RELAZIONE DI PROGRAMMAZIONE AVANZATA

Progetto: Mastermind

Abstract

Implementazione di Mastermind in Java

Help Desk Utenti

matteo.belenchia@studenti.unicam.it

Sommario

Introduzione

Il progetto è una implementazione del gioco Mastermind[1]. Viene permessa una elevata libertà di customizzazione della partita che va dal numero di pioli alla dimensione del codice, dal tipo di pioli al tipo di interfaccia di gioco. Sono definite 3 tipologie di giocatori artificiali, di cui 2 che sfruttano algoritmi risolutivi più o meno efficaci. Uno di questi è l'algoritmo di Knuth che riesce sempre a vincere (a meno di impostazioni proibitive) concessi i lunghi tempi di esecuzione. L'altro algoritmo è un tipo di algoritmo genetico che non sempre riesce nella risoluzione ma che ha un comportamento molto simile a quello di un giocatore "vero". Sono disponibili due tipi di interfacce per giocatori umani : una da linea di comando utilizzando il terminale e l'altra mediante una GUI scritta su misura. Questa varietà di giocatori e di interfacce tuttavia sono gestite indifferente dal modulo di controllo del gioco.

Selezionando due giocatori umani è possibile sfidarsi sullo stesso PC utilizzando la stessa interfaccia (testuale o grafica); il progetto quindi implementa la modalità "hotseat".[2] Purtroppo invece per via di mancanza di tempo non è stata implementata la modalità multiplayer in rete; viene tuttavia trattata una sua possibile implementazione nell'ultimo capitolo.

Nella classe `Tests` è definito un test che può essere eseguito utilizzando impostazioni diverse delle Properties. La classe `robot` viene utilizzata per premere un tasto (f2) automaticamente e far proseguire la partita senza che sia necessario premere tasti manualmente.

Nel corso della trattazione si utilizzeranno in modo equivalente i termini codice e `Code`; tentativo e guess; `Outcome`, responso, indizio e risposta; giocatore e `Player`; inoltre viene utilizzata in maniera intercambiabile il nome di una variabile e il suo tipo quando non ambiguo. Inoltre con l'espressione CodeMaker ci si riferisce al giocatore che ha composto il codice segreto nel round correntre, mentre CodeBreaker è il suo avversario.

Package : core

CodePeg

L'interfaccia `CodePeg` tratta del singolo elemento che fa parte di un codice, ovvero il singolo piolo. Le classi implementatrici di questa interfaccia dunque definiscono una categoria di pioli e l'unico metodo richiesto è `toString` per poter salvare ogni piolo in una `Map` che rappresenta il set di pioli per la partita corrente. L'interfaccia inoltre contiene una `Enum` composta da un solo elemento, `EMPTY`, che viene aggiunto alla map quando la modalità di gioco specifica che è possibile utilizzare spazi vuoti nei codici : in questa maniera lo spazio vuoto è trattato allo stesso modo di un piolo "classico", semplificando sostanzialmente la gestione di questa modalità

Le `Enum` che al momento implementano questa interfaccia sono `LetterPeg`, `ColorPeg` e `DigitPeg` nel package `Pegs`, che rispettivamente rappresentano un `Enum` di lettere dell'alfabeto (dalla A alla F), alcuni colori (in questo caso 8 colori diversi) e le 10 cifre. L'implementazione dell'interfaccia attraverso `Enum` mi sembra quella più naturale ma non si esclude la possibilità di implementarla in altri modi; tuttavia implicitamente resta evidente la necessita di avere in queste classi una struttura dati che contenga un elenco di pioli, ognuno dei quali ha una rappresentazione a stringa diversa (quest'ultima parte già garantita dal metodo `toString` ereditato da `Object`).

PegSetFactory e BasePegSetFactory

L' interfaccia funzionale `PegSetFactory` implementa un l'abstract factory pattern, dove la funzione `getPegSet` ha lo scopo di restituite al chiamante una `Map` che rappresenta il set di pioli in uso nella partita

[1] https://en.wikipedia.org/wiki/Mastermind_(board_game)
[2] https://en.wikipedia.org/wiki/Hotseat_(multiplayer_mode)

corrente. La Map utilizza come chiave la rappresentazione stringa dei pioli e come valore un' istanza di CodePeg, e quale effettiva implementazione di CodePeg è utilizzata viene deciso dal metodo. I parametri richiesti dalla funzione sono il numero di pioli e se lo spazio vuoto è consentito.

La classe BasePegSetFactory implementa la factory per ottenere il pegSet.Nel costruttore è richiesta la classe estendente CodePeg da utilizzare come fonte dei pioli per fissarla come attributo della classe, quindi viene registrata la funzione baseFactory per la produzione dei pegSet delle classi ColorPeg, DigitPeg e LetterPeg. La funzione baseFactory accetta 2 argomenti : il numero di pioli nel set (che deve essere consono rispetto al tipo di pioli, ovvero per esempio se si usano LetterPeg non possono essere più di 6) e se gli spazi vuoti sono permessi nei codici. La funzione crea una LinkedHashMap di dimensione adatta e aggiunge il numero di pioli richiesti prendendoli dall'Enum della classe impostata dal costruttore, ed eventualmente aggiungendo il piolo vuoto a rappresentare il fatto che è possibile lasciare spazi vuoti nei codici. Purtroppo non è possibile obbligare una classe implementatrice di CodePeg ad essere Enum, quindi se la classe passata non ha un Enum, il metodo lancia una RuntimeException. Il metodo getPegSet ereditato dall'interfaccia ritorna il risultato della funzione associata alla pegClass impostata nel costruttore.

L'utilizzo di una LinkedHashMap è dovuto al fatto che questa mantiene l'ordine di inserimento delle Entry, così che quando viene scorsa gli elementi arrivano nell'ordine definito dalle corrispondenti Enum.

PegClassFactory e BasePegClassFactory

Anche l'interfaccia funzionale PegClassFactory implementa un abstract factory pattern dove la funzione getClass data una stringa restituisce la classe estendente CodePeg associata. Nel progetto serve essenzialmente per stabilire data una String ricevuta dalle Properties a quale classe tale stringa si riferisce.

L'interfaccia è implementata da BasePegClassFactory, che nel costruttore registra nella Map di classi le classi ColorPeg, Digitpeg e LetterPeg del package Pegs. A questo punto la funzione getClass consiste semplicemente nel restituire la classe nella Map corrispondente alla Stringa passata in input.

PlayerFactory e BasePlayerFactory

Analogamente all'interfaccia precedente, PlayerFactory implementa un abstract factory pattern con la funzione makePlayer che ha lo scopo di restituire un'istanza di Player data una serie di parametri.

L'interfaccia è implementata da BasePlayerFactory che nel costruttore registra i giocatori artificiali da me implementati nella Map AIPlayers. Tuttavia i giocatori umani hanno bisogno di parametri diversi per essere costruiti e quindi non possono essere aggiunti alla Map insieme agli altri, ma vanno gestiti a parte in una maniera non estendibile senza mettere mano al sorgente. La funzione createPlayer per prima cosa stabilisce se c'è da creare un giocatore umano, ed in tal caso crea il giocatore appropriato passando la GUI o l'InputStream e PrintStream appropriato. In implementazioni successive la funzione potrebbe accettare anche un parametro di GUI, InputStream o PrintStream per costruire i Player anche tenendo conto di questi parametri. Nel caso il giocatore da costruire non è umano, basta recuperare la referenza del costruttore dalla Map AIPlayers.

