Progetto di Paradigmi di Programmazione (parte Java) o Programmazione ad Oggetti 2019/2020 Realizzazione di Battaglia Navale

Il progetto va svolto in un Gruppo di massimo **3 persone**. Il progetto deve essere consegnato **2/3 giorni prima dell'esame** scritto (vi metterò un contenitore nel sito del corso) e vale 1/3 del voto complessivo (per il corso di paradigmi di programmazione) e 1/2 del voto complessivo (per il corso di programmazione ad oggetti). Tutti i gruppi devono implementare il progetto come richiesto in questa guida.
Alcune funzionalità sono indicate come opzionali (**OPZ**). Tali funzionalità sono le uniche facoltative. Il voto massimo di laboratorio, nel caso in cui siano state implementate anche le funzionalità opzionali, è 33 (30 e lode), mentre senza le funzionalità opzionali è 27[1]. Nella relazione consegnata (vedi sotto) deve essere indicato se le funzionalità opzionali sono state implementate o no.

Requisiti generali

Lo svolgimento del progetto deve tenere in considerazione i seguenti requisiti:

- le classi del modello (non le viste, né i controller) devono essere corredate da classi di test;
- la documentazione deve essere prodotta con javadoc;
- ogni progetto deve essere accompagnato da una breve relazione (massimo 2 pagine escluse le figure) in cui sono spiegate e illustrate le scelte adottate;
- per quanto riguarda l'interfaccia utente (che deve essere separata dalle classi che realizzano la logica del sistema) è solo necessario che permetta di accedere alle funzionalità del sistema.

Nello svolgere il progetto usare i principi di buona programmazione. In particolare,

- buona strutturazione delle classi, uso di classi astratte e interfacce se necessario;
- ogni classe ha un compito preciso (e fa solo quello), ha campi (e metodi) con visibilità “adeguate” (privati se non devono essere usati al di fuori della classe, protected in caso si pensi possa avere sottoclassi e queste debbano accedere a quei campi, public solo se costanti);
- uso di eccezioni e non "stampe" per segnalazioni di malfunzionamenti;
- metodi di dimensione (possibilmente) limitata. Se un metodo è troppo grande, probabilmente può essere diviso in più parti;
- non ripetizioni di codice

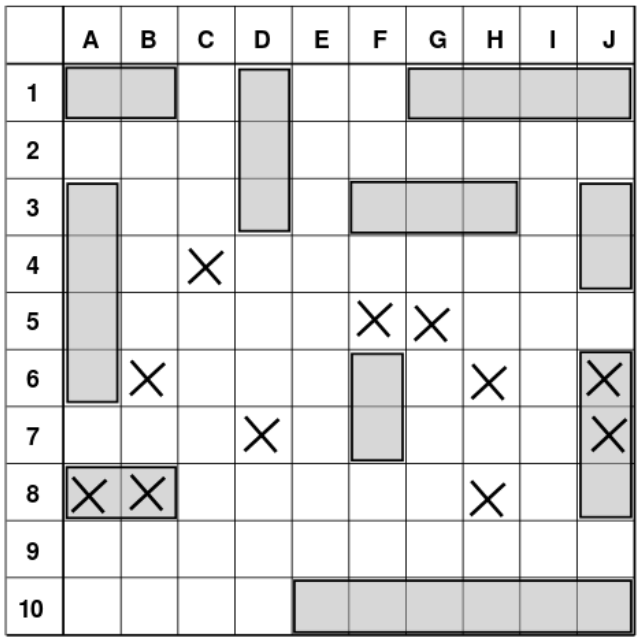
Valutazione

La discussione del progetto con i singoli studenti verterà sulle modalità di implementazione adottate e sulla padronanza di alcuni dei concetti necessari per realizzare il progetto e/o spiegati a lezione. La valutazione del progetto sarà fatta in base alla: conformità dell’implementazione scelta per risolvere il problema con il paradigma di programmazione a oggetti e alla conformità del codice presentato alle regole di buona programmazione.

