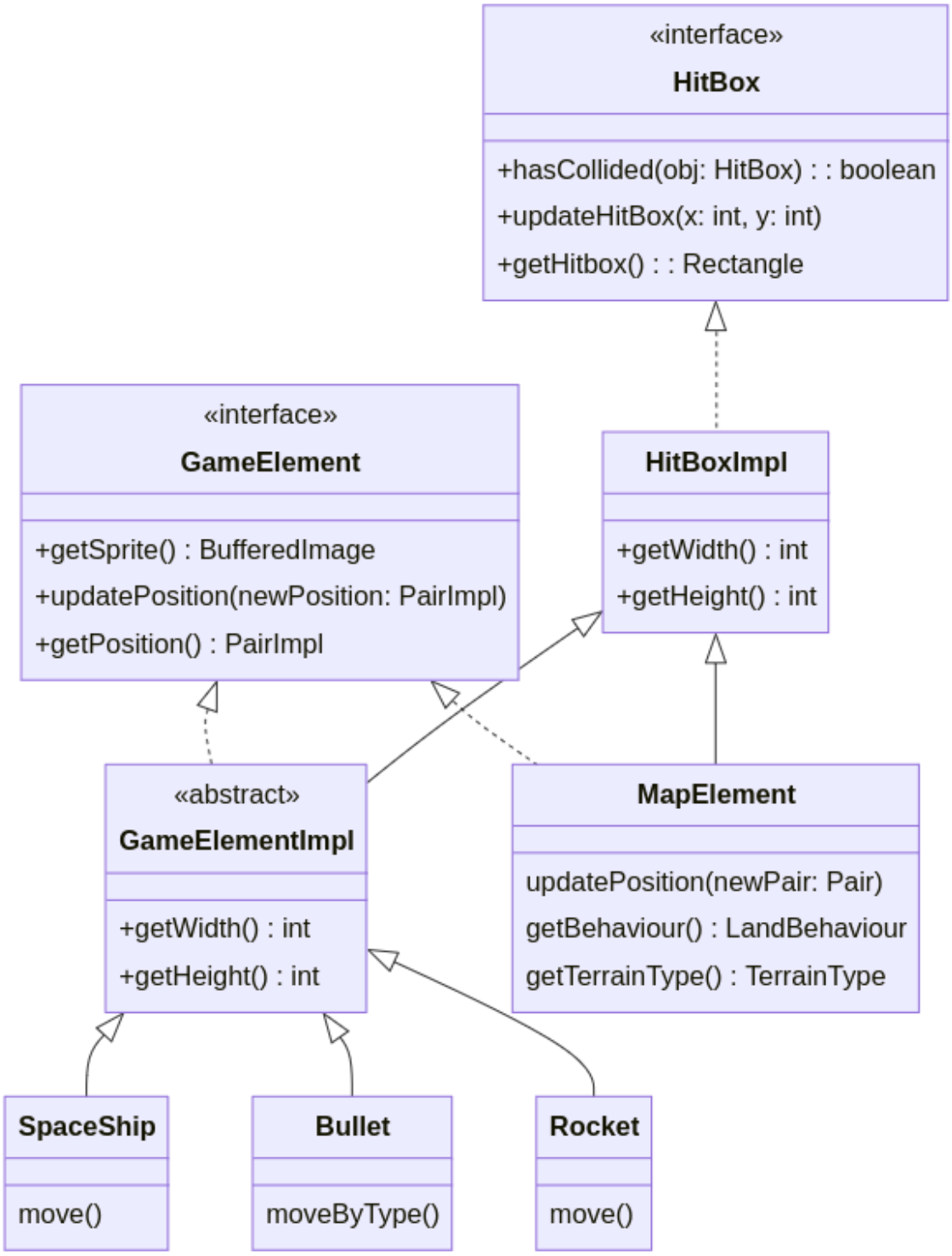

Figura 2.1: UML Architetturale

2.2 Design Dettagliato

A seguire un dettaglio per ogni singolo componente del gruppo, sviluppato in autonomia. Facciamo presente che nonostante siamo a suddividere qui il lavoro, seguendo lo scopo del progetto in sè, molta della programmazione effettiva è stata eseguita in comune. Partendo quindi da basi solide ma condivise, la progettazione rimane divisa in sezioni personali che tuttavia potrebbero avere alcuni punti di contatto, come la classe GameElement e GamePanel.

2.2.1 Lasse Werpers: nemici e collisioni tramite hitBox

HitBox è una delle classi portanti della logica di gioco in quanto qualsiasi collisione gestita da LocigController passa da questa classe.

Problema: Come creare entità che collidono tra loro?
Soluzione:
Per gestire le HitBox è stato scelto di appoggiarsi sulla classe Rectangle.
Al rectangle vengono passati le dimensioni e la posizione di un oggetto e lui traccia un rettangolo attorno ad esso. La classe Rectangle a sua volta ha un metodo [intersects] che controlla se 2 oggetti Rectangle hanno dei punti in comune (intersezione) che noi andiamo ad utilizzare per rilevare una collisione tra hitbox.
Inizialmente avevamo pensato di implementare le HitBox utilizzando Polygon per avere una HitBox più accurata. Ripensandoci abbiamo però ritenuto questa implementazione inutilmente complessa dato che nel caso avessimo deciso di modificare le dimensione di un oggetto avremmo dovuto ricalcolare i punti e le proporzioni da passare nella fase di costruzione del GameElement. Inoltre gli spostamenti con il conseguente update della hitbox, risulta di gran lungo più veloce a runtime e concettualmente più facile, in quanto basta solo passare le nuove coordinate del centro del GameElement, la HitBox si muove di conseguenza.
Per la gestione delle hitbox e delle collisioni, abbiamo adottato il principio di **Composition over Inheritance**. Invece di affidarmi a una gerarchia rigida di classi, ogni `GameElement` contiene una `Hitbox` come componente separato. Questo approccio rende più semplice e flessibile l'aggiornamento e la gestione delle hitbox, poiché è possibile modificare la hitbox di un elemento senza influenzare la struttura o il comportamento generale dell'oggetto. In particolare, ciò facilita l'aggiornamento delle coordinate e delle dimensioni delle hitbox a runtime.

