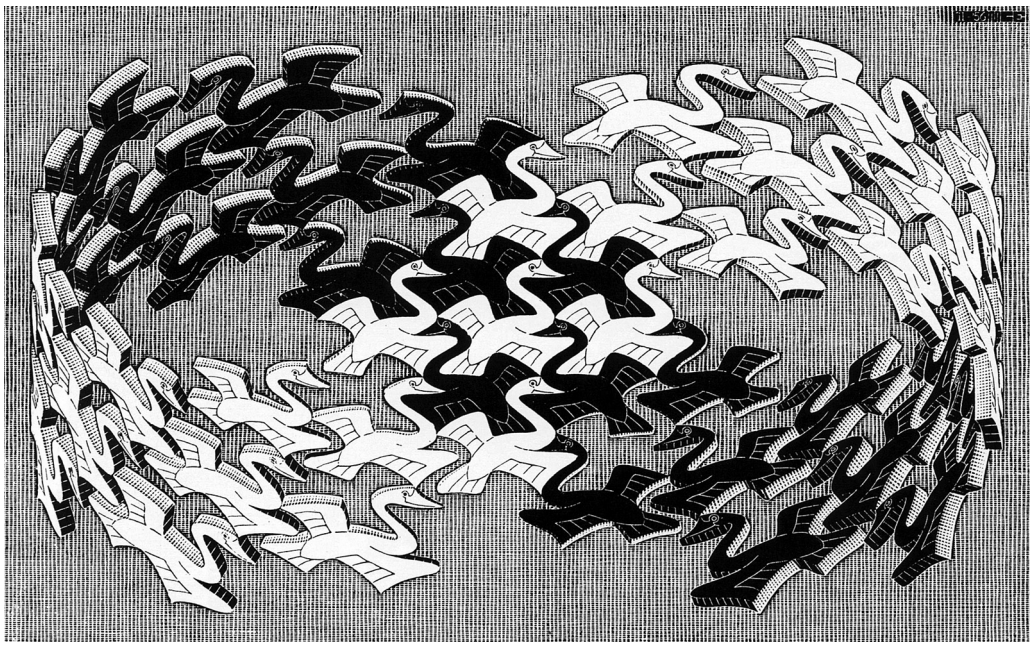
Esame di Design Patterns

Andrea Balbo Mossetto
Davide Marietti
Piaget Bouaka
Edoardo Pastori



Master IT Full Stack Design and Development, a.a. 2022/2023

Indice

Sommario

In questo documento é proposto il lavoro relativo alla realizzazione di un *e-shop* di abbigliamento *innovativo* per conto dell'*Atelier Splendor*. Una definizione piú precisa di questo aggettivo ha richiesto una fase convergente, legata alla presa di coscienza delle caratteristiche proprie di quelli che sarebbero diventati i nostri *competitors*, nonché una fase divergente, attraverso la quale ci siamo soffermati sulle caratteristiche in linea con quella che iniziava a delinearsi con la nostra definizione di *innovativo* mutuate da domini non appartenenti al mondo dell'*e-shop* e dell'abbigliamento.

Lo sviluppo di questo progetto si divide in fasi distinte: si parte dall'elicitazione dei requisiti [1] e da un'analisi di mercato, proponendo idee innovative sul prodotto e analizzando alcuni competitors. Si procede con l'intervista al cliente, da cui abbiamo ottenuto i risultati in base ai quali é stata successivamente prodotta l'analisi dei requisiti. A questo punto vengono definiti attori, obiettivi e casi d'uso. Si passa dunque alla stesura di un documento di vision [2].

Una volta conclusa la parte di progettazione, passiamo all'implementazione vera e propria in linguaggio Java, concentrandoci sull'utilizzo di quattro design patterns: il pattern Decorator, il pattern Strategy, il pattern Composite ed il pattern Observer [3].

Tutto il codice e il materiale relativo a questo la lavoro sono visionabili e scaricabili da un repository pubblico GitHub [4].

1 Elicitazione dei requisiti

Questo capitolo raccoglie i requisiti che andranno poi analizzati, modellati e specificati nella fase di **Analisi dei Requisiti**.

I problemi comuni da evitare in questa fase iniziale, per uno sviluppo florido del progetto, si riferiscono a 3 domini principali [2]:

1. **Scopo**: i requisiti devono riflettere i bisogni del cliente,
2. **Comprensione**: é necessaria una buona comunicazione tra clienti, sviluppatori ed utenti per portare sul pratico i bisogni del cliente, che rifletteranno quelli degli utenti a cui vuole rivolgersi,
3. **Volatilitá**: i requisiti possono cambiare, non essere completi all'inizio dello sviluppo ed evolvere nel tempo; é quindi necessario definirli con quanta piú perizia possibile in questa fase del processo.

Si rende quindi necessaria una stretta comunicazione con il cliente, rappresentato dalla Professoressa Bono in qualitá di proprietario dell'*Atelier Splendor*, per giungere ad un insieme di conoscenze che permetta di superare i problemi esposti poco sopra.

La strategia adottata é quella dell'**intervista**, alla quale siamo giunti dopo un **brainstorming** che ci ha visto partecipi in un'iniziale definizione delle caratteristiche che, a nostro parere, dovrebbero appartenere ad un *e-shop innovativo*.

Abbiamo optato per questo strumento poiché il committente è un singolo imprenditore a capo di un'azienda priva di una struttura di gestione complessa. Inoltre nessuno nel team di sviluppatori aveva esperienza pregressa nel campo della moda, né informazioni abbastanza precise sul committente, dunque risultava fondamentale un incontro conoscitivo diretto.

In una prima fase abbiamo posto numerose domande aperte al fine di ottenere più informazioni generali possibili sul passato, sul presente e sui suoi obiettivi per il futuro. In base alle risposte ottenute abbiamo iniziato la seconda fase dell'intervista, dove, delle tante idee elaborate come team nel brainstorming precedente, abbiamo proposto noi a risposta chiusa solo quelle che abbiamo ritenuto più adatte.

Dopo questa fase propositiva, che ha visto selezionate dal committente solo le proposte migliori, abbiamo discusso sui loro possibili sviluppi futuri, sulle priorità e preso nota delle successive proposte del cliente, lasciando aperta la possibilità di un successivo incontro. Le nostre riflessioni sono sintetizzate nella sezione successiva.

1.1 Analisi di mercato e idee di prodotto

In questa fase abbiamo tratteggiato i contorni, ancora sfumati, dell'applicazione. Servono più informazioni possibili per avere una **visione ampia del mercato** in cui il prodotto richiesto si collocherà, ma anche idee originali a cui ispirarsi, che non devono necessariamente venire dal mondo dell'abbigliamento. Bisogna guardare al di là del *main stream*, l'applicazione dovrebbe creare l'**effetto wow!**

Un'analisi di mercato approfondita ci ha permesso di individuare e confrontare le caratteristiche dei competitors e di pensare a possibili **solu-**

	Capi su misura	Riparazione usato	Resi per voucher
YOOX	×	×	×
Net a Porter	×	×	×
Patagonia	×	✓	×
SSENSE	×	×	×
Garment Workshop	×	×	×

Tabella 1: Tabella comparativa sulla capacità sartoriale dei competitors.

zioni innovative non ancora presenti sul mercato, da poter proporre al committente.

Le aziende di cui abbiamo analizzato i servizi offerti sono YOOX, Net a Porter, Patagonia, SSENSE e Garment Workshop, per citarne alcune. Abbiamo innanzitutto preso in considerazione la possibilità di fare leva sulle capacità sartoriali del committente, per offrire vestiti su misura e la possibilità di mandarli in riparazione, o ancora, di effettuare un reso in cambio di voucher da poter utilizzare per nuovi acquisti. Ad eccezione della riparazione post vendita di Patagonia, nessuno di questi servizi è offerto dai competitors analizzati, come mostrato in tabella 1.

Superando gli aspetti non prettamente sartoriali, abbiamo notato un interesse da parte di ciascuno dei competitors ai temi etici e della sostenibilità, sebbene nessuno di questi abbia abbracciato definitivamente la causa, limitandosi solo ad alcuni (vedi il packaging) e tralasciandone altri (vedi l'economia circolare), come si può notare dalla tabella 2.

Risulta comune a tutti la presenza di una **linea green** nella propria offerta, ma considerando ad esempio il tema della sostenibilità ambientale, sociale e dell'economia circolare, solo Patagonia può dirsi promossa. Per quanto riguarda il tema della trasparenza finanziaria, solo Garment Workshop si distingue dagli altri.

Passando invece ad aspetti più legati alla tecnologia, si nota un utilizzo di algoritmi data driven per la profilazione dei clienti online da parte della maggioranza delle aziende (mentre solo due utilizzano metodologie **human driven**).

Infine, è ancora abbastanza limitato l'utilizzo di strumenti innovativi, quali algoritmi di **realtà aumentata** che consentano di vedersi i vestiti digitali addosso dal proprio smartphone, l'invio di capi di abbigliamento senza che questi siano stati scelti dal cliente, o ancora l'abbandono del concetto di negozio fisso, spostandosi ad un approccio più delocalizzato come gli **spot stores** di Garment Workshop.

	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	✓	×	×	×	✓	✓	×	×	×	✓	✓

Tabella 2: Tabella comparativa risultato dell'analisi di mercato fatta sui competitors dell'*Atelier Splendor*.

1.2 Intervista

L'intervista al committente dell'applicazione segue l'iniziale fase di analisi di mercato e di individualizzazione di possibili sviluppi innovativi, la quale è stata essenziale per **strutturare l'intervista** e raccogliere le idee innovative da proporre al committente. Le informazione raccolte nell'intervista serviranno a definire chiaramente i contorni dell'applicazione e a selezionare i moduli innovativi da sviluppare. L'intervista è stata strutturata nella seguente forma a macro blocchi [2]:

Chi: in questa prima fase conoscitiva abbiamo voluto inquadrare meglio la figura del committente. Abbiamo chiesto:

- *Descrizione dell'azienda. La sua dimensione e il fatturato? La localizzazione?* Il cliente è l'*Atelier Splendor*, una micro azienda di abbigliamento a gestione familiare, caratterizzata dalla presenza di una sartoria interna che produce vestiti con il proprio marchio. La sartoria comprende un unico punto vendita nel centro storico di Torino, con produzione nel retrobottega, dove, allo stato attuale, risultano impiegate solamente due sarte, le quali si occupano del loro ruolo in maniera tradizionale. Il cliente non ha specificato il proprio fatturato.
- *Chi sono gli utenti? Target utente finale è basso/medio/alto spendente? giovane/meno giovane, con quali valori e cultura)? Il target futuro deve rimane lo stesso?* Allo stato attuale, il target di clienti risulta alto spendente ed è concentrato nella fascia di età tra i 35 e i 60 anni. In generale il cliente dell'*Atelier Splendor* è colto e molto sensibile ai valori della sostenibilità. Emerge l'obiettivo futuro di ampliare il pro-

prio target, estendendolo soprattutto in direzione di una fascia di età più giovane, comunque alto spendente, individuata più precisamente in neo assunti dai 25 anni in su.

- *Su quale mercato opera attualmente? Su quale mercato punta di operare in futuro?* L'*Atelier Splendor* è attualmente presente come vendite sul solo mercato locale di passanti. Possiamo dunque identificare una clientela locale, prettamente circoscritta alla provincia di Torino. Dalle risposte del committente emerge la volontà di estendere il proprio bacino di clienti vendendo anche come negozio online, puntando con la giusta consapevolezza anche al mercato internazionale.

Che cosa: in questa seconda fase, sempre squisitamente conoscitiva, l'obiettivo è quello di comprendere nel dettaglio le esigenze del committente. Risulta a questo punto necessario carpire qual è la sua idea di applicazione finale e le caratteristiche salienti per lui imprescindibili. La raccolta di queste informazioni è stata essenziale per comprendere il contesto in cui si opera, al fine di selezionare le idee innovative (inizialmente pensate nella fase di analisi di mercato) più adatte da proporre al cliente.

- *Cosa motiva lei e gli stakeholders? Quali sono i vostri valori? In particolare, quali le aspettative in merito al prodotto?* Il cliente ha ribadito come durante questo processo di rapido sviluppo non voglia per nessun motivo allontanarsi dai suoi valori caratterizzanti: la cultura, la sostanza e la consapevolezza per se stessi e per l'ambiente. Il cliente dell'*Atelier Splendor* sceglie infatti un vestito per dire chi è, non lasciando decidere alla massa cosa dovrebbe indossare o chi dovrebbe essere. E' perciò emersa la tendenza al paradigma dello slow-fashion e della consapevolezza, contrapposta a strategie marketing di *hype* adoperate da altre aziende del mercato della moda (*e.g.* Supreme), rifiutando nettamente i canoni della GDO. L'azienda al centro di questo progetto desidera entrare nel mercato online seguendo il proprio stile e rispettando i propri valori fondanti.

