ESAME DI FONDAMENTI DI INFORMATICA T-2 del 22/7/2021

Proff. E. Denti – R. Calegari – A. Molesini

Tempo a disposizione: 3 ore

NOME PROGETTO ECLIPSE: CognomeNome-matricola (es. RossiMario-0000123456)
NOME CARTELLA PROGETTO: CognomeNome-matricola (es. RossiMario-0000123456)
NOME ZIP DA CONSEGNARE: CognomeNome-matricola.zip (es. RossiMario-0000123456.zip)
NOME JAR DA CONSEGNARE: CognomeNome-matricola.jar (es. RossiMario-0000123456.jar)

Si devono consegnare DUE FILE: *l'intero progetto Eclipse* e *il JAR eseguibile*

Si ricorda che compiti *non compilabili* o *palesemente lontani da 18/30* NON SARANNO CORRETTI e causeranno la verbalizzazione del giudizio "RESPINTO"

È stata richiesta una app per giocare a ***Battaglia Navale***, **nella versione *solitario***. L'obiettivo è giocare contro il computer, indovinando la posizione delle navi mediante l'uso intelligente delle informazioni fornite.

DESCRIZIONE DEL DOMINIO DEL PROBLEMA

La Battaglia navale in solitario è un gioco di logica e abilità costituito da una *griglia* di 8×8 celle, ciascuna delle quali può contenere o un ***elemento di nave*** o il ***mare***. Inizialmente, quasi tutte le celle della griglia del giocatore sono ***vuote***, tranne alcune fornite come base di partenza.

Le navi possono essere di quattro ***tipi***:

- *Portaerei* (4 elementi: 2 estremi + 2 centrali)
- *Incrociatori* (3 elementi: 2 estremi + 1 centrale)
- *Cacciatorpedinieri* (2 elementi, entrambi estremi)
- *Sommergibili* (un singolo elemento)

Gli ***elementi*** che costituiscono le navi possono essere:

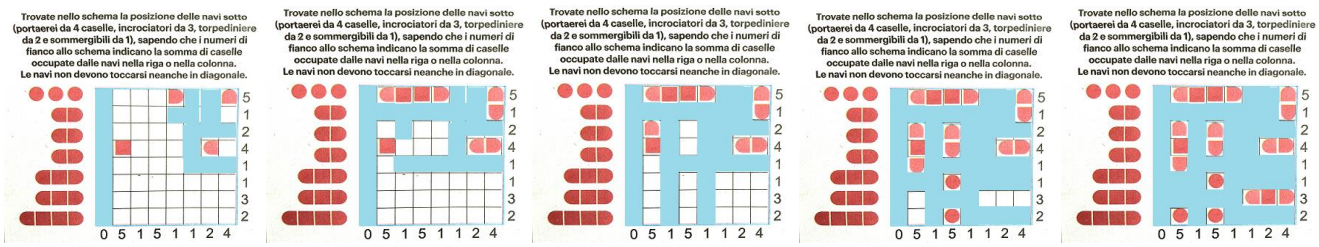
- Elementi orizzontali (estremo sinistro, estremo destro)
- Elementi verticali (estremo superiore, estremo inferiore)
- Elemento centrale (un quadrato)
- Elemento singolo (un cerchio = sommergibile)

Lo scopo del gioco è capire dove siano le varie navi, sapendo che:

- Il totale di elementi di ogni riga/colonna è riportato a destra/sotto la riga/colonna corrispondente
- Intorno a ogni nave deve esserci del mare: due navi non possono mai toccarsi, neanche in diagonale

STRATEGIA: il giocatore inizia collocando il mare intorno agli elementi di nave noti e prosegue poi deducendo via via le possibili posizioni degli altri tenendo conto dei totali di riga/colonna forniti.

