Progetto Programmazione 3

-TELEPASS S.p.A-

A cura di :

Mario Scognamiglio 0124002032

A.A 2020/2021

1.Descrizione del progetto

Si vuole sviluppare un programma che ricrea il sistema Telepass presente nelle autostrade.
Il sistema Telepass è in grado di gestire l'accesso degli utenti(veicoli) all'autostrada, la gestione avviene quando un veicolo attraversa il casello. Il sistema calcola le varie tratte che un veicolo percorre e quindi addebita all'utente un costo del pedaggio (varia a seconda dell'entrata e uscita di un casello). Il programma è strutturato in modo che ci siano due modalità, la modalità amministratore, dove si gestisce il sistema Telepass e la modalità veicolo che sarebbe l'utente che percorre l'autostrada.

Modalità Amministratore

Se utilizziamo la modalità amministratore, bisogna accedere tramite password (per default, impostata a: 0000). Questa modalità permette di effettuare operazioni privilegiate, ad esempio l'approvazione di un nuovo dispositivo telepass, la rimozione di un dispositivo telepass e la richiesta al sistema di creare delle statistiche aggiornate.

Modalità Veicolo

Ogni veicolo rappresenta un utente, ogni veicolo è dotato di un dispositivo (trans-ponder). Quest'ultimo permette di interagire con tutto il sistema Telepass e di accedere all'autostrada.
Quando si connette un nuovo utente quindi un nuovo veicolo, esso viene registrato nel sistema.
Ogni utente inserisce il nome, il cognome, la targa e il metodo di pagamento che preferisce.
Il veicolo può entrare in autostrada fare il proprio percorso ed uscire dall'autostrada tramite i caselli presenti nel sistema.
Il veicolo per entrare in autostrada ha bisogno dell'approvazione da parte dell'amministratore di sistema.
Quando il veicolo esce dall'autostrada il sistema Telepass calcola il pedaggio ed addebita automaticamente la cifra all'utente.
Ogni utente ha possibilità di cambiare il veicolo, il sistema chiederà solamente la nuova targa. Il cambio veicolo non può essere effettuato quando il veicolo sta viaggiando in autostrada.
Infine un utente può aderire al servizio esteso "Telepass Plus "con possibilità di assistenza in autostrada.

Sistema

Il sistema ha traccia di tutti i dispositivi connessi, dei caselli disponibili, delle entrate e delle uscite dall'autostrada questo è possibile grazie all'utilizzo di un database. Ogni volta che viene avviato il programma il database viene azzerato.

2.Informazioni

Il programma è scritto in linguaggio java.

L'editor di codice che è stato utilizzato è Eclipse.

Il database è stato creato con SQLite, si trova nella cartella del progetto /sqlite.

Nella cartella lib invece si trova il file sqlite-jdbc-3.34.0 che sarebbe il file jar contenente il driver JDBC con tutte le classi per il funzionamento del database.

Per quanto riguarda il supporto grafico è stato utilizzato Java Swing e vengono utilizzati anche i thread.

Javadoc si trova nella cartella del progetto /javadoc.

Il diagramma completo si trova nella cartella del progetto /uml.

Sono stati utilizzati i seguenti design pattern: Command, Builder, Singleton.

I design pattern Command e Builder sono stati utilizzati principalmente per la costruzione dell'interfaccia.

In particolare, il Command è stato utilizzato per la generazione dei pulsanti, dove ogni pulsante aziona un determinato comando.

Il pattern Builder è stato utilizzato per la costruzione delle componenti dell'interfaccia, come ad esempio spazi di inserimento, pulsanti e testo.

Il pattern singleton **Singleton** (versione **eager**) è stato utilizzato per creare una sola istanza della classe avente lo scopo di astrarre il database e le relative operazioni.

3.Diagrammi

Siccome non è stato possibile inserire il diagramma UML completo per motivi di spazio, si è preferito inserire dei diagrammi UML semplificati.

