UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE ED ELETTRICA E MATEMATICA APPLICATA


CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

**SOFTWARE ENGINEERING**

# DIEM INVADERS

GRUPPO 9

Carmen Fucile

Elena Guzzo

Edoardo Maffucci

Marco Preziosi

Stefano Saldutti

Salvatore Scala

Raffaele Squitieri

Antonio Tozza

Bruno Vento

ANNO ACCADEMICO 2019/2020

## Sommario

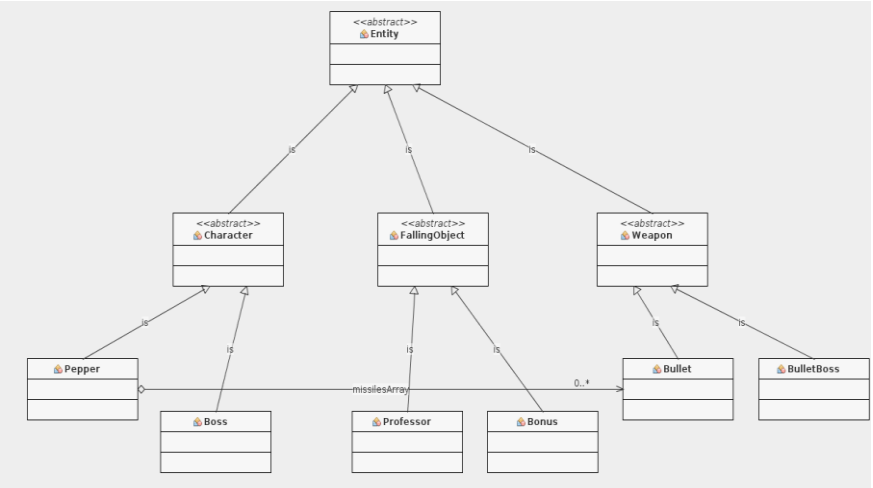
# Architettura

Per dividere il codice in blocchi dalle funzionalità ben distinte, l'architettura del gioco segue il pattern **MVC** (Model, View e Controller). In particolare:

- ***Model***: contiene i metodi di accesso ai dati.
- ***View***: si occupa di visualizzare i dati all'utente e gestisce l'interazione fra quest'ultimo e l'infrastruttura sottostante.
- ***Controller***: riceve i comandi dell'utente attraverso il View e reagisce eseguendo delle operazioni che possono interessare il Model e che portano generalmente ad un cambiamento di stato del View.

## Model

La classe base del Model è la classe ***Entity,*** la quale astrae ogni componente del gioco
Ogni oggetto *Entity* ha una coordinata x e y che servono per identificare la posizione del model nella finestra di gioco; un'altezza e una larghezza relativa all'immagine che viene utilizzata per visualizzare il model; infine, ogni *Entity* mantiene una lista di Observer e di una lista di interi che rappresentano un particolare stato durante la sessione di gioco.




*Entity* viene estesa da tre classi astratte: ***Character***, ***FallingObject*** e ***Weapon***.

*Character* è l'astrazione dei personaggi presenti nel gioco, ovvero *Pepper* e *Boss*. Tale classe aggiunge ad *Entity* gli attributi "health" e "dx" che rappresentano rispettivamente la salute e lo spostamento lungo l'asse x del personaggio.

*FallingObject*, invece, è l'astrazione di ogni oggetto che cade dall'alto, cioè *Professor* e *Bonus*. In tale classe vengono inseriti nuove variabili d'istanza come "y0" (che rappresenta la coordinata lungo y iniziale da cui comincia a cadere un oggetto) e "checkCollision", la quale vale true solo se l'oggetto collide con un altro. Inoltre, vi sono due variabili costanti FALLING_OBJECT_SPEED e FALLING_OBJECT_SPEED_UDPATE che rispettivamente rappresentano il valore iniziale per impostare la velocità e il valore utilizzato per aggiornare la velocità quando aumenta la difficoltà del gioco.
***Weapon*** è un'altra classe che astrae i missili che vengono sparati dai Character ed è estesa da ***Bullet*** e ***BulletBoss***.

Ciascun model astrae al suo interno la logica di "move" e, per i Character, anche di "fire".

### Controller

Tutte le classi che ricoprono il ruolo di Controller nel pattern MVC sono situate nel package controller.
Ciascuna di esse istanzia i relativi model e implementa l'interfaccia *Controller,* che dichiara un solo metodo update() e che serve ad aggiornare il rispettivo Model.

Inoltre, è presente la classe ***PlayerController*** che gestisce una **collezione di Controller** secondo il pattern Composite: nel momento in cui questa classe richiama il metodo update() (definito nell'interfaccia *Controller*, che agisce come interfaccia Component del pattern) , viene richiamato lo stesso metodo dei "Sotto-Controller" (Component objects) per aggiornare i rispettivi Model.
Inoltre, il metodo update() di *PlayController* aggiorna la *MainView* e richiama il metodo manageTime().
Questo metodo si occupa della gestione di alcune dinamiche del gioco legate al tempo. In particolare:

- fa in modo che la velocità degli oggetti che cadono aumenti dopo che siano trascorsi i secondi indicati da una costante quando il *PlayController* è abilitato (cioè dal momento viene premuto Start nel menu)
- fa in modo che i bonus scompaiano dopo un determinato numero di secondi definito da un'appropriata costante
- fa in modo che tutti i *FallingObject* scompaiano prima dell'entrata del *Boss*
- gestisce il cambiamento di immagine del modello *Pepper* quando lo scudo è attivo e resetta l'immagine quando termina il tempo di immunità
- aggiorna un set di variabili dopo la morte del *Boss*.