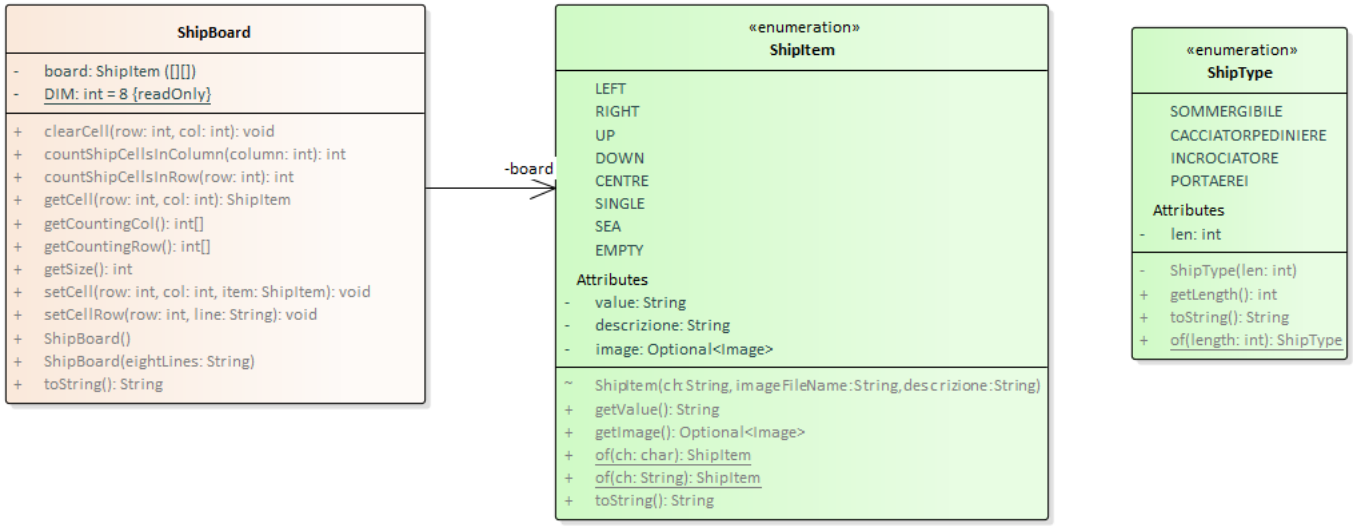
Per curiosità, sotto viene mostra il corrispondente schema via via risolto.



TEMPO STIMATO PER SVOLGERE L'INTERO COMPITO: 1h40 – 2h10

Parte 1 (punti: 17)

Modello dei dati (package battleship.model) [TEMPO STIMATO: 20-25 minuti] *(punti: 7)*



SEMANTICA:

a) L'enumerativo ***ShipType*** (fornito) definisce i quattro tipi possibili di nave, ciascuno con associata la propria lunghezza recuperabile tramite il metodo *getLength*. Un metodo factory *of* consente di ottenere l'enumerativo "giusto" per una lunghezza data. Un'apposita *toString* completa il tutto.

b) L'enumerativo ***ShipItem*** (fornito) definisce gli otto tipi possibili di elementi, di cui 6 elementi di nave più il mare e il caso della casella vuota. Ognuno è caratterizzato da varie proprietà, in particolare

- la stringa (singolo carattere) corrispondente, recuperabile tramite il metodo *getValue*
- l'immagine associata, recuperabile tramite il metodo *getImage*
- la descrizione testuale corrispondente, restituita da *toString*

Anche in questo caso una coppia di metodi factory *of* consente di ottenere l'enumerativo "giusto" a partire dal carattere dato (uno degli otto possibili).

c) la classe ***ShipBoard*** (**fornita parzialmente realizzata ma da completare**) rappresenta lo schema di gioco: ne verranno usate due istanze, una per rappresentare la soluzione (immutabile), l'altra per rappresentare la situazione attuale della scacchiera del giocatore (ovviamente modificabile). Per ipotesi, la scacchiera è *sempre 8x8*. La classe mette a disposizione i seguenti metodi:

- *un costruttore* per la scacchiera inizialmente vuota
- *un costruttore* con argomento una stringa di otto righe, corrispondenti al contenuto iniziale della scacchiera; ogni riga è costituita da una sequenza di singoli caratteri separati fra loro da spazi.
  I caratteri ammessi sono soltanto **<**, **>**, **^**, **v**, **x** , **o** per gli elementi-nave, più **~** per il mare e **#** per denotare la cella vuota. [NB: il carattere ~ si ottiene con ALT+126 nelle tastiere italiane].
  Il costruttore, come anche il reader, fa uso del sottostante metodo *setCellRow*.
- *setCellRow* consente di caricare nello schema un'intera riga di valori, secondo le medesime convenzioni sopra indicate. Il metodo deve controllare i valori di riga/colonna ricevuti, che ovviamente devono essere compresi nel range 0..DIM-1, lanciando ***IllegalArgumentException*** in caso di non conformità, con adeguata messaggistica.
- *getCell* restituisce lo ***ShipItem*** corrispondente all'attuale contenuto della cella relativa agli indici ricevuti come argomento. Come sopra, il metodo deve controllare accuratamente i parametri ricevuti, lanciando ***IllegalArgumentException*** in caso di non conformità.
- *getSize* restituisce la dimensione della scacchiera

