Relazione progetto

“Programmazione ad Oggetti”

Beatrice Ricci

Nicola Santolini

31/05/2015

Capitolo 1

Analisi

1.1 Requisiti

Lo scopo di questo progetto è quello di realizzare un clone del popolare videogioco-rompicapo Candy Crush Saga. L'idea di base è molto semplice: dato un campo di gioco con delle caramelle di vari colori bisogna scambiarle di posizione in modo da formare combinazioni di tre, quattro o cinque caramelle dello stesso colore. Dopo essere state combinate le caramelle sparisco e lasciano scendere quelle al di sopra, oltre a nuove caramelle generate per riempire gli spazi vuoti che si sono andati a creare. Ciò può portare, a catena, alla formazione di altre combinazioni, che porteranno altre caramelle a spostarsi ed altre ancora ad essere generate, e così via, con la possibilità di avere più sequenze di combinazioni in cascata prima che il gioco si trovi di nuovo in una fase di "quiete" e permetta al giocatore di fare una nuova mossa. Attraverso la formazione di particolari combinazioni, come quella da quattro, da cinque o di due tris incrociati, si posso generare caramelle particolari, con utili bonus da sfruttare nelle mosse successive. Ogni volta che una caramella viene eliminata e che vengono realizzate particolari combinazioni il punteggio viene incrementato. Nella versione originale vengono poi inserite alcune varianti, per ridurre la ripetitività del gioco (viste anche le centinaia di livelli disponibili), come per esempio ostacoli presenti sul campo di gioco o differenti modalità di vittoria del livello oltre al semplice raggiungimento di un certo punteggio.

Ciò che ci proponiamo con il nostro progetto è di realizzare un'applicazione che dopo una breve interfaccia iniziale permetta di accedere a un livello del gioco. Il livello in questione sarà un livello "base", con un campo di gioco composto da 81 caramelle senza particolari ostacoli/varianti, e sarà possibile ottenere la vittoria semplicemente raggiungendo un certo punteggio con le mosse a disposizione. Questo punteggio varierà in base alla difficoltà scelta prima di iniziare a giocare, e potrà poi essere cambiata qualora

si intenda rigiocare. Durante il gioco si potranno combinare le caramelle solo formando combinazioni di almeno 3 elementi dello stesso colore, e si potranno realizzare tutte le combinazioni presenti anche nel gioco originale; non sarà invece possibile effettuare ulteriori combinazioni particolari abbinando caramelle speciali (ad esempio non si potranno abbinare due caramelle ottenute con combinazioni di quattro elementi dello stesso colore), cosa invece prevista dal gioco originale.

Ulteriore funzionalità prevista è la già citata interfaccia iniziale in cui sarà presentato un menu da cui si potrà accedere al gioco, consultare le istruzioni oppure chiudere l'applicazione.

Per comodità di espressione in seguito ci riferiremo a:

- *"tris"* come semplice combinazione di tre caramelle dello stesso colore
- "*caramella wrapped"* come caramella ottenuta dalla combinazione di due tris incrociati. Questa caramella ha un bonus: quando viene attivata con una combinazione elimina tutte e otto le caramelle a lei adiacenti.
- "*caramella striped"* come caramella ottenuta da una combinazione di quattro caramelle dello stesso colore. Quando viene attivata con una combinazione elimina tutte le caramelle sulla sua stessa riga o colonna, a seconda se si tratta di una striped orizzontale o verticale (ciò dipende da come erano orientate le quattro caramelle che l'hanno generata).
- *"caramella special"* come caramella ottenuta da una combinazione di cinque caramelle dello stesso colore. Quando viene accoppiata con una qualsiasi caramella, elimina tutte le altre caramelle del suo stesso colore presenti sul campo.

1.2 Problema

Il nostro progetto presenta alcuni elementi di difficoltà. Per prima cosa, bisogna gestire il campo di gioco con le sue caramelle, incluse tutte le loro possibili varianti. La parte di modello dovrà occuparsi di gestire lo spostamento delle caramelle, evitando errori, imprecisioni o variazioni impreviste dei dati (ad esempio un errore nel colore o nel tipo di una caramella) che inficerebbero l'esperienza di gioco. Inoltre non sarà sufficiente che il modello gestisca semplicemente gli scambi tra caramelle, poiché il gioco prevede vincoli che non concedono di eseguire qualsiasi mossa liberamente. Infatti è possibile scambiare la posizione di due caramelle solo se sono vicine, e solo in senso verticale e/o orizzontale, non essendo previste mosse in diagonale. Inoltre per poter eseguire uno scambio bisogna che questo vada a generare almeno una combinazione di tre caramelle dello stesso colore, altrimenti la mossa non avrà nessun effetto e le due caramelle che si intendeva scambiare resteranno nella loro posizione. Trattandosi poi di un videogioco, o comunque di un suo clone, subentrano varie problematiche di coordinamento tra le varie parti dell'applicazione, di gestione di sincronismi e di presentazione all'utente, anche a livello estetico e, per quanto possibile, di "animazioni". In particolare, sia in fase di proposta valutando il volume di cose da realizzare rispetto al monte ore, sia poi durante la lavorazione effettiva del progetto, abbiamo capito le difficoltà nel gestire elementi grafici complessi ed animazioni. Abbiamo comunque cercato di gestire questi aspetti in modo coerente col monte ore e con quanto dichiarato in fase di proposta, tenendo presente che difronte a scelte condizionate dal tempo rimanente abbiamo preferito migliorare e dedicare maggiore attenzione ad altri aspetti dell'applicazione che consideravamo di maggiore rilevanza.

Capitolo 2

Design

2.1 Architettura

Per realizzare il nostro progetto abbiamo deciso di utilizzare il pattern MVC, organizzandolo come mostrato nel UML in figura 1.

