Forza 4

Gianmarco Caldaroni - Matricola 1994231

13 luglio 2022

Figure 1: Una tipica griglia di Forza 4.

Contents

1 Introduzione **3**

2 Funzionalità del programma **4**
2.1 Creazione di una nuova partita 4
2.1.1 I giocatori 4
2.1.2 I pezzi 4
2.1.3 La griglia 5
2.2 Gestione della partita 5
2.2.1 Nuova partita 6
2.2.2 Salvataggio 7
2.2.3 Classi di testing 9
2.2.4 Uscire dal programma 10

3 Progettazione delle classi **11**
3.1 Descrizione delle classi 11
3.1.1 Classi e metodi principali 11
3.1.2 Classi e metodi di testing 15
3.2 Diagramma UML 17
3.2.1 Pacchetto game 17
3.2.2 Pacchetto sequence 19
3.2.3 Pacchetto tests 19
3.2.4 Pacchetto main 20
3.3 Programmazione ad oggetti e principi SOLID 21
3.3.1 Regole e convenzioni della programmazione ad oggetti . . 21
3.3.2 I principi SOLID 22

1 Introduzione

La seguente relazione vuole avere lo scopo di illustrare nel dettaglio il processo di realizzazione, con annesse tutte le specifiche e funzionalità, del **programma Forza 4**.

Prodotto dalla Milton Bradley Company (abbreviato MB), celebre per aver ideato altri giochi da tavolo diffusi in tutto il mondo come *Battaglia navale*, **Forza 4** venne commercializzato per la prima volta nel 1974, con il nome originale di *Connect 4*.

Di seguito ne sono descritte regole e curiosità:

1. In Forza 4 si gioca in due: ognuno dei due giocatori avrà una propria pedina che sarà di colore diverso da quella dell'avversario.

2. Si dispone una griglia, normalmente di dimensioni 6x7 (6 righe e 7 colonne). Ogni giocatore ha il semplice scopo di allineare (*connect*) a turno, prima dell'avversario, quattro pezzi del proprio colore in orizzontale, verticale o in diagonale. Il gioco risulta essere molto simile a *Tetris* (infatti conosciuto anche come *Vertical Tic-Tac-Toe* oppure *Gravitational Tic-Tac-Toe*), con la grande differenza che la griglia è disposta in verticale e che quindi le pedine saranno soggette a forza di gravità, posandosi di conseguenza sul fondo.

3. Forza 4 è un gioco "risolto": ovvero uno di quei giochi che con una sequenza di mosse pre-calcolate porta inevitabilmente alla vittoria. In particolare è un *first-player-win*, quindi, il giocatore che inizia la partita, giocando in maniera perfetta, può "forzare" la vittoria.

Lo scopo del programma è quello di riprodurre fedelmente una partita di Forza 4 **su console**.

Per la realizzazione del progetto sono stati utilizzati:

- Eclipse IDE: per la scrittura del codice in Java.
- git, GitHub: per la gestione facilitata delle varie versioni e modifiche apportate al programma. (link repository)
- Overleaf: per la scrittura di questa relazione in LaTeX.

2 Funzionalità del programma

Per una buona implementazione e realizzazione del progetto, ho ritenuto essenziale che il programma dovesse avere **due** macro-funzioni:

- Creazione di una nuova partita (2.1).
- Gestione della partita (2.2).

2.1 Creazione di una nuova partita

Per il set up di una nuova partita abbiamo bisogno di tre componenti fondamentali:

- Due giocatori (2.1.1).
- Due pezzi di colore diverso (2.1.2).
- La griglia in cui andranno inseriti i pezzi (2.1.3).

2.1.1 I giocatori

Per la creazione di due nuovi giocatori quello di cui abbiamo bisogno è semplicemente conoscerne i nomi: tramite un **oggetto Scanner** assegnamo al primo giocatore, che scriverà il proprio nome su console, il pezzo con cui giocherà. Lo stesso vale per il secondo giocatore che vedrà assegnatosi il pezzo rimanente.

2.1.2 I pezzi

Contrariamente a quanto il nostro intuito sembri suggerire, nella griglia ci sono non due, ma tre tipi di pezzi:

1. Il pezzo rosso.
2. Il pezzo giallo.
3. Anche se non "fisicamente" presente, ma solo "virtualmente" vi è anche la presenza di un pezzo *vuoto*: dove non presenti uno dei due pezzi, per la creazione successiva della griglia (2.1.3), abbiamo bisogno di un pezzo non visisbile all'utente, ma comunque presente, da sostituire poi con uno dei due colorati.

