

# GIOCO AUTO 2

Relazione progetto **Programmazione III e  
Laboratorio di Programmazione III**

Prof. Angelo Ciaramella

Emanuel Di Nardo

A cura di

Lorenzo Guerrini:0124002183

Pasquale Marzocchi:0124001891

# Descrizione Progetto

Il progetto prevede la realizzazione di un'applicazione che riguarda un gioco d'auto.  
È un progetto di tipo gioco che permette agli utenti di registrarsi  
e giocare le proprie partite ,  
il gioco consiste nel superare un percorso con ostacoli  
e la possibilità di raccogliere oggetti bonus.

# Per chi è sviluppata l'applicazione?

L'attore principale che avrà modo di interfacciarsi con l'applicazione sarà l'utente, dopo aver effettuato la registrazione, inserendo nome e cognome. Dopodiché potrà iniziare a interfacciarsi con il gioco.

# Cosa è stato usato?












- Il linguaggio di programmazione usato è Java
- Sono stati usati i Design Pattern, di importanza fondamentale per lo sviluppo dell'applicazione
- Per le componenti grafiche è stato usato JavaFx con editor Scene Builder
- File per memorizzare i dati utenti e le partite

# DESIGN PATTERN

I Design Pattern usati per la realizzazione sono:

- Singleton
- State
- Decorator
- Iterator
- Factory

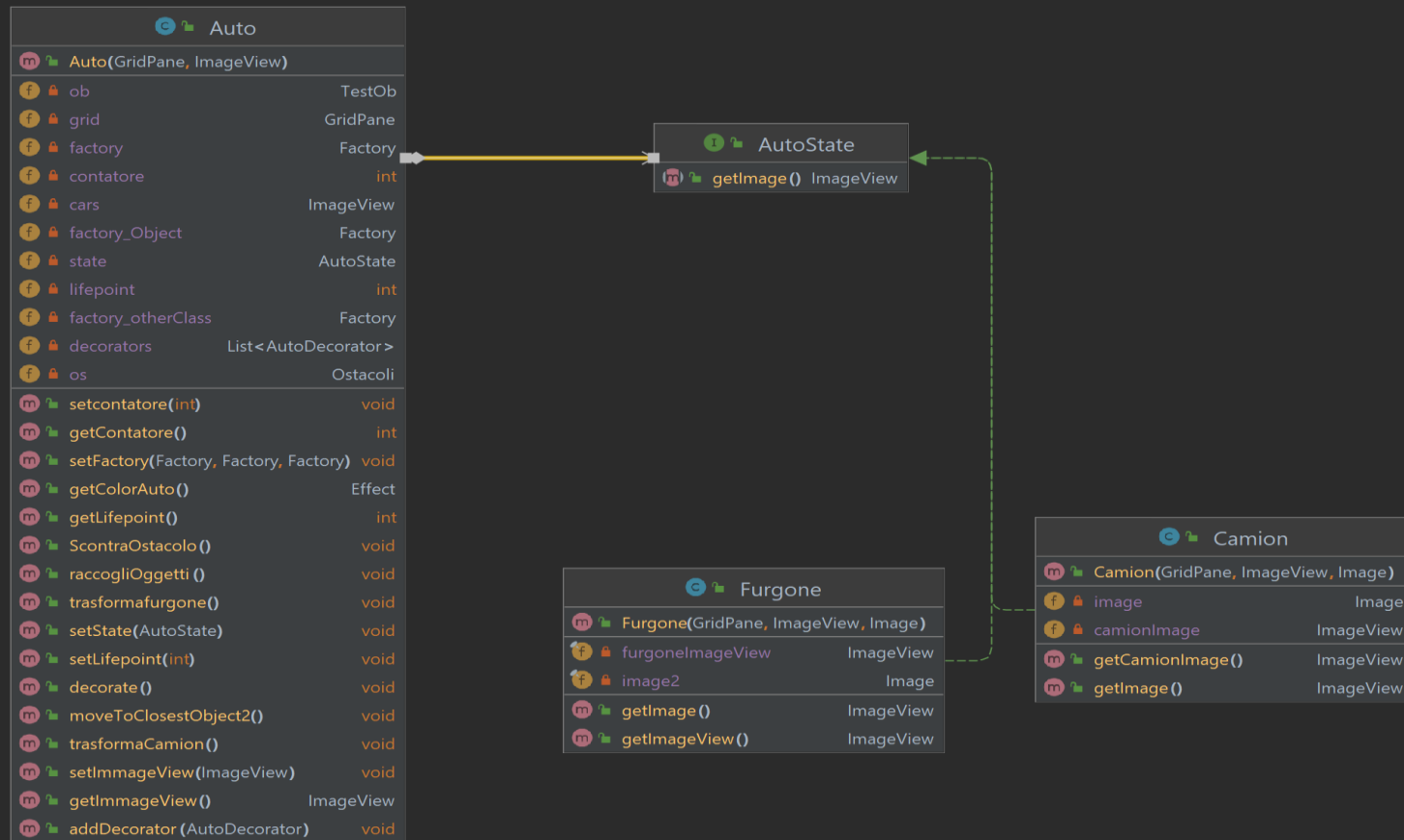
# SINGLETON

c  GameManager		
f 	isPaused	boolean
f  	instance	GameManager
m 	pause()	void
m  	getInstance()	GameManager
m 	isPaused()	boolean
m 	stop()	void
m 	setPaused(boolean)	void
m 	resume()	void

L'intento di Singleton è quello di assicurarsi che la classe abbia una sola istanza fornendo nel contempo un punto d'accesso globale.

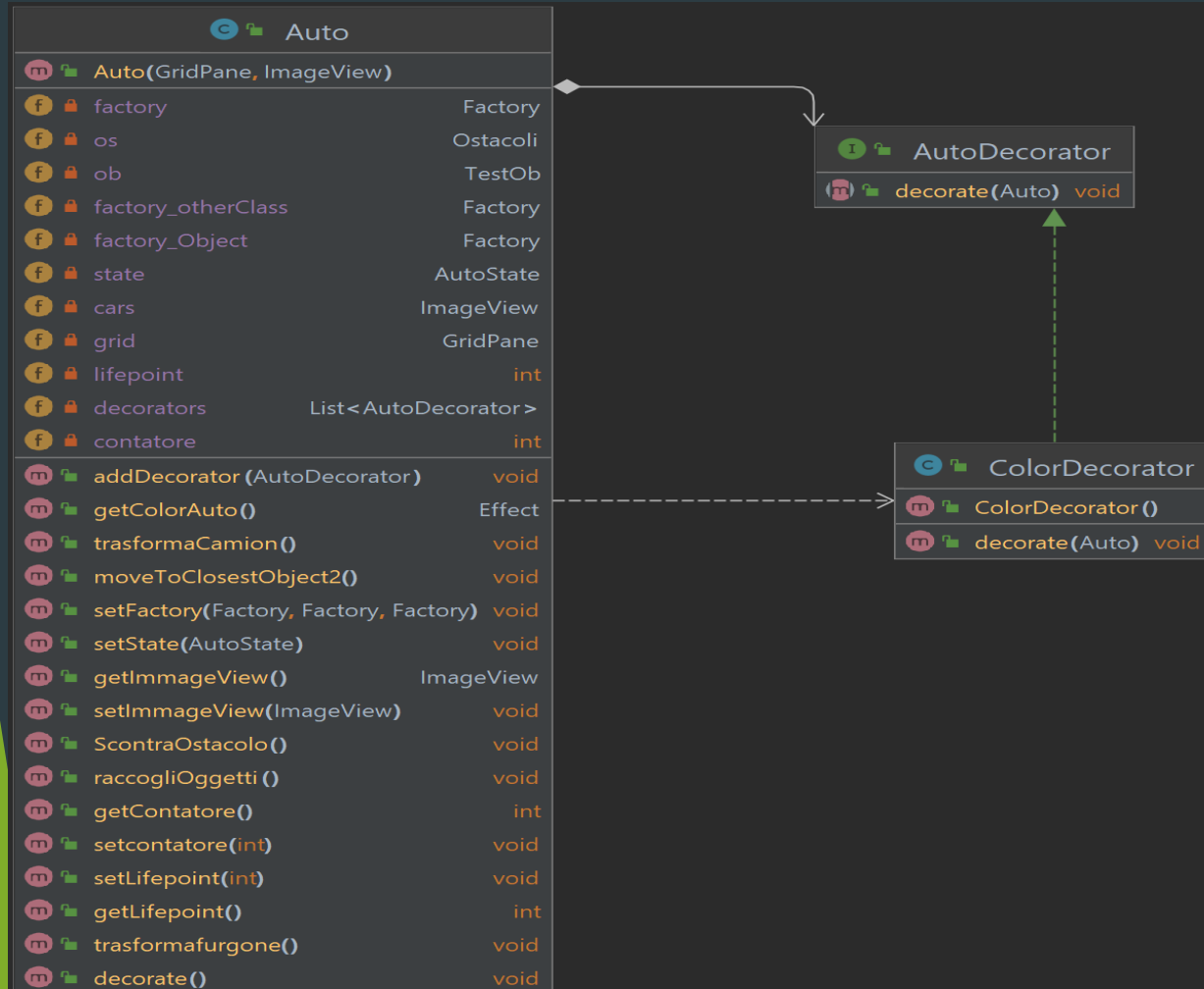
Nel nostro caso Singleton si occupa di gestire la pausa e la ripresa del sistema di gioco.