LogicController va poi a richiamare HitBox dalle varie implementazioni di GameElement per controllare le collisioni con altri elementi di gioco.

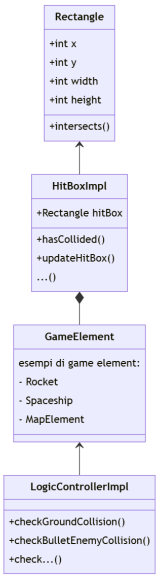

Figura 2.2: Hitbox Implementation

Figura 2.3: HitBox in game

Problema: Come far muovere il Rocket con un Delay nella partenza.
Soluzione:
Per gestire il delay è stato deciso di utilizzare un timer che parte nel momento in cui viene chiamato il metodo turnOnMove. Nel momento in cui viene chiamato turnOnMove viene associato ad un timer un task e un delay apposito. Il delay viene generato in modo pseudorandomico. alla creazione del Rocket.

Il rocket viene creato appena entra nel FOV (field of view) del giocatore e quindi parte subito il timer del task.
Il task cambia semplicemente lo stato del rocket da PREMOVE a MOVING.

Nello stato PREMOVING il Rocket scorre verso sinistra con la stessa velocità del background, dando l'impressione di stare fisso a terra.

Nello stato MOVING il Rocket inizia effettivamente il suo movimento verticale aggiornando la velocità Y mantenendo la velocità X del PREMOVE per dare l'impressione che decolli il razzo.

Ulimo stato è EXPLODED dove il razzo riprende il movimento solo della x e la y si blocca, dando così l'impressione che il razzo esploso rimanga fermo sul background scorrevole.

2.2.2 Federico Capponi: astronave e logica di gioco

Problema: Ciclo di gioco.
Tra le difficoltà più grandi incontrate nello sviluppo del gioco c'è stato sicuramente la stesura di un ciclo di gioco strutturato. Si è dovuta definire la gestione corretta di tutti gli elementi concorrenti allo svolgimento del gameplay.

Soluzione: Nella classe LogicController, tramite uso di Timer adatti (Adapter), si verificano tutti i controlli per la gestione delle vite e soprattutto del Game Over (vite = 0). La classe gestisce, inoltre, i punti di salvataggio generati dalla view per permettere di procedere nel gioco nel caso in cui non sia sopraggiunto il game over ma si sia persa una vita.

Problema: Movimento oggetti sullo schermo.
Per quanto riguarda, invece, il movimento degli oggetti sullo schermo e il movimento dell'astronave (l'unica parte controllata dal giocatore insieme al lancio dei proiettili) che inizialmente non poteva muoversi in diagonale, le specifiche del gioco prevedevano non potesse superare circa la metà dello schermo al massimo.

Soluzione: Anche altri elementi del gioco si muovono sullo schermo come l'astronave ma seguono logiche diverse alla base. Nella classe FuelTank, che estende GameElementImpl e che può essere presa come esempio, il movimento è solo orizzontale per seguire lo scorrimento della mappa. Per l'astronave, invece, si è dovuto gestire il movimento a parte, in modo che fosse fluido e omogeneo.
A tutti gli effetti, a parte l'astronave, il movimento avviene in maniera trasparente. Al giocatore sembrerà che gli oggetti siano fermi e che sia l'astronave a muoversi in avanti e a lasciare il resto indietro. Nella realtà, l'astronave si muove solo all'interno della porzione di schermo occupata dal pannello ad essa dedicato.