[1] Come detto nelle slide introduttive, il voto è “a scalare” in base ai tentativi. Per chi fa la parte opzionale, i voti sono [33, 27, 23]. Per chi non fa la parte opzionale, i voti sono [27, 23, 19].

Regole del gioco (da Wikipedia)

Per giocare a battaglia navale occorrono quattro tabelle (due per giocatore), tutte di uguali dimensioni (per esempio 10×10 o un'altra dimensione concordata dai giocatori). I quadretti della tabella sono identificati da coppie di coordinate, corrispondenti a riga e colonna; tradizionalmente si usano lettere per le colonne e numeri per le righe (perciò le celle sono "A-1", "B-6", e così via). **Nota: in questo laboratorio, per semplicità, useremo interi sia per le righe che per le colonne (quindi [0,0], [0,1], …).** All'inizio, i giocatori devono "posizionare le proprie navi" segnandole su una delle loro due griglie (che terranno nascoste all'avversario per tutta la durata del gioco).



Una "nave" occupa un certo numero di quadretti adiacenti in linea retta (orizzontale o verticale) sulla tabella. Due navi non possono toccarsi. I giocatori si accordano preliminarmente su quante navi disporre e di quali dimensioni. Si può notare che molti giocatori utilizzano (anche non sempre in modo consistente) una particolare terminologia per riferirsi alle navi delle varie dimensioni; per esempio un sottomarino è di solito una nave di dimensione 3, insieme all'incrociatore, un cacciatorpediniere è di dimensione 2 e le navi di lunghezza superiore sono corazzate (dimensione 4) e portaerei (dimensione 5).

Una volta posizionate le navi, il gioco procede a turni. Il giocatore di turno "spara un colpo" dichiarando un quadretto (per esempio, "B-5"). L'avversario controlla sulla propria griglia se quella cella è occupata da una sua nave. In caso affermativo risponde "colpito!", e marca quel quadretto sulla propria tabella; in caso negativo risponde "acqua" o "mancato". Sulla seconda tabella in dotazione i giocatori prendono nota dei colpi che hanno sparato e del loro esito. Quando un colpo centra l'ultimo quadretto di una nave non ancora affondata, il giocatore che subisce il colpo dovrà dichiarare "colpito e affondato!", e la nave si considera persa. Vince il giocatore che per primo affonda tutte le navi dell'avversario.

Funzionalità

Per la parte di laboratorio (Java) è richiesto di implementare una versione di battaglia navale nella quale l'utente umano "sfida" il computer. Potete aggiungere funzionalità a piacimento (senza esagerare), ma le seguenti sono richieste:

- **INTERFACCIA:** tutti devono implementare una GUI che permetta all'utente umano di visualizzare le sue due mappe (nella sezione dei consigli c'è una descrizione di come potrebbero essere). Per chi decide di implementare la versione semplice, l'interazione con l'utente (es. quale azione fare successivamente) può essere fatta da terminale (con un menu come quello che avete usato per Rubrica), per chi decide di fare la parte **OPZ**, tutta l'interazione con l'utente deve essere fatta tramite GUI. Ad esempio, potete usare delle JTextField per chiedere le coordinate, o rendere "cliccabili" le caselle della mappa tramite gli eventi del mouse.
- **CARICAMENTO/NUOVA PARTITA**: se si decide di implementare la parte opzionale (**OPZ**) all'utente deve essere permesso di salvare e caricare partite (faremo questa parte il 13/01). Sia per la parte opzionale che per quella semplice, l'utente deve invece poter creare una nuova partita, se volete, selezionando la dimensione delle tabelle (deve comunque essere disponibile una dimensione di default,

di 10 caselle per lato) e (se volete) il numero di elementi per ciascuna nave possibile (altrimenti, ci devono essere dei valori di default). È obbligatorio per tutti che siano disponibili almeno sottomarini (dimensione 3) e portaerei (dimensione 5). Le altre tipologie di navi le potete aggiungere a piacimento, ma non influiscono sul voto. Dopo aver iniziato una nuova partita, l'utente deve poter posizionare tutte le sue navi. Per posizionale, deve scegliere la casella "di testa" (es. [3,5]) e l'orientamento (VERTICALE o ORIZZONTALE). La disposizione della nave avviene dalla casella di testa, dall'alto verso il basso o da sinistra verso destra. Il sistema deve verificare che i posizionamenti siano leciti, e comunicare all'utente eventuali errori.

