



# EasyWander



*Scooter-Rental*



# Il problema

**Negli ultimi anni è diventato sempre più popolare l'utilizzo di monopattini per spostarsi nei centri urbani.**

**Le persone vogliono potersi spostare comodamente ma senza aver bisogno di portare con sé il proprio mezzo, per tal motivo si è diffuso il noleggio di monopattini.**



# La soluzione

Per questo motivo è nata EasyWander, una PWA che permette il noleggio dei monopattini (convenzionati).

L'utente noleggerà un qualsiasi monopattino libero, il quale verrà visualizzato sulla mappa.



# Cos'è EasyWander?

EasyWander è un sistema pensato per essere usato da tre tipologie di utenti:

- **User:** è l'utente che usufruisce del servizio di noleggio.
- **Employee:** è l'utente che si occupa della gestione del sistema e del servizio.
- **Operator:** è l'utente che si occupa delle operazioni su campo.

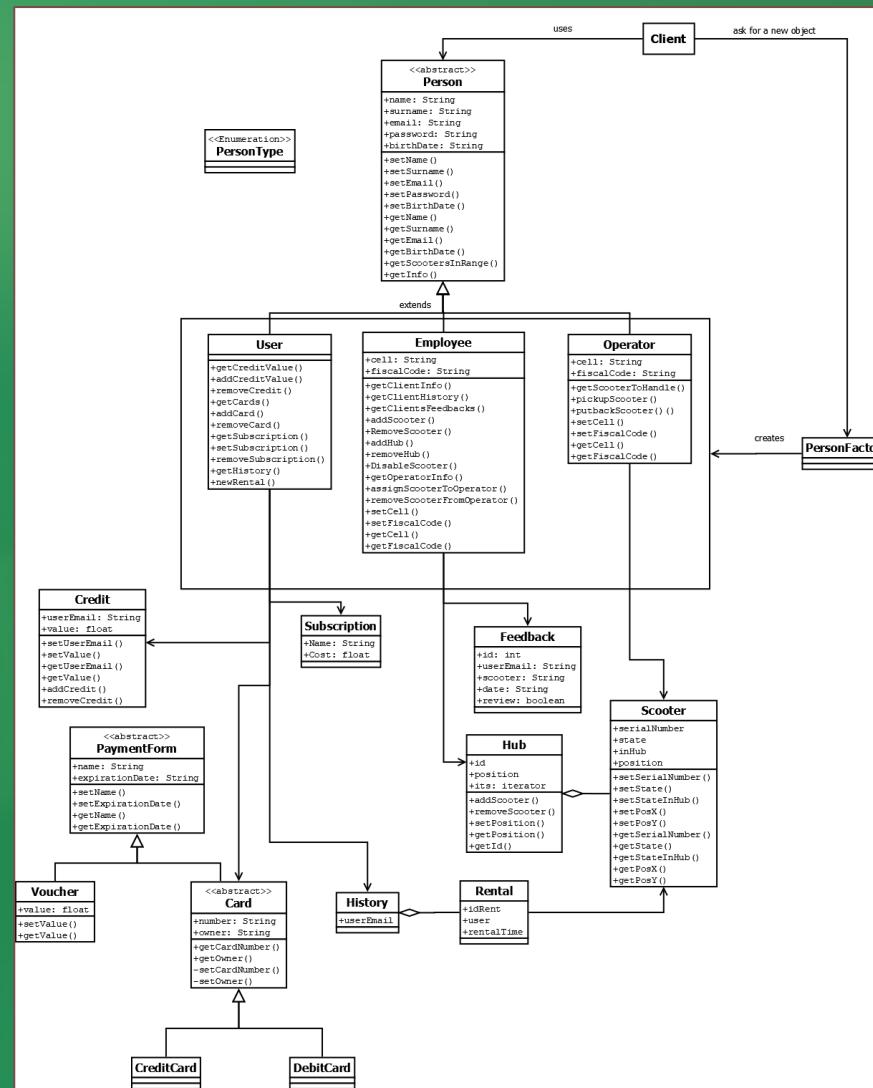
Per lo sviluppo di questo sistema sono stati utilizzati 5 pattern:

- **Factory Pattern;**
- **Strategy;**
- **Memento;**
- **Chain of Responsibility;**
- **Iterator**

