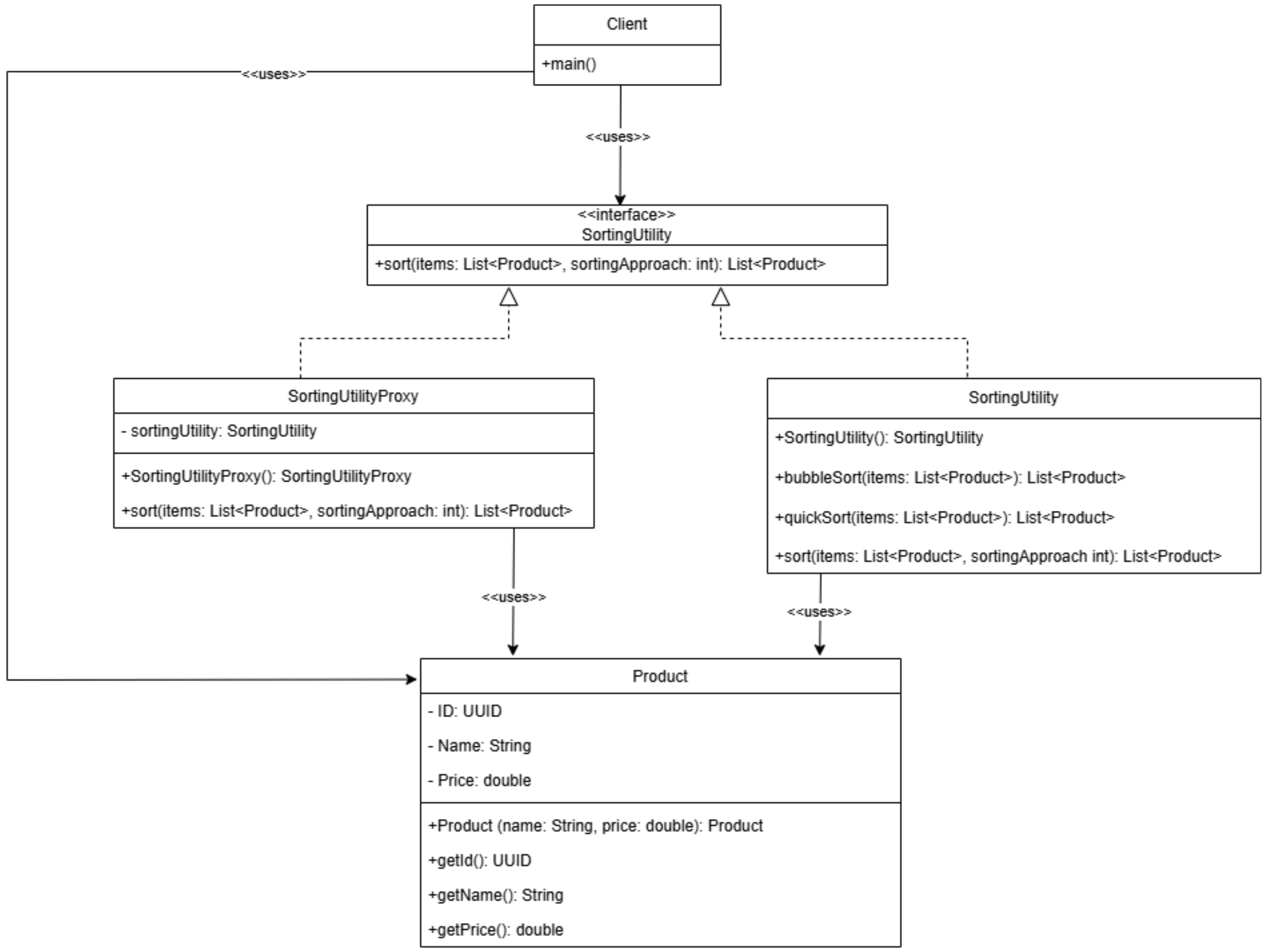
# SE 471 - Lab 1 - Report

| Name | CSUSM Email | Contribution Percentage |
|---|---|---|
| Jaedon Spurlock | spurl004@csusm.edu | 100% |

## Problem 1

This problem covers a situation where the client wants to log the results from a sorting algorithm but the main class that implements it cannot be modified. The design pattern we can use is the `PROXY` pattern, which is a structural design pattern that provides an object (proxy) that acts as an intermediary for another object.

