RELAZIONE PROGETTO DI METODOLOGIE DI PROGRAMMAZIONE
2023 – 2024

GIULIO ALLEGRETTI, MATRICOLA 2029763
CORSO IN TELEDIDATTICA

PROGETTO JBOMBERMAN – TEAM DI 1 PERSONA

È con orgoglio e gratitudine che consegno questo mio progetto: esso è il mio primo programma ed è completamente scritto da me dall'inizio alla fine. È programmato esclusivamente in Java e la sua GUI è implementata a partire dalla libreria Java SWING.

Ho affrontato questa avventura guidato dai seguenti principi direttivi:

- Approfondire la mia conoscenza e capacità applicativa dei Design Patterns.
- Progettare una applicazione il più possibilmente estensibile nelle sue parti, dando per scontato che se ne vogliano sviluppare versioni aggiornate in futuro.
- Sviluppare una mentalità professionale nell'ambito della Programmazione ad Oggetti.
- Il mio modesto videogioco vuole rendere omaggio per quanto mi è stato possibile, al massimo delle mie capacità, alla versione originale di Bomberman.
- Vorrei utilizzare questa applicazione nel curriculum a favore della ricerca di lavoro nel campo della programmazione.

La applicazione che ho scritto è stata testata più volte, per molte ore. Io sono stato il beta-tester.
Alla versione attuale è possibile completare tutto il gioco con successo, usando il personaggio che si preferisce, nello Stage che si desidera. È possibile sfruttare le operazioni di salvataggio, carimento, creazione di Account: tutto ciò che è disponibile funziona. Completare il gioco significa superare tutti gli otto livelli dello Stage che si è scelto. L'ultimo livello di ogni Stage è una Boss-fight.
Mi sono sforzato di scrivere JavaDoc tutto in inglese, poiché Java è in inglese.

In questi 6 mesi mi sono documentato e ho studiato da molte fonti per poter sviluppare la percezione, la capacità e la maturità minima necessaria per progettarlo. Sono partito da zero, prima di questo corso non conoscevo Java, bensì solo Python (sempre grazie al contesto universitario). Ogni step nello sviluppo è stato un salto nel vuoto, ma questo mi ha anche regalato grandi emozioni. È incredibile per me pensare a tutto quello che è accaduto in questo lasso di tempo e quanto io sia cambiato. Questa applicazione è cresciuta con me, ha vissuto i miei errori e le mie paure. Alcune sue parti sono state riscritte più volte nel corso del tempo in un processo di refactoring. Altre invece sono rimaste, come strati più antichi sopra a quelli più recenti, come una strana riproduzione virtuale della città di Atene. Progettare una applicazione a partire da zero garantisce poteri decisionali assoluti che corrispondono ad altrettante assolute responsabilità. Ho vissuto all'interno di questo codice che da piccola casetta qual'era ora mi sembra un paese intero, con la sua geografia, politica e i cittadini che vi abitano. Infatti, il più grande ostacolo che ho dovuto fronteggiare durante lo sviluppo di questo progetto è stato il dovermi confrontare con una mole di

codice maggiore di quella che potessi ricordare di avere scritto. Ad un certo punto, sapevo di non essere più completamente conscio di ciò che avevo prodotto tempo addietro. Ho imparato che esiste una memoria temporale in me. Non avevo mai superato questo limite. Ho sperimentato il fatto di non riconoscermi, di dovermi relazionare con la qualità del codice scritto da una versione passata di me. Alcune parti sono state rifatte per questo motivo, in modo da risultare più ordinate. Questo mi ha all'inizio traumatizzato, ma mi ha anche fatto capire che la prima responsabilità che ho nello scrivere un programma è nei confronti di me stesso, mentre la seconda è nei confronti di coloro che andranno a leggere ciò che ho scritto – capire quello che ha scritto un'altro dev'essere ancor più difficile. È quindi fondamentale lasciare traccia del significato del codice mediante appunti e commenti all'interno dello stesso. Utilizzare i Design Patterns è stato come introdurre dei punti di riferimento all'interno della struttura generale, come dei monumenti, obelischi, fari. Il principio dell'incapsulamento è risultato utile non solo per compartimentare il codice durante la sua stesura, ma soprattutto in seguito per favorirne la manutenibilità e definire i contesti operativi.

Se riguardo a tutto quello che ho fatto fino ad adesso sorge in me il desiderio di rifare tutto da capo. Non perché non sia soddisfatto del risultato attuale, ma perché penso che ora potrei approciarmi al progetto completo con una mente capace di pensare in modo più ampio e dotata del suo bagaglio di errori risolti. Credo che sia la maledizione di Sisifo che affligge la programmazione: quella di voler riscrivere godendo delle ultime cose apprese. Per me questo è accaduto quando, a metà del progetto, capii di dover riprogettare tutti i decorator della View per rendere più fruibili i tipi agli altri package. Sperimentare, soprattutto. Se penso a quanto io sia ancora così inesperto, sono molto felice. Quanto ho ancora da scoprire e da elucubrare, se già dopo 6 mesi sono arrivato qui ! Quanto mi sono divertito !

NOTE PROGETTUALI E DI SVILUPPO

Di questo percorso ricordo benissimo gli ostacoli più grandi che ho dovuto superare, l'ansia scaturita nel confrontarmi con essi e infine la più grande soddisfazione nel riuscire a risolverli.

(a) DEFINIZIONE DEL LOOP TEMPORALE

- La definizione del loop temporale e l'arrivare a muovere il personaggio in una maniera che fosse godibile e fluida. Per raggiungere questo traguardo ho dovuto sviluppare un Delta Loop (lo Sleep Loop è risultato meno efficiente) che stabilizzasse gli FPS ad un valore desiderato. Lo Sleep Loop sfrutta il metodo sleep della classe Thread, benchè possa funzionare anche nel mio contesto Multi-threading (giacché join blocca ThreadMenu fino alla morte di ThreadGame e lo sleep non è considerato "morte") il metodo sleep risulta impreciso per garantire un feeling efficace nelle movenze del player. Avevo sviluppato Delta Loop per ThreadGame e Sleep Loop per ThreadMenu, tuttavia alla fine ho preferito usare solamente il Delta Loop e generalizzare il codice nella superclasse astratta ThreadDelta, cosa che ha portato ad avere l'architettura attuale (applicando poi il Template Method Pattern sul metodo run). Qui di seguito un esempio di Sleep Loop.

```
// Parametri temporali per stabilizzare gli fps
double bar = 1_000_000_000 / FPS; // intervallo di scrittura, la battuta del game loop
double delta = 0;
long now = System.nanoTime();           // istante attuale
double nextBar = System.nanoTime()+bar; // prossimo intervallo di scrittura
double quintupleClock = 0;
double decimaClock = 0;
```

```
boolean up, down, enter;
up = down = enter = false;

while(JBomberMan.getInstance().isAppOn()) {

        while(JBomberMan.getInstance().isMenuOn()) {

                DynaSlaveCard ds = dm.getActivePanel();
                int mod = ds.getMod();

                System.out.println(dm.getActivePanelName());

                Dir key = klm.getDir();
                up = key==Dir.UP;
                down = key==Dir.DOWN;
                enter = klm.getEnter();
```

```
            try {
                    double rest = (nextBar - System.nanoTime())/1_000_000;
                    if(rest<0) rest = 0;
                    Thread.sleep((long)rest);
                    nextBar += bar;
                    decimaClock += 0.01;
            } catch (InterruptedException e) {
                    e.printStackTrace();
            }
    }
```

- Il ThreadDelta si basa sulla registrazione del tempo corrente e quindi sulla differenza di tempo intercorsa tra la registrazione precedente e quella attuale. Tale differenza concorre a definire All'interno di ThreadDelta ho scritto un meccanismo di animazione che si basa su tre ingredienti principali: Iterator Pattern per regolare la successione di sprite (che ho chiamato footage) all'interno dell'animazione, il Decorator Pattern che ho dovuto usare per poter rispettare MVC ottenendo versioni disegnabili di oggetti a partire dai tipi scritti nel Model, l'Observable-Observer che ha permesso al Controller di comunicare alla View i momenti esatti in cui si devono animare gli elementi grafici grazie alle notifiche ricevute dal Model.

(b) ESPLOSIONE

- La generazione dell'effetto esplosivo della bomba definito all'interno del Modello mediante ricorsione, ma da doversi applicare alle versioni Decorate sia della bomba sia delle sue fiamme. L'evento "esplosione" ha reso necessaria la presenza di uno spazio tra le funzioni a cui gli oggetti scenici sono suscettibili tale che ogni oggetto, prima di venire distrutto, avesse uno spazio apposito in cui effettuare le sue operazioni pre-distruzione. Da questa necessità è nato il metodo trespasser() nella classe Prop. La particolarità dell'evento

esplosione e del metodo trespasser è la seguente: l'esplosione di una bomba innesca l'immediata esplosione di una bomba nel suo raggio d'azione (anche se quest'ultima è stata deposta da poco). Questo accade perché si innesca il trespasser della bomba nel momento in cui gli oggetti Fire generati dall'esplosione vanno a influenzare la bomba recente ancora in fase di animazione. Penso che questa proprietà sia molto importante per rispecchiare fedelmente il funzionamento del gioco originale. Un'altra caratteristica particolare delle esplosioni è il fatto che le fiamme di una eplosione impediscono la prosecuzione delle fiamme generate da un'altra esplosione, come a disporre di un muro di fuoco. Questa proprietà può avere risvolti strategici in fasi avanzate del gameplay. Sta di fatto che gli oggetti Fire sono molto importanti ai fini del funzionamento delle esplosioni, sono i protagonisti. Il loro funzionamento sostanzialmente è questo: sono gli unici oggetti che possono avere letalità uguale a true e le Tessere restituiscono la letalità del loro Prop in bundle qual'ora presente, ergo la presenza di un oggetto Fire all'interno di una Tessera la rende letale fintanto che l'oggetto Fire è ad essa avviluppato. Le fasi riguardati un evento "esplosione" si possono riassumere nei seguenti termini: (1) Viene eseguita l'animazione della bomba. Quando l'animazione è completa (ovvero l'iteratore di DecoratorBomb ritorna hasnext() → false) vengono registrati gli oggetti da far distruggere e le posizioni ove fare scaturire le fiamme. (2) Si fa eseguire il metodo trespasser agli oggetti da distruggere in modo da far eseguir loro le loro "ultime volontà" (il testamento). (3) Si applicano le operazioni nelle posizioni registrate e le fiamme provvedono alla loro animazione. Il LevelAbstract dispone poi del metodo erase che permette di annullare a tutti gli effetti il bundle delle Tessere i cui Prop compositi devono essere tolti dalla Griglia. Questo si applicherà solo al termine delle animazioni, infatti i Decorator alterano il significato del metodo trespasser del loro componente e invocano super.trespasser() solo al termine della loro animazione. Tutto può funzionare perché le coordinate che restituisce il Decorator sono le stesse del suo componente.

(c) FULLSCREEN

- La mia ostinazione nel voler ottenere il fullscreen ha causato non pochi problemi. Primo fra tutti la necessità di dover ridimensionare tutti gli elementi rispettando le proporzioni. Questo è stato risolto disegnando tutto il gioco in una immagine a dimensioni canoniche (il defScreen). Essa viene poi ridimensionata secondo i desiderata dell'Utente. Ho scoperto un modo per richiedere i parametri del monitor al sistema operativo e quindi il defScreen viene disegnato nella DynaMasterCard, invece la Card di volta in volta attiva nel Layout non disegna sullo schermo, bensì modifica il contesto grafico del defScreen. Per come è implementato il fullscreen ora sarebbe teoricamente possibile a questo punto generalizzare questa procedura a una qualunque risoluzione richiesta. Ho capito che poter risolvere problemi in maniera generale (piuttosto che in maniera specifica per una determinata situazione) permette di ampliare le possibilità attuali, anche dapprima impensate; alcune idee sono sorte come digressioni di altre. Le digressioni vanno gestite perché possono richiedere più tempo del previsto per essere portate a termine. Un secondo problema in merito al fullscreen è stato il riposizionamento corretto dei tasti invisibili che poggiano ora esattamente sulle scritte visibili a schermo e riferite a quelli (tali scritte sono le nomunclature delle opzioni all'interno dei vari pannelli). Infatti, ogni opzione scritta all'interno dei pannelli della View è solo un disegno. Sopra ognuna di esse si trova un

pulsante invisibile. Perchè ho voluto complicarmi così la vita ? Per una ragione estetica: non ero soddisfatto della resa estetica dei pulsanti di Java Swing, ritengo tutt'ora che l'applicazione perda di personalità. Pertanto, i pulsanti esistono, ma non si vedono e l'Utente può premere le stringhe anche col mouse. Questo risultato è stato possibile anche sfruttando il fattore di correzione che si trova nel DynaFrame.

(d) FOCUS DEI PANNELLI DELLA VIEW

- Il focus dei pannelli del CardLayout per la GUI che veniva perduto durante l'interazione tra Utente e pannelli nested. Alcuni pannelli del CardLayout infatti possiedono al loro interno un'altro pannello, denominato Nested. Interagire con il pannello Nested faceva perdere il focus a tutto il pannello. La chiave è stato capire che il focus viene gestito unicamente dal Master (il pannello che ospita il CardLayout come suo layout). Pertanto, le interazioni nei pannelli Nested devono concludersi con la richiesta di focus al Master. Ho fatto tutto ciò per dare la possibilità all'Utente di utilizzare anche solamente la tastiera o fargli usare tastiera e mouse assieme, in modo da ottenere una esperienza di generazione di input piacevole. In particolar modo, nel pannello in cui si seleziona il personaggio per giocare si può selezionare col mouse il personaggio e premere enter con la tastiera per tornare ad utilizzare le funzioni del pannello container. Così avviene anche nel pannello di selezione dello Stage, dove selezionare lo Stage desiderato col click del mouse non impedisce di premere subito dopo i tasti della tastiera con successo per passare alla opzione successiva. Lavorare sulla view, sul front-end, è stato duro e mi ha fatto capire quanto lavoro e quanto tempo sia necessario per creare una applicazione la cui responsività sia godibile per l'utenza. Penso ad ogni applicazione con cui ho interagito durante la vita ed al fatto che anche quei loro meccanismi che considerai banali, "scontati", richiesero impegno per poter essere realizzati. Quanto è stato semplice banalizzare ignorando tutto ciò ! Nei momenti in cui tutto mi sembrava disfatto ho trovato la forza di continuare attingendo all'amore per ciò che volevo produrre, ottenere: ho immaginato il risultato finale. Io non ho utilizzato nessun software aggiuntivo per creare la mia interfaccia: tutto è frutto di prove e di errori.