N.B. Per evitare problemi di sorta, nel programma ho scelto i colori rosso e giallo solamente per **indicare** i due pezzi, non utilizzando i cosiddetti ANSII escape code per la colorazione di testo in console, dato che potrebbero dare problemi al codice in fase di output. Siccome la presenza o l'assenza di pezzi colorati non influenza in alcun modo il funzionamento del programma, rifacendomi al gioco del *Tetris* (1) ho deciso che il carattere 'O' sta a rappresentare il pezzo rosso, il carattere 'X' il pezzo giallo ed infine uno spazio ' ' a rappresentare il pezzo vuoto.

Essendo, infine, che il pezzo può assumere solamente tre valori costanti possibili, sono stati implementati sfruttando il tipo *Enum*.

2.1.3 La griglia

Ora che abbiamo definito i giocatori e soprattutto i pezzi, possiamo inizializzare una nuova griglia.

Definiamo innanzitutto delle dimensioni costanti di quella che non è altro che una *matrice*, rappresentata in Java con un **array a due dimensioni** avente due componenti:

- La larghezza: il numero di colonne, ovvero sette.
- L'altezza, che di fatto rappresenta il numero di righe, ovvero sei.

Usando un costruttore realizziamo quindi la nostra griglia, che inizialmente sarà composta solamente da pezzi vuoti. Sfruttando il concetto del **polimorfismo** (3.3.1), nel codice è stato inserito anche un *overloading* del costruttore: esso prenderà un *ArrayList* di stringhe in modo tale da costruire una griglia con vari pezzi già preimpostati, che ci sarà utile successivamente per le classi di testing (2.2.3).

Nella classe sono stati poi inseriti numerosi metodi che ci serviranno per il funzionamento del gioco (vedi 3 e 3.2 per i dettagli).

Infine la griglia sarà stampata nella console tramite *l'override* del *toString()* della superclasse Object, facendo uso di:

- **Stringbuilder class**: per implementare in maniera facilitata la rappresentazione a schermo dello scheletro della griglia.
- Lo scheletro in sé, è stato invece costruito tramite caratteri speciali, sfruttando degli **ALT codes shortcut** (reperibili su questo link), per una raffigurazione più elegante.

  ATTENZIONE: Per la visualizzazione della griglia con i suddetti caratteri, assicurarsi che *l'encoding* del testo sia impostato su "**UTF-8**".

Figure 2: Visualizzazione di una griglia vuota su console.

2.2 Gestione della partita

Avendo completato i preparativi necessari descritti nella precedente sezione, è arrivato il momento di iniziare un'effettiva partita. Attraverso la classe Menu, all'avvio del gioco l'utente si troverà di fronte a quattro possibili scelte:

1. Iniziare una nuova partita (2.2.1).
2. Caricare una partita esistente precedentemente salvata (2.2.2).
3. Avviare la classi di testing (2.2.3).
4. Uscire dal programma (2.2.4).

2.2.1 Nuova partita

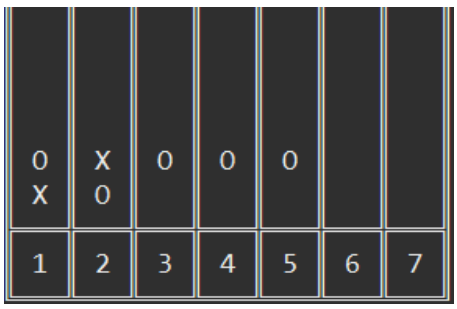
Selezionando l'opzione uno, i due giocatori dovranno innanzitutto scrivere i loro nomi, quindi il programma svolgerà le seguenti funzioni:

- Sceglie il giocatore che inizierà la partita, attraverso un valore *booleano*. Ho deciso in questo modo data l'assenza di regole ufficiali su chi dei due debba partire per primo; sfruttando un oggetto di tipo **Random** si genera casulmente un booleano: se sarà *true* il giocatore rosso partirà per primo, altrimenti sarà il giallo. In questo modo ogni giocatore ha un 50% di probabilità di incominciare.
- Attraverso un **while loop**, sempre grazie all'ausilio di uno Scanner, leggeremo quello che l'utente ci darà in input ed in base ad esso, attraverso vari controlli, il programma svolgerà varie azioni:
  - Se l'input è un numero valido, ovvero compreso tra 1 e 7: si passerà al comando successivo (vedi prossimo punto).
  - Se l'input non è un numero valido: ovverosia un numero minore di 1 o maggiore di 7, oppure un numero scelto dove un colonna è già stata riempita, sarà stampato un errore che comunicherà all'utente di riprovare a scrivere un numero appropriato.
  - Se l'input è del tipo "save filename": in questo caso entra in gioco la funzione di salvataggio del programma (2.2.2).
  - Se l'input è qualsiasi altra cosa: lettere, parole, simboli... il programma anche in questo caso stamperà un messaggio di errore comunicando all'utente di riprovare ad inserire un numero.
- Tornando nel caso in cui l'input sia un numero valido: il pezzo sarà aggiunto alla griglia nella posizione desiderata, ovviamente partendo dal fondo e man mano posandosi sull'ultimo pezzo della colonna scelta.
- Ad ogni turno il programma si occuperà di controllare la presenza di un vincitore attraverso il seguente algoritmo:

  per ogni verso in cui bisogna verificare la presenza di un vincitore sono stati implementati i rispettivi metodi: *checkHorizontal()*, *checkVertical()*, *checkDiagonal()* e *checkAntiDiagonal()*. Se c'è un vincitore viene visualizzato a schermo il suo nome, altrimenti si continua finchè non se ne trova uno oppure finchè la griglia sarà piena, finendo in parità.