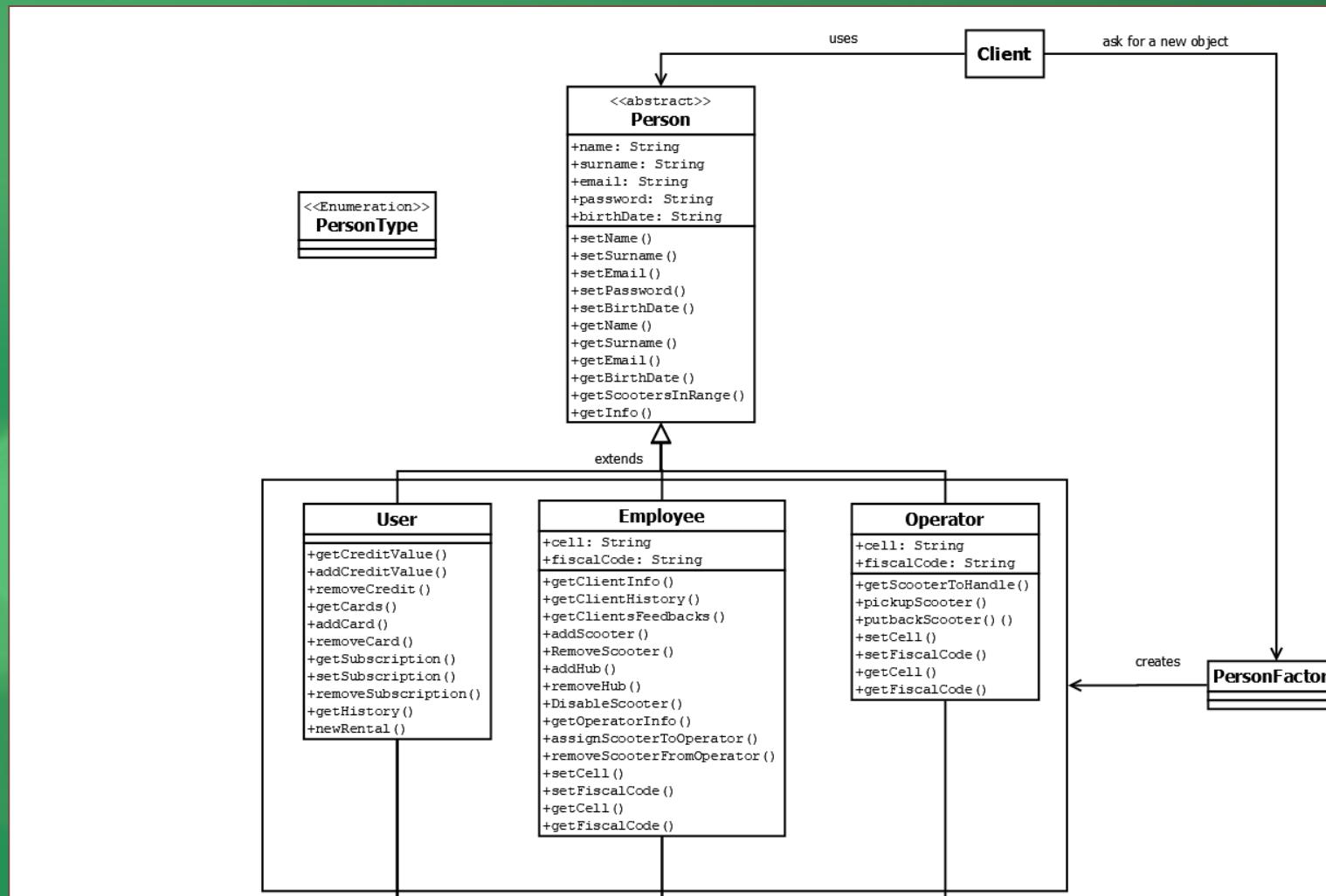
È stato fatto utilizzo inoltre di mySQL per la base di dati e SWING per l'interfaccia grafica.

