Relazione Progetto OOP

JBomberman

Componenti: Battistini Luca, Cangini Loris, Salvi Loris

L'intenzione era di creare una versione Java del famoso gioco giapponese Bomberman, creato dalla Hudson Soft.

L'idea di base è semplice, un giocatore controlla un personaggio all'interno di una mappa, di grandezza variabile a seconda del livello, con lo scopo di eliminare ogni singolo mostro o giocatore nemico, tramite l'utilizzo di bombe, all'interno della mappa. Quest'ultima è composta da tre tipi di blocchi: *unbreakable*, *breakable* e *walkable*, che condizioneranno l'intera partita del giocatore.
Tutti i blocchi del primo tipo limiteranno lo spazio di azione del giocatore, costringendolo a scegliere una direzione anziché un'altra, mentre il secondo imporrà al player l'impiego di una bomba per liberare il passaggio e continuare la sua missione. Il terzo, invece, differenzia lo spazio su cui è possibile camminare da tutto il resto.
Di particolare importanza è il secondo blocco descritto. Questo perché dopo la sua distruzione, può rilasciare un *power up*. Ogni *power up* modifca delle carattristiche del player o della sua bomba, ed ogni cambiamento può influire sia positivamente che negativamente.

Oltre alla modalità singleplayer, esiste anche la modalità multiplayer dove due o più giocatori possono sfidarsi nella stessa mappa.
Lo scopo di questa modalità è di raccogliere più power up possibili per poi avere un vantaggio sugli avversari e poterli sconfiggere.

Progettazione Architetturale

Abbiamo deciso di strutturare la fase di apertura del gioco tramite un menu a scelta, dove l'utente potrà scegliere la modalità (singleplayer o multiplayer) tramite le frecce della tastiera, inibendo l'utilizzo del mouse durante tutto l'utilizzo dell'applicazione.

Si è deciso di creare una classe GameStateManager che si occuperà di controllare e di mantenere attivo l'intero processo dell'applicazione.

Tra gli stati possibili c'è il LevelState, a cui verrà affidato il lavoro di gestire tutte le varie fasi del gioco, come ad esempio gli stati dei player, dei mostri o dei vari blocchi.

Abbiamo applicato sei differenti pattern per risolvere diverse problematiche di sviluppo, e sono:

- **Prototype**: permette la creazione di un nuovo oggetto partendo da uno di base. Il suo utilizzo è servito nella creazione del livello, dove vengono caricati e salvati tutti i tile una sola volta, poi, quando necessario, vengono clonati (mediante utilizzo di interfaccia *Cloneable*) per inserirli nelle celle che andranno a comporre

la mappa finale.

- **Command**: il concetto è rendere possibile la dinamicità dei destinatari che possono eseguire una certa azione, senza conoscere i dettagli di come tale azione verrà eseguita. È stato necessario il suo impiego in diverse situazioni, come ad esempio l'utilizzo del metodo kill(). In tale metodo, infatti, la classe Tile controlla il suo tipo e nel caso sia distruttibile eseguirà determinate azioni, il player si toglierà una vita se non è invulnerabile, etc... Dalla cella, che è il luogo dove vengono chiamati i kill() delle varie classi, tutto ciò è trasparente.
- **State**: permette all'applicazione di eseguire determinate azioni a seconda dello stato in cui si trova.

  Abbiamo strutturato un GameState che definisce l'interfaccia dei vari stati, rappresenta lo *state* del pattern.
  Questo verrà implementato da tutti gli stati, come Level1State e MenuState, che sono i nostri *concreteState*.

  Tutto ciò sotto il controllo del GameStateManager, il nostro *Context*, che si preoccuperà di mantenere lo stato attivo dell'applicazione e di reindirizzare eventuali eventi ad esso.
- **Template**: rende possibile creare un algoritmo comune tra due o più sottoclassi, permettendo, successivamente, a queste ultime di poter aggiungere o ridefinire parte del codice. Viene applicato in diverse parti del progetto, come, ad esempio, tra le classi MovableEntity, Player ed Enemy.
- **Singleton**: assicura che di una certa classe sia creata un'unica istanza e, rendendola accessibile all'esterno, se ne permette l'utilizzo.

  Nel nostro caso è servito fare ciò con la classe GameStateManager, perchè, effettivamente, non avrebbe senso creare più di un gestore per lo stato dell'applicazione.

N.B.

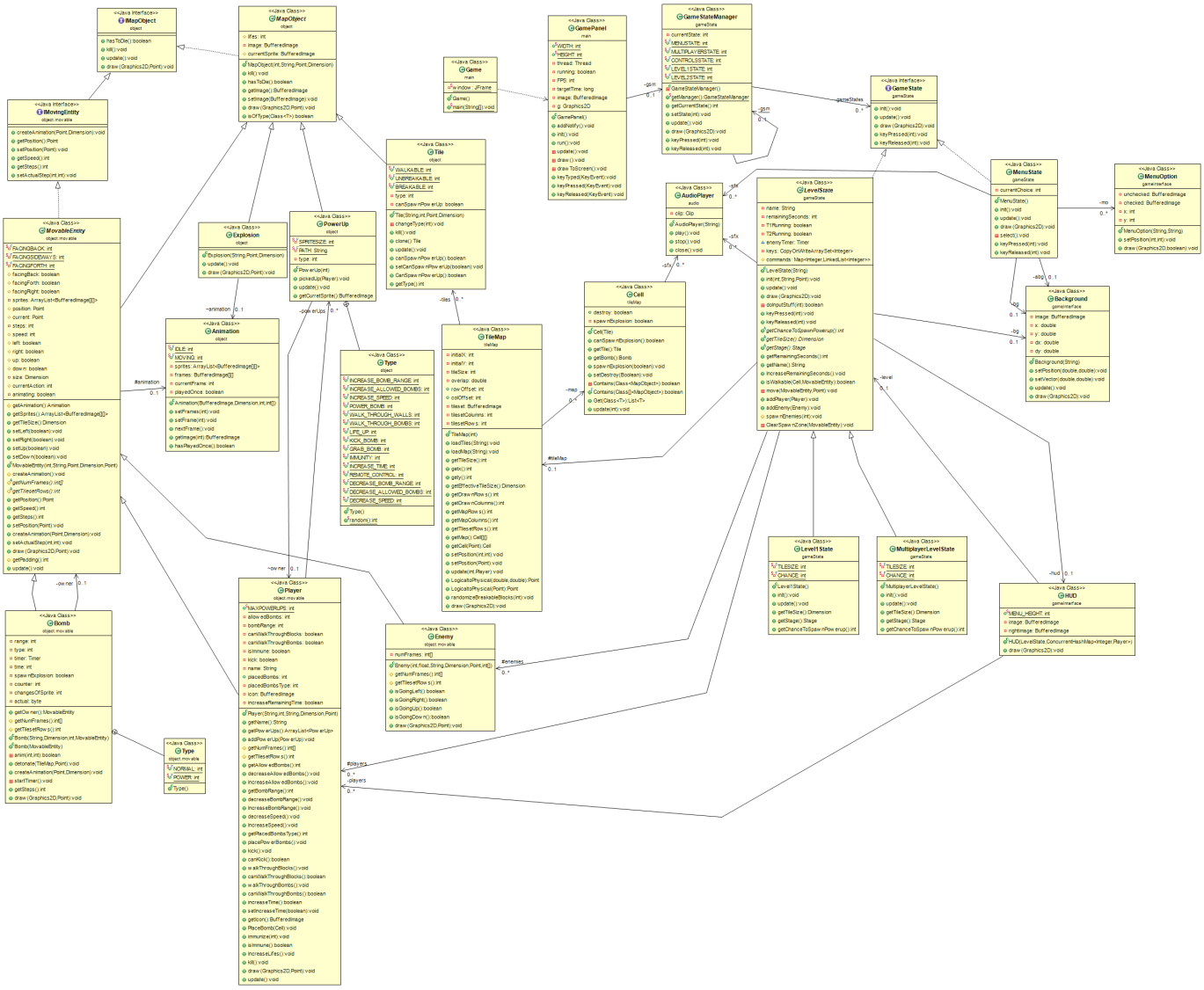
Abbiamo usato java8 per lo sviluppo della nostra applicazione, più precisamente abbiamo usato la versione ***jre1.8.0_25*** per quanto riguarda le librerie di sistema di JRE.