Code

Questa classe rappresenta un generico codice del gioco, che sia il codice segreto o un tentativo di indovinarlo. La classe svolge il ruolo di wrapper rispetto alla reale implementazione interna di un codice, che è il suo attributo di tipo LinkedList<CodePeg>. Nel suo costruttore viene accettata una List di qualsiasi tipo, quindi un player può decidere di assemblare un codice nel tipo di lista che preferisce, ma internamente questa lista viene poi convertita in una LinkedList.

La classe sovrascrive alcuni metodi ereditati da `Object`. Prima di tutto, il `toString` di un codice è una stringa composta dalle rappresentazioni a stringa di ogni piolo contenuto nella `LinkedList` separate da uno spazio. Il metodo `equals` invece, oltre ai confronti tipici sui riferimenti e sul tipo, controlla anche che ogni elemento delle liste corrisponda nello stesso ordine, utilizzando semplicemente il metodo `equals` fornito dalle `LinkedList` stesse. Analogamente il metodo `hashCode` restituisce l'hash della `LinkedList` utilizzando la versione di `hashCode` fornita da `List`.

Outcome.

La classe Outcome rappresenta l'indizio che viene dato all'utente a seguito di un tentativo di indovinare il codice segreto. Al contrario dei pioli che compongono il codice, che possono essere di vari tipi, questa classe è stabilita come universale per ogni tipo di gioco e rappresenta i due tipi di indizio, ovvero piolo presente ma in posizione sbagliata e piolo presente in posizione corretta, attraverso gli attributi interi whitePegs e blackPegs.

La classe presenta due costruttori, uno di default che crea un Outcome senza pioli e uno con il numero di pioli passato in input. I metodi mutator sono particolari nel senso che permettono solo di aggiungere un piolo di un dato tipo alla volta, perché durante l'esecuzione è l'unico tipo di modifica necessaria; per poter settare valori in modo diverso si dovrebbe far riferimento al costruttore con parametri.

Nella classe vengono sovrascritti i metodi equals e toString. Il metodo equals, oltre ai soliti controlli, considera due Outcome uguali se e solo se il numero di whitePegs e blackPegs corrispondono.

GuessRow

Questa classe rappresenta una riga di una `Board` contenente un tentativo di indovinare il codice e il responso associato.

Le righe di tentativo, oltre al codice del tentativo, hanno l'indizio di risposta di tipo `Outcome` e un numero intero che rappresenta il numero del turno di gioco associato alla riga. Chiamate successive a `setOutcome` non modificano il valore di `Outcome` impostato.

Board

La classe `Board` rappresenta la plancia di gioco e la considero il modello del pattern `Model-Control-View`.

Gli attributi della classe sono una `LinkedList` di `GuessRow` che rappresenta le righe dei tentativi, un `Code` che rappresenta il codice segreto, una variabile di numero di turno e un parametro di dimensione che corrisponde al numero massimo di tentativi permessi. Nella `LinkedList` le nuove righe vengono via via aggiunte in coda. La `Board` estende la classe `Observable` per poter notificare il suo `Observer`, cioè le GUI, delle sue modifiche e quindi aggiornare la sua rappresentazione grafica. In questo modo è implementato l'Observer pattern.

Il costruttore della `Board` richiede la dimensione della stessa e il codice segreto associato.

setGuess e setResponse

I due metodi principali sono `setGuess` e `setResponse`. Il primo prende il codice di un tentativo e lo inserisce in fondo alla `LinkedList` mentre il secondo prende l'indizio di risposta a un tentativo e lo inserisce nella riga in fondo alla `LinkedList`. In entrambi i casi l'Observer è notificato del cambiamento passandogli il nuovo codice o il nuovo indizio. Ovviamente questi due metodi devono essere chiamati dal Controller nell'ordine giusto e nel modo adeguato per mantenere la consistenza, ovvero si chiama prima `setGuess` e poi `setResponse` e per ogni `setGuess` viene chiamato un `setResponse` subito dopo. Alcuni controlli vengono eseguiti ma si potrebbero rendere ancora più stringenti.

hasEnded

Il metodo `hasEnded` verifica se la `Board` è stata completata: ciò avviene se i turni sono terminati o se l'ultimo indizio di risposta corrisponde a un `Outcome` con un numero di pioli neri pari alla dimensione del codice.

getLastRow e getVisibleBoard

Il metodo `getLastRow` restituisce la `GuessRow` in fondo alla `LinkedList`, mentre `getVisibleBoard` restituisce l'intera lista.

Player

L'interfaccia `Player` contiene i metodi che ogni giocatore deve implementare per poter essere compatibile con la classe `Match`. Il metodo `getGuess` chiede al giocatore un tentativo passandogli come dato la lista dei tentativi precedenti. Il metodo `getCode` chiede un codice segreto, mentre `checkLastRow` chiede al giocatore l'indizio di risposta al codice dato (assumendo che il giocatore conosca il codice segreto con cui confrontarlo). Il metodo `endRound` viene chiamato alla fine di una `Board` prima di iniziarne una nuova e di cambiare i ruoli ai giocatori e dovrebbe contenere codice di clean-up per i giocatori artificiali e pulizia della GUI per quelli umani. Il metodo `endMatch` viene chiamato alla fine del match e dovrebbe contenere le operazioni (aggiuntive) di chiusura del programma come la chiusura della GUI.

L'interfaccia quindi contiene il metodo statico di utilità `getRandomCode` che genera un codice casuale data la sua dimensione e la lista dei pioli in uso.

Settings

La classe `Settings` si occupa di ottenere dall'utente le impostazioni della partita necessarie per costruire il `Match`. L'attributo fondamentale è la variabile settings di tipo `Properties`, che contiene un mapping di impostazioni e valori associati. La classe implementa un Singleton pattern perché le impostazioni della partita sono univoche durante la stessa e devono poter essere facilmente acceduta sia dal match che dalla GUI (se presente), tuttavia la funzione per accedervi, `getGameProperties`, non restituisce l'oggetto di tipo `Settings` ma direttamente l'attributo `Properties settings`. La classe ha una rappresentazione grafica corredata di slider, checkbox e radiobuttons per rendere immediata la selezione delle impostazioni.

L'unico metodo accedibile dall'esterno è `getGameProperties`. Questo metodo è acceduto per la prima volta nella prima istruzione del costruttore del `Match`, ed in questo caso il valore di `Settings` è null e quindi il Singleton viene istanziato e la GUI delle impostazioni viene generata e visualizzata. A questo punto il thread `main` che ha richiamato il metodo chiama la `wait` sull'oggetto `Settings istance`, in attesa che le impostazioni siano raccolte dalla GUI.