- *Quali sono gli obiettivi dell'azienda? L'obiettivo principale, gli obiettivi a breve, medio e lungo termine (possibilmente in ordine di importanza)?* Il cliente è stato molto chiaro nella definizione dei propri obiettivi, ponendo come punto imprescindibile nel breve termine quello di avere un negozio online pienamente funzionante per espandere il proprio business. Guardando invece a un periodo più ampio, l'obiettivo principale è di farsi conoscere e apprezzare sulla scena internazionale della vendita online di abbigliamento, ritagliandosi una propria fetta di mercato che cresca nel tempo.

- *Quali sono i criteri di successo del prodotto? Quali potrebbero essere gli indicatori da monitorare durante il progetto e quale sarebbe un indicatore chiave per stabilire che l'applicazione ha avuto successo?* Il cliente ha preferito non stabilire a priori un fatturato minimo da raggiungere o una quota di mercato da conquistare. Anche a causa della natura di realtà legata al territorio e alla volontà di sviluppo soprattutto come azienda, si è stabilito come indicatore chiave, cartina al tornasole del successo del progetto nella sua prima macro fase, l'impiego a tempo

pieno con contratto a tempo indeterminato di nuovo personale nella sartoria, nello specifico di almeno due nuove sarte per il retrobottega.

Quando: in questa terza fase conoscitiva, puntiamo a definire con precisione le tempistiche relative alla finalizzazione del progetto. Lo scopo di queste domande è di carpire quali/quante idee innovative selezionare nella (successiva) fase di brainstorming con il committente. Il tempo a disposizione determina una selezione che andrà discussa con il cliente e una ripartizione delle idee di valore in moduli "core", moduli "pilota" e moduli non presi in considerazione in questa fase.

- *Quando l'applicazione dovrebbe uscire sul mercato?* La scadenza per la pubblicazione online del sito di e-commerce è stata fissata a tre mesi dalla definizione dei dettagli definitivi da parte di tutti gli stakeholders.
- *Quando l'applicazione dovrebbe essere pienamente operativa?* A seguito della pubblicazione, viene stabilito un tempo di due mesi per risolvere tutte le eventuali criticità e per sviluppare definitivamente tutti gli aspetti tecnici non essenziali, non presenti al momento del lancio.
- *Quando il progetto dovrebbe dirsi concluso?* La fine del progetto non è stata ancora definita, in quanto è stata lasciata aperta la possibilità di ulteriori fasi di sviluppo, che potranno essere prese in considerazione sulla base dei risultati raggiunti nel medio termine.

1.3 Brainstorming

In questa fase, sulla base di tutte le informazioni raccolte attraverso le precedenti domande durante l'intervista, abbiamo proposto al cliente tutte le nostre **idee innovative** relative a diversi possibili sviluppi del progetto. Le idee proposte sono state avvalorate da una precedente analisi di mercato e dalle conoscenze in merito del nostro esperto di dominio. L'insieme delle idee così raccolte, è poi stato ridiscusso con il cliente e suddiviso in gruppi (moduli core, moduli pilota e moduli attualmente non presi in considerazione), come presentato nella successiva sezione sui risultati.

1.4 Risultati

I risultati dell'intervista possono essere suddivisi in tre macro gruppi. Le idee che per motivi strategici o di tempo non vengono ulteriormente sviluppate sono finite nel gruppo dei **moduli attualmente non presi in considerazione**, e di seguito non presentati.

La progettazione e lo sviluppo completo dei **moduli "core"**, strategici per raggiungere gli obiettivi del cliente e contribuire al tasso di innovazione dell'applicazione. L'investimento del cliente e le forze in gioco del team rendono questi obiettivi realistici e realizzabili.

- *"Pop-Up Stores"*: non molti negozi fisici, ma spot stores temporanei, localizzati dove sono concentrati la maggior parte dei clienti. La creazione di una rete di aziende amiche che condividono gli stessi ideali di sostenibilità permette numerosi vantaggi. *Atelier Splendor* ospiterà

virtualmente i capi di queste aziende nel suo futuro negozio virtuale, queste aziende ospiteranno mostre temporanee dei nostri capi nei loro negozi fisici. La scelta delle città in cui organizzare le mostre verrà suggerita dai dati raccolti.

- *"Vestirsi con la realtà aumentata"*: Algoritmo di realtà aumentata stile Ikea che consente di vestirsi virtualmente per provare i vestiti e trovare la propria taglia. Ciò sorride all'ambiente e fa l'occhiolino al consumatore consapevole perché permetterà di ridurre il numero dei resi. Cliente soddisfatto e ambiente pure.
- *"Magazzino Snello"*: rivedere il data model del magazzino da integrare in un data base pinco pallino su cui poggerà la nostra applicazione. Questo consentirà agli impiegati dell'azienda di loggarsi come tali e di monitorare meglio le risorse e i flussi di magazzino, abbattendo i costi generati dalla merce invenduta.
- *"Go Circular!"*: sezione dell'applicazione dedicata ai vestiti usati che viene alimentata dai vestiti creati dall'atelier e dati indietro dai clienti in cambio di un vaucher da spendere sui vestiti nuovi confezionati dall'atelier. In questo modo il negozio ospiterà sì brand diversi e sostenibili, ma cerchiamo di favorire l'acquisto di abiti con il nostro brand. Questa politica, che sorride all'ambiente, sorride anche all'espansione della sartoria e all'assunzione di nuovo personale. Inoltre, gli abiti dati indietro spesso necessiteranno una sistemata da parte delle nostre sarte prima di essere caricato online.

La progettazione e lo sviluppo almeno parziale dei **moduli "pilota"**, strategici ai fine del marketing e strategici in futuro, non appena verranno raggiunti gli obiettivi prefissati più importanti (assunzioni di nuove sarte e aumento del numero di clienti). Le condizioni attuali non consentono lo sviluppo completo di questi moduli. Lo sviluppo futuro di questi moduli consentirà una fruttuosa e duratura collaborazione con l'*Atelier Splendor*.

- *"Algoritmo di consiglio all'acquisto human-driven"*: l'output dell'algoritmo non viene direttamente fornito al cliente, ma è mediato dall'essere umano, nello specifico un esperto di moda, che spiega il perché di questo consiglio di acquisto.
- *"Marketing"*: Nell'ambito del marketing, prodotti mirati saranno associati a un link di un contento TikTok di un influencer che starà indossando quel vestito. Questo per ampliare il target verso una fascia più giovane.
- *"Sartoria dedicata e confezionamento su misura"*: la sartoria interna potrà essere ulteriormente allargata introducendo due nuovi servizi al cliente. Primo, i vestiti che necessiteranno di manutenzione (orli, pence, sostituzione cerniere, rattoppi, ...) potranno essere spediti alla sede centrale per essere aggiustati. Due, la sartoria offrirà la possibilità di avere abiti confezionati su misura.

2 Analisi dei requisiti

Proseguiamo riportando i requisiti che la nostra applicazione dovrà rispettare e seguire nella sua implementazione.

L'analisi dei requisiti infatti è un'attività preliminare allo sviluppo di un prodotto, dove si definiscono le **funzionalità che questo dovrà offrire**, nel nostro caso i requisiti che devono essere soddisfatti dal software per rispondere alle esigenze del committente, e si conclude con la stesura di una dettagliata specifica dei requisiti, che guideranno le fasi di sviluppo successive [2].

Da una buona analisi dei requisiti si ottiene una maggiore qualità del prodotto finale, soprattutto per quanto riguarda caratteristiche non visibili all'utente, come ad esempio la modularità: suddividendo da subito un progetto in diversi sottosistemi, ciascuno dei quali svolge un compito ben definito e dipende dagli altri in modo semplice, lo si rende nel complesso più comprensibile, e la comprensibilità, a sua volta, insieme alla semplicità delle interdipendenze tra questi sottosistemi, rende il sistema più facile da verificare e quindi più affidabile. Infine, mantenere ciascun sottosistema il più indipendente possibile dagli altri, ne rende più facili la modifica e il riuso, rendendo efficiente tutto il processo di sviluppo [2].

2.1 Attori, Obiettivi e Casi d'Uso

L'applicazione e tutto il sistema che traduce deve essere pensata per essere sfruttata da una serie di attori, umani o meno, che, con le loro attivitá, determinano come essa sará sviluppata. Possiamo classificare gli attori nella tassonomia seguente [1]:

- gli **attori principali** sono quelli che utilizzano direttamente l'applicazione per uno specifico use case (sono i fruitori dell'applicazione, e.g. un cliente, un impiegato),
- gli **attori secondari** sono sistemi esterni (tipicamente software) che integriamo all'interno dell'applicazione (sono API che inseriamo per implementare il tutto e su cui non abbiamo controllo),
- gli **attori fuori scena** sono entità che non usano direttamente l'app, ma che ne possono trarre benefici o ci danno dei vincoli da rispettare (l'università è l'attore fuori scena che beneficia di Moodle, la legge con il GDPR).

I Casi d'Uso, invece, descrivono i possibili scenari di utilizzo dell'applicazione, il macro-contesto all'interno del quale si muovono gli attori di cui sopra. Una comune rappresentazione grafica di un Caso d'Uso (o **UC**) é ravvisabile nell'*UC diagram*, ovvero un'*interfaccia* astratta della nostra applicazione (vedi figura 1).

La definizione degli attori deriva da quella degli UC, pertanto procediamo elencando quelli che saranno di nostro interesse progettuale:

- Acquisto di capi di vestiario online (*UC1*),
- Accesso al magazzino snello (*UC2*),

- Aggiornamento dei clienti rispetto all'installazione temporanea di Pop-Up store (*UC3*).

A questo punto possiamo enunciare gli attori, a piú livelli, che entreranno in contatto con gli UC che abbiamo appena definito. In particolare:

- **UC1**
  - Principali: Cliente che vuole acquistare - Dipendente che monitora acquisti,
  - Secondari: Metodi di pagamento,
  - Fuori scena: Compagnia di trasporti .
- **UC2**
  - Principali: Cliente nella sua forma di avatar - Dipendente sarto che gestisce il proprio lavoro rispetto al traffico degli avatar,
  - Secondari: Algoritmo che coadiuva i dipendenti nella gestione del magazzino,
  - Fuori scena: Fornitori di materie prime.
- **UC3**
  - Principali: Cliente che vuole essere aggiornato rispetto alle attivitá della Compagnia,
  - Secondari: Database collegato a software tramite API per gestire informazioni di contatto dei clienti interessati alla newsletter,
  - Fuori scena: GDPR.

2.2 Requisiti

Con *requisiti* intendiamo caratteristiche del sistema, da un punto di vista puramente logico o da un punto di vista tecnico [1]. Piú nello specifico, distinguiamo:

- **requisiti funzionali**, che riguardano ciò che il sistema fa e sono i moduli da implementare o integrare per lo sviluppo dell'applicazione,
- **requisiti non funzionali**, che riguardano come é fatto il sistema (specificano ad esempio quali sono i tools che vengono utilizzati per la sua realizzazione, il linguaggio di programmazione con cui é scritto, il database a cui si appoggia, ...).

