

# CLASSE GIOCO

---

## ✓ `verificaFinePartita()`

Controlla se la partita è finita.

In pratica:

- Se il giocatore ha totalizzato **17 colpi a segno** (cioè ha affondato tutte le navi del bot), chiude la finestra di gioco (`dispose()`) e apre la schermata di fine partita vincente (`FinePartita(true)`).
  - Se invece il bot ha totalizzato 17 colpi a segno, chiude la finestra e apre la schermata di sconfitta (`FinePartita(false)`).
- 

## ✓ `provaAttacco(int x, int y)`

Viene chiamato quando il giocatore clicca su una cella della griglia del bot per attaccare.

Passaggi:

1. Chiama `bot.riceviAttacco(x, y)` per sapere se il colpo ha colpito una nave.
2. Se ha colpito:
  - Incrementa `colpiGiocatore`.
  - Colora la cella colpita di **rosso**.
3. Se ha mancato:
  - Colora la cella di **blu**.
4. In entrambi i casi, disattiva il pulsante per impedirne un nuovo clic.
5. Dopo l'attacco del giocatore, il **bot risponde**:
  - Usa `bot.effettuaAttacco()` per scegliere un punto da colpire.

- Chiama `giocatore.riceviAttacco(...)` per verificare se ha colpito.
  - Se colpisce, colora la cella in **rosso** e incrementa `colpiBot`.
  - Se manca, colora la cella in **blu**.
6. Registra il risultato dell'attacco del bot tramite `bot.registraEsitoAttacco(...)`.
  7. Dopo entrambi gli attacchi, chiama `verificaFinePartita()` per vedere se qualcuno ha vinto.

---

### ✓ `posizionaNave(int x, int y)`

Serve per posizionare le navi del giocatore all'inizio della partita.

Funziona così:

1. Controlla se hai già posizionato tutte le navi (`indiceNaveCorrente >= dimensioniNavi.length`). Se sì, mostra un messaggio e non fa nulla.
  2. Crea una nuova nave con la dimensione corretta (`new Nave(...)`).
  3. Chiede al giocatore di posizionarla con `giocatore.posizionaNave(...)`.
  4. Se il posizionamento va a buon fine:
    - Aggiorna la griglia visivamente colorando di **nero** le celle in cui è posizionata la nave.
    - Incrementa l'indice della prossima nave.
    - Mostra nella sidebar un messaggio sulla nave posizionata e su quella successiva.
  5. Se il posizionamento non è valido (per esempio uscirebbe dai bordi o si sovrapporrebbe), mostra un messaggio di errore.
-

### ✓ `mostraPreview(int x, int y, boolean mostra)`

Quando il mouse passa sopra la griglia, mostra o nasconde un'**anteprima** della nave che stai per posizionare.

- Se `mostra` è `true`, colora di **nero** le celle dove la nave verrebbe posizionata.
  - Se `mostra` è `false`, ripristina i colori:
    - Se c'è già una nave nella cella, la lascia nera.
    - Altrimenti, la riporta al colore azzurro di base.
  - Calcola la posizione delle celle della nave in base all'orientamento (`orizzontale`) e alla lunghezza della nave corrente (`dimensioniNavi[indiceNaveCorrente]`).
  - Non mostra nulla se hai già posizionato tutte le navi.
- 

### ✓ `creaGriglia(JButton[][] griglia, boolean isGiocatore, JPanel panel)`

Crea e visualizza la griglia di gioco (10x10) con i pulsanti.  
Esegue queste operazioni:

1. Svuota il pannello grafico (`panel.removeAll()`).
2. Imposta un layout a griglia 10x10.
3. Crea ogni bottone (`JButton`) con dimensione, colori, bordo e altri parametri grafici.
4. Salva il bottone nella matrice `griglia[i][j]`.
5. Aggiunge i comportamenti:
  - **Se `isGiocatore` è `true`:**
    - Clic: posiziona la nave chiamando `posizionaNave`.