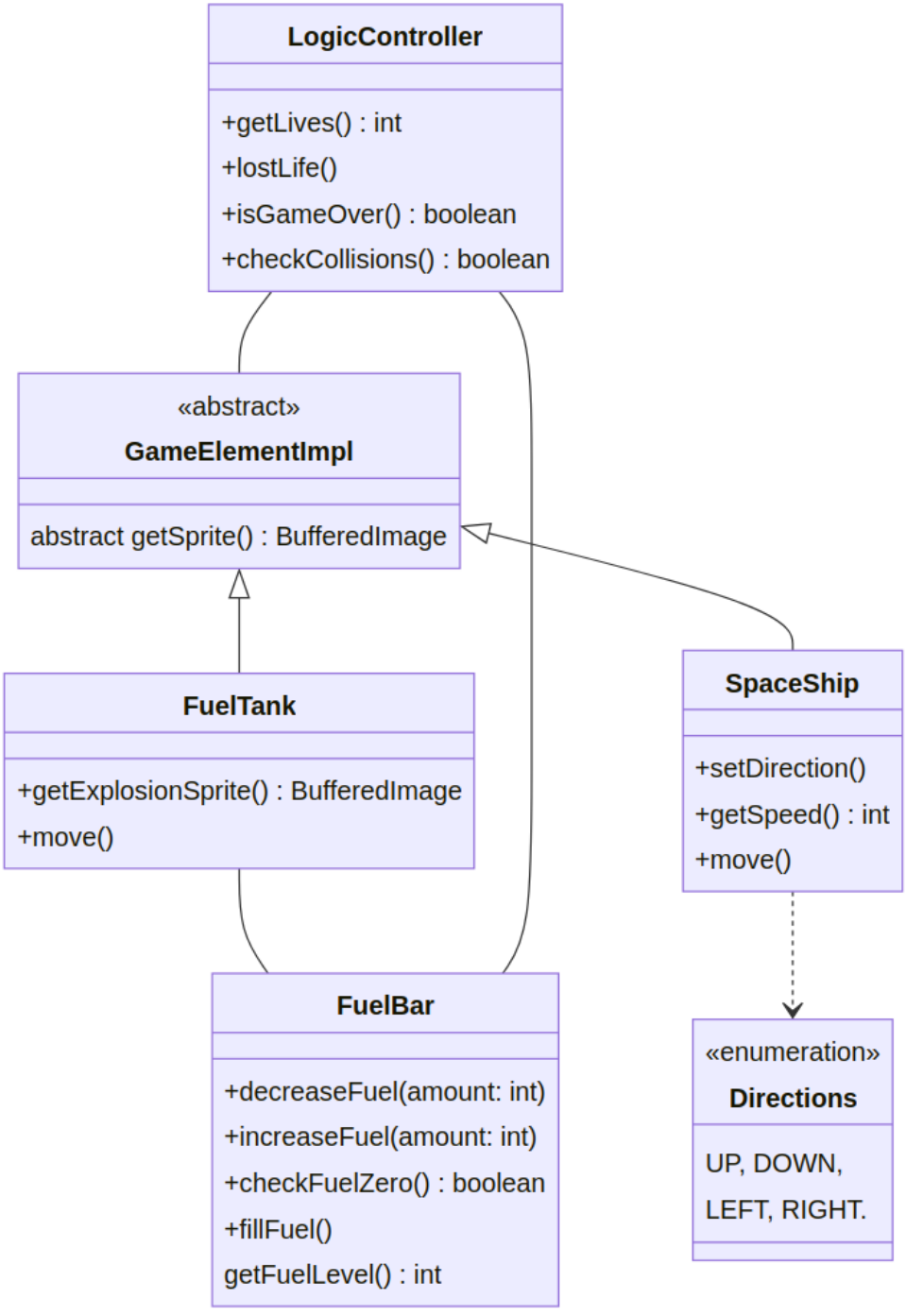

Figura 2.4: Collisions and game logic

Problema: Generazione e carimento sprite di gioco.
Oltre ai punti precedenti, un'altra problematica consisteva nell'implementazione dell'acquisizione e della gestione delle sprite per l'animazione: in una classe infatti nella visualizzazione delle immagini sarà statica, mentre nelle altre potrebbe essere dinamica e, alle volte anche casuale al fine di generare una specie di animazione.

Soluzione: Seguendo il pattern Template Method, il metodo di ottenimento dell'immagine è dichiarato astratto: in questo modo le implementazioni successive sono libere di risolvere questo inghippo successivamente. Le immagini del gioco sono fortunatamente state facilmente reperibili online, da un repository GitHub di una versione del gioco scritta in GDScipt per l'IDE di gioco Godot. [1]

[1] `https://github.com/simonalanjones/godot-scramble`

2.2.3 Francesco Teo Calzolari: mappa e relativo controller

Il mio ruolo nel gruppo riguardava la generazione dei vari stage di gioco e la visualizzazione di essi in un'unica mappa.

Problema: Come generare una mappa bidimensionale divisa in stage dove per ogni vi è una componente soffitto e una componete terra che assumono divesi comportamenti?

Soluzione: Per la generazione dei vari stage, ne ho descritto il comportamento in file csv divisi per stage e componente (soffitto e pavimento) e ho creato una classe astratta CSVReader per la lettura e l'immagazzinamento dei vari stage in dati grezzi, composti a loro volta da segmenti che poi verranno raffinati per la creazione dello stage di gioco. Ogni classe RawData (contenente i dati greszzi) contiene sia il pavimento che il soffitto dello stage.

Problema: Non esiste un solo tipo di stage ma ne esistono di varii tipi(alcuni sono una landa verde, altri sono composti da colonne di mattoni).

Soluzione: Come risoluzione a questo problema ho reso generica la classe astratta CSVReader e ho creato delle classi adHoc per la lettura di file CSV che descrivono comportamenti in modo diverso ma che restituiscono tutti una lista di SegmentRawData che costituisce uno dei due componenti di RawData (pavimento o soffitto).

