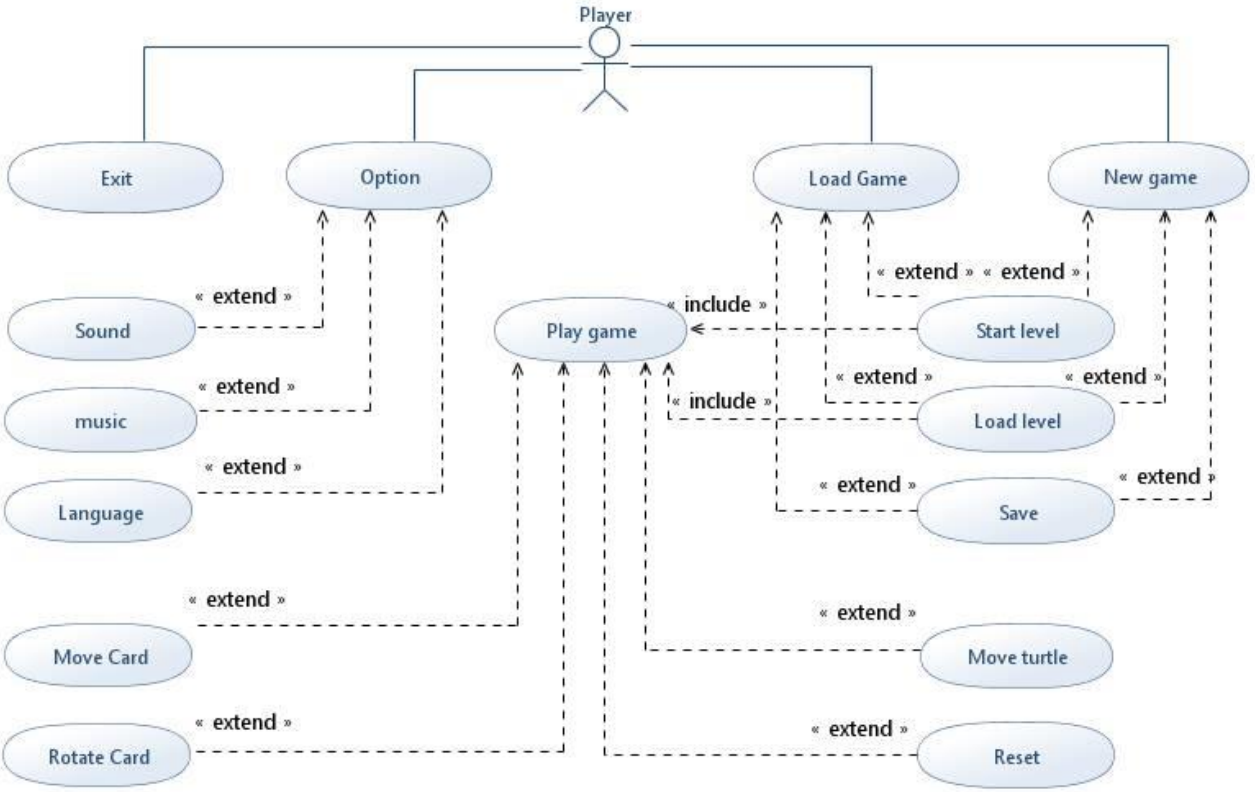
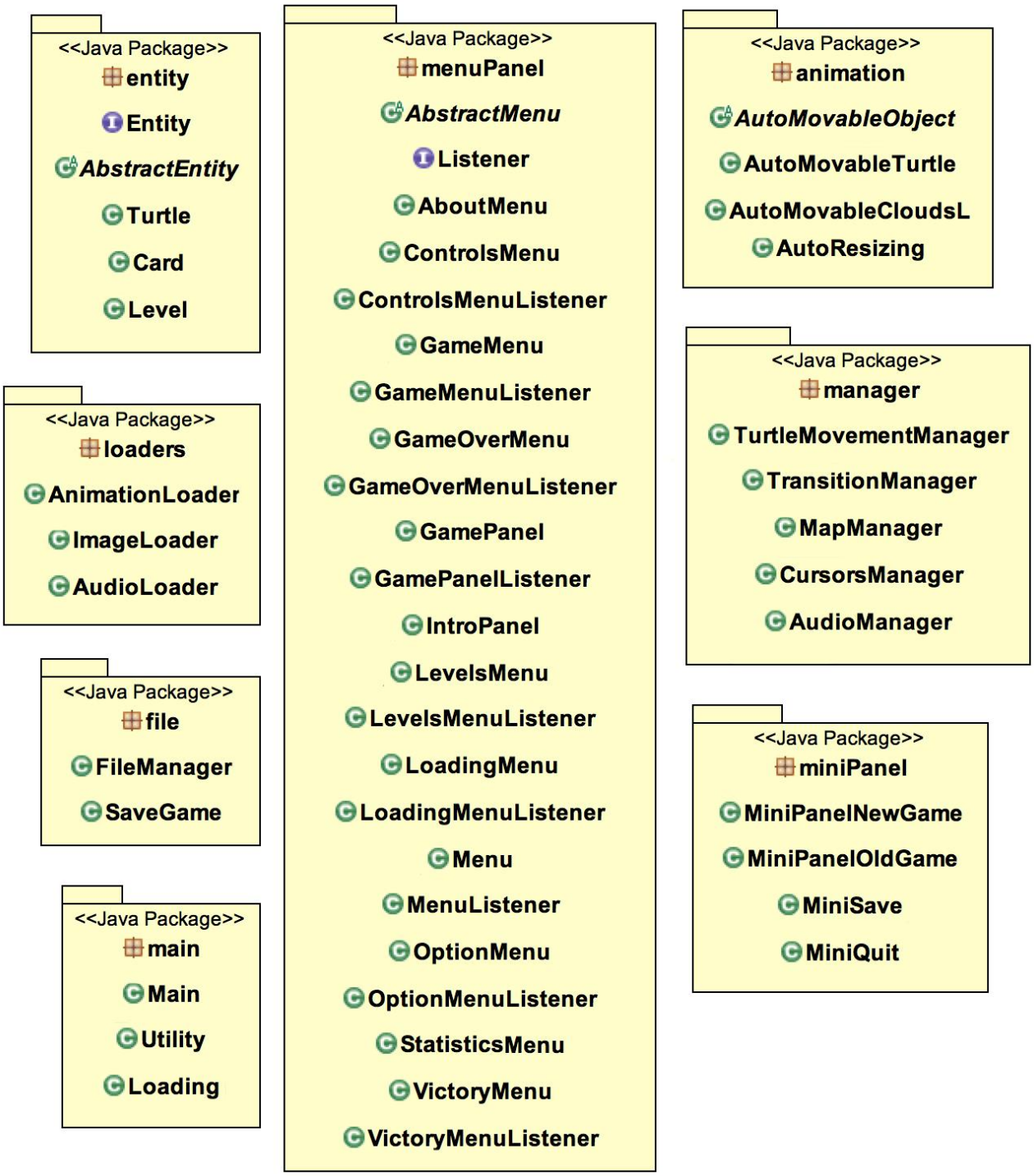
TURTLE REVOLUTION

Il progetto che si è deciso di realizzare è un gioco, al quale si è dato il nome di “Turtle Revolution”. È uno di quei giochi definiti rompicapo in quanto lo scopo principale è di far arrivare in ogni livello la tartaruga “protagonista” del gioco stesso a destinazione nel laghetto. Per riuscire in ciò si hanno a disposizione tre tessere che possono e devono essere riusate per creare un percorso che dalla partenza arriva al traguardo, identificato con la tessera del laghetto. Le tessere, a discrezione del livello che si sta giocando, possono o meno ruotare, cosa che è indicata dal cambio del puntatore del mouse quando lo si muove sulla tessera in questione. Per rendere il gioco più difficile, oltre alla tessera di partenza e arrivo, sono presenti un numero significativo di tessere su cui non è possibile transitare, gli “ostacoli”, e un numero di mosse massime oltre il quale si perde il livello. Tale numero non è legato ai movimenti della tartaruga in sé, ma al numero di allocazioni che si fanno con le tre tessere disponibili, allocazioni che ne cambiano chiaramente la posizione precedentemente assunta. È evidente che non è possibile muovere una tessera nel caso in cui la tartaruga si trovi al suo interno.

Di seguito si vede il diagramma Use Case che mostra le possibili interazioni dell’utente con il gioco.