(e) MULTI-THREADING

- Ottenere una applicazione Multithreading. Questa idea si è innestata in me a poco a poco. All'inizio volevo gestire il tutto usando unicamente un ciclo while. Poi ho capito che erano necessarie almeno due situazioni differenti perché tutto potesse venire elaborato con ordine: il contesto del Menù (con tutti i suoi pannelli, opzioni, possibilità di salvataggio e di creazione di Account nuovi, con la scelta di personaggio e di livello da giocare…) ed il contesto del Game vero e proprio (che nella view è il DynaSlaveGame). Quindi sarebbero dovuti esistere due cicli while distinti e si sarebbero dovuti attivare e disattivare nel modo corretto, ma come rispettare l'Observable-Observer ? Poi come per gioco ho trasformato il ciclo while in un Thread: un oggetto Runnable, molto più comodo per poter disporre di libertà in merito all'unica classe da cui estendere. Quindi ho creato un'altro thread per gestire il contesto Menù. I due thread tuttavia non comunicavano correttamente all'inizio, la View andava predisposta prima che avvenisse il passaggio effettivo al contesto Game. In effetti, lo stesso JFrame di Java Swing dispone anch'esso del suo Thread. Quindi il passaggio da un Thread all'altro del Controller avviene grazie alle notifiche del GameModel: ho predisposto degli oggetti di tipo Order a questo scopo. L'applicazione

comincia con il thread del main, il quale attiva il ThreadMenu e carica qualche livello (quelli iniziali per ogni Stage) prima di concludersi. Nel mentre, l'Utente dispone già dell'interfaccia grafica per operare. I pannelli che predispongono la sessione di gioco servono a salvare all'interno di DynaFrame i valori scelti dall'Utente con i quali iniziare la partita (il playerCharacter, lo Stage) dopodichè DynaSlaveStage predispone la View al passaggio al contesto di gioco. Il ThreadMenu recepisce queste informazioni e quindi arriva al punto di congelarsi tramite il metodo join() : resta in attesta della morte del nascente ThreadGame. Il ThreadGame viene generato usando un metodo custodito all'interno di JbomberMan (dove c'è anche il main dell'applicazione). Ogni sessione di gioco vede la generazione di un nuovo oggetto ThreadGame: esso si consuma, non viene mantenuto. Il ThreadGame termina se una delle tre condizioni di terminazione viene raggiunta: gameOver, escape (uscita volontaria, senza aver mai perduto una vita), gameWin (vittoria di tutti e otto i livelli). Quindi le notifiche del GameModel tramite observable-observer cambiano DynaSlaveGame (Card attiva durante la sessione di gioco) e riportano a DynaSlaveStart (Card di inizio dell'applicazione) e ripristinano il funzionamento di ThreadMenu da dove era rimasto congelato. Per ultima cosa ho applicato Template Method Pattern sull'astrazione ThreadDelta, perché in ThreadMenu e ThreadGame (i sottotipi di ThreadDelta) vi fosse un metodo run che beneficiasse dell'incapsulamento degli algoritmi del loro supertipo. Sono molto soddisfatto del funzionamento dei Thread, del loro metodo Run e delle loro interazioni mediante Observable-Observer.

- Mi è stato di grande utilità progettare a mano gli algoritmi riguardanti il ThreadGame. Scrivere mi ha aiutato a ragionare con cautela. Propongo qui di seguito 4 fogli inerenti lo sviluppo di ThreadGame dai quali risulta una evoluzione del mio pensiero da programmazione con if/else a progettazione con oggetti.

BOZZA 0.1

run ThreadGame:

```
while (isAppOn) {
    while (isGameOn) {
        /* load assets and load from View the first level from Stage selected by User */   (as the PC)
        while (attempt >= 0 && isGameOn) {
            // draw initial Screen → name of Level and Stage   // temporized! with Thread.sleep.
            // reset Timer Game
            boolean gameOver, gameWin = false;      { gameRun = !gameOver && !gameWin
            (A) while (!gameOver && !gameWin) {
                ( paused = Keylist.getPause();
                  draw PausedScreen { exit from game: }
                (B) while (!gameOver && !gameWin && !paused) {
                    // Δ ≥ 1 → if (winConditions) gameWin = true;
                    // timerGame → if (timerGame == 0) death = true;
                    // if (death) {
                        lives--;
                        if (lives >= 1) {
                            invincibility (3 seconds);
                            if (timerGame == 0) restore Partial TimerGame;
                        } death = false;
                    // if (death) {
                        gameOver = true;
                        death = false;
                        gameWin = false;
                    } else if (gameWin) ...
                }
            }
            if (isGameOn) {
                if (gameOver) {
                    boolean choiceMade = (attempt < 0) ? true : false;
                    while (!choiceMade) {
                        → draw Continue Screen : continue?
                            → yes: { choiceMade = true;
                                     attempt--;
                                     lives = 1;
                                   }
                            → no: { [REGISTRA SCONFITTA] * choiceMade = true;
                                    isGameOn = false;
                                    draw GameOverScreen;
                                    changePanel("start");
                                  }
                else { // gameWin == true
                    [REGISTRA VITTORIA DI LEVEL] *
                    // load new assets for level. PC remains the same.
                }
```

|| generarli solo quando serve! Quindi questa parte si da togliere poi...

|| per rendere il codice più leggibile bisogna usare i campi dell'oggetto Thread Game a mio favore e registrarsi gli assets!

{ death = true because no lives || !stillInTime

→ if (deathConditions) death = true;

{ // exit by Paused Screen
[REGISTRA FUGA] *
isGameOn = false
break // for exit from (A)
changePanel ("start")
→ e fare prima pannello di fuga?

NO — NO — NO — Sì — Sì

Sì: da mettere lì dove è il tempo commensurabile

!! Using a Nested DynaSlave Cond?!

{ unificare dunque (A) con (B)

DA RICONTROLLARE

(mettere anche registrazione n° volte di stage completati) → [REGISTRA VITTORIA DI LEVEL]

BOZZA 0.0 → Vedi Sx

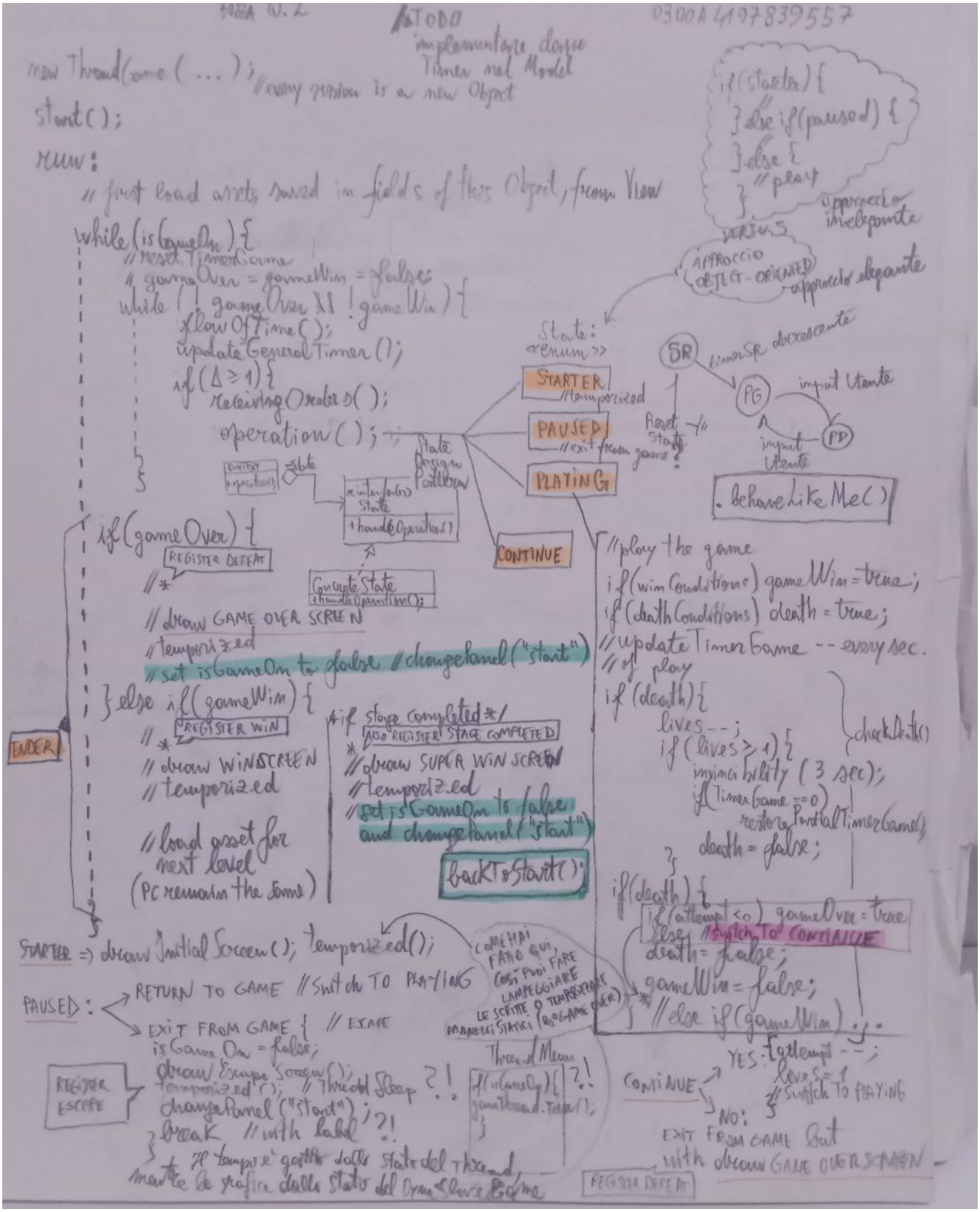
run ThreadGame :

```
while (isAppOn) {

  while (isGameOn) {
    // load from View the first level from Stage selected by User
    while (isGameOn) {
      pause = KeyLis.getPause();
      while (! pause && isGameOn()) {
        if Δ≥1 {
          // game
        }
        if (timerFps>= second)) {
          //
        }
      }

    while (attempt ≥ 0 && isGameOn) {
      // reset TimerGame (3:00)
      while (paused) {   — error!
        paused = KeyLis.getPause();
        draw pausedScreen;
        while (! nolives && stillOnTime && ! paused) {
          // Δ ≥ 1        /* play the game */
          // timerGame
          // if (death) {
              lives--;
              invincibility (3 secondi di durata)
              if (lives < 1) nolives = true      migliorare!
              else death = false
          }
        }
      }
      if (attempt >= 0) {
      draw Continue screen : continue ? {
          yes → attempt --
          lives = 1
          no → isGameOn = false → registra sconfitta
      }
    // lighten up, exit from game, changePanel("start")
       go to main menu
```

implementare
*) vittoria e proseguo nei livelli.


TODO
implementare classe Timer nel Model
new Thread(game(...)); // every version is a new Object
start();
run:
// first load assets saved in fields of this Object, from View
while(isGameOn){
// reset Timer&Game
// gameOver = gameWin = false;
while(!gameOver && !gameWin){
flowOfTime();
updateGeneralTimer();
if(Δ ≥ 1){
receivingOrders();
operation();
}
if(gameOver){
REGISTER DEFEAT
// *
// draw GAME OVER SCREEN
// temporized
// set isGameOn to false // changePanel("start")
}else if(gameWin){
REGISTER WIN
// *
// draw WINSCREEN
// temporized
// load asset for next level (PC remain the same)
// if stage completed */
ALSO REGISTER STAGE COMPLETED
*
// draw SUPER WIN SCREEN
// temporized
// set isGameOn to false; and changePanel("start")
backToStart();
ENDER
STARTER => drawInitialScreen(); temporized();
PAUSED: RETURN TO GAME // switch TO PLAYING
EXIT FROM GAME { // ESCAPE
isGameOn = false;
drawEscapeScreen();
temporized(); // ThreadSleep ?!
changePanel("start");
break // with label ?!
}
REGISTER ESCAPE
// temporale gestite dallo stato del thread
mentre le grafiche dallo stato del DrawSlaveGame
COME HAI FATTO QUI, COSÌ POI FARE LAMPEGGIARE LE SCRITTE O TEMPORIZZARE (PRO GAME OVER)
Thread Menu
if(isGameOn){
gameThread.Take();
}
?!
?!
State:
<<enum>>
STARTER
// temporized
PAUSED
// exit from game?
PLAYING
Reset Stage ?
State Design Pattern
<<interface>> State
+handleOperation()
ConcreteState
+handleOperation();
CONTEXT (operation())
State
CONTINUE
// play the game
if(winConditions) gameWin = true;
if(deathConditions) death = true;
// updateTimerGame -- every sec.
// of play
if(death){
lives--;
if(lives ≥ 1){
invincibility (3 sec);
if(TimerGame == 0)
restorePartialTimerGame();
death = false;
}
checkDeath()
if(death){
if(attempt < 0) gameOver = true
else // switch TO CONTINUE
death = false;
gameWin = false;
} // else if(gameWin) ...
CONTINUE
YES: attempt--;
lives = 1
// switch TO PLAYING
NO:
EXIT FROM GAME but with draw GAME OVER SCREEN
REGISTER DEFEAT
if(starter){
}else if(paused){
}else{
// play
}
VERSUS
APPROCCIO OBJECT-ORIENTED
approccio alternativo
approccio imperativo
SR
timerSR decrescente
PG
input Utente
PP
input Utente
behaveLikeMe()

new ThreadGame(...);

start();

run();

(Model the objects selected in the V.ow)

// found from the level and PC. Save them in the fields of this Object.

// currentState = StartGameState.getInstance(this);

while (!isGameOver) {

// gameOver = gameWin = false

// reset TimerGame and TimerGeneral

while (!gameOver && !gameWin) {

flowOfTime(); // with updateTimerGeneral

if (Δ ≥ 1) {

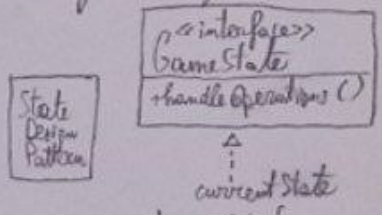
receivingOrders();

operations();

}

}

}



State design Pattern

public void operations() {
currentState.handleOperations()
}

Every implementations of GameState is a singleton.

ThreadGame

public void setCurrentState(
GameState gS) {
this.gS = gS
}

But I need the reference of the current ThreadGame object, so...

in a GameState implementation: ...

private static GameState instance;

private ThreadGame actualTG;

private constructor() {}

public static GameState getInstance(ThreadGame actualTG) {

if (instance == null) instance = new ***State();

this.actualTG = actualTG;

return instance;

}

and final!

// getter for the actualTG ...

example: in the code of a GameState implementation, to code the transition of states of the TG:

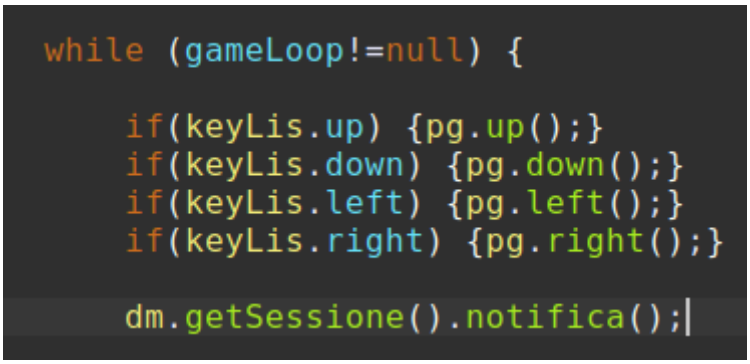
//

actualTG.setCurrentState(nextState.getInstance(actualTG));

- *Postilla*. La velocità standard di movimento dei personaggi è stata stabilita in modo da essere in relazione costante con gli FPS grazie ad una costante detta Costante di Alacrità. Esiste infatti questa relazione, visibile nel MyJBombermanFormat, tale per cui: FPS * SPEED_STANDARD = ALACRITY_CONSTANT ovvero SPEED_STANDARD = ALACRITY_CONSTANT / FPS.