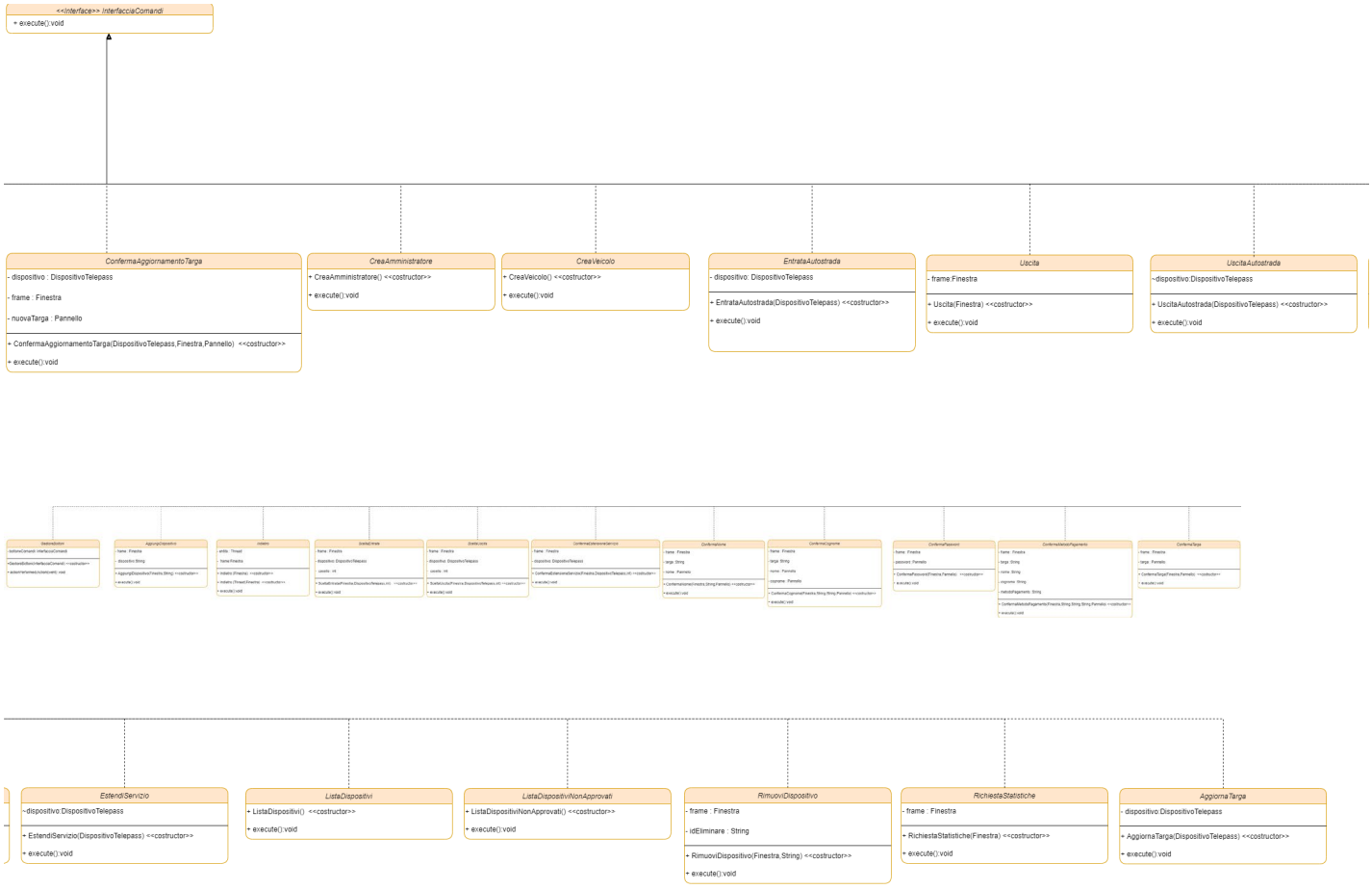
All'interno della cartella del progetto ci sarà anche il diagramma delle classi.