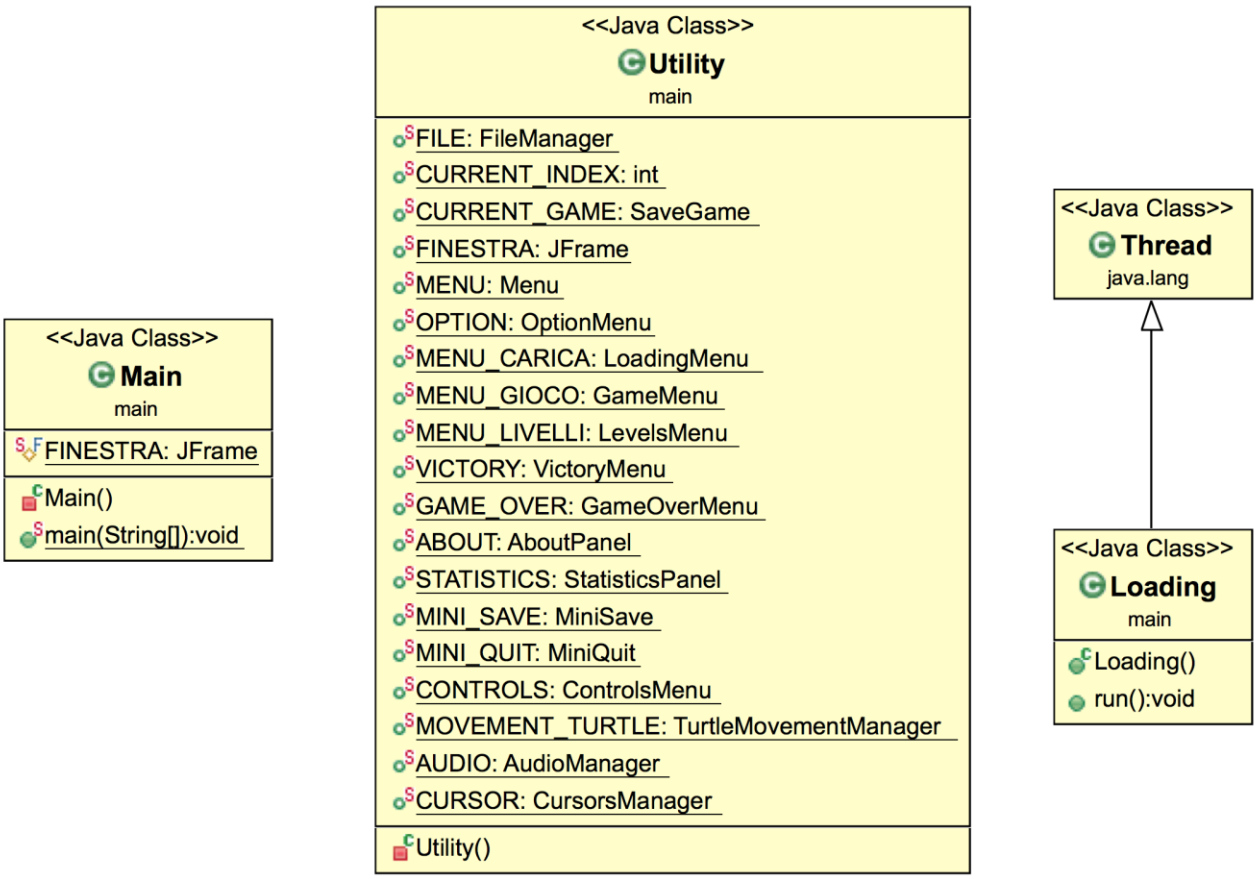
A livello di programmazione si è deciso di dividere il progetto stesso in più package, ognuno dei quali caratterizzato da classi simili tra di loro, per motivi di chiarezza e ordine all'interno del progetto stesso. Di seguito sono riportati i suddetti.



Si vedono ora nel dettaglio tutti i package.

MAIN

Il package ***main*** viene rappresentato con il seguente UML delle classi.



Tale package rappresenta la classe in cui è presente il main vero e proprio e le classi utili all'avvio del gioco.

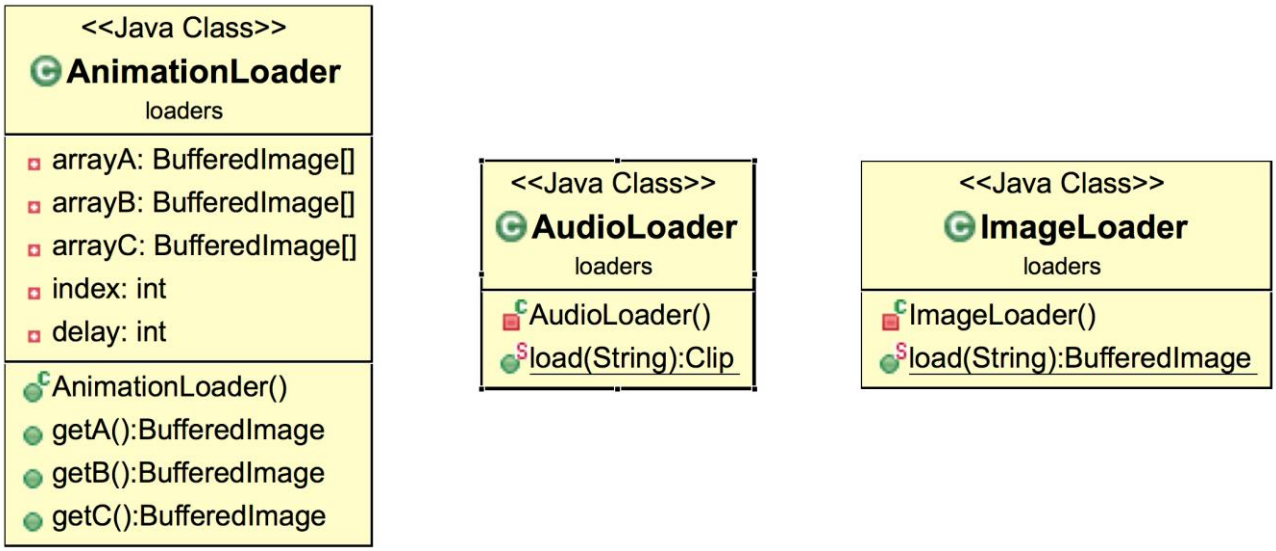
Nella classe ***Main*** vera e propria si crea il JFrame, su cui si alternano un numero di JPanel pari al numero dei menù presenti nel gioco stesso, si impostano le caratteristiche standard della finestra stessa, si carica il gioco invocando il metodo *start()* dell'oggetto di tipo *Loading* precedentemente istanziato e si avvia il gioco stesso creando l'*IntroPanel*.

La classe ***Utility*** è una classe di utilità che inizializza l'indice e il salvataggio della partita corrente, inizializza, poi, a null come variabili static tutti i JPanel, corrispondenti ai menù, e alcuni manager in modo tale da crearli solo una volta e riutilizzarli di continuo durante il gioco.

La classe ***Loading***, che è in sé un Thread, istanzia tutte le variabili static della classe *Utility* nel metodo *run()*, che ridefinisce per tale scopo, prima di interrompersi al termine di tale processo che richiede qualche secondo.

LOADERS

Il package ***loaders*** ha la seguente forma.



Tutti le classi di tale package hanno la comune caratteristica di essere classi di utilità atte a caricare immagini, animazioni o suoni.

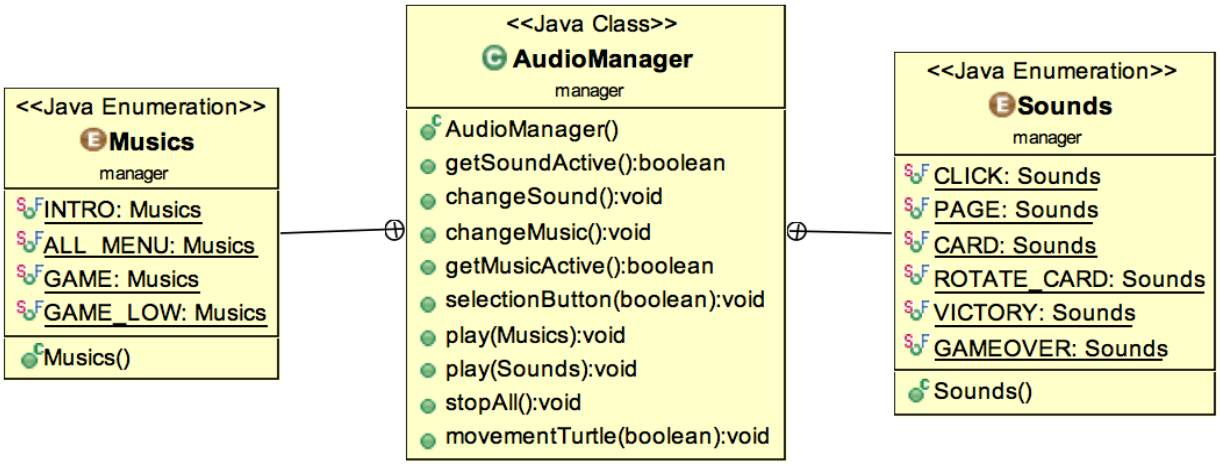
Le classi ***AudioLoader*** e ***ImageLoader*** sono classi di utilità con il compito di caricare, sfruttando il metodo *load()*, rispettivamente l'audio e le immagini, presenti nella directory specificata come parametro in ingresso, utilizzate dai vari menù.

La classe ***AnimationLoader*** si occupa, invece, di caricare le immagini relative all'animazione di una tartaruga volante presente in alcuni menù. In tale classe sono presenti tre array che rappresentano tre differenti animazioni. Per ogni array c'è un metodo *get()* che ritorna l'immagine associata, prelevano un'immagine dal vettore e incrementano l'indice utilizzando l'aritmetica modulare.

MANAGER

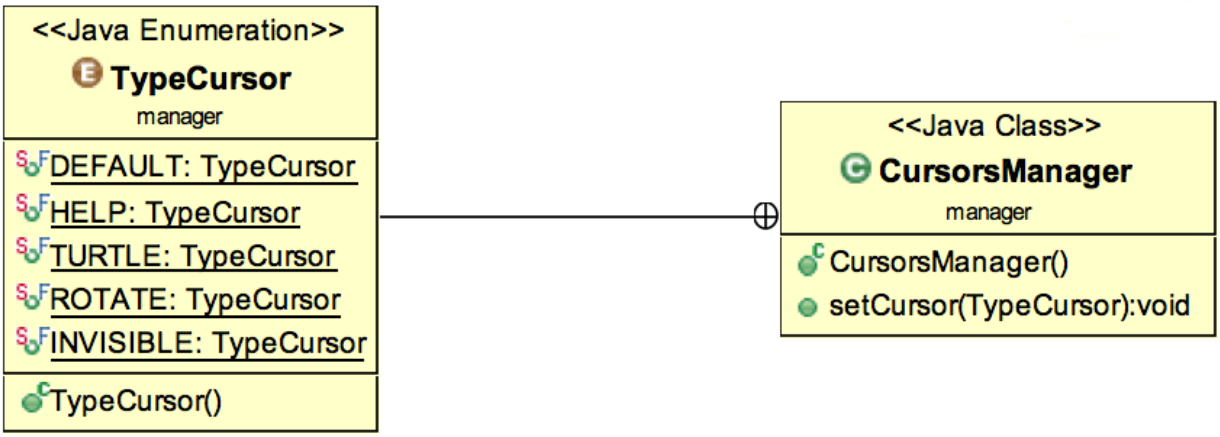
Il package ***manager*** presenta classi con il compito di caricare alcuni elementi del gioco, quali audio e cursori, e di gestire la mappa di gioco, i movimenti della tartaruga e le transizioni tra i vari menù. Si vedono ora nel dettaglio.