Nella GUI `l'ActionListener` del `JButton` "Start" istanzia l'oggetto `Properties` e raccoglie le informazioni dai vari elementi della GUI, quindi elimina il `JFrame` e risveglia il thread in attesa su `istance`. A questo punto `getGameProperties` può ritornare la variabile `Properties settings` attraverso il getter `getProperties`.

Le possibili impostazioni di gioco sono:

- Pegs : il numero di pioli diversi che possono comporre il codice; questo valore dipende anche dal tipo di pioli (la GUI si aggiorna in modo consistente a ciò)
- Rounds : il numero di round da giocare che per regola è un numero pari
- BoardSize : il numero di tentativi che una Board può contenere
- CodeSize : la dimensione di un codice
- EmptyAllowed : se sono permessi spazi vuoti in un codice
- CodeBreaker : se il giocatore selezionato come Player 1 inizia come CodeBreaker
- PegType : il tipo di pioli, tra Lettere, Numeri e Colori.

- Player 1 e 2 : i giocatori che si sfideranno, possono essere sia umani che artificiali
- UI : il tipo di interfaccia se grafica o testuale
- Match : se è un match locale, un istanza di server o di client; non implementato

ResourceLoader

La classe si occupa di caricare le risorse grafiche richieste dalle varie GUI. Si utilizza una classe a sé per evitare di caricare le stesse risorse più volte. Viene offerta quindi una interfaccia univoca e semplice mediante l'utilizzo di un Singleton pattern; tuttavia come nel caso di `Settings` anche qui abbiamo dei metodi che invece di restituire l'istanza restituiscono direttamente la `Map<String,BufferedImage>` richiesta. Nel caso la `Map` richiesta sia null, viene creato un nuovo `ResourceLoader` anonimo il cui costruttore chiama il metodo `loadFixedResources` o `loadDynamicResources`. Il primo metodo carica nelle mappe `keyp` e `struct` le risorse grafiche che sono le stesse per ogni tipo match, mentre il secondo carica le risorse grafiche col nome corrispondente alle chiavi della `Map` `pegSet`.

Status

Questo enum rappresenta i possibili stati di un match. Lo stato `AWAITINGCODE` significa che il match si aspetta il codice segreto dal giocatore di turno, `AWAITINGGUESS` significa che si aspetta un tentativo mentre `AWAITINGOUTCOME` significa che si aspetta un indizio di risposta (all'ultimo tentativo). Nell'implementazione corrente questi stati servono solamente alla `GUI`, principalmente per capire alla pressione del bottone "Send" da quale tipo di bottoni prelevare la mossa.

Match

La classe `Match` è la classe principale del programma e svolge la funzione di Controller nel pattern MCV. La classe è inoltre implementata seguendo il pattern Singleton, visto che in un' ottica di partita locale ad ogni istanza del programma è associato uno e un solo match e dopodiché il programma si chiude. Con una semplice modifica, resettando il parametro `currentRound`, si possono avere più match consecutivi sempre con un' unica istanza di `match`. Da un punto di vista di un' architettura client-server invece il server dovrebbe poter avere più istanze di `match` in esecuzione, una per ogni coppia di client. Sempre mantenendo il singleton pattern, si può avere invece di una variabile statica di istanza `match`, una `List` statica di istanze di `match` create allo stesso modo chiamando `init`, dove la posizione i-esima nella `List` (0,1,2,…,N) identifica univocamente il match tra la coppia i-esima di client.

La funzione `init` è la funzione di avvio del match ed è l'unica che effettivamente istanzia l'attributo `match`. Quindi avvia la procedura `initializePlayers` ed avvia il match tramite la funzione `runMatch`. La funzione `getMatch` che ritorna l'istanza di Match lancia un'eccezione se il `match` non è stato istanziato. Di norma nel singleton pattern in questi casi si istanzia il `match` e lo si ritorna, ma volevo costringere gli utenti a dover chiamare init prima di usare `match`. Nel contesto di una partita locale ciò ha senso perché i `Player` che fanno riferimento a `match` dovrebbero farlo solo nel contesto di una partita avviata, anche se nulla osta il contrario. Nel contesto di una partita online invece queste considerazioni non sono valide.

Nel costruttore di `Match` si ottengono le impostazioni di gioco dalla classe `Settings` e quindi si istanziano le corrispondenti variabili e le varie factory (`BasePegClassFactory`, `BasePegSetFactory` e `BasePlayerFactory`). Infine, se nelle impostazioni è stata impostata una interfaccia grafica senza definire giocatori umani, il match crea una nuova `ObserverGUI`.

La funzione `initializePlayers` istanzia l'array dei `Player` e vi assegna i `Player` creati da `playerfactory`. Dato che la variabile `codeBreaker` è `true` se si è selezionato che il giocatore designato come Player 1 inizia come CodeBreaker, ma per come è strutturata l'esecuzione di un round il CodeMaker del primo round è sempre il `Player` nella posizione 0 dell'array, si fa in modo che il `Player` CodeBreaker del primo round sia sempre nella posizione 1 dell'array. Viene anche inizializzato l'array dei punteggi.

runMatch

La funzione `runMatch` contiene la corretta sequenza delle operazioni per lo svolgimento del match. La sequenza prevede di inizializzare il round con `initRound`, eseguire il turno con `execTurn` finché la `Board` `currentBoard` non è conclusa. Quindi eseguire le operazioni di fine round con `finalizeRound`. Le precedenti oeprazioni sono svolte per ognuno dei round, quindi si termina il match chiamando la funzione `endMatch`.

initRound

La funzione imposta lo `Status` del match come `AWAITINGCODE` quindi chiede al player corrispondente al modulo 2 del turno corrente il codice segreto. Quindi al primo round chiederà sempre a `player`[0], poi al `player`[1] e così via. Il codice segreto viene salvato in una variabile non accessibile dall'esterno della classe. Il codice viene quindi passato a una funzione che ne verifica la validità e in caso di errori si lancia un'eccezione. Ciò avviene solo per giocatori artificiali, dato che i giocatori umani utilizzano un'altra funzione di controllo che ritorna eccezioni gestite nelle loro classi e quindi le mosse di giocatori umani, quando arrivano al match, sono sempre corrette e passano sempre questo controllo e quelli definiti successivamente.

Dopodiché si istanzia una nuova `Board` in `currentBoard` passando la dimensione richiesta e il codice segreto, quindi si chiama la funzione `setObservers` per impostare, se presenti, le GUI (collegate a `HumanGUIPlayer` o la `ObserverGUI` di `Match`) come `Observer` della `Board` appena creata.

execTurn

Questa funzione viene chiamata per ogni turno di gioco. Inizialmente imposta lo `Status` del match in `AWAITINGGUESS`, quindi chiama la funzione `getPlayerguess` per ottenere il tentativo dal giocatore di turno. Quindi lo `Status` viene impostato come `AWAITINGRESPONSE` e la funzione `getPlayerOutcome` ottiene l'indizio di risposta dal giocatore adatto. Quindi avanza il turno di `currentBoard`.