- Nel caso in cui ci sia un vincitore, non ne viene solamente stampato a schermo il nome, ma viene salvata anche la **winning sequence**, ovvero il colore del pezzo vincitore e le coordinate dei quattro pezzi connessi. Quest'ultima viene utilizzata per un sistema di animazione: sfruttando il metodo *Thread.sleep()* che manda in pausa l'esecuzione del programma per vari millisecondi, insieme ad un sistema di **blinking** che sostitusice temporaneamente i pezzi vincitori con degli spazi (ovvero pezzi vuoti) e viceversa, si crea un effetto "lampeggiante" che mette in risalto la sequenza vincitrice.

- Se nel caso di una vittoria da parte di un gioctore in due o più sensi possibili (ovvero in contemporanea verticale ed orizzontale ad esempio), l'animazione verrà effettuata sulla sequenza orizzontale dato che i metodi *check* sono stati implementati nel seguente ordine: orizzontale, verticale, digonale e antidiagonale, quindi anche se esiste una sequenza di vittoria in verticale verrà visualizzata solamente quella orizzontale.



(a) Primo stato dell'animazione.

(b) Secondo stato dell'animazione.

2.2.2 Salvataggio

Creazione del file di salvataggio Per la funzione di salvataggio del programma ho deciso di implementare un'interfaccia *Savable*: l'unico metodo definito al suo interno, *toSaveState()*, verrà implementato nelle componenti del programma effettivamente utili per la creazione di una partita:

- Nella classe Player per salvarci il nome dei giocatori.
- Nell'enum Piece per salvarci i pezzi della partita
- Nella Board per salvarci l'effettivo stato della griglia.

Nella classe Game con il metodo **saveSession**, che prende il nome del file come parametro, scriviamo il file di salvataggio tramite la classe PrintWriter. Successivamente il metodo *parseCommand()* ci darà la possibilità di salvare una partita in qualunque momento del gioco tramite un semplice comando: "save

filename". **Save** sarà quindi l'effettivo comando che dirà al programma di salvare la partita e, chiaramente, filename è il nome del file.

Il file appena scritto verrà salvato e posizionato direttamente nello stesso percorso della directory del progetto. Impicitamente, quindi, il programma sarà in grado di immagazzinare più file di salvataggio contemporanemante.

Apertura di un file di salvataggio Nel menù principale ad avvio del programma, la seconda opzione ci permetterà di caricare un file salvato. Il programma leggerà il file attraverso uno Scanner. Sfruttando un ArrayList<String> ci salviamo i nomi dei giocatori, il booleano e la griglia, per poi successivamente **rimuovere i primi due** dal file, in modo tale che quando andremo a caricare il file, visualizzeremo solamente la griglia, costruendo una nuova partita con le informazioni lette.

Di seguito la visualizzazione di un file tipo che verrà letto dal nostro programma:

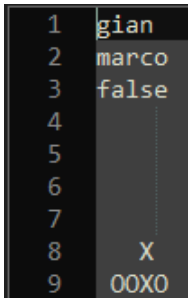


Figure 4: Prototipo di una partita salvata.

Accorgimenti:

- Le prime due informazioni che notiamo sono i nomi dei rispettivi giocatori: alla riga uno il nome del giocatore con il pezo rosso, alla due il nome del giocatore con il giallo.
- Come seconda informazione importante abbiamo un booleano: per capire chi dovrà effettuare la prima mossa sfrutteremo il booleano *isRedTurn* generato casualmente con Random (vedi 2.2.1). In pratica se è il turno del rosso il booleano salvato sarà true, se tocca al giallo, come nel nostro esempio, toccherà al giocatore "marco", quindi il booleano sarà false.
- Le righe di ogni file salvato notiamo corrispondere esattamente a nove (6 per la griglia, il booleano e i due nomi). Ogni file con un numero di righe diverso da 9 (sia maggiore o minore) non verrà letto del programma e lancierà un'eccezione, comunicando all'utente che il file selezionato non è valido; stesso concetto se presente un carattere estraneo nella griglia o qualsiasi altra cosa scritta al posto di true o false nella terza riga. Notare che al contrario delle due ultime informazioni, i nomi dei giocatori non

sono essenziali per ricaricare una partita, quindi i due utenti se hanno la volontà di cambiare nomi possono farlo modificando direttamente il file sorgente, facendo attenzione a non alterare il numero di righe.