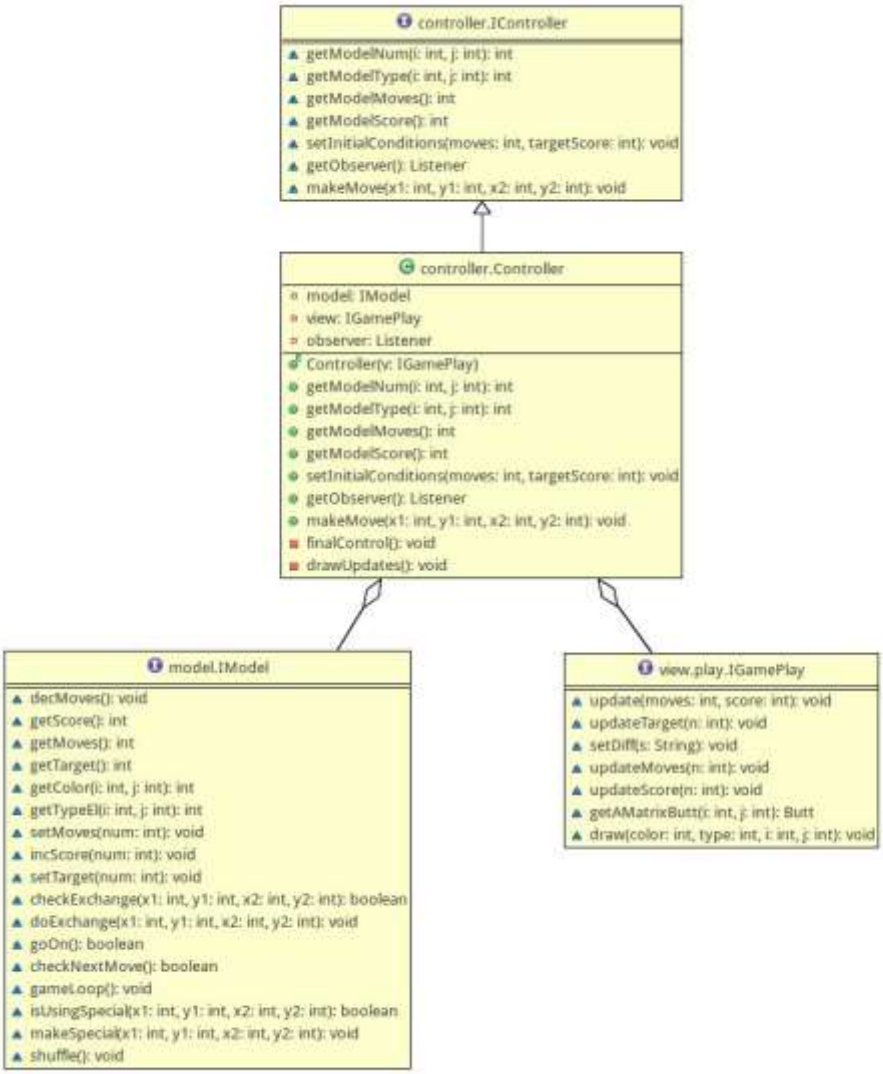

Figura 1 - implementazione MVC

Nel nostro caso la parte di model gestisce il campo di gioco e implementa tutti gli algoritmi necessari alla gestione di una partita, la parte di view invece si occupa di gestire i vari menu e finestre che possono essere utilizzati dall'applicazione e della presentazione dei dati all'utente durante la partita; queste due parti sono coordinate dal controller.
Il controller, una volta istanziato, è collegato alla parte di modello tramite l'interfaccia **IModel** e alla parte di view per mezzo dell'interfaccia **IGamePlay**.
In particolare durante una partita la view riceve gli input dell'utente, li fornisce al controller che a sua volta li comunica al model che si occupa di valutarli e di applicare gli algoritmi necessari a rappresentare il comportamento del gioco in risposta a tali input (ovvero come il gioco si sviluppa conseguentemente a una mossa, sia che essa venga poi completata o meno). Quando il model ha terminato le sue operazioni interviene nuovamente il controller che prende i nuovi dati dal model e li fornisce alla view così che questa possa presentare all'utente lo stato aggiornato del gioco.
Per quanto riguarda una eventuale sostituzione della view con un'altra versione, riteniamo che tale cambiamento non produrrebbe risultati disastrosi sul resto dell'applicazione. Le interazioni tra controller e view infatti sono limitate. La view provvede alla notifica al controller di semplici dati relativi a coordinate sul campo di gioco. Nel senso opposto il controller fornisce alla view i dati di gioco aggiornati di volta in volta, e come questi vengano poi gestiti e "disegnati" a schermo non impatta il controller stesso in maniera rilevante.

2.2 Design di dettaglio

Dopo alcune valutazioni, abbiamo deciso di far partire l'applicazione lanciando il menu principale. La successiva navigazione nelle finestre disponibili viene poi gestita all'interno della parte di view dell'applicazione; quando poi si decide di iniziare a giocare, e scelto il livello di difficoltà, viene istanziato il controller, agganciato alla view tramite **IGamePlay**, e in comunicazione con il modello per mezzo di **IModel**.

La parte di modello è stata suddivisa in tre parti principali, che possono essere identificate con le interfacce **IModel**, **IBoard** e **IGame.**

IModel è l'unica parte del modello in collegamento diretto col controller, che si rivolge a questa interfaccia per le sue richieste di informazioni alle altre parti del modello, e contiene quindi metodi necessari a queste comunicazioni. Nella classe **Model**, che implementa l'interfaccia **IModel,** vengono inoltre gestiti alcune informazioni importanti per il gioco, ad esempio mosse rimanenti e punti accumulati.

IBoard è l'entità che modella il campo di gioco. Concretamente questo concetto è rappresentato dalla matrice di gioco, contenuta nella classe **Board.** Tale matrice è composta di oggetti di tipo **Candy**, rappresentati dalla propria interfaccia, **ICandy.**

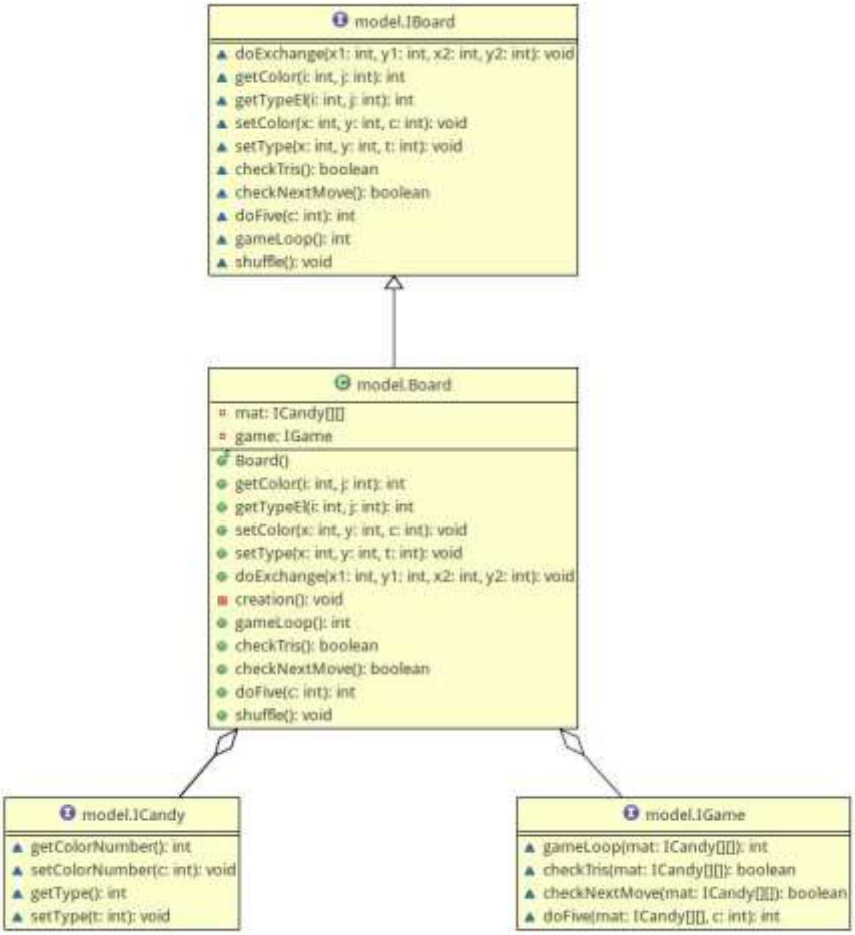

Figura 2 – Diagramma della classe Board, che rappresenta il campo di gioco

Tali elementi, "le caramelle", sono caratterizzate da un colore (rosso, arancio, viola, ...) e da un tipo (normale, striped, wrapped, special). **IBoard** espone i metodi necessari a interagire con la matrice, ad esempio quelli necessari a raccogliere le informazioni su una caramella in una certa posizione, per poterla poi rappresentare.

Per quanto riguarda l'interfaccia **IGame** (e la classe **Game** che la implementa), questa entità che inizialmente non era prevista, è stata aggiunta successivamente per suddividere la classe **Board**, che aveva a nostro parere raggiunto dimensioni proibitive. In particolare la classe Game si compone di alcuni campi, che vengono utilizzati per svolgere alcuni compiti. Si utilizza ad esempio la classe **Checks** (modellata dall'intefaccia **IChecks**) contiene tutte le funzionalità di controllo sulle combinazioni all'interno della matrice, e quindi i metodi per verificarne la presenza sul campo di gioco. Sempre nella classe **Checks**, viene implementata anche la funzionalità, molto importante, che verifica se sul campo di gioco è presente almeno una combinazione, di qualsiasi tipo e colore; questa condizione viene verificata frequentemente, visto che finché rimane vera, il loop di gioco deve continuare.
La classe **Game** si compone anche di altri quattro campi, delle classi **TrisBehaviour**, **PokerBehaviour**, **FiveBehaviour** e **WrappedBehaviour**. Queste classi contengono i comportamenti da applicare qualora si generi un tris o si creino caramelle striped, wrapped e special. Il codice comune a tutte e quattro le classi è stato racchiuso nella classe astratta **AbstractBehaviourController**, ad esempio i metodi *descend()* e *resolve()*. Questi metodi sono fondamentali per lo svolgimento della partita; in particolare il metodo *descend()* è molto importante perché dopo aver formato una combinazione ed eliminato le caramelle interessate, interviene e simula la "forza di gravità" facendo scendere quelle al di sopra per occupare lo spazio lasciato vuoto, creando così un "buco" nella parte alta del campo, visto che è da lì le nuove caramelle che vengono generate entrano in gioco. Il metodo *resolve(),* una volta che le aree vuote sono affiorate nella parte alta del campo e che tutte le caramelle sono scese nella nuova posizione, provvede a inserire le nuove caramelle dove necessario.

