CC-Tan!

Paolo Baldini, Gianmarco Magnani, Nicolas Pasolini, Lorenzo Sutera

Agosto 2018

Contents

Chapter 1

Analisi

In questo capitolo tratteremo l'analisi dei requisiti e quella del problema elencando ciò che l'applicazione dovrà fare (requisiti) e l'ambito in cui si colloca (analisi del problema).

1.1 Requisiti

L'applicazione si pone come obiettivo quello di realizzare un gioco ispirato all'app per smartphone CC-Tan. Il giocatore vestirà i panni di una navicella nello spazio fissa al centro che dovrà abbattere i blocchi provenienti da tutte le direzioni prima che questi la colpiscano.

Requisiti funzionali

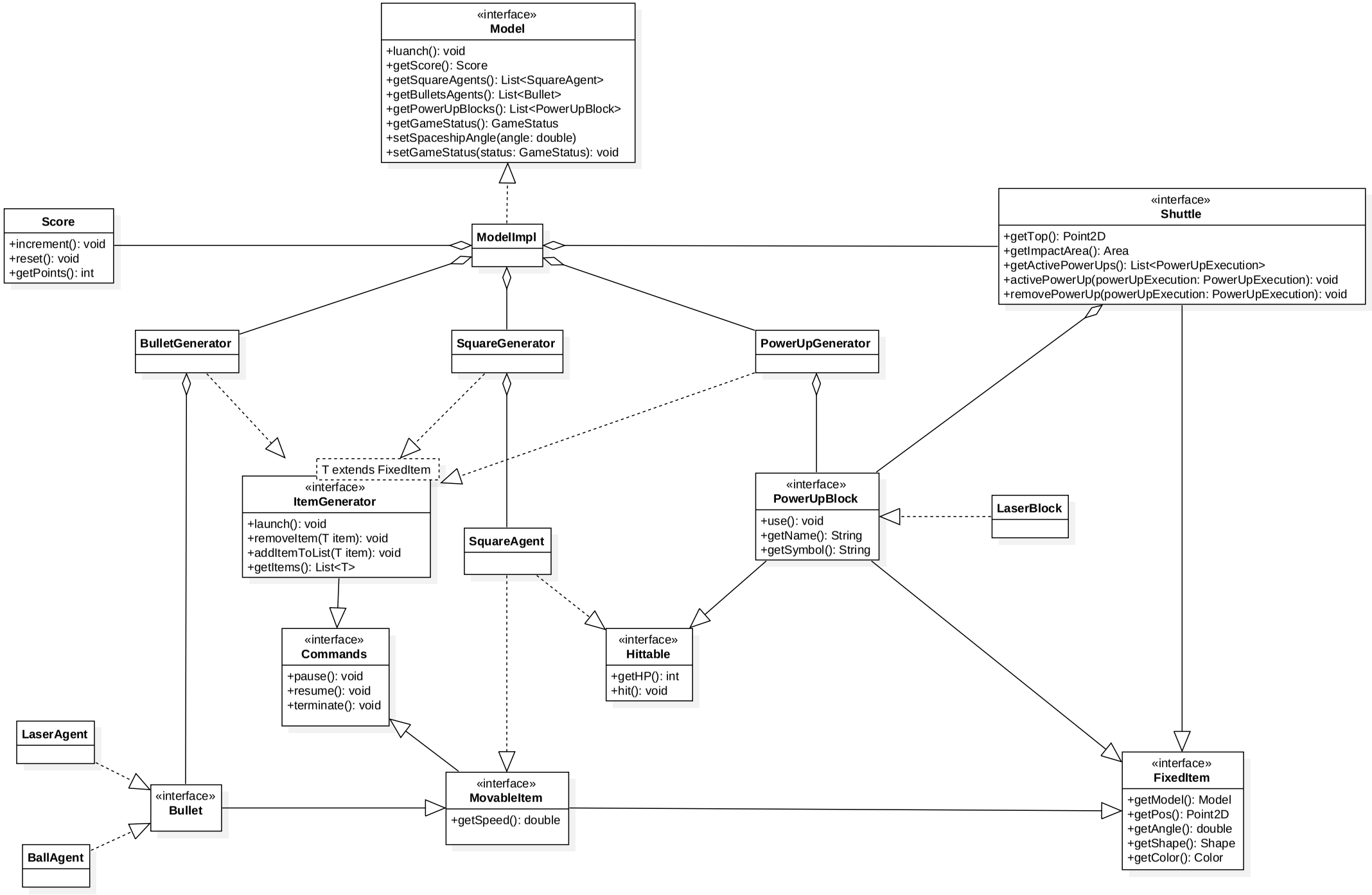
- Creazione dei quadrati con una determinata vita (numero di colpi che deve subire per essere distrutto), generati casualmente;
- generazione uniforme dei quadrati da tutti e quattro i bordi esterni dell'area di gioco, dai quali si dirigeranno verso il centro, dove è presente la navicella;
- all'aumentare del tempo di gioco si deve avere un incremento graduale della frequenza di creazione dei quadrati, della velocità e dei punti vita;
- la navicella sparerà delle palle (proiettili) che dovranno abbattere i quadrati;

- all'aumentare del tempo di gioco si deve avere un incremento graduale della frequenza di creazione delle palle e della loro velocità;
- ogni pallina, all'impatto con un quadrato dovrà rimbalzare adeguatamente, decrementando di uno la sua vita;
- generazione di un blocco power-up laser (potenziamento), che consente di cambiare il tipo di proiettile, sostituendo le palline con un raggio laser in grado di recare maggior danno ai quadrati;
- anche il power-up, come i quadrati ha una certa vita e quando viene distrutto sarà possibile attivare la sua abilità;
- la generazione del power-up deve essere più frequente all'aumentare del tempo di gioco
- quando una pallina colpisce un blocco power-up non deve rimbalzarci, ma semplicemente attraversarlo, proseguendo la sua traiettoria e decrementando di uno la vita rimanente del power-up;
- il raggio laser, quando viene attivato, trapasserà tutti i quadrati, infliggendo loro un danno ogni volta che una sua parte ne è a contatto;
- ogni volta che un quadrato viene distrutto, il punteggio del giocatore viene incrementato di una unità;
- lo scopo del gioco è quello di accumulare quanti più punti possibili stabilendo di volta in volta nuovi record personali;
- realizzazione di un menù iniziale che permetta all'utente di sceglier la dimensione della finestra di gioco in base alla risoluzione del proprio schermo;
- fornire la possibilità all'utente di mettere il gioco in pausa durante la partita premendo i tasti "p", "esc" o "spazio". Nel caso in cui venga premuto "esc" deve comparire un menù che dia la possibilità di riiniziare il gioco, oppure di uscire dalla partita corrente e di consultare la classifica di tutti i punteggi globali;
- memorizzazione di tutti i risultati ottenuti da tutti gli utenti globali (dello stesso computer che hanno giocato almeno una partita);
- visualizzazione di una classifica con i primi dieci punteggi globali dei rispettivi utenti;
- visualizzazione del punteggio massimo e della media dei punteggi totali dell'utente corrente;

- realizzazione di una grafica user-friendly dell'applicazione.

1.2 Analisi e modello del dominio

Il dominio dell'applicazione è ambientato in una raffigurazione dello spazio nella quale l'attenzione sarà concentrata sulla missione di una navicella che dovrà sopravvivere il più a lungo possibile all'avvicinarsi dei nemici. La navicella utilizzerà proiettili come le palline, per cercare di abbattere quanti più avversari (quadrati) possibile. Inoltre, viene offerta la possibilità di potenziare la navicella con opportuni power-up, come ad esempio il laser, da utilizzare come proiettile. Questi power-up avranno un diverso effetto sui quadrati o sulla navicella; un laser potrà ad esempio abbattere più facilmente i nemici di una normale pallina. Fra le entità individuate dal dominio si riscontrano quindi una navicella (Shuttle), dei nemici (Square), dei proiettili (Bullet) ed eventuali potenziamenti (PowerUp). Si intuisce quindi che i quadrati dovranno essere in grado di sopportare un certo numero di colpi (hit points) prima di essere distrutti. Allo stesso modo la navicella verrà distrutta al contatto con un qualsiasi nemico. Una delle difficoltà di questa parte sarà coordinare l'input dell'utente col comportamento effettivo della navicella, ad esempio muovendola di conseguenza. In aggiunta, si dovrà gestire correttamente la fisicità delle entità coinvolte (rimbalzi, collisioni, ecc). Nello schema UML successivo si evidenziano le relazioni tra le entità individuate fin qua, con l'aggiunta di appositi generatori che si occuperanno di creare, al momento opportuno e in determinate posizioni, gli oggetti desiderati.


«interface»
Model
+luanch(): void
+getScore(): Score
+getSquareAgents(): List<SquareAgent>
+getBulletsAgents(): List<Bullet>
+getPowerUpBlocks(): List<PowerUpBlock>
+getGameStatus(): GameStatus
+setSpaceshipAngle(angle: double)
+setGameStatus(status: GameStatus): void
Score
+increment(): void
+reset(): void
+getPoints(): int
ModelImpl
«interface»
Shuttle
+getTop(): Point2D
+getImpactArea(): Area
+getActivePowerUps(): List<PowerUpExecution>
+activePowerUp(powerUpExecution: PowerUpExecution): void
+removePowerUp(powerUpExecution: PowerUpExecution): void
BulletGenerator
SquareGenerator
PowerUpGenerator
T extends FixedItem
«interface»
ItemGenerator
+launch(): void
+removeItem(T item): void
+addItemToList(T item): void
+getItems(): List<T>
SquareAgent
«interface»
PowerUpBlock
+use(): void
+getName(): String
+getSymbol(): String
LaserBlock
«interface»
Commands
+pause(): void
+resume(): void
+terminate(): void
«interface»
Hittable
+getHP(): int
+hit(): void
LaserAgent
«interface»
Bullet
BallAgent
«interface»
MovableItem
+getSpeed(): double
«interface»
FixedItem
+getModel(): Model
+getPos(): Point2D
+getAngle(): double
+getShape(): Shape
+getColor(): Color

Chapter 2

Design

2.1 Architettura

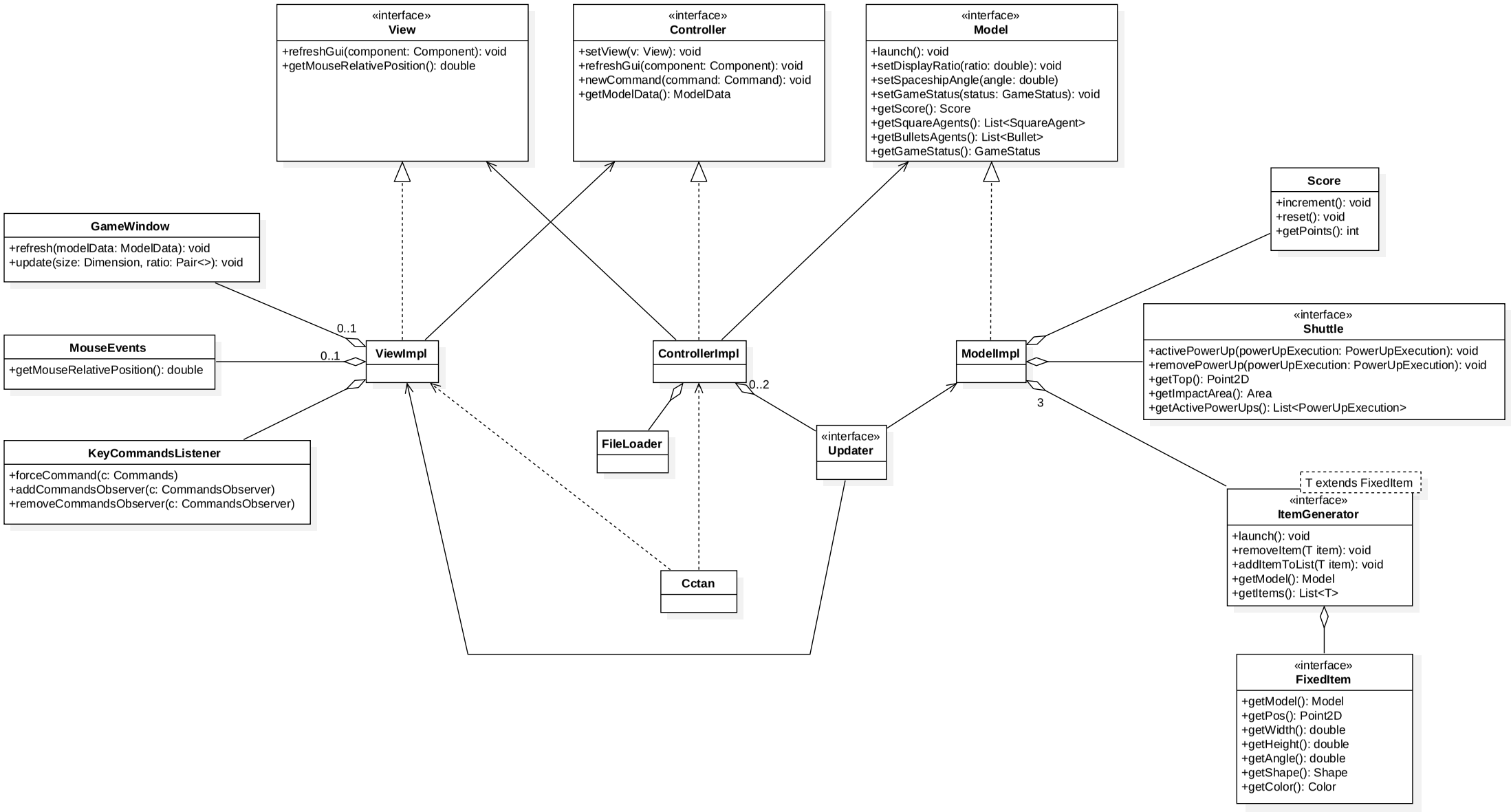
L'architettura dell'applicazione CC-Tan segue il pattern architetturale **MVC** (Model, View, Controller). Il sistema è stato suddiviso in componenti il più possibile isolati e indipendenti tra loro. Ognuno di questi deve poter essere intercambiabile con altre possibili implementazioni. La **View** si occupa di gestire le interazioni con l'utente e la parte di presentazione attraverso una finestra di gioco che mostri lo stato attuale del modello. Il **Model** ha il compito di gestire i dati e la logica del dominio dell'applicazione. Infine il **Controller** dovrà invece occuparsi di coordinare l'interazione tra View e Model ed eventualmente interagire col sistema operativo. Nello specifico, l'interazione tra Controller e View sarà relativa a:

- passaggio di risorse caricate dal Controller alla View;
- sincronizzazione dell'aggiornamento della View da parte del Controller;
- trasmissione dei comandi impartiti dall'utente dalla View verso il Controller (tramite pattrn *Observer*);
- informazione dello stato attuale della partita da Controller a View (running/end-game).

Mentre le interazioni tra Model e Controller sono relative a:

- richiesta da parte del Controller degli elementi da stampare a video;
- richiesta dello stato di gioco al Model;

Non ci sono interazioni dirette tra View e Model, le quali sono invece mediate dal Controller. Mentre, per quanto riguarda il compito specifico del Model avrà quello di gestire il movimento dei vari oggetti nell'area di gioco, comprese le eventuali collisioni o rimbalzi, attivazioni di power-up, generazione di nuovi elementi in determinate posizioni, ecc.
Nella pagina seguente è riportato uno schema UML che mette in evidenza le principali interazioni appena citate, con alcune delle interfacce e metodi più utilizzati.


«interface»
View
+refreshGui(component: Component): void
+getMouseRelativePosition(): double
«interface»
Controller
+setView(v: View): void
+refreshGui(component: Component): void
+newCommand(command: Command): void
+getModelData(): ModelData
«interface»
Model
+launch(): void
+setDisplayRatio(ratio: double): void
+setSpaceshipAngle(angle: double)
+setGameStatus(status: GameStatus): void
+getScore(): Score
+getSquareAgents(): List<SquareAgent>
+getBulletsAgents(): List<Bullet>
+getGameStatus(): GameStatus
Score
+increment(): void
+reset(): void
+getPoints(): int
GameWindow
+refresh(modelData: ModelData): void
+update(size: Dimension, ratio: Pair<>): void
0..1
MouseEvents
+getMouseRelativePosition(): double
0..1
ViewImpl
ControllerImpl
0..2
ModelImpl
3
«interface»
Shuttle
+activePowerUp(powerUpExecution: PowerUpExecution): void
+removePowerUp(powerUpExecution: PowerUpExecution): void
+getTop(): Point2D
+getImpactArea(): Area
+getActivePowerUps(): List<PowerUpExecution>
FileLoader
«interface»
Updater
KeyCommandsListener
+forceCommand(c: Commands)
+addCommandsObserver(c: CommandsObserver)
+removeCommandsObserver(c: CommandsObserver)
Cctan
T extends FixedItem
«interface»
ItemGenerator
+launch(): void
+removeItem(T item): void
+addItemToList(T item): void
+getModel(): Model
+getItems(): List<T>
«interface»
FixedItem
+getModel(): Model
+getPos(): Point2D
+getWidth(): double
+getHeight(): double
+getAngle(): double
+getShape(): Shape
+getColor(): Color

2.2 Design dettagliato

Prima di addentrarci nelle specifiche sottoparti spendiamo due parole su come è stata effettuata la suddivisione in package.
Come detto nella sezione precedente, poiché si è utilizzato il pattern MVC abbiamo cercato di suddividere il progetto in package che rispettassero questo pattern architetturale. Quindi è stato creato un package per ogni componente, vale a dire uno per il Model, uno per il Controller e uno per la View. Da questa macro suddivisione ne sono state svolte delle altre che si addentrano man mano in un concetto specifico di un componente. Ad esempio, per quanto riguarda il Model, oltre il suo package principale, ne sono stati creati altri due più specifici in cui sono racchiuse delle classi di geometria ed un altro che contiene i generatori. Nella View, invece, si era pensato inizialmente di creare un sotto-package contenente le classi di menù e uno con le classi per disegnare, mentre in un secondo momento non è stato fatto per non dover rendere pubbliche alcune classi. Mentre per l'*interpackage_comunication*, essendo di natura pubblico, è stato suddiviso in sotto-package contenenti il primo tutte le interfacce e le implementazioni dei *CommandsObserver*, il secondo quelle del *SizeObserver* e l'ultimo con le classi di "dati". Il Controller invece non è stato scomposto in ulteriori sotto-package.

Passiamo dunque ora ad analizzare singolarmente le varie parti rilevanti di ciascun membro del team.

Gianmarco Magnani - Model

Vediamo di seguito alcune parti salienti della mia sottoparte in cui sono stati impiegati i pattern di progettazione. Partiamo dalla classe *MovableItemImpl*, che rappresenta gli oggetti in movimento all'interno dell'area di gioco. Poiché a seconda dell'oggetto in esame potrebbero cambiare la sua velocità e le sue condizioni di non esistenza, sono stati creati i due metodi protetti astratti applyConstraints() e getDefaultSpeed(). Ad esempio, nel caso delle palline, esse non dovranno più esistere non appena escono dai margini della mappa, mentre un quadrato termina la sua esecuzione all'impatto con la navicella, provocando inoltre la fine del gioco. Si è quindi realizzato il **template method** *updatePos()* al cui interno viene richiamato il metodo applyConstraints(), per accertarsi che ad ogni spostamento siano soddisfatte le condizioni di esistenza. Mentre per quanto riguarda getDefaultSpeed(), esso viene richiamato direttamente all'interno del costruttore per inizializzare la velocità iniziale dell'oggetto, nel caso in cui non ve ne sia specificata

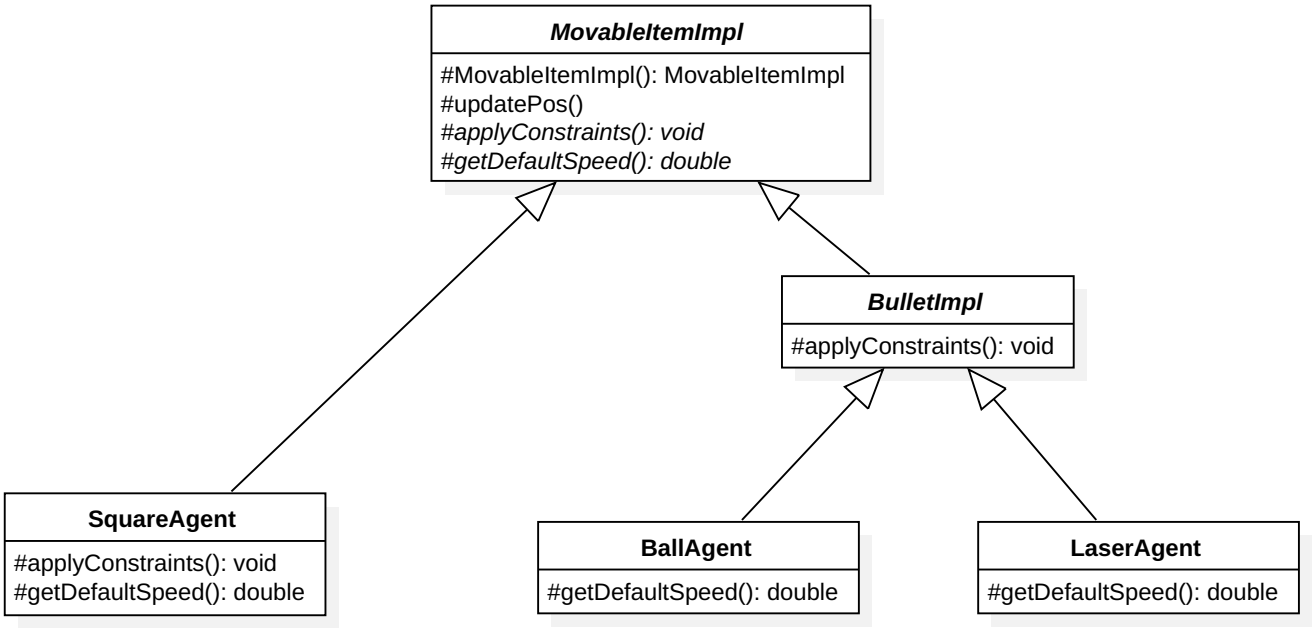


Figure 2.1: Template method in MovableItemImpl.

una. Dalla classe astratta *MovableItemImpl* è stata creata un'altra sottoclasse anch'essa astratta, *BulletImpl*, che implementa già il metodo applyConstraints(), uguale per ogni tipo di proiettile (quindi palline comprese), mentre il restante getDefaultSpeed() sarà specializzato dalle classi concrete *BallAgent* e *LaserAgent*. D'altra parte, *SquareAgent* implementerà direttamente i due metodi di cui sopra. Uno schema UML che rappresenta quanto appena detto è rappresentato dalla figura 2.1.

Passiamo ora alla classe astratta *HittableImpl*, che gestisce le operazioni eseguibili su oggetti colpibili. In questo caso l'azione da intraprendere a seguito della distruzione dell'oggetto potrebbe cambiare da oggetto a oggetto: nel caso di un quadrato, non appena i suoi punti vita scendono a 0, cioè viene distrutto, dovrà semplicemente scomparire dall'area di gioco ed essere rimosso dalla lista dei quadrati attivi, incrementando inoltre di una unità il punteggio del giocatore. Mentre nel caso dei blocchi-powerup, quando vengono distrutti dovranno attivare il loro potere, come ad esempio il laser proiettile. Dunque troviamo in *HittableImpl* tutti i metodi in comune (come hit(), getHP()), mentre viene lasciato astratto il metodo destroyed(), richiamato all'interno del **metodo template** *hit()* non appena i punti vita dell'oggetto in considerazione scendono a 0. In questo modo l'implementazione sarà demandata

alle sottoclassi, che la realizzeranno nel modo a loro consono. Una nota rispetto a quanto osservato in precedenza: poiché come visto *SquareAgent* estende già dalla classe astratta *MovableItemImpl* e *PowerUpBlockImpl* da *FixedItemImpl* (non mostrate qui), non è possibile un'estensione diretta anche da *HittableImpl*. Per tale ragione si è scelto di realizzare una aggregazione, dove le due classi avranno un oggetto di tipo *HittableImpl*, costruito con una *implementazione anonima* nella quale si definisce l'unico metodo destroyed(). In questo modo, l'implementazione dei metodi hit() e getHP() sarà semplicemente una delegazione all'istanza di *HittableImpl*, ottenendo così un riuso della suddetta classe. La figura 2.2 seguente rappresenta quanto appena detto.

