Università di Bologna – Sede di Cesena

Corso di Laurea in Scienze ed Ingegneria Informatica a.a. 2013-2014

Programmazione ad Oggetti

Relazione relativa all'ingegnerizzazione di Space Rush

Componenti del gruppo: Tommaso Bodini, Damiano Galassi

1 Descrizione del problema

Si vuole realizzare un semplice videogioco di strategia come consegna per l'elaborato d'esame. Il gioco proposto, denominato "S*pace Rush*" ha come scopo il posizionamento strategico di torrette difensive per bloccare l'avanzata di asteroidi nel percorso che li porta dall'inizio all'uscita del gioco. Qualora non si riesca a posizionare le torrette in maniera tale da consentire la distruzione degli asteroidi, questi ultimi potrebbero raggiungere l'uscita nella mappa, e qualora si raggiunga il limite di asteroidi che possono uscire dalla mappa di gioco, si va incontro alla sconfitta

La costruzione di torrette difensive comporta per il giocatore una spesa, al contrario, la distruzione di un asteroide farà guadagnare una somma di denaro; la dotazione monetaria del giocatore viene salvata alla fine di ogni round. Il giocatore vince qualora riesca a piazzare strategicamente le torrette per distruggere tutte le asteroidi previste dalla modalità di gioco.

2 Progettazione architetturale

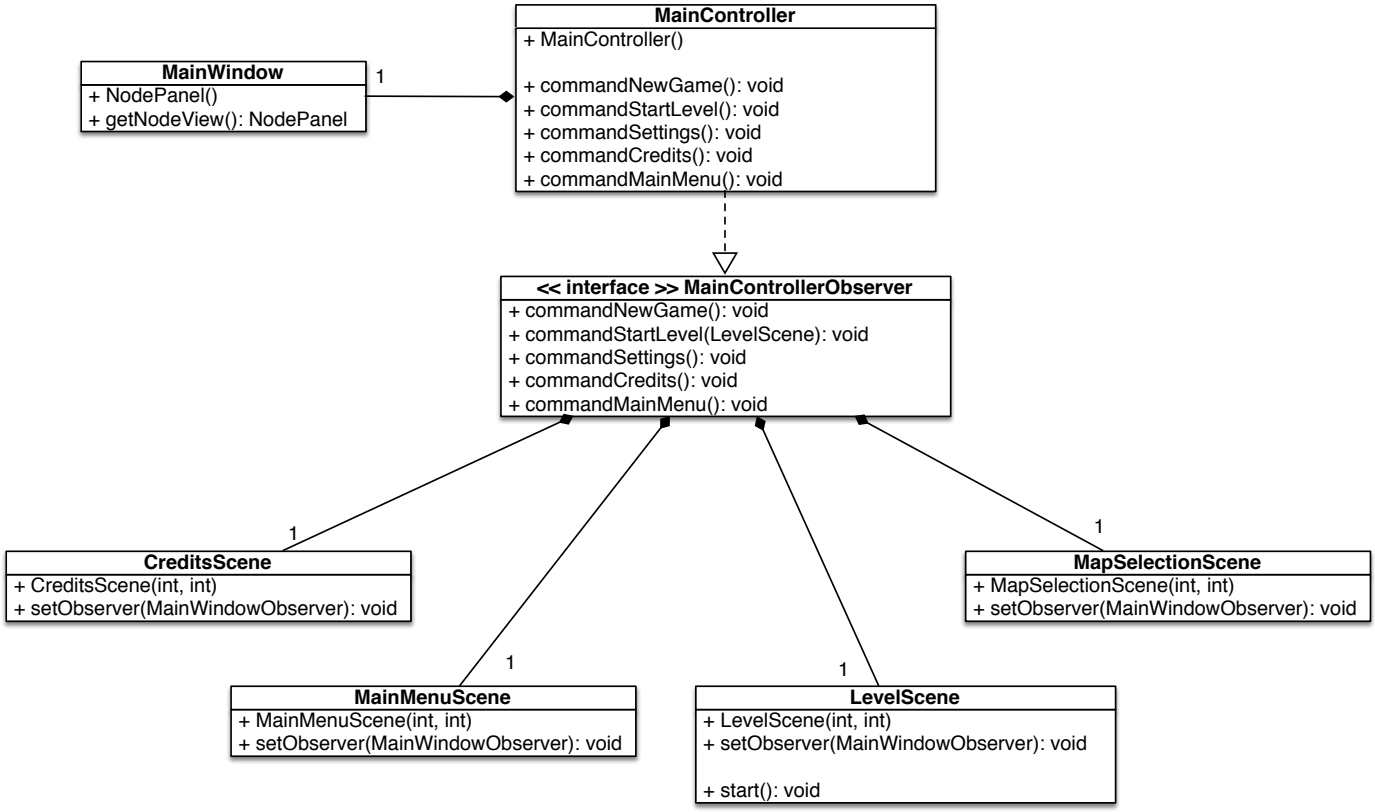
L'applicazione è stata progettata in modo da rendere il più indipendente possibile lo sviluppo delle varie parti dell'applicazione tramite il pattern MVC.

