UNIVERSITÀ DEGLI STUDI DI NAPOLI PARTHENOPE

DIPARTIMENTO DI SCIENZE E TECNOLOGIE
CORSO DI PROGRAMMAZIONE 3 E LABORATORIO DI PROGRAMMAZIONE 3

DISTRIBUTORE BEVANDE

CATEGORIA: GESTIONALE

CODICE GRUPPO: v3bgp3z697e

DOCENTE

Angelo Ciaramella

CANDIDATI

Viscillo Nicola 0124002557
Galiero Nicola 0124002671

Anno accademico 2023 - 2024

Indice

Introduzione

Traccia del progetto

Si vuole simulare un distributore automatico di Bevande. Il distributore ha 6 tipologie di bevande: caffè, thè, latte, camomilla, cioccolata e acqua calda. Per ogni tipologia sono previste delle sottotipologie (e.g., caffè e latte, latte e caffè, caffè e cioccolato, thè e latte, . . .). L'utente può scegliere anche la quantità di zucchero da erogare. Scrivere un programma per la gestione del distributore. L'accesso deve avvenire sia in modalità amministratore che in modalità utente.

L'amministratore può effettuare le seguenti operazioni:

- periodicamente aggiungere bevande alla scorta. Il sistema controlla automaticamente se la bevanda è sotto scorta (minore di 1 litro)
- definire il prezzo per ogni tipo di bevanda
- fare un report sui consumi mensili delle diverse tipologie di bevande
- aggiungere una nuova tipologia di bevanda partendo da quelle già esistenti (e.g., thè con limone)

L'utente può effettuare le seguenti operazioni:

- scegliere, prelevare e pagare una bevanda. Il pagamento può avvenire secondo le modalità: contanti (5, 10, 20, 50 centesimi, 1 e 2 euro), chiavetta ricaricabile o carta di credito
- ricaricare una chiavetta inserendo contanti (5,10,20,50 euro)

Proposta di realizzazione

Abbiamo scelto la modalità di sviluppo **programma standalone con supporto grafico** , usando l'IDE IntelliJ e il suo strumento grafico **GUI Designer**.

All'interno del progetto, facciamo uso dei file come componente per immagazzinare i dati relativi alle bevande, alle sottotipologie delle bevande e ai consumi mensili.

1 Teoria

1.1 Breve descrizione dei requisiti del progetto

Il progetto consiste in un simulatore di distributore automatico di bevande con accesso in modalità amministratore e utente.

Modalità amministratore:

- Rifornimento periodico di bevande;
- Definizione dei prezzi;
- Report mensili sui consumi;
- Aggiunta di nuove tipologie di bevande.

Modalità utente:

- Selezione della bevanda e quantità di zucchero;
- Pagamento in contanti, chiavetta ricaricabile o carta di credito;
- Ricarica della chiavetta con contanti.

1.2 Pattern

I pattern vengono definiti come soluzioni architetturali che possono risolvere problemi in contesti etegorenei. Ci sono tre tipi di Pattern fondamentali:

- Creational Patterns;
- Structural Patterns;
- Behavioral Patterns.

I pattern da noi scelti sono:

- Singleton Pattern: per garantire che esista una sola istanza della classe Distributore in modo che tutte le richieste di accesso al distributore siano gestite da un'unica istanza. Questo è utile per garantire coerenza e centralizzazione nella gestione delle operazioni di distribuzione e ricarica;
- Strategy Pattern: per gestire diversi algoritmi di pagamento. Abbiamo diverse strategie di pagamento per contanti, chiavetta ricaricabile e carta di credito. Abbiamo creato un'interfaccia PaymentStrategy con implementazioni concrete per ciascun metodo di pagamento, permettendo all'utente di selezionare la strategia desiderata;
- State Pattern: abbiamo utilizzato lo State Pattern per gestire le diverse modalità del distributore. Ogni classe che implementa l'interfaccia "ModalitaDistributore" è responsabile delle sue azioni specifiche;
- Command Pattern: utilizzato per incapsulare richieste specifiche di amministrazione (come l'aggiunta di una bevanda alla scorta) in oggetti comando, facilitando l'aggiunta di nuove funzionalità e la gestione delle azioni richieste dall'utente.

