Relazione per il progetto

Programmazione e modellazione ad oggetti

Michelangelo Ungolo

Sette Miriana

Docente Sara Montagna

2022/2023

Analisi

Lo scopo è quello di realizzare il famoso gioco del Monopoly in una versione cittadina dedicata ad Urbino, da cui ne proviene lo stesso nome del software: Urbinopoly.
Urbinopoly si ispira alla versione originale del gioco, dove, nel corso dei decenni, ogni Paese e comunità ha rilasciato la propria variante inerente al proprio territorio, motivo per cui si ritiene opportuno realizzare il famoso gioco da tavola in onore di Urbino.
Il gioco, essendo da tavola, ammette dai 2 ai 4 partecipanti che si riuniscono per prevalere sugli altri; infatti lo scopo principale è quello di arricchirsi maggiormente lungo il cammino sul *tabellone* portando gli avversari alla bancarotta. Gli elementi principali del gioco sono:

- I *dadi,* una volta lanciati a turno, indicano il numero di caselle che si devono attraversare dalla posizione corrente in cui ci si trova. Il gioco infatti presuppone che tutti i partecipanti partino dalla casella indicata dal via, per poi spostarsi lanciando i dadi uno per volta secondo un giro dei turni logico e invariato lungo tutta la partita.
  Durante il turno di un giocatore è possibile tirare una sola volta i dadi, a meno che entrambe le facciate mostrino lo stesso valore, in quel caso sarà possibile ritirare e continuare il proprio turno.
- Il *tabellone,* ovvero il percorso illustrativo che i partecipanti dovranno effettuare e tenere d'occhio per applicare le proprie strategie. Il tabellone contiene diverse caselle ognuna delle quali riserva comportamenti differenti che potranno essere, a seconda del momento della partita, vantaggiosi o meno.

Esso presenta un insieme di caselle che si possono raggruppare in:

- *Proprietà* che racchiudono tre diverse tipologie tra cui terreni, stazioni e servizi ognuno dei quali si differenzia dall'altro ma che in comune hanno diversi aspetti data ad esempio l'appartenenza alla stessa famiglia.

  Le proprietà, se libere, possono essere acquistate da chi ci cade con la propria pedina. Chi invece cade in una proprietà non sua deve pagare il prezzo di affitto.

  Le proprietà possono essere ipotecate non permettendo più i loro vantaggi ma permettendo d'altra parte di ricavare dalla banca la metà del prezzo d'acquisto speso. Si possono ipotecare sia case/hotel (elementi che contraddistinguono i terreni dalle stazioni e dai servizi), sia le intere proprietà. Per disipotecare però è obbligatorio restituire la somma ricevuta in precedenza dalla banca in aggiunta al 10% di tasse.

  Come già accennato nella famiglia delle proprietà si distinguono i *terreni* per la possibilità di costruire case (fino ad un massimo di quattro) ed hotel (unico e solo se si sono costruite precedentemente 4 case). La difficoltà però sorge nel fatto che in un terreno è possibile costruire se e solo se il giocatore ha già acquistato tutti i terreni dello stesso colore, ovvero ha ottenuto il monopolio o urbinopoly. Il grande vantaggio è che i ricavi per ogni casa costruita crescono vertiginosamente.

  Successivamente si hanno le *stazioni,* in totale quattro, che a seconda del numero di stazioni possedute, i ricavi raddoppiano. Infine i *servizi,* presenti in due sul

tabellone, hanno la particolarità che il ricavo corrisposto al giocatore che la possiede non dipende solo dal numero di servizi, ma esso è da moltiplicare al numero ottenuto dai dadi del giocatore che ci è caduto sopra.

- *Imprevisti* e *probabilità* sono caselle particolari che implicano di pescare una carta dal loro preciso mazzo che potrà presentare azioni positive o negative a seconda del messaggio contenuto, obbligatorio da rispettare.
- La *prigione* indica 3 turni di stop in cui si possono solo lanciare dadi; questo permette in caso di doppio valore uguale dei dadi di uscire direttamente. Un altro modo per scagionarsi è pagare una cauzione di 125€ oppure mediante una carta specifica trovata fortunatamente nelle carte probabilità o imprevisti.

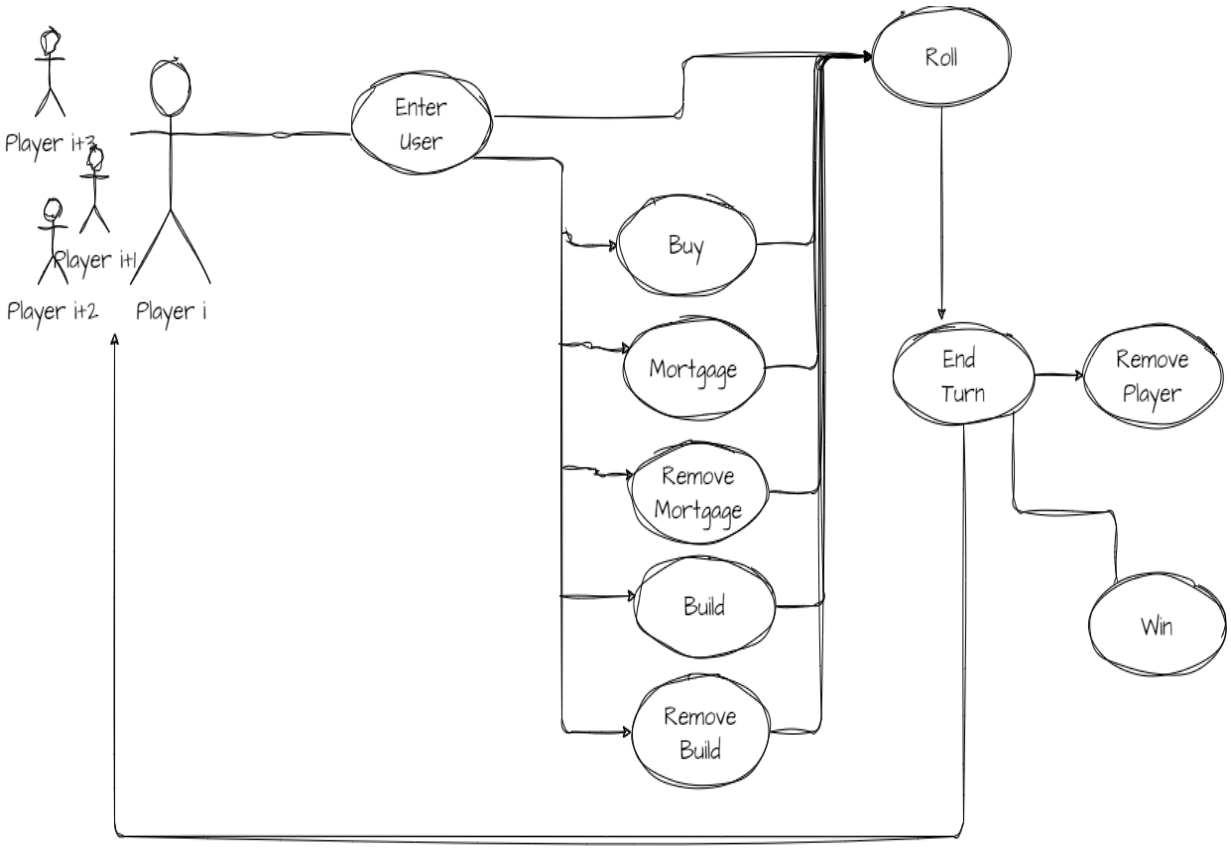
  E' importante scagionarsi il prima possibile poichè il giocatore che viene bloccato in prigione sarà impossibilitato a riscuotere il proprio bottino qualora un avversario faccia visita ad una delle sue proprietà.

  In prigione è possibile finirci in tre diversi modi: cadendo direttamente sopra la casella che indica la prigione, con tre lanci consecutivi dei dadi con valore delle facciate uguali oppure mediante carte probabilità o imprevisti che obbligano la carcerazione.
- *Tasse,* presenti due varianti, una su cui viene applicata una percentuale del 10% del bilancio complessivo, l'altra in cui vengono scalati 200€ netti.
- *Parcheggio,* invece unica casella dove non accade nulla.

Requisiti

I diversi requisiti funzionali che l'applicativo dovrà essere in grado di fare:

- I diversi partecipanti al gioco potranno indicare i propri username prima di procedere alla vera e propria partita.
- Il tabellone viene predisposto già in condizione di gioco in modo tale che la partita possa iniziare quando i giocatori lo ritengono opportuno.
- Ogni giocatore avrà a disposizione diverse mosse da compiere che saranno valutate a seconda della propria condizione, visualizzando solo quelle che può effettuare in quel dato istante.
- L'applicativo sarà in grado di gestire tutti quei comportamenti obbligatori che dovranno verificarsi per le regole del gioco
- Il tabellone sarà aggiornato costantemente ad ogni azione in modo tale che, in tempo reale, i giocatori possono avere tutto sotto controllo.


Player i+3
Player i+1
Player i+2
Player i
Enter User
Roll
Buy
Mortgage
Remove Mortgage
Build
Remove Build
End Turn
Remove Player
Win

Modello del dominio

Il software dovrà gestire una classica partita (Urbinopoly) garantendo che ogni evenienza di gioco sia conforme alle regole.

Ogni giocatore (Player) deve essere in grado di compiere tutte le mosse e le azioni a lui disponibili in modo tale che possa esprimere qualsiasi strategia di gioco che ritenga opportuna.