Il programma si compone di un controller e di una vista principale che permette di presentare a schermo le varie scene in cui è suddiviso il gioco.

All'avvio viene caricato il controller principale e la scena iniziale (MainMenuScene), dalla quale si può passare alle altre scene. Dalla scena MapSelectionScene un utente può selezionare la mappa e le opzioni di gioco ed iniziare una nuova partita.

Per creare una interfaccia grafica animabile si è realizzata una libreria, spriteKit, ottimizzata per la gestione di sprite bidimensionali. Ciò ha permesso di ottenere con facilità effetti come trasparenze, dissolvenze, traslazioni, rotazioni e sequenze di animazioni mantenendo il codice snello e comprensibile. La maggior parte dell'interfaccia del programma è stata realizzata con questa libreria (per una analisi approfondita si veda la sezione 5.2).

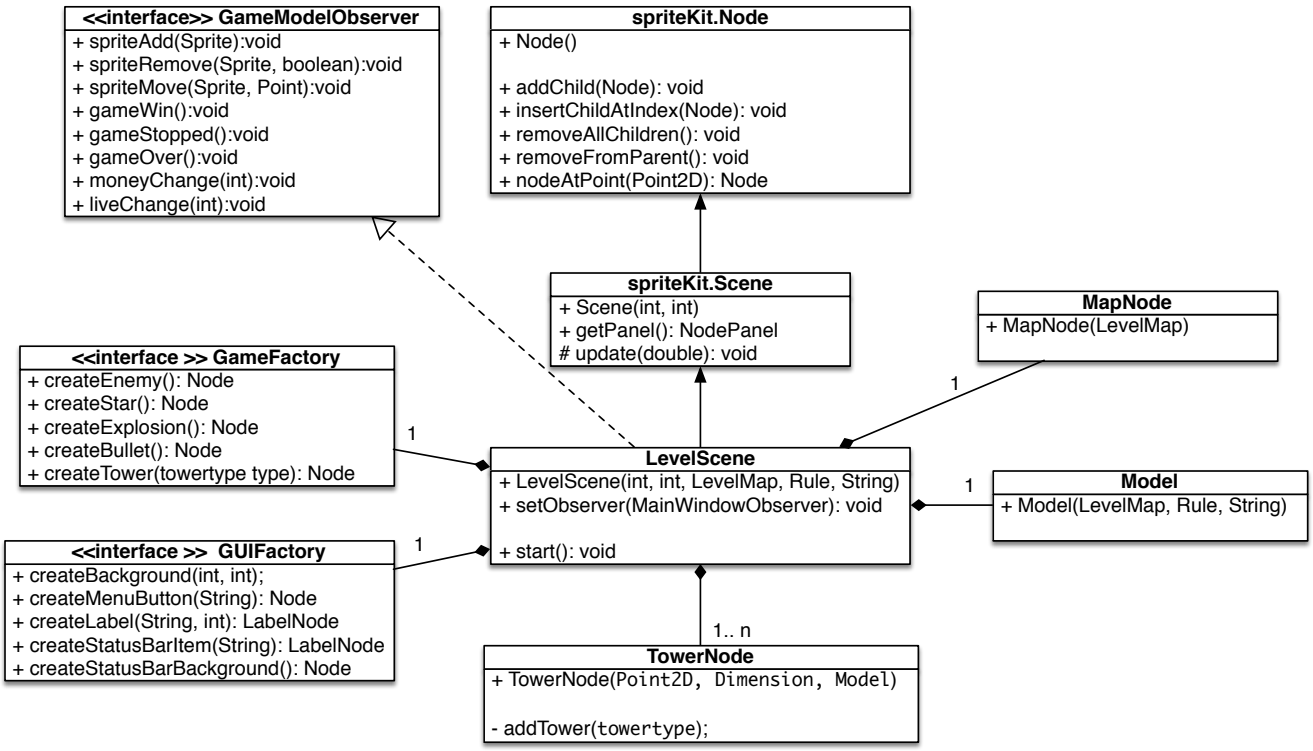
Diagramma UML semplificato delle parti di view e controller principale dell'applicazione:



Descrizione degli aspetti principali:

- **MainController**: Il controller principale, che si occupa di costruire e mostrare le seguenti scene:
  - MainMenuScene
  - MapSelectionScene
  - CreditsScene
  - LevelScene
- **MainMenuScene**: la scena iniziale, mostra il titolo del programma e un menu per passare alle altre scene;
- **CreditsScene**: una scena che mostra i crediti.
- **MapSelectionScene:** si occupa del caricamento da disco delle mappe e delle regole del gioco in formato xml, permettendo al giocatore di selezionarle e di iniziare una nuova partita.