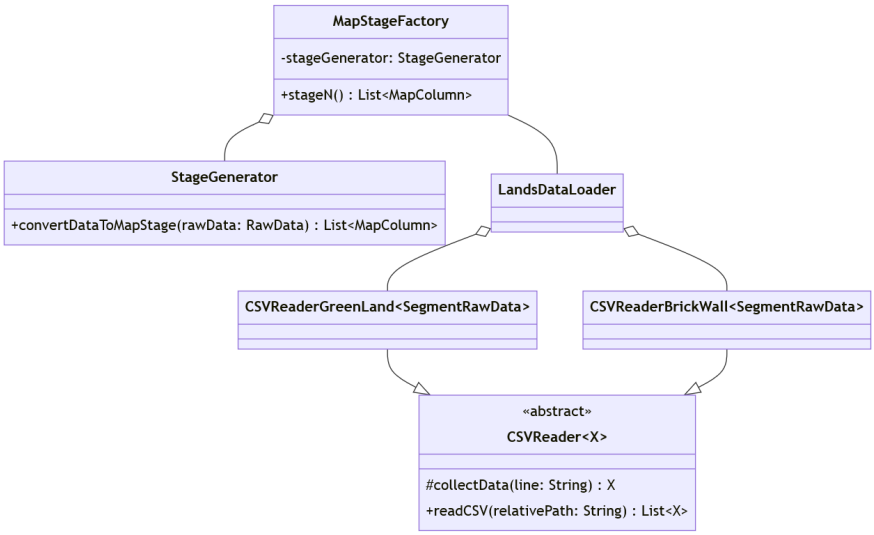

Figura 2.5: CSVReader and MapStageFactory

Problema: Generazione degli stage

Soluzione: Ho deciso di utilizzare il design pattern factory method per la generazione dei vari stage creando un'interfaccia MapStageFactory composta da metodi quali prestage, stage1, ecc.. che restituiscono gli stage estratti dai CSVReader tramite una classe LandsDataLoader.

Per fare ciò utilizza una classe StageGenerator la quale legge i RawData e ne genera una mappa.

Problema: Come assegnare le sprite giuste al terreno, dal momento che nei csv vi è solo il comportamento del terreno ma non l'aspetto visivo?

Soluzione: Per la scelta delle sprite del terreno ho deciso di randomizzare le sprite in base al comportamento del terreno. Nel caso, ad esempio, il terreno si comporti in modo piatto (LandBehaviour.FLAT), lo StageGenerator gli assegnerà una sprite idonea al tipo di comportamento, se invece il terreno è crescente (LandBehaviour.UP), gli verrà assegnata una sprite che ne descriva visivamente il comportamento.

Problema: Rappresentazione dei vari stage per evitare di caricare tutta la mappa nella parte di view.

Soluzione: Ho deciso di utilizzare una lista che conterrà le colonne che

compongono uno stage in modo sequenziale. Per le colonne ho creato la classe MapColumn che fornisce tutti i metodi necessari per la gestione e visualizzazione di una colonna.
In MapColumn vi è una lista di MapElement, ovvero un implementazione di GameElement per seguire la filosofia comune dove tutto qello che si vede a sechermo è un GameElement.

Il MapController è il controllore che gestisce la mappa. Il suo compito principale è quello di fornire al LandscapePanel (un GamePanel che si occupa in modo specifico della mappa di gioco) le colonne che andranno visualizzatem, onde evitare di caricare l'intera mappa per poi doverla ridisegnare continuamente.

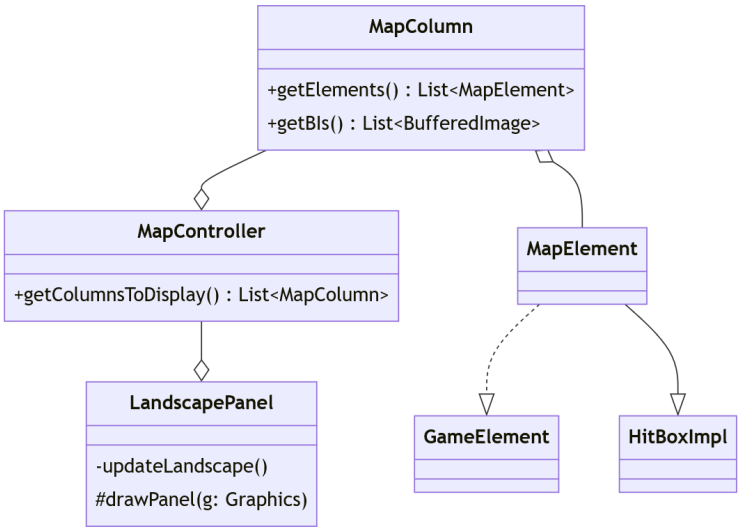

Figura 2.6: MapController

2.2.4 Claudia Falcone: input e proiettili

Problema: Come separare l'input del giocatore dalla logica di esecuzione di queste azioni o modificare i controlli senza influire sul resto del codice?

Soluzione: Per la gestione dell'input si è deciso di utilizzare il pattern Command per suddividere completamente il rilevamento degli input dalla logica di esecuzione dei comandi. Invece di legare direttamente l'input a metodi specifici (ad esempio, premere un tasto per sparare), l'input è mappato a un oggetto Command che sa come eseguire l'azione corrispondente.

InputControl non è tenuto a conoscere i dettagli del comando ma il suo compito è che quando rileva un input, crea un oggetto di tipo Command (come SpaceshipCommand o BulletCommand). SpaceshipCommand e BulletCommand implementano la logica effettiva. Ricevono i parametri necessari dall'InputControl o da altre parti del sistema e li utilizzano per eseguire l'azione. Ad esempio, quando l'utente preme il tasto per sparare, InputControl potrebbe generare un'istanza di BulletCommand e chiamare execute(). BulletCommand conosce i dettagli su come creare un proiettile e aggiungerlo al gioco.

