Relazione relativa al progetto d'esame PROGRAMMAZIONE E MODELLAZIONE A OGGETTI

sessione estiva — 2022/2023

Corso di Laurea in Informatica Applicata
Università di Urbino

DOCENTE:

Sara Montagna

STUDENTI:

Kristina Volkova
matricola: 313349

Gabriele Ramagnano
matricola: 315439

Indice

1 Analisi

L'obiettivo del progetto è realizzare un simulatore del gioco della roulette che ne testi le diverse strategie per raccogliere i dati e analizzare l'andamento del giocatore.
La roulette è un gioco d'azzardo che si svolge intorno ad un tavolo (table de jeu) di forma rettangolare, con un piano ricoperto quasi interamente da un tappeto verde (tapis verte) sul quale è disegnato il tracciato (tableau) formato da numeri e combinazioni esterne. La "roulette" è un cilindro in legno (le cylindre) il cui interno ruota ed è suddiviso in trentasette caselle (cases), le quali comprendono lo 0, diciotto numeri rossi e diciotto numeri neri alternati tra loro dall'1 al 36. Il gioco è molto semplice: si scommette contro il banco cercando di indovinare in quale numero cadrà la pallina lanciata dal croupier (le bouler). Le puntate vincenti daranno diritto a dei premi proporzionati sia al valore dei gettoni che al tipo di combinazione scelta.
Esistono varie tipogie di roulette, ma le più conosciute sono la roulette francese, l'inglese e l'americana. Il simulatore poggia le basi del suo funzionamento sulla roulette francese. La roulette francese è considerata la variante della roulette dove la vincita del banco è minore, ossia le chance di vincita dei giocatori sono maggiori.
Dal momento che è matematicamente impossibile vincere contro il banco sulla lunga distanza (cioè con un numero di giocate molto elevato), il compito di ogni strategia è mantenere il bilancio del giocatore più alto possibile dato un numero finito di giocate. In pratica la strategia indica come verranno gestite le puntate ad ogni giro della roulette a fronte di una possibile vincita o perdita. Ci sono due principali tipi di strategie: le strategie basate sulle puntate esterne oppure sulle combinazioni di numeri. Il simulatore del gioco prenderà in considerazione solamente le prime.

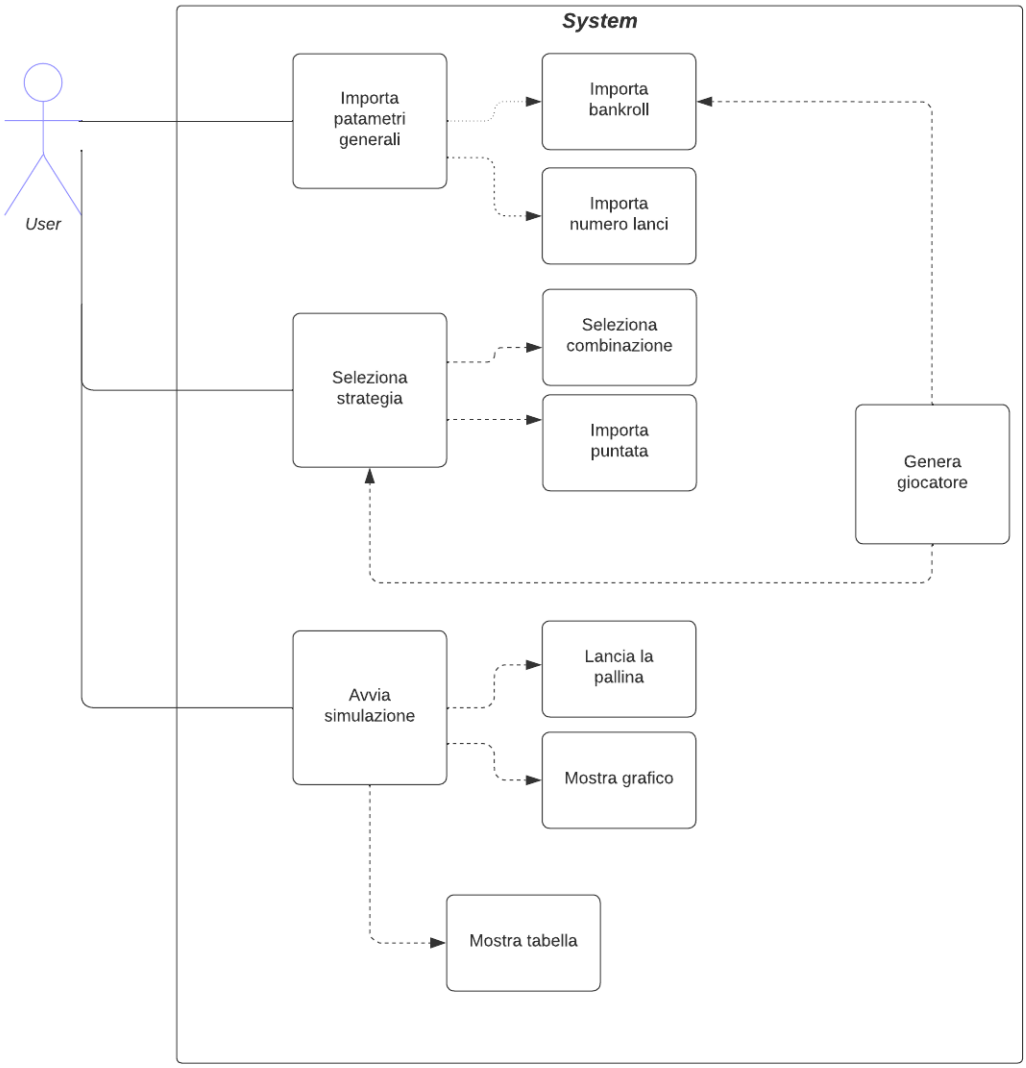
1.1 Requisiti

Requisiti funzionali

- L'applicazione permetterà la creazione di un numero arbitrario di giocatori, ciascuno caratterizzato da un codice identificativo, un bankroll, una puntata e una strategia.
- L'applicazione dovrà impostare per tutti i giocatori un bankroll e un numero di giocate uguali.
- L'applicazione potrà gestire in automatico la scelta delle combinazioni esterne relative alle puntate di ogni giocatore.
- L'utente potrà inoltre scegliere a piacimento le combinazioni esterne per ogni strategia selezionata.
- L'utente potrà impostare manualmente il bankroll e il numero di giocate (lanci della pallina) che verranno effettuati all'avvio della simulazione.
- L'utente sceglierà liberamente quante strategie testare fra quelle proposte. Difatti una strategia può essere selezionata più di una volta, ad esempio con valori di puntata differenti per simularne i diversi esiti nel gioco.
- All'avvio della simulazione non sarà possibile cambiare i dati impostati.
- Sarà possibile, a simulazione terminata, visualizzare su un grafico l'andamento del bilancio di ogni giocatore ad ogni giocata effettuta.
- Terminata la simulazione si potrà visualizzare una tabella che raccoglie per ogni giocatore i seguenti dati: id, strategia, puntata iniziale, puntata finale, combinazione, bankroll, bilancio, lanci.