La classe ***AudioManager*** ha la seguente forma.



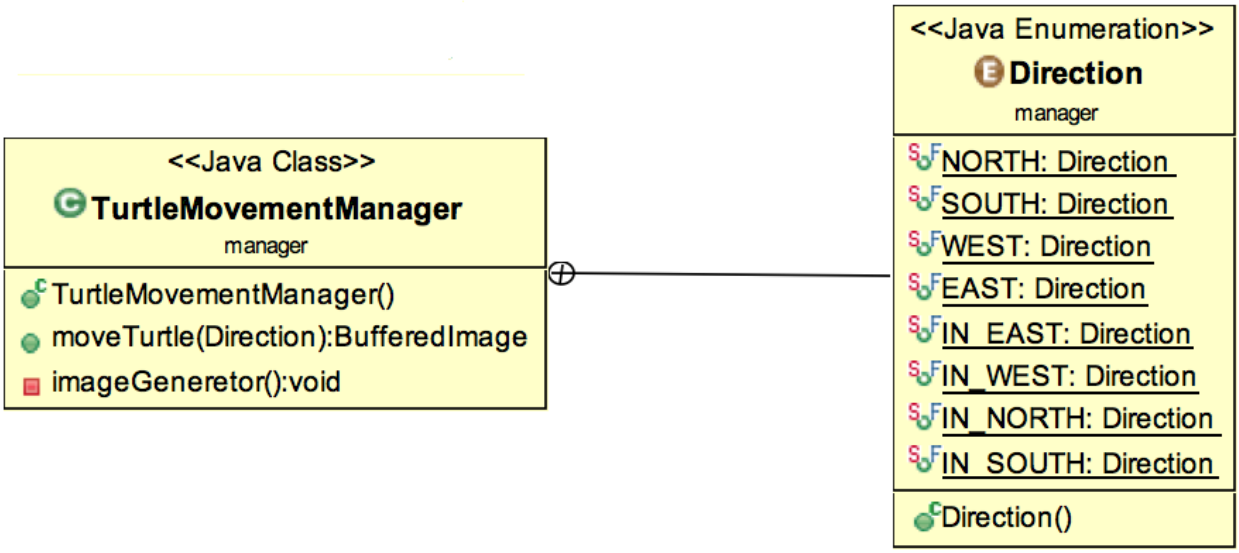
Carica tutti i suoni e la musica presenti nel gioco. Presenta metodi quali *getSoundActive()* e *getMusicActive()*, che resituiscono il valore delle rispettive variabili private SoundActive e MusicActive che servono per gestire l'attivazione o meno dei suoni e delle musiche all'interno del gioco, *changeSound()* e *changeMusic()*, che invertono il valore delle variabili private, e *stopAll()*, che ha il compito di fermare tutte le musiche. Altri metodi rilevanti sono *play()*, che ha il compito di avviare la musica o il suono associato al tipo dell'enumerazione ricevuto in ingresso, e *movementTurtle()*, che si occupa di gestire il suono relativo al movimento della tartaruga di gioco nel *GamePanel*: se la variabile ricevuta come parametro è true, allora il suono parte, se false si ferma.
Necessità di particolare attenzione il metodo *selectionButton()* che si occupa della gestione del suono relativo al passaggio del puntatore del mouse su un pulsante attivo in base al valore che riceve in ingresso: se true allora il suono parte, altrimenti controlla se il suono è attivo, nel qual caso riavvolge il file audio. Ciò viene fatto per evitare il sovrapporsi di suoni qualora si muova il mouse troppo velocemente sui tasti, sovrapposizione che risulterebbe poco gradevole, se non fastidiosa.

La classe ***CursorManager*** presenta un diagramma che è il seguente.



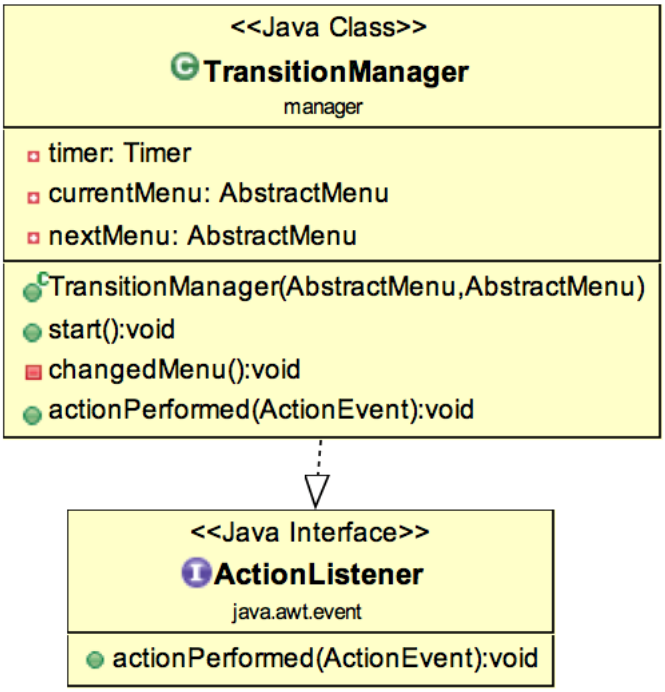
Possiede un costruttore all'interno del quale vengono creati tutti i tipi di puntatori utilizzati nel gioco e presenta un solo metodo, *setCursor()*, che imposta il cursore della finestra a seconda del tipo di puntatore richiesto ricevuto come parametro.

La classe ***TurtleMovementManager*** ha la seguente forma.



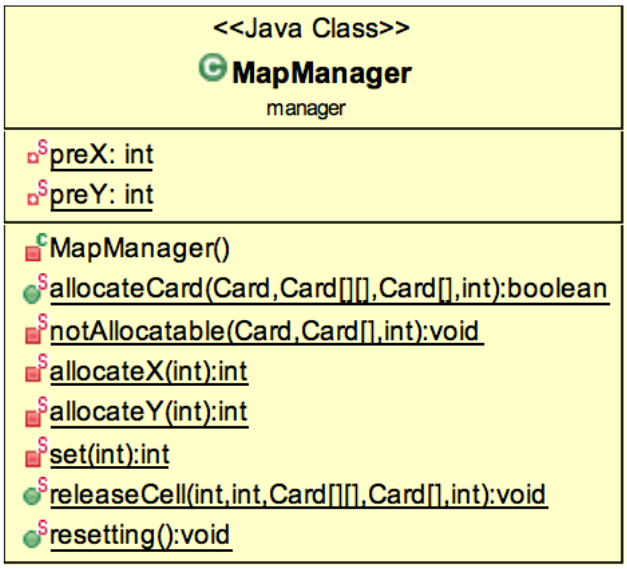
Si occupa delle animazioni della tartaruga di gioco all'interno del pannello di gioco *GamePanel* qualora venissero utilizzati i tasti direzionali.
Il costruttore carica tutte le immagini. Attraverso il metodo privato *imageGeneretor()* le immagini vengono di volta in volta ruotate e caricate in appositi array, creando, di fatto, nuovi tipi di immagini. In questo modo si evita di caricare tante immagini, ma si riusano sempre le stesse, opportunamente ruotate. Il metodo *moveTurtle()* non fa altro che restituire l'immagine del vettore associata alla direzione indicata dalla variabile ricevuta come parametro. È bene notare che gli indici dei vettori vengono incrementati attraverso l'utilizzo dell'aritmetica modulare.

La classe ***TransitionManager*** gestisce la transizione da un menu all'altro, attraverso l'utilizzo di alcune caratteristiche insite agli AbstractMenu. Ha la seguente forma.



Il suo funzionamento è abbastanza semplice. Il costruttore ha il computo di creare e istanziare un oggetto di tipo Timer e di associare alle variabili private currentMenu e nextMenu i pannelli tra i quali si sta effettuando la transizione. Il metodo *start()* non fa altro che avviare il Timer precedentemente istanziato. Tale fatto così banale è, in realtà, di grande importanza. Infatti, la classe *TransitionManager* implementa *ActionListener* e ridefinisce il metodo *actionPerformed()* che, basato sugli eventi creati dal Timer, invoca, di continuo, il metodo privato *changedMenu()* che ha il compito di verificare che la transizione in uscita del currentMenu sia terminata. Quando questo è effettivamente successo, sostituisce il pannello con il nextMenu e chiama la transizione in ingresso di tale pannello appena aggiunto. Finito tale processo il Timer viene fermato. È bene notare che le transizioni effettive in entrata e in uscita non sono effettuate dal *TransitionManager*, ma sono gestite dall'*AbstractMenu* in quanto la possibilità di smaterializzarsi è una proprietà insita ai menù (per l'*AbstractMenu* si veda più avanti, ndr).

La classe ***MapManager*** è una classe di utilità che si occupa di gestire la mappa di gioco, verificando che le mosse siano possibili. La maggior parte dei metodi di tale classe necessitano di una descrizione approfondita. Si vede, innanzitutto, il diagramma.