- *clearCell* imposta a ***ShipItem.EMPTY*** il contenuto della cella relativa agli indici di riga e colonna specificati, che ovviamente devono essere controllati, lanciando ***IllegalArgumentException*** in caso di non conformità.
- *setCell* imposta il contenuto della cella relativa agli indici di riga e colonna specificati al valore di ***ShipItem*** specificato; anche in questo caso ovviamente gli indici devono essere controllati come sopra, lanciando ***IllegalArgumentException*** in caso di non conformità.
- *countShipCellsInRow* e *countShipCellsInColumn* restituiscono rispettivamente il numero di caselle di tipo nave (quindi, *non mare e non vuote)* nella riga o colonna specificata.
- *getCountingRow* e *getCountingCol* restituiscono rispettivamente l'array di interi relativo ai "suggerimenti" verticali (riga extra da posizionare sotto la scacchiera) / orizzontali (colonna extra da posizionare alla destra della scacchiera).
- *toString* restituisce una stringa con la struttura della scacchiera (in righe) intesa come sequenza di valori che etichettano le varie caselle, utilizzando i caratteri associati a ogni ***ShipItem.***

IMPORTANTE: data la presenza di una ***Image*** JavaFX in ***ShipItem***, e quindi di riflesso anche in ***ShipBoard***, **per far girare i test è indispensabile aggiungere i soliti argomenti alla run configuration di ogni test**

```
-ea --module-path ……\javafx-sdk-15.0.1\lib --add-modules javafx.controls
```

Persistenza (battleship.persistence) **[TEMPO STIMATO: 35-45 minuti]** *(punti 10)*

Ci sono due file di testo:

- **battlefield.txt** contiene la soluzione, ossia la disposizione di navi e mare che il giocatore deve indovinare
- **initialfield.txt** contiene la configurazione iniziale della scacchiera del giocatore.

Entrambi sono formattati secondo lo stesso schema, ovvero con esattamente otto righe costituite ciascuna da una sequenza di otto singoli caratteri separati fra loro da spazi. I caratteri ammessi sono soltanto **<**, **>**, **^**, **v**, **x**, **o** per gli elementi-nave, più **~** per il mare e **#** per denotare la cella vuota. Per ipotesi:

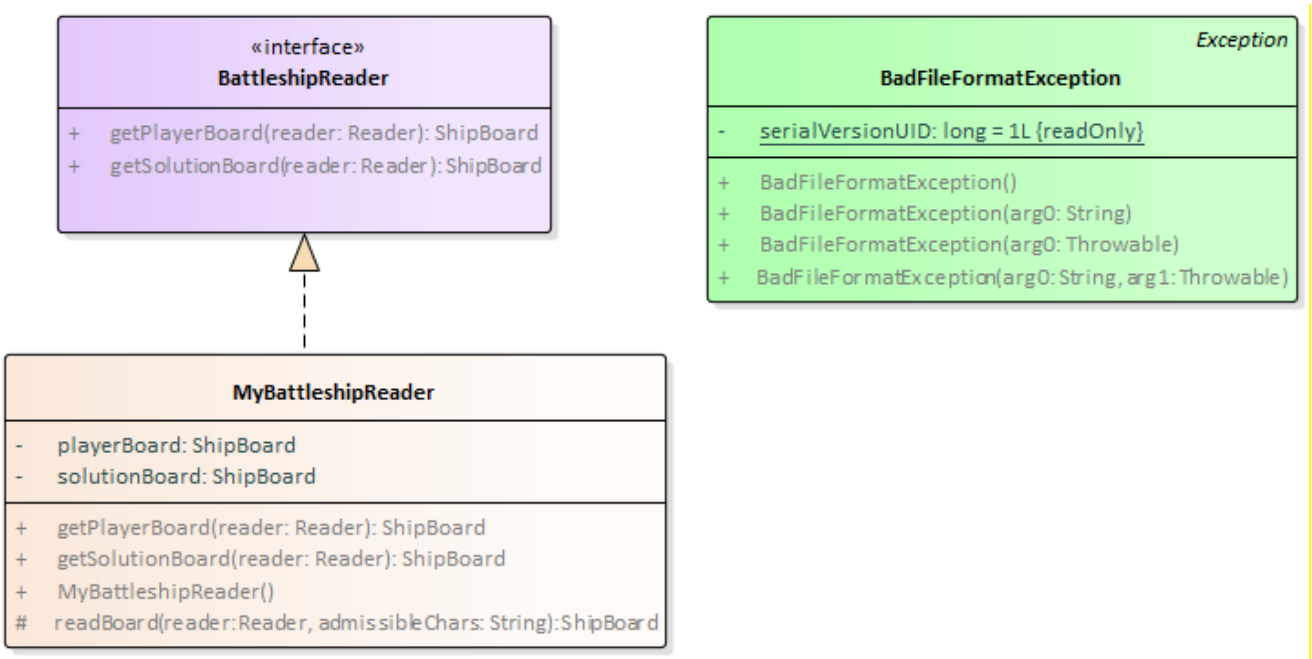
- nel file **battlefield.txt** non vi sono celle vuote (quindi, non è mai presente il carattere **#**)
- nel file **initialfield.txt** non vi sono solo celle di mare (quindi, non è mai presente il carattere **~**).