1.3 Diagramma UML delle classi

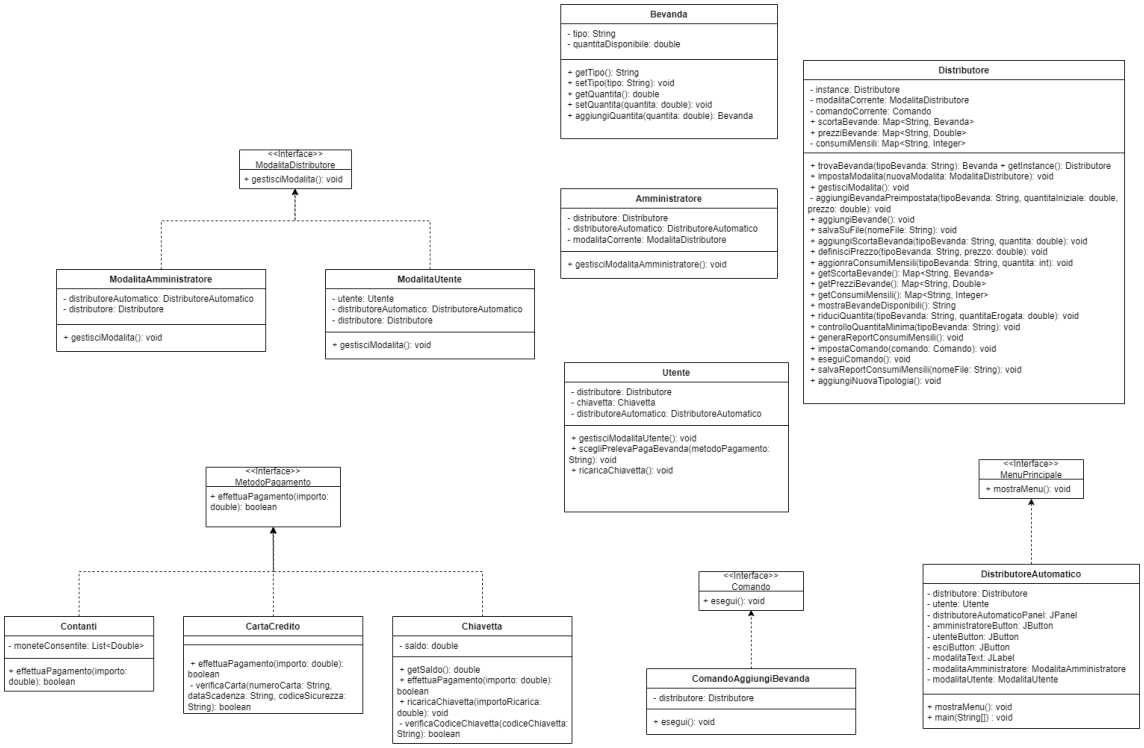


Diagramma delle classi

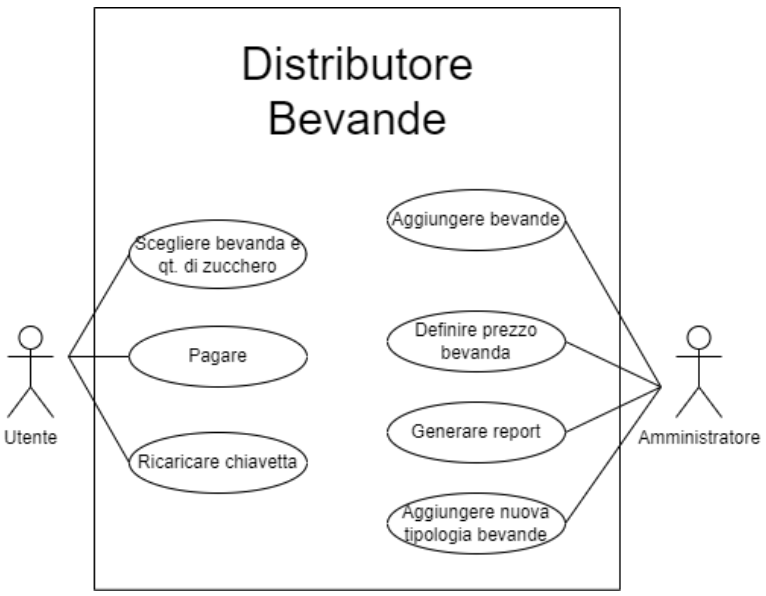


Diagramma use case

2 Pratica

In questa sezione verranno implementate parti rilevanti del codice sviluppato.

