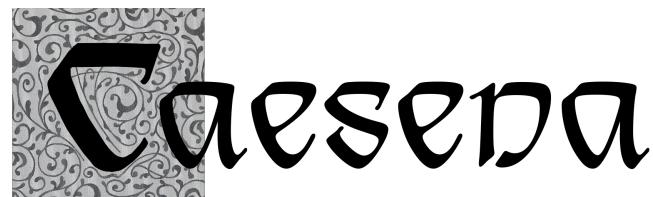


Caesena

Progetto di "Programmazione ad Oggetti"
Caesena



**Mauro Pellanara, Alessandro Martini, Davide
Speziali, Samuele Giancarli**

Programmazione ad Oggetti

Università di Bologna

09/04/2023

Indice

1 Analisi	2
1.1 Requisiti	2
1.2 Analisi e modello del dominio	3
2 Design	5
2.1 Architettura	5
2.2 Design dettagliato	7
3 Sviluppo	22
3.1 Testing automatizzato	22
3.2 Metodologia di lavoro	23
3.3 Note di sviluppo	27
4 Commenti finali	28
4.1 Autovalutazione e lavori futuri	28
5 Guida utente	30
6 Esercitazioni di laboratorio	32
Riferimenti bibliografici	34

1 Analisi

1.1 Requisiti

Il gruppo si pone come obiettivo quello di realizzare la trasposizione video-ludica del gioco da tavolo Carcassonne [1]. Si tratta di un gioco da 2 a 6 giocatori dove ognuno deve posizionare delle tessere che formano strutture; ovvero strade, praterie, città e monasteri. Su queste strutture possono essere posti dei seguaci, o pedine, con lo scopo di realizzare punti. Chi alla fine della partita avrà totalizzato più punti verrà proclamato vincitore.

Requisiti funzionali

Il gioco dovrà permettere ai giocatori che ne partecipano, ognuno identificato univocamente da nome e colore, di piazzare e ruotare tessere ed eventualmente posizionare un seguace. E' necessario:

- Permettere il posizionamento delle tessere solo in posizioni adiacenti a tessere già piazzate con la quale combaciano i lati
- Permettere il posizionamento di seguaci su strutture appartenenti alla tessera appena piazzata e senza altri seguaci
- Calcolare il punteggio dei giocatori, sia nel momento in cui viene chiusa una struttura, che alla fine della partita
- Permettere la navigazione del tabellone di gioco
- Permettere di scartare una tessera che non possa essere piazzata in nessuna delle posizioni libere
- Permettere di ruotare la tessera prima di piazzarla sul tabellone di gioco
- Permettere la selezione del numero dei giocatori e assegnare un nome ed un colore ad ognuno
- Gestire l'ampliamento e l'unione di praterie, città e strade

Requisiti non funzionali

- Menù di pausa
- Visualizzazione grafica e dinamica dei seguaci rimanenti

- Pannello per la scelta del colore del giocatore
- Decorazioni, sfondi e distanziamento fra i vari componenti grafici
- Supporto di espandibilità futura per l’implementazione di immagini di profilo dei giocatori
- Supporto alla lingua italiana e inglese
- Supporto per partecipare alla stessa partita da più interfacce
- Supporto per future espansioni con l’aggiunta di nuove tessere e seguaci

1.2 Analisi e modello del dominio

L’analisi e il modello del dominio di Caesena possono essere suddivisi in diversi componenti chiave: i giocatori, il tabellone di gioco, le tessere, i seguaci e le strutture.

Il tabellone di gioco si occupa di tenere traccia delle tessere posizionate, delle strutture costruite e dei seguaci piazzati su di esse. Posizionando un seguace su una struttura, il giocatore ne prende la proprietà, guadagnandone poi i punti alla sua chiusura. A fine partita ogni proprietario di una struttura ne guadagna i punti anche se non chiusa, con il caso speciale delle città nel quale ne guadagna la metà. Una struttura è chiusa quando:

- nel caso delle città, questa non abbia parti non murate. Fa guadagnare al proprietario 2 punti per ogni tessera all’interno della città. Nel caso una tessera contenga uno stemma, allora varrà il doppio;
- nel caso del monastero, questo sia circondato da 8 tessere. Fa guadagnare al proprietario 1 punto per ogni tessera dalla quale è circondato, più 1 punto per se stesso;
- nel caso della strada, sia presente un incrocio o un’entrata ad un monastero/città da ambo i lati. Fa guadagnare al proprietario 1 punto per ogni tessera, considerando anche gli estremi.

Nel caso della prateria, questa non può essere chiusa e fa guadagnare punti al proprietario solo a fine partita, esattamente 3 per ogni città chiusa confinante. Siccome i seguaci rimangono sulle strutture fino alla loro chiusura, questo comporta che un seguace posizionato su una prateria ne rimarrà all’interno fino alla fine.

È possibile che due strutture in cui sono piazzati due o più seguaci (anche dello stesso colore) vengano unite tramite una tessera di congiunzione. In questo caso alla chiusura della struttura il giocatore che avrà la maggioranza di seguaci otterrà i punti vincendo su tutti gli altri. In caso di pareggio ogni giocatore otterrà la totalità dei punti.

In fase di analisi le difficoltà primarie che abbiamo individuato sono:

- rappresentare il concetto di tessera coerentemente con la sua rappresentazione grafica;
- rappresentare il concetto di struttura;
- unire strutture dello stesso tipo;
- fare il conteggio dei punti rispettando le regole del gioco.

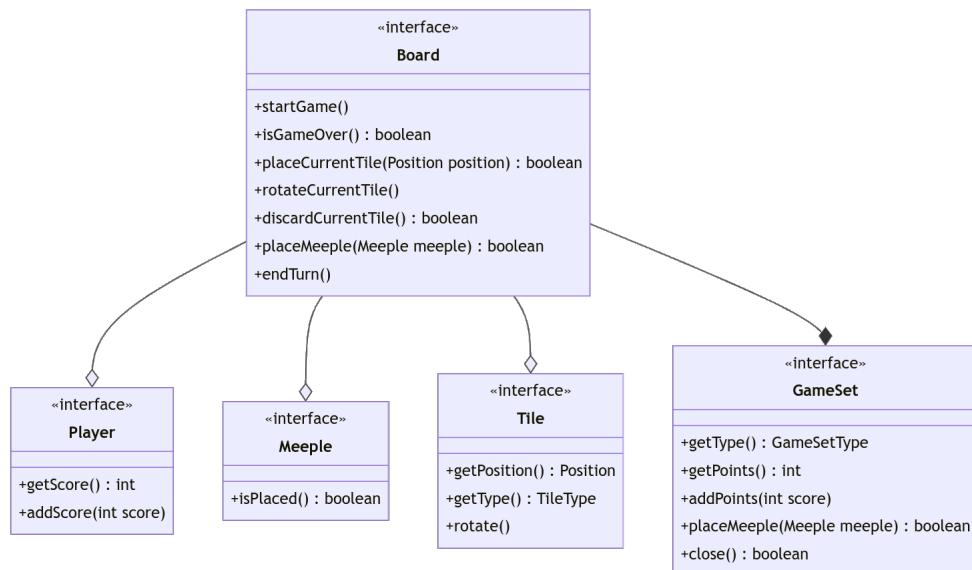


Figura 1: Schema UML dell'analisi del problema.