(f) RESPONSIVITÀ

- La responsività a seguito dell'immisione di input da parte dell'Utente. Per quanto riguarda il contesto Menu questo problema tratta la responsività del puntatore dell'opzione corrente. Quando l'Utente premeva il tasto S (giu) o W (su) non si otteneva sempre lo stesso risultato in termini di tempistiche e di ritardo di ricezione dell'input. Passare dallo Sleep Loop al Delta Loop ha aiutato, come ha aiutato anche la scelta di intervalli ricettivi intorno ai due decimi di secondo, ma ciò che ha risolto davvero la situazione è stato utilizzare un Getter che esaurisse il segnale di cui fosse stato portatore. Questo Getter "di Scarico" restituisce il valore ricevuto dal Listener, ma nel momento in cui questo è vero allora al campo viene assegnato il valore falso, cioè il segnale viene annullato automaticamente. In questo modo solo ripremendo il tasto, cioè togliendo il dito e riposizionandolo, si può effettivamente dare il segnale di modifica del cursore. La formula del Getter di Scarico è tanto semplice quanto funzionale. Chiaramente questo Getter non poteva essere utilizzato in un contesto di Gameplay: se l'Utente mantiene premuto un tasto il suo personaggio deve continuare a muoversi. Per quanto riguarda il contesto Game c'è voluto un po' di tempo per arrivare a trovare la soluzione che mi soddisfasse. Ci potevano essere diversi modi per controllare la direzionalità mediante if...else.



Questo controllo di direzionalità genera un vettore di movimento che è la sommatoria delle direzioni impresse al personaggio. Questo comporta anche il movimento diagonale che sarebbe interessante, ma nel mio caso non andava bene. Inoltre, la pressione contemporanea di due tasti con associata direzione opposta causava uno stallo del personaggio.

```
while (gameLoop!=null) {

    if(keyLis.up) {pg.up();}
    else if(keyLis.down) {pg.down();}
    else if(keyLis.left) {pg.left();}
    else if(keyLis.right) {pg.right();}

    dm.getSessione().notifica();
```

Questo controllo di direzionalità conferisce una priorità non desiderata alle direzioni applicate. In questo caso il primo controllo annulla tutte le direzioni degli altri else/if. Non va bene, poiché se il tasto per la direzione up sta venendo premuto allora nessun'altra direzione verrà controllata. Se sto lasciando il tasto up mentre premo il testo right ci sarà un ritardo di tempo causato dalla meccanica della tastiera e dalla flessione delle mie dite. Io invece cerco un modo per applicare la direzione voluta a tempo di intuizione di gioco: appena il tasto voluto viene premuto. In questo modo il giocatore non si sente truffato dall'applicazione.

Quelli qui sotto di seguito sono i metodi del KeyListener che venivano utilizzati con questei controlli if/else.

```
@Override
public void keyPressed(KeyEvent e) {
    int keyCode = e.getKeyCode(); // ritorna l'integer keycode associato all'evento della tastiera

    switch(keyCode) {
    case KeyEvent.VK_W , KeyEvent.VK_UP : direction = Direction.UP; break;
    case KeyEvent.VK_D , KeyEvent.VK_RIGHT : direction = Direction.RIGHT; break;
    case KeyEvent.VK_S , KeyEvent.VK_DOWN : direction = Direction.DOWN; break;
    case KeyEvent.VK_A , KeyEvent.VK_LEFT : direction = Direction.LEFT; break;
    default: direction = Direction.NONE;
    }
```

```
    @Override
    public void keyReleased(KeyEvent e) {
//      direction = Direction.NONE;

//      up = down = left = right = false;

        int keyCode = e.getKeyCode(); // returns the integer keycode associated with the key in this event

        switch(keyCode) {
        case KeyEvent.VK_W , KeyEvent.VK_UP : up = false; break;
        case KeyEvent.VK_D , KeyEvent.VK_RIGHT : right = false; break;
        case KeyEvent.VK_S , KeyEvent.VK_DOWN : down = false; break;
        case KeyEvent.VK_A , KeyEvent.VK_LEFT : left = false; break;
        default: break;
        }
    }
```

```
if(keyLis.getDirection()!=Direction.NONE) {
    switch(keyLis.getDirection()) {
    case UP : pg.up();break;
    case RIGHT : dm.getSessione().pgRight();break;
    case DOWN : dm.getSessione().pgDown();break;
    case LEFT : dm.getSessione().pgLeft();break;
    case NONE : break;
    default : break;
    }
}
```

- All’inizio si trattava di un unico if… else if ... else if… cioè di una catena unica di if seguito da tre else if che controllava quale direzione fosse stata premuta (un if per ogni direzione possibile delle quattro disponibili). Questo approccio così elementare imponeva austerità nel movimento: il personaggio poteva cambiare direzione di movimento solo se nessun tasto in relazione ad altre direzioni fosse stato premuto. In termini di gioco capita spesso di mantenere un tasto direzionale premuto e di premere il successivo *mentre* si sta togliendo il dito dal primo. Questo fatto, poiché non funzionava, era frustrante. L’idea di separare la catena di 4 controlli in due catene da 2 controlli ciascuno (sue giu in una, destra e sinistra nell’altra) migliorava un poco la situazione, ma non risolveva il problema di fondo. Ho capito alla fine che quello che cercavo di ottenere non era altro che una pila, mentre io stavo applicando erroneamente una coda decisa dalla successione delle righe di codice. Quindi, mentre il KeyListenerMenu fa del Getter di Scarico il suo cavallo di battaglia, il KeyListenerGame trova il motivo della sua esistenza nella sua pila di KeyMemo. Ho incapsulato ogni direzione generata dalla pressione del suo tasto in un oggetto statico e ho fatto in modo che il Getter del KeyListenerGame restituisse il valore in cima alla pila dei tasti direzionali (premuti in sequenza). In cima c’è l’ultimo tasto che si sta premendo. Il Getter non elimina tale direzione una volta che la ha resitutita. La espone soltanto. La direzione viene tolta dalla pila solo se il suo tasto viene rilasciato dall’Utente. Questo Getter espositore di pila permette di ottenere una responsività più simile a quella dei videogiochi reali: si possono mantenere premuti più tasti direzionali: quello che conta è l’ultimo immesso nella pila. L’aver incapsulato le direzioni possibili in oggetti KeyMemo (classe innestata nel KeyListenerGame) apre orizzonti nuovi: sarebbe possibile registrare il numero di volte in cui un tasto è stato premuto o riconoscere se un tasto è sia stato premuto un numero limite di volte perché si scateni una certa condizione. L’atto di incapsulare in un oggetto una situazione gli conferisce dignità: disporre di una identità su cui operare è un principio cardine che ho ritrovato alla base di molti Design Pattern (come l’ Iterator, lo Strategy, …) , l’atto di dare un nome ad una cosa permette di chiamarla quando serve (e di richiamarla, invocarla in altre righe di codice).

(g) TESTING

- Il Testing. Tutti i bug che ho trovato li ho risolti, tranne uno: il disinnesco di una bomba di tipo Remote che avviene quando un nemico di tipo BombEater mangia una bomba deposta successivamente a quella. Questo fatto tuttavia non mi dispiace che accada. Alcune situazioni impreviste hanno risvolti positivi. Il funzionamento delle bombe Remote è piuttosto godibile. È stato inserito verso la fine del completamento dell’applicazione ed ha messo in luce il fatto che tutto sommato sarebbe stato progettualmente intelligiente disporre di una collection all’interno del PlayerCharacter delle bombe deposte. Questo fatto causerebbe tuttavia un refactoring importante. Un bug particolare che è comparso solo dopo diversi minuti di gioco durante la fase di testing ha riguardato un errore scaturito dal KeyListener. La gestione dell’eccezione tuttavia non impediva il gioco (veniva notata solo su consolle di Eclipse). La situazione è stata comunque risolta. Alcuni bug sono comparsi solo a seguito di particolari situazioni: avere una applicazione già sviluppata rende più lento il processo di testing. Più l’applicazione diventa complessa, più è necessario spendere tempo, impiegare risorse, inventare dei modi e delle strategie per mettere in campo una situazione efficiente di testing. Ad esempio, studiare il comportamento dei Boss è risultato più semplice se temporaneamente la Boss-Fight è posizionata al primo livello dello Stage:

questo atto che sembra banale richiede il riposizionamento di diversi elementi e righe di codice (nelle Factory ad esempio) e la necessità di essere in grado di riportare tutto alla situazione precedente alla fase di testing. Altro esempio, aggiungere un nuovo personaggio a quelli disponibili potrebbe essere fatto teorizzando una serie corretta di passaggi da eseguire. Insomma, all'aumentare della complessità dell'applicazione insorgono problematiche che erano in precedenza sconosciute, serve più tempo, organizzazione e comprensione di ciò che è stato scritto.

- **WORA (Write Once Run Anywhere).** JBomberman è stato sviluppato con Eclipse su un Sistema Operativo Ubuntu 22.04.3 LTS 64 bit e versione GNOME 42.9, windowing System Wayland. È stato testato e funziona correttamente anche sul sistema operativo Linux Mint (perfettamente) e su Window 11 (ha laggato un po', soprattuto all'inizio. Molto meglio su Linux).

(g) GENERARE IL JAR

- Il Jar. La generazione di un Jar di Jbomberman è ora possibile, ma ha richiesto un modo ben preciso di gestire l'ottenimento delle risorse audio e le immagini mediante il metodo:

  ImageIO.read(getClass().getResourceAsStream(#String che definisce il path dentro la cartella Res));

  Inoltre è stato necessario configurare correttamente la cartella Res contenente le risorse tramite la funzione Build Path presente in Eclipse.

DECISIONI DI PROGETTAZIONE RELATIVE ALLE SPECIFICHE

(1) Gestione del profilo utente, nickname, avatar, partite giocate, vinte e perse, livello.

Benchè questa fosse la prima specifica richiesta, ho trovato più congeniale nella mia situazione svilupparla nelle fasi finali dell'applicazione. Disponevo già di una astrazione funzionante di DynaSlaveCard e si è trattato di costruire una architettura collaterale nella View che potesse ospitare ciò che concerne il profilo di un plausibile Utente. Il DecoratorAccount e la sua forma concreta DecoratedAccount sono l'applicazione del Decorator Pattern al tipo Account presente nel Model. Account dispone di una miriade di campi, getter e setter ed incrementatori riguardanti le varie statistiche di gioco. L'applicazione di questi metodi è stata inserita nei punti strategici di ThreadGame. Game Model si comporta come una base di dati, disponendo della collection di Account correnti. Questa collection viene salvata in un file esterno mediante Serializzazione, al contempo è possibile caricare un file di questo tipo per conservare i risultati ottenuti. La Serializzazione ha richiesto di implementare l'interfaccia Serializable a tutte le variabili d'istanza che componevano l'Account decorato. La decorazione ha associato mediante composizione l'oggetto Account all'oggetto AvatarAccount (una enum) che stabilisce quale oggetto AvatarUser utilizzare per l'animazione dell'Account.

Le interazioni tra Model e View avvengono nel rispetto di MVC e di Observable-Observer. L'utente vede la View e vi interagisce utilizzando in verità i Listener del Controller, i quali manipolano la collezione di Account del Game Model, i quali metodi causano un cambiamento di stato

nell'Observable e quindi notificano gli Observer. In questo contesto l'Observer attivo è il Master che aggiorna le versioni presenti nella View di tale Collectione dell'Account Corrente.
Alcune specifiche, come la possibilità di settare un nuovo Account come Account corrente nella View ha reso necessario che la View esponesse al Controller dei valori booleani che identificassero se fosse necessario notificare la View con i valori correnti del Model.

(2) Gestione di una partita completa con almeno due livelli giocabili, due tipi di nemici con grafica e comportamento di gioco differenti, con gestione del punteggio, delle vite, di 3 powerups, schermata hai vinto, game over, continua.

(a) Entity: quelle che reagiscono e si muovono nella grid (di appartenenza del Model) e quelle che invece lo fanno nel Panel (di competenza della View).

Definisco Entity l'astrazione di un oggetto facente parte del gameplay e definita all'interno del Model. Il primo contesto di progettazione ha interessato il posizionamento di queste Entità di gioco. Ho postulato l'esistenza di due contesti di spazio: la Griglia (Grid), di appartenenza del Model, e il Pannello (Panel), li ho denfiniti a priori nel Modello. Il Pannello però trova la sua realizzazione concreta nella View. Ogni posizione, sia essa appartenente all'insieme delle posizioni possibili della Griglia o del Pannello, è identificata mediante un oggetto CartesianCoordinate che identifica un vero e proprio punto cartesiano. Le Entità sono di due sottotipi: Figure e GameCharacter. Le Figure esistono nella Griglia, i GameCharacter esistono invece nel Pannello. Chiaramente i due spazi interagiscono tra di loro, le Figure non si spostano, mentre i GameCharacter possono farlo. I GameCharacter ridefiniscono ad ogni loro ciclo la casella della Griglia nella quale si trovano, a partire dai valori di coordinate che possiedono nel Pannello. C'è dunque una corrispondenza implicita tra la Griglia e il Pannello, ovvero esiste una funzione suriettiva che mette in relazione ogni punto della Griglia con più punti del Pannello (cioè un insieme finito di punti nel Pannello). Tale relazione non è iniettiva, in quanto ogni punto della Griglia ha un insieme ben definito di punti del Pannello e tutti questi insiemi di punti del Pannello hanno intersezione nulla fra di loro, ovvero non esiste un punto del Pannello che sia riconducibile a più di un punto della Griglia (ogni punto del Pannello è riconducibile a uno e un solo punto della Griglia, mentre ad un punto della Griglia corrisponde un insieme di punti del Pannello). In termini di gameplay un punto della Griglia è esattamente una sua casella, mentre un punto del Pannello è un pixel, quindi in una casella esiste un insieme finito di pixel che sono quelli al suo interno.

Ho trovato utile fissare i parametri che definiscono la Griglia e quelli che definiscono il Panello in interfacce contenenti questi campi (intrinsecamente public static final) che sono rispettivamente MyGridFormat e MyJBombermanFormat. L'implementazione di tali interfacce permette alla classe che lo fa di accedere a queste disposizioni. Inoltre, in caso si voglia modificare uno di questi parametri lo si può fare molto semplicemente cambiando esclusivamente il valore all'interno della specifica interfaccia.