- Se il file viene salvato con un nome di un altro salvataggio esistente, allora quest'ultimo verrà sovrascritto con i nuovi dati.

- Per il nome del file sono ammesse solamente lettere maiuscole, minuscole e numeri. Simboli del tipo "/()=!" ecc. non sono supportati e il programma comunicherà che non è stato possibile salvare il file per la presenza di caratteri non ammessi. Chiaramente per la scrittura delle estensioni il punto è un carattere valido. Per accertarmi della presenza di caratteri invalidi ho utilizzato la tavola ASCII.

2.2.3 Classi di testing

Per una verifica della correttezza del codice, dovendomi assicurare il funzionamento del programma per tutti i casi limite possibili, ho implementato le seguenti classi che fungono da *tester* (avviabili direttamente dal menù scegliendo l'opzione tre):

- **BoardTester**: per ogni senso in cui è possibile conseguire una vittoria, ho creato i seguenti casi limite: in basso ed in alto a destra e sinistra, e il caso di pareggio. Come detto in 2.1 traminte *overloading* del costruttore, è stato possibile preimpostare una griglia con i pezzi già posizionati prima scritti tramite un *Array* e poi convertiti in una *List* di stringhe.

- **GameTester**: questa classe è stata invece utilizzata per testare i vari tipi di input sbagliati scritti dall'utente. In particolare numeri minori di 1 o maggiori di 7; se il numero scelto ricade su una colonna già riempita; se l'input è del tipo "save filename"; se l'input sono lettere o simboli qualsiasi ed infine se il numero è compreso fra 1 e 4 se siamo nel caso del menù iniziale.

- **WinningSequenceTester**: ho ritenuto opportuno realizzare anche una classe che mi assicuri che la sequenza di vittoria del giocatore sia effettivamente corretta. Sfruttando lo stesso approccio per il BoardTester, creando quattro casi generici di vittoria, ho anche in questo caso, fatto uso di due costruttori:

  con un primo mi sono creato un *oggetto winningSequence* passandogli come parametri i risultati attesi. Poi usando il metodo *checkWinner()*, ho creato una seconda sequenza con un secondo costruttore vuoto della classe WinningSequence. Infine ho comparato le due sequenze: se sono uguali il test è passato. Per la comparazione è stato necessario effettuare l'*override* del metodo **equals** della classe Object: dato che il metodo restituisce *true* solo se si tratta di due riferimenti allo stesso oggetto, per un confronto più appropriato è stato necessario ridefinire equals nella classe *Coordinates*,

che confronta le componenti delle coordinate (x, y), ed anche nella classe WinningSequence, che confronta se il pezzo e le quattro coordinate siano gli stessi.

2.2.4 Uscire dal programma

Nel caso in cui l'utente voglia scegliere di uscire dal programma e terminare la partita, il semplice metodo *exit()*, definito nella classe **Main** scegliendo l'opzione quattro, e in **parseCommand()** della classe Game scrivendo "exit" oppure "quit", sfrutta il metodo della java.lang *System.exit(0)* che termina l'esecuzione, segnalando (per convenzione usando lo 0) l'uscita dal programma con successo.

3 Progettazione delle classi

3.1 Descrizione delle classi

Legenda:

- classe NomeClasse (affianco scritto se implementano una classe o lanciano un'eccezione):

  Attributi:

  - attributi della classe

  Metodi:

  - nome metodo(TipoParametro parametro): valore ritornato (se non *void*)

3.1.1 Classi e metodi principali

- **class Main**:

  Attributi:

  - **private static** Scanner scanner

  Metodi:

  - **private static void** gameLoop(Game game)
  - **private static void** game()
  - **private static void** load()
  - **private static void** tests()
  - **private static void** exit()
  - **private static void** main(String[] args)

- **class Menu**:

  Attributi:

  - **private static** Scanner scanner

  Metodi:

  - **public** Menu()
  - **public static Integer** chooseOption(): Integer playerInput
  - **private static boolean** isValidInput(Integer playerInput): boolean

- **enum Piece** implements **Savable**:

  Attributi:

  - YELLOW
  - RED
  - EMPTY

  Metodi:

  - **public static Piece** fromCharacter(char character) *throws* CharConversionException: YELLOW or RED or EMPTY
  - (@Override) **public String** toSaveState(): String this.toString()
  - (@Override) **public String** toString(): String

- **class Player** implements **Savable**:

  Attributi:

  - **private static** Scanner scanner
  - **String** name

  Metodi:

  - **public** Player()
  - **public** Player(String name)
  - **public String** getName(): String name
  - **public void** setName(String name)
  - **public void** setNameFromInput()
  - **public static** Player fromSaveState(String input): new Player(input)
  - (@Override) **public String** toSaveState(): String
  - (@Override) **public String** toString(): String name

- **class Board** implements Savable:

  Attributi:

  - **private final static int** WIDTH
  - **private final static int** HEIGHT
  - **private Piece**[][] pieces

  Metodi:

  - **public static int** getWidth(): WIDTH
  - **public static int** getHeight(): HEIGHT
  - **public** Board()
  - **public** Board(ArrayList<String> lines) *throws* CharConversionException
  - **public boolean** isFilled(): boolean

- **public boolean** isColoumnFull(Integer playerInput): boolean
- **public void** makeMove(Integer playerInput, Piece turn)
- **public Player** checkWinner(Player redPlayer, Player yellowPlayer, WinningSequence winningSequence): winner or null
- **private Piece** checkHorizontal(WinningSequence winningSequence): color or null
- **private Piece** checkVertical(WinningSequence winningSequence): color or null
- **private Piece** checkDiagonal(WinningSequence winningSequence): color or null
- **private Piece** checkAntiDiagonal(WinningSequence winningSequence): color or null
- **private boolean** checkEquals(Piece piece1, Piece piece2, Piece piece3, Piece piece4): boolean
- **private Player** checkColor(Player redPlayer, Player yellowPlayer, Piece color): redPlayer or yellowPlayer or null
- **public void** blink(WinningSequence winningSequence)
- **public static** Board fromSaveState(ArrayList<String> lines) *throws* CharConversionException: board
- **public void** setPieces(Piece[][] pieces)
- **public Boolean** evaluateTurn(boolean firstTurn): boolean
- (@Override) **public String** toSaveState(): StringBuilder outputString
- (@Override) **public String** toString(): String outputString

- **class Coordinates**:

  Attributi:

  - **private int** x
  - **private int** y

  Metodi:

  - **public** Coordinates(int x, int y)
  - **public int** getX(): x
  - **public void** setX(int x)
  - **public int** getY(): y
  - **public void** setY(int y)
  - (@Override) **public boolean** equals(Object object): boolean
  - (@Override) **public String** toString(): String

- **class WinningSequence**:

  Attributi:

  - **private Piece** piece
  - **private Coordinates** first
  - **private Coordinates** second
  - **private Coordinates** third
  - **private Coordinates** fourth

  Metodi:

  - **public** WinningSequence(Piece piece, Coordinates first, Coordinates second, Coordinates third, Coordinates fourth)
  - **public** WinningSequence()
  - **public Piece** getPiece(): piece
  - **public void** setPiece(Piece piece)
  - **public Coordinates** getFirst(): first
  - **public void** setFirst(Coordinates first)
  - **public Coordinates** getSecond(): second
  - **public void** setSecond(Coordinates second)
  - **public Coordinates** getThird(): third
  - **public void** setThird(Coordinates third)
  - **public Coordinates** getFourth(): fourth
  - **public void** setFourth(Coordinates fourth)
  - (@Override) **public boolean** equals(Object object): boolean
  - (@Override) **public String** toString(): String

- **interface Savable**:

  Metodi:

  - **public String** toSaveState()

- **final class Game**:

  Attributi:

  - **private boolean** isRedTurn
  - **private boolean** firstTurn
  - **private static Scanner** scanner
  - **private Player** redPlayer
  - **private Player** yellowPlayer

- **private Board** board

Metodi:

- **public** Game()
- **public** Game(boolean isRedTurn, boolean firstTurn, Player redPlayer, Player yellowPlayer, Board board)
- **public void** setUp()
- **private Piece** updateTurn(): turn
- **public Player** run(): winner or run
- **private void** winningAnimation(WinningSequence winningSequence)
- **private boolean** chooseFirstPlayer(): boolean
- **private static void** clearScreen()
- **private int** chooseMove(): playerInput
- **private boolean** parseCommand(String rawInput): boolean
- **private static boolean** isValidFileName(String fileName): boolean
- **private void** saveSession(String fileName) *throws* FileNotFoundException
- **private boolean** isValidInput(Integer playerInput): boolean
- **private static ArrayList<String>** readLines(String fileName) *throws* FileNotFoundException, IllegalArgumentException: ArrayList<String> lines
- **public static Game** fromSaveState(String fileName) *throws* FileNotFoundException, CharConversionException, IllegalArgumentException: new Game(isRedTurn, firstTurn, redPlayer, yellowPlayer, board)

3.1.2 Classi e metodi di testing

- **class BoardTester**:

  Metodi:

  - **public static void** test()
  - **private static void** isPassed(String[] strings)
  - **private static void** testHorizontal()
  - **private static void** testVertical()
  - **private static void** testDiagonal()
  - **private static void** testAntiDiagonal()
  - **private static void** testDraw()

