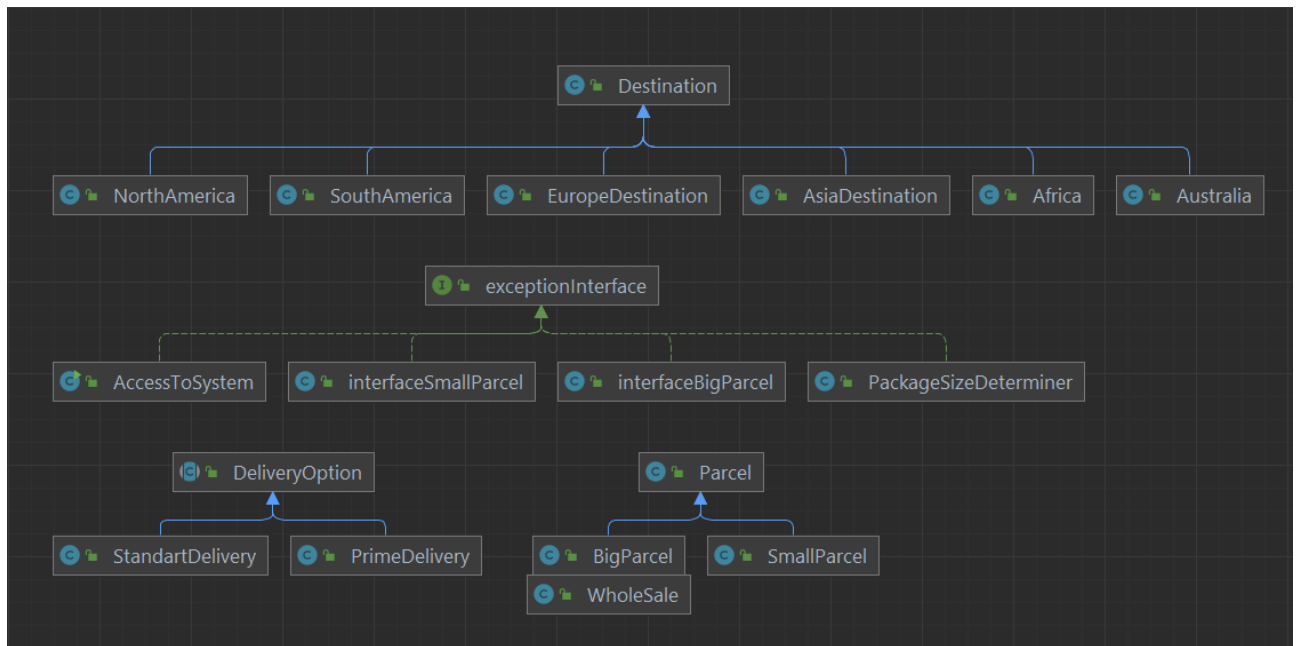


Documentation to the project

Object-Oriented Programming

Andrii Ilkiv

The whole project diagram



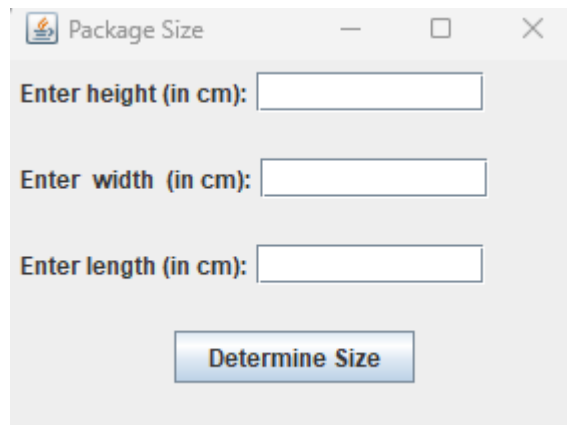
About the project

My project focuses on facilitating the transportation of parcels from China to Europe.