2.1 Pattern

Singleton Pattern

```
public class Distributore {
    private static Distributore instance;

    private Distributore() {
    .
    .
    .
    }

    public static Distributore getInstance() {
        if(instance == null) {
            instance = new Distributore();
        }
        return instance;
    }
}
```

```
public class DistributoreAutomatico extends JFrame implements
    MenuPrincipale {

    private Distributore distributore = Distributore.getInstance();
    .
    .
    .
}
```

La classe Distributore rappresenta il distributore automatico di bevande. Gestisce la scorta di bevande, i prezzi, i consumi mensili e fornisce funzionalità per aggiungere bevande, effettuare pagamenti e generare report.

State Pattern

```
public interface ModalitaDistributore {
    void gestisciModalita();
}
```

```
public class ModalitaAmministratore extends JPanel implements
    ModalitaDistributore {

        .
        .
        .

        @Override
        public void gestisciModalita() {
                JOptionPane.showMessageDialog(null, "Modalit ␣
                    amministratore␣selezionata", "Successo",
                    JOptionPane.INFORMATION_MESSAGE);

        }
}
```

Questa classe rappresenta la modalità amministratore del distributore automatico. Fornisce funzionalità come l'aggiunta di bevande alla scorta, la definizione dei prezzi delle bevande e la generazione di report sui consumi mensili.

```
public class ModalitaUtente extends JPanel implements ModalitaDistributore{
        private Utente utente;
        .
        .
        .
        @Override
        public void gestisciModalita() {
                JOptionPane.showMessageDialog(null, "Modalit  utente 
                    selezionata", "Successo",
                    JOptionPane.INFORMATION_MESSAGE);

        }
}
```

Questa classe rappresenta la modalità Utente del distributore automatico. Gli utenti possono scegliere, prelevare e pagare una bevanda, ricaricare la chiavetta, o tornare al menu principale.

Strategy Pattern

```
public interface MetodoPagamento {
    boolean effettuaPagamento(double importo);
}
```

```
public class CartaCredito implements MetodoPagamento {
    @Override
    public boolean effettuaPagamento(double importo) {
        String numeroCarta = JOptionPane.showInputDialog("Inserisci il 
            numero della carta:");
        String dataScadenza = JOptionPane.showInputDialog("Inserisci la 
            data di scadenza della carta:");
        String codiceSicurezza = JOptionPane.showInputDialog("Inserisci il 
            cvv della carta:");

        if(verificaCarta(numeroCarta, dataScadenza, codiceSicurezza)) {
            JOptionPane.showMessageDialog(null, "Pagamento effettuato con 
                successo. Importo pagato: " + importo + " euro",
                "Successo", JOptionPane.INFORMATION_MESSAGE);
            return true;
        } else {
            JOptionPane.showMessageDialog(null, "Errore durante il 
                pagamento. Riprova", "Errore", JOptionPane.ERROR_MESSAGE);
        }
        return false;
    }

    //Metodo privato utilizzato per verificare la validit  della carta di
        credito.
    //In questa implementazione, la verifica    simulata e restituisce
        sempre true.
    private boolean verificaCarta(String numeroCarta, String dataScadenza,
        String codiceSicurezza) {
        return true;
    }
}
```