# STATE



Il pattern State viene utilizzato per gestire lo stato di un oggetto e consentire di cambiare il suo comportamento dinamicamente in base allo stato corrente. Nel nostro caso, stiamo utilizzando il pattern State nella classe Auto per gestire lo stato dell'auto e cambiare la sua rappresentazione visuale. In questo modo quando l'auto raccoglie un oggetto passa dallo stato base furgone e camion tramite State.

# DECORATOR



Il pattern Design Decorator viene utilizzato per modificare dinamicamente il comportamento di un oggetto senza doverlo modificare direttamente. Nel nostro caso, abbiamo utilizzato Decorator per cambiare il colore dell'immagine dell'auto nel metodo `scontraOstacoli` della classe `Auto`.

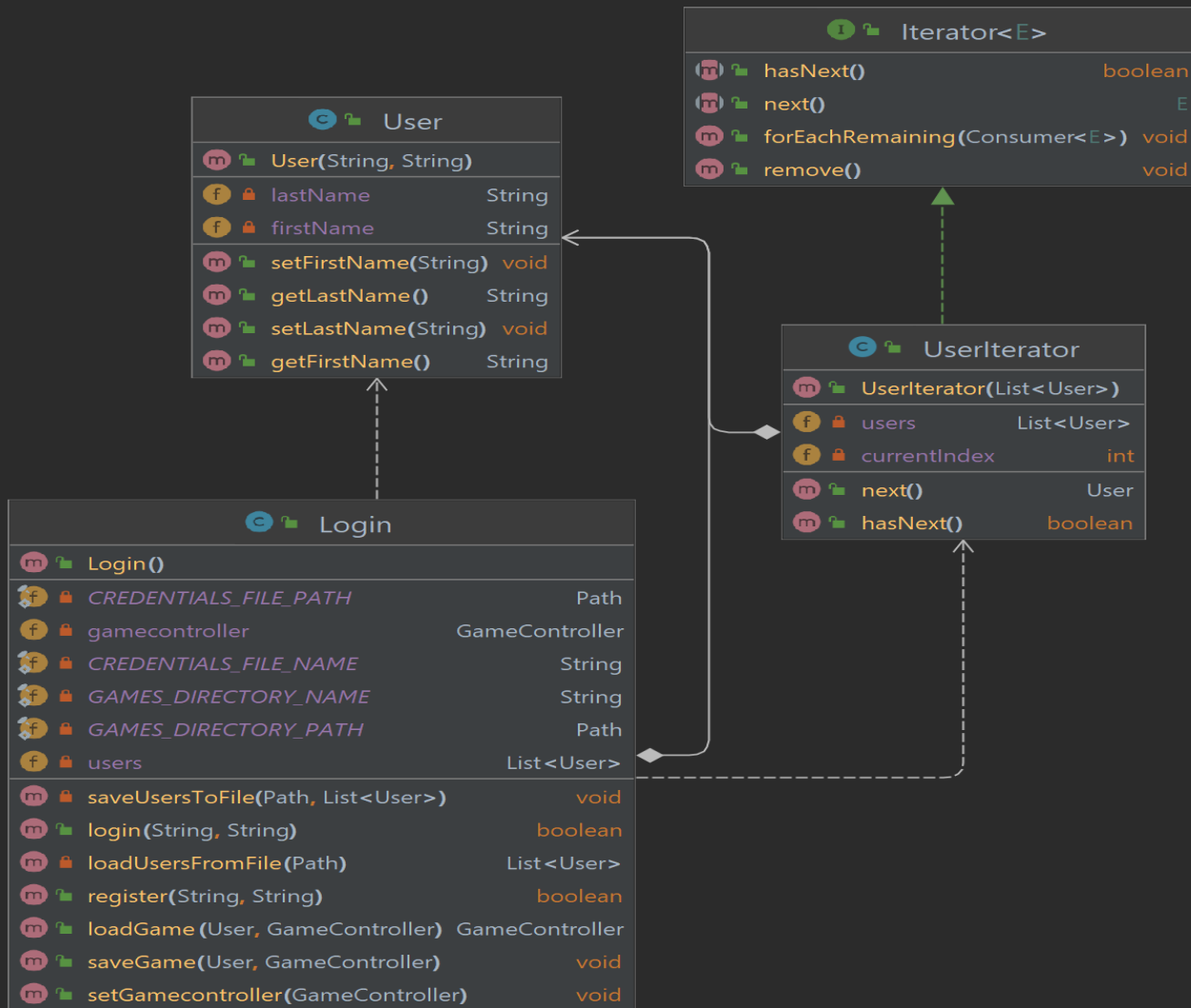
Abbiamo definito un'interfaccia chiamata `AutoDecorator` che ha un metodo `decorate` che prende un oggetto di tipo `Auto` come argomento e creato una classe chiamata `ColorDecorator` che implementa l'interfaccia `AutoDecorator`. All'interno del metodo `decorate`, viene implementato e gestito il codice per modificare il colore dell'auto. Nella classe `auto` con i metodi `addDecorator` e `decorate()` possiamo gestire il cambiamento del colore dell'auto quando si scontra con un ostacolo.