# Factory Pattern

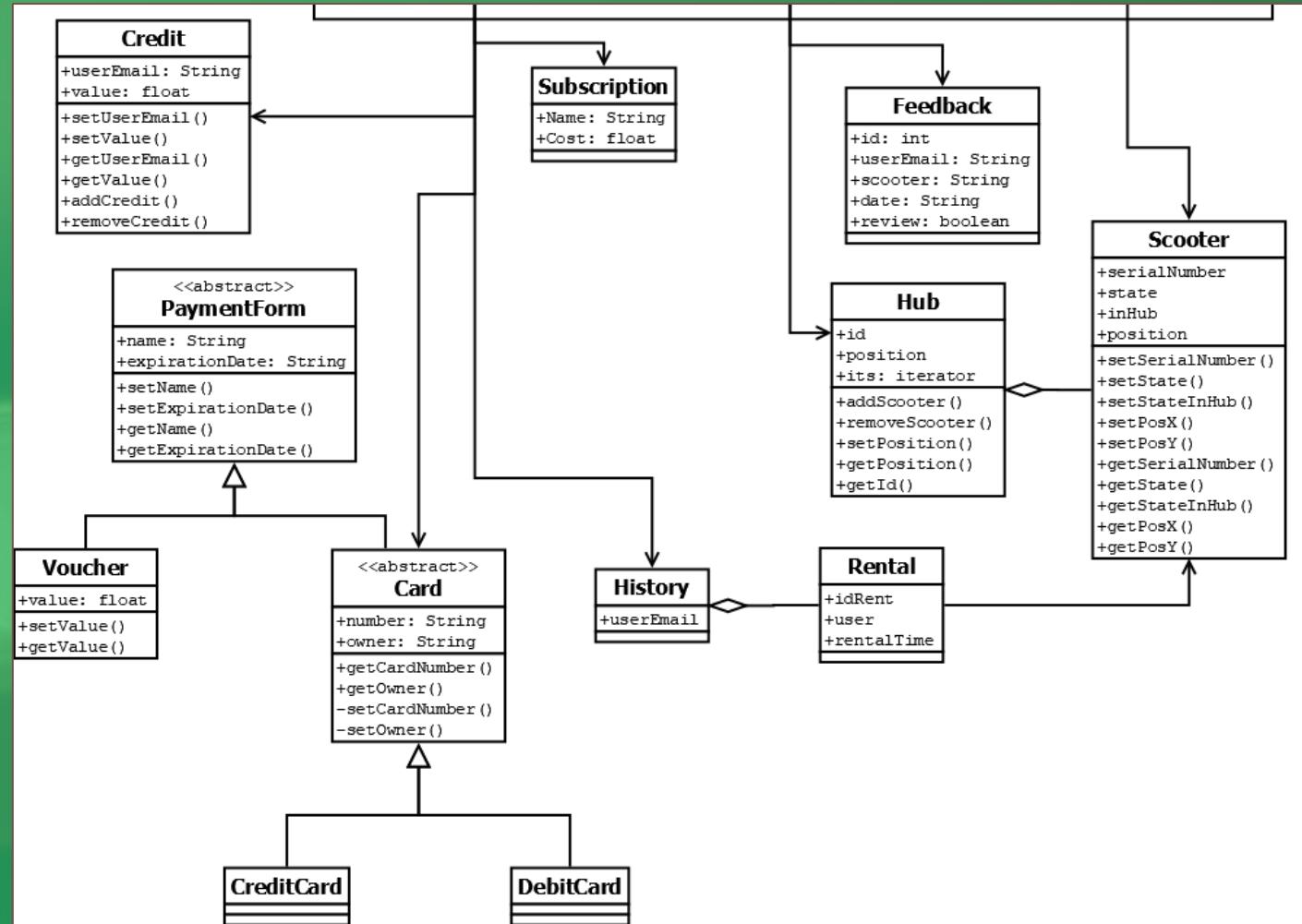
# Diagramma Factory Pattern (Intero)



# Diagramma Factory Pattern (zoom 1)



# Diagramma Factory Pattern (zoom 2)



**Il Factory Pattern ha lo scopo di creare oggetti senza esporre la logica di istanziazione al client. La creazione degli oggetti avviene attraverso un’interfaccia comune.**

**È stato usato in questo caso in quanto si voleva delegare la responsabilità della creazione di una “tipologia di utente” dal client alla classe factory.**

**Questo pattern ci permette di avere un codice più robusto, meno accoppiato e facile da estendere.**

**La superclasse può essere un’interfaccia o una classe astratta. In questo caso, la superclasse è rappresentata dalla classe astratta Person. Le sottoclassi sono invece User, Employee e Operator.**

**Una volta definiti superclassi e sottoclassi, definiamo la classe factory PersonFactory, la quale offre un metodo statico che ritorna la sottoclasse. Il tipo di sottoclasse restituita dipenderà dagli input inseriti.**

## PERSON

Person è una classe astratta che verrà ereditata da User, Employee e Operator.