La classe CartaCredito implementa l'interfaccia MetodoPagamento e consente di effettuare pagamenti utilizzando una carta di credito. Richiede all'utente di inserire il numero della carta, la data di scadenza e il codice di sicurezza per completare il pagamento.

```java
public class Chiavetta implements MetodoPagamento {
    private double saldo;       //Saldo attuale sulla chiavetta

    public Chiavetta(double saldoIniziale) {
        this.saldo = saldoIniziale;
    }

    public double getSaldo() {
        return saldo;
    }

    //Metodo per effettuare il pagamento utilizzando la chiavetta.
    //Richiede all'utente di inserire il codice della chiavetta e verifica se il saldo    sufficiente.
    @Override
    public boolean effettuaPagamento(double importo) {
        //Verifica se il saldo    sufficiente per effettuare il pagamento
        if(saldo >= importo) {
            String codiceChiavetta =
                JOptionPane.showInputDialog("Inserisci il codice della chiavetta");
            if(verificaCodiceChiavetta(codiceChiavetta)) {
                saldo -= importo;
                JOptionPane.showMessageDialog(null, "Pagamento effettuato con successo. Importo pagato: " + importo + " euro.\nSaldo rimanente: " + saldo + " euro",
                    "Successo", JOptionPane.INFORMATION_MESSAGE);
                return true;
            } else {
                JOptionPane.showMessageDialog(null, "Codice chiavetta non valido", "Errore", JOptionPane.ERROR_MESSAGE);
            }
        } else {
            JOptionPane.showMessageDialog(null, "Saldo insufficiente",
                "Errore", JOptionPane.ERROR_MESSAGE);
        }
        return false;
    }

    //Metodo privato per verificare la validit  del codice della chiavetta.
    //In questa implementazione, la verifica    simulata e restituisce sempre true.
    private boolean verificaCodiceChiavetta(String codiceChiavetta) {
        return true;
    }

    //Metodo per ricaricare la chiavetta con un importo specifico.
    public void ricaricaChiavetta(double importoRicarica) {
        if(importoRicarica == 5.0 || importoRicarica == 10.0 ||
           importoRicarica == 20.0 || importoRicarica == 50.0) {
            saldo += importoRicarica;
            JOptionPane.showMessageDialog(null, "Ricarica effettuata con successo. Nuovo saldo: " + saldo + " euro", "Successo",
                JOptionPane.INFORMATION_MESSAGE);
        } else {
            JOptionPane.showMessageDialog(null, "Importo ricarica non valido", "Errore", JOptionPane.ERROR_MESSAGE);
        }
    }
}
```

La classe Chiavetta implementa l'interfaccia MetodoPagamento e consente di effettuare pagamenti utilizzando una chiavetta prepagata con un saldo associato. È inoltre possibile ricaricare la chiavetta.

```
public class Contanti implements MetodoPagamento {
    //Lista dei valori delle monete consentite
    private List<Double> moneteConsentite = Arrays.asList(0.05, 0.10,
        0.20, 0.50, 1.0, 2.0);

    //Metodo per effettuare il pagamento utilizzando le monete.
    //Richiede all'utente di inserire monete fino a raggiungere l'importo
        specificato.
    //Restituisce il resto, se presente.
    @Override
    public boolean effettuaPagamento(double importo) {
        double importoRimanente = importo;
        double importoPagato = 0.0;

        while(importoRimanente > 0) {
            String input = JOptionPane.showInputDialog("Inserire moneta:");
            double monetaInserita;

            try {
                monetaInserita = Double.parseDouble(input);
            } catch(NumberFormatException exc) {
                JOptionPane.showMessageDialog(null, "Inserire un valore
                    numerico valido", "Errore", JOptionPane.ERROR_MESSAGE);
                continue;
            }

            if(moneteConsentite.contains(monetaInserita)) {
                importoPagato += monetaInserita;
                importoRimanente -= monetaInserita;
                if(importoRimanente > 0) {
                    JOptionPane.showMessageDialog(null, "Importo
                        rimanente: " + importoRimanente + " euro",
                        "Importo Rimanente",
                        JOptionPane.INFORMATION_MESSAGE);
                }
            } else {
                JOptionPane.showMessageDialog(null, "Moneta non
                    consentita", "Errore", JOptionPane.ERROR_MESSAGE);
            }
        }
        JOptionPane.showMessageDialog(null, "Pagamento effettuato con
            successo. Importo pagato: " + importo + " euro", "Successo",
            JOptionPane.INFORMATION_MESSAGE);

        //Restituzione resto
        double resto = importoPagato - importo;
        if(resto > 0) {
            JOptionPane.showMessageDialog(null, "Resto: " + resto + "
                euro", "Resto", JOptionPane.INFORMATION_MESSAGE);
        }
        return true;
    }
}
```

La classe Contanti implementa l'interfaccia MetodoPagamento e consente di effettuare pagamenti utilizzando monete di diverso valore.

