Объектно-ориентированное программирование

Задание 1.

Рефакторинг проекта с семейным деревом с учетом принципов SOLID.

Подсказка № 1

Принципы SOLID и их применение:

1. Single Responsibility Principle (SRP): Каждый класс должен иметь одну ответственность.
2. Open/Closed Principle (OCP): Классы должны быть открыты для расширения, но закрыты для модификации.
3. Liskov Substitution Principle (LSP): Объекты должны быть заменяемы их подтипами без изменения правильности программы.
4. Interface Segregation Principle (ISP): Клиенты не должны зависеть от интерфейсов, которые они не используют.
5. Dependency Inversion Principle (DIP): Зависимости должны быть инвертированы, т.е. модули высшего уровня не должны зависеть от модулей низшего уровня, а обе группы должны зависеть от абстракций.

Подсказка № 2

Разделение интерфейсов (ISP). Разделите интерфейс `TreeView` на несколько специализированных интерфейсов (`MessageView`, `PersonView`, `InputView`), каждый из которых будет отвечать только за один аспект взаимодействия с пользователем. Убедитесь, что класс `ConsoleTreeView` реализует все необходимые интерфейсы, чтобы сохранить полную функциональность, но без излишней зависимости от ненужных методов.

Подсказка № 3

Избавление от сложной логики в презентере (SRP). Перенесите логику обработки команд и ввода пользователя из `TreePresenter` в отдельный класс, например, `CommandHandler`. Это позволит `TreePresenter` сосредоточиться на управлении данными и бизнес-логике. Убедитесь, что `CommandHandler` берет команды из `TreeView` и делегирует их `TreePresenter`, сохраняя код чистым и разделенным по ответственности.

Подсказка № 4

Инверсия зависимостей (DIP). Убедитесь, что `TreePresenter` и другие классы зависят от абстракций, а не от конкретных реализаций. Например, `TreePresenter` должен зависеть от интерфейсов `MessageView`, `PersonView`, и `InputView`, а не от конкретной реализации `ConsoleTreeView`. Проверьте, что `TreePresenter` и другие

классы используют интерфейсы для взаимодействия с внешними сервисами (например, `FileOperations`), чтобы можно было легко подменить реализации без изменения основной логики.

Подсказка № 5

В `Main` классе создайте и свяжите все компоненты, следуя новым интерфейсам и классам. Убедитесь, что все зависимости инжектируются правильно. Используйте конструкторы для передачи зависимостей, а не жестко кодируйте их в классе.

Эталонное решение:

Пакет `view`:

1. **FamilyTree.java:**

```
package view;



import model.Person;

import java.util.List;



public interface TreeView extends MessageView, PersonView, InputView
{

    void setPresenter(TreePresenter presenter);

}
```

2. **MessageView.java:**

```
package view;



public interface MessageView {

    void displayMessage(String message);

}
```

3. **MersonView.java:**

```
package view;


import model.Person;

import java.util.List;


public interface PersonView {

    void displayPersons(List<Person> persons);

}
```

4. **PersonView.java:**

```
package view;


import model.Person;

import java.util.List;


public interface PersonView {

    void displayPersons(List<Person> persons);

}
```

5. **InputView.java :**

```
package view;


public interface InputView {

    String getUserInput();

}
```

6. **ConsoleTreeView.java :**

```
package view;
```

```
import model.Person;
import presenter.TreePresenter;
import java.util.List;
import java.util.Scanner;

public class ConsoleTreeView implements TreeView {
    private TreePresenter presenter;
    private Scanner scanner;

    public ConsoleTreeView() {
        this.scanner = new Scanner(System.in);
    }

    @Override
    public void displayMessage(String message) {
        System.out.println(message);
    }

    @Override
    public void displayPersons(List<Person> persons) {
        for (Person person : persons) {
            System.out.println(person.getName() + ", born in " +
person.getBirthYear());
        }
    }
```

```
    @Override
    public String getUserInput() {
        return scanner.nextLine();
    }

    @Override
    public void setPresenter(TreePresenter presenter) {
        this.presenter = presenter;
    }
}
```