Ogni GameCharacter ha dei punti (CartesianCoordinate) che lo identificano in una determinata posizione del Pannello e della Griglia. Innanzitutto dispone di getGravityCoord() : CartesianCoordinate che restituisce il punto del Pannello che definisce, tramite una conversione, la casella della Griglia nella quale si trova. Per il PlayerCharacter tale conversione viene effettuata mediante il metodo updateActualXYGrid() del DynaManager nel Controller, mentre per gli NPC(NonPlayableCharacter) il discorso è un poco più elaborato: ho ideato un NpcMediator che gestisce il movimento degli NPC tramite un Visitor (il VisitorCharacterMover) che dispone del metodo cartesianGridConverter().
Il metodo getActualSolidPoints() : CartesianCoordinate restituisce invece il Set dei punti che caratterizzano il rettangolo la cui area definisce lo spazio in-game che occupa quel personaggio.

Questo metodo è importante soprattutto per controllare se il PlayerCharacter si trova all'interno dell'area di un NPC (in tal caso verrebbe scatenato l'effetto di risonanza di quell'NPC nei confronti del PC). Tale area può essere intesa come la collision-box di quel personaggio.
Un'altra famiglia di punti molto importante è quella dei projectedPoints ottenibili tramite analoghi metodi presenti nella classe GameCharacter e applicando determinati offset stabiliti dall'interfaccia MyJBombermanFormat nella View. I projected points permettono di stabilire dove si troverebbero due dei quattro punti dell'area che un personaggio avrebbe se la direzione su di esso applicata venisse concessa. Quali siano i due punti dei quattro possibili è un fatto stabilito dalla direzione in questione (la direzione è una delle quattro facce del rettangolo). Il movimento viene concesso se i due punti di proiezione non appartengono ad aree di solidità.

(b) Organizzazione di una Sessione di Gioco

Il secondo contesto di progettazione ha definito cos'è una sessione di Gioco. Ho studiato per diverso tempo come si svolgeva il gameplay reperibile su youtube di Super Bomberman. Si tratta di una successione di otto livelli in cui la scena interessa l'eliminazione di un gruppo di nemici oppure di un unico nemico, più forte (un Boss). Ho deciso di organizzare la mia sessione di gioco chiamandola Stage. Ogni Stage è costituito da otto livelli, l'ottavo è quello in cui si presenta il Boss. Ogni livello lo ho chiamato Scene. In questo modo uno specifico livello è identificabile come un oggetto Theater il quale è composto da un oggetto Stage e un oggetto Scena. Theater è una enum, in questo modo tutti i livelli sono unici e ben distinguibili. Ho pensato all'idea di Teatro poiché la prospettiva di gioco mi ha dato l'impressione che l'Utente sia come seduto su una poltrona intento a guardare un'opera Teatrale, infatti i personaggi sono figurati in prospettiva centrale ed è solo la parte inferiore della loro sprite a definirne l'area di appoggio. Possedere la definizione di Theater ha tra l'altro permesso di gestire ordinatamente le Factory nel Controller.

(c) La gestione della Griglia di gioco

Gli elementi della Griglia sono le Figure. I sottotipi di Figure sono: TesseraAbstract e Prop.
La Griglia di per sé è una matrice di array di 13 colonne e 11 righe. Gli elementi di cui è composta sono le 143 TesseraAbstract (le ho pensate come delle maioliche). Le Tessere si comportano come dei wrapper speciali, l'oggetto che avvolgono deve essere di tipo Prop e viene inteso come l'oggetto bundle della Tessera. La capacità delle Tessere è quella di manifestare le caratteristiche dell'oggetto wrapped, se presente. Poichè ogni Figura dispone di booleani che ne definiscono le proprietà, una molto importante è la solidity. Ad esempio, ogni Tessera ha solidity false decisa in fase di costruzione, ma se ospita un bundled Prop allora manifesta come valore di solidity quello del wrapped. La solidity rende falsi i controlli effettuati sui projected points (impedendo quindi i movimenti verso aree di solidità). Prop sta per Theatrical Property, poiché un Prop è pensato essere un oggetto di scena di natura effimera. Mentre le Tessere rimangono fino alla fine del gameplay i Prop possono essere presenti e poi venire tolti (es. Obstacle), essere messi in fase di gameplay (es. Fire), essere teoricamente sempre presenti (es.Wall) o comunque distruttibili solo grazie a particolari poteri [In questo caso in ipotetiche versioni future del gioco][I Wall sono indistruttibili durante il gameplay ma possono venire messi o tolti durante la fase di creazione di un Livello. Questo contesto riguarda quelli che ho chiamato ExtraWalls]. Tutte queste entità sono decorate nella View e la sprite da disegnare riferita ad una Tessera segue un ragionamento analogo a quello del bundle, ovvero viene disegnata la sprite del Prop in bundle se presente, altrimenti quella della Tessera.

(d) Risultati Concreti

Ho sviluppato 3 Stage ognuno di 8 livelli, in cui ogni ottavo è una Boss fight. Ogni Stage ha una sua specifica tassellatura, definita mediante i sottotipi della classe Arches. Ogni Stage ha le sue sprite contestuali e un equilibrio nei gruppi di nemici stabilito a seconda della difficoltà ad essi assegnata. La particolarità a mio avviso più interessante dei vari livelli è la presenza degli ExtraWall. Ogni livello dispone di regular Wall, cioè di oggetti Wall in bundle con le Tessere alle posizioni di riga e colonna dispari. Per questo motivo i regular Wall sono teoricamente 30. Ad essi si aggiunge un numero casuale (ma il cui range di casualità è stabilito all'interno della Factory dei Level a seconda dello specifico Theater di riferimento) di Wall mediante un algoritmo di backtracking ricorsivo che evolve concettualmente da quello visto a lezione (Teseo e il Minotauro) e quello presente allo Scritto di Settembre, ma al quale si aggiungono ulteriori metodi ricorsivi di appoggio per poter posizionare sempre in modo casuale e ricorsivo il gruppo di NPCs di quello specifico livello in modo che al playerCharacter sia possibile raggiungere la Tessera di uscita e ogni NPC presente in gioco. Infatti per vincere è necessario posizionarsi sopra la tessera di uscita, senza bundle, dopo aver eliminato tutti i nemici. Per fare in modo che ciò sia possibile ogni nemico deve essere in grado di raggiungere l'uscita. In sostanza, il playerCharacter e ogni NPC si danno appuntamento all'uscita durante la ricorsione. Quando la densità di extrawall è eccessiva c'è anche un meccanismo che riduce il numero di wall presenti nella nascente griglia. Alla fine di questi processi si stabilisce la blackened delle Tessere in base al risultato finale (quelle che devono essere ombreggiate o meno). Complessivamente è stata una salvezza applicare il Factory Pattern per incapsulare tutte queste procedure in un unica classe, spazialmente vicine. Il risultato finale è che rigiocare uno Stage può mostrare griglie con murature differenti ogni volta, in questo modo miglioro un poco la longevità del videogioco.

Ho sviluppato 9 powerups capaci di modificare il personaggio giocante, le sue bombe o i nemici in gioco. Un discorso importante da applicare sia ai PowerUp sia agli NPC è l'utilizzo delle classi anonime. Ho utilizzato un factory method nella classe astratta dei PowerUp per fare in modo che sia possibile ottenere PowerUp concreti solamente usando lo specifico metodo di generazione di un powerUp il quale si avvale di generare PowerUp tramite classi anonime a partire dalla classe PowerUp astratta. Ho ritenuto davvero comodo utilizzare le classi anonime in qeusto modo, mi sono risparmiato di dover scrivere 9 classi concrete in cui a tutti gli effetti riportare solo poche righe di codice. Le classi anonime mi hanno aiutato a mantenere vicini tutti i powerUps concreti. Inoltre ho usato le classi anonime anche nella Factory degli NPC allo stesso scopo. Dopotutto non ho dovuto aggiungere metodi alle classi concrete oltre a quelli già stabiliti nel loro super tipo.

Esistono 7 tipologie di Nemici la cui comparsa è diluita nel corso dell'avanzamento dei livelli. Queste tipologie sono:

1. Gregari. Si muovono lentamente e dispongono di una vita sola.
2. Standard. Si muovono più velocemente e dispongono di due vite.
3. Runner. Possono muoversi sopra gli Obstacles. Dispongono di due vite.
4. Bomb Eater. Eliminano le bombe non ancora esplose nelle loro direzioni possibili di movimento (quando si muovono).
5. Shooter. Sono nemici che possono sparare, ovvero rilasciare, Pyro.
6. Pyro. Sono una categoria speciale di nemici che vanno in una unica direzione, superando qualunque bariera solida e si esauriscono quando superano i confini della griglia di gioco. Rappresentano i "proiettili"/"dardi" dei nemici.
7. Boss. I Boss sono nemici dotati di un numero maggiore di vite, capaci di sparare molti Pyro, ma la loro caratteristica unica è quella definita dall'interfaccia Hound e nell'NpcMediator: quella di inseguire il playerCharacter durante le loro fasi di movimento.

Ogni NPC dispone di un Template Method chiamato doYourThing. L'implementazione dei metodi accorpati in questo è lasciata all'implementazione nella ereditarità dei vari sottotipi.

(4) Adozione di Java Swing.

Ho studiato diverse opzioni prima di arrivare a capire che il CardLayout faceva al caso mio. Usare un unico pannello all'interno del Frame e gestire in esso ogni possibilità di disegno avrebbe causato rapidamente problemi, si sarebbe creato un God Object di difficile comprensione e manutenibilità. Il CardLayout non è utile solo per quanto riguarda il mostrare il pannello attivo data una sequenza di pannelli, ma soprattutto per gestirne il focus, per incapsularne le dinamiche e per automatizzare la successione dei pannelli.

Dispongo di un unico JFrame che rimane lo stesso per tutta la vita della applicazione. Nel JFrame è posizionato un Jpanel che ho chiamato Master e il quale Layout è il CardLayout. All'interno di questo Layout ho tutti i Jpanel della mia applicazione che sono tutti sottotipi concreti di DynaSlaveCard, una astrazione nella quale ho definito tutti i metodi e i campi necessari a impaginare le opzioni e i pulsanti presenti. Per ogni concretizzazione di DynaSlaveCard esiste uno e un solo campo chiamato modulo che stabilisce il numero massimo del valore che può avere il puntatore delle stringhe di menù di quello specifico pannello. Se tale puntatore è maggiore del modulo della Card allora viene resettato a zero, se invece diventerebbe minore di zero allora viene valutato pari al modulo meno uno. Questa meccanica è gestita all'interno del Master, il quale, assieme a DynaSlaveGame (uno dei JPanel nel CardLayout), sono i due Observer di Game Model. L'applicazione viene inizializzata con Observer il Master, durante le procedure che si concludono con la creazione di un ThreadGame e il congelamento del ThreadMenu il Master viene tolto dalla lista degli Observer del Game Model e vi viene aggiunto DynaSlaveGame. Il tutto avviene tramite notifiche del Game Model. Quando la sessione di gioco termina, DynaSlaveGame viene tolto dalla lista degli Observer di Game Model e viene reinserito DynaMasterCard. C'è sempre dunque un solo Observer attivo in un dato momento. Tramite il metodo update dell'Observer attivo viene gestita la View. In alcuni casi si utilizzano i campi del Game Model raggiungibili grazie ai getter chiamati sulla istanza Singleton di Game Model. L'istanza di Game Model è stata salvata all'interno di una sua Nested Class, ovvero non si tratta di un Singleton Lazy (che inizializza l'istanza tramite la chiamata del metodo getInstance), bensì viene inizializzata staticamente la prima occasione in cui Game Model viene acceduto.

In base al modo tramite il quale ridimensiono la finestra di gioco e quindi disegno sul pannello Master del CardLayout è stato fondamentale arrivare a capire come ridimensionare le componenti del pannello. Le operazioni di disegno avvengono in un pannello di dimensioni standard il quale viene poi disegnato nel pannello Master secondo le proporzioni attuali del Jframe. Pertanto le coordinate iniziali di ogni operazione di disegno si riferiscono a valori standard, indipendenti dalle dimensioni del Jframe. Invece, le coordinate Bounds dei componenti devono essere ridefinite ogni volta che il Jframe cambia di dimensioni.

Mentre ThreadMenu viene eseguito, quando il Controller riceve un input da tastiera tale da dover modificare il puntatore attuale dell'opzione nel Jpanel attivo della View, il Controller modifica il puntatore salvato all'interno di GameModel usando un metodo dello stesso. Il metodo di Game Model sa come gestire un puntatore nella situazione in cui sia presente il modulo, quindi modifica il puntatore secondo l'esigenza attuale, ma tale metodo notifica anche gli Observer inviando l'oggetto ModelPointer attuale, nel qual caso avremo come Observer il Master. Il Master aggiorna dunque il valore del suo puntatore e la DynaSlaveCard attiva in quel momento lo disegnerà utilizzando tale valore.

Tra tutti i vari Layout per la gestione del posizionamento degli elementi grafici sicuramente il GridBagLayout è stato il layout su cui ho fatto più affidamento per definire i singoli pannelli.

(6) Riproduzione di audio sample.

La riproduzione di audio, sia esso musica o effetti sonori è completamente incapsulata in una classe, DynaSound. Ho deciso di utilizzare Clip per gestire i suoni, in particolare il problema riguardante l'accavallamento di suoni è stato gestito usando una Clip specifica per le musiche (da ripetersi in loop) e una Clip specifica per gli effetti sonori (da doversi esaurire). Era sorto un bug in riferimento alla riproduzione di più effetti sonori uno di seguito all'altro e in occasione di effetti sonori di durata inferiore al secondo che è stato risolto utilizzando un LineListener che aggiornasse la Clip in modo che si chiudesse correttamente nel momento in cui la risorsa audio fosse stata scorsa completamente.

(7) Animazione ed Effetti Speciali.