- Passaggio del mouse: mostra l'anteprima con `mostraPreview`.
  - Se `isGiocatore` è `false` (quindi griglia del bot):
    - Clic: effettua l'attacco chiamando `provaAttacco`.
6. Aggiunge ogni bottone al pannello.
  7. Alla fine rinfresca il pannello con `revalidate()` e `repaint()`.

### ✓ Costruttore `public Gioco(boolean diff)`

Questo metodo viene eseguito **quando viene creato un oggetto `Gioco`**, cioè quando si avvia la partita.

Riceve come parametro `diff`, un valore booleano che indica la **difficoltà** (facile o difficile).

Ecco cosa fa, **passo per passo**:

1. `initComponents();`
  - Inizializza i componenti grafici generati da NetBeans (interfaccia, pannelli, bottoni, ecc.).
2. `this.setLocationRelativeTo(null);`
  - Posiziona la finestra **al centro dello schermo**.
3. `difficolta = diff;`
  - Salva il valore della difficoltà in un campo della classe.

**Imposta il titolo della finestra** in base alla difficoltà:

```
setTitle(difficolta ? "Gioco -> Difficile" : "Gioco -> Facile");
```

- 4.

**Mostra nella sidebar** il messaggio che invita a posizionare la prima nave:

```
jTextAreaSideBar.append("Posiziona le tue navi: dimensione attuale " +  
dimensioniNavi[indiceNaveCorrente] + "\n");
```

5.

### **Inizializza le griglie:**

```
grigliaGiocatore = new JButton[10][10];  
grigliaBot = new JButton[10][10];
```

6. Due matrici 10x10 di bottoni: una per il giocatore, una per il bot.

### **Crea graficamente le griglie:**

```
creaGriglia(grigliaGiocatore, true, jPanelGrigliaGiocatore);  
creaGriglia(grigliaBot, false, jPanelGrigliaBot);
```

7.

### **Crea gli oggetti **Giocatore** e **Bot**:**

```
giocatore = new Giocatore(grigliaGiocatore);  
bot = new Bot(grigliaBot, difficolta);
```

8.

### **Fa posizionare automaticamente le navi al bot:**

```
bot.posizionaNaviAutomaticamente();
```

9.

---

## **Gestione tasto "R" per ruotare le navi**

Permette al giocatore di premere il tasto **"R"** per cambiare l'orientamento delle navi (orizzontale ↔ verticale).

Funziona così:

**Si intercetta il tasto R** anche se non c'è il focus su un bottone:

```
getRootPane().getInputMap(JComponent.WHEN_IN_FOCUSED_WINDOW).put(KeyStroke.getKeyStroke("R"), "ruota");
```

1.

**Si associa l'azione chiamata "ruota" a un blocco di codice:**

```
getRootPane().getActionMap().put("ruota", new AbstractAction() {  
  
    @Override  
  
    public void actionPerformed(ActionEvent e) {  
  
        orizzontale = !orizzontale;  
  
        jTextAreaSideBar.append("Orientamento ruotato a " + (orizzontale ?  
"orizzontale" : "verticale") + "\n");  
  
    }  
  
});
```

2.

3. Quando premi il tasto R:

- Cambia il valore della variabile **orizzontale** (da **true** a **false** e viceversa).
  - Mostra nella sidebar il nuovo orientamento scelto.
- 

## CLASSE GIOCATORE

---

## ✓ `public class Giocatore`

Questa classe rappresenta il **giocatore umano** nella battaglia navale.

Contiene:

- `boolean[][] griglia`: una **matrice 10x10** che memorizza dove sono posizionate le navi (`true` = cella occupata da una nave).
- `JButton[][] pulsantiGriglia`: la **griglia di pulsanti grafici** del giocatore (per colorare le celle).
- `List<Nave> navi`: una lista di oggetti `Nave`, che tiene traccia di tutte le navi posizionate dal giocatore.

---

## ✓ `public Giocatore(JButton[][] pulsantiGriglia)`

È il **costruttore**: riceve in input la griglia grafica (bottoni) del giocatore e la salva nella variabile della classe.

---

## ✓ `public boolean posizionaNave(Nave nave, int x, int y, boolean orizzontale)`

Serve per **posizionare una nave** del giocatore sulla griglia.