Per questo motivo risulta essere necessario anche ***JDK 8*** ed una versione di Eclipse che lo supporti (versione *Luna*).

Dopo aver ottenuto tutto ciò è necessario impostare Ecplipse in modo che usi il compilatore 1.8, una volta fatto sarà possibile usare le nuove funzionalità come le lambda expression (che abbiamo usato!).

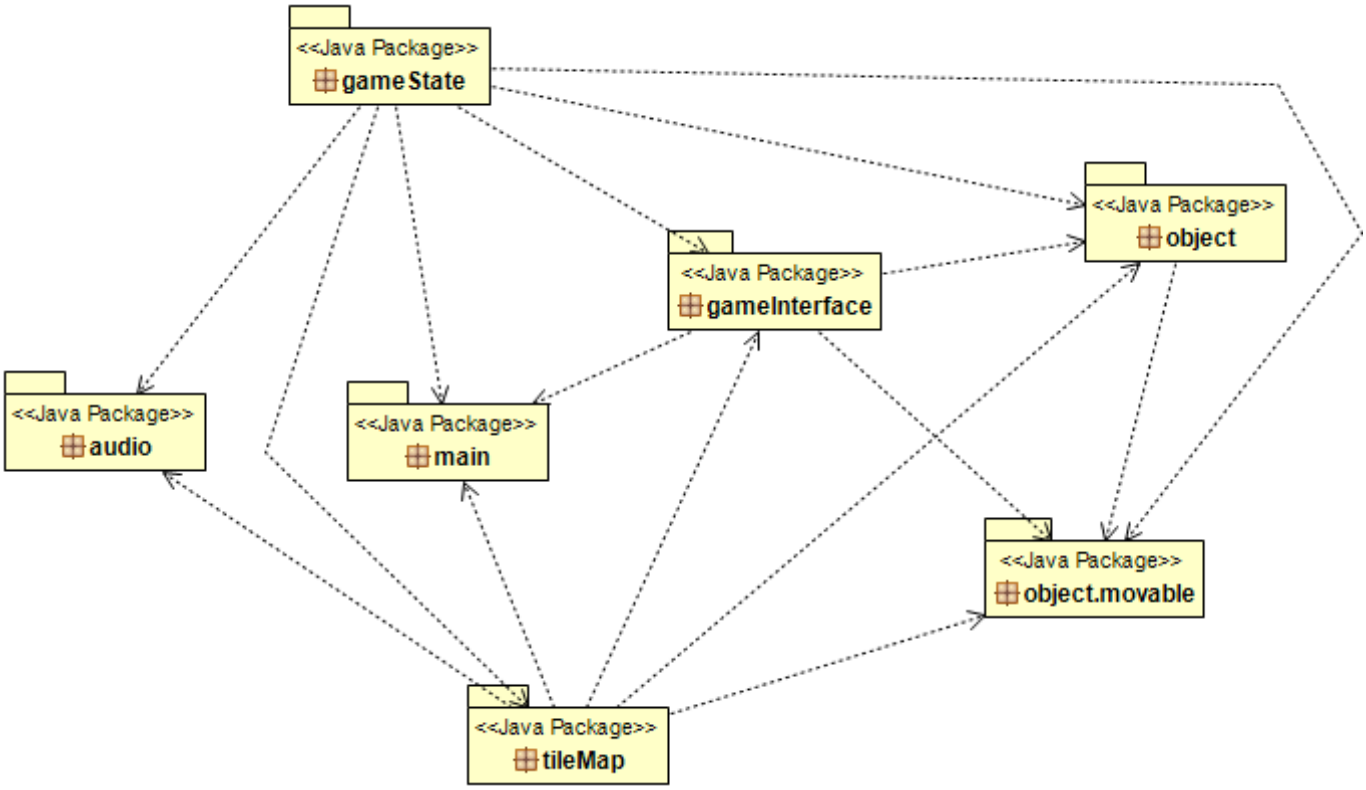
UML generale

Riportiamo di seguito il diagramma UML della nostra applicazione:

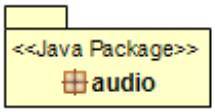


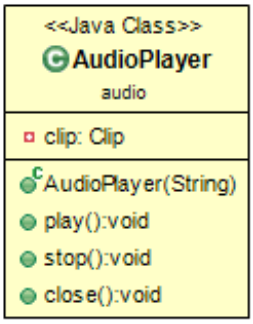
Organizzazione dei Package

Il progetto è suddiviso in vari packages come da schema seguente:



Passiamo ora ad analizzarli ed a spiegare il loro significato:





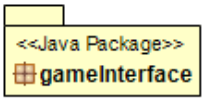
Il package audio contiene tutte le classi che si occupano di far partire degli effetti sonori in certi momenti dell'applicazione.

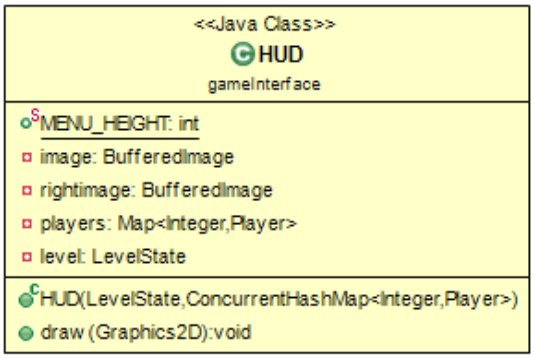
- AudioPlayer:

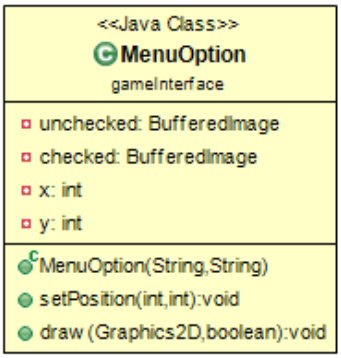
  Prende in ingresso una stringa col percorso del file audio da eseguire, poi, al momento opportuno, il suono potrà essere fatto partire tramite il metodo play().

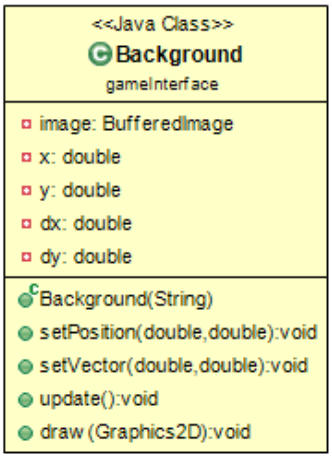
  stop(), come immaginabile, ferma l'attuale esecuzione e close() chiude il file audio, da questo momento, se sarà necessario rieseguire lo stesso suono

bisognerà riaprirlo.









Questo package contiene le classi utili a visualizzare sullo schermo componenti grafici

- HUD:

  Crea il menu ingame, dove verranno visualizzate informazioni relative alla partita in corso.

- MenuOption:

  Serve per creare le opzioni nel menu iniziale, *checked* ed *unchecked* sono 2 immagini e rappresentano rispettivamente la versione selezionata e non selezionata dell'opzione.

