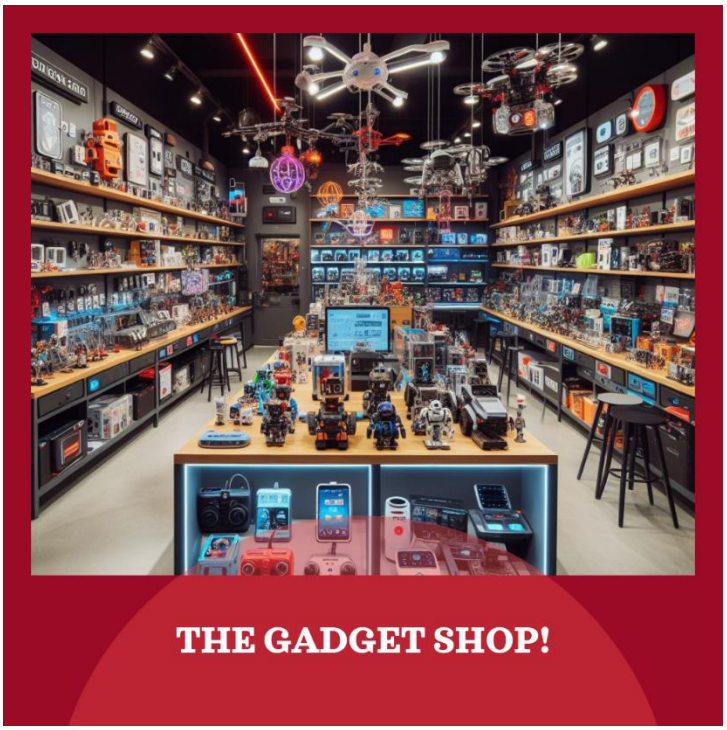
DEVELOPMENT THE GADGETSHOP APPLICATION IN JAVA

COURSEWORK PROGRAMING



By

Emilio Antonio Barrera Sepúlveda
22047090

London Metropolitan University
BSc Computing
CC4001 Programming
LONDON
2024

Table of Content

1. INTRODUCTION

The coursework entails crafting a comprehensive gadget management system for a retail shop, employing Object-Oriented Programming (OOP) principles in Java. It structures a class hierarchy with a parent class (Gadgets) and two subclasses (Phone and MP3), each tailored to encapsulate distinct gadget attributes and functionalities. Mastery of OOP concepts such as inheritance, encapsulation, and polymorphism are showcased, with emphasis on practical implementation through a Java Swing-based graphical user interface (GUI). Robust error handling mechanisms ensure system reliability, while adopting a Model-View-Controller (MVC) architecture fosters code modularity. The accompanying report encompasses a class diagram, pseudocode, and comprehensive details on testing procedures.

2. The GitHub links

https://github.com/emiliobs/TheGadgetShopJavaCoursework

3. Pseudocode

The pseudocode establishes a gadget management system with a base Gadget class and subclasses for Mobile and MP3 gadgets. It includes methods to add, display, and manipulate gadgets in a GadgetShop class. This framework enables adding, displaying, making calls (for Mobile), downloading, and deleting music (for MP3) in a gadget shop setting.

```
class Gadget:
    attributes:
        model: string
        price: decimal
        weight: integer
        size: string

    constructor(model, price, weight, size):
        initialize model, price, weight, size attributes with given values

    method getModel():
        return model

    method getPrice():
        return price

    method getWeight():
        return weight

    method getSize():
        return size
```

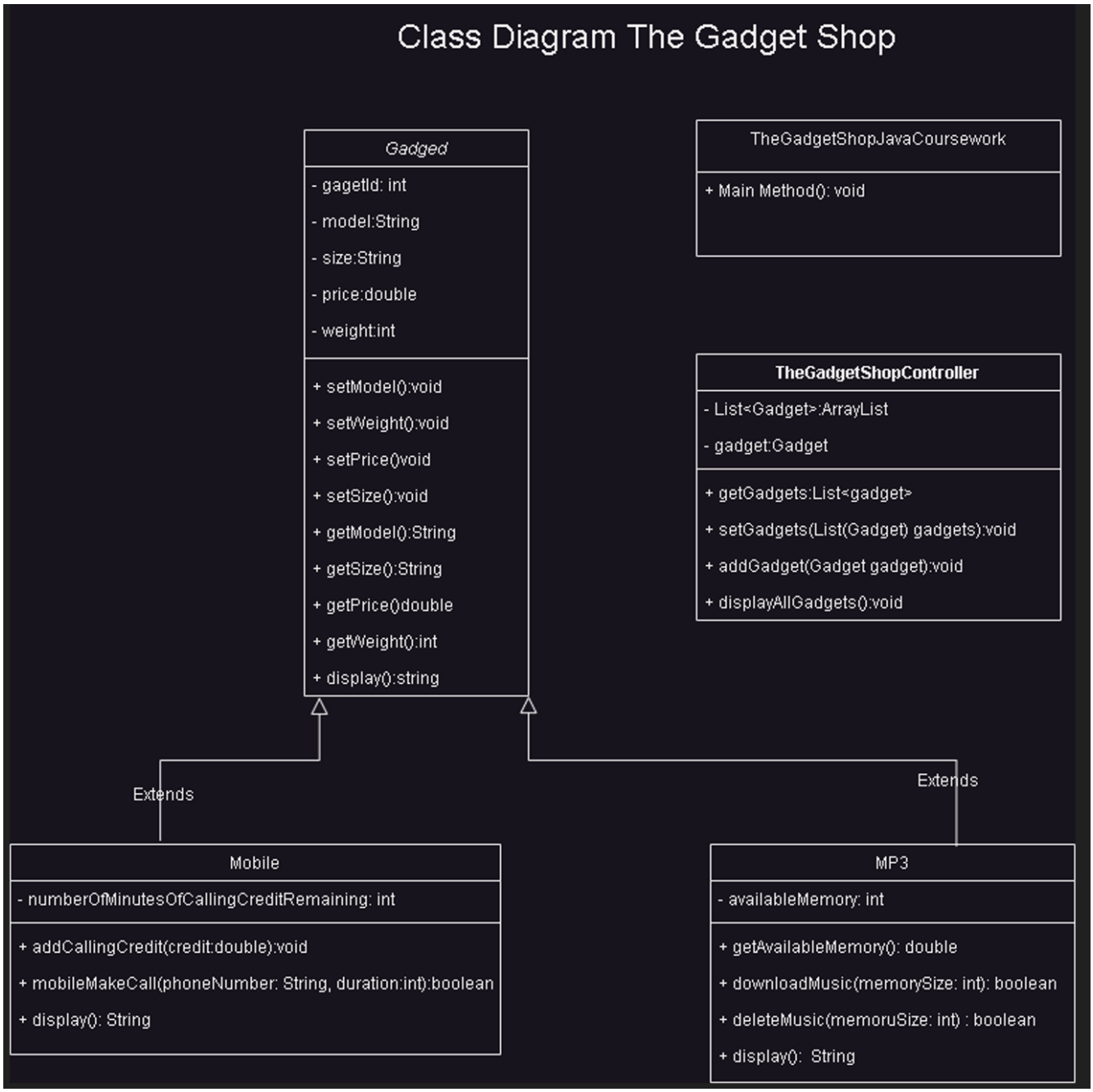
```
    method display():
        print "Model:", model
        print "Price:", price
        print "Weight:", weight
        print "Size:", size

class Mobile inherits Gadget:
    attribute:
        callingCredit: integer

    constructor(model, price, weight, size, callingCredit):
        call super constructor with model, price, weight, size
        initialize callingCredit attribute with given value

    method addCallingCredit(credit):
        if credit > 0:
            increase callingCredit by credit
        else:
            print "Please enter a positive amount for credit."

    method makeCall(phoneNumber, duration):
        if callingCredit >= duration:
            print "Making call to", phoneNumber, "for", duration, "minutes."
            decrease callingCredit by duration
        else:
            print "Insufficient credit to make the call."

    method display():
        call super display method
        print "Calling Credit:", callingCredit, "minutes"

class MP3 inherits Gadget:
    attribute:
        availableMemory: integer

    constructor(model, price, weight, size, availableMemory):
```

```
        call super constructor with model, price, weight, size
        initialize availableMemory attribute with given value

    method downloadMusic(memory):
        if memory <= availableMemory:
            decrease availableMemory by memory
            print "Music downloaded successfully."
        else:
            print "Not enough memory to download the music."

    method deleteMusic(memory):
        increase availableMemory by memory
        print "Music deleted successfully."

    method display():
        call super display method
        print "Available Memory:", availableMemory, "MB"

class GadgetShop:
    attribute:
        gadgets: array of Gadget objects

    method addMobile(model, price, weight, size, callingCredit):
        create a new Mobile object with given parameters
        add the Mobile object to gadgets array

    method addMP3(model, price, weight, size, availableMemory):
        create a new MP3 object with given parameters
        add the MP3 object to gadgets array

    method clearTextFields():
        clear gadgets array

    method displayAll():
        for each gadget in gadgets array:
            call gadget's display method
```

```
method makeCall(displayNumber, phoneNumber, duration):
    if displayNumber is valid:
        retrieve the gadget from gadgets array using displayNumber
        if gadget is Mobile:
            call makeCall method with phoneNumber and duration

method downloadMusic(displayNumber, memory):
    if displayNumber is valid:
        retrieve the gadget from gadgets array using displayNumber
        if gadget is MP3:
            call downloadMusic method with memory

method deleteMusic(displayNumber, memory):
    if displayNumber is valid:
        retrieve the gadget from gadgets array using displayNumber
        if gadget is MP3:
            call deleteMusic method with memory
```

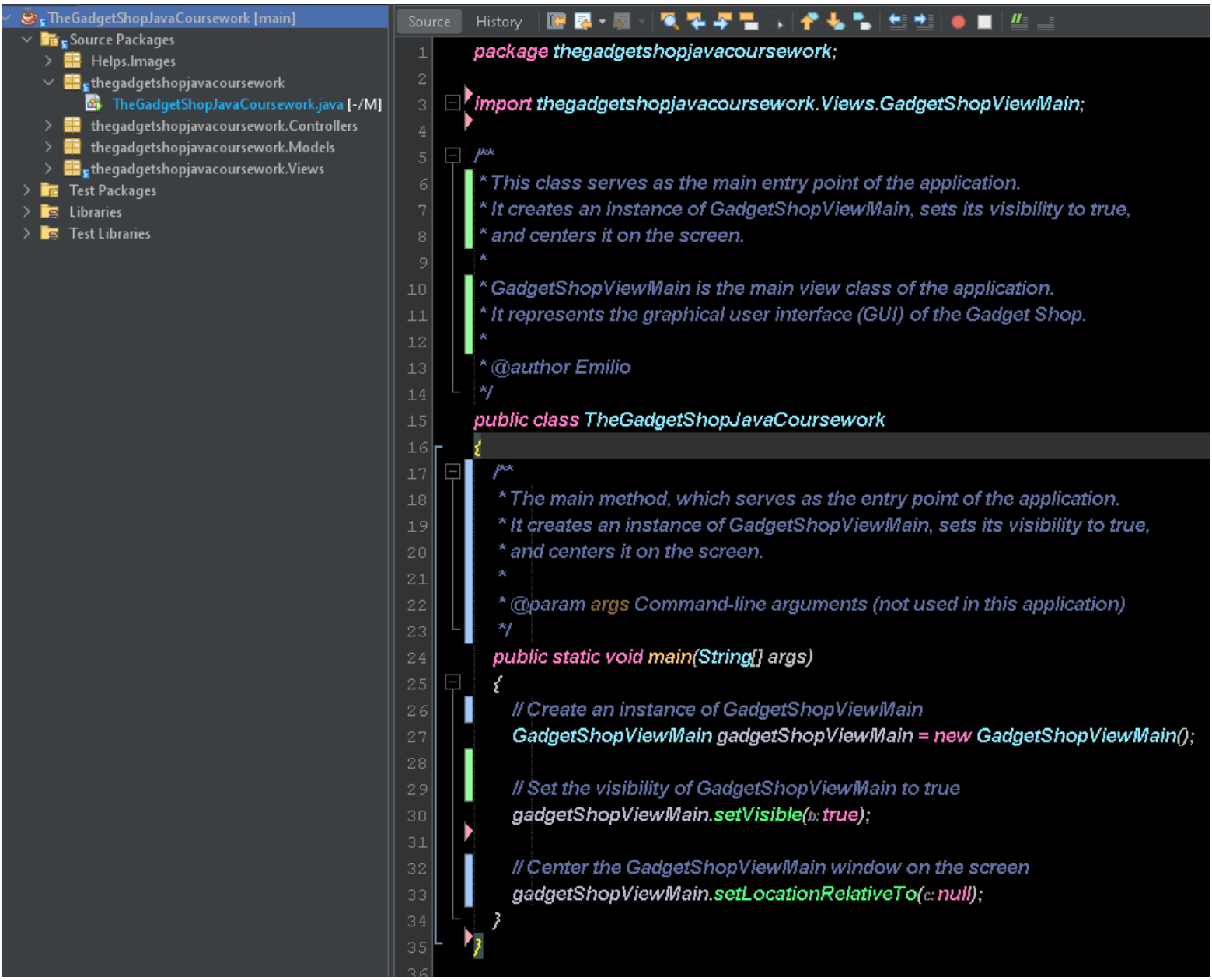
4. UML Class diagrams the gadget shop application

This class figure shows five classes as parent child relationship shown below:



5. Main entry point of the application

This class serves as the main entry point of the application. It creates an instance of GadgetShopViewMain, sets its visibility to true, and centers it on the screen. GadgetShopViewMain is the main view class of the application. It represents the graphical user interface (GUI) of the Gadget Shop.

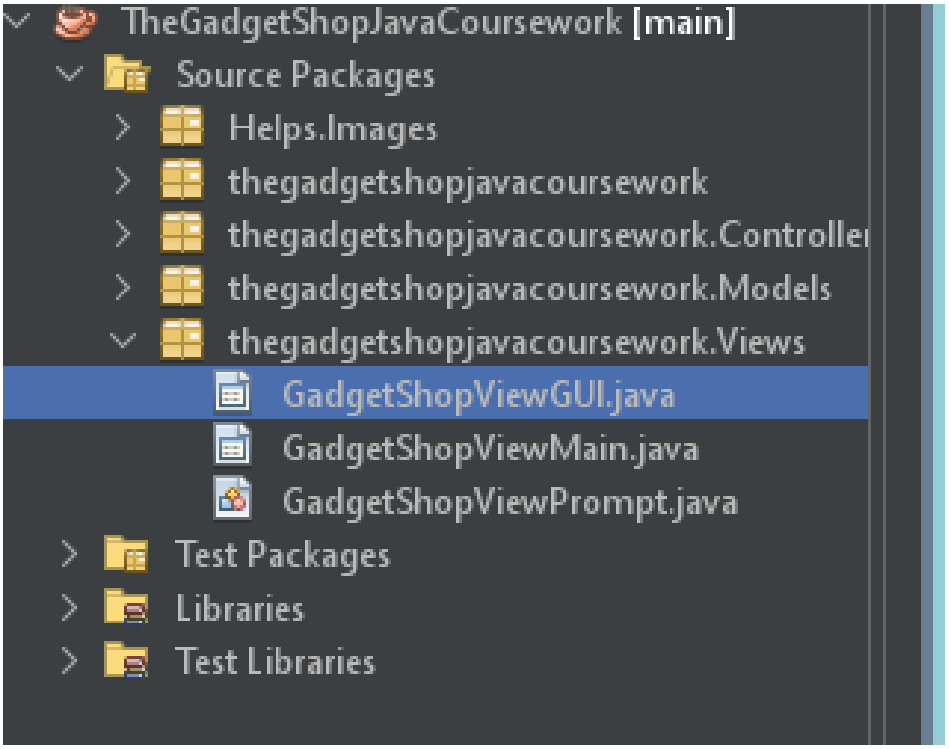


6. Design pattern MVC

MVC is a design pattern commonly used in software development to separate the application logic into three interconnected components: Model, View, and Controller. Here's a brief explanation of each component and how they work together:

6.1 View

- Represents the user interface (GUI) and displays the data to the user.
- Listens for changes in the Model and updates the UI accordingly.
- Should not contain business logic.



6.1.1 The Gadget Shop GUI Main

The provided code appears to be a snippet from a Java Swing application, likely representing the declaration of components (buttons, labels, text fields, etc.) in a graphical user interface (GUI). Each component seems to be associated with a specific functionality.

