Documentazione del progetto di ingegneria del software

Rota Gabriele [1079894]

Lorenzi Alessandro [1075244]

Rocco Alessandro [1081179]

Indice

1. Project Plan

https://github.com/alorenzi10/Monopoly/blob/main/documentazione/Project%20Plan.pdf

Differenze emerse a progetto concluso:

https://github.com/alorenzi10/Monopoly/wiki/Conclusioni-di-fine-progetto

2. Software lifecycle

Per lo sviluppo del progetto software abbiamo scelto di adottare il Rational Unified Process (RUP), adattato con pratiche agili per una maggiore flessibilità. Questa combinazione ci consente di mantenere una solida struttura metodologica, tipica di RUP, bilanciandola con la capacità di rispondere rapidamente ai cambiamenti e ottimizzare il lavoro in base al tempo a disposizione.
Il requisito principale del progetto è quello di ricreare il gioco Monopoly, che costituisce un obiettivo fisso e immutabile. Tuttavia, alcuni requisiti secondari (funzionalità aggiuntive o miglioramenti) saranno considerati opzionali e implementati solo se le tempistiche lo consentiranno, secondo MoSCoW e l'idea del CutOver di Rad.

La fase iniziale si è concentrata sul fissare obiettivi chiari: scopo del progetto, i suoi confini e i criteri di accettazione che verranno utilizzati per la valutazione del sistema. Abbiamo inoltre stimato i tempi e i rischi, consapevoli che una parte significativa delle difficoltà sarebbe derivata dalla nostra inesperienza nello sviluppo collaborativo e nella gestione di progetti complessi. Questo si è tradotto in una sottostima di alcuni rischi, come la difficoltà di sincronizzazione del lavoro di gruppo, la scrittura di codice di alta qualità secondo i requisiti non funzionali e la creazione delle interfacce grafiche.

Nella fase di elaborazione si analizza il dominio del problema e si mira all'ottenimento di un'architettura solida che farà da base per eventuali implementazioni aggiuntive future. Abbiamo studiato a fondo le regole del gioco identificando i vari attori, modellato il progetto con UML e deciso di applicare l'architettura MVC.
La fase di costruzione è il processo di fabbricazione. L'enfasi è sullo sviluppo dei componenti implementabili. I componenti sono stati integrati e al termine dello sviluppo sono stati testati. Al termine di questa fase era pronta la prima versione operativa del sistema, la versione *beta*.

Nella fase di transizione il sistema completato è sottoposto a beta test, coinvolgendo, oltre al team, anche amici che hanno giocato per individuare problematiche e fornire feedback

critici. Abbiamo analizzato e risolto bug segnalati, concentrandoci su stabilità, affidabilità e usabilità.

In attesa dell'esame orale la documentazione verrà aggiornata e completata, ed effettueremo piccole ottimizzazioni all'architettura e al codice per massimizzare la qualità del sistema.

3. People management and Team organization

Anche se avevamo un piano dettagliato da seguire per l'implementazione, nella pratica si è rivelato più una guida orientativa che un vincolo rigido, fungendo da faro piuttosto che da salvagente.

Abbiamo adottato un approccio misto, ispirandoci sia ai principi delle squadre agili che al modello del team SWAT.

Avevamo canali di comunicazione brevi e un atteggiamento orientato alle persone piuttosto che formalistico, spesso abbiamo lavorato in coppia (pilota e copilota). Abbiamo utilizzato strumenti collaborativi come workshop e sessioni di brainstorming per sviluppare idee, mentre il software è stato costruito attraverso iterazioni incrementali.

Quando era necessario implementare una nuova funzionalità non familiare, il processo seguiva un iter ben definito: inizialmente ciascun membro studiava la problematica individualmente, compatibilmente con i propri impegni in corso. Successivamente, organizzavamo riunioni in cui venivano presentate le proposte. Le idee venivano poi valutate collettivamente, e, insieme, si sceglieva la soluzione migliore da adottare.

Non c'è stata una rigida suddivisione dei compiti, ma ogni membro ha contribuito attivamente a tutte le fasi dello sviluppo software, con livelli di coinvolgimento variabili a seconda delle necessità e delle competenze individuali.

4. Configuration management

Tutta la documentazione ufficiale, così come il codice, è stata regolarmente salvata nel repository di GitHub.

Per le prime fasi del progetto, la documentazione in sviluppo e l'organizzazione del lavoro sono state gestite tramite una cartella su Google Drive.

Con il proseguire del progetto, la cartella su Drive è diventata principalmente un deposito di file, link utili e documentazione in fase di sviluppo.

Inizialmente, il lavoro è stato organizzato tramite issue, ma poiché il team era piccolo e la comunicazione tra i membri era frequente (tramite messaggi e chiamate), abbiamo notato che sarebbe stato più semplice dividere il lavoro usando fogli di Google e successivamente con la Kanban Board di GitHub.

La board ci ha permesso di visualizzare chiaramente lo stato delle attività, tenere traccia dei bug da risolvere e dei miglioramenti da implementare, senza la necessità di un sistema di issue formale.

La Kanban Board è divisa in 3 colonne:

- **ToDo**: sono i task che devono essere iniziati. Le attività in questa sezione provengono dai requisiti iniziali del progetto, da nuovi obiettivi definiti in corso d'opera, oppure da dubbi o malfunzionamenti emersi durante l'implementazione.
- **In Progress**: le attività che sono in fase di sviluppo.
- **Done**: i task conclusi e funzionanti.

Non abbiamo seguito uno standard rigido per i commit, ma ci siamo sforzati di mantenerli il più coerenti e descrittivi possibile. Ogni commit aveva un titolo e una descrizione che riassumevano le modifiche, anche se, spesso, controllavamo direttamente le modifiche nel codice. La nostra attenzione principale era quella di mantenere i commit piccoli e facilmente comprensibili.

All'inizio, avevamo poca esperienza con Git e la gestione dei branch, quindi li abbiamo usati con cautela, creando il primo branch per ripulire il codice. Successivamente, ne abbiamo creati altri per separare la logica del controller e dalla view.

Con il tempo, abbiamo iniziato a utilizzare un branch di sviluppo principale chiamato “develope”, da cui ne sono stati creati alcuni separati per sviluppare nuove funzionalità. Una volta completati, i branch venivano uniti (merged) nel“develope” tramite pull request. Le pull request venivano approvate sia dai compagni di team che da chi aveva creato il branch.