Requisiti non funzionali

- L'applicazione dovrà garantire che i dati numerici inseriti nella simulazione siano coerenti con il tipo richiesto e della grandezza idonea (ad esempio il valore della posta non può superare il valore del bankroll).
- L'applicazione dovrà garantire che per ogni successiva modifica apportata al bankroll vengano eliminate le strategie precedentemente selezionate dal giocatore.
- L'organizzazione dei giocatori prima dell'inizio della simulazione dovrà essere ottimizzata in maniera tale da rendere più veloce il lavoro di ricerca.



1.2 Modello del dominio

Il programma dovrà essere in grado di simulare il comportamento di varie tipologie di strategie che utilizzano le combinazioni esterne.
Prima dell'inizio della simulazione vera e propria ci troveremmo dinanzi ad un tavolo da gioco (*TableManager*) composto da un tabellone che consta di una matrice di caselle (*Number*) e di appositi spazi esterni (combinazioni esterne) dove ogni singolo giocatore potrà registrare la sua puntata. Ogni giocatore sarà dotato di un bankroll comune ma di una sola strategia. Ciascuna strategia aggiornerà lo stato della propria puntata (*BetOutside*) impostata su una particolare combinazione esterna (*OutsideComb*).
Una volta partita, la simulazione terminerà nei seguenti casi:

- Nel momento in cui si esaurirà il numero di giocate (*launches*)
- Quando non ci saranno più giocatori rimasti in partita.

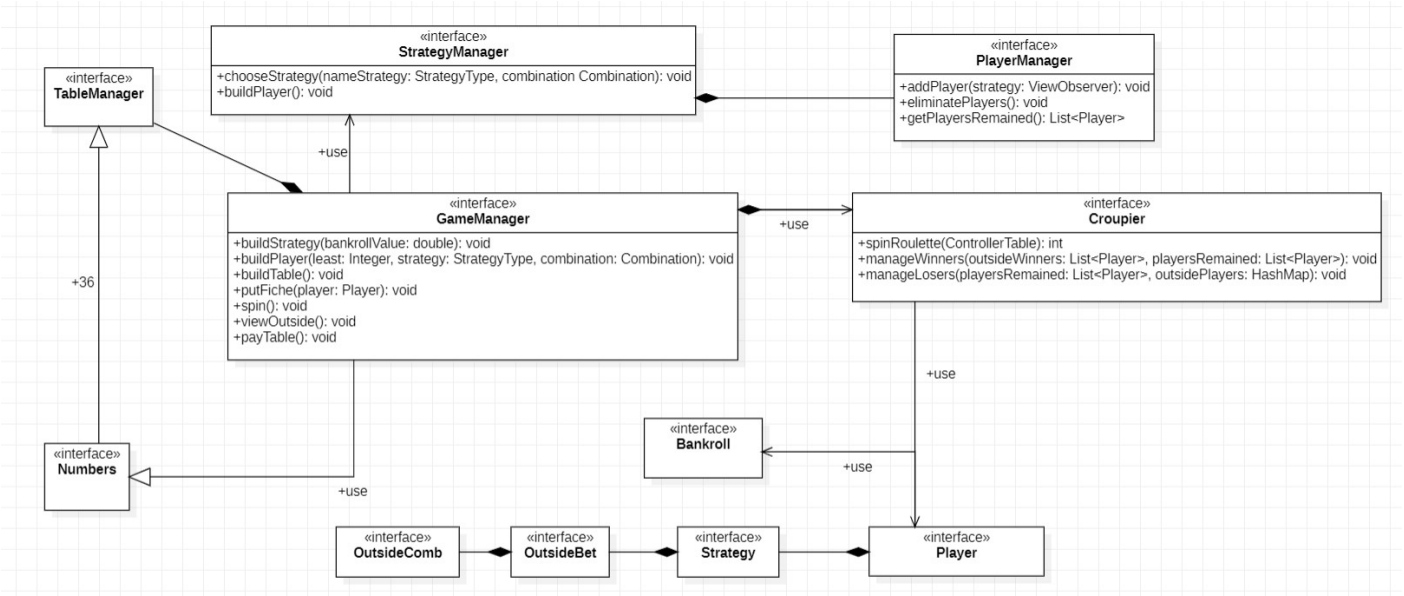
Ad ogni turno di gioco:

- Il Croupier lancierà la pallina ed estrarrà il numero.
- Verrà controllata la corrispondente casella e si vedrà quanti giocatori hanno puntato il numero uscito.
- Per ogni giocatore che ha puntato su quel numero e che sarà quindi risultato vincitore:

1. Si incrementerà il suo bankroll con la somma vinta.
2. Si deciderà se, data la sua futura puntata, ha ancora sufficiente bankroll per continuare a giocare. Se l'esito è positivo allora il giocatore passa al prossimo turno altrimenti esce dal gioco.

- Per tutti coloro che hanno scommesso diversamente (i perdenti) si prenderanno le medesime decisioni, tranne il fatto che il loro bankroll verrà decrementato.

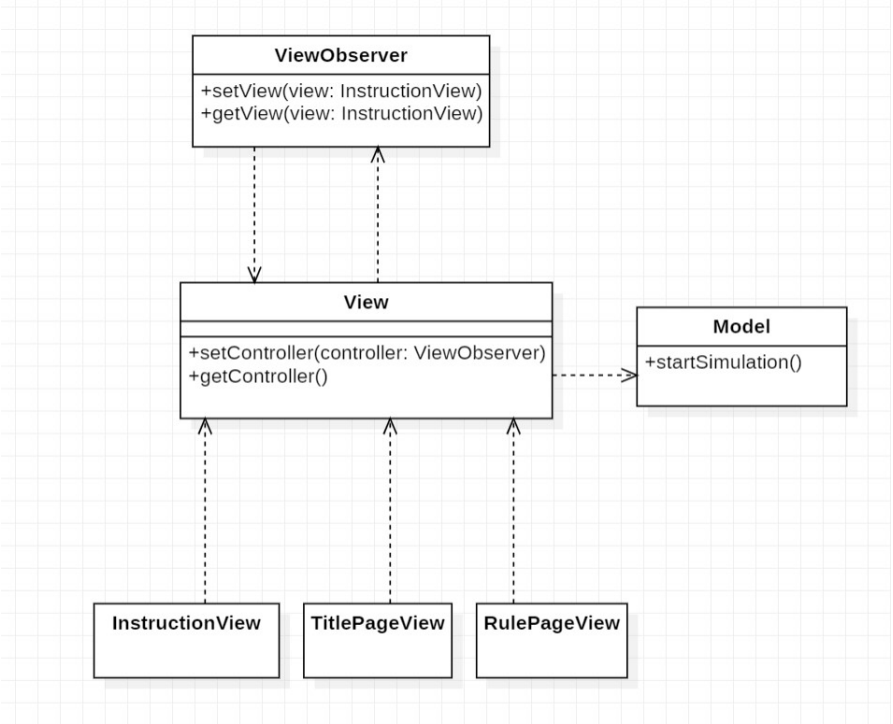
Al termine della simulazione ogni *Player* avrà la propria funzione di bilancio della partita, la quale descriverà il comportamento della strategia adottata.



