Tetraj

Progetto di Programmazione ad Oggetti (12 CFU)

Patrizio Bertozzi
patrizio.bertozzi@studio.unibo.it

Università di Bologna
Corso di Laurea in Ingegneria e Scienze Informatiche

Anno Accademico 2024/2025

Indice

1 Analisi **2**
- 1.1 Descrizione e requisiti . . . 2
  - 1.1.1 Requisiti funzionali . . . 2
  - 1.1.2 Requisiti non funzionali . . . 3
- 1.2 Modello del Dominio . . . 4

2 Design **6**
- 2.1 Architettura . . . 6
- 2.2 Design dettagliato . . . 7
  - 2.2.1 Selezione dei pezzi . . . 7
  - 2.2.2 Rendering delle View . . . 8
  - 2.2.3 Tetromini: gerarchia e creazione . . . 9
  - 2.2.4 Gestione degli stati di gioco . . . 11

3 Sviluppo **14**
- 3.1 Testing automatizzato . . . 14
- 3.2 Note di sviluppo . . . 15

4 Commenti finali **18**
- 4.1 Autovalutazione e lavori futuri . . . 18
- 4.2 Difficoltà incontrate e commenti per i docenti . . . 19

A Guida utente **20**
- A.1 Avvio del gioco . . . 20
- A.2 Schermata iniziale . . . 21
- A.3 Durante la partita . . . 22
- A.4 Punteggio . . . 23
- A.5 Fine partita e classifica . . . 24

B Esercitazioni di laboratorio **26**

Capitolo 1

Analisi

1.1 Descrizione e requisiti

Tetraj è una riproduzione del videogioco Tetris [1]. Il giocatore manipola forme geometriche chiamate tetromini (pezzi composti da quattro blocchi quadrati) che cadono dall'alto dell'area di gioco. L'obiettivo è posizionare i tetromini completando linee orizzontali senza lasciare spazi vuoti: le linee completate scompaiono e il giocatore guadagna punti. La partita termina quando i pezzi si accumulano fino a raggiungere la parte superiore dell'area di gioco.

I tetromini sono sette, identificati da una lettera che ne richiama la forma: I (linea di 4 blocchi), O (quadrato), T, S, Z, J e L. Ogni tetromino ha un colore distintivo. L'area di gioco ha dimensioni standard: 10 celle in larghezza e 20 in altezza.

1.1.1 Requisiti funzionali

- **Gestione dei tetromini**
  - Supporto per tutti e sette i tetromini standard, ciascuno con il proprio colore
  - Rotazione in senso orario e antiorario
  - Caduta automatica con velocità crescente in base al livello
  - Movimento orizzontale, a sinistra e a destra
  - Soft drop (discesa accelerata) e hard drop (caduta istantanea)
- **Selezione dei pezzi**
  - Due modalità di selezione: "7-bag randomizer" (moderna) dove i sette tetromini vengono mescolati e distribuiti ciclicamente, oppure selezione completamente casuale (classica) come nel Tetris originale
  - Anteprima del prossimo pezzo in arrivo
- **Hold**
  - Possibilità di trattenere il pezzo corrente per usarlo successivamente
  - La funzione hold è utilizzabile una sola volta per ogni pezzo che cade

- **Ghost piece**
  - Visualizzazione di un'ombra che indica dove atterrerà il tetromino corrente
- **Wall kick**
  - Spostamento automatico del pezzo in una posizione valida quando una rotazione causerebbe collisione
- **Linee e punteggio**
  - Eliminazione delle linee completamente piene
  - Punteggio basato sul numero di linee eliminate simultaneamente
  - Bonus significativo per il "Tetris" (4 linee contemporanee)
  - Punti bonus per il soft drop proporzionali alla distanza percorsa
  - Punti bonus per l'hard drop proporzionali alla distanza di caduta
- **Livelli e velocità**
  - Aumento del livello ogni 10 linee completate
  - Due curve di velocità: valori predefiniti per livello (classica) oppure formula di crescita continua (moderna)
- **Stati e controlli**
  - Menu principale, partita in corso, game over
  - Pausa durante il gioco
  - Effetti sonori per le azioni principali
- **Classifica**
  - Memorizzazione dei 10 migliori punteggi ottenuti
  - Richiesta del nickname al giocatore se il punteggio entra in classifica
  - Persistenza della classifica
  - Visualizzazione della classifica al termine della partita

1.1.2 Requisiti non funzionali

- **Portabilità**
  - Esecuzione su Windows, macOS e Linux
- **Prestazioni**
  - Frame rate fluido e costante
  - Assenza di flickering durante il rendering
- **Usabilità**
  - Visualizzazione chiara di: area di gioco, pezzo corrente, prossimo pezzo, pezzo trattenuto, punteggio, livello, linee completate

1.2 Modello del Dominio

Il dominio di Tetraj ruota attorno a poche entità fondamentali.

L'elemento centrale è il **tetromino**, una forma geometrica composta da quattro blocchi. Esistono sette tipi di tetromino (I, O, T, S, Z, J, L), ciascuno con una forma e un colore caratteristici [2]. Ogni tetromino possiede una posizione nell'area di gioco e uno stato di rotazione. Un tetromino può muoversi orizzontalmente, cadere verso il basso e ruotare.

L'**area di gioco** è una griglia rettangolare (10 celle in larghezza e 20 in altezza) che contiene i blocchi dei tetromini già posizionati. L'area di gioco ha la responsabilità di verificare se una posizione è valida (non fuori dai bordi, non sovrapposta ad altri blocchi), di accogliere i tetromini quando si depositano, e di rilevare e rimuovere le linee complete.

La **partita** rappresenta una sessione di gioco completa. Una partita tiene traccia del punteggio, del livello corrente, del numero di linee eliminate, e gestisce la successione dei tetromini: il pezzo corrente in caduta, il prossimo pezzo in attesa, e l'eventuale pezzo trattenuto. La partita determina quando il gioco termina, ovvero quando un nuovo tetromino non può essere posizionato in cima all'area di gioco.

La **strategia di selezione** determina quale sarà il prossimo tetromino. Le due modalità (classica e moderna) differiscono nel criterio di scelta ma condividono lo stesso ruolo: fornire il prossimo pezzo alla partita.

La **strategia di velocità** determina quanto rapidamente i tetromini cadono in base al livello corrente. Anche qui le due modalità (classica e moderna) differiscono nella formula di calcolo ma hanno lo stesso scopo.

La **classifica** mantiene l'elenco dei migliori punteggi ottenuti. Ogni voce della classifica è rappresentata da un **elemento** che associa un punteggio ad altre informazioni rilevanti (ad esempio la data). La classifica è ordinata in modo decrescente per punteggio e può avere un numero massimo di 10 voci. La classifica deve essere persistente tra le sessioni di gioco, delegando la memorizzazione a un sistema di storage esterno. Al termine di una partita, se il punteggio ottenuto è sufficientemente alto, il giocatore può inserire il proprio record in classifica.

Infine il **giocatore** interagisce con la partita attraverso comandi: spostare il pezzo, ruotarlo, farlo cadere rapidamente, metterlo in hold, mettere in pausa.