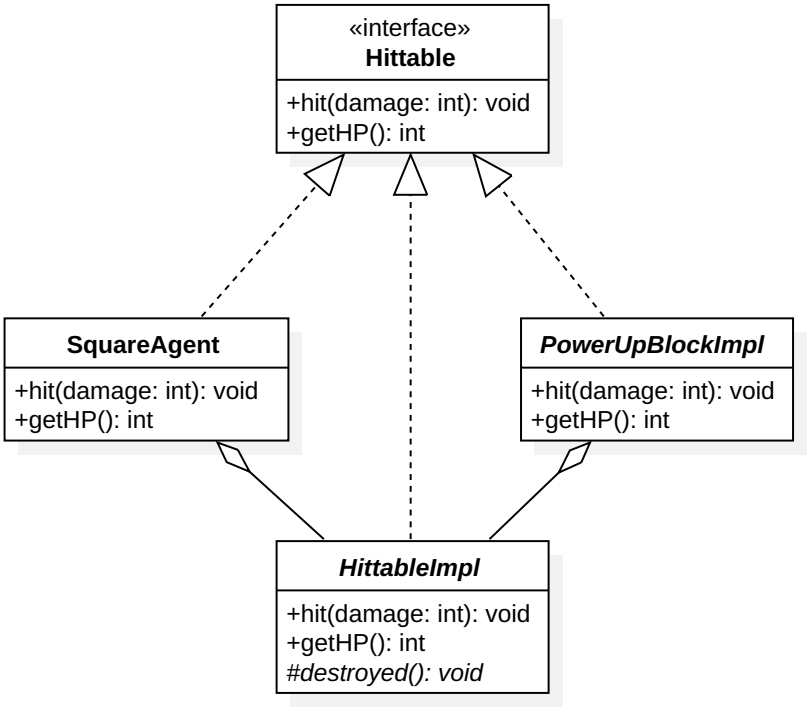


Figure 2.2: Template method in HittableImpl.

Vediamo di seguito la classe astratta *BulletImpl*, che implementa il comportamento di un generico proiettile, in particolare si occupa di controllare se questo impatta contro qualche oggetto colpibile e in tal caso richiama il suo metodo hit(). È stato realizzato a tal fine il **metodo template** *checkIntersecate()* che, nel caso si verifichi una collisione con un quadrato, richiama il metodo astratto updateAngle(), per consentire ad ogni proiettile

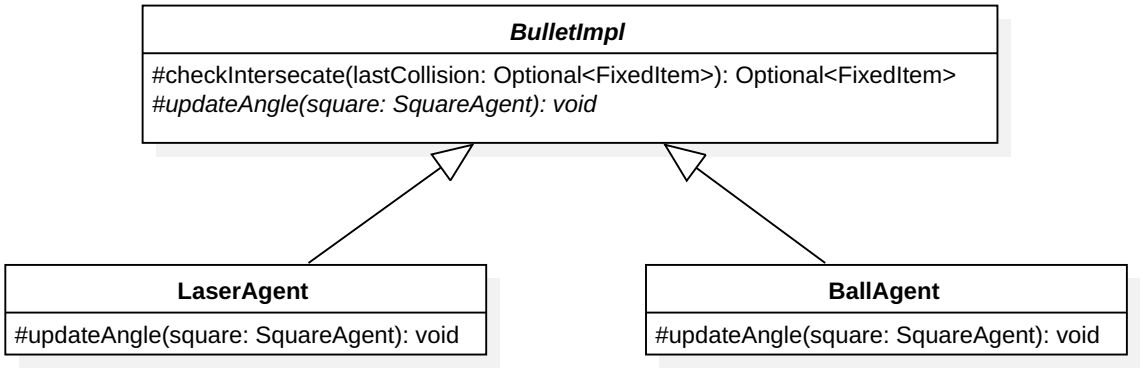


Figure 2.3: Template method in BulletImpl.

specifico di modificare la sua traiettoria in maniera differente a seguito di un impatto con esso. Infatti, come notiamo, i due tipi specifici di proiettile, *BallAgent* e *LaserAgent*, estendono da questa classe ed implementano in modo differente il metodo astratto: il primo calcola il nuovo angolo in modo da realizzare un rimbalzo, mentre il secondo non effettua cambiamenti, lasciando quindi che il laser attraversi liberamente il quadrato. Per l'effettiva implementazione di updateAngle() è possibile consultare i sorgenti, dove si noterà proprio che quella concreta in *LaserAgent* non conterrà nessun tipo di operazione, a differenza di quella in *BallAgent*. Si rimanda alla figura 2.3 per una rappresentazione grafica del concetto.

Infine, dato che si hanno a disposizione diversi parametri da settare per ogni tipo di oggetto, si è scelto di realizzare il **pattern Builder** per la creazione delle diverse entità presenti nell'area di gioco. In questo modo si evita di avere tanti costruttori differenti per ogni esigenza, lasciando la possibilità di creare oggetti personalizzati, purché siano specificati quanto meno il model e la posizione di partenza. In questo caso, sono stati realizzati due builder astratti: uno per oggetti fissi (che quindi non si spostano), *AbstractBuilderFI*, e uno per oggetti in movimento, *AbstractBuilderMI*. Quest'ultimo, così come *MovableItemImpl* estende da *FixedItemImpl*, estenderà da *AbstractBuilderFI*, ereditando così tutti i metodi del builder di partenza. A questo punto, i builder concreti potranno scegliere di estendere da uno di questi due astratti, a seconda delle esigenze chiaramente, ricordandosi di implementare il metodo astratto build(), che alla fine produrrà l'oggetto vero e proprio con i parametri richiesti. Ovviamente sarà possibile aggiungere anche altri metodi all'interno del builder concreto per impostare eventuali campi specifici di un oggetto, non presenti in nessuno dei due astratti. È il caso ad esempio di

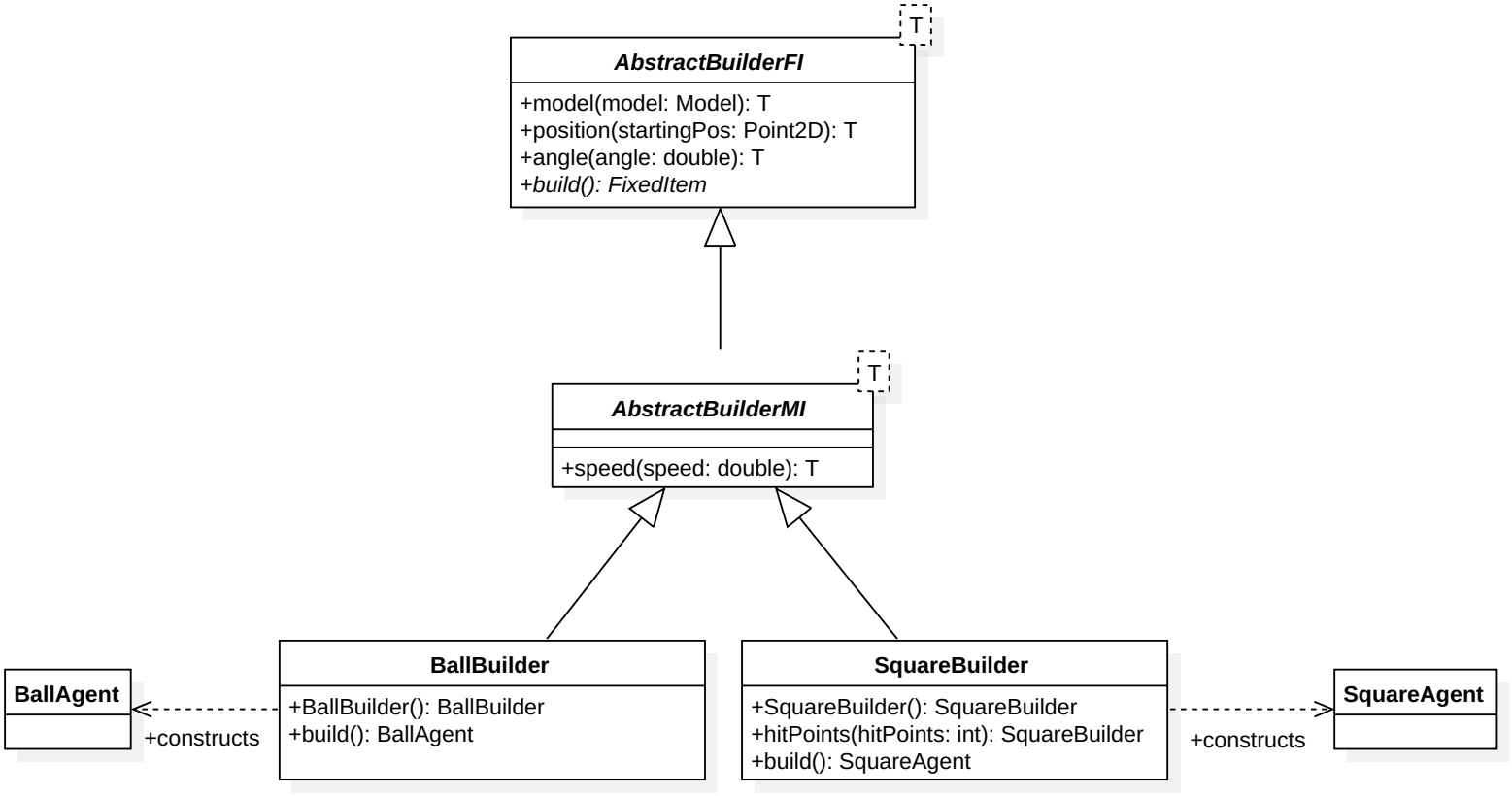


Figure 2.4: Pattern builder per costruire nuovi oggetti di vario tipo.

SquareBuilder, che aggiunge il metodo hitPoints() per specificare i punti vita iniziali del quadrato. Viceversa, *BallBuilder* si limiterà solamente a implementare il metodo build(). L'utilizzo dei generici sarà discusso in seguito, nella sezione 3.3 - Gianmarco Magnani. Uno schema UML riassuntivo di quanto detto è rappresentato nella figura 2.4.

Spendiamo ora qualche parola sulle eventuali estensioni future possibili grazie ad una tale implementazione. L'utilizzo del pattern **template method** agevola il riuso delle classi astratte che ne fanno uso, lasciando che le sottoclassi implementino le funzioni astratte nel modo a loro consono, modificando quindi in parte il comportamento dell'oggetto. In questo modo tutte le azioni in comune possono continuare ad essere utilizzate senza la necessità di re-implementarle, mentre quelle che differiscono verranno realizzate mediante una ridefinizione dei metodi astratti. Consideriamo ad esempio il template method **updatePos()** visto sopra, in cui viene richiamato al suo interno il metodo astratto *applyConstraints()*. Supponiamo ora che vogliamo

creare un oggetto in movimento che dopo un certo numero di refresh scompaia dall'area di gioco. Per fare questo sarà sufficiente aggiungere un campo all'interno della sottoclasse che sarà settato inizialmente col numero di refresh che può supportare, dopo i quali esso sarà eliminato. A questo punto in applyConstraint() si andrà a verificare, dopo aver decrementato la variabile, se il suo valore sia ancora maggiore di 0; in tal caso l'oggetto continuerà ad esistere, altrimenti sarà rimosso. In questo modo abbiamo potuto utilizzare interamente la classe astratta *MovableItemImpl* adeguando semplicemente il metodo *applyConstraints()* e inserendo un campo.

Un discorso simile lo si può fare anche nel caso del secondo template method: **hit()**, con all'interno il metodo astratto *destroyed()*. A seconda dell'oggetto in esame possiamo volere eseguire azioni differenti quando esso viene distrutto. Un caso di questo tipo lo troviamo proprio già nell'applicazione: quando un quadrato viene distrutto, come già detto, dovrà semplicemente scomparire ed incrementare il punteggio di uno, mentre nel caso del blocco del power-up, alla sua distruzione dovrà essere attivata la sua abilità. Si può dunque implementare il metodo destroyed() nei modi più svariati, per eseguire azioni differenti a seguito della distruzione dell'oggetto colpibile (Hittable).