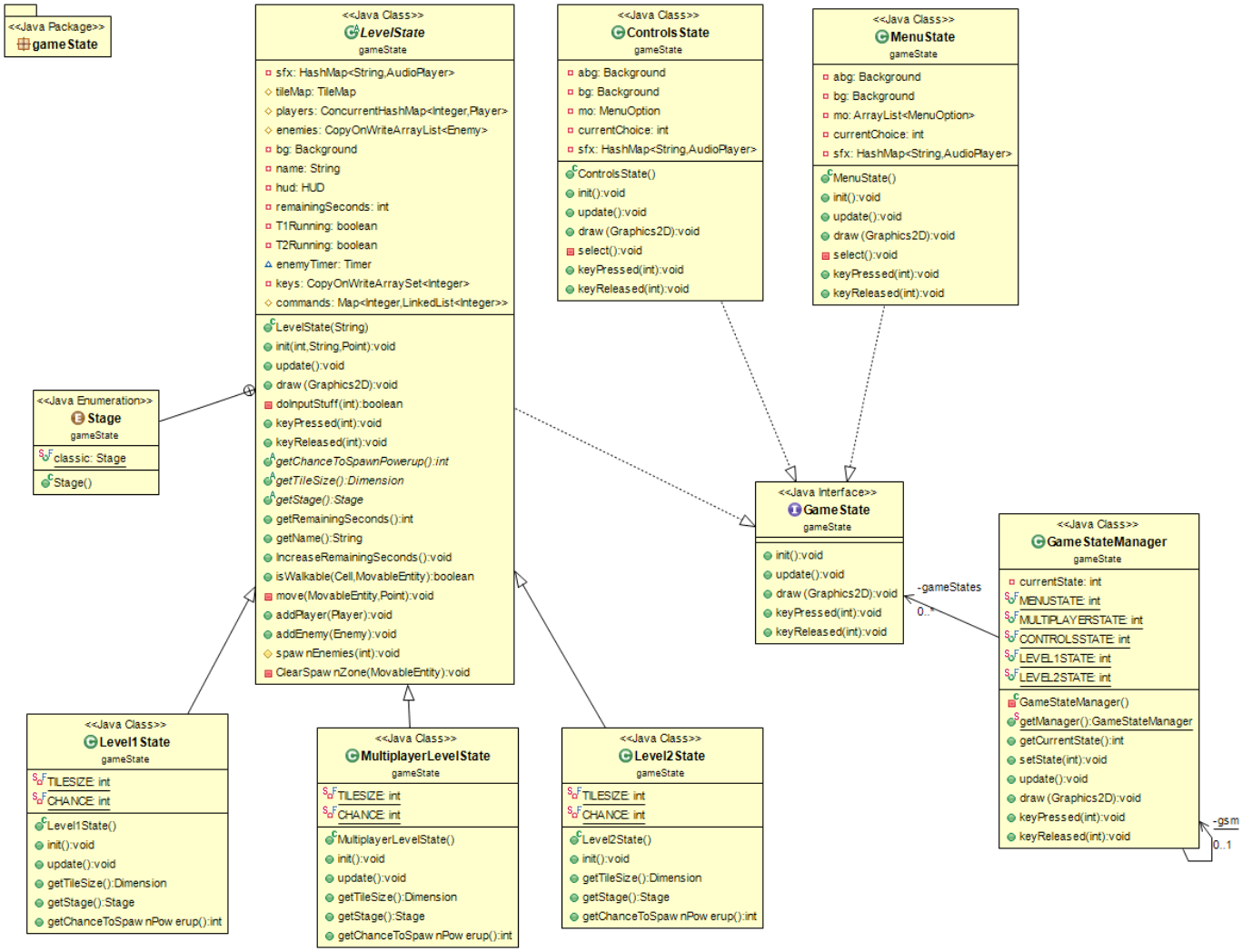
  x ed y sono le coordinate dell'opzione nella schermata del menu.

- Background:

  Crea il background della finestra, anche con la possibilità di renderlo scorrevole.

  Tramite setVector() si indica la velocità di scorrimento nelle 2 direzioni, poi tramite l'update() la posizione del background verrà modificata.

  In seguito, chiamando draw(), il backround verrà disegnato.



Questo package contiene tutti gli stati in cui il gioco si può trovare

- GameStateManager:

  è la classe che si occupa di mantenere lo stato attuale dell'applicazione ed a mandare eventuali messaggi allo stato corrente.

  Da questa classe passano tutti i keypressed, draw, update..., poi, avendo lo stato attuale del gioco, tali messaggi verranno reindirizzati verso le classi giuste.

- GameState:

  Interfaccia che indica come un generico stato dovrà essere fatto, sarà implementata in tutti i *concreteState*.

- LevelState:

  Classe astratta che indica come un generico livello di gioco dovrà essere fatto implementando i tratti comuni ad essi, come la gestione dei nemici, dei giocatori, della mappa, degli input...

- Stage:

    enumeratore utile al LevelState per sapere quali immagini caricare, tutto ciò per dare l'effetto di diversi Stage nel corso dei livelli.

- MenuState:

    Rappresenta lo stato del gioco alla partenza o quando una partita si conclude.

    Permette all'utente di scegliere tra varie opzioni, ad esempio se giocare in single player o multiplayer.

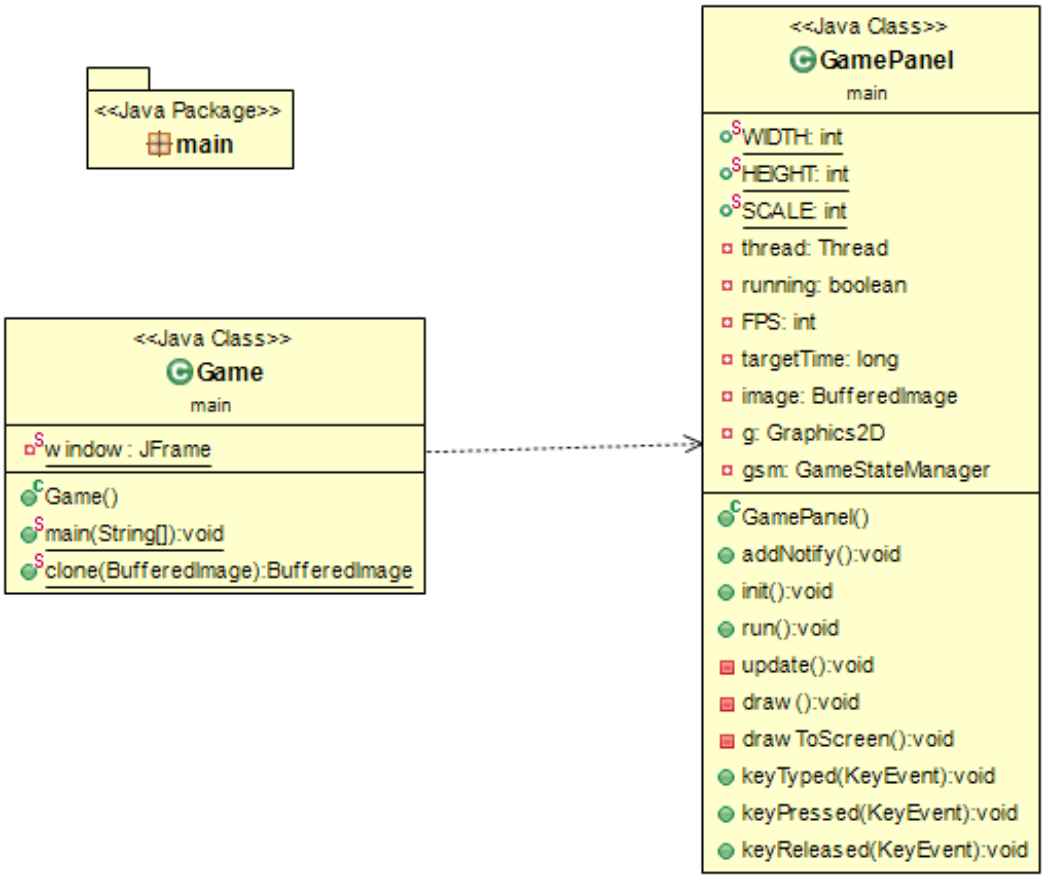
- ControlsState:

    Si tratta di un mini-stato usato per visualizzare i comandi quando richiesto scegliendo l'opzione nel menu.

- Level1State, Level2State, MultiplayerLevelState:

    3 implementazioni di LevelState, specificano le informazioni rimanenti (non implementate in LevelState) utili al lancio del livello.

    Specificano dove e quali giocatori e/o nemici dovranno spawnare, che stage usare per il livello, la probabilità che un powerup compaia alla rottura di un blocco.