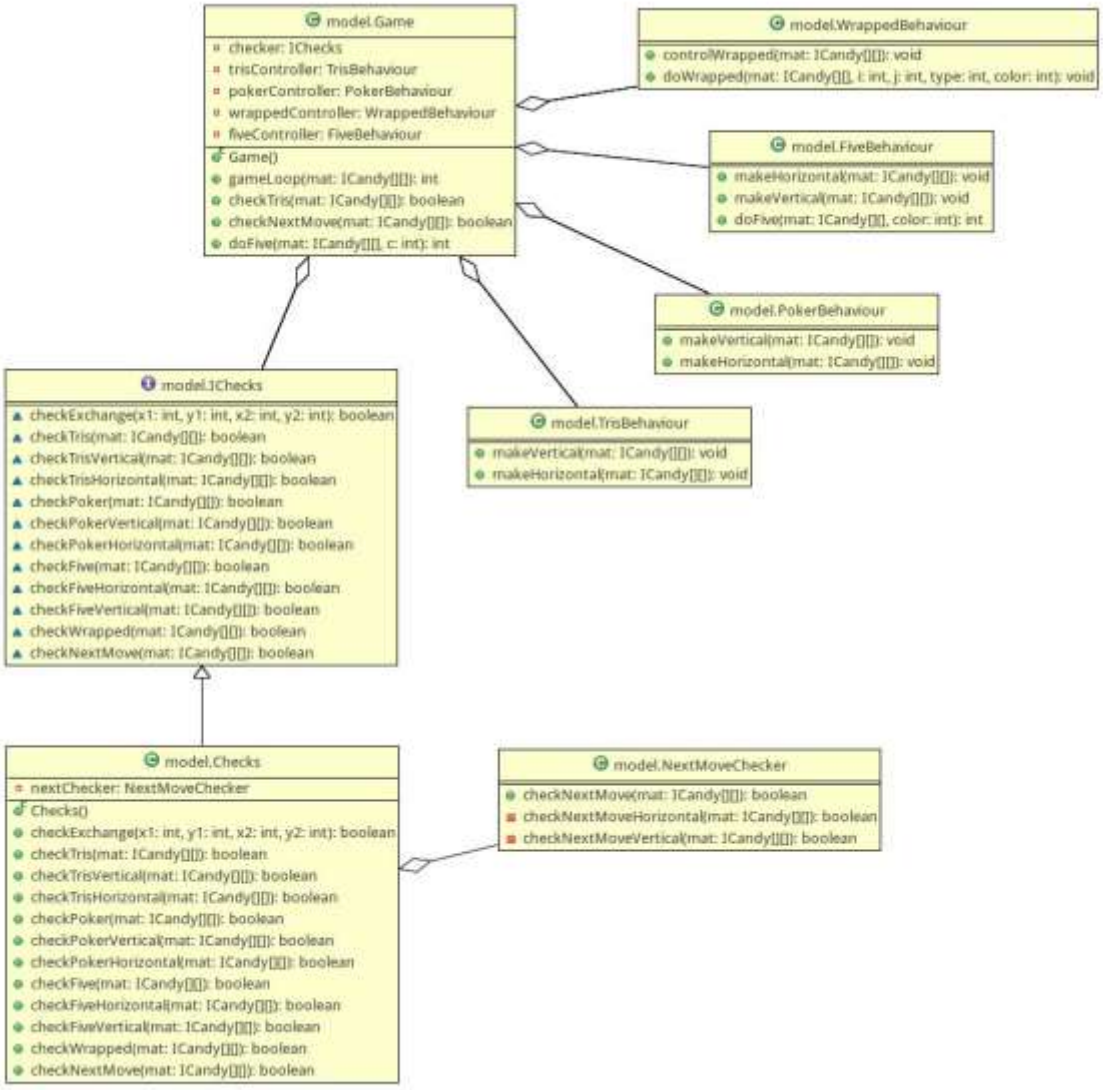

Figura 3 – Diagramma della classe Game e di quelle di cui si compone

Altri metodi comuni però non riguardavano la classe **WrappedBehaviour**, essendo questo tipo di combinazione sostanzialmente diversa dalle altre perché "non lineare", visto che è composta da due tris perpendicolari. Per ovviare a questo problema si è realizzata un'altra classe astratta, **AbstractLinearBehaviour** che estende da **AbstractBehaviourController** e contiene i metodi *makeHorizontal*() e *makeVertical().* Come evidenziato dal diagramma UML (figura 4) le classi **TrisBehaviour**, **PokerBehaviour**, **FiveBehaviour**, che necessitano dei suddetti metodi poiché realizzano combinazioni di caramelle allineate, estendono dalla classe **AbstractLinearBehaviour**,

mentre la classe **WrappedBehaviour**, che non utilizza quei metodi, estende solo da **AbstractBehaviourController**. Nella realizzazione dei due metodi *makeVertical()* e *makeHorizontal()* si è poi utilizzato il pattern Template Method: questi metodi, anche se svolgono funzioni logicamente molto simili, hanno bisogno di implementazioni diverse a seconda di quale combinazione si deve andare a realizzare (tre, quattro o cinque elementi). Tali implementazioni sono infatti delegate alle sottoclassi. E' poi il metodo che si occupa di gestire il loop di gioco che, dopo aver verificato con i metodi di check quale combinazione deve considerare, si rivolge alla classe corretta richiedendo *makeVertical()* o *makeHorizontal().*

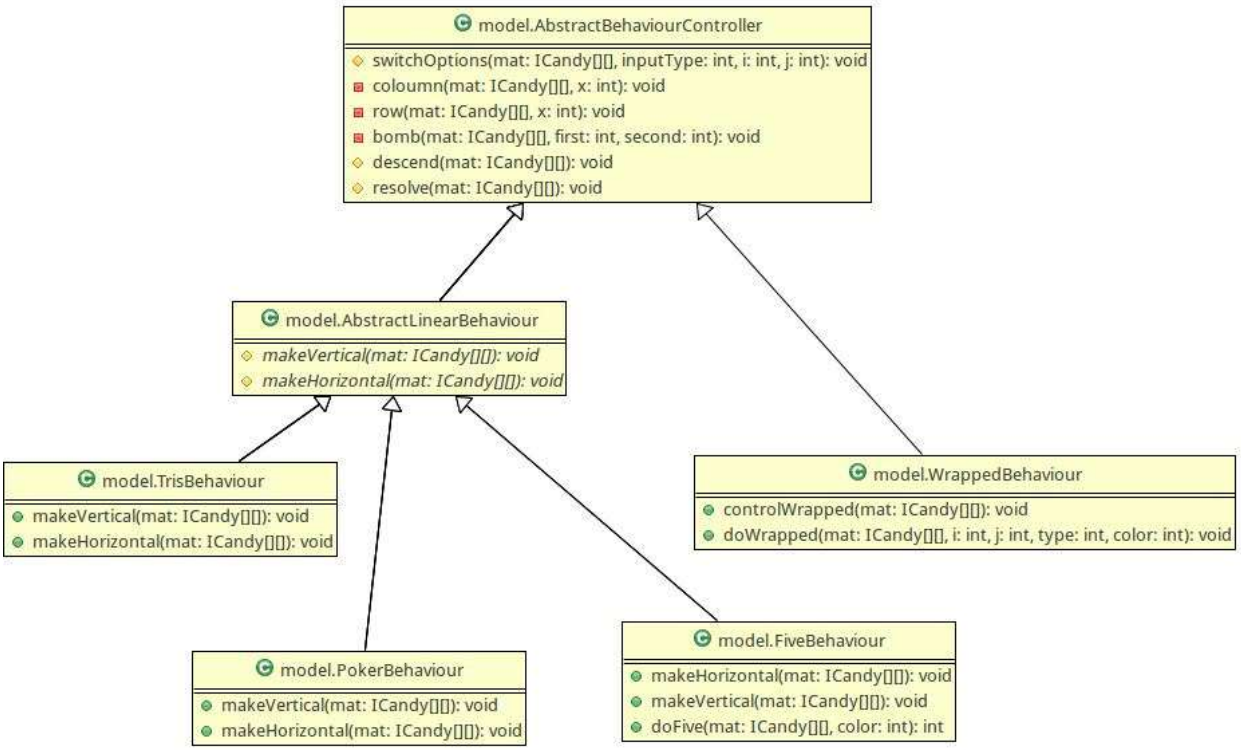

Figura 4 – Gerarchia delle classi astratte per la gestione dei behaviour

Nella classe **Game** si è optato per questa composizione di classi proprio per permettere questo tipo di realizzazione, soprattutto del metodo *gameLoop()*, che finché sul campo di gioco è presente almeno una combinazione, continua a individuare di quale tipo si tratta utilizzando i metodi di check e invocare il behaviour corrispondente. Essendo

questo sistema un po' macchinoso, e coinvolgendo diverse classi/metodi, si è poi pensato, traendo anche spunto dal pattern facade, di "nascondere" questo tipo di implementazione dalla classe **Board** e di mostrarle solo il metodo *gameLoop()*, per rendere così più semplice il suo utilizzo, e delegando queste funzionalità a un livello sottostante.