2 Design

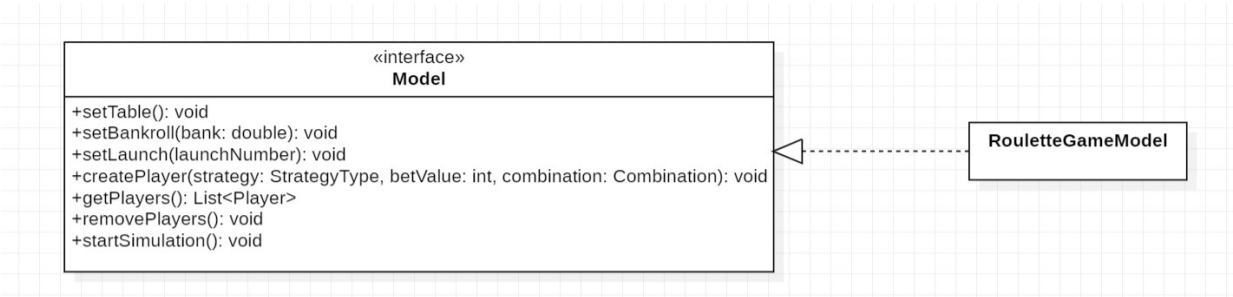
2.1 Architettura

Per lo sviluppo del programma di simuazione abbiamo deciso si seguire il pattern architetturale MVC (Model - View - Controller).



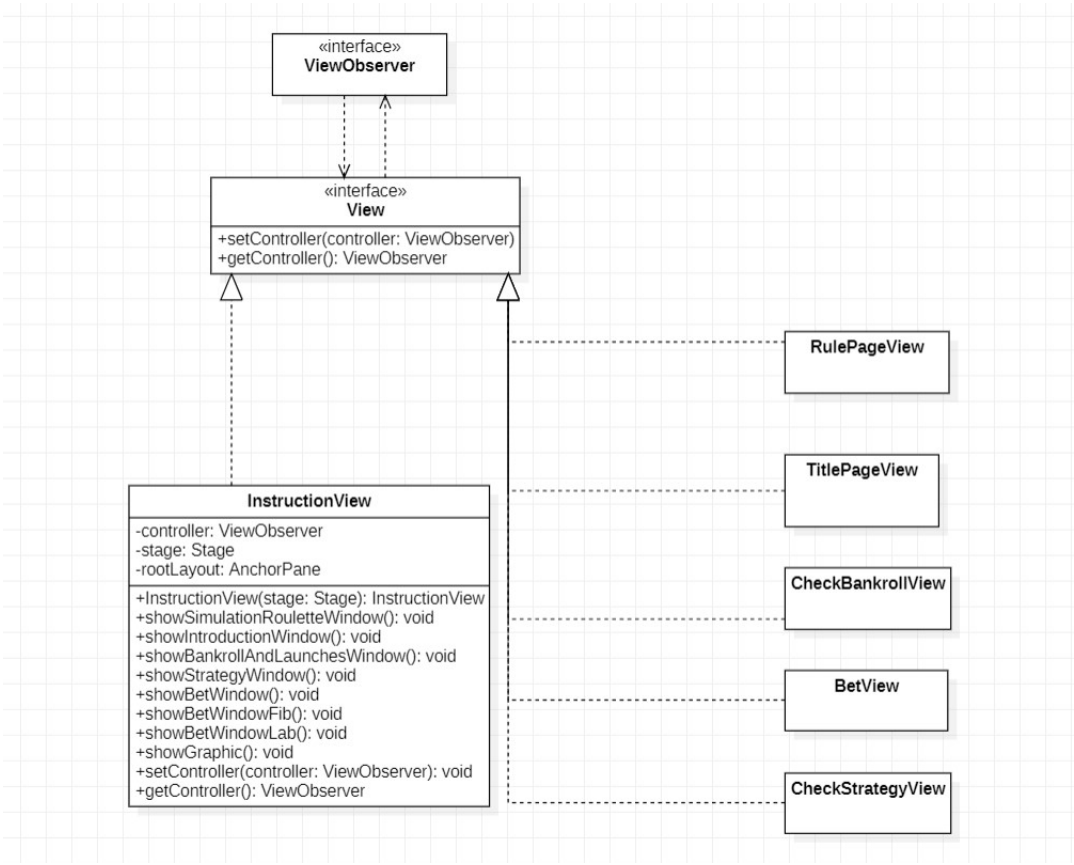
2.1.1 Model

Il Model rispecchia il dominio applicativo del sistema, ossia l'implementazione del modello astratto precedentemente discusso. Come tale esso rappresenta il *core* dell'applicazione, una sezione tanto centrale quanto indipendente dal resto del sistema.
Funzione principale del Model è difatti offrire dei servizi per mezzo dei quali compiere un insieme di sotto-operazioni che andranno ad aggiornare lo stato corrente del sistema ad ogni richiesta da parte dell'utente. Tutto ciò, tradotto in questo programma simulativo, significa predisporre funzionalità per la costruzione di giocatori e l'avvio della simulazione. Tale insieme di metodi è offerto dall'interfaccia *Model* implementata dalla corrispondente classe *RouletteGameModel*.