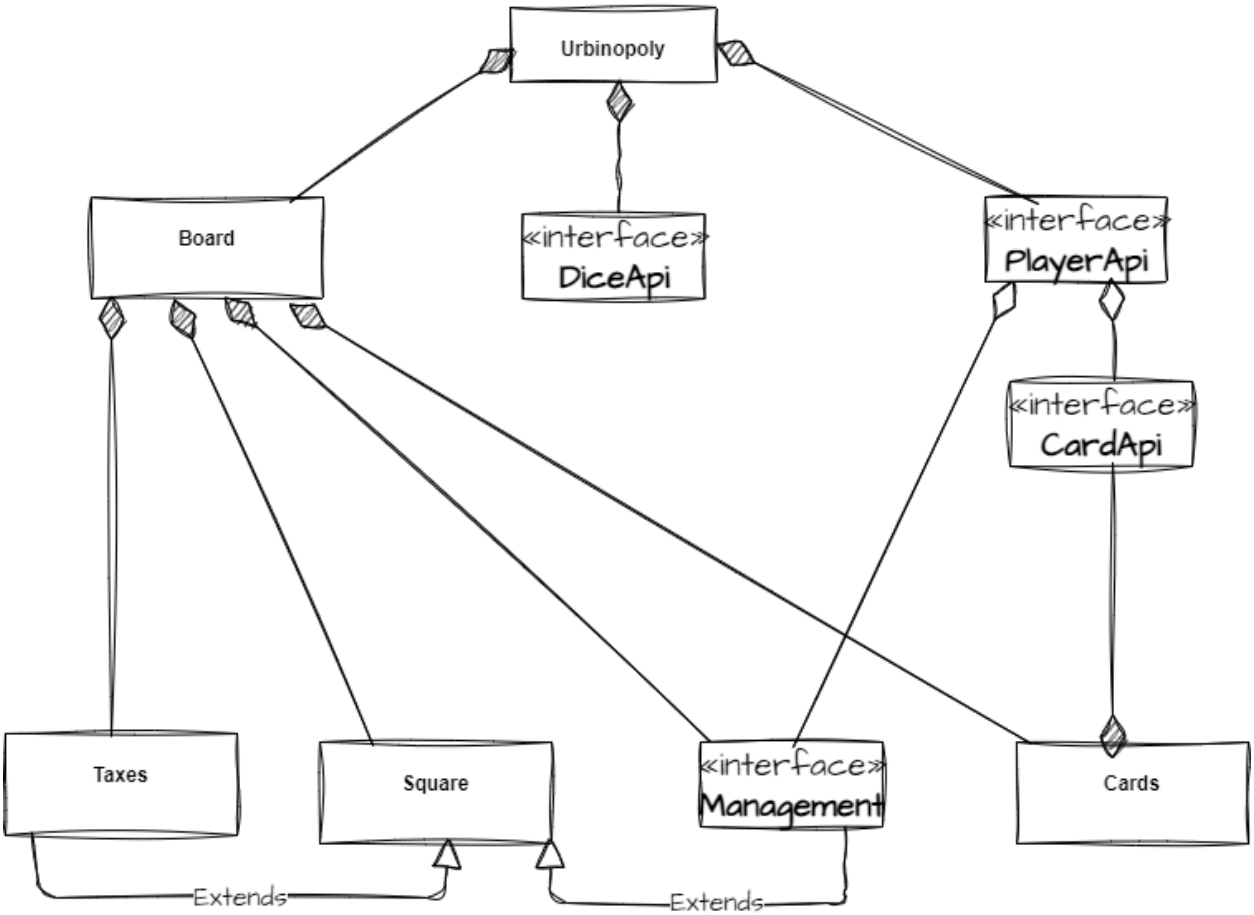
Ogni player potrà fare riferimenti a tutte le sue informazioni come il bilancio a disposizione e la posizione corrente in cui si trova in un dato momento della partita.

A caratterizzare quindi una partita sono proprio i giocatori, che insieme ai dadi (Dice) e il tabellone (Board) hanno tutto il necessario per procedere.

I dadi sono in grado di essere rollati per poter assegnare il proprio valore, che influirà direttamente sul giocatore stesso e sul tabellone di gioco.

Il tabellone Board contiene tutte le informazioni sui diversi quadrati che lo compongono (Square), mostrando ordine, tipo e valori di ogni singolo quadrato. Inoltre presenta i due mazzi di carte (Cards) delle probabilità e degli imprevisti per completare l'entità del tabellone.

La difficoltà del gioco è proprio quella di garantire che ogni azione sia legale ai suoi fini, azione a cui corrisponderà una reazione nel corso della partita dove le informazioni saranno in continuo aggiornamento.


Urbinopoly
Board
«interface»
DiceApi
«interface»
PlayerApi
«interface»
CardApi
Taxes
Square
«interface»
Management
Cards
Extends
Extends

Design

Architettura

Lo schema architetturale dell'applicativo Urbinopoly viene gestito da uno dei design pattern architetturali più utilizzati e che più si abbina all'intero software: il Model View Controller, (o abbreviato MVC). La scelta è ricaduta immediatamente su questa architettura per evidenziare e separare perfettamente il modello dalla vista, in modo tale che il software possa essere aggiornato a posteriori, senza modificare la parte di design ed inoltre, sarà possibile implementare una nuova interfaccia grafica senza la necessità di modificare il modello. In altre parole grazie a questo pattern architetturale si rendono assolutamente indipendenti la Gui e la logica del software.

Model

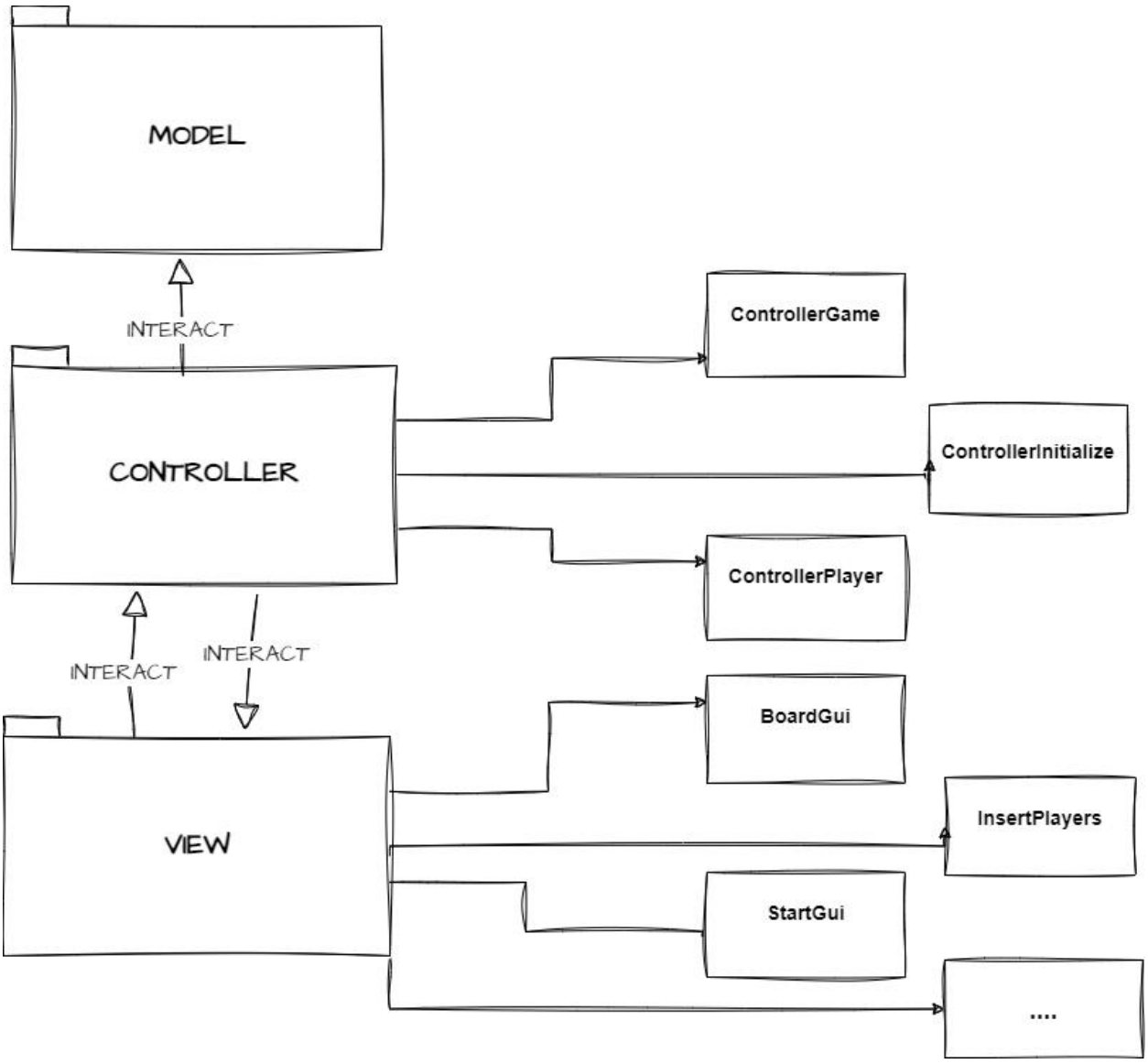
Come si evince dal diagramma di flusso rappresentato, il modello è nettamente separato dalla view e l'unico in grado di interagire con esso andando a scambiare le informazioni è il controller. Lo scopo principale del modello è quello di fornire un punto di accesso fornendo la propria logica di business, ovvero il core dell'elaborazione dei dati che rende possibile il funzionamento del software.

View

La vista o comunemente chiamata GUI è l'interfaccia grafica dedicata interamente all'utente, molto importante per rendere il nostro gioco più accattivante. E' importantissimo dedicare alla view questo unico compito, così da estraniarla da qualsiasi forma di logica e soprattutto evitare qualsiasi forma di interazione con il modello poiché altrimenti l'intera architettura verrebbe compromessa. Il compito della view quindi, oltre rendere appetibile l'interazione con l'utente, è quello di informare il controller di eventuali azioni dell'utente che comunicherà solo ed esclusivamente attraverso le diverse pagine della view che scorreranno durante una partita.

Controller

E' il mezzo con il quale il tutto riesce a comunicare, scambiando informazioni tra la vista e il modello, tra le azioni dell'utente e le reazioni dalla parte della logica. Il proprio comportamento quindi risulta tanto semplice quanto fondamentale per il funzionamento di tutta l'applicazione. Il controller sarà in costante ascolto sui comandi che l'utente segnalerà mediante la Gui e che, una volta ricevuti, verranno inviati al nostro modello che elaborerà le richieste o le informazioni relative e le riporterà finite al controller il quale avrà il compito di rispedire il messaggio alla view per poter essere aggiornata. Con questi passaggi l'utente sarà sempre a conoscenza dell'evoluzione del gioco, poiché ad ogni sua mossa tramite Gui sarà sempre corrisposta una reazione da parte del modello.