Un'altra funzionalità che si vuole esporre è quella legata alla condizione imposta dal gioco secondo cui, per eseguire una mossa questa deve formare almeno una combinazione. Questo comporta che, se con le caramelle sul campo di gioco non fosse possibile eseguire una combinazione in una sola mossa, il gioco non potrebbe proseguire e ci si troverebbe in una fase di stallo.
Per gestire queste situazioni e garantire il rispetto di questa condizione per prima cosa l'applicazione deve essere in grado di valutare la matrice di gioco e di capire quando si entra in una, per così dire, fase di stallo. A questo scopo, dopo un breve studio teorico sulle diverse posizioni e combinazioni che con uno scambio generano un tris, abbiamo iniziato, in particolare Nicola, a progettare un algoritmo che le individuasse tutte, ma che al tempo stesso non segnalasse dei falsi positivi. Si sono iniziate delle prove di implementazione di tale algoritmo, prima su matrici di dimensioni limitate ed elementi semplici, per poi adattarlo alle dimensioni del nostro campo di gioco e agli elementi che dovevamo modellare. Dopo diversi tentativi, si è arrivati alle versione finale, contenuta nei metodi *checkNextMoveVertical()* e *checkNextMoveHorizontal()* della classe **NextMoveChecker**. L'algoritmo esegue dei semplici controlli sui colori delle caramelle, ed alcuni di essi sono volutamente ridondanti. Durante la fase di costruzione dell'algoritmo si è infatti visto che anche applicando correttamente i controlli per individuare tutte le possibili combinazioni interessate, percorrendo la matrice una sola volta era impossibile evidenziare tutte le situazioni che dovevamo individuare, se non complicando notevolmente l'algoritmo creando numerosi casi particolari per controllare i bordi, gli angoli e la parte centrale della matrice in maniere anche molto diverse per evitare errori nel range degli indici di scorrimento o omissione di qualche combinazione utile. Si è quindi optato per eseguire i controlli partendo da tutti e quattro gli angoli della

matrice. Come detto questo sistema rende alcuni controlli ridondanti poiché tra le varie visite della matrice all'interno dello stesso metodo si creano zone di sovrapposizione, che vengono controllate più volte, e alcune combinazioni possono essere individuate partendo sia da un angolo che da un altro. Si è comunque deciso di adottare questa soluzione poiché garantisce di individuare tutte le possibili conformazioni della matrice di interesse, tenendo conto che le performance rimangono buone visto che si tratta di semplici confronti su valori interi e che è altamente improbabile che tutti i controlli vengano eseguiti visto che al primo possibile tris individuato il metodo ritorna vero e conclude l'esecuzione, e quindi solo nel caso pessimo il pieno costo computazionale dell'algoritmo verrebbe sostenuto.

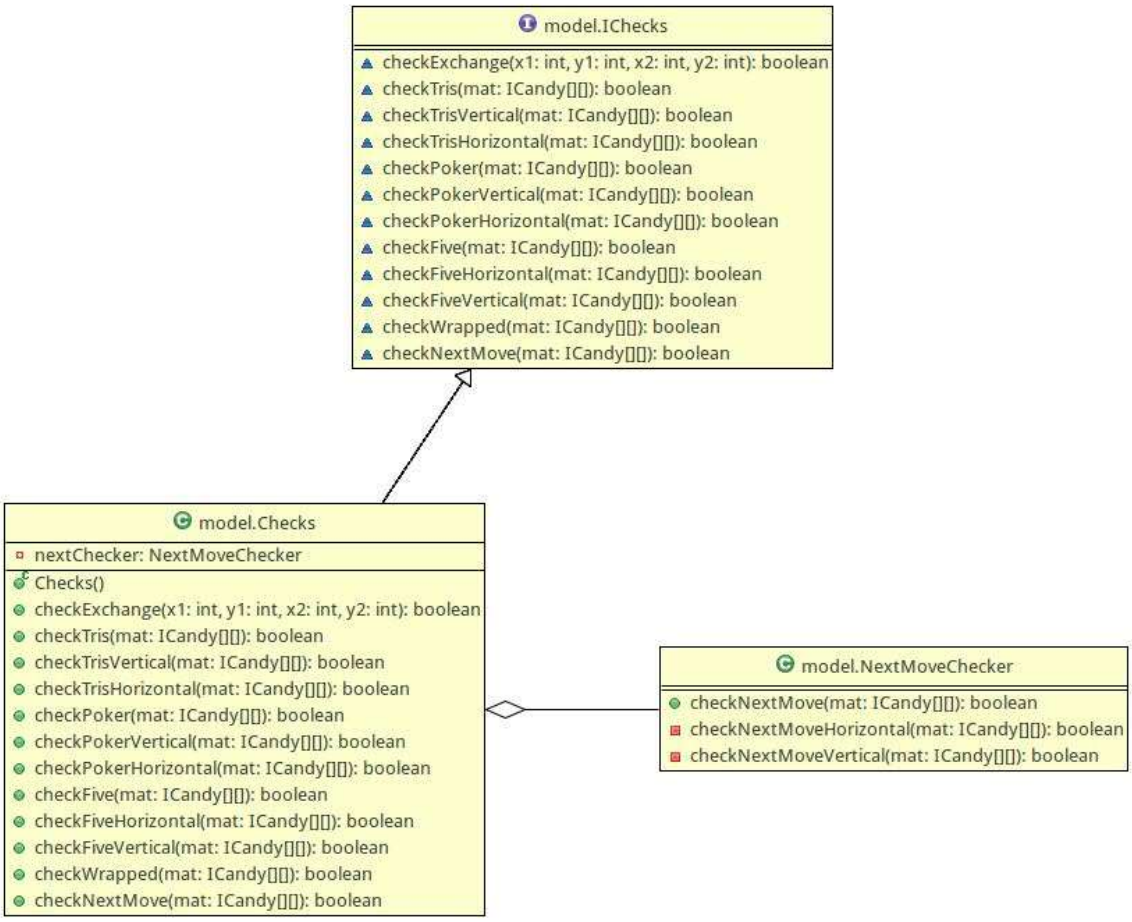

Figura 5 – Diagramma della classe Checks

Una volta garantito che le situazioni di stallo vengano individuate, bisogna anche implementare il comportamento necessario alla loro risoluzione. Nella versione originale il gioco, in queste situazioni, attiva un sistema con cui rimescolano le caramelle sul campo di gioco finché non ci si trova in una situazione in cui il gioco non è più in fase di stallo.
Anche noi abbiamo cercato di implementare una soluzione simile, nonostante sia da notare che nelle lunghe prove che abbiamo fatto del gioco tale situazione non ci è mai capitata nello svolgimento normale di una partita. Per altro si tratta di una condizione poco probabile anche statisticamente, in rapporto al numero di caramelle sul campo ed ai possibili colori che possono assumere. Abbiamo dovuto provvedere a dei test automatici per verificare che il sistema si accorgesse di questa situazione (anche se creata "artificialmente") e che effettivamente attivasse la risposta corretta, oltre al fatto che questa risposta fosse poi davvero funzionante e il "rimescolamento" avvenisse. Per completezza viene presentato il diagramma UML della classe **Checks,** che oltre all'algoritmo precedente contiene anche i metodi che si occupano di individuare tutte le combinazioni effettivamente in atto sul campo di gioco. Infine mostriamo un UML, anche se molto corposo, che rappresenta gran arte del modello, per dare anche una visione d'insieme.

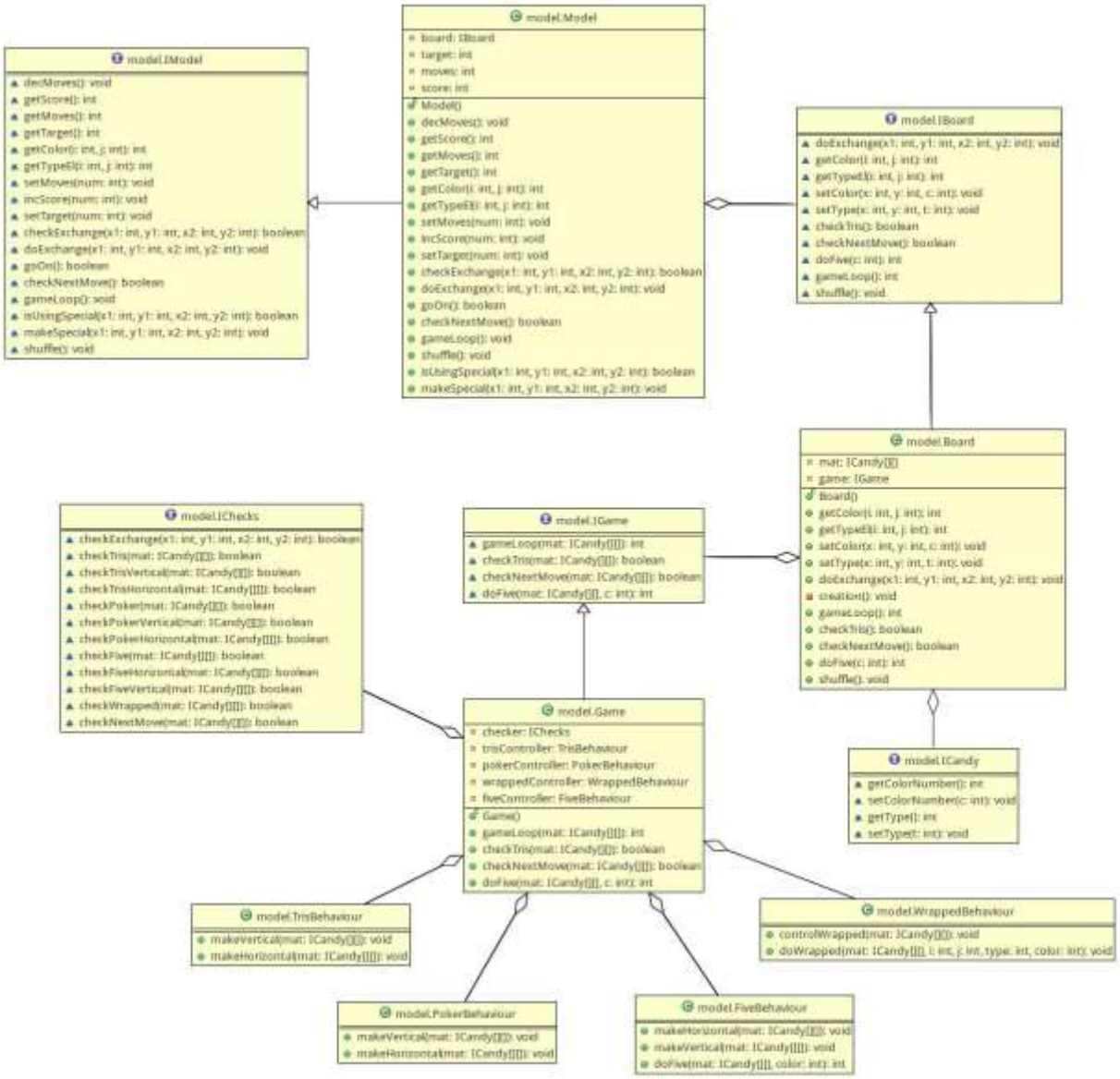

Figura 6 – Diagramma UML del model

La parte di view è composta sostanzialmente da:

- Due menù iniziali, uno principale e uno per selezionare la difficoltà di gioco
- La schermata principale di gioco, che mostra le caramelle in campo
- Una schermata finale che può comunicare la vittoria o la sconfitta

Inizialmente l'applicazione propone all'utente un menu a tre scelte: iniziare direttamente il gioco, leggere le istruzioni oppure chiudere l'applicazione.

Premendo il tasto "INSTRUCTION" l'utente sceglie di visualizzare le istruzioni, verrà quindi aperta una nuova schermata dove si possono leggere le regole del gioco, per poi tornare al menu principale. Premendo il tasto "START" invece, verrà proposto all'utente un secondo menu in cui è necessario scegliere un livello di difficoltà per poter procedere. La differenza principale fra i tre livelli di difficoltà è il diverso obiettivo posto per vincere. Abbiamo deciso di stabilire un unico numero di mosse per tutte le diverse difficoltà, ma sarebbe facilmente modificabile. Una volta scelto il livello di difficoltà, compare la schermata di gioco disegnata dall'interfaccia **IGamePlay** e implementata dalla classe **GamePlayView** e dalla classe **SetPlayPanels**, di cui la stessa **GamePlayView** si compone. Graficamente questa classe è composta da diversi elementi necessari all'utente per capire come si evolve la situazione di gioco. Questi sono il numero di mosse che l'utente ha ancora a disposizione, il punteggio da raggiungere e il totale dei punti accumulati grazie ogni mossa.

Vi è poi la matrice di gioco. A livello implementativo è stato scelto di considerare la matrice di caramelle con una matrice di **Butt**, cioè una classe ideata appositamente per le esigenze di gioco. **Butt** infatti estende la classe **JButton** (javax.swing.JButton) facendo in modo però che ad ogni **Butt** siano associati due numeri interi che andranno a rappresentare le sue coordinate sulla matrice degli elementi. Nella fase di creazione della matrice, all'interno del metodo *drawFirstMatrix(),* viene contestualmente legato ogni **Butt** ad un listener che sfrutta i due campi interi inseriti nell'oggetto per notificare le coordinate della caramella selezionata al Controller.

Per fare in modo che il Controller si interfacci con la view solamente tramite l'interfaccia **IGamePlay** è stato necessario creare all'interno della classe **GamePlay** il metodo *draw()* che a sua volta richiama l'omonimo metodo presente in **IUpdate** che si occupa di ridisegnare la matrice di gioco dopo ogni mossa in base ai dati forniti dal controller. In una prima fase era la view stessa, quando riceveva l'ordine di eseguire l'update, a richiedere, tramite il controller, i dati necessari per ridisegnare la schermata di gioco aggiornata in tutti i suoi elementi. Successivamente abbiamo deciso di attuare la soluzione attuale, in cui è invece il controller che, per ogni elemento della matrice di gioco, dà

ordine di disegnare una caramella con certe caratteristiche in una precisa posizione, poiché ci sembrava una modalità più corretta.

Nell'UML seguente (figura 7) è rappresentata la struttura che sorge attorno alla classe **GamePlayView**.

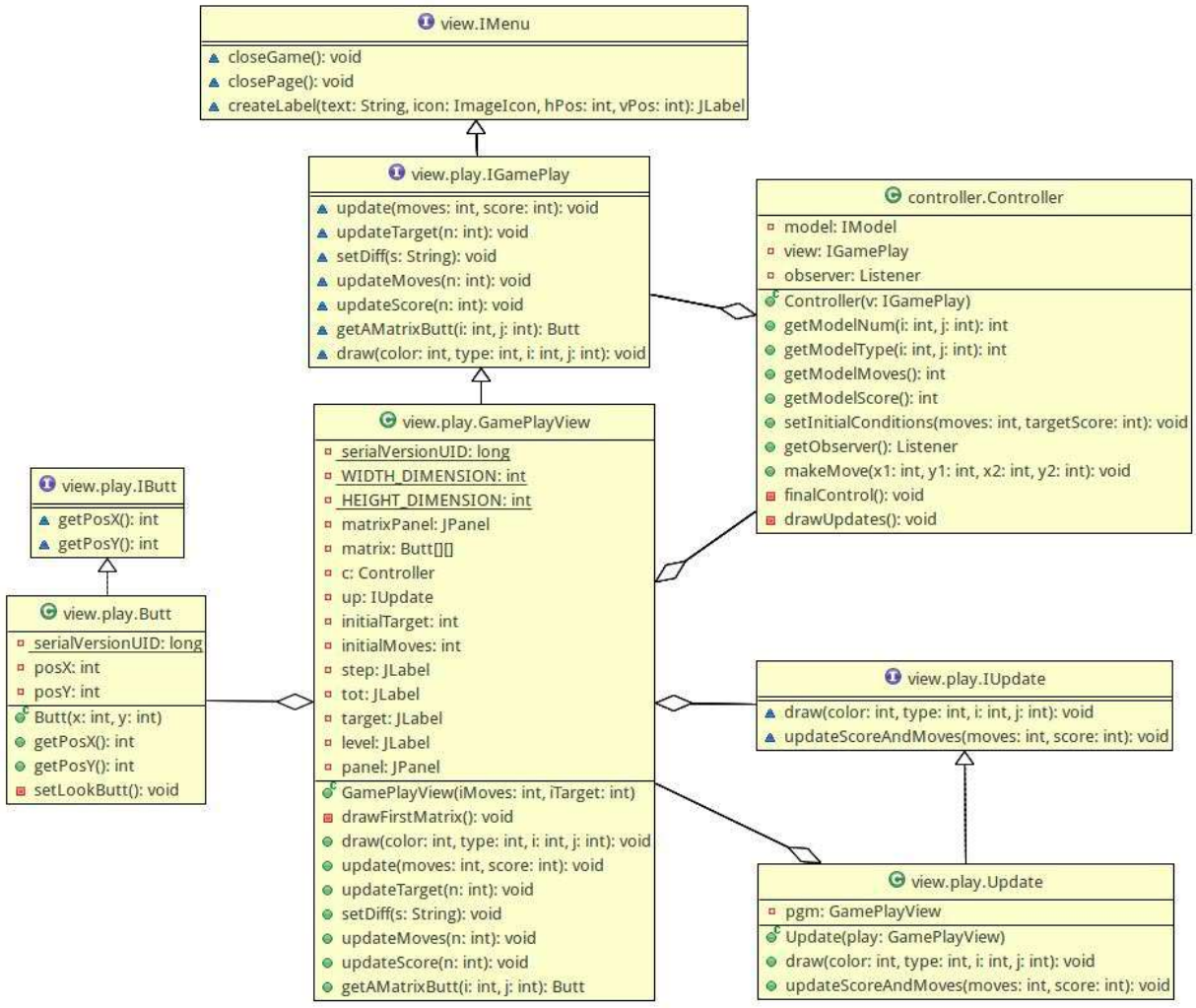

Figura 7 – UML della parte di view fondamentale per il gioco

Quando il controller stabilisce che la partita è terminata (tramite il metodo che esegue i controlli finali sul punteggio e le mosse rimanenti) verrà lanciata la classe **Win** o **GameOver** in base al risultato ottenuto, visualizzando così la schermata corrispondente. Entrambe le classi sopra citate estendono dalla classe astratta **AbstractFinal** che a sua volta implementa l'interfaccia **IFinal**. **AbstractFinal** contiene l'implementazione dei metodi

comuni delle classi figlie, che differenziano solo per il messaggio da comunicare all'utente e il colore di sfondo.

Per quanto riguarda i livelli di difficoltà, questi corrispondono alle tre classi **EasyLevel**, **MediumLevel** e **DifficultLevel**. Queste classi svolgono funzioni molto simili a livello logico, infatti estendono tutte dalla classe **AbstractLevel**, che contiene il metodo astratto *setLevel()*, che viene poi concretamente implementato dalle sottoclassi. Si è quindi applicato il pattern Template Method (come da figura 8) per ridurre le ripetizioni e ridondanze del codice.