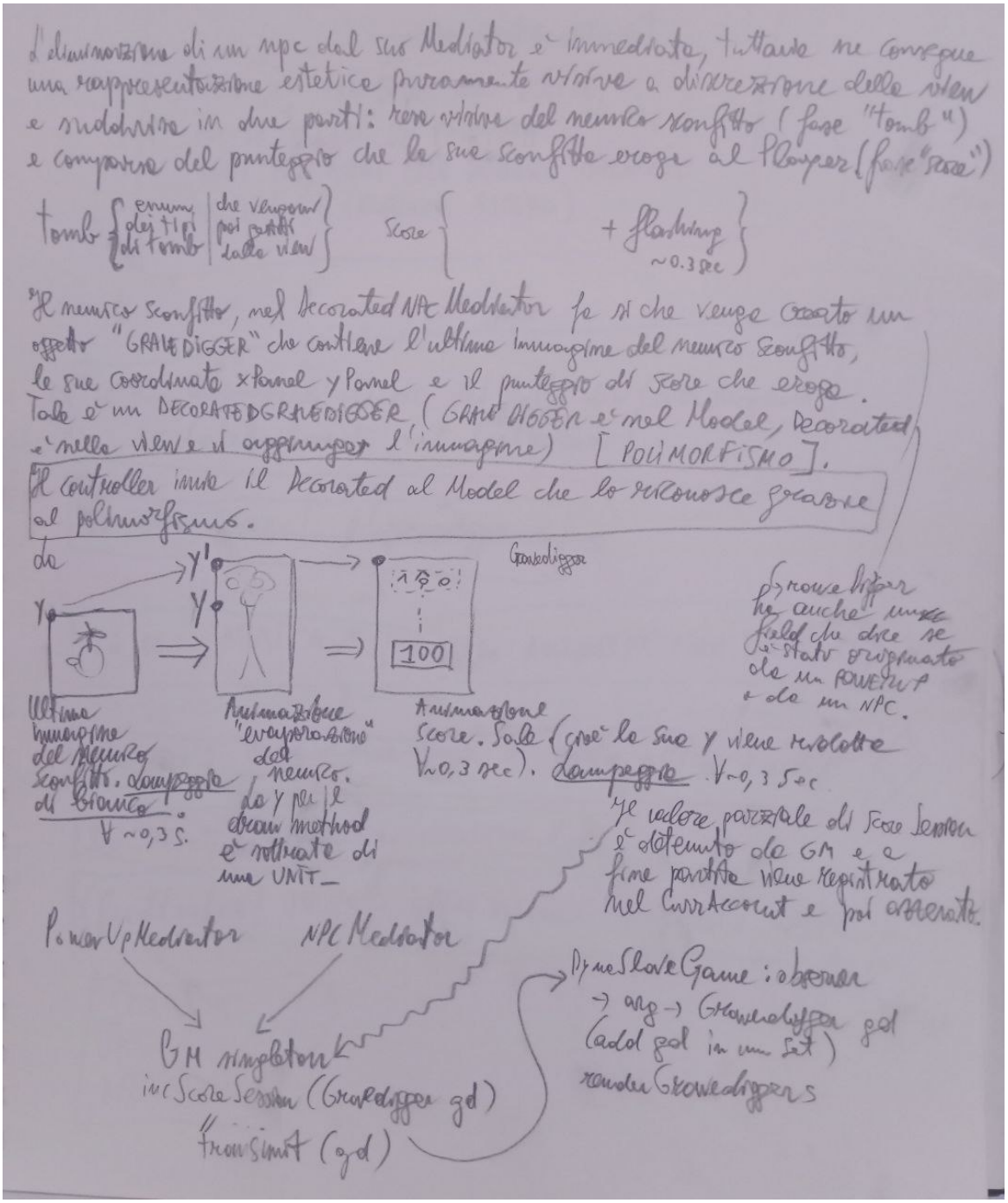
L'animazione dei personaggi e gli effetti speciali è stata ottenuta mediante l'uso estensivo della combinazione di Iterator Pattern e Decorator Pattern. È stato animato il dock della game session: il timer, l'acquisizione di punteggio, il contatore delle vite e dei tentativi del personaggio in quel momento. DynaSlaveGame incapsula queste dinamiche nei suoi metodi render.

È stato sviluppata la classe ColorTwister per applicare filtri ai colori delle immagini in modo da assomigliare alla versione originale del gioco. Ho scelto di applicare un filtro di grigi ai nemici (quando subenti un danno, ma non morenti), mentre un filtro sbiancante per il player che si attiva quando la sua modalità "invincibile" risulta attiva.

Gli Effetti Speciali non sarebbe tali se non fossero anche accompagnati da effetti sonori: questi vengono prodotti da una specifica Clip all'interno di DynaSound. L'applicazione dell'effetto sonoro avviene a seguito di una invocazione del metodo playEffect di DynaSound nei punti strategici delle classi della View o del Controller interessate a questo fenomeno. Un array è stato usato per gestire gli effetti sonori con l'idea che possa performare meglio essendo tali suoni molto brevi e spesso ravvicinati tra loro ed avendo incrontato alcune difficoltà nel rendere questa dinamica il più sincrona possibile con l'evento che la produce. Tutte le meccaniche sonore sono incapsulate in DynaSound, mentre le invocazioni dei suoi metodi si trovano nelle classi che ne fanno uso (sono solo i metodi play che vengono utilizzati al di fuori di DynaSound).

Le situazioni seguenti alla morte di un NPC generano invece oggetti specifici per gestire questi eventi, i Gravedigger. I Gravedigger lato Model gestiscono l'incremento del punteggio di Score del playerCharacter. Questo valore è registrato nel GameModel e solo alla morte di ThreadGame viene registrato all'interno dell'Account corrente. Le forme decorate di Gravedigger si occupano degli eventi grafici che conseguono al raccoglimento di un PowerUp (quindi alla comparsa a schermo del punteggio acquisito raccogliendolo) o all'eliminazione di un NPC. Il polimorfismo mi garantisce come sempre il corretto funzionamento di questi oggetti in package differenti: il supertipo astratto garantisce una interpretazione comune.
Nel caso dell'eliminazione dell'NPC si genera un Gravedigger per gestirne l'eliminazione lato grafico: porta con sé l'ultima sprite di quell'NPC e stabilisce secondo le sue regole interne il susseguirsi degli effetti scenici (le sublimations). Dopodichè lo stesso Gravedigger ne rilascia un altro per gestire il punteggio a schermo.
I Gravedigger generati alla sconfitta dei Boss sono lo stesso tipo che per gli altri NPC, questo è stato possibile generalizzando adeguatamente il codice all'interno di Gravedigger. È stato necessario

La collezione di Gravedigger attivi in un dato momento è in seno a DynaSlaveGame, all'interno del suo metodo update. DynaSlaveGame dispone di metodi contraccettivi nei confronti dell'eccezione di ConcurrentModificationException poichè i Gravedigger sono oggetto di iterazione continua. Tale problematica è stata risolta utilizzando array di appoggio e mettendo a disposizione una collezione (Set) a parte ove segnalare i Gravedigger da dover rimuovere.



DESIGN PATTERNS ADOTTATI

(3) Uso appropriato di MVC con Observer-Observable e altri Design Patterns.

(a) MODEL-VIEW-CONTROLLER (MVC PATTERN)

Come da requisiti è implementato il Pattern Architetturare MVC. È garantito: il Model non dipende da nessun altro package. Utilizza certamente Java.util, Java.io (ad esempio per la Serializzazione) e le Collections. Il Model è completamente riutilizzabile e riadattabili in altre applicazioni. Il Controller invece dipende in parte dal Model e in parte dalla View quindi è meno riutilizzabile. La View dipende dal Model e utilizza i Listeners che si trovano nel Controller. La View è l'aspetto più volatile e non riutilizzabile, ovvero altamente specifica, di questa applicazione.
Il Modello contiene tutti i dati correnti dell'applicazione, immagazzina le informazioni che è necessario depositare e costituisce la logica necessaria per regolare le interazioni tra le varie entità in gioco. Il GameModel nello specifico risulta essere il punto di riferimento per tutta l'applicazione, dove conrgono le informazioni da potersi ritenere valide in ogni momento.
Il Controller manipola i campi, i dati, del Model per conformarli alle richieste dell'Utente, utilizzando i metodi del Model nella maniera più opportuna. Il Controller ha la funzione di stabilire il ritmo nella logica applicativa, grazie a ThreadDelta, e grazie all'Observable-Observer tra Model (Observable) e View (Observers).
La View si occupa di gestire l'aspetto grafico e sonoro dell'applicazione in modo da essere il più semplicemente fruibile e piacevole per l'Utenza. La View disegna la grafica basandosi sulo stato e sui campi presenti nel Model, o perché presenti in un dato momento (utilizza il Model grazie al Singleton Pattern) o in seguito all'aver ricevuto una notifica mediante l'Observable GameModel.
I metodi update dei due possibili Observers: DynaMasterCard e DynaSlaveGame sono fondamentali per regolare la disposizione di componenti, le operazioni di disegno e gli aggiornamenti dei dati mostrati a schermo (prelevati dal Model).

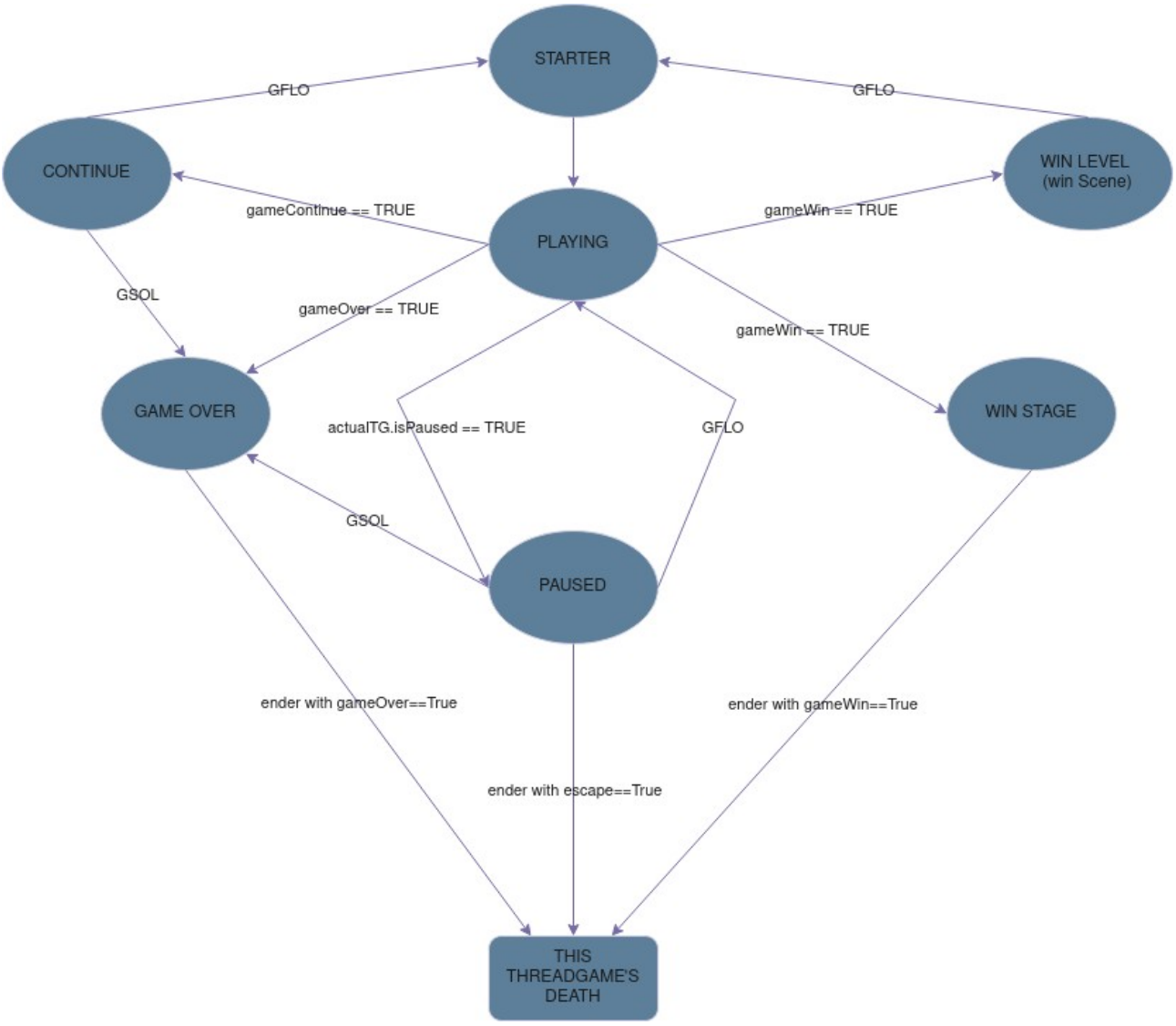
L'Utente percepisce gli output presentati dalla View, interagisce su di essa mediante i meccanismi dotati in essa e provenienti dal Controller. Il Controller controlla – per l'appunto – le modifiche degli elementi esposti dalla View e in base anche alle regole definite dallo sviluppatore chiede al Modello di cambiare stato e ne manipola i campi. Il Controller può a sua volta modificare alcuni aspetti della View (ad esempio l'effetto dei Commands sui Receiver che possono anche essere elementi della View). Il Modello notifica la View quando il suo stato cambia. La View infine riceve tali notifiche e a sua volta propositivamente chiede al Modello uali siano le informazioni corretti in un dato momento.

L'utilizzo di Observable-Observer all'interno di MVC permette di mantenere il Model complemtanete indipendente da View e Controller. Inoltre, potrebbe permettere in linea teorica di gestire molteplici View assieme, senza rischio di interferenze. È il vantaggio di una relazione Uno-a-Molti.

(b) STATE PATTERN

Dovendo sviluppare una architettura condizionale per il funzionamento del ThreadGame mi sono dovuto confrontare con la gestione di un flusso di informazioni In altri termini sono giunto alla conclusione che questa architettura potesse venire rappresentata mediante un Automa a Stati Finiti. Studiando i Design Patterns della Gang of Four ho scoperto lo State Pattern. Lo State Pattern è una variante dello Strategy Pattern, entrambi i patterns si basano sulla composizione. Lo Strategy incapsula una famiglia di algoritmi, dei comportamenti, invece, lo State Pattern serve a definire un Automa di Stati. Ogni Stato concreto si occupa di gestire, tramite implementazione dei metodi definiti nell' interfaccia dello State Pattern, alcuni metodi della classe di cui si vuole ottenere

l’Automa. L’Automa infatti demanda il risultato dei metodi il cui comportamento varia ai metodi dell’oggetto Stato Concreto che lo compone. Nel mio caso ho utilizzato lo State Pattern nel ThreadGame e nella DynaSlaveGame. Ho utilizzato una interfaccia presente nel Model, GameState, ma ho sviluppato in seguito delle classi astratte che la implementano per gestire gli Automi desiderati. Gli stati concreti da realizzare sono dichiarati tramite una enum nel Model, StateLabel. La forma concreta di questi è stata lasciata ai packages corrispondeti. L’astrazione di State Pattern per il ThreadGame nel Controller è StateAbstract, mentre per DynaSlaveGame nella View è State Panel. L’Automa vero è proprio è quello rappresentato in ThreadGame. Lo State Pattern in DynaSlaveGame serve ad emulare il comportamento dello stato corrente in ThreadGame, visibile in Game Model. DynaSlaveGame infatti muta lo stato corrente in base alle notifiche di GameState ricevute tramite Observer-Observable ed inviate dal GameModel. Il Controller infatti modifica lo stato corrente del Game Model quando il suo Automa effettua una transizione di stato.
È indubbio il vantaggio conferito da State Pattern nell’architettura generale dell’applicazione, non avrei potuto svilupparla altrimenti.
È demandato al DynaManager il compito di resettare il livello corrente e il riposizionamento del playerCharacter (nonché la restituzione di una vita) nel momento in cui si scelga di continuare la partita dopo una sconfitta.



GFLO (GameFirstOptionListener), GSOL (GameSecondOptionListener).