Diagramma UML della classe LevelScene:



LevelScene: il cuore vero e proprio del gioco. È la classe che implementa l'interfaccia **GameModelObserver**: costruisce un nuovo modello del gioco e le varie parti che ne compongono l'interfaccia:

- **TowerNode**: il controller della vista dei punti delle torrette.
- **MapNode**: la vista della mappa del gioco.

gli elementi del gioco sono generati da una classe factory (spacerush.gameViews.GameFactory) visto che non contenevano elementi significati per renderli classi separati.

La classe Model si occupa della simulazione dello gioco e dell'aggiornamento dei salvataggi. Comunica con il controller tramite il pattern Observer, il controller implementa l'interfaccia GameModelObserver, e ad ogni evento generato dalla simulazione di gioco aggiorna i rispettivi elementi dell'interfaccia. Questo disaccoppiamento permette al modello di essere totalmente indipendente dal suo utilizzatore, e aumenta le possibilità di riuso dei componenti dell'applicazione.

Descrizione degli aspetti principali del modello:

- **Model**: implementazione della simulazione del gioco. Permette di essere inizializzata fornendogli una mappa, un set di regole e un observer.
- **GameModelObserver:** l'interfaccia implementata dall'observer del modello del gioco.

Per il salvataggio delle preferenze si è deciso di usare un singleton implementanti nella classe **Settings**, mentre SettingsController permette di presentare a schermo una finestra per la modifica delle impostazioni. È possibile tramite questa finestra attivare/disattivare l'audio di sfondo, gli effetti sonori, reimpostare il gioco e impostare il livello di difficoltà.

Considerazioni sulla divisione controller e view all'interno di una scena:

L'utilizzo della libreria SpriteKit ha comportato una leggera deviazione dal modello MVC usato normalmente in Java (View ↔ Controller ↔ Model) per la progettazione di programmi con interfaccia grafica. Un oggetto di tipo Scene (descritta in dettaglio più avanti) è una sottoclasse di Node che può essere visualizzata in un NodePanel. Scene aggiunte a Node un metodo update() chiamato ogni tot secondi utile per aggiornare la logica di gioco. Una scena è quindi ideale per essere usata come controller principale. Ma allo stesso tempo contiene elementi visuali che non possono essere disassociati.

Per le scene create per questa applicazione e per la loro semplicità si è deciso di aggiungere i 5-6 nodi che normalmente compongono una scena direttamente senza creare un'altra sottoclasse Node finalizzata solamente alla creazione degli elementi visuali.

Librerie esterne:

Si è utilizzata la libreria Xstream (http://xstream.codehaus.org/) che consente di serializzare gli oggetti in formato XML permettendo così la modifica e la creazione di nuove mappe e regole di gioco tramite un semplice editor di testi.

3 Organizzazione in package

Al fine di organizzare il progetto in maniera ordinata, ogni parte funzionale è stata inserita in un package separato; si illustrano di seguito i package del progetto e se ne spiegano le funzionalità da essi fornite.

1.spacerush.main: contiene solamente la classe che lancia l'intero applicativo

2.spacerush.controller: contiene le varie classi che si occupano di disegnare le varie scene di cui è composto il gioco, oltre ad intercettare gli eventi per gestire l'interazione con l'utente

3.spacerush.gameViews: in questo package sono inserite le classi che implementano le varie funzioni per semplificare la creazione dell'interfaccia grafica del gioco vero e proprio.

4.spacerush.model: ingloba tutte le classi che gestiscono la logica del gioco e le impostazioni

5.spacerush.soundEffect: contiene le classi che permettono l'esecuzione degli effetti sonori e della musica in background

7.spacerush.view: contiene classi che implementano la vista e gli elementi dell'interfaccia grafica.

8.**spriteKit**: è una package per comporre un grafo di una scena bidimensionale.

4 Suddivisione del lavoro

Le varie parti di cui e composto il progetto sono state realizzare dai componenti del gruppo in base alla seguente suddivisione:

- Tommaso Bodini si è occupato di:
  - Modello
  - Sistema degli effetti audio
  - Schermata selezione impostazioni
- Damiano Galassi si è occupato di:
  - Vista con relative funzioni per permettere una più agevole creazione degli oggetti grafici
  - Controller che capta gli eventi e li notifica al modello

5.1 Progettazione in dettaglio: Tommaso Bodini

Modello

Come si può notare dallo schema UML sottostante, il modello si compone di:

-**LevelMap**: descrive il livello con le sue caratteristiche: punto di ingresso e di uscita e mappa, memorizzata come matrice di sprite che viene generata dal metodo generateMap() a partire da una serie di coppie di punti che descrivono i segmenti di cui è composta la mappa, caricati da un file esterno XML tramite xstream; tale soluzione si è resa necessaria per avere dei tempi di caricamento accettabili (in quanto matrici di sprite di grosse dimensioni necessitano di parecchi secondi per essere caricati). Estende GameOption che contiene nome e descrizione del livello.