- **class GameTester**:

  Attributi:

  - **private static boolean** isNotValid

  Metodi:

  - **public static void** test()
  - **private static void** isPassed(String rawInput)
  - **private static void** testFullColumnInput(Integer playerInput)
  - **private static void** isSaveInput(String rawInput)
  - **private static void** menuInput(String rawInput)
  - **private static void** testInput()

- **class WinningSequenceTester**

  Metodi:

  - **public static void** test()
  - **private static WinningSequence** winningSequenceBuilder(String[] strings): winningSequence
  - **private static void** isPassedHorizontal()
  - **private static void** isPassedVertical()
  - **private static void** isPassedDiagonal()
  - **private static void** isPassedAntiDiagonal()

3.2 Diagramma UML

Nella seguente sezione verrà mostrato un overview del diagramma UML, per poi analizzarlo nel dettaglio diviso in pacchetti. Per la creazione del diagramma UML sono state scartate, almeno nel diagramma generale e nei pacchetti più complessi, le associazioni, ovvero qualsiasi tipo di relazione di base tra le classi, per fare in modo che il diagramma non risulti confusionario andando a creare troppe freccie (che stanno appunto a rappresentare l'associazione tra le classi) intersecate fra loro. Si è posto invece l'accento sulle associazioni **dirette** dove vi è una forte realzione tra le due classi (vedi ad esempio 3.2.2), e l'implementazione, o *realization*, dove una classe implementa un metodo definito da un'altra classe, nel nostro caso un'interfaccia (vedi 3.2.1).

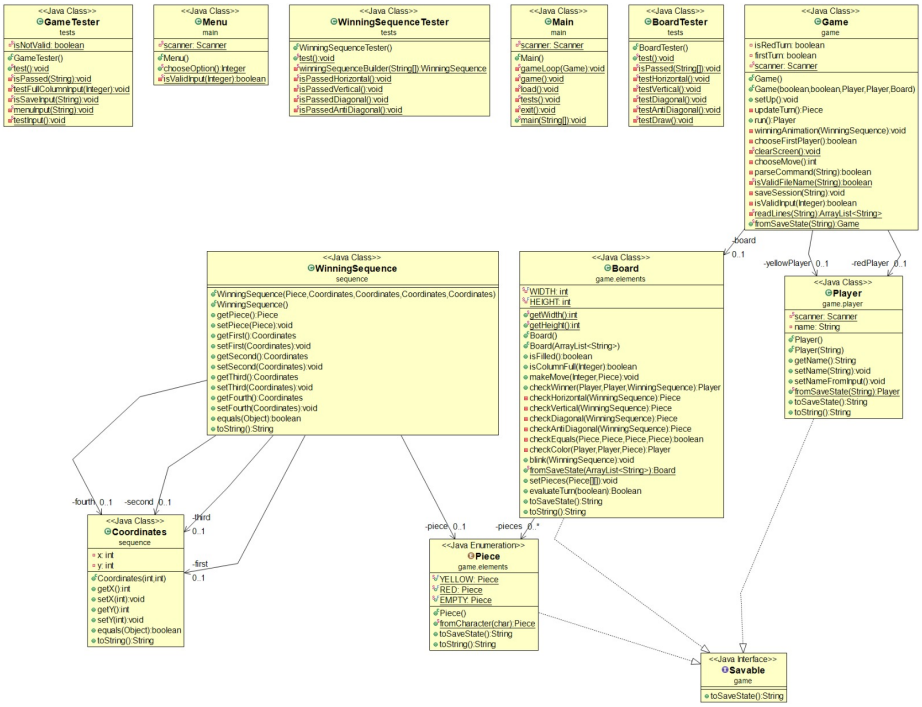


Figure 5: Un overview del diagramma UML.

3.2.1 Pacchetto game

Il pacchetto game è il più articolato del nostro programma. Esso contiene una classe e un'interfaccia, più due ulteriori pacchetti.