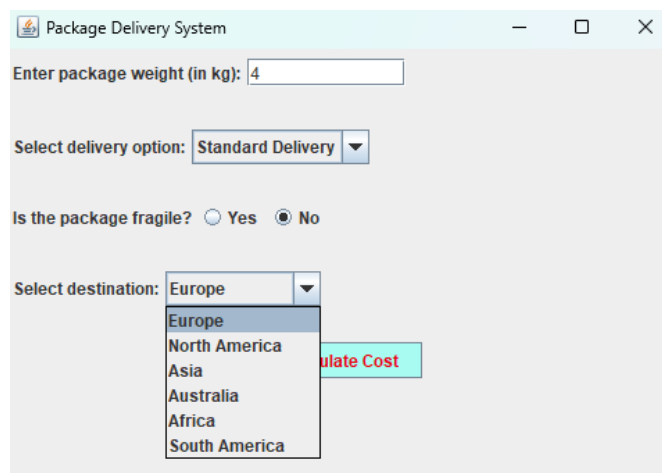
To achieve this goal, we have developed two main classes: Parcel and Delivery. The Parcel class represents the different types of parcels that may require transportation, encompassing both small and large parcels. For small parcels, there is an option to specify whether they are fragile or standard. In the case of large parcels, there is also the possibility of becoming a wholesale buyer. However, this decision is contingent upon the parcel prices. The Delivery class oversees the entire delivery process, encompassing both standard and prime deliveries. These two options differ in terms of price and delivery speed.

- **Program visualization**

After simple login into the system we are going to the package size determiner, it determines if your parcel is big or small.



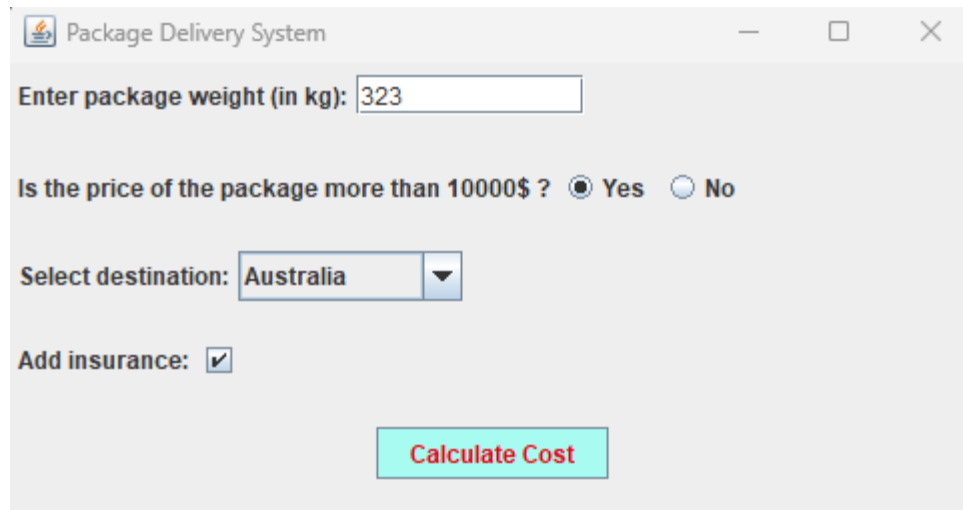
If your parcel is small, it will open this interface:



Here you can choose which delivery you need (Prime or Standard), shipping continent, weight of the parcel and if the parcel is fragile or not.

It will count all the conditions and will calculate a shipping cost.

If your parcel is big, it will open the next interface:

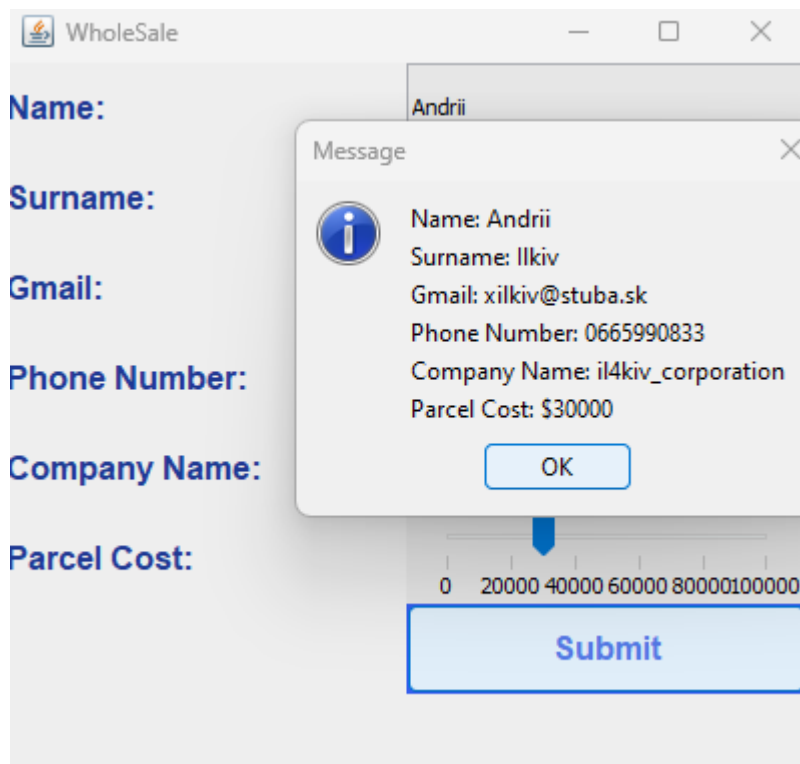


The screenshot shows a window titled "Package Delivery System". It contains the following elements:

- A text input field labeled "Enter package weight (in kg):" with the value "323" entered.
- A radio button group for "Is the price of the package more than 10000\$?" with "Yes" selected.
- A dropdown menu for "Select destination:" showing "Australia".
- A checkbox for "Add insurance:" which is checked.
- A red "Calculate Cost" button at the bottom.

It's almost the same as in small parcel but you can add insurance(+\$).
If the price of the package is more than 10000\$ you need to become a WholeSale person.

You can become WholeSale person in the next interface:



The screenshot shows a window titled "WholeSale" with a message dialog box open over it.

WholeSale Window:

- Fields for "Name:", "Surname:", "Gmail:", "Phone Number:", "Company Name:", and "Parcel Cost:".
- A "Submit" button at the bottom.
- A slider for "Parcel Cost:" ranging from 0 to 100,000, with a blue arrow pointing to approximately 30,000.

Message Dialog:

- Title: "Message"
- Content:
 - Name: Andrii
 - Surname: Ilkiv
 - Gmail: xilkiv@stuba.sk
 - Phone Number: 0665990833
 - Company Name: il4kiv_corporation
 - Parcel Cost: \$30000
- Buttons: "OK"

Code:

Shortly, how i used an aggregation:

```
import Code.Destinations.Destination;
import Code.Parcel.Parcel;

2 inheritors  Ilkiv Andrii *
public abstract class DeliveryOption {
    1 usage
    Parcel parcel;
    3 usages
    protected double baseCost;
    3 usages
    protected double fragileCost;

    Ilkiv Andrii *
    public DeliveryOption(Parcel parcel, double baseCost, double fragileCost) {
        this.parcel = parcel;
        this.baseCost = baseCost;
        this.fragileCost = fragileCost;
    }

    2 usages  2 implementations  Ilkiv Andrii
    public abstract double calculateCost(Parcel parcel, Destination destination);

    3 usages  Ilkiv Andrii
    public static DeliveryOption createDeliveryOption(String option) {
        switch (option.toLowerCase()) {
            case "prime":
                return new PrimeDelivery();
            case "standart":
                return new StandartDelivery();
            default:
                throw new IllegalArgumentException("Invalid delivery option: " + option);
        }
    }
}
```

The DeliveryOption class aggregates the Parcel object as a member variable. The DeliveryOption class has a member variable named parcel of type Parcel. It demonstrates the usage of aggregation by having the DeliveryOption class aggregate the Parcel object.