-**Rule**: descrive le impostazioni di gioco, ovvero le ondate di nemici previsti e il numero di nemici che posso raggiungere l'uscita prima di perdere il gioco. Il metodo inizialize() ha il compito di creare effettivamente gli oggetti di tipo Unit. Le impostazioni di gioco sono anch'esse serializzate in file esterni (in XML tramite le librerie xstream). Analogamente a LevelMap, Rule estende GameOption per le stesse ragioni.

-**Unit**: rappresentano le unità nemiche (asteroidi) da colpire, vengono mantenuti in una lista all'interno del modello. La classe Unit estende Entity che modella le sprite animabili (vive), definisce quini il metodo live() che descrive il comportamento del nemico. A sua volta la classe si specializza con delle classi statiche innestate che definiscono i vari tipi di unità nemiche; sono proprio le classi che definiscono il metodo getDir(...) che restituisce la direzione di movimento dell'unità in base ad un algoritmo che permette di differenziare “l'intelligenza” del nemico.

-**Tower**: in maniera del tutto simile Unit definisce il comportamento delle torri difensive; unica differenza sono i metodi astratti getTarget() e detDamage(...), che implementati nelle estensioni restituiscono rispettivamente il nemico da colpire e il danno da infliggerli.

-**Bullet**: classe che modella il comportamento dei proiettili sparati dalle torrette, estende Entity e quindi implementa live() che definisce il comportamento dei proiettili.

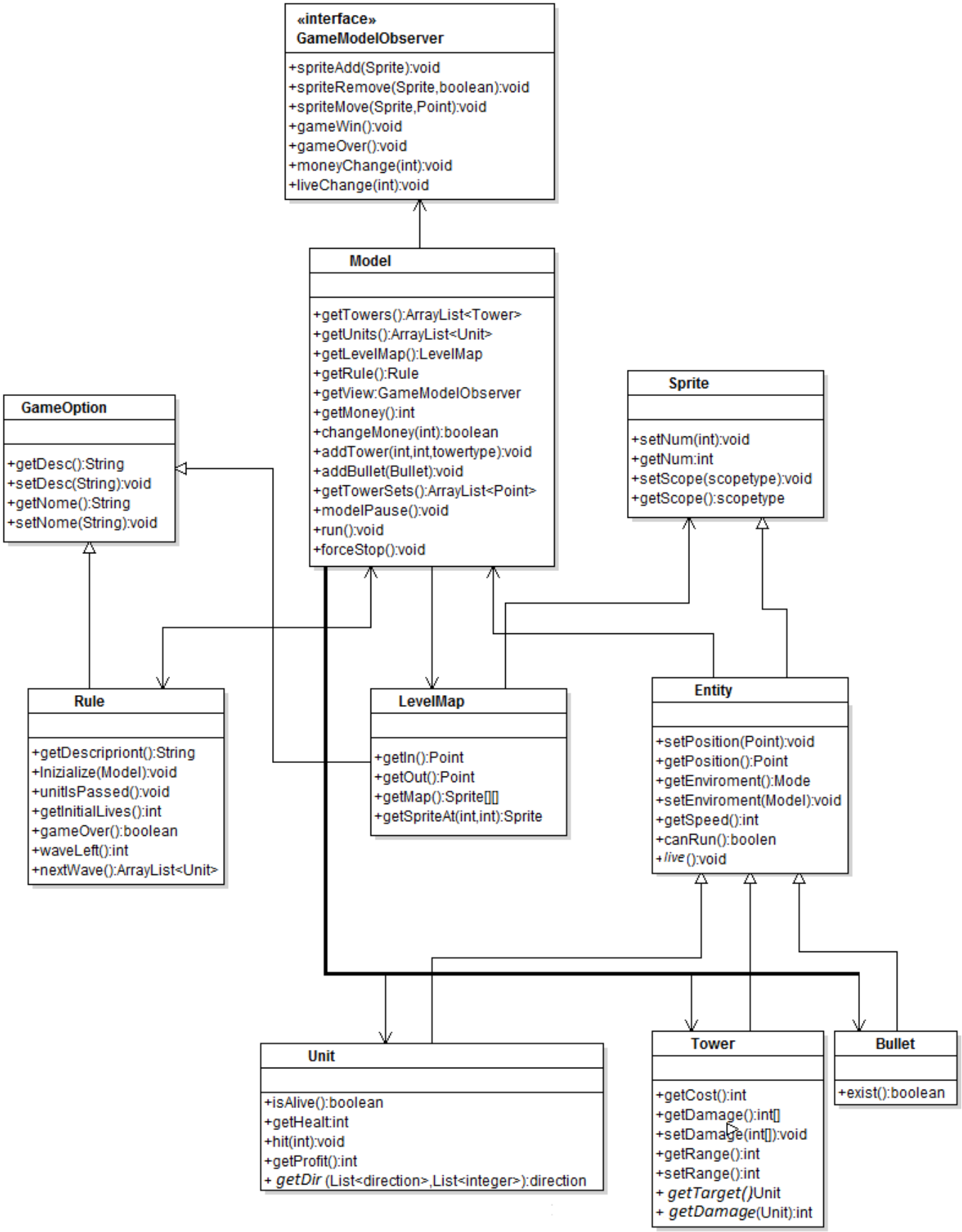
Altre classi presenti nel package sono:

-**Setting**: memorizza le impostazioni correnti del gioco, impostate all'inizio dello stesso quali difficoltà ed effetti sonori(silenziosi o meno). Utilizza il pattern Singleton poiché è evidente che vi è una e una sola istanza di impostazioni di gioco. Implementa Serializable per far si che le impostazioni salvate siano disponibili all’apertura del gioco. (non presente nel UML poichè di poco interesse)

-**GameOption**: fattorizza i campi comuni tra LevelMap e Rule.

-**StateSaved**: incapsula il denaro disponibile, essendo serializzabile salva alla fine di ogni partita le disponibilità monetarie.(non presente nel UML poichè di poco interesse)

Il modello (**Model**) è progettato per essere un thread se stante (estendendo Thread), il metodo run() si occupa di gestire le ondate di nemici e l'avanzamento del gioco, chiamando ciclicamente il metodo live() su torri, nemici e proiettili; gestisce inoltre l'eliminazione di nemici(asteroidi) distrutti o usciti dalla mappa di gioco. Il modello incapsula quindi tutti gli elementi che ne fanno parte.


«interface»
GameModelObserver
+spriteAdd(Sprite):void
+spriteRemove(Sprite,boolean):void
+spriteMove(Sprite,Point):void
+gameWin():void
+gameOver():void
+moneyChange(int):void
+liveChange(int):void
Model
+getTowers():ArrayList<Tower>
+getUnits():ArrayList<Unit>
+getLevelMap():LevelMap
+getRule():Rule
+getView:GameModelObserver
+getMoney():int
+changeMoney(int):boolean
+addTower(int,int,towertype):void
+addBullet(Bullet):void
+getTowerSets():ArrayList<Point>
+modelPause():void
+run():void
+forceStop():void
GameOption
+getDesc():String
+setDesc(String):void
+getNome():String
+setNome(String):void
Sprite
+setNum(int):void
+getNum:int
+setScope(scopetype):void
+getScope():scopetype
Rule
+getDescripriont():String
+Inizialize(Model):void
+unitIsPassed():void
+getInitialLives():int
+gameOver():boolean
+waveLeft():int
+nextWave():ArrayList<Unit>
LevelMap
+getIn():Point
+getOut():Point
+getMap():Sprite[][]
+getSpriteAt(int,int):Sprite
Entity
+setPosition(Point):void
+getPosition():Point
+getEnviroment():Mode
+setEnviroment(Model):void
+getSpeed():int
+canRun():boolen
+live():void
Unit
+isAlive():boolean
+getHealt:int
+hit(int):void
+getProfit():int
+ getDir (List<direction>,List<integer>):direction
Tower
+getCost():int
+getDamage():int[]
+setDamage(int[]):void
+getRange():int
+setRange():int
+ getTarget()Unit
+ getDamage(Unit):int
Bullet
+exist():boolean

Sistema audio

Il sistema di gestione degli effetti sonori proposto, presente nel package soundEffect, si basa su due parti:

Musica di background da file MIDI, si basa sulla creazione di un oggetto della classe Sequencer definita di javax.sound.midi, che esegue il file MIDI di interesse, i due metodi nella classe MIDISoundBackground sono play() e stop() che molto semplicemente fanno partire e bloccano l'esecuzione del file MIDI.

Gli effetti sonori invece sono gestiti con una classe più sofisticata. La classe SoundEffect è una enum che ha come elementi i possibili effetti sonori. Il costruttore della classe provvede ad aprire il file .wav passato come parametro del costruttore e creare un oggetto di tipo Clip (definito anch'esso in javax.sound.sampled) associato al file audio. Attraverso il metodo play() e possibile iniziare l'esecuzione dell'effetto sonoro. Questa implementazione permette di caricare il file audio al momento del caricamento da parte di JVM della classe, ottenendo in seguito le massime performance in termini di tempo di esecuzione.

Tramite le impostazioni di gioco è possibile disabilitare sia gli effetti sonori che la musica di sottofondo.