Esempi:

battlefield.txt	initialfield.txt
~ < x x > ~ ~ ^ ~ ~ ~ ~ ~ ~ ~ v ~ ^ ~ ^ ~ ~ ~ ~ ~ x ~ v ~ ~ < > ~ v ~ ~ ~ ~ ~ ~ ~ ~ ~ o ~ ~ ~ ~ ~ ~ ~ ~ ~ < x > ~ o ~ o ~ ~ ~ ~	# # # # > # # ^ # # # # # # # # # # # # # # # # # x # # # # < #

Poiché la struttura dei due file è identica, l'architettura prevede un unico reader con due distinti metodi di lettura, che costituiscono due entry point per lo stesso metodo di lettura fisica, con diverso set di caratteri ammissibili.

Il diagramma UML è illustrato alla pagina seguente.



SEMANTICA:

a) L'interfaccia ***BattleShipReader*** (fornita) dichiara i due metodi *getSolutionBoard e getPlayerBoard* che restituiscono rispettivamente la scacchiera-soluzione e la scacchiera iniziale lette dal file; al fine di evitare letture ripetute, la prima volta che essi vengono invocati memorizzano nello stato del reader la ***ShipBoard*** letta, che viene poi restituita a ogni invocazione successiva dello stesso metodo.

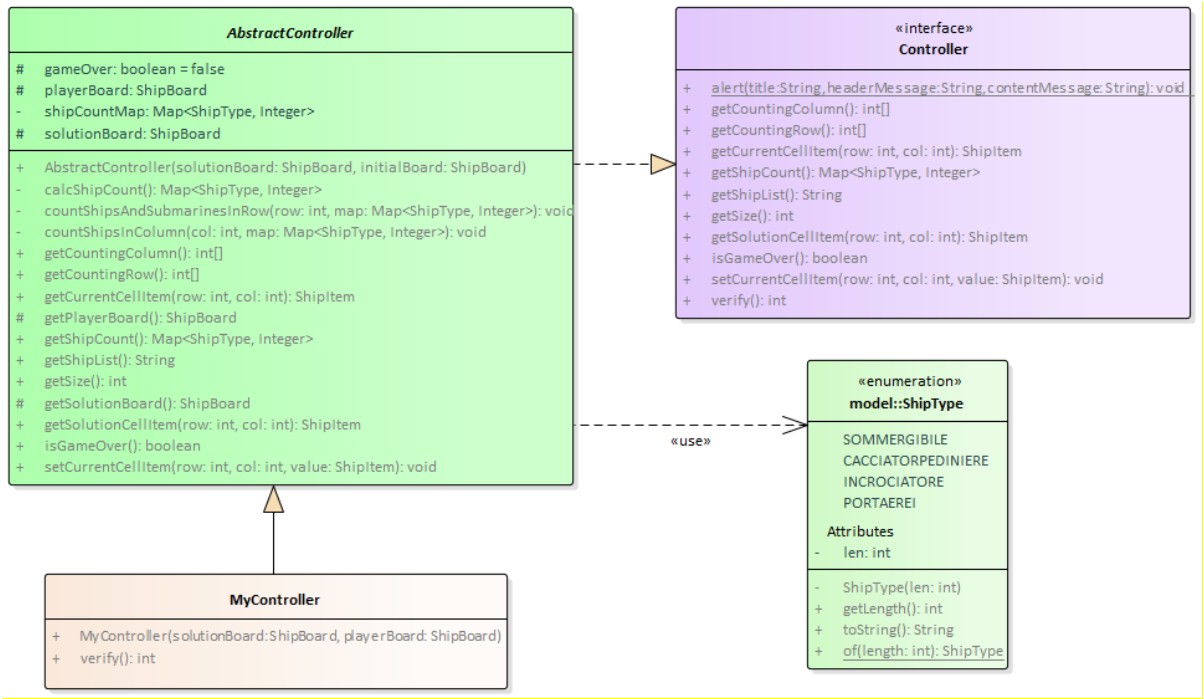
b) La classe ***MyBattleShipReader*** (**da realizzare**) implementa ***BattleShipReader***