MODEL
INTERACT
CONTROLLER
ControllerGame
ControllerInitialize
ControllerPlayer
INTERACT
INTERACT
VIEW
BoardGui
InsertPlayers
StartGui
....

Design dettagliato

Ungolo Michelangelo

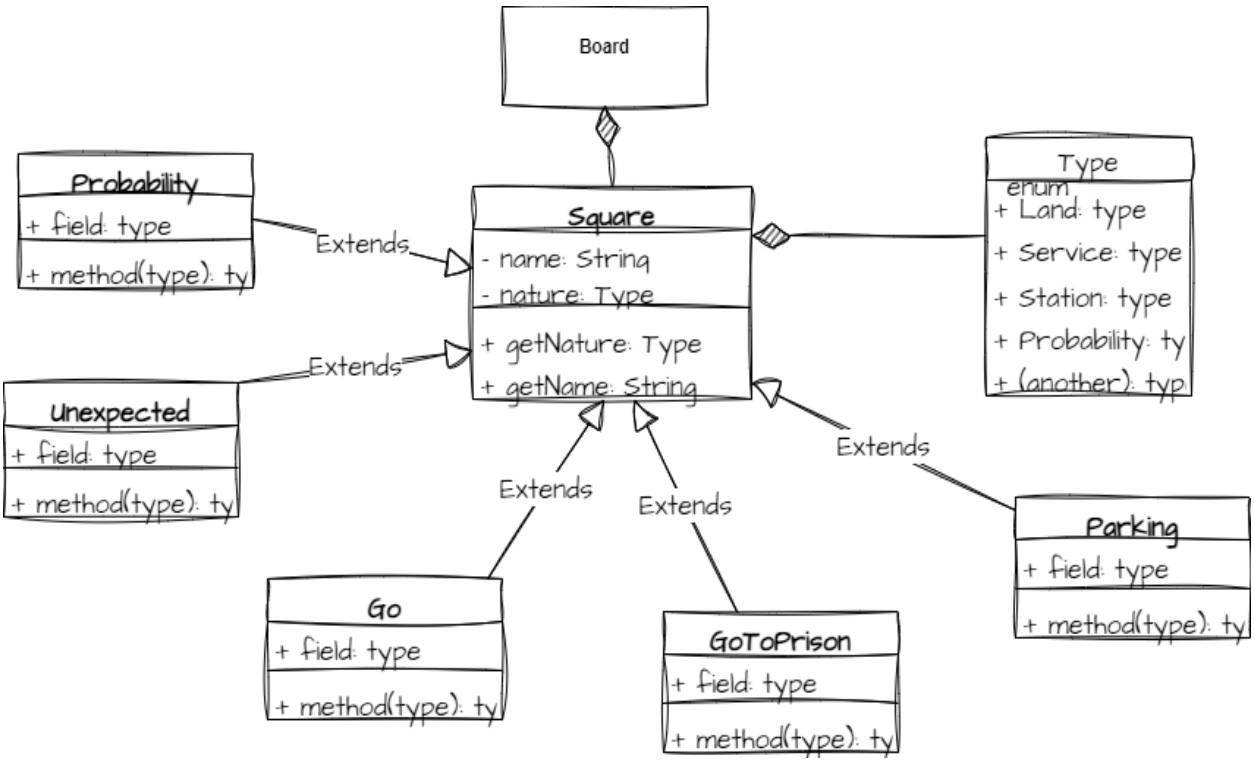
Square Problem

Il primo passo verso l'implementazione del software è stato quello di identificare la struttura dell'applicazione, quindi il primo approccio è stato realizzare i diversi quadrati che avrebbero composto il nostro tabellone, pensandoli come scheletro di tutto il gioco.
Per la realizzazione di tutti i quadrati l'idea è ricaduta fin da subito nell'implementazione di una classe padre che potesse racchiudere in sé l'insieme di tutti i quadrati, infatti tutti gli Square avevano in comune 2 caratteristiche: il nome e la loro natura.
Per quanto riguarda la natura di un quadrato è stato semplice pensare alla realizzazione di una classe enum che andasse a racchiudere tutte le tipologie delle caselle presenti nel gioco.
A livello implementativo Square è stata realizzata come classe astratta per due motivi principali:

- in primo luogo sarebbe stato molto efficace utilizzare il polimorfismo per rappresentare l'intero tabellone come un insieme di Square così che ogni quadrato appartenesse alla stessa famiglia, come logicamente verrebbe da immaginare. Quindi l'idea è stata quella di realizzare tutti gli altri quadrati, distinti l'uno dall'altro, andando ad ereditare dalla classe Square così da poter utilizzare il polimorfismo.

- In secondo luogo non occorreva istanziare oggetti di tipo Square dato che avremmo disposto successivamente di oggetti ad hoc per i loro compiti all'interno del gioco, quindi una classe astratta sposava perfettamente il concetto pensato per la realizzazione della base di tutto il progetto.

Altri tipi di quadrati come le caselle Probability, Unexpected, Go, Parking, Prison e GoToPrison non necessitano di ulteriori implementazioni rispetto a tutto ciò che era definito già nella classe Square, quindi sono state solo personalizzate rispetto al loro nome e la loro natura.



Property Problem

Come pezzi del tabellone e come identità principale del gioco si è dovuto implementare la famiglia delle proprietà composta da terreni, stazioni e servizi.

Si è pensato che queste ultime sono tutte proprietà con caratteristiche diverse l'una dall'altra in termini di strutturazione interna e concetto di ricavi, ma con appartenenza allo stesso insieme; questo ha portato all'immediata implementazione.

Anche in questo caso tramite ereditarietà si sono intersecati i concetti di Property con le classi da esso dipendenti Land, Station e Service. E' stata realizzata una classe Property per andare a racchiudere tutti i concetti uguali che ogni proprietà riserva in quanto tale: si identifica ad esempio la logica di compera, ipoteca e disipoteca, e di appartenenza o meno ad un determinato acquirente o proprietario.

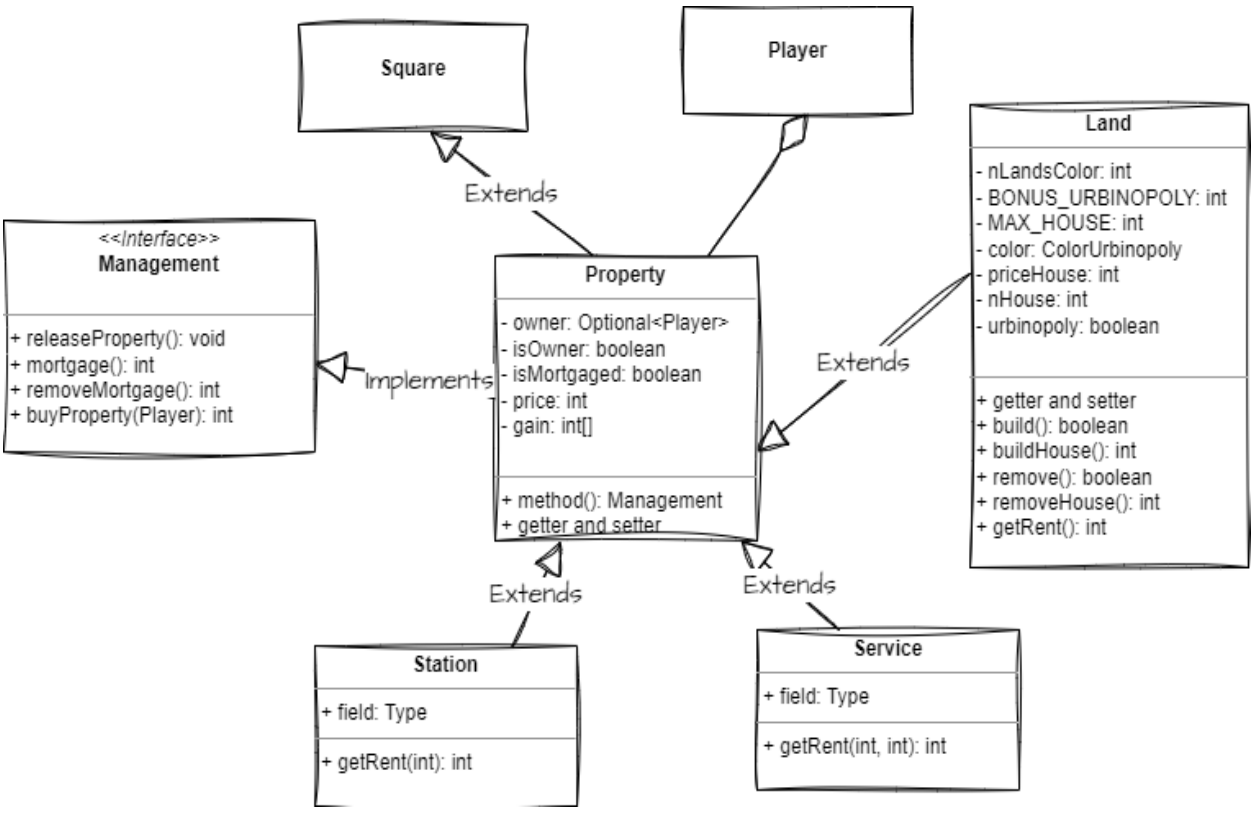
In questo caso l'implementazione non è ricaduta su una classe astratta poiché le proprietà non racchiudono poche informazioni, come Square (che ha proprio il senso di concetto astratto), ma molto della logica di gioco, che le ha rese in quanto tali istanze vere e proprie per il controllo e la loro stessa gestione.

Come anticipato per ogni proprietà si hanno dei concetti diramati che si riducono nella creazione delle sue dirette sottoclassi, riferito per l'appunto ai terreni, stazioni e servizi.