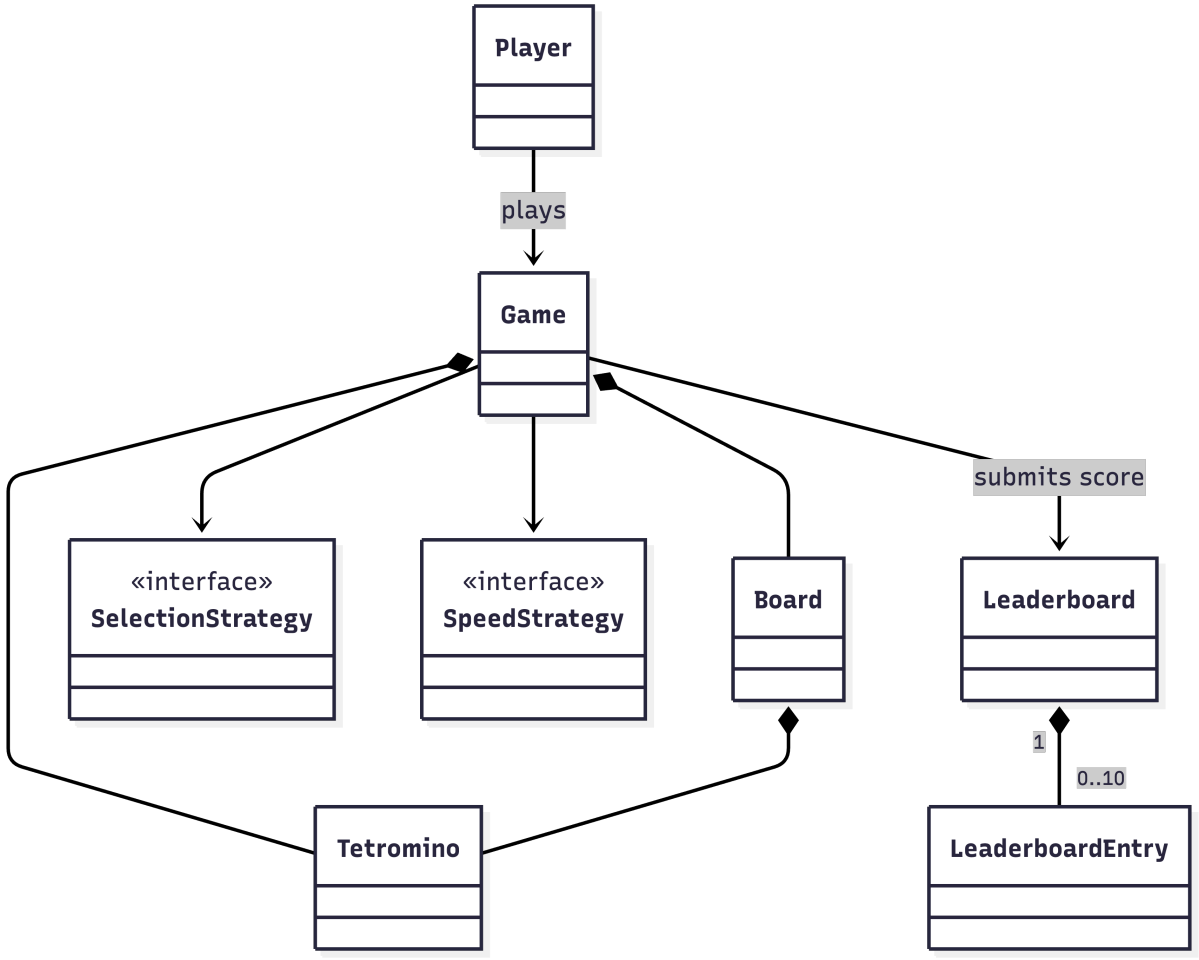

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

L'architettura di Tetraj segue il pattern architetturale **MVC** (Model-View-Controller) [3] combinato con una **macchina a stati finiti** (FSM) [4] per la gestione delle transizioni tra le diverse schermate del gioco.

Il coordinamento tra i componenti è affidato a un **Service Locator** [5] che funge da punto centralizzato per l'accesso ai servizi condivisi; crea e collega i componenti all'avvio, e li rilascia allo spegnimento.

Il **Game Loop** [6] costituisce il cuore pulsante dell'applicazione: ad ogni iterazione interroga la macchina a stati per ottenere il controller attivo, ne aggiorna la logica e ne richiede il rendering.

La gestione dell'input è disaccoppiata dalla logica di gioco tramite il pattern **Command** [7]: un gestore di input mappa i tasti a comandi, permettendo configurazioni diverse per ogni stato senza modificare la logica dei controller.

Ogni schermata del gioco ha il proprio trittico MVC, composto dal **Model** (`PlayModel`, `MenuModel`, ecc.) contenente lo stato e la logica di dominio, dalla **View** (`PlayView`, `MenuView`, ecc.) che è responsabile esclusivamente del rendering ed estende una classe base comune per la gestione delle risorse grafiche, e infine dal **Controller** (`PlayController`, `MenuController`, ecc.) che implementa l'interfaccia `Controller` definendo il ciclo di vita (`enter`, `exit`), l'aggiornamento (`update`), il rendering (`render`), la gestione dell'input (`handleInput`, `handleInputRelease`) e l'esposizione del componente grafico (`getCanvas`) che il Game Loop monta nella finestra principale. Ogni controller possiede il proprio Model e la propria View, coordinandone l'interazione.

Con questa architettura, aggiungere una nuova schermata al gioco richiede di creare una nuova terna MVC, registrare il nuovo stato nella macchina a stati e aggiungere il controller al Service Locator. Il Game Loop non necessita di alcuna modifica, garantendo un'estensione modulare del sistema senza impatti sulle componenti esistenti.

L'architettura garantisce inoltre la completa sostituibilità della View senza impatti su Controller e Model. Le View non mantengono riferimenti al Model, ricevendolo solo come parametro per il rendering, e i Controller interagiscono con esse esclusivamente tramite l'interfaccia comune. I Model sono completamente agnostici rispetto alla tecnologia

grafica. Per transitare ad una diversa libreria grafica sarebbe sufficiente reimplementare le classi View e adattare la gestione della finestra nel Game Loop; Controller e Model rimarrebbero invariati.