Package di partenza del progetto

- Game:

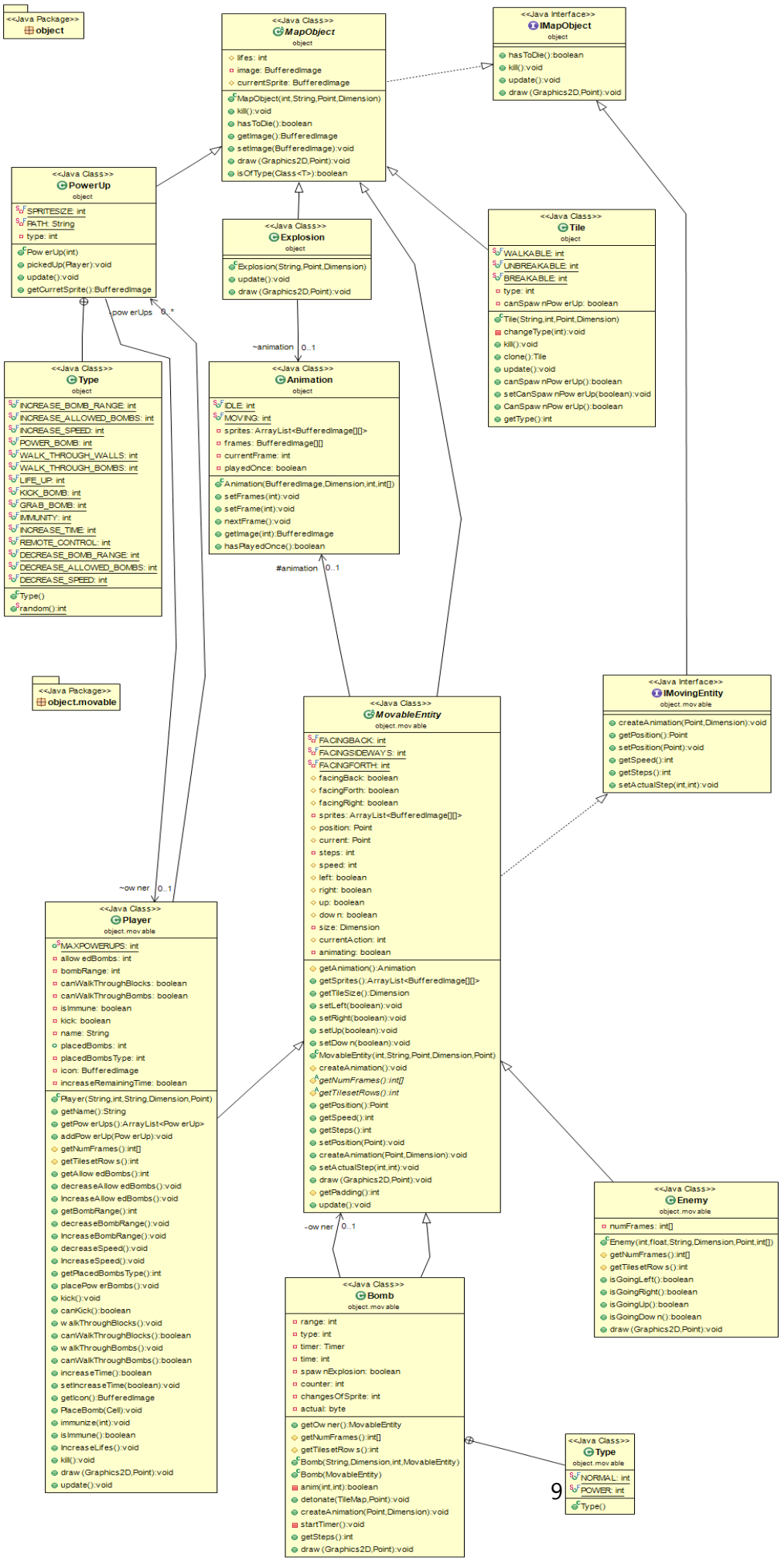
  contiene il main() del progetto, è la prima classe ad essere eseguita, ciò che fa è creare la finestra ed aggiungergli un GamePanel.

- GamePanel:

  è questa classe che si preoccupa di creare il Thread principale del gioco e di ricevere gli input.

  Al suo interno crea un GameStateManager al quale invia i messaggi di keyPressed() e keyReleased() quando necessario e update() e draw() ad ogni "giro" del Thread.

  Oltre a questo disegna il gioco stesso tramite drawToScreen sempre dentro al Thread, così da dare l'impressione non di immagini separate ma di un video continuo.


<<Java Package>>
object
<<Java Class>>
MapObject
object
lifes: int
image: BufferedImage
currentSprite: BufferedImage
MapObject(int,String,Point,Dimension)
kill():void
hasToDie():boolean
getImage():BufferedImage
setImage(BufferedImage):void
draw (Graphics2D,Point):void
isOfType(Class<T>):boolean
<<Java Interface>>
IMapObject
object
hasToDie():boolean
kill():void
update():void
draw (Graphics2D,Point):void
<<Java Class>>
PowerUp
object
SPRITESIZE: int
PATH: String
type: int
PowerUp(int)
pickedUp(Player):void
update():void
getCurretSprite():BufferedImage
<<Java Class>>
Explosion
object
Explosion(String,Point,Dimension)
update():void
draw (Graphics2D,Point):void
<<Java Class>>
Tile
object
WALKABLE: int
UNBREAKABLE: int
BREAKABLE: int
type: int
canSpawnPowerUp: boolean
Tile(String,int,Point,Dimension)
changeType(int):void
kill():void
clone():Tile
update():void
canSpawnPowerUp():boolean
setCanSpawnPowerUp(boolean):void
CanSpawnPowerUp():boolean
getType():int
-powerUps
0..*
~animation
0..1
<<Java Class>>
Type
object
INCREASE_BOMB_RANGE: int
INCREASE_ALLOWED_BOMBS: int
INCREASE_SPEED: int
POWER_BOMB: int
WALK_THROUGH_WALLS: int
WALK_THROUGH_BOMBS: int
LIFE_UP: int
KICK_BOMB: int
GRAB_BOMB: int
IMMUNITY: int
INCREASE_TIME: int
REMOTE_CONTROL: int
DECREASE_BOMB_RANGE: int
DECREASE_ALLOWED_BOMBS: int
DECREASE_SPEED: int
Type()
random():int
<<Java Class>>
Animation
object
IDLE: int
MOVING: int
sprites: ArrayList<BufferedImage[][]>
frames: BufferedImage[][]
currentFrame: int
playedOnce: boolean
Animation(BufferedImage,Dimension,int,int[])
setFrames(int):void
setFrame(int):void
nextFrame():void
getImage(int):BufferedImage
hasPlayedOnce():boolean
#animation
0..1
<<Java Package>>
object.movable
<<Java Class>>
MovableEntity
object.movable
FACINGBACK: int
FACINGSIDEWAYS: int
FACINGFORTH: int
facingBack: boolean
facingForth: boolean
facingRight: boolean
sprites: ArrayList<BufferedImage[][]>
position: Point
current: Point
steps: int
speed: int
left: boolean
right: boolean
up: boolean
down: boolean
size: Dimension
currentAction: int
animating: boolean
getAnimation():Animation
getSprites():ArrayList<BufferedImage[][]>
getTileSize():Dimension
setLeft(boolean):void
setRight(boolean):void
setUp(boolean):void
setDown(boolean):void
MovableEntity(int,String,Point,Dimension,Point)
createAnimation():void
getNumFrames():int[]
getTilesetRows():int
getPosition():Point
getSpeed():int
getSteps():int
setPosition(Point):void
createAnimation(Point,Dimension):void
setActualStep(int,int):void
draw (Graphics2D,Point):void
getPadding():int
update():void
<<Java Interface>>
IMovingEntity
object.movable
createAnimation(Point,Dimension):void
getPosition():Point
setPosition(Point):void
getSpeed():int
getSteps():int
setActualStep(int,int):void
~owner
0..1
<<Java Class>>
Player
object.movable
MAXPOWERUPS: int
allowedBombs: int
bombRange: int
canWalkThroughBlocks: boolean
canWalkThroughBombs: boolean
isImmune: boolean
kick: boolean
name: String
placedBombs: int
placedBombsType: int
icon: BufferedImage
increaseRemainingTime: boolean
Player(String,int,String,Dimension,Point)
getName():String
getPowerUps():ArrayList<PowerUp>
addPowerUp(PowerUp):void
getNumFrames():int[]
getTilesetRows():int
getAllowedBombs():int
decreaseAllowedBombs():void
increaseAllowedBombs():void
getBombRange():int
decreaseBombRange():void
increaseBombRange():void
decreaseSpeed():void
increaseSpeed():void
getPlacedBombsType():int
placePowerBombs():void
kick():void
canKick():boolean
walkThroughBlocks():void
canWalkThroughBlocks():boolean
walkThroughBombs():void
canWalkThroughBombs():boolean
increaseTime():boolean
setIncreaseTime(boolean):void
getIcon():BufferedImage
PlaceBomb(Cell):void
immunize(int):void
isImmune():boolean
increaseLifes():void
kill():void
draw (Graphics2D,Point):void
update():void
-owner
0..1
<<Java Class>>
Enemy
object.movable
numFrames: int[]
Enemy(int,float,String,Dimension,Point,int[])
getNumFrames():int[]
getTilesetRows():int
isGoingLeft():boolean
isGoingRight():boolean
isGoingUp():boolean
isGoingDown():boolean
draw (Graphics2D,Point):void
<<Java Class>>
Bomb
object.movable
range: int
type: int
timer: Timer
time: int
spawnExplosion: boolean
counter: int
changesOfSprite: int
actual: byte
getOwner():MovableEntity
getNumFrames():int[]
getTilesetRows():int
Bomb(String,Dimension,int,MovableEntity)
Bomb(MovableEntity)
anim(int,int):boolean
detonate(TileMap,Point):void
createAnimation(Point,Dimension):void
startTimer():void
getSteps():int
draw (Graphics2D,Point):void
<<Java Class>>
Type
object.movable
NORMAL: int
POWER: int
Type()
9