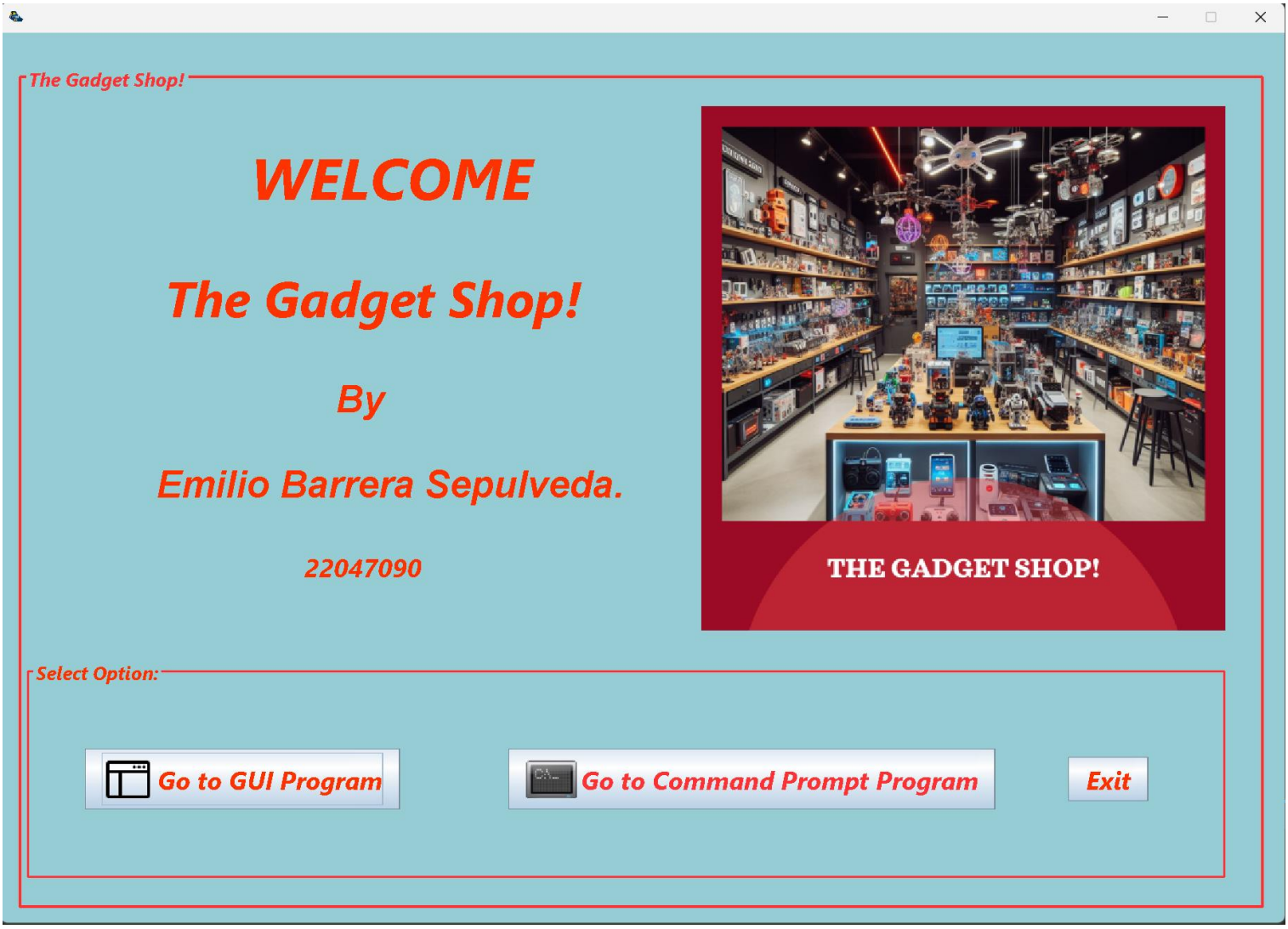
When launching the program, you're welcomed by the main GUI featuring a personalized message from the developer. Additionally, you're presented with three options:

- Navigate to the GUI version of the program.
- Enter the Prompt Command Program to access the program without GUI.
- Exit the program to terminate its execution.

6.1.1.1 Main Jpanel Container.

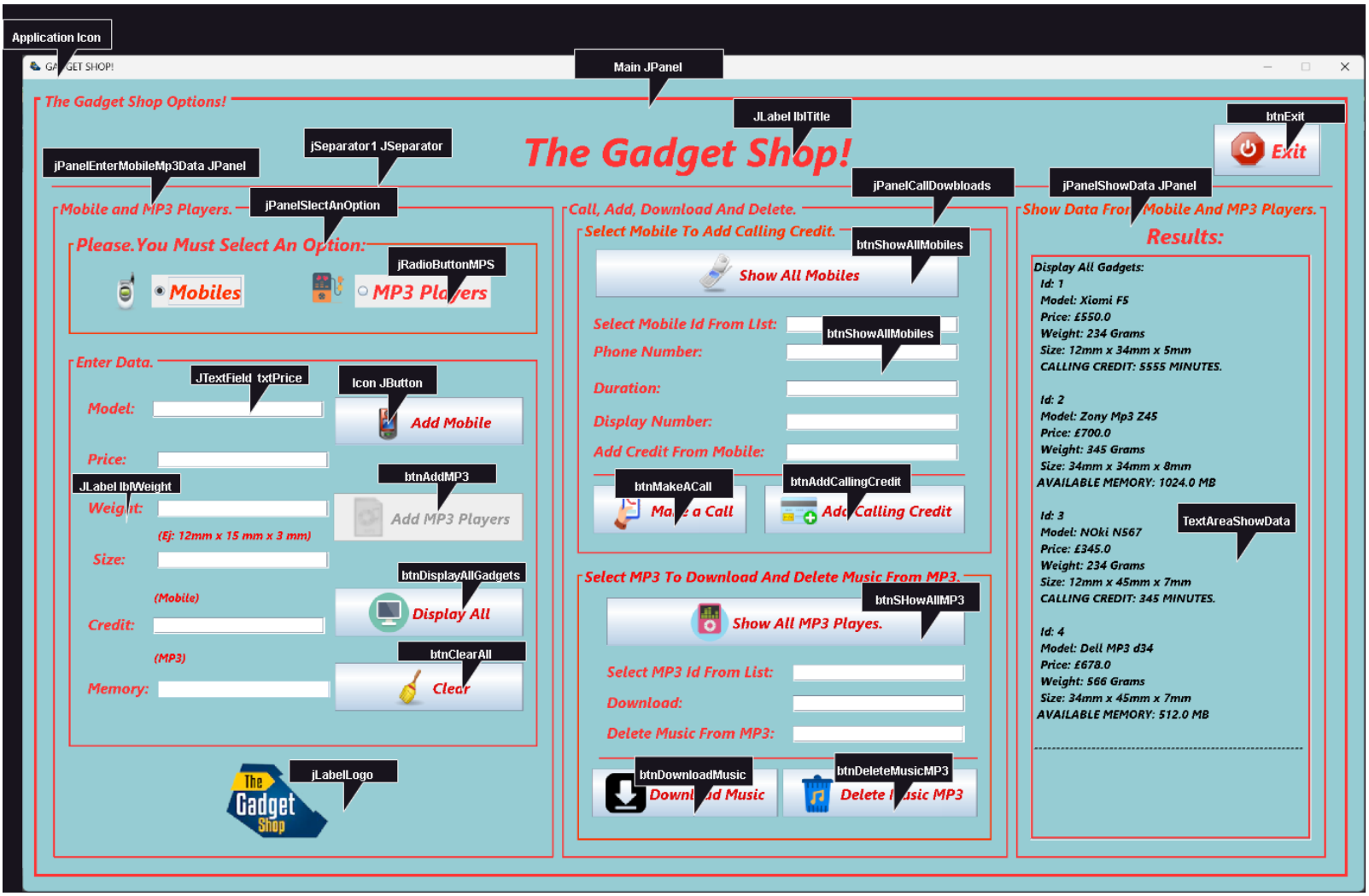
This panel consists of 6 JLabel, 1other JPanels with three 3 JButton:

- jPanelContainer
- lblTitleGadgetShop
- lblTitleGadgetShop1
- lblBy
- lblName
- lblIdCard
- lblImage
- btnCommandPrompt
- btnGuiProgram
- btnExit.


The Gadget Shop!
WELCOME
The Gadget Shop!
By
Emilio Barrera Sepulveda.
22047090
THE GADGET SHOP!
Select Option:
Go to GUI Program
Go to Command Prompt Program
Exit

6.1.2 The Gadget Shop GUI

Upon selecting "Go to GUI program," users access the GUI program interface, initially featuring disabled controls. This prompts users to select between adding mobile phones or MP3 players, activating the controls accordingly. Enabled controls facilitate various data manipulation tasks, including adding devices, making calls, deleting music, and displaying all devices conveniently. The provided text further elaborates on the JPanel declarations within the GUI, organizing components for options selection, data entry, downloads, and information display. Sub-panels, such as jPanelEnterData and jPanelCallDownloads, house components like radio buttons, text fields, and buttons for specific actions, ensuring efficient user interaction and data management.



6.1.3 The Gadget Shop Command Prompt Program

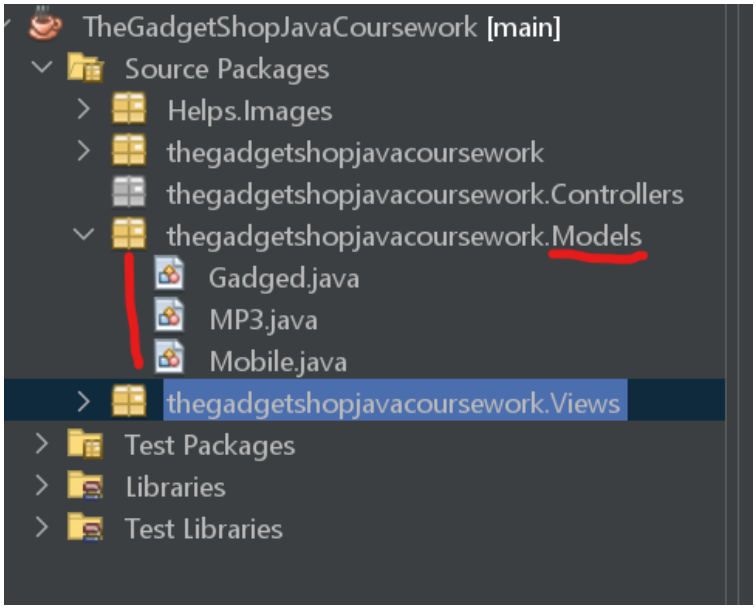
After selecting "Go to Command Prompt Program," you'll be directed to the Command Prompt interface. Here, you'll find the Main Menu of the program, encompassing all the functionalities of The GadgetShop program. This includes options such as adding items, displaying all available products, managing calling credits, making calls, downloading music, and deleting music from mobile devices and MP3 players.

Important Note:

As the focus of the coursework is on the development of the program in GUI (Graphical User Interface) rather than the Command Prompt interface, detailed discussion regarding the application's functionality in the Command Prompt will not be provided.

6.2 Models

- Represents the application's data and business logic.
- Manages the data, logic, and rules of the application.
- Notifies the View when the data changes.



6.2.1 Gadget.java Class

The Gadget class, located within the thegadgetshopjavacoursework.Models package, serves as a foundation for managing gadgets in the application. It encompasses essential attributes such as gadgetId, model, price, weight, and size. Constructors facilitate gadget creation with default or specified values. Accessor methods ensure seamless retrieval and modification of attributes, while the display method presents a formatted summary of gadget details. This class provides a structured approach to handle gadget information within the Gadget Shop application, enabling creation, manipulation, and presentation of gadget data with clarity and efficiency.

```java
package thegadgetshopjavacoursework.Models;

/**
 * Model class representing a Gadget. A Gadget object encapsulates information
 * about a particular gadget, including its ID, model, price, weight, and size.
 *
 * Gadget objects are used to represent gadgets within the Gadget Shop
 * application.
 *
 * @author Emilio
 */
public class Gadget
{

    // Instance Variables
    private static int gadgetCounter = -1; // Counter for generating unique gadget IDs

    private int gadgetId; // Unique ID of the gadget
    private String model; // Model of the gadget
    private double price; // Price of the gadget
    private int weight; // Weight of the gadget
    private String size; // Size of the gadget

    // Constructors
    /**
     * Default constructor for creating a Gadget object. Increments the gadget
     * counter to generate a unique ID for the gadget.
     */
    public Gadget()
    {
        // Set the gadgetId using gadgetCounter and increment gadgetCounter
        this.gadgetId = gadgetCounter++;
```

```java
 * Increments the gadget counter to generate a unique ID for the gadget.
 *
 * @param gadgetId Unique ID of the gadget
 * @param model Model of the gadget
 * @param price Price of the gadget
 * @param weight Weight of the gadget
 * @param size Size of the gadget
 */
public Gadget(int gadgetId, String model, double price, int weight, String size)
{
    // Set the gadgetId using gadgetCounter and increment gadgetCounter
    this.gadgetId = gadgetCounter++;

    // Initialize other attributes with the provided values
    this.model = model;
    this.price = price;
    this.weight = weight;
    this.size = size;
}

// Accessor methods for Gadget attributes
public int getGadgetId()
{
    return gadgetId;
}

public void setGadgetId(int gadgetId)
{
    this.gadgetId = gadgetId;
}

public String getSize()
{
    return size;
}
```

```
public void setSize(String size)
{
    this.size = size;
}

public String getModel()
{
    return model;
}

public void setModel(String model)
{
    this.model = model;
}

public double getPrice()
{
    return price;
}

public void setPrice(double price)
{
    this.price = price;
}

public int getWeight()
{
    return weight;
}

public void setWeight(int weight)
{
    this.weight = weight;
}

// Method to display Gadget details
```

```
    // Method to display Gadget details
    /**
     * Generates a string representation of the Gadget object, including its
     * details.
     *
     * @return A string containing details of the gadget (ID, model, price,
     * weight, size)
     */
    public String display()
    {
        // Construct and return a string containing the gadget's attributes
        return " Id: " + gadgetId + "\n " + " Model: " + model + "\n " + " Price: £" + price + "\n
" + " Weight: " + weight + " Grams" + "\n " + " Size: " + size;
    }
}
```

6.2.2 Mobile.java Class

The Mobile class extends Gadget, focusing on mobile representation in the Gadget Shop. It includes numberOfMinutesOfCallingCreditRemaining to track calling minutes. Constructors set mobile attributes and manage calling credit. Accessors handle credit retrieval/modification, with methods ensuring non-negative values and managing calls based on credit. Display method showcases mobile details, enabling seamless integration into the application's framework with inherited and specific functionalities.

```java
package thegadgetshopjavacoursework.Models;

/**
 * Subclass Mobile inheriting from Gadget. Represents a mobile device with
 * additional functionality such as calling credit management. Extends the
 * functionality of the Gadget class.
 *
 * Mobile objects are used to represent mobile devices within the Gadget Shop
 * application. They inherit attributes and methods from the Gadget class.
 *
 * @author Emilio
 */
public class Mobile extends Gadget
{

    private int numberOfMinutesOfCallingCreditRemaining;  // Number of calling credit
minutes remaining

    // Constructors
    /**
     * Default constructor for creating a Mobile object. Initializes the object
     * with default values.
     */
    public Mobile()
    {

    }
    /**
     * Constructor for initializing Mobile attributes and Gadget attributes with
     * specified values.
     *
     * @param numberOfMinutesOfCallingCreditRemaining Number of calling credit
     * minutes remaining
     * @param gadgetId Unique ID of the mobile gadget
     * @param model Model of the mobile gadget
     * @param price Price of the mobile gadget
```

```
    * @param weight Weight of the mobile gadget
    * @param size Size of the mobile gadget
    */
   public Mobile(int numberOfMinutesOfCallingCreditRemaining, int gadgetId, String
model, double price, int weight, String size)
   {
      // Call the constructor of the superclass (Gadget) with specified parameters
      super(gadgetId, model, price, weight, size);

      // Initialize the numberOfMinutesOfCallingCreditRemaining attribute with the
provided value
      this.numberOfMinutesOfCallingCreditRemaining =
numberOfMinutesOfCallingCreditRemaining;
   }

   /**
    * Constructor for initializing only Mobile attributes with specified
    * values.
    *
    * @param numberOfMinutesOfCallingCreditRemaining Number of calling credit
    * minutes remaining
    */
   public Mobile(int numberOfMinutesOfCallingCreditRemaining)
   {
      this.numberOfMinutesOfCallingCreditRemaining =
numberOfMinutesOfCallingCreditRemaining;
   }
   /**
    * Retrieves the remaining calling credit for the mobile device.
    *
    * @return The number of minutes of calling credit remaining.
    */
   public int getNumberOfMinutesOfCallingCreditRemaining()
   {
      // Return the current value of numberOfMinutesOfCallingCreditRemaining attribute
      return numberOfMinutesOfCallingCreditRemaining;
```

```
    }

    /**
     * Sets the remaining calling credit for the mobile device.
     *
     * @param numberOfMinutesOfCallingCreditRemaining The number of minutes of
     * calling credit remaining.
     */
    public void setNumberOfMinutesOfCallingCreditRemaining(int
numberOfMinutesOfCallingCreditRemaining)
    {
        // Check if the provided number of minutes of calling credit remaining is non-
negative
        if (numberOfMinutesOfCallingCreditRemaining >= 0)
        {
            // If it's non-negative, set the value of
numberOfMinutesOfCallingCreditRemaining
            this.numberOfMinutesOfCallingCreditRemaining =
numberOfMinutesOfCallingCreditRemaining;
        }
        else
        {
            // If it's negative, print an error message
            System.out.println("");
            System.out.println("***** - Sorry!. Please enter a positive amount for adding
credit. - *****");
        }
    }

    /**
     * Adds calling credit to the mobile device.
     *
     * @param credit The amount of calling credit to add.
     */
    public void addCallingCredit(int credit)
    {
```

```
        // Check if the credit amount is non-negative
        if (credit >= 0)
        {
            // If the credit amount is non-negative, add it to the remaining calling credit
            numberOfMinutesOfCallingCreditRemaining += credit;
        }
        else
        {
            // If the credit amount is negative, print an error message
            System.out.println("");
            System.out.println("***** - Sorry!. Please enter a positive amount for adding
credit. - *****");
        }
    }

    /**
     * Simulates making a call from a mobile device.
     *
     * @param phoneNumber The phone number being called.
     * @param duration The duration of the call in minutes.
     * @return True if the call can be made (sufficient calling credit), false
     * otherwise.
     */
    public boolean mobileMakeCall(int phoneNumber, int duration)
    {
        // Check if there is sufficient calling credit remaining and if the duration is within the
available credit
 if (numberOfMinutesOfCallingCreditRemaining != 0 &&
numberOfMinutesOfCallingCreditRemaining >= duration)
        {
            // Output information about the call to the console
            System.out.println("CALLING NUMBER: " + phoneNumber + " FOR " + duration
+ " MINUTES.");
            // Deduct the call duration from the remaining calling credit
            numberOfMinutesOfCallingCreditRemaining -= duration;
            // Return true indicating the call was successful
```

```
            return true;
        }
        else
        {
            // Return false indicating the call cannot be made due to insufficient calling credit
            return false;
        }
    }
    /**
     * Overrides the display method to include information about the remaining
     * calling credit.
     *
     * @return A string representation of the mobile device's information,
     * including the calling credit.
     */
    @Override
    public String display()
    {
        // Calls the display method of the superclass (presumably to display basic gadget
information)
        // Concatenates information about the remaining calling credit to the returned string
        return super.display() + "\n " + " CALLING CREDIT: " +
numberOfMinutesOfCallingCreditRemaining + " MINUTES.";

    }

}
```

