

Table of Contents

1. Single Responsibility Principle.....	2
1.1. Bad package	2
1.2. Good Package	3
2. Open-Closed Principle.....	5
2.1. Bad Package	5
2.2. Good Package	5
3. Liskov Substitution Principle.....	7
3.1. Bad package	7
3.2. Good package.....	7
4. Interface Segregation Principle.....	9
4.1. Bad package	9
4.2. Good package.....	9
5. Dependency Inversion Principle	11
5.1. Bad package	11
5.2. Good Package	11

Table of Figures

Figure 1: Room class structure showcasing SRP	2
Figure 2: Room class structure after fixing SRP	3
Figure 3: User (abstract) class structure after fixing SRP	3
Figure 4: Student class structure after fixing SRP	4
Figure 5: Teacher class structure after fixing SRP.....	4
Figure 6: Start event function inside EventManager showcasing OCP violation	5
Figure 7: Start event method implementation after fixing OCP	6
Figure 8: Set profile image method implementation showcasing LSP violation	7
Figure 9: Set profile image implementation after fixing LSP.....	8
Figure 10: Message interface declaration for ISP violation demonstration	9
Figure 11: Message interface after fixing ISP	10
Figure 12: Edited interface after fixing ISP	10
Figure 13: Replied interface after fixing ISP	10
Figure 14: GitManager constructor showcasing DIP violation	11
Figure 15: GitManager constructor after fixing DIP	12

1. Single Responsibility Principle

1.1. Bad package

Files:

- **JitsiMeet**: Mock class of third-party library that provides support for hosting meeting with specific configurations, adding user, removing user, etc.
- **ScreenSharingOptions**: Helper enum class for defining all options while sharing screen in a meeting.
- **OnlineMeeting**: Interface defining set of functionalities that are needed by any meeting host class needs to provide, which is JitsiMeet in my case.
- **Room**: Room is online classroom where teacher teaches student remotely.
- **OnlineLearningPlatform**: Main java class to showcase dummy usage of all classes. Real world implementation/usage for all classes added vary according to requirement, this is just an example to make sure code is successfully compiled and ready to run.

Violation:

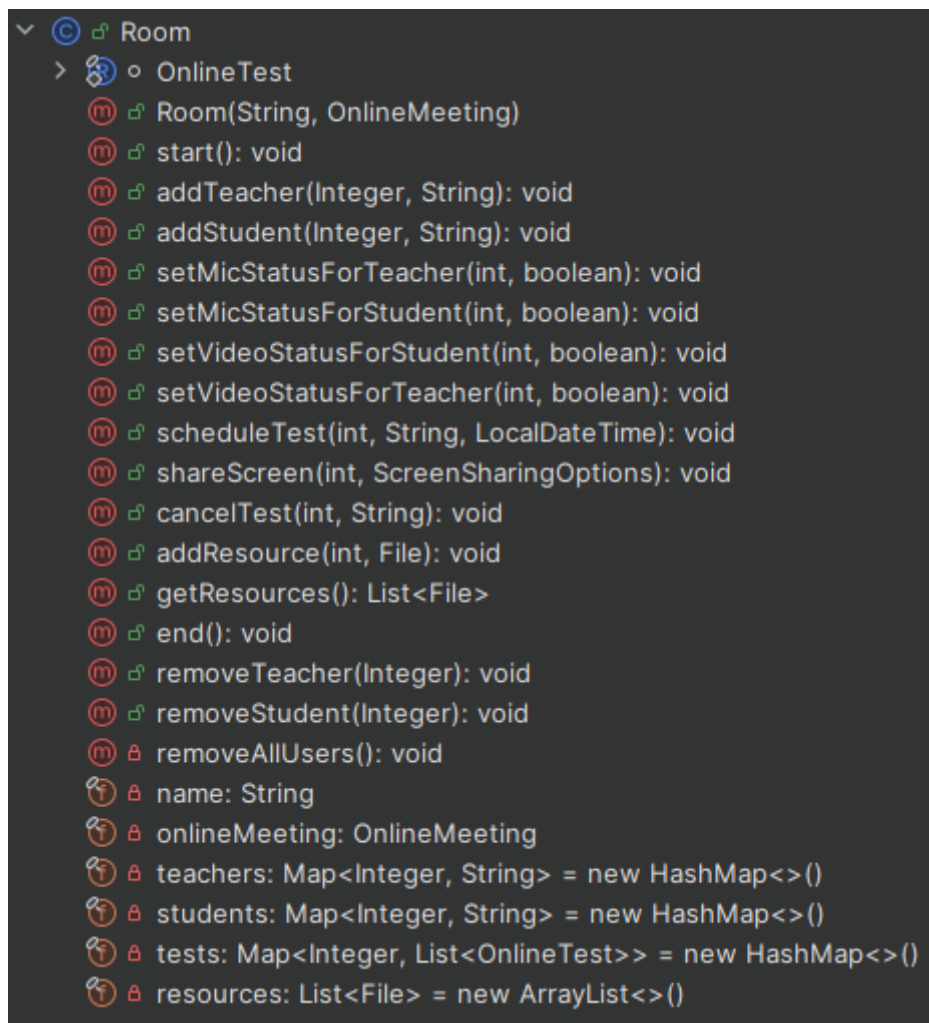


Figure 1: Room class structure showcasing SRP violation

Here, Room class handles lots of responsibilities including its own and of Teacher and Student which is violating SRP which states a class should have only one and only one reason to change. Regarding teacher's responsibility, it is providing schedule test and cancel test functionality which is totally dependent on teacher and not on room. Also, for each type of user, it provides separate functions for handling each users responsibility which again violates SRP.

1.2. Good Package

Files added apart from files in bad package:

- **User:** Abstract class which defines a user that can join into a Room.
- **Student:** Concrete class extending User class and providing its responsibilities too.
- **Teacher:** Concrete class extending User class and providing its responsibilities too.

To fix Single Responsibility Principle, I split the responsibilities into individual classes from Room to Room, Student and Teacher.

```

Room
  Room(String, OnlineMeeting)
  start(): void
  admit(int): void
  remove(int): void
  setMicStatus(int, boolean): void
  setVideoStatus(int, boolean): void
  shareScreen(int, ScreenSharingOptions): void
  addResource(File): void
  getResources(): List<File>
  end(): void
  removeAllUsers(): void
  name: String
  onlineMeeting: OnlineMeeting
  attendees: ArrayList<Integer> = new ArrayList<>()
  resources: List<File> = new ArrayList<>()

```

Figure 2: Room class structure after fixing SRP

```

User
  User(int, String)
  getName(): String
  joinRoom(Room): void
  leaveRoom(Room): void
  setMicStatus(Room, boolean): void
  setVideoStatus(Room, boolean): void
  shareScreen(Room, ScreenSharingOptions): void
  userId: int
  userName: String

```

Figure 3: User (abstract) class structure after fixing SRP

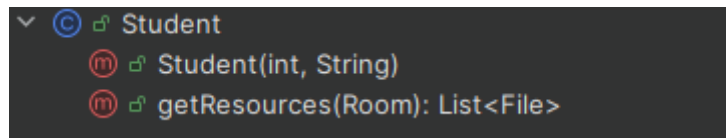


Figure 4: Student class structure after fixing SRP

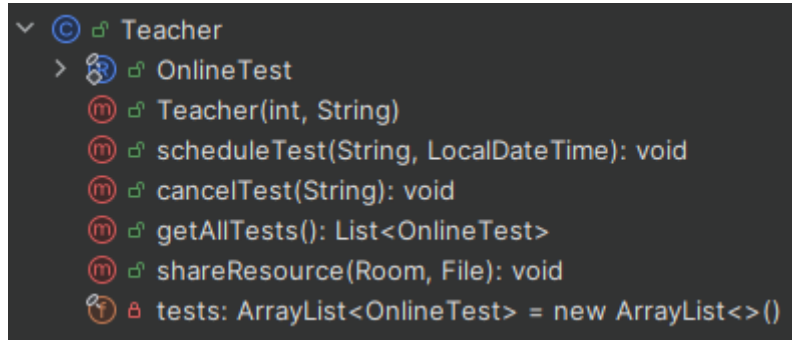


Figure 5: Teacher class structure after fixing SRP

In this example, there isn't much about students to mention but this is general example of how we should split multiple responsibility from one class to multiple classes as horizontally scaling your code is always better than vertical scaling.