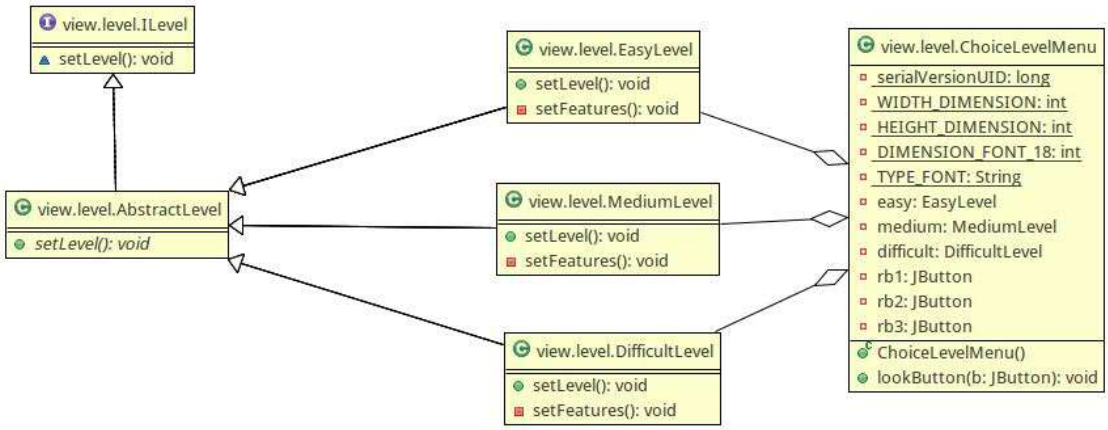

Figura 8 – schema UML dello sviluppo di ChoiseLevelMenu

Lo stesso pattern è stato utilizzato anche in gran parte delle classi della view per astrarre e organizzare in maniera più efficace comportamenti comuni a più componenti grafici, permettendo però agli stessi di adattarli poi a esigenze specifiche.

Si fa riferimento alle classi astratte **AbstractMenuButton**, **AbstractMenuPanel** e **AbstractMenuPanelAndButton**. Si tratta di classi che estendono tutte da **AbstractMenu**, e che poi contengono alcuni metodi astratti specifici, come si può notare anche dal relativo diagramma UML, in figura 9, relativi alle funzionalità che devono poi essere "personalizzati" dalle sottoclassi.

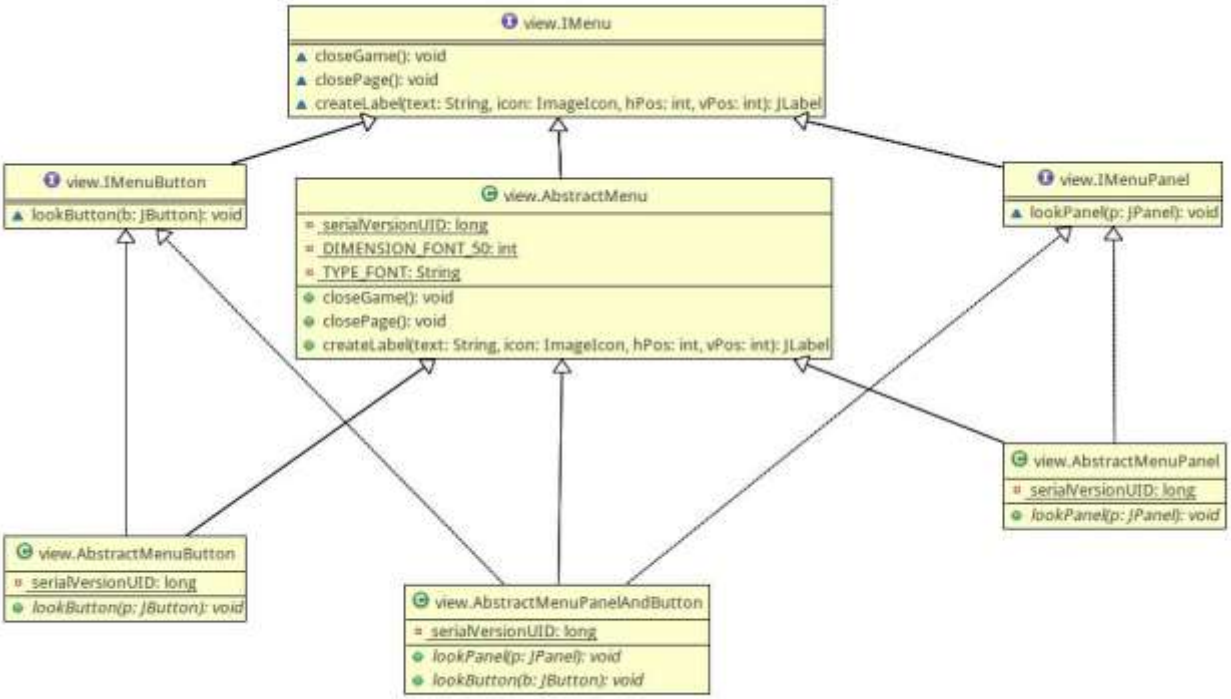

Figura 9 – schema dello sviluppo gerarchico delle classi "padri" delle classi principali

Questo ci ha permesso una maggiore flessibilità nella gestione dei componenti grafici, in particolare la gestione del *look and feel* di **JButton** e **JPanel.** Si è reso necessario utilizzare più classi astratte poiché non tutti i componenti necessitavano delle stesse possibilità di personalizzazione, e si è quindi costruita la gerarchia finale mostrata negli UML seguenti.