2 Design

2.1 Architettura

In fase di progettazione è stato scelto di adottare il pattern architettonico MVC, traendone quindi tutti i suoi vantaggi, principalmente che model e controller rimangono praticamente uguali a prescindere dalla view utilizzata. Infatti, per la view è stato scelto di usare il pattern Strategy con delle interfacce generiche per i singoli componenti che potranno perciò adottare qualsiasi GUI framework. Per passare per esempio da Swing a JavaFX basterebbe solo sostituire le implementazioni delle interfacce presenti nella parte di view.

Ogni modulo del pattern Model-View-Controller ha quindi una propria organizzazione e scopo:

- Nella parte di model sono state definite le entità che modellano: tessere (tile), seguaci (meeple), strutture (gameset) e giocatori (player). Esse vengono instanziate e gestite dal controller.
- Nella parte di view è presente un componente principale, la UserInterface, che si occupa di gestire tutti i componenti grafici e di renderli visibili all'utente quando necessario. Quest'ultima si occupa anche di fornire ai vari componenti, ognuno con uno scopo ben preciso, l'accesso regolarizzato alla parte di controller. Nella maggior parte dei casi questi componenti sono annidati l'uno dentro l'altro.
- La parte di controller si occupa della gestione della partita e di propagare sul model i risultati delle azioni compiute dall'utente attraverso la view.

Siccome ad ogni modifica di stato della partita, il Controller notifica le UserInterface collegate ad esso, è possibile giocare alla stessa partita contemporaneamente tramite varie interfacce utente aggiungendole tutte allo stesso controller. Queste possono essere diverse tra loro e di qualsiasi tipo, per esempio: Web, CLI o come nel nostro caso, GUI.

Per la configurazione iniziale della partita viene utilizzato un file JSON. All'interno di esso vengono memorizzate informazioni riguardo quali e quante tile generare all'inizio della partita e quale deve essere la tile iniziale.

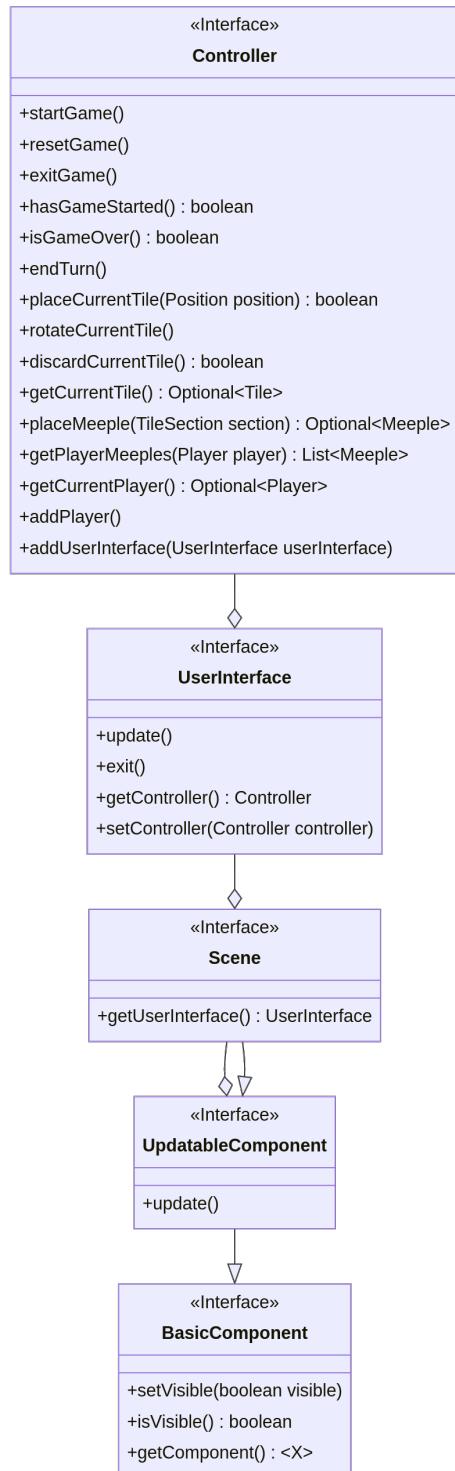


Figura 2: Schema UML architetturale.

2.2 Design dettagliato

Mauro Pellonara

Permettere di sviluppare Meeple con diverse caratteristiche

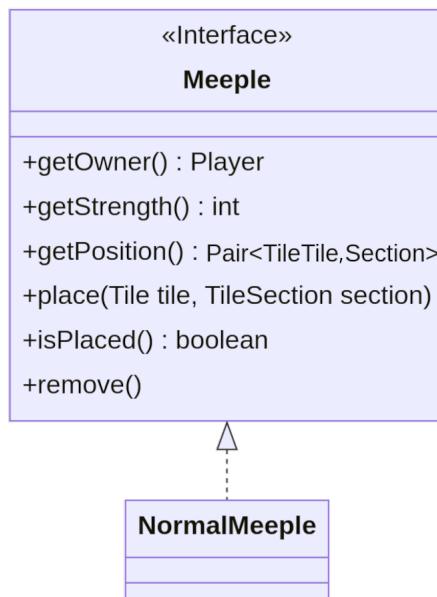


Figura 3: Rappresentazione UML dell'applicazione del pattern Strategy per Meeple e NormalMeeple.

Problema La versione classica del gioco supporta solo seguaci normali che non hanno funzioni speciali. Alcune espansioni introducono invece dei seguaci che valgono il doppio di quelli normali, è quindi necessario garantire espandibilità futura in caso di nuove espansioni.

Soluzione Usare il pattern Strategy sulla classe NormalMeeple introducendo un'interfaccia Meeples che può essere usata in futuro per introdurre nuovi seguaci di tipo diverso, un esempio sono quelli che hanno valore doppio rispetto ai normali.

Facilitare la creazione di istanze di MutableTile assieme ai relativi GameSet

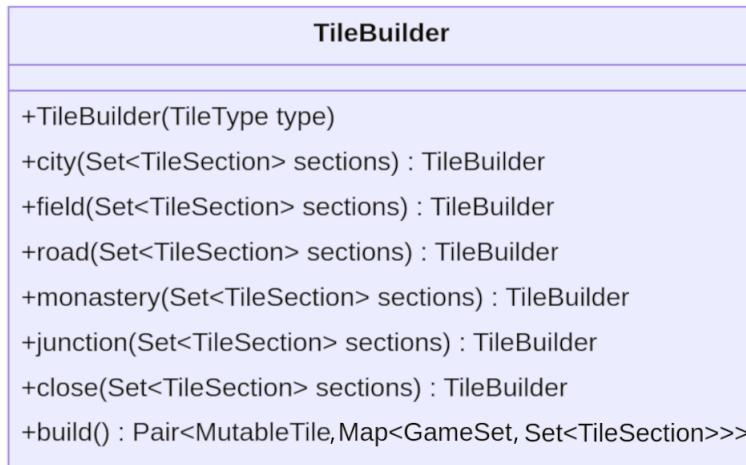


Figura 4: Rappresentazione UML dell'applicazione del pattern Builder per MutableTile.

Problema Creare istanze di MutableTile assieme ai relativi GameSet in modo facile e leggibile.

Soluzione Usare il pattern Builder per la creazione di oggetti MutableTile rendendo più esplicite le istruzioni da eseguire. Assieme alla creazione di un'istanza di MutableTile possono anche essere creati i relativi GameSet e restituiti tutti assieme con il metodo build().