Un' altra importante funzione svolta dal *PlayController* è quella di aggiornare i model a seconda delle varie tipologie di collisioni che avvengono nel gioco e che sono notificate dalla View.

### View

La logica di presentazione dei dati all'utente viene gestita dalla View, che è stata realizzata mediante la libreria ***javax.swing*.**
La classe *GameFrame* rappresenta il frame principale in cui vengono incapsulate le singole viste (*MenuView*, *MainView* e *GameOverView*).

- ***MenuView*** costituisce il menù di avvio dell'applicazione e offre all'utente la possibilità di scegliere la possibilità con cui iniziare il gioco o di uscire da esso
- ***MainView*** rappresenta la vista principale del gioco in cui sono mostrati tutti i personaggi.
- ***GameOverView*** mostra la schermata di game over da cui è possibile accedere nuovamente al menù

## Pattern

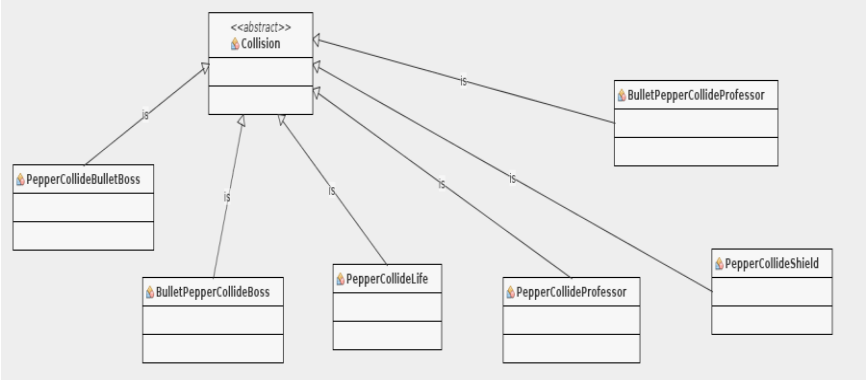
### Observer

La notifica dei cambiamenti di stato della View e del Model è realizzata mediante **il pattern dell'observer.**
In particolare, **la View è osservata dal Controller** al quale comunica le azioni di Pepper (dovuto all'input da tastiera) e le collisioni tra i vari oggetti presenti nella scena. Ricevute queste modiche, il **Controller attua gli**

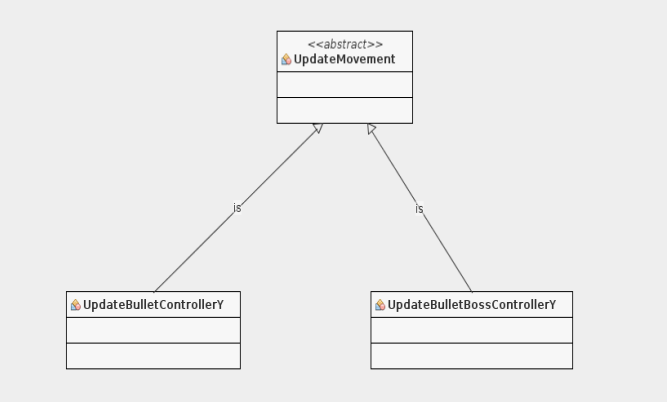
**appropriati cambiamenti del Model** (secondo la logica incapsulata in esso) e **sollecita l'aggiornamento della View.**
In alcuni casi, il **Model notifica la View** dei suoi cambiamenti in modo take da poter essere mostrati all'utente.
Nel package *Observer* è presente l'interfaccia *Observer* che estende *EventListener*, fornita da Java per la gestione degli eventi sulla GUI. **L'interfaccia *Observer* viene implementata da *MainView*** (che osserva Boss, Pepper tramite i Controller) e **da *PlayController*** (che osserva MainView).
La classe *StateChangement* estende *EventObject* (una classe che rappresenta un evento generico in Java) e contiene un array di stati.

## Template

Il pattern Template è stato sfruttato per gestire le **collisioni** tra gli oggetti presenti nella View.
In particolare, la classe astratta Collision definisce un'implementazione di default per collisioni tra più entità e dichiara un metodo hook che viene implementato nelle sottoclassi a seconda delle entità a cui fanno riferimento.




Inoltre, Il pattern Template è stato usato anche per **gestire il movimento** dei Bullet di Pepper e dei Bullet del Boss




## Composite

Come spiegato nel paragrafo dei Controller, la classe *PlayerController* che gestisce una collezione di Controller secondo il **pattern Composite**: nel momento in cui questa classe richiama il metodo update() (definito nell'interfaccia *Controller*, che agisce come interfaccia Component del pattern) , viene richiamato lo stesso metodo dei "Sotto-Controller" (Component objects) per aggiornare i rispettivi Model.

## Singleton

Il pattern creazione del **singleton** è stato utilizzato allo scopo di garantire che della classe PlayController venga creata una e una sola istanza, e di fornire un punto di **accesso globale a tale istanza**.