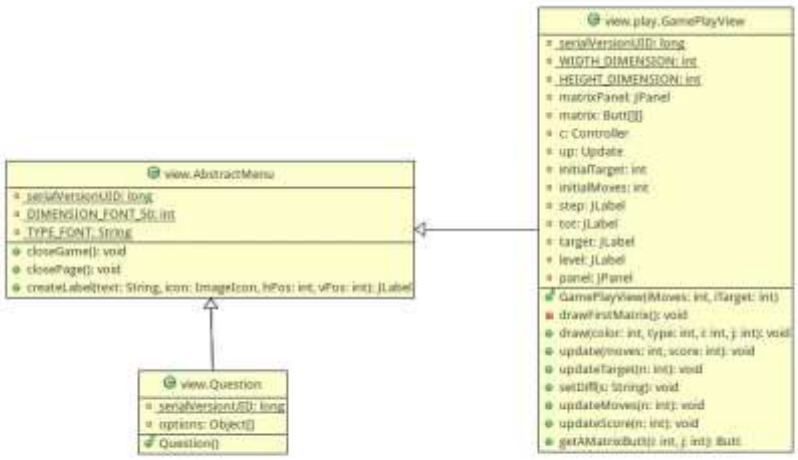

Figura 10 – AbstractMenu è classe padre di GamePlayView e Question

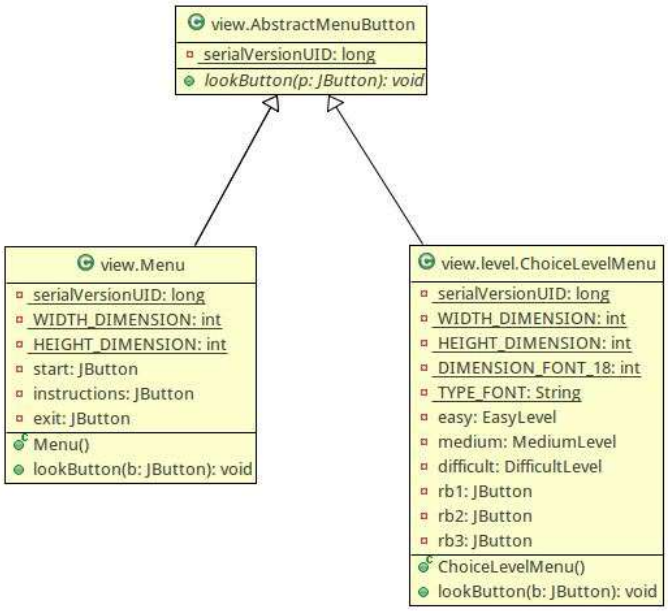

Figura 11 – AbstractMenuButton è classe padre di Menu e ChoiceLevelMenu

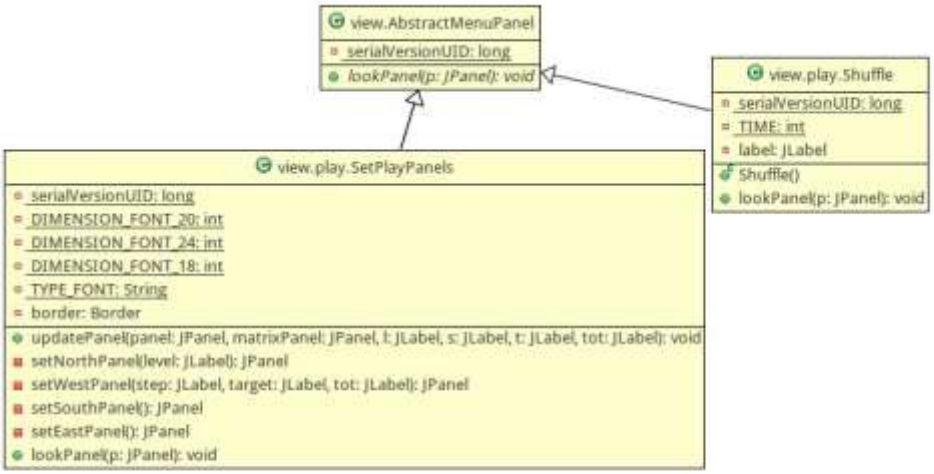

Figura 12 – AbstractMenuPanel è classe padre di SetPlayPanels e Shuffle

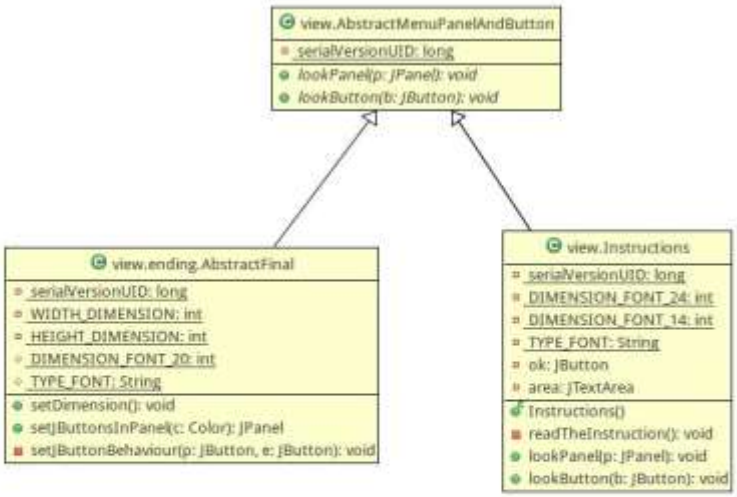

Figura 13 – AbstractMenuPanelAndButton è classe padre di AbstractFinal e Instructions

Un'altra funzionalità che si è gestita anche a livello grafico è quella legata alla procedura, precedentemente trattata nella parte di relazione riguardante il model, di mescolamento delle caramelle in campo quando necessario. Tale funzione di shuffle coinvolge anche la view poiché quando viene utilizzata dal controller, questo richiama anche una classe specifica che si occupa di mostrare a video per qualche secondo una schermata idonea per comunicare all'utente quanto sta avvenendo.

Mentre per le parti di modello e view abbiamo lavorato in maniera fortemente autonoma, per realizzare il controller (figura 14) abbiamo deciso di collaborare maggiormente.

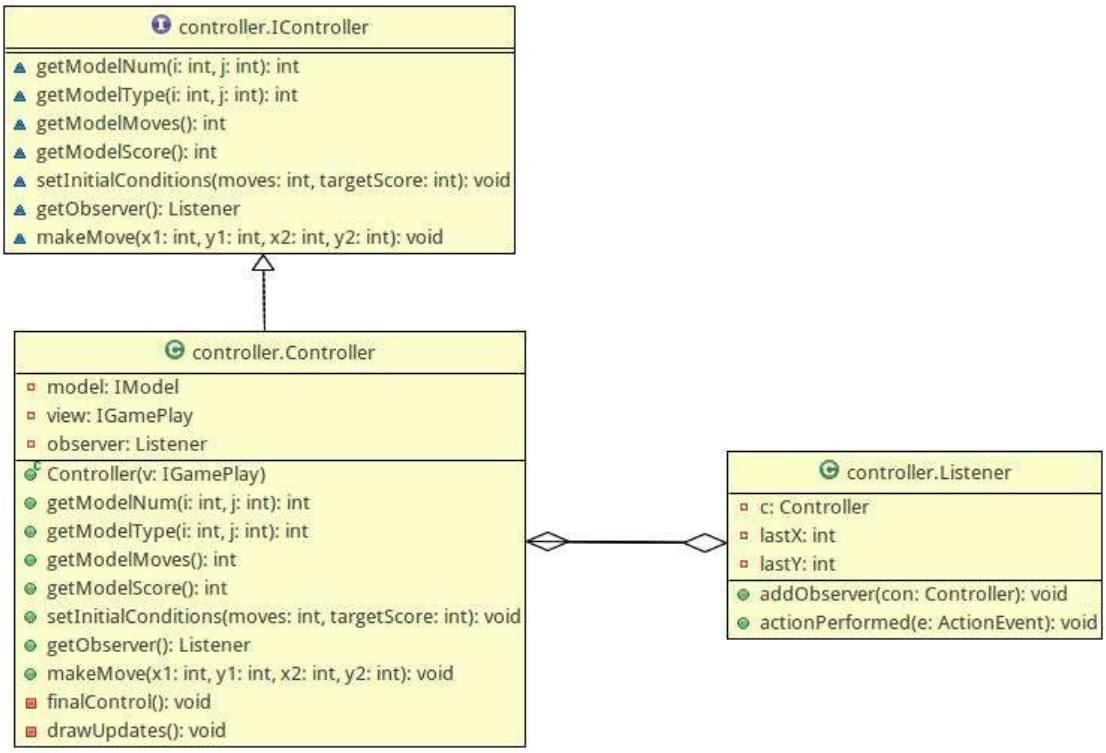

Figura 14 – UML del controller

Una volta istanziato, come già detto, tramite il menu principale, il controller crea il listener, che userà per osservare della view. Infatti la classe **GamePlayView**, che come già visto contiene la matrice di **Butt**, ovvero la rappresentazione a livello grafico del campo di gioco; a ogni elemento della matrice viene agganciato un listener, lo stesso a cui è legato anche il controller. Quando una caramella viene selezionata, se un'altra a lei adiacente lo è già stata, le due relative coordinate vengono notificate al controller, mentre se la caramella selezionata precedentemente non è vicina alla seconda, la prima viene deselezionata e sostituita. Il controller riceve quindi coppie di coordinate da processare, ovvero le posizioni sul campo di due caramelle. Per prima cosa verifica se una delle due è una caramella special, il cui behaviour deve essere attivato indipendentemente dal fatto di essere coinvolta in una combinazione o meno. Se nessuna delle due caramelle è special, il controller dà ordine al modello di invertirne la posizione, e fornisce i dati

aggiornati alla view per rappresentare a schermo il cambiamento avvenuto. Procede poi a verificare, tramite il modello, se lo spostamento ha generato una combinazione. In caso negativo ordina di rieseguire lo scambio sia al modello che alla view, fornendo successivamente dei nuovi dati aggiornati. In questo modo, in caso l'utente esegua uno scambio non permesso, cioè che non genera combinazioni, a livello grafico ha comunque la possibilità di vedere le caramelle che prima si scambiano e poi ritornano alla loro posizione senza alcun effetto, e di capire che la mossa non è corretta. Abbiamo ben presto notato che a velocità normale tutto ciò avveniva troppo rapidamente, e che un utente comune non avrebbe potuto capire cosa fosse successo. Abbiamo quindi sentito la necessità di "rallentare" la risposta grafica a questa situazione. Dopo alcuni tentativi e ragionamenti, viste le tempistiche e le nostre possibilità in quanto a conoscenze grafiche e di animazione, abbiamo optato per una soluzione che fosse per lo meno efficace nel risultato. Sfruttando quanto visto anche durante il corso, abbiamo utilizzato il metodo *invokeLater()* di **SwingUtilities** insieme ad un breve sleep del thread per permettere all'utente di visualizzare per qualche istante lo spostamento e capire cosa avviene. Consci che probabilmente non si tratta di una soluzione ottimale, e che avremmo potuto fare meglio per quanto riguarda la parte di animazione e thread-safety, abbiamo preferito tenerci tempo, che poi abbiamo effettivamente sfruttato, per successive opere di miglioramento e ottimizzazione.