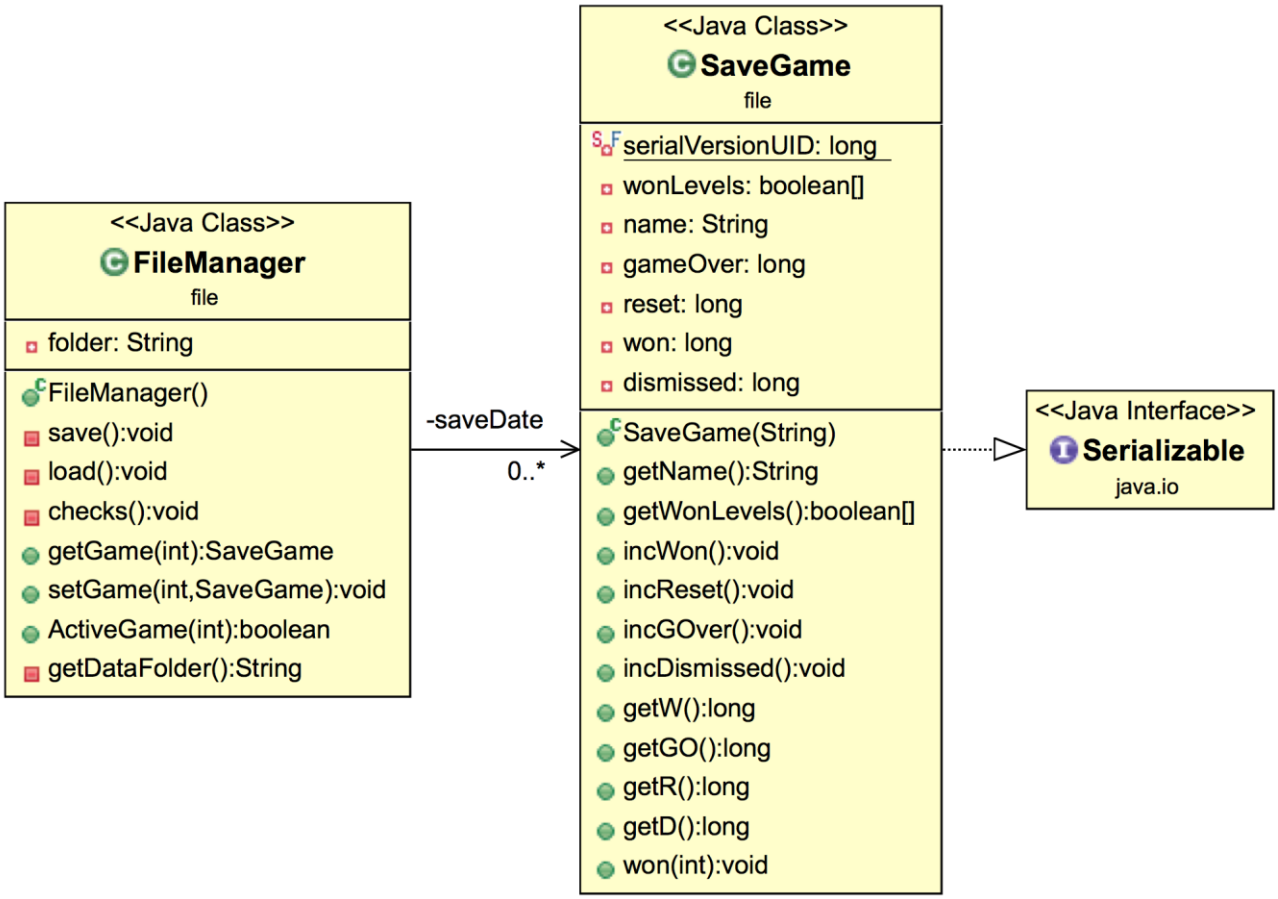


Il metodo privato *notAllocatable()* è invocato nel momento in cui la tessera ricevuta come parametro non si può collocare nella posizione in cui il giocatore sta tentando di metterla. Il suo compito è quello di riposizionare la tessera nella sua posizione originaria.
I metodi privati *allocateX()* e *allocateY()* hanno il compito di trasformare la coordinata grafica ricevuta come parametro nel corrispettivo indice associato alla matrice. Nel caso in cui il valore non fosse ammissibile restituisce -1. Tali metodi fanno riferimento rispettivamente all'asse orizzontale e all'asse verticale.
I metodi privati *set()* non ha altro che la funzione opposta ai metodi precedentemente descritti: ricevuto un indice, ne restituiscono la coordinata grafica associata.
È bene fare una precisazione sulle variabili statiche preX e preY che rappresentano le precedenti coordinare dell'ultima tessera utilizzata.
Il metodo privato *releaseCell()* ha il compito di liberare dalla mappa la posizione indicata dalle coordinate grafiche ricevute come parametro, aggiornando, inoltre, le variabili preX e preY, alle quali viene sommato il valore di dell'intero n, ricevuto come parametro, che rappresenta la tessera corrente che si sta considerando (se la 0, la 1 o la 2). Ciò è di particolare rilevanza perché serve a distinguere le tessere una dall'altra ed evitare poi errori di logica nel momento del gioco della partita.
Il metodo *allocateCard()* posiziona se possibile una tessera nelle posizioni xy della tessera stessa. Tale metodo ha, infatti, il compito di verificare la posizione corrente della tessera. Se tale posizione non è valida o se la cella della matrice è già occupata chiama il metodo *notAllocatable()* e ritorna false, altrimenti tale cella viene

occupata dalla tessera stessa ricevuta come parametro, le cui coordinate correnti vengono impostate in modo opportuno. In un if di controllo viene fatto un confronto fra i valori delle variabili preX e preY con i valori correnti delle coordinate della tessera con l'aggiunta della variabile n ricevuta anch'essa come parametro: se le posizioni precedenti e correnti sono uguali il metodo ritorna false, perché si sta mettendo la tessera nella posizione che occupava subito prima che il metodo venisse chiamato, quindi subito prima di essere spostata e essere riallocata. Questo è un fatto di straordinaria rilevanza perché è strettamente collegato alla struttura del gioco, struttura che prevede che se si prende una tessera da una posizione e la si rimette nella stessa posizione le mosse non decrescono. A questo punto diventa necessaria una distinzione delle tessere perché è vero che le mosse non devono decrementarsi se si rimette una tessera nella posizione assunta subito prima lo spostamento, ma posizionare una tessera diversa da essa in quella stessa posizione viene considerata una nuova allocazione, quindi una nuova mossa. Ad assolvere questo compito è chiamata proprio la variabile n che evita di posizionare una tessera nella posizione precedentemente assunta da un'altra tessera senza decrementare le mosse. È bene notare che preX e preY vengono aggiornati nel *releaseCell()* con le coordinate ricevute come parametro in aggiunta della variabile n: proprio grazie a questo il confronto di cui si è parlato assume un senso. È bene notare che tale situazione di cui si è parlato non è gestita direttamente da un metodo solo, ma da *releaseCell()* e *allocateCard()* insieme.

FILE

Il package ***file*** è caratterizzato dalle classi necessarie al salvataggio su file del gioco ed è così suddiviso.



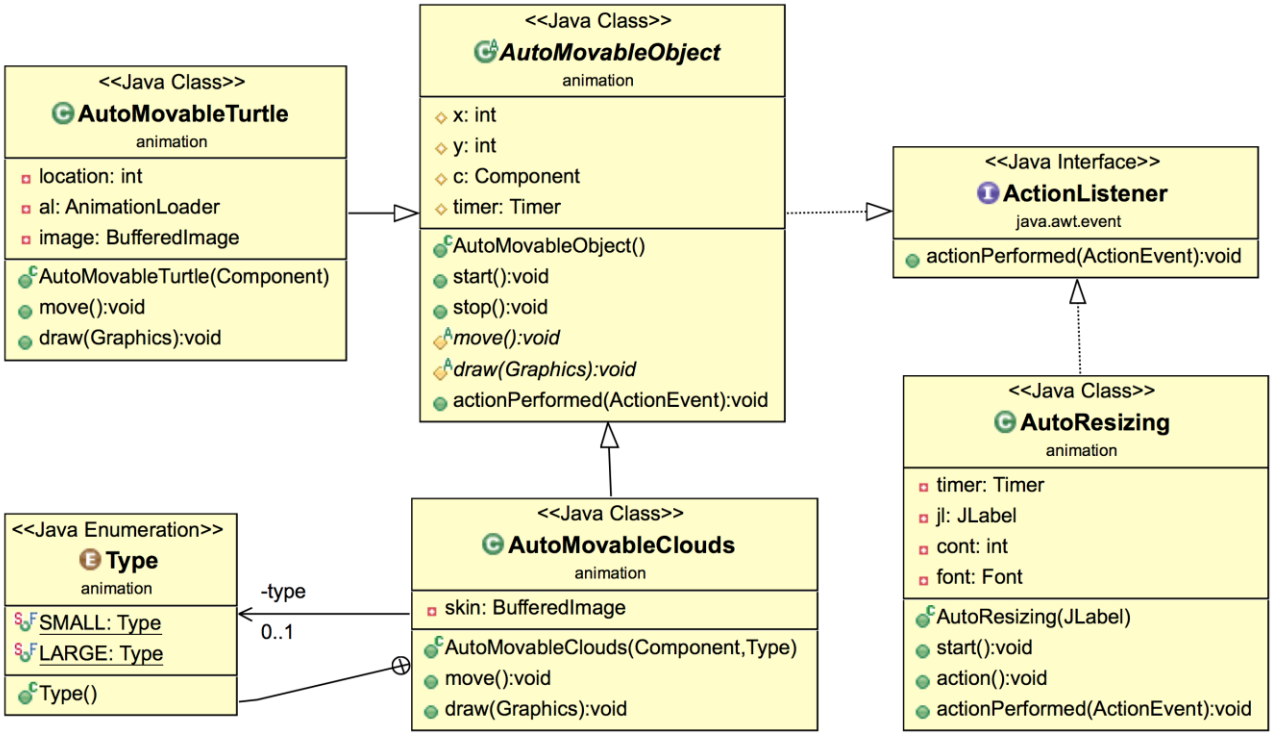
La classe ***FileManager*** gestisce le interazioni con il file di salvataggio. Il costruttore crea un array di tre oggetti di tipo *SaveGame* che rappresentano le partite salvate: tre perché tre sono gli slot che il gioco mette a disposizione per le partite salvate. Dopodiché invoca il metodo *getDataFolder()*, che crea la directory in cui effettuare i salvataggi in base al sistema operativo correntemente in uso, e il metodo *checks()*, che effettua

controlli sul file: crea l'oggetto di tipo *File*, se tale oggetto non esiste lo crea invocando il metodo *save()*, se esiste lo carica attraverso il metodo *load()*, se è danneggiato lo cancella e invoca, anche in questo caso, il metodo *save()*. Intuitivamente il metodo *getGame()* restituisce l'n-esima partita dell'array, con n ricevuto come parametro, di partite salvate, mentre *setGame()* sostituisce l'n-esima partita con quella ricevuta come parametro e salva il gioco.