Il file si trova all'interno della cartella uml ed è stato nominato come diagramma_delle_classi.png

Controller

Il package ‘’controller” contiene l’interfaccia ’’InterfacciaComandi’’ e tutte le classi che la implementano. Ogni classe di questo package rappresenta un comando che viene eseguito alla pressione di un pulsante dell’interfaccia.

Di seguito possiamo visualizzare il diagramma uml diviso in più immagini per facilitarne la lettura.



Model

Il package ‘’model’’ contiene la classe amministratore, dispositivo telepass, globals, sistema telepass e veicolo.

La classe amministratore accede al sistema immettendo una password e sarà in grado di effettuare alcune operazioni.

Successivamente vi troviamo la classe dispositivo telepass, dove ogni istanza rappresenta un dispositivo telepass. Il dispositivo ha un codice univoco, il quale contiene una variabile che punta al veicolo associato, cronologia dei pagamenti e lo stato del dispositivo.

La classe globals è un’utility che contiene campi generali utilizzabili da tutte le classi.

La classe SistemaTelepass contiene tutti i metodi per effettuare le operazioni dei veicoli e degli amministratori. E’ l’unica classe che ha accesso ad un database locale in SQLite, tiene traccia di tutti i caselli, dispositivi connessi al sistema e delle entrate e uscite dall’autostrada. Attraverso il singleton generato all’ avvio del programma, si ha l’accesso al sistema.

Infine vi è la classe veicolo, dove per ogni istanza, verrà generato un menu con informazioni e pulsanti.