Descriviamo quindi i requisiti che abbiamo determinato relativamente agli UC individuati nella sezione precedente:

- UC1
  - funzionali: voglio permettere all'utente di accedere al mio sito web e alla mia applicazione, visualizzare l'elenco degli articoli a mia disposizione, fargli scegliere quelli che piú lo aggradano, inserirli nel carrello, pagare e completare l'ordine. Voglio permettere al dipendente di visualizzare gli ordini,
  - non funzionali: tools Java (pattern decorator + strategy), html + javascript + css (front-end), Oracle.
- UC2

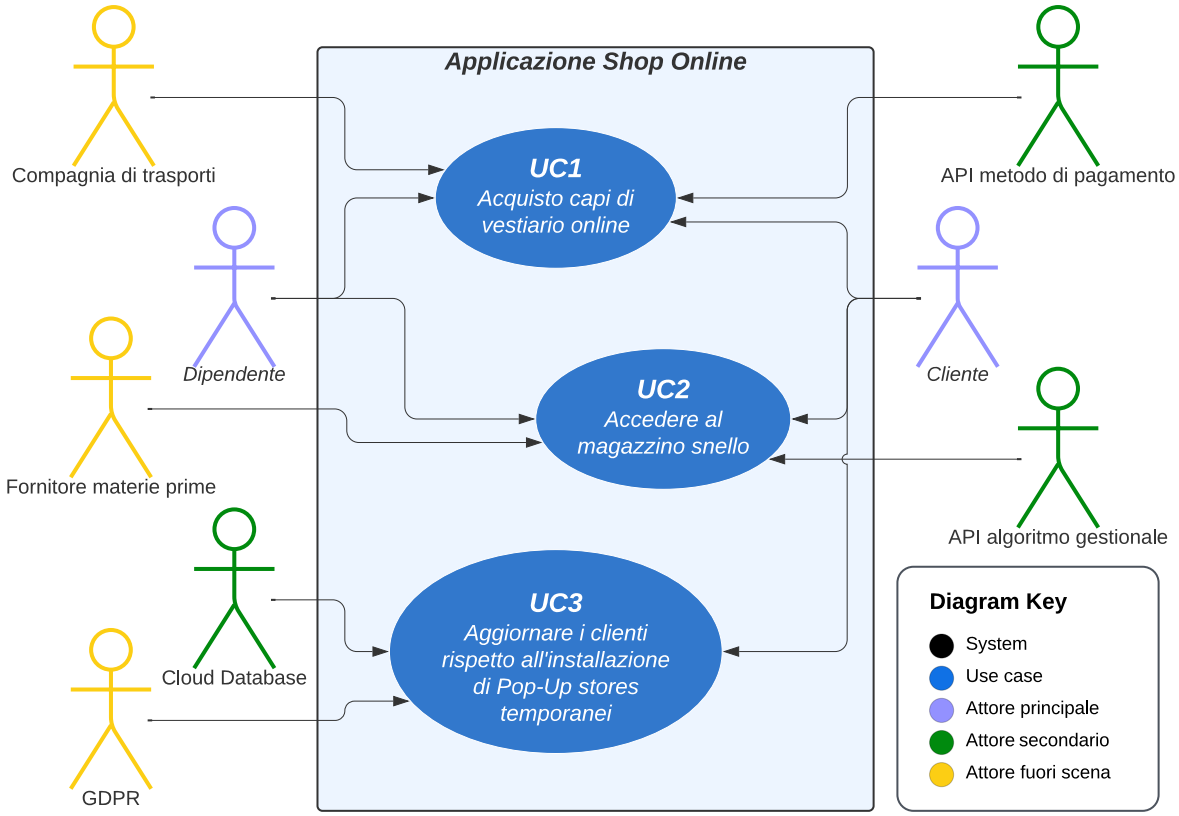


Figura 1: Esempio di UC diagram relativo all'applicazione del negozio online. All'interno delle *system boundaries* sono posizionati i casi d'uso esaminati in questa sezione. Gli attori principali (azzurro), quelli secondari (verde) e quelli fuori scena (giallo) sono rappresentati dalle figure al di fuori del *system* e sono collegati ai relativi casi d'uso. Una più approfondita analisi del diagramma è presentata in sezione 2.1

  - funzionali: voglio permettere all'utente di gestirsi (eventualmente) l'avatar, tramite cui provera' le diverse combinazioni di capi. I dati immagazzinati serviranno al sarto per una gestione piú lean del magazzino (vedi anche ordini con i fornitori di materie prime),
  - non funzionali: Oracle, Java (pattern Composite), Three-js (libreria js sul browser per realtá aumentata) / webGL (grafica 3D).

- UC3
  - funzionali: voglio permettere ai miei "iscritti" di ricevere informazioni riguardanti i prossimi eventi pop-up store nella zona geografica di competenza,
  - non funzionali: Oracle, Java (pattern Observer), client mail, API di geolocalizzazione.

2.3 Glossario degli attori

Viene fornito al lettore un glossario dei termini tecnici utilizzati nelle sezioni precedenti [1]:

- API: *"le API sono meccanismi che consentono a due componenti software di comunicare tra loro usando una serie di definizioni e protocolli. Ad esempio, il sistema software dell'ufficio meteorologico contiene dati meteorologici giornalieri. L'applicazione del meteo sul cellulare "parla" con questo sistema tramite API e mostra gli aggiornamenti meteorologici quotidiani sul telefono"* [5].
- Client mail: un programma che permette di gestire il servizio di posta elettronica.
- Database: *un database é una collezione organizzata e strutturata di dati. Il database é controllato da un Database Management System (DMBS), che funge da interfaccia tra l'utente che vuole avere accesso ai dati e il pc su cui questi dati sono salvati* [6]; un celebre esempio di DBMS e' Oracle.
- Libreria: insieme di funzioni e strutture dati che permette allo sviluppatore di avere a propria disposizione codice giá pronto alle proprie necessitá.
- Linguaggio di marcatura: insieme di regole che definiscono e strutturano l'impaginazione di un testo. Faremo in particolare ricorso ad *HTML* e *CSS*. La stesura di codice in questi linguaggi definirá ció che gli utenti vedranno tramite interfaccia.
- Linguaggio di programmazione: linguaggio formale che specifica un insieme di istruzioni che vorranno interpretate e/o compilate dal computer, restituendo dei dati in uscita. Ci avvaliamo di *Javascript* e *Java*.
- Pattern (*Design Pattern*): modello logico a cui si ricorre per proporre una soluzione ad un problema che riguarda l'architettura e la progettazione del programma.

3 Documento di vision

3.1 Vision

Il progetto nasce dal desiderio di una piccola sartoria di Torino, fortemente legata al territorio, di investire a lungo termine sul proprio sviluppo commerciale.

L'azienda in questione esiste per soddisfare il bisogno di una clientela selezionata di vestire alla moda, con classe e stile. Possiede infatti il know-how necessario per la produzione di capi di alta qualità e per la riparazione sartoriale degli stessi. Risulta inoltre presente da anni sul mercato, di cui conosce le dinamiche e i bisogni, con risultati che hanno rispettato le aspettative. Vengono ricercate soluzioni ad un obiettivo di crescita economica e di brand, facendo leva sui nuovi strumenti che l'attuale tecnologia consente, in maniera sostenibile.

Il contatto con il cliente nasce a seguito di un piano a lungo termine di sviluppo di soluzioni innovative totalmente slegate dal mercato del fashion. Una delle principali ambizioni è quella di diffondere oltre i confini urbani di Torino il brand ed il business, entrando in modo innovativo nel mondo della vendita online, cogliendo tutte le opportunità che questa può offrire a chi investe nel proprio mercato. Tutto nasce dal bisogno di allargare il bacino di clienti ad un pubblico più variegato, slegandosi dal solo mercato tradizionale di over 35, facendo breccia sul mercato dei giovani over 25. La sartoria sogna infatti di sviluppare una rete commerciale internazionale, che metta al centro i propri prodotti e servizi, ma che allo stesso tempo commercializzi prodotti di comprovata qualità per conto di terzi.

Alla base di ogni aspetto del progetto vi è l'uso della tecnologia, attraverso l'utilizzo di un'applicazione sviluppata ad hoc che raccolga dati che possano guidare anche geograficamente l'espansione commerciale, o il sito internet per coinvolgere gli utenti online con un'esperienza nuova di acquisto di capi di abbigliamento rispetto al resto del mercato della moda. Viene presa infatti ispirazione da mercati completamente slegati da quello di riferimento, per creare un nuovo approccio degli utenti all'acquisto online di vestiti e accessori, con l'obiettivo di creare un rapporto forte e duraturo sia con i nuovi clienti che con quelli di lungo corso.

L'intero progetto mira dunque a creare uno sviluppo sociale ed economico, come diretta conseguenza della nuova posizione dell'azienda sul mercato online, offrendo una valida alternativa ai modelli di crescita insostenibili che caratterizzano ormai l'intero settore della moda.

3.2 Valori

Nel progetto è centrale il rispetto dei valori sempre più all'attenzione del grande pubblico. Risultano perciò imprescindibili in tutte le fasi di sviluppo del progetto:

- trasparenza nella raccolta di dati degli utenti online,
- condizione dell'ambiente di lavoro e dei lavoratori,
- provenienza delle materie prime e condizioni degli allevamenti di bestiame per la produzione di lana,
- utilizzo di coloranti conformi a tutte le normative e non dannosi per l'ambiente.

4 Design Patterns utilizzati

Un *pattern* è una soluzione generale ad un problema ricorrente, che va opportunamente adattato al problema in esame (cfr.[7], p.12). Un Pattern può essere riassunto in: problema, soluzione, pro e contro della soluzione ed esempi d'uso. I Design pattern visti a lezione sono i **GoF Design Pattern** e permettono di sfruttare al massimo le caratteristiche object oriented di Java. Essi costituiscono di fatto una mappa per scrivere codice Java [3].

In questa sezione riportiamo sotto forma di micro prototipi le implementazioni dei design pattern applicati ai casi d'uso della nostra applicazione innovativa.

- Con riferimento al caso d'uso dell'acquisto di capi di vestiario online, un dipendente dell'atelier può anche essere un cliente e utilizzare l'applicazione con due scopi differenti: per controllare gli ordini da evadere o per acquistare capi sullo store online. Abbiamo quindi pensato di modellare la figura del *dipendente-cliente* tramite due pattern alternativi: il **pattern Decorator** (cfr.[7], p.196) e il **pattern Strategy** (cfr.[7], p.349). Ciò ci ha permesso di confrontare e discutere i vantaggi/svantaggi di questi due pattern.
- Con riferimento al caso d'uso dell'accesso al magazzino snello, possiamo ricollegarci al modulo sull'approccio human-driven per implementare il **pattern Composite** (cfr.[7], p.183). Lato tecnico usiamo il pattern composite. Lato umano, i sottoinsiemi generati dal composite verranno filtrati dall'umano esperto. Alternativamente, il pattern composite potrebbe essere applicato semplicemente al caso d'uso dell'accedere al magazzino snello con riferimento al modulo del *magazzino snello*.
- Con riferimento al caso d'uso dell'aggiornare i clienti rispetto all'installazione di Pop-Up stores temporanei, l'idea di allestire una newsletter ci ha indirizzati nell'implementazione del **pattern Observer** (cfr.[7], p.326).

4.1 Pattern Decorator

Il pattern *Decorator*, noto anche come *Wrapper*, è un design pattern **strutturale** che permette di aggiungere dinamicamente responsabilità addizionali ad un oggetto individuale. In questo modo si possono estendere le funzionalità di oggetti particolari senza coinvolgere intere classi (cfr.[3], p.65).

Un modo per aggiungere responsabilità ad un oggetto potrebbe essere tramite la semplice *ereditarietà*. Ciò consente una certa flessibilità, tuttavia rimane una certa staticità dovuta al fatto che il client non può controllare quanto e come aggiungere tale proprietà all'oggetto. Un approccio più flessibile è dato dall'includere l'oggetto che si vuole decorate in una *wrapper classes* (il *Decorator*) che aggiunge la nuova responsabilità voluta. Gli oggetti decorati assieme al Decorator devono sottostare a una interfaccia comune, che rende la sua presenza *trasparente* e permette all'applicazione di continuare ad interagire con gli oggetti decorati [3]. Tale proprietà di trasparenza rende iterativa e potenzialmente illimitata l'applicazione di responsabilità aggiuntive tramite nuovi Decorators. La proprietà di trasparenza permette inoltre ai decorators di apparire ovunque l'oggetto da decorare possa apparire, in modo che i clients non siano in grado di distinguere un oggetto decorato da uno che non lo sia [7].

Un esempio illuminante di possibile utilizzo ci è proposto dalla *Gang of Four* (cfr.[7], p.197): Supponiamo di avere un oggetto TextView che mostra del testo in una finestra. Di default, la TextView non ha scroll bar, perché non sempre ne abbiamo bisogno. Quando necessario possiamo "decorare" la TextView con una scroll bar. Supponiamo di volere anche un bordo più spesso attorno alla finestra. Possiamo "decorare" l'oggetto TextView anche

con quello e con tutte le altre caratteristiche aggiuntive fino a ottenere il risultato voluto.

Con riferimento invece al caso d'uso dell'acquisto di un abito sull'applicazione, si pensi ad un modello di oggetti che rappresenti i dipendenti dell'*Atelier Splendor*. Un impiegato può essere un cliente interessato a comprare un abito sulla piattaforma. Nasce così l'esigenza di modellare la figura dell'**dipendente-cliente**, il quale in aggiunta alle operazioni definite per i dipendenti, definisce anche quelle relative ai clienti.