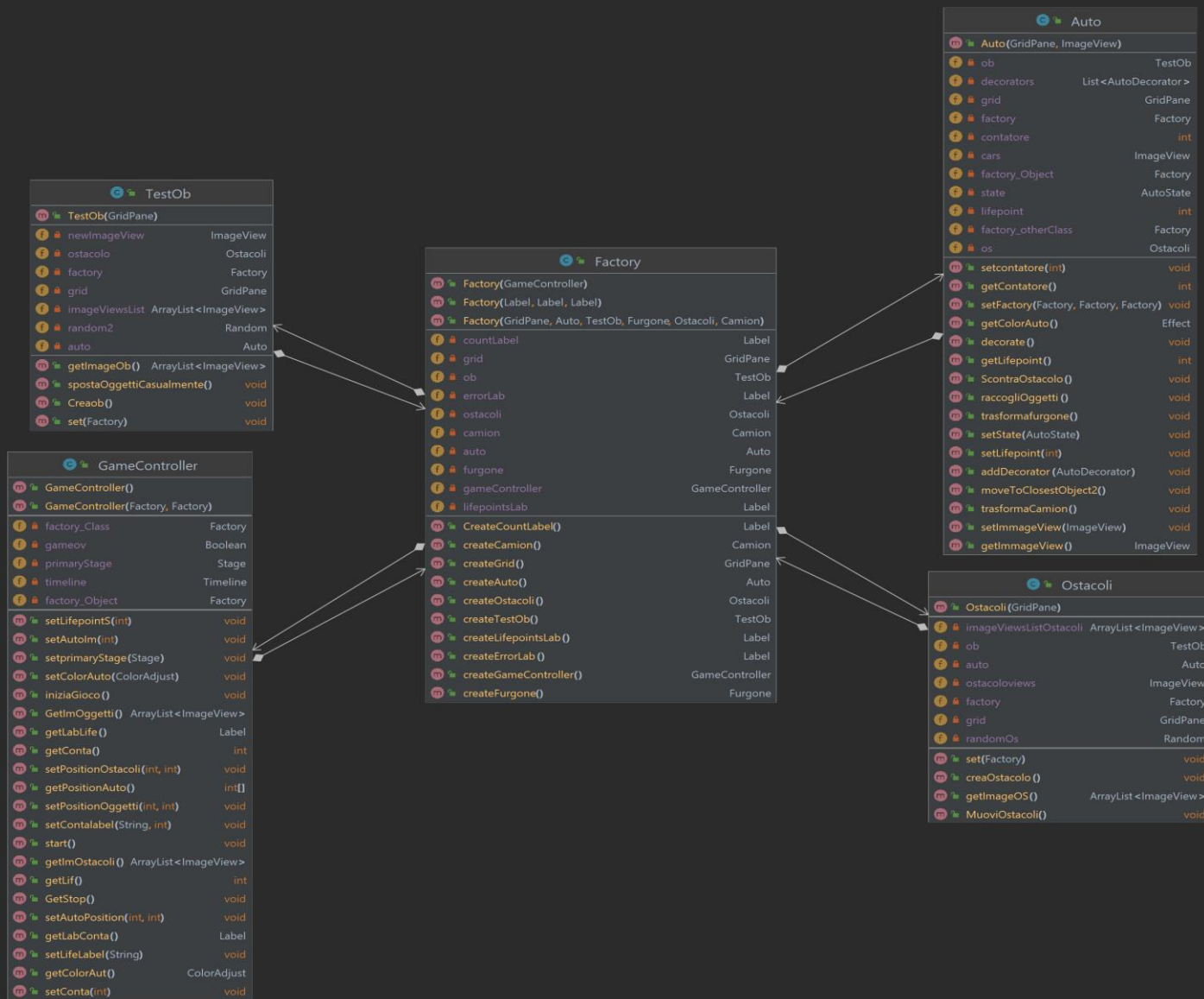
# ITERATOR

Il pattern Iterator è un design pattern comportamentale che consente di accedere agli elementi di una collezione senza rivelare la sua implementazione sottostante. In Java, il pattern Iterator è implementato mediante l'interfaccia `Iterator` e la sua implementazione personalizzata.

La classe `UserIterator` implementa l'interfaccia `Iterator<User>` e fornisce un'implementazione personalizzata per l'iterazione attraverso una lista di oggetti `User`.



# FACTORY



Utilizzando Factory, si possono creare oggetti senza preoccuparsi della logica specifica di creazione e della classe concreta degli oggetti. Factory si occupa di restituire l'oggetto appropriato in base al metodo di creazione chiamato.

Il pattern Factory ti permette di creare oggetti senza dover conoscere la classe concreta e di delegare la responsabilità di creazione degli oggetti a una classe Factory dedicata. Nel nostro codice fornisce metodi per creare diversi oggetti utilizzati nel gioco, semplificando così il processo di creazione degli oggetti.

# UML PROGETTO

# Come si presenta il gioco



Il gioco si presenta con un menu contenente 3 button

- **START**: permette di avviare la partita (soltanto dopo aver eseguito il login).
- **LOG IN**: permette all'utente di eseguire il login per iniziare il gioco e salvare i propri dati.
- **SIGN IN**: permette all'utente di eseguire la registrazione.