Per tutte le loro classi viene evidenziato il concetto di ereditarietà poiché prenderanno tutte le informazioni dalla classe Property. Quelle che saranno nel corso del gioco le istanze Land ovvero i terreni, sono state fornite del concetto di costruzione: infatti sarà possibile costruire,

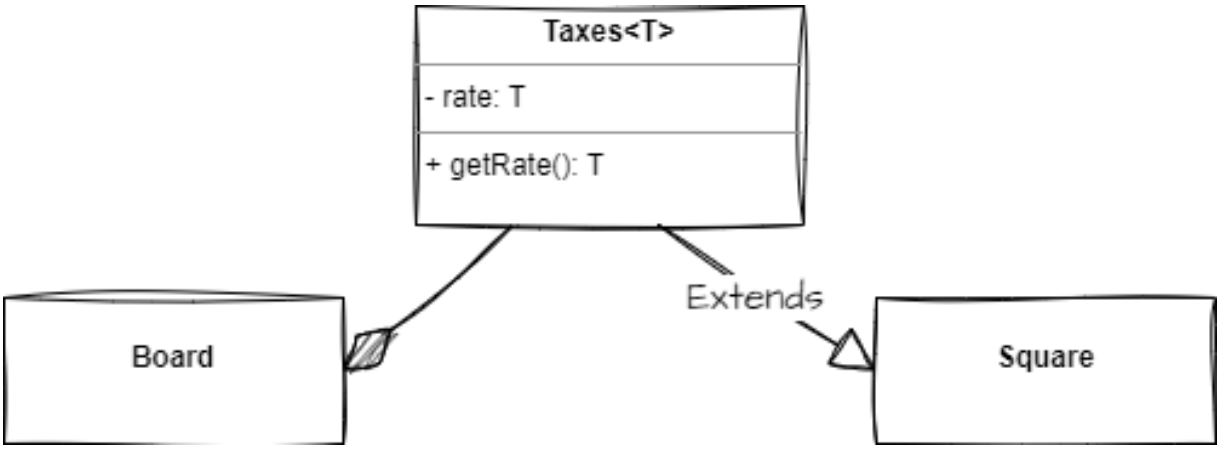
ipotecare e disipotecare le stesse case/hotel comprate. Dall'analisi fatta in precedenza per realizzare tutto ciò si è pensato al concetto di urbinopoly dove, un player possiede un monopolio se e solo se è il proprietario di tutti i terreni dello stesso colore.

In seguito per tutte le classi figlie di Property viene ridefinito il concetto di ricavo, dove viene definito per ognuna: i terreni derivano il loro ricavo dal numero di costruzioni e dal loro monopolio, le stazioni invece moltiplicano il loro ricavo rispetto al numero di stazioni possedute dallo stesso proprietario e i servizi moltiplicano il numero di servizi posseduti al valore dei dadi del player che cade sul servizio.



Taxes Problem

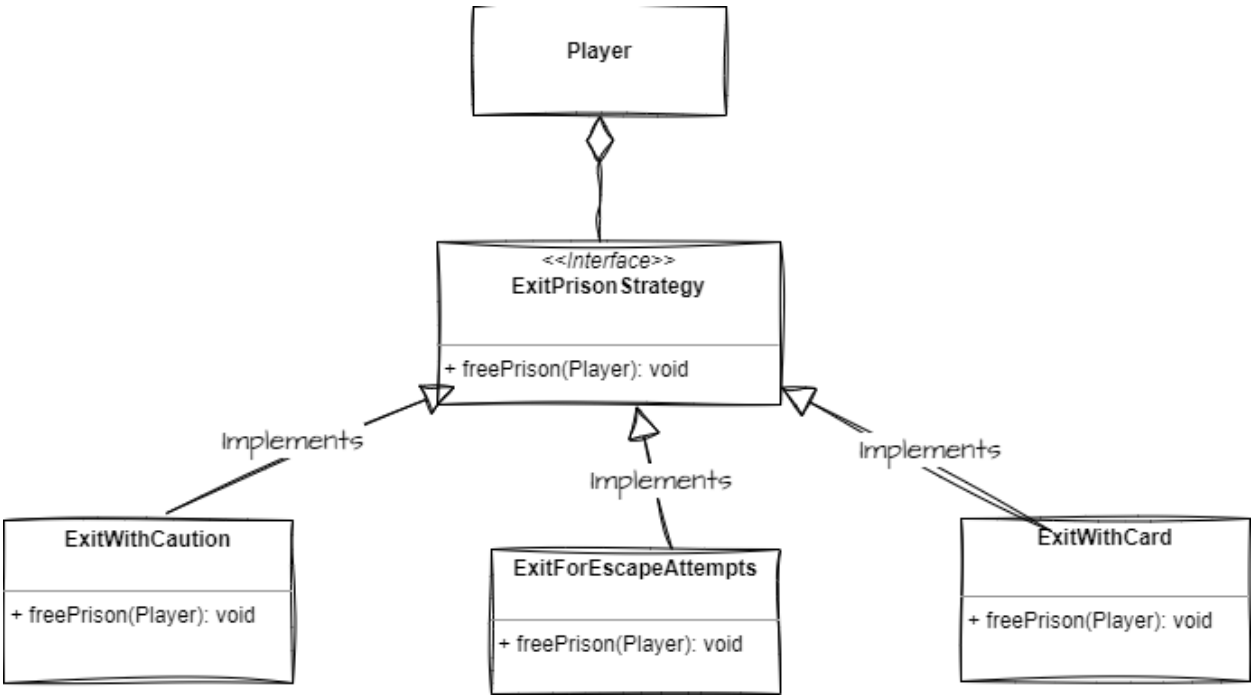
Le tasse come altra istanza del tabellone si scompongono nelle tasse sul patrimonio e una tassa fissa. Si differenziano quindi per la loro attribuzione ma data la loro caratteristica comune in quanto tasse, si è proceduto con l'implementazione di una classe generica per modellare la loro forma. Come si può vedere dal grafico uml seguente, il tutto sarà fatto girare intorno al parametro generico definito, che porterà alla creazione di una classe con parametro Double per le tasse sul patrimonio in quanto calcolano la percentuale di pagamento sul bilancio corrente del player, mentre si userà la classe wrapper Integer per l'assegnazione e la creazione della tassa fissa.



Prison

La prigione è un elemento del tabellone, fondamentale poiché definisce una grande fetta dell'intero gioco. Analizzando il modello però si è deciso di separare la casella dedicata per la prigione, definita all'interno di Square, dalla sua logica poiché essa dipende esclusivamente dal Player interessato che, durante il suo i-esimo turno, qualora si trova in prigione, può decidere in quale modo gestire le proprie mosse come da descrizione delle regole di gioco.
Da questa analisi ne deriva la sua diretta implementazione all'interno della classe Player; infatti viene utilizzato il noto design pattern Strategy che mediante la propria interfaccia comunica il metodo pubblico per uscire di prigione mediante diversi modi i quali saranno implementati nelle classi che si occuperanno di implementare le diverse strategie secondo la logica di gioco.
Questo ha permesso di semplificare molto la classe del Player, una delle classi principali per l'economia dell'intero gioco, delegando, alle classi che seguono, il compito di implementare la corretta strategia della logica della prigione attraverso il giocatore.
Si forniscono così le classi che implementano le 3 diverse strategie:

- ExitForEscapeAttempts: qualora il Player decidesse di usarla, tenta la scarcerazione tramite i dadi, provando il doppio numero uguale, dove nel peggiore dei casi attenderà i 3 turni di stop
- ExitWithCard: tale strategia potrà essere utilizzata dal Player se possiede una delle carte probabilità o imprevisti che regalano l'uscita automatica
- ExitWithCaution: il giocatore viene scarcerato pagando obbligatoriamente la cauzione



Controller

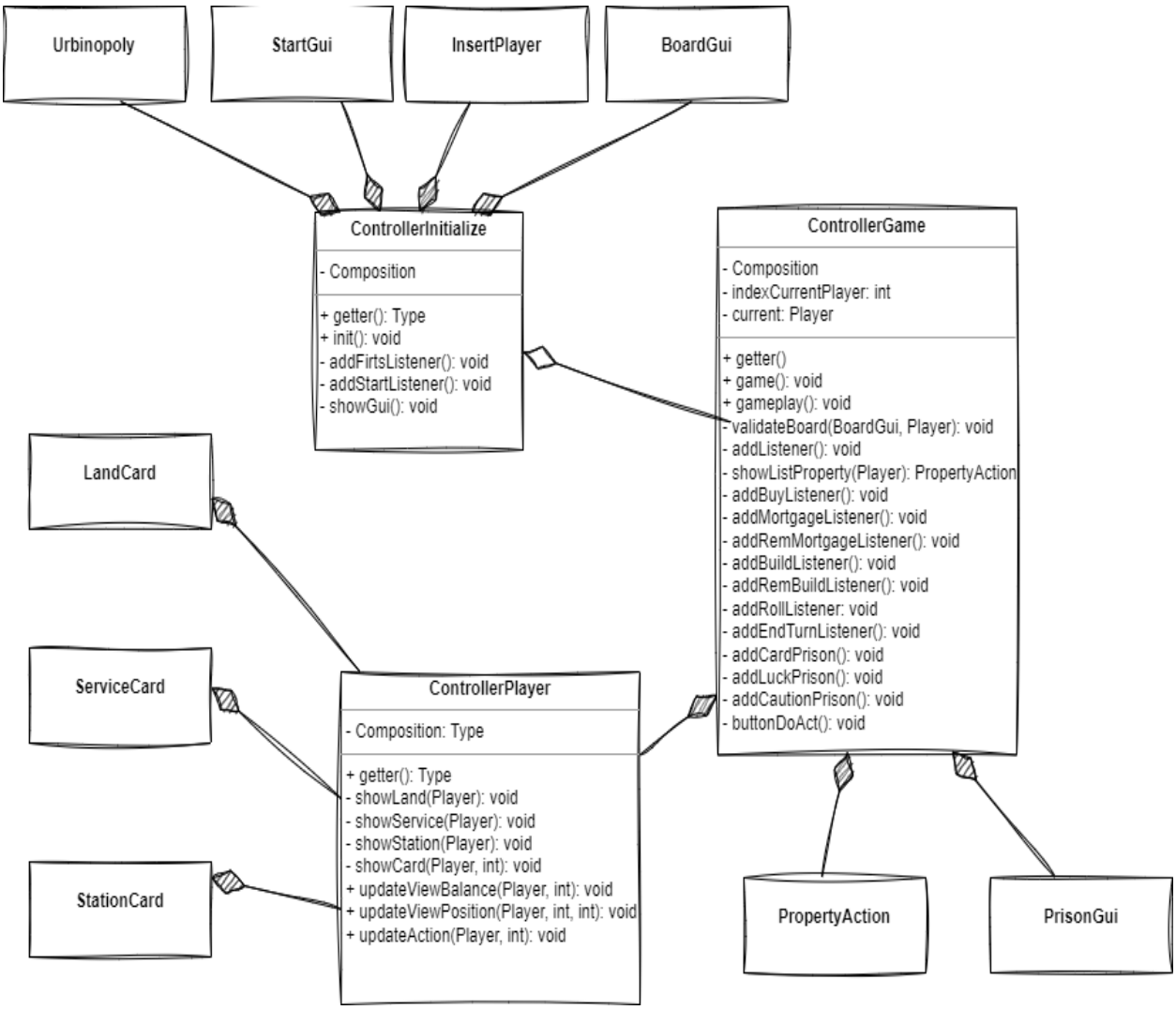
Il controller come già spiegato in precedenza ha il compito di far comunicare la view e il controller mediante la propria architettura rendendo allo stesso tempo indipendente la view dal modello e viceversa; questo è stato possibile mediante il pattern MVC.