Facilitare la creazione di istanze di MutableTile di tipo diverso

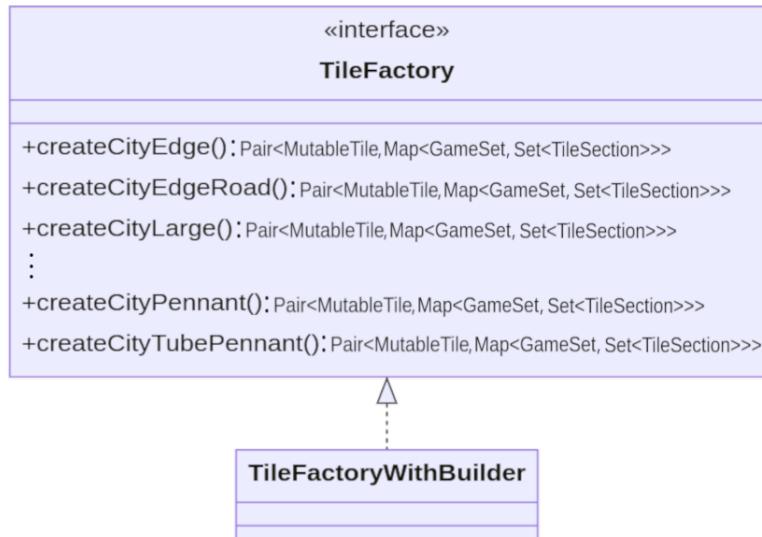


Figura 5: Rappresentazione UML dell'applicazione del pattern Factory Method per MutableTile.

Problema All'inizio di ogni partita vengono generate numerose MutableTile per ogni tipo rispettando il file di configurazione, è quindi necessario che per ogni tipo di Tile ci sia una procedura facile e ripetibile per la generazione.

Soluzione Usare il pattern Factory Method includendo nella classe TileFactoryWithBuilder, per ogni tipo di Tile, un metodo per la creazione di MutableTile di quel tipo. In aggiunta, è stato usato anche il pattern Strategy aggiungendo un'interfaccia per le Factory di MutableTile, in questo modo ogni Factory può impiegare algoritmi e strategie diverse per conseguire lo stesso risultato. In questo caso è stata sviluppata una factory che usasse la classe TileBuilder per la creazione di MutableTile.

Evitare di avere riferimenti doppi tra GameSet e Tile

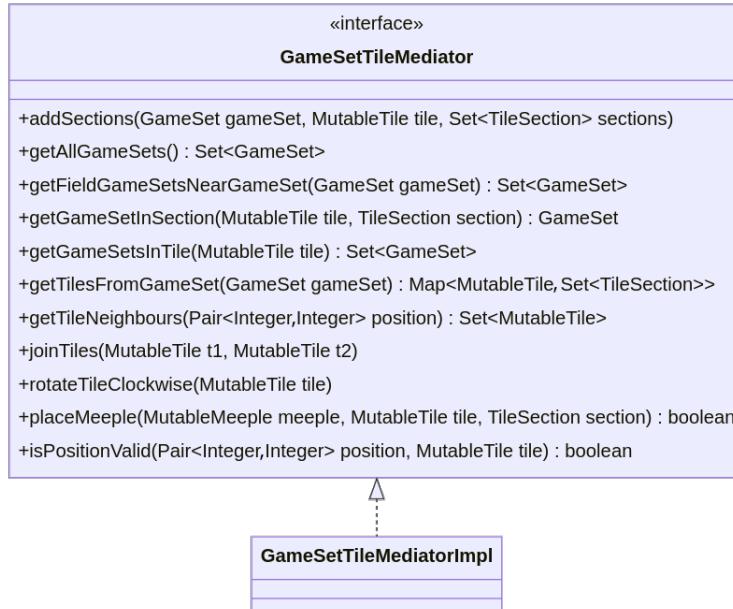


Figura 6: Rappresentazione UML dell'applicazione del pattern Mediator per GameSet e Tile.

Problema Ogni singola Tile alla sua creazione ha su di essa diversi GameSet. Questi però non esistono solo sulle singole Tile ma riguardano il tabellone di gioco in generale, in quanto, con l'avanzamento progressivo della partita, vengono estesi ed uniti tra loro. Perciò, ogni Tile contiene dei GameSet e ogni GameSet contiene delle Tile, serve quindi un modo di gestire questo legame.

Soluzione Usare il pattern Mediator per creare una classe che si occupi del legame tra GameSet e Tile, permettendo quindi di eseguire operazioni come ottenere i GameSet presenti in una Tile o ottenere le Tile sulla quale si estende un certo GameSet. Quindi, usando questo pattern non sarà necessario tenere e sincronizzare dei riferimenti doppi tra GameSet e Tile. In aggiunta, è stato usato anche il pattern Strategy aggiungendo un'interfaccia per il Mediator, in questo modo ognuno può impiegare algoritmi e strategie diverse per conseguire lo stesso risultato.

Permettere diversi tipi di campi di input per la creazione di Player

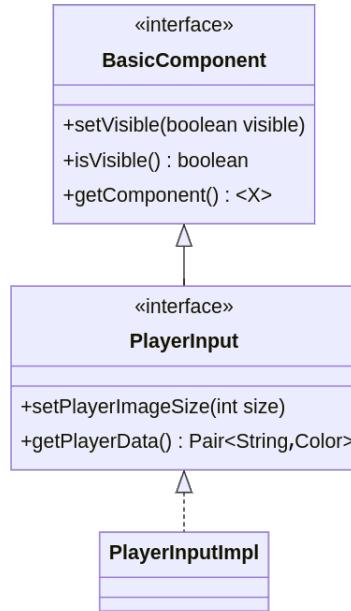


Figura 7: Rappresentazione UML dell'applicazione del pattern Strategy per `PlayerInput`.

Problema Creare un componente grafico per l'inserimento delle informazioni necessarie alla creazione di un nuovo Player, rispettando quindi il Single Responsibility Principle. Inoltre, far sì che questo principio venga rispettato da qualsiasi view con interfaccia grafica.

Soluzione Creare una classe `PlayerInputImpl` che si occupi di ottenere e gestire le informazioni relative alla creazione di un nuovo Player. Ulteriormente, applicare il pattern Strategy alla classe `PlayerInputImpl` generalizzandone il comportamento e usando un tipo generico rappresentante il componente grafico utilizzato, che varia a seconda del GUI framework scelto.

Alessandro Martini

Definizione di operazioni che vengono utilizzati su GameSet differenti

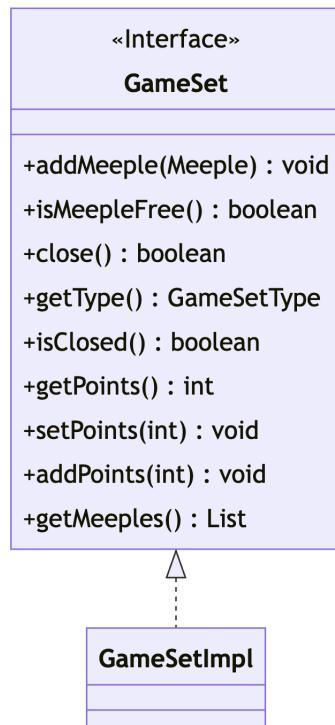


Figura 8: Rappresentazione UML dell'applicazione del pattern Strategy per GameSet e GameSetImpl

Problema: Assegnare logiche differenti per GameSet differenti. Si era posto il problema della scalabilità delle funzioni che ogni GameSet dovesse avere, anche per aggiornamenti futuri

Soluzione: Tramite il pattern Strategy, sono andato a creare un'interfaccia GameSet che definisse le funzioni generali di ogni GameSet, così facendo, ho reso scalabile anche per upgrade futuri la possibilità di creare nuove funzioni e nuove regole per i vari tipi di GameSet.