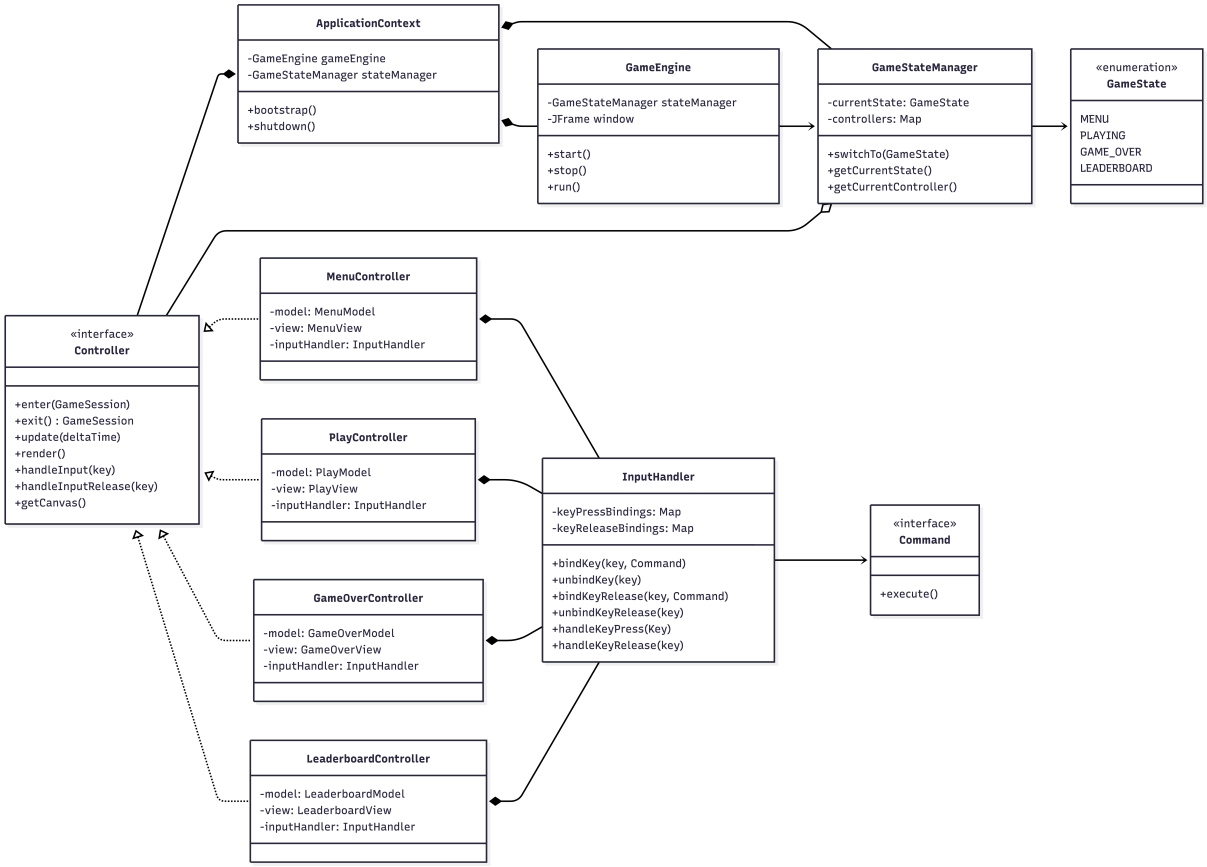

Figura 2.1: Diagramma UML dell'architettura del sistema

2.2 Design dettagliato

2.2.1 Selezione dei pezzi

Problema

Il gioco deve supportare diverse modalità di selezione del prossimo tetromino. La versione classica di Tetris usa una selezione puramente casuale, mentre le versioni moderne adottano il sistema "7-bag randomizer" che garantisce una distribuzione equa dei pezzi. La scelta della modalità deve essere configurabile senza modificare il codice.

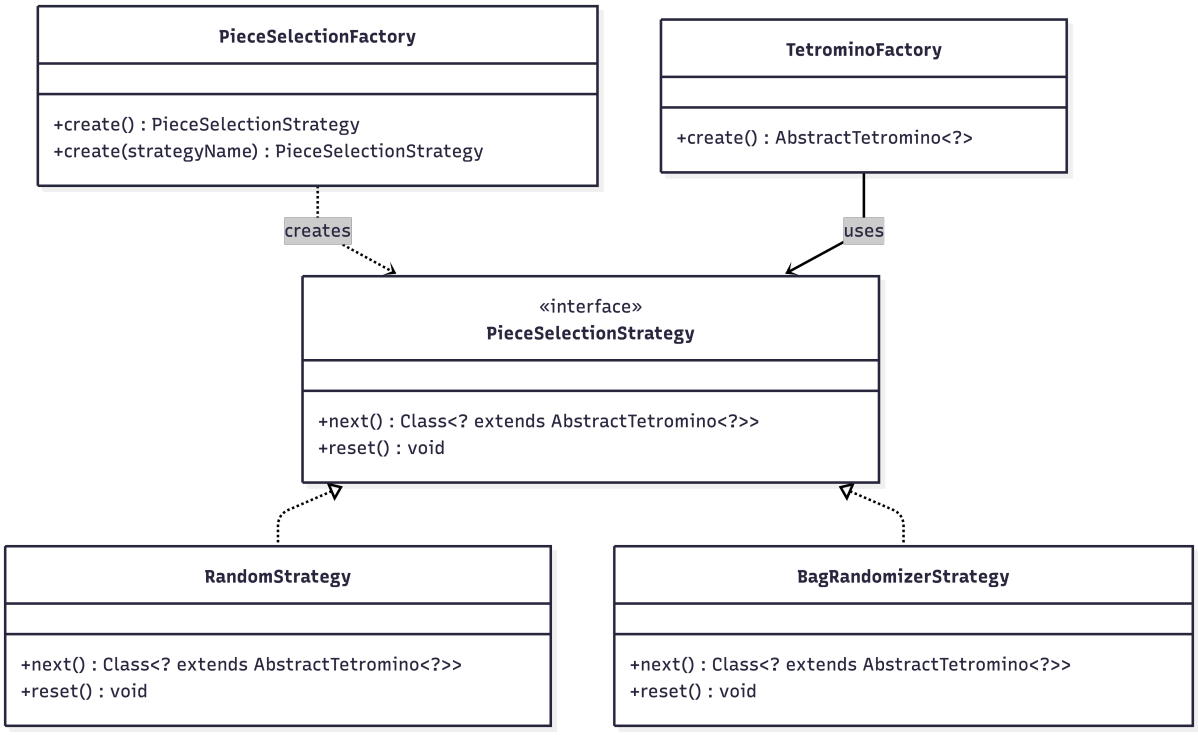

Figura 2.2: Diagramma UML del pattern Strategy per la selezione dei pezzi

Soluzione

Come da Figura 2.2 si è adottata una combinazione di **Strategy** [8] e **Simple Factory** [9]. L'interfaccia `PieceSelectionStrategy` definisce il contratto per la selezione: `next()` restituisce il prossimo tipo di tetromino, `reset()` reinizializza lo stato. Mentre le implementazioni concrete (`RandomStrategy`, `BagRandomizerStrategy`) incapsulano i diversi algoritmi. Infine `PieceSelectionFactory`, in `create()`, legge la configurazione e istanzia la strategia appropriata.

La soluzione risulta estensibile (è possibile inserire nuove strategie senza modificare codice esistente), testabile (strategie isolabili), e configurabile a runtime. Al contrario di alternative basate su If/Else/Switch-case o con l'utilizzo di enum.

2.2.2 Rendering delle View

Problema

Le diverse schermate del gioco (menu, partita, game over, classifica) condividono la stessa logica di inizializzazione del canvas e del buffer strategy, ma differiscono nel contenuto da disegnare. Duplicare questa logica in ogni view violerebbe il principio DRY e renderebbe difficile la manutenzione.

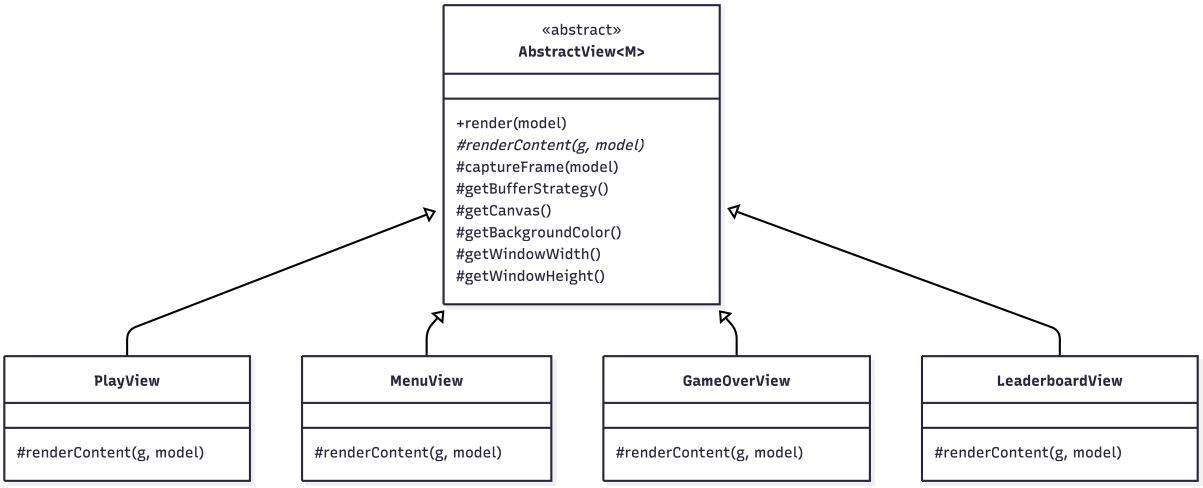

Figura 2.3: Diagramma UML del pattern Template Method per il rendering delle View

Soluzione

Come da Figura 2.3 si è adottato il pattern **Template Method** [10]. `AbstractView<M>` definisce il metodo template `render(M model)` che gestisce l'inizializzazione del buffer strategy per poi delegare a `renderContent(Graphics2D g, M model)`. Le sottoclassi (`PlayView`, `MenuView`, `GameOverView`, `LeaderboardView`) implementano solo `renderContent(Graphics g, M model)` con la propria logica di disegno; la classe base fornisce anche getter protetti per risorse comuni (font, dimensioni, colori).