## UML Diagram


Client
+main()
<<uses>>
<<uses>>
<<interface>>
SortingUtility
+sort(items: List<Product>, sortingApproach: int): List<Product>
SortingUtilityProxy
- sortingUtility: SortingUtility
+SortingUtilityProxy(): SortingUtilityProxy
+sort(items: List<Product>, sortingApproach: int): List<Product>
SortingUtility
+SortingUtility(): SortingUtility
+bubbleSort(items: List<Product>): List<Product>
+quickSort(items: List<Product>): List<Product>
+sort(items: List<Product>, sortingApproach int): List<Product>
<<uses>>
<<uses>>
Product
- ID: UUID
- Name: String
- Price: double
+Product (name: String, price: double): Product
+getId(): UUID
+getName(): String
+getPrice(): double

## Code implementation (Full code attached in submission)

First we define our Product model,

```
public class Product {
    private final UUID ID;
    private String name;
    private double price;

    public Product(String name, double price) {
        this.ID = UUID.randomUUID();
        this.name = name;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public double getPrice() {
        return price;
    }

    public UUID getID() {
        return ID;
    }
}
```

Then we create sorting algorithms to sort those products. This code cannot be modified later on to fit specific needs of the client,

```
public interface SortingUtilityInterface {
    List<Product> sort(List<Product> items, int sortingApproach);
}

public class SortingUtility implements SortingUtilityInterface {
    @Override
    public List<Product> sort(List<Product> items, int sortingApproach) {
        if (sortingApproach == 1) {
            return bubbleSort(items);
        } else if (sortingApproach == 2) {
            return quickSort(items);
        } else {
            // ... Throw error
    }

    private List<Product> bubbleSort(List<Product> items) {
        // ... Sort items
        return items;
    }

    private List<Product> quickSort(List<Product> items) {
        // ... Sort items
        return items;
    }
}
```