Dato un applicativo non banale e con tante situazioni da controllare ho optato per la realizzazione di tre classi con la funzione di controllore:

- Controller Initialize per andare a rendere operativo tutto il sistema di gioco, un controller che si occupa solamente di creare le istanze di gioco e le diverse finestre di interfaccia utente in modo tale da delegare l'inizializzazione ad un intero controller, responsabile di

solo questo compito per l'avvio della partita di gioco. Sarà in grado quindi di avviare le 3 finestre principali della view e creare il tabellone, donando sia la relativa view che il modello con la lista dei giocatori indispensabile per la propria inizializzazione.

- Controller Game sarà il controllore dell'intera partita di gioco che quindi avrà il compito di sincronizzare il modello con la sua logica di business con la successiva reazione alla finestra di gioco con costanti aggiornamenti. Tale controllore aggiunge ascoltatori presso i bottoni istanziati nella view per recepire le informazioni volute dall'utente e trasformarle in una conseguenza per il modello dove successivamente il controller invierà le informazioni elaborate alla view per il proprio aggiornamento. Questa è la tecnica principale utilizzata nel model view controller per sincronizzare ed è la più adatta per l'intera applicazione nella sua semplicità.
- Controller Player invece è il terzo e ultimo controllore che viene realizzato per controllare tutte le azioni del giocatore separatamente, scelta dovuta soprattutto per leggibilità del codice e per separare i diversi compiti nel controller non delegando all'unica classe la responsabilità di tutta la sincronizzazione di una partita di gioco.
  In questo modo questo controller regola tutte le azioni che vedono l'elaborazione delle informazioni per il giocatore i-esimo. Dato che la propria sincronizzazione è parte integrante del Controller Game viene naturale che quest'ultimo si componga di esso in modo tale da usufruire di tutte le sue funzionalità.


Urbinopoly
StartGui
InsertPlayer
BoardGui
ControllerInitialize
- Composition
+ getter(): Type
+ init(): void
- addFirtsListener(): void
- addStartListener(): void
- showGui(): void
ControllerGame
- Composition
- indexCurrentPlayer: int
- current: Player
+ getter()
+ game(): void
+ gameplay(): void
- validateBoard(BoardGui, Player): void
- addListener(): void
- showListProperty(Player): PropertyAction
- addBuyListener(): void
- addMortgageListener(): void
- addRemMortgageListener(): void
- addBuildListener(): void
- addRemBuildListener(): void
- addRollListener: void
- addEndTurnListener(): void
- addCardPrison(): void
- addLuckPrison(): void
- addCautionPrison(): void
- buttonDoAct(): void
LandCard
ServiceCard
StationCard
ControllerPlayer
- Composition: Type
+ getter(): Type
- showLand(Player): void
- showService(Player): void
- showStation(Player): void
- showCard(Player, int): void
+ updateViewBalance(Player, int): void
+ updateViewPosition(Player, int, int): void
+ updateAction(Player, int): void
PropertyAction
PrisonGui

Sette Miriana

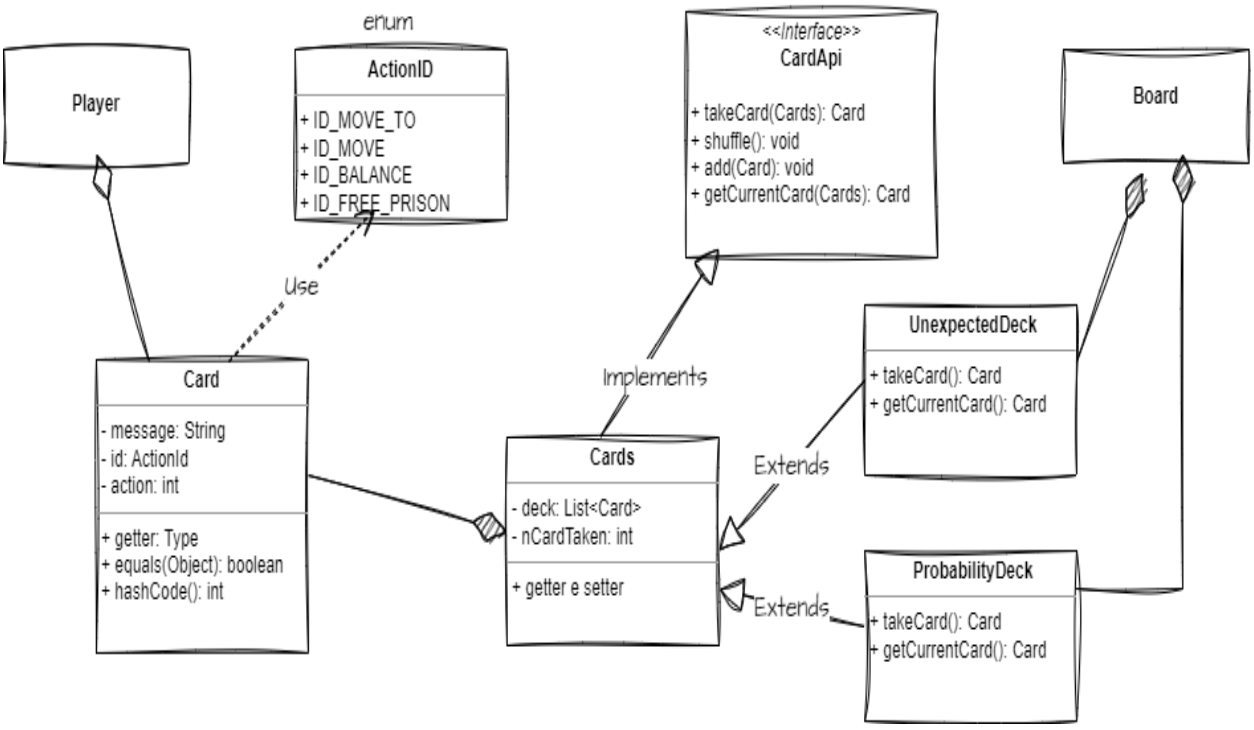
Cards Problem

I mazzi di carte sono un elemento protagonista sul tabellone di gioco, con la caratteristica che ogni carta nel suo mazzo ha un compito differente dall'altro. Lo scopo è quello di creare due mazzi di carte tra probabilità e imprevisti, dinamici, ovvero capaci di restituire una carta differente ad ogni pescata a simulare un vero mazzo.

Si è notato che occorreva prima definire la carta in quanto tale e poi comporla nel suo mazzo generico da cui si specializzano successivamente i due mazzi del gioco. Durante la fase di modellazione non occorrerà l'utilizzo dei soli mazzi specifici al gioco, ma sarà estremamente utile anche la modellazione della singola carta o deck generico per l'elaborazione delle singole informazioni.