La classe ***SaveGame*** rappresenta la partita di gioco nel file di salvataggio. È una classe in sé molto semplice, il cui costruttore associa il nome della partita a quella ricevuta come parametro. Possiede una serie di metodi *get()* di facile intuizione, tra i quali, in particolare, si ha il metodo *getWonLevels()* che restituisce il contenuto della variabile wonLevels, un vettore di boolean che corrisponde ai livelli superati. I metodi che riportano il prefisso "*inc*" incrementano i valori delle corrispondenti variabili e, infine, il metodo *won()* segna come vinto il livello ricevuto come parametro nel vettore di boolean di cui sopra.

ANIMATION

Il package ***animation*** è così definito.



Ha in sé le classi che garantiscono delle animazioni all'interno del gioco.

La classe ***AutoMovableObject*** rappresenta un oggetto, la cui immagine, la skin nelle varie classi concrete, è capace di muoversi autonomamente sullo sfondo di un Component. La classe astratta, che implementa *ActionListener*, presenta una serie di metodi legati alla variabile di tipo Timer di cui è dotata. Implementa poi il metodo *actionPerformerd()* che si occupa di chiamare continuamente il metodo *move()* e il *repaint()* del Component su cui si sta lavorando. La classe lascia poi da implementare i metodi astratti *move()* e *draw().*

La classe ***AutoMovableTurtle*** rappresenta un oggetto auto movibile di tipo *Turtle*, estensione dell'oggetto astratto la cui classe è sopra descritta. È di particolare rilevanza la presenza dell'*AnimationLoader* tra le

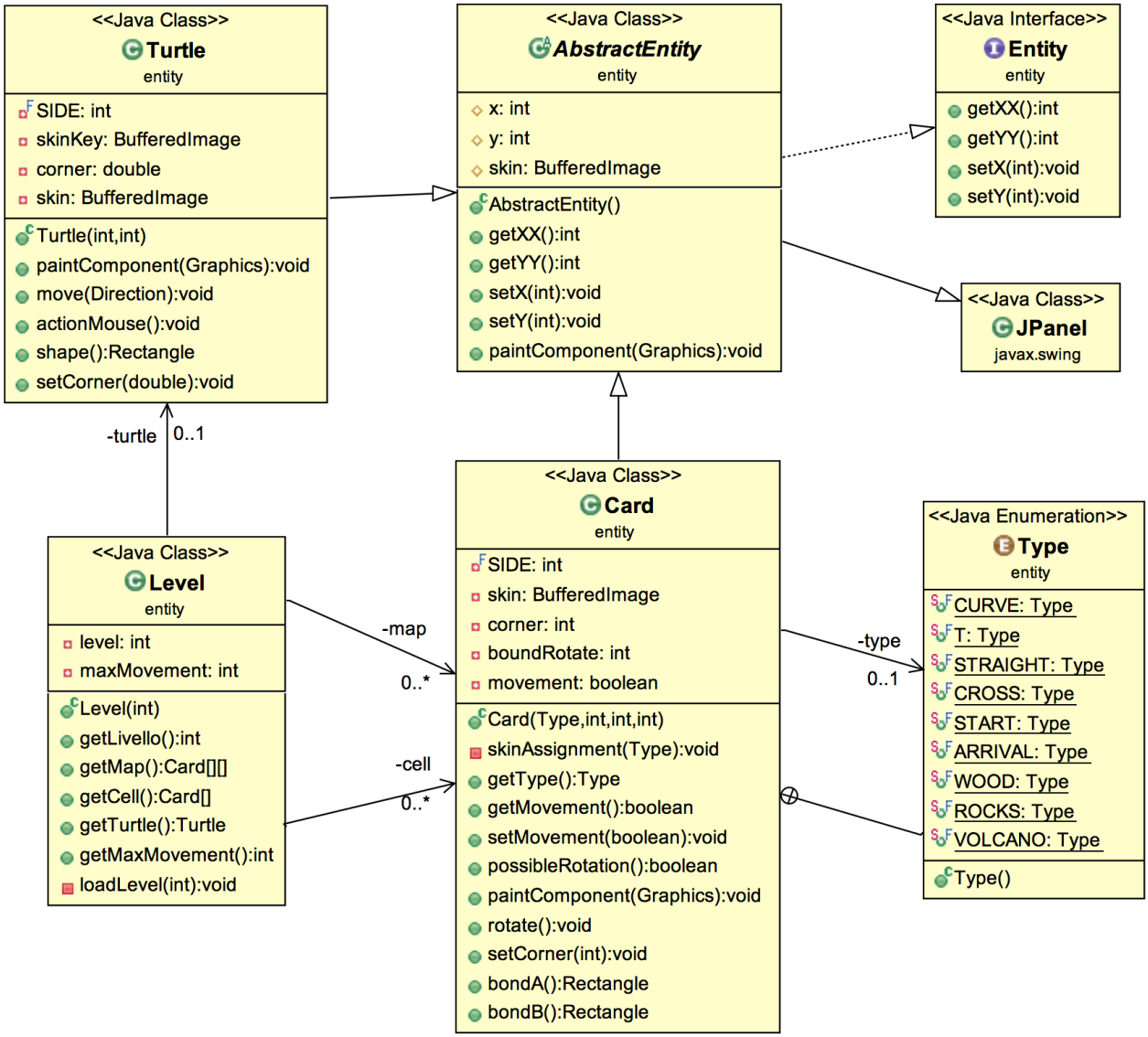
variabili d'istanza, cosa che la differenzia dall'altro erede di *AutoMovableObject*. Il costruttore imposta l'immagine nelle posizioni predefinite, quelle di partenza, associa il Component, che riceve come parametro, alla specifica variabile d'istanza e inizializza il Timer ricevuto dalla classe astratta. Ridefinisce quindi i metodi astratti ricevuti: *move()* cambia il valore delle variabili x e y e di location in casi gestiti da if specifici, il metodo *draw()* disegna l'immagine dell'oggetto in base al valore di location, ovvero in base alla posizione dell'oggetto stesso.

La classe ***AutoMovableClouds*** rappresenta un oggetto auto movibile di tipo *Clouds*, estensione dell'oggetto astratto la cui classe è sopra descritta. La classe presenta un'enumerazione in quanto esistono due tipi di *Clouds*. Il costruttore associa il Component e il tipo ricevuti come parametri con le variabili opportune, in base al tipo di *Clouds* colloca l'immagine dell'oggetto in posizioni predefinite, inizializza il timer con valori opportuni e carica la skin con l'immagine appropriata attraverso l'uso della classe di utilità *ImageLoader*. Il metodo *move()* cambia i valori della variabile x in base al tipo, in quanto tale oggetto si muove solo sull'asse orizzontale, a differenza di *Turtle* che si spostava anche in diagonale sullo schermo, mentre il metodo *draw()* disegna semplicemente la skin dell'oggetto nelle posizioni x e y.

La classe ***AutoResizing***, a differenza delle altre classi concrete presenti nel package, non estende *AutoMovableObject*, ma è comunque un'animazione, pertanto la sua presenza in tale package è giustificata. Rappresenta un oggetto in grado di ridimensionare il contenuto della JLabel. In particolare è usata, all'interno del gioco, per un'animazione quando le mosse decrescono: ogni volta che l'allocazione di una tessera risulta valida le mosse diminuiscono con l'animazione data dall'oggetto *AutoResizing*. Propriò ciò giustifica la presenza dell'avvio del suono di allocazione di una tessera nel metodo *start()*: ogni volta che si ha un'animazione di questo tipo significa che la tessera viene allocata, pertanto il suono deve partire.
Il costruttore, che riceve la JLabel sui cui lavorare, imposta le variabili in modo opportuno, associando il parametro ricevuto, prelevandone il font, inizializzando un contatore e un Timer.
Il metodo *start()* da avvio all'animazione: reimposta a zero il valore del contatore, ingrandisce il font e lo imposta alla JLabel e avvia il timer. Avviando il timer fa sì che l'animazione prosegua: la classe implementa, infatti, *ActionListener*, pertanto, nel momento in cui il timer viene avviato il metodo *actionPerformer()* inizia a lavorare, invocando, a sua volta, il metodo *action().* Quest'ultimo incrementa il contatore che, quando raggiunge un valore prefissato, permette di entrare in un if nel quale si blocca il contatore e si reimposta la dimensione del font a quella precedente l'animazione. In definitiva, tale animazione viene usata per ingrandire per un brevissimo periodo di tempo il numero di mosse disponibili di ogni livello ogniqualvota vengono decrementate.

ENTITY

Il package ***entity***, che presenta al suo interno le classi che rappresentano le entità fondamentali del gioco, ha la seguente forma.



L'interfaccia ***Entity*** rappresenta il generico elemento presente all'interno del pannello di gioco. Prevede una serie di metodi, quali i *get()* e *set()* per le coordinate x e y. Degno di nota è il nome dei *get* che presentano doppia lettera: questo per evitare qualsiasi tipo di contrasto tra i metodi propri dell'interfaccia e quelli di JPanel: si sarebbero ridefiniti i metodi del JPanel stesso, metodi che in realtà servono come servono quelli dell'interfaccia.

La classe astratta ***AbstractEntity*** rappresenta il generico elemento astratto necessario ai fini del gioco. La classe estende JPanel. Tale estensione, piuttosto che l'utilizzo di una semplice immagine, è un fatto puramente pratico. Tra le *AbstractEntity* figurano, infatti, sia la tartaruga di gioco che le tessere che supportano e/o necessitano di essere mosse attraverso il drag del mouse: se si fossero utilizzate delle immagini nel momento in cui durante il drag si avesse un repentino movimento del mouse stesso, l'immagine e il puntatore non avrebbero assunto la stessa posizione e nel momento dell'allocazione della tessera ciò avrebbe portato a errori all'interno del gioco. Questo perché durante suddetto movimento non tutte le coordinate vengono percepite, problema che, invece, non accade con l'estensione di JPanel in quanto il drag del mouse è legato strettamente ad esso e non incorre in nessun caso a problemi di cui sopra. Vengono definiti i metodi dell'interfaccia *Entity* e si lascia da definire, invece, il metodo *paintComponent()*.