6.2.3 PM3.java Class

The MP3 class, a subclass of Gadget, manages MP3 player gadgets in the Gadget Shop. It adds availableMemory attribute to store MP3 memory. Constructors initialize with default or specific memory. Accessors retrieve and modify memory. Methods handle music operations based on memory. Display method showcases MP3 details, extending Gadget's functionality for MP3 management.

```
package thegadgetshopjavacoursework.Models;

/**
 * Subclass MP3 inheriting from Gadget. Represents an MP3 player gadget with
 * additional functionality such as music management. Extends the functionality
 * of the Gadget class.
 *
 * MP3 objects are used to represent MP3 players within the Gadget Shop
 * application. They inherit attributes and methods from the Gadget class.
 *
 * @author Emilio
 */
public class MP3 extends Gadget
{

    private double availableMemory; // Available memory in the MP3 player

    // Constructors
    /**
     * Default constructor for creating an MP3 object. Initializes the object
     * with default values.
     */
    public MP3()
    {
    }

    /**
     * Constructor for initializing only the available memory attribute of the
     * MP3 player.
     *
     * @param availableMemory Available memory of the MP3 player in megabytes
     * (MB)
     */
    public MP3(double availableMemory)
    {
        // Set the available memory for the MP3 player using the provided value
```

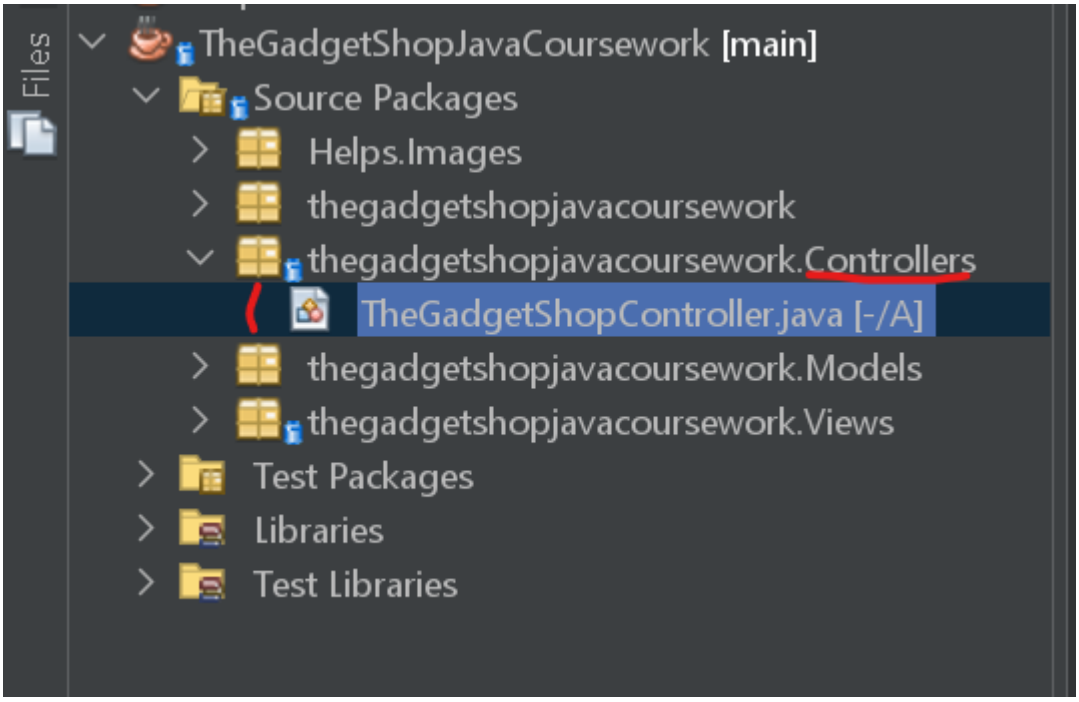
```
        this.availableMemory = availableMemory;
    }

    /**
     * Constructor for initializing both MP3 attributes and Gadget attributes
     * with specified values.
     *
     * @param availableMemory Available memory of the MP3 player in megabytes
     * (MB)
     * @param gadgetId Unique ID of the MP3 gadget
     * @param model Model of the MP3 gadget
     * @param price Price of the MP3 gadget
     * @param weight Weight of the MP3 gadget
     * @param size Size of the MP3 gadget
     */
    public MP3(double availableMemory, int gadgetId, String model, double price, int
weight, String size)
    {
        // Call the constructor of the superclass (Gadget) with specified parameters
        super(gadgetId, model, price, weight, size);

        // Initialize the available memory attribute for the MP3 object with the provided
value
        this.availableMemory = availableMemory;
    }

    // Accessor methods for availableMemory
    public double getAvailableMemory()
    {
        return availableMemory;
    }

    public double setAvailableMemory(double availableMemory)
    {
        return this.availableMemory = availableMemory;
    }
```

```
    // Method to download music
    /**
     * Downloads music to the MP3 player.
     *
     * @param memoryRequired Memory required for downloading the music in
     * megabytes (MB)
     * @return true if the music is successfully downloaded, false otherwise
     */
    public boolean downloadMusic(double memoryRequired)
    {
        // Check if the required memory for downloading the music is less than or equal to
the available memory
        if (memoryRequired <= availableMemory)
        {
            // If there is sufficient available memory, deduct the required memory from the
available memory
            availableMemory -= memoryRequired;
            // Return true indicating the music was successfully downloaded
            return true;
        }
        // Return false indicating the music cannot be downloaded due to insufficient
available memory
        return false;
    }
    // Method to delete music (Free up memory in the MP3 player)
    /**
     * Deletes music from the MP3 player, freeing up memory.
     *
     * @param memoryFreed Memory freed by deleting music in megabytes (MB)
     * @return true if the music is successfully deleted, false otherwise
     */
    public boolean deleteMusic(int memoryFreed)
    {
        // Check if the memory freed after deleting the music is non-negative
        if (memoryFreed >= 0)
```

```
        {
            // If it's non-negative, add the freed memory to the available memory
            availableMemory += memoryFreed;
            // Return true indicating the music was successfully deleted
            return true;
        }
        else
        {
            // If it's negative, return false indicating deletion failure
            return false;
        }
    }
    // Override display method to include available memory
    /**
     * Generates a string representation of the MP3 player, including its
     * details and available memory.
     *
     * @return A string containing details of the MP3 player (ID, model, price,
     * weight, size) and available memory
     */
    @Override
    public String display()
    {
        // Calls the display method of the superclass (presumably Gadget) to get basic
gadget information,
        // Concatenates information about the available memory to the returned string
        return super.display() + "\n " + "AVAILABLE MEMORY: " + availableMemory + "
MB";
    }

}
```

6.3 Controllers

- Acts as an intermediary between the Model and the View.
- Receives user input from the View and updates the Model accordingly.
- Listens for changes in the Model and updates the View.



6.3.1 TheGadgetShopController.java Class

The TheGadgetShopController class, nestled in the thegadgetshopjavacoursework.Controllers package, centrally manages gadgets in the Gadget Shop application. It features an ArrayList to hold Gadget objects and a reference variable for a Gadget. With a default constructor initializing necessary components, it provides methods to add gadgets to the list, retrieve them, and display their details. Acting as an intermediary between the user interface and data model, this controller ensures smooth gadget management, facilitating tasks such as addition and display, crucial for seamless operation and user interaction within the application's framework.

```java
package thegadgetshopjavacoursework.Controllers;

import java.util.ArrayList;
import java.util.List;

import thegadgetshopjavacoursework.Models.Gadget;

/**
 * Controller class responsible for managing gadgets within the Gadget Shop
 * application. TheGadgetShopController class provides methods to add gadgets to
 * a list, retrieve the list of gadgets, and display all gadgets in the list.
 *
 * This class serves as an intermediary between the user interface and the data
 * model, facilitating the management of gadgets.
 *
 * It contains functionality to interact with Gadget and Mobile objects,
 * including adding gadgets to a list and displaying all gadgets.
 *
 * @author Emilio
 */
public class TheGadgetShopController
{

    // Create an empty ArrayList to store Gadget objects
    private List<Gadget> gadgets;
    Gadget gadget;

    /**
     * Default constructor for creating a TheGadgetShopController object.
     * Initializes the gadgets list and a reference to a Gadget object.
     */
    public TheGadgetShopController()
    {
        gadgets = new ArrayList<>();
```

```java
/**
 * Retrieves the list of gadgets.
 *
 * @return The list of gadgets.
 */
public List<Gadget> getGadgets()
{
    return gadgets;
}

/**
 * Sets the list of gadgets.
 *
 * @param gadgets The list of gadgets to set.
 */
public void setGadgets(List<Gadget> gadgets)
{
    this.gadgets = gadgets;
}

/**
 * Adds a gadget to the list of gadgets.
 *
 * @param gadget The gadget to add to the list.
 */
public void addGadget(Gadget gadget)
{
    gadgets.add(gadget);
}

/**
 * Displays all gadgets in the list. If the list is empty, it prints a
 * message indicating that no gadgets are available.
 */
public void displayAllGadgets()
```

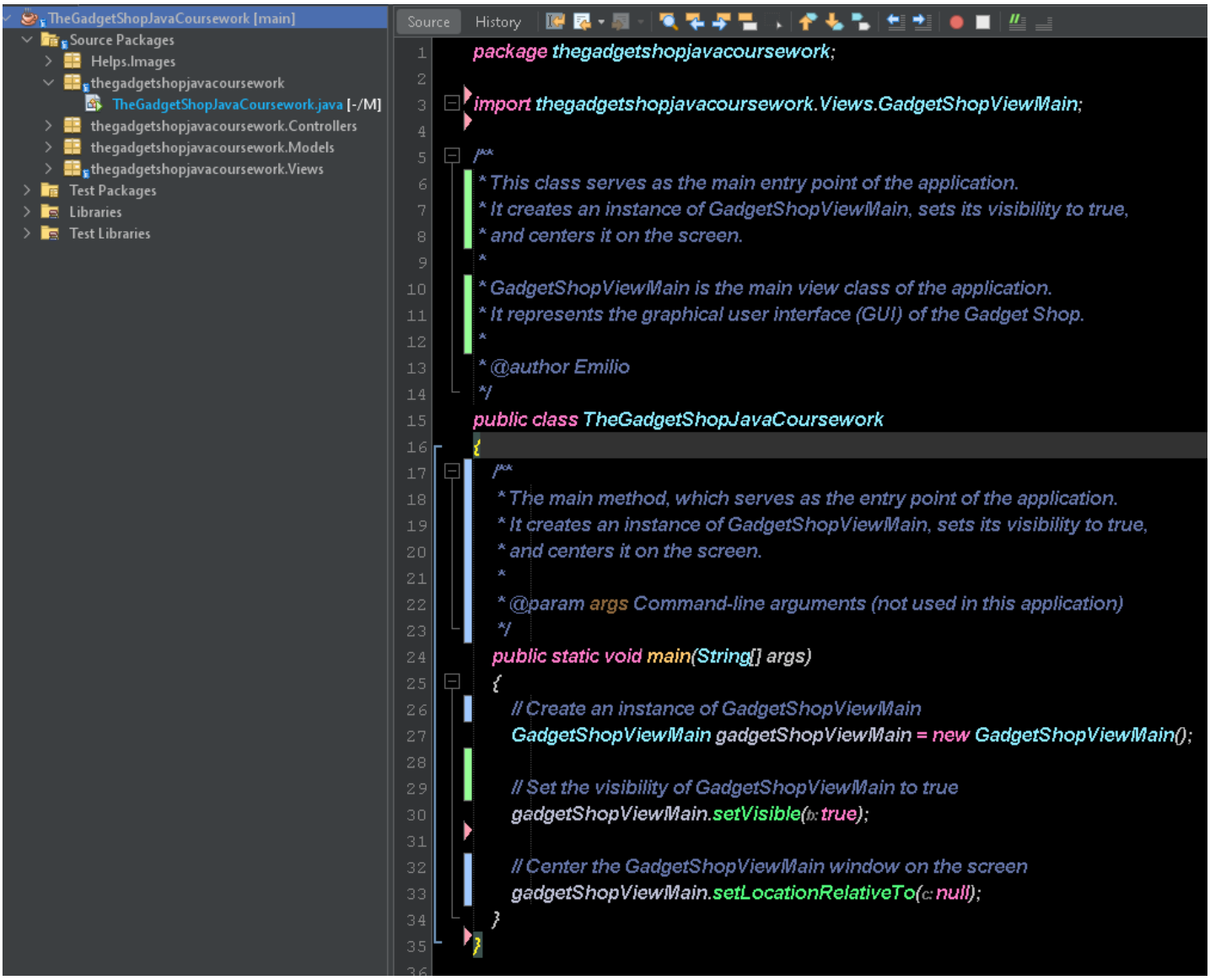
```
    {
        // Print a header for the display
        System.out.println("============= DISPLAY ALL GADGETS
==============");

        // Check if the collection of gadgets is empty
        if (gadgets.isEmpty())
        {
            // If the collection is empty, print a message indicating no gadgets are available
            System.out.println("Sorry!. No Gadgets available to display.");
        }

        // Iterate over each gadget in the collection
        for (Gadget gadget : gadgets)
        {
            // Display information about the current gadget
            System.out.println(gadget.display());
            System.out.println(""); // Print a blank line for better readability
        }

        // Print a footer for the display
        System.out.println("==================================================
");
    }
}
```

7. Functionality

Code is provided for the following button handling methods, and I will now proceed to describe each method in detail.

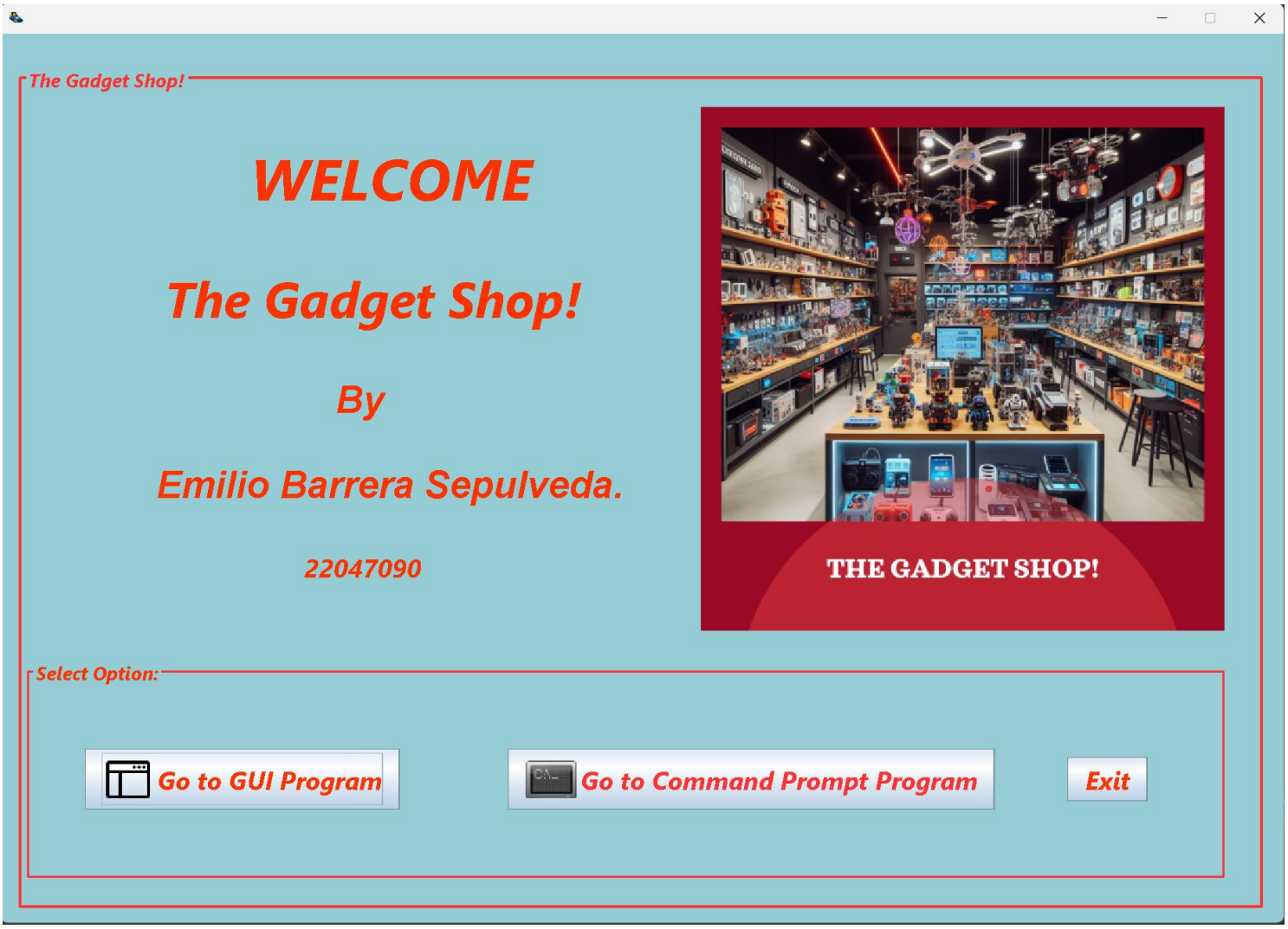
7.1 Main entry point of the application

This class serves as the main entry point of the application. It creates an instance of GadgetShopViewMain, sets its visibility to true, and centers it on the screen. GadgetShopViewMain is the main view class of the application. It represents the graphical user interface (GUI) of the Gadget Shop.



7.2 Getting the display number from the GUI (Main)

The GadgetShopViewMain class, located in the thegadgetshopjavacoursework.Views package, extends javax.swing.JFrame, indicating it as a GUI window. Its constructor initializes GUI components and sets the application icon. Event handlers include btnGuiProgramActionPerformed() for launching a GUI program, btnCommandPromptActionPerformed() for activating a command prompt, and btnExitActionPerformed() for closing the window. Variable declarations include Swing components like buttons and panels. This code serves as the main GUI interface for the gadget shop application, facilitating user interaction through various event handlers and components. It likely collaborates with controllers and other views within the application's framework.