L'idea dell'enum per gli effetti sonori e la documentazione sull'esecuzione dei midi è stata tratta dal sito dell'università di Nanyang:(http://www3.ntu.edu.sg/home/ehchua/programming/java/J8c_PlayingSound.htm)

5.2 Progettazione in dettaglio: Damiano Galassi

Per la realizzazione dell'interfaccia grafica si è deciso di realizzare una libreria, package spriteKit, sulla falsariga di cocos2d (http://www.cocos2d-x.org/) e Apple SpriteKit (https://developer.apple.com/library/mac/documentation/GraphicsAnimation/Conceptual/SpriteKit_PG/Introduction/Introduction.html), le quali permettono di semplificare lo sviluppo fornendo una solida astrazione degli aspetti legati alla creazione di giochi 2d.

La libreria è composta da 4 classi fondamentali: Node, SpriteNode, Scene, NodePanel.

La primitiva fondamentale della libreria è la classe "Node". Un nodo può essere pensato come un'area rettangolare sullo schermo contenente per esempio una immagine o altre istanze della classe Node. Un nodo in se non disegna nulla a video, ma contiene comunque una serie di proprietà visuali visto che le sue sottoclassi disegnano a video, quindi deve capire alcuni aspetti visuali.

Ogni nodo contiene diverse proprietà modificabili (posizione, dimensione, punto di ancoraggio, scala, rotazione, trasparenza, pausa) , una lista di riferimenti ai nodi figli, un riferimento al padre, ed è estendibile tramite due template method: drawInContext() e shouldRasterize().

I nodi sono disegnati a video tramite sottoclassi di Scene e NodePanel. Una scena è il nodo radice di una gerarchia di nodi. NodePanel è una sottoclasse di JPanel che mostra a video una scena alla volta.

Tramite una scena è possibile modellare una scena di un gioco o una interfaccia grafica in modo indipendente dalla dimensione di una finestra o schermo, è possibile ridimensionare la scena tramite due fattori di scala, orizzontale e verticale.

NodePanel modella un classico loop di rendering in tre fasi:

- chiama il metodo update() sulla scena attualmente visualizzata;
- chiama il metodo evaluateActions() per far avanzare lo stato delle eventuali azioni associate ai nodi;
- disegna ricorsivamente la scena a schermo.

Agisce infine da adattatore per la gestione degli eventi del mouse tra Swing e spriteKit.

Un normale programma basato su spriteKit si compone di un NodePanel e una serie di Scene, ognuna modellante una diverso momento del programma (menù principale, preferenze, livello del gioco, etc…).

SpriteNode e LabelNode sono due sottoclassi fornite dalla libreria. La prima estende Node per permettere la visualizzazione di immagini bidimensionali, e la configurazione di una serie di proprietà tra le più utilizzate nello sviluppo di interfacce grafiche: colore di sfondo, colore e spessore del bordo, smussamento degli angoli dello sfondo. La seconda, LabelNode, permette di visualizzare stringhe di caratteri a video, e consente di modificare l'allineamento orizzontale e verticale del testo.

I nodi sono animabili tramite la classe Action. Ad ogni nodo possono essere associate una o più azioni, che vengono eseguite ad ogni aggiornamento della vista. È possibile raggruppare azioni per essere eseguite contemporaneamente o in sequenza. Sono fornite delle azioni per animare le proprietà principali dei nodi. Tramite le azioni è possibile creare animazioni come dissolvenze,

rotazioni o spostamenti, e inoltre rimuovere un nodo dal nodo padre. È fornita anche un'azione per animare uno SpriteNode tramite un array di texture.

La classe Action è implementata con una serie di metodi statici per la creazione di nuove azioni predefinite, e non è pensata per essere estesa. Una nuova azione contiene solo i parametri di configurazione necessari in modo tale da rendere semplice e veloce la sua clonazione. Ciò permette di associare una azione e più nodi, i quali eseguiranno una copia al momento dell'associazione. Al momento dell'esecuzione verrà creata internamente ad Action una sottoclasse che implementa l'interfaccia IActionInternal. NodePanel ad ogni ciclo di aggiornamento chiamerà il metodo evaluateActions() su un nodo, il quale a sua volta chiamerà update() sulle azioni contenute al suo interno; update() infine creerà una nuova istanza di ActionInternal se necessario utilizzando i parametri di configurazione presenti nell'azione, ed eseguirà l'azione per il tempo stabilito dal NodePanel.

Questa architettura permette di combinare ricorsivamente le azioni per creare animazioni complesse.

La classe Texture infine si occupa di caricare le immagini bidimensionali da disco e convertirle nel formato utilizzato dalla libreria grafica sottostante, in questo caso java2d.

Texture fornisce un metodo factory, in modo da ottimizzare i tempi di caricamento tenendo in memoria le texture già utilizzate.