object e object.movable contengono tutte le entità del gioco:

- IMapObject:

  Interfaccia contenente le operazioni comuni a tutte le entità.

  (ia è stato tolto, risultava inutile, tanto poi chi deve essere comandato da ia lo si capisce in altri modi)

- MapObject:

  Classe astratta che implementa IMapObject, dà un'implementazione a 3 dei 4 metodi richiesti dall'interfaccia e aggiunge supporto per le immagini.

  kill() leva una vita all'entità in questione

  hasToDie() controlla se le vite sono finite o meno

  draw() disegna lo sprite attuale alla posizione passatagli come parametro

- Powerup, Tile, Explosion:

  3 estensioni di MapObject per rappresentare diversi oggetti.

  Un Powerup è un oggetto che può comparire alla distruzione di un Tile e dona effetti (positivi o negativi) ad un Player che lo raccoglie. Può essere di uno dei tipi specificati nella sottoclasse Type.

  Il Tile è un blocco costituente della mappa, può essere di 3 tipi (WALKABLE, BREAKABLE ed UNBREAKABLE), rispettivamente camminabile, cioè senza effetto per le entità che ci passano sopra, distruttibile, cioè blocca le entità che tentano di passargli attraverso ma, tramite una bomba, tale blocco viene rimosso, e indistruttibile, cioè viene messo li alla generazione della mappa e non c'è modo di spostarlo o di passarci attraverso.

  Explosion rappresenta l'effetto di una bomba al momento dell'esplosione.

- Animation:

  Classe che si preoccupa del cambiamento di sprite di una entità quando necessario.

  Tutti gli sprites vengono caricati alla creazione dell'oggetto, così da non dover andare a recuperarli al momento del bisogno, poi, quando serve lo sprite corrente, è ottenibile tramite getImage().

Ora passiamo alle classi di object.movable, sono in un package separato dal precedente per indicare che le seguenti sono, per un qualche motivo, in grado di spostarsi.

- IMovingEntity:

  Interfaccia che estende la precedente (IMapObject) ed aggiunge le signature dei metodi utili allo spostamento.

- MovableEntity:

  Classe astratta che implementa IMovingEntity ed estende MapObject, questo per indicare che le entità in grado di muoversi possono essere considerate normali entità ove necessario.

  Si preoccupa di specificare come le varie entità dovranno muoversi, e lo fa tramite l'implementazione di createAnimation, in pratica, sapendo dove l'entità deve arrivare e la sua posizione attuale, e conoscendo il numero di passi che tale entità deve fare per raggiungere il "traguardo", modifica la posizione e il frame (sfruttando un oggetto Animation) ad ogni intervallo di tempo calcolato in base alla velocità dell'entità stessa.

- Player:

  Rappresenta l'entità comandata da un giocatore tramite tastiera, principalmente contiene i metodi necessari ad applicare a se stesso gli effetti dei powerup che raccoglie.

- Enemy:

  Questi, invece, sono i nemici controllati dall'ia, ciò che fanno è scegliere una direzione casuale e muoversi in tale direzione se possibile.

- Bomb:

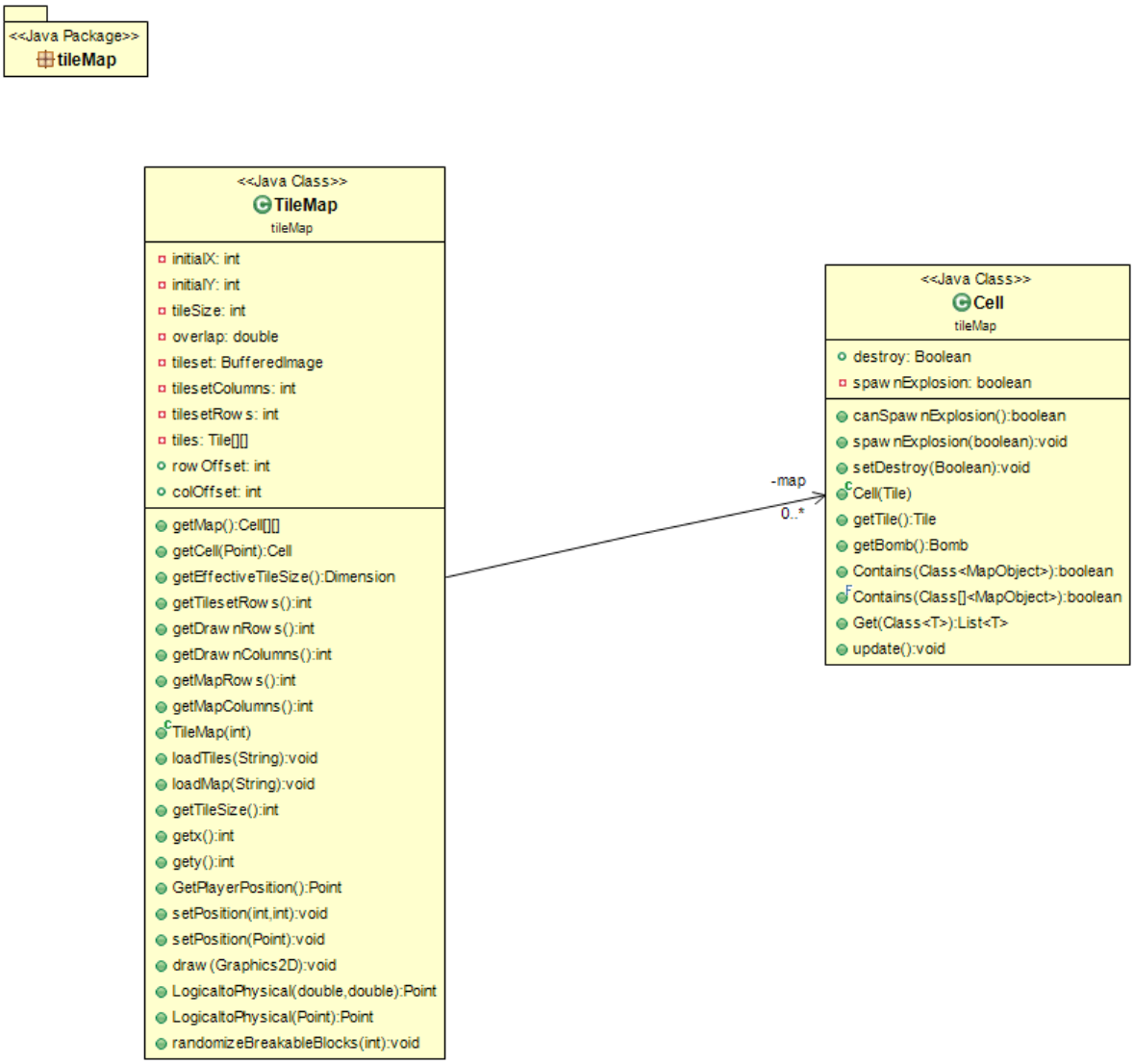
  Rappresenta la bomba rilascata dai Player alla pressione di un certo tasto.