Se invece dopo aver effettuato il primo scambio il controller riceve conferma dal model che è stata realizzata una combinazione, inizia una catena di operazioni che coinvolge sia la parte di modello che la view, con il controller che le coordina. Infatti finché il model rileva combinazioni, che posso generarsi in cascata a causa di combinazioni precedenti, questo continua ad elaborarle. Ad ogni combinazione realizzata il controller manda dati aggiornati alla view affinché li possa rappresentare a video. Per implementare tutto ciò abbiamo realizzato un sistema in cui il controller interroga il model chiedendogli se è presente almeno una combinazione (indipendentemente da quale sia), e finché riceve una risposta positiva ordina al model stesso di individuarla e gestirla nella maniera corrispondente. Questo procedimento può concludersi con una sola combinazione,

oppure con più cicli di operazioni qualora si formino combinazioni a catena o si utilizzino behaviour di caramelle particolari. Una volta terminato il loop il controller provvede a verificare che sia comunque possibile realizzare una combinazioni con la nuova disposizione del campo di gioco, in caso contrario deve provvedere a notificare model e view affinché avviino le rispettive procedure per gestire le operazioni di shuffle di cui abbiamo parlato. Quando necessario il controller applica anche la procedura che verifica se le mosse a disposizione sono terminate e che compara il punteggio raggiunto dopo ogni mossa con quello che deve essere raggiunto (in base alla difficoltà) per controllare se la partita deve terminare e con quale esito.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per assicurarci che la parte di modello funzionasse correttamente e non generasse errori imprevisti abbiamo deciso di eseguire, oltre ai numerosi test manuali provando l'applicazione, anche dei test automatizzati utilizzando gli strumenti, visti anche durante il corso, di **JUnit**. Abbiamo realizzato una classe appositamente per questo scopo. In particolare abbiamo testato:

- Il funzionamento dei metodi di "check", ovvero quelli che si occupano di ricercare combinazioni di caramelle sul campo di gioco. Nello specifico dopo aver generato delle matrici senza combinazioni ne abbiamo inserite "a mano" cambiando i “colori” di alcune caramelle in specifiche posizioni per poi accertaci che tali combinazioni venissero individuate.
- La correttezza dei metodi di "check"; in particolare abbiamo verificato, lanciando tali metodi svariate volte, che non generassero mai errori relativi al range di lavoro degli indici, per assicurarci che fossero stati costruiti correttamente.
- Analogamente ai punti precedenti, anche per il sistema che verifica che sia sempre possibile una mossa successiva, abbiamo verificato sia che il metodo riconosca effettivamente l'eventuale assenza di questa condizione, sia che non presenti errori di costruzione per quanto riguarda gli indici.

Si sono comunque realizzati ulteriori test manuali per verificare a fondo il funzionamento dell'intera applicazione.

3.2 Metodologia di lavoro e divisione dei compiti

Nella fase iniziale del progetto, prima di formalizzare la proposta, ci siamo ritrovati per alcune ore per realizzare alcune prove pratiche su alcuni aspetti, in particolare di Swing, per verificare la praticabilità concreta di alcune cose che intendevamo realizzare. A questo proposito, dopo aver valutato la possibilità di utilizzare framework più complessi per implementare la parte grafica, abbiamo deciso di appoggiarci interamente a Swing e cercare di ottenere il massimo delle possibilità offerte da questa libreria, avvalendoci talvolta ad Awt. Fatto ciò, abbiamo deciso di formalizzare la proposta e ci siamo quindi di nuovo riuniti per iniziare a organizzare il lavoro. Fin da subito abbiamo deciso di dare una separazione il più netta e logica possibile alle parti di lavoro da svolgere, soprattutto per evitare di perdere successivamente molto tempo per integrare quanto avremmo realizzato; data per acquisita la necessità di utilizzare il pattern MVC, abbiamo quindi optato per affidare a Nicola la parte di model, a Beatrice la parte di view, per poi collaborare nella realizzazione del controller, in proporzioni da decidere poi in corso d'opera anche in base a eventuali problemi o squilibri sorti durante lo sviluppo delle rispettive parti.

Abbiamo quindi proceduto sviluppando le nostre parti di progetto, cercando di arrivare il prima possibile a versioni abbastanza funzionanti da poter provare una prima integrazione delle due parti. Una volta che entrambi abbiamo raggiunto questo obiettivo abbiamo provato ad integrare il tutto con una prima versione del controller, realizzata inizialmente da Nicola, che poi però Beatrice ha dovuto completare essendo l'interazione con la view di sua competenza. Abbiamo quindi fatto combaciare le varie parti e realizzato una prima struttura dell'applicazione in cui erano presenti tutte le parti dell'MVC, e tutte interagivano tra loro, anche se ovviamente solo le minime funzionalità necessarie erano state realizzate. Una volta completato questo "prototipo" abbiamo poi continuato autonomamente nello sviluppo delle rispettive parti di progetto, confrontandoci però spesso, vista anche la concomitanza dello sviluppo con le lezioni. Successivamente, dopo un ulteriore sviluppo del model e della view abbiamo di nuovo collaborato per aggiustare il controller. Una volta realizzate le funzionalità principali che ci eravamo proposti viste

anche il numero di ore trascorse abbiamo deciso di non iniziare ad implementarne altre, soprattutto per quanto riguardava aspetti grafici e di animazione. Abbiamo quindi iniziato, autonomamente, una fase di raffinamento e ottimizzazione del codice, poiché anche se le funzionalità erano state realizzate in molti casi non si erano adottate le soluzioni migliori o comunque il codice non era organizzato in maniera sufficientemente corretta. Abbiamo poi dedicato alcune ore al controllo e "aggiustamento" del codice con gli strumenti, visti durante il corso, di *FindBugs*, *PMD* e *CheckStyle*.
Con rammarico non siamo riusciti a sfruttare le possibilità del DVCS poiché dopo alcuni tentativi per provarne le funzionalità nella prima fase di lavoro, a causa di diversi problemi tecnici e difficoltà nell'utilizzarlo, abbiamo deciso di procedere allo sviluppo e all'integrazione delle nostre parti di lavoro utilizzando metodi meno sofisticati e anche scambiandoci parti di codice manualmente quando necessario.

3.3 Note di sviluppo

Tutto il codice contenuto nel progetto è stato interamente realizzato da noi, ad eccezione di un metodo utilizzato da Beatrice per la creazione di **JLabel** contenenti testo e immagine, segnalato anche nel codice (classe **AbstractMenu**, metodo *createLabel()*). Il progetto è stato sviluppato da Beatrice su una macchina Windows, mentre Nicola ha lavorato utilizzando Ubuntu; non sono state riscontrate differenze rilevanti nel risultato finale su entrambe le macchine.

Capitolo 4

Commenti finali

4.1 Conclusioni e lavori futuri

Analizzando a posteriori quanto abbiamo realizzato, tenendo presente che si trattava della prima volta in cui ci misuravamo con un progetto di queste dimensioni, possiamo dirci nel complesso soddisfatti. Il lavoro è stato lungo ed impegnativo, ma riteniamo di aver realizzato un'applicazione abbastanza completa e aderente con quanto si intendeva fare e si era dichiarato in fase di proposta, cercando anche di produrre un codice di qualità migliore possibile.

Per quanto riguarda i limiti del nostro progetto non siamo del tutto soddisfatti del risultato grafico e di animazione in alcune fasi della partita. Nello specifico dobbiamo riconoscere che in certe situazioni, in particolare quando si verificano più combinazioni a catena, il gioco può diventare un po' caotico e non siamo stati in grado di rappresentare a livello visivo tutte le combinazioni che vengono a formarsi, per quanto al termine di ogni successione di mosse il risultato finale mostrato a schermo sia sempre quello corretto. Dobbiamo quindi riconoscere di non essere riusciti, nel tempo a disposizione, a lavorare a sufficienza sull'aspetto di sincronizzazione, forse anche un po' condizionati dalla scelta iniziale di avvalerci quasi esclusivamente di Swing.

Un ulteriore problema che abbiamo riscontrato nella fase finale di test riguarda la caramella wrapped, il cui behaviour specifico per esigenze implementative, abbiamo deciso di rendere utilizzabile solo se tale caramella non si trova su un bordo del campo. E' una problematica molto marginale visto che comunque buona parte del comportamento non potrebbe essere applicato partendo da un bordo, ma visto che la abbiamo rilevata abbiamo voluta segnalarla. Va comunque precisato che ciò non comporta problemi al funzionamento dell'applicazione e che la caramella in questione se si viene a trovare su un bordo reagisce come una qualsiasi altra caramella.

Per quanto riguarda l'estendibilità e possibili lavori futuri, posto che non conosciamo bene le norme relative a copyright e diritti di autore, non pensiamo di divulgare su particolari piattaforme il nostro lavoro; potremmo eventualmente valutare in futuro la possibilità di rimettere mano al progetto sostituendo la parte grafica con una versione più performante o realizzando una versione del modello maggiormente ottimizzata ed efficiente.

4.2 Difficoltà incontrate e commenti per i docenti

Anche se non abbiamo riscontrato difficoltà particolari possiamo sicuramente dire che è stato un progetto molto impegnativo. I problemi relativi a BitBucket sono sicuramenti dovuti a nostre mancanze. Un aspetto che ci ha messo un po' in difficoltà è stata la realizzazione del pattern MVC, che è stato più volte visto durante il corso, ma che quando poi ci siamo ritrovati a doverlo implementare veramente con le nostre mani ci è apparso meno chiaro di come pensavamo. In fase di integrazione tra le parti svolte autonomamente abbiamo notato una certa differenza nelle soluzioni implementative adottate: Beatrice ha più spesso fatto ricorso a dipendenze tra le classi, mentre Nicola ha più volte utilizzato numerosi passaggi di parametri. Abbiamo comunque cercato, nella parte finale del lavoro, di rendere il più coerente possibile il lavoro svolto dai due membri del gruppo.