- *costruttore* di default che si limita a inizializzare lo stato interno, senza ancora effettuare letture
- *getSolutionBoard* e *getPlayerBoard* che si appoggiano al metodo protetto *readBoard*, il cui primo argomento è un ***Reader*** già aperto, il secondo è una stringa che specifica i caratteri ammissibili per quella scacchiera; come già anticipato, ognuno di questi metodi effettua realmente la lettura solo la prima volta che viene invocato, memorizzando il risultato nello stato interno del reader, così da poterlo facilmente restituire alle invocazioni successive senza dover rifare alcuna lettura.

Parte 2 (punti: 13)

Controller (package battleship.controller) [TEMPO STIMATO: 15-20 minuti] *(punti: 5)*

Il Controller è organizzato secondo il diagramma UML in figura.



SEMANTICA:

a) L'interfaccia ***Controller*** (fornita) dichiara dieci metodi, otto dei quali costituiscono semplici entry point ad omonimi metodi di ***ShipBoard***:

- *getSize* restituisce la dimensione della scacchiera
- *getCountingRow / getCountingCol* si rimappano sui quasi-omonimi metodi di ***ShipBoard***
- *getCurrentCellItem / setCurrentCellItem* si rimappano sui quasi-omonimi metodi di ***ShipBoard*** relativamente alla scacchiera giocatore
- *getSolutionCellIItem* si rimappano sul quasi-omonimo metodo di ***ShipBoard*** relativamente alla scacchiera soluzione
- *isGameOver* restituisce lo stato del controller relativamente all'eventuale raggiungimento della fine del gioco (nessuna cella vuota, tutte le celle della scacchiera giocatore identiche a quelle della soluzione)
- *getShipList* restituisce una stringa che elenca numero e tipo della navi presenti nella soluzione

Sono invece peculiari del controller i due metodi:

- *verify,* che confronta lo stato attuale della scacchiera giocatore con la soluzione, contando e restituendo il numero di celle non vuote diverse (ossia, sbagliate): contemporaneamente, aggiorna lo stato interno del controller relativamente al campo-dati *gameOver*, che diviene *true* solo se nella scacchiera giocatore non vi è più alcuna cella vuota e tutte le celle sono identiche a quelle della soluzione.
- *getShipCount*, che restiuisce una mappa *<ShipType, Integer>* che conta quante navi ci sono nella scacchiera-soluzione per ogni tipo di nave

NB: il Controller contiene anche il **metodo statico ausiliario *alert***, utile per mostrare avvisi all'utente.