  Ciò che una bomba fa è, immediatamente dopo essere stata piazzata, attendere un certo tempo per poi esplodere, danneggiando tutto ciò che colpisce.

  Anche la bomba sfrutta un oggetto Animation per cambiarsi di sprite ad intervalli regolari.

  Esistono 2 tipi di bombe, NORMAL e POWER, la prima distrugge blocchi attorno a se al momento dell'esplosione fino al primo blocco distruttibile (BREAKABLE) che incontra in ognuna delle 4 direzioni, la seconda prosegue fino al raggiungimento del suo range.

  Entrambi i tipi si fermano se incontrano un blocco del tipo UNBREAKABLE.



Questo package contiene le classi utili alla gestione delle mappe.

- tileMap:

  è la classe che si occupa del caricamento e del disegnamento di una mappa da file a schermo.

  Una mappa viene caricata mediante il metodo loadMap, che prende in ingresso un file contenente la mappa sotto forma di numeri (di base sono 0, 1 e 2) che indicano che tipo di blocco andare a mettere in quella posizione, li converte in Tile e ritorna una matrice di Cell.

  Quali tiles usare vengono specificati usando loadTiles, in base alla stringa in ingresso andrà a caricare i corrispettivi tile. Questo è l'unico punto dove vengono caricati i Tile della mappa, così da non dover andare a cercarli e caricarli quando servono.

  Questo metodo verrà richiamato da LevelState e la stringa sarà dipendente da quale Stage il livello corrente ha deciso di usare.

- Cell:

Rapresenta la zona atomica della mappa, in pratica un immaginario cubo contenente ciò che in quel momento sta in quella posizione della mappa.

Alla creazione clona il Tile passato come parametro, in tal modo i TIle di base restano pochi e si risparmia in prestazioni.

Una cella è una lista di MapObject, si accorge di particolari eventi che possono accadere su di essa e agisce di conseguenza.

è la cella che si accorge se una certo oggetto deve morire (se ha finito le sue vite), e nel caso la rimuove.

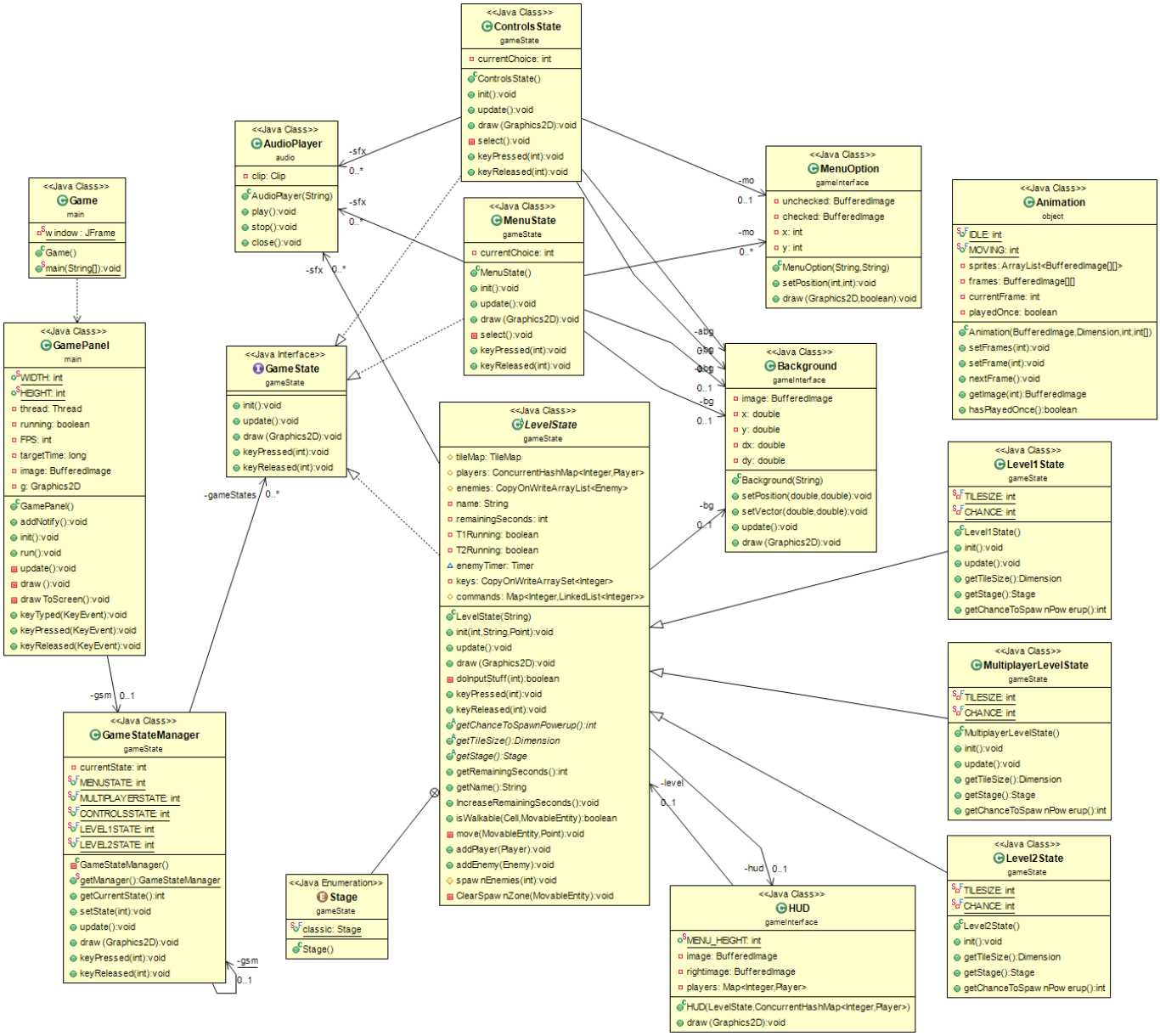
Si accorge anche se una bomba è esplosa nelle vicinanze e, nel caso, danneggia tutte le entità su di essa. Alla stessa maniera, se un nemico (classe Enemy) e un Player sono sulla stessa cella, fa in modo che il Player perda una vita.

In maniera simile, se un Player e un Powerup sono sulla stessa cella, il Player raccoglie il Powerup.

Suddivisione del lavoro

Il carico di lavoro è stato suddiviso come segue:

Luca Battistini:



Mi sono occupato del cosiddetto "cuore del gioco", ovvero le basi su cui tutte le classi ruotano intorno ed le classe di "decorazione", quali Animation, Background, AudioPlayer, ecc..

Il gioco non è altro che una semplice form con al suo interno un unico componente, la classe GamePanel, ovvero l'estensione della classe JPanel (messa a disposizione dal linguaggio) implementando due interfacce Runnable e KeyListener.

La prima rende l'istanza del nuovo componente eseguita da un thread, mentre la secondo implementa i metodi per il riconoscimento di inputs da tastiera.

Per la creazione di questa classe mi sono avvalso di qualche dritta scovata in un web tutorial su youtube di **ForeignGuyMike**. In poche parole il vero "core" è il metodo run(), cioè il game loop il quale eseguirà, finché l'applicazione non sarà terminata, le funzioni di update(), draw() e drawToScreen() per ogni singolo frame.

Per fare in modo che gli FPS siano costanti e di conseguenza non ci sia lagg, ho sfruttato un timer il quale, raggiunto un targetTime, eseguita un thread sleep di

qualche frazione di secondo in modo tale che l'esecuzione di ogni frame occupi lo stesso lasso di tempo affinché le funzioni vengano eseguite correttamente.

All'interno di questa classe è inoltre istanziata un istanza della classe GameStateManager, ovvero il gestore degli stati del gioco; gli stati variano dal semplice menù iniziale allo stato di gioco.

Il GameStateManager definisce un'ArrayList di GameState, ovvero l'interfaccia che definisce le funzioni che ogni stato deve implementare, cioè init(), update(), draw(), keyPressed() e KeyReleased(), le quali cambieranno a differenza dello stato corrente.

Inoltre il GameStateManager si occuperò direttamente di chiamare le funzioni appena citate per lo stato di gioco corrente, il quale vero impostato da mediante le azioni dell'utente finale.

Lo stato iniziale selezionato è il menuState.

Il menuState, avvalendosi delle classi Background (definisce il comportamento di sfondi statici, ma anche animati animati) e menuOption, creerà l'ambiente visivo con il quale l'utente verrà ad interfacciarsi, in particolare il metodo select() sarà il tramite per l'impostazione del Level1State oppure del MultiplayerLevelState, le due modalità di gioco.

Ogni livello di gioco, in quanto ha funzioni comuni, estende la superasse LevelState.

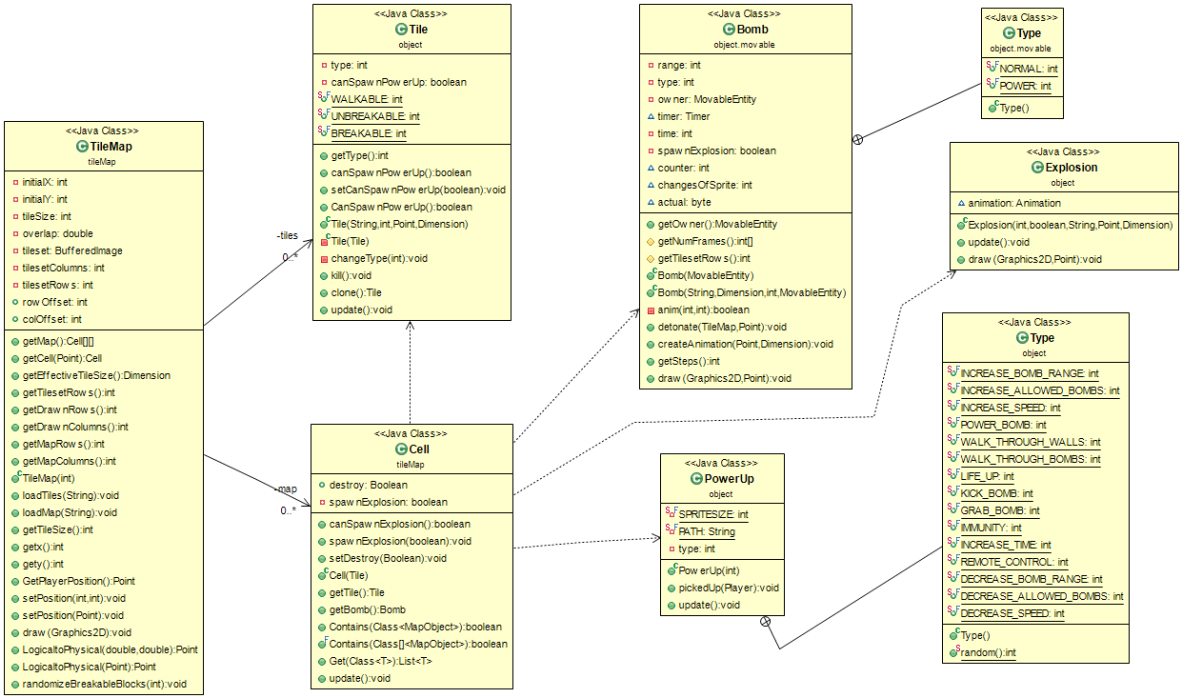
LevelState si occupa delle creazione del livello di gioco, ciò implica che faccia da mediatore fra la mappa e le entità. Vi sono presenti quindi metodi per l'aggiunta di giocatori: addPlayer() e per l'aggiunta di nemici: spawnEnemies(), addEnemy(), e metodi per il controllo dei movimenti di essi.

Per gestire gli input vengono utilizzati keyPressed() e keyReleased() i quali imposteranno a true o false le variabili di movimento (left, right, up, down) dei giocatori definite nelle relative entità. Nel costruttore sono definiti i tasti amessi per la gestione del personaggio.

Le classi Animation e AudioPlayer hanno entrambe un funzionamento molto semplice, la prima incrementerà ad ogni tick del timer un indice indicate il numero del frame attuale di un determinato array di BufferedImage (uno per ogni stato, nel nostro caso IDLE e MOVING); mentre la seconda dopo avere caricato una clip audio da file implementerà le funzioni play() e stop().

Oltre alla mia parte ho supervisionato, in parte, il lavoro degli altri due miei colleghi, aiutandoli anche a risolvere alcune problematiche.

Loris Cangini:



Mi sono preoccupato principalmente della parte relativa alla creazione della mappa di gioco e degli oggetti costituenti di essa.

La classe principale è TileMap, è lì che viene caricato il file .map che indica quali blocchi usare e dove.

La prima cosa che viene fatta, all'interno di un livello, è usare LoadTiles() per caricare le immagini dei blocchi da usare per disegnare la mappa a schermo.

Dopo può essere usato LoadMap() per caricare la mappa da file, dove non si fa altro che scorrere il file per righe e mettere nella matrice interna alla classe delle Cell contenenti dei TIle corrispondenti al valore letto.

SetPosition() può essere usato per spostare tutta la mappa, utile perchè ai bordi della finestra ci va il menu ingame (creato dalla classe HUD), la mappa finirebbe sotto ad esso se non spostata opportunamente.

Tramite il draw, che disegna tutte le entità contenute nelle celle nelle loro posizioni.

C'è anche un update(), in tale metodo TIlemap controlla se una bomba ha concluso il suo timer e quindi deve esplodere, in tal caso la fa detonare e la rimuove.

Nell'update si occupa anche di trovare da quali coordinate cominciare a disegnare la mappa, questo perchè è possibile avere mappe più grandi della zona di gioco e quindi si rende necessario disegnare solo la parte attualmente visibile in base alla posizione del giocatore.

Altra classe importante è Cell, rappresenta un cubetto della mappa, contiene tutte le entità che al momento sono in quella zona della mappa.

Praticamente tutto il lavoro viene fatto nell'update, dove controlla tutte le cose scritte durante l'analisi dei package.

Fatto ciò mi sono dedicato alla creazione delle bombe e ai loro effetti, quindi powerup ed explosion.

Una bomba è un entità particolare che appena creata setta un timer per distruggersi. Sfruttando la classe Animation crea un animazione durante il periodo di attesa prima dell'esplosione.

All'esplosione, setta un flag nelle celle colpite, in modo che tali Cell sappiano che ciò che c'è sopra deve essere danneggiato.

Le Cell con quel flag acceso (flag destroy) danneggiano le entità contenute in essa e creano un Explosion, che non è altro che l'effetto grafico dell'esplosione.

Similmente alla bomba, l'esplosione fa partire un Timer per eseguire un animazione, una volta finita si uccide.

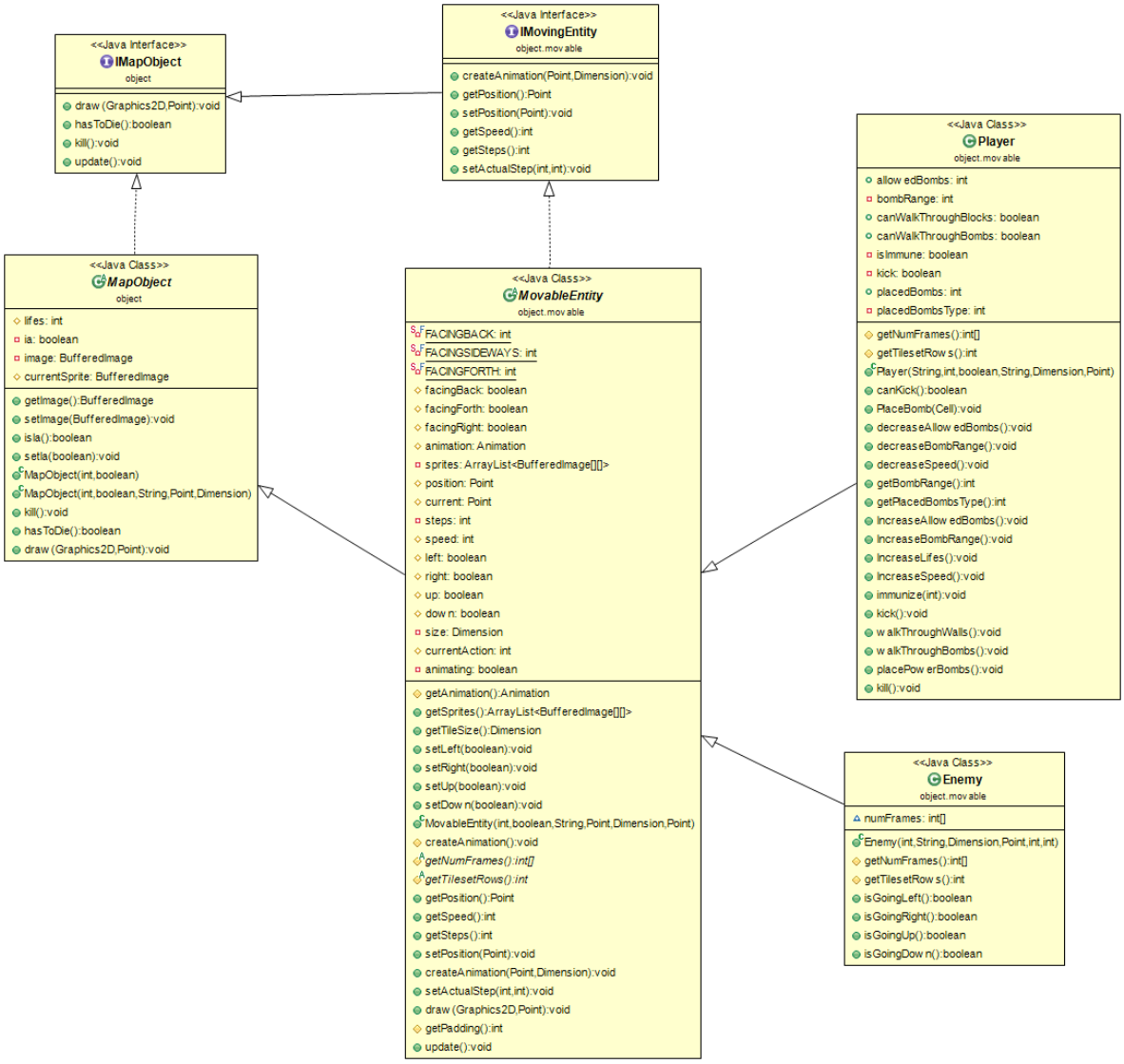
Un altro effetto del flag destroy è quello di distruggere i TIle distruttibili, tale meccanismo è trasparente alla cella, che non fa altro che chiamare i kill() delle varie entità, ciò che fa in modo che ciò sia possibile è il kill stesso dei Tile, che sovrascrive quello generico della superclasse.

Nel kill() dei Tile si controlla se è distruttibile, nel caso cambio il tipo del Tile in WALKABLE e setto un flag che dice alla Cell che in questo blocco potrà comparire un PowerUp.

Sarà ancora una volta la Cell ad accorgersi di tale flag e a far comparire un PowerUp, dipendentemente dalla possibilità che ciò accada, che cambia di livello in livello.

Un Powerup, similmente alla bomba, crea un Timer al suo interno per autodistruggersi dopo un certo tempo dopo essere comparso, se prima di tale periodo un Player raccoglie quel Powerup guadagnerà l'effetto associato ad esso.

Loris Salvi:



La parte su cui mi sono concentrato maggiormente è stata la creazione delle entità che sarebbero andate a comporre il gioco.

Per realizzare ciò, sono partito creando due *interfacce* principali: **IMapObject** e **IMovingEntity**. L'obbiettivo era definire una base di partenza per definire una reale divisione tra le diverse entità che si sarebbero andate a creare.

Creando **IMapObject** si voleva generare uno standard per tutte le entità. Lo standard, infatti, definisce quattro metodi principali che serviranno a tutti gli oggetti che verrano creati in futuro: *draw*, *hasToDie*, *kill* e *update*, sono le fondamenta che permettono il funzionamento del gioco.

Da questa interfaccia, infatti, si diramano due tipi di entità: quelle che una volta create potranno in qualche maniera muoversi e quelle che non potranno. Da qui la creazione di un'interfaccia come **IMovingEntity** per creare lo standard di tutte gli oggetti moventi.
Per realizzare questo, all'interno dell'interfaccia si sono definiti: *createAnimation*, *getPosition*, *setPosition*, *getSpeed*, *getSteps* e *setActualSteps*, che permettono all'oggetto di memorizzare e sfruttare i dati che permettono di generare l'effetto del

movimento.

Dopo aver generato le due interfacce di partenza, si è potuto definire le classi che racchiudono i concetti descritti appena sopra: **MapObject** e **MovableEntity**.
Tra le due la più complessa è la seconda, dato che la prima non fa altro che definire l'ambiente di partenza senza aggiungere funzioni più complesse.
La classe **MovableEntity**, diversamente, genera la struttura principale per gestire il movimento. Di relativa importanza il definire un'unica architettura che renda possibile il movimenti tramite tastiera, quindi tramite input dell'utente, e quella in automatico, decisa da un'intelligenza artificiale, senza creare conflitti o ridondanze di codice.

Mirando a questo obiettivo, si sono create delle variabili che tenessero traccia del verso di movimento del player, della sua posizione, dell'eventuale velocità variabile e anche dalla creazione di un intero, chiamato *steps*, per poter conoscere il numero di passi che l'entità dovrà subire tra una cella ed un'altra durante il movimento.
Di altrettanta importanza il metodo *createAnimation*, che collegato alla classe *Animation*, realizza l'effetto di spostamento.

Da MovableEntity, infine, si riesce a sviluppare le classi effettive di **Player** ed **Enemy**.
A differenza dall'ultima, la prima citata ha tutti i metodi che servono per attivare, disattivare, potenziare o indebolire le statische del giocatore che un certo power up scatena.
Di relativa importanza il metodo *kick* attivato dal proprio flag che viene settato *canKick*. Questo metodo, ammettendo che il player abbia preso il relativo power up, innescherà le relative operazioni che permetteranno di spostare la bomba.
Enemy, diversamente, implementa i metodi per muoversi senza ricevere alcun tipo di input.

Da notare come questa parte di programma sia strettamente collegata alla parte descritta dal mio collega Cangini.
Si sono effettuati, infatti, alcune correzioni per permettere alle classi di funzionare nel miglior modo possibile, come durante la creazione della classe *Cell* o durante lo sviluppo di *MovableEntity*.

Sono soddisfatto, infine, del risultato ottenuto, ma rimango ancora convinto che diverse parti di codice sarebbero potute essere sviluppate meglio. A mio parere la comunicazione è stata soddisfacente, anche se non sempre continua. Sono riuscito a notare, a volte, la poca esperienza nel lavorare in gruppo, almeno dalla mia parte. Ho trovato utile, però, una prova di questo genere, che sicuramente aiuta a mettere in pratica tutto ciò che ho assimilato durante le lezioni.

Note finali:

è stato inevitabile che in certe classi un secondo membro del gruppo, diverso dal principale fautore di tali classi, abbia dovuto aggiungere o modificare parti di codice.

Un esempio è Animation, inizialmente creava un Timer al suo interno per cambiare Sprite autonomamente, poi, per sincronizzare il tutto col movimento, il Timer è stato rimosso e lo sprite cambia tramite la chiamata di un altro metodo (nextFrame()), usato

nel codice di MovableEntity dove si crea lo spostamanto da una cella ad un altra.

Sono state fatte alcune modifiche alle classi dopo averle create in un altra maniera, ad esempio la classe GameStateManager è stata resa un Singleton in un secondo momento.

La classe astratta LevelState è stata anch'essa creata in un secondo momento, prima tutto era dentro Level1State, poi ci siamo resi conto che molti dei comportamenti che aveva tale livello erano in realtà comuni a tutti i livelli, questo ci ha portato lla creazione di una classe astratta che fattorizzasse tali comportamenti.