DALHOUSIE
UNIVERSITY

ASSIGNMENT 2

Course: CSCI 5308 ASDC

Manan Mistry

Banner ID: B00948831

Repository Link: https://git.cs.dal.ca/courses/2023-summer/csci-5308/assignment2/mannm

Single-Responsibility Principle:

"Every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class, module or function. All its services should be narrowly aligned with that responsibility."[1]

Bad code example:

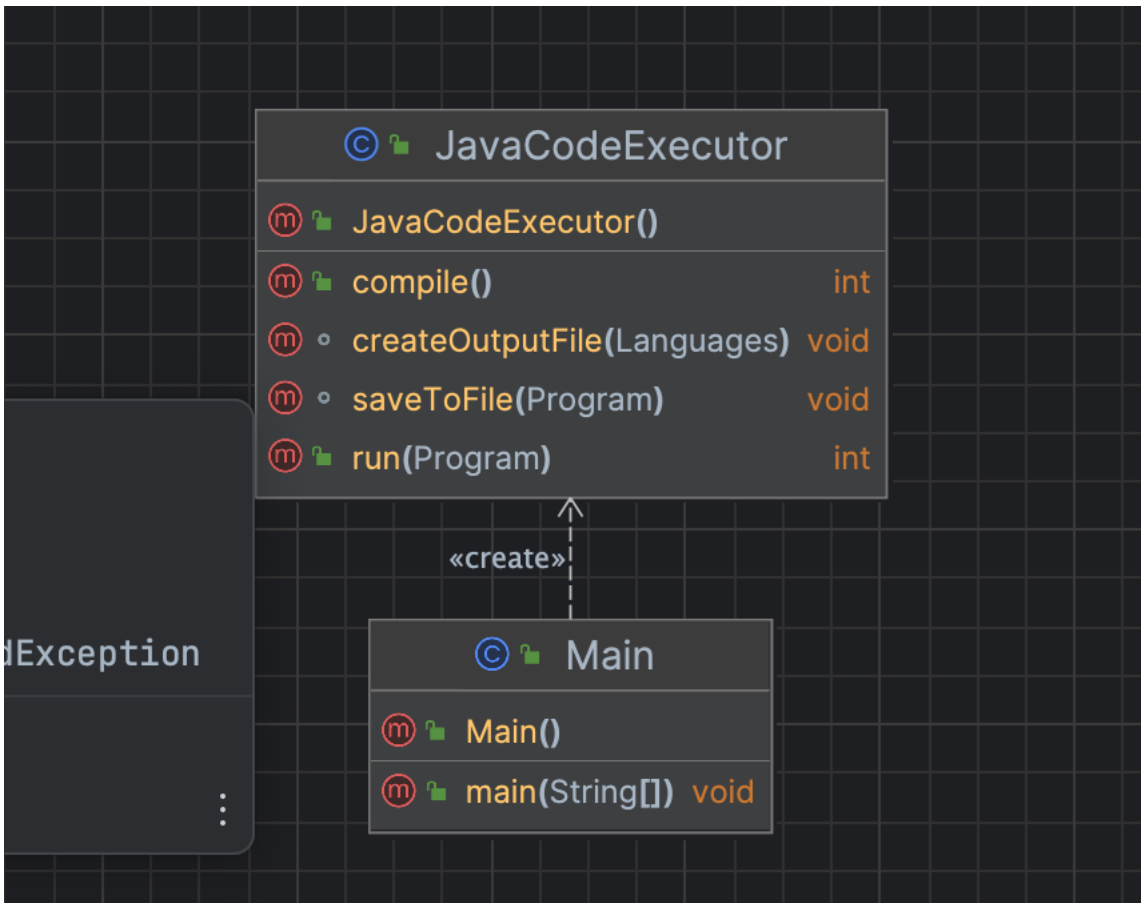

Figure 1UML Diagram of Single Responsibility Principle

Here in UML diagram you can see that there is a class called as JavaCodeExecutor, which is not only having logic related to compiling the program but it also has logic of saving the program into file and also generating outputfile which will store the result.

Now if we want to change anything related to file system, then we have to perform changes in JavaCodeExecutor class that violates the principle of Single-Responsibility. If we refine the problem then we can have following pattern.

Good code example:

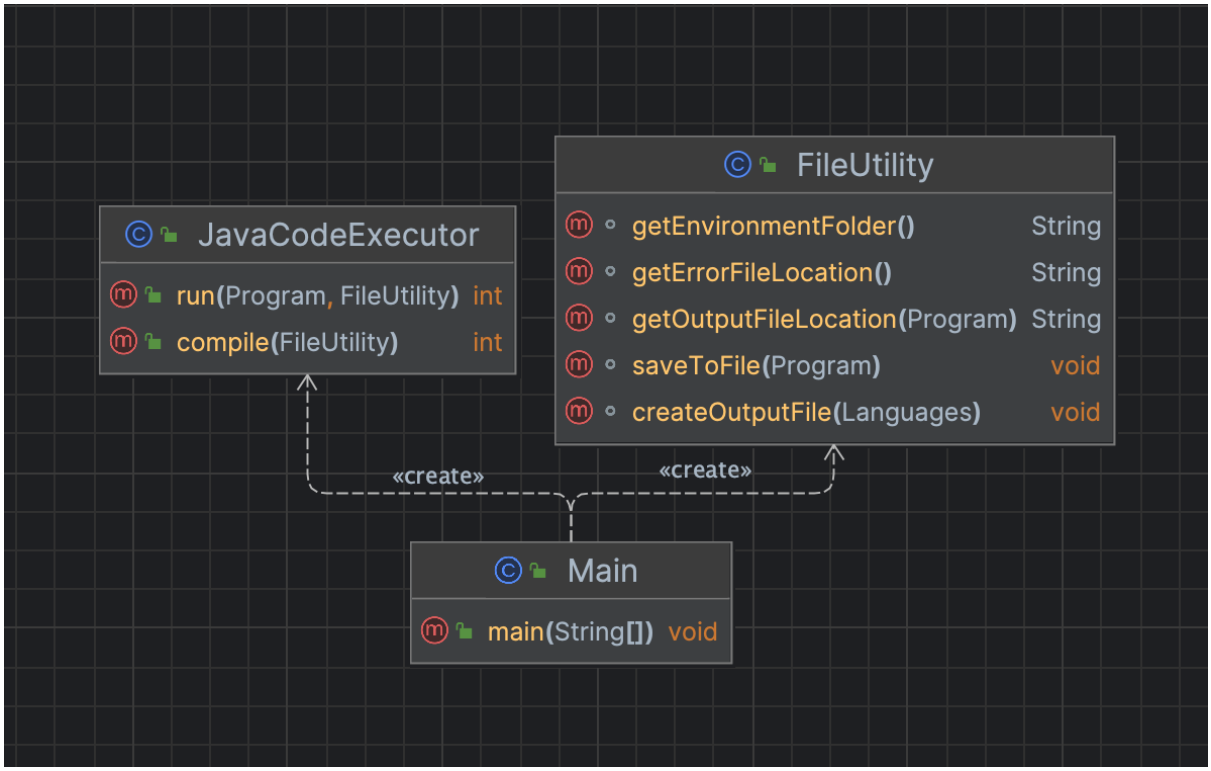

Figure 2 UML Diagram of Good Code of Single Responsibility Principle

As per the above UML diagram, FileUtility class has been created which will have logic related to File system, so in future if we have any changes in file system structure then, FileUtility class needs change, not the JavaCodeExecutor which should only be responsible for running the Java Program.

Open/Close:

Open/close principle states that, the Class should be open for extension but it should be closed for modification. Following example of notification sender will firstly show bad code and then how to optimize it.

Bad code example:

