Analisi di mercato

Confronto con i competitors rispetto a soluzioni innovative

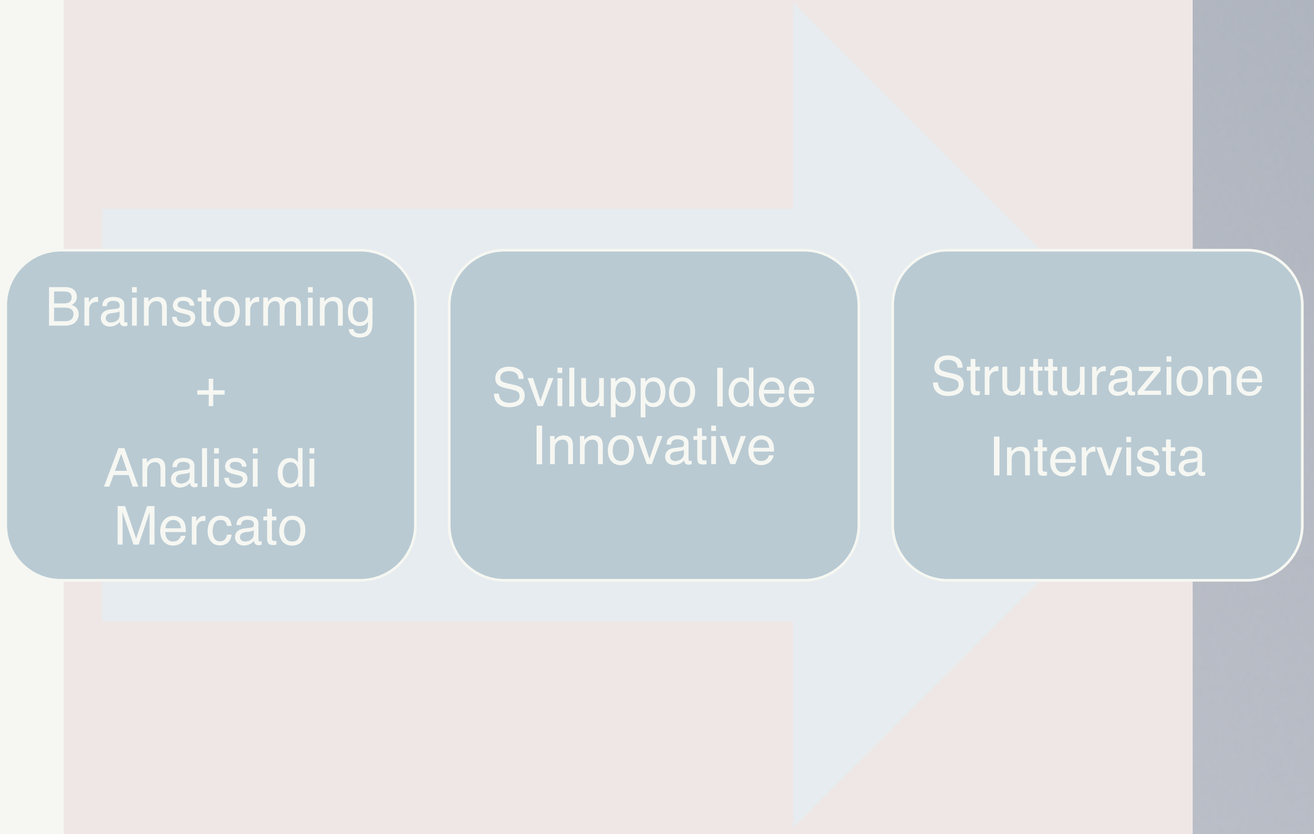
- Ragionamento sui valori chi si vogliono perseguire
- Analisi dei competitors rispetto ai suddetti valori
- Considerazioni finali sull'appetibilità degli stessi

	Packaging	Sost. Ambientale	Sost. Sociale	Economia Circolare	Linea Green	Test Di Invio Capi	Realtà Aumentata	Consigli Human Driven	Consigli Data Driven	Trasparenza Finanziaria	Spot Stores Temporanei
YOOX	✓	×	×	×	✓	×	×	×	✓	×	×
Net a Porter	✓	×	×	×	✓	×	×	×	✓	×	×
Patagonia	✓	✓	✓	✓	✓	×	×	✓	✓	×	×
SSENSE	✓	×	×	×	✓	×	×	✓	×	×	×
Garment Workshop	✓	×	×	×	✓	✓	×	×	×	✓	✓

Comparazione con i competitors

Intervista

Struttura Intervista



- **Chi:** per inquadrare meglio la figura del committente.
- **Che cosa:** per comprendere nel dettaglio le esigenze del committente e la sua idea di applicazione finale.
- **Quando:** per definire con precisione le tempistiche relative alla finalizzazione del progetto. Il tempo a disposizione determina una selezione dei moduli innovativi da sviluppare.

Risultati

Moduli Core

- Pop-up stores
- Magazzino snello
- Vestirsi con la realtà aumentata
- Go circular!

Moduli Pilota

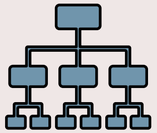
- Algoritmo di consigli all'acquisto huma-driven
- Sartoria dedicata e confezionamento su misura
- Social marketing

Attori, obiettivi, casi d’uso e requisiti

Tassonomia degli attori

Principali: coloro che utilizzano direttamente l'applicazione


Secondari: sistemi esterni che si integrano all'interno dell'applicazione

Fuori scena: entità che traggono benefici o costituiscono dei vincoli


Casi d'uso (UC)

- **Acquisto di capi di vestiario online** (UC1)
- **Accesso al magazzino snello** (UC2)
- **Aggiornamento dei clienti sui nuovi Pop-Up Stores** (UC3)

Requisiti

- **Funzionali:** cio' che il sistema compie
- **Non funzionali:** come e' fatto il sistema

UC1 - Acquisto di capi online

- **A. Principali:** Cliente (acquisto) - Dipendente (monitoraggio)
- **A. Secondari:** Metodi di pagamento
- **A. Fuori Scena:** Compagnia di trasporti
- **R. Funzionali:** Accesso al sito e all'applicazione, visualizzazione del catalogo, aggiunta al carrello, completamento e visualizzazione dell'ordine
- **R. Non funzionali:** Java (Strategy), HTML+CSS, DB Oracle

UC2 - Accesso al magazzino snello

- **A. Principali:** Cliente (avatar) - Dipendente sarto (monitoraggio)
- **A. Secondari:** Algoritmo per gestione magazzino (dipendenti)
- **A. Fuori Scena:** Fornitori
- **R. Funzionali:** Gestione dell'avatar per provare i capi; il sarto sfrutterà i dati per gestione lean del magazzino
- **R. Non funzionali:** Java (Composite), Three-JS (AR), webGL (3D-Graphics)

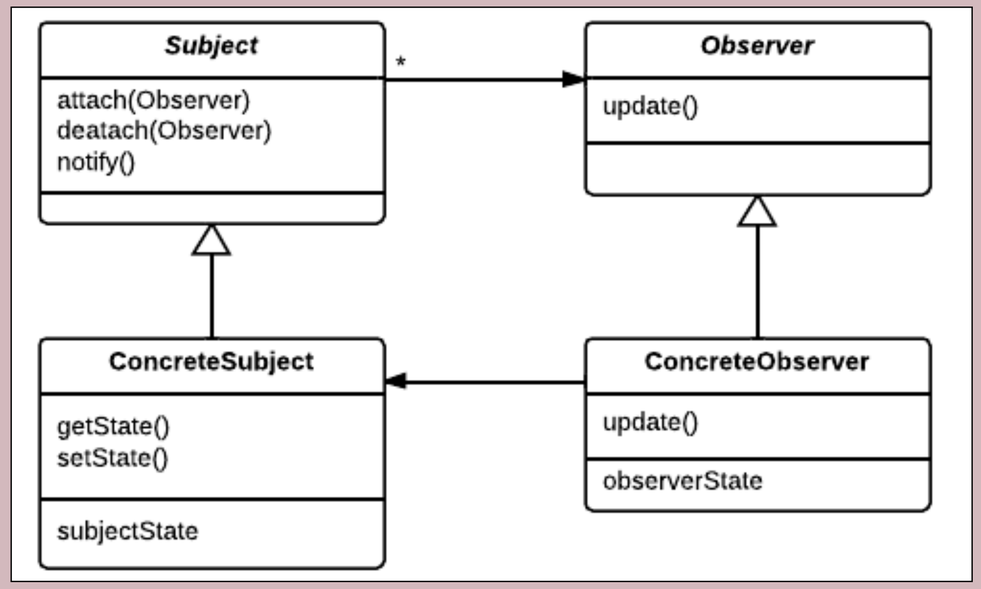
UC3 - Aggiornamento utenti per Pop-Up Stores

- **A. Principali:** Cliente (newsletter)
- **A. Secondari:** DB+API per gestione contatti
- **A. Fuori Scena:** GDPR
- **R. Funzionali: gli iscritti alla newsletter riceveranno aggiornamenti sui Pop-Up Stores**
- **R. Non funzionali:** Java (Observer), client mail, DB Oracle, API di geolocalizzazione

Pattern Observer

Definisce una relazione uno-a-molti tra oggetti, in modo tale che, quando un oggetto cambia stato, tutti i suoi dipendenti siano notificati e aggiornati automaticamente

Gamma, E. et Al. - Design Patterns: Elements of Reusable Object-Oriented Software (p.326)



Consta di:

- **Subject (1):** interfaccia per aggiungere, rimuovere, notificare osservatori
- **ConcreteSubject (1):** modifica il proprio stato e notifica del cambiamento gli osservatori
- **Observer (1...n):** interfaccia per aggiornare osservatori circa i cambi di stato
- **ConcreteObserver (1...n):** viene informato dei cambi di stato e aggiorna le info a propria disposizione

Le interfacce

Insieme di nomi di metodi che verranno implementati (in modo potenzialmente differente) su più classi

L'adozione delle interfacce e' legata ad una questione di **flessibilita'**: l'implementazione all'interno delle classi può essere fatta in più modi, senza modificare l'architettura complessiva del sistema. **Principio open-closed**: gli oggetti devono essere aperti ad implementazioni ma chiusi a modifiche

Vengono qui preferite rispetto alle *abstract classes* perche' le ultime costringerebbero ad usare le implementazioni definite nella classe stessa. Torna la **flessibilita'**

```
// The Subject interface
interface Subject {
  void registerObserver(Observer o);
void removeObserver(Observer o);
  void notifyObservers();
}
// The Observer interface
interface Observer {
void update(String location, String store, Date openingdate,
  Date closingdate, double openingtime, double closingtime);
}


interface DisplayElement {
void display();
}
```

Le classi concrete - Subject

Mantiene una lista dei dipendenti (***observers*** - Array di Observers) e li notifica tramite il seguente flusso:

- ***setMeasurements(...)*** aggiorna gli attributi e quindi chiama measurementsChanged()

- ***measurementsChanged()*** chiama notifyObservers()

- ***notifyObservers()*** esegue un loop per l'array di Osservatori (*observers*), aggiornando i dati a loro disposizione e notificandoli

```
// The PopUpStore class is the Subject
class PopUpStore implements Subject {
// attributes go here
public PopUpStore() {
observers = new ArrayList<Observer>();
}
public void registerObserver(Observer o) {
   observers.add(o);
}

public void removeObserver(Observer o) {
   int i = observers.indexOf(o);
if (i >= 0) {
      observers.remove(i);
} }

public void notifyObservers() {
   for (Observer observer : observers) {
observer.update(location, store, openingdate,
closingdate, openingtime, closingtime);
} }

public void measurementsChanged() {
notifyObservers();
}
public void setMeasurements(String location, String store, Date
openingdate, Date closingdate, double openingtime, double
closingtime) {
this.location = location;
this.store = store;
this.openingdate = openingdate;
this.closingdate = closingdate;
this.openingtime = openingtime;
this.closingtime = closingtime;
measurementsChanged();
   }
}
```

Le classi concrete - Observer

Rappresenta un oggetto interessato allo stato del ***Subject***, ricevendo aggiornamenti a modifiche dello stesso.

Ha, in particolare:

- un ***costruttore***

- ***update(...)***, il metodo che viene chiamata quando il Subject (*PopUpStore*) seguito dall'osservatore si aggiorna, aggiorna le info a disposizione di quest'ultimo e lo notifica chiamando display()

- ***display()***, il metodo che mostra le informazioni

```
// The Observer class
class CurrentConditionsDisplay implements Observer, DisplayElement {
// some attributes here
private Subject PopUpStore;
SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/dd");
String formattedOpeningDate = sdf.format(openingdate);
String formattedClosingDate = sdf.format(closingdate);
public CurrentConditionsDisplay(Subject PopUpStore) {
  this.PopUpStore = PopUpStore;
  PopUpStore.registerObserver(this);
}

public void update(String location, String store, Date
openingdate, Date closingdate, double openingtime, double
closingtime) {
  this.location = location;
this.store = store;
this.openingdate = openingdate;
this.closingdate = closingdate;
this.openingtime = openingtime;
this.closingtime = closingtime;
display();
}

public void display() {
System.out.println("We look forward to seeing you at our new Pop
-Up store in " + location + ", kindly hosted by  " + store +
", on the following dates: " + formattedOpeningDate + " - " +
  formattedClosingDate + ", at the following times:" + openingtime
   + " - " + closingtime);
  }
}
```

Il Main

- Viene creata un'istanza del PopUpStore e del SimpleDateFormat (stringa-data-stringa)

- Crea un osservatore (CurrectConditionDisplay) che monitori un PopUpStore

- Aggiorna gli stati del Subject con senMeasurement, che poi chiamerà' gli altri metodi per notificare l'osservatore

E' stato aggiunto un blocco try-catch nel caso in cui le stringhe di input non fossero nel formato corretto

```
// The main method
public class Newsletter {
  public static void main(String[] args) {
    PopUpStore PopUpStore = new PopUpStore();
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/dd");
    try {
      Date openingDate = sdf.parse("2023/01/12");
Date closingDate = sdf.parse("2023/01/19");
    CurrentConditionsDisplay currentDisplay = new
CurrentConditionsDisplay(PopUpStore);
    PopUpStore.setMeasurements("Turin", "Hannibal",
}}
openingDate, closingDate, 9.00, 19.00);
  } catch (ParseException e) {
  e.printStackTrace();
```

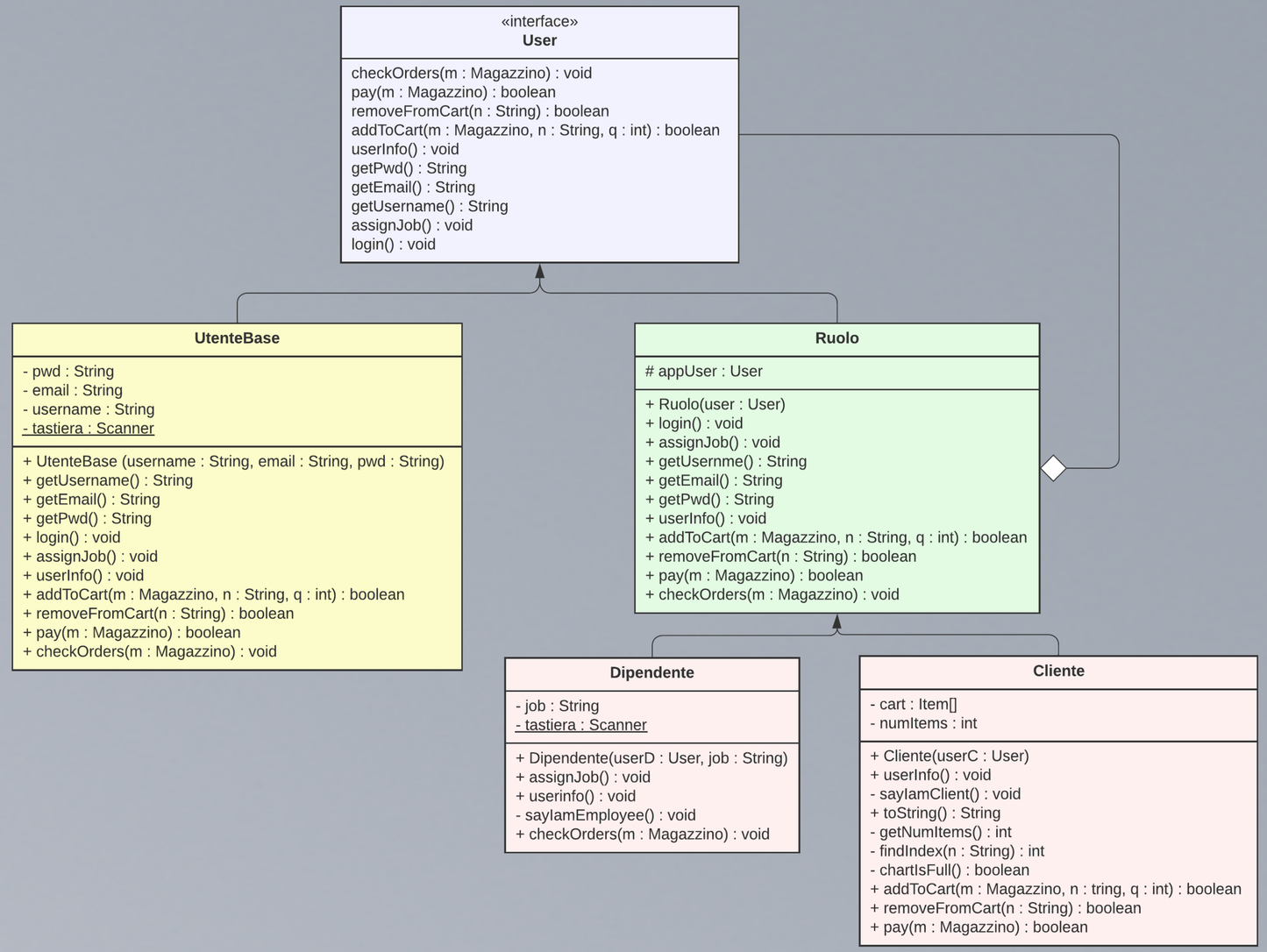
Il Pattern Decorator

Vantaggi:

- È più flessibile della ereditarietà statica (multipla)
- Approccio “pay as you go” aggiungendo responsabilità

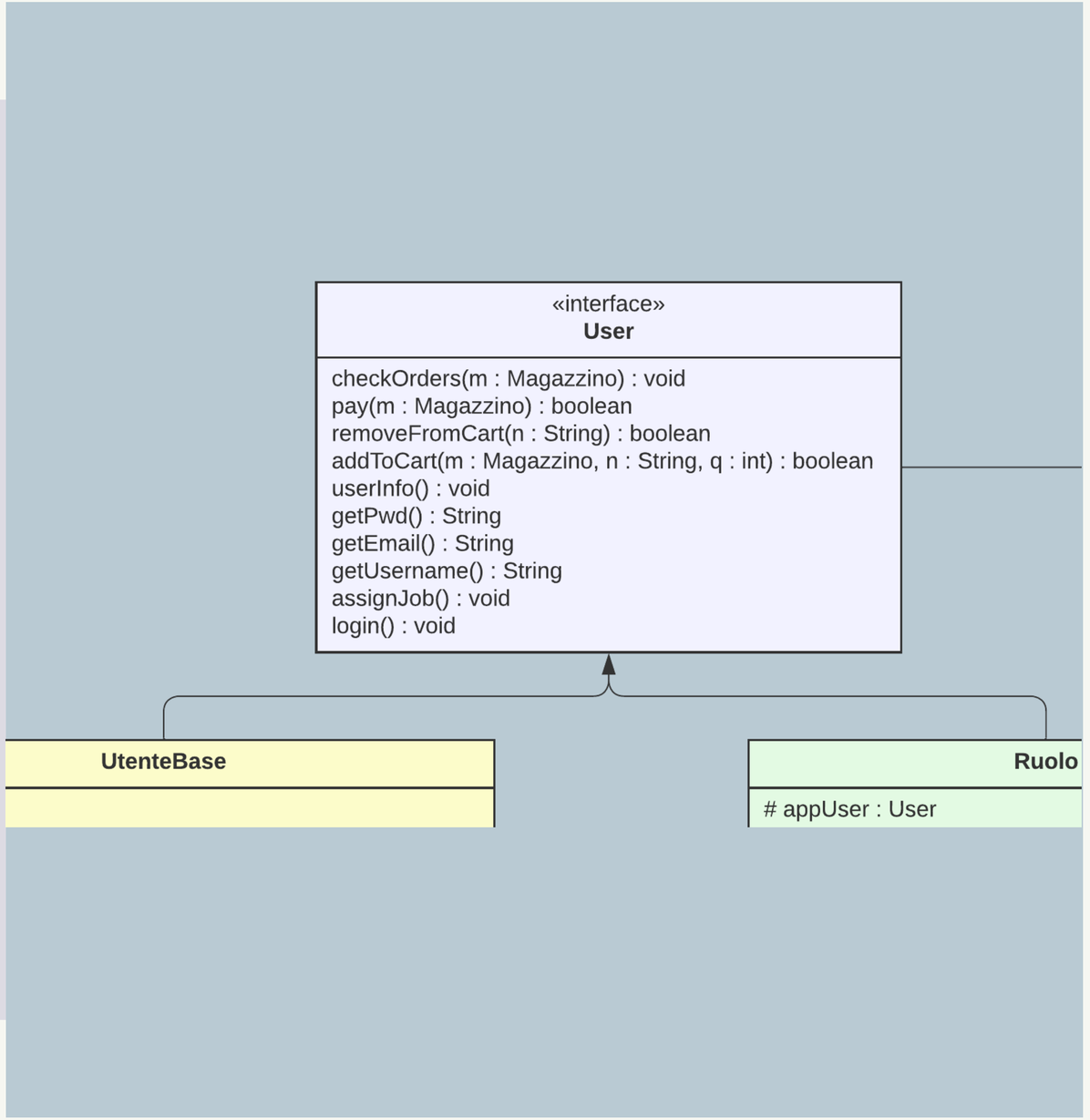
Svantaggi:

- Il decorator e i suoi component non sono identici (==)
- Genera tanti “piccoli” oggetti simili tra di loro



Component

Definisce l’interfaccia per gli oggetti a cui verranno aggiunte responsabilità in modo dinamico



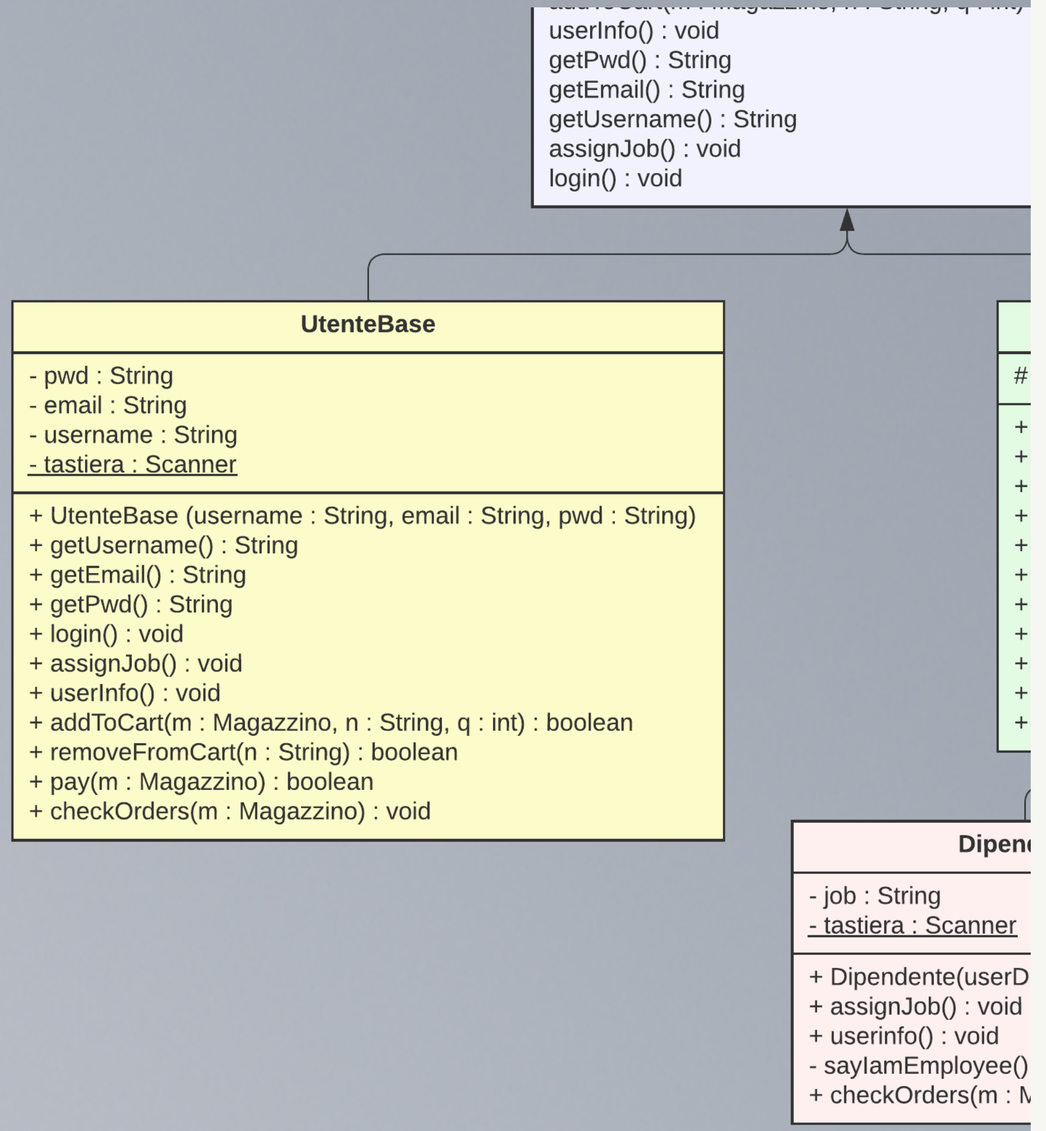
È importante mantenere l'interfaccia User "**leggera**": il suo scopo è la definizione di una interfaccia conforme, non deve contenere altri dati.

Mettere troppe funzionalità nella classe Component aumenta la possibilità che le sottoclassi concrete "**paghino**" per caratteristiche di cui non hanno un effettivo bisogno.

```java
public interface User {

    public void login();

    public void assignJob();

    public String getUsername();

    public String getEmail();

    public String getPwd();

    public void userInfo();

    public boolean addToChart(Magazzino m, String n, int q);

    public boolean removeFromChart(String n);

    public boolean pay(Magazzino m);

    public void checkOrders(Magazzino m);
}
```

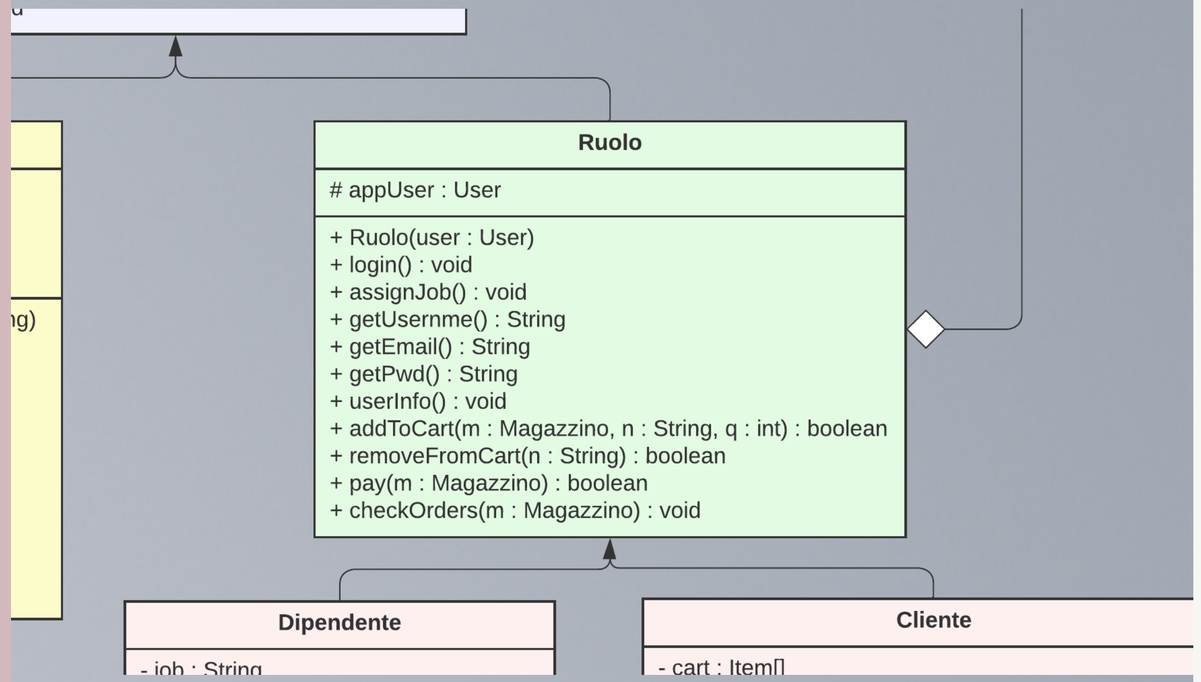
ConcreteComponent

Definisce l'oggetto su cui verranno aggiunte dinamicamente responsabilità



Decorator

Mantiene una referenza al Componenti implementando una interfaccia conforme al Component



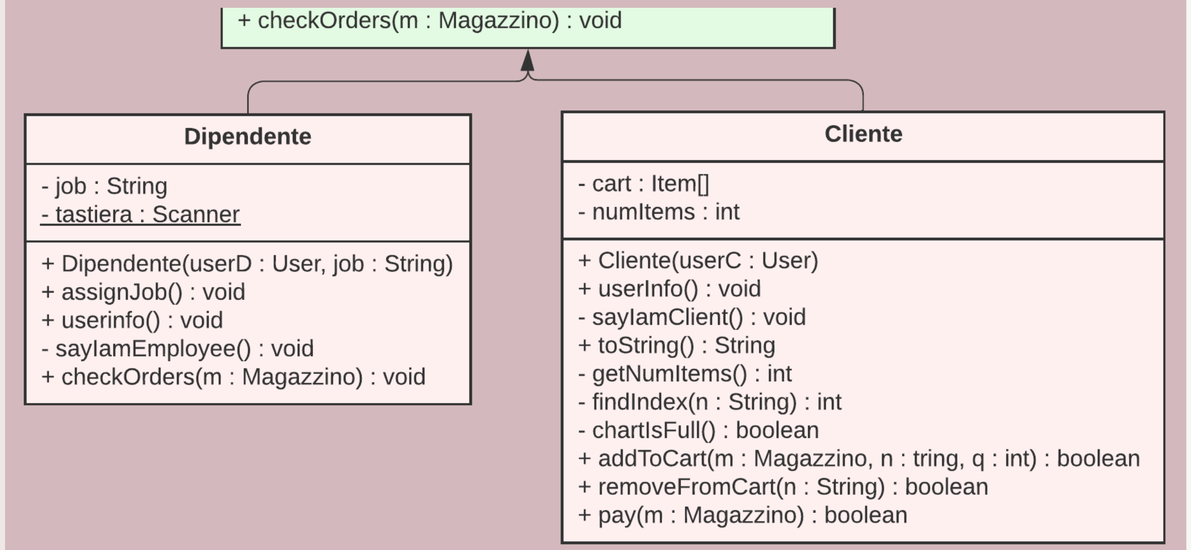
Contiene il codice necessario per immagazzinare al suo interno l'oggetto decorato User, mappa verso di lui le operazioni richieste. Implementa l'interfaccia User, che utilizza per **comunicare** con il Component.

Notare il ruolo chiave del **binding dinamico** che si ottiene ridefinendo i metodi applicati all'oggetto User.

```java
public abstract class Ruolo implements User {

   protected User appUser;

   public Ruolo(User user) {
      appUser = user;
   }

   public void login() {
   }

   public void assignJob() {
   }

   public String getUsername() {
      return appUser.getUsername();
   }

   public String getEmail() {
      return appUser.getEmail();
   }

   public String getPwd() {
      return appUser.getPwd();
   }

   public void userInfo() {
      appUser.userInfo();
   }

   public boolean addToChart(Magazzino m, String n, int q) {
      return appUser.addToChart(m, n, q);
   }

   public boolean removeFromChart(String n) {
      return appUser.removeFromChart(n);
   }
```

ConcreteDecorator

Aggiunge le responsabilità al Component



Le responsabilità specifiche dell'utente **dipendente** sono codificate nella classe Dipendente.

```
40    public void checkOrders(Magazzino m) {
41       if (m.hasOrders()) {
42          System.out.println(m.getOrders());
43       } else {
44          System.out.println("Nessun ordine presente.");
45       }
46    }
47 }
```