`getPlayerGuess` imposta la variabile `currentGuess` prendendo dal giocatore opposto a quello che ha creato il codice segreto il suo tentativo, il tentativo viene passato a una funzione che ne verifica la validità (la stessa usata in `initRound`) e si lancia un eccezione in caso di errore. Quindi il tentativo viene passato alla funzione `setGuess` della `Board`.

`getPlayerOutcome` funziona in modo analogo ma viene chiamata sullo stesso giocatore che ha creato il codice segreto e utilizza una funzione diversa per verificare la sua risposta, quindi la risposta viene passata alla funzione `setResponse` della `Board`.

finalizeRound

La funzione aggiorna il punteggio del CodeMaker sommando il numero di turni di `currentBoard` più 1 se CodeBreaker non ha indovinato il codice. Quindi richiama la funzione `endRound` su entrambi i giocatori e se la `ObserverGUI` è stata settata richiama il suo metodo `endRound`, altrimenti se non ci sono giocatori umani né `ObserverGUI` si richiama la funzione `printScores` che stampa sul terminale la `Board` e i punteggi del round. Quindi si incrementa il valore di `currentRound` di 1.

endMatch

Al termine del match si richiamano le corrispondenti funzioni su entrambi i giocatori e, se presente, la `ObserverGUI`.

verifyCode

Questa funzione verifica la validità di un codice, che sia segreto o un tentativo. La funzione viene richiamata da `execTurn` su ogni mossa, anche se per i giocatori umani essa viene già richiamata da una funzione intermedia per permettere di ripetere la mossa in caso di errore. Se la funzione rileva nel codice di input un

piolo non presente in pegSet, lancia una NoSuchElementException, mentre se la dimensione del codice è diversa dal parametro codeSize lancia una IllegalArgumentException. Infine ritorna true.

verifyOutcome

La funzione ha un ruolo analogo per gli indizi di risposta, lanciando una IllegalArgumentException se il numero di pioli supera la dimensione di un codice o se la risposta ricevuta in input non corrisponde con la risposta che il match si attendeva. Infine ritorna true.

parseCodeString

Questa è la funzione che i giocatori umani usano per verificare i loro codici di input. La String passata in input viene separata in token utilizzando lo spazio come separatore mediante uno Scanner, quindi il codice viene assemblato aggiungendo il piolo di pegSet ottenuto utilizzando il token come chiave della Map. A questo punto il codice viene passato alla funzione verifyCode e la funzione lo ritorna se il controllo viene passato, mentre in caso contrario ci sarà stata una Exception.

parseOutcomeString

Questa funzione è equivalente alla precedente ma si usa per gli indizi. Per ogni token corrispondente a "WHITE" o "BLACK" si aggiunge il corrispondente piolo all' Outcome, mentre se si incontra un token inatteso viene lanciata una NoSuchElementException. La stringa "EMPTYKEYSLOT" è l'actionCommand dei bottoni lasciati vuoti nella GUI. L'Outcome viene comunque passato alla funzione verifyOutcome e viene ritornato se essa risolve a true.

Check

La funzione check produce l'Outcome verificando il codice guess sul codice code. Tramite questa funzione il match verifica la correttezza degli Outcome ricevuti dai giocatori e i giocatori artificiali implementati si affidano su di questa per produrre i loro Outcome. La funzione crea due LinkedList temporanee dei codici di input e ottiene i rispettivi ListIterator. I ListIterator vengono scorsi parallelamente e se viene trovato lo stesso CodePeg in entrambi i codici nella stessa posizione si aggiunge un piolo nero ad Outcome e si rimuovono i pioli combacianti da entrambe le liste. Finito questo ciclo, abbiamo ottenuto tutti i pioli neri.

Ora mediante uno stream per ogni elemento del codice segreto se questo è contenuto nel codice tentativo si aggiunge un piolo bianco e si rimuove il piolo dal codice tentativo. In questa maniera le regole di Mastermind sono applicate correttamente anche per quanto riguarda codici contenenti colori ripetuti.

Main

La classe main contiene il metodo main che imposta il LookAndFeel di sistema e richiama la funzione init di Match.

Package : players

AIPlayer

Questa classe astratta implementa l'interfaccia Player ed è la classe da cui ereditano i giocatori artificiali da me implementati. Gli attributi definiti per questa classe sono la dimensione di un codice e una List di pioli di tipo CodePeg, in più un attributo secretCode inizializzato a null che contiene, quando il giocatore ha il ruolo di creare il codice segreto, il codice che l'avversario deve indovinare. Il costruttore della classe necessita della dimensione del codice e della Map dei pioli pegSet generato dalla factory dei pegSet. Tuttavia per un giocatore artificiale è più pratica una List di pioli che può essere acceduta con valori numerici ad accesso casuale, quindi nel costruttore la Map viene convertita in un' ArrayList.

I due metodi implementati sono `checkLastRow`, che semplicemente richiama il metodo statico `check` della classe Match usando il tentativo passato in input e il valore di `secretCode`, e `getCode` che restituisce il codice segreto in modo casuale col metodo `getRandomCode` di `Player`.

RandomPlayer

Il giocatore `RandomPlayer` gioca in modo completamente casuale, e entrambi i metodi `getGuess` e `getCode` richiamano semplicemente il metodo statico `getRandomCode` definito nell'interfaccia `Player`.

KnuthPlayer

Il giocatore `KnuthPlayer` gioca implementando l'algoritmo di Knuth, un algoritmo enumerativo abbastanza lento che usa una tecnica di minimax[3] [4]. L'algoritmo è studiato per codici di dimensione 4 con 6 tipi di pioli e termina sempre in 5 mosse o meno in questo caso. L'utilizzo per dimensioni o tipi di pioli maggiori è assolutamente sconsigliato perché è un algoritmo di tipo enumerativo e i tempi di calcolo possono diventare proibitivi.

Gli attributi aggiuntivi necessari sono il numero dei diversi pioli `alphabet`, che viene recuperato dalla `Map` `pegSet`, i `Set` `possibleSolutions` e `AvailableGuesses` che rappresentano rispettivamente i possibili codici risolutivi e i tentativi di codice ancora non utilizzati, infine una `List` `possibleOutcomes` che contiene tutte le possibili combinazioni di `Outcome`. Nel costruttore vengono richiamati metodi `generateSets` e `generateOutcomes` popolano questi `Set` e la `List`.

generateSets e generateOutcomes

I due `Set` sono generati in modo identico come `HashSet` a partire da un codice di partenza `String` composto interamente dal primo piolo della lista `pegSet`. Quindi si itera un numero di volte pari al numero delle possibili combinazioni, dove ad ogni iterazione prima si aggiunge il `Code` generato dalla `String` corrente ai `Set`, quindi un ciclo for annidato cicla per la lunghezza di un codice e ad ogni iterazione scambia il j-esimo piolo con quello successivo in `pegSet`, e, se questo piolo non è il primo della lista, interrompe il ciclo. Questo metodo genera tutte le possibili combinazioni di codici dato il `pegSet`.