La gestione degli eventi del mouse è realizzata implementando l'interfaccia Responder in Node.

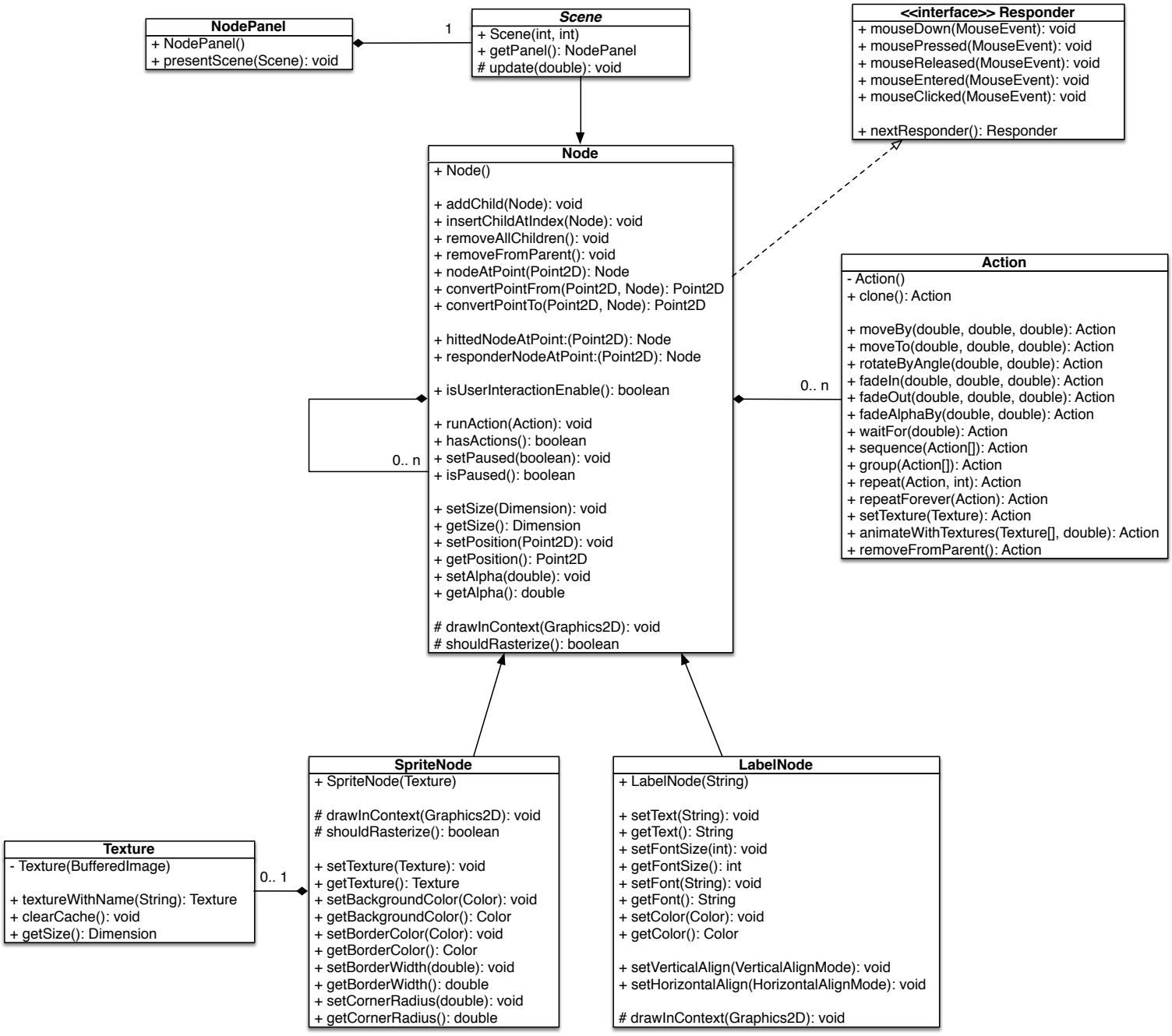
Responder contiene 5 metodi da implementare che mappano gli eventi generati dal mouse e un metodo nextResponder() con il quale una istanza può passare l'evento a un'altra istanza, solitamente l'istanza padre.

Node implementa due metodi per cercare nella sua gerarchia i nodi presenti ad un punto:

- Node nodeAtPoint(Point2D)
- Node responderNodeAtPoint(Point2D)

responderNodeAtPoint() in particolare fornisce in output solamente il nodo il cui metodo isUserInteractionEnable() ritorna un valore vero, ciò per semplificare la gestione degli eventi nel caso di gerarchie di nodi profonde.