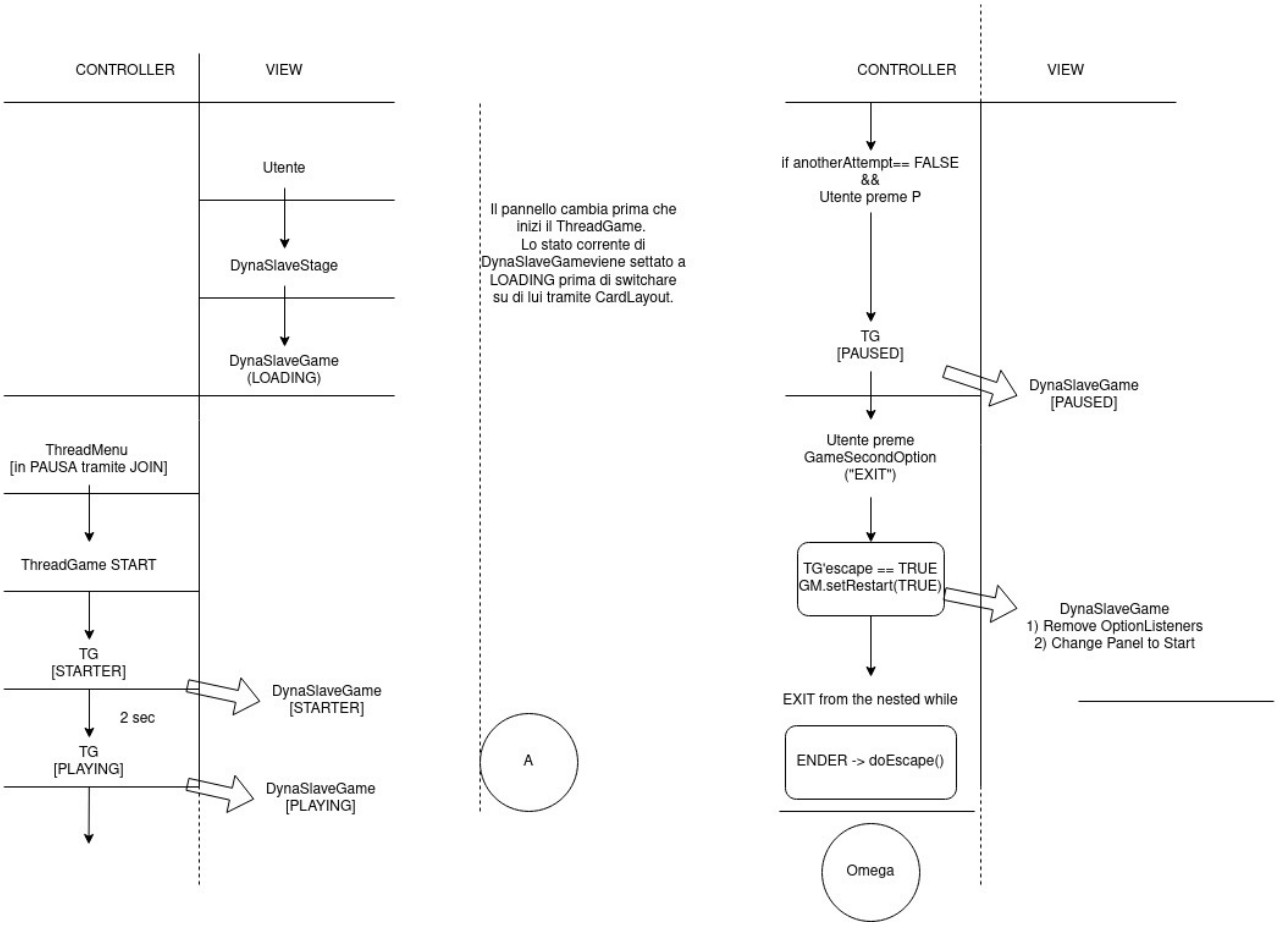
(c) OBSERVABLE-OBSERVER PATTERN

Tutte le comunicazioni tra package avvengono per mediazione del Game Model, unico Observable. Il Controller recepisce gli input dell'Utente (input generati a seguito di ciò che ha visto nella View) e richiama i metodi necessari del Game Model per aggiornare lo stato dell'applicazione. Il Game Model si comporta sia come informatore, sia come tesoriere delle informazioni da considerarsi attendibili (ovvero aggiornate più di recente) nell'applicazione.

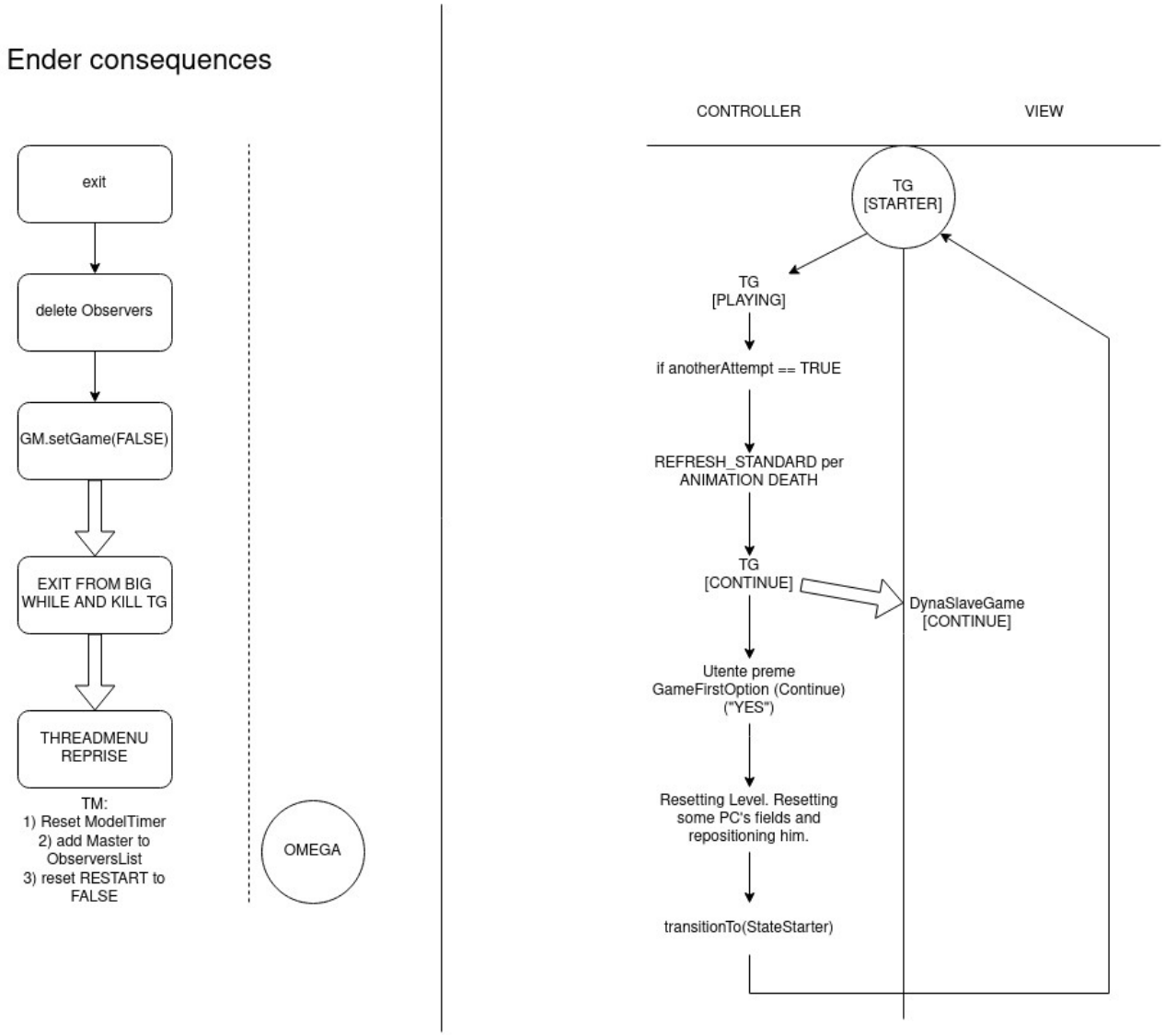
Observable-Observer dota il Model della capacità di ordinare alla View quando aggiornarsi. Le notifiche di GameModel ordinano alla View di ridisegnarsi. Il ritmo di ripittura è stabilito dall'intervallo tra le chiamate di questo pattern. È il Controller che sfrutta i metodi del Model (Observable) per aggiornare la View mediante le notifiche. Durante il contesto ThreadMenu è il Master a invocare repaint all'interno del suo metodo update. Durante il contesto ThreadGame invece è la DynaSlaveGame a invocare repaint nel suo update a seguito della ricezione di una notifica con arg di tipo Order e con label GAME_ON (questo accade ogni bar).

Observable-Observer gestisce gli stati della sessione di gioco. Quando l'Utente interagisce con la View per cambiare stato di gioco, il Controller carpisce questo segnale, cambia il suo stato e aggiorna la definizione di stao corrente nel Model. Questo viene eseguito tramite i metodi transmit del Model, che notificano come conseguenza tutti gli Observers inviando oggetti GameState. Quindi la View aggiorna il suo stato grazie a questa notifica, in modo da disegnare correttamente lo stato di gioco.

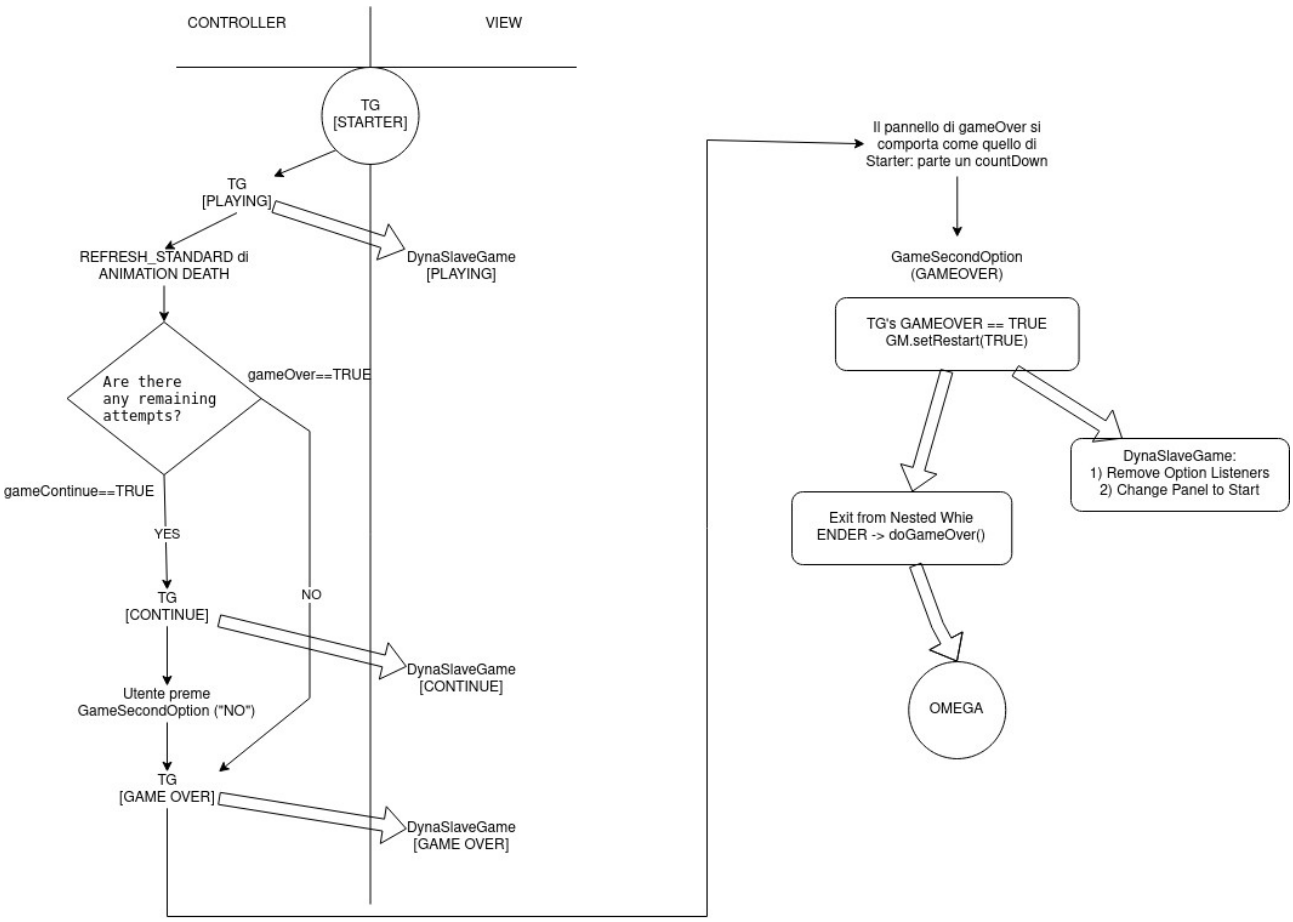
Questi sono i principali flussi di informazioni che vengono gestiti durante un ciclo di vita di un ThreadGame.



Questo è il Flusso di Informazioni dell'uscita con escape



A sinistra il flusso Omega,
a destra il Flusso per riprovare il Livello [CONTINUE = “YES” e ATTEMPTS>0]



Flusso per Uscire con Continue=”NO” o in caso di ATTEMPTS ESAURITI

*Observable-Observer gestisce l’attivazione/disattivazione dei Thread (ThreadMenu, ThreadGame) nell’applicazione.*Il GameModel dispone dei metodi per inviare un tipo specifico di oggetto, il tipo Order, per gestire ad esempio l’interruzione del ThreadGame e garantire il corretto passaggio, come Card attiva nel CardLayout, da DynaSlaveGame a DynaSlaveStarter.

Observable-Observer notifica l’avvenuto passaggio di tempo pari a un decimo di secondo al DynaMasterCard, mentre ogni bar durante ThreadGame alla DynaSlaveGame. Questo avviene grazie all’invio di ModelTimer nel primo caso e di un Order (GameOn) nel secondo caso.

Oltre a tutto ciò tramite Observable-Observer vengono inviate le informazioni aggiornate con cui la View disegna gli oggetti, ad esempio le coordinate reali del PlayerCharacter attuale.

I limiti di questo Pattern sono legati al fatto che gli Observer non possono limitare la ricezione di notifiche: è pertanto necessario un severo controllo di ciò che scatena notifiche e quindi che utilizza i metodi di GameModel (il Controller). Inoltre, il limite al significato di una notifica è racchiuso nel tipo che viene inviato e dunque riconosciuto come tale nell’update dell’Observer. Quindi ogni tipo è portatore di un significato ben preciso, per disporre di più tipi e quindi più messaggi è utile sfruttare le enum che garantiscono un insieme ben preciso di elementi che si possono usare come linguaggio.