Le variabili sono protette perché dovranno essere ereditate.

Contiene un metodo “getScootersInRange()” in quanto tutti possono cercare, per ragione diverse, uno scooter.

Rappresenta il “Product” nel Pattern: Factory Pattern.

## USER

Lo User è l’utente che usufruisce del servizio di noleggio.

Ogni utente ha un credito (che dovrà ricaricare attraverso una forma di pagamento), delle carte, al più un abbonamento e uno storico di noleggi.

Uno User può effettuare un noleggio attraverso il metodo “newRental()”.

## EMPLOYEE

**Un Employee è l'utente che si occupa della gestione del sistema e del servizio.**

**Si occupa di aggiungere, rimuovere o disattivare monopattini.**

**Si occupa anche di aggiungere o rimuovere account (infatti, ad eccezione dell'utente che può registrarsi autonomamente, sarà lui ad occuparsi di aggiungere nuovi employee o operator).**

**Può ottenere informazioni sull'utente, sul suo storico di noleggi, sui feedback degli utenti.**

**Può assegnare o levare uno scooter ad un Operator.**

## OPERATOR

**Un Operator è l'utente che si occupa delle operazioni su campo.**

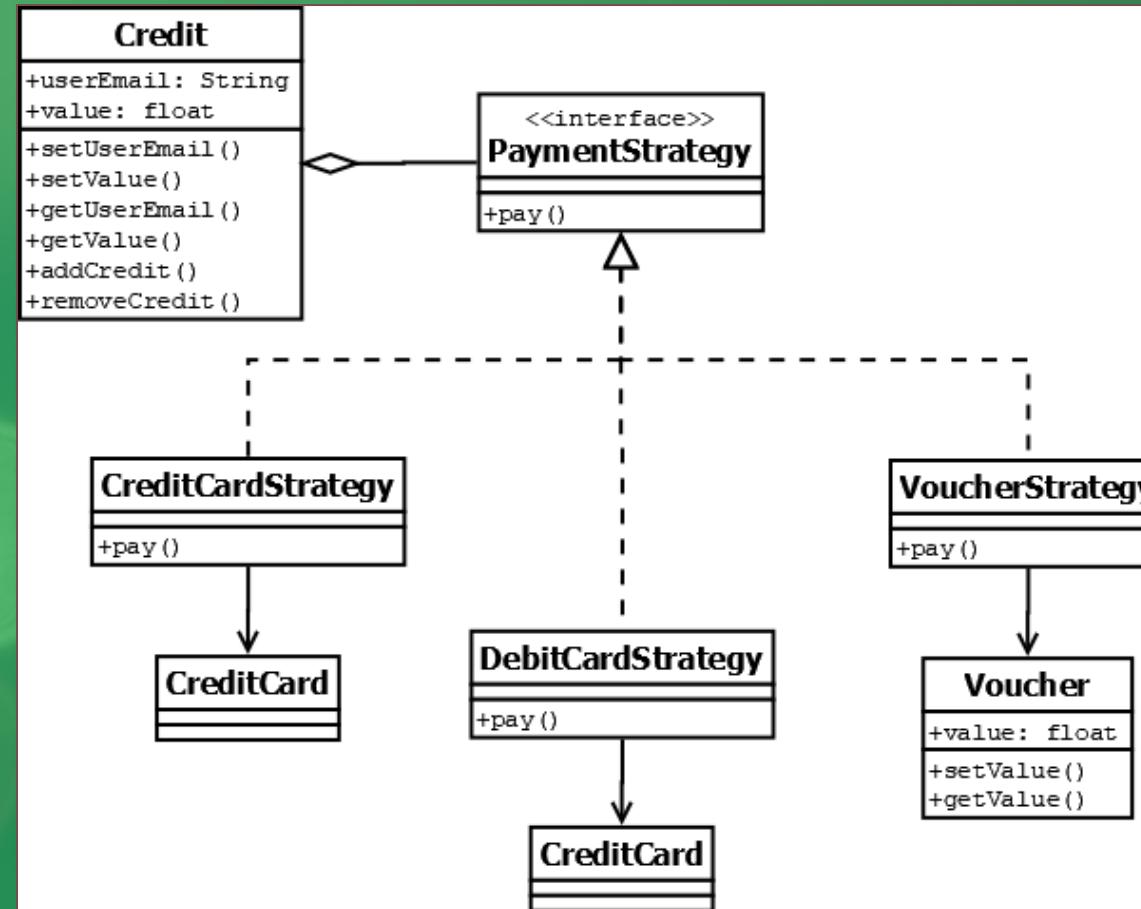
**Si tratta di colui che può prelevare o rimettere a posto gli scooter.**