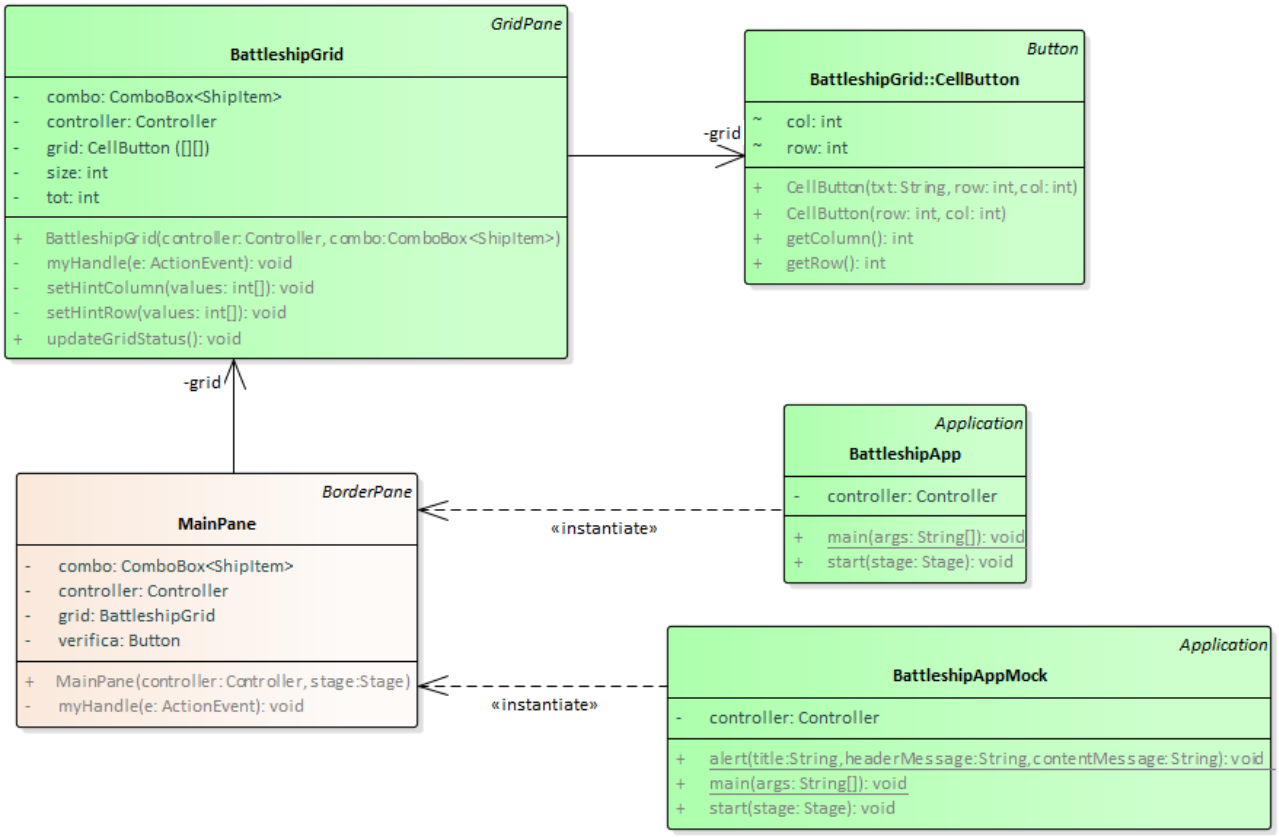
b) `La classe` ***AbstractController*** (fornita) implementa quasi totalmente tale interfaccia

- il *costruttore* riceve le due scacchiere (***ShipBoard***) relative alla soluzione e alla configurazione iniziale
- i due metodi protetti *getSolutionBoard* / *getPlayerBoard* restituiscono le ***ShipBoard*** memorizzate nello stato del controller
- rimane astratto il metodo *verify*

c) La classe ***MyController*** (**da realizzare**) estende e completa ***AbstractController*** implementando i metodi mancanti, secondo le specifiche dell'interfaccia ***Controller***.

Interfaccia utente (package battleship.ui) **[TEMPO STIMATO: 25-40 minuti]** ***(punti: 8)***

L'interfaccia utente è illustrata nelle figure seguenti e segue il modello sotto illustrato:



La classe ***BattleshipApp*** (fornita) costituisce l'applicazione JavaFX che si occupa di aprire i file, creare il controller e incorporare il ***MainPane***. Per consentire di collaudare la GUI anche in assenza / in caso di malfunzionamento della parte di persistenza, è possibile avviare l'applicazione mediante la classe ***BattleshipAppMock***.

SEMANTICA:

a) La classe ***BattleshipGrid*** **(fornita)** fornisce un componente pronto per l'uso che mostra e gestisce la griglia di pulsanti che costituiscono la GUI della scacchiera del giocatore, con le due colonne/righe di ausilio a destra e sotto. In particolare:
   - il *costruttore* riceve il ***Controller*** e un riferimento alla ***ComboBox*** del ***MainPane,*** utile per estrarre l'elemento scelto dall'utente per poi impostare al giusto ***ShipItem*** la casella premuta;
   - il metodo *updateGridStatus* aggiorna la visualizzazione e lo stato interno della griglia: va chiamato dopo ogni modifica che implichi una gestione di eventi del ***MainPane***

b) La classe ***MainPane*** (da realizzare) estende ***BorderPane*** e prevede:
   1) nel lato sinistro, in verticale, prima una ***ComboBox*** popolata con gli ***ShipItem***, poi alcune ***Label*** che riportano l'elenco delle navi, indi un pulsante VERIFICA che scatena il confronto fra l'attuale scacchiera giocatore e la soluzione retrostante (che non viene però mai mostrata).
   2) nella parte centrale, una ***BattleshipGrid***.