Creazione di più GameSet con tipologie differenti

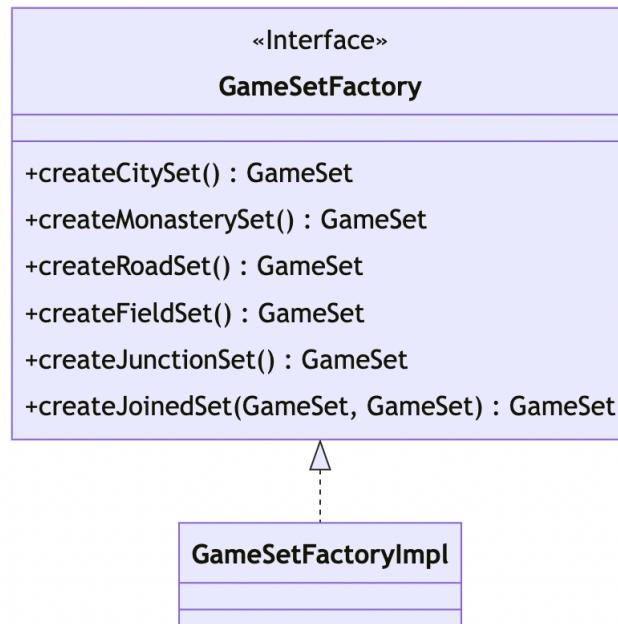


Figura 9: Rappresentazione UML dell'applicazione del pattern Strategy per il gamesetfactory e GameSetFactoryImpl

Problema: Creare una uguale implementazione di GameSet ma di tipi differenti

Soluzione: Tramite il pattern Strategy, siamo andati a implementare dei metodi che vanno a definire come i vari tipi di GameSet vengono creati, e inoltre come si possono creare dei GameSet da altri GameSet

Creazione di più BasicComponent con caratteristiche differenti

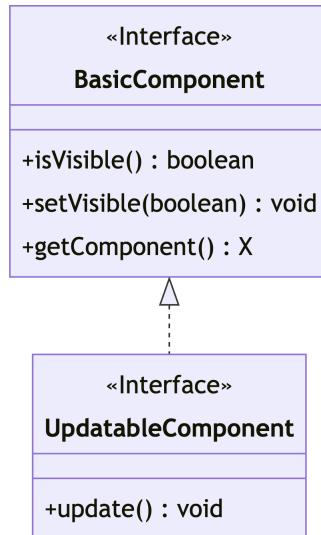


Figura 10: Rappresentazione UML dell'applicazione del pattern Strategy per la classe interfaccia BasicComponent, che viene estesa dall'interfaccia UpdatableComponent

Problema: Erano presenti dei componenti Grafici che, oltre ai metodi in comune con gli altri, avevano necessità di essere aggiornati durante il gioco

Soluzione: Sempre tramite il pattern Strategy, sono andato ad implementare i metodi che vanno a definire i comportamenti (metodi) in comune tra i vari componenti della view, e successivamente ho creato un'altra interfaccia che, estende BasicComponent, e inoltre va a definire il comportamento (metodo) di update del componente.

Davide Speziali

Associazione tra View e Model per quanto riguarda le immagini di tasselli e seguaci

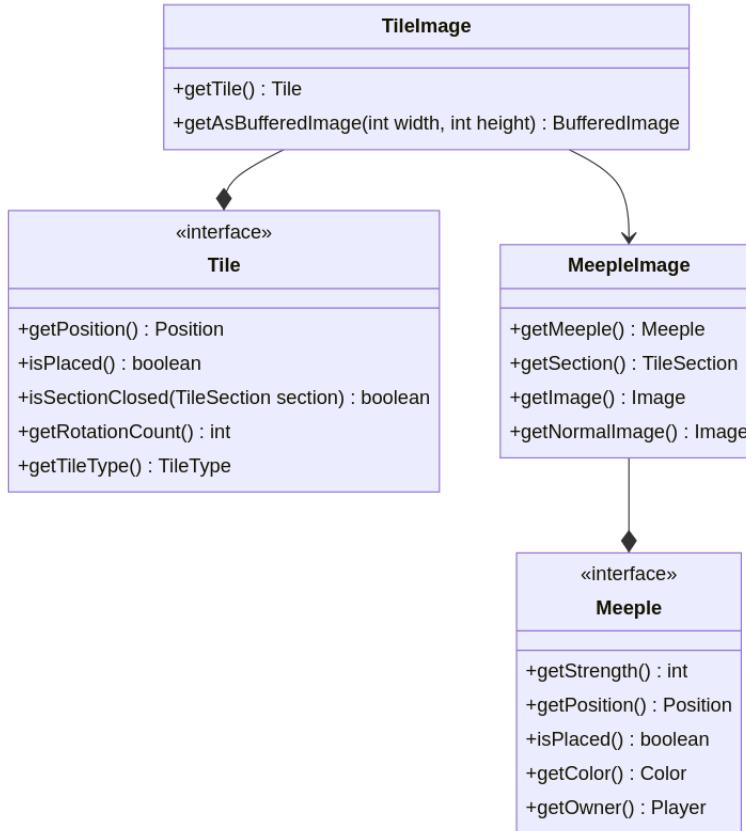


Figura 11: Rappresentazione UML della gestione delle immagini di tile e meeple

Problema: Necessità di avere delle imamgini aggiornate sulla base dello stato dei singoli tasselli e seguaci presenti nel gioco. Che devono inoltre essere presenti sia nel footer che nel tabellone.

Soluzione: Per quanto riguarda i taselli la soluzione trovata è stata quella di creare una classe **TileImage** che al suo interno contenesse una referenza ad oggetto della classe **Tile**. In questo modo viene fatta un'associazione diretta tra ogni singola istanza della classe **Tile** e la sua immagine, senza necessità che il model sia a conoscenza dell'esistenza di quest'ultima. All'interno di

TileImage viene generata un’immagine corrispondente alla tile che viene rigenerata ogni volta che avviene una modifica: rotazione, piazzamento di meeple o rimozione di meeple. Queste ultime due modifiche vengono gestite attraverso la classe MeepleImage che TileImage usa per ottenere l’overlay dei seguaci da applicare alle immagini dei tasselli. La classe MeepleImage è gestita in maniera analoga a TileImage, contiene e si associa in maniera diretta a un’istanza della classe Meeple. Questo sistema, oltre a garantire la gestione dinamica delle immagini, ci permette di attuare una forma di caching, non essendo necessario generare una nuova immagine ogni qual volta un tassello viene girato o un seguace viene piazzato su di esso. Questa soluzione garantisce che sia MeepleImage che TileImage vengano utilizzati anche indipendentemente.