Gli `Outcome` sono invece generati a partire dall'osservazione che il numero di pioli neri è pari al più alla dimensione del codice meno il numero di pioli bianchi e viceversa. Dunque per ogni numero di pioli neri i abbiamo abbiamo al più n-i pioli bianchi e quindi per ogni i ci sono n-i +1 combinazioni; i possibili valori di i sono pari da n+1. Dunque $\sum_{i=0}^{n} n - i + 1 = \sum_{i=0}^{n} i + 1 = \sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$. Il metodo crea `Outcome` da aggiungere alla lista tenendo conto della prima osservazione.

initialGuess

Il metodo `initialGuess` genera la mossa iniziale, che è fissata nell'implementazione a due pioli del primo colore e due pioli del secondo. Genericamente, ho posto che la mossa iniziale sia per metà del primo colore e per metà del secondo.

getGuess

L'algoritmo è contenuto nel metodo `getGuess`. Il nuovo tentativo nextMove viene inizializzato con la mossa fissa iniziale se siamo al primo turno mentre dal secondo tentativo in poi si applica l'algoritmo. Come prima cosa si rimuovono dal `Set` `possibleSolutions` tutti i codici che se fossero stati il codice segreto non avrebbero dato lo stesso `Outcome` ricevuto con l'ultimo tentativo. In questo modo vengono quindi escluse tutte le soluzioni che non sono valide. A questo punto la mossa successiva viene determinata in questo modo:

[3] https://en.wikipedia.org/wiki/Mastermind_(board_game)#Five-guess_algorithm
[4] https://www.cs.uni.edu/~wallingf/teaching/cs3530/resources/knuth-mastermind.pdf

Ad ogni codice non utilizzato come tentativo è associato un numero intero, calcolato prendendo in considerazione tutti i possibili `Outcome`. Per ogni `Outcome` si calcola se prendendo il tentativo non utilizzato come codice segreto e confrontandolo con una possibile soluzione si ottenga `quell'Outcome`, e in caso affermativo un contatore viene incrementato. Quindi per ogni `Outcome` si ha il numero di volte che il tentativo non utilizzato restituisca `quell'Outcome`, e il numero associato al tentativo non utilizzato è il massimo valore tra questi numeri. Il tentativo migliore è quello con il minimo valore associato, perché può escludere il maggior numero di codici in `possibleSolutions`. I tentativi non utilizzati sono raggruppati a seconda di questo numero in una `List` e le `List` sono inserite in una `TreeMap`, il tutto attraverso la funzione `collect` applicata a uno stream parallelo. Si accede quindi al primo elemento della `TreeMap`, che corrisponde quindi alla `List` di `Code` con numero associato minore e utilizzando `min` seleziono quando possibile un tentativo non utilizzato che è anche una possibile soluzione. Questo tentativo sarà la prossima mossa che l'algoritmo esegue e viene rimosso da `availableGuesses`.

GeneticPlayer

Questo giocatore artificiale sfrutta un algoritmo genetico[5] per avere buoni risultati in un tempo migliore rispetto agli algoritmi enumerativi. Malgrado i risultati mostrati nell'articolo sono affini a quelli di altri algoritmi, come quello di Knuth, nella mia implementazione l'algoritmo non riesce consistentemente a vincere con la stessa media di turni proposta, tuttavia riesce in poche mosse a scoprire tutti i colori del codice segreto e si comporta in modo molto simile a un essere umano. I parametri secondo il quale l'algoritmo opera sono `POPSIZE`, cioè il numero di codici che fanno parte della popolazione, `MAXGEN`, il numero massimo di generazioni prodotte per turno, `ELIGIBLES`, il massimo numero di possibili tentativi presi in considerazione e `BIAS` che è un valore che serve per ottenere una probabilità influenzata in un certo modo.

getGuess

La funzione per ottenere un tentativo di codice gioca al primo turno un codice prestabilito chiamando `initialGuess`, mentre nei turni successivi si azzera l'insieme dei tentativi giocabili e finché questo insieme è vuoto si genera una nuova popolazione di codici casuale mediante `generateInitialPop` e poi chiamando `generatePop` questa popolazione passa attraverso varie generazioni conseguenti, durante le quali eventualmente dei tentativi giocabili verranno generati ed aggiunti al `Set eligibles`. Quindi quando almeno un tentativo è stato generato si chiama la funzione `selectGuess` per selezionare quale tentativo giocare.

generateInitialPop

La popolazione iniziale viene generata aggiungendo al `Set population` codici casuali diversi fra loro, mediante la funzione `getRandomNewCode` che utilizzando la classe `Random` genera codici casuali finché non ne genera uno non già presente in `population`.

generatePop

La funzione fa riprodurre la popolazione currente per un numero di generazioni stabilite mediante la funzione `breed`.

Breed

Prima di far riprodurre la popolazione si devono selezionare i codici "genitori" più adatti chiamando `generateParents`. Una volta ottenuti i genitori prescelti per riprodursi la popolazione viene azzerata, quindi si producono altrettanti "figli" per ripopolarla. Presi due genitori casuali, viene generato un nuovo codice nato dall'unione tra i due con la funzione `cross`, quindi in questo nuovo codice c'è una probabilità

---

5 https://lirias2repo.kuleuven.be/bitstream/handle/123456789/164803/kbi_0806.pdf;jsessionid=FB774472417C4D86005574CE1E9F5301?sequence=1

del 3% di avere una mutazione o una permutazione, e una probabilità del 2% di avere una inversione. Quindi se il codice risulta essere un tentativo giocabile e non si è superato il limite di numero dei tentativi giocabili, viene aggiunto al `Set eligibles`. Infine, se questo codice è già presente nella nuova popolazione che sta venendo generata, si crea un nuovo codice casuale univoco con `getRandomNewCode`, altrimenti il codice viene aggiunto alla popolazione.

generateParents

I genitori vengono selezionati in base al loro valore di `fitness`. Dato il `Set` della popolazione attuale `population`, mediante uno stream ordino la popolazione per il loro valore di `fitness`. Quindi ciclo un numero di volte pari alla dimensione della popolazione e ad ogni iterazione seleziono un elemento casuale della lista ordinata utilizzando la funzione `randomBiased`, che utilizzando il parametro di `BIAS` ha una probabilità di restituire un indice di valore basso più alta rispetto agli indici di valore alto. Se l'elemento selezionato non è già presente nella lista dei genitori prescelti, viene aggiunto ad essi. In questo modo il numero di genitori sarà al più `POPSIZE` e nel caso peggiore pari a 1.

Fitness e eligibility

Il valore di `fitness` di un codice è pari al valore di `eligibility` sommato a una costante dipendente dalla lunghezza di un codice e il numero di turno corrente. Il valore di `eligibility` è pari alla somma dei valori assoluti della differenza riga per riga della somma dei pioli di risposta generati da ogni tentativo meno la somma dei pioli di risposta generati utilizzando il tentativo come codice e il codice di cui vogliamo la `eligibility` come tentativo.

isEligible

Un codice è considerato un tentativo giocabile se non è già stato giocato e il suo valore di `eligibility` è 0, ossia quindi se per ogni tentativo già avvenuto il codice comparato coi tentativi avrebbe dato lo stesso `Outcome` e quindi risulta consistente con la sua possibilità di essere il codice segreto.