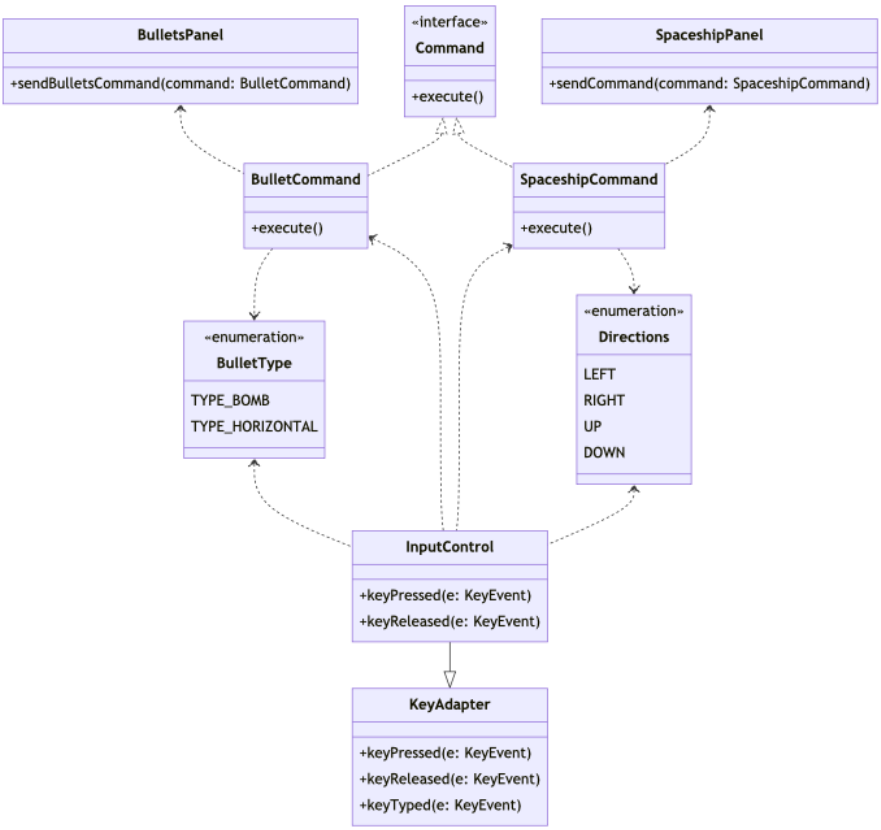

Figura 2.7: InputController

Problema: Come far sparire dopo un certo tempo l'animazione dell'esplosione di una bomba?
Soluzione: Con una lista personalizzata dove ogni elemento viene rimosso automaticamente dopo un determinato intervallo di tempo. La classe TimedLinkedListImpl utilizza internamente una LinkedList ma la adatta per fornire funzionalità aggiuntive: ogni elemento viene automaticamente rimosso dopo un intervallo di tempo, grazie al ScheduledExecutorService. TimedLinkedListImpl funge da adapter che estende una lista tradizionale con una funzionalità di temporizzazione.

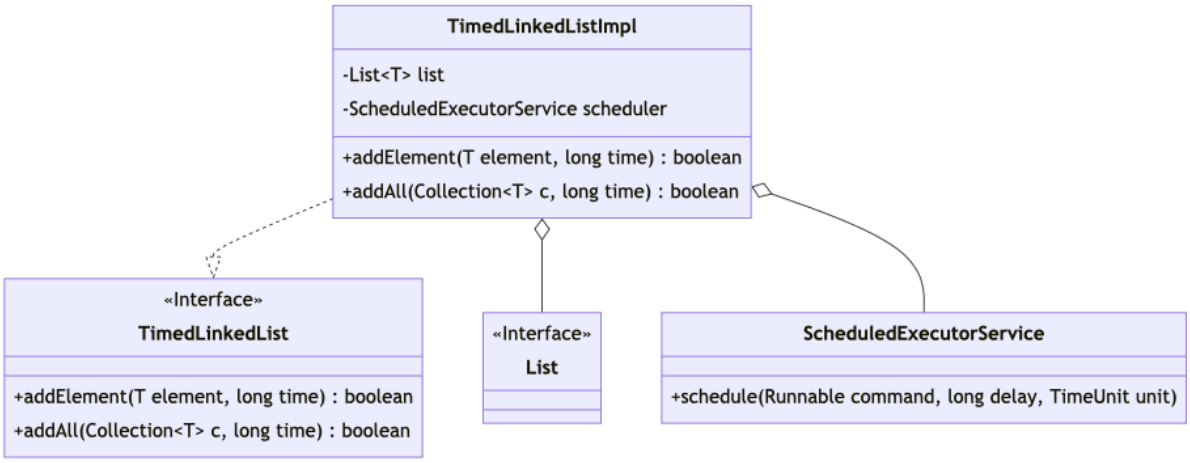

Figura 2.8: TimedLinkedList

2.3 View

Per la parte di View, come già detto, abbiamo deciso di utilizzare la libreria grafica java.swing. Abbiamo deciso quindi di utilizzare il pattern di Adapter Decorator per creare JPanel che si adattassero ad ogni elemento presente sullo schermo. Abbiamo quindi creato una classe astratta, GamePanel, che ci permettesse di adattare JPanel alle nostre esigenze.
Abbiamo poi esteso la classe astratta utilizzando il pattern Decorator per creare pannelli ad hoc a seconda delle funzioni che lo stesso pannello debba eseguire. Esisterà, quindi, un pannello dedicato per il background, uno per l'astronave, uno per i nemici, uno per il carburante, uno per i proiettili e via dicendo.