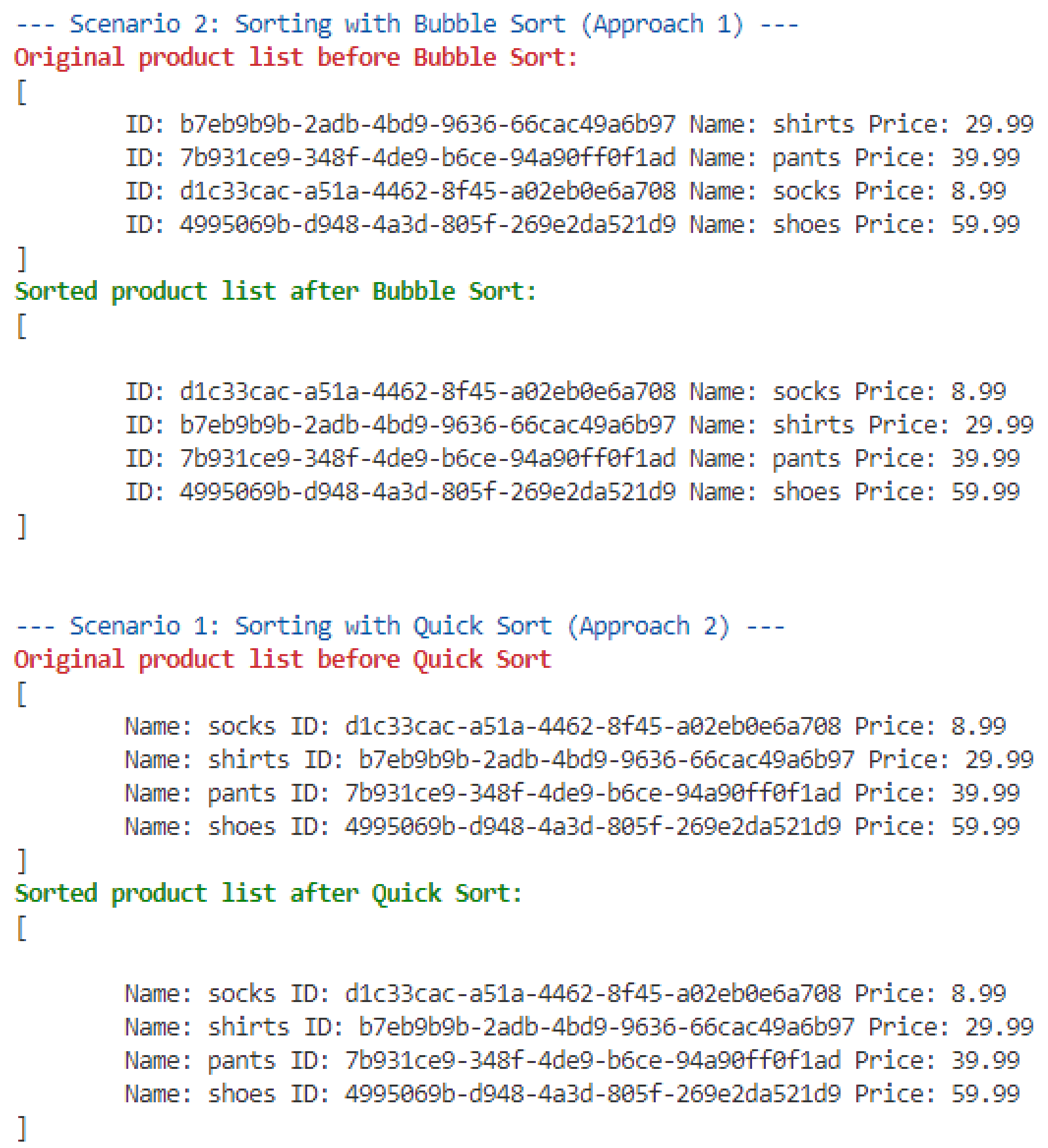
Since the client wants to log the items after sorting, we can use a proxy class to log the results after calling the algorithm,

```
public class SortingUtilityProxy implements SortingUtilityInterface {
    private final SortingUtilityInterface sortingUtility = new SortingUtility()

    @Override
    public List<Product> sort(List<Product> items, int sortingApproach) {
        List<Product> sortedItems = sortingUtility.sort(items, sortingApproach)

        if (sortingApproach == 1) {
            for (Product item : sortedItems) {
                // ... Log the item
           }
        } else if (sortingApproach == 2) {
            for (Product item : sortedItems) {
                // ... Log the item
            }
        }

        return sortedItems;
    }
}
```

Now, the client can use the proxy to both sort the items as well as log the results.

```
public class Main {
    public static void main(String[] args) {
        // Create products
        List<Product> products = new ArrayList<>();
        products.add(new Product("shirts", 29.99));
        products.add(new Product("pants", 39.99));
        products.add(new Product("socks", 8.99));
        products.add(new Product("shoes", 59.99));

        SortingUtilityProxy proxy = new SortingUtilityProxy();

        // Scenario 1: Sorting with Bubble Sort (Approach 1)
        for (Product product : products) {
            // ... Print out products before sorting
        }

        System.out.println("Sorted product list after Bubble Sort: ");
        proxy.sort(products, 1);

        // Scenario 2: Sorting with Quick Sort (Approach 2)
        for (Product product : products) {
            // ... Print out products before sorting
        }
        System.out.println("Sorted product list after Quick Sort: ");
        proxy.sort(products, 2);
    }
}
```