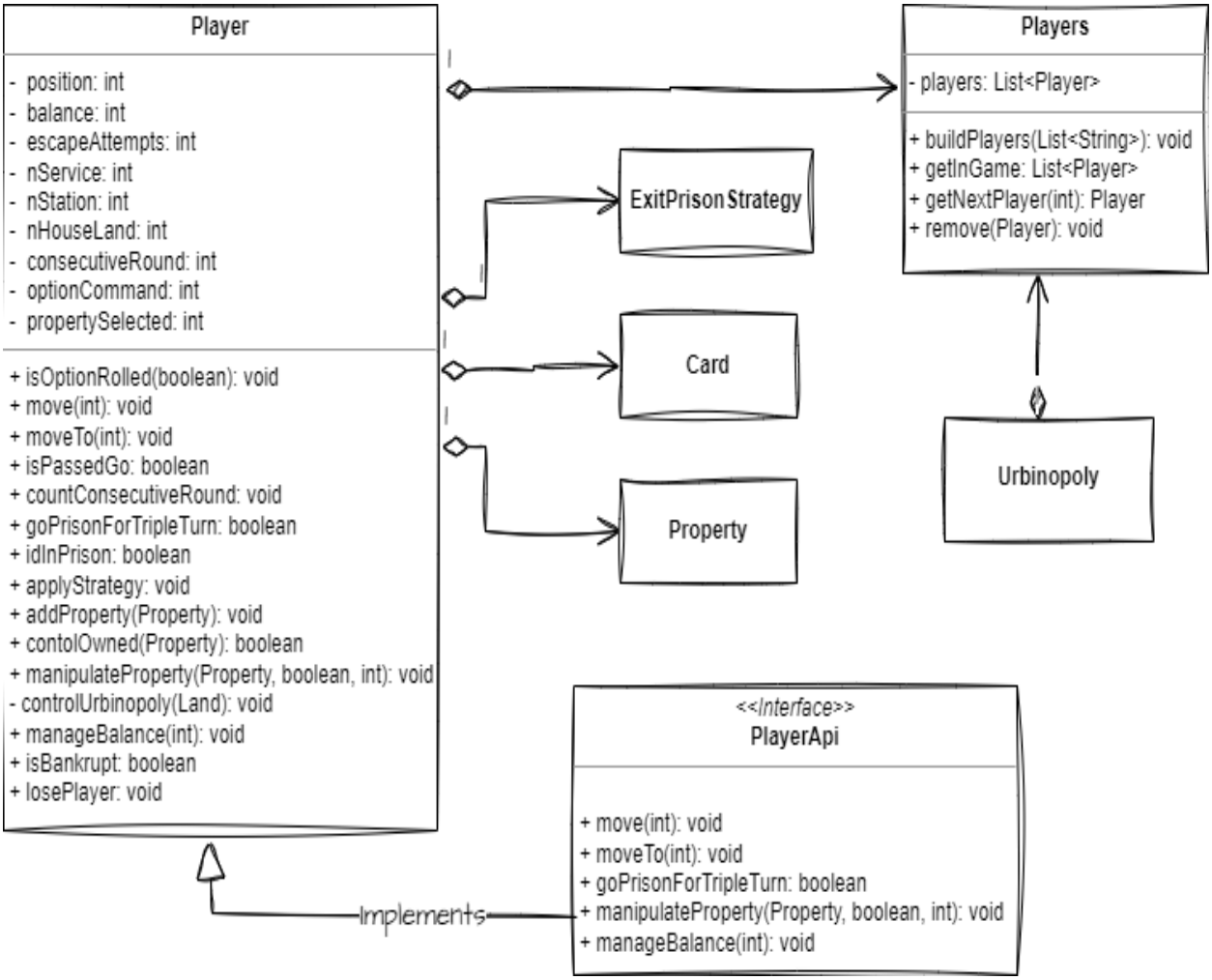
La carta viene dotata di un codice id molto importante per distinguerla dalle altre mediante la sua caratteristica più importante; infatti tutte le carte estratte da un deck avranno un'azione, identificata proprio dal codice id dato dai tipi enum. Questo permette non solo di confrontare le diverse carte tra loro, ma anche di associare la singola carta estratta al suo preciso compito durante il gioco.

Il mazzo di carte invece va a definire il comportamento che i due mazzi poi avranno, in particolare stabilisce la logica di pescaggio di una carta e del mescolamento di un deck, in modo tale che il tutto possa essere randomico come una vera partita.


Player
enum
ActionID
+ ID_MOVE_TO
+ ID_MOVE
+ ID_BALANCE
+ ID_FREE_PRISON
<<Interface>>
CardApi
+ takeCard(Cards): Card
+ shuffle(): void
+ add(Card): void
+ getCurrentCard(Cards): Card
Board
Use
Card
- message: String
- id: ActionId
- action: int
+ getter: Type
+ equals(Object): boolean
+ hashCode(): int
Implements
Cards
- deck: List<Card>
- nCardTaken: int
+ getter e setter
UnexpectedDeck
+ takeCard(): Card
+ getCurrentCard(): Card
Extends
ProbabilityDeck
+ takeCard(): Card
+ getCurrentCard(): Card
Extends

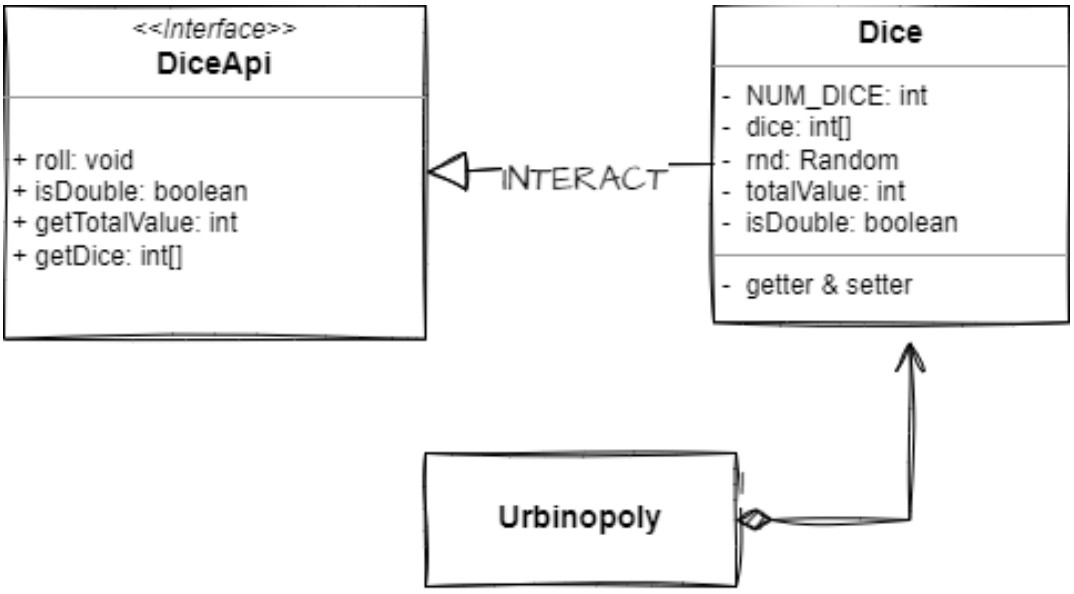
Player

Come per ogni gioco che si rispetti, particolare attenzione è da prestare ai giocatori che si interfacciano su esso. Infatti sono loro ad essere la chiave di tutto, ogni loro scelta e azione deve essere seguita con precisione per garantire il giusto flusso di esecuzione.
Il singolo giocatore ha dalla sua parte molte azioni che può effettuare ad ogni singolo turno poiché appunto andrà a stabilire la propria strategia, potendo pensare a qualsiasi mossa possibile. In particolare il giocatore dovrà essere in grado di muoversi sul tabellone di gioco attraverso movimenti circolari, dovrà poter manipolare il proprio bilancio e applicare le proprie strategie come ad esempio quelle relative alla prigione.
Il tutto viene fatto senza troppe difficoltà avendo donato a tale entità tutto l'occorrente per essere modellato, come le collezioni delle sue proprietà e carte pescate, o le condizioni di monopolio e carcerazione.
L'unico parametro che occorrerà per inizializzare un Player è proprio il nome che sarà dato a runtime proprio dall'utente che vorrà iniziare una partita.
Proprio da questo nasce l'idea del multiplayer, ovvero la necessità di far comunicare i 2-4 giocatori assieme sul tavolo di gioco virtuale in modo tale da rendere realistico l'intero sviluppo. Infatti viene garantita una loro collaborazione attraverso una classe più astratta che prevede la loro gestione come insieme, in modo tale da creare un sistema che potesse coordinare i turni e le possibili eliminazioni al termine del gioco. Il tutto viene gestito principalmente da due enti come viene mostrato in seguito.


Player
- position: int
- balance: int
- escapeAttempts: int
- nService: int
- nStation: int
- nHouseLand: int
- consecutiveRound: int
- optionCommand: int
- propertySelected: int
+ isOptionRolled(boolean): void
+ move(int): void
+ moveTo(int): void
+ isPassedGo: boolean
+ countConsecutiveRound: void
+ goPrisonForTripleTurn: boolean
+ idInPrison: boolean
+ applyStrategy: void
+ addProperty(Property): void
+ contolOwned(Property): boolean
+ manipulateProperty(Property, boolean, int): void
- controlUrbinopoly(Land): void
+ manageBalance(int): void
+ isBankrupt: boolean
+ losePlayer: void
Players
- players: List<Player>
+ buildPlayers(List<String>): void
+ getInGame: List<Player>
+ getNextPlayer(int): Player
+ remove(Player): void
ExitPrisonStrategy
Card
Property
Urbinopoly
<<Interface>>
PlayerApi
+ move(int): void
+ moveTo(int): void
+ goPrisonForTripleTurn: boolean
+ manipulateProperty(Property, boolean, int): void
+ manageBalance(int): void
Implements

Dice

I dadi vengono creati attraverso un array poiché l'unica cosa di cui hanno bisogno è un accesso rapido ed efficiente per fornire i risultati velocemente. Il tutto viene fatto girare attraverso un oggetto dell'Api che produce numeri casuali in modo tale da simulare la randomicità del gioco. Inoltre vengono implementati i controlli per verificare se le due facciate dei dadi presentano valori uguali, caratteristica che occorre conoscere per molte fasi del gioco. Questo permette di avere sempre a disposizione, dopo il lancio dei dadi, i valori a conoscenza grazie al contratto stabilito, senza dover implementare ulteriori passaggi all'occorrenza.