Diagramma UML semplificato del package spriteKit

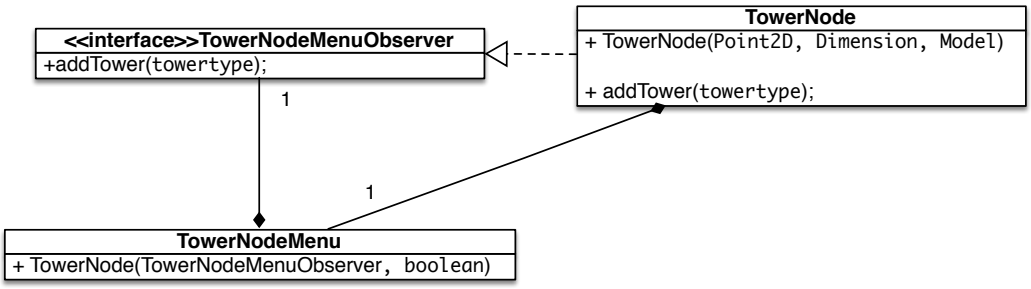


L’interfaccia del programma è composta da un controller principale, MainController, composto da una MainWindow contenente un NodePanel, utilizzato per presentare 4 scene principali:

- **MainMenuScene**: la scena con il menu principale mostrata all’avvio del programma
- **CreditsScene**: la scena di informazioni
- **MapSelectionScene**: la scena di selezione del livello del gioco
- **LevelScene**: la scena del gioco vero e proprio

LevelScene in particolare è la classe adibita al controllo di una partita del gioco. Crea una istanza della classe Model inizializzandola con un LevelMap, Rule e un path a un salvataggio. Implementa GameModelObserver, e dal modello riceve le notifiche di aggiunta, spostamento e rimozione delle unità del gioco; l’aggiornamento delle vite e delle risorse e le notifiche di fine partita.
(lo schema UML di LevelScene è incluso della sezione “progettazione architetturale)

Tramite la classe **MapNode** si disegna a video il percorso della mappa. Una serie di **TowerNode** modellano i punti possibili delle torrette e permettono l’aggiunta delle stesse. Un TowerNode è inizializzato passandogli il riferimento al modello, in questo modo può agire direttamente e può essere modificata senza dipendere dalla classe LevelScene. TowerNode usa internamente una istanza di **TowerNodeMenu** che implementa il menu di selezione del tipo di torretta.



Per gli elementi del gioco si è deciso di utilizzare una classe factory contenuta nel package **spacerush.gameViews**. Questa factory genera tutti i più comuni elementi grafici del gioco.

LevelScene utilizza due viste per il menu di pausa e di fine gioco:
- **PauseMenuNode**: prende nel costruttore una istanza di PauseMenuNodeObserver e presenta a video un menù che permette di riprendere il gioco o tornare al menù principale.
- **EndGameNode**: prende in input una istanza di EndGameNodeObserver e presenta a video un testo con l’esito della partita e un pulsate per tornare al menu principale.

Il package **spacerush.generator** fornisce una interfaccia e due classi per creare dei generatori di sfondi. Il metodo advanceSimulationTime() permette di far avanzare a piacere il tempo del generatore in modo da avere una simulazione già ben avviata a coprire l’intero nodo target con nodi figli alla presentazione di una nuova scena.

6 Testing

Viene ora presentato il testing, per mezzo dell'uso di JUnit, di alcune delle classi principale di spriteKit.

- **spriteKit.node**: il testing di questa classe viene effettuato dalla classi di testing **spriteKit.test.NodeTest**. In particolare:
    - **testNodeInsertion:** controlla il corretto inserimento di un nuovo nodo come figlio di un padre.
    - **testNodeInsertionAtIndex:** controlla il corretto inserimento di un nuovo nodo al giusto indice nella lista dei figli.
    - **testNodeCoordinateConversion:** controlla la corretta conversione delle coordinate dal sistema di riferimento di un nodo a quello di un nodo padre/figlio.
    - **testActionsInsertion:** controlla il corretto inserimento di azioni in un nodo.
    - **testActionsRemoval**: controlla la corretta rimozione delle azioni e del metodo hasActions().

Il test manuale dell'applicazione è stato condotto provando quante più possibili di combinazioni di percorsi nell'interfaccia e combinazione di esiti ed eventi nelle partite del gioco.

7 Note finali

Il processo di sviluppo è iniziato in parallelo, con lo sviluppo del prototipo del controller/vista e del modello. Appena è stato reso possibile dall'avanzare dei progressi, sono state integrate le due parti.

Il progetto ha sofferto in parte a causa di una insufficiente analisi iniziale delle caratteristiche necessarie al gioco, che ha reso necessaria un ampliamento in corso d'opera dell'interfaccia observer del modello per migliorare la comunicazione degli eventi.

La scelta di java2d come motore di rendering si è rilevata soddisfacente per le necessità del gioco, il consumo di CPU si attesta a livelli bassi, con sporadici picchi durante la creazione di nuove scene.