La classe ***Turtle*** rappresenta l'entità della tartaruga di gioco. Il costruttore imposta le dimensioni dell'oggetto, la sua posizione e carica le immagini delle skin, una delle quali è usata come supporto.
Il metodo *move()* muove la skin della tartaruga attraverso il metodo *moveTurtle()* della classe TurtleMovementManager, passandogli il parametro che esso stesso riceve. Il metodo *actionMouse()* ruota la skin della tartaruga quando viene invocato in base al valore della variabile corner attraverso la creazione e l'uso di un AffineTransform. Tale valore di corner si imposta con il metodo *setCorner()*.
Altri metodi presenti nella classe sono *shape()* che restituisce il rettangolo rappresentante la tartaruga, di fondamentale importanza per gestirne i movimenti all'interno delle tessere (per questo si veda poi la descrizione del *GamePanelListener*, ndr), e il *paintComponent()* che disegna la skin della tartaruga.

La classe ***Card*** rappresenta l'entità di gioco tessera. Presenta un'enumerazione in quanto si hanno più tipi di tessera con immagini differenti. Il costruttore associa alle variabili di istanza opportune il numero di rotazioni della tessera, le sue coordinate e il tipo di tessera che riceve come parametro, imposta a 0 di default l'angolo di rotazione della tessera stessa e il movimento possibile della tessera stessa a false. Infine, per assegnare l'immagine al tipo di tessera invoca il metodo *skinAssignment().*
È bene notare che la tessera è un elemento fondamentale del gioco in quanto al suo interno la tartaruga si deve muovere per raggiungere il traguardo. Chiaramente il movimento della tartaruga all'interno della tessera è vincolato in alcune direzioni ben precise. Per permettere ciò, l'oggetto *Card* è dotata di vincoli, i *bonds*, il cui numero è legato al tipo di tessera stessa, che sono rappresentati da rettangoli restituiti dai metodi *bondA()* e *bondB()* che lavorano attraverso l'uso di switch.
Seguono alcuni metodi di semplice comprensione, quali *possibleRotation()*, che informa sulla possibilità di effettuare una rotazione sulla tessera in questione, i *get* e i *set* relativi a variabili specifiche dell'oggetto e il *paintComponent()* che disegna la skin dopo averla ruotata in base al valore della variabile corner.
Una nota di rilievo è doverosa farla per il metodo *rotate()* che si occupa di ruotare la tessera in base alla variabile boundRotate:

- Se pari a 0, allora il metodo si interrompe e nessuna rotazione è consentita;
- Se minore di 0, allora può effettuare due rotazioni adiacenti, ovvero la tessera in posizione normale e poi ruotata di 270°;
- Se pari, allora può effettuare due rotazioni di 180° per volta;
- Altrimenti è dispari e può effettuare tutte le rotazioni.

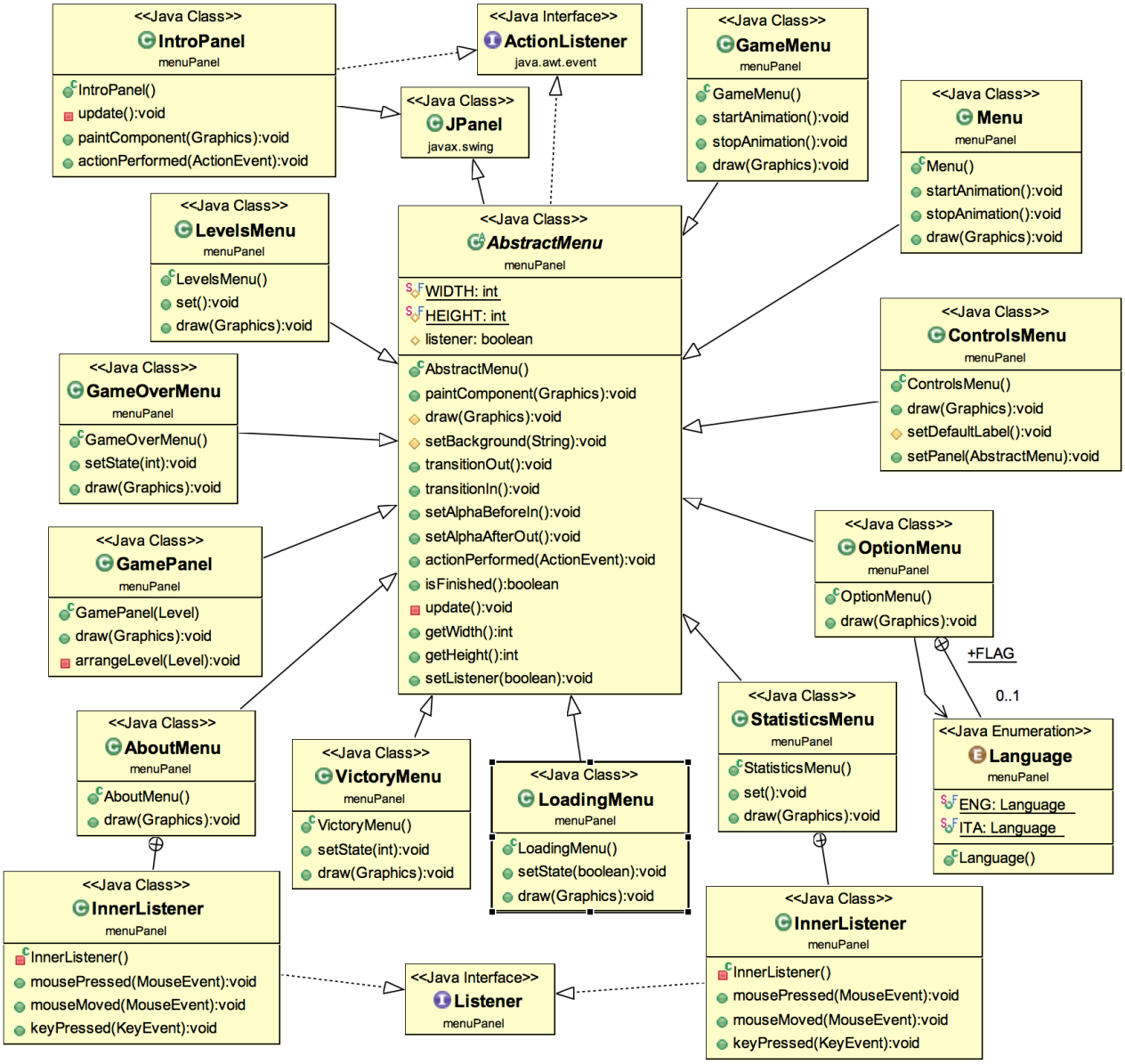
Dopo aver ruotato il metodo effettua il *repaint()* della tessera per rendere effettivamente visibile la rotazione e invoca il *play()* dell'audio relativo alla rotazione stessa.

La classe *Level* rappresenta l'oggetto livello necessario ai fini del gioco. La classe è, in sé, molto semplice. Il costruttore riceve un intero da associare alla variabile level che corrisponde al livello corrente da creare e giocare. Il costruttore richiama quindi *loadLevel()* che, attraverso uno switch, carica effettivamente il livello in base al valore del parametro ricevuto, posizionando le tessere rappresentanti gli ostacoli, impostando le rotazioni, le mosse massime e i tipi di tessera da utilizzare e posizionando la tartaruga di gioco. I metodi *get()* ritornano le opportune variabili di istanza.

MENUPANEL

Il package ***menuPanel*** presenta tutte le classi relative ai menù presenti nel gioco e i loro ascoltatori. In particolare, quest'ultimi non sono stati inseriti come InnerClass all'interno dei relativi menù, ma come classi separate per rendere il codice più leggibile. Tuttavia, poiché utilizzano per la maggior parte variabili dei menù ai quali sono legati, non si è potuti dividerli in package diversi a meno di non rendere pubbliche suddette variabili; si è deciso, quindi, di renderle protected e di inserire tutto nello stesso package. Si vede, innanzitutto, la descrizione dell'*AbstractMenu* e delle relative classi astratte e, inseguito, la descrizione dell'interfaccia *Listener* con relative classi.

È così rappresentabile.



La classe ***AbstractMenu*** rappresenta il generico menù presente all'interno del gioco che possiede determinate delle caratteristiche, quali la possibilità di effettuare transizioni in entrata e in uscita (gestite e usate dal *TransitionManager*, ndr) e le dimensioni del pannello stesso.
Il metodo *paintComponent()* disegna lo sfondo del menù e invoca il metodo astratto *draw()* che i metodi concreti dovranno poi ridefinire. Il metodo *setBackground()* imposta l'immagine di sfondo, caricandola attraverso l'*ImageLoader* nella directory ricevuta come parametro. La classe implementa *ActionListener* e ridefinisce il metodo *actionPerformed()* che, basato sul Timer avviato nei metodi *transitionIn()* e *transitionOut()*, invoca il metodo *update()* e il *repaint()*. Proprio il metodo *update()* ha il compito di smaterializzare, in entrata e/o in uscita, i menù durante le transizioni da uno all'altro: non fa altro che aggiornare continuamente, sfruttando il timer, il valore della variabile alpha che si occupa della trasparenza di un rettangolo che viene disegnato davanti tutto il pannello: nella transizione in uscita si passa da un valore alpha nullo, pari alla trasparenza, a un valore massimo che coincide con il bianco, viceversa per la transizione in entrata. Il metodo *isFinished()* verifica che la transizione sia terminata verificando lo stato del timer, che si ferma in *update()* in casi ben precisi regolati da if.

I metodi *get* e *set* sono di facile comprensione. In particolare, il *setListener()* è usato per rendere o meno attivi gli ascoltatori del menù in questione nel caso in cui ad esso venga aggiunto un *miniPanel*.
Un'altra nota va fatta per la presenza del *setLayout(null)* nei costruttori di alcuni menù: questo avviene perché il JPanel fornisce un layout predefinito che si è deciso di non utilizzare in alcuni casi in quanto risultava inadeguato, da qui la necessità di impostarlo a null.