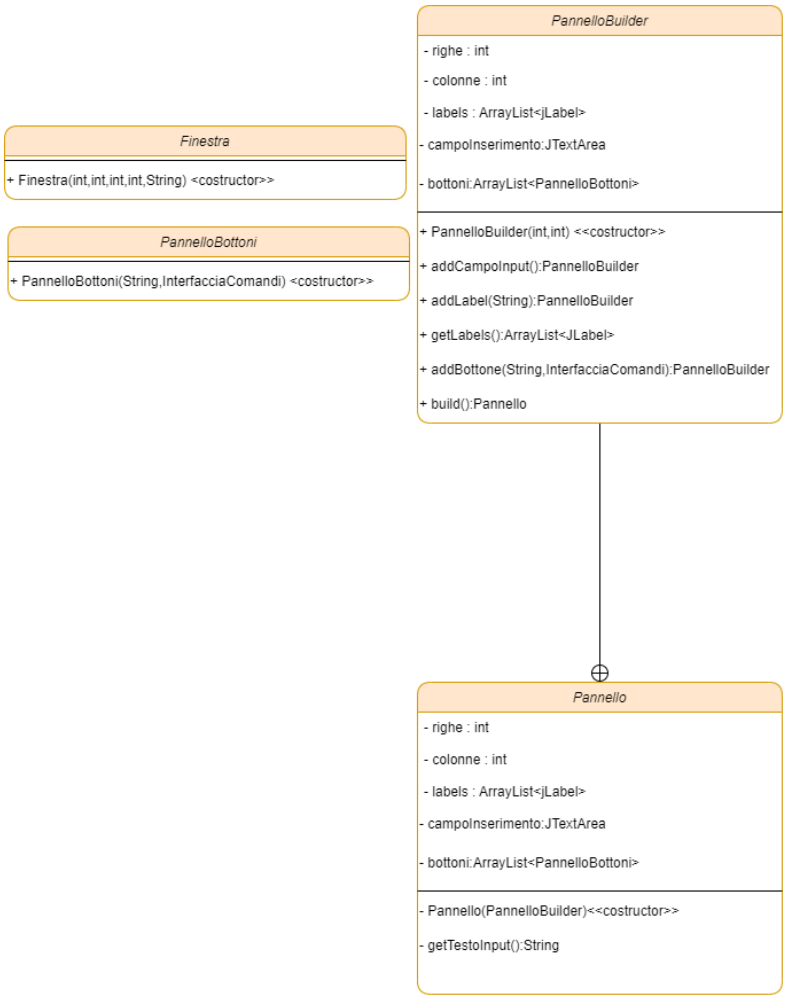
View

Contiene tutte le classi utilizzate per la costruzione dell'interfaccia, tranne i comandi che sono associati alla pressione dei bottoni.

La classe Pannello viene utilizzata per la costruzione dell'interfaccia durante l'esecuzione del programma. Ogni istanza di questa classe rappresenta un pannello che deve essere aggiunto ad un frame, che deve essere mostrato. Ogni pannello può contenere degli spazi di input, dei pulsanti e del testo.

La classe Finestra è un frame, con un costruttore più comodo ai fini del programma.

La classe PannelloBottoni permette di generare tutti i pulsanti dell'interfaccia, facendo in modo che ognuno di essi esegua delle operazioni indipendenti.



4.Pattern

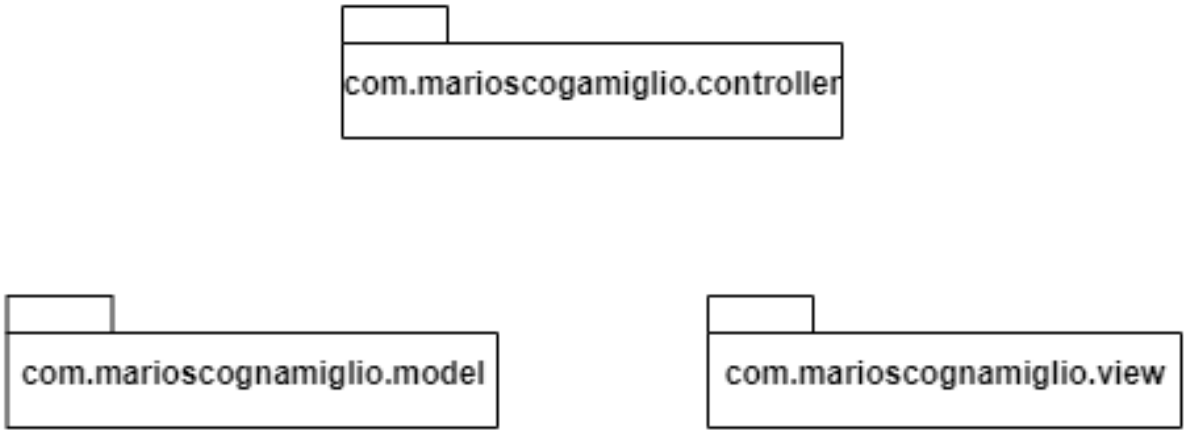
PATTERN ARCHITETTURALE: MVC

Il pattern architetturale MVC è stato implementato in quanto rende il progetto più leggibile e lo rende più solido per futuri aggiornamenti.

Il pattern è basato sulla separazione dei compiti fra i componenti software che interpretano tre ruoli principali:

- il *model* fornisce i metodi per accedere ai dati utili all'applicazione;
- la *view* visualizza i dati contenuti nel model e si occupa dell'interazione con utenti e agenti;
- il *controller* riceve i comandi dell'utente (in genere attraverso la view) e li attua modificando lo stato degli altri due componenti.

Nel progetto il pattern è stato implementato nel seguente modo:



Singleton

E'stato applicato il pattern di tipo creazionale: Singleton. Tale pattern è stato utilizzato per creare una sola istanza della classe avente lo scopo di astrarre il database e le relative operazioni (Versione eager). SistemaTelepass contiene tutti i metodi che permettono il funzionamento del sistema, inoltre è l'unica classe che ha accesso diretto al database.

Builder

E'stato applicato il pattern di tipo creazionale : Builder. Tale pattern ha lo scopo di separare la costruzione di un oggetto complesso dalla sua rappresentazione, in modo tale che lo stesso processo di costruzione può creare differenti rappresentazioni.

Quindi, nel progetto è stato implementato il pattern in questione, in quanto ci permette di costruire un oggetto complesso come il pannello, permettendone differenti rappresentazioni.

Il builder, è stato implementato attraverso una classe statica innestata, che si occupa di costruire l'istanza superiore, solo dopo aver raccolto le informazioni passate dinamicamente.

Pannello, invece, è una classe privata in quanto, l'unica classe in grado di accederci, deve essere la classe builder innestata.

PannelloBuilder ha un costruttore che chiede in input I parametri obbligatori per creare una qualsiasi istanza di Pannello .

Successivamente, è possible chiamare diversi metodi del builder in modo da aggiungere componenti completamente facoltativi. Solo quando è stato aggiunto tutto il necessario, bisognerà chiamare il metodo build per ritornare un'istanza di Pannello.

Command

E' stato applicato il pattern di tipo comportamentale : Command. Che incapsula una richiesta all'interno di un oggetto, consentendo così di parametrizzare i client con richieste diverse, accordare e rintracciare le richieste, oppure supportare operazioni di uno.

In particolare è stato implementato tale pattern, in quanto vi era la necessità di richiedere dei compiti ad oggetti (bottone), senza conoscere tutto sulle operazioni richieste.

Per tal motivo, Il pattern Command, è stato utilizzato per la creazione dei pulsanti, ogni pulsante aziona un determinato comando.

Infatti, sono stati implementati un gran numero di comandi ed ogni comando deve implementare un metodo ''execute'', dove, ogni pulsante, quando premuto, deve eseguire un comando. Un pulsante viene creato, genericamente, sfruttando l'interfaccia, contenente il singolo metodo execute.In questo modo, si rispettano i principi SOLID ed in particolare il principio di segregazione delle interfacce (interface segregation principle).

5.Parti rilevanti del codice

Segue un'analisi di alcune parti del codice.

Command

Il pattern command necessita prima di tutto di comandi. È stato implementato un gran numero di comandi e ogni comando deve implementare un metodo "execute". Si sfrutta dunque un'interfaccia come segue:

com.MarioScognamiglio.contoller/InterfacciaComandi.java

```
public interface InterfacciaComandi {
    public void execute();
}
```

Ogni pulsante, quando premuto, deve eseguire uno di questi comandi. Creare un pulsante significa dunque passare al suo costruttore un comando, e lo si fa in maniera generale sfruttando l'interfaccia come segue:

com.MarioScognamiglio.view/PannelloBottoni.java

```
public class PannelloBottoni extends JButton {

    public PannelloBottoni(String nomeBottone, InterfacciaComandi
comandiBottone) {
        this.setText(nomeBottone);
        this.setBackground(Color.DARK_GRAY);
        this.setForeground(Color.WHITE);
        this.setBorder(new LineBorder(Color.WHITE));
        this.addActionListener(new GestoreBottoni(comandiBottone));
```

Il costruttore imposta l'actionListener sfruttando il comando ricevuto in input. Entra dunque in gioco GestoreBottoni, che imposta il proprio metodo actionPerformed (pulsante premuto) in modo che richiami il metodo execute del comando passato in input, come segue:

com.MarioScognamiglio.contoller/GestoreBottoni.java

```
public class GestoreBottoni implements ActionListener {

   private InterfacciaComandi bottoneComandi; // Comando generico

          //bottoneComandi Comando da associare al pulsante

 public GestoreBottoni(InterfacciaComandi bottoneComandi) {
        this.bottoneComandi = bottoneComandi;
    }

   // Quando il pulsante viene premuto, il comando associato viene lanciato.

    public void actionPerformed (ActionEvent e) {
        this.bottoneComandi.execute();   }
```