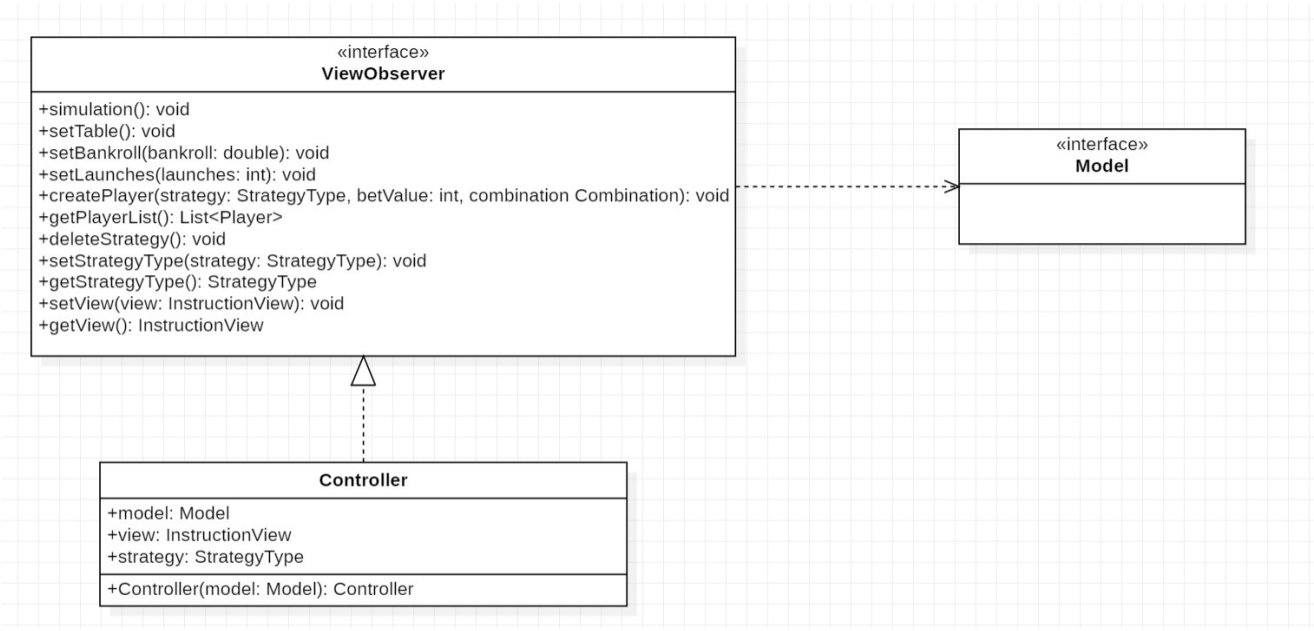
2.1.2 View

La View si presenta come l'interfaccia grafica del sistema attraverso la quale l'utente comunica con il dominio applicativo agendo su un insieme di dati. La View si occupa di informare il Controller delle scelte effettuate dall'utente e attende le successive istruzioni.
Si è deciso di suddividere la View in un numero di classi pari alla quantità di frame utilizzati. Per rendere più semplice il passaggio da una finestra all'altra si è utilizzata la classe *InstructionView* la quale predispone una lista di metodi ognuno collegato ad uno specifico frame. L'interfaccia mediante la quale ogni "ViewClass" dialoga con il Controller è la *View*.



2.1.3 Controller

Il Controller è la parte del sistema che fa da tramite fra View e Model. Si occupa di informare il Model riguardo alle richieste provenienti dalla View e coordina quest'ultima in maniera tale che possa aggiornarsi rispetto ai successivi cambiamenti di stato del sistema.
Nel suddetto programma si è scelto di utilizzare un solo Controller che dialoga con ogni singola *ViewClass* e gestisce, mediante l'*InstructionView*, metodi per il cambiamento di finestra. L'interfaccia utilizzata è *ViewObserver* implementata dalla classe *Controller*.



2.2 Design dettagliato

2.2.1 Ramagnano Gabriele

PlayerManager

Si è deciso di rappresentare il tabellone di gioco come un oggetto composto da caselle (*Number*) e spazi per le combinazioni esterne (*mappingOutside*).

Vantaggi:

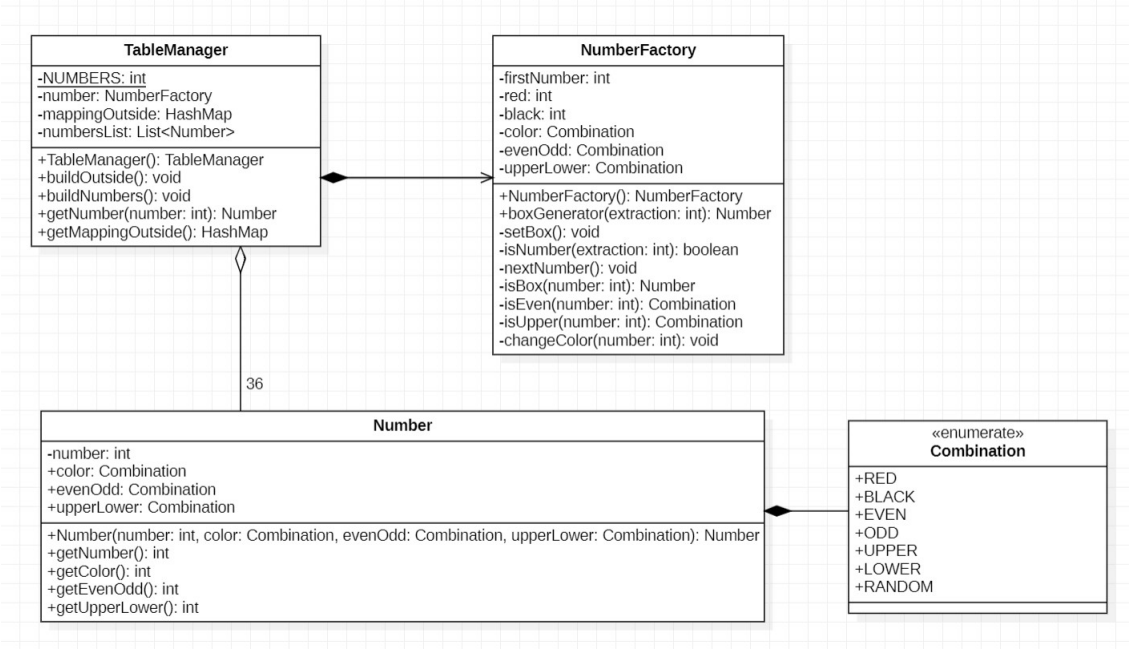
- Soluzione che rimappa fedelmente il problema reale (problem space) riproponendo gli stessi "oggetti" presenti nel gioco della Roulette.
- Si ottimizza la ricerca dei giocatori una volta partita la simulazione: occorre semplicemente visitare gli spazi dove ciascun giocatore si è registrato.
- Architettura estendibile anche nel caso in cui si voglia implementare anche le combinazioni interne (puntata singola, terzina, colonna, ecc.). Ad esempio, per le puntate singole, estendendo la classe *Number* si può ricavare un ulteriore spazio di "indirizzamento" (dove si depositano gli indirizzi degli oggetti *Player*) per i giocatori.

Svantaggi

- La creazione ed il posizionamento del giocatore richiedono molto tempo (una quantità comunque inferiore a quella richiesta alle operazioni effettuate dall'utente in fase di input/output nella scelta del giocatore).