- **AVVERSARIO:** Il sistema fa da avversario (lo chiameremo CPU). CPU posiziona i suoi pezzi in maniera casuale, dopo che lo ha fatto l'utente.
- **PARTITA:** Dopo i posizionamenti, inizia la partita, che funziona a turni. Ad ogni turno, l'utente sceglie una casella ed il sistema verifica se ha colpito/affondato o no una nave avversaria (comunicandolo all'utente) e poi la stessa cosa succede per CPU. Nella versione base, CPU sceglie una casella casuale che non ha ancora scelto[2]. Se decidete di implementare le funzionalità opzionali (**OPZ**), oltre alla versione base, deve esserci una versione "intelligente" di CPU. Lascio a voi eventuali "strategie evolute", non necessarie, ma almeno una volta colpita una nave, deve colpire caselle adiacenti finché non la ha affondata. **CONSIGLIO:** dato che sia l'utente che CPU ad ogni turno devono scegliere una mossa, vi conviene implementare il tutto con un metodo (es. getProssimaMossa), che sarà implementato in maniera diversa per CPU e per l'utente.
- **TERMINE PARTITA:** Dopo ogni turno, il sistema controlla se tutte le navi di un giocatore sono state affondate. In caso positivo, decreta il vincitore. Per chi decide di fare la parte opzionale (**OPZ**), c'è un'altra condizione di terminazione della partita. Il giocatore umano inizia la partita con un certo tempo a disposizione (decidete voi se far iniziare il timer con un tempo richiesto all'utente, o dipendente dalla dimensione delle tabelle). Il tempo è decrementato mentre l'utente sta effettuando la scelta su quale casella dell'avversario colpire. Se il timer arriva a 0, l'utente umano ha perso.

Classi (obbligatorie)

Tutte le classi usate per questo progetto, esclusa eventualmente una classe con il main che vi permette di eseguirlo, devono essere contenute in un package chiamato upo.battleship.cognome (cognome è il cognome di un componente del gruppo, tutto minuscolo). È richiesto che la loro architettura rispecchi il modello MVC, e devono esserci almeno le seguenti classi:

- BattleshipModel
- BattleshipView
- BattleshipController

Siete però liberi di aggiungere (e fortemente consigliati a farlo) altre classi al package, decidendo i loro campi, metodi e visibilità degli stessi.

Consigli

Nota bene: i seguenti consigli sono dati per aiutarvi ad applicare una metodologia di progettazione Object-Oriented "ordinata", tuttavia includono scelte di progettazione fatte dal sottoscritto. Per questo, non

[2] Potete implementare la scelta casuale come volete. Online troverete diversi modi per generare un intero random all'interno di un certo intervallo. Potete generarne 2, corrispondenti alle coordinate della casella, e se poi tale casella è già stata colpita, potete sceglierne una adiacente che non lo sia già. E così via finché non ne trovate una da colpire. Un'altra soluzione, più facile da implementare, è generare inizialmente una lista che contenga tutte le coordinate non ancora colpite (es, {[0,0], [0,1], … [1,0], …}) e poi scegliere, usando un intero random tra 0 e la lunghezza attuale della lista, la posizione della lista da scegliere. Una volta presa la casella, la dovete rimuovere dalla lista per non riscegliarla.

devono essere intese come l'unico (o il più giusto) metodo di fare le cose. Se, una volta letti i requisiti sopra, avete già ben chiaro come procedere, non avete bisogno dei consigli seguenti.