## Codice di riferimento:

- **com/easywander/people:**
  - **Person.java**
  - **User.java**
  - **Employee.java**
  - **Operator.java**
- **com/easywander/factory:**
  - **PersonFactory.java**
  - **PersonType.java**

# Strategy

# Diagramma Strategy



**Lo Strategy ha lo scopo di definire una famiglia di algoritmi, incapsularli e renderli intercambiabili. L'algoritmo cambia indipendentemente dai client che lo usano.**

**È usato perché talvolta è necessario modificare dinamicamente gli algoritmi (o comunque le strategie) utilizzati da un'applicazione.**

**In questa applicazione, il noleggio e gli abbonamenti devono essere pagati attraverso un credito ricaricabile (per motivi di sicurezza), e tale credito può essere ricaricato attraverso 3 strategie di pagamento: carta di credito, carta di debito e voucher.**

## CREDIT

Credit rappresenta il credito ricaricabile che può essere speso per noleggiare monopattini o acquistare abbonamenti

## PAYMENTSTRATEGY

PaymentStrategy rappresenta l'interfaccia del pattern, in questo caso lo usiamo per definire un metodo “pay()” attraverso cui passiamo l'importo da ricaricare.

## CREDITCARD/DEBITCARD/VOUCHER STRATEGY

Implementiamo 3 diverse strategie per il pagamento, tutte molto simili fra di loro. Esse definiscono al loro interno un oggetto CreditCard, DebitCard o Voucher in base alla strategia utilizzata, il quale sarà passato come argomento attraverso il costruttore. Tutte e 3 le strategie faranno poi l'Override del metodo “pay()” implementato da PaymentStrategy.

## **PAYMENT FORM**

PaymentForm è una classe astratta che definisce il nome e la data di scadenza delle forme di pagamento. Notare come anche i metodi setter siano privati in quanto una volta inseriti i dati di una forma di pagamento valida, questi non possono essere modificati (sarà ovviamente possibile rimuovere successivamente la forma di pagamento).

Ha due sottoclassi: Voucher e Card.

## **VOUCHER**

Voucher è una classe che rappresenta i buoni. Ogni buono ha un certo valore.

## CARD

Card è una classe astratta che verrà usata come base per tipologie di carte più specifiche.

Definisce inoltre il numero e l'intestatario della carta.

Ha due sottoclassi: CreditCard e DebitCard.