Cross, mutate, permute e invert

La funzione `cross` genera un codice figlio dati due codici genitori. Nella metà dei casi l'incrocio è a singolo punto e nell'altra a doppio punto. Negli incroci a singolo punto si sceglie un punto tra 1 e la dimensione del codice - 1 e nella parte sinistra del punto si prende la parte di codice del genitore 1 mentre nella parte destra quella del genitore 2. Negli incroci a doppio punto si prende un punto tra 1 e la dimensione del codice – 2 e un altro tra il primo punto e la dimensione del codice – 1. Tra questi due punti si prende la parte di codice del genitore 2, mentre negli altri quella del genitore 1.

La funzione `mutate` seleziona un elemento casuale del codice e lo sostituisce con un altro elemento a caso tra quelli disponibili nel `pegSet`.

La funzione `permute` prende due indici casuali diversi del codice e scambia gli elementi di quelle posizioni.

La funzione `invert` seleziona un indice tra 0 e la dimensione del codice – 2 e un altro indice tra il primo indice e la dimensione del codice – 1, quindi inverte la sequenza degli elementi del codice tra questi due indici.

selectGuess

Questa funzione seleziona quale tra i possibili tentativi validi giocare. La funzione seleziona tale tentativo considerando quale tra essi in media esclude il maggior numero di altri tentativi validi, affidandosi alle funzioni `averageEsclusions` e `countEligibles`. In pratica dato un tentativo, ogni altro tentativo ha la funzione di codice segreto che viene confrontato col tentativo in `averageEsclusions` e in `countEligibles` si conta quanti altri tentativi data questa situazione rimarrebbero validi, e `averageEsclusions` computa la media per ogni volta.

HumanPlayer

Questa è la classe astratta estesa dai giocatori umani da me definiti. Oltre al costruttore identico a quello di `AIPlayer` non implementa alcuna funzione dell'interfaccia `Player`.

HumanGUIPlayer

Questa classe gestisce un giocatore umano che si interfaccia attraverso una GUI. Nel suo costruttore è richiesta una istanza di `GUI` in input, la quale viene salvata come attributo di classe e sulla quale viene chiamata il metodo `setPlayer` passando l'istanza di `HumanGUIPlayer`. In questo modo le classi possono comunicare a vicenda e la `GUI`, con la chiamata a `setPlayer`, viene visualizzata. Si può vedere questa classe come implementatrice di un `Adapter` pattern, dove adatta la classe `GUI` ad avere una interfaccia `Player`. La classe implicitamente delega tutte le operazioni alle funzioni definite in `GUI` e svolge solo una funzione da tramite tra essa e il match. In questo modo quindi il match può trasparentemente operare su giocatori artificiali, giocatori umani senza GUI o con GUI.

getGuess, getCode e checkLastRow

Queste funzioni dell'interfaccia `Player` sono implementate in modo simile. Se necessario, si richiede alla `GUI` di sbloccare i bottoni necessari chiamando su di essa `nextTurn` (la GUI inoltre si basa sullo `Status` del `match` per capire quali), quindi il thread corrente viene messo in stato di `wait`. Quando la `GUI` ha la mossa del giocatore, ovvero dopo che il giocatore ha premuto il bottone "Send" e la sua mossa è stata considerata valida, essa richiama una delle corrispondenti funzioni `receiveCode`,`receiveGuess` o `receiveOutcome` che settano i rispettivi valori ricevuti e risvegliano il thread in attesa su `HumanGUIPlayer`. Il thread risvegliato ritorna quindi il valore al `match` che ne aveva fatto richiesta.

endRound e endMatch

La classe delega queste funzioni alle corrispondenti funzioni della classe `GUI`.

HumanTextPlayer

Questa classe gestisce un giocatore umano che si interfaccia utilizzando un `InputStream` e un `PrintStream`. Nell'implementazione corrente la factory di `Player` istanzia questa classe passando `System.in` e `System.out`, quindi l'interfaccia utilizzata è quella di un terminale. A tutti gli effetti questa classe rappresenta una vista del pattern MCV (senza tuttavia utilizzare il pattern Observer).

Nel costruttore l'`InputStream` viene trasformato in uno stream di caratteri con `InputStreamReader` e quindi istanziato in uno `BufferedReader` per leggere una riga di caratteri efficientemente.

getGuess, getCode e checkLastRow

Queste funzioni sono implementate in modo simile. Dopo le stampe necessarie sul `PrintStream` per fornire contesto all'utente si effettua un ciclo while su una chiamata a `readLine` sul `BufferedReader`. La stringa ricevuta deve passare i controlli che vengono effettuati chiamando `parseCodeString` o `parseOutcomeString` altrimenti il ciclo while prosegue. Infine il valore viene ritornato.

endRound e endMatch

La prima funzione richiama `printScores` sull'istanza di `Match` per stampare `Board` e punteggi quindi si mette in attesa della pressione del tasto invio per procedere al round successivo. La seconda funzione chiude il programma.

Package : GUI

ObserverGUI

La classe `ObserverGUI` genera una GUI di base per visualizzare lo stato di gioco. Viene utilizzata quando si seleziona l'interfaccia grafica ma senza avere giocatori umani nella partita. L'interfaccia non contiene bottoni cliccabili e si aggiorna mediante la funzione `update` dell'interfaccia `Observer`. Sono definiti molti `getter` per potervi accedere dalla classe `GUI`, la quale estende `ObserverGUI` aggiungendo funzioni e componenti grafiche aggiuntive.

Nel costruttore si recuperano le `Properties` e da queste interessa la lunghezza di un codice `codeSize` e la dimensione della board `boardSize`, quindi mediante l'argomento `pegSet` si inizializzano le `Map` `codep`, `keyp` e `struct` con le risorse grafiche ottenute dalla classe `ResourceLoader`. A questo punto si crea il frame impostandone i contenuti con la funzione `assemble`.

Assemble

Le funzioni `codeRow`,`leftSide` e `rightSide` creano rispettivamente i `Jpanel` per la riga del codice segreto, la plancia di gioco e la sezione di destra contenente la console e i punteggi. La creazione dei bottoni è inoltre snellita utilizzando la funzione `createButton`. La funzione `assemble` si occupa di riunire i `Jpanel` creati dalle funzioni precedenti in un unico `Jpanel` che utilizza un `GridBagLayout`.