Consigli sul modello:

Ricordate che il modello deve contenere tutte le informazioni relative alla partita in corso. "Contenere" significa che un oggetto BattleshipModel deve o avere uno o più campi che modellano le informazioni, o avere dei riferimenti ad altri oggetti che modellano le informazioni. Ciascun oggetto deve poi fornire dei metodi che gli permettono di modificare il suo stato in base alle azioni "lecite" del gioco/dell'applicazione

Per capire cosa modellare e come, vi consiglio di ragionare in maniera "mista". Partendo in modo top-down, mi chiederei se ci sono dei gruppi di informazioni omogenee tra di loro. Alcune di queste che mi vengono in mente sono:

- Informazioni "generiche" della partita in corso. Ad esempio, quanto sono grandi le tabelle dei giocatori, oppure di chi è il turno (se serve), quante e quali navi sono posizionabili inizialmente, …
- Le informazioni che riguardano lo stato dell'utente (e le azioni che si possono fare su di esse). Ad esempio, quali sono le sue navi e dove sono posizionate.
- Quelle che riguardano lo stato di CPU (e le azioni che si possono fare su di esse)
- Se la strategia di CPU ha uno stato (ad esempio, se richiede di tenere conto delle caselle già colpite) va mantenuta anche quel genere di informazione.
- Se decido di fare anche la parte opzionale, dovrò tenere anche conto del tempo che passa.
- …?

A questo punto, mi focalizzerei sulle informazioni riguardanti l'utente, che (forse) possono essere raggruppate in un singolo oggetto. Sicuramente, sarà necessario modellare:

- Quali e dove sono tutte le sue navi
- Quali sono state colpite e dove
- Quali punti della mappa avversaria ha tentato di colpire e quali di questi hanno avuto esito positivo
- …?

Inoltre, ci sono operazioni che possono essere effettuate, ad esempio:

- Tentare di colpire una casella della mappa avversaria. Questo comporta di aggiungere l'informazione che l'utente ha tentato di colpire la casella, e anche con quale esito
- Ricevere un tentativo di colpo su una propria casella. In questo caso bisogna controllare se in tale casella è presente una nave, e restituire un esito. Inoltre, se una nave è stata colpita, bisogna aggiungere l'informazione al proprio stato.
- …?

Ragionando successivamente su CPU, mi accorgo che le informazioni e le azioni che ho deciso di modellare per l'utente umano sono le stesse (o buona parte) che devo modellare per CPU. Probabilmente, oltre a raggruppare le informazioni in singoli oggetti, questi oggetti possono in qualche maniera avere un modello comune (una classe? Un'interfaccia? Dipende dalle vostre scelte).

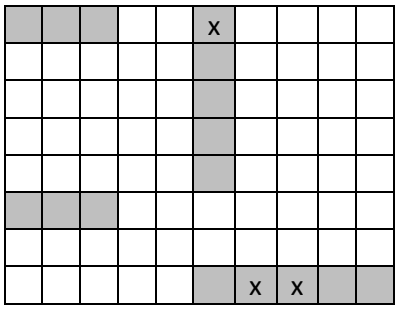
Analogamente, mi potrei accorgere che una nave (che ha diversi sottotipi) ha un suo stato: ha una posizione, una dimensione e può essere colpita in qualche punto, illesa o affondata. Molte di queste informazioni devono essere accessibili sia in lettura (get) sia in scrittura (set). Inoltre, deve essere posizionata nella tabella di un certo utente…. E avanti così.

Prima di passare all'implementazione di vista e controller, vi consiglio di implementare la parte del modello.

Consigli sulla vista:

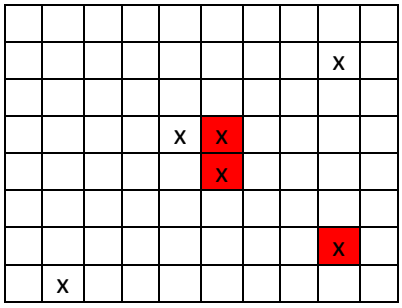
Sia che decidiate di implementare la vista semplice, sia che decidiate di fare quella opzionale, ricordate che la vista deve occuparsi sono di visualizzare lo stato (aggiornato) del modello, e di permettere all'utente di eseguire le azioni permesse.
Un consiglio: non fate cose troppo complicate, dato che non è richiesto. Fate riferimento ai package Java java.awt e javax.swing per cercare eventuali componenti grafici.
Partiamo dalla parte grafica comune ad entrambe le versioni, e analizziamo di cosa è composta:

Una griglia dove il giocatore vede lo stato delle sue navi (posizione, e se sono state colpite). Più o meno, potrebbe essere fatta così:



Dove le celle colorate di grigio sono celle con una nave, mentre quelle con la x sono state colpite.