Il pattern Decorator ha i seguenti **vantaggi** evidenti:

- Il pattern Decorator è più **flessibile** dell'ereditarietà statica (multipla). Con il Decorator le responsabilità possono essere infatti aggiunte/tolte dinamicamente a *run time*, semplicemente attaccando o staccando il Decorator. Mentre utilizzando l'ereditarietà servirebbe creare una nuova classe per ciascuna nuova responsabilità addizionale, con il conseguente proliferare di classi che aggiungo complessità al sistema. Inoltre, con il Decorator avrei il vantaggio di poter combinare liberamente tra loro le varie responsabilità. Infine, il Decorator rende più semplice e meno prono ad erri l'aggiungere più volte una stessa proprietà a un dato oggetto (e.g. nel caso della TextView, di aggiungere due volte il bordo) [7].
- Evita che classi ricche di caratteristiche scalino la gerarchia. Il Decorator offre infatti un **approccio "pay-as-you-go"** all'aggiunta di responsabilità: invece di cercare di gestire tutte le caratteristiche di un oggetto in un'unica classe grande e complessa, possiamo definire una classe semplice e aggiungerci le caratteristiche in modo **incrementale** con gli oggetti Decorator. Le funzionalità vengono composte da pezzi piccoli e semplici. Come risultato un'applicazione non deve "pagare" per caratteristiche che non usa [7]. Questo approccio semplifica anche la definizione di nuove caratteristiche che non erano state pensate o incluse in partenza, evitando che l'estensione di una classe complessa possa esporre dettagli non connessi alle responsabilità che vogliamo aggiungere.

Mentre gli **svantaggi** più evidenti sono:

- Un Decorator e i suoi *component* **non sono identici (==)**. Esso si comporta come un "contenitore trasparente", ma da un punto di vista dell'identità tra oggetti, un oggetto decorato non è uguale all'oggetto stesso non decorato. Quindi, non bisogna fare affidamento sull'identità tra oggetti (==) quando si usa questo pattern [3].
- Questo pattern genera tanti "piccoli" oggetti simili tra di loro. Tali oggetti differiscono solo dal modo in cui sono interconnessi, non dalla classe o dal valore delle loro variabili. Seppur semplici da customizzare per chi li ha creati, risultano **difficili da comprendere e debuggare** da terzi [7].

Per implementare il pattern Decorator contestualmente al caso d'uso dell'acquisto di abiti online, abbiamo bisogno delle seguenti componenti minimali (cfr.[3], p.65):

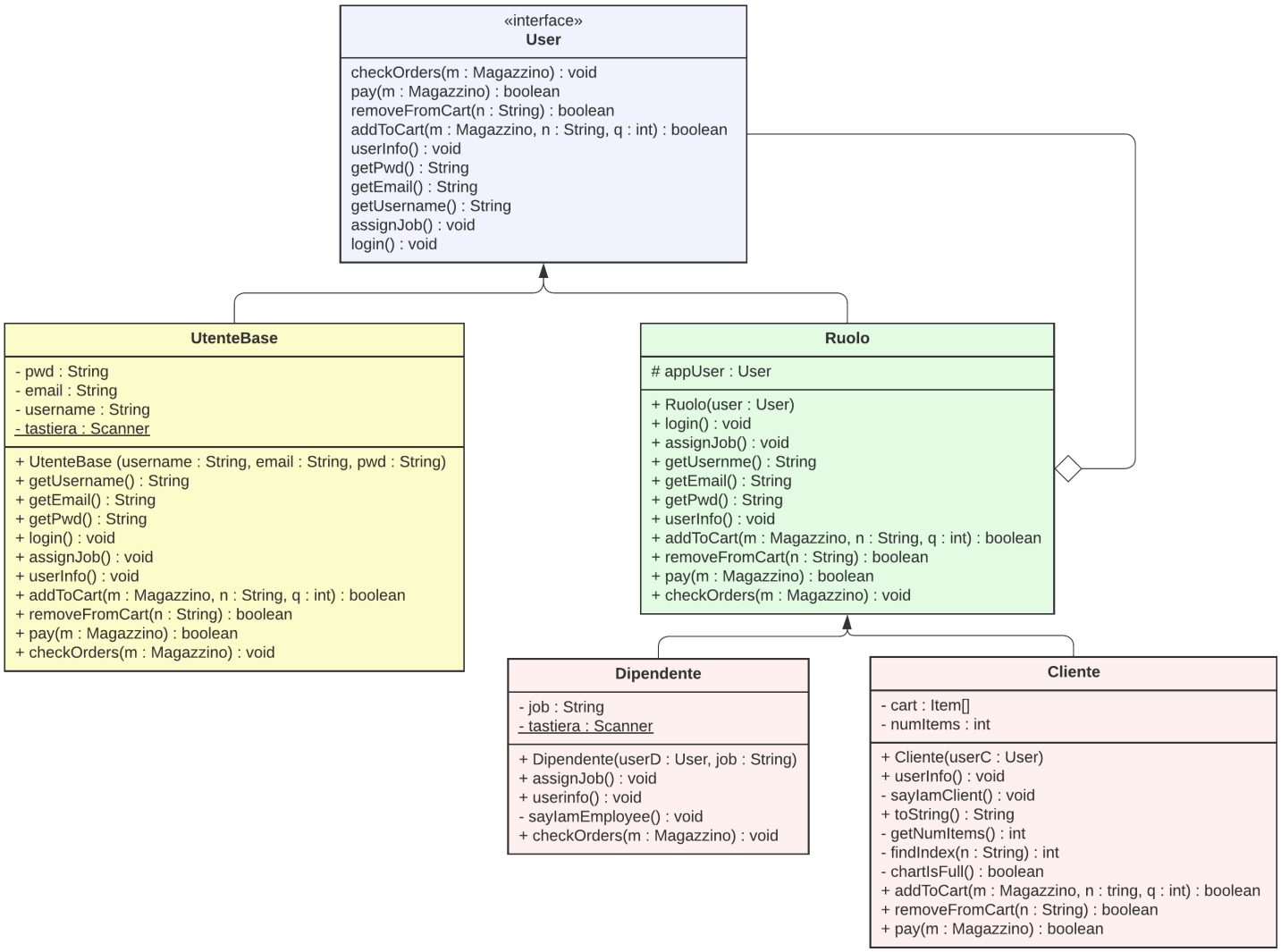


Figura 2: Rappresentazione in UML (Unified Modeling Language) del pattern Decorator impiegato per modellare la figure del dipendente-cliente [8].

- Un *Component* (`User`): definisce l'interfaccia per gli oggetti a cui verranno aggiunte proprietà in modo dinamico.
- Un *ConcreteComponent* (`UtenteBase`): definisce l'oggetto su cui verranno dinamicamente aggiunte le proprietà.
- Un *Decorator* (`Ruolo`): mantiene una referenza al *Component* e definisce un'interfaccia conforme a quella del *Component*.
- Uno o più *ConcreteDecorator* (`Dipendente, Cliente`): aggiunge le responsabilità al *Component*.

La relazione tra le componenti del decorator implementato per l'applicazione dell'*Atelier Splendor* è illustrata sotto forma di UML diagram in figura 2.

Listing 1: User.java – Component del pattern Decorator [3].

```java
/*
Specifica la interfaccia degli objs User, che possono essere
    decorati dinamicamente con nuove caratteristiche. Tutti le
    componenti del pattern Decorator implementano la interfaccia
    User, viene detta "Component".
*/

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);
}
```

Con riferimento a list.1 e a figura 2, tutti le componenti di questo pattern implementano l'interfaccia `User`, che specifica soltanto una serie di metodi pubblici. Notare come sia importante mantenere l'interfaccia `User` (*Component*) "leggiera", ossia: **il suo scopo è la definizione di un'interfaccia conforme, non deve contenere altri dati**. La definizione dei dati rappresentativi è competenza delle sottoclassi, altrimenti la complessità della classe *Component* rende i *decorators* troppo pesanti per essere applicati ricorsivamente. Inoltre, Mettere troppe funzionalità nella classe *Component* aumenta la possibilità che le sottoclassi *concrete* "paghino" per caratteristiche di cui non hanno un bisogno effettivo [7].

Listing 2: UtenteBase.java – ConcreteComponent del pattern Decorator [3].

```java
/*
Definisce un obj UtenteBase implementando la interfaccia User, a cui
     si possono aggiungere nuove caratteristiche. Questa componente
    del pattern Decorator viene detta "ConcreteComponent".
 */

import java.util.Scanner;

public class UtenteBase implements User {

```

```
9     private static Scanner tastiera = new Scanner(System.in);
10
11    private String username, email, pwd;
12
13    public UtenteBase(String username, String email, String pwd) {
14       this.username = username;
15       this.email = email;
16       this.pwd = pwd;
17    }
18
19    public String getUsername() {
20       return username;
21    }
22
23    public String getEmail() {
24       return email;
25    }
26
27    public String getPwd() {
28       return pwd;
29    }
30
31    public void login() {
32       System.out.println("username: ");
33       this.username = tastiera.nextLine();
34       System.out.println("email: ");
35       this.email = tastiera.nextLine();
36       System.out.println("password: ");
37       this.pwd = tastiera.nextLine();
38       tastiera.nextLine();
39    }
40
41    public void assignJob() {
42    }
43
44    public void userInfo() {
45       System.out.println("Username: " + getUsername() + ", Email: "
      + getEmail());
46    }
47
48    public boolean addToChart(Magazzino m, String n, int q) {
49       return false;
50    }
51
52    public boolean removeFromChart(String n) {
53       return false;
54    }
55
56    public boolean pay(Magazzino m) {
57       return false;
58    }
59
60    public void checkOrders(Magazzino m) {};
61 }
```

Con riferimento a list.2 e a figura 2, la classe `UtenteBase` implementa l'interfaccia `User`.

Listing 3: Ruolo.java – Decorator del pattern Decorator [3].

```
/*
La classe astratta Ruolo corrisponde al Decorator del pattern.
    Contiene il codice necessario per immagazzinare al suo interno
    loggetto decorato (Component), e mappa verso di lui le
    operazioni richieste. Si noti che questa classe implementa la
    intrerfaccia User, e al suo interno utilizza questa stessa
    interfaccia per comunicare con il Component.
 */

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);
   }

   public boolean pay(Magazzino m) {
      return appUser.pay(m);
   }

```

```
47      public void checkOrders(Magazzino m) {
48          appUser.checkOrders(m);
49      }
50  }
```

Con riferimento a list.3 e a figura 2, La classe astratta `Ruolo` corrisponde al *Decorator* del modello. Contiene il codice necessario per immagazzinare al suo interno l'oggetto decorato `User`, e mappa verso di lui le operazioni richieste. Si noti che questa classe implementa l'interfaccia `User`, e al suo interno utilizza questa stessa interfaccia per comunicare col *Component* [3].

Notare l'importanza della conformità delle interfacce: l'interfaccia di un oggetto decorato deve essere conforme all'interfaccia del *Component* che decora. Di conseguenza, la classe *ConcreteDecorator* deve ereditare da una classe comune [7].

Nella classe `Ruolo`, notare come il **Binding Dinamico** che si ottiene ridefinendo i metodi applicati all'oggetto `appUser` sia la chiave per il funzionamento corretto del pattern Decorator.

Listing 4: Dipendente.java – Primo ConcreteDecorator del pattern Decorator [3].

```
1  /*
2  La classe Dipendente, invece estende le variabili di stato
       delloggetto decorato, e ne modifica il comportamento.
3  */
4
5  /*
6  Il dipendente deve poter controllare gli ordini effettuati dai
       clienti e avere accesso alla gestione del magazzino.
7   */
8
9  import java.util.Scanner;
10
11 public class Dipendente extends Ruolo {
12
13     private static Scanner tastiera = new Scanner(System.in);
14
15     private String job;
16
17     public Dipendente(User userD, String job) {
18         super(userD);
19         this.job = job;
20     }
21
22     public void assignJob() {
23         System.out.println("Mansione: ");
24         this.job = tastiera.nextLine();
25     }
26
27     public void userInfo() {
28         sayIamEmployee();
29         super.userInfo();
30     }
31
32     private void sayIamEmployee() {
```

```
33      System.out.print("Sono un utente dipendente con la mansione di
        " + job + ". ");
34    }
35
36    /*
37    La responsabilit aggiuntiva principale assegnata a un dipendente
        quella
38    di controllare gli ordini di vendita
39    */
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 }
```

Con riferimento a list.4 e a figura 2, Le responsabilità specifiche riguardo la figura dell'utente dipendente sono codificate nella classe `Dipendente` [7]. Questa classe estende le funzioni del *Decorator*, particolarmente aggiungendo la possibilità di controllare gli ordini presenti nel magazzino tramite il metodo `checkOrders`.

Listing 5: Cliente.java – Secondo ConcreteDecorator del pattern Decorator [3].

```
1  /*
2  La caratteristica particolare di un utente con il ruolo di cliente
       sono codificate nell obj Cliente. Questa classe estende le
       funzioni del Decorator, particolarmente aggiungendo...
3  */
4
5  /*
6  Il cliente deve poter scegliere i propri vestiti, aggiungerli ad un
       carrello e acquistarli tramite pagamento online.
7   */
8
9
10 public class Cliente extends Ruolo {
11
12    private int numItems = 0;
13
14    private Item[] cart; // Il carrello  definito come un array di
       oggetti Item
15
16    public Cliente(User userC) {
17       super(userC);
18       this.cart = new Item[10]; // Il carrello non pu avere pi di 10
        items
19    }
20
21    public void userInfo() {
22       sayIamClient();
23       super.userInfo();
24       System.out.println("Numero di items nel carrello: " +
```

```
    getNumItems());
  }

  private void sayIamClient() {
     System.out.print("Sono un utente cliente. ");
  }

  // Ridefinizione del metodo toString() come visto nel corso di
   Java base
  public String toString() {
     String s = "Carrello: ";

     if (numItems > 0) {
        s += "\n";
        int i = 0;
        int items_counter = 0;
        while (i < numItems) {
           items_counter += cart[i].getQuantity();
           s = s + cart[i].toString();
           ++i;
        }
        s += "Numero totale di items: " + items_counter + "\n";
     } else {
        s += "vuoto\n";
     }

     return s;
  }

  private int getNumItems() {
     return numItems;
  }

  private int findIndex(String n) {
     int i = 0;
     while (i < numItems) {
        if (cart[i].getName().equalsIgnoreCase(n)) return i;
        ++i;
     }

     return numItems;
  }

  private boolean chartIsFull() {
     return (numItems == cart.length);
  }

  /*
La responsabilit aggiuntiva principale assegnata a un cliente
  quella di poter aggiungere o togliere vestiti dal carrello.
*/
  public boolean addToChart(Magazzino m, String n, int q) {
     if (chartIsFull()) return false;
     if (m.isPresent(n)) {
        if (q < m.checkAvailability(n)) {
           cart[numItems] = m.sell(n, q);
```

```
78              ++numItems;
79              return true;
80           } else {
81              return false;
82           }
83        } else {
84           return false;
85        }
86     }
87
88     public boolean removeFromChart(String n) {
89        int i = findIndex(n);
90        if (i == numItems) return false;
91        --numItems;
92        cart[i] = cart[numItems];
93        return true;
94     }
95
96     /*
97    Un utente decorato con il ruolo di cliente puo anche ordinare la
       merce scelta pagando.
98    */
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 }
```

Con riferimento a list13 e a figura 2, la classe `Cliente` codifica la figure dell'utente cliente, dando la possibilitá all'utente cliente di aggiungere vestiti al carrello (`addToChart`), di rimuoverne (`removeFromChart`) e di pagare effettuando l'ordine (`pay`).

Si noti che i *ConcreteDecorators* definiti in questo microprototipo:

- estendono operazioni esistenti del *Component* (*e.g.* il metodo `userInfo`),
- estendono lo stato del Component (*e.g.* i campi `numItem`, `numItems`),
- definiscono nuove operazioni non condivise tra i *ConcreteDecorators* (*e.g.* i metodi `addToChart`, `removeFromChart`, `checkOrders`)

Per vedere il funzionamento del micro prototipo che utilizza il Decorator ho sviluppato altre tre classi, rispettivamente `Magazzino`, `Item`, `Ordine`, che integrate con le classi sopra presentate, simulano nel client (`DecoratorExample`) un esempio di scenario del caso d'uso preso in esame. Tale codice viene riportato per comodità in appendice A.

4.1.1 Comparativa Decorator vs Strategy

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

Cambiare la pelle di un oggetto, non le viscere. Possiamo pensare al *Decorator* come a una pelle che ricopre l'oggetto e che ne cambia il comportamento. Se desideriamo un cambiamento più viscerale, la scelta giusta è il **pattern Strategy**.

Rispetto al Decorator, il pattern Strategy è la scelta giusta quando la classe *Component* è contenutisticamente "pesante" (vedi commetti sopra), e quindi rende il pattern Decorator troppo costoso da applicare. Nel patter Strategy il *component* passa alcune delle sue caratteristiche a un oggetto *Strategy* separato [3]. La differenza principale rispetto il decorator è che ci permette di alterare/estendere le responsabilità andando a sostituire l'oggetto *strategy* (cfr.[7], p.201). Proseguire con riferimento al libro.

4.2 Pattern Strategy

Il *pattern Strategy* serve a definire una famiglia di algoritmi simili tra loro e dunque **intercambiabili**. In particolare, è tipica della situazione in cui vogliamo isolare l'algoritmo che svolge un certo compito, per farlo variare in modo indipendente dal resto dell'implementazione della classe.

Questo pattern suggerisce l'incapsulamento della logica di ogni particolare algoritmo, in apposite classi, dette *ConcreteStrategy*, le quali implementano l'interfaccia che consente agli oggetti di interagire con loro. Questa interfaccia deve fornire un accesso efficiente ai dati del Context, richiesti da ogni ConcreteStrategy, e viceversa [3].

Listing 6: User.java – Componente Context del pattern Strategy [3].

```
1 /*
2 Componente Context del pattern Strategy. Contiene la parte di
      algoritmo comune sia al cliente che al dipendente.
3  */
4
5 public class User {
6
7    public Dipendente dipendente;
8
```

```
9     private String username, email, pwd;
10
11    public User(String username, String email, String pwd) {
12       this.username = username;
13       this.email = email;
14       this.pwd = pwd;
15    }
16
17    public String getUsername() {
18       return username;
19    }
20
21    public String getEmail() {
22       return email;
23    }
24
25    public String getPwd() {
26       return pwd;
27    }
28
29    public String toString() {
30       return "username: " + this.username + ", email: " + this.email
        + ", pwd: " + pwd + "\n";
31    }
32
33    public void assegnaRuoloDipendente(Dipendente d) {
34       dipendente = d;
35    }
36
37    public void display() {
38       dipendente.userInfo(this);
39    }
40
41    public void checkOrders(Magazzino m) {
42       dipendente.checkOrders(m);
43    }
44 }
```

Listing 7: Dipendente.java – Prima componente Strategy del pattern Strategy [3].

```
1  /*
2  Primo componente Strategy del pattern Strategy.
3  */
4  public interface Dipendente {
5
6     public void assegnaRuoloCliente(Cliente c);
7
8     public void userInfo(User u);
9
10    public String toString();
11
12    public void display(User u);
13
14    public void checkOrders(Magazzino m);
15
```

```
16    public boolean addToChart(Magazzino m, String n, int q);
17
18    public boolean pay(User u, Magazzino m);
19 }
```

Listing 8: ConcreteDipendente.java – ConcreteStrategy della prima componente Strategy [3].

```
1  /*
2  Componente ConcreteStrategy della prima componente Strategy.
3   */
4  public class ConcreteDipendente implements Dipendente {
5
6     public Cliente cliente;
7
8     private String job;
9
10    public ConcreteDipendente(String job) {
11       this.job = job;
12    }
13
14    public String toString() {
15       return cliente.toString();
16    }
17
18    public String getJob() {
19       return job;
20    }
21
22
23    public void assegnaRuoloCliente(Cliente c) {
24       cliente = c;
25    }
26
27    public void userInfo(User u) {
28       System.out.print(u);
29       sayIamEmployee();
30    }
31
32    private void sayIamEmployee() {
33       System.out.print("Sono un utente dipendente con la mansione di
        " + job + ".\n");
34    }
35
36    public void display(User u) {
37       this.userInfo(u);
38       cliente.userInfo();
39    }
40
41    public void checkOrders(Magazzino m) {
42       if (m.hasOrders()) {
43          System.out.println(m.getOrders());
44       } else {
45          System.out.println("Nessun ordine presente.");
46       }
```

```
47    }
48
49    public boolean addToChart(Magazzino m, String n, int q) {
50       return cliente.addToChart(m, n, q);
51    }
52
53    public boolean pay(User u, Magazzino m) {
54       return cliente.pay(u, m);
55    }
56 }
```

Listing 9: Cliente.java – Seconda componente Strategy del pattern Strategy [3].

```
1  /*
2  Seconda componente Strategy del pattern Strategy.
3  */
4  public interface Cliente {
5
6     public void userInfo();
7
8     public String toString();
9
10    public boolean addToChart(Magazzino m, String n, int q);
11
12    public boolean pay(User u, Magazzino m);
13 }
```

Listing 10: ConcreteCliente.java – ConcreteStrategy della seconda componente Strategy [3].

```
1  /*
2  Componente ConcreteStrategy della seconda componente Strategy.
3   */
4  public class ConcreteCliente implements Cliente {
5
6     private int numItems = 0;
7
8     private Item[] cart; // Il carrello  definito come un array di
       oggetti Item
9
10    public ConcreteCliente() {
11       this.cart = new Item[10]; // Il carrello non pu avere pi di 10
        items
12    }
13
14    public void userInfo() {
15       System.out.print("Ora sono anche un utente cliente.\n");
16       System.out.println("Numero di items nel carrello: " +
       getNumItems());
17    }
18
19    public String toString() {
20       String s = "Carrello: ";
21
```

```
22      if (numItems > 0) {
23         s += "\n";
24         int i = 0;
25         int items_counter = 0;
26         while (i < numItems) {
27            items_counter += cart[i].getQuantity();
28            s = s + cart[i].toString();
29            ++i;
30         }
31         s += "Numero totale di items: " + items_counter + "\n";
32      } else {
33         s += "vuoto\n";
34      }
35
36      return s;
37   }
38
39   private int getNumItems() {
40      return numItems;
41   }
42
43   private int findIndex(String n) {
44      int i = 0;
45      while (i < numItems) {
46         if (cart[i].getName().equalsIgnoreCase(n)) return i;
47         ++i;
48      }
49
50      return numItems;
51   }
52
53   private boolean chartIsFull() {
54      return (numItems == cart.length);
55   }
56
57   /*
58  La responsabilit aggiuntiva principale assegnata a un cliente
      quella di poter aggiungere o togliere vestiti dal carrello.
59  */
60   public boolean addToChart(Magazzino m, String n, int q) {
61      if (chartIsFull()) return false;
62      if (m.isPresent(n)) {
63         if (q < m.checkAvailability(n)) {
64            cart[numItems] = m.sell(n, q);
65            ++numItems;
66            return true;
67         } else {
68            return false;
69         }
70      } else {
71         return false;
72      }
73   }
74
75   /*
76  Un utente decorato con il ruolo di cliente puo anche ordinare la
```

```
    merce scelta pagando.
77   */
78    public boolean pay(User u, Magazzino m) {
79       if (numItems == 0) {
80          System.out.println("Cestino vuoto, pagamento non andato a
     buon fine.");
81          return false;
82       } else {
83
84          int i = 0;
85          int bill = 0;
86          while (i < numItems) {
87             m.addOrder(u.getEmail(), cart[i].getName(), cart[i].
     getPrice(), cart[i].getQuantity());
88             bill += cart[i].getQuantity() * cart[i].getPrice();
89             ++i;
90          }
91
92          System.out.println("Saldo totale pagato: " + bill + "$");
93
94          this.numItems = 0;
95          this.cart = new Item[10];
96          return true;
97       }
98    }
99 }
```

Il resto del codice é disponibile in appendice B.

4.3 Pattern Composite

Questo pattern ci consente di costruire una gerarchie di oggetti composti. Gli oggetti composti possono essere formati da oggetti singoli, oppure da altri oggetti composti. è utile nei casi in cui si vuole rappresentare gerarchie di oggetti tutto-parte, essere in grado di ignorare le differenze tra oggetti singoli e oggetti composti [7]. Il fatto che l'*Atelier Splendor* mostra il desiderio di vendere articoli di altri brand abbiamo visto molto opportuno proporre il composite perché nell'architettura potrá aggiungere gli articoli suoi e altri brand in modo facile e molto separato.