2. Open-Closed Principle

2.1. Bad Package

Files:

- **Event:** Record class for storing all details of the event.
- **EventManager:** Manger class to manage all events, basically CRUD on events and handling starting event.
- **EventValidator:** Validator class for event which deals with all event validations.
- **EventManagementApplication:** Main java class to showcase dummy usage of all classes. Real world implementation/usage for all classes added vary according to requirement, this is just an example to make sure code is successfully compiled and ready to run.

Violation:

```
62     public void startEvent(int eventId) {
63         // Check if event ID is valid or not
64         if (eventValidator.isEventIdNotAdded(eventId, events)) {
65             return;
66         }
67         String eventType = getEventType(eventId);
68         // Check if event type is valid or not
69         if (eventType == null) {
70             System.out.println("Cannot get event type for event with id " + eventId + " not found");
71             return;
72         }
73         switch (eventType) {
74             case "GamesNight" -> setupGamesNightEvent();
75             case "MondayNightMeal" -> setupMondayNightMealEvent();
76             case "MovieNight" -> setupMovieNightEvent();
77             case "SnowballFight" -> setupSnowballFightEvent();
78             case "CanadaDayFireworks" -> setupCanadaDayFireworksEvent();
79             default -> System.out.println("Invalid event type to start");
80         }
81         removeEvent(eventId);
82     }
```

Figure 6: Start event function inside EventManager showcasing OCP violation

Here event type is a static string on which the function decides on which function to call further on for setting up the event. Above code is violating OCP as, whenever a new event type is registered, developer will have to add new case over here and create separate function for setting up event, violating the closed for modification rule.

2.2. Good Package

Files added apart from files in bad package:

- **EventType:** Interface defining a separate type of event and defines its functionalities.
- 5 different classes implementing EventType and providing its functionalities

To fix OCP, I created this interface EventType and passed it's child class instance in Event record class. By doing so, my method startEvent will look like:

```

60     public void startEvent(int eventId) {
61         // Check if event ID is valid or not
62         if (eventValidator.isEventIdNotAdded(eventId, events)) {
63             return;
64         }
65         Optional<Event> event = events.stream().filter(e -> e.id() == eventId).findFirst();
66         if (event.isEmpty()) {
67             System.out.println("Event not found with id: " + eventId);
68             return;
69         }
70         event.get().type().setupEventEnvironment();
71         removeEvent(eventId);

```

Figure 7: Start event method implementation after fixing OCP

This way each sub class of EventType provides implementation of setup method and we won't have to modify implementation here but only need to create new class if we want to extend our event type, this way fixing Open Closed Principle.

3. Liskov Substitution Principle

3.1. Bad package

Files:

- **ImageUploader:** Interface defining blueprint for a class that provides image upload functionality.
- **FirebaseImageUploader & GoogleImageUploader:** Imager uploader classes that implement ImageUploader and provides URL in return once image uploaded.
- **AmazonImageUploader:** Imager uploader classes that implement ImageUploader and does not provide URL in return once image uploaded.
- **UserProfileManager:** Manager class for handling user profile.
- **UserProfileManagement:** Main java class to showcase dummy usage of all classes. Real world implementation/usage for all classes added vary according to requirement, this is just an example to make sure code is successfully compiled and ready to run.