Builder

Il pattern Builder garantisce l'immutabilità e ci permette di costruire un'istanza di Pannello come più ci aggrada. Il builder è una classe statica interna che si occupa di costruire l'istanza superiore solo dopo aver raccolto tutto il necessario. Analizziamo quindi il costruttore di Pannello, che è privato proprio perché l'unico in grado di richiamarlo deve essere il builder interno. Il costruttore inizializza i membri sulla base di ciò che gli fornisce il builder.

com.MarioScognamiglio.view/Pannello.java

```
    private Pannello(PannelloBuilder builder) {
        this.righe = builder.righe;
        this.colonne = builder.colonne;
        this.labels = builder.labels;
        this.campoInserimento = builder.campoInserimento;
        this.bottoni = builder.bottoni;

        this.setLayout(new GridLayout(this.righe, this.colonne));

        try {       // Provo ad inserire degli spazi di inserimento
            this.add(this.campoInserimento);
        } catch (Exception e) {
             e.printStackTrace();
        }

        for (JLabel label : this.labels) // Inserisco le label
            this.add(label);

        for (PannelloBottoni bottoni : this.bottoni) // Inserisco i pulsanti
            this.add(bottoni);
    }
```

PannelloBuilder possiede un proprio costruttore che chiede in input i parametri obbligatori per creare una qualsiasi istanza di Pannello. Successivamente, è possibile chiamare diversi metodi del builder in modo da aggiungere componenti completamente facoltative. Solo quando è stato aggiunto tutto il necessario, bisognerà chiamare il metodo build per creare effettivamente l'istanza di Pannello.

com.MarioScognamiglio.view/Pannello.java

```
//classe del builder l'unica in grado di costruire i pannelli
public static class PannelloBuilder {
        private int righe;
        private int colonne;
        private ArrayList<JLabel> labels;
        private JTextArea campoInserimento;
        private ArrayList<PannelloBottoni> bottoni;
```

```
//costruttore del builder ,qui vengono caricati i campi obbligatori della
//classe pannello
public PannelloBuilder(int righe, int colonne) {
            this.righe = righe;
            this.colonne = colonne;
            this.labels = new ArrayList<JLabel>();
            this.bottoni = new ArrayList<PannelloBottoni>();
        }


//capo facoltativo, spazio di inserimento
    public PannelloBuilder addCampoInput() {
            this.campoInserimento = new JTextArea();

            campoInserimento.setBackground(new Color(189, 247, 183));//colore
            campoInserimento.setForeground(Color.darkGray);//colore
            campoInserimento.setFont(new Font("Calibri", Font.BOLD, 18));//font

            return this;
        }

//campo facoltativo,testo
Public PannelloBuilder addLabel(String label) {
            JLabel labelStats = new JLabel(label, JLabel.CENTER);
            labelStats.setFont(new Font("Calibri", Font.BOLD, 18));//font
            labelStats.setBackground(Globals.secondaryColor);//colore
            labelStats.setForeground(new Color(57, 67, 183));//colore
            labelStats.setOpaque(true);

            this.labels.add(labelStats);

            return this;
        }


//campo facoltativo, pulsante
   public PannelloBuilder addBottone(String nomeComando, InterfacciaComandi
comandi) {
            this.bottoni.add(new PannelloBottoni(nomeComando, comandi));

            return this;
        }


//chiamo il costruttore del pannello
public Pannello build() {
            return new Pannello(this);
        }
```

ESEMPIO

Analizzo due esempi per il pattern Builder. Il codice che segue è incluso nel metodo run della classe Veicolo. Vengono creati 2 pannelli, uno contente solo label(testo), l'altro contente solo pulsanti. I due pannelli vengono successivamente aggiunti a Finestra per costruire l'interfaccia.