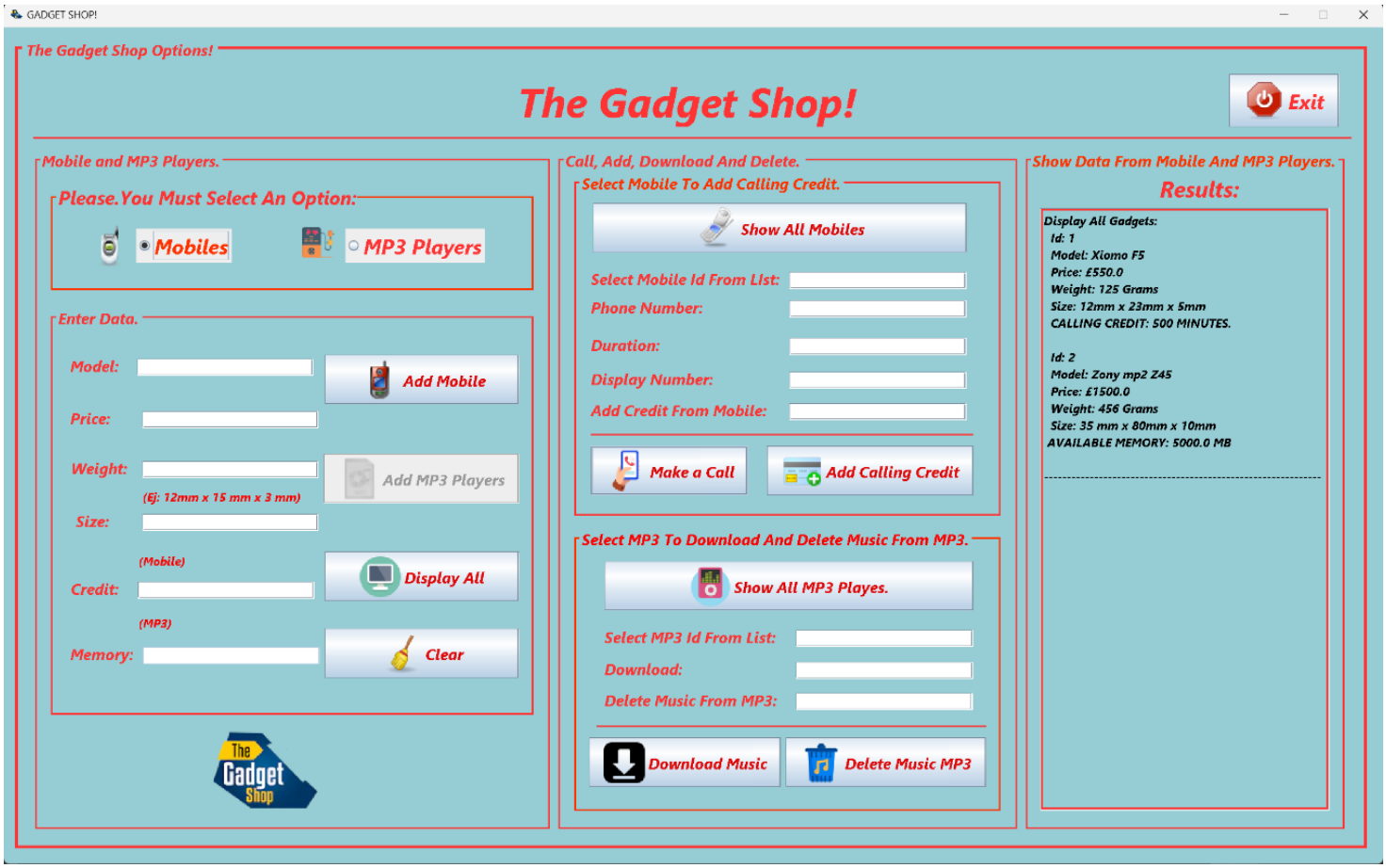
```
package thegadgetshopjavacoursework.Views;

import javax.swing.ImageIcon;
import thegadgetshopjavacoursework.Controllers.TheGadgetShopController;

/**
 *
 * @author Emilio
 */
public class GadgetShopViewMain extends javax.swing.JFrame
{

    /**
     * Creates new form GadgetShopViewMain
     */
    public GadgetShopViewMain()
    {
        initComponents(); // Initialize the GUI components (presumably generated by a
GUI builder tool)

        // Set the application icon using the ImageIcon class and getResource() method
        // The path to the icon resource is assumed to be "/Helps/Images/logo.png" relative
to the class package
        setIconImage(new
ImageIcon(getClass().getResource("/Helps/Images/logo.png")).getImage());
    }

 /**
     * Event handler for the action performed when the "Exit" button is clicked.
     * Disposes of the current window.
     */
    private void btnExitActionPerformed(java.awt.event.ActionEvent
evt)
    {
        // Disposes of the current window
```

```
    }

    /**
     * Event handler for the action performed when the "Command Prompt" button
     * is clicked. Shows the program in the command prompt.
     */
    private void btnCommandPromptActionPerformed(java.awt.event.ActionEvent
evt)
    {
        // Create an instance of TheGadgetShopController
        TheGadgetShopController gadgetShopController = new
TheGadgetShopController();

        // Create an instance of GadgetShopViewPrompt
        GadgetShopViewPrompt viewPrompt = new GadgetShopViewPrompt();

        // Run the view in the command prompt
        viewPrompt.run();
    }

    /**
     * Event handler for the action performed when the "GUI Program" button is
     * clicked. Displays the program in the graphical user interface (GUI).
     */
    private void btnGuiProgramActionPerformed(java.awt.event.ActionEvent
evt)
    {

/// Create an instance of GadgetShopViewGUI, which is presumably a GUI view for the
program
        GadgetShopViewGUI gadgetShop = new GadgetShopViewGUI();

        // Set the visibility of the GUI view to true, making it visible to the user
        gadgetShop.setVisible(true);

        // Set the location of the GUI view to be centered on the screen
```

```
        gadgetShop.setLocationRelativeTo(null);
    }

    /**
     * @param args the command line arguments
     */

    // Variables declaration - do not modify
    public javax.swing.JButton btnCommandPrompt;
    public javax.swing.JButton btnExit;
    public javax.swing.JButton btnGuiProgram;
    public javax.swing.JPanel jPanelContainer;
    public javax.swing.JPanel jPanelMain;
    public javax.swing.JPanel jPanelOptionProgram;
    public javax.swing.JLabel lblBy;
    public javax.swing.JLabel lblIdCard;
    public javax.swing.JLabel lblImage;
    public javax.swing.JLabel lblName;
    public javax.swing.JLabel lblTitleGadgetShop;
    public javax.swing.JLabel lblTitleGadgetShop1;
    // End of variables declaration
}
```

7.3 The Gadget Shop GUI

Upon selecting "Go to GUI Program" from the main interface, you will seamlessly transition to The Gadget Shop's GUI program interface. Here, you'll have access to a comprehensive array of functionalities including adding mobile devices, incorporating MP3 players, displaying all items, and effortlessly clearing fields, alongside other program features.



7.4 Please. You Must Select an Option: Radio button

The method jRadioButtonMPSActionPerformed manages the selection of the "MP3" radio button, updating the UI accordingly. It confirms the selection with a message, enables/disables relevant text fields (txtModel, txtPrice, txtWeight, txtSize, txtCredit, txtMemory), and adjusts button accessibility (btnAddMobile enabled, btnAddMP3 disabled) based on the choice between mobiles and MP3s.

```
/**
    * Event handler triggered when the "Mobile" radio button is selected.
    *
    * @param evt The ActionEvent representing the action performed event.
    */

   private void
jRadioButtonMObileActionPerformed(java.awt.event.ActionEvent
evt)
   {
       // Display a message indicating that mobiles are selected
       JOptionPane.showMessageDialog(null, "Brilliant!. Your Select Was Add
Mobiles.");

       // Enable text fields related to mobile information
       txtModel.setEnabled(true);
       txtPrice.setEnabled(true);
       txtWeight.setEnabled(true);
       txtSize.setEnabled(true);
       txtCredit.setEnabled(true);

       // Disable text field related to MP3 memory (assuming it's related to
MP3s)
       txtMemory.setEnabled(false);

       // Enable buttons related to adding mobiles and disable buttons related to
adding MP3s
       btnAddMobile.setEnabled(true);
       btnAddMP3.setEnabled(false);
   }
```

```
 /**
   * Event handler triggered when the "MP3" radio button is selected.
   * @param evt The ActionEvent representing the action performed event.
   */
private void jRadioButtonMPSActionPerformed(java.awt.event.ActionEvent
evt)
  {
      // Display a message indicating that MP3 players are selected
      JOptionPane.showMessageDialog(null, "Brilliant!. Your Select Was Add MP3
Players.");

      // Enable text fields related to MP3 player information
      txtModel.setEnabled(true);
      txtPrice.setEnabled(true);
      txtWeight.setEnabled(true);
      txtSize.setEnabled(true);

      // Disable text fields related to mobile credit (assuming it's related to mobiles)
      txtCredit.setEnabled(false);

      // Enable text field related to MP3 memory
      txtMemory.setEnabled(true);
      // Enable button related to adding MP3s and disable button related to adding
mobiles
      btnAddMobile.setEnabled(false);
      btnAddMP3.setEnabled(true);
}
```

7.5 Adding a mobile

The btnAddMobileActionPerformed method manages adding a new mobile gadget. It retrieves input from text fields, validates for empty fields, and creates a Mobile object with input properties. Exception handling ensures reliability, displaying error messages if needed. Upon success, it confirms with a message and clears input fields.

```
/**
    * This method is an event handler for the button "btnAddMobile". It is
    * invoked when the user clicks on the button to add a mobile gadget.
    *
    * @param evt The ActionEvent representing the user's action (clicking the
    * button)
    */
   private void btnAddMobileActionPerformed(java.awt.event.ActionEvent
evt)
   {

       // Retrieving input values from text fields
       String model = txtModel.getText();
       String price = txtPrice.getText();
       String weight = txtWeight.getText();
       String size = txtSize.getText();
       String credit = txtCredit.getText();

       try
       {
           // Input validation
           if (model.trim().isBlank() && model.trim().isEmpty())
           {
               // Display an error message if the model field is empty
               JOptionPane.showMessageDialog(null, "Please, Enter a Gadget Mobile
Model.");
               return;
           }
           else
           {
               // Enabling buttons and text fields related to mobile gadgets
               btnDisplayAll.setEnabled(true);
               btnShowAllMobiles.setEnabled(true);
               txtSelectMobileId.setEnabled(true);
               txtPhoneNumber.setEnabled(true);
               txtDuration.setEnabled(true);
```

```
            txtDisplayNumber.setEnabled(true);
            txtAddCallingCreditToMObile.setEnabled(true);
            btnMakeACall.setEnabled(true);
            btnAddCallingCredit.setEnabled(true);
        }

        if (price.trim().isBlank() && price.trim().isEmpty())
        {
            // Display an error message if the price field is empty
            JOptionPane.showMessageDialog(null, "Please, Enter a Gadget Mobile Price
(£).");
            return;
        }

        if (weight.isBlank() && weight.isEmpty())
        {
            // Display an error message if the weight field is empty
            JOptionPane.showMessageDialog(null, "Please, Enter a Gadget Mobile
Weight (Grams).");
            return;
        }

        if (size.isBlank() && size.isEmpty())
        {
            // Display an error message if the size field is empty
            JOptionPane.showMessageDialog(null, "Please, Enter a Gadget Mobile Size
(12mm X 17mm x 4mm).");
            return;
        }

        if (credit.isBlank() && credit.isEmpty())
        {
            // Display an error message if the credit field is empty
            JOptionPane.showMessageDialog(null, "Please, Enter Calling Credit
(Minutes).");
            return;
```

```
        }

        // Creating a new Mobile object
        Mobile mobile = new Mobile();
        // Setting properties of the Mobile object
        mobile.setModel(model);
        mobile.setPrice(Double.parseDouble(price));
        mobile.setWeight(Integer.parseInt(weight));
        mobile.setSize(size);
        mobile.setNumberOfMinutesOfCallingCreditRemaining(Integer.parseInt(credit));
        // Adding the Mobile object to the gadget shop controller
        gadgetShopController.addGadget(mobile);

    }
    catch (Exception e)
    {
        // Exception handling
        JOptionPane.showMessageDialog(null, "Error: Please, Enter a Number (e.g., 1,
2, or 5.5): " + e.getMessage());
        return;
    }
    // Displaying a success message
    JOptionPane.showMessageDialog(null, "Great!. Mobile Added Successfully!");
    // Clearing input fields
    txtModel.setText("");
    txtPrice.setText("");
    txtWeight.setText("");
    txtSize.setText("");
    txtCredit.setText("");
  }
```

7.6 Adding an MP3

The btnAddMP3ActionPerformed method adds an MP3 gadget to the shop. It validates input fields for model, price, weight, size, and memory. After enabling relevant features, it creates an MP3 object, sets its properties, and adds it to the shop. Exception handling ensures reliability. Upon success, it displays a confirmation message and clears input fields.

```
/**
   * This method is an event handler for the button "btnAddMP3". It is invoked
   * when the user clicks on the button to add an MP3 gadget.
   *
   * @param evt The ActionEvent representing the user's action (clicking the
   * button)
   */

  private void btnAddMP3ActionPerformed(java.awt.event.ActionEvent
evt)
  {

     // Retrieving input values from text fields
     String model = txtModel.getText();
     String price = txtPrice.getText();
     String weight = txtWeight.getText();
     String size = txtSize.getText();
     String memory = txtMemory.getText();
     try
     {
        // Input validation
        if (model.trim().isBlank() && model.trim().isEmpty())
        {
           // Display an error message if the model field is empty
           JOptionPane.showMessageDialog(null, "Please, Enter a MP3 Model.");
           return;
        }
        else
        {
           // Enabling buttons and text fields related to MP3 gadgets
           btnDisplayAll.setEnabled(true);
           btnSHowAllMP3.setEnabled(true);
           txtSelectMP3Id.setEnabled(true);
           txtDownload.setEnabled(true);
           txtDeleteMusicFromMp3Player.setEnabled(true);
           btnDownloadMusic.setEnabled(true);
```

```
            btnDeleteMusicMP3.setEnabled(true);
        }

        if (price.trim().isBlank() && price.trim().isEmpty())
        {
            // Display an error message if the price field is empty
            JOptionPane.showMessageDialog(null, "Please, Enter a MP3 Price (£).");
            return;
        }

        if (weight.isBlank() && weight.isEmpty())
        {
            // Display an error message if the weight field is empty
            JOptionPane.showMessageDialog(null, "Please, Enter a MP3 Weight
(Grams).");
            return;
        }

        if (size.isBlank() && size.isEmpty())
        {
            // Display an error message if the size field is empty
            JOptionPane.showMessageDialog(null, "Please, Enter a MP3 Size (12mm X
17mm x 4mm).");
            return;
        }

        if (memory.isBlank() && memory.isEmpty())
        {
            // Display an error message if the memory field is empty
            JOptionPane.showMessageDialog(null, "Please, Enter Available Memory
(MB).");
            return;
        }
        // Creating a new MP3 object
        mp3 = new MP3();
        // Setting properties of the MP3 object
```

```
        mp3.setModel(model);
        mp3.setPrice(Double.parseDouble(price));
        mp3.setWeight(Integer.parseInt(weight));
        mp3.setSize(size);
        mp3.setAvailableMemory(Double.parseDouble(memory));
        // Adding the MP3 object to the gadget shop controller
        gadgetShopController.addGadget(mp3);

    }
    catch (Exception e)
    {
        // Exception handling
        JOptionPane.showMessageDialog(null, "Error: Please, You Must Enter a
Number (e.g., 1, 2, or 5.5): " + e.getMessage());
        return;
    }

    // Displaying a success message
    JOptionPane.showMessageDialog(null, "Great!. MP3 Added Successfully!");

    // Clearing input fields
    txtModel.setText("");
    txtPrice.setText("");
    txtWeight.setText("");
    txtSize.setText("");
    txtMemory.setText("");
  }
```

7.7 Displaying all gadgets in the array list

The btnDisplayAllActionPerformed method responds to the "Display All" button click, presenting a list of all gadgets in the shop. It begins by clearing the TextAreaShowData to ensure a fresh display. Then, it retrieves the gadget list from the controller, checks if it's empty, and exits if so. Otherwise, it adds a header, iterates through each gadget, appending its formatted information to the text area. Finally, it adds a visual separator for clarity. This method facilitates user interaction, utilizing the controller to access gadget data and presenting it in a user-friendly format.

```
/**
    * Handles the action event when the user clicks the "Display All" button.
    * This method clears the TextAreaShowData and then iterates through the
    * gadget list. It displays information for all gadgets. If the gadget list
    * is empty, it displays an error message.
    *
    * @param evt The ActionEvent representing the user's action
    */
private void btnDisplayAllActionPerformed(java.awt.event.ActionEvent
evt)
   {
      // Clearing the text area
      TextAreaShowData.setText("");

      // Checking if the list of gadgets is empty
      if (gadgetShopController.getGadgets().isEmpty())
```

```
        {
            // Displaying a message if the list is empty
            JOptionPane.showMessageDialog(null,  "Sorry! There Is Not gadgets To Display
(Empty List.)");
            // Exiting the method
            return;
        }

        // Appending a header for displaying all gadgets
        TextAreaShowData.append("Display  All Gadgets:\n");


        // Iterating over each gadget in the list
        for (Gadget gadget : gadgetShopController.getGadgets())
        {
            // Displaying information about the gadget
            TextAreaShowData.append("  " + gadget.display() + "\n");
            // Adding a newline after displaying each gadget
            TextAreaShowData.append("\n");
        }

        // Appending a separator line
        TextAreaShowData.append("-------------------------------------------------------\n");
}
```

7.8 Clear

The btnClearActionPerformed method serves as an event handler for the "Clear" button click. While it doesn't directly perform the clearing functionality, it delegates this task to a private method named Clear(). Since the implementation details of Clear() are not visible, it likely contains specific logic for resetting data or components in the application. This approach separates the handling of user interaction from the actual clearing process, enhancing code organization and readability.

```
/**
    * This method is an event handler for the button "btnClear". It is invoked
    * when the user clicks on the button to clear/reset some data or
    * components.
    *
    * @param evt The ActionEvent representing the user's action (clicking the
    * button)
    */
private void btnClearActionPerformed(java.awt.event.ActionEvent
evt)
{
  Clear(); //method is defined as private, indicating that it is only accessible within the
same class
}
```