Inizialmente, il pannello mostra la configurazione iniziale (Fig. 1). Per inserire un elemento nella griglia, l'utente deve prima selezionarlo dalla combo, poi premere il pulsante-cella corrispondente (Fig. 2).

Proseguendo nel gioco, in qualunque momento l'utente può premere il pulsante VERIFICA che scatena, tramite il metodo *verify* del controller, la verifica sull'eventuale presenza di celle errate, mostrandone il numero (Figg. 3 e 4).

Nel caso vi siano celle errate, esse NON vengono immediatamente resettate a livello grafico: la gestione dell'evento deve invece far comparire un apposito dialogo (Figg. 3 e 4) che riporta l'esito della verifica. SOLO DOPO che l'utente preme il tasto OK nel dialogo si procede ad aggiornare lo stato della griglia, "sbiancando" le celle errate (Fig. 5). In questo modo è impossibile proseguire nel gioco con configurazioni errate: il sistema convalida e accetta solo configurazioni corrette, guidando via via verso la soluzione.

Il gioco termina quando, dopo aver riempito tutte le caselle, la verifica finale dà esito positivo (Fig. 6): anche in tal caso comunque la scacchiera giocatore resta attiva, permettendo eventualmente all'utente di continuare a modificare celle per divertimento.

La gestione dell'evento relativo al pulsante VERIFICA deve:

- effettuare la verifica della situazione, tramite il metodo *verify* del controller
- mostrare la finestra di dialogo (utile il metodo statico *alert* del controller) con idonea messaggistica adeguata alla specifica situazione
- aggiornare lo stato della griglia.

Cose da ricordare

- salva costantemente il tuo lavoro: l'informatica a volte può essere "subdolamente ostile"..
- in particolare: se ora compila e stai per fare modifiche, salva la versione attuale (non si sa mai)

Checklist di consegna

- Hai fatto un **JAR eseguibile**, che contenga cioè l'indicazione del main?
- Hai controllato che **si compili** e **ci sia tutto**? [NB: non includere il PDF del testo]
- Hai **rinominato** IL PROGETTO, lo ZIP e il JAR esattamente come richiesto?
- Hai **chiamato** la cartella del progetto esattamente come richiesto?
- Hai fatto un **unico file ZIP (NON .7z, rar o altri formati)** contenente l'intero progetto? In particolare, ti sei assicurato di aver incluso tutti i file .java (e non solo i .class)?
- **Hai consegnato DUE file distinti, ossia lo ZIP col progetto e il JAR eseguibile?**
- Su EOL, hai **premuto** il tasto "CONFERMA" per inviare il tuo elaborato?