## CREDITCARD e DEBITCARD

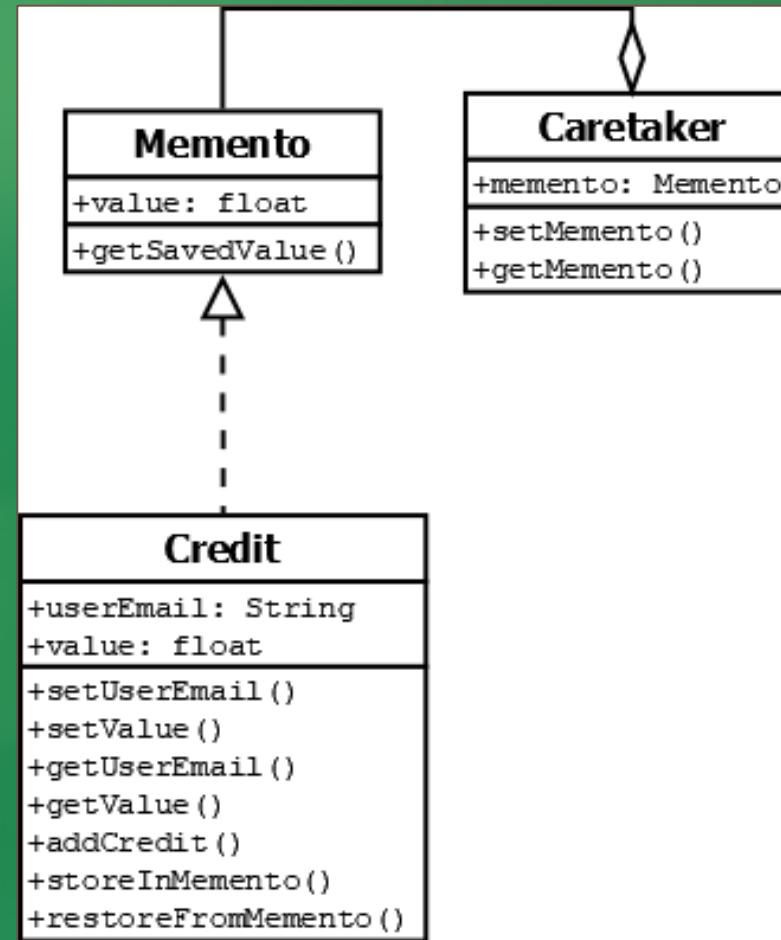
Rappresentano rispettivamente le carte di credito e le carte di debito.

## Codice di riferimento:

- **com/easywander/economy:**
  - **Card.java**
  - **Credit.java**
  - **CreditCard.java**
  - **DebitCard.java**
  - **PaymentForm.java**
  - **Voucher.java**
- **com/easywander/paymentstrategy:**
  - **CreditCardStrategy.java**
  - **DebitCardStrategy.java**
  - **VoucherStrategy.java**
  - **PaymentStrategy.java**

# Memento

# Diagramma Memento



**Il Memento ha lo scopo di catturare lo stato interno di un oggetto senza violare l'incapsulamento e fornendo così un mezzo per ripristinare l'oggetto allo stato iniziale quando necessario.**

**È utilizzato quando si vuole ripristinare una situazione precedente in caso di errore, come nell'undo. Nel nostro caso, lo useremo per ripristinare lo stato precedente del credito.**