Si vedono ora nel dettaglio tutti i menù che hanno tra le caratteristiche comuni l'utilizzo dei rettangoli come bottoni e l'uso di più tipi di immagini per i bottoni stessi: il gioco è progettato per avere due lingue, Italiano e Inglese, pertanto le immagini cambiano di conseguenza. Tali immagini vengono tutte caricate sempre nei costruttori e poi disegnate sul Component in base alla lingua corrente nei *draw()* di tutte le classi.

La classe ***AboutMenu*** rappresenta il menù delle informazioni sul gioco. Presenta un costruttore in cui imposta le caratteristiche standard del pannello, aggiunge l'ascoltatore in modo appropriato dopo averlo creato e crea l'unico bottone presente all'interno di tale menù. In questo caso l'ascoltatore della classe, poiché veramente semplice, è inserito come inner class.

La classe ***ControlsMenu*** rappresenta il menù delle informazioni sui controlli mouse e tastiera all'interno di tutto il gioco. Come per l'*AboutPanel* presenta un costruttore in cui imposta le caratteristiche standard della finestra e, in più, crea delle JLabel, ne imposta le condizioni di default anche attraverso il metodo *setDefaultLabel()*, carica due font, uno per il testo normale dei comandi, uno più grande e in grassetto per i nomi dei menù di cui si sta facendo la descrizione, e aggiunge le JLabel stesse al pannello.
Il metodo *draw()* imposta il testo e le immagini in base alla pagina in cui si è correntemente e in base alla lingua in uso. Il metodo *setPanel()* imposta la variabile AbstractMenu che possiede protected come variabile d'istanza con quella ricevuta come parametro e imposta la pagina a zero. Va dedicata una nota di rilievo per questo metodo: tale menù può essere raggiunto da due menù, pertanto nel momento in cui si preme il tasto back per tornare indietro si deve sapere a quale dei due tornare e proprio questo è il compito di tale metodo che viene invocato prima di avviare la transizione in entrata. Poiché è invocato sempre prima che il menù effettivamente sia visibile se ne imposta la pagina a zero cosicché da vedere sempre la pagina iniziale al momento della sua apertura.

La classe ***OptionMenu*** rappresenta il menù delle opzioni. Presenta un'enumerazione legata al tipo di lingua possibile alla quale tutte le altre classi possono fare riferimento. Il costruttore imposta le caratteristiche standard del pannello e carica le immagini opportune che vengono disegnate nel metodo *draw()* dopo appropriati controlli sulla lingua e sull'audio.

La classe ***StatisticsMenu*** rappresenta il menù delle statistiche. Il costruttore imposta le caratteristiche standard, crea dei font e delle JLabel e le aggiunge dopo averle posizionate. Il metodo *set()* imposta, in base alla lingua, il testo presente all'interno delle JLabel, prelevando da file dalla partita corrente i valori delle statistiche, mentre *draw()* disegna le immagini seguendo la stessa logica. In questo caso l'ascoltatore della classe, poiché veramente semplice, è inserito come inner class.

La classe ***IntroPanel*** rappresenta il pannello di intro del gioco. È una sorta di caricamento che fornisce il tempo necessario alla classe *Loading* di creare tutti i menù presenti nel gioco, processo che richiede qualche secondo.
A differenza degli altri menù, non estende *AbstractMenu*, ma estende *JPanel* e *ActionListener*, in quanto non si tratta effettivamente di un menù, ma di una semplice intro al gioco.
Il costruttore imposta le caratteristiche standard del pannello, inizializza e avvia il Timer su cui l'animazione si basa e carica le immagini. La classe implementa *ActionListener* e ridefinisce il metodo *actionPerformed()* che richiama continuamente i metodi *update()* e *repaint()*. Il metodo *update()* aggiorna continuamente, sfruttando il timer, il valore della variabile alpha che si occupa della trasparenza di un rettangolo che viene disegnato davanti tutto il pannello: si passa da un valore di alpha massimo, pari al bianco, a un valore di alpha

nullo, pari alla trasparenza, per poi effettuare nuovamente una transizione in uscita verso il menù successivo, ovvero il menù iniziale del gioco. Oltre alla transizione sono presenti due animazioni:

- la prima consiste nella corsa di una tartaruga che funge da percentuale di caricamento della pagina di intro stessa. È data dal susseguirsi di una sequenza di immagini le cui coordinate cambiano, dando così l'impressione del movimento;
- la seconda consiste nella comparsa dei nomi: si taglia un'immagine di supporto, posizionata sopra le immagini che devono apparire, in modo tale da creare un'animazione per la comparsa delle scritte delle immagini. Ciò è creato attraverso l'uso di contatori e di subImage.

La classe ***VictoryMenu*** rappresenta il menù di vittoria. L'unico metodo rilevante è il *setState()* che imposta la variabile nextLevel con quella passatagli come parametro. Il costruttore e il metodo *draw()* si comportano esattamente come negli altri menù.

La classe ***GameOverMenu*** rappresenta il menù che appare quando si perde una partita. Anche in questo caso il costruttore e il *draw()* si comportano come negli altri menù, mentre differisce il metodo *setState()* che imposta il valore della variabile level con quella ricevuta come parametro.

La classe ***LevelsMenu*** rappresenta il menù dei livelli, nel quale è possibile selezionare e/o vedere i livelli sbloccati. Di rilievo è il metodo *set()* che imposta la pagina a zero e associa alla variabile games i livelli superati o meno della partita corrente.

La classe ***LoadingMenu*** rappresenta il menù di caricamento e di creazione di una nuova partita. Nel costruttore, oltre alle operazioni standard, crea i font e le JLabel, con delle caratteristiche opportune, necessarie all'interno del menù stesso. Il numero delle JLabel è pari al numero massimo di partite salvabili contemporaneamente.
Rilevante è la variabile state alla quale il metodo *setState()* fa riferimento. Grazie ad essa, infatti, si riesce a capire se si sta utilizzando il *LoadingMenu* come menù di caricamento delle partite salvate o come menù in cui creare nuove partite o sovrascrivere quelle precedenti.

La classe ***Menu*** rappresenta il menù iniziale del gioco. Oltre alle operazioni standard, nel costruttore vengono creati tre oggetti di tipo *AutoMovableObject*. I metodi *startAnimation()* e *stopAnimation()* rispettivamente iniziano e fermano le animazioni di suddetti oggetti. Il metodo *draw()* sfrutta il *repaint()* invocato costantemente dalle animazioni per aggiornare il valore di flashTitle per permettere il lampeggiamento del titolo del gioco, oltre alle ordinarie operazioni comuni a tutti i menù.

La classe ***GameMenu*** rappresenta il menù di gioco dal quale è possibile avviare effettivamente la partita. Presenta un costruttore simile a quello della classe *Menu*, che oltre alle procedure ordinarie, crea tre tipi di *AutoMovableObject*. Anche questa classe presenta i metodi *startAnimation()* e *stopAnimation()* identici in tutto e per tutto a quelli di Menu. Il metodo *draw()* oltre a disegnare gli oggetti auto movibili non presenta caratteristiche diversi rispetto a quello degli altri menù.

Infine, si ha la classe ***GamePanel*** che rappresenta il pannello di gioco vero e proprio. Il costruttore presenta operazioni standard con l'aggiunta dell'invocazione del metodo private *arrangeLevel()*. Tale metodo ha il compito di creare effettivamente il livello e disporre gli elementi al suo interno. Il *draw()* si comporta come tutti gli altri *AbstractMenu*.

Si vede ora la struttura dei ***listener***.

L'interfaccia ***Listener*** rappresenta l'ascoltatore mouse e tastiera dei *menuPanel*. Estende più classi di Java e implementa i loro metodi a vuoto di default, reso possibile grazie a Java 8. Si è deciso di implementare a vuoto tali metodi direttamente nell'interfaccia per una questione puramente estetica e di leggibilità del codice all'interno dei *listener* specifici delle classi, ascoltatori che non necessitano, infatti, di tutti i metodi che l'interfaccia mette a disposizione: ogni classe ridefinisce solo ed esclusivamente i metodi consoni al suo corretto funzionamento logico.

Gli ascoltatori si comportano quasi tutti allo stesso modo, proprio per questo se ne parlerà in generale e si entrerà nel dettaglio solo per metodi e/o comportamenti specifici di alcuni *listener*.
Nei costruttori di tutti i *listener* vengono creati gli oggetti Rectangle, i rettangoli che fungono da tasti, e viene associato il pannello a cui l'ascoltatore fa riferimento così da poter avere a disposizione le variabili protected. In generale tutti i *listener* implementano, almeno, il *mousePressed()*, il *mouseMoved()* e il *keyPressed()*. Il *mousePressed()* avvia l'audio specifico e si occupa di creare un oggetto *TransitionManager* per effettuare la transizione o di aggiungere un *miniPanel* in base al caso specifico del tasto che si sta premendo. Compito analogo lo ha chiaramente il metodo *keyPressed()*. Il metodo *mouseMoved()* avvia l'audio specifico e si occupa di cambiare in modo opportuno l'immagine relativa al puntatore del mouse. Leggermente diversi

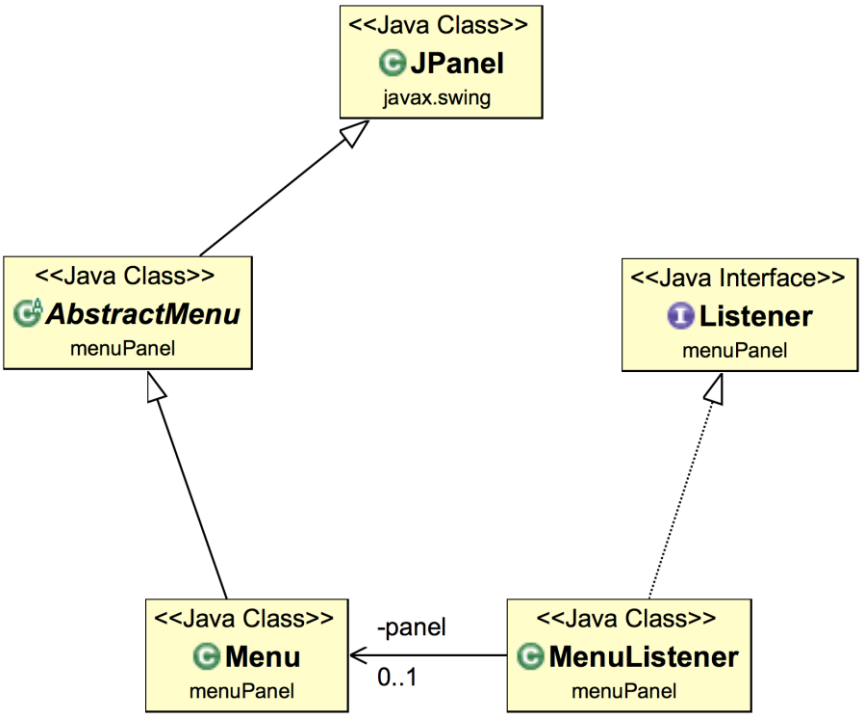
risultano il *mousePressed()* e il *keyPressed()* delle classi *ControlsMenuListener* e *LevelsMenuListener* che gestiscono anche il cambio di pagina con conseguente *repaint()* della finestra stessa.

Più complesso e, pertanto, degno di nota è il ***GamePanelListener***, ovvero l'ascoltatore della finestra di gioco vera e propria. Tale ascoltatore gestisce i movimenti della tartaruga all'interno delle tessere sulle quali ciò è chiaramente possibile. Si ricorda, infatti, che le tessere presentano dei rettangoli all'interno dei quali la tartaruga, anch'essa un rettangolo, ha la possibilità di muoversi.
Il *mousePressed()* sfrutta il metodo *releaseCell()* della classe *MapManager* per gestire la de-allocazione di una delle tre tessere di gioco cliccata. Gestisce, poi, i tasti in accordo con gli altri *listener*.
Il *mouseClicked()* si occupa di gestire le rotazioni delle tessere, invocando l'opportuno metodo della classe *Card*.
Il metodo *mouseReleased()* alloca la tessera nella posizione in cui è stata rilasciata, sfruttando il metodo *allocateCard()* in *MapManager*. Se l'allocazione è andata a buon fine il numero di mosse viene decrementato con l'animazione data dall'*AutoResizing*. Il metodo si occupa anche di verificare che le mosse non scendano sotto lo zero: in tal caso si transita nel menù di game over.
Il *mouseDragged()* invoca il metodo privato *generateBounds()* che genera i rettangoli relativi alle tessere e alla tartaruga. Dopodiché, se l'azione riguarda una tessera essa si muove semplicemente, altrimenti, se sta avvenendo sulla tartaruga viene invocato il metodo *possibleMovement()* che effettua numerosi controlli per gestire due tipi di movimenti della tartaruga:

- all'interno della singola tessera, sfruttando il metodo *verify1()*. Quest'ultimo verifica che nell'ipotesi in cui la tartaruga si muovesse nelle coordinate xy ricevute come parametro il movimento della tartaruga sia possibile all'interno della tessera stessa e, cioè, che essa sia ancora contenuta nei rettangoli in cui tale movimento è possibile. Nel caso in cui ciò è verificato si invoca il metodo *locateTurtle()* che semplicemente alloca la tartaruga in tale posizione, altrimenti il movimento non viene effettuato;
- tra due tessere adiacenti, sfruttando il metodo *verify2()*. Quest'ultimo effettua controlli per verificare che il rettangolo della tartaruga stia intersecando i rettangoli relativi ai percorsi delle due tessere adiacenti. Se ciò è verificato, allora il movimento è possibile e viene effettuato, altrimenti esso non viene effettuato.

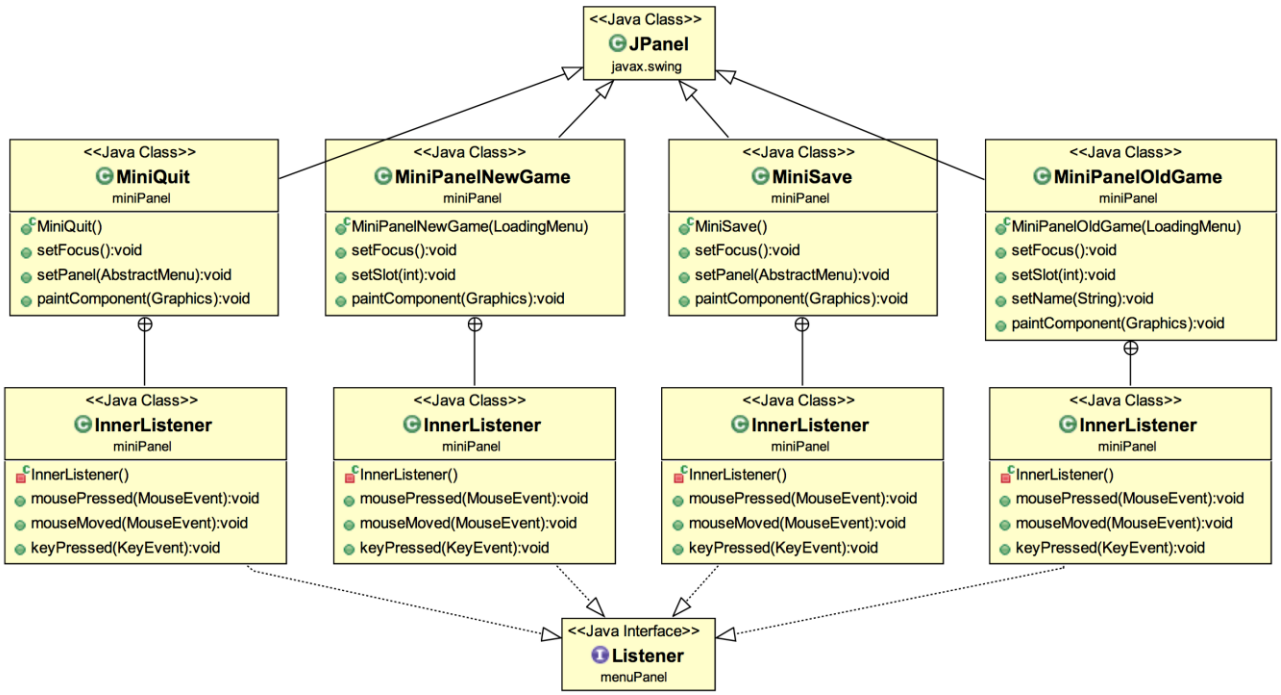
Il gioco è ideato per permettere il movimento della tartaruga utilizzando anche i tasti direzionali della tastiera. Si hanno, pertanto, il metodo *keyPressed()*, che gestisce il movimento, l'animazione e il suono della tartaruga nelle varie direzioni, e il metodo *keyReleased()*, che gestisce l'animazione della tartaruga che rientra nel guscio in base all'ultimo tasto direzionale premuto.
Si vede, in ultima analisi, il metodo privato *busyCard()* che ha il semplice compito di rendere occupata una o entrambe le tessere che riceve come parametro per evitarne il movimento nel momento in cui la tartaruga si trovi al loro interno, caso gestito non nel metodo in sé, ma da tutti quelli che lo invocano.

La struttura dei collegamenti che intercorre tra le classi rappresentanti i menù e i relativi ascoltatori è basata sul seguente esempio.



MINIPANEL

Il package ***miniPanel*** è così strutturato.



All'interno di questo package sono presenti pannelli più piccoli che vengono utilizzati nel gioco per funzioni specifiche. Hanno dimensioni ridotte rispetto ai pannelli relativi ai menù veri e propri e vengono agganciati e rimossi direttamente dal pannello che li invoca.

La classe ***MiniQuit*** rappresenta il pannello di conferma che appare quando si cerca di uscire dal gioco. In modo analogo a quanto avveniva nelle classi relative ai menù il costruttore imposta le caratteristiche standard del pannello oltre a creare il font che si utilizza al suo interno. Di particolare rilevanza è il metodo *setFocus()* che fa sì che il focus sia impostato sul *miniPanel* in questione. Il metodo *setPanel()* associa la variabile AbstractMenu che possiede come variabile di istanza con quella ricevuta come parametro, utile per sapere a quale menù fare riferimento quando il pannello viene rimosso, mentre il metodo *paintComponent()* si comporta esattamente come il metodo *draw()* delle classi di *menuPanel*.

La classe ***MiniSave*** rappresenta il pannello di conferma avvenuto salvataggio. Risulta essere analogo in tutto e per tutto al *miniQuit*, con gli appropriati cambiamenti.

La classe ***MiniPanelNewGame*** rappresenta il pannello che appare ogniqualvolta si crea un nuova partita. Esso permette di inserire il nome della partita stessa creandone una nuova. Presenta pertanto una JTextField in cui poter scrivere il nome della partita che si vuole iniziare a giocare. I metodi *setFocus()* e *paintComponent()* sono identici a quelli degli altri *miniPanel,* mentre presenta un metodo diverso che è il *setSlot()* che non fa altro che associare lo slot del metodo con quello passatogli come parametro: ciò è utile per conoscere quale dei tre slot di salvataggio si sta utilizzando, in modo tale sa sapere dove inserire il nome che verrà digitato all'interno del **miniPanel**.

La classe ***MiniPanelOldGame*** rappresenta il pannello che appare ogniqualvolta si sovrascrive una partita già esistente. Presenta, pertanto, una JLabel in cui appare il nome della partita che si sta sovrascrivendo e una JTextField in cui poter scrivere. È identica alla classe di cui sopra, presenta, in più, il metodo *setName()* che modifica il contenuto della JLabel con la stringa che riceve come parametro.

Le classi all'interno del package presentano tutte un *listener* come inner class simile in tutto e per tutto agli ascoltatori degli *AbstractMenu*, che si comporta, chiaramente, in modo opportuno.

Un'ultima nota va fatta per le risorse del gioco, quali immagini, audio e via dicendo. Esse si trovano nella cartella ***res*** all'interno del progetto stesso, suddivise in package appropriati in base al tipo di risorsa che rappresentano.

Stefano Gualtieri, matricola 169148

Alessandra Gagliardi, matricola 169097

Lorenzo Morelli, matricola 169153

Cuconato Francesco, matricola 169095