I branch creati durante lo sviluppo includevano attività come:

- Gestione delle costruzioni.
- Scambi e gestione della bancarotta.
- Salvataggio dei dati su JSON.
- Trasformazione delle view e dei controller del menu in Singleton.

Una volta raggiunta la prima versione operativa del sistema, il codice è stato integrato nel branch principale (main). In caso di manutenzione o modifiche, verrà creato un branch alternativo dedicato, su cui saranno implementate e testate le modifiche necessarie. Dopo aver verificato che tali modifiche non introducano malfunzionamenti, il branch alternativo sarà nuovamente integrato nel main, garantendo così la stabilità e la continuità operativa del sistema.

5. Software quality

Il team si è impegnato a creare un programma che rispettasse i fattori di qualità definiti da McCall.

Funzionamento del programma:

- **Correttezza**: Il programma soddisfa pienamente le specifiche iniziali e gli obiettivi definiti all'avvio del progetto, garantendo l'aderenza ai requisiti funzionali previsti.
- **Affidabilità**: Tutte le funzionalità sono state sottoposte a numerosi test, garantendo che il software esegua correttamente le attività previste in condizioni normali di utilizzo. Tuttavia, come per qualsiasi applicazione software, non si possono escludere del tutto eventuali malfunzionamenti imprevisti in scenari non considerati durante la fase di sviluppo e testing.
- **Efficienza**: Il programma, pur non essendo completamente ottimizzato in termini di prestazioni, funziona correttamente su qualsiasi computer in grado di eseguire Java, senza richiedere risorse hardware o software particolarmente avanzate.
- **Integrità**: Il programma è sicuro anche perché non vengono richiesti i dati degli utenti per funzionare.
- **Usabilità**: L'interfaccia del programma è intuitiva e di facile comprensione. Durante il gioco, i giocatori ricevono messaggi di output in base all'operazione che viene effettuata, aiutandoli nel gestire i turni.

Revisione del programma:

- **Manutenibilità**: Il software, nonostante la complessità, è risultato semplice da correggere in caso di errori o bug. Questo risultato è stato raggiunto grazie all'adozione di solide pratiche di ingegneria del software, divisione in mvc, modularizzazione del codice e l'inserimento di commenti dettagliati.
- **Testabilità**: Il programma è testabile. Il team si è concentrato sul testare solo il modello.
- **Flessibilità**: L'architettura del software è stata sviluppata con un buon grado di modularità, consentendo l'aggiunta di nuove funzionalità in futuro, senza modificare eccessivamente il codice esistente.

Transizione del programma:

- **Portabilità**: Il programma è compatibile con tutte le macchine che supportano Java, garantendo un'elevata portabilità indipendentemente dalla piattaforma. Un fattore limitante è l'interfaccia grafica, essendo fissa e pensata per schermi con risoluzione di 1920x1080.
- **Riutilizzabilità**: Il programma che è stato creato è difficile che possa essere riutilizzato in altre applicazioni che non riguardano il Monopoly.

6. Requirement engineering

I requisiti del sistema sono stati derivati attraverso un processo che ha combinato diverse tecniche di analisi. In particolare:

- Studio delle regole ufficiali del Monopoly: per garantire che tutte le funzionalità rispettino fedelmente le meccaniche e le dinamiche del gioco originale.

- Osservazione etnografica: giocando a Monopoly e osservando direttamente le interazioni tra i giocatori, le decisioni prese e le sfide comuni, con l'obiettivo di identificare esigenze e comportamenti ricorrenti.
- Analisi dei casi d'uso: mappando le possibili interazioni tra i giocatori e il sistema, esplorando i flussi di gioco principali e le varianti per identificare le funzionalità richieste.
- Progettazione di scenari: immaginando situazioni di gioco reali e come rappresentare a livello di interfaccia grafica, al fine di garantire una copertura completa delle attività e delle regole.

Queste attività ci hanno permesso di tradurre le regole e le dinamiche del Monopoly in requisiti dettagliati per il sistema, assicurando una progettazione che riproduca fedelmente l'esperienza del gioco originale, adattandola al contesto digitale.

I requisiti sono stati classificati secondo MoSCoW e tenendo a mente il modello di Kano.

Must have	**Should have**	**Could have**	**Won't have**
Rappresentazione accurata del gioco da tavolo Monopoli	Interfaccia fedele	Giocatori automatici (bot)	La versione mobile per telefono
Salvare e ripristinare una partita	Modalità multi dispositivo	Salvataggio automatico	Le modalità alternative
Interfaccia base			Statistiche di gioco

6.1 Specifica dei requisiti:

6.1.1 Introduzione

Il presente documento descrive la specifica dei requisiti per il progetto di digitalizzazione del gioco da tavolo Monopoly.

6.1.1.1 Obiettivo

L'obiettivo principale del progetto è realizzare una versione digitale che simuli il gioco da tavolo Monopoly, permettendo ai giocatori di partecipare localmente tramite dispositivi elettronici. Il sistema deve replicare le regole e le dinamiche del gioco originale, garantendo un'esperienza utente fluida e immersiva.

6.1.1.2 Scopo

Il progetto intende inoltre migliorare la gestione del gioco, riducendo gli errori umani, velocizzando le dinamiche e semplificando le interazioni. Questo progetto servirà come caso di studio per applicare concetti chiave di ingegneria del software, tra cui modellazione dei requisiti, progettazione architetturale, sviluppo iterativo e test del software.

6.1.2. Descrizione generale

6.1.2.1 Funzioni del prodotto

Le principali funzioni includono:

- Simulazione delle dinamiche di gioco: il prodotto implementa le regole originali del Monopoly;
- Assistenza guidata: il sistema supporta i giocatori durante la partita, impedendo azioni non valide e fornendo suggerimenti, rendendo il gioco accessibile anche a chi non conosce approfonditamente le regole;
- Gestione automatica delle operazioni: elimina la necessità di calcoli manuali o interventi fisici, ad esempio per il pagamento di affitti, l'acquisto di proprietà o la costruzione di case e alberghi;
- Salvataggio delle partite: il sistema consente ai giocatori di salvare lo stato della partita in qualsiasi momento, permettendo loro di riprendere il gioco in seguito senza perdere i progressi.