**Possiamo distinguere tre componenti:**

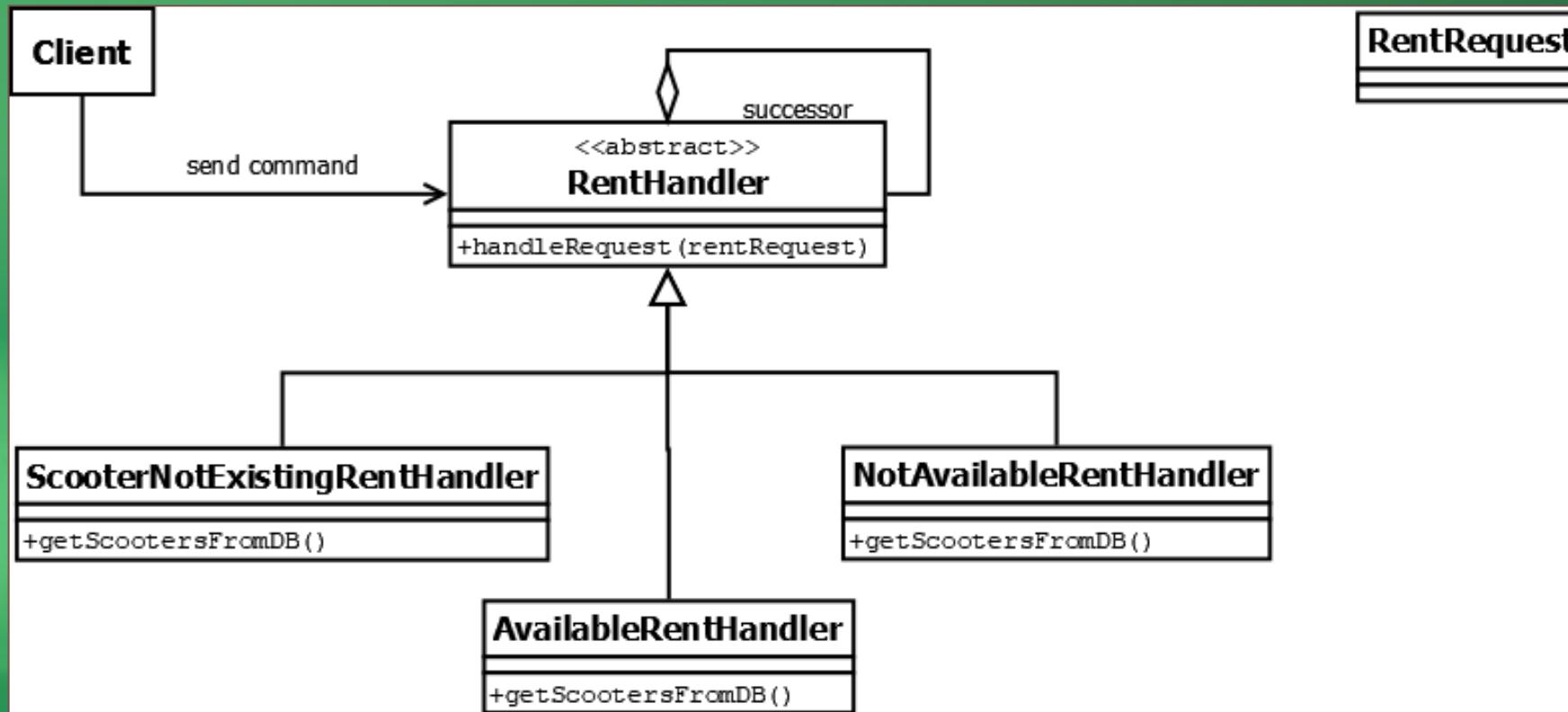
- **MEMENTO**: l'oggetto base conservato in stati diversi;
- **ORIGINATOR (Credit)**: imposterà ed otterrà i valori (set e get) dal memento correntemente selezionato. Verrà inoltre usato per creare nuovi memento e per assegnare loro;
- **CARETAKER**: mantiene un elemento (o lista di elementi) che conterrà tutte le versioni precedenti del Memento. Può conservare e recuperare Memento.

## Codice di riferimento:

- **com/easywander/economy:**
  - **Credit.java**
- **com/easywander/creditmemento:**
  - **Caretaker.java**
  - **Memento.java**

# Chain of Responsibility

# Diagramma CoR



**La Chain of Responsibility ha lo scopo di consentire di separare il mittente di una richiesta da un destinatario, in modo da consentire al più ad un oggetto di gestire la richiesta.**

**Gli oggetti destinatari vengono messi in “catena” e la richiesta viene trasmessa fino a trovare un oggetto che la gestisca.**

**La CoR permette ad un oggetto di inviare un comando senza conoscere quale oggetto la riceverà e la gestirà. La responsabilità viaggia per la catena finché non trova qualcuno che potrà gestirla.**

**Nel nostro caso, usiamo tale pattern per gestire le richieste di noleggio di monopattino.**

RentHandler rappresenta la classe astratta dalla quale andremo a definire le sottoclassi per gestire i vari casi, nello specifico distinguiamo:

- ScooterNotExistingRentHandler: gestisce la situazione in cui lo scooter richiesto non è presente all'interno del DB;
- AvailableRentHandler: gestisce la situazione in cui lo scooter è disponibile;
- NotAvailableRentHanlder: gestisce la situazione in cui lo scooter NON è disponibile.

La richiesta sarà rappresentata da un oggetto RentRequest, al quale dovremo passare il seriale del monopattino che vogliamo noleggiare (presente sul monopattino stesso).