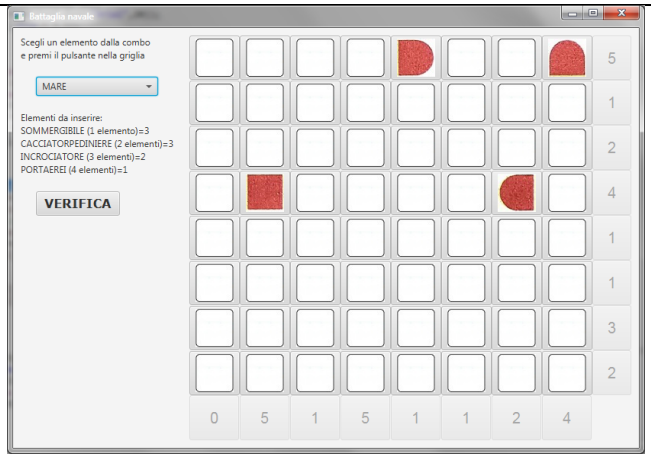


Fig.1

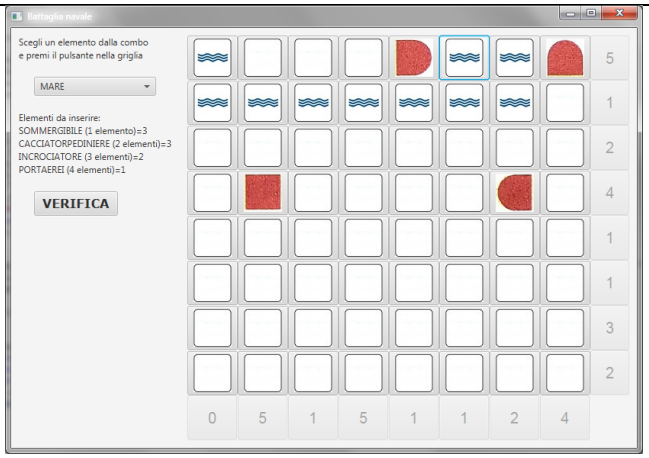


Fig.2

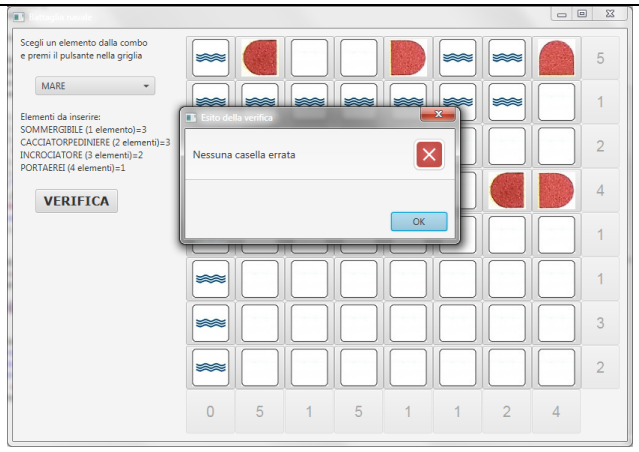


Fig.3

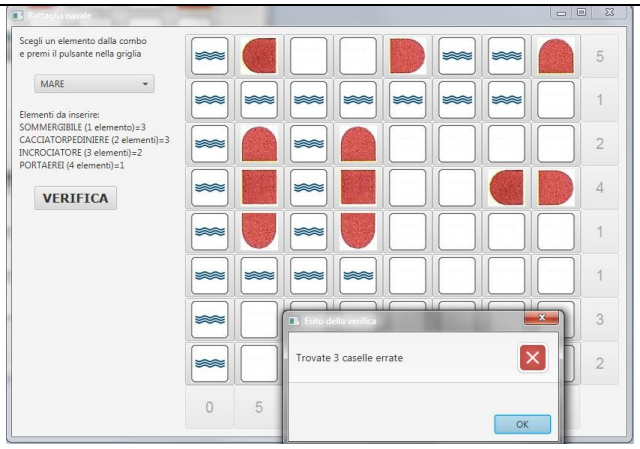


Fig.4

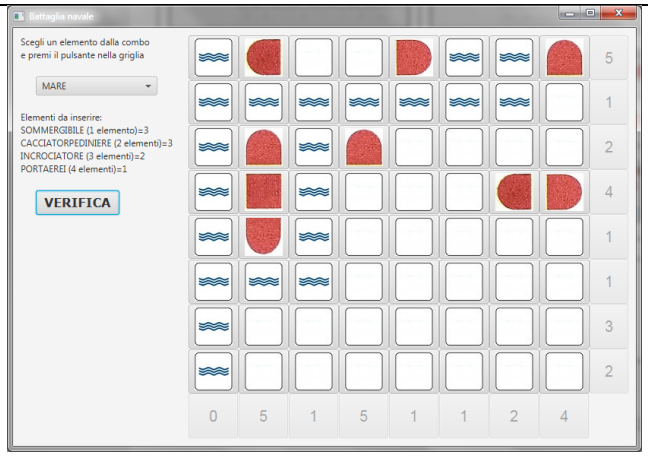


Fig.5

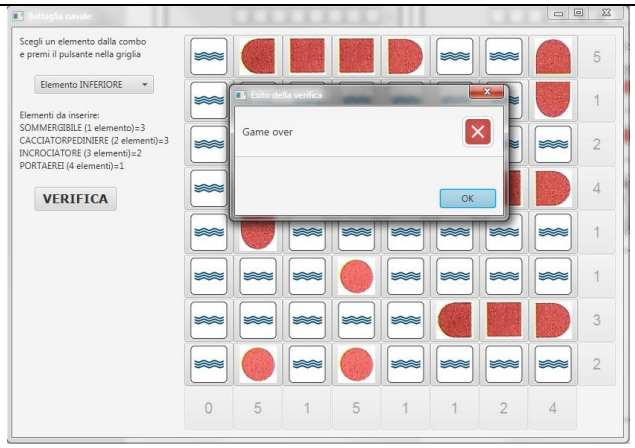


Fig.6