Una importante considerazione da fare in merito al tipo di argomento inviato nelle notifiche è quella riguardante al suo tipo. Non è necessario che l’Observable conosca il tipo dell’argomento che inoltra agli Observers. Questo è verificabile osservando l’architettura del processo di animazione che prevede gli oggetti Animations, originari del package View, venire inviati dal Controller tramite le notiche del Model a DynaSlaveGame. Questo è un esempio limite di un oggetto della View che segue comunque il ciclo completo di MVC per arrivare ad agire sulla View stessa. Avrei potuto

spostare l'interfaccia Animate nel Model, giacché ha un significato prevalentemente di label pur essendo una interfaccia funzionale col metodo animate, tuttavia ho preferito non farlo: l'aspetto grafico è totalmente a discrezione della View. Per garantire l'unicità delle operazioni di animazione Animations è stato sviluppato come una enum, ogni elemento della enum è una implementazione concreta dell'interfaccia Animate.

(d) ITERATOR PATTERN

La prima idea di footage prevedeva un array contenete le BufferedImage delle Sprite e un campo che scorresse in ordine questo array di volta in volta aggiornandosi. L'intuizione è stato associare questo processo al funzionamento dell'Iterator e quindi incapsulare tutto questo fenomeno all'interno di tali oggetti. Questo è stato il primo Design Pattern che ho adottato all'interno di qeusta applicazione. Quando lo feci mi sembrò geniale, ora invece mi sembra la cosa più naturale del mondo: sarà diventata naturale col corso del tempo. La verità è che la necessità di avere un modo più maneggevole di gestire la sequenzialità delle sprite è sorta nel momento in cui ho studiato l'animazione di Bomberman. Le sue sprite non si susseguono come A, B, C bensì si comportano secondo l'ordine: A, B, A, C. Sarebbe stato davvero brutto aggiungere la stessa immagine ripetuta due volte all'interno dell'animazione del personaggio principale: non lo avrei mai permesso. Così è nato l'iterator che restituisce i valori posizionali dell'array nel modo in cui desidero, senza far ripetere le sprite. L'interfaccia FootageFuncs ha sostanzialmente l'obiettivo di far implementare l'iterator ai Decorator che devono essere animati.

A questo discorso si può aggiungere una parentesi riguardante il tempo che ha richiesto la modifica delle sprite di personaggi, nemici e ambientazione. Ho scoperto il mondo della pixel art. È sorprendente l'effetto che si possa ottenere guardando da lontano quella che sembra una macchia di quadrati colorati da vicino. Devono esserci persone che hanno dedicato la vita per fare questo. Ringrazio tutte le persone che hanno prodotto le sprite che ho utilizzato o modificato: ringrazio tutta la comunity di www.spriters-resource.com e tutti gli artisti dai quali ho attinto. Ho usato sprite provenienti da tutti i 5 bomberman e ad altre versioni di Bomberman. Senza il sito Piskel www.piskelapp.com non sarebbe stato possibile manipolare tutte le sprite, riadattarle. Ad esempio, la scritta iniziale BOMBER MAN era unita, ho dovuto separarle. La scritta JAVA nella pergamena è stata scritta a partire dalla scritta originale SUPER. La scritta SET delle opzioni è stata fatta modificando lettere prese dalle altre scritte. Le sprite per la tassellatura della griglia sono state riadattate per rispettare le proporzioni che utilizzo, così sono state riadattate nei colori e nelle proporzioni molte sprite dei personaggi e dei nemici, una per una. Ho testato diversi font prima di scegliere PixelBug.ttf, ringrazio la community di www.dafont.com.

(d) DECORATOR PATTERN

Ho imparato ad amare Java per la sua forte tipizzazione. Ciò che scrivo si comporta esattamente come ho deciso. Questo aspetto deterministico permette di ottenere risultati stabili. Questo lo dico contestualmente al Design Pattern che più di tutti mi ha messo a confronto con il polimorfismo: il Decorator Pattern. Il Decorator Pattern è una architettura logica in cui viene applicata la composizione all'interno di una classe tarmite un oggetto del suo stesso tipo. Questo permette di incapsulare i metodi originari dell'oggetto wrapped e restituire quindi una versione modificata degli stessi tramite il Decorator. Il Decorator ha un atteggiamento ricorsivo nei confronti dei metodi dell'oggetto che contiene, infatti il decoratore dispone dello stesso insieme dei metodi del decorato, essendo entrambi dello stesso Tipo. Ad esempio, un DecoratorPassword potrebbe essere un modo progettualmente efficace per criptare l'oggetto che contiene. Nella mia applicazione i Decorator permettono di utilizzare invece tutto il codice che è stato scritto nel Model e di ampliarlo con le richieste dell'ambiente grafico della View. Il Controller ha avuto come scopo principale anche

quello di fare da medium comunicativo tra Model e View: è nel Controller che ho posto le Factory che si occupano infine di ottenere le versioni decorate di oggetti prodotti a partire dal Model.

Per funzionare a dovere il Decorator deve estendere la classe che decora: questo serve a fare in modo che il decorator venga riconosciuto esattamente come quella classe quando vengono utilizzati i meccanismi di casting. Il tipo definisce l'insieme dei metodi che possono essere utilizzati, quindi è fondamentale per un decorator decorare la classe che contenga tutti i metodi richiesti dalle procedure di casting. Ogni funzione deve essere avvolta ricorsivamente alla sua controparte che viene richiamata tramite il componente del Decorator. Questo fatto deve essere fatto manualmente per ogni funzione. Solo così si può ottenere un mascheramento completo. Dimenticarsi di effettuare questo passaggio, giacché è da fare manualmente per ogni metodo, è stato fonte di disagio causando bug "nascosti". Questo accade perché se si aggiungono metodi ai tipi nel Model bisogna ricordarsi di aggiornare anche la controparte Decorator, altrimenti la funzione del Decorator si applicherà sui campi del Decorator e non su quelli del tipo del Model (componente).

Un altro erorre delle fasi iniziali di progettazione che mi costò caro fu quello di lesinare nei Decorator. Definire correttamente i Decorator senza doverne scrivere troppi è una sfida. Per quelle classi che vengono utilizzate e chiamate in molti metodi conviene fare un Decorator preposto, esattamente di quel tipo: così ho fatto per Bomb con DecoratorBomb. Per altre classi, come ad esempio per PowerUp, non ho creato un Decorator preposto, bensì ho sfruttato un più astratto DecoratorProp: questo comporta la necessità di disporre di una interfaccia, Powerupper, che sia la summa dei metodi appartenenti all'insieme dei metodi di PowerUp non presenti in Prop.
I Decorator mi hanno fatto pensare in modo insiemistico ai metodi riferiti ai Tipi. DecoratorProp è utile perché molti metodi che sono scritti in lui sono gli stessi che sono da riproporre qualora si volesse fare un Decorator di un'altro Prop (come quello di Bomb detto sopra). Infatti, c'è del codice ripetuto in DecoratorBomb che è anche presente in DecoratorProp. Tuttavia DecoratorProp non potrà mai essere un Bomb, a meno di non avere i metodi di Bomb in una interfaccia, come ho fatto per i PowerUp. Questo perché in Java non è possibile l'eredità multipla – tuttavia non è detto che applicare l'eredità multipla possa apportare dei vantaggi in tal senso, sarebbe cosa da sperimentare. Scrivere un unico decorator di un tipo astratto supertipo di altri tipi astratti che possa valere per le forme concrete di questi ultimi si è rivelato generalmente insoddisfacente. Ho lasciato questa soluzione solamente per i PowerUp, come ho detto sopra. Tale operazione richiede l'uso di interfacce che raccolgano i metodi extra dei figli astratti. Il problema è che queste interfacce raccolgono solo quelli ! Il casting dei Decorator infatti, in tale situazione, deve avvenire a tali interfacce e non alla classe effettiva che decorano. Questo risulta nella perdita del codice che il supertipo astratto più alto nella gerarchia avrebbe conferito ai suoi sottotipi, rendendolo a tutti gli effetti pressochè inutilizzabile, se non in maniera complicata (usando diversi casting), non immediatamente visibile allo sviluppatore e quindi anche rischioso. Ciò può compromettere il codice e sicuramente lo rende meno leggibile quando lo si utilizza nel Controller. Pertanto, ho deciso di fare un Decorator per ogni livello di astrazione dell'applicazione, ove le forme concrete di tali astrazioni lo richiedessero. Alla fine, al costo di un Decorator in più il risultato è stato un codice più facile da leggere.

Ho chiamato i sottotipi dei Decorator col nome Decorated, ma come si evince ad esempio dall'UML di DecoratorNonPlayableCharacter, ho sviluppato diversi sottotipi di Decorated in alcuni casi cercando il più possibile di evitare la ripetizione di codice.

(e) SIMPLE FACTORY PATTERN

Ho implementato 5 Factories nel Controller. Ognuna di esse si comporta come anello di congiunzione tra il tipo proveniente dal Model e la sua forma Decorator proveniente dalla View.
Incapsulare la procedura di generazione delle istanze necessarie al gameplay in classi specifiche mi ha permesso di sviluppare strategie di creazione abbstanza approfondite, in particolar modo nella Factory riguardante gli oggetti LevelAbstract.

Il Factory Pattern mi ha permesso di evitare di accopiare tramite dipendenza la classe di un oggetto dalla sua forma concreta. Questo è stato portato all'estremo mantenendo le classi astratte nel model e creandone una versione concreta mediante classi anonime nelle Factories del Controller.
Tutta logica di creazione e le regole che concernono quale prodotto rilasciare in quale momento vengone tutte incapsulate nelle Factory, rispettando la Single Responsability Principle (poiché tutta la logica è incapsulata in un oggetto specifico) e l'Open/Close Principle che in questo caso è tale in quanto l'introduzione di nuovi tipi all'interno del programma può avvenire senza dover disturbare il Client (nel Controller).

(f) BUILDER PATTERN

Il Builder Pattern è stato applicato alla creazione di istanze di classi concrete quali PlayerCharacterConcrete e LevelConcrete.

L'utilizzo di Builder Pattern ha nettamente migliorato la chiarezza espositiva delle Factory che lo utilizzano.

Grazie al Builder Pattern la costruzione di un PlayerCharacterConcrete non prevede l'utilizzo di un costruttore immenso, bensì è possibile selezionare le specifiche che interessano lo sviluppatore in quel momento per quel particolare tipo di prodotto.

La creazione di PlayerCharacterConcrete può avvenire step-by-step: è possibile evitare di specificare ciò che non serve e riproporre persino più volte di seguito talune specifiche (ad esempio il miglioramento del parametro Luck).

(g) SINGLETON PATTERN

Ho applicato il Singleton Pattern per quelle classi di cui è sensato che esista una unica istanza. Primo fra tutti il Game Model che è depositario di tutti i valori e i parametri correnti e validi dell'applicazione. Si comporta come una sorta di base di dati, in particolar modo per quanto riguarda la collezione di Account di cui dispone.

Il Signleton offre un ulteriore vantaggio: quello di poter disporre del riferimento dell'istanza desiderata chiamandola attraverso la sua classe, il cui accesso è regolato dalle dinamiche di visibilità dei packages e delle classi stesse.

Il Singleton Pattern è il Design Pattern che ho utilizzato più di ogni altro. Lo ho applicato sia secondo un approccio "Lazy initialization", sia mediante un approccio "Eager initialization". L'approccio "Lazy initialization" prevede che l'istanza venga creata solo nel momento in cui viene richiesta mediante il metodo getInstance() ovvero quando serve. Questo metodo tuttavia non è Thread-safe, pertanto per le classi Singleton in cui è importante il fattore Thread-safe (quali Game Model e DynaManager) ho utilizzato l'approccio "Eager initialization". Questo approccio prevede che l'istanza della classe venga creata quando la classe viene caricata in memoria dalla JVM. Il

riferimento all'istanza è definito staticamente tramite un campo statico o una classe innestata con un campo statico al suo interno.
Una architettura ispirata al Singleton Pattern è stata quella usata ad esempio in StateAbstract. Questa classe astratta è depositaria delle istanze Singleton dei suoi sottotipi concreti. Tali istanze vengono create ed assegnate alla prima chiamata del getInstance() dello stato concreto che invoca questo metodo. Oltre a ciò, getInstance() aggiorna il riferimento all'oggetto ThreadGame corrente.
Lazy Singleton è stato usato per GameFirstOptionListener e GameSecondOptionListener che sono i listener che vengono utilizzati per i pulsanti durante la sessione di gioco e sono attivati sia tramite pressione di tasti della tastiera, sia mediante mouse. Lazy Singleton è stato usato anche per KeyListenerMenu, KeyListenerGame.
Nella View invece è stato applicato il Singleton sulla classe Vocabulary, DynaSound e sulle forme concrete dei Mediator. Analogamente a quello che è stato fatto per StateAbstract (come detto sopra) la classe AvatarUser ha accorpato le istanze di AvatarUser possibili.

(h) STRATEGY PATTERN

Il rilascio di oggetti Bomb da parte del PlayerCharacter è il fulcro del gioco. Il comportamento del PlayerCharacter in merito al rilascio di un oggetto Bomb è stato incapsulato grazie allo Strategy Pattern. I diversi tipi di comportamenti di rilascio di oggetti Bomb sono una famiglia di algoritmi con lo scopo comune di restituire una forma polimorfica della classe astratta Bomb. Ho definito tre tipi di Comportamenti, ognuno incapsulato nella sua specifica classe: DropBombFaster, DropBombStandard, DropBombRemote. Questi comportamenti sono una implementazione concreta della interfaccia ReleaseBehavior. Grazie alla composizione, runtime, è possibile per l'istanza corrente di PlayerCharacter attiva in una sessione di gioco di cambiare il suo comportamento. In qeusta versione del gioco ho deciso di lasciare che solo un personaggio disponga del comportamento di DropBombRemote, ma mediante un PowerUp sarebbe possibile cambiare il comportamento di rilascio delle bombe sostituendo il comportamento corrente del PlayerCharacter con quello desiderato.

(i) PATTERNS IN JAVA SWING: COMMAND PATTERN E COMPOSITE PATTERN

Composite Pattern e Command Pattern sono intrinsecamente implementati all'interno di Java SWING e sono consoni alle funzionalità che deve avere la View.

Composite Pattern. L'interfaccia grafica, ovvero la View nella sua interezza, consiste di un JFrame composto da una varietà di Componenti Grafici quali JPanel, JButton, JTextField, JLabel, JList (e gli oggetti per personalizzarla), JOptionPane, JFileChooser, JScrollPane, JTable ...
La composizione è rappresentabile come un albero n-ario in cui ogni nodo ha come figli i suoi componenti. Le foglie infatti sono i componenti che non contengono (non hanno) altri componenti. Ogni Componente è dunque o un Composite di altri componenti o una foglia. Le regole di gestione dei componenti del Composite Pattern in Java Swing sono definite all'interno di questa libreria. Ad esse ho aggiunto i metodi per riallocare i componenti: i metodi relocate all'interno di DynaSlaveCard, relocateComponents() e relocateSpecialComponents(). Questi ultimi vengono utilizzati anche dai metodi in DynaFrame che si occupano di riposizionare gli elementi dei pannelli a seguito di un cambio di risoluzione dell'applicazione (normal screen ↔ fullscreen).