Board

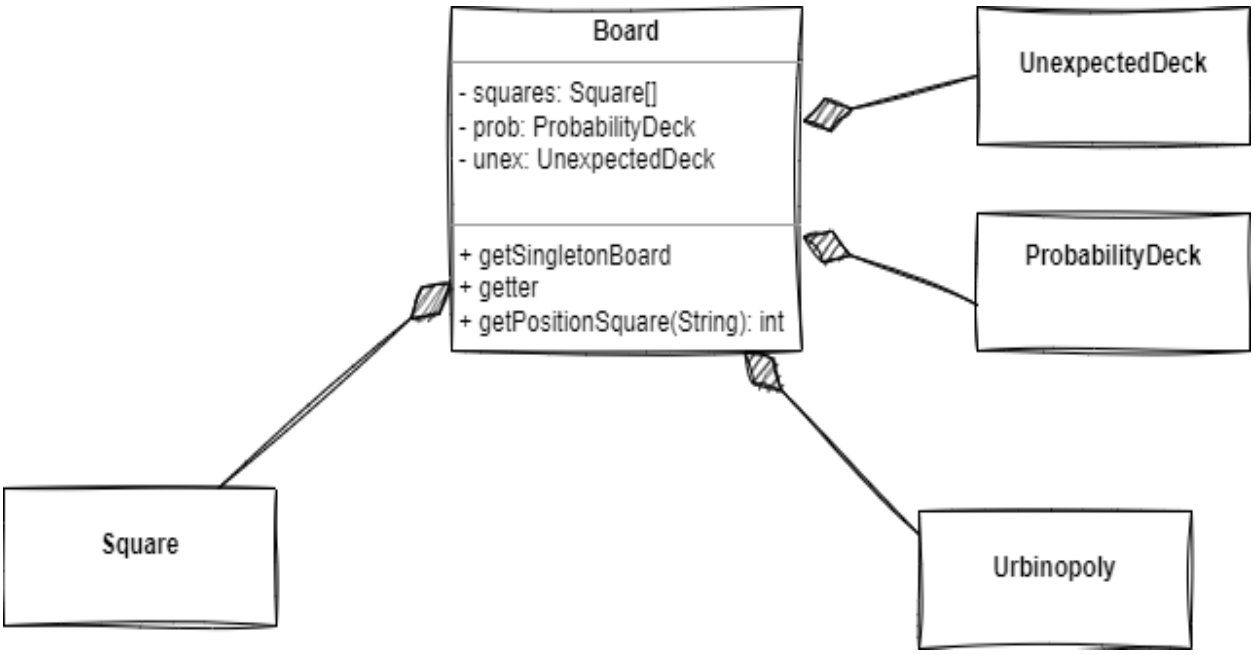
Si punta a creare il tabellone di gioco che dovrà racchiudere tutte le istanze presenti su di esso. Pensando a diverse strategie è stata scartata l'idea di rendere il tabellone una collezione di oggetti ognuno diverso dall'altro poiché questo passaggio richiedeva troppi controlli superflui che andavano a peggiorare le leggibilità del codice.

Si è optato quindi per l'uso prevalente del polimorfismo grazie all'ereditarietà proposta dalla classe padre di tutti i pezzi descritta in precedenza. Si riesce così a creare un tabellone in cui ogni pezzo ad esso collegato è dello stesso tipo, inizializzato poi attraverso la propria entità. Questo ci permette una manipolazione dei pezzi in ogni momento di gioco avendo tutto l'occorrente per la lavorazione.

Grazie alle funzionalità che il tabellone condivide con l'esterno, sarà possibile ricevere sempre il quadrato corrente mediante la posizione che si desidera. Questo permette una grande velocità di risposta in un dato momento, usato ad esempio per la modellazione di una fase di gioco a seconda della casella occupata da un player.

Proprio da questo pensiero ne deriva il fatto che il tabellone viene implementato mediante un array per ottenere una richiesta di accesso in un tempo costante considerando che il tabellone di gioco ha un numero fisso di caselle che non potrà mai cambiare nel tempo.

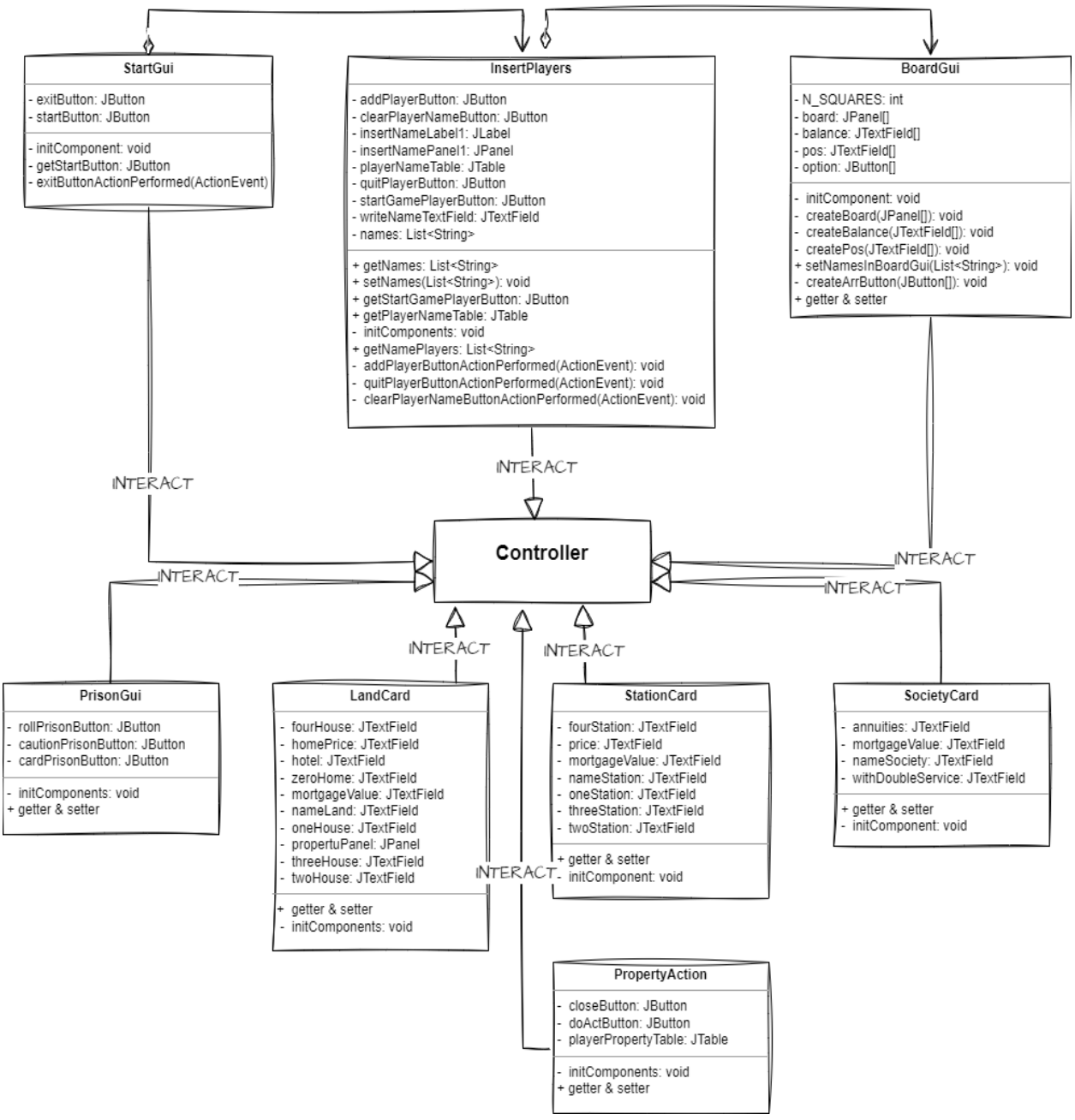
Si garantisce inoltre grazie all'applicazione del pattern Singleton che il tabellone di gioco sia l'unica entità presente nell'applicativo: si possono pensare a più entità dei dadi, player, pezzi, ma sicuramente non è possibile avere due tabelloni in quanto unico nel gioco.


Board
- squares: Square[]
- prob: ProbabilityDeck
- unex: UnexpectedDeck
+ getSingletonBoard
+ getter
+ getPositionSquare(String): int
UnexpectedDeck
ProbabilityDeck
Square
Urbinopoly

Gui

Per realizzare l'architettura citata in precedenza, mi sono occupata dell'elemento che sicuramente ha il compito di attirare l'attenzione dell'utente, l'interfaccia grafica.
Subito dopo aver completato a dovere la parte dedicata al modello, ho iniziato a pensare alle diverse finestre di gioco che si sarebbero dovute presentare nel nostro applicativo. Il tutto è realizzato attraverso la libreria Swing di Java che contiene tutto il necessario e oltre per la realizzazione di giochi 2d come questo.
Sono presenti 3 videate principali: una pagina di benvenuto, una dedicata all'inserimento dei giocatori e l'ultima invece al tabellone di gioco che è sicuramente il più complesso da realizzare.
Lo scheletro del sistema è stato realizzato mediante l'IDE di sviluppo che ha reso possibile l'inserimento e soprattutto la posizione degli elementi in maniera rapida e precisa.
Nel tabellone di gioco sono state realizzate altre videate che permettessero la comunicazione con il tabellone ogni qual volta si presentassero situazioni di carcerazione o caduta su proprietà.
Questo ha permesso di migliorare non solo l'aspetto estetico del software ma anche di garantire una sincronizzazione efficiente offrendo nuovi bottoni per la comunicazione nella nostra architettura.
Nelle diverse classi che rappresentano la vista vengono in ogni caso realizzate delle collezioni di oggetti dello stesso tipo, come JText relativi allo stesso insieme di operazioni, in modo tale che nella sincronizzazione del controller, questo potesse facilitare l'elaborazione di alcune operazioni