Il pattern viene usato nel nostro caso perché si può rappresentare il nostro magazzino come segue [3]:

- Un *Item* (è una classe astratta che fornisce le operazione base del nostro magazzino/articolo)
- Un *CompoundItem* (La classe estende Item e si comporta sia come una parte del magazzino o come il magazzino intero a secondo di come è usato)
- Un *SaleableItem* (Questa classe è l'articolo proprio, estende Item e ha un costruttore piu completo di CompoundItem anh)
- Un *SaleableItemException* (Per gestire un' *exception* quando l'utente prova ad aggiungere in un *SaleableItem* un *Item*)

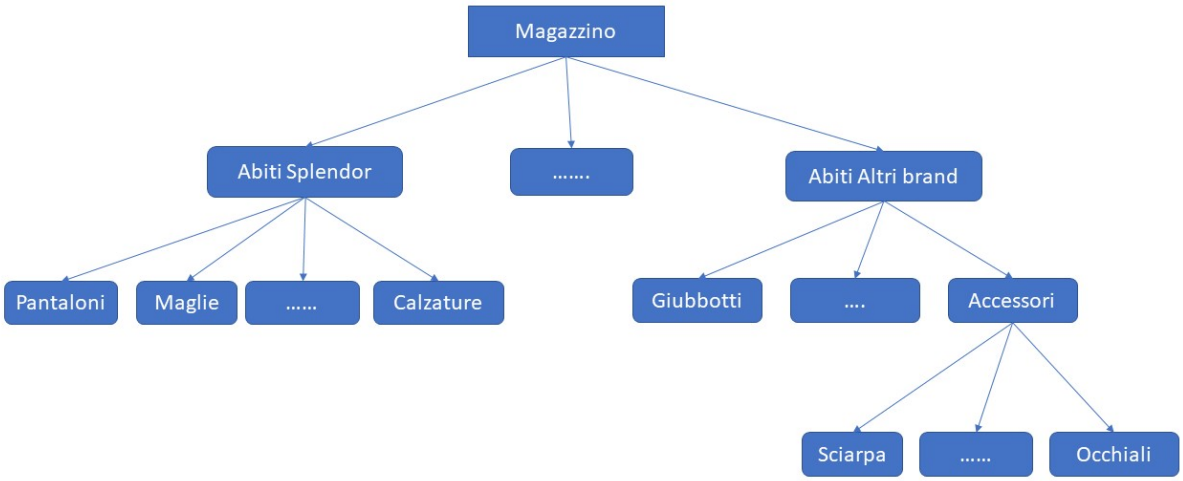


Figura 3: In figura è mostrata la struttura gerarchica delle merce in vendita presso lo shop online dell'*Atelier Splendor*.

```
// The Item abstract class
public abstract class Item {
    protected String name;

    public Item(String name) {
        this.name = name;
    }

    public abstract void describe();

    public void addItem(Item c) throws SaleableItemException {
        if (this instanceof SaleableItem)
            throw new SaleableItemException();
    }

    public void removeItem(Item c) throws SaleableItemException {
        if (this instanceof SaleableItem)
            throw new SaleableItemException();
    }

    public Item getItem(int n) {
        return null;
    }
}

//CompoundItem class
public class CompoundItem extends Item {
    private ArrayList<Item> items;

```

```
30      public CompoundItem(String name) {
31          super(name);
32          items = new ArrayList<Item>();
33      }
34
35      @Override
36      public void describe() {
37          System.out.println(name);
38          System.out.println("\tComposed by:");
39          System.out.println("\t\t{");
40          items.forEach(Item::describe);
41          System.out.println("}");
42      }
43
44      public void addItem(Item c) throws SaleableItemException {
45          items.add(c);
46      }
47
48      public void removeItem(Item c) throws SaleableItemException {
49          items.remove(c);
50      }
51
52      public Item getItem(int n) {
53          if (n<0 || n>items.size()) return null;
54          return (Item) items.get(n);
55      }
56  }
57
58  // SaleableItem class
59  public class SaleableItem extends Item{
60      private double weight;
61      private String type;
62      private double price;
63      private String size;
64
65      public SaleableItem(String name,String type, double weight,
        double price,String size) {
66          super(name);
67          this.weight = weight;
68          this.price = price;
69          this.type = type;
70          this.size = size;
71      }
72
73      @Override
74      public void describe() {
75           System.out.println( "SaleableItem: " + name  +", type: "+
        type +", weight: "+weight +"kg"+", price: "+price +"" +", size:"
         + size);
76
77      }
78  }
79
80  // SaleableItemException class
81  public class SaleableItemException extends Exception {
82      public SaleableItemException() {
```

```
83          super("Not supported method, you cant add Item on
        SaleableItem");
84      }
85  }
```

Come detto prima, la classe *CompoundItem* può rappresentare l'intero magazzino o semplicemente una parte (categoria) di alcuni articoli. La classe *CompoundItem* è usata per rappresentare tutti gli articoli dell'*Atelier Splenldor* e quelli di altri brand che il loro insieme forma tutto il magazzino che è sempre rappresenatato dalla classe *CompoundItem* [3].

La classe *SaleableItem* rappresenta l'articolo venduto con tutto le caratteristiche possibile, ogni volta che si vuole aggiungere un articolo (*SaleableItem*) si chiama la `addItem` sul Magazzino (*CompoundItem*).

Nota bene: chiamando la `addItem` o `removeItem` su *SaleableItem* lancerà la *SaleableItemException* [7].

Il resto del codice é disponibile in appendice C.

4.4 Pattern Observer

Il pattern *Observer* (noto anche come *Publish-Subscribe*) é un pattern che "*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*" (cfr.[7], p.326).

L'adozione di questo pattern trova le sue radici nella necessitá di trovare una *consistenza* tra oggetti appartenenti a classi che *cooperano* tra loro; una soluzione che sarebbe possibile adottare sarebbe quella di rendere queste classi strettamente dipendenti le une dalle altre, ma questo ridurrebbe la loro riusabilitá. La *Gang of Four* cita come esempio la situazione in cui una diversa rappresentazione grafica (un grafico a barre, un grafico a torta e una rappresentazione tabellare) prendano le informazioni circa i dati da rappresentare dal medesimo *data object*. Queste rappresentazioni grafiche non comunicano direttamente tra loro, ma devono comportarsi come se lo potessero fare. Quindi, nel momento in cui l'utente cambia i dati, questo cambiamento deve riflettersi in tutte le rappresentazioni grafiche di cui sopra. (cfr.[7], p.327)

Questo comportamento implica quindi, accettando il fatto di non voler creare dipendenze tra questi tre oggetti, che essi vengano notificati e coerentemente aggiornati rispetto a modifiche ed aggiornamenti che si verificano per il *data object* soggiacente che li popola. Il pattern Observer viene incontro a questa esigenza.

Nel nostro caso specifico, essendoci immaginati, oltre allo store on-line per la vendita dei nostri prodotti, anche l'allestimento di PopUp-Store temporanei in diverse cittá, l'idea che gli utenti iscritti al nostro sito per effettuare acquisti potessero essere interessati ad aggiornamenti relativi alla nostra presenza fisica sul territorio, per poter toccare direttamente con mano i capi che forniamo loro, ha portato alla presa di coscienza della necessitá di allestire una *newsletter* che li potesse aggiornare in tal senso. Essi saranno aggiornati rispetto ad informazioni chiave quali la cittá ed il negozio entro cui il PopUp-Store verrá allestito, nonché gli orari.

Procediamo ora con il commento del codice utilizzato.
L'implementazione prevede la creazione di:

- Un *subject* (fornisce un'interfaccia che permette di aggiungere, rimuovere e notificare gli osservatori)
- Un *concreteSubject* (notifica gli osservatori circa i cambiamenti del proprio stato e modifica quest'ultimo all'occorrenza)
- Un (...n) *observer* (fornisce un'interfaccia per aggiornare gli osservatori rispetto ai cambi di stato del soggetto)
- Un (...n) *concreteObserver* (viene informato dei cambi di stato avvenuti sul concreteSubject e usa le informazioni ottenute per aggiornarsi appropriatamente rispetto ad esso)

```
1  // The Subject interface
2  interface Subject {
3      void registerObserver(Observer o);
4      void removeObserver(Observer o);
5      void notifyObservers();
6  }
7
8  // The Observer interface
9  interface Observer {
10     void update(String location, String store, Date openingdate,
       Date closingdate, double openingtime, double closingtime);
11 }
```

Come abbiamo detto precedentemente, queste due interfacce permettono, rispettivamente:

- SI:
  - aggiungere un osservatore
  - rimuovere un osservatore
  - notificare agli osservatori i cambi di stato
- OI:
  - essere aggiornati rispetto ai cambi di stato del soggetto

Ribadiamo che abbiamo definito i seguenti cambi rispetto ai quali i clienti dovranno essere aggiornati:

- cittá entro cui si svolgerá il PopUp-Store
- il negozio che lo ospiterá
- la data di apertura e quella di chiusura
- l'orario entro cui esso si potrá trovare all'interno del negozio sopracitato

Possiamo ora passare al *ConcreteSubject* e al *ConcreteObserver*:

```
1
2  // The PopUpStore class is the Subject
3  class PopUpStore implements Subject {
4
```

```
    private List<Observer> observers;
    private String location;
    private String store;
    private Date openingdate;
    private Date closingdate;
    private double openingtime;
    private double closingtime;

    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();
    }
}


// The Observer class
class CurrentConditionsDisplay implements Observer, DisplayElement {
    private String location;
    private String store;
    private Date openingdate;
    private Date closingdate;
    private double openingtime;
    private double closingtime;
```

```
58      private Subject PopUpStore;
59
60      SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/dd");
61      String formattedOpeningDate = sdf.format(openingdate);
62      String formattedClosingDate = sdf.format(closingdate);
63
64      public CurrentConditionsDisplay(Subject PopUpStore) {
65          this.PopUpStore = PopUpStore;
66          PopUpStore.registerObserver(this);
67      }
68
69      public void update(String location, String store, Date
        openingdate, Date closingdate, double openingtime, double
        closingtime) {
70          this.location = location;
71          this.store = store;
72          this.openingdate = openingdate;
73          this.closingdate = closingdate;
74          this.openingtime = openingtime;
75          this.closingtime = closingtime;
76          display();
77      }
78
79      public void display() {
80      System.out.println("We look forward to seeing you at our new Pop
        -Up store in " + location + ", kindly hosted by  " + store +
81      ", on the following dates: " + formattedOpeningDate + " - " +
        formattedClosingDate + ", at the following times:" + openingtime
         + " - " + closingtime);
82      }
83  }
```

Commentando questa parte di codice, notiamo che il *ConcreteSubject* é rappresentato dal PopUp-Store che, come anticipato, puó aggiungere, rimuovere e notificare gli *Observers*, andando ad aggiornare i dati relativi ai campi che definiscono il nostro PopUp-Store (tramite la comunicazione diretta con l'interfaccia *Observer*). Vi é poi un metodo per settare questi dati.

Passando invece al *ConcreteObserver*, vediamo che esso ha modo di registrarsi al *Subject* di nostro interesse, di aggiornare le informazioni a sua disposizione relativamente ai campi che definiscono il PopUp-Store, nonché averle a disposizione tramite il suo metodo *display*.

Il resto del codice é disponibile in appendice D.

Riferimenti bibliografici

[1] F. Landorno. *Requirements elicitation tool*. Tesi di laurea triennale. Università degli Studi di Torino, 2018.

[2] Craig Larman. *Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, 2004.

[3] F. Guidi Polanco. *Dispense sui design pattern GoF in Java*. Politecnico di Torino, 2002.

[4] Bouka Marietti Balbo Mossetto e Pastori. *GitHub repository for the Design Pattenrs exam project*. URL: https://github.com/Balbone/ITFS---Design-Pattern-Exam.

[5] AWS Team. *What is an API*. 2022. URL: https://aws.amazon.com/it/what-is/api/.

[6] Oracle Team. *What is a Database*. 2022. URL: https://www.oracle.com/database/what-is-database/.

[7] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1ª ed. Addison-Wesley Professional, 1994.

[8] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition)*. 3ª ed. MR: UML(2) schnell eingeführt. Addison-Wesley Professional, 2003.

A Decorator

In appendice presento le altre classi necessarie per far funzionare il micro prototipo del decorator applicato al caso d'uso dell'acquisto di vestiti sullo store online (rispettivamente `Magazzino`, `Item`, `Ordine`). L'output in console del client (`DecoratorExample`) è presentato in figura 4.