## SIGN IN

NOME

COGNOME

Sign in

Come si  
presentano le  
due finestre  
all'utente

## LOGIN

NOME

COGNOME

Log in

```
public boolean register(String firstName, String lastName) throws IOException {
    UserIterator iterator = new UserIterator(users);
    while (iterator.hasNext()) {
        User user = iterator.next();
        if (user.getFirstName().equals(firstName) && user.getLastName().equals(lastName)) {
            System.out.println("Utente già registrato!");
            return false;
        }
    }
    User newUser = new User(firstName, lastName);
    users.add(newUser);
    saveUsersToFile(CREDENTIALS_FILE_PATH, users);
    System.out.println("Registrazione effettuata con successo!");
    return true;
}
```

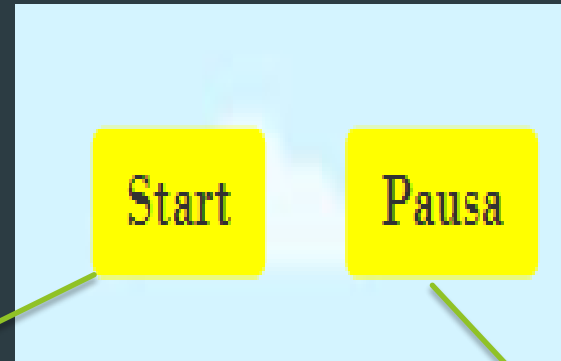
```
public boolean login(String firstName, String lastName) {
    UserIterator iterator = new UserIterator(users);
    while (iterator.hasNext()) {
        User user = iterator.next();
        if (user.getFirstName().equals(firstName) && user.getLastName().equals(lastName)) {
            System.out.println("Accesso effettuato con successo!");
            return true;
        }
    }
    System.out.println("Credenziali errate!");
    return false;
}
```

# Il gioco



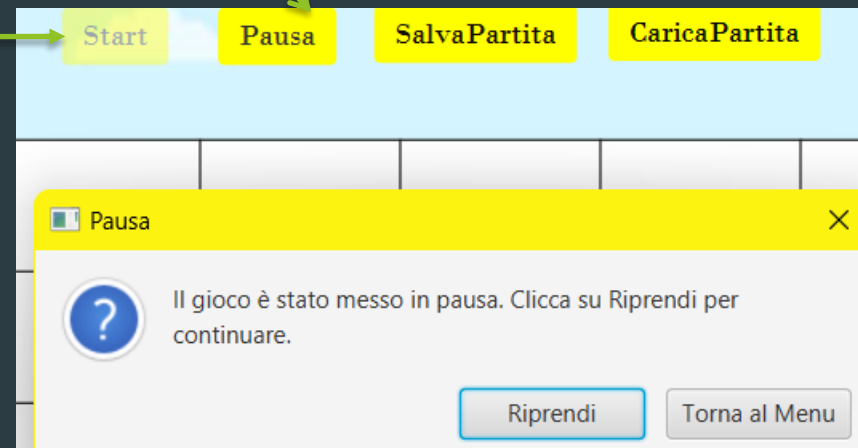
Il gioco offre all'utente 4 button per interagire. Inoltre mostra anche le vite disponibili e gli oggetti raccolti durante la corsa

# Start e Pausa



Pausa permetterà all'utente di riprendere a giocare o di tornare al menu principale

Start avvia la partita.  
NB: il button start viene disabilitato dopo essere stato premuto per evitare errori durante la partita.





# SalvaPartita e CaricaPartita

## GIOCO D'AUTO

SalvaPartita

CaricaPartita

**loadGame** caricherà la partita tramite i dati ricevuti da GAMES.DIRECTORY\_PATH, leggendoli tramite il BufferedReader

```
public GameController loadGame(User user, GameController game) throws IOException {
    String gameFileName = user.getFirstName() + "_" + user.getLastName() + ".txt";
    Path gameFilePath = Paths.get(GAMES_DIRECTORY_PATH.toString(), gameFileName);
    if (!Files.exists(gameFilePath)) {
        throw new FileNotFoundException("File not found: " + gameFilePath);
    }
    BufferedReader reader = new BufferedReader(new FileReader(gameFilePath.toFile()));
    String line;
```