Command Pattern. Il mio compito è stato quello di implementare i Command concreti, ovvero i Listener, all'interno del Controller e quindi dotare i componenti della View di tali Listener durante le fasi di creazione dell'interfaccia grafica.

(j) TEMPLATE METHOD PATTERN

Il Template Method più elegante presente in Jbomberman è sicuramente quello definito nella classe ThreadDelta. Il Template riguarda il metodo Run. Il metodo primitivo del Template è quello che regola il calcolo del delta ed il susseguirsi delle bar che regolano il flusso del tempo. Gli altri metodi sono quelli astratti che devono implementarsi nei sottotipi. ThreadMenu segue fedelmente la formula definita in ThreadDelta, implementando i metodi astratti secondo le sue necessità. ThreadGame invece va oltre, poiché esso ha necessità di un ciclo while annidato in un while più generale per poter gestire correttamente i suoi stati, il metodo Run viene utilizzato come super.run() mentre la formula di Run viene sovrascritta con Override. La chiamata super.run può funzionare correttamente perché non è il metodo run a istanziare un nuovo Thread, bensì il metodo start (quindi chiamare super.run() non crea un nuovo Thread).
Un altro Template Method è il metodo doYourThing per la gestione del comportamento attivo in un dato momento per gli NPC.

(k) MEDIATOR PATTERN

Nel momento in cui mi sono trovato a dover implementare i contesti operativi dei PowerUp e degli NPC mi sono domandato innanzitutto come incapsularli. Per non dover trovarmi a gestire ogni cosa all'interno di DynaManager o di GameModel (ancora peggio). Entrambi i contesti presentavano un sistema in effetti ben chiuso, costituito da una Collection di entità ben precise. Studiando i Design Patterns della Gang of Four sono venuto a conoscenza di una implementazione di una chat mediante Mediator Pattern. Il sistema che regolano i miei Mediatori è molto simile a quello di una chat: vengono gestiti gli elementi presenti attualemente runtime nella griglia, in particolare vengono regolate le dinamiche di accesso di nuovi elementi a quelli presenti o alla eliminazione/uscita di quelli. Questo in analogia ad una chat. Gli oggetti che vogliono interagire con un elemento gestito da Mediator non devono preoccuparsi dello stato o addirittura dell'esistenza di tale target. Sarà il mediator a farlo. Il problema che viene risolto da questo pattern e che mi ha interessato maggiormente è stata infatti la riduzione delle dipendenze da Molti-a-Molti ad Molti-a-Uno.
Ho sviluppato quindi due Mediatori per gestire la collezione di PowerUp presenti sulla Griglia in un dato momento, e per gestire gli NPC presenti sul Pannello in un dato momento. Quindi situazioni a runtime. Gli oggetti della griglia o i GameCharacter che vogliono interagire con uno specifico PowerUp o con uno specifico NPC non agiscono direttamente su di esso, bensi inviano un messaggio al Mediatore preposto, informandolo della coordinata d'interesse. Il Mediatore gestirà la situazione. Il Mediator Pattern permette di incapsulare tutte le dinamiche e i metodi relativi a una specifica collezione di oggetti impedendo ad oggetti esterni di farlo direttamente. Si tratta dunque di un filtro alla operatività. Questo è risultato molto vantaggioso per compartimentare tali dinamiche nel videogioco.
I Mediatori hanno una differenza sostanziale. Il PowerUpMediator viene inizializzato con una collezione vuota all'inizio di una sessione di gioco, invece l'NpcMediator viene inizializzato con il set di NPC forniti dal LevelAbstract in quel momento attivo (quello corrente secondo il GameModel).
Inoltre, NpcMediator verifica anche se l'area di collisione di un personaggio si interseca con quella di almeno un NPC: questa condizione risulta essere un metodo alternativo alla letalità delle Tessere per danneggiare il playerCharacter.
Le astrazioni dei mediatori presenti nel Model favoriscono il riuso di codice, mentre le loro versioni concrete si trovano nella View ed effettuano Override dei metodi in cui si rende necessario l'utilizzo di tipi decorati.
Lo svantaggio dei Mediatori è quello di dover dotare gli oggetti che vogliono interagire che le entità che essi regolano di un riferimento ai mediatori. Per semplificare questa dinamica ho trovato molto utile il Sigleton Pattern applicato solo alle forme concrete nella View. Nel Controller queste

vengono usate per caricare gli oggetti originari del Model: essi le vedono basandosi sul loro supertipo astratto definito nel Model. Il Singleton mi permette di avere a disposizione a priori il riferimento ai Mediatori, senza dover passare il loro riferimento attraverso varie funzioni a catena.

Ho deciso di definire i metodi di render dei PowerUp e degli NPC nei rispettivi Mediatori. Il DynaSlaveGame, che si occupa di disegnare sul defScreen che verrà disegnato nelle corrette proporzioni dal DynaMasterCard, utilizza questi metodi. Questo incapsulamento è stato fatto per rispettare le dipendenze definite dal Mediator Pattern.

Esempio di Funzionamento di PowerUpMediator. Incapsulo la gestione amministrativa di inserimento, comparsa, applicazione dei PowerUp nel loro mediator. A seguito della distruzione di un Obstacle, durante il trespasser di Obstacle viene inviato un messaggio al Mediator in cui si specifica la coordinata (Grid) dell'Obstacle distrutto. Viene passata la corodinata alla funzione possibleSpawn la quale impone una certa probabilità (influenzata dalla Luck del PlayerCharacter corrente) che si aggiunga la coordinate a quelle papabili per essere degli spawn. Il metodo setPowerUp() mediante Stream converte ogni coordinata di spawn in un PowerUp. Questo mediatore dispone anche di un metodo a cui viene passato in argomento il PlayerCharacter e vi applica il PowerUp, se presente, alle sue coordinate attuali.

NpcMediator, tramite il suo metodo managementCore che riceve in input diverse informazioni dal Controller si occupa di aggiornare lo stato di tutte le entità (della Griglia ed il PC) calcolando tutte le interazioni tra di essi e gli NPC attualmente presenti.

(l) VISITOR PATTERN

Ad un certo punto dello sviluppo, poiché i metodi necessari a gestire il movimento del PlayerCharacter erano diventati molti e risultavano sparigliati in diverse classi la situazione era diventata confusa. Prevedendo che avrei dovuto sviluppare le meccaniche di movimento di altri sottotipi di GameCharacter (gli NPC, che non avevo ancora sviluppato) ho pensato ad un modo per incapsulare tutte queste meccaniche in un unica classe e al contempo mi sono domandato se fosse possibile disporre di un metodo la cui specificità potesse essere determinata durante lo svolgimento dell'applicazione in base al tipo di oggetto su cui applicare tale metodo. Ho studiato i Design Patterns della Gang of Four e ho trovato davvero soddisfacente il funzionamento di Visitor Pattern tanto da utilizzarlo per gestire le dinamiche di movimento di tutti i GameCharacter. Sostanzialmente, la classe a cui si vuole aggiungere una meccanica (un metodo) la si rende accetrice di un tipo Visitor, implementando una interfaccia preposta (VisitorAcceptor). Il metodo accept esegue il metodo visit dell'oggetto di tipo Visitor nel quale si passa come argomento l'istanza della classe accettrice. Il Visitor in questione nella sua interfaccia ha predisposto i metodi di visit in overloading con ogni tipo visitabile. La visita corretta viene attivata grazie al polimorfismo. Ho verificato che questo funziona anche qualora il visitor disponga di visite diverse per un oggetto che dispone di più tipi visitabili: è possibile definire il tipo della visita per tale oggetto mediante casting dell'istanza da visitare in argomento al metodo di visita.
L'aggiunta di un solo metodo (accept) ad una classe permette di dotarla teoricamente di una infinità di funzioni, incapsulate all'interno dei Visitor che le concretizzano. Questo espande le funzionalità di un programma all'inverosimile, tuttavia, ben lungi dall'abusare di ciò, mi sono concentrato a sviluppare un solo Visitor concreto, per gestire il movimento da applicarsi ai personaggi: VisitorCharacterMover. La visita di un GameCharacter controlla se la direzione corrente sia conforme alle solidità della griglia e possa quindi essere applicata. Se il controllo viene passato, allora il Visitor esegue il metodo move di quel personaggio.

NpcMediator dispone di una funzione che fa effettuare il moveimento a tutte gli NPC richiamandone visit ed usando il visitor apposito.

VisitorCharacterMover non solo dispone dei metodi per visitare PlayerCharacter, NonPlayableCharacter e Runner, ma è fondamentalmente il sito in cui vengono caricati i valori della View. L'eleganza di tutto ciò sta nel fatto che tutte le dinamiche del Visitor sono esclusiva del Model, non serve altro. Si può adattare a qualunque valore la View scelga. Chiaramente tali valori vengono caricati nel Controller dal ThreadGame. Ogni ThreadGame, nel suo stato PLAYING inizializza il suo Visitor.
Nel VisitorCharacterMover ho implementato una ulteriore dinamica a cui mi riferisco col nome di Camminata Assistita. Il concetto dell'algoritmo scatenato da una visita si basa sui punti di soldità del movimento qualora fosse applicato. I punti proiettati in quella direzione dal personaggio sono sempre 2 poiché sono i due vertici della faccia del rettangolo identificata dal punto cardinale riferito alla direzione in oggetto. Se nessun punto è solido allora il movimento accade. Se entrambi sono solidi il personaggio non si deve muovere, o meglio, lo si muove ma con velocità 0 (questo permette il prosieguo dell'animazione, tipica dei videogiochi di un personaggio che spintona un muro). Se solamente uno dei punti è solido allora accade il meccanismo della Camminata Assisitita per cui il personaggio viene si mosso, ma non nella direzione impressa, bensì nella direzione necessaria ad allontanarsi dal punto di solidità. Questa meccanica è visibile nel videogioco originale e benchè io non abbia mai avuto l'occasione di giocare la versione originale del gioco, penso che nella mia modesta replica questa correzione di movimento migliori di molto l'esperienza di gioco, poiché cerca di evitare, ridurre, la frustrazione che si ha quando il proprio personaggio si blocca contro un muro.

USO DI STREAM

(5) Utilizzo appropriato di Stream.

(a) FILTER FUNCTIONS

Il Pannello Statistics della View è quello che fa più ampio uso di Stream.
I metodi basati sugli Stream che utilizza si trovano nel Model e sono definiti dalla classe AccountFilter.

Il fatto che li abbia definiti completamente nel Model va a vantaggio del riuso di tale package nella View. Questi vengono utilizzati infatti per filtrare in vari modi la collections di Account presente nel Game Model e per calcolare dati statistici riguardanti i risultati complessivi. A questi scopi ho trovato gli Stream davvero congeniali.

Molto utili sono stati gli entrySet per poter ottenere uno Stream di un set a partire da una mappa (entrySet è sostenuto ("backed") dai dati presenti nella mappa).

(b) STREAM NEI MEDIATORS

Un'altro spazio in cui le Collezioni risultano essere protagoniste sono stati i Mediatori. Gli Stream offrono un modo molto coinciso per effettuare iterazioni (si può vedere un esempio di ciò in setPowerUps, metodo in PowerUpMediator) ed è possibile sfruttarli per possedere funzionalità di comodo (come sorted).

In particolare, NpcMediator sfrutta gli stream ad esempio mediante la funzione NobodyLeft(), che sfrutta noneMatch, o sendPosition che controlla se esiste almeno un NPC avente le stesse coordinate del PC e in caso positivo vi applica l'effetto di risonanza di quell'NPC.

(c) STREAM PER CONVERTIRE STRING IN BUFFERED IMAGES

Ho trovato un modo alternativo per convertire String rappresentanti il path per caricare una immagine nella BufferedImage corrispondente rispetto a quello che ho utilizzato nelle altre classi. Questo metodo si trova in DecoratedGravediggerScore e si chiama bufferedImagesConverter.

(d) STREAM PER ORDINARE I DISEGNI DEI GAME CHARACTERS

Lo Stream mi ha comodamente conferito un metodo per ordinare la collection di tutti i GameCharacter presenti sulla griglia. Questo metodo è renderNPCsWithPlayer ed è presente all'interno di EnjoyNpcMediator (la versione concreta della View del tipo Astratto NpcMediator presente nel Model). Questo metodo serve a disegnare nella prospettiva corretta i personaggi sulla griglia: il risultato grafico è estremamente efficiente, avviene senza che sia possibile accorgersene. L'ordine si è basato sull'override che ho scritto di compareTo nella classe GameCharacter. Questo ordina i GameCharacter basandosi principalmente sulla gravityCoords di ognuno (quindi si rifà al compareTo presente in CartesianCoordinate, a sua volta gestito con un altro override). L'ordinamento che si ottiene permette di disegnare correttamente i personaggi a schermo (PlayerCharacter e NPC) rispettando gli accavallamenti imposti dalla prospettiva del gioco originale. È importante sottolineare come questo risultato si possa raggiungere utilizzando anche un TreeSet dei GameCharacter presenti, il quale utilizza intrinsecamente il compareTo. Attenzione tuttavia ho posto nello sperimentare che se due GameCharacter risultano uguali tramite il compareTo allora non vengono inseriti nel TreeSet. Questo bug è stato difficile da capire, si era evidenziato durante una Boss-fight di testing in cui il Boss Clown non generava gli 8 Pyro in direzioni diverse bensì una successione di 8 tutti nella stessa direzione. Aggiungendo una riga all'interno del compareTo di GameCharacter questo bug è stato risolto. Poi, alla fine, ho deciso di utilizzare gli Stream per evidenziare tale ordinamento.

RISORSE DI STUDIO UTILIZZATE

Ho utilizzato inanzitutto le risorse offerte dal Corso di Studio, il forum, gli appunti, le dispense e i programmi visti a lezione e per esercizio, quelli fatti assieme in laboratorio, le lezioni del corso in presenza e quelle per la teledidattica.

Ho utilizzato i seguenti libri:

(1) "Il Nuovo Java" Claudio De Sio Cesari, HOEPLI informatica
(2) "Concetti di informatica e fondamenti di Java" settima edizione, Cay Horstmann, MAGGIOLI editore
(3) Java The Complete Reference Twelfth Edition, Herbert Schildt, Mc Graw Hill
(4) "Design Patterns" II edizione, Eric Freeman & Elisabeth Robson, Head First O'REILLY
(5) "Design Patterns Elements of Reusable Object-Oriented Software" E.Gamma, R.Helm, R.Johnson, J.Vlissides, ADDISON-WESLEY Professional Computing Series
(6) "The Unified Modeling Language Reference Manual", Ii edition, J.Rumbaugh, I. Jacobson, G. Booch, ADDISON-WESLEY
(7) "Effective Java", III edition by Joshua Bloch, PEARSON ADDISON WESLEY

Ho effettuato ricerche su internet, in particolar modo StackOverflow mi ha aiutato tantissimo. Ho approfondito Design Patterns sulla piattaforma Udemy mediante alcuni corsi specifici ed ho studiato il sito Refactoring Guru. Youtube mi ha fornito svariati video inerenti l'utilizzo di Java Swing, Design Patterns, Multithreading e inerenti la creazione di videogiochi.