Le responsabilità specifiche dell'utente **cliente** sono codificate nella classe Cliente.

```
99    public boolean pay(Magazzino m) {
100      if (numItems == 0) {
101         System.out.println("Cestino vuoto, pagamento non andato a
       buon fine.");
102         return false;
103      } else {
104
105         int i = 0;
106         int bill = 0;
107         while (i < numItems) {
108            m.addOrder(this.getEmail(), cart[i].getName(), cart[i].
       getPrice(), cart[i].getQuantity());
109            bill += cart[i].getQuantity() * cart[i].getPrice();
110            ++i;
111         }
112
113         //System.out.println("Prodotti acquistati:\n" + this + "
       Saldo totale pagato: " + bill + "$");
114         System.out.println("Saldo totale pagato: " + bill + "$");
115
116         this.numItems = 0;
117         this.cart = new Item[10];
118         return true;
119      }
120   }
121 }
```

Problema del Diamante

```java
    User utente2 = new UtenteBase("chiara.genovese", "chiara.genovese@me.it", "querty99");
    //chiara.userInfo();

    User chiara = new Dipendente(utente2, "sarta");
    //chiara.assignJob();
    System.out.println("\nInformazioni sull'utente dipendente:");
    chiara.userInfo();

    System.out.println("\nL'utente dipendente Chiara controlla gli ordini da evadere.");
    chiara.checkOrders(magazzinoSplendor);


    // L'utente dipendente Chiara diventa anche un cliente dell'Atelier Splendor

    System.out.println("\nL'utente dipendente Chiara diventa anche un cliente dell'Atelier Splendor.");
    User chiara_cliente = new Cliente(chiara);
    System.out.println("Informazioni sull'utente dipendente-cliente:");
    chiara_cliente.userInfo();

    // Chiara aggiunge i seguenti items al suo cestino
    chiara_cliente.addToChart(magazzinoSplendor, "t-shirt", 1);
    chiara_cliente.addToChart(magazzinoSplendor, "giacca", 2);

    System.out.println("\nL'utente cliente-dipendente Chiara aggiunge i seguenti items al carrello:");
    System.out.println(chiara_cliente);

    System.out.println("L'utente cliente-dipendente Chiara effettua l'ordine ed  lei stessa che controlla gli ordini da evadere.");
    chiara_cliente.pay(magazzinoSplendor);
    chiara_cliente.checkOrders(magazzinoSplendor);
}
```

```
Informazioni sull'utente dipendente:
Sono un utente dipendente con la mansione di sarta. Username: chiara.genovese, Email: chiara.genovese@me.it

L'utente dipendente Chiara controlla gli ordini da evadere.
Ordini in attesa di essere evasi:
cliente: marco.rossi@me.it, item: t-shirt, price: 33$, quantity: 3
cliente: marco.rossi@me.it, item: giacca, price: 256$, quantity: 1
cliente: marco.rossi@me.it, item: jeans, price: 80$, quantity: 1
cliente: marco.rossi@me.it, item: camicia, price: 90$, quantity: 2
Numero totale di items da spedire: 7

L'utente dipendente Chiara diventa anche un cliente dell'Atelier Splendor.
Informazioni sull'utente dipendente-cliente:
Sono un utente cliente. Sono un utente dipendente con la mansione di sarta. Username: chiara.genovese, Email: chiara.genovese@me.it
Numero di items nel carrello: 0

L'utente cliente-dipendente Chiara aggiunge i seguenti items al carrello:
Carrello:
item: t-shirt, price: 33$, quantity: 1
item: giacca, price: 256$, quantity: 2
Numero totale di items: 3

L'utente cliente-dipendente Chiara effettua l'ordine ed è lei stessa che controlla gli ordini da evadere.
Saldo totale pagato: 545$
Ordini in attesa di essere evasi:
cliente: marco.rossi@me.it, item: t-shirt, price: 33$, quantity: 3
cliente: marco.rossi@me.it, item: giacca, price: 256$, quantity: 1
cliente: marco.rossi@me.it, item: jeans, price: 80$, quantity: 1
cliente: marco.rossi@me.it, item: camicia, price: 90$, quantity: 2
cliente: chiara.genovese@me.it, item: t-shirt, price: 33$, quantity: 1
cliente: chiara.genovese@me.it, item: giacca, price: 256$, quantity: 2
Numero totale di items da spedire: 10
```

Decorator vs Strategy

Il pattern Decorator e il pattern Strategy costituiscono due modi **alternativi** per cambiare il comportamento di un oggetto.

- Possiamo pensare al Decorator come a una **pelle** che ricopre l'oggetto e che ne cambia il comportamento.
- Quando la classe Component è "**pesante**" (ricca di contenuti) il pattern Decorator è troppo costoso da applicare iterativamente e si preferisce lo strategy.
- Per un cambiamento più **viscerale**, serve invece lo Strategy.
- La differenza principale rispetto al decorator è che ci permette di alterare/estendere le responsabilità andando a **sostituire** l'oggetto Strategy.