7.9 Show all Mobiles

The btnShowAllMobilesActionPerformed method clears TextAreaShowData and displays all mobile gadgets. It checks for an empty list, appends a header, iterates through each gadget, checking if it's a mobile, and displays its formatted information. Finally, it adds a separator and resets specific input fields.

```
/**
    * Handles the action event when the user clicks the "Show All Mobiles"
    * button. This method clears the TextAreaShowData and then iterates through
    * the gadget list. It displays information only for the gadgets that are
    * instances of Mobile. If no mobile gadgets are found, it displays an error
    * message.
    *
    * @param evt The ActionEvent representing the user's action
    */
   private void btnShowAllMobilesActionPerformed(java.awt.event.ActionEvent
evt)
   {
      // Clearing the text area
      TextAreaShowData.setText("");

      // Checking if the list of gadgets is empty
      if (gadgetShopController.getGadgets().isEmpty())
      {
         // Displaying a message if there are no gadgets available
         JOptionPane.showMessageDialog(null,  "Sorry!. No Gadgets Available From
Mobiles");
         // Exiting the method
         return;
      }

      // Appending a header for the Mobiles List
      TextAreaShowData.append("Mobiles  List:\n");

      // Iterating over each gadget in the list
      for (Gadget gadget : gadgetShopController.getGadgets())
```

```
    {
        // Checking if the gadget is an instance of Mobile
        if (gadget instanceof Mobile)
        {
            // Displaying information about the mobile gadget
            TextAreaShowData.append(" " + gadget.display() + "\n");
            // Adding a newline after displaying each mobile gadget
            TextAreaShowData.append("\n");
        }
    }
    // Appending a separator line
    TextAreaShowData.append("-------------------------------------------------------------\n");
    // Resetting input fields
    txtSelectMP3Id.setText("");
    txtDeleteMusicFromMp3Player.setText("");
    txtDownload.setText("");
}
```

7.10 Making a call

The btnMakeACallActionPerformed method responds to the "Make A Call" button click, simulating a call from a selected mobile gadget. It retrieves input values, validates them, attempts the call, and handles errors.

```
 /**
   * Action performed when the make a call button for mobile is clicked.
   *
   * @param evt Action event generated when the button is clicked.
   *
   * The instanceof operator in Java is used to test whether an object is an
   * instance of a particular class or interface. It also checks if an object
   * is an instance of a subclass of the specified class or interface.
   */

private void btnMakeACallActionPerformed(java.awt.event.ActionEvent
evt)
{
      // Extract mobile ID, phone number, and call duration from text fields
      String mobileId = txtSelectMobileId.getText();
      String phoneNumber = txtPhoneNumber.getText();
      String duration = txtDuration.getText();

      // Check if the gadget list is empty
      if (gadgetShopController.getGadgets().isEmpty())
      {
         JOptionPane.showMessageDialog(null, "Sorry!. No Gadgets Available to Make
Call From Mobile.");
         return;
      }
      // Check for empty or invalid mobile ID
      if (mobileId.trim().isBlank() && mobileId.trim().isEmpty())
      {
         JOptionPane.showMessageDialog(null, "Error: Please Enter A Mobile Id From
The List. ");
         return;
```

```
    }

    // Validate if mobile ID is a non-negative numeric value
    if (!isNonNegativeNumeric(mobileId))
    {
        JOptionPane.showMessageDialog(null, "Error: Please Enter a Valid Mobile Id
Number (1,2...) Not A: " + mobileId);
        return;
    }

    // Check for empty or invalid phone number
    if (phoneNumber.trim().isBlank() && phoneNumber.isEmpty())
    {
        JOptionPane.showMessageDialog(null, "Error: Please Enter A Phone
Number  To Make A Call (Minutes).");
        return;
    }

    // Validate phone number format
    if (phoneNumber.length() != 9 || !isNonNegativeNumeric(phoneNumber))
    {
        JOptionPane.showMessageDialog(null, "Error: Please Enter a Valid Phone
Number To Make A Call  (9-Digits) Not A: " + phoneNumber);
        return;
    }

    // Check for empty or invalid call duration
    if (duration.trim().isBlank() && duration.isEmpty())
    {
        JOptionPane.showMessageDialog(null, "Error: Please Enter A Duration The
Call. (Minutes).");
        return;
    }

    // Validate if call duration is a non-negative numeric value
    if (!isNonNegativeNumeric(duration))
```

```
        {
            JOptionPane.showMessageDialog(null, "Error: Please Enter a Valid Duration
Number (1,2... (Minutes)) Not A: " + duration);
            return;
        }

        try
        {
            // Attempt to parse values to integers
            int mobileID = Integer.parseInt(mobileId);
            int NumberPhone = Integer.parseInt(phoneNumber);
            int callDuration = Integer.parseInt(duration);

            // Check if the mobile ID is within the range and if it's a Mobile object
            if (mobileID >= 1 && mobileID <= gadgetShopController.getGadgets().size() &&
gadgetShopController.getGadgets().get(mobileID - 1) instanceof Mobile)
            {
                // Make the call and handle the result
                boolean resultMakeCAll = ((Mobile)
gadgetShopController.getGadgets().get(mobileID - 1)).mobileMakeCall(NumberPhone,
callDuration);
                if (resultMakeCAll)
                {
                    // Display success message and updated details
                    TextAreaShowData.append("Making A Call From Mobile\n");
                    JOptionPane.showMessageDialog(null, "Great!. Call Was Successfully");
                    TextAreaShowData.append("CALLING NUMBER: " + phoneNumber + "
FOR: " + duration + " MINUTES.\n");
                    TextAreaShowData.append(((Mobile)
gadgetShopController.getGadgets().get(mobileID - 1)).display() + "\n");

                    // Reset text fields
                    txtDisplayNumber.setText(txtPhoneNumber.getText());
                    txtSelectMobileId.setText("");
                    txtPhoneNumber.setText("");
                    txtDuration.setText("");
```

```
                txtAddCallingCreditToMObile.setText("");
                TextAreaShowData.append("-----------------------------------------------------------
\n");

            }
            else
            {
                // Display error message for insufficient credit
                JOptionPane.showMessageDialog(null, "Sorry!. Insufficient Credit To Make
The Call.");
            }
        }
        else
        {
            throw new IndexOutOfBoundsException();
        }
    }
    catch (IndexOutOfBoundsException | InputMismatchException e)
    {
        // Handle exceptions
        JOptionPane.showMessageDialog(null, "Error: Invalid Choice. Please Enter A
Valid Number On The List.: " + e.getMessage());
    }
}
```

7.11 Add Calling Credit

The method adds calling credit to a mobile gadget, validating user input, ensuring mobile ID existence, updating credit, displaying success messages, and handling errors.

```
/*
        This method appears to handle adding calling credit to
a mobile gadget in a shop's inventory.
        It validates user input, performs necessary checks,
        updates the gadget's calling credit, and displays
appropriate messages.
     */
    private void
btnAddCallingCreditActionPerformed(java.awt.event.ActionEvent
evt)
    {
        // Step 1: Extracting Input
        String mobileId = txtSelectMobileId.getText();
        String creditToAdd =
txtAddCallingCreditToMObile.getText();

        // Step 2: Checking for Empty Gadget1 List
        if (gadgetShopController.getGadgets().isEmpty())
        {
            JOptionPane.showMessageDialog(null, "Sorry!. No
Gadgets Available to Add Calling Credit..");
            return;
        }

        // Step 3: Validating Mobile ID
        if (mobileId.trim().isBlank() &&
mobileId.trim().isEmpty())
        {
            JOptionPane.showMessageDialog(null, "Error: Please
Enter A Mobile Id From The List. ");
            return;
        }

        // Step 4: Validating Mobile ID Format
        if (!isNonNegativeNumeric(mobileId))
        {
```

```
            JOptionPane.showMessageDialog(null, "Error: Please
Enter a Valid Mobile Id Number (1,2...) Not Un: " + mobileId);
            return;
        }

        // Step 5: Validating Credit to Add
        if (creditToAdd.trim().isBlank() &&
creditToAdd.isEmpty())
        {
            JOptionPane.showMessageDialog(null, "Error: Please
Enter A Calling Credit  To Mobile (MInutes).");
            return;
        }

        // Step 6: Validating Credit to Add Format
        if (!isNonNegativeNumeric(creditToAdd))
        {
            JOptionPane.showMessageDialog(null, "Error: Please
Enter a Valid Add Credit From Mobile Minutes  (1,2...) Not A: "
+ creditToAdd);
            return;
        }
        try
        {
            // Step 7: Parsing Mobile ID
            int id = Integer.parseInt(mobileId);

            // Step 8: Checking Mobile ID Range and Type
            if (id >= 1 && id <=
gadgetShopController.getGadgets().size() &&
gadgetShopController.getGadgets().get(id - 1) instanceof
Mobile)
            {
                // Step 9: Updating Mobile's Calling Credit
                TextAreaShowData.append("Mobiles Update with
Calling Credit:\n");
```

```
                ((Mobile)
gadgetShopController.getGadgets().get(id -
1)).addCallingCredit(Integer.parseInt(creditToAdd));

                // Step 10: Displaying Updates
                JOptionPane.showMessageDialog(null, "Great!.
The Calling Credit To Add Was Successfully." + creditToAdd + "
Minutes");
                TextAreaShowData.append(((Mobile)
gadgetShopController.getGadgets().get(id - 1)).display() +
"\n");

                // Step 11: Clearing Text Fields
                txtSelectMobileId.setText("");
                txtPhoneNumber.setText("");
                txtDuration.setText("");
                txtAddCallingCreditToMObile.setText("");
            }
            else
            {
                throw new IndexOutOfBoundsException();
            }

        }
        catch (IndexOutOfBoundsException |
InputMismatchException e)
        {
            // Step 12: Exception Handling
            JOptionPane.showMessageDialog(null, "Error: Invalid
Choice. Please Enter a Valid Number On The List.): " +
e.getMessage());
        }
        // Step 13: Appending Separator
        TextAreaShowData.append("-----------------------------
------------------------------\n");
    }
```

7.12 Show All MP3 Players

The btnShowAllMP3ActionPerformed method clears TextAreaShowData and displays all MP3 gadgets. It checks for an empty list and appends a header if not. Iterating through gadgets, it checks if each is an MP3, displaying formatted information. Finally, it adds a separator. This method effectively filters and displays MP3 gadgets.

```
/**
    * Action performed when the button to show all MP3 gadgets is clicked.
    *
    * @param evt Action event generated when the button is clicked.
    */

private void btnSHowAllMP3ActionPerformed(java.awt.event.ActionEvent
evt)
{
        // Clear the text area
        TextAreaShowData.setText("");
        // Check if the gadget list is empty
        if (gadgetShopController.getGadgets().isEmpty())
        {
            JOptionPane.showMessageDialog(null,  "Sorry!. No Gadgets Available From
MP3s.");
            return;
        }
        else
        {
            // Iterate through the list of gadgets
            // Display MP3 gadgets
            TextAreaShowData.append("MP3s List:\n");
            for (Gadget gadget : gadgetShopController.getGadgets())
            {
                if (gadget instanceof MP3)
                {
                    // Display MP3 gadget details
                    TextAreaShowData.append("  " + gadget.display() + "\n");
                    TextAreaShowData.append("\n");
                }
            }
            // Append separator to the text area
            TextAreaShowData.append("-------------------------------------------------------------\n");
        }
}
```

7.13 Downloading music

The btnDownloadMusicActionPerformed method handles the "Download Music" button, retrieving input values, validating them, and attempting music download to an MP3 player. Exception handling ensures robustness, and the display is updated accordingly.

```
  /**
     * Handles the action event when the user clicks the
"Download Music"
     * button. This method retrieves the MP3 ID and the memory
to free from the
     * UI components. It performs validations and then attempts
to download
     * music to the specified MP3 player. If successful, it
updates the UI with
     * the information of the MP3 player and the download
status. If
     * unsuccessful, it displays an error message.
     *
     * @param evt The ActionEvent representing the user's
action
     */

    private void
btnDownloadMusicActionPerformed(java.awt.event.ActionEvent
evt)
    {
        String mp3Id = txtSelectMP3Id.getText();
        String downloadMusic = txtDownload.getText();

        // Check if the gadget list is empty
        if (gadgetShopController.getGadgets().isEmpty())
        {
```

```
            JOptionPane.showMessageDialog(null, "Sorry!. No
Gadgets Available To Download Music MP3.");
            return;
        }

        // Validate MP3 ID input
        if (mp3Id.trim().isBlank() && mp3Id.trim().isEmpty())
        {
            JOptionPane.showMessageDialog(null, "Error: Please
Enter A MP3 Id From The List. ");
            return;
        }

        // Validate if MP3 ID is a non-negative number
        if (!isNonNegativeNumeric(mp3Id))
        {
            JOptionPane.showMessageDialog(null, "Error: Please
Enter a Valid MP3 Id Number (1,2...) Not Un: " + mp3Id);
            return;
        }

        // Validate memory input
        if (downloadMusic.trim().isBlank() &&
downloadMusic.isEmpty())
        {
            JOptionPane.showMessageDialog(null, "Error: Please
Enter Memory To Free From The Download Music (MB).");
            return;
        }

        // Validate if memory input is a non-negative number
        if (!isNonNegativeNumeric(downloadMusic))
        {
            JOptionPane.showMessageDialog(null, "Error: Please
Enter a Valid Memory  Number  From MP3 (1,2...) Not A: " +
downloadMusic);
```

```
            return;
        }

        try
        {
            // Parsing MP3 ID
            int idMp3 = Integer.parseInt(mp3Id);

            // Validation and Checking MP3 ID
            if (idMp3 >= 1 && idMp3 <=
gadgetShopController.getGadgets().size() &&
gadgetShopController.getGadgets().get(idMp3 - 1) instanceof
MP3)
            {
                // Attempting to download music
                boolean resultDownload = ((MP3)
gadgetShopController.getGadgets().get(idMp3 -
1)).downloadMusic(Integer.parseInt(downloadMusic));

                // Handling Download Result
                if (resultDownload)
                {
                    // Actions if download is successful
                    TextAreaShowData.append("MP3 Update With
Download Music In MB:\n");
                    JOptionPane.showMessageDialog(null,
"Great!. The Download Music To MP3 Was Successfully: " +
downloadMusic + " MB");
                    TextAreaShowData.append(((MP3)
gadgetShopController.getGadgets().get(idMp3 - 1)).display());

                    txtSelectMP3Id.setText("");
                    txtDownload.setText("");
                    txtDeleteMusicFromMp3Player.setText("");
                }
                else
```

```
                {
                    // Actions if download fails
                    JOptionPane.showMessageDialog(null,
"Sorry!. Not Enough Memory To Download Music!");
                    return;
                }
            }
            else
            {
                // Invalid MP3 ID
                throw new IndexOutOfBoundsException();
            }
        }
        catch (IndexOutOfBoundsException |
InputMismatchException e)
        {
            // Exception Handling
            JOptionPane.showMessageDialog(null, "Error: Invalid
Choice. Please Enter a Valid Number On The List.): " +
e.getMessage());
            return;
        }

        // Appending to TextArea
        TextAreaShowData.append("\n-----------------------------
--------------------------------\n");
    }
```

7.14 Delete Music MP3

The method manages the "Delete Music from MP3 Player" button, validating input, attempting music deletion, and updating the display accordingly.

```
 /*
      This method handles the deletion of music from an MP3 player in a Java GUI
application.
      It takes input from text fields for MP3 ID and the amount of memory to delete,
performs validation checks,
      updates the MP3 player's memory, and displays appropriate messages. Finally,
      it appends a separator to the text area for visual distinction.
    */
private void btnDeleteMusicMP3ActionPerformed(java.awt.event.ActionEvent
evt)
{
      // Extract MP3 ID and memory to delete from text fields
      String mp3Id = txtSelectMP3Id.getText();
      String deleteMusic = txtDeleteMusicFromMp3Player.getText();

      // Check if the gadget list is empty
      if (gadgetShopController.getGadgets().isEmpty())
      {
         JOptionPane.showMessageDialog(null,  "Sorry!. No Gadgets Available To Delete
Music MP3.");
         return;
      }

      // Check for empty or invalid MP3 ID
      if (mp3Id.trim().isBlank() && mp3Id.trim().isEmpty())
      {
         JOptionPane.showMessageDialog(null,  "Error: Please Enter A MP3 Id From The
List. ");
         return;
      }

      // Validate if MP3 ID is a non-negative numeric value
      if (!isNonNegativeNumeric(mp3Id))
      {
         JOptionPane.showMessageDialog(null,  "Error: Please Enter a Valid MP3 Id
Number (1,2...) Not Un: " + mp3Id);
```

```
            return;
        }

        // Check for empty or invalid memory value
        if (deleteMusic.trim().isBlank() && deleteMusic.isEmpty())
        {
            JOptionPane.showMessageDialog(null, "Error: Please Enter Memory To Free
From The Delete Music (MB).");
            return;
        }

        // Validate if memory value is a non-negative numeric value
        if (!isNonNegativeNumeric(deleteMusic))
        {
            JOptionPane.showMessageDialog(null, "Error: Please Enter a Valid
Memory Number From MP3 (1,2...) Not A: " + deleteMusic);
            return;
        }

        try
        {
            // Attempt to parse MP3 ID to an integer
            int idMp3 = Integer.parseInt(mp3Id);

            // Check if the ID is within the range and if it's an MP3
            if (idMp3 >= 1 && idMp3 <= gadgetShopController.getGadgets().size() &&
gadgetShopController.getGadgets().get(idMp3 - 1) instanceof MP3)
            {
                // Display update message
                TextAreaShowData.append("MP3 Update with Delete Music Memory:\n");

                // Delete music from MP3 and display success message
                ((MP3) gadgetShopController.getGadgets().get(idMp3 -
1)).deleteMusic(Integer.parseInt(deleteMusic));
                JOptionPane.showMessageDialog(null, "Great!. The Delete Music To MP3
Was Successfully." + deleteMusic + " Minutes");
```

```
            // Display updated MP3 details
            TextAreaShowData.append(((MP3)
gadgetShopController.getGadgets().get(idMp3 - 1)).display() + "\n");

            // Reset text fields
            txtSelectMP3Id.setText("");
            txtDownload.setText("");
            txtDeleteMusicFromMp3Player.setText("");
        }
        else
        {
            throw new IndexOutOfBoundsException();
        }
    }
    catch (IndexOutOfBoundsException | InputMismatchException e)
    {
        // Handle exceptions
        JOptionPane.showMessageDialog(null, "Error: Invalid Choice. Please Enter a
Valid Number On The List.): " + e.getMessage());
        return;
    }

    // Append separator to the text area
    TextAreaShowData.append("-------------------------------------------------------\n");
}
```

7.15 Exit

The btnExitActionPerformed method handles the "Exit" button click event, closing the application window. It directly calls the dispose () method, responsible for deallocating resources and removing the window from the screen. This ensures a straightforward and efficient way to exit the application.

```
/**
    * This method is an event handler for the button "btnExit". It is invoked
    * when the user clicks on the button to exit the application. It disposes
    * of the current window, effectively closing it.
    *
    * @param evt The ActionEvent representing the user's action (clicking the
    * button)
    */
   private void btnExitActionPerformed(java.awt.event.ActionEvent
evt)
   {
      dispose(); // Dispose of the current window
   }
```