La soluzione elimina duplicazione e garantisce un'inizializzazione corretta e consistente, lasciando le sottoclassi focalizzate solo sul proprio disegno. Rispetto a una soluzione basata su composizione, risulta meno flessibile in scenari più complessi.

2.2.3 Tetromini: gerarchia e creazione

Problema

I sette tetromini condividono comportamenti comuni (movimento, rotazione, posizione) ma differiscono per forma, colore e stati di rotazione. Serve una struttura che eviti duplicazione, garantisca copie type-safe, centralizzi la creazione e permetta estensioni future.

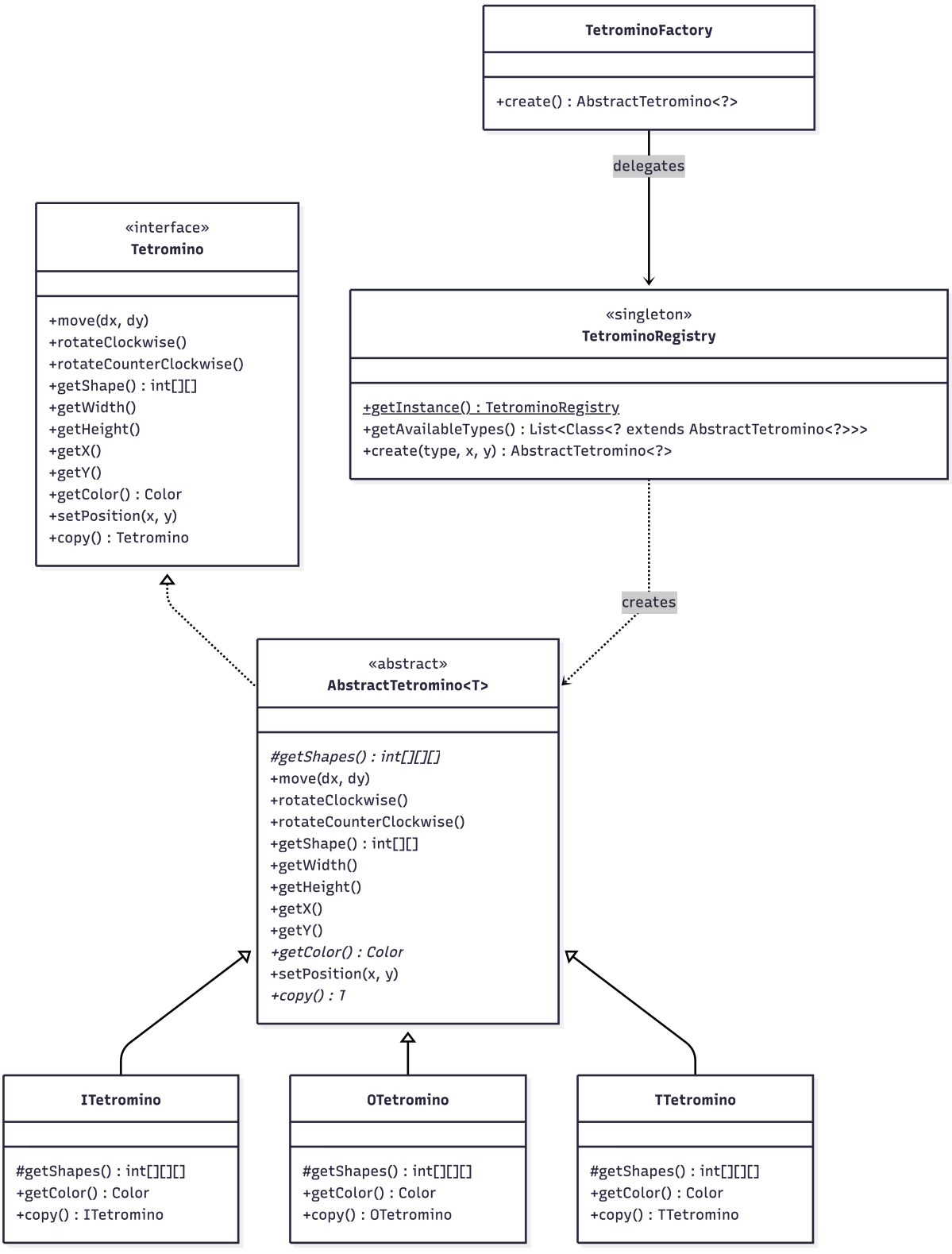

Figura 2.4: Diagramma UML della gerarchia dei tetromini e del sistema di creazione (per semplicità sono mostrati solo 3 dei 7 tetromini)

Soluzione

Come da Figura 2.4 si è strutturata una gerarchia a tre livelli combinata con un sistema di creazione centralizzato.

L'interfaccia `Tetromino` definisce il contratto pubblico. La classe astratta `AbstractTetromino<T extends AbstractTetromino<T>>` implementa i comportamenti comuni usando **F-bounded Polymorphism** [11]: il parametro di tipo ricorsivo garantisce che `copy()` restituisca il tipo concreto invece di un generico `Tetromino`. I tetromini concreti (`ITetromino`, `OTetromino`, `TTetromino`, ecc.) estendono la classe astratta implementando solo i metodi specifici: `getShapes()`, `getColor()`, `copy()`.

Per la creazione, `TetrominoRegistry` implementa il pattern **Registry** [12] (come **Singleton** [13]) mantenendo una mappa che associa ogni classe di tetromino alla sua funzione di creazione. `TetrominoFactory` usa la strategia di selezione per ottenere il tipo da creare e delega al registry per l'inizializzazione effettiva.

La soluzione garantisce type-safety a compile-time, codice DRY ed estensibilità: aggiungere un nuovo tetromino richiede solo di creare la classe e registrarla. Al contrario, alternative senza generici avrebbero richiesto cast espliciti, mentre uno Switch-case sulla classe avrebbe violato l'Open/Closed Principle.

2.2.4 Gestione degli stati di gioco

Problema

Il gioco è composto da più schermate (menu principale, partita, game over, classifica) con transizioni vincolate: ad esempio, dalla partita si può passare solo a game over o al menu, mai direttamente alla classifica. È necessario gestire queste transizioni in modo centralizzato, validandole per evitare stati incoerenti, e propagare le informazioni rilevanti da uno stato al successivo senza accoppiare i controller tra loro.