Classi nel pacchetto game

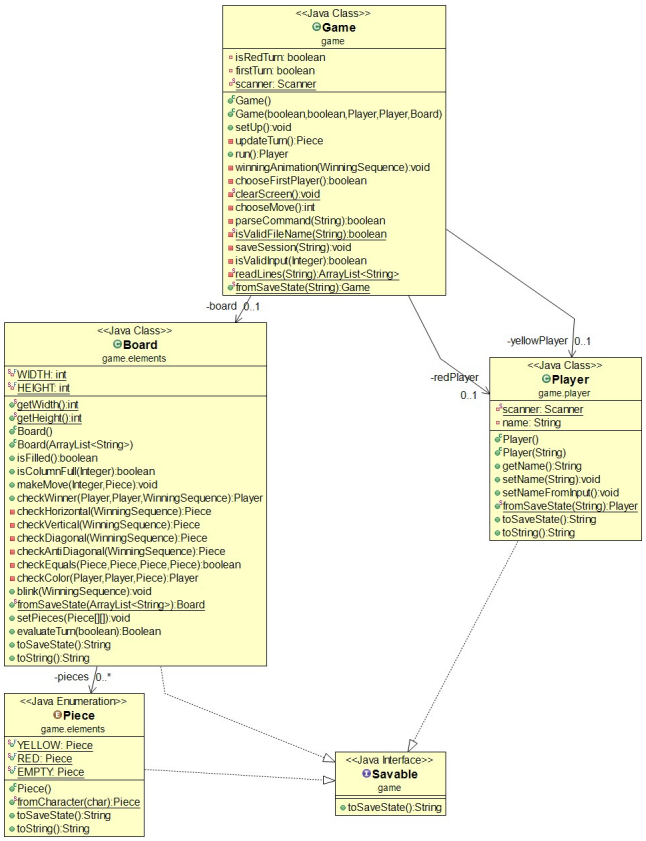


Figure 6: Diagramma UML del pacchetto game.

- **Game**: inizializza una nuova partita e definisce anche i metodi utili per il salvataggio.
- **Savable**: un'*interfaccia* che definisce il metodo per salvare lo stato della griglia, dei giocatori e dei pezzi.

Pacchetto elements

- **Board**: classe che ha il compito di costriure una nuova griglia e definirne tutti i metodi per il gioco.
- **Piece**: un *enum* che definisce i tre pezzi principali che possono apparire nella griglia.

Pacchetto player

- **Player**: classe che definisce due nuovi giocatori.

3.2.2 Pacchetto sequence

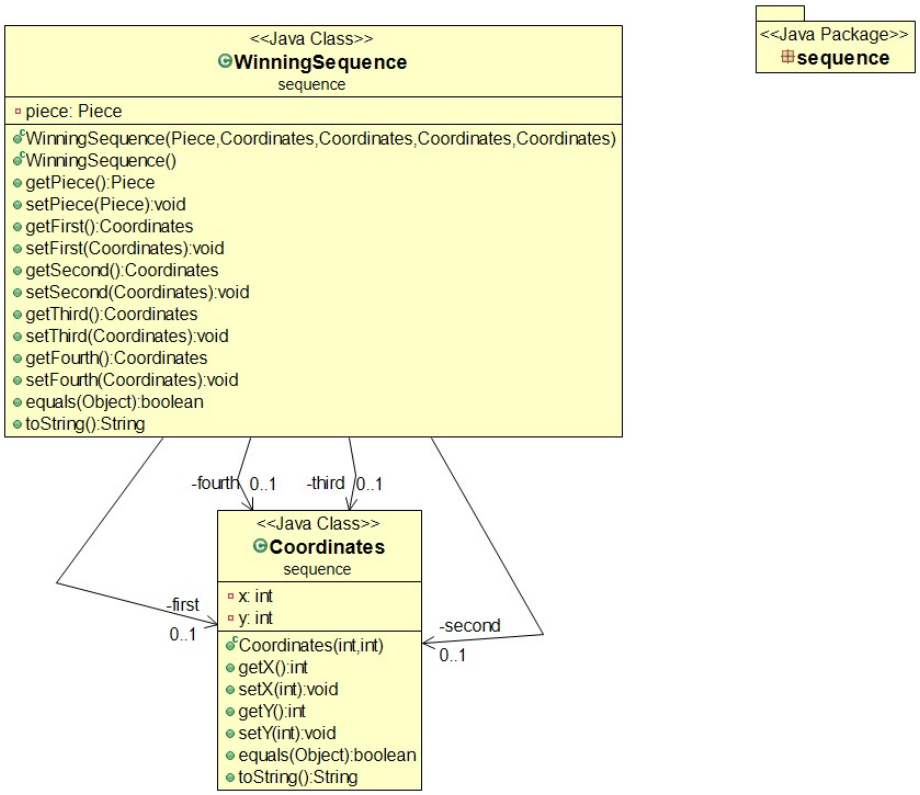


Figure 7: Diagramma UML del pacchetto sequence.

Il pacchetto sequence si occupa del salvataggio della *winning sequence*.

Classi nel pacchetto sequence

- **Coordinates**: salva le componenti delle coordinate dei quattro pezzi della sequenza.
- **WinningSequence**: crea la sequenza utile per l'animazione di vittoria, contenente il pezzo vincitore (rosso o giallo) e le quattro coordinate.

3.2.3 Pacchetto tests

Il pacchetto tests di occupa dei test.