com.MarioScognamiglio.contoller/Veicolo.java

```
This.frame = new Finestra(Globals.frameHeight, Globals.frameWidth, 3, 1,
"MODALITA' VEICOLO"); // Finestra del veicolo

this.labels = new Pannello.PannelloBuilder(3, 1).addLabel("ID DISPOSITIVO
TELEPASS  " + this.dispositivo.getID())

  .addLabel("TARGA " + this.targa).addLabel("TIPO DI SERVIZIO " +
this.dispositivo.getTipoServizio())

 .addLabel("METODO DI PAGAMENTO UTILIZZATO " +
this.metodoPagamento).addLabel("NOME  " + this.nome)

 .addLabel("COGNOME" + this.cognome).build();

 this.bottoni = new Pannello.PannelloBuilder(5, 1).addBottone("ENTRATA IN
AUTOSTRADA", new EntrataAutostrada(this.dispositivo))

 .addBottone("USCITA AUTOSTRADA", new UscitaAutostrada(this.dispositivo))

 .addBottone("CAMBIA VEICOLO", new AggiornaTarga(this.dispositivo))

.addBottone("ESTENDI SERVIZIO IN TELEPASS PLUS", new
EstendiServizio(this.dispositivo))

 .addBottone("USCITA", new Indietro(this, this.frame)).build();

        this.frame.add(this.labels);
        this.frame.add(this.bottoni);
        this.frame.setVisible(true);
    }
```

In seguito troviamo un altro esempio, proveniente dalla classe ListaDispositivi. In questo caso, viene creato un pannello con un solo pulsante, poi un pannello che contiene un testo e un pulsante sulla stessa riga, per ogni stringa presente in una lista, e che viene subito aggiunto al frame.

com.MarioScognamiglio.contoller/ListaDispositivi.java

```
 public void execute() {
Finestra frame = new Finestra(Globals.frameHeight, Globals.frameWidth, 1, 1,
"SCEGLI IL DISPOSITIVO CHE VUOI RIMUOVERE");

 Pannello indietro = new Pannello.PannelloBuilder(1, 1).addBottone("INDIETRO",
new Indietro(frame)).build(); //bottone per tornare indietro

        Pannello item;

ArrayList<String> dispositivi = SistemaTelepass.getSystem().getDispositivi();
//richiesta dispositivi gia' approvati dal sistema Telepass


for (String dispositivo : dispositivi) { //Per ogni dispositivo gia' approvato
genero un bottone di scelta
item = new Pannello.PannelloBuilder(2, 1).addLabel(dispositivo)


.addBottone("RIMUOVI", new RimuoviDispositivo(frame, dispositivo)).build();
//codice dispositivo e pulsante di scelta


frame.add(item);
}

frame.add(indietro);
frame.setVisible(true); //mostro lista solo quando sono stati aggiunti   tutti
i dispositivi gia' approvati
    }
}
```

6.Interfaccia grafica: Java Swing

L’interfaccia grafica è stata implementata con java swing.
Swing è un framework per Java orientato allo sviluppo di interfacce grafiche.
Parte delle classi del framework Swing sono implementazioni di widget (oggetti grafici) come caselle di testo, pulsanti, pannelli e tabelle.
La libreria Swing viene utilizzata come libreria ufficiale per la realizzazione di interfacce grafiche in Java. È un'estensione del precedente Abstract Window Toolkit. La differenza principale tra i due è che i componenti Swing sono scritti completamente in codice Java.
La libreria di tipo Swing, si presta egregiamente alla realizzazione di un'interfaccia adatta a qualunque piattaforma.
Ecco un esempio di implementazione della libreria java swing:
com.MarioScognamiglio.view/Pannello.java

```
public class Pannello extends JPanel {
    private int righe;
    private int colonne;
    private ArrayList<JLabel> labels;
    private JTextArea campoInserimento;
    private ArrayList<PannelloBottoni> bottoni;
    …
```