Passiamo ora invece ad un'altra classe: *BulletImpl*, che implementa l'interfaccia *Bullet*. La parte sulla quale ci si vuole concentrare è il vantaggio fornito da questa classe e i possibili riusi. La navicella (Shuttle nel gioco) non sparerà per forza delle palline (*BallAgent*), ma dei proiettili generici, *Bullet* per l'appunto. In questo modo è possibile cambiare il tipo effettivo di proiettile che la navicella sparerà molto semplicemente andando ad utilizzare una qualsiasi classe che implementi *Bullet* (o estenda da *BulletImpl*). Infatti, come si può notare dai sorgenti, il tipo di proiettile sparato dalla navicella potrà mutare più volte nel corso del tempo, ad esempio per l'attivazione di un power-up, grazie all'astrazione appena descritta.
Inoltre, grazie al template method **checkIntersecate()** di BulletImpl che richiama al suo interno il metodo astratto *updateAngle()*, è possibile specificare un comportamento differente da proiettile a proiettile al momento dell'impatto con un quadrato. Ad esempio, un proiettile laser quando impatta con un quadrato continuerà regolarmente lungo la sua traiettoria senza subire variazioni, mentre una pallina dovrà rimbalzare e quindi ricalcolare adeguatamente l'angolo.

Nicolas Pasolini - Model

Uno degli obiettivi principali dell'applicazione che è stato portato a termine è quello di generare dinamicamente nel tempo le palline e i quadrati con una determinata frequenza. Sin dall'inizio dell'analisi del problema è stata presa in considerazione la possibilità e la volontà di far variare nel tempo sia la velocità di spostamento dei singoli oggetti, sia la frequenza con la quale i diversi oggetti vengono generati.
La prima scelta, è stata una scelta a livello concettuale. Inizialmente non si è data importanza alle singole palline e ai singoli quadrati, ma si è pensato di astrarre questi due concetti nel significato più generale che può assumere un oggetto. A tal proposito è stata creata l'interfaccia *ItemGenerator*, contenente tutti i metodi necessari per aggiungere un qualsiasi tipo di nuovo oggetto all'interno dell'applicazione. Dato che al suo interno, in futuro, potrebbe presentarsi la necessità di voler generare diversi tipi di oggetti, è stato utilizzato il pattern **Template Method** per massimizzare il riuso, come da Figura 2.5. La classe astratta *AbstractItemGenerator* è un thread, in quanto ogni volta, allo scadere di un determinato lasso di tempo, deve generare un nuovo oggetto avente specifiche caratteristiche. Il template method è il metodo *run()*, che al suo interno chiama un metodo astratto e protetto *createNewItem()*.

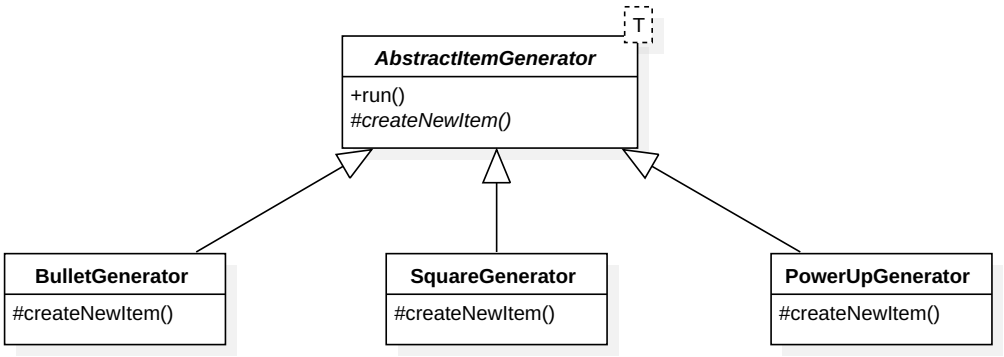


Figure 2.5: Pattern template method in AbstractItemGenerator.

Per quanto riguarda la variazione nel tempo della frequenza di generazione e la velocità di spostamento dei singoli oggetti, anche in questo caso è stato utilizzato il pattern **Template Method** come da Figura 2.6. La classe astratta *AbstractTimerRatio* è anch'essa un thread, in quanto allo scadere di ogni minuto si occupa di aumentare sia la frequenza di generazione, sia la

velocità dell'oggetto. Il template method è il metodo *run()*, che al suo interno chiama un metodo astratto e protetto *operationRatio()*. Questa scelta è stata fatta pensando che in futuro potrebbero esserci nuovi oggetti con differenti caratteristiche che devono variare nel tempo. In questo modo si riuscirà a riutilizzare comodamente il codice.

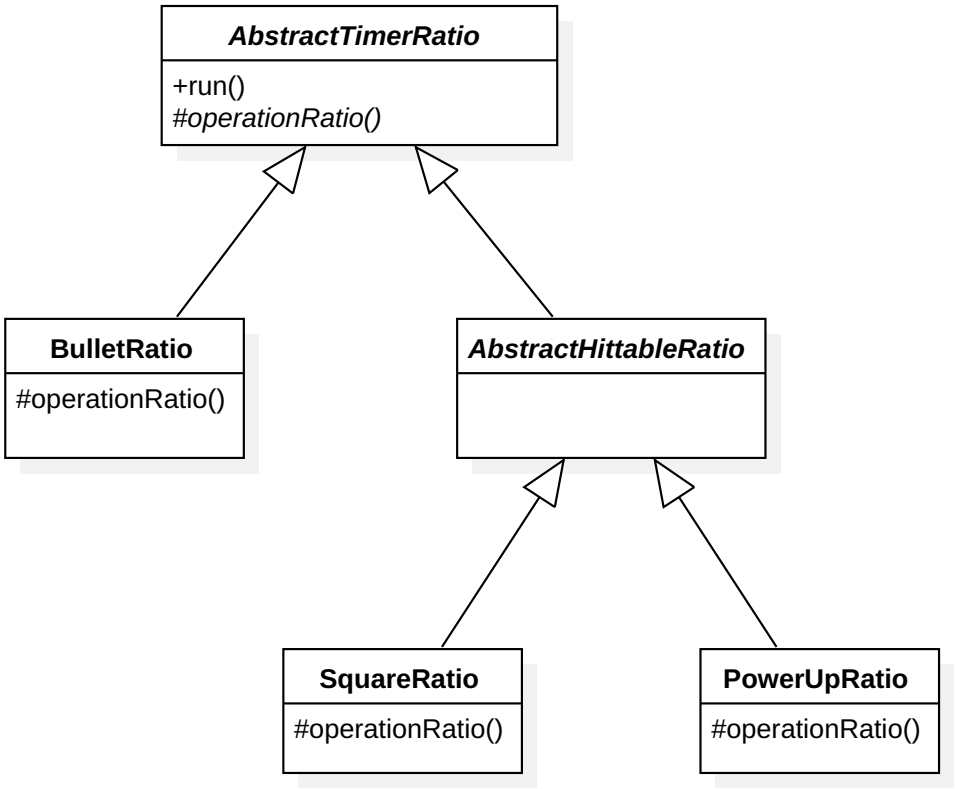


Figure 2.6: Pattern template method in AbstractTimerRatio.

Tra le due classi astratte sopra descritte si ha una stretta iterazione in quanto, come possiamo notare dalla Figura 2.7, l'oggetto *AbstractItemGenerator* è composto da un oggetto di tipo *AbstractTimerRatio*, dal quale riuscirà a reperire a runtime tutti i valori che gli servono.

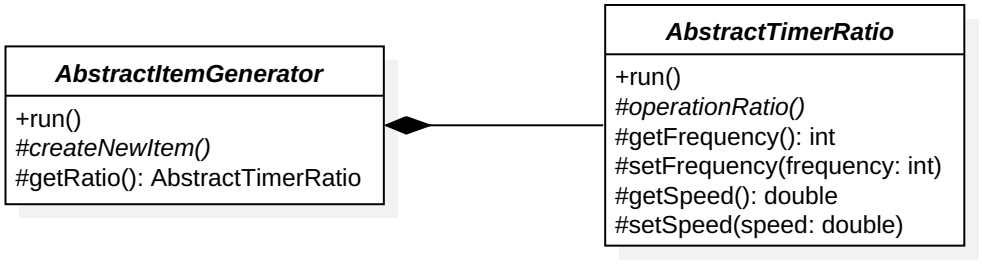


Figure 2.7: Composizione in AbstractItemGenerator.

Il pregio di aver utilizzato questo pattern di progettazione lo si è notato anche durante lo sviluppo stesso dell'applicazione. Durante l'implementazione opzionale di un potenziamento (laser) in cui c'era la necessità di generare alcuni Power-Up, grazie a questo pattern siamo riusciti a svolgere l' implementazione in breve tempo, riutilizzando il precedente codice. Tutte queste classi che si occupano di generare dinamicamente nel tempo i diversi tipi di oggetti e tutte quelle che estendono da queste ultime, sono contenute all'interno del package "*it.unibo.oop.cctan.model.generator*".

Inoltre, è stato utilizzato il pattern **Singleton** nella classe *Score*, per tenere traccia del punteggio di gioco. La scelta è ricaduta su questo pattern in quanto ci garantisce l'univocità all'interno dell'applicazione di quel tipo di oggetto. È stata scelta anche un'implementazione thread-safe perché il nostro è un ambiente multi-threading. Nel caso in cui non avessimo usato questa implementazione, alcuni thread all'inizio, avrebbero potuto creare contemporaneamente più istanze di quell'oggetto.

Lorenzo Sutera (View + Controller)

La GUI è organizzata in finestre che permettono di eseguire varie azioni e aprirne delle altre. Le finestre si adattano alla risoluzione del monitor del computer in utilizzo, quindi in questo caso il problema è quello di ridimensionare le finestre. La soluzione a tale problema è quella di realizzare un metodo che calcola, in base ai componenti utilizzati, qual è la dimensione ottimale per la finestra (il metodo pack() non è utilizzabile nel nostro caso dato che l'immagine per il background ha dimensioni proprie, e si adatterebbe a queste).
Inoltre la GUI deve poter notificare gli altri componenti dell'applicazione

sull'avvenuta conferma di certe scelte nella sezione impostazioni, per questo si è deciso di utilizzare il **pattern Observer**.

I punteggi vengono salvati in un file che rimarrà memorizzato all'interno del computer in oggetto. Una funzionalità della GUI è mostrare i migliori punteggi tramite una tabella, incluse alcune statistiche personali, per fare ciò, si utilizza la classe Records che estrae, inserisce, elabora e riordina i punteggi, e mostra quindi i migliori nella tabella.
È stata inserita la possibilità di avere una traccia audio come sottofondo durante il gioco, che è possibile stoppare e riprendere a piacimento.

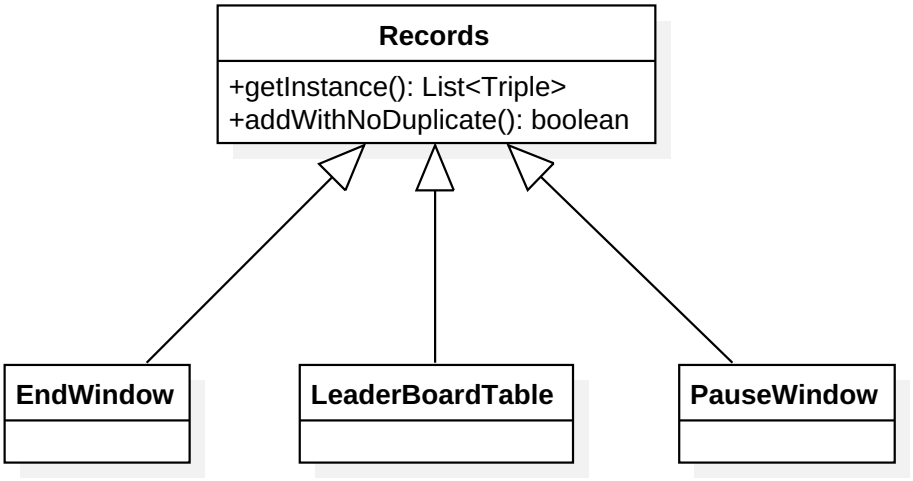
Per identificare le fasi del gioco sono state create delle enum (*GameStatus*), le quali sono utilizzate per indicare in che stato è attualmente il gioco, e permettere o meno certe azioni. Per avvisare gli observer di che azione (comando) il giocatore ha scelto di eseguire, viene utilizzate una enum: *Commands*. Alla ricezione i componenti observer agiranno di conseguenza.

Alla schermata di gioco è stato aggiunto un *KeyListener* che permette al giocatore di utilizzare i tasti della tastiera per mettere il gioco in pausa o accedere ad un menù.

Per far si che le classi del MVC conoscano quali impostazioni sono state scelte dall'utente tramite la GUI, è stato implementato un **pattern Observer** sia per il dato di tipo *size* (Dimension) che per il *ratio* dello schermo. In questa maniera, alla conferma della selezione di queste impostazioni, le classi interessate, cioè gli *observer*, verranno aggiornati dal gestore degli stessi. Esso possiede una lista di tutti gli oggetti interessati al dato, e li avvisa non appena avviene un cambiamento. Le classi interessate si possono aggiungere a questa lista. Questa soluzione ha permesso alla View di comunicare con il Model in modo conforme al pattern MVC e permette facili cambiamenti futuri.