7.16 Method Clear ()

The Clear () method in a Java GUI application resets text fields by emptying their content and clears a text area by removing displayed text, facilitating user input and information display.

```
/** Clears the content of all text fields and text area. */
   private void Clear()
   {
      // Clear the text in each text field and text area by setting their text to an empty
string
      txtModel.setText("");
      txtPrice.setText("");
      txtWeight.setText("");
      txtSize.setText("");
      txtCredit.setText("");
      txtMemory.setText("");
      txtSelectMobileId.setText("");
      txtSelectMP3Id.setText("");
      txtPhoneNumber.setText("");
      txtDuration.setText("");
      txtDisplayNumber.setText("");
      txtAddCallingCreditToMObile.setText("");
      txtDownload.setText("");
      txtDeleteMusicFromMp3Player.setText("");
      // Clear the text area
      TextAreaShowData.setText("");
   }
```

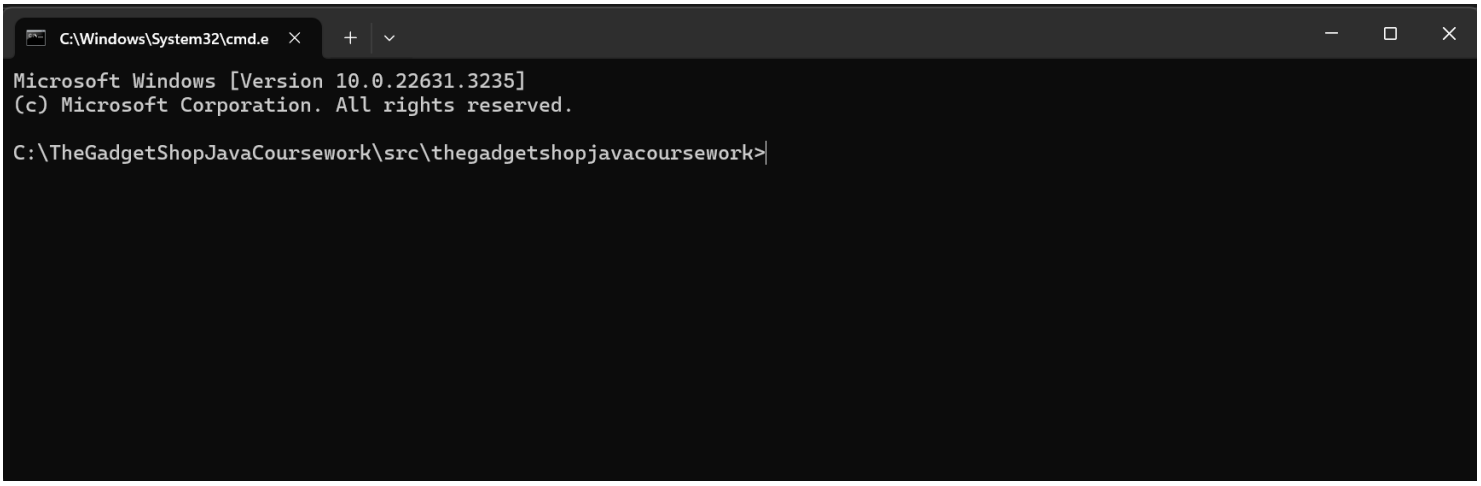
8. Testing

The Gadget Shop program's testing evidence includes screenshots of dialog boxes and the GUI interface, displaying interactions, messages, entered values, and the program's state.

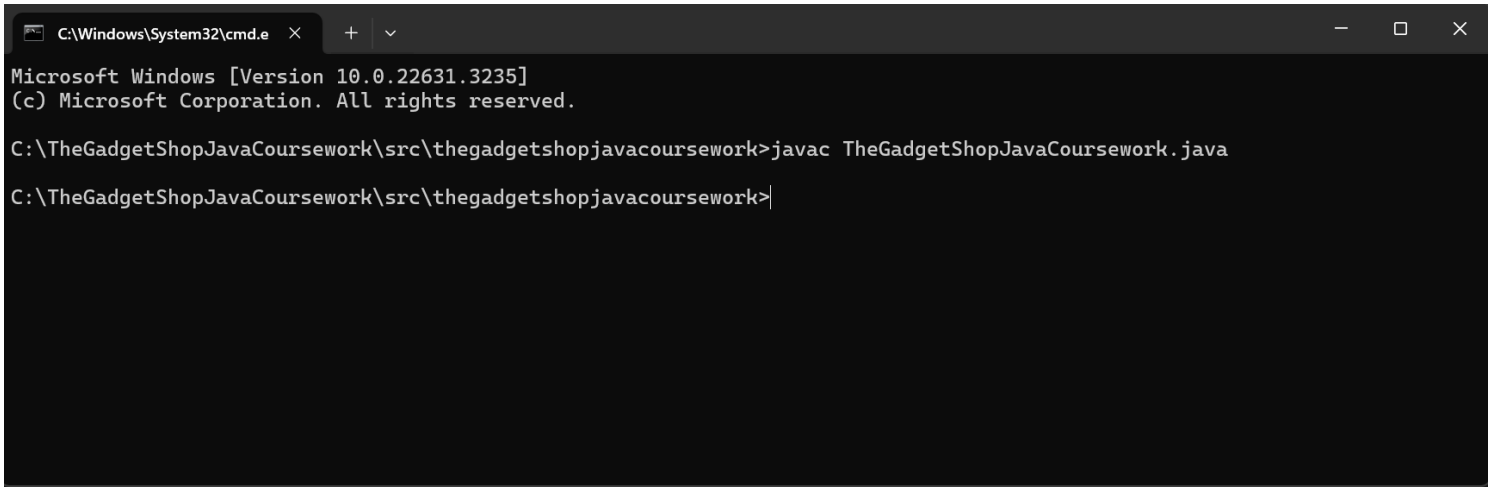
8.1 Test 1: Running my program from Terminal

Test that the program can be compiled and run using the command prompt, including a screenshot similar to Figure 1 in the command prompt learning aid.

8.1.1 My first step was changing the directory to the location of my project.

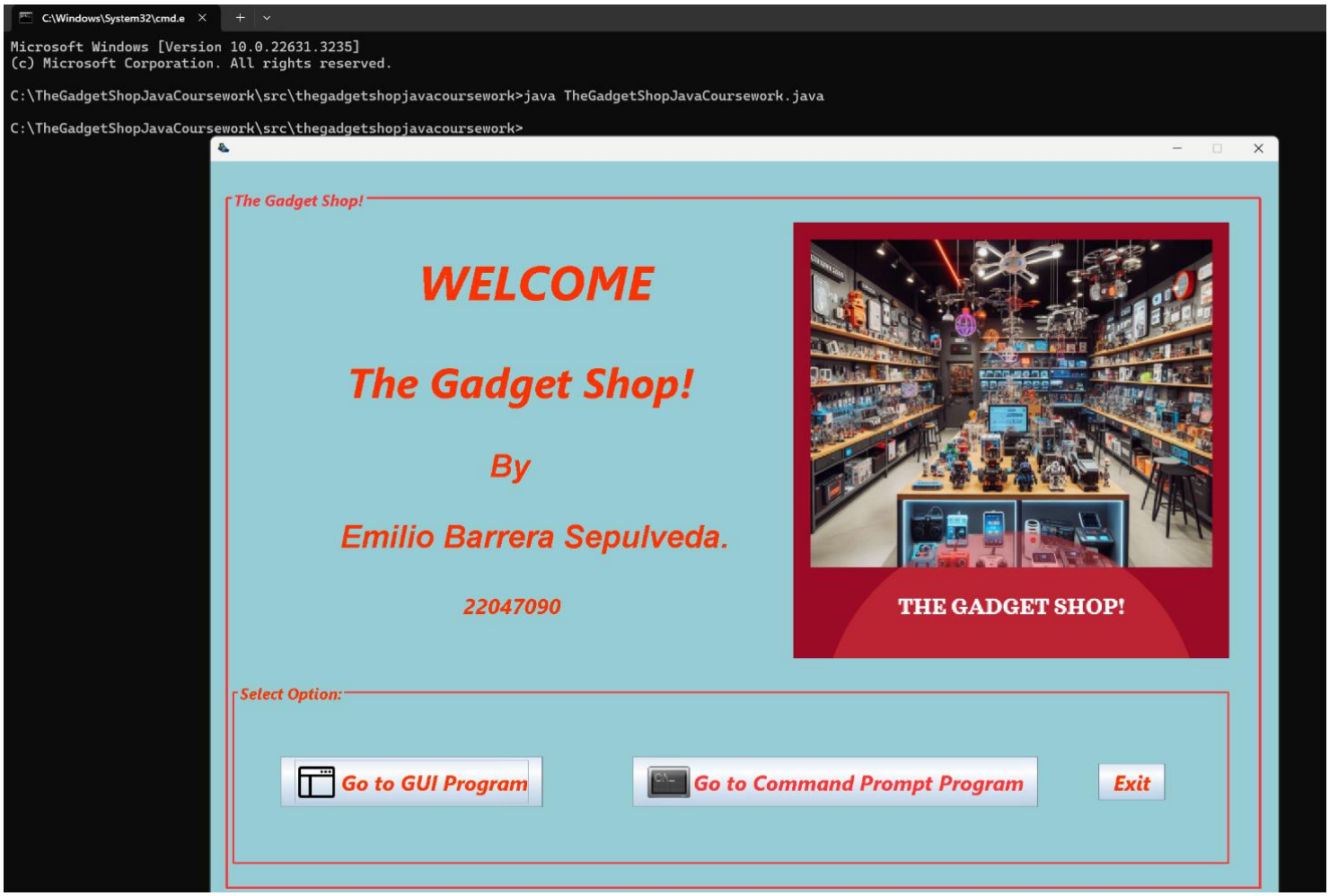


8.1.2 My next step was compiling my main Java file.



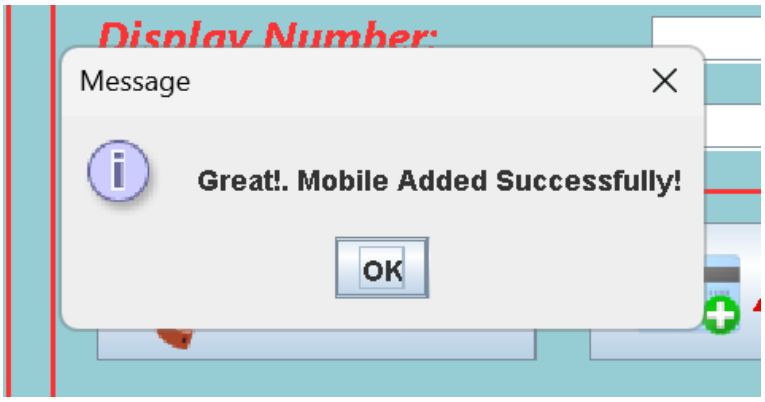
8.1.3 My final step was running the file.

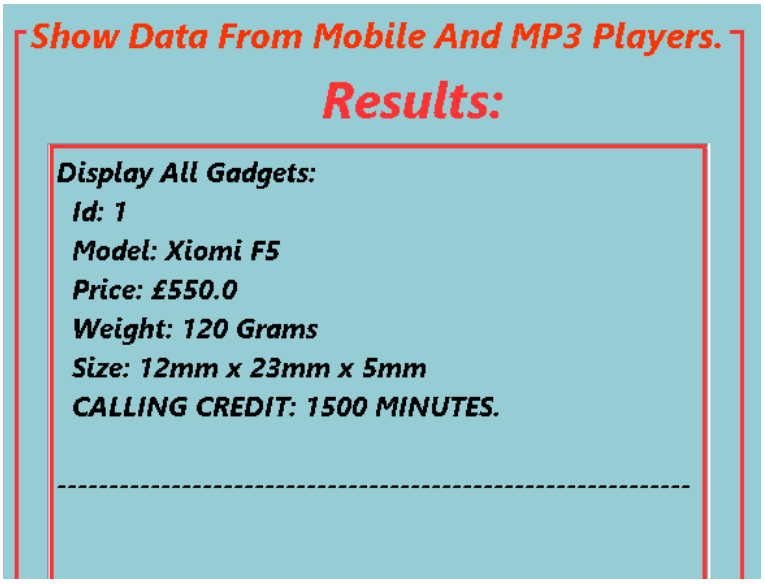
Program ran successfully.



8.2 Test 2: Adding a mobile to the Array List

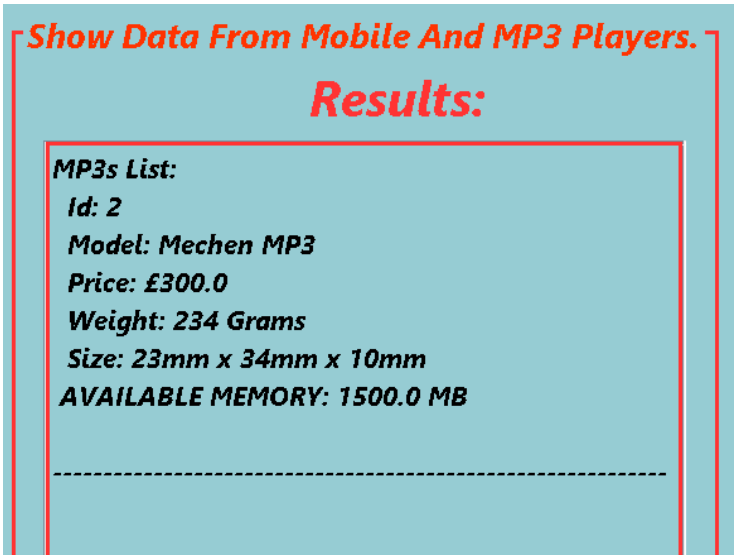
After entering all values into the fields, you can click on the 'Add Mobile' button. If successful, you'll receive a confirmation message indicating that the data was added to the list.





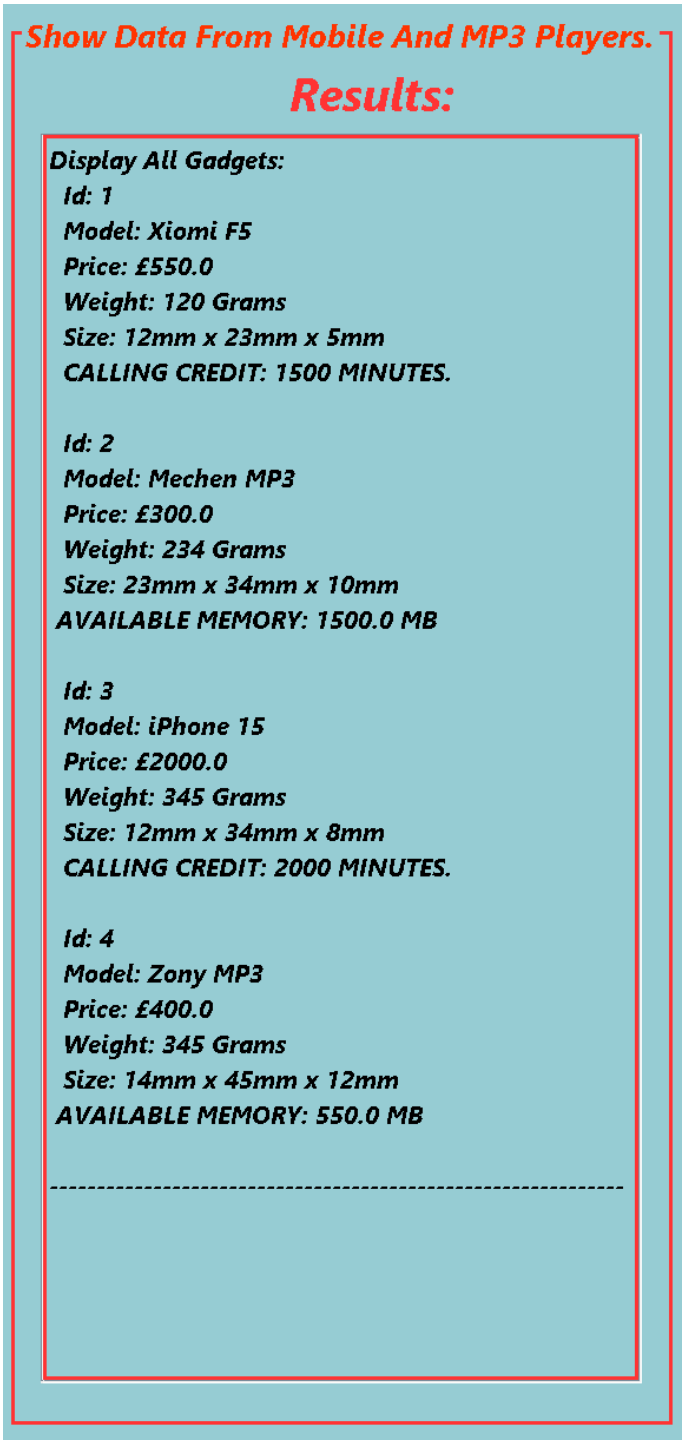
8.3 Test 3: Adding an MP3 Players to the Array List

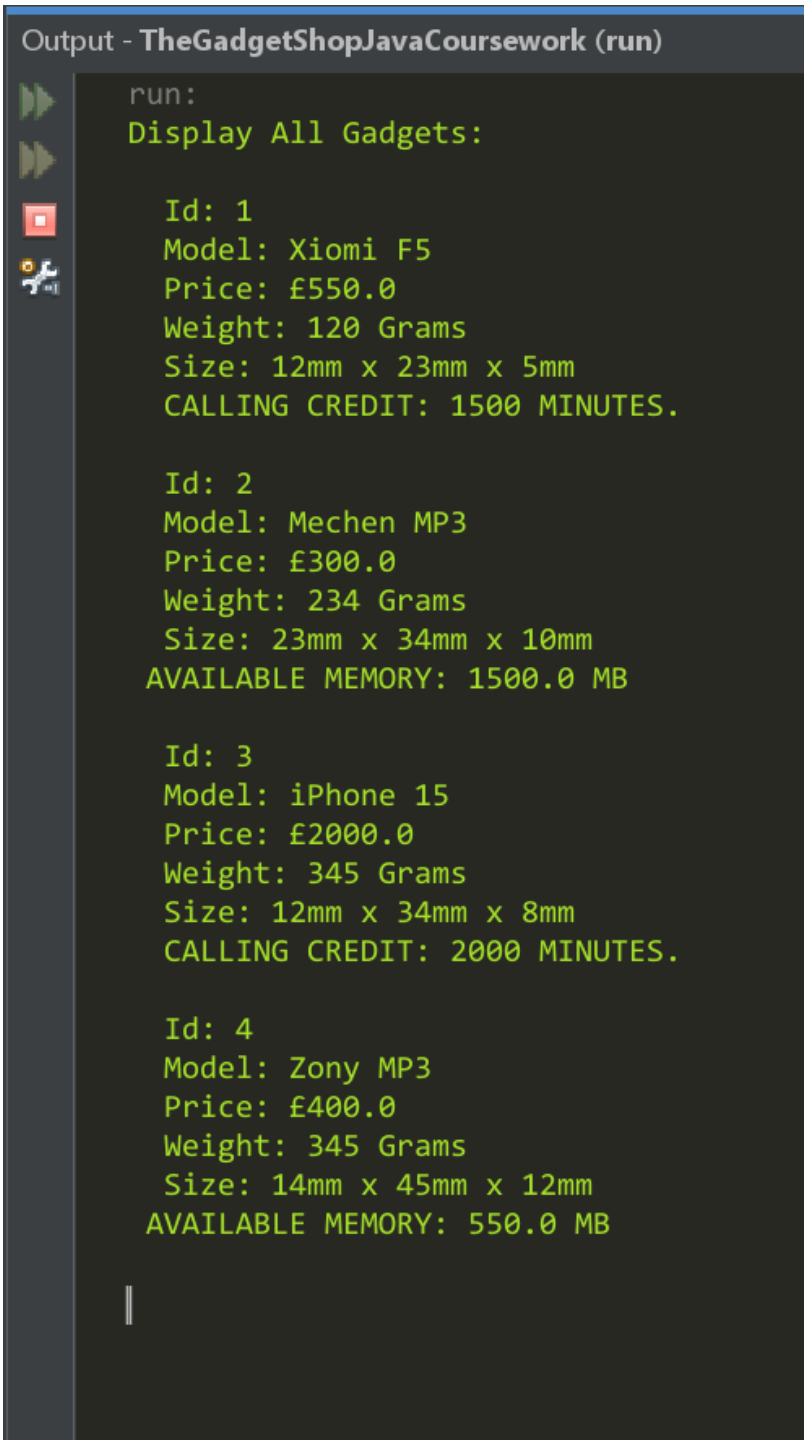
After entering all values into the fields, you can click on the Add MP3 Player button. If successful, you'll receive a confirmation message indicating that the data was added to the list.