## Screenshots of code running



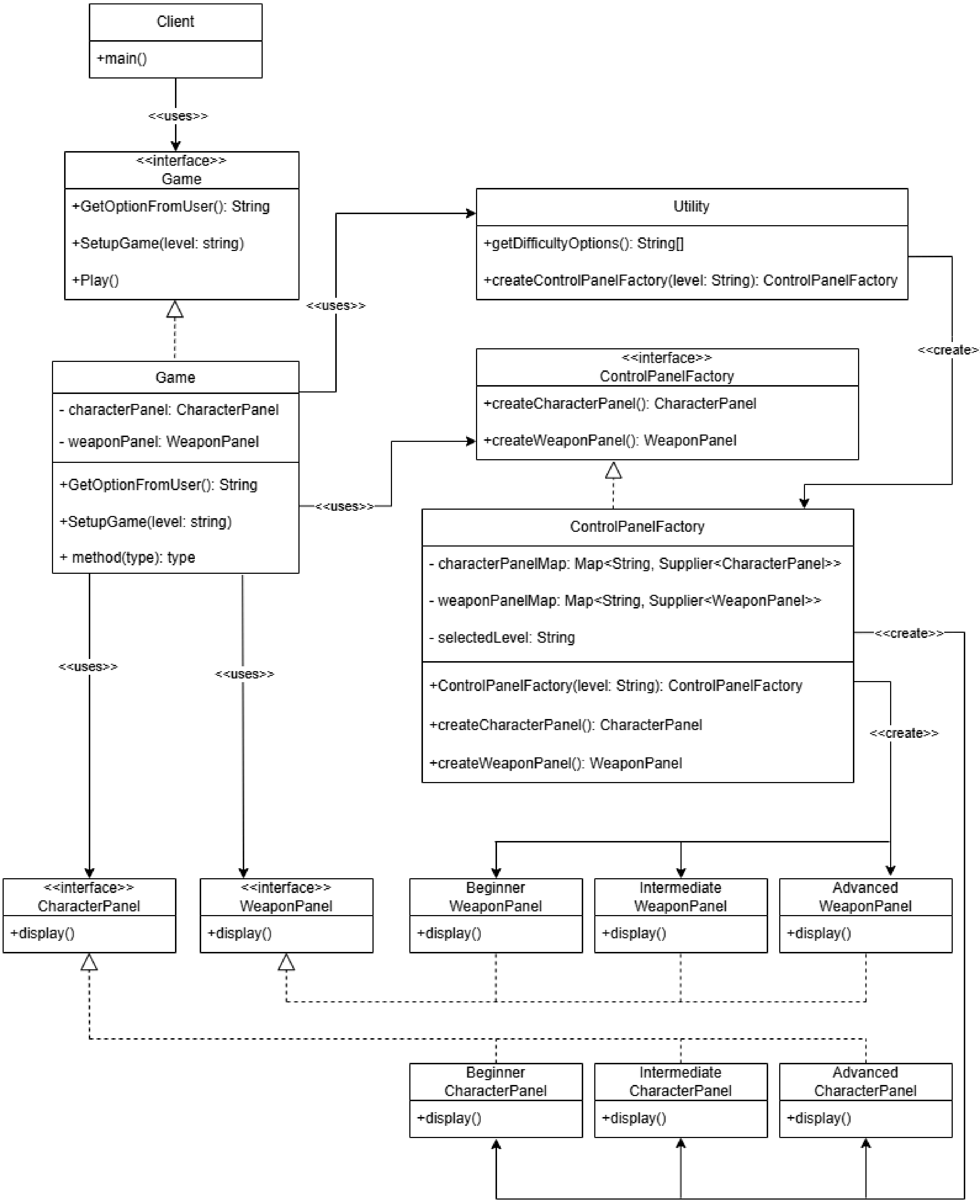
```
--- Scenario 2: Sorting with Bubble Sort (Approach 1) ---
Original product list before Bubble Sort:
[
        ID: b7eb9b9b-2adb-4bd9-9636-66cac49a6b97 Name: shirts Price: 29.99
        ID: 7b931ce9-348f-4de9-b6ce-94a90ff0f1ad Name: pants Price: 39.99
        ID: d1c33cac-a51a-4462-8f45-a02eb0e6a708 Name: socks Price: 8.99
        ID: 4995069b-d948-4a3d-805f-269e2da521d9 Name: shoes Price: 59.99
]
Sorted product list after Bubble Sort:
[

        ID: d1c33cac-a51a-4462-8f45-a02eb0e6a708 Name: socks Price: 8.99
        ID: b7eb9b9b-2adb-4bd9-9636-66cac49a6b97 Name: shirts Price: 29.99
        ID: 7b931ce9-348f-4de9-b6ce-94a90ff0f1ad Name: pants Price: 39.99
        ID: 4995069b-d948-4a3d-805f-269e2da521d9 Name: shoes Price: 59.99
]


--- Scenario 1: Sorting with Quick Sort (Approach 2) ---
Original product list before Quick Sort
[
        Name: socks ID: d1c33cac-a51a-4462-8f45-a02eb0e6a708 Price: 8.99
        Name: shirts ID: b7eb9b9b-2adb-4bd9-9636-66cac49a6b97 Price: 29.99
        Name: pants ID: 7b931ce9-348f-4de9-b6ce-94a90ff0f1ad Price: 39.99
        Name: shoes ID: 4995069b-d948-4a3d-805f-269e2da521d9 Price: 59.99
]
Sorted product list after Quick Sort:
[

        Name: socks ID: d1c33cac-a51a-4462-8f45-a02eb0e6a708 Price: 8.99
        Name: shirts ID: b7eb9b9b-2adb-4bd9-9636-66cac49a6b97 Price: 29.99
        Name: pants ID: 7b931ce9-348f-4de9-b6ce-94a90ff0f1ad Price: 39.99
        Name: shoes ID: 4995069b-d948-4a3d-805f-269e2da521d9 Price: 59.99
]
```