I pannelli oltre all'interazione dei vari elementi sullo schermo, si occupano anche della generazione degli oggetti di gioco contenuti al loro interno, mantenendo un riferimento ai relativi oggetti del model.

Le uniche "classi pannello" che differiscono dalle altre sono StartMenu e GameOverPanel. Questo in quanto entrambe per certi periodi di gioco prendono il sopravvento su tutti gli altri Panel che, invece, tipicamente sono attivi durante il gioco.
Difatti la prima viene presentata all'inizio del gioco e nel caso il giocatore perda tutte le vite a sua disposizione dopo una run (il gioco inizia premendo start su questa schermata). La seconda, invece, compare solo alla fine di tutti i livelli, ed avviene quindi al completamento del gioco (compare per qualche secondo per poi restituire il comando al menu start).

Per modulare il controllo delle schermate, la classe GameView (che è il JFrame, la finistra di gioco) aggiunge e rimuove i pannelli all'occorrenza.

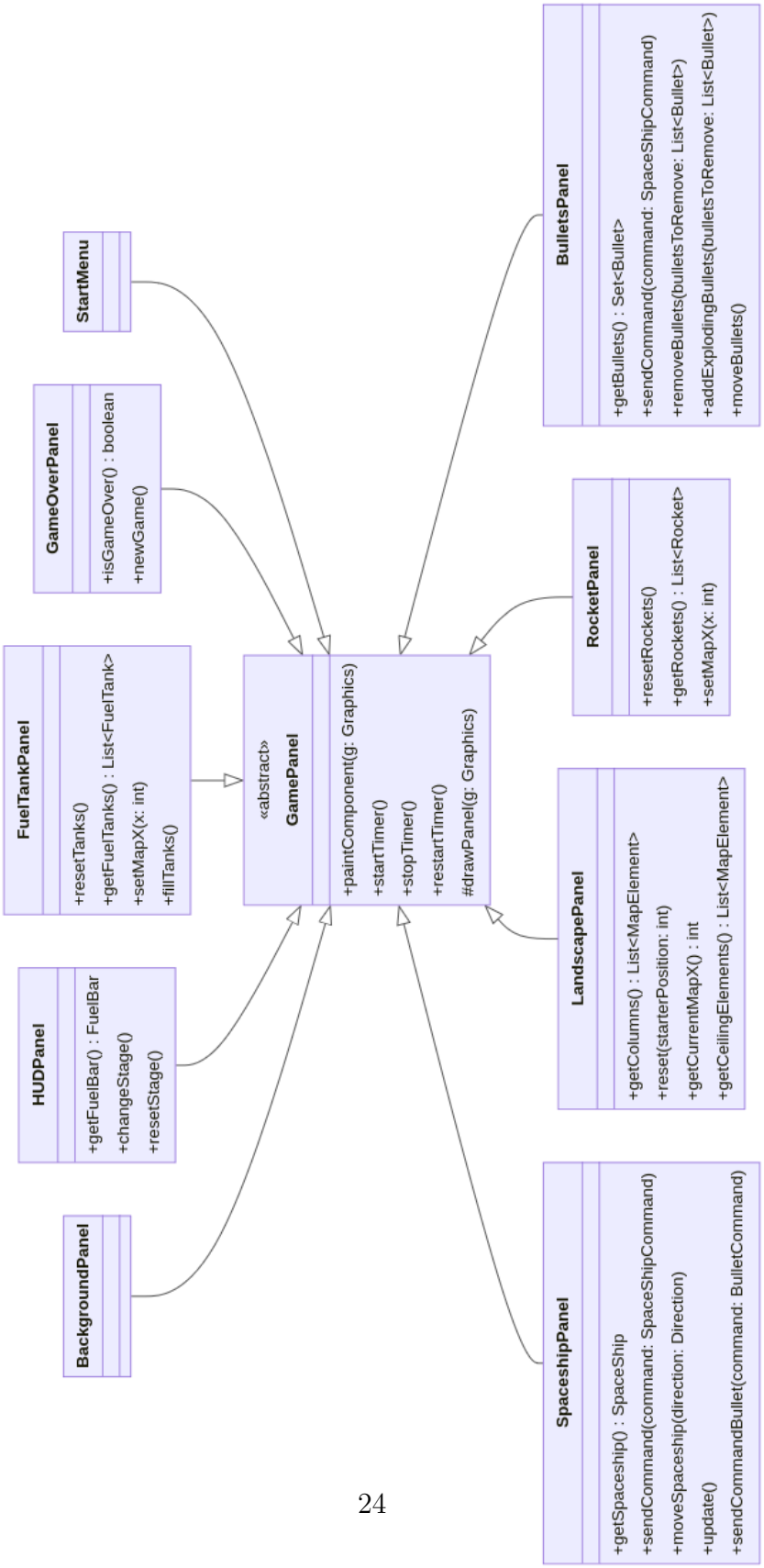
StartMenu
GameOverPanel
+isGameOver() : boolean
+newGame()
FuelTankPanel
+resetTanks()
+getFuelTanks() : List<FuelTank>
+setMapX(x: int)
+fillTanks()
HUDPanel
+getFuelBar() : FuelBar
+changeStage()
+resetStage()
BackgroundPanel
«abstract»
GamePanel
+paintComponent(g: Graphics)
+startTimer()
+stopTimer()
+restartTimer()
#drawPanel(g: Graphics)
BulletsPanel
+getBullets() : Set<Bullet>
+sendCommand(command: SpaceShipCommand)
+removeBullets(bulletsToRemove: List<Bullet>)
+addExplodingBullets(bulletsToRemove: List<Bullet>)
+moveBullets()
RocketPanel
+resetRockets()
+getRockets() : List<Rocket>
+setMapX(x: int)
LandscapePanel
+getColumns() : List<MapElement>
+reset(starterPosition: int)
+getCurrentMapX() : int
+getCeilingElements() : List<MapElement>
SpaceshipPanel
+getSpaceship() : SpaceShip
+sendCommand(command: SpaceShipCommand)
+moveSpaceship(direction: Direction)
+update()
+sendCommandBullet(command: BulletCommand)