È stato utilizzato il **pattern Observer** anche per i *Commands*, i quali indicano al Model in che modo deve agire basandosi su cosa il giocatore sceglie di fare.

Per la classe *Records* è stato utilizzato il **pattern Singleton**, in questo modo chiunque richieda l'istanza della classe ottiene comunque la stessa. Siccome *Records* interagisce con il file Scores, si è notato che applicando il pattern singleton, il numero di accessi al file è ridotto. Inoltre utilizzare un singleton, permetterà in futuro cambiamenti tali che Records lavori in modo concorrente e therad safe senza troppe complicazioni.



Paolo Baldini (View + Controller)

In questa sezione verranno analizzate le metodologie scelte per lo svolgimento dei compiti di gestione della componente grafica del gioco, di caricamento risorse e di comunicazione fra i vari package del progetto. La prima parte che andremo ad affrontare sarà relativa alla "stampa a video" degli elementi del dominio applicativo (astronave, nemici, etc.), nella view. Non ci occuperemo subito di come questi elementi giungano alla view, ma per il momento ci limiteremo a descrivere come questi vengono utilizzati per la stampa. Occorre subito notare che durante la strutturazione di questa porzione di codice non si è voluto sviluppare la soluzione per rispondere alle sole richieste del dominio, ma si è invece cercato di generare un codice che potesse rispondere alle richieste di una più generica stampa cartesiana in due dimensioni. Questo fa in modo che la parte di codice dedicata alla graficazione astragga dai concetti di nemici, navicella etc. per focalizzarsi su una graficazione di "figure" generiche, che possono acquisire quindi qualsivoglia forma. Il vantaggio di questa strutturazione è risultato evidente all'introduzione dei PowerUp nell'applicazione, quando non è stato necessario modificare nessuna parte di codice relativa alla stampa per adattare i nuovi cambiamenti. La filosofia alla base di questa struttura è: "non c'è differenza fra nemico, astronave o altro, tutto è uguale, tutto è una figura". Si ritiene che questa scelta logica abbia favorito e favorisca notevolmente il riuso e l'estendibilità dell'applicazione. Tornando alla descrizione del blocco di classi che si occupano della stampa, possiamo notare che questo è gestito da una classe chiamata *GameWindow*, che utilizza, per svolgere la sua funzione, l'insieme

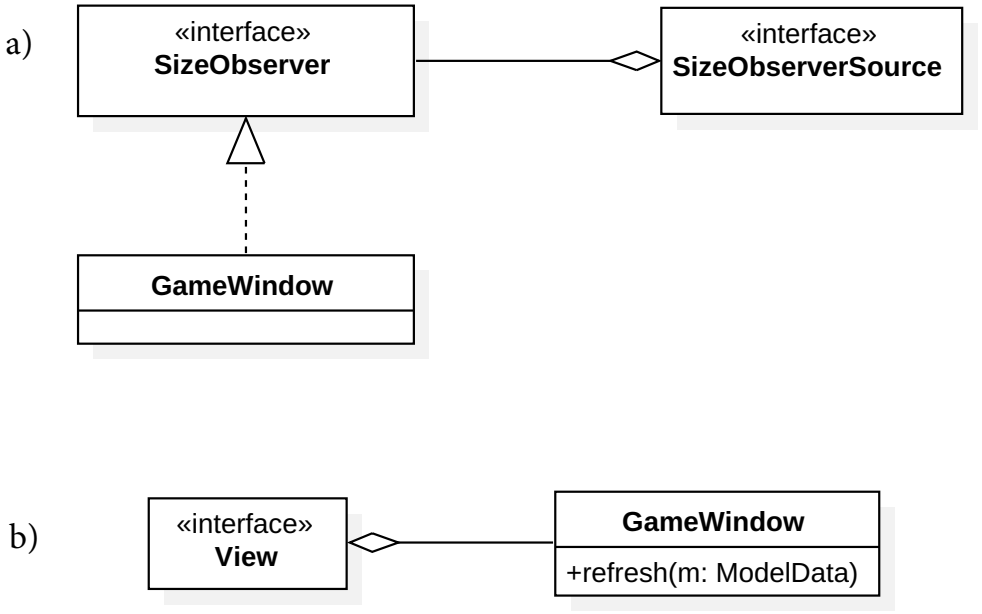


Figure 2.8: a) La classe *GameWindow* "osserva" variazioni di dimensione; b) la View contiene una GameWindow sulla quale chiamerà refresh

dei dati graficabili provenienti (ancora non sappiamo come) dal model, e passati tramite il metodo refresh(). Questa classe non svolge propriamente la funzione di disegno su schermo, ma invece si occupa di ricevere gli ordini di update e le informazioni su possibili cambiamenti nelle impostazioni di gioco (dimensione della finestra) attraverso il **pattern Observer**, notificandolo successivamente alle sue due sotto-classi che si occupano invece della graficazione vera e propria. L'interazione di queste sotto-classi è logicamente semplice: una, *GraphicPanel*, decide la strategia di disegno, l'altra, *Drawer*, lo effettua in modo vero e proprio con tutte le attenzioni del caso. Questa implementazione include quindi il pattern Observer.

La seconda parte di questa analisi è invece relativa al caricamento delle risorse. Questo avviene tramite una classe presente nel controller (*FileLoader*) che si occupa della creazione delle directory nel pc dell'utente e del caricamento o conversione dei file utilizzati dall'applicazione. Per questa classe di load è stata scelta un'implementazione basata sul threading; questa scelta è stata fatta per permettere, in una possibile implementazione futura, di poter lanciare simultaneamente varie classi di load che, in un sistema che supporti il multi-threading, permetterebbe un caricamento più rapido dei file, essendo questo fatto in contemporanea (N.B. con caricamento si intendono anche

possibili conversioni di file, che potrebbero essere anche molto onerose in termini di tempo e che beneficerebbero dell'esecuzione contemporanea). Questi file vengono poi caricati in una classe con la funzione di "store". Tenendo presente che si è deciso di tenere in considerazione la possibilità di lancio di più loader contemporaneamente (non presente nell'implementazione attuale, tuttavia), sarebbe scomodo e complicato tenere in memoria varie versioni di queste classi di store e di volta in volta cercare quella corretta in cui siano contenuti i file (che potrebbero oltretutto essere outdated rispetto a quelli contenuti in un altra classe). La soluzione a questo problema è stata trovata nell'utilizzo del **pattern Singleton**. Verrà quindi creata una sola istanza di questa classe, a cui tutti i loader potranno accedere e che conterrà quindi tutti i file caricati. Questa soluzione si rivela inoltre molto comoda, non dovendosi trasportare un riferimento all'oggetto fra le varie classi. Per la natura multi-threading dell'architettura studiata però, bisogna garantire una creazione del singleton thread-safe; questa è ottenuta con la creazione tramite il pattern **Lazy Initialization**, come visto a lezione, nella sua versione thread-safe. Il Singleton è ora creato, ma una classe che provi ad accedere ai dati ivi contenuti potrebbe trovarli non ancora caricati. La soluzione a questo evento è stata trovata rendendo il Singleton (che è poi la classe *LoadedFilesSingleton*) allo stesso tempo un observable (nel codice indicato col suffisso -ObserverSource). Al caricamento di un nuovo dato, questa classe di store notifica quindi a tutti coloro che sono in attesa, che un nuovo file è stato caricato. Quest'implementazione include quindi ben tre pattern: l'Observer, il Singleton e il Lazy Initialization.

Come terzo approfondimento analizzeremo l'aggiornamento di view e model e il passaggio dei dati mappabili da l'uno all'altro. Nel rispetto del pattern MVC, il model non ha nessuna interazione attiva con controller e view, il passaggio dei dati deve quindi essere gestito esternamente. Per questo compito sono state create nel controller due apposite classi. Essendo queste molto simili, è stato deciso di creare una classe scheletro astratta e utilizzare il pattern template method, che lavora su di un metodo astratto e protetto exec(). La differenza sostanziale fra le due sono infatti le operazioni che svolgono, mentre il comportamento generale è il medesimo. Una di queste si occuperà di prendere dati dal model e notificare alla view di aggiornarsi, l'altra viceversa prenderà dati (eg: angolo navicella) dalla view e li notificherà al model. Queste classi, dovendosi occupare di refreshare la view e di informare e ricevere periodicamente dati dal model, dovranno essere implementate quindi come thread. Questa scelta implica però che queste classi debbano essere sensibili ai comandi di gioco quali la pause, lo start, il resume e l'end game. Per fare ciò, si è utilizzato nuovamente il pattern observer, questa volta ap-

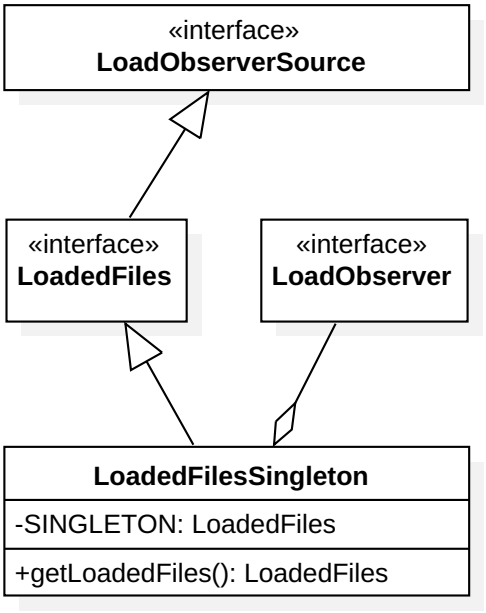


Figure 2.9: *LoadedFilesSingleton* è un "*Observable*" e un "Initialization-on-demand holder idiom"

plicato all'osservazione dei comandi di gioco. Il *ViewUpdater*, oltre a inviare il refreshGui(), ha anche il compito di controllare che lo stato di gioco nel model non sia ended (si è persa la partita), nel qual caso dovrà informare il gestore dei comandi dell'evento straordinario. Non inizialmente previsto, durante lo sviluppo dell'applicativo è stato riscontrato, a causa di una mancata comprensione, una discrepanza fra i metodi di ritorno presenti nel model e gli accordi presi inizialmente. A seguito di ciò è stato aggiunto al package controller, e nello specifico a una di queste due classi (ViewUpdater), una classe basata sul pattern Adapter, con lo scopo di risolvere le discrepanze nei dati ritornati. In queste classi si riscontrano quindi altri tre pattern: Observer, Template Method e Adapter.

Un punto interessante e non ancora affrontato nella trattazione, è l'utilizzo di un pattern (o una sua sintesi) non presente fra quelli spiegati a lezione: il "**Chain Of Responsibility**". Questo pattern è stato inizialmente usato per una facilitazione di utilizzo del pattern Observer, andando a richiamare a ogni classe della "catena" un metodo per ottenere la sorgente delle osservazioni, fino a trovarne una che effettivamente lo contenga. Abbiamo scritto "una sua sintesi" perchè questa è effettivamente una leggera variazione di questo pattern. Se nel pattern originale ogni "anello" della catena ha un suo

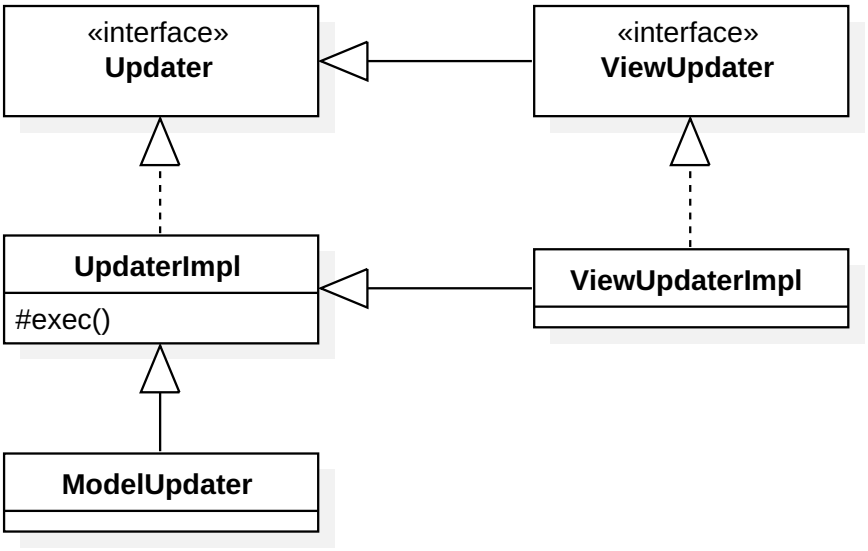


Figure 2.10: *ModelUpdater* e *ViewUpdater* implementano il pattern Template Method.