6.1.2.2 Caratteristiche dell'utente

L'utente non è obbligato a conoscere le regole del gioco per partecipare, poiché il sistema è progettato per impedire azioni non consentite e fornire assistenza durante il gioco. Tuttavia, una conoscenza preliminare delle regole può migliorare l'esperienza di gioco.

6.1.2.3 Vincoli

Per giocare bisogna aver installato JDK e/o JRE a seconda di come verrà consegnato.
L’interfaccia grafica attualmente è progettata per schermi con risoluzione 1920x1080.
In modalità locale su un singolo computer è fondamentale che i giocatori mantengano un comportamento onesto, poiché il sistema non può impedire che un giocatore esegua azioni per conto di un altro o forzi scambi non desiderati.

6.1.3 Requisiti specifici

6.1.3 Requisiti dell'interfaccia esterna e richieste funzionali

6.1.3.1 Interfaccia utente

L'utente è la persona che interagisce con il programma fuori dalle fasi di gioco. Una volta avviata l'applicazione, può compiere scelte iniziali tramite un’interfaccia con bottoni, che consente di:

- Creare una nuova partita:
    - L’utente seleziona questa opzione per iniziare una nuova sessione di gioco.
    - Dopo aver scelto questa opzione, dovrà specificare il numero di giocatori partecipanti o tornare indietro. Dopo la scelta c’è l’interfaccia di gioco (6.1.3.2);
- Caricare una partita salvata:

    - Viene mostrata una tabella contenente i salvataggi disponibili, catalogati per nome del salvataggio, numero di giocatori, data e orario di salvataggio.
    - L'utente può digitare il nome di un salvataggio (case-sensitive) e scegliere se caricarlo per riprendere la partita o eliminarlo.
    - In caso di caricamento, il gioco riprenderà dal punto esatto in cui era stato interrotto, ripristinando lo stato di gioco.
- Uscire dal programma:
    - L'utente può scegliere di chiudere l'applicazione senza avviare o riprendere una partita.

6.1.3.2 Interfaccia di gioco

Dopo che l'utente ha configurato il numero di partecipanti, i giocatori inseriranno il loro nome e uno di loro confermerà la scelta per procedere. I giocatori sono le persone che partecipano attivamente alla partita.
Successivamente in base all'ordine di inserimento dei nomi sceglieranno la propria pedina secondo la disponibilità.
Finita la configurazione della partita apparirà a schermo l'interfaccia di gioco, contenente:

- Una **console per i messaggi di gioco** per notificare gli eventi e le azioni svolte durante la partita;
- Il **tabellone**;
- I **bottoni di gioco**: per eseguire le diverse azioni disponibili;
- I **dati dei giocatori**: saldo disponibile, numero di carte per uscire gratis di prigione, la pedina relativa al giocatore e la lista delle proprietà che possiede.

Durante il turno il player potrà decidere cosa fare:

- Tirare i dadi: verrà mostrata la pedina sulla casella di arrivo e in base al tipo di casella si possono attivare eventi;
- Scambi tra giocatori: l'accordo viene preso a voce, e, in caso di consenso, il giocatore corrente seleziona con chi scambiare, le proprietà e/o il denaro necessario per concludere lo scambio.
- Eseguire operazioni sulle proprietà: si possono gestire le proprietà possedute decidendo se:
    - Costruire: per acquistare case o alberghi;
    - Demolire: per rimuovere case o alberghi per recuperare parte del denaro;
    - Ipotecare: per mettere proprietà in ipoteca per ottenere denaro;
    - Disipotecare: per riscattare proprietà ipotecate.

Successivamente bisogna selezionare la proprietà in base al gruppo colore e al nome.

- Bancarotta: il giocatore può ritirarsi dalla partita volontariamente o in caso di incapacità di ripagare i debiti con la banca;
- Fine turno: quando il giocatore ha completato tutte le azioni desiderate, può passare il turno al successivo;

- Salva: decide di salvare la partita e che nome dare al salvataggio;
- Esci: decide di concludere la partita;
- Paga cauzione: se in prigione, può pagare la cauzione per uscire senza attendere un tiro di dadi doppi;
- Esci gratis di prigione: se in possesso della carta speciale, può usarla per uscire di prigione senza pagare la cauzione o tirare i dadi doppi;

6.1.3.3 Richieste funzionali del sistema:

- Gestire il movimento del giocatore sul tabellone e i diversi scenari:
    - se la casella è una proprietà e non è posseduta da nessuno allora si può decidere di:
        - comprarla, la transazione verrà eseguita automaticamente;
        - mandarla all'asta per provare a pagarla meno oppure se non si è interessati o se non si hanno fondi sufficienti all'acquisto;
    - se la casella è una proprietà ed è posseduta allora l'affitto sarà pagato automaticamente, in base al numero e al tipo di costruzioni presenti, se è ipotecata o meno e all'identità del proprietario;
    - se la casella è un imprevisto o una probabilità allora verrà pescata una carta dal rispettivo mazzo ed eseguita l'azione indicata;
- Gestione delle proprietà (costruzione, demolizione, ipoteca, disipoteca, scambio, bancarotta);
- Gestione delle aste;
- Gestione della bancarotta;
- Gestione delle carte speciali e del mazzo che le contiene;
- Gestione della prigione;
- Aggiornare i dati in base agli eventi;
- Gestione della vittoria: deve mostrare la schermata di vittoria con i bottoni per uscire dal programma o tornare al menu quando resta un solo giocatore dato che tutti gli altri player hanno dichiarato la bancarotta;
- Gestire i salvataggi e i caricamenti delle partite: serializzando e deserializzando un file JSON;

Più in generale deve essere in grado di gestire gli eventi e le operazioni secondo le regole di gioco.

6.1.4 Requisiti di prestazione

Non è richiesta un'alta potenza di calcolo per questo applicativo e non necessita di connessione internet.

6.1.5 Vincoli di progettazione

Il gioco deve essere sviluppato utilizzando il linguaggio di programmazione Java. Questo implica che il codice sorgente sarà scritto interamente in Java, sfruttando le sue librerie standard per la gestione della grafica, delle interazioni e della logica di gioco. Inoltre, si dovranno utilizzare strumenti come JavaFX o Swing per l'interfaccia grafica, a seconda delle esigenze e delle preferenze tecniche.