Figura 2.9:

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Abbiamo utilizzato la suite JUnit per il test automatico. Di seguito le classi in esame:

- BulletTest in cui viene testato il movimento del proiettile in base alla tipologia, la sua collisione con la mappa e il corretto funzionamento delle sprite.
- GameElementImplTest, verifica l'inizializzazione dei valori del costruttore, l'aggiornamento della posizione dell'oggetto GameElementImpl e l'implementazione del metodo getSprite nelle sue sottoclassi.
- HitBoxTest nel quale si testa il corretto funzionamento della collisione tra le hitbox, l'aggiornamento della posizione della hitbox e la corrispondenza tra il rettangolo restituito e quello previsto.
- RocketImplTest in cui viene testato il movimento del Rocket in base alla velocità predefinita, che le sprite del razzo si alternino correttamente (utile per l'animazione) e che il razzo rilevi correttamente le collisioni con i proiettili.
- MapColumnTest in cui viene testato la corretta inizializzazione della classe che rappresenta una colonna di tile nella mappa, il corretto comportametno nel caso vi siano inseriti elementi di vario tipo (esempio: colonne di mattoni o landa verde) e la restituzione del comportamento del terreno utile per lo spawn dei nemici e dei fuel tank.
- MapElementTest in cui viene testata l'inizializzazione di un map element e l'aggiornamento di hitbox e posizione.

- LandscapePanelTest in cui si verifica che la posizione corrente della mappa sia stata correttamente aggiornata al valore fornito.
- FuelBarTest in cui viene testato che la barra del carburante inizi con il valore massimo predefinito, la riduzione e l'aumento del carburante dell'importo atteso.
- SpaceShipTest, in cui viene testata la corretta creazione dell'istanza di SpaceShip e la corretta impostazione delle sue proprietà iniziali, la corretta rilevazione delle collisioni tra l'astronave e i nemici e il suo movimento in base al tipo di direzioni.

3.2 Note di sviluppo

Utilizzo di di file .json per il salvataggio degli scores
Permalink:
`https://github.com/2Skali3/Scramble1981_Remastered/blob/master/app/src/main/java/scramble/model/scores/Scores.java#L136-L152`

Utilizzo di lambda expressions
Permalink:
`https://github.com/2Skali3/Scramble1981_Remastered/blob/master/app/src/main/java/scramble/view/compact/LandscapePanel.java#L53`

Utilizzo di generici quando opportuno:
Permalink:
`https://github.com/2Skali3/Scramble1981_Remastered/blob/master/app/src/main/java/scramble/model/common/api/Pair.java#L9`

Utilizzo di stream:
Permalink:
`https://github.com/2Skali3/Scramble1981_Remastered/blob/master/app/src/main/java/scramble/controller/mediator/LogicController.java#L248-L253`
`https://github.com/2Skali3/Scramble1981_Remastered/blob/master/app/src/main/java/scramble/view/compact/BulletsPanel.java#L81-L84`
`https://github.com/2Skali3/Scramble1981_Remastered/blob/master/app/src/main/java/scramble/view/compact/BulletsPanel.java#L108-L110`
`https://github.com/2Skali3/Scramble1981_Remastered/blob/master/app/src/main/java/scramble/view/compact/BulletsPanel.java#L193`

https://github.com/2Skali3/Scramble1981_Remastered/blob/master/app/src/main/java/scramble/model/common/impl/TimedLinkedListImpl.java#L48
https://github.com/2Skali3/Scramble1981_Remastered/blob/master/app/src/main/java/scramble/model/enemy/Rocket.java#L123
https://github.com/2Skali3/Scramble1981_Remastered/blob/master/app/src/main/java/scramble/model/enemy/Rocket.java#L133