"range di responsbilità/influenza", qui la responsabilità è affidata alle sole classi che contengono il sorgente dell'observer. Si potrebbe dire che in questa implementazione chiunque contenga un riferimento al sorgente abbia la "responsabilità", mentre tutti gli altri non l'abbiano. Questo pattern ha perso poi la sua utilità a causa di una piccola ristrutturazione del codice, che ha reso l'unico "anello" della catena la classe ViewImpl. E' stato mantenuto per l'utilità in una possibile estensione del progetto e perchè complessivamente non appesantiva di molto il codice, venendo comunque utilizzato, anche se in una sola classe.

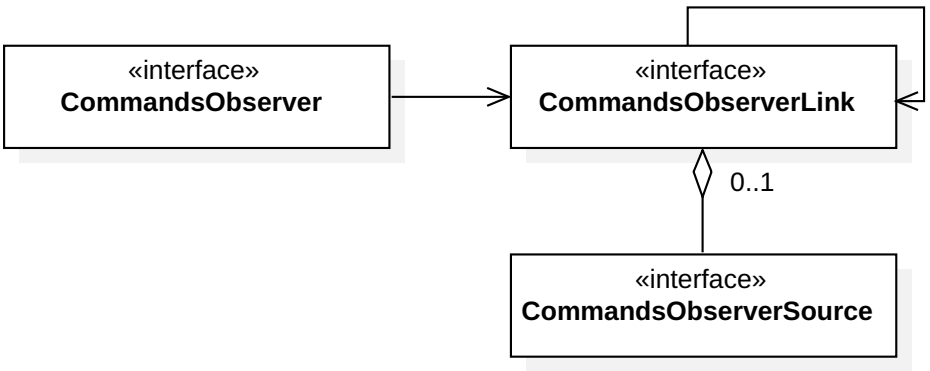


Figure 2.11: La struttura della nostra variante della "chain of responsibility".

Chapter 3

Sviluppo

3.1 Testing automatizzato

Di seguito si elencano i componenti sottoposti a test automatici e successivamente saranno fornite alcune indicazioni a riguardo.

- Test di risposta ai comandi:
  - *ViewUpdater*
  - *ModelUpdater*
  - *KeyCommandsListener*
- Creazione di directory e conversione di file .svg:
  - *FileLoader*
- Test di "stampa" a video (simulazione casistiche di gioco):
  - *GameWindow* (e conseguentemente *GraphicPanel* e *Drawer*)
- Test avanzamento barra di caricamento:
  - *Loader*
- Test calcolo posizione angolare del mouse:
  - *MouseEvents*
- Test corretta rotazione della navicella:
  - *ShuttleImpl*

Mentre non sono stati effettuati test manuali di rilievo. L'unico riscontro è un implicito test della componente grafica su sistemi operativi diversi (Windows 10, OSX, Linux Mint) a seguito di visualizzazioni non corrette su alcuni sistemi.

Durante la creazione del progetto sono stati strutturati vari test per il controllo di correttezza del codice. Vediamo ora più approfonditamente quanto riassunto sopra.

- Test di risposta ai comandi: si verifica il comportamento delle classi thread ViewUpdaterImpl e ModelUpdaterImpl in presenza di comandi di gioco (PAUSE, START, RESUME, END). Il test *KeyListenerJTest* verifica che alla pressione di specifici tasti, l'applicazione reagisca in modo corretto.

  Per rendere il test automatizzato, dato che l'applicazione è pensata per interagire con l'utente, è stata utilizzata la classe *Robot* che simula sia la pressione dei tasti dalla tastiera che il movimento e il click del mouse.

- Creazione di directory e conversione di file .svg: si controlla che le directory vengano effettivamente create dal FileLoader e che questo converta efficacemente i file .svg in jpg. A onor del vero il test sulla conversione è di difficile attuazione, per cui si è optato per un approccio più semplice di check di esistenza e lettura del file.

- Test di stampa a video: un test riguardante la GameWindow e le sue sotto-classi GraphicPanel e Drawer. Questo test riproduce vari scenari di funzionamento, dai più generali e comuni ai casi limite. Purtroppo, essendo un test per una parte grafica ed avendo questa molti metodi privati, non è stato possibile effettuare un vero test con controllo di ritorno. Il test infatti, nonostante proponga casi limite, si basa sul disegno di questi su schermo, ed è quindi necessario un minimo spirito critico per capire se il codice è strutturato correttamente. Eccetto per il controllo visivo, non è però richiesto nessun controllo manuale durante il test.

- Test barra progressione loading dati: come per il test sulla GameWindow, anche questo è un test grafico. E' un test basilare e durante l'applicazione si vedrà semplicemente la barra di caricamento progredire. In realtà però, questo test controlla anche il funzionamento del pattern observer applicato al loader, dovendo questo far avanzare la barra solo ad un nuovo input passato da un fittizio LoadedFile.

- Test di calcolo della posizione angolare del mouse rispetto alla finestra:

l'intento è di verificare che l'angolo di inclinazione del mouse calcolato rispetto alla finestra (per l'inclinazione della navicella) sia corretto. A differenza dei due test precedenti, basa la sua riuscita solo ed unicamente su test JUnit. Il lato negativo di questo test è che, come per quello sulla GameWindow, la classe contiene metodi privati, rendendo il test tramite spostamento fisico (AUTOMATICO) del mouse, l'unico metodo per provarlo. A questo scopo occorre una precisazione: nonostante il test sia completamente automatizzato, è richiesto che l'utente NON muova il mouse durante il test. Se il mouse dovesse venire mosso c'è caso che questo interferisca con la riuscita o meno.

- Test corretta rotazione della navicella: in questo test viene verificato che lo Shuttle (navicella) si posizioni correttamente a seguito di una sua rotazione. In particolare, poiché i 3 punti vengono calcolati da un "algoritmo" implementato basandosi sulle formule trigonometriche, si verifica che effettivamente il suo funzionamento sia corretto. Nel test vengono controllati i 3 vertici della navicella in alcuni angoli noti: 0°, 90°, 180°, 270°, accertandosi che la navicella si disponga nel modo adeguato. Per eseguire correttamente il test è necessario impostare l'altezza e la larghezza della navicella (in *ShuttleImpl*) a 1.

3.2 Metodologia di lavoro

Suddivisione dei ruoli

La suddivisione dei ruoli è stata fatta nel seguente modo:

- **Gianmarco Magnani**: implementazione di una parte della logica dell'applicazione (Model), con gestione del movimento dei blocchi nell'area di gioco ed eventuali collisioni dei proiettili con quadrati o blocchi power-up. Attivazione e disattivazione del potere dei power-up al momento opportuno.
- **Nicolas Pasolini**: implementazione della restante parte di logica (Model), per quanto riguarda la generazione dei diversi oggetti, rispettando i vincoli descritti in fase di analisi, come l'aumento della frequenza di creazione, della velocità e della vita dei nemici.
- **Lorenzo Sutera**: realizzazione della parte di grafica relativa al menu iniziale e di pausa (View + Controller). Memorizzazione di tutti i punteggi globali su file e statistiche di gioco descritte durante l'analisi.

Possibilità di scegliere la risoluzione di gioco in base al display in uso. Coordinamento finale tra View, Controller e Model.

- **Paolo Baldini**: implementazione della grafica dell'applicazione per quanto riguarda la parte dell'area di gioco (View). Gestione di una parte delle interazioni del Controller.

Per una buona riuscita della fase di sviluppo abbiamo utilizzato diverse interfacce, in modo da avere a disposizione sin dall'inizio una buona parte dei metodi necessari. Nel corso delle implementazioni ci siamo sicuramente accorti che sarebbe stato necessario aggiungere nuovi metodi, oppure ridurre la visibilità di alcuni pubblici, per poter far interagire correttamente le diverse componenti. In questo modo le interfacce originali di partenza hanno subito qualche variazione nel corso del tempo, mantenendo però sempre il loro compito generale di partenza.

Impiego del DVCS

Come software di controllo versione distribuito abbiamo utilizzato **git** e la piattaforma di *Atlassian* **bitbucket** per la gestione del repository remoto. Subito dopo aver creato il repository era presente un unico ramo di sviluppo, *master*, nel quale abbiamo settato le impostazioni generali per Eclipse dei diversi plug-in, tra cui PMD, Checkstyle e FindBugs. A questo punto dal ramo principale ne è stato creato uno di sviluppo, *develop*, a partire dal quale abbiamo iniziato lo sviluppo dell'applicazione. Tutta la fase di implementazione si è dunque svolta in questo ramo, dove ciascun membro periodicamente caricava le proprie modifiche e scaricava quelle fatte dagli altri. In questo modo tutti quanti eravamo sempre aggiornati e allo stesso passo.

Verso la fine della fase di sviluppo è stato creato un nuovo ramo, *threadUpgrade*, in cui si è creata una classe astratta, *PausableThread*, per la gestione ottimizzata di tutti i thread presenti nell'applicazione. Si è dunque implementata questa nuova classe laddove era necessario l'utilizzo di un thread. Di tanto in tanto, in questo ramo sono state unite le modifiche del ramo develop, per far sì che i due branch non divergessero, cioè avessero contenuti troppo differenti. Alla fine del progetto, per via di una performance leggermente peggiore rispetto a quella del ramo develop, si è scelto di non fondere questo su quello di sviluppo, non alterando così quanto di buono era già stato completato. In conclusione, è stato fuso il ramo develop su quello principale, master, rilasciando così la prima versione ufficiale del gioco **CC-Tan**. In seguito sarà sempre possibile continuare l'ottimizzazione sul ramo *threadUp-*

grade per poi portarla in develop e infine rilasciare una nuova versione che migliori la precedente.

3.3 Note di sviluppo

Gianmarco Magnani

In questa sottoparte si è cercato di mantenere una struttura il più chiara possibile, con un linguaggio pulito e formattato, aumentando così la leggibilità del codice. Per il nome di variabili, metodi o classi sono stati utilizzati nomi il più possibile autoesplicativi, evitando di usare abbreviazioni incomprensibili. Laddove si sono impiegati stream è stato utilizzato uno stile "*fluent*" per una migliore visualizzazione e chiarezza. Inoltre, troviamo diverse featuring avanzate del linguaggio Java, che saranno di seguito elencate con qualche motivazione e anche il riferimento alla classe (o classi) in cui è possibile osservarle.

- Uso avanzato dei **generici**: sono stati impiegati, come accennato nella sezione 2.2 - Magnani, nella realizzazione del pattern builder. In particolare, siccome si hanno più builder che si ereditano "ricorsivamente", senza l'utilizzo dei generici avremmo avuto un vincolo sull'ordine dei metodi da richiamare per costruire l'oggetto. Vale a dire che si sarebbero dovute rispettare le gerarchie, ovvero partire subito a richiamare i metodi del builder più specifico per poi andare via via verso quelli superiori. Consideriamo un esempio con le classi viste in precedenza per chiarire il concetto: supponiamo di voler costruire un oggetto utilizzando il builder concreto *SquareBuilder*. Se da questo partiamo a richiamare un metodo del builder superiore, diciamo speed(), a questo punto otterremmo un oggetto del tipo di tale builder, *AbstractBuilderMI*, dalla figura 2.4 vista sopra. Da qui, dunque, potremmo solamente richiamare i metodi di questo builder, oppure di quello ancora più superiore, non avendo quindi la possibilità di vedere il metodo *hitPoints()*, che sarebbe nel costruttore di partenza. Questo fatto non è molto elegante, poiché vorremmo impostare le diverse proprietà dell'oggetto in qualsiasi ordine, senza avere questo vincolo. Per tale ragione sono stati realizzati dei builder generici, che fanno l'uso dei **bounded generics**, i quali limitano il tipo che si può passare, mediante la keyword *extends*. Anche in questo caso non riportiamo il codice, che è possibile comunque notare nei sorgenti, in particolare all'interno

delle classi *FixedItemImpl*, *MovableItemImp*, *SquareAgent*, *BallAgent*, all'interno delle quali si troveranno, innestati, i rispettivi costruttori.