Command Pattern

```
public interface Comando {
    void esegui();
}
```

```
public class ComandoAggiungiBevanda implements Comando {
    //Riferimento al distributore su cui eseguire il comando
    private Distributore distributore;

    public ComandoAggiungiBevanda(Distributore distributore) {
        this.distributore = distributore;
    }

    //Metodo per eseguire il comando di aggiunta bevande sul distributore.
    @Override
    public void esegui() {
        distributore.aggiungiBevande();
    }
}
```

La classe ComandoAggiungiBevanda implementa l'interfaccia Comando e rappresenta un comando per aggiungere bevande al distributore.

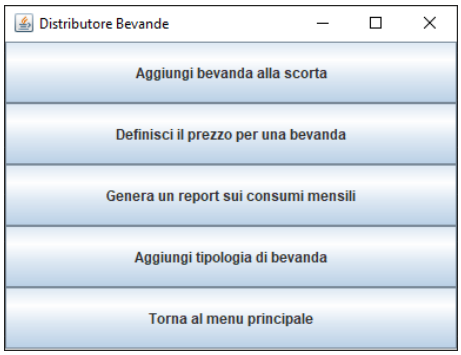
```
public class Distributore {
    private Comando comandoCorrente;

    public void impostaComando(Comando comando) {
        this.comandoCorrente = comando;
    }

    public void eseguiComando() {
        if(comandoCorrente != null) {
            comandoCorrente.esegui();
        } else {
            System.out.println("Nessun comando impostato");
        }
    }
}
```

```
public class ModalitaAmministratore extends JPanel implements
    ModalitaDistributore {
    public ModalitaAmministratore(DistributoreAutomatico
        distributoreAutomatico, Distributore distributore) {
        .
        .
        JButton aggiungiBevandaButton = new JButton("Aggiungi␣bevanda␣alla␣
            scorta");

        aggiungiBevandaButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                //Crea un nuovo comando per aggiungere una bevanda al
                    distributore
                Comando comandoAggiungiBevanda = new
                    ComandoAggiungiBevanda(distributore);

                //Imposta il comando corrente nel distributore
                distributore.impostaComando(comandoAggiungiBevanda);

                //Esegue il comando corrente
                distributore.eseguiComando();
            }
        });
    }
}
```

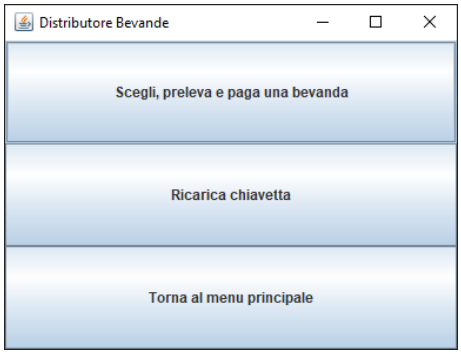
2.2 Rappresentazione grafiche

In questa sezione ci sono alcuni esempi delle nostre interfacce grafiche sviluppate con IntelliJ IDEA:

Menu



Amministratore



Utente

2.3 File.txt

Qui ci sono le bevande prefissate: caffè, thè, latte, camomilla, cioccolata calda e acqua.

bevande.txt

```
Nome bevanda: The, quantità: 5.0, prezzo: 2.5
Nome bevanda: Camomilla, quantità: 5.0, prezzo: 2.5
Nome bevanda: Acqua, quantità: 5.0, prezzo: 0.5
Nome bevanda: Caffe, quantità: 5.0, prezzo: 1.0
Nome bevanda: Cioccolata calda, quantità: 5.0, prezzo: 3.0
Nome bevanda: Latte, quantità: 5.0, prezzo: 1.5

```

Questo è un esempio del file che contiene il report sui consumi delle bevande.

consumiMensili.txt

```
Bevanda: Latte, acqua, Quantità consumata: 3 unità
Bevanda: Cioccolata calda, Quantità consumata: 4 unità
Bevanda: The, camomilla, Quantità consumata: 1 unità

```

Qui ci sono le sottotipologie che derivano dalle tipologie principali. Alcuni esempi:caffè e latte, latte e caffè, caffè e cioccolato, thè e latte.