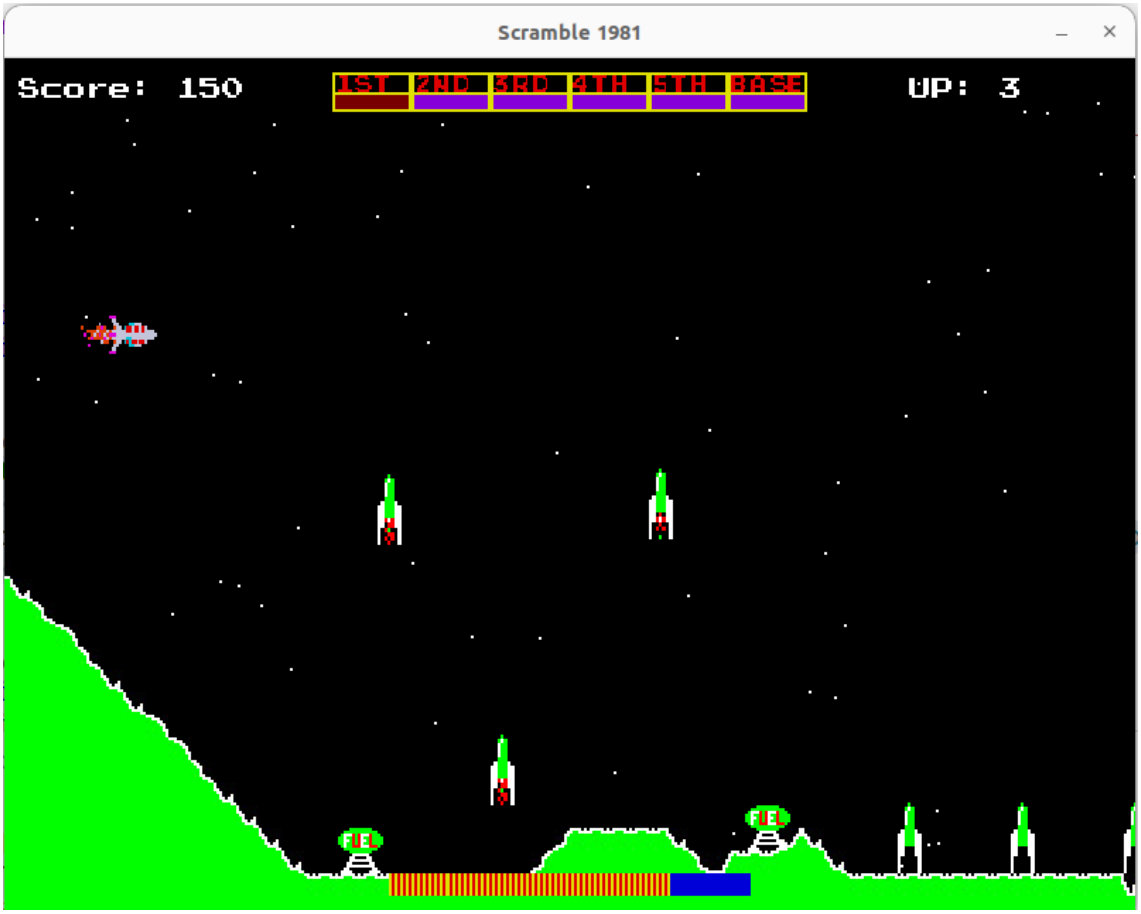

Figura 3.1: Gameplay screenshot

3.3 Note finali

Vogliamo sottolineare ancora come lo svolgimento di questo progetto sia stato fatto molto in collaborazione l'un l'altro. Abbiamo usufruito appieno di questa opportunità per crescere come persone ed anche come programmatori, imparando e confrontandoci l'un l'altro.
Nel nostro caso, forse più che in altri, il lavoro di tutti i membri del gruppo confluiva nel git come un unico flusso di pensieri, e modifiche collaborative del codice erano all'ordine del giorno (come mostrato anche dalla sequenza dei commit).

In conclusione, ci siamo divertiti molto e ringraziamo di averci diciamo "spinti" verso questa opportunità.
Di seguito una piccola descrizione e valutazione del lavoro svolto da parte di ognuno di noi.

3.3.1 Lasse

Mi piace il gioco e lo farò vedere a chiunque mi chieda cosa si fa ad Informatica. Detto ciò, non so se sono portato a fare progetti di gruppo se non per lavoro con orari fissi. Preferisco gestirmi il tempo io e non sentirmi in colpa con i compagni quando ho orari diversi da loro. Un'altra cosa che ho notato è che non mi piace fare solo la mia parte e accontentarmi li, ma preferisco non avere una sola cosa della quale so tutto nel minimo dettaglio, ma sapere un po' meno in dettaglio, ma di qualsiasi aspetto di un progetto in modo da poter proporre idee e miglioramenti a varie parti di codice anche se non sono mie.

3.3.2 Federico

Credo che il codice sia molto resource-savvy, anche se forse un po' confusionario ma piuttosto pulito. Forse avremmo potuto gestire meglio il tempo di sviluppo e di design ma, essendo costretti da una scadenza più corta del solito abbiamo comunque fatto del nostro meglio. Link GitHub personale (per esercitazioni svolte) : `https://github.com/Ke1p5`

3.3.3 Francesco

Questo progetto è stato un'ottima esperienza per comperndere meglio le dinamiche della programmazione in gruppo. Mi ritengo soddisfatto dal codice prodotto e dalle soluzioni adottate da me e dai miei compagni. Un aspetto particolarmente positivo è stata la comunicazione efficace e la disponibilità reciproca, che hanno contribuito a creare un ambiente di lavoro armonioso e collaborativo.

3.3.4 Claudia

Mi ritengo soddisfatta del lavoro svolto da me e dai miei colleghi: le sfide affrontate durante il progetto sono state diverse, alcune anche particolarmente complesse, ma insieme siamo riusciti a gestirle e superarle con successo nonostante le tempistiche limitate. In futuro mi piacerebbe tornare a lavorarci per migliorare il codice e aggiungere features più avanzate.

Capitolo 4

Guida Utente

Da menu iniziale premere "invio" per far partire il gioco. Premere "barra spaziatrice" per lo sparo orizzontale, premere "1" per la bomba. Le freccette direzionali servono per muoversi.

Scopo del gioco è fare più punti possibili, i quali vengono salvati ogni tre vite o all'arrivo della fine dei livelli. I nemici, una volta distrutti dall'astonave valgono tutti (o quasi) 50 punti. Ogni giocata si hanno tre vite, si riparte dall'inizio di un livello raggiunto quando si perde una vita.