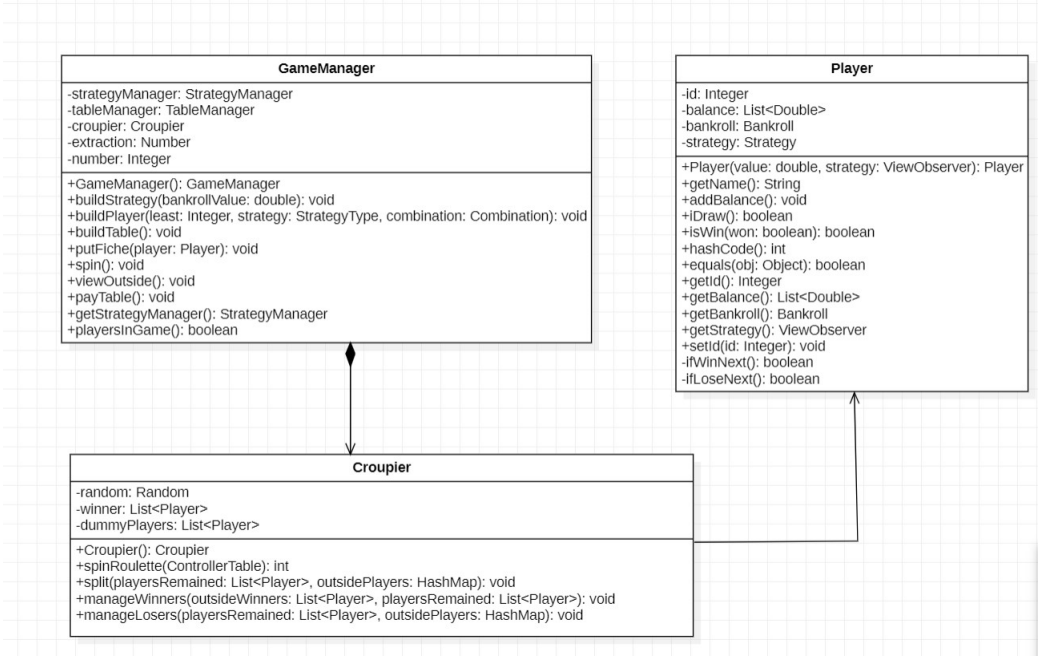
Le alternative considerate vedevano l'oggetto *Number* dotato solamente di due campi (colore e numero) e non sfruttavano l'utilizzo del posizionamento dell'oggetto *Player* negli appositi spazi collocati sul tabellone. Ciò avrebbe permesso di semplificare e snellire il modello del dominio, anche se la fase di simulazione sarebbe risultata più lenta per la presenza di maggiori operazioni da eseguire nel controllo per ogni giocatore della combinazione scelta.

In questa sezione è stato utilizzato il design patter creazionale *Simple Factory.*



Croupier

Una volta partita la simulazione l'attenzione si sposta dal *TableManager* al *Croupier* che diviene il cuore pulsante del modello. A lui infatti spetta il compito di gestire i pagamenti con i giocatori e aggiornarne le liste allorquando risulti che un *Player* non possiede più denaro a sufficienza per la prossima puntata.
Sarebbe stato possibile incapsulare le funzionalità del *Croupier* all'interno del *GameManager* ma ciò avrebbe di sicuro oberato tale classe di troppe responsabilità. In questi modo invece il design rimane più pulito oltre ad acquisire più coerenza rispetto al modello reale di riferimento. Così, se il *GameMananger* coordina il *Croupier*, a quest'ultimo sarà assegnata la gestione dei giocatori.
Per quanto essenziali tuttavia le operazioni eseguite dal Croupier risultano lente dal momento che, ad ogni giro di roulette, si dovranno aggiornare due liste di giocatori: la lista dei giocatori rimasti in gioco (*playersRemained*) e la lista dei giocatori presenti sul tabellone (*outsidePlayers*). Il limite che l'ottimizzazione di questa sezione presenta è dovuto alla rappresentazione del modello decisa in precedenza.



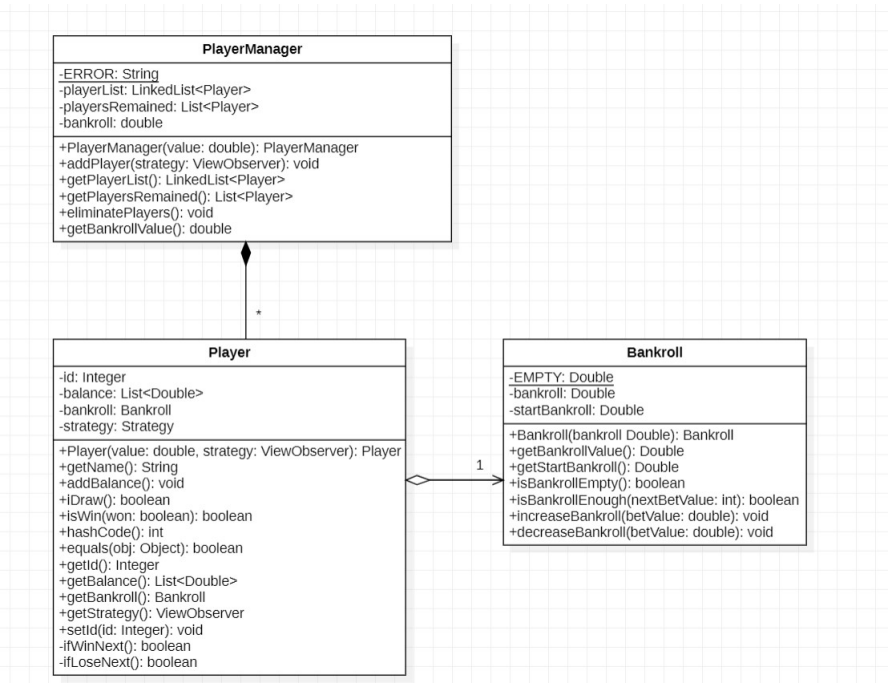
Player

Il programma di simulazione è nato con l'intento di analizzare il comportamento delle varie strategie che nel corso degli anni sono state elaborate allo scopo di ricavare dal banco un guadagno sempre più elevato. In tale ottica sarebbe inutile considerare nell'organizzazione del sistema simulativo la presenza di giocatori. Ciononostante l'inclusione della classe *Player* viene giustificata dalle seguenti osservazioni:

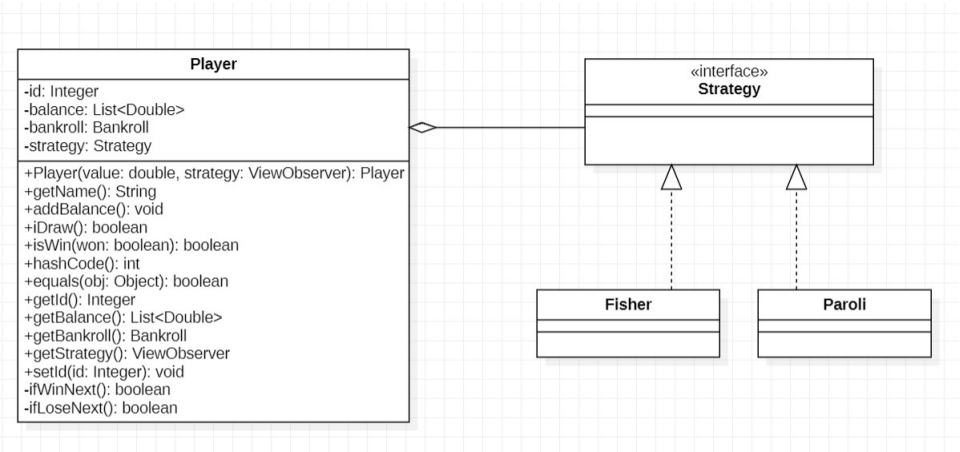
1. Si alleggerisce la porzione di codice delle strategie rendendo tali classi più leggibili e dotate di maggiore linearità e coesione rispetto alle funzionalità principali: decidere che scelte prendere quando si perde o si vince.

2. Si proietta il programma verso future vie di estensione. Si potrebbe ad esempio estendere la classe *Player* dotandola di una puntata che sia indipendente dalla strategia e della possibilità di cambiare più combinazioni durante il corso della partita.

3. La gestione del *Bankroll* diventa molto più semplice come pure l'aggiornamento della funzione di bilancio che descrive il comportamento della strategia.



L'inclusione della classe *Player* ci ha permesso di utilizzare il design pattern *Strategy*.



2.2.2 Volkova Kristina

Strategy

Le strategie sono la parte principale del progetto dal punto di vista concettuale (proprio la simulazione di diverse strategie rappresenta lo scopo di progetto), sia dal punto di vista funzionale (contando la quantità di classi). La particolarità di queste classi è proprio la presenza al loro interno degli algoritmi sui quali si basa il resto del programma visto che proprio questi algoritmi dovranno essere testati nel corso della simulazione.

Nel progettare le classi delle strategie vengono alla luce tre problemi:

1. quali strategie scegliere
2. la presenza di strategie alcune differenti, altre simili, cioè con funzionalità comuni
3. ci sono algoritmi che hanno bisogno di un testing accurato per assicurare la loro correttezza funzionale

Il primo problema è stato risolto nella fase di discussione delle scelte progettuali. Abbiamo selezionato solo le strategie che puntano su combinazioni esterne siccome la maggioranza delle strategie ha questa caratteristica.

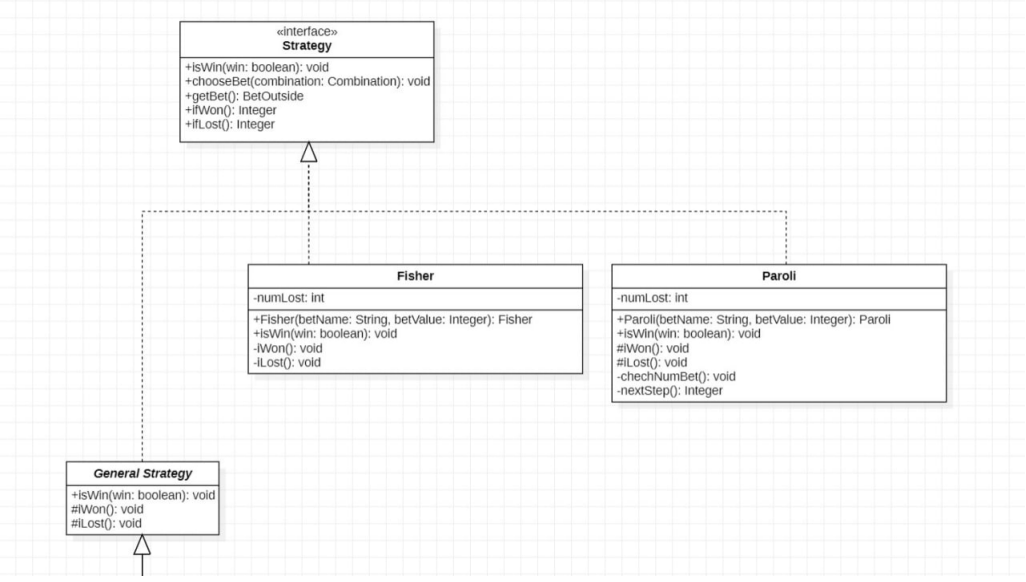
Il secondo problema si presenta nella elaborazione dei metodi delle strategie. Era evidente già nella prima fase di progettazione la necessità di una interfaccia per rappresentare tutti i metodi comuni di tutte le strategie, mentre la necessità di usare l'ereditarietà è apparsa evidente durante l'analisi approfondita del carattere delle strategie. Per esempio AntiMartingale estende Martingale, ContrLabouchere estende Labouchere, siccome usano le stesse funzionalità ma in situazioni diverse. Allo stesso tempo le classi Martingale e Labouchere estendono GeneralStrategy dal momento che sfruttano metodi comuni.

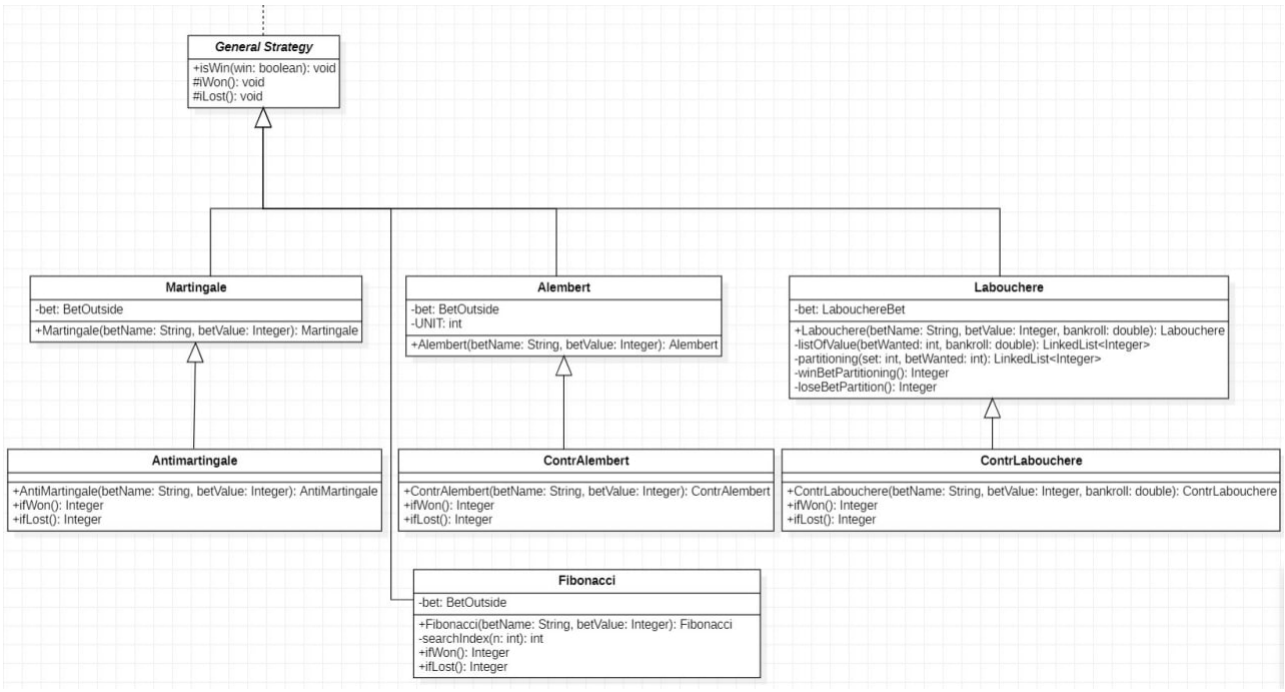
Il terzo problema è stato affrontato sia durante lo studio e sviluppo del algoritmo, sia durante la fase di testing.