## Problem 2

The game has three modes (beginner, intermediate, advanced), each displaying a different character selection panel and weapon selection panel. Future extensions might introduce new modes or new types of control objects, making it necessary to design a system that supports flexibility and scalability without modifying the client-side code.

As such, the design pattern that is ideal is the `ABSTRACT FACTORY` pattern , which is a design pattern that allows the creation of families of related objects without specifying their concrete classes in the client code.

## UML Diagram



Client
+main()
<<uses>>
<<interface>>
Game
+GetOptionFromUser(): String
+SetupGame(level: string)
+Play()
<<uses>>
Utility
+getDifficultyOptions(): String[]
+createControlPanelFactory(level: String): ControlPanelFactory
<<create>>
Game
- characterPanel: CharacterPanel
- weaponPanel: WeaponPanel
+GetOptionFromUser(): String
+SetupGame(level: string)
+ method(type): type
<<interface>>
ControlPanelFactory
+createCharacterPanel(): CharacterPanel
+createWeaponPanel(): WeaponPanel
<<uses>>
ControlPanelFactory
- characterPanelMap: Map<String, Supplier<CharacterPanel>>
- weaponPanelMap: Map<String, Supplier<WeaponPanel>>
- selectedLevel: String
+ControlPanelFactory(level: String): ControlPanelFactory
+createCharacterPanel(): CharacterPanel
+createWeaponPanel(): WeaponPanel
<<create>>
<<create>>
<<uses>>
<<uses>>
<<interface>>
CharacterPanel
+display()
<<interface>>
WeaponPanel
+display()
Beginner
WeaponPanel
+display()
Intermediate
WeaponPanel
+display()
Advanced
WeaponPanel
+display()
Beginner
CharacterPanel
+display()
Intermediate
CharacterPanel
+display()
Advanced
CharacterPanel
+display()

## Code implementation (Full code attached in submission)

First we define our Abstract Panels,

```
abstract public interface CharacterPanel {
    void display();
}

abstract public interface WeaponPanel {
    void display();
}
```

Then, we create different concrete class difficulty of those panels

```
public class AdvancedCharacterPanel implements CharacterPanel {
    public void display() {
        System.out.println("Advanced Character Panel");
    }
}

public class AdvancedWeaponPanel implements WeaponPanel {
    public void display() {
        System.out.println("Advanced Weapon Panel");
    }
}
```

Afterwards, we create a factory class that returns these panels based on its difficulty from its constructor,

```
interface ControlPanelFactoryInterface {
    CharacterPanel createCharacterPanel();
    WeaponPanel createWeaponPanel();
}

// Concrete Factory Class that creates control panel instances
public class ControlPanelFactory implements ControlPanelFactoryInterface {
    private static Map<String, Supplier<CharacterPanel>> characterPanelMap
    private static Map<String, Supplier<WeaponPanel>> weaponPanelMap
    private String selectedLevel;

    static {
        characterPanelMap.put("advanced", AdvancedCharacterPanel::new);
        weaponPanelMap.put("advanced", AdvancedWeaponPanel::new);

        //  ... Add more panels here
    }

    public ControlPanelFactory(String level) {
        this.selectedLevel = level.toLowerCase();
    }

    public CharacterPanel createCharacterPanel() {
        Supplier<CharacterPanel> s = characterPanelMap.get(selectedLevel);

        if (s != null) {
            return s.get();
        }

        // ... Throw error
    }

    public WeaponPanel createWeaponPanel() {
        Supplier<WeaponPanel> s = weaponPanelMap.get(selectedLevel);

        if (s != null) {
            return s.get();
        }

        // ... Throw error
    }
}
```

We will now create a utility class that defines the difficulties and also creates the factory based on the given level. In addition, we can also check for invalid inputs here as well,

```
public class Utility {
    // ... Add more difficulty levels here
    private static String[] difficulties = new String[] { "advanced" }

    public static ControlPanelFactory createControlPanelFactory(String level) {
        if (!Arrays.asList(difficulties).contains(level.toLowerCase())) {
            // ... Throw error
        }
        return new ControlPanelFactory(level);
    }

    public static String[] getDifficultyOptions() {
        return difficulties;
    }
}
```

Now, we can finally define a game class that defines the basic functionalities of the game

```
interface GameInterface {
    void SetupGame(String level);
    String GetOptionFromUser();
    void Play();
}

// Concrete class that implements GameInterface
public class Game implements GameInterface {
    private CharacterPanel characterPanel;
    private WeaponPanel weaponPanel;

    public void SetupGame(String level) {
        ControlPanelFactory factory = Utility.createControlPanelFactory(level)
        characterPanel = factory.createCharacterPanel();
        weaponPanel = factory.createWeaponPanel();
    }

    public String GetOptionFromUser() {
        String[] options = Utility.getDifficultyOptions();
        // ... User selects option
        return selectedOption;
    }

    public void Play() {
        characterPanel.display();
        weaponPanel.display();
    }
}
```

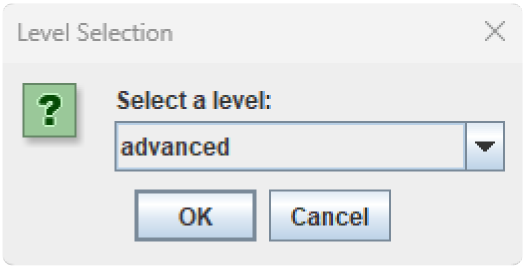
The client can now use this game interface to set up and play the game,

```
public class Client {
    public static void main(String[] args) {
        Game game = new Game();
        String level = game.GetOptionFromUser();
        game.SetupGame(level);
        game.Play();
    }
}
```

Note that, the client and the game itself does not need to be changed if a new difficulty is added. Thus adding an abstraction layer that allows great scalability for new additions.

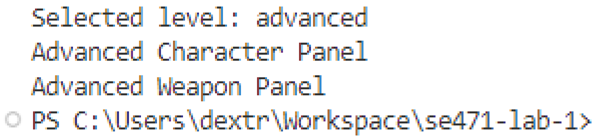
## Screenshots of code running

1. User selects a level



Select a level:
advanced

2. The game displays the panels based on the level chosen by the user



```
Selected level: advanced
Advanced Character Panel
Advanced Weapon Panel
PS C:\Users\dextr\Workspace\se471-lab-1>
```