1. Usa il metodo `posiziona()` della nave per verificare se può essere inserita nella posizione richiesta.
2. Se il posizionamento ha successo:
  - Aggiunge la nave alla lista `navi`.
  - Colora di **grigio** le celle corrispondenti sulla griglia visiva.
  - Ritorna `true`.

3. Se non è possibile, ritorna `false`.
- 

### ✓ `public boolean haNave(int x, int y)`

Controlla se nella cella `(x, y)` è presente **una nave**.

1. Scorre tutte le navi.
  2. Per ogni nave, controlla se una delle sue coordinate corrisponde a `(x, y)`.
  3. Se trova una corrispondenza, ritorna `true`; altrimenti `false`.
- 

### ✓ `public boolean riceviAttacco(int x, int y)`

Serve per **gestire un attacco del bot** contro il giocatore.

1. Scorre tutte le navi e le loro posizioni.
  2. Se trova che l'attacco colpisce una nave **non ancora affondata**:
    - Chiama `subisciColpo()` sulla nave.
    - Colora di **rosso** la cella colpita.
    - Se la nave è stata affondata, colora **tutte le sue celle di blu scuro**.
    - Aggiorna la griglia logica (`griglia[x][y] = false`).
    - Ritorna `true` (colpo a segno).
  3. Se nessuna nave è colpita:
    - Colora la cella di **bianco**.
    - Ritorna `false`.
-



### ✓ `public boolean haPerso()`

Controlla se il giocatore ha **perso la partita**.

Usa lo **stream** Java per verificare se **tutte le navi sono affondate**:

```
return navi.stream().allMatch(Nave::isAffondata);
```

- 

---

### ✓ `public List<Nave> getNavi()`

Restituisce la **lista delle navi del giocatore**.

---

### ✓ `public boolean puoPosizionare(Nave nave, int x, int y, boolean orizzontale)`

Verifica **se è possibile posizionare una nave** in una certa posizione senza uscire dalla griglia o sovrapporsi ad altre navi.

1. Per ogni cella in cui si estende la nave:
  - Calcola le coordinate `nx` e `ny`.
  - Controlla se è **fuori dalla griglia** (`nx >= 10` o `ny >= 10`).
  - Controlla se c'è già una nave (`griglia[nx][ny] == true`).
2. Se tutte le celle sono valide, ritorna `true`; altrimenti `false`.

---

## CLASSE BOT

---

## ◆ public class Bot extends Giocatore

La classe `Bot` rappresenta il **giocatore automatico**, cioè il computer, nella modalità **Player vs Bot**.

### 🔧 Variabili principali

- `Random random`: per generare numeri casuali.
  - `boolean difficile`: se `true`, il bot usa una strategia più intelligente.
  - `Deque<Point> bersagliAdiacenti`: coda di coordinate da colpire vicino a un punto colpito con successo.
  - `boolean[][] celleColpite`: tiene traccia delle celle già attaccate dal bot.
  - `Point ultimoColpo`: memorizza l'ultima cella colpita.
  - `Point direzioneAttacco`: direzione in cui il bot sta provando a colpire (es. destra, sinistra, su, giù).
- 

## ◆ Costruttore

```
public Bot(JButton[][] pulsantiGriglia, boolean difficile)
```

Inizializza il bot, salvando:

- La griglia di pulsanti,
  - Se deve usare la modalità difficile,
  - La mappa delle celle già colpite,
  - La lista dei bersagli adiacenti da provare.
- 

## ◆ posizionaNave(...)

Sovrascrive (`@Override`) il metodo del `Giocatore` per il posizionamento delle navi.

- Tenta di posizionare una nave nella griglia logica (`griglia`), senza colorare bottoni (perché è un bot).
  - Se riesce, la aggiunge alla lista `navi` e ritorna `true`.
- 

## ♦ `posizionaNaviAutomaticamente()`

Metodo chiamato per far sì che il bot **posizioni automaticamente le sue navi**.