Listing 11: Magazzino.java – Classe che astrae il concetto di magazzino dell'*Atelier Splendor*.

```
1  public class Magazzino {
2
3     private int numItemsMag;
4
5     private int numItemsOrd;
6
7     private Item[] magazzino; // Un magazzino contiene Items
       sottoforma di array di objs
8
9     private Ordine[] ordini; // Un magazzino contiene gli ordini
       sottoforma di array di Ordini
10
11    public Magazzino() {
12       numItemsMag = 0;
13       magazzino = new Item[20];
14       ordini = new Ordine[20];
15    }
16
17    public String toString() {
18       int i = 0;
19       int counter = 0;
20
21       String s = "Magazzino:\n";
22
23       while (i < numItemsMag) {
24          s = s + magazzino[i].toString();
25          counter = counter + magazzino[i].getQuantity();
26          ++i;
27       }
28       s = s + "Numero totale di items presenti: " + counter;
29
30       return s;
31    }
32
33    private int findIndex(String n) {
34       int i = 0;
35
36       while (i < numItemsMag) {
37          if (magazzino[i].getName().equalsIgnoreCase(n)) return i;
38          ++i;
39       }
40
41       return numItemsMag;
42    }
43
```

```
44    public boolean isPresent(String n) {
45       return (findIndex(n) < numItemsMag);
46    }
47
48
49    public int checkAvailability(String n) {
50       int i = findIndex(n);
51       if (i < numItemsMag) return magazzino[i].getQuantity();
52       else return 0;
53    }
54
55    // Inizializzo il magazzino con una serie di oggetti che contiene
        nizialmente
56    public boolean initMagazzino(String n, int p, int q) {
57       magazzino[numItemsMag] = new Item(n, p, q);
58       ++numItemsMag;
59       return true;
60    }
61
62    public Item sell(String n, int q) {
63       int i = 0;
64
65       while (i < numItemsMag) {
66          if (magazzino[i].getName().equalsIgnoreCase(n)) break;
67          ++i;
68       }
69
70       int old_quantity = magazzino[i].getQuantity();
71       magazzino[i].setQuantity(old_quantity - q);
72
73       return new Item(n, magazzino[i].getPrice(), q);
74    }
75
76    public boolean addOrder(String e, String n, int p, int q) {
77       ordini[numItemsOrd] = new Ordine(e, n, p, q);
78       ++numItemsOrd;
79       return true;
80    }
81
82    public boolean hasOrders() {
83       return (numItemsOrd > 0);
84    }
85
86    public String getOrders() {
87       int i = 0;
88       int counter = 0;
89
90       String s = "Ordini in attesa di essere evasi:\n";
91
92       while (i < numItemsOrd) {
93          s = s + ordini[i].toString();
94          counter = counter + ordini[i].getQuantity();
95          ++i;
96       }
97       s = s + "Numero totale di items da spedire: " + counter;
98
```

```
99          return s;
100      }
101  }
```

Listing 12: Item.java – Classe che astrae il concetto di item venduto dall'*Atelier Splendor*.

```
1  public class Item {
2
3      private String name;
4
5      private int price, quantity;
6
7      public Item(String name, int price, int quantity) {
8          this.name = name;
9          this.price = price;
10         this.quantity = quantity;
11     }
12
13     public String getName() {
14         return name;
15     }
16
17     public int getPrice() {
18         return price;
19     }
20
21     public int getQuantity() {
22         return quantity;
23     }
24
25     public void setQuantity(int quantity) {
26         this.quantity = quantity;
27     }
28
29     public String toString() {
30         return "item: " + name + ", price: " + price + "$, quantity: "
       + quantity + "\n";
31     }
32 }
```

Listing 13: Ordine.java – Classe che astrae il concetto di ordine effettuato da un cliente dell'*Atelier Splendor*.

```
1  public class Ordine extends Item {
2
3      private String email;
4
5      public Ordine(String email, String name, int price, int quantity)
       {
6          super(name, price, quantity);
7          this.email = email;
8      }
9
```

```
10    public String toString() {
11       return "cliente: " + email + ", " + super.toString();
12    }
13 }
```

Listing 14: DecoratorExample.java – Classe client che contine il main del microprototipo che utilizza il pattern Decorator.

```
1  /*
2  Client per testare il micro prototipo.
3  */
4
5  public class DecoratorExample {
6     public static void main(String arg[]) {
7
8        // Inizializzo il magazzino
9        Magazzino magazzinoSplendor = new Magazzino();
10       System.out.println("Il magazzino dell'Atelier Splendor  vuoto.
      ");
11       System.out.println(magazzinoSplendor);
12
13       System.out.println("\nIl magazzino dell'Atelier Splendor viene
       riempito la sua nuova collezione autunno-inverno.");
14       magazzinoSplendor.initMagazzino("t-shirt", 33, 30);
15       magazzinoSplendor.initMagazzino("polo", 67, 20);
16       magazzinoSplendor.initMagazzino("camicia", 90, 40);
17       magazzinoSplendor.initMagazzino("giacca", 256, 10);
18       magazzinoSplendor.initMagazzino("jeans", 80, 65);
19
20       System.out.println(magazzinoSplendor);
21
22
23       // Inizializzo un utente cliente
24
25       //System.out.println("SIGN UP per accedere all'applicazione
      dell'Atelier Splendor.");
26       User utente1 = new UtenteBase("marco.rossi", "marco.rossi@me.
      it", "pollo123");
27       //marco.login();
28
29       User marco = new Cliente(utente1);
30       System.out.println("\nInformazioni sull'utente cliente:");
31       marco.userInfo();
32
33       // Marco aggiunge i seguenti items al suo cestino
34       marco.addToChart(magazzinoSplendor, "t-shirt", 3);
35       marco.addToChart(magazzinoSplendor, "giacca", 1);
36       marco.addToChart(magazzinoSplendor, "jeans", 1);
37       marco.addToChart(magazzinoSplendor, "camicia", 2);
38
39       System.out.println("\nL'utente cliente Marco aggiunge i
      seguenti items al carrello:");
40       System.out.println(marco);
41
42       System.out.println("L'utente cliente Marco effettua l'ordine."
```

```
    );
43      marco.pay(magazzinoSplendor);
44
45      System.out.println("\nCestino degli acquisti dell'utente
    cliente Marco dopo l'acquisto:");
46      System.out.println(marco);
47      System.out.println("Il magazzino dell'Atelier Splendor dopo l'
    acquisto dell'utente cliente Marco:");
48      System.out.println(magazzinoSplendor);
49
50
51      // Inizializzo un utente dipendente
52
53      User utente2 = new UtenteBase("chiara.genovese", "chiara.
    genovese@me.it", "querty99");
54      //chiara.userInfo();
55
56      User chiara = new Dipendente(utente2, "sarta");
57      //chiara.assignJob();
58      System.out.println("\nInformazioni sull'utente dipendente:");
59      chiara.userInfo();
60
61      System.out.println("\nL'utente dipendente Chiara controlla gli
     ordini da evadere.");
62      chiara.checkOrders(magazzinoSplendor);
63
64
65      // L'utente dipendente Chiara diventa anche un cliente dell'
    Atelier Splendor
66
67      System.out.println("\nL'utente dipendente Chiara diventa anche
     un cliente dell'Atelier Splendor.");
68      User chiara_cliente = new Cliente(chiara);
69      System.out.println("Informazioni sull'utente dipendente-
    cliente:");
70      chiara_cliente.userInfo();
71
72      // Chiara aggiunge i seguenti items al suo cestino
73      chiara_cliente.addToChart(magazzinoSplendor, "t-shirt", 1);
74      chiara_cliente.addToChart(magazzinoSplendor, "giacca", 2);
75
76      System.out.println("\nL'utente cliente-dipendente Chiara
    aggiunge i seguenti items al carrello:");
77      System.out.println(chiara_cliente);
78
79      System.out.println("L'utente cliente-dipendente Chiara
    effettua l'ordine ed  lei stessa che controlla gli ordini da
    evadere.");
80      chiara_cliente.pay(magazzinoSplendor);
81      chiara_cliente.checkOrders(magazzinoSplendor);
82    }
83 }
```

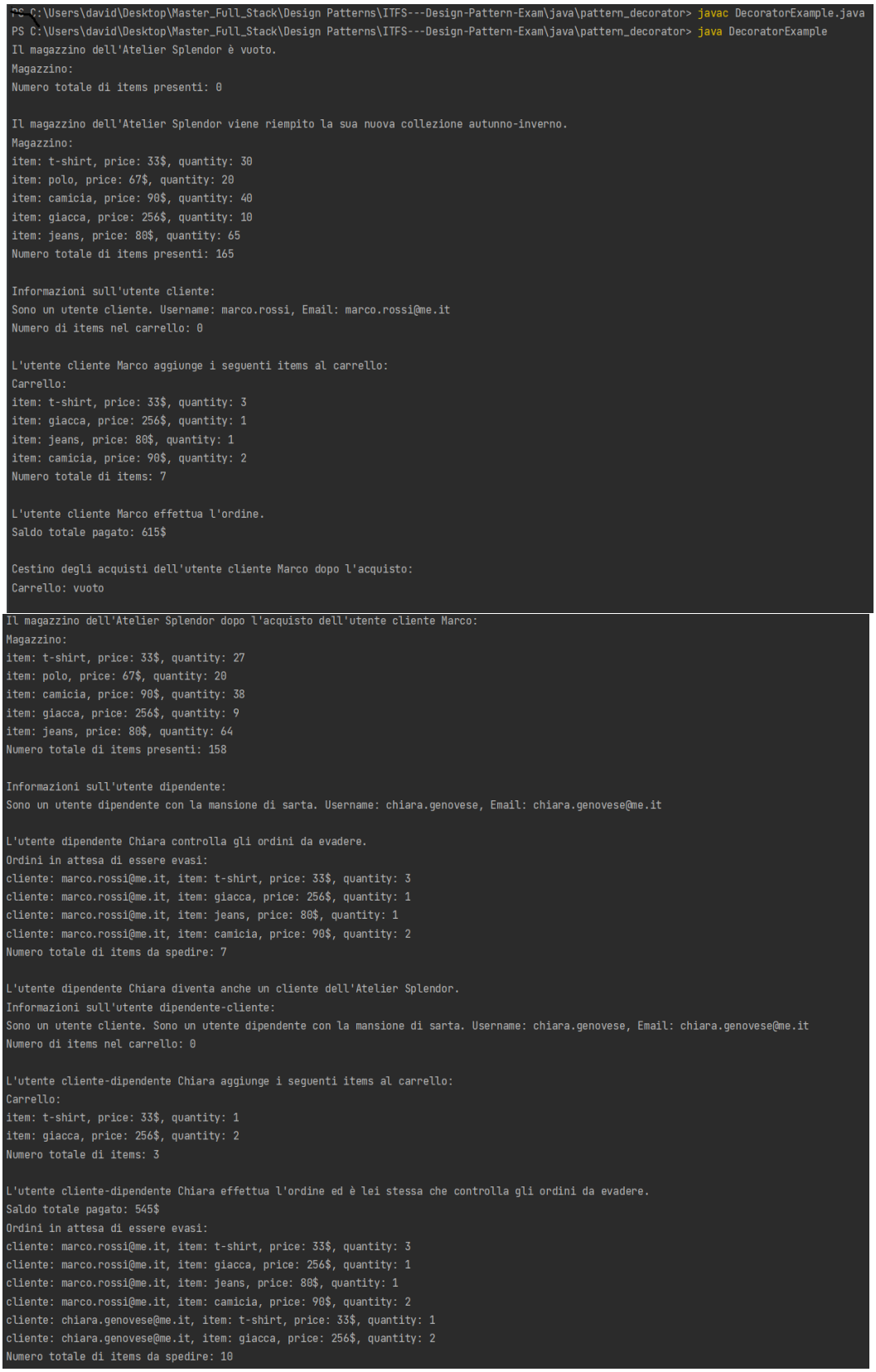


Figura 4: Output del micro prototipo che utilizza il pattern Decorator.

B Strategy

In appendice l'output in console del client (`StartegyExample`) è presentato in figura 5. Ho scelto di utilizzare le stesse classi di test sviluppate per il decorator (le classi `Magazzino`, `Item`, `Ordine` riportate in appendice A) proprio per evidenziare al massimo le differenze tra i due diversi approcci allo stesso caso d'uso.