- Uso delle **lambda expressions**: molto comode quando si deve definire un metodo sul momento, senza dover riscrivere tutto il prototipo. Sono state utilizzate in diverse occasioni, come ad esempio nella classe *ShuttleImpl*, nel metodo getShape(), per specificare l'azione da eseguire per ogni elemento della lista. Ne troviamo una anche in *BallAgent*, nel metodo updateAngle() per la creazione "al volo" di un comparatore, senza la necessità di creare una classe o nemmeno il bisogno di effettuare una implementazione anonima.

- Uso degli **stream**: nella classe *ShuttleImpl* sono stati utilizzati per eseguire delle operazioni sulle liste di *Pair* (si veda di seguito), tra cui filtraggio o mapping da un tipo a un altro, per ottenere infine una lista contenente un certo sottoinsieme degli elementi iniziali, a seconda delle esigenze. Tali manipolazioni sono presenti nei metodi *getActivePowerUps()* e *activePowerUp()*.
  Troviamo inoltre l'utilizzo di **Optional** nella classe *BulletImpl* e nel suo metodo *checkIntersecate()*. Esso dovrà restituire sempre l'ultimo oggetto contro il quale il proiettile corrente ha impattato, per evitare di contare due volte lo stesso danno (nel caso delle palline), qualora a seguito di un refresh rimanga ancora un minimo contatto tra i due oggetti. Ma, poiché, per diversi istanti un proiettile può non intersecare mai nessun elemento, tale metodo potrebbe non avere nulla da restituire. Per tale ragione si è scelto l'utilizzo di un opzionale, onde evitare valori null, che potrebbero generare eccezioni in un secondo momento.

Inoltre è stata utilizzata la libreria **Apache Commons Lang** per l'utilizzazione dell'interfaccia *Pair* e della classe *ImmutablePair*. Essa è stata impiegata in *BallAgent*, nel metodo *updateAngle()*, per memorizzare la distanza di ogni vertice del quadrato in cui la pallina ha impattato dal centro della palla stessa. Il *Pair* manterrà quindi la coppia (<nome vertice>, <distanza >) e da tale lista si andrà poi ad estrarre il Pair più piccolo, usando come criterio di confronto chiaramente la distanza (un comparatore implementato "al volo", specificato grazie ad una *lambda expression*). Da questo Pair ottenuto si estrarrà poi il vertice nel quale la pallina ha toccato e a seconda di quale sia verrà modificato adeguatamente l'angolo della pallina, per simulare un rimbalzo sullo spigolo.

Nicolas Pasolini

Le feature del linguaggio che sono state utilizzate per l'implementazione dei generatori, sono i generici, l'utilizzo delle espressioni lambda e l'utilizzo delle interfacce funzionali.

- L'uso dei **generici** è stato impiegato nell'interfaccia *ItemGenerator*<T extends *FixedItem* >e nella classe astratta *ItemGeneratorImpl* <T extends *FixedItem* >. L'utilizzo di questa feature permette di generare qualsiasi tipo di oggetto, purchè sia un FixedItem. Quindi, se in futuro si ha la necessità di creare e generare degli esagoni, dei rombi, oppure altre figure particolari, è possibile farlo attraverso un'implementazione chiara e semplice, sfruttando in modo appropriato questa feature di programmazione molto efficace. Nel caso in cui non avessimo usato i generici, se qualcuno avesse voluto reperire la lista di quadrati attraverso il metodo presente nel model, *MovableItem getSquareAgents()*, gli sarebbe stata restituita una lista di MovableItem, mentre, grazie ai generici gli viene restituito il tipo che è stato specificato alla creazione dell'oggetto, evitando eventuali ambiguità di tipo.
- Durante lo studio riguardante all'implementazione del generatore di palline uscenti dalla navicella, è emersa la possibilità, in futuro, di voler generare altri tipi di proiettili come per esempio un laser, delle mine, ecc. A fronte di questa considerazione, il concetto di palline è stato astratto fino ad arrivare all'idea di avere un unico generatore di proiettili. In questo modo, in futuro, sarà possibile utilizzare la medesima classe *BulletGenerator* per generare più tipi di proiettili. All'interno di questa classe, per riuscire a capire quale tipo di proiettile deve essere generato, è stata creata una particolare enumerazione, *BulletGeneratorSettings*. Tale enumerazione fa uso sia dell'interfaccia funzionale Supplier<T>, sia delle **espressioni lamda**. Lo scopo di questa enumerazione è quello di contenere tutti i tipi di proiettili che possono essere generati all'interno dell'applicazione e per ciascuno di essi è associato il rispettivo builder. Il builder, implementato da Magnani, viene fornito tramite il costruttore privato dell'enumerazione attraverso un'espressione lambda. In questo modo è possibile ottenere il giusto builder da utilizzare alla creazione di un nuovo proiettile, richiamando il metodo dell'enumerazione getBulletType() sull'oggetto di tipo BulletGeneratorSettings. Quest'ultimo oggetto è un componente fondamentale della classe *BulletGenerator*, in quanto permette appunto di capire quale sia il giusto proiettile che dovrà essere generato. Per cam-

biare il tipo di proiettile, quando il blocco del power-up viene distrutto, viene utilizzato il metodo setBulletType(BulletGeneratorSettings bulletSettings) proprio della classe BulletGenerator, andando a settare il giusto proiettile.

Lorenzo Sutera

- **lambda expressions** in *KeyCommandListener*;
- **Optionals** sono presenti in vari punti del codice;
- **stream** nel metodo bestScore della classe *Records*
- org.apache.commons.lang3.tuple che ha permesso la creazione di liste di Pair e Triple contenenti i dati dei *Records*.

Paolo Baldini

- Features avanzate utilizzate:
  - **Generici** (*ObserverSource*)
  - **Lambda expression**: si è sfruttata al massimo questa potente funzionalità.
  - **Stream e Optional**: si è cercato di dare un elevato peso all'utilizzo di questi costrutti nella scrittura del codice.
- Approfondimenti non trattati nel corso:
  - Utilizzo della libreria **Apache Batik** per la conversione di file .svg
- Librerie utilizzate nel progetto:
  - Librerie attualmente utilizzate:
    - Apache Batik
    - Apache Commons-io
    - Apache Commons-lang3
  - Librerie aggiunte in passato (spesso per testare la soluzione migliore) ma poi rimpiazzate/non più utilizzate:
    - SVGSalamander

    - TwelveMonkeys
    - JHotDraw
    - Google Guava
    - material-ui-swing
- Snippet:
    - Opportuna implementazione della pausa nei Thread

Nella stesura del codice si è prestata particolare attenzione all'utilizzo, dove possibile, delle lambda expression e degli stream. Si è evitato l'utilizzo di cicli for e while, preferendo lavorare con gli stream e non si sono riscontrati casi in cui questo non fosse possibile. Le lambda hanno avuto particolare applicazione in tutti i contesti dove gli opzionali ne facevano da padrone, riducendo al minimo l'utilizzo degli if tranne quando si è ritenuto che questi potessero migliorare la leggibilità del codice (vadasi lambda applicate ai metodi ifPresent di più opzionali innestati). L'utilizzo della lambda e degli stream è stato quindi molto intenso durante la stesura del codice. Fra gli aspetti avanzati utilizzati è presente anche l'utilizzo dei generici, sebbene in una sola interfaccia. Questi sono stati usati per creare una interfaccia generica di tipo ObserverSource (sorgente degli observer) che contenesse i metodi add e remove observer, comuni a tutte le implementazioni del pattern. Durante la programmazione si è ritenuto utile l'utilizzo di librerie esterne, come le classiche librerie di Apache "Commons-lang3", "Commons-io" e "Batik". Durante lo sviluppo si è fatto anche uso di altre librerie, come la libreria "JHotDraw", "Guava" di Google, "SVGSalamander" e TwelveMonkey; anche se poi, con l'evolvere del progetto, hanno perso la loro utilità o sono state trovate soluzioni alternative reputate migliori e sono state rimosse. E' stato anche proposto l'utilizzo della libreria grafica "material-ui-swing" anche se poi il gruppo ha preferito un approccio grafico più classico, considerata la natura retrò del gioco. La libreria studiata più approfonditamente durante il progetto è stata senza dubbio "Batik" di java, per il suo approccio non proprio intuitivo e user-friendly. Grazie alle guide di Apache e alla JavaDoc non è risultato però particolarmente complicato capirne l'utilizzo e il funzionamento. Sono poi stati aggiunti ulteriori .jar necessari per alcune dipendenze in altre librerie. Per quanto riguarda l'utilizzo di snippet di codice, l'unico caso degno di nota è stato nell'implementazione della pausa dei thread come descritto a questo link (*). L'utilizzo di questo snippet è stato dettato dalla volontà di implementare propriamente la pausa nei thread senza creare busy waiting. Non sono presenti altri snippet nel vero senso del termine, ma non si esclude di poter aver preso spunto nella risoluzione di problemi da altri codici

presenti nella rete. L'utilizzo del termine snippet non è comunque ritenuto corretto in questi altri casi dato che è stato comunque compiuto un lavoro di ragionamento e modifica per ottenere la soluzione considerata ottimale per lo scopo. Non è stato quindi effettuato un puro lavoro di copia-incolla ma una rielaborazione di idee a fronte dell'esperienza acquisita tramite lo studio dei metodi risolutivi altrui.

(*) https://www.tutorialspoint.com/java/java_thread_control.htm

Chapter 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Gianmarco Magnani

Per quanto riguarda la mia sottoparte mi ritengo soddisfatto del lavoro svolto e del risultato finale. Ho cercato sin da subito di raggruppare oggetti con caratteristiche comuni nella stessa entità, come un'interfaccia, tentando di isolare i diversi aspetti degli oggetti gli uni dagli altri. In questo modo si è creata una sorta di gerarchia tra le diverse interfacce e classi astratte, cosicché l'implementazione di quelle concrete è stata molto semplice, dato che molti metodi erano già stati definiti. Tale suddivisione rende possibile futuri cambiamenti, permettendo di ereditare dalla classe astratta più adatta tutti i metodi comuni, definendo solo quelli che hanno discrepanze. Un punto di forza, come visto in precedenza, sono sicuramente i pattern di progettazione, che permettono un elevato riuso delle classi superiori, senza la necessità di ricrearle quando vi sono piccole differenze. Inoltre, il fatto di avere una scomposizione gerarchica, dagli oggetti fissi fino a quelli movibili, proiettili ecc, permette di avere ben chiara l'organizzazione degli elementi presenti nella mappa di gioco ed eventualmente per aggiungere nuovi elementi basterà scegliere la classe astratta più adatta da cui estendere, dovendo così scrivere solamente poche righe di codice. Le parti nelle quali ho speso più tempo sono state quelle della gestione della rotazione della navicella al variare dell'angolo del suo asse, dove ho dovuto adattare le formule trigonometriche dei triangoli rettangoli per calcolare le coordinate dei tre vertici della navicella, e quella del rimbalzo delle palline. Anche in quest'ultimo caso il problema principale

è stato dovuto alla geometria piuttosto che alla programmazione in sè, infatti la difficoltà è stata capire come modificare l'angolo a seguito di un impatto in uno dei quattro lati del quadrato, per simulare il rimbalzo. Un'altra complicazione è stato il rimbalzo sugli spigoli, che ho approssimato vedendo il vertice come una retta perpendicolare alla diagonale passante per esso. Il rimbalzo quindi è stato calcolato su questa retta immaginaria, per avere una buona idea di tale concetto.

Passando ora all'aspetto organizzativo, personalmente mi sono trovato molto bene con tutti gli altri tre componenti del team, sempre molto disponibili e volenterosi di portare a termine il lavoro. All'interno del gruppo abbiamo avuto tutti quanti lo stesso ruolo, senza quindi alcun tipo di gerarchia. Ci coordinavamo di tanto in tanto poi ciascun componente si dedicava alla sua sottoparte, per poi unirle tutte e avviare l'applicazione. Ho trovato inoltre molto utile l'utilizzo di git per restare sempre sincronizzati e avere tutte le modifiche aggiornate, avendo a disposizione la possibilità di effettuare dei checkout per tornare a vecchie versioni, nel caso ci fossero stati problemi. Un aspetto molto vantaggioso è stato quello di poter lavorare su branch paralleli, così da effettuare modifiche aggiuntive su un nuovo ramo, senza intaccare la restante parte già completa e funzionante.