- Usa un array di dimensioni `[5, 4, 3, 3, 2]` per rappresentare le navi da piazzare.
  - Per ogni nave, sceglie finché non trova una posizione valida.
- 

## ♦ `effettuaAttacco()`

Questo è il **cuore dell'intelligenza del bot**: sceglie dove attaccare il giocatore.

### **Comportamenti:**

1. **Se in modalità difficile** e ha una direzione già impostata:
  - Prova a continuare in quella direzione (es. se ha colpito una nave, cerca di affondarla seguendo la linea).
  - Se la direzione è invalida o va fuori dai limiti, la azzera.
2. **Se ha bersagli adiacenti in coda:**
  - Li estrae uno a uno finché trova una cella non ancora colpita.
3. **Se non ha niente di utile (o è in modalità normale):**

- Colpisce una cella casuale, ma in **modalità difficile** segue una strategia "a scacchiera" (colpisce solo celle pari in somma  $x+y$ ).

Alla fine:

- Segna la cella come colpita in `celleColpite`.
- Salva il punto come `ultimoColpo`.
- Ritorna il punto da colpire.

---

## ◆ `registraEsitoAttacco(Point punto, boolean colpito)`

Serve per **informare il bot se il suo ultimo attacco è andato a segno**.

**Se ha colpito:**

- Se c'è almeno un altro bersaglio adiacente nella coda, calcola la **direzione dell'attacco**.
- Aggiunge alla coda `bersagliAdiacenti` tutte le celle non ancora colpite **vicine al punto colpito**.

**Se ha mancato:**

- Reset della direzione.

---

## ◆ `getAdiacentiNonColpite(Point p)`

Metodo di supporto:

- Ritorna una lista di celle **adiacenti non ancora colpite** al punto `p`.

- Usa le direzioni **sopra, sotto, sinistra, destra**.

---

## CLASSE NAVE

---

### **public class Nave**

Questa classe rappresenta una **nave nel gioco**. Ogni nave ha una lunghezza, può essere posizionata su una griglia e può ricevere colpi fino a essere affondata.

---

### **Attributi principali**

```
private int lunghezza;  
private int colpiSubiti = 0;  
private List<Point> posizioni = new ArrayList<>();
```

- **lunghezza**: la lunghezza della nave (es. 5 per la portaerei, 3 per il sottomarino, ecc.).
  - **colpiSubiti**: quanti colpi ha già ricevuto. Quando raggiunge **lunghezza**, la nave è affondata.
  - **posizioni**: lista delle coordinate (**Point**) che la nave occupa nella griglia.
- 

### **Costruttore**

```
public Nave(int lunghezza)
```

Crea una nuova nave con la lunghezza specificata.


---

## **boolean posiziona(int x, int y, boolean orizzontale, boolean[][] griglia)**

Questo metodo serve a **posizionare la nave** sulla griglia.

### **Cosa fa:**

1. **Svuota** la lista delle posizioni (`posizioni.clear()`).
2. Per ogni cella che dovrebbe occupare:
  - Calcola le coordinate in base all'orientamento (orizzontale o verticale).
  - Controlla che non vada fuori dalla griglia (oltre il bordo) o su una cella già occupata (`griglia[nx][ny]`).
  - Se tutto è valido, aggiunge il punto alla lista `posizioni`.
3. Se tutte le posizioni sono valide:
  - Segna ogni cella come occupata (`griglia[x][y] = true`).
  - Restituisce `true`.

 Se anche una sola posizione non è valida, la nave **non viene piazzata** e il metodo restituisce `false`.

---

## **void subisciColpo()**

Incrementa il contatore `colpiSubiti`. Viene chiamato ogni volta che la nave viene colpita.

---

## **boolean isAffondata()**

Controlla se la nave è affondata:

```
return colpiSubiti >= lunghezza;
```

Una nave è affondata quando ha subito tanti colpi quanto la sua lunghezza.

---

## Altri metodi

- `List<Point> getPosizioni()`: restituisce la lista delle coordinate occupate dalla nave.
  - `int getLunghezza()`: restituisce la lunghezza della nave.
-