Violation:

```
64     public String setProfileImage(File profileImage, ImageUploader imageUploader) {  
65         if (profileImage == null) {  
66             System.out.println("Profile image is null");  
67             return null;  
68         }  
69         // URL is expected in return  
70         profileImageUrl = imageUploader.uploadImage(profileImage);  
71         return profileImageUrl;  
72     }
```

Figure 8: Set profile image method implementation showcasing LSP violation

Here, when the function, setProfileImage is called, it expects URL string in return when it uploads the profile image for a user. When Firebase and Google's class is provided as ImageUploader, it will work as expected but when Amazon's class is provided as ImageUploader, the class won't have any value in return and will lead to empty profile image even after uploading. This way, Liskov Substitution Principle is violated as on substituting child will lead to inconsistency and lead to errors in program going further.

3.2. Good package

Files added apart from files in bad package:

- **SimpleImageUploader:** Abstract class implementing ImageUploader which states that implementing class only uploads image and does not provide anything in return.
- **UrlProvidingImageUploader:** Abstract class implementing ImageUploader which states that implementing class uploads image and provides URL in return.

```

63     public String setProfileImage(File profileImage, UrlProvidingImageUploader imageUploader) {
64         if (profileImage == null) {
65             System.out.println("Profile image is null");
66             return null;
67         }
68         // URL is expected in return
69         profileImageUrl = imageUploader.uploadImage(profileImage);
70         return profileImageUrl;
71     }
72
73     public void setProfileImage(File profileImage, SimpleImageUploader imageUploader) {
74         if (profileImage == null) {
75             System.out.println("Profile image is null");
76             return;
77         }
78         // URL is NOT expected in return
79         imageUploader.uploadImage(profileImage);
80         profileImageUrl = null;
81     }

```

Figure 9: Set profile image implementation after fixing LSP

Doing so I ended up making two separate methods this will fix Liskov Substitution Principle as now any child provided for individual method, feature will work as expected.

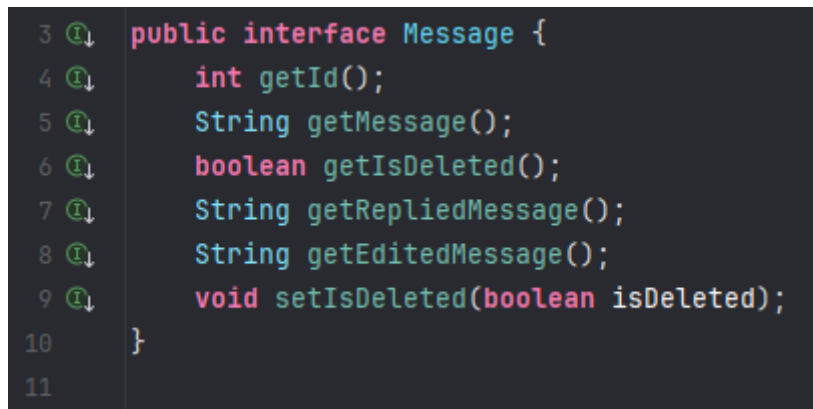
4. Interface Segregation Principle

4.1. Bad package

Files:

- **Message:** Interface (God interface) for a message shown in chat.
- **NormalMessage:** Class representing normal message which implements Message.
- **EditedMessage:** Class representing edited message which also implements Message.
- **RepliedMessage:** Class representing replied message which also implements Message.
- **Pair:** Utility record class to hold a pair of generic data types. (Similar to Pair in Kotlin language).
- **Participant:** Represents a chat participant
- **ChatManager:** Manager class for managing messages of a room, basically CRUD on messages + edit/reply/delete a message.
- **ChatApplication:** Application main class to demonstrate dummy usage of all classes. Real world implementation/usage for all classes added vary according to requirement, this is just an example to make sure code is successfully compiled and ready to run.

Violation:



```
3 public interface Message {
4     int getId();
5     String getMessage();
6     boolean getIsDeleted();
7     String getRepliedMessage();
8     String getEditedMessage();
9     void setIsDeleted(boolean isDeleted);
10 }
11
```

Figure 10: Message interface declaration for ISP violation demonstration

This interface is acting as God interface which basically has a lot of functionalities defined which not all children might be able to provide. For example, `getEditedMessage()` is only applicable to `EditedMessage` class and `getRepliedMessage()` is only applicable to `RepliedMessage` class yet all children implementing this interface needs to override all methods and handle extra methods which they don't have to provide. Hence, violating Interface Segregation Principle.