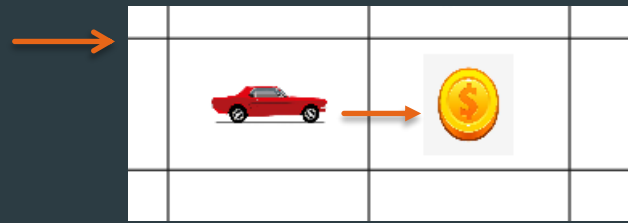
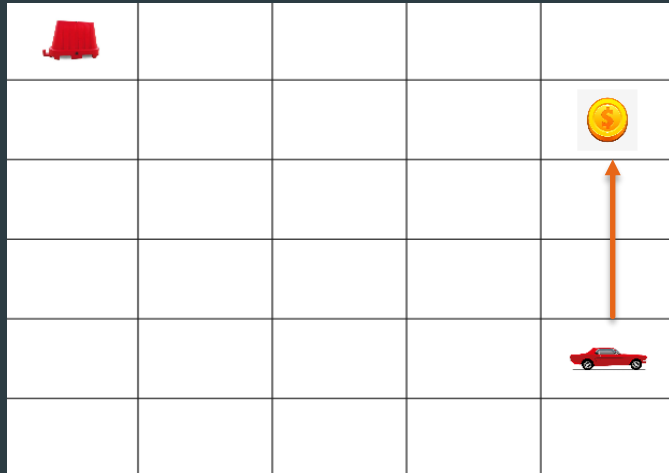
```
public void saveGame(User user, GameController game) throws IOException {
    String gameFileName = user.getFirstName() + "_" + user.getLastName() + ".txt";
    Path gameDirectoryPath = Files.createDirectories(GAMES_DIRECTORY_PATH);
    Path gameFilePath = gameDirectoryPath.resolve(gameFileName);
    BufferedWriter writer = new BufferedWriter(new FileWriter(gameFilePath.toFile()));

    // salva set stato auto, furgone, camion
    int stateA=game.getConta();
    int viteA=game.getLif();
    writer.write("CarLife:"+viteA);
    writer.newLine();
```

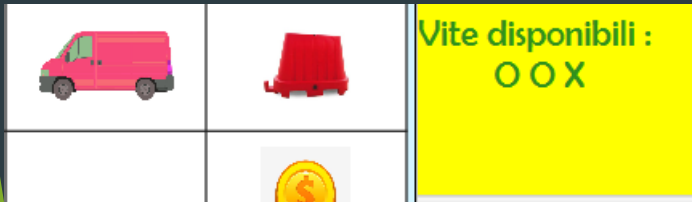
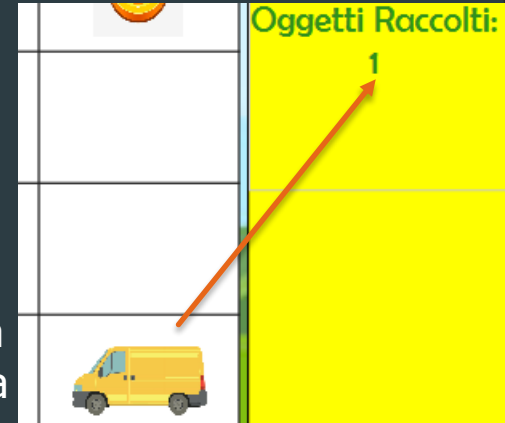
**saveGame** salva i dati della partita insieme al nome e cognome dell'utente ,creando il file del salvataggio  
GAMES\_DIRECTORY\_PATH



# Raccogli Oggetti e Scontra Ostacoli



L'auto si muoverà subito verso l'oggetto più vicino fino alla fine della partita, una volta raccolto cambierà la sua forma in furgone e camion.



Quando l'auto colpirà un ostacolo diminuirà di 1 le sue vite(3) e cambierà colore ogni volta.

## GAME OVER

Torna al menu

# GIOCO D'AUTO

Pausa

SalvaPartita

CaricaPartita

**FINE**

Vite disponibili :

Oggetti Raccolti :