Пакет presenter

1. **TreePresenter.java:**

```
package presenter;

import model.FamilyTree;
import model.Person;
import service.FileOperations;
import view.MessageView;
import view.PersonView;
import view.InputView;
import java.io.IOException;

public class TreePresenter {
    private FamilyTree<Person> familyTree;
    private MessageView messageView;
```

```
    private PersonView personView;

    private InputView inputView;

    private FileOperations<Person> fileOperations;



    public TreePresenter(FamilyTree<Person> familyTree, MessageView
messageView, PersonView personView, InputView inputView,
FileOperations<Person> fileOperations) {

        this.familyTree = familyTree;

        this.messageView = messageView;

        this.personView = personView;

        this.inputView = inputView;

        this.fileOperations = fileOperations;

    }



    public void addPerson(String name, int birthYear) {

        Person person = new Person(name, birthYear);

        familyTree.addMember(person);

        messageView.displayMessage("Person added: " + name);

    }



    public void showAllPersons() {

        personView.displayPersons(familyTree.getMembers());

    }



    public void sortPersonsByName() {

        familyTree.sortByName();

        messageView.displayMessage("Persons sorted by name:");
```

```
        showAllPersons();
    }

    public void sortPersonsByBirthYear() {
        familyTree.sortByBirthYear();
        messageView.displayMessage("Persons sorted by birth year:");
        showAllPersons();
    }

    public void saveTree(String fileName) {
        try {
            fileOperations.saveToFile(familyTree, fileName);
            messageView.displayMessage("Family tree saved to " +
fileName);
        } catch (IOException e) {
            messageView.displayMessage("Error saving family tree: "
+ e.getMessage());
        }
    }

    public void loadTree(String fileName) {
        try {
            familyTree = fileOperations.loadFromFile(fileName);
            messageView.displayMessage("Family tree loaded from " +
fileName);
        } catch (IOException | ClassNotFoundException e) {
            messageView.displayMessage("Error loading family tree: "
+ e.getMessage());
```

```
        }

    }


    public void handleCommand(String command) {

        switch (command) {

            case "add":

                messageView.displayMessage("Enter name:");

                String name = inputView.getUserInput();

                messageView.displayMessage("Enter birth year:");

                int birthYear =
Integer.parseInt(inputView.getUserInput());

                addPerson(name, birthYear);

                break;

            case "list":

                showAllPersons();

                break;

            case "sortByName":

                sortPersonsByName();

                break;

            case "sortByBirthYear":

                sortPersonsByBirthYear();

                break;

            case "save":

                messageView.displayMessage("Enter file name:");

                saveTree(inputView.getUserInput());

                break;

            case "load":
```

```
                messageView.displayMessage("Enter file name:");
                loadTree(inputView.getUserInput());
                break;
            case "exit":
                System.exit(0);
            default:
                messageView.displayMessage("Unknown command");
        }
    }
}
```

2. **CommandHandler.java:**

```
package presenter;


import view.PresenterView;


public class CommandHandler {
    private TreePresenter presenter;
    private PresenterView view;


    public CommandHandler(TreePresenter presenter, PresenterView
view) {
        this.presenter = presenter;
        this.view = view;
    }


    public void handleUserInput() {
        while (true) {
```

```
            view.displayMessage("Enter command (add, list,
sortByName, sortByBirthYear, save, load, exit):");

            String command = view.getUserInput();

            presenter.handleCommand(command);

        }

    }

}
```

Пакет `main`:

1. **Main.java:**

```
package main;



import model.FamilyTree;

import model.Person;

import presenter.CommandHandler;

import presenter.TreePresenter;

import service.FileOperationsImpl;

import view.ConsoleTreeView;



public class Main {

    public static void main(String[] args) {

        FamilyTree<Person> familyTree = new FamilyTree<>();

        ConsoleTreeView view = new ConsoleTreeView();

        FileOperationsImpl<Person> fileOperations = new
FileOperationsImpl<>();



        TreePresenter presenter = new TreePresenter(familyTree,
view, view, view, fileOperations);
```

```
        CommandHandler commandHandler = new
CommandHandler(presenter, view);



        commandHandler.handleUserInput();

    }

}
```

Пакет service:

1. **FileOperations.java:**

```
package service;



import model.FamilyTree;

import java.io.IOException;



public interface FileOperations<T> {

    void saveToFile(FamilyTree<T> familyTree, String fileName)
throws IOException;

    FamilyTree<T> loadFromFile(String fileName) throws IOException,
ClassNotFoundException;

}
```

2. **FileOperationsImpl.java:**

```
package service;



import model.FamilyTree;

import java.io.*;



public class FileOperationsImpl<T> implements FileOperations<T> {
```

```
    @Override

    public void saveToFile(FamilyTree<T> familyTree, String
fileName) throws IOException {

        try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(fileName))) {

            oos.writeObject(familyTree);

        }

    }



    @Override

    public FamilyTree<T> loadFromFile(String fileName) throws
IOException, ClassNotFoundException {

        try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(fileName))) {

            return (FamilyTree<T>) ois.readObject();

        }

    }

}
```

Пакет `model`:

1. **Person.java**

```
package model;



import java.io.Serializable;



public class Person implements Serializable {

    private String name;

    private int birthYear;
```

```
    public Person(String name, int birthYear) {
        this.name = name;
        this.birthYear = birthYear;
    }

    public String getName() {
        return name;
    }

    public int getBirthYear() {
        return birthYear;
    }
}
```

2. **FamilyTree.java**

```
package model;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

public class FamilyTree<T extends Person> implements Serializable {
    private List<T> members = new ArrayList<>();

    public void addMember(T person) {
        members.add(person);
```

```
    }

    public List<T> getMembers() {
        return new ArrayList<>(members);
    }

    public void sortByName() {
        members.sort(Comparator.comparing(Person::getName));
    }

    public void sortByBirthYear() {
        members.sort(Comparator.comparingInt(Person::getBirthYear));
    }
}
```