Nicolas Pasolini

Il lavoro che è stato svolto a riguardo della generazione degli oggetti all'interno dell'applicazione ha sia dei punti di forza, ma anche dei difetti. I punti di forza sono l'estensibilità del codice e il suo riuso. Come già spiegato anche precedentemente, grazie al pattern Template Method, abbiamo già una base sulla quale possiamo costruire in futuro nuove feature riguardanti l'applicazione in oggetto. Un punto a sfavore sono le performance dell'applicazione, in particolare quando un thread viene messo in pausa mentre sta effettuando una sleep. Nell'implementazione attuale, quando ad un thread viene chiesto di mettersi in pausa, e quest'ultimo sta effettuando la sleep, si verifica un ritardo nella sospensione dell'attività e questo può essere un difetto. Infine, non sono stati riuniti tutti i thread in un'unica classe astratta superiore che consentisse di definire tutte le parti in comune che essi hanno, come per esempio la possibilità di mettere in pausa, di terminare e di riprendere l'esecuzione dello stesso thread oppure di creare un unico metodo run() in comune. Penso infine di aver dato un buon contributo all'interno del gruppo e credo che il mio ruolo sia stato abbastanza centrale e molto importante per lo svolgimento del progetto.

Lorenzo Sutera

È presentata una GUI semplice e intuitiva, modificabile senza troppe complicazioni. Il lavoro effettuato sulla comunicazione tra i componenti MVC è molto ben riuscito. La creazione e il salvataggio su file dei punteggi e il calcolo delle statistiche su quest'ultimi è ben realizzato, e incline a modifiche future. Il metodo per calcolare la giusta dimensione delle finestre è realizzato grossolanamente, non in modo ottimale. Il key listener è ben realizzato e incline a modifiche future, quali nuove funzionalità. La riproduzione della musica poteva essere realizzata meglio e estesa allo scopo di riprodurre suoni di interazione durante il gioco. In conclusione credo di aver svolto un lavoro di livello Base-Intermedio.

Paolo Baldini

Nel complesso ritengo il risultato di questo progetto di buon livello. Per quanto riguarda le mie parti di codice ho opinioni contrastanti. Avrei in mente molte modifiche che sarebbe possibile apportare per migliorare qualitativamente il prodotto, ma che a causa dei tempi che si sono allungati non mi è possibile implementare lasciando al contempo il giusto tempo per il testing. Ritengo la struttura delle mie parti di codice di buona qualità. Prima della stesura ho ragionato molto su come implementare efficacemente la logica delle mie classi e, nonostante durante il progetto sia stato necessario modificarla leggermente, sono particolarmente contento di essere riuscito a seguirla quasi pedissequamente. La struttura logica che si delinea adesso alla fine del progetto è sostanzialmente quella definita a monte. Le differenze sono riguardanti le sezioni di update della view, che inizialmente era stato pensato di inserire all'interno del package della View, ma che è stato poi ritenuto corretto muovere nel controller per una questione logica, e l'update del model, che era inizialmente stato pensato di inserire all'interno dell'implementazione del controller, ma che è poi stato scomposto in una sotto classe. Un rammarico in questo progetto deriva dal fatto che avendo lavorato principalmente su view e controller ho avuto un campo di lavoro più limitato rispetto a quello del model per poter approfondire maggiormente alcune funzionalità avanzate come i generici. Non che nella view non sia possibile utilizzarne, ovviamente, ma il dominio del model si presta ovviamente meglio a questo genere di utilizzi. Tuttavia, per quanto la mia parte implementativa risulti forse un po meno complessa a livello logico della parte di model, ritengo di aver fatto un buon lavoro, che mi ha permesso di far fronte a imprevisti implementativi durante la realizzazione e di relazionarmi con le parti di codice dei miei

compagni in maniera tutto sommato fluida. Il mio giudizio finale sul lavoro è comunque fortemente positivo. Anche se avrei avuto immensamente piacere di potermi mettere maggiormente in gioco in alcune parti del progetto, ritengo comunque che il mio contributo sia stato particolarmente importante.

4.2 Difficoltà incontrate e commenti per i docenti

Le nostre considerazioni finali sul progetto sono positive, anche se, durante lo svolgimento di questo ultimo, non nascondiamo il fatto di aver riscontrato alcune criticità a livello organizzativo. La prima parola che riassume il nostro stato d'animo al termine di tutto è "soddisfazione". C'è soddisfazione perché all'inizio alcuni membri del gruppo erano critici a riguardo del progetto finale. Nonostante quella poca fiducia che era stata attribuita all'inizio, ciascun membro si è dato da fare sin dall'inizio per studiare accuratamente la fattibilità di tutto il progetto e se ciascuno di noi avesse avuto le competenze necessarie per sostenere la propria parte. Fortunatamente Paolo aveva già svolto alcuni progetti di grafica, quindi l'idea di dover sviluppare interamente la grafica dell'applicativo non lo spaventava. Mentre, anche Magnani aveva già avuto esperienze passate nello svolgere un progetto in comune con altri ragazzi su git, quindi a livello organizzativo è stato di molto aiuto per quanto riguarda la fase di coordinazione iniziale. Al termine del progetto, ci siamo chiesti se ciascuno di noi fosse stato in grado di svolgere alcune parti che sono state svolte da altri. La risposta finale per alcuni è stato un secco no. Questo no nasce dal fatto che il corso, a nostro parere, non ci ha fornito un approfondimento sufficiente per quanto riguarda la programmazione concorrente in java. Le sole due lezioni tenute dal professor Ricci sono state insufficienti, sebbene molto interessanti e proficue. Infine ci saremmo aspettati di più per quanto riguarda la grafica, argomento trattato in maniera molto superflua e sbrigativa. Questa idea forse è frutto della nostra inesperienza in questo campo e quindi all'inizio ci può sembrare tutto insormontabile, pertanto un vantaggio per i prossimi allievi sarebbe tenere qualche approfondimento in più su come implementare una grafica multithreading, mostrando le diverse iterazioni che si vengono a creare tra le classi coinvolte, magari assegnandogli qualche esercizio per casa e proporre una soluzione online. Per quanto riguarda le nozioni base della programmazione ad oggetti sono state spiegate in modo molto dettagliato ed esaustivo dal professore Viroli. A partire dall'oggetto in sé fino ad arrivare al polimorfismo, ai generici e per concludere

ai pattern. Ogni lezione tenuta dal professor Viroli è risultata molto chiara e precisa sotto ogni punto di vista. Il suo rigore, la sua professionalità e la sua fermezza ci ha coinvolto sin dal primo giorno e motivato nel curare tutti i diversi aspetti della programmazione ad oggetti. Forse parte della nostra determinazione a portare a termine questo progetto è anche grazie al vostro comportamento diligente e rigoroso che avete sempre tenuto in aula con ciascuno di noi. Per questo ve ne siamo grati e vi ringraziamo.

Appendix A

Guida utente

L'applicazione si presenta inizialmente con una finestra di impostazioni, in cui il giocatore inserisce il proprio username e sceglie la dimensione della finestra di gioco a seconda della risoluzione del proprio monitor. Dopo aver configurato opportunamente queste impostazioni, premendo il pulsante "done", si ha accesso al gioco. Tale finestra è rappresentata nella seguente figura A.1.

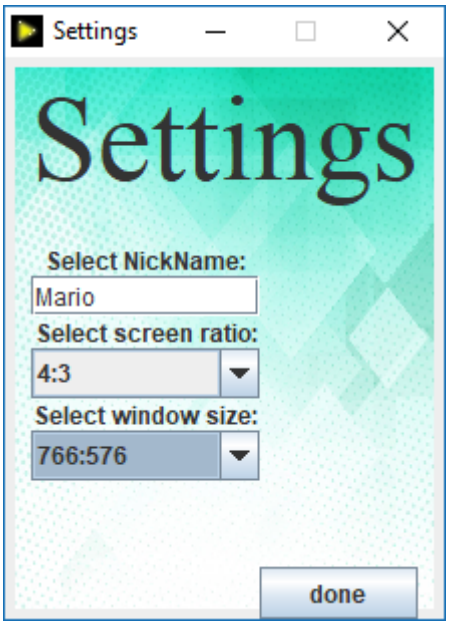


Figure A.1: Schermata iniziale, nonché finestra Settings.

Nella nuova schermata che si aprirà, come illustrato dalla figura A.2, viene

fornita la possibilità di modificare le impostazioni della finestra di gioco che sono state selezionate precedentemente (cliccando "*Settings*"), inoltre l'utente ha anche la possibilità di consultare la classifica globale dei 10 punteggi più alti cliccando sul pulsante "*View Leaderboard*". Mentre, nel caso in cui abbia intenzione di iniziare una nuova partita, può premere sul pulsante "*Start*". Diversamente, per uscire dall'applicazione è possibile cliccare sul pulsante "*Exit*".

Figure A.2: Menù principale, con tutte le diverse azioni da eseguire.

Mentre l'utente sta giocando può decidere di mettere in pausa la partita cliccando sia sulla barra spaziatrice da tastiera, sia sulla lettera "P". Infine, nel caso in cui venga premuto il pulsante "Esc", sempre da tastiera, apparirà un menù in cui viene fornita la possibilità di modificare le impostazioni della finestra di gioco (uscendo dalla partita in corso), riprendere la partita corrente (resume), iniziare una nuova partita da zero (restart), oppure di uscire completamente dall'applicazione (exit). In quest'ultimo caso, il punteggio corrente della partita *non* verrà salvato in memoria, mentre in tutti gli altri casi sì, contribuendo così alla classifica globale. L'ultima figura A.3 rappresenta come si presenta il gioco vero e proprio, con la navicella che sta sparando le palline contro i quadrati nemici.

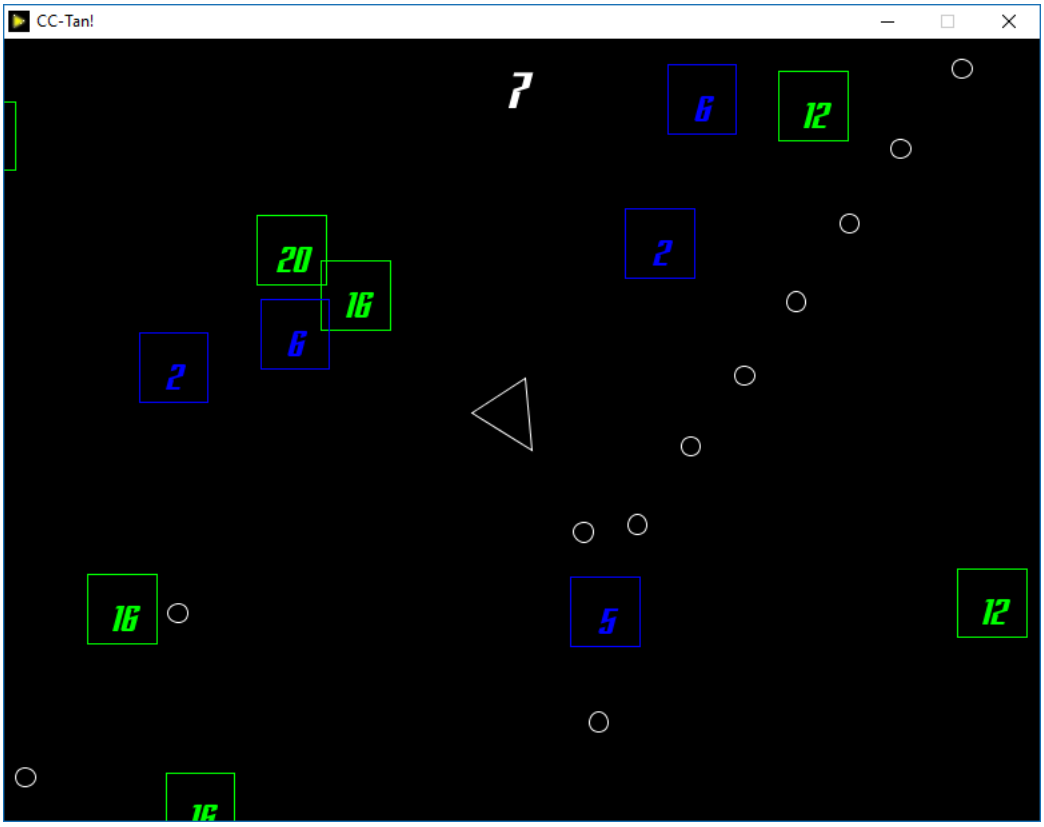


Figure A.3: Finestra di gioco, con la navicella (triangolo nel centro) che sta sparando le palline (proiettili) contro i nemici.