8.4 Test 4: Displaying All Gadgets the Array List

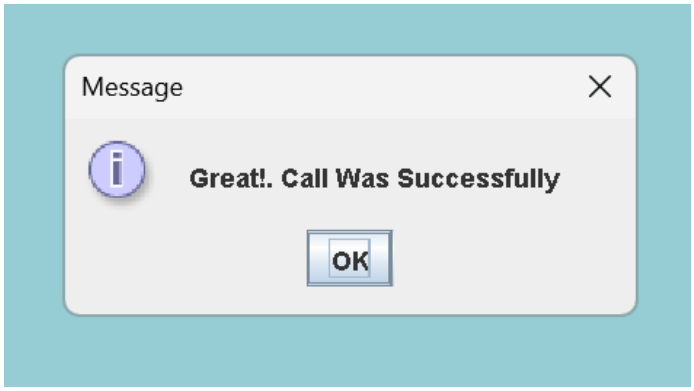
This method manages the "Display All" button action, clearing the display area, iterating through gadgets to show their information, and handling empty lists with an error message.

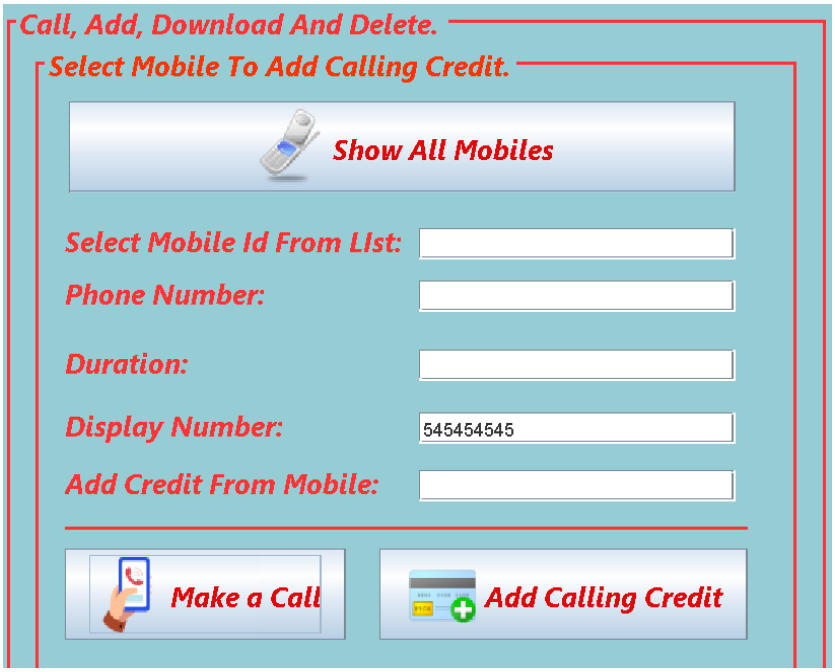


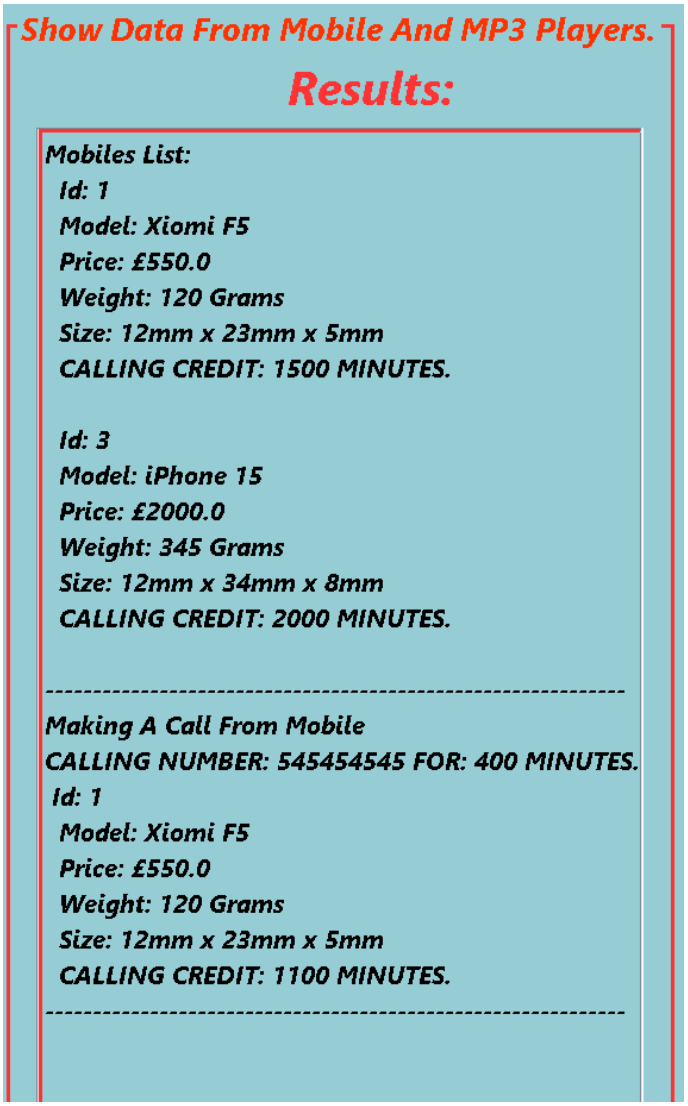


8.5 Test 5: Making a call

Initiates a call from the mobile phone, displays the phone number in the designated field, deducts the call duration from available credit, and prompts an insufficient credit message if necessary.

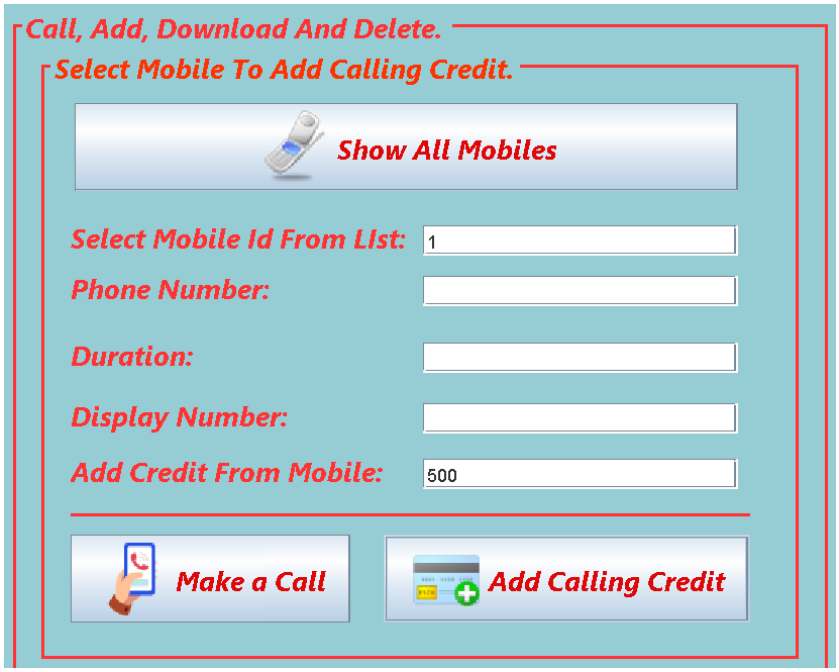


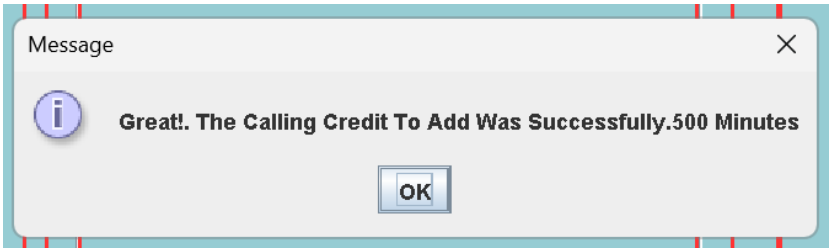


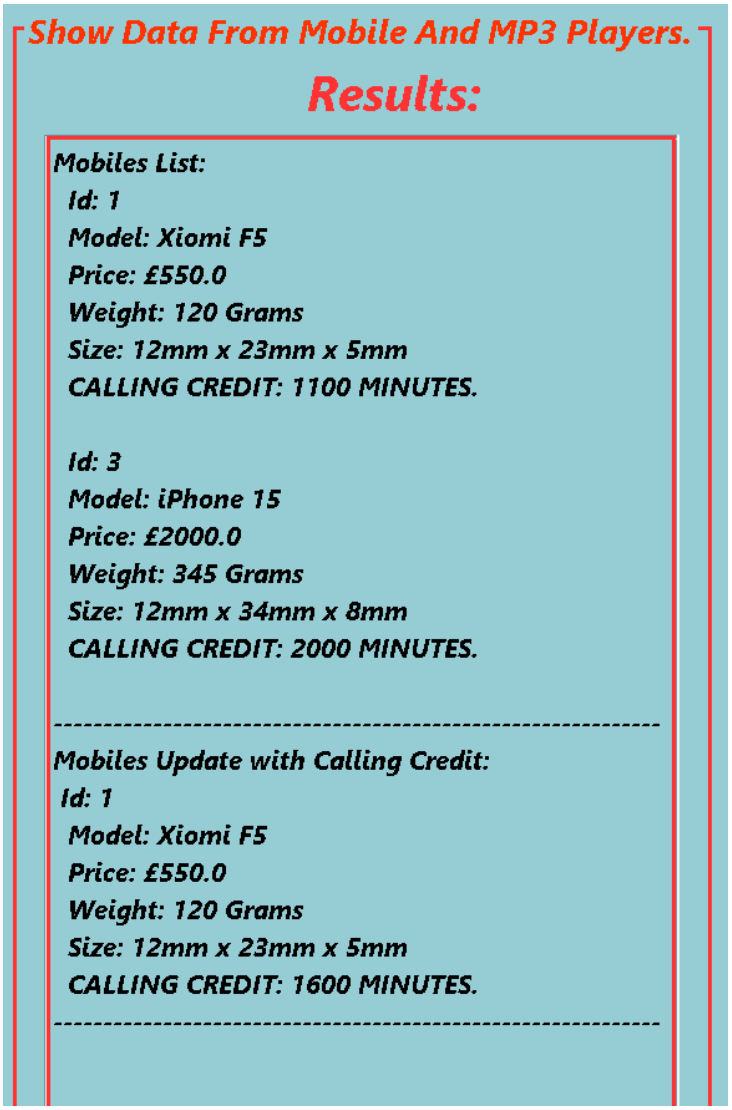


8.6 Test 5.1: Add Calling Credit

Increases the calling credit on the mobile phone by the specified amount if a positive value is provided; otherwise, prompts the user to enter a positive amount.

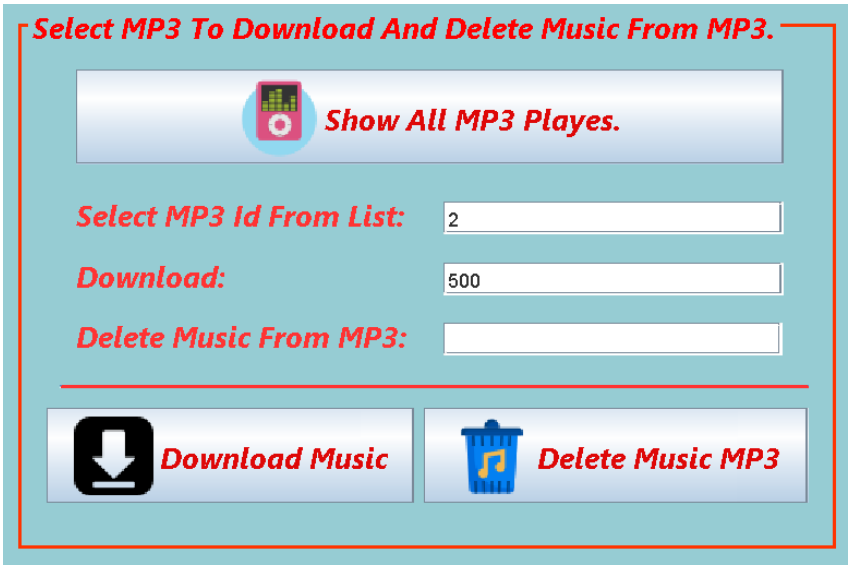


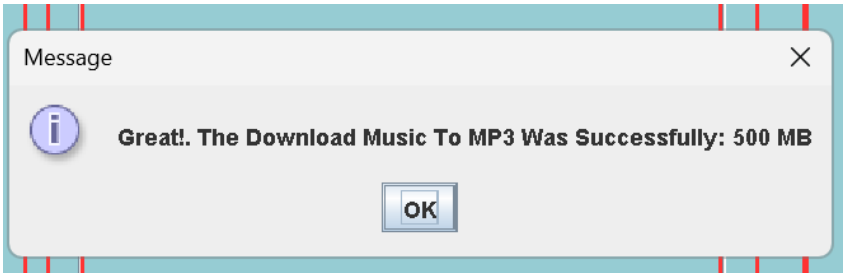


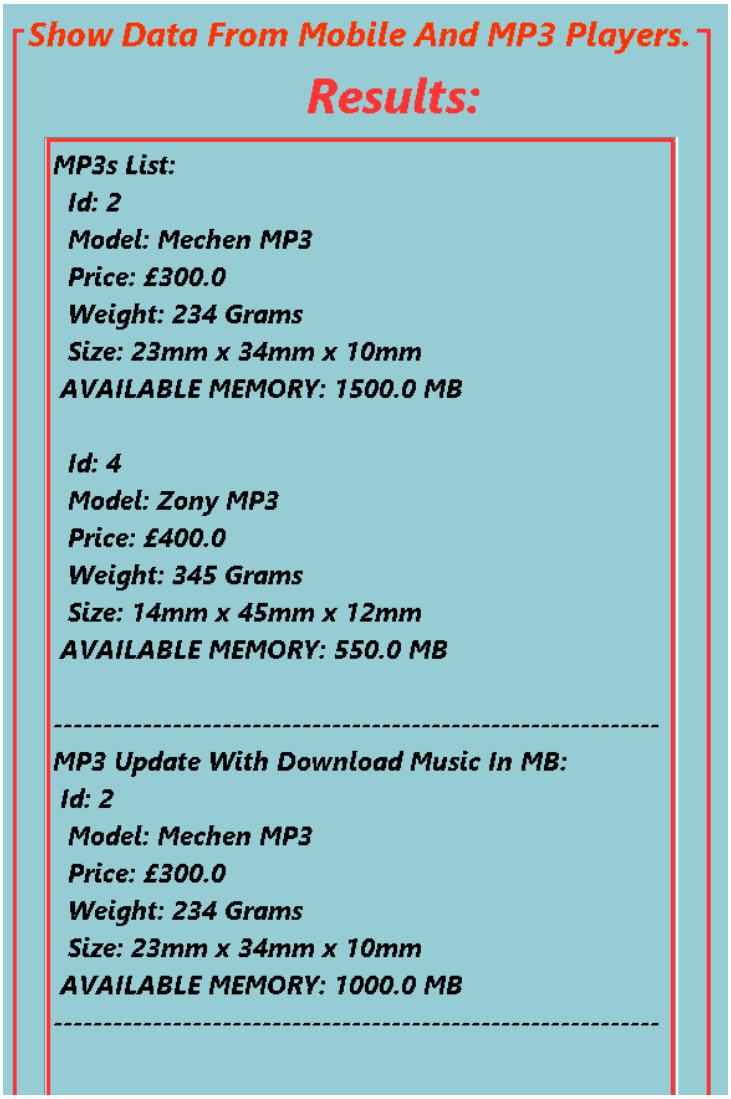


8.7 Test 6: Downloading music

Downloads music to the MP3 player, reducing available memory if sufficient space is available; otherwise, prompts an error message indicating insufficient memory.