Further criteria explaining:

- using default method implementation in interfaces

```
1 package Code.Interfaces;
2
3 /**
4  * The exceptionInterface interface defines a contract for classes that handle exceptions.
5  * It provides a default method for printing an exception message.
6  */
7 public interface exceptionInterface {
8
9     /**
10      * Prints the specified message to the standard output.
11      *
12      * @param message the message to be printed
13      */
14     default void printMessage(String message) {
15         System.out.println(message);
16     }
17 }
```

I created a public interface with default method for writing the exceptions to the console.

- explicit use of RTTI

```
class Code.Interfaces.AccessToSystem
class Code.Interfaces.PackageSizeDeterminer
333.0
3.0
3.0
class Code.Interfaces.interfaceBigParcel
class Code.Interfaces.Wholesale
```

When we are changing the interfaces it is writing to the console at runtime.

It's called Run-Time Type Indetification.

- explicit use of multithreading

```
new SwingWorker() {
    public Object doInBackground() {
        if (height <= 50 && width <= 50 && length <= 50) {
            printMessage(String.valueOf(height));
            printMessage(String.valueOf(width));
            printMessage(String.valueOf(length));
            JOptionPane.showMessageDialog( parentComponent: PackageSizeDeterminer.this, message: "Your package is small");
            dispose();
            interfaceSmallParcel actionWindow = new interfaceSmallParcel();
            actionWindow.setVisible(true);
        } else {
            printMessage(String.valueOf(height));
            printMessage(String.valueOf(width));
            printMessage(String.valueOf(length));
            JOptionPane.showMessageDialog( parentComponent: PackageSizeDeterminer.this, message: "Your package is big");
            dispose();
            interfaceBigParcel actionWindow = new interfaceBigParcel();
            actionWindow.setVisible(true);
        }
        return null;
    }
}.execute();
```

In the actionPerformed method, a SwingWorker is created and its doInBackground method is overridden. The computation inside the doInBackground method, which determines whether the package is small or big based on user inputs, is executed in the background thread.

- handling exceptional states using own exceptions

```
public class LocalException extends Exception {
    no usages
    @Serial
    private static final long serialVersionUID = -3121055327488048910L;

    /**
     * Constructs a new LocalException with a default error message.
     */
    20 usages
    public LocalException() {
        super("Shipping can only be Prime Delivery for packages over 100kg.");
    }
}
```

This exception works when we try to choose StandardDelivery in small parcel if the weight is more than 100kg.

```
if (weight > 100) {
    try {
        throw new LocalException();
    } catch (LocalException ex) {
        costLabel.setText(ex.getMessage());
        costLabel.setForeground(Color.RED);
        return;
    }
}
```

- Factory Pattern

```
3 usages  Ilkiv Andrii
public static DeliveryOption createDeliveryOption(String option) {
    switch (option.toLowerCase()) {
        case "prime":
            return new PrimeDelivery();
        case "standart":
            return new StandartDelivery();
        default:
            throw new IllegalArgumentException("Invalid delivery option: " + option);
    }
}
```

```
    deliveryOption = createDeliveryOption("standart");
} else {
    deliveryOption = createDeliveryOption("prime");
}
```

It takes a string parameter option and uses a switch statement to determine the appropriate delivery option based on the value of option. It creates and returns an instance of the corresponding DeliveryOption subclass (PrimeDelivery or StandartDelivery).

And the last done

- providing a graphical user interface separated from application logic and with at least part of the event handlers created manually – counts as a fulfillment of two further criteria

Conclusion:

In conclusion, this Java Swing project has been a valuable learning experience, helping me gain proficiency in GUI development and software design patterns. I have successfully implemented a user-friendly interface for determining package sizes, utilizing layout managers, event handling, and user input validation. Throughout the project, I applied the Factory Pattern and the Model-View-Controller (MVC) pattern to improve code organization and reusability. Additionally, I explored multithreading techniques for efficient background computations and utilized Run-Time Type Information (RTTI) for class information during runtime. Overall, this project has enhanced my skills in GUI development, design patterns, multithreading, and Java fundamentals. I am excited to apply this knowledge in future projects and continue growing as a software developer.