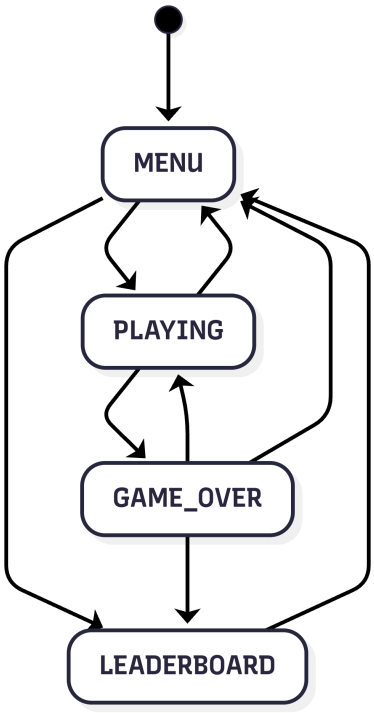

Figura 2.5: Diagramma degli stati e delle transizioni valide nel gioco

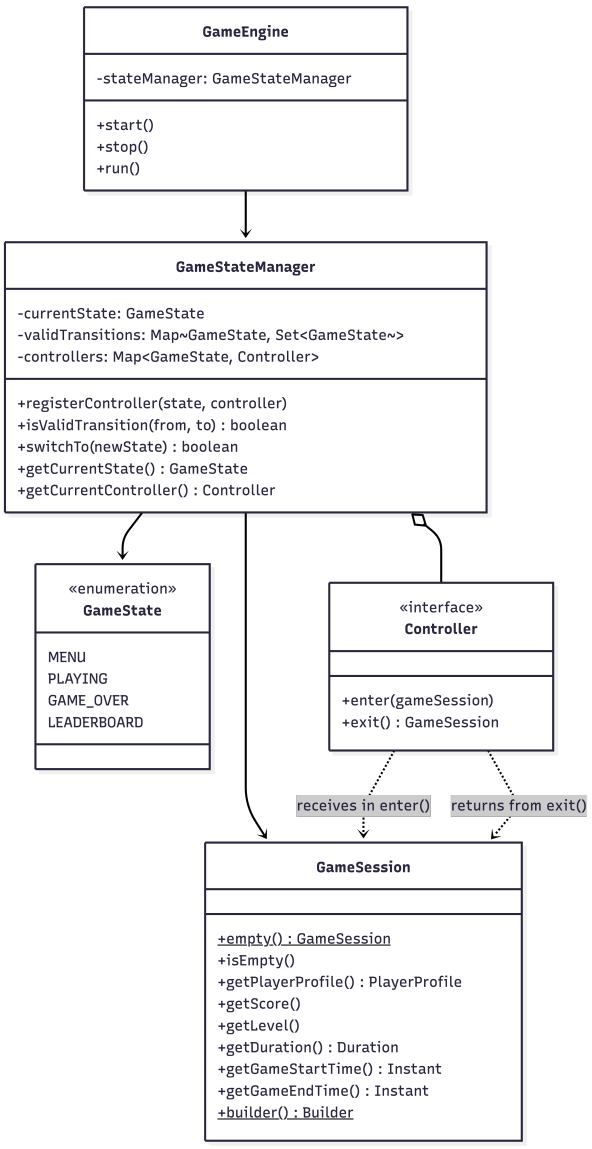

Figura 2.6: Diagramma UML della macchina a stati finiti per la gestione delle schermate

Soluzione

Come da Figura 2.5 e Figura 2.6, si è adottata una **macchina a stati finiti** (FSM) [4] per governare le transizioni tra le schermate. La classe `GameStateManager` mantiene lo stato corrente tramite l'enum `GameState` e una mappa di transizioni valide (`EnumMap<GameState, Set<GameState>>`) inizializzata nel costruttore. Il metodo `switchTo(GameState)` verifica la validità della transizione prima di effettuarla: se non è valida, la rifiuta e logga un warning; altrimenti invoca `exit()` sul controller uscente — che restituisce una `GameSession` con i dati della partita — e `enter(GameSession)` sul controller entrante, propagando il contesto senza che i due controller si conoscano.

`GameSession` è un oggetto immutabile (con copie difensive per le immagini) che funge da **data transfer object** (DTO) tra gli stati: contiene punteggio, livello, linee completate, ultimo frame renderizzato e tempi di gioco. L'uso di un Builder semplifica la costruzione nelle varie transizioni. Il `GameEngine`, nel game loop, interroga il `GameStateManager` per rilevare cambiamenti di stato e aggiornare il canvas nella finestra.

Questa soluzione centralizza la logica di transizione in un unico punto, rendendo impossibili transizioni non previste e disaccoppiando completamente i controller tra loro. Aggiungere un nuovo stato richiede solo di estendere l'enum, definire le transizioni valide e registrare il controller associato, senza modificare il `GameEngine` né gli altri controller.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Il progetto utilizza diverse librerie per garantire la qualità del codice attraverso test automatizzati:

- **JUnit 5**: framework principale per la scrittura e l'esecuzione dei test unitari.
- **JUnit Jupiter Params**: estensione di JUnit per la creazione di test parametrizzati, che consente di eseguire lo stesso test con diversi input riducendo la duplicazione del codice.
- **Mockito**: libreria per la creazione di mock objects, utilizzata per isolare le unità sotto test dalle loro dipendenze.
- **JaCoCo**: strumento per la misurazione della code coverage, integrato nel processo di build per generare report sulla percentuale di codice effettivamente esercitata dai test.

I test si concentrano sulle componenti che contengono la logica di gioco più critica e sulle classi che orchestrano il comportamento di altre entità.

Il cuore del testing riguarda il **model**: i tetromini vengono verificati nella loro creazione, rotazione e posizionamento, con test parametrizzati che coprono tutti e sette i tipi di pezzi. La **Board** è testata per la validazione delle posizioni, il rilevamento delle collisioni e l'eliminazione delle linee complete. Le strategie di selezione dei pezzi sono verificate nelle loro invarianti — ad esempio, che il bag randomizer distribuisca tutti i pezzi prima di ripetere.

Per quanto riguarda la **persistenza**, la **Leaderboard** è testata sia nella logica di ordinamento e limite delle voci, sia nel funzionamento dei diversi provider di storage (JSON su file e Redis), utilizzando Mockito per isolare i test dalle dipendenze esterne.

Infine le classi **GameStateManager** e **GameSession** sono testate per verificare le transizioni tra gli stati del gioco e la corretta gestione del ciclo di vita della partita.

3.2 Note di sviluppo

Utilizzo di Log4j2 per il logging
Permalink: https://github.com/xpicio/OOP24-tetraj/blob/273676fb34af5e40427604d965e7eca11c81e7b8/src/main/java/it/unibo/tetraj/GameStateManager.java#L17

Utilizzo di Jackson per serializzazione e deserializzazione JSON
Permalink: https://github.com/xpicio/OOP24-tetraj/blob/273676fb34af5e40427604d965e7eca11c81e7b8/src/main/java/it/unibo/tetraj/model/leaderboard/JsonFileStorageProvider.java#L119-L146

Utilizzo di Jedis per connessione a Redis
Permalink: https://github.com/xpicio/OOP24-tetraj/blob/273676fb34af5e40427604d965e7eca11c81e7b8/src/main/java/it/unibo/tetraj/model/leaderboard/RedisStorageProvider.java#L70-L92

Utilizzo di JUnit 5 con test parametrizzati (@MethodSource)
Permalink: https://github.com/xpicio/OOP24-tetraj/blob/273676fb34af5e40427604d965e7eca11c81e7b8/src/test/java/it/unibo/tetraj/model/piece/TetrominoTest.java#L201

Utilizzo di JUnit 5 con test parametrizzati (@CsvSource)
Permalink: https://github.com/xpicio/OOP24-tetraj/blob/273676fb34af5e40427604d965e7eca11c81e7b8/src/test/java/it/unibo/tetraj/model/PlayModelTest.java#L661

Utilizzo di Mockito per mock objects
Permalink: https://github.com/xpicio/OOP24-tetraj/blob/273676fb34af5e40427604d965e7eca11c81e7b8/src/test/java/it/unibo/tetraj/model/leaderboard/RedisStorageProviderTest.java#L51-L69