Update

La funzione `update` viene dall'interfaccia `Observer` e si occupa di aggiornare lo stato della GUI. L'argomento `arg` che ci si aspetta di ricevere è un `Code` di tentativo o un `Outcome` di risposta, quindi a seconda di quale sia viene chiamata la funzione `updateGuess` o `updateOutcome`. Negli altri casi viene lanciata una `RuntimeException`. Le due funzioni appena citate recuperano il turno corrente dall'istanza di `Match` e quindi modificando l'icona dei bottoni corrispondenti a quella riga utilizzando le risorse grafiche collegate ai pioli del `Code` o `dell'Outcome`.

setCode

Questa funzione imposta il codice segreto nella `ObserverGUI`. Questo ovviamente ha senso perché dato che questa GUI viene utilizzata per osservare giocatori artificiali è lecito osservare anche il codice segreto. Il suo funzionamento è affine a `updateGuess`.

endRound

Questa funzione viene chiamata alla fine di un round di gioco e svolge delle operazioni di clean-up. Vengono aggiornati i punteggi dei giocatori chiamando `updateScores`, si scrive che il round è terminato nella console di gioco e si avvia un `EventListener` sui tasti della keyboard. Quindi il thread chiamante viene messo in attesa. Non appena l'Event Dispatch Thread rileva la pressione di un bottone il thread viene risvegliato e chiama la funzione `reset` che utilizza uno stream per reimpostare a null le immagini dei bottoni del codice segreto e della plancia, e riporta all'icona di default le immagini dei bottoni delle risposte.

endMatch

La funzione chiude il frame e il programma.

GUI

Questa classe estende `ObserverGUI` ed implementa l'interfaccia grafica collegata alla classe `HumanGUIPlayer`. Nel pattern MCV considero come View la classe `GUI`, mentre `HumanGUIPlayer` implementa un Adapter pattern per rendere questa vista compatibile con l'interfaccia `Player` che il Controller, ovvero `Match`, si aspetta. La classe modifica il `JFrame` ottenuto da `ObserverGUI` aggiungendo

nuovi bottoni laterali cliccabili e un bottone “Send” per inviare la mossa corrente. Vengono definiti vari actionListener collegati ai bottoni della plancia, ai bottoni laterali, ai bottoni degli indizi e al bottone “Send”.

Il costruttore recupera da settings il numero di pioli diversi che si utilizzano nei codici (per poter inizializzare il `GridBagLayout` con il valore giusto di righe quando si va a costruire la sidebar), attiva un eventListener sugli eventi correlati al cursore e costruisce il frame chiamando la funzione `assemble`. Il parametro booleano `codeBreaker` passato al costruttore è `true` se il giocatore associato alla `GUI` ha il ruolo di CodeBreaker nel primo round. Dato che per come è strutturato il match il giocatore 1 inizia sempre come CodeMaker, si può stabilire che si è il giocatore 1 quando il valore negato di `codeBreaker` è `true`, e questo valore è salvato nella variabile `amIP1`. Con questa informazione si richiama la funzione `updateScores`, che ora farà mostrare la dicitura “(you)” affianco a “Player 1” o “Player 2” nella GUI. Quindi si richiama la funzione `initialize` passando il valore di `codeBreaker`, che è anche necessario per stabilire la prospettiva della GUI, dato che CodeMaker e CodeBreaker hanno viste diverse.

Initialize

La funzione setta il valore di `codeBreaker` passato in input e se questo è `false`, e cioè la GUI è collegata a un player col ruolo di CodeMaker, si sbloccano i bottoni del codice segreto chiamando `toggle`, si sblocca il bottone “Send” e si inserisce il suggerimento di inserire il codice. Questa funzione viene chiamata tra un round e l’altro, ed ad ogni round il valore di `codeBreaker` ricevuto è invertito. La prospettiva di un CodeBreaker non mostra cambiamenti. Si sarebbe potuta invertire la GUI, mostrando in questo caso la riga del codice nella parte alta e gli spazi tentativi sulla parte bassa, ma per semplicità ho mantenuto questa implementazione.

setPlayer

Questa funzione imposta lo `HumanGUIPlayer` passato come parametro nell’attributo player della classe, quindi richiama il metodo `showGUI` per visualizzare la GUI. In questo modo quindi la GUI viene mostrata solo quando il `Player` associato è stato definito.

nextTurn

Il metodo `nextTurn` serve per sbloccare i bottoni che l’utente deve usare nel turno corrente; se il player è un CodeBreaker vengono sbloccati i bottoni della riga corrente corrispondente alla parte di codice, altrimenti si sbloccano i corrispondenti bottoni corrispondenti alla parte di responso.

Toggle

Il metodo `toggle` su cui si basa lo sblocco inverte lo stato dell’array di `JButton` passato in input, da `enabled` a `disabled` e viceversa, inoltre inverte anche il loro bordo tra rosso (quando attivi) e un bordo vuoto (quando disattivi).

Update

Il metodo `update` viene sovrascritto per aggiungere un controllo su `codeBreaker`: in questo modo l’implementazione di `update` della superclasse viene richiamata solo se c’è da visualizzare qualcosa di nuovo, ovvero nei casi se si è CodeBreaker ed arriva un nuovo `Outcome`, o se si è CodeMaker e arriva una nuova `Code guess`.

Reset

Il metodo `reset` viene sovrascritto per aggiungere una chiamata a `initialize` con valore di `codeBreaker` invertito; il metodo viene chiamato alla fine di ogni round.

makeCursor

La funzione crea un nuovo cursore prendendo come immagine l’elemento corrispondente alla stringa di input dalla `Map` dei pioli di codice o dei pioli di risposta (a seconda del valore del secondo parametro), e dà al cursore un nome pari al nome del piolo più `CodePeg` se è di codice o `KeyPeg` se è di risposta. Questa

distinzione è necessaria perché se si gioca con i colori allora il nome dei pioli dei codici e delle risposte si sovrappongono (coi pioli "WHITE" e "BLACK").

mousePressed

Questo è l'handler collegato all' eventListener associato al cursore nel costruttore, e resetta il cursore all'icona di default quando viene premuto il tasto destro del mouse.

Assemble e ActionListener

Il metodo disegna l'interfaccia grafica, recuperando il `Jpanel` contenuto nel frame creato dalla superclasse. Si aggiungono gli actionListener ai bottoni della plancia, delle risposte e del codice segreto, si aggiunge un bottone "Send" con un suo actionListener e si creano due nuovi `Jpanel`, l'uno a contenere i pioli di codice disponibili e l'altro i pioli di risposta. Questi `Jpanel` sono creati dalle funzioni `makeSidePanelPegPool` e `makeSidePanelKeyPool`. Anche ai bottoni di questi `Jpanel` sono associati dei rispettivi actionListener.

I vari actionListener descrivono le funzionalità della GUI.

`btnSideHandler` e `btnLowSideHandler` gestiscono i nuovi bottoni laterali, che rappresentano rispettivamente i pioli di codice disponibili e i pioli di risposta. Cliccandoli, il cursore cambia diventando il piolo selezionato (la dimensione del cursore rimane però 32x32 in Windows) mediante la funzione `makeCursor` a cui viene passato l'actionCommand associato al bottone. L'actionCommand dei bottoni collegati a questi handler altro non è che il valore di chiave nella `Map` delle risorse grafiche generate da `ResourceLoader`.

`btnMainHandler` gestisce la plancia di gioco e i bottoni del codice segreto. Un click su uno di questi bottoni, se ha l'actionCommand "EMPTY" (ossia è vuoto) e il cursore è un cursore custom il cui nome contiene "CodePeg", allora si imposta l'icona del bottone come l'icona presa dalla `Map` delle risorse grafiche dei pioli di codice corrispondente alla chiave del nome del cursore senza la dicitura "CodePeg". Ossia se il cursore si chiama "REDCodePeg", da questa `Map` si va a prendere l'icona corrispondente alla chiave "RED". Nell'altro caso un click resetta l'icona del bottone e il suo actionCommand a "EMPTY"

`btnKeyHandler` si comporta in modo analogo, con la differenza che si controlla nel cursore la presenza di "KeyPeg" nel nome.