Listing 15: StrategyExample.java – Classe client che contine il main del micro prototipo che utilizza il pattern Strategy.

```
1  public class StrategyExample {
2     public static void main(String[] arg) {
3
4        // Inizializzo il magazzino
5        Magazzino magazzinoSplendor = new Magazzino();
6        System.out.println("Il magazzino dell'Atelier Splendor  vuoto.
       ");
7        System.out.println(magazzinoSplendor);
8
9        System.out.println("\nIl magazzino dell'Atelier Splendor viene
        riempito la sua nuova collezione autunno-inverno.");
10       magazzinoSplendor.initMagazzino("t-shirt", 33, 30);
11       magazzinoSplendor.initMagazzino("polo", 67, 20);
12       magazzinoSplendor.initMagazzino("camicia", 90, 40);
13       magazzinoSplendor.initMagazzino("giacca", 256, 10);
14       magazzinoSplendor.initMagazzino("jeans", 80, 65);
15
16       System.out.println(magazzinoSplendor);
17
18       // Inizializzo l'utente
19       User chiara = new User("chiara.genovese", "chiara.genovese@me.
       it", "qwerty99");
20
21       // Ci assegno il ruolo di dipendente
22       System.out.println("\nDefinisco un utente Chiara con il ruolo
       di dipendente.");
23       Dipendente chiara_dipendente = new ConcreteDipendente("sarta")
       ;
24       chiara.assegnaRuoloDipendente(chiara_dipendente);
25       chiara.display();
26
27       System.out.println("\nControllo se ci sono ordini da evadere
       nel magazzino:");
28       chiara.checkOrders(magazzinoSplendor);
29
30       // Ci assegno il anche ruolo di cliente
31       System.out.println("\nAssegno a Chiara anche il ruolo di
       cliente.");
32       Cliente chiara_cliente = new ConcreteCliente();
33       chiara_dipendente.assegnaRuoloCliente(chiara_cliente);
34       chiara_dipendente.display(chiara);
35
36       // Chiara aggiunge i seguenti items al suo cestino
37       chiara_dipendente.addToChart(magazzinoSplendor, "t-shirt", 3);
```

```
38        chiara_dipendente.addToChart(magazzinoSplendor, "giacca", 1);
39        chiara_dipendente.addToChart(magazzinoSplendor, "jeans", 1);
40        chiara_dipendente.addToChart(magazzinoSplendor, "camicia", 2);
41
42        System.out.println("\nChiara aggiunge i seguenti items al
      carrello:");
43        System.out.println(chiara_dipendente);
44
45        // Chiara effettua l'ordine
46        System.out.println("Chiara effettua l'ordine.");
47        chiara_dipendente.pay(chiara, magazzinoSplendor);
48
49        System.out.println("\nCestino degli acquisti di Chiara dopo l'
      acquisto:");
50        System.out.println(chiara_dipendente);
51
52        System.out.println("Il magazzino dell'Atelier Splendor dopo l'
      acquisto di Chiara:");
53        System.out.println(magazzinoSplendor);
54
55        // Chiara stessa controlla gli ordini
56        System.out.println("\nChiara ricontrolla gli ordini da evadere
       dopo il suo acquisto.");
57        chiara.checkOrders(magazzinoSplendor);
58    }
59 }
```

```
PS C:\Users\david\Desktop\Master_Full_Stack\Design Patterns\ITFS---Design-Pattern-Exam\java\patter_strategy> javac StrategyExample.java
PS C:\Users\david\Desktop\Master_Full_Stack\Design Patterns\ITFS---Design-Pattern-Exam\java\patter_strategy> java StrategyExample
Il magazzino dell'Atelier Splendor è vuoto.
Magazzino:
Numero totale di items presenti: 0

Il magazzino dell'Atelier Splendor viene riempito la sua nuova collezione autunno-inverno.
Magazzino:
item: t-shirt, price: 33$, quantity: 30
item: polo, price: 67$, quantity: 20
item: camicia, price: 90$, quantity: 40
item: giacca, price: 256$, quantity: 10
item: jeans, price: 80$, quantity: 65
Numero totale di items presenti: 165
item: jeans, price: 80$, quantity: 1
item: camicia, price: 90$, quantity: 2
Numero totale di items: 7

Chiara effettua l'ordine.
Saldo totale pagato: 615$

Cestino degli acquisti di Chiara dopo l'acquisto:
Carrello: vuoto

Il magazzino dell'Atelier Splendor dopo l'acquisto di Chiara:
Magazzino:
item: t-shirt, price: 33$, quantity: 27
item: polo, price: 67$, quantity: 20
item: camicia, price: 90$, quantity: 38
item: giacca, price: 256$, quantity: 9
item: jeans, price: 80$, quantity: 64
Numero totale di items presenti: 158

Chiara ricontrolla gli ordini da evadere dopo il suo acquisto.
Ordini in attesa di essere evasi:
cliente: chiara.genovese@me.it, item: t-shirt, price: 33$, quantity: 3
cliente: chiara.genovese@me.it, item: giacca, price: 256$, quantity: 1
cliente: chiara.genovese@me.it, item: jeans, price: 80$, quantity: 1
cliente: chiara.genovese@me.it, item: camicia, price: 90$, quantity: 2
Numero totale di items da spedire: 7
```

Figura 5: Output del micro prototipo che utilizza il pattern Strategy.

C Composite

In appendice l'output in console del client (`Main`) è presentato in figura 6.

Listing 16: CompoundItem.java

```java
public class CompoundItem extends Item {
  private ArrayList<Item> items;

  public CompoundItem(String name) {
    super(name);
    items = new ArrayList<Item>();
  }

  @Override
  public void describe() {
    System.out.println(name);
    System.out.println("\tComposed by:");
    System.out.println("\t\t{");
    items.forEach(Item::describe);
    System.out.println("}");
  }

```

```
18    public void addItem(Item c) throws SaleableItemException {
19      items.add(c);
20    }
21
22    public void removeItem(Item c) throws SaleableItemException {
23      items.remove(c);
24    }
25
26    public Item getItem(int n) {
27      if (n<0 || n>items.size()) return null;
28      return (Item) items.get(n);
29    }
30
31    public Item getItem(String name) {
32      List<Item> tmp= items.stream().filter(x->x.name.equals(name)).
        collect(Collectors.toList());
33      return tmp.size() == 0 ? null : tmp.get(0);
34    }
35  }
```

Listing 17: Item.java

```
1  public abstract class Item {
2    protected String name;
3
4    public Item(String name) {
5      this.name = name;
6    }
7
8    public abstract void describe();
9
10   public void addItem(Item c) throws SaleableItemException {
11     if (this instanceof SaleableItem)
12       throw new SaleableItemException();
13   }
14
15   public void removeItem(Item c) throws SaleableItemException {
16     if (this instanceof SaleableItem)
17       throw new SaleableItemException();
18   }
19
20   public Item getItem(int n) {
21     return null;
22   }
23 }
```

Listing 18: SaleableItem.java

```
1  public class SaleableItem extends Item{
2    private double weight;
3    private String type;
4    private double price;
5    private String size;
6    private int quantity;
```

```
7
8   public SaleableItem(String name,String type, double weight, double
        price,String size,int quantity) {
9     super(name);
10    this.weight = weight;
11    this.price = price;
12    this.type = type;
13    this.size = size;
14    this.quantity=quantity;
15  }
16
17  @Override
18  public void describe() {
19     System.out.println( "SaleableItem: " + name  +", type: "+type +
      ", weight: "+weight +"kg"+", price: "+price +"" +", size:" +
      size);
20
21  }
22
23  public double getWeight() {
24    return weight;
25  }
26
27  public void setWeight(double weight) {
28    this.weight = weight;
29  }
30
31  public String getType() {
32    return type;
33  }
34
35  public void setType(String type) {
36    this.type = type;
37  }
38
39  public double getPrice() {
40    return price;
41  }
42
43  public void setPrice(double price) {
44    this.price = price;
45  }
46
47  public String getSize() {
48    return size;
49  }
50
51  public String getName() {
52    return super.name;
53  }
54
55  public void setSize(String size) {
56    this.size = size;
57  }
58
59  public int getQuantity() {
```

```
60      return quantity;
61    }
62
63    public void setQuantity(int quantity) {
64      this.quantity = quantity;
65    }
66
67  }
```

Listing 19: SaleableItemException.java

```
1  public class SaleableItemException extends Exception {
2
3    private static final long serialVersionUID = 1L;
4
5    public SaleableItemException() {
6      super("Not supported method, you can't add Item on SaleableItem"
       );
7    }
8  }
```

Listing 20: Main.java

```
1  public class Main {
2
3    public static void main(String[] args) {
4
5      CompoundItem store = new CompoundItem("Atelier Splendor");
6
7      CompoundItem splendorCreation = new CompoundItem("
       splendorCreation");
8
9      CompoundItem otherBrand = new CompoundItem("OtherBrand");
10
11     CompoundItem accessories = new CompoundItem("Accessories");
12
13     try {
14
15       store.addItem(splendorCreation);
16       store.addItem(otherBrand);
17       otherBrand.addItem(accessories);
18     } catch (SaleableItemException e) {
19
20       e.printStackTrace();
21     }
22
23
24     SaleableItem pantalone = new SaleableItem("pantalone","vestito"
       ,0.5,12.00,"42",30);
25     SaleableItem maglia = new SaleableItem("maglia","vestito"
       ,0.3,14.00,"XS",2);
26     SaleableItem cintura = new SaleableItem("cintura","accessories"
       ,0.1,5.00,"1m",4);
```

```
27      SaleableItem giubbotto = new SaleableItem("giubbotto","vestito"
        ,2,75.00,"S",6);
28      SaleableItem stivale = new SaleableItem("stivale","calzature"
        ,2,25.00,"42",16);
29
30      SaleableItem sciarpa = new SaleableItem("sciarpa","accessories"
        ,0.1,5.00,null,5);
31
32      try {
33        splendorCreation.addItem(pantalone);
34        splendorCreation.addItem(maglia);
35        splendorCreation.addItem(stivale);
36
37        otherBrand.addItem(giubbotto);
38        accessories.addItem(cintura);
39        accessories.addItem(sciarpa);
40
41
42      } catch (SaleableItemException e) {
43
44        e.printStackTrace();
45      }
46
47      store.describe();
48
49
50
51    }
52
53  }
```

```
Atelier Splendor
        Composed by:
                {
splendorCreation
        Composed by:
                {
SaleableItem: pantalone, type: vestito, weight: 0.5kg, price: 12.0€, size:42
SaleableItem: maglia, type: vestito, weight: 0.3kg, price: 14.0€, size:XS
SaleableItem: stivale, type: calzature, weight: 2.0kg, price: 25.0€, size:42
}
OtherBrand
        Composed by:
                {
Accessories
        Composed by:
                {
SaleableItem: cintura, type: accessories, weight: 0.1kg, price: 5.0€, size:1m
SaleableItem: sciarpa, type: accessories, weight: 0.1kg, price: 5.0€, size:null
}
SaleableItem: giubbotto, type: vestito, weight: 2.0kg, price: 75.0€, size:S
}
}
```

Figura 6: Output del micro prototipo che utilizza il pattern Composite.

D Observer

```
  import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.text.ParseException;


// 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);
}

// The DisplayElement interface just has one method, display, that
    we will call when the display element
// needs to be displayed.
interface DisplayElement {
    void display();
}

// The PopUpStore class is the Subject
class PopUpStore implements Subject {

    private List<Observer> observers;
    private String location;
    private String store;
    private Date openingdate;
    private Date closingdate;
    private double openingtime;
    private double closingtime;

    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);
        }
    }
```

```
51
52      public void notifyObservers() {
53          for (Observer observer : observers) {
54              observer.update(location, store, openingdate,
        closingdate, openingtime, closingtime);
55          }
56      }
57
58      public void measurementsChanged() {
59          notifyObservers();
60      }
61
62      public void setMeasurements(String location, String store, Date
        openingdate, Date closingdate, double openingtime, double
        closingtime) {
63          this.location = location;
64          this.store = store;
65          this.openingdate = openingdate;
66          this.closingdate = closingdate;
67          this.openingtime = openingtime;
68          this.closingtime = closingtime;
69          measurementsChanged();
70      }
71  }
72
73  // The Observer class
74  class CurrentConditionsDisplay implements Observer, DisplayElement {
75      private String location;
76      private String store;
77      private Date openingdate;
78      private Date closingdate;
79      private double openingtime;
80      private double closingtime;
81      private Subject PopUpStore;
82
83
84
85      public CurrentConditionsDisplay(Subject PopUpStore) {
86          this.PopUpStore = PopUpStore;
87          PopUpStore.registerObserver(this);
88      }
89
90      public void update(String location, String store, Date
        openingdate, Date closingdate, double openingtime, double
        closingtime) {
91          this.location = location;
92          this.store = store;
93          this.openingdate = openingdate;
94          this.closingdate = closingdate;
95          this.openingtime = openingtime;
96          this.closingtime = closingtime;
97          display();
98      }
99
100     public void display() {
101
```

```
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/dd");
        String formattedOpeningDate = sdf.format(openingdate);
        String formattedClosingDate = sdf.format(closingdate);

        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);
    }
}



// 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();
        }
    }
}
```

Viene allegato l'output in console:

Figura 7: Output della console relativo all'implementazione del Pattern Observer