4.2. Good package

Files added apart from files in bad package:

- **Replied:** Interface specifically denoting replied message and moving its related function from Message to this interface.
- **Edited:** Interface specifically denoting edited message and moving its related function from Message to this interface.
- **BaseNormalMessage:** Abstract class which is parent class for all message implementation classes that provides common shared functionalities.

Rest files stay the same with minor modifications. By segregating the Message interface into 3 separate interfaces i.e. Message, Edited, and Replied, I gave flexibility to implementing classes in such a way, that they only need to provide implementation for methods they want to. This way, BaseNormalMessage class implements Message interface, EditedMessage class extends BaseNormalMessage and implements Edited interface and RepliedMessage class extends BaseNormalMessage and implements Replied interface. This was ISP violation is fixed.

```

3  public interface Message {
4
5      > /** Provides ID of the message ...*/
9      int getId();
10
11     > /** Provides the actual message ...*/
15     String getMessage();
16
17     > /** Provides if the message is deleted or not ...*/
21     boolean getIsDeleted();
22
23     > /** Set if the message is deleted or not ...*/
27     void setIsDeleted(boolean isDeleted);
28 }

```

Figure 11: Message interface after fixing ISP

```

3  public interface Edited {
4
5      > /** Provides the edited string of the message ...*/
9      String getEditedMessage();
10 }

```

Figure 12: Edited interface after fixing ISP

```

3  public interface Replied {
4
5      > /** Provides the replied string of the message ...*/
9      String getRepliedMessage();
10 }

```

Figure 13: Replied interface after fixing ISP

5. Dependency Inversion Principle

5.1. Bad package

Files:

- **GitManager:** Manager class to handle git operations locally.
- **GitHubManager:** Utility class to handle git operations remotely.
- **CustomIdeApplication:** Application main class to demonstrate dummy usage of all classes. Real world implementation/usage for all classes added vary according to requirement, this is just an example to make sure code is successfully compiled and ready to run.

Violation:

```
7      public class GitManager {  
8  
9          private final GitHubManager gitHubManager;  
10         private final List<String> remotes = new ArrayList<>();  
11         private List<File> stagedFiles = new ArrayList<>();  
12         private List<File> committedFiles = new ArrayList<>();  
13         private List<File> bufferedFiles = new ArrayList<>();  
14         private final List<String> branches = new ArrayList<>(List.of( e1: "main"));  
15         private String currentBranch = "main";  
16  
17         public GitManager(String email, String password) {  
18             gitHubManager = new GitHubManager(email, password);  
19             gitHubManager.loginUser();  
20         }
```

Figure 14: GitManager constructor showcasing DIP violation

Here, not only the class GitManager is directly dependent on concrete class GitHubManager but also creates its variable instance locally which violates Dependency Inversion Principle which state a class should not depend directly on its low-level modules.

5.2. Good Package

Files added apart from files in bad package:

- **RemoteGitProvider:** Interface providing blueprint for all operations a remote git provider must implement.

To fix DIP, I created this separate interface RemoteGitProvider and injected it into GitManager class making sure high-level modules do not depend on low level modules instead they depend on asbstraction (RemoteGitInterface). Doing this, it gives flexibility to GitManager class to adapt to changes in remote git provider class as it is only dependent on functionalities and not in concrete class.

```

7 public class GitManager {
8
9     private final RemoteGitProvider remoteGitProvider;
10    private final List<String> remotes = new ArrayList<>();
11    private final List<String> branches = new ArrayList<>(List.of("main"));
12
13    private List<File> stagedFiles = new ArrayList<>();
14    private List<File> committedFiles = new ArrayList<>();
15    private List<File> bufferedFiles = new ArrayList<>();
16    private String currentBranch = "main";
17
18    public GitManager(RemoteGitProvider remoteGitProvider) {
19        this.remoteGitProvider = remoteGitProvider;
20        this.remoteGitProvider.loginUser();
21    }

```

Figure 15: GitManager constructor after fixing DIP