StategyManager è una classe che assegna strategie al player, partecipando alla fase di creazione dell'entità Player. Come si vedere dall"UML la interfaccia Strategy fa da tramite tra il Player e combinazione esterne, quindi le puntate. Un'alternativa potrebbe essere sviluppare le strategie come classi utilities, ma visto l'importanza concettuale delle strategie nel progetto è stato deciso di utilizzare un approccio object oriented. In tal modo si è data più centralità alle strategie dotando ciascuna di una puntata (*Bet*), la quale altrimenti sarebbe stata inclusa nella classe *Player*.

Nello stesso tempo per motivi di validazione è stata presa una decisione in comune, ossia includere alcuni metodi delle strategie nelle classi Utilities (package model.main.utilities).

Come visto in precedenza questa progettazione delle classi permette l'utilizzo del design pattern Strategy.

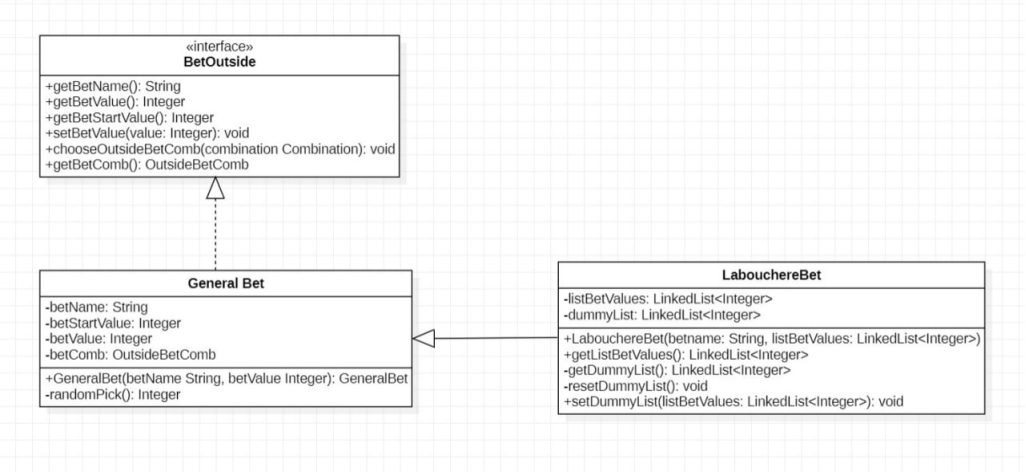




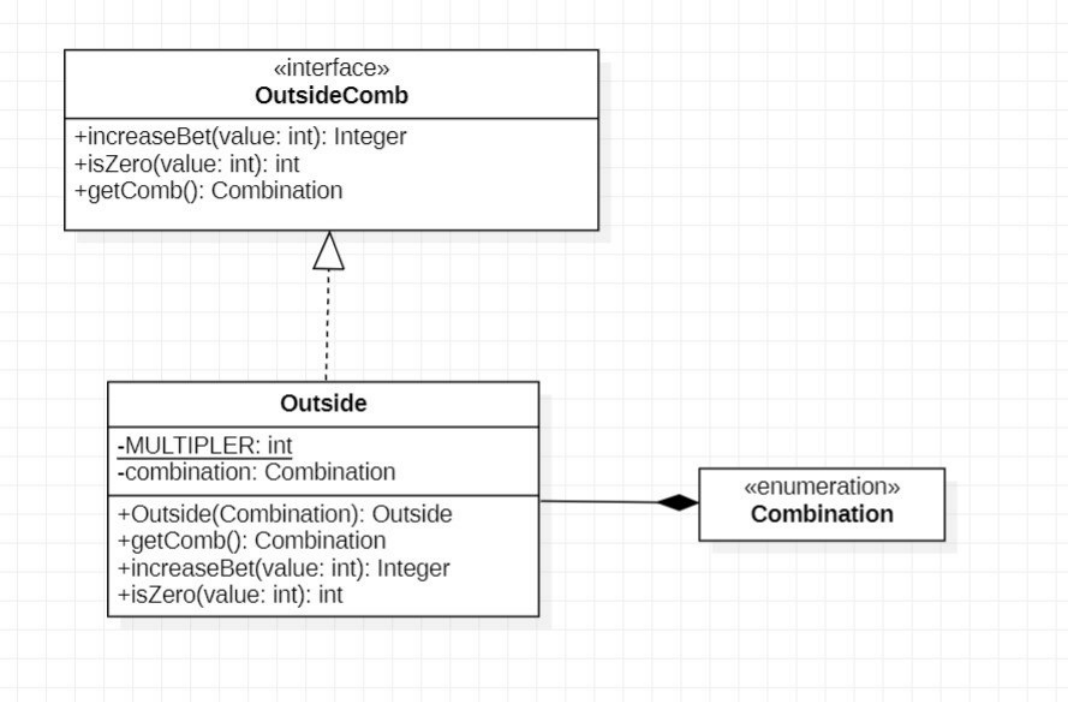
Bet

Nel package *Bet* viene gestita la parte del cambiamento del valore della puntata durante il gioco e la scelta della combinazione. In effetti i metodi principali della interfaccia si occupano dell'impostazione del valore iniziale della puntata e del valore della puntata finale (o corrente) e anche della scelta della combinazione da puntare (rosso, nero, pari, dispari etc).