Classi nel pacchetto tests

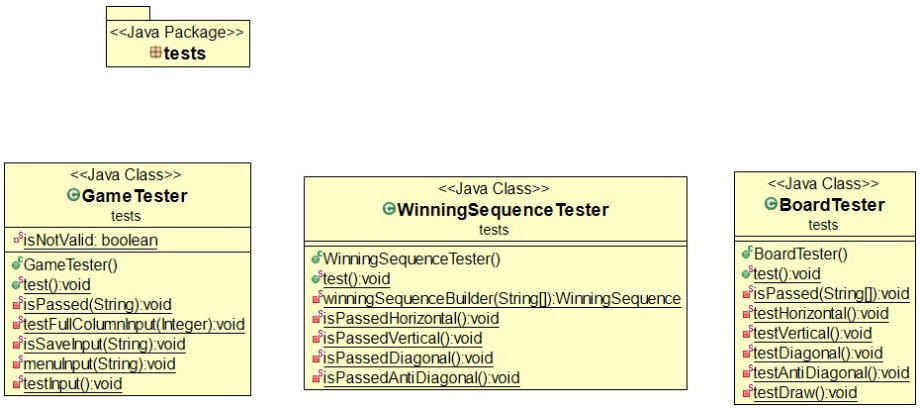


Figure 8: Diagramma UML del pacchetto tests.

- **GameTester**: si occupa dei possibili input errati che l'utente può fare, sia in fase di gioco che in menù.
- **WinningSequenceTester**: classe che testa la correttezza delle sequenze di vittoria.
- **BoardTester**: testa la correttezza dei metodi di ricerca di un possibile vincitore o di parità.

3.2.4 Pacchetto main

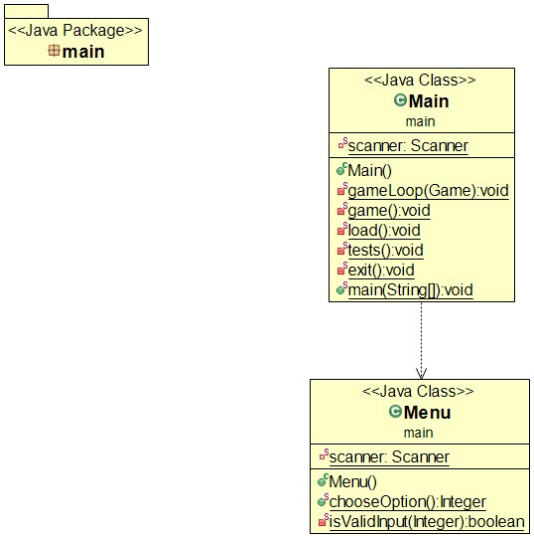


Figure 9: Diagramma UML del pacchetto main.

Il pacchetto main è il principale in quanto contiene il main del programma.

Classi del pacchetto game

- **Menu**: il menu si occupa della creazione delle opzioni che appariranno all'avvio della partita.
- **Main**: la nostra classe starter.

3.3 Programmazione ad oggetti e principi SOLID

Come ultima sezione della relazione porterò l'attenzione sui principi SOLID e le convenzioni della programmazione ad oggetti rispettati nel progetto.

3.3.1 Regole e convenzioni della programmazione ad oggetti

Polimorfismo Tra i vantaggi maggiori della programmazione ad oggetti vi è il polimorfismo: la possibilità di un metodo di assumere valori diversi in base al tipo di dato su cui viene applicata, tramite overloading (implementazione di più metodi con lo stesso nome, ma con argomenti e possibili valori di return diversi) e overriding (completa sovrascrittura di metodi ereditati da una superclasse). In paricolare:

- Overloading con:
  - Il costruttore della Board.
  - Il costruttore del Game.
  - Il costruttore del Player.
  - Il costruttore della WinningSequence.
- Overriding con:
  - Il metodo toString() della superclasse Object in Board, Piece, Player, Coordinates e WinningSequence.
  - Il metodo equals() della superclasse Object in Coordinates e WinningSequence.
  - L'implementazione del toSaveState() dall'interfaccia Savable in Piece, Player e Board.

Modificatori Durante la scrittura del codice è stata posta particolare attenzione alla scelta dei modificatori di visibilità public e private di ogni attributo e metodo della classe. Stesso discorso per i modificatori static e final, per esempio il numero di righe e colonne definite come final static. Quindi non possono essere modificate e saranno condivise fra tutte le istanze di quella classe.

Consistenza I nomi delle variabili e classi sono stati assegnati secondo le convenzioni della Java Code Convention:

- Nomi delle classi scritti in PascalCase.
- Attributi scritti in camelCase.
- Attributi costanti scritti in MAIUSCOLO.

3.3.2 I principi SOLID

Per una buona programmazione ad oggetti sono stati rispettati tutti i principi SOLID, ma si è posta l'attenzione principalmente su:

- S-Single responsability principle: ogni classe deve avere una ed una sola responsabilità, interamente incapsulata al suo interno.
- I-Interface segragation principle: interfacce specifiche e non generiche (Savable).

GRAZIE