F-bounded polymorphism per copie type-safe
Permalink: https://github.com/xpicio/OOP24-tetraj/blob/273676fb34af5e40427604d965e7eca11c81e7b8/src/main/java/it/unibo/tetraj/model/piece/AbstractTetromino.java#L10

Strutture generiche complesse innestate
Permalink: https://github.com/xpicio/OOP24-tetraj/blob/273676fb34af5e40427604d965e7eca11c81e7b8/src/main/java/it/unibo/tetraj/model/piece/TetrominoRegistry.java#L14-L16

TypeReference di Jackson per preservare tipi generici a runtime
Permalink: https://github.com/xpicio/OOP24-tetraj/blob/273676fb34af5e40427604d965e7eca11c81e7b8/src/main/java/it/unibo/tetraj/model/leaderboard/JsonFileStorageProvider.java#L33-L36

Inizializzazione lazy thread-safe con ConcurrentHashMap.computeIfAbsent()
Permalink: https://github.com/xpicio/OOP24-tetraj/blob/273676fb34af5e40427604d965e7eca11c81e7b8/src/main/java/it/unibo/tetraj/util/LoggerFactory.java#L50

Shutdown hook con watchdog thread per cleanup risorse
Permalink: https://github.com/xpicio/OOP24-tetraj/blob/273676fb34af5e40427604d965e7eca11c81e7b8/src/main/java/it/unibo/tetraj/ApplicationContext.java#L176-L201

Utilizzo di lambda expressions
Permalink: https://github.com/xpicio/OOP24-tetraj/blob/273676fb34af5e40427604d965e7eca11c81e7b8/src/main/java/it/unibo/tetraj/model/piece/TetrominoRegistry.java#L20-L31

Utilizzo di method references
Permalink: https://github.com/xpicio/OOP24-tetraj/blob/273676fb34af5e40427604d965e7eca11c81e7b8/src/main/java/it/unibo/tetraj/model/PlayModel.java#L189

Utilizzo di Stream
Permalink: https://github.com/xpicio/OOP24-tetraj/blob/273676fb34af5e40427604d965e7eca11c81e7b8/src/main/java/it/unibo/tetraj/model/leaderboard/RedisStorageProvider.java#L175-L179

Chaining null-safe con Optional
Permalink: https://github.com/xpicio/OOP24-tetraj/blob/273676fb34af5e40427604d965e7eca11c81e7b8/src/main/java/it/unibo/tetraj/model/PlayModel.java#L240

Utilizzo di functional interfaces (Consumer, BiFunction, Runnable) come parametri
Permalink: https://github.com/xpicio/OOP24-tetraj/blob/273676fb34af5e40427604d965e7eca11c81e7b8/src/main/java/it/unibo/tetraj/model/PlayModel.java#L325-L338

Costruzione di oggetti immutabili complessi
Permalink: https://github.com/xpicio/OOP24-tetraj/blob/273676fb34af5e40427604d965e7eca11c81e7b8/src/main/java/it/unibo/tetraj/controller/PlayController.java#L61-L69

Codice riadattato

Un aspetto che sicuramente ha semplificato notevolmente la gestione dei pezzi e del wall kick è stato l'utilizzo di matrici tridimensionali con stati di rotazione pre-calcolati. Invece di computare le rotazioni matematicamente a runtime tramite trasformazioni geometriche, ogni tetromino definisce esplicitamente i suoi 4 stati rotazionali come layer di una matrice 3D:

```
/** The 4 rotation states of the T piece. */
private static final int[][][] SHAPES = {
  {{0, 1, 0}, {1, 1, 1}, {0, 0, 0}},
  {{0, 1, 0}, {0, 1, 1}, {0, 1, 0}},
  {{0, 0, 0}, {1, 1, 1}, {0, 1, 0}},
  {{0, 1, 0}, {1, 1, 0}, {0, 1, 0}},
};
```

Questo approccio, ispirato al Super Rotation System (SRS) definito nelle Tetris Guideline [14], è lo standard de facto nelle implementazioni del gioco, come documentato anche nella libreria Tetris Gymnasium [15]. La rotazione diventa un semplice cambio di indice nella matrice, eliminando calcoli trigonometrici e casi speciali per pezzi asimmetrici.

Anche le formule per la velocità di caduta dei pezzi sono riprese da fonti ufficiali: la `ClassicSpeedStrategy` utilizza la tabella frames-per-row del Tetris NES (NTSC, 60 FPS), mentre la `ModernSpeedStrategy` implementa la formula delle Tetris Guideline: $(0.8 - ((level - 1) \times 0.007))^{level-1}$.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Complessivamente il progetto ha raggiunto gli obiettivi che mi ero prefissato: un clone funzionante di Tetris con un'architettura pulita e manutenibile. Ci sono tuttavia alcuni aspetti dell'implementazione che, col senno di poi, avrei affrontato diversamente.

Gestione della GameSession. La classe `GameSession` è stata introdotta in una fase avanzata dello sviluppo, quando l'architettura dei controller era già consolidata. Il suo scopo è propagare le informazioni di una partita (punteggio, livello, ultimo frame) tra i diversi stati del gioco, ad esempio da `PLAYING` a `GAME_OVER`. La soluzione funziona, ma risulta un po' rigida: ogni controller deve esplicitamente costruire e restituire una `GameSession` nel metodo `exit()`, e riceverla nel metodo `enter()`. Un approccio alternativo, progettato fin dall'inizio, avrebbe potuto essere un sistema ad eventi: i controller pubblicano eventi (es. `GameEndedEvent`) e gli interessati si sottoscrivono, disaccoppiando completamente produttori e consumatori di informazioni.

Inizializzazione statica dei controller. Nell'`ApplicationContext`, i controller vengono istanziati e registrati manualmente nel metodo `setupGameComponents()`. Ogni nuova coppia stato-controller richiede modifiche in questo punto. Una soluzione più flessibile avrebbe potuto sfruttare annotazioni custom (es. `@GameState(GameState.MENU)`) e reflection per scoprire automaticamente i controller e associarli ai rispettivi stati nel `GameStateManager`, seguendo un approccio simile a quello di framework come Spring.

Accoppiamento dei colori con AWT. Nei model dei tetromini il colore è rappresentato direttamente con `java.awt.Color`, creando una dipendenza dal framework grafico. Per una maggiore portabilità sarebbe stato preferibile utilizzare un enum o un record `Color(int r, int g, int b)`, lasciando alla view il compito di convertirlo nel tipo specifico del framework.

Gestione del nickname. Nei requisiti iniziali era previsto che il nickname del giocatore fosse richiesto tramite input. Per semplificare l'implementazione e ridurre i tempi di sviluppo, ho optato per la generazione automatica del nickname e la sua persistenza in un file nella home directory dell'utente.

Sviluppi futuri. Se dovessi portare avanti il progetto, oltre a risolvere i punti sopra citati, aggiungerei:

- Schermata delle opzioni per configurare modalità di gioco e impostazioni a runtime
- Personalizzazione grafica dei tetromini e dello sfondo
- Ottimizzazione delle prestazioni e dell'utilizzo della memoria

4.2 Difficoltà incontrate e commenti per i docenti

Premetto che la mia situazione non è quella dello studente "tipo": sono fuori corso da diversi anni e lavoro a tempo pieno. Questa condizione ha rappresentato la difficoltà principale nello svolgimento del progetto, non tanto per complessità tecniche o lacune nella preparazione, quanto per l'impossibilità di garantire un impegno continuativo nel tempo.

Il lavoro frammentato ha reso più complesso mantenere il filo logico del progetto: a volte le sessioni di sviluppo si concentravano nel fine settimana, seguite da un'intera settimana senza poter toccare il codice. Riprendere il lavoro dopo giorni di pausa significava ogni volta ricostruire mentalmente il contesto: dove ero arrivato, quali decisioni avevo preso e perché, quali erano i prossimi passi. Questo overhead ha inevitabilmente rallentato lo sviluppo e, in alcuni casi, portato a riscrivere parti di codice che con un lavoro più continuo sarebbero state affrontate diversamente fin dall'inizio.

Non intendo con questo cercare giustificazioni o attenuanti: la responsabilità di non aver completato il percorso universitario nei tempi previsti è esclusivamente mia. Tuttavia, ritengo utile condividere questa esperienza perché rappresenta una realtà concreta di chi, per scelte di vita o necessità, si trova a conciliare studio e lavoro. Il corso e il progetto sono stati comunque un'occasione preziosa per consolidare competenze, e per approfondire aspetti della programmazione ad oggetti che nella pratica lavorativa spesso si danno per scontati o si dimentica di applicare.

Appendice A

Guida utente

A.1 Avvio del gioco

Per avviare Tetraj è necessario avere Java 21 (o versione successiva) [16] installato sul proprio sistema. Il gioco si avvia eseguendo il file `tetraj.jar` con il comando:

```
java -jar tetraj.jar
```

In alternativa, è possibile scaricare l'ultima versione ed eseguire direttamente il gioco con un singolo comando.

macOS / Linux (zsh/bash):

```
curl -L https://tinyurl.com/tetrajar -o /tmp/tetraj.jar && \
  java -jar /tmp/tetraj.jar
```

Windows (PowerShell):

```
Invoke-WebRequest -Uri "https://tinyurl.com/tetrajar" ‘
  -OutFile "$env:TEMP\tetraj.jar"; java -jar "$env:TEMP\tetraj.jar"
```

A.2 Schermata iniziale

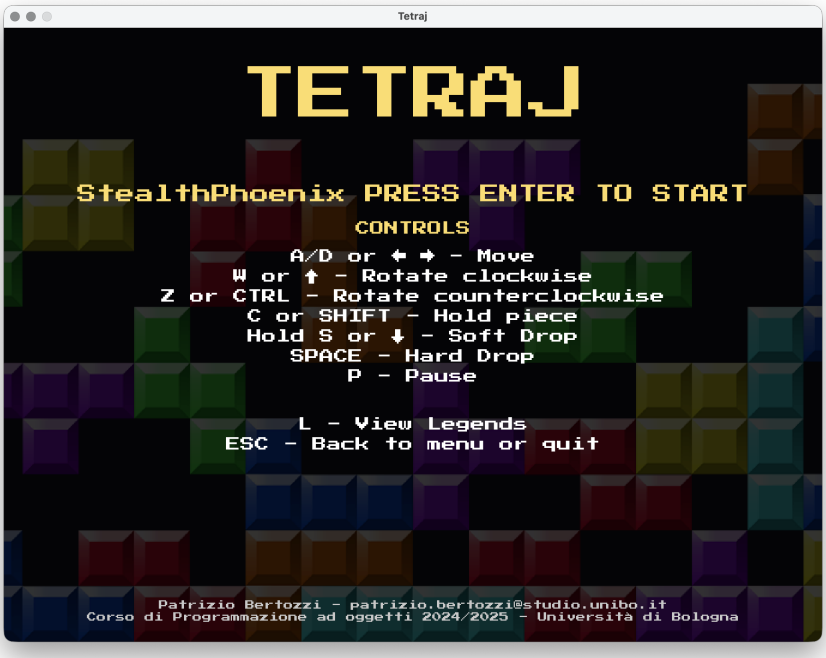

Figura A.1: Schermata iniziale del gioco

All'avvio viene mostrato il menu principale. Da qui è possibile:

- Premere **INVIO** per iniziare una nuova partita
- Premere **L** per visualizzare la classifica dei punteggi
- Premere **ESC** per uscire dal gioco

A.3 Durante la partita

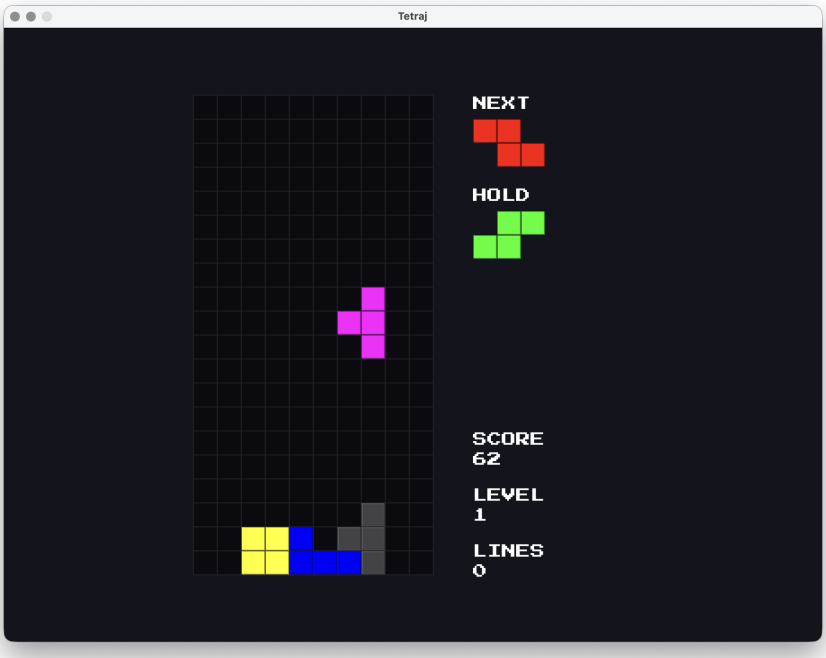

Figura A.2: Schermata di gioco

Una volta avviata la partita, sullo schermo sono visibili: l'area di gioco centrale dove cadono i pezzi, il prossimo pezzo in arrivo, l'eventuale pezzo messo da parte, il punteggio, il livello e il numero di linee completate.

I controlli disponibili sono:

- **Freccia sinistra** o **A**: sposta il pezzo a sinistra
- **Freccia destra** o **D**: sposta il pezzo a destra
- **Freccia giù** o **S**: accelera la caduta del pezzo (soft drop)
- **Barra spaziatrice**: fa cadere istantaneamente il pezzo (hard drop)
- **Freccia su** o **W**: ruota il pezzo in senso orario
- **CTRL** o **Z**: ruota il pezzo in senso antiorario
- **SHIFT** o **C**: mette da parte il pezzo corrente per usarlo dopo
- **P**: mette in pausa la partita
- **ESC**: torna al menu principale

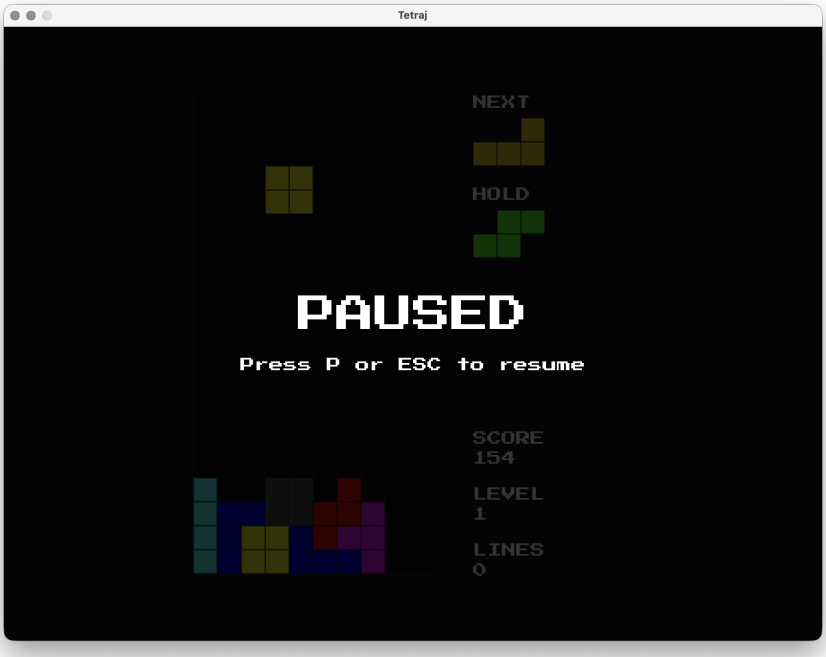

Figura A.3: Schermata di gioco in pausa

Durante la pausa, premere **P** o **ESC** per riprendere a giocare.

A.4 Punteggio

Il punteggio aumenta completando linee orizzontali. Più linee si completano contemporaneamente, più punti si ottengono:

- 1 linea: 100 punti
- 2 linee: 300 punti
- 3 linee: 500 punti
- 4 linee (Tetris): 800 punti

I punti vengono moltiplicati per il livello corrente. Inoltre, il soft drop assegna 1 punto bonus per ogni cella percorsa, mentre l'hard drop ne assegna 2 per ogni cella di altezza da cui cade il pezzo.

Il livello aumenta ogni 10 linee completate, rendendo la caduta dei pezzi progressivamente più veloce.

A.5 Fine partita e classifica

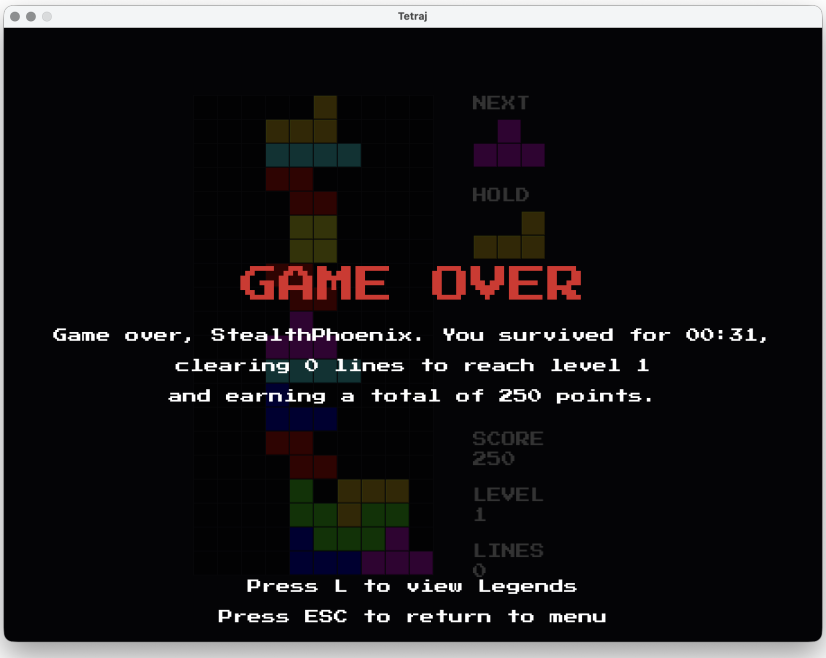

Figura A.4: Schermata di fine della partita

La partita termina quando non c'è più spazio per posizionare un nuovo pezzo. Viene mostrata la schermata di fine partita con il punteggio ottenuto. Da qui è possibile:

- Premere **INVIO** per iniziare una nuova partita
- Premere **L** per visualizzare la classifica
- Premere **ESC** per tornare al menu principale

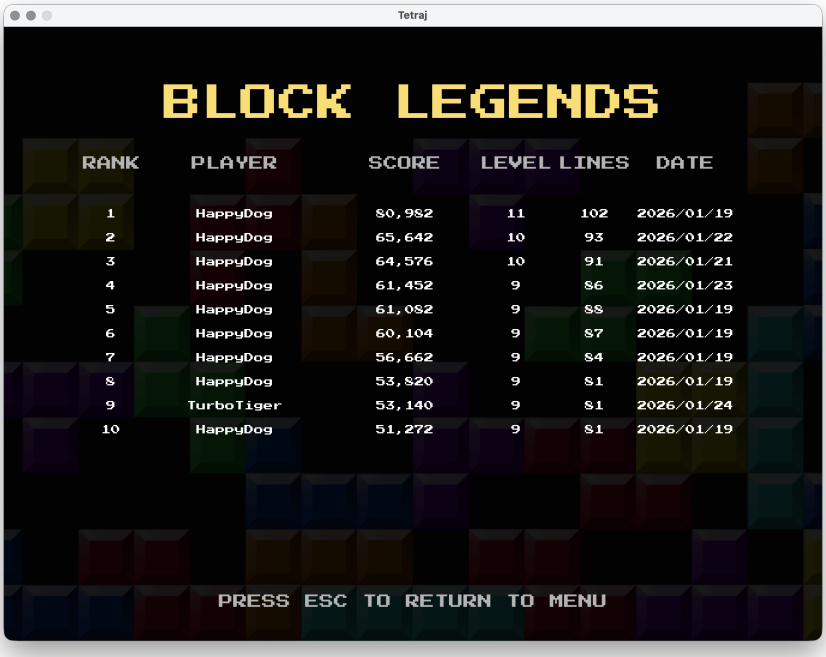

Figura A.5: Schermata con i dieci punteggi migliori "Block legends"

Dalla schermata della classifica, premere **ESC** per tornare al menu.

Appendice B

Esercitazioni di laboratorio

patrizio.bertozzi@studio.unibo.it

- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=177162#p246015
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=179154#p247849
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=180101#p248934
- Laboratorio 11: https://virtuale.unibo.it/mod/forum/discuss.php?d=181206#p251421

Bibliografia

[1] TetrisWiki, *Enciclopedia della community dedicata al gioco Tetris*, https://tetris.wiki/Tetris.wiki

[2] TetrisWiki, *Tetromino – Descrizione delle forme e dei colori ufficiali dei sette tetromini*, https://tetris.wiki/Tetromino

[3] Wikipedia, *Pattern Model–view–controller*, https://en.wikipedia.org/wiki/Model-view-controller

[4] Wikipedia, *Pattern Finite-state machine*, https://en.wikipedia.org/wiki/Finite-state_machine

[5] Wikipedia, *Pattern Service locator*, https://en.wikipedia.org/wiki/Service_locator_pattern

[6] Wikipedia, *Game loop*, https://en.wikipedia.org/wiki/Game_loop

[7] Wikipedia, *Pattern Command*, https://en.wikipedia.org/wiki/Command_pattern

[8] Wikipedia, *Pattern Strategy*, https://en.wikipedia.org/wiki/Strategy_pattern

[9] Wikipedia, *Pattern Factory*, https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)

[10] Wikipedia, *Pattern Template Method*, https://en.wikipedia.org/wiki/Template_method_pattern

[11] Wikipedia, *Generici – Bounded quantification, F-bounded quantification*, https://en.wikipedia.org/wiki/Bounded_quantification#F-bounded_quantification

[12] Martin Fowler, *Pattern Registry*, https://martinfowler.com/eaaCatalog/registry.html

[13] Wikipedia, *Pattern Singleton*, https://en.wikipedia.org/wiki/Singleton_pattern

[14] TetrisWiki, *Tetromino – Super Rotation System*, https://tetris.wiki/Super_Rotation_System

[15] Tetris Gymnasium, *Tetromino – Documentazione delle forme e rotazioni*, https://max-we.github.io/Tetris-Gymnasium/components/tetromino/

[16] Eclipse Adoptium, *Temurin JDK 21 – Pagina di download*, https://adoptium.net/temurin/releases?version=21