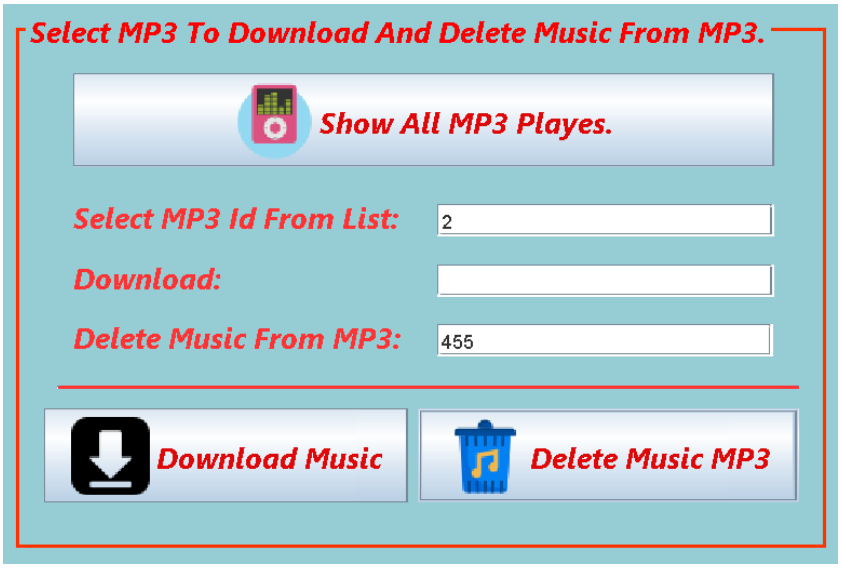


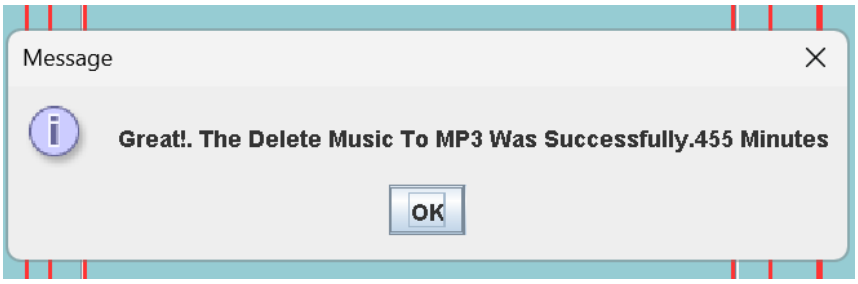


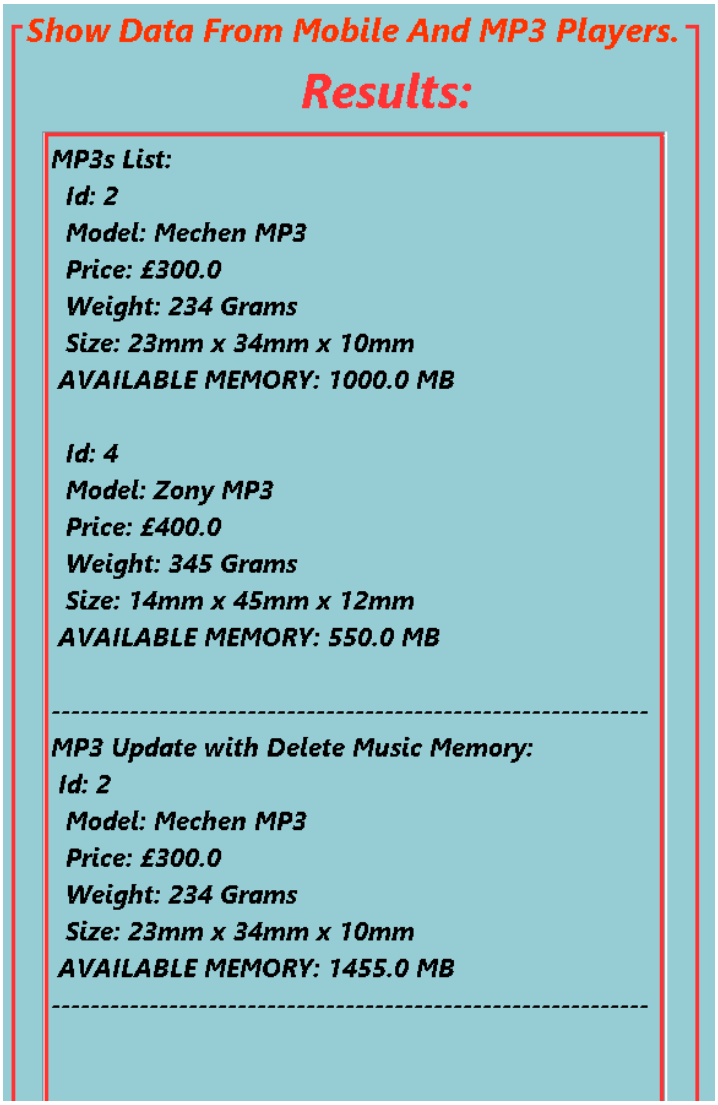


8.8 Test 6.1: Delete Music MP3

Deletes music from the MP3 player, increasing available memory by the specified amount**.**

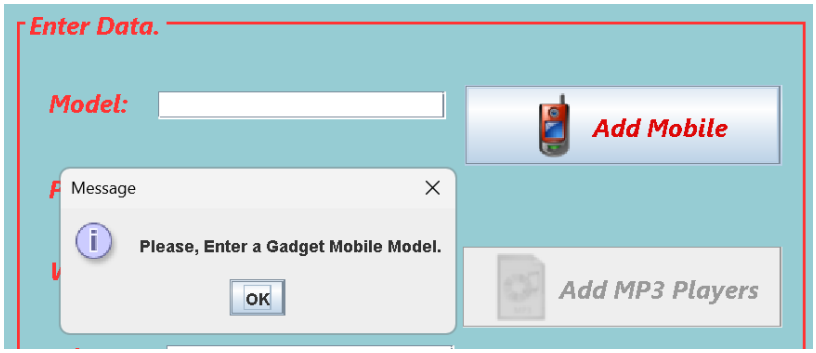
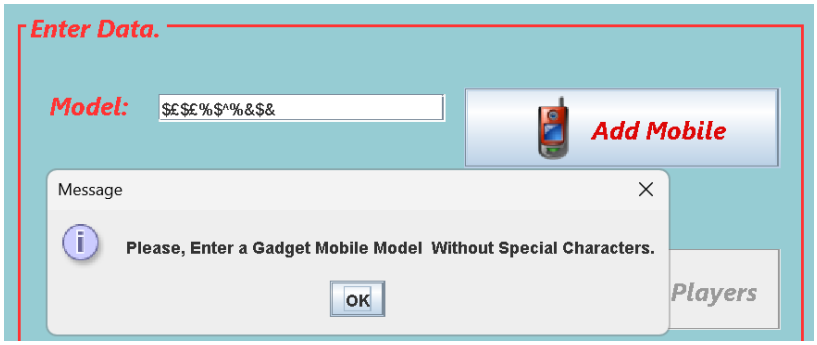


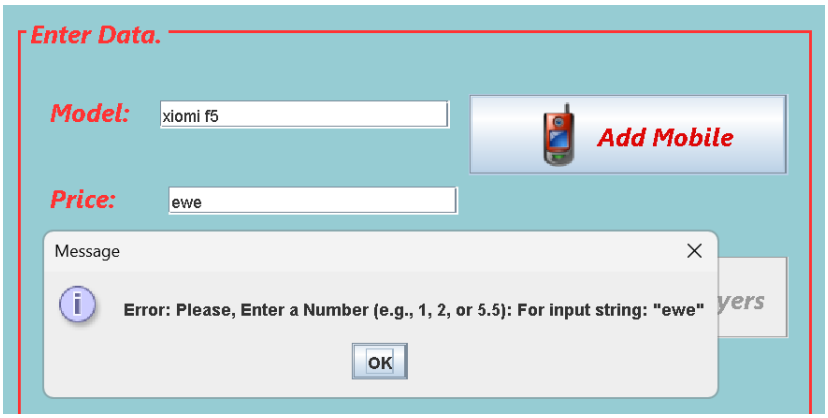




8.9 Test 7: Test that appropriate dialog boxes appear when unsuitable values are entered for the display number.

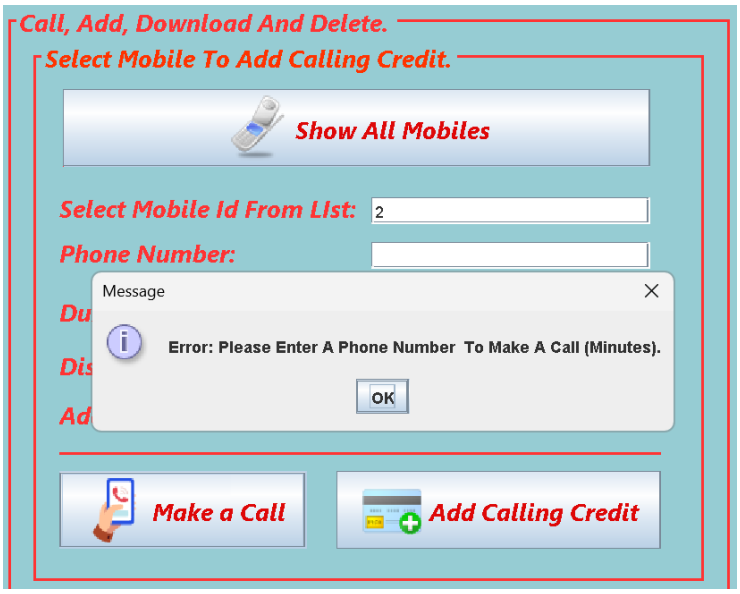
Try-catch blocks were utilized in most form field validations to handle errors thrown by the system.

Error Capture Message	Observations
	Ensure all form fields are filled to avoid empty values. Display an alert message if any field is left empty.
	Handle special characters to prevent input validation issues and ensure data integrity.

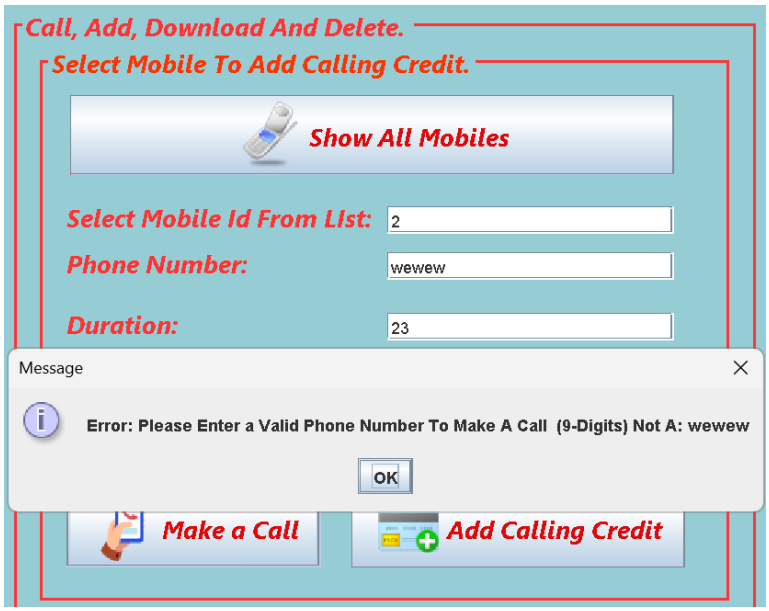


Allow only numeric input in specific fields and display an alert message if non-numeric characters are entered.

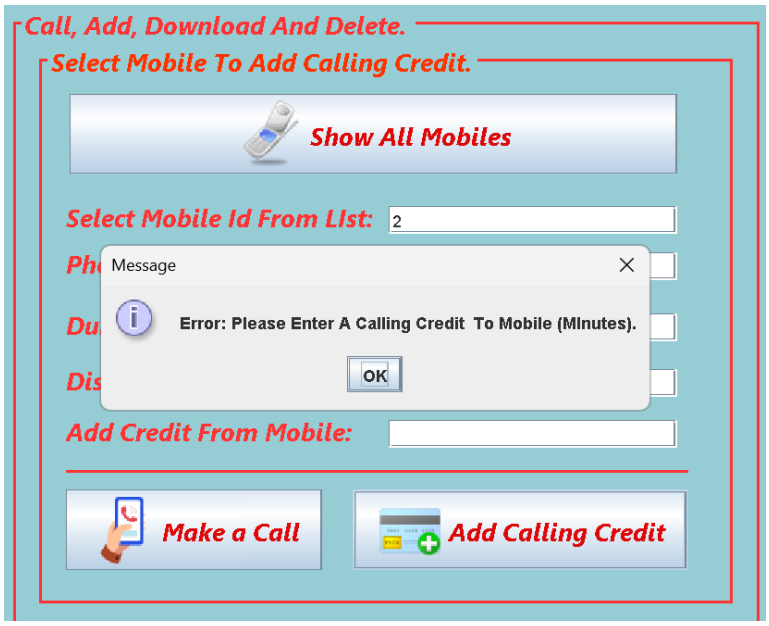
Before performing any operation on the phone, such as making a call or adding credit, ensure the selection of an existing phone number from the list. Prompt a message if the selected number does not exist, instructing to choose an existing one.



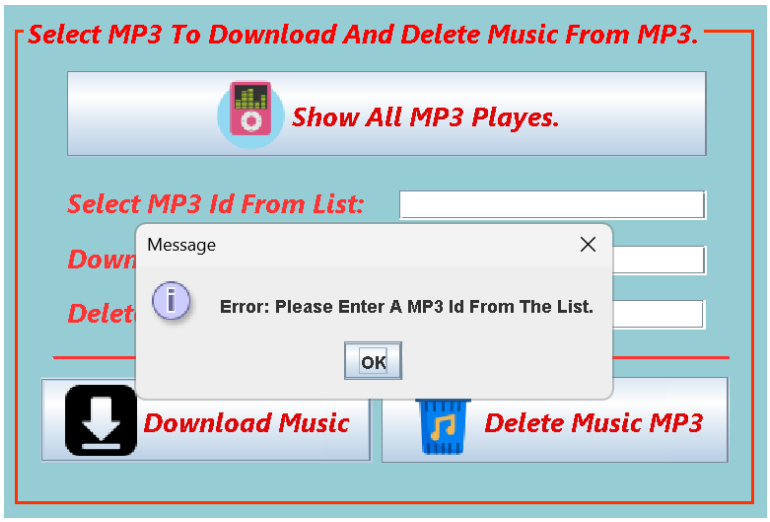
Please enter a phone number in the field to avoid triggering the empty field message.



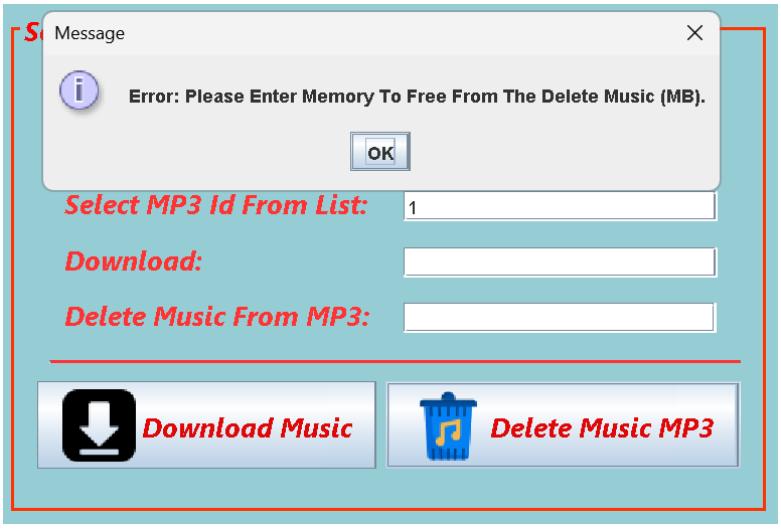
Please enter a phone number without any letters to prevent the message from being triggered.



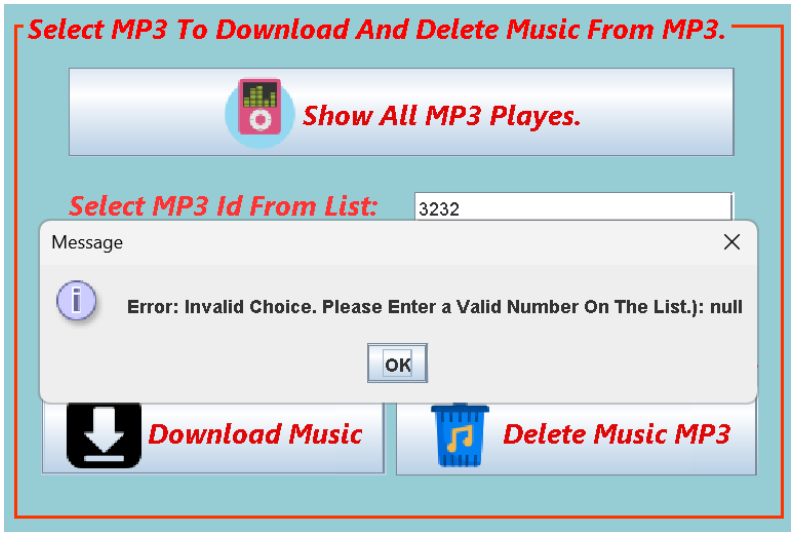
To prevent the alert message, ensure you input the credit amount before proceeding.



Please ensure that you select an MP3 player ID before attempting any download or music deletion operations.



Validation ensures the deletion field is not left empty. If it is empty, a message prompts you to enter data.



If you enter an ID for an MP3 player that does not exist in the list, a message indicating non-existence will appear.

If a negative value is entered in the MP3 download field, it is considered invalid and triggers an error message.

Entering data in the MP3 fields is mandatory; otherwise, an alert message will appear.

9. CONCLUSION

The Java-based GadgetShop application epitomizes Object-Oriented Programming (OOP) principles, showcasing adeptness in inheritance, encapsulation, and polymorphism. Through parent class (Gadgets) and subclasses (Phone and MP3) with customized attributes and methods, it demonstrates mastery in code organization. Integration of Java Swing for the GUI enhances user experience, while adherence to the Model-View-Controller (MVC) pattern ensures modularity. The accompanying report, featuring detailed diagrams, pseudocode, and testing documentation, underscores commitment to functionality and reliability. This coursework reflects proficiency in advanced Java programming, illustrating practical application of OOP concepts in the development of the GadgetShop application.