È presente inoltre la classe *LabouchereBet* che estende *GeneralBet* per particolari esigenze della strategia stessa nella gestione della terminazione del gioco per il singolo player. È evidente che la strategia si esaurisce nel momento in cui il bankroll si azzera oppure la prossima puntata porta ad un bankroll negativo. L'entità Croupier richiede l'informazione riguardante la puntata successiva; nella Labouchere i valori delle puntate sono predefiniti (tramite un algoritmo che opera randomicamente) e si trovano in una lista. Per simulare la prossima puntata serviva una lista d'appoggio per visualizzare la puntata successiva senza modificare la puntata reale corrente che troviamo nella *LabouchereBet*.



Combination
Nell'interfaccia *OutsideBet* si trovano i metodi per definire la combinazione scelta (rosso, nero, pari, dispari etc) e il valore della vincita (increase bet) o del pareggio (is zero).



3 Sviluppo

3.1 Testing automatizzato

L'attività di testing è stata svolta utilizzando JUnit. I test non sono stati effettuati per ciascuna classe di ogni singolo package ma principalmente su un insieme di classi che presentano un sistema di operazioni articolato e il cui esito è di difficile previsione.
Lo sviluppo dei test ha seguito un approccio bottom-up: si sono verificati per prima i requisiti basilari del programma, come la creazione degli oggetti *Number*, per poi passare ad attività più corpose (costruzione dei *Player*) ed infine alle unità principali e complesse del sistema, ovvero le strategie.
Durante i test abbiamo cercato di "mettere alla prova" l'applicativo e molto spesso ci siamo trovati difronte a degli errori che, con un numero ristretto di operazioni (lato interfaccia utente), difficilmente sarebbero stati scovati.

3.1.1 Ramagnano Gabriele

Essendomi perlopiù cimentato in sezioni del programma riguardanti la creazione di entità (*Player*, *Number*), mi sono occupato di eseguire dei test volti ad appurare il corretto funzionamento delle factory-class. In particolare i test effettuati sulla *NumberFactory* mi hanno permesso di rilevare un'anomalia presente nel metodo utilizzato per il cambio dei colori.
Oltre a ciò ho ritenuto essenziale testare l'intero Model su un numero di giocatori molto elevato con l'obiettivo di misurare i tempi di risposta del sistema a fronte di una ingente quantità di dati in ingresso. Tale test si è rilevato utile per la cattura di un errore tanto nascosto quanto insidioso. L'errore rilevato era di natura "concorrente" (ConcurrentModificationException) e si verificava ogniqualvolta veniva estratto il numero zero. In breve, il programma si arrestava nel momento in cui veniva attraversata e modificata allo stesso tempo una lista di giocatori. In realtà successivamente si è notato che tale difetto si riproponeva anche nel metodo *manageWinners()* della classe *Croupier*; ma con una quantità molto piccola di giocatori in partita (non certo dell'ordine del migliaio), questa problematica non sarebbe probabilmente emersa così spontaneamente.

3.1.2 Volkova Kristina

Le strategie hanno richiesto molti test simili per testare principalmente come cambia il valore della puntata dopo la vincita o la perdita del player. In particolare, errori nel test della Alembert e ContAlembert hanno permesso di rilevare alcuni metodi inutilizzati della classe estesa. I test più difficili da realizzare sono stati quelli di Labouchere e ContrLabouchere dove non era possibile prevedere il valore della puntata data una vincita o una perdita. Perciò i test sono stati svolti solo in alcuni casi: all'uscita dello zero o per verificare che la puntata calcolata con valori casuali sia inferiore al Bankroll.

3.2 Metodologia di lavoro

Lo sviluppo del progetto ha seguito il tradizionale iter di analisi - progettazione - implementazione. Le prime due fasi, in particolar modo la progettazione, hanno richiesto riunioni abbastanza frequenti ed un lavoro a stretto contatto. In seguito ad una sostanziosa ricerca di informazioni per delineare al meglio lo spazio del problema e costruire il primo abbozzo di quello che sarebbe stato il modello del dominio, la fase di progettazione ha occupato un'importante fetta di tempo nel monte orario complessivo del progetto. Questa è stata la parte che, come già detto, ha richiesto più collaborazione, caratterizzata soprattutto da un prolifico sfogo di idee.
Ma una volta messo ordine e costruito per intero il diagramma UML finale, siamo riusciti ha suddividere il progetto in maniera tale che ciascuno lavorasse in maniera indipendente sull'insieme di classi assegnate.
Ovviamenteanche durante il corso dell'implementazione non sono mancati confronti e rapide, piccole modifiche al modello originale ma, in generale, gran parte della struttura è rimasta invariata e non ha subito ulteriori ritocchi.

3.2.1 Ramagnano Gabriele

L'inisieme di classi delle quali mi sono occupato è contenuto nei seguenti packages:

- Bankroll
- Table
- Croupier
- Player

Per quanto riguarda i primi tre il lavoro è stato svolto in sostanziale autonomia.
La classe *StrategyManager* , poichè legata strettamente con la classe *PlayerManager* dell'ultimo package, ha richiesto in fase implementativa un maggior confronto fra i componenti del team.
Sul resto dell'architettura (Controller, View) non c'è stata una particolare divisione dei compiti e per la maggior parte delle occasioni il codice è stato scritto insieme.

Version Control

Durante la costruzione del progetto si è utilizzato come DVCS Git. Ciò ha permesso uno scambio veloce e organico del materiale che è stato organizzato inizialmente su un unico branch per poi essere diviso in sotto-branch a seconda della suddivisione del lavoro precedentemente descritta. Il link è il seguente: https://github.com/volkovalolka/Roulette.git

3.2.2 Volkova Kristina

Le classi di cui mi sono occupata sono contenute in questi package:

- Strategy
- Bet
- Combination
- Main.Utilities

Per quanto riguarda le mie classi il lavoro in autonomia è stato svolto per progettare gli algoritmi delle strategie, mentre per quanto riguardo le Utilities è stato necessario un confronto in team.

3.3 Note di sviluppo

3.3.1 Ramagnano Gabriele

- **Stream**: ampiamente utilizzati nella classe *Croupier* per gestire le liste di giocatori. Hanno permesso la scrittura di un codice più sintetico ed elegante evitando strutture più "ingombranti" come *forEach* e l'utilizzo di continui *if*.
- **Lambda expression**: sfruttati esclusivamente nelle classi della *View*.
- **JavaFx**: mi sono occupato in particolar modo delle classi che legano il controller con la *View* (ad esempio, *InstructionView*) e dell'organizzazione generale delle medesime.

3.3.2 Volkova Kristina

- **Lambda expression**: sfruttati esclusivamente nelle classi della *View*.
- **JavaFx**: ho curato la parte visuale del progetto prestando particolare attenzione alla disposizione e interazione degli elementi in schermata. L'inserimento del grafico è stato il compito piu difficile da affrontare rispetto al resto della *View*.