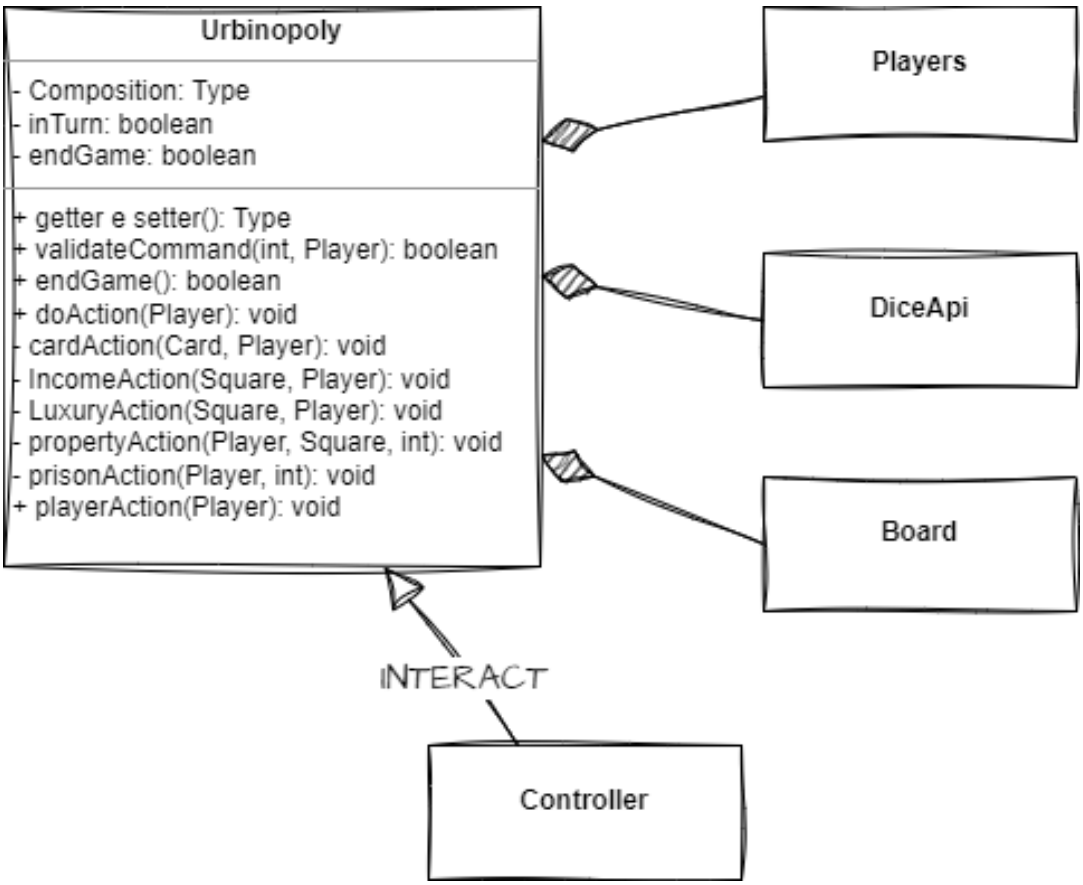
mediante la sola occorrenza dell'indice relativo al giocatore. Si fa riferimento ad esempio agli array dedicati ai JText per l'aggiornamento immediato dei bilanci, posizioni e così via.



Il seguente problema invece durante lo sviluppo è stato affrontato in egual modo da entrambi, dato lo sviluppo intersecato di tutti i lavori svolti in precedenza nella modellazione.

Game Problem

Una volta realizzate tutte le classi e implementate tutte le regole del gioco, il compito fondamentale è stato quello di dover unire tutto in maniera armoniosa in modo tale che si potesse sposare con un'entità a rappresentanza di tutto il gioco. Questo viene allora realizzato nella classe a capo di tutto il modello che sarà poi anche l'entità responsabile della comunicazione con il controller. Urbinopoly quindi, come si può notare, viene composta dal suo tabellone, i relativi dadi e il gruppo di giocatori che controllano le loro strategie di gioco. La difficoltà quindi è stata proprio quella di coordinare tutte le entità di un livello di astrazione più elevato per farle comunicare in modo sincronizzato. Si definiscono così i termini per automatizzare il gioco, con la capacità di ascoltare le azioni volute dal player (playerAction), e le reazioni del gioco all'avvenire di una determinata condizione del giocatore (doAction). Inoltre si rende responsabile il gioco, di valutare passo dopo passo, se determinate azioni possono essere svolte o meno da un giocatore.


Urbinopoly
- Composition: Type
- inTurn: boolean
- endGame: boolean
+ getter e setter(): Type
+ validateCommand(int, Player): boolean
+ endGame(): boolean
+ doAction(Player): void
- cardAction(Card, Player): void
- IncomeAction(Square, Player): void
- LuxuryAction(Square, Player): void
- propertyAction(Player, Square, int): void
- prisonAction(Player, int): void
+ playerAction(Player): void
Players
DiceApi
Board
INTERACT
Controller

Sviluppo

Testing automatizzato

I test sono stati uno dei mezzi più importanti per la realizzazione dell'applicativo, questo perché in progetti più complessi il numero di errori in fase di esecuzione è molto elevato.

Il lavoro di testing è stato effettuato grazie alle librerie JUnit di Java, metodo più efficiente per testare le diverse classi e i metodi. Fondamentale soprattutto nella parte iniziale dove l'intero core di interesse è stato riservato alla parte di modello, poiché non disponendo di una view era impossibile testare tramite l'esecuzione classica.

Il procedimento lavorativo generale può essere sintetizzato come segue:

- Realizzazione di classi che coinvolgono lo stesso problema.
- Test molteplici per il ritrovamento di errori.
- Risoluzione di questi ultimi mediante l'utilizzo del debugger (indispensabile per osservare lo stato del modello step by step e comprendere i punti critici di errore)
- Rivisitazione del codice per aumentare la sua leggibilità.

Solo dopo aver testato un insieme di classi inerenti allo stesso problema la fase di implementazione proseguiva, anche perchè risulta molto più complicato testare un numero elevato di classi assieme al termine di tutta la parte di modello, rischiando di commettere ulteriori errori invece di migliorare.

In maniera inerente alle classi di modello implementate ognuno si è dedicato al testing di quelle specifiche classi. Il lavoro è stato diviso quindi come segue.

Sette Miriana

- CardsTest
- PlayerTest
- PlayersTest
- DiceTest

Ungolo Michelangelo

- LandsTest
- PropertyTest
- TaxesTest
- UrbinopolyTest

Metodologia di lavoro

La metodologia si è basata su una stretta collaborazione: tutta la parte di modello, iniziando dalle prime fasi di analisi, l'assegnazione dei compiti e le prime implementazioni è stata fatta interamente in maniera sincrona, in modo tale che anche non lavorando sulle altre zone di codice non spettanti eravamo sempre a conoscenza del lavoro che veniva svolto.

Non solo questo ci ha permesso di aiutarci a vicenda nei momenti di stallo che durante l'implementazione sorgevano, ma anche di imparare l'uno dall'altro il diverso approccio di ragionamento su un determinato problema.

Solo dopo una grande fetta di lavoro abbiamo continuato il lavoro singolarmente, dove la logica che girava intorno al software era a buona conoscenza di entrambi.

Conclusa la fase di modello abbiamo separato i due macro compiti che restavano suddividendo la view e il controller, rimanendo comunque in contatto data la fase implementativa di sincronizzazione che doveva realizzarsi.

A giochi finiti possiamo confermare che il nostro approccio al lavoro è stato molto soddisfacente, poiché lavorando per molto tempo alla fase di modello assieme abbiamo potuto apprendere modi di approccio alla logica e all'implementazione diversi l'uno dall'altro, potendo così sbloccare nuove skills lavorative.

Importantissimo l'uso di git e github, che ci ha permesso una fluidità e velocità di lavoro notevole dovendo lavorare su macchine differenti. Questo ci ha permesso di avere l'uno i

progressi dell'altro sul proprio ramo di riferimento ad ogni progresso fatto, dove alla fine si è unito il tutto nel main branch.
In particolare una volta concordati sul lavoro da svolgere, si è proceduti nell'implementazione restando comunque in contatto; completato il lavoro di riferimento ognuno testava le proprie funzionalità, migliorava la leggibilità del codice e effettuava la procedura di commit e push tramite git. In questo modo tutti i progressi erano continuamente disponibili per la visualizzazione e la continuazione del progetto.

Ungolo Michelangelo

Parte iniziale inerente ai pezzi del tabellone, donando l'ossatura che in seguito occorre per la creazione del tabellone.

Realizzata tutta la parte inerente alle proprietà del gioco, rispettando le regole della sua documentazione.

Implementazione delle tasse come elemento costitutivo del tabellone.

Logica della prigione con scelta di separazione di essa da una eventuale classe prison, ma inserita come applicativo del player.

Sincronizzazione del modello per la classe coordinatrice Urbinopoly.

Sincronizzazione dell'intero applicativo mediante l'implementazione del controller.

Sette Miriana

Realizzazione delle carte di gioco, separando e distinguendo le normali carte, dai mazzi e la creazione personalizzata dei due mazzi di gioco principi: probabilità e imprevisti.

Implementazione dei giocatori, definendo tutte le sue azioni possibili e le caratteristiche di ognuno mediante la propria classe inerente.

Realizzazione dei dadi di gioco e implementazione del tabellone raggruppando tutta l'ossatura svolta sui singoli pezzi creati.

Sincronizzazione del modello per la classe coordinatrice Urbinopoly.

Implementazione dell'intera view ponendo l'attenzione sia all'efficienza che all'attrazione.

Note di sviluppo

Sette Miriana

- Utilizzo di stream
- Utilizzo di lambda expression fondamentali per un ottimo lavoro con gli stream
- Algoritmo manipulateProperty in Player per la realizzazione di un metodo in grado di potersi adattare ad ogni controllo di una proprietà ponendo attenzione al principio DRY.

Ungolo Michelangelo

- Utilizzo di stream
- Lambda Expression in abbinamento agli stream
- Utilizzo degli Optional per evitare la gestione di campi con null
- Costruzione di tipi generici per l'implementazione delle tasse.
- Runnable per la sincronizzazione dei diversi thread creati nel controllo e vista

Considerazioni ed osservazioni

Note sul progetto

Si consiglia l'avvio dell'applicazione solo dopo aver impostato la risoluzione del proprio schermo su 1366x768 dalle corrispondenti impostazioni del proprio dispositivo. Contrariamente la view non è capace di adattarsi in qualunque monitor, portando una visualizzazione non completa del gioco implementato.

Opinioni personali sul corso

Entrambi riteniamo che questo corso sia stato uno dei più importanti, nonché molto piacevole da seguire. Per la nostra personale opinione è stato un corso complesso, direttamente proporzionato allo spessore dello stesso, il che sicuramente lo ha reso ancora più entusiasmante.
Il progetto è stato realizzato con un occhio differente rispetto ad un normale esame universitario, proprio perché si è rivelato molto ambizioso sin dall'inizio.
Ci è dispiaciuto non aver avuto più ore a disposizione per approfondire alcuni aspetti trattati velocemente ma rivelatisi fondamentali durante lo svolgimento del progetto e soprattutto per la programmazione ad oggetti e non solo. Le difficoltà incontrate maggiormente sono state legate ad aspetti come l'architettura model view controller o l'implementazione dei diversi pattern. Si è capito che anche per la docente questo è stato un grande ostacolo, causato dalle ore a disposizione.
Ringraziamo la professoressa Montagna per l'intero corso e la passione a noi trasmessa.