L'altra griglia, uguale per dimensione, deve far vedere ciò che l'utente conosce la mappa dell'avversario (quindi quali celle l'utente ha cercato di colpire, e in quali di queste ha trovato qualcosa)



Ad esempio, la "x" può segnalare un tentativo, ed il colore rosso il fatto che sia stato colpito qualcosa in quella cella.

Entrambe le griglie possono essere implementate tramite delle estensioni di JPanel. Contestualmente a questo progetto ho caricato un file (MyGridPanel) in cui vi mostro brevemente come si può creare un pannello a griglia, e come si possono modificare i suoi elementi. Il file non rispetta l'organizzazione MVC ed è fatto a solo scopo illustrativo.

I due pannelli così costruiti, possono essere posizionati all'interno di uno stesso JFrame, magari affiancati, o anche in container top level differenti. Ricordate che potete usare anche dei container di tipo JDialog.

Consiglio importante sulla modularità di MVC:

Un ultimo consiglio sull'utilizzo del pattern MVC e sulla modularità. Immaginiamo che nel modello abbiamo deciso che ci sono due oggetti che modellano le mappe di un giocatore: **mappaMieNavi**, e **mappaTentativi** (in questo momento, non ci interessa come sono implementati). Dovrebbe essere evidente che le due immagini disegnate qui sopra (quella con alcune caselle grigie – che chiameremo **vistaMieNavi** - e quella

con alcune caselle rosse - **vistaTentativi**) siano ognuna la vista di uno dei due oggetti del modello. Come mappaMieNavi, e mappaTentativi saranno contenute in BattleshipModel, vistaMieNavi e vistaTentativi saranno contenute in BattleshipView. Tuttavia, ai singoli oggetti non serve sapere che il loro rispettivo modello/vista è contenuto dentro ad un'altra classe. Quindi, se usate il pattern Observer/Observable, potete tranquillamente fare in modo che vistaMieNavi, ad esempio, sia Observer di mappaMieNavi, senza far passare i messaggi da BattleshipModel e BattleshipView. Questo dovrebbe semplificare di molto il vostro codice.

Ho caricato un package upo.aiutobattleship.aiutoMVC che illustra più o meno questa cosa. In pratica, voglio gestire 2 elementi: un pannello colorato (con un menu a tendina – JComboBox – che permette di modificare il colore) ed un pannello con un testo modificabile tramite un JTextField. Per prima cosa, ho implementato il modello MVC per entrambi gli elementi separatamente:

- SemaforoModel, SemaforoView e SemaforoController
- TestoModel, TestoView e TestoController

Successivamente, verificato che funzionassero correttamente, ho implementato il modello MVC che li trattava insieme:

- AiutoMVCModel: che inizializza e gestisce SemaforoModel e TestoModel
- AiutoMVCView: che inizializza e gestisce SemaforoView e TestoView
- AiutoMVCController: che inizializza e gestisce SemaforoController e TestoController

In questo modo, tutti gli elementi sono isolati ed il loro codice è molto semplice. Nella vostra implementazione, il modello, la vista ed il controller ad alto livello possono anche fare qualcosa in più oltre a "contenere" gli elementi a basso livello. Il mio è ovviamente solo un esempio.

Importante: uso di DIR

Chi avesse dubbi sul laboratorio può (e deve) **usare il forum su DIR**, controllando prima che non siano già state fatte domande simili. Questo permetterà a tutti di avere le stesse informazioni. Risponderò velocemente alle domande fatte prima del 27/12 e dopo il 4/01. Nella settimana in mezzo avrò difficile accesso ad internet, quindi se non vi risponderò aspettatevi una risposta dopo il 4/01.