## Codice di riferimento:

- **com/easywander/handlerental:**
  - **RentHandler.java**
  - **RentRequest.java**
  - **ScooterNotExistingRentHandler.java**
  - **AvailableRentHandler.java**
  - **NotAvailableRentHandler.java**

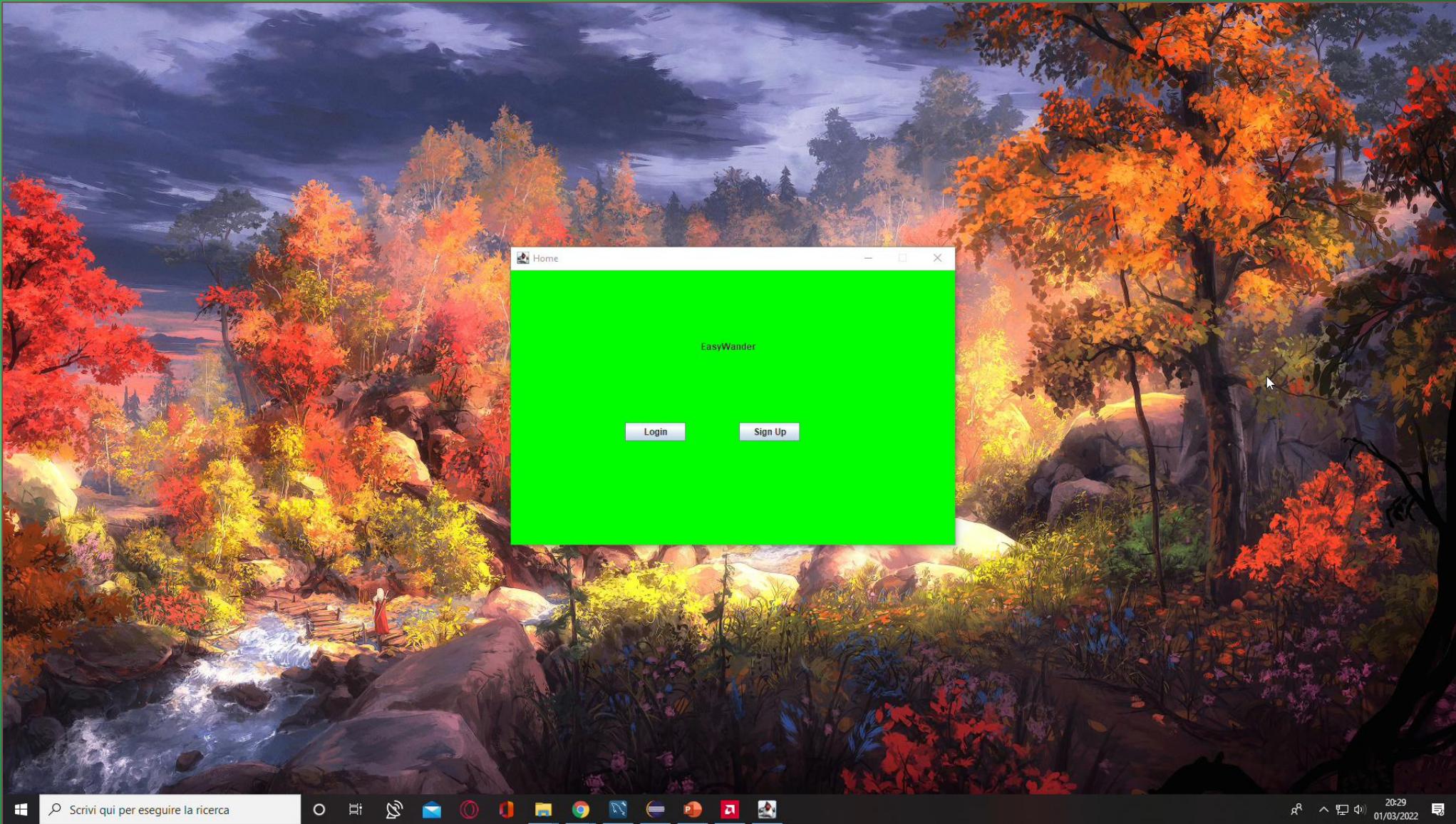
# Iterator

**L'iterator ha lo scopo di fornire un modo per accedere agli elementi di un oggetto aggregato nascondendo i dettagli di implementazione.**

**Iterator è un pattern già presente in Java, per tale motivo abbiamo preferito usare quello più che ricreare il pattern da zero.**

**È stato usato, ad esempio, per iterare sulle liste di scooter.**

# Video Dimostrativo



**Grazie per la vostra attenzione.**

- Fiorentino Michele 0124002085