Il `sendHandler` gestisce il bottone "Send" ed opera in modo diverso a seconda dello stato del `Match`. Se lo stato è `AWAITINGCODE`, allora si esegue la funzione `getCode` passando i `JButton` del codice segreto; se lo stato è `AWAITINGGUESS`, allora si esegue `getCode` passando i `JButton` della parte di codice della riga corrente; se lo stato è `AWAITINGRESPONSE`, allora si esegue `getResponse` passando i `JButton` della parte di risposta della riga corrente. Le funzioni vengono eseguite da un nuovo thread passato `all'ExecutorService`, per evitare di sovraccaricare l'EDT.

getCode

La funzione `getCode` reperisce un codice (che sia un tentativo o il codice segreto) dalla `GUI` e lo manda al player associato. Per prima cosa il codice viene reperito come stringa concatenando gli actionCommand dei bottoni ricevuti in input, quindi vengono passati alla funzione `parseGuessString` del `match` per verificarne la correttezza. Eventuali eccezioni lanciate in caso di stringhe erronee vengono catturate e il loro testo riportato nella `textArea` ereditata da `ObserverGUI` che ha la funzione di console. Se non sono state lanciate eccezioni allora il valore di `code` sarà non null, quindi vengono disattivati i bottoni dal quale è stato prelevato e il bottone di "Send", quindi a seconda dello `status` del `match` si richiama `receiveCode` (se il match si aspettava un codice segreto) o `receiveGuess` (se il match si aspettava un tentativo di `HumanGUIPlayer`. In caso di errori nel codice non avviene nulla, quindi il giocatore può riprovare.

getResponse

Questa funzione agisce in modo analogo alla precedente, con la differenza che la stringa viene controllata invece da `parseOutcomeString` di `match`, per verificare che il responso è consono con il tentativo e il codice segreto. Nel caso il controllo viene passato si disattivano i bottoni come nel caso precedente e si passa `response` alla funzione `receiveOutcome` di `HumanGUIPlayer`. Nell'altro caso viene mostrato un messaggio di errore nella `textArea` e si può riprovare.

endRound e endMatch

Queste funzioni sono ereditate da `ObserverGUI` senza modifiche, tuttavia vale la pena notare che `updateScores` chiamato dalla prima è quello definito nella classe `GUI` (senza overriding perché privato).

Package : networking

Per sviluppare una modalità multigiocatore di rete si può utilizzare un architettura client-server, con un server che ascolta su una determinata porta e che non appena due giocatori si connettono avvia la partita tra loro.

La comunicazione tra client e server si dovrebbe basare su oggetti di tipo `Message` che incapsulano il tipo di messaggio `MessageType`, l'argomento `Object` e una variabile di status e turno del match in esecuzione sul server. `MessageType` è un `enum` che rappresenta il tipo di messaggio e si potrebbe utilizzare per fare il casting adeguato dell'argomento di tipo `Object`. Possiamo immaginare `MessageType` di tipo `GUESS` quando il messaggio richiede (server->client) o contiene (client->server) un tentativo e considerazioni analoghe si possono fare per `MessageType` di tipo `CODE` o `OUTCOME`. Un `MessageType` `INFO` si potrebbe usare a inizio partita per trasmettere le informazioni sulle regole di partita in corso (definite rigorosamente dal server) e un `MessageType` `ENDROUND` o `ENDMATCH` per segnalare ai client il termine del round o della partita.

Il server dovrebbe essere avviato dopo la definizione delle `Properties` della partita e prima di avviare il `match`. La classe `Server` al momento utilizza array di due elementi quindi può gestire una sola partita, ma si potrebbero senza problemi gestirne di più in contemporanea (modificando opportunamente anche match per avere più istanze). Il server creato su una porta predefinita si può mettere in ascolto tramite la funzione `listen` (che può essere lanciata su un thread nel caso di partite multiple) e una volta che questa funzione termina la partita può iniziare.

Il `Server` si può incapsulare in una classe `RemotePlayer` che implementa l'interfaccia `Player` (che agisce come un `Adapter`) e il `Match` in esecuzione sul server si troverebbe a operare con due istanze di `RemotePlayer` che incapsulano lo stesso `Server` ma hanno `id` diverso. La classe quindi ridireziona le richieste del `match` su corrispondenti funzioni del server `getCode`, `getGuess` e `getResponse`. Queste funzioni sul server provocano la compilazione di un nuovo messaggio contenente il tipo di richiesta, l'argomento necessario per il suo soddisfacimento (se presente) e le informazioni sul turno corrente e lo stato corrente del `Match`. Quindi il `Server` invia il messaggio alla socket corrispondente `all'id` del `RemotePlayer` che ha fatto richiesta e si mette in attesa di risposta su quella socket per poi ritornarla al `RemotePlayer` che la ritornerà al `Match`.

Dal lato client si può istanziare un oggetto della classe `HumanRemotePlayer` che implementa un Decorator pattern su una istanza di `HumanPlayer` che può essere sia `HumanTextPlayer` che `HumanGUIPlayer`. Nel costruttore si connette alla porta prestabilita dal server quindi avvia la procedura `run` che si mette in ascolto di messaggi dal server sulla socket e chiama la funzione appropriata a seconda del `MessageType`, quindi invia la risposta al server. Tuttavia sia `HumanTextPlayer` che `HumanGUIPlayer` dipendono dal `Match` per alcune informazioni e funzioni : fanno richiesta di turno e

status, richiamano le funzioni `parseCodeString` e `parseOutcomeString` e quest'ultima richiede di accedere al parametro del codice segreto contenuto nel `Match`. Per risolvere questo problema è necessario istanziare il `Match` sul lato client, utilizzando le `Properties` contenute nei messaggi `INFO`. Quindi ad ogni turno aggiornare il valore di turno e status di questo `Match` sul lato client utilizzando le informazioni di status e turno che il server invia nei messaggi. Infine quando un `Player` lato client ha il ruolo di CodeMaker, deve poter settare il codice segreto nell'istanza di `Match`.

Per risolvere il problema senza modificare la classe `Match` si potrebbe creare una nuova classe che fa da tramite per le classi che utilizzano i metodi di `Match` : la classe nel caso di una partita locale inoltrerebbe le richieste all'istanza di `Match`, mentre nel caso di una partita dal punto di vista di un client la classe istanzierebbe `Match` con le `Properties` ottenute dal `Message` di tipo `INFO`, quindi avrebbe metodi per ritornare e settare i valori di turno, stato e codice segreto su se stessa. Le chiamate a `parseCodeString` sarebbero inoltrate all'istanza creata di `Match`, mentre per `parseOutcomeString` si dovrebbe riscrivere il metodo per poterlo risolvere senza basarsi su `Match`, richiamando tuttavia il metodo `check` già implementato.