Permettere al giocatore l’utilizzo del tabellone di gioco

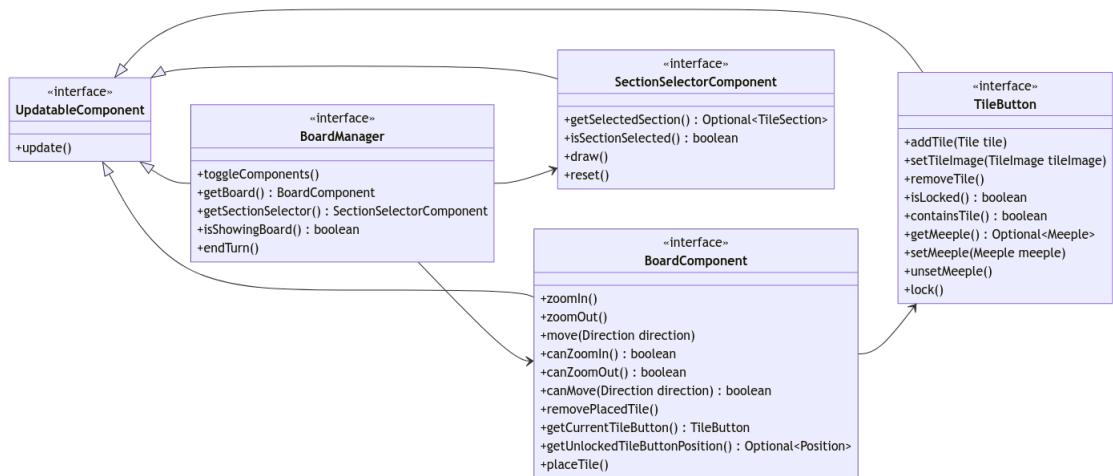


Figura 12: Rappresentazione UML della gestione dei componenti grafici del tabellone di gioco

Problema: È necessaria la creazione di una GUI in grado di permettere al giocatore di:

- Posizionare tessere
- Posizionare seguaci su tessere appena piazzate

Soluzione: Per garantire il *single responsibility principle* la soluzione trovata è stata quella di creare 4 componenti:

- TileButton, rappresenta una posizione nella board, può quindi contenere tasselli e seguaci
- BoardComponent, contiene i vari TileButton
- SectionSelectorComponent, permette di selezionare dove piazzare esattamente i seguaci
- BoardManager, gestisce la visualizzazione alternata tra BoardComponent e SectionSelectorComponent

Inoltre per facilitare espandibilità e refactoring è stato utilizzato il pattern *Strategy* per definire le operazioni richieste a ogni componente. Nel mio caso in particolare è stato utilizzato per definire il comportamento di tutti e quattro i componenti sopracitati.

Condivisione di elementi del Model nella View

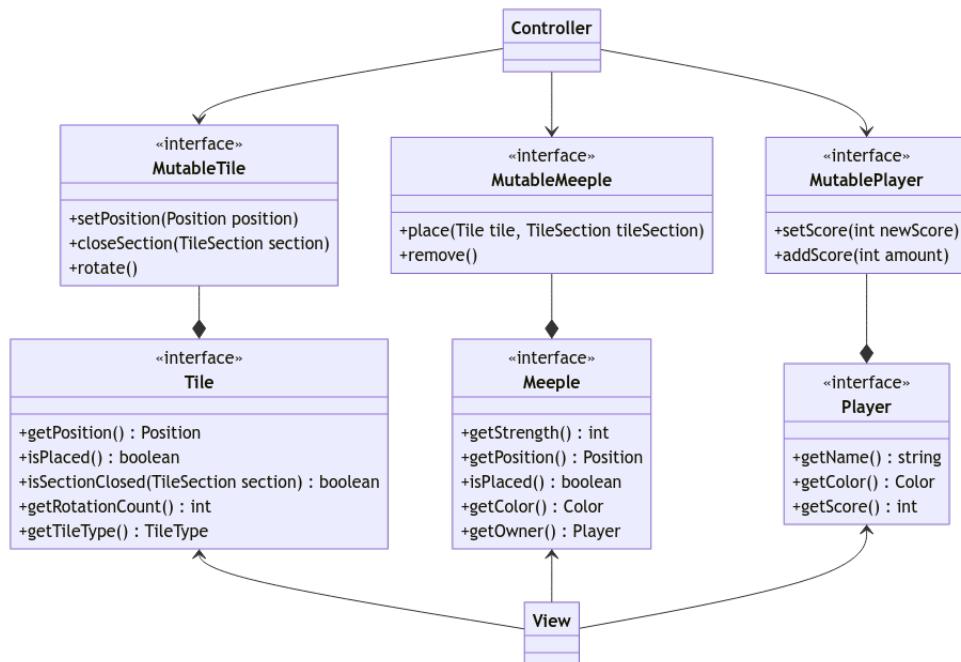


Figura 13: Rappresentazione UML dei componenti del Model nella gestione della mutabilità

Problema: Come gestire la necessità di avere riferimenti a componenti del Model all'interno della View, evitando di dare la possibilità a quest'ultima di apportare modifiche al Model.

Soluzione: La soluzione trovata è stata quella di utilizzare il concetto di *Immutability Through Interfaces*. Esso consiste nell'avere un'interfaccia, contenente soltanto metodi di richiesta di informazioni, che viene estesa dall'ulteriore interfaccia contenente anche metodi che comportano la modifica delle informazioni delle istanze. Le classi implementeranno direttamente questa seconda interfaccia mutabile, la prima interfaccia verrà solo utilizzata come tipo all'interno della View, in modo che quest'ultima abbia accesso ai dati aggiornati riguardo il Model ma allo stesso tempo non possa in nessun modo modificarli.

Questa tecnica è stata utilizzata per le interfacce:

- Meeple → MutableMeeple
- Player → MutablePlayer
- Tile → MutableTile

Samuele Giancarli

Permettere di sviluppare Tile con determinate peculiarità

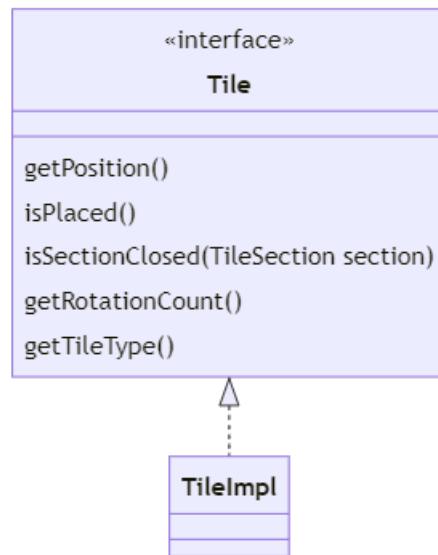


Figura 14: Rappresentazione UML dell'applicazione del pattern Strategy per Tile.

Problema: Nonostante le tile siano definite solamente dalle loro sezioni indipendenti (TileSection) ognuna definita da una tipologia di struttura (GameSet), è necessario garantire espandibilità considerando che futuri aggiornamenti potrebbero necessitare l'implementazione di nuove meccaniche di gioco.

Soluzione: utilizzo del Pattern Strategy per la rappresentazione e l'implementazione di nuove tessere.

Gestione della HUD

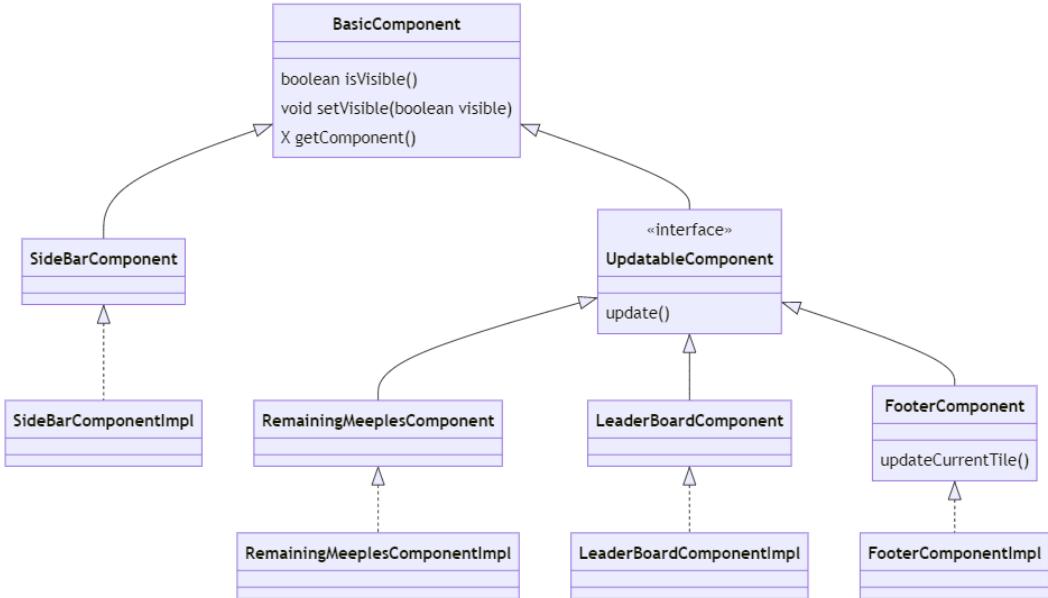


Figura 15: Rappresentazione UML dei componenti grafici della HUD

Problema: è necessaria la creazione di una HUD che permetta al giocatore di compiere azioni all'interno del tabellone e visualizzarne i dati di gioco

Soluzione: Per garantire il single responsibility sono stati creati 4 componenti principali:

- RemainingMeepleComponent per la visualizzazione dei propri seguaci rimanenti
- LeaderBoardComponent per la visualizzazione la classifica dei giocatori
- SideBarComponent per la gestione del tabellone di gioco
- FooterComponent per la visualizzazione dei dati di gioco

è stato utilizzato uno strategy pattern che vede un'interfaccia BasicComponent al suo vertice. Questa permette già l'implementazione di componenti con nessuna particolare necessità, come SideBarComponent: unicamente utilizzato per l'inserimento in input dei comandi da parte dei giocatori. I restanti componenti dovendo invece mostrare dati di gioco hanno necessitato

della funzione update(), ricavata dall’interfaccia UpdatableComponent che ha esteso BasicComponent. Da notare come LeaderBoardComponent, per esempio, sia un componente a se stante e quindi ottimo per essere utilizzato in altri contesti di view, per esempio come in gameOverScene.

3 Sviluppo

3.1 Testing automatizzato

Per effettuare i test necessari per verificare la correttezza delle funzionalità del nostro software abbiamo utilizzato la suite dedicata JUnit 5. I test che siamo andati ad implementare sono:

- GameSetTest: questa classe permette di testare la corretta creazione di vari GameSet (esempio: citySet, FieldSet), l'unione di due GameSet e il piazzamento di meeple all'interno dei GameSet. Sono presenti test anche per il controllo della corretta assegnazione e modifica dei punti relativi ai GameSet, è inclusa anche la verifica della gestione della corretta chiusura di un GameSet e dell'assegnazione dei punti ai Player aventi Meeple sul GameSet appena chiuso.
- NormalMeepleTest: questa classe permette di testare i metodi appartenenti al NormalMeeple e il posizionamento di esso all'interno di una Tile.
- PlayerTest: questa classe ci permette di testare i metodi appartenenti al Player e la modifica progressiva dei punti a lui assegnati.
- TileTest: questa classe ci permette di testare la corretta creazione di Tile e i metodi ad essa associata, più in specifico la chiusura di una TileSection e il suo scorrimento circolare all'interno della Tile.
- ToStringBuilderTest: questa classe ci permette di testare la classe ToStringBuilder, in particolar modo l'ottenimento di una stringa rappresentativa per un qualsiasi Object. Infine verifica la consistenza delle rappresentazioni letterali degli oggetti, garantendo che rimangano invariate.
- DirectionTest: questa classe ci permette di testare l'enum Direction e il suo metodo match che controlla che la differenza delle coordinate di due punti combaci con la direzione fornita.
- GameSetTileMediatorTest: questa classe ci permette di testare i vari metodi della classe GameSetTileMediator, per esempio la rotazione di una Tile con i GameSet ad essa associata, e la corretta corrispondenza tra i GameSet e le Tile con le proprie TileSection.

Per quanto riguarda la parte di view non sono stati scritti dei veri e propri test JUnit, ma si è ritenuto più opportuno testare manualmente le funzionalità principali:

- utilizzando schermi di diverse dimensioni e risoluzioni per verificare che il gioco funzionasse ugualmente per ognuno di essi e che il ridimensionamento fosse corretto.
- utilizzando variabili di debug per accedere direttamente ad una Scene da testare evitando tutte le altre Scene che secondo la logica di gioco sarebbero dovute essere state visualizzate prima.

3.2 Metodologia di lavoro

Ci siamo focalizzati molto sulla fase di progettazione in modo da avere interfacce ben definite una volta iniziata la fase di sviluppo. Sempre in questa fase, una parte che ha richiesto altrettanta attenzione è stata la ricerca di tecniche per effettuare una trasposizione informatica fedele al gioco da tavolo originale.

Inizialmente abbiamo suddiviso il lavoro in modo che ognuno potesse lavorare parallelamente. Ci sono stati però dei casi in cui è stato deciso, per efficienza e semplicità, che un componente del gruppo lavorasse su classi non di sua competenza, questo è avvenuto per due possibili ragioni:

- Individuazione e risoluzione tempestiva e continuativa di bug o errori di sorta.
- Implementazioni di tecniche o pattern da applicare nel medesimo modo in più classi e interfacce.

Per quanto riguarda il DVCS abbiamo deciso di utilizzare Git, in particolare abbiamo creato due diversi branch, uno per la stesura della relazione e l'altro per lo sviluppo del codice. Ogni componente ha quindi avuto la responsabilità di clonare localmente la repository e di fare le operazioni di *pull*, *push* ed eventualmente *merge*.

Mauro Pellonara

In autonomia mi sono occupato di:

- Interfaccia grafica principale (GUI in `it.unibo.caesenaview`)
- Menù principale e di pausa (StartScene e PauseScene in `it.unibo.caesenaview.scene`)
- Vari componenti con scopi generici (in `it.unibo.caesenaview.components.common`)

- Componenti relativi al player (PlayerInput e PlayerImage in `it.unibo.caesena.view.components.player`)
- Ridimensionamento dinamico di FooterComponentImpl (in `it.unibo.caesena.view.components`)
- "Caching" delle immagini (RemainingMeeplesComponent in `it.unibo.caesena.view.components.meeple`)
- Traduzione in italiano del gioco (LocaleHelper in `it.unibo.caesena.view`)
- Gestione di inizio e reset della partita (Controller in `it.unibo.caesena.controller`)
- Controllo della chiusura dei GameSet (Controller in `it.unibo.caesena.controller`)
- Gestione delle UserInterfaces (Controller in `it.unibo.caesena.controller`)
- Creazione di Tile (TileFactory e TileBuilder in `it.unibo.caesena.model.tile`)
- GameSetTileMediator con il relativo test (in `it.unibo.caesena.model`)
- Meeple con il relativo test (in `it.unibo.caesena.model.meeple`)

In collaborazione mi sono occupato di:

- Gestione della fine della partita con Alessandro Martini (Controller in `it.unibo.caesena.controller`)
- Tile e il relativo test con Samuele Giancarli (in `it.unibo.caesena.model.tile`)
- TileType con Davide Speziali (in `it.unibo.caesena.model.tile`)
- Gestione del dimensionamento della GUI, in particolare del font, con Davide Speziali
- Color con Davide Speziali (package `it.unibo.caesena.model`)
- Gestione delle immagini dei seguaci con Davide Speziali (package `it.unibo.caesena.view.components.meeple`)

Alessandro Martini

In autonomia mi sono occupato di:

- Implementazione e gestione dei vari GameSet (package it.unibo.caesena.model.gameset)
- Implementazione della GameOverScene (package it.unibo.caesena.view.scene)
- Implementazione BasicComponent (package it.unibo.caesena.view;)
- Implementazione UpdatableComponent (package it.unibo.caesena.view;)
- Implementazione Configuration Loader (package it.unibo.caesena.controller)
- Controller.endTurn() (package it.unibo.caesena.controller)
- Controller.discardCurrentTile (package it.unibo.caesena.controller)
- Controller.isCurrentTilePlaceable (package it.unibo.caesena.controller)
- Controller.getEmptyNeighbouringPositions (package it.unibo.caesena.controller)
- Controller.isPositionOccupied (package it.unibo.caesena.controller)

In collaborazione mi sono occupato di:

- Gestione della fine della partita con Mauro Pellonara (Controller in it.unibo.caesena.controller)

Davide Speziali

In autonomia mi sono occupato di:

- Gestire la parte di view riguardante la visualizzazione del campo di gioco (package it.unibo.caesena.view.components.board)
- Gestire la parte di view riguardante il piazzamento dei seguaci (package it.unibo.caesena.view.components.board)
- Gestire la parte di view riguardante cosa visualizzare tra board e section selector (package it.unibo.caesena.view.components.board)
- Gestire la parte di view riguardante la visualizzazione e piazzamento/-rimozione delle tessere (package it.unibo.caesena.view.components.tile)
- Avere un modo di creare stringhe standardizzate da utilizzare nei metodi `toString` dei vari oggetti del model utilizzando Reflection API (package it.unibo.caesena.utils)

- Player con relativo test (package it.unibo.caesena.model.player)
- Gestire delle immagini delle tessere (package it.unibo.caesena.view.components.tile)

In collaborazione mi sono occupato di:

- Gestione del dimensionamento della GUI, in particolare del font, con Mauro Pellonara
- Color con Mauro Pellonara (package it.unibo.caesena.model)
- Gestione delle immagini dei seguaci con Mauro Pellonara (package it.unibo.caesena.view.components.meeple)

Samuele Giancarli

In autonomia mi sono occupato di:

- Gestione della rotazione e del piazzamento tessere (package it.unibo.caesena.controller)
- Inserimento delle Tessere pennant in TileType (it.unibo.caesena.model.tile)
- Implementazione delle Factory per le tessere PENNANT in TileFactoryWithBuilder (it.unibo.caesena.model.tile)
- Implementazione del FooterComponent (package it.unibo.caesena.view.components;)
- Implementazione del SideBarComponent (package it.unibo.caesena.view.components)
- Implementazione del LeaderBoardComponent (package it.unibo.caesena.view.components.p)
- Implementazione del RemainingMeeplesComponent (package it.unibo.caesena.view.componen
- Implementazione del ResourceUtil (it.unibo.caesena.utils)
- Implementazione di TileSection (package it.unibo.caesena.model.tile)

In collaborazione mi sono occupato di:

- Sviluppo dell'intero package Tile e relativi test con Mauro Pellonara

3.3 Note di sviluppo

Mauro Pellonara

- Utilizzo di ResourceBundle e Locale per supporto di più lingue: [Permalink](#) dell'esempio
- Utilizzo di tipi generici: [Permalink](#) dell'esempio
- Utilizzo di Stream, Lambda e Method reference, alcuni esempi sono:
 - [Permalink](#) del 1° esempio
 - [Permalink](#) del 2° esempio
 - [Permalink](#) del 3° esempio
 - [Permalink](#) del 4° esempio
 - [Permalink](#) del 5° esempio

Alessandro Martini

- Utilizzo di Stream: sono presenti all'interno delle classi sviluppati porzioni di codice contenente il costrutto funzionale Stream: [Permalink](#)
- Uso di JSON Parser: utilizzo di un JSON parser per prendere tutte le tile presenti nel gioco (memorizzate nel file config.json) [Permalink](#)

Davide Speziali

- Uso di Reflection API, esempi:
 - [Permalink](#) del 1° esempio
 - [Permalink](#) del 2° esempio
- Utilizzo di Stream, lambda e method reference: [Permalink](#)
- Utilizzo di Optional, alcuni esempi:
 - [Permalink](#) del 1° esempio
 - [Permalink](#) del 2° esempio

Samuele Giancarli

- Utilizzo di stream e lambda: [Permalink](#)
- Esempio di codice leggibile e pulito: [Permalink](#)

4 Commenti finali

4.1 Autovalutazione e lavori futuri

Mauro Pellonara

Sono molto soddisfatto di questo progetto in quanto mi ha permesso di allenare le mie capacità di progettazione, astrazione e soprattutto programmazione. È stato molto motivante lavorare con gli altri colleghi facenti parte del gruppo, in quanto ci siamo spronati continuamente a cercare di scrivere codice sempre più pulito e corretto. Probabilmente la fase che ho apprezzato di più è stata quella di progettazione, iniziata con una partita al gioco da tavolo originale e conclusa spendendo numerose ore alla lavagna abbozzando il progetto. Devo però anche dire che in questa fase ci saremmo dovuti concentrare di più sul progettare la parte grafica, che invece è stata fatta inizialmente in modo confusionario e poi aggiustata e corretta. All'interno del gruppo, oltre a programmare, ho anche avuto il compito di organizzare e coinvolgere tutti i miei colleghi, promuovendo obiettivi parziali ogni settimana e cercando di distribuire al meglio il lavoro da fare. Avrei decisamente apprezzato una partecipazione più attiva e puntuale da parte di tutti i miei colleghi, cosa che invece non c'è stata fin dall'inizio e in alcuni casi nemmeno alla fine.

Alessandro Martini

Sono soddisfatto del mio lavoro all'interno del progetto, mi ritengo molto fortunato e orgoglioso per aver lavorato con colleghi che ritengo molto validi e collaborativi a questo progetto. Lo scambio di idee e il supporto reciproco sono stati cardine durante l'intero lavoro di progetto. Questo è stato il mio primo vero grande progetto scritto, e ho imparato l'utilità della comunicazione tra colleghi, ma soprattutto ho compreso l'utilità di scrivere codice pulito, riusabile all'interno di progetti, soprattutto per una corretta interpretazione futura di esso. Ritengo la parte grafica il tallone d'achille di questo progetto che, una volta terminato, cercheremo di migliorare.

Davide Speziali

Sono assolutamente soddisfatto del progetto e del risultato finale. Ha sicuramente permesso di farmi rendere conto delle mie reali capacità da programmatore e da designer del codice. Reputo però che non ci sia stato da tutti i componenti del gruppo l'impegno che andava riposto e questo ha secondo me tremendamente rallentato lo sviluppo e portato a un risultato finale che

potrebbe essere migliorato sotto molti aspetti. Anche durante la fase di progettazione l'impegno non è arrivato allo stesso modo da tutti i componenti, cosa che ha portato alla non totale soluzione di certi problemi, in particolar modo sulla progettazione della componente grafica, che è stata quasi del tutto ignorata inizialmente. Se non altro questi problemi mi hanno aiutato a capire quali sono i campanelli di allarme da notare nelle prime fasi di sviluppo.

Samuele Giancarli

Ho trovato estremamente gratificante questo lavoro in quanto mi ha mostrato di avere già le capacità e gli strumenti per poter sviluppare un'applicazione in java completamente funzionante, mettendo inoltre in pratica l'utilizzo di pattern e stili di programmazione appresi a lezione. Essendo l'elemento del team con la maggior consapevolezza del gioco originale ho avuto un importante ruolo nella sua progettazione iniziale, che mi ha stimolato in quanto ha necessitato di un grande lavoro di astrazione delle meccaniche di gioco da parte di tutto il gruppo. Ho trovato il gruppo di facile decisamente supportivo e valido. Probabilmente la possibilità di aggiungere altre tessere al gioco con annesse nuove meccaniche mi porterà ad aggiornare il programma quando sarà possibile, magari per poterlo anche utilizzare come portfolio personale. Mi sento di muovere una piccola critica sulla distribuzione dei compiti in fase di programmazione che sarebbe potuta essere più omogenea.

5 Guida utente

La schermata iniziale del gioco presenta la scelta del numero di giocatori, e per ciascuno, dei campi in cui inserirne il nome e il relativo colore. Dopo aver selezionato e impostato a piacimento le informazioni dei giocatori, si dovrà semplicemente cliccare sul pulsante "Inizia una nuova partita". A questo punto si avrà accesso alla schermata di gioco, che si presenta come in figura.

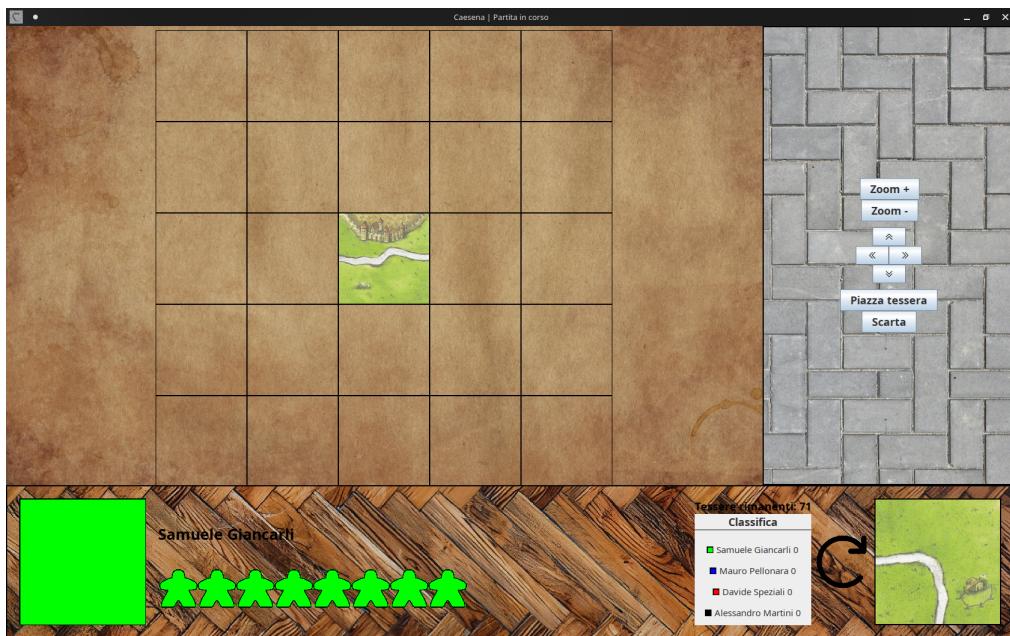


Figura 16: Schermata di gioco.

L'elemento principale del gioco è il tabellone che mostrerà al suo centro la tessera iniziale vicino alla quale dovranno essere piazzate le altre tessere. In basso a sinistra dello schermo vengono mostrati colore e nome del giocatore corrente oltre al numero dei suoi seguaci rimanenti, mentre alla destra si trovano la classifica dei giocatori e l'immagine della tessera corrente con il pulsante per ruotarla. A destra dello schermo si presentano invece i controlli del tabellone che ne gestiscono lo Zoom e ne permettono lo spostamento all'interno. Premendo il pulsante "Piazza tessera" questa verrà piazzata solo in caso sia stata selezionata una posizione nel tabellone. Premendo invece il pulsante "Scarta" verrà scartata la tessera solo se non piazzabile. Il pulsante "Piazza seguace" mostra invece l'apposita schermata che permette il piazzamento del seguace. Premendo "Finisci turno" si passa al turno successivo. Mentre si sta giocando è anche possibile aprire o chiudere il menù di pausa premendo ESC.

Alla fine della partita, verrà mostrata la classifica finale nella quale ogni giocatore vedrà il proprio punteggio conclusivo e avrà la possibilità di tornare alla schermata iniziale o di uscire dal gioco.

Il gioco supporta anche la lingua inglese nel caso il sistema operativo non utilizzi la lingua italiana, il testo dei pulsanti alla quale si riferisce la guida si traduce nel seguente modo:

- "Inizia una nuova partita" → "Start a new game"
- "Piazza tessera" → "Place tile"
- "Scarta" → "Discard"
- "Piazza seguace" → "Place meeple"
- "Finisci turno" → "End turn"

6 Esercitazioni di laboratorio

mauro.pellonara@studio.unibo.it

- Laboratorio 03: <https://virtuale.unibo.it/mod/forum/discuss.php?d=112846#p168591>
- Laboratorio 04: <https://virtuale.unibo.it/mod/forum/discuss.php?d=113869#p168941>
- Laboratorio 05: <https://virtuale.unibo.it/mod/forum/discuss.php?d=114647#p169516>
- Laboratorio 06: <https://virtuale.unibo.it/mod/forum/discuss.php?d=115548#p171293>
- Laboratorio 07: <https://virtuale.unibo.it/mod/forum/discuss.php?d=117044#p172650>
- Laboratorio 08: <https://virtuale.unibo.it/mod/forum/discuss.php?d=117852#p173721>
- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=118995#p175119>
- Laboratorio 10: <https://virtuale.unibo.it/mod/forum/discuss.php?d=119938#p175962>
- Laboratorio 11: <https://virtuale.unibo.it/mod/forum/discuss.php?d=121130#p177669>
- Laboratorio 12: <https://virtuale.unibo.it/mod/forum/discuss.php?d=121885#p178399>

alessandro.martini10@studio.unibo.it

- Laboratorio 03: <https://virtuale.unibo.it/mod/forum/discuss.php?d=112846#p168455>
- Laboratorio 05: <https://virtuale.unibo.it/mod/forum/discuss.php?d=114647#p169858>
- Laboratorio 07: <https://virtuale.unibo.it/mod/forum/discuss.php?d=117044#p172897>

- Laboratorio 08: <https://virtuale.unibo.it/mod/forum/discuss.php?d=117852#p174406>
- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=118995#p174952>
- Laboratorio 10: <https://virtuale.unibo.it/mod/forum/discuss.php?d=119938#p175992>

davide.speziali@studio.unibo.it

- Laboratorio 03: <https://virtuale.unibo.it/mod/forum/discuss.php?d=112846#p168604>
- Laboratorio 05: <https://virtuale.unibo.it/mod/forum/discuss.php?d=114647#p169876>
- Laboratorio 07: <https://virtuale.unibo.it/mod/forum/discuss.php?d=117044#p173007>
- Laboratorio 08: <https://virtuale.unibo.it/mod/forum/discuss.php?d=117852#p174111>
- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=118995#p174935>
- Laboratorio 10: <https://virtuale.unibo.it/mod/forum/discuss.php?d=119938#p175870>
- Laboratorio 11: <https://virtuale.unibo.it/mod/forum/discuss.php?d=121130#p177286>

samuele.giancarli@studio.unibo.it

Riferimenti bibliografici

- [1] Wikipedia. Informazioni dettagliate su Carcassonne. URL: <https://it.wikipedia.org/wiki/Carcassonne>.