sottotipologie.txt

```
Nome bevanda: Latte, acqua, quantità: 5.0, prezzo: 2.0
Nome bevanda: Latte, cioccolata calda, quantità: 5.0, prezzo: 4.5
Nome bevanda: Cioccolata calda, the, quantità: 5.0, prezzo: 5.5
Nome bevanda: The, cioccolata calda, quantità: 5.0, prezzo: 5.5
Nome bevanda: Caffe, latte, quantità: 5.0, prezzo: 2.5
Nome bevanda: Cioccolata calda, acqua, quantità: 5.0, prezzo: 3.5
Nome bevanda: Acqua, cioccolata calda, quantità: 5.0, prezzo: 3.5
Nome bevanda: Camomilla, the, quantità: 5.0, prezzo: 5.0
Nome bevanda: The, quantità: 5.0, prezzo: 2.5
Nome bevanda: Caffe, cioccolata calda, quantità: 5.0, prezzo: 4.0
Nome bevanda: Acqua, camomilla, quantità: 5.0, prezzo: 3.0
Nome bevanda: Caffe, acqua, quantità: 5.0, prezzo: 1.5
Nome bevanda: Caffe, the, quantità: 5.0, prezzo: 3.5
Nome bevanda: The, acqua, quantità: 5.0, prezzo: 3.0
Nome bevanda: Cioccolata calda, quantità: 5.0, prezzo: 3.0
Nome bevanda: Acqua, latte, quantità: 5.0, prezzo: 2.0
Nome bevanda: Acqua, the, quantità: 5.0, prezzo: 3.0
Nome bevanda: The, camomilla, quantità: 5.0, prezzo: 5.0
Nome bevanda: Latte, the, quantità: 5.0, prezzo: 4.0
Nome bevanda: The, latte, quantità: 5.0, prezzo: 2.5
Nome bevanda: Acqua, caffe, quantità: 5.0, prezzo: 1.5
Nome bevanda: The, caffe, quantità: 5.0, prezzo: 3.5
Nome bevanda: Caffe, camomilla, quantità: 5.0, prezzo: 3.5
Nome bevanda: Latte, quantità: 5.0, prezzo: 1.5
Nome bevanda: Camomilla, latte, quantità: 5.0, prezzo: 4.0
Nome bevanda: Latte, camomilla, quantità: 5.0, prezzo: 4.0
Nome bevanda: Latte, caffe, quantità: 5.0, prezzo: 2.5
Nome bevanda: Camomilla, caffe, quantità: 5.0, prezzo: 3.5
Nome bevanda: Camomilla, quantità: 5.0, prezzo: 2.5
Nome bevanda: Cioccolata calda, caffe, quantità: 5.0, prezzo: 4.0
Nome bevanda: Acqua, quantità: 5.0, prezzo: 0.5
Nome bevanda: Camomilla, cioccolata calda, quantità: 5.0, prezzo: 5.5
Nome bevanda: Caffe, quantità: 5.0, prezzo: 1.0
Nome bevanda: Cioccolata calda, camomilla, quantità: 5.0, prezzo: 5.5
Nome bevanda: Camomilla, acqua, quantità: 5.0, prezzo: 3.0
Nome bevanda: Cioccolata calda, latte, quantità: 5.0, prezzo: 4.5
```

3 Conclusioni

Il programma di gestione del distributore automatico di bevande presentato in questo progetto rappresenta una soluzione completa per le esigenze di amministratori e utenti.

L'amministratore ha a disposizione diverse funzionalità per gestire efficientemente il distributore, tra cui la possibilità di:

- Mantenere la scorta delle bevande;
- Stabilire i prezzi;
- Monitorare i consumi mensili;
- Aggiungere nuove tipologie di bevande;

Queste funzionalità consentono di personalizzare il distributore in base alle esigenze specifiche dell'attività e di garantire un'esperienza utente fluida e conveniente.

Gli utenti, invece, possono comodamente selezionare, prelevare e pagare le bevande secondo le loro preferenze. Il sistema offre diverse opzioni di pagamento, tra cui contanti, chiavette ricaricabili e carte di credito.

In sintesi, il programma presentato in questo progetto offre un'interfaccia intuitiva e funzionalità avanzate che rendono il distributore automatico di bevande un servizio efficiente e adattabile alle esigenze degli utenti.

La modularità del codice permette future espansioni e miglioramenti, assicurando la durabilità e la flessibilità del sistema nel tempo.