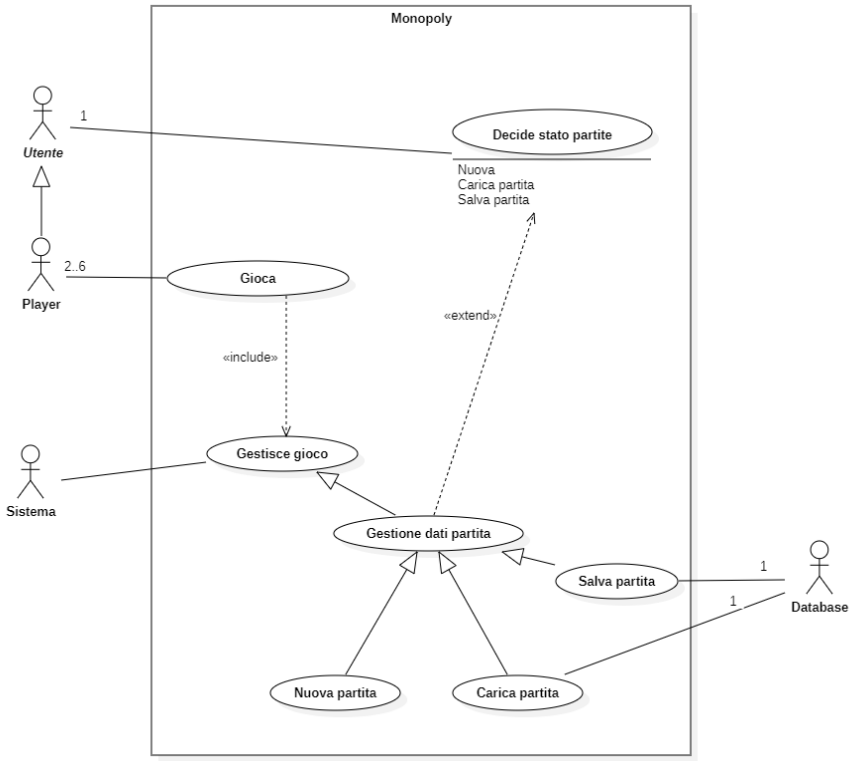
Il progetto deve essere gestito tramite GitHub, utilizzando il sistema di versionamento per tracciare ogni modifica al codice e gestire le diverse versioni del gioco.

Il progetto deve essere configurato e gestito tramite Maven, un sistema di build e gestione delle dipendenze per progetti Java.

Il gioco digitale deve aderire rigorosamente alle regole tradizionali di Monopoly. L'obiettivo è garantire che l'esperienza di gioco rimanga fedele al prodotto originale, con eventuali aggiustamenti fatti solo per facilitare l'adattamento digitale, senza alterare la sostanza del gioco.

L'interfaccia grafica del gioco deve essere intuitiva, facile da usare e piacevole. I giocatori devono essere in grado di navigare attraverso le diverse fasi del gioco senza confusione, con un layout chiaro e comandi facilmente riconoscibili.

Diagramma dei casi d'uso



7. Modelling con UML

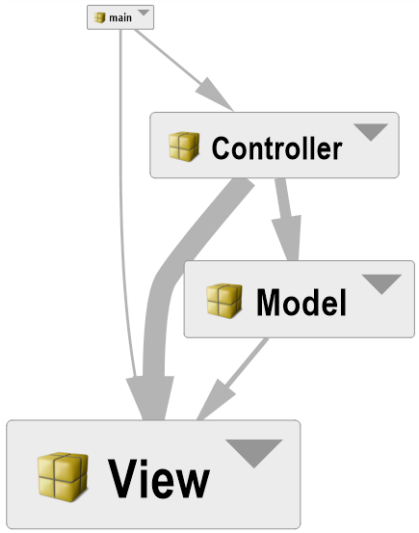
Link agli UML del progetto:

https://github.com/alorenzi10/Monopoly/tree/main/documentazione/uml

8. Software architecture

Il nostro programma è basato sulla architettura MVC (Model-View-Controller):

- **Model**: rappresenta la funzione logica del gioco. Ha il compito di eseguire i calcoli e gestire i dati. Ad esempio, quando vengono lanciati i dadi, il modello cambia la posizione della pedina, controlla in che tipo di casella si è atterrati e in base a quello effettua una determinata operazione;
- **Controller**: ha il compito di gestire gli input dell'utente; quasi sempre sono eventi ricevuti dai bottoni della View. Interagisce quindi con il Model e/o modifica la View. Ad esempio quando un giocatore decide di cliccare sul bottone "Acquista", la proprietà diventa del giocatore e viene aggiornato l'elenco a schermo delle proprietà.
- **View**: ha il compito di mostrare l'output all'utente, contiene le varie schermate e le alterna/modifica in base alle chiamate effettuate dal Model e/o dal Controller. Ad esempio, quando un giocatore decide di giocare una nuova partita premendo un bottone, il controller rende invisibile il menu principale e chiama la schermata successiva.



La separazione delle responsabilità dell'interfaccia grafica dalla logica, ci ha permesso di ridurre la dipendenza tra i vari componenti e questo ha facilitato la manutenzione e l'incremento del codice. Inoltre, ha semplificato il lavoro in team, permettendo una maggiore indipendenza tra i vari sviluppatori.
Questa modularità ha agevolato anche una futura manutenzione del codice.

9. Software design

Abbiamo utilizzato tre design pattern principali nel nostro progetto:

Singleton pattern

Nel progetto è stato adottato Singleton per garantire che le schermate del menu, la schermata di vittoria e i relativi controller siano rappresentati da una singola istanza univoca. Questa scelta progettuale ha permesso di evitare la creazione di istanze duplicate, migliorando la stabilità del programma e l'efficienza nella gestione della memoria e delle risorse.

Tuttavia, il pattern Singleton non è stato implementato nelle classi MonopolyGUI e MonopolyController per motivi di tempistiche e di priorità. L'implementazione non cambierebbe l'esperienza di gioco dell'utente in modo significativo.
In future sessioni di manutenzione, potrebbe essere utile valutare l'introduzione di questa ottimizzazione come parte di una revisione più ampia dell'architettura, al fine di migliorare ulteriormente la coerenza e la robustezza del progetto.

Abstract Occurrence pattern

L'Abstract Occurrence è stato applicato per ottimizzare la gestione dei mazzi di carte Imprevisti e Probabilità, che condividono caratteristiche comuni come la lista delle carte e le azioni associate. Questo approccio ha permesso di ridurre la duplicazione del codice e di centralizzare le logiche comuni migliorando la manutenibilità e la leggibilità del codice.

Delegation pattern

Quando un utente interagisce con un bottone, l'oggetto JButton delega la gestione dell'evento associato ad un listener, implementando così il Delegation Pattern. Questo approccio è comune nella gestione degli eventi in Java Swing.

In Asta, il metodo getName() utilizza una delegazione, poiché chiama giCorrente.getName() per ottenere il nome, delegando l'azione al metodo della classe Player.

L'utilizzo di questo pattern aumenta la flessibilità del sistema e incoraggia il riutilizzo del codice. Tuttavia, comporta un incremento nel numero di livelli di comunicazione, il che potrebbe avere un impatto sulle prestazioni complessive e determinare un aumento della complessità del sistema.

9.1 Analisi della qualità del codice

Per valutare la complessità delle classi e la qualità del codice, abbiamo utilizzato diversi strumenti. Tra questi:

CodeMR: è stato utilizzato per visualizzare ed analizzare le metriche di qualità del software, ed è stato utile per identificare e comprendere i problemi presenti nel codice;

JArchitect: per generare grafici dettagliati che evidenziano i punti critici del codice sorgente.

Checkstyle, UCDetector, SpotBugs: per identificare problemi di struttura, utilizzo inefficiente del codice e potenziali bug;

Le analisi automatiche sono state integrate con revisioni manuali, per garantire che il codice fosse conforme alle migliori pratiche e facile da mantenere. Grazie a questo approccio, siamo riusciti a individuare e risolvere punti deboli significativi, migliorando la robustezza e la leggibilità del progetto.

L'attuale complessità di alcuni metodi è giustificata dalla natura stessa del gioco, come ad esempio creazioneCaselle(), in MonopolyGUI. Abbiamo gestito ogni singola casella del tabellone come JLabel perché questo in futuro può permettere una maggiore

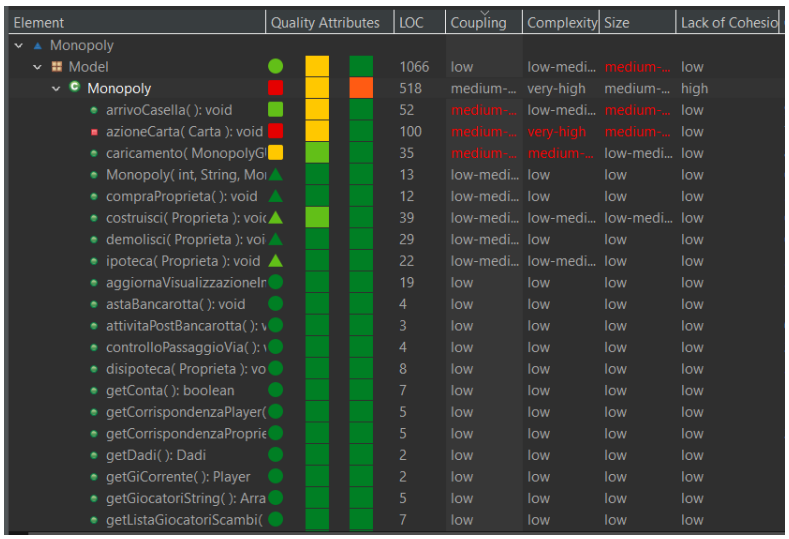
personalizzazione a livello grafico, rispetto all'utilizzo di un'immagine statica del tabellone completo. Analizzando la sua struttura si nota infatti la ripetitività del pattern di creazione e aggiunta delle caselle insieme alle rispettive immagini.

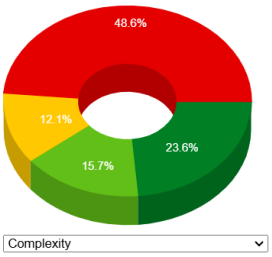
Un altro metodo complesso è arrivoCasella() questo metodo gestisce le diverse tipologie di caselle su cui il giocatore può atterrare dopo il lancio dei dadi, utilizzando numerosi if. La complessità è dovuta alla necessità di coprire le diverse interazioni previste dal gioco, garantendo che ogni tipo di casella sia gestita correttamente. Questo discorso vale anche per il metodo azioneCarta().

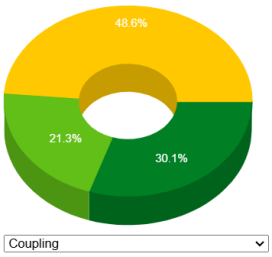
9.2 CodeMR

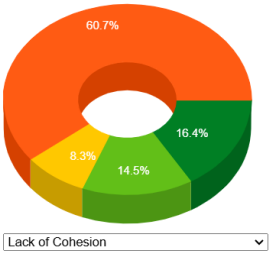
L'analisi dei grafici ottenuti ha evidenziato la presenza di alcune classi che incidono negativamente sulla qualità generale del codice a causa della loro struttura necessariamente complessa, dovuta alle specifiche esigenze del gioco.
Un esempio significativo è rappresentato dalla classe CaseAlberghiView, incaricata di gestire la creazione di 110 edifici posizionati in punti diversi del tabellone. Anche i metodi discussi nel paragrafo precedente, come creazioneCaselle() e arrivoCasella(), contribuiscono a questa complessità intrinseca.
Nonostante ciò, le metriche globali non sono compromesse in modo critico. Infatti, i dati forniti da JArchitect evidenziano un buon livello di manutenibilità del codice, indicando che il sistema è strutturato in modo tale da poter essere aggiornato e modificato efficacemente, pur rispettando le esigenze di complessità richieste dal progetto.

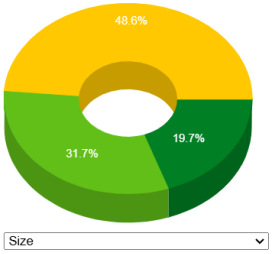
Package Model



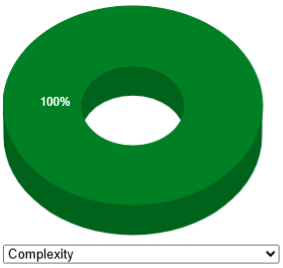


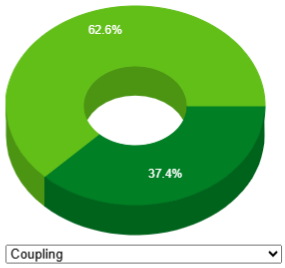


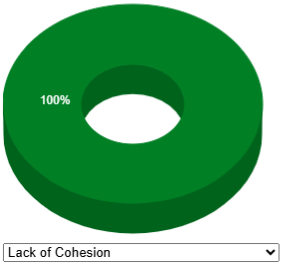


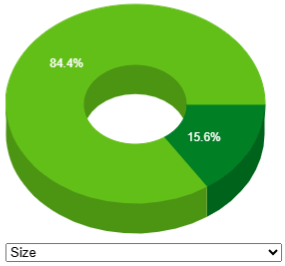


Package Controller senza la classe MonopolyController

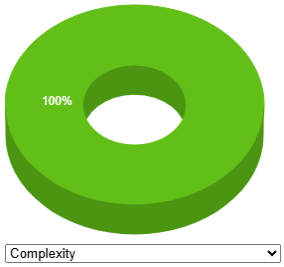


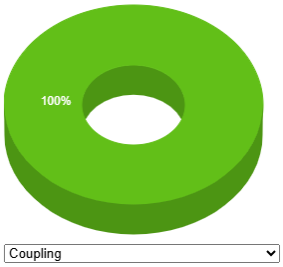


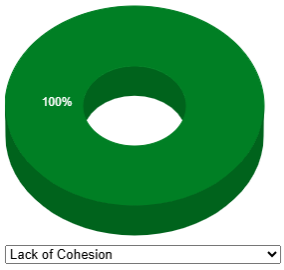


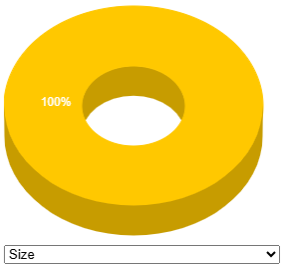


Classe MonopolyController

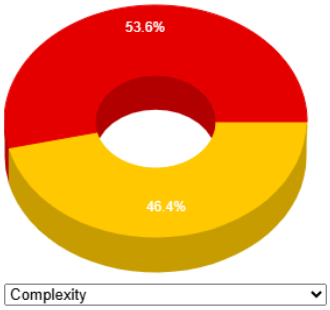


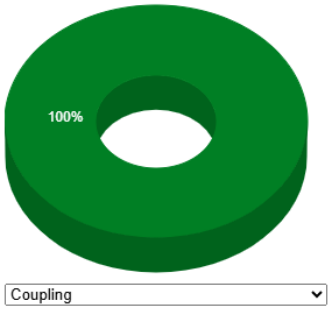


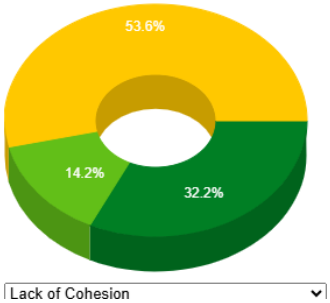


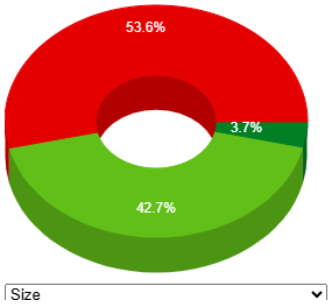


Package View







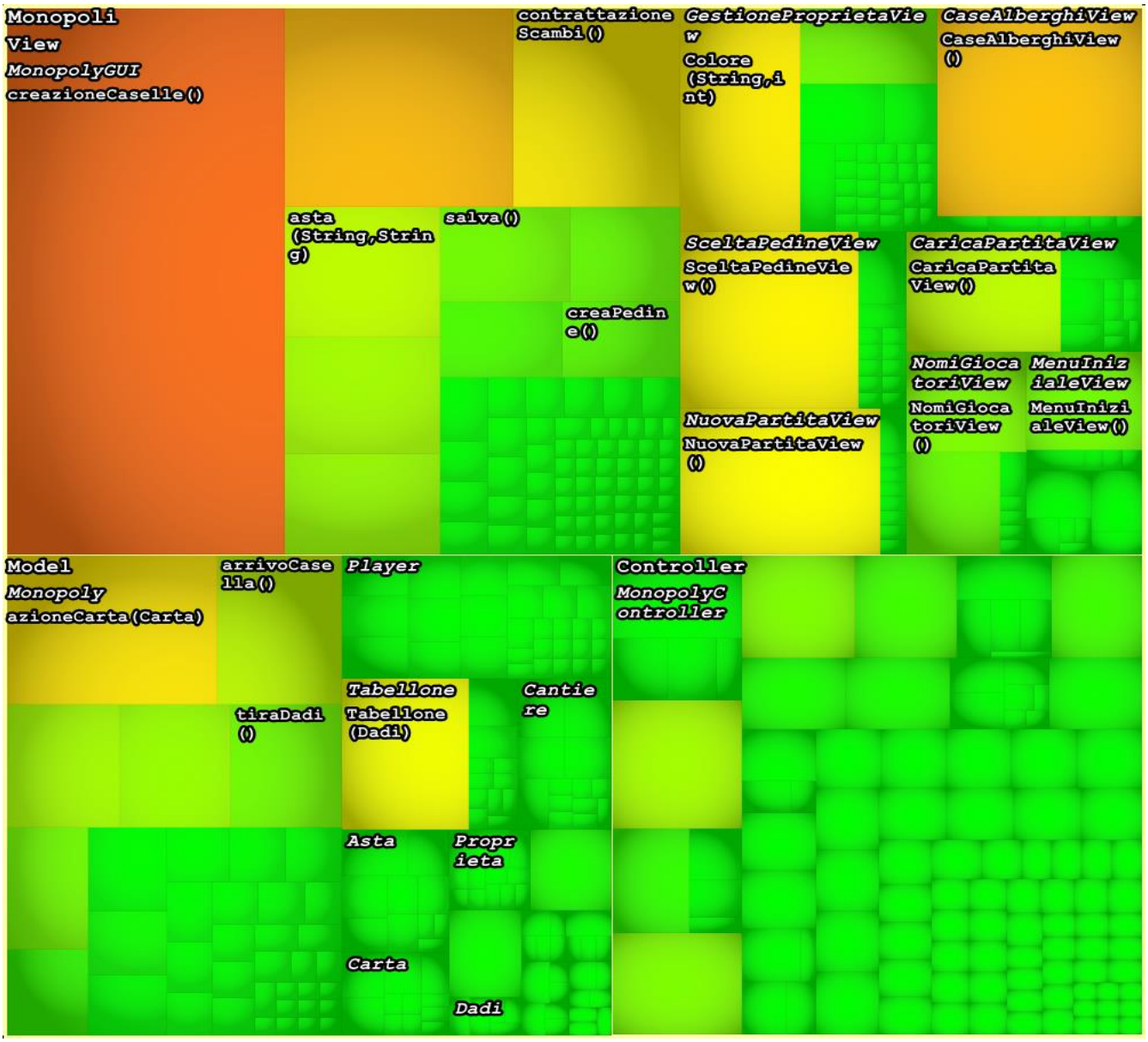


- **Lack of Cohesion**: indica quanto poco i metodi di una classe sono coesi tra di loro. È preferibile che la coesione sia elevata poiché favorisce caratteristiche desiderabili al software come robustezza, affidabilità, riusabilità e facilità di comprensione.
- **Complexity**: è la difficoltà di comprensione e descrive le interazioni tra le entità del sistema. Una complessità maggiore aumenta la possibilità di avere interazioni

accidentali e di conseguenza la probabilità di introdurre errori durante le modifiche del codice.

- **Size**: misura la quantità di righe e metodi presenti in una classe.
- **Coupling**: indica il livello di dipendenza tra i componenti del sistema del software. È preferibile che ci sia un basso accoppiamento, poiché consente ad ogni componente di essere modificabile senza influire sugli altri.

Manutenibilità del codice secondo JArchitect



10. Software testing

I nostri test si sono concentrati interamente sulla parte logica del progetto, rappresentata dal pacchetto Model.

L'obiettivo era quello di controllare la corretta gestione e lettura delle variabili condivise tra classi. In questo modo ci siamo assicurati del coerente sviluppo dei dati, accorgendoci di alcune imprecisioni tralasciate nella fase di implementazione delle classi.

Per eseguirli abbiamo utilizzato due paradigmi:

- JUnit: utilizzato per identificare e correggere eventuali errori in modo sistematico, assicurandoci che i metodi soddisfacessero i requisiti previsti;
- Test manuali: Impiegati per verificare che gli output del programma corrispondessero alle attese, garantendo un controllo diretto sul comportamento delle funzionalità implementate.

Abbiamo testato che non venissero istanziati oggetti nulli, che le proprietà venissero assegnate e/o scambiate correttamente, che i valori d'affitto richiesti fossero corretti, che le pedine si spostassero nella casella giusta dopo il lancio dei dadi, ecc…

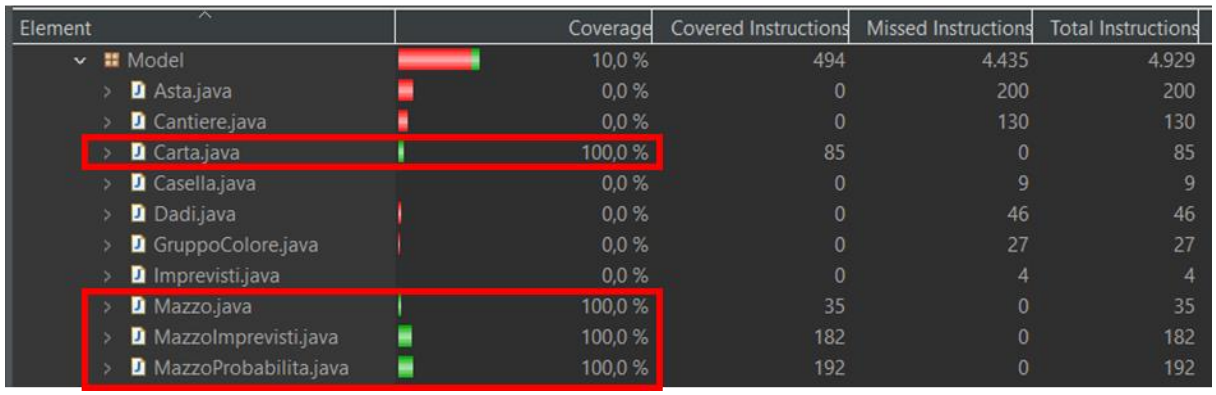
In generale quindi abbiamo testato che i metodi non lanciassero eccezioni, che ritornassero i valori come previsto e le modifiche venissero effettuate.

I test scritti hanno una buona copertura, calcolata con gli strumenti forniti da Eclipse.

MonopolyTest

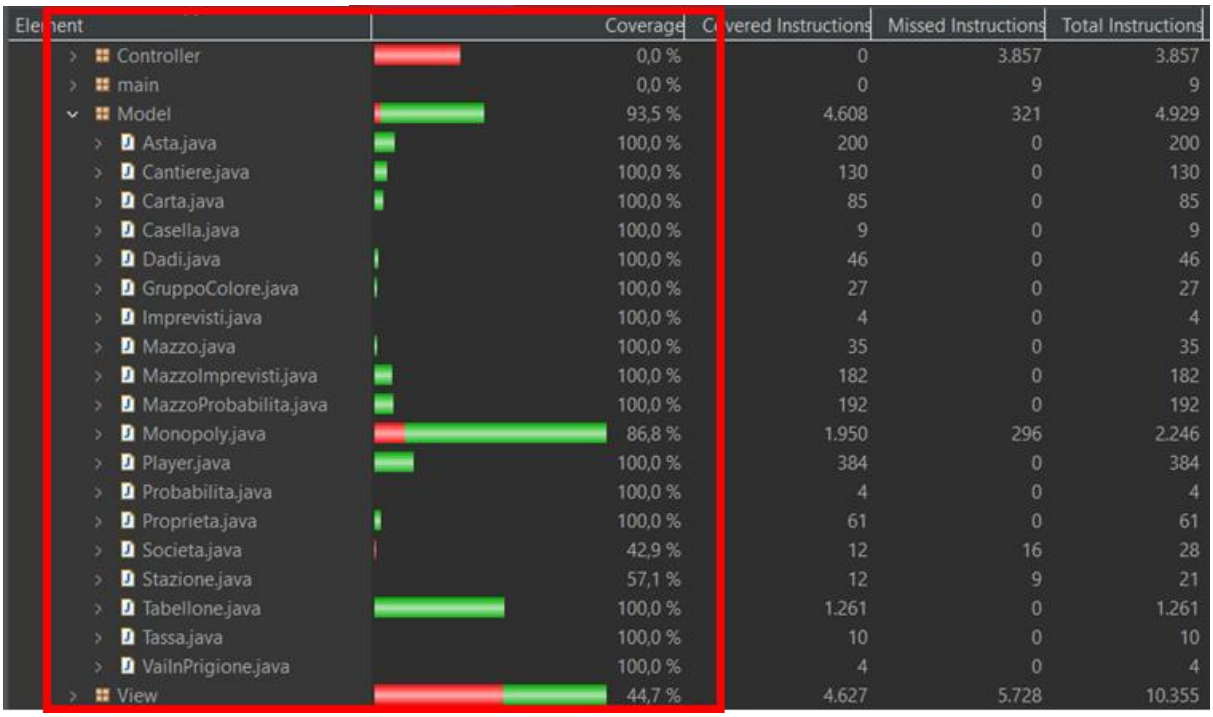
Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
Model	77,8 %	3.834	1.095	4.929
Asta.java	25,0 %	50	150	200
Cantiere.java	69,2 %	90	40	130
Carta.java	81,2 %	69	16	85
Casella.java	100,0 %	9	0	9
Dadi.java	100,0 %	46	0	46
GruppoColore.java	74,1 %	20	7	27
Imprevisti.java	100,0 %	4	0	4
Mazzo.java	82,9 %	29	6	35
MazzoImprevisti.java	100,0 %	182	0	182
MazzoProbabilita.java	100,0 %	192	0	192
Monopoly.java	73,4 %	1.648	598	2.246
Player.java	55,5 %	213	171	384
Probabilita.java	100,0 %	4	0	4
Proprieta.java	90,2 %	55	6	61
Societa.java	42,9 %	12	16	28
Stazione.java	57,1 %	12	9	21
Tabellone.java	94,0 %	1.185	76	1.261
Tassa.java	100,0 %	10	0	10
VaiInPrigione.java	100,0 %	4	0	4

MazzoECarteTest



Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
Model	10,0 %	494	4.435	4.929
Asta.java	0,0 %	0	200	200
Cantiere.java	0,0 %	0	130	130
Carta.java	100,0 %	85	0	85
Casella.java	0,0 %	0	9	9
Dadi.java	0,0 %	0	46	46
GruppoColore.java	0,0 %	0	27	27
Imprevisti.java	0,0 %	0	4	4
Mazzo.java	100,0 %	35	0	35
MazzoImprevisti.java	100,0 %	182	0	182
MazzoProbabilita.java	100,0 %	192	0	192

Copertura di tutti i test



Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
Controller	0,0 %	0	3.857	3.857
main	0,0 %	0	9	9
Model	93,5 %	4.608	321	4.929
Asta.java	100,0 %	200	0	200
Cantiere.java	100,0 %	130	0	130
Carta.java	100,0 %	85	0	85
Casella.java	100,0 %	9	0	9
Dadi.java	100,0 %	46	0	46
GruppoColore.java	100,0 %	27	0	27
Imprevisti.java	100,0 %	4	0	4
Mazzo.java	100,0 %	35	0	35
MazzoImprevisti.java	100,0 %	182	0	182
MazzoProbabilita.java	100,0 %	192	0	192
Monopoly.java	86,8 %	1.950	296	2.246
Player.java	100,0 %	384	0	384
Probabilita.java	100,0 %	4	0	4
Proprieta.java	100,0 %	61	0	61
Societa.java	42,9 %	12	16	28
Stazione.java	57,1 %	12	9	21
Tabellone.java	100,0 %	1.261	0	1.261
Tassa.java	100,0 %	10	0	10
VaiInPrigione.java	100,0 %	4	0	4
View	44,7 %	4.627	5.728	10.355

11. Software maintenance

Nel corso dello sviluppo del progetto Monopoly, ci siamo basati inizialmente su un repository preesistente, da cui abbiamo tratto lo scheletro delle principali funzioni tramite un processo di reverse engineering. Tuttavia, questa strategia si è rivelata meno vantaggiosa del previsto, poiché la struttura ereditata non era pienamente conforme alle esigenze del nostro progetto. Questo ha comportato la necessità di ripulire e riorganizzare il codice in maniera significativa.

Considerate le prime difficoltà incontrate con il pacchetto Swing, alcuni elementi logicamente simili della View sono stati implementati in modi differenti tra di loro. Questa

mancanza di uniformità potrebbe complicare la manutenzione futura, poiché manca una standardizzazione nei metodi utilizzati per rappresentare elementi strutturalmente affini.

Il processo di manutenzione prima della consegna ha visto:

- **Refactoring** (vedi 11.1);
- **Aggiornamento della documentazione**;
- **Aggiunta di commenti** per rendere il codice più intuitivo e facilitarne la comprensione da parte di futuri sviluppatori;
- **Miglioramenti grafici** per rendere il gioco più accattivante e migliorare l'esperienza utente.

In generale abbiamo applicato manutenzione correttiva, perfettiva e preventiva.

11.1 Refactoring

Abbiamo adottato un approccio iterativo al refactoring, eseguendolo sia alla fine del progetto sia durante le fasi di implementazione, con l'obiettivo di mantenere il codice ordinato e manutenibile.

Le azioni effettuate:

- La rinominazione delle variabili e dei metodi in modo tale da averli tutti in italiano e con nomi auto esplicativi;
- Eliminazione del codice che risultava inutilizzato;
- Miglioramento della struttura riorganizzando le classi e i package per seguire una logica più coerente e modulare;
- Rimozione di duplicazioni di codice anche tramite l'estrazione di metodi o classi comuni;
- Revisione e semplificazione dei costrutti condizionali complessi;

Come già indicato nel punto 9, insieme alla nostra poca esperienza a riconoscere i cattivi odori sono stati utilizzati anche i plugin SpotBugs, Unnecessary Code Detector, Checkstyle e il programma JArchitect con lo scopo di migliorare la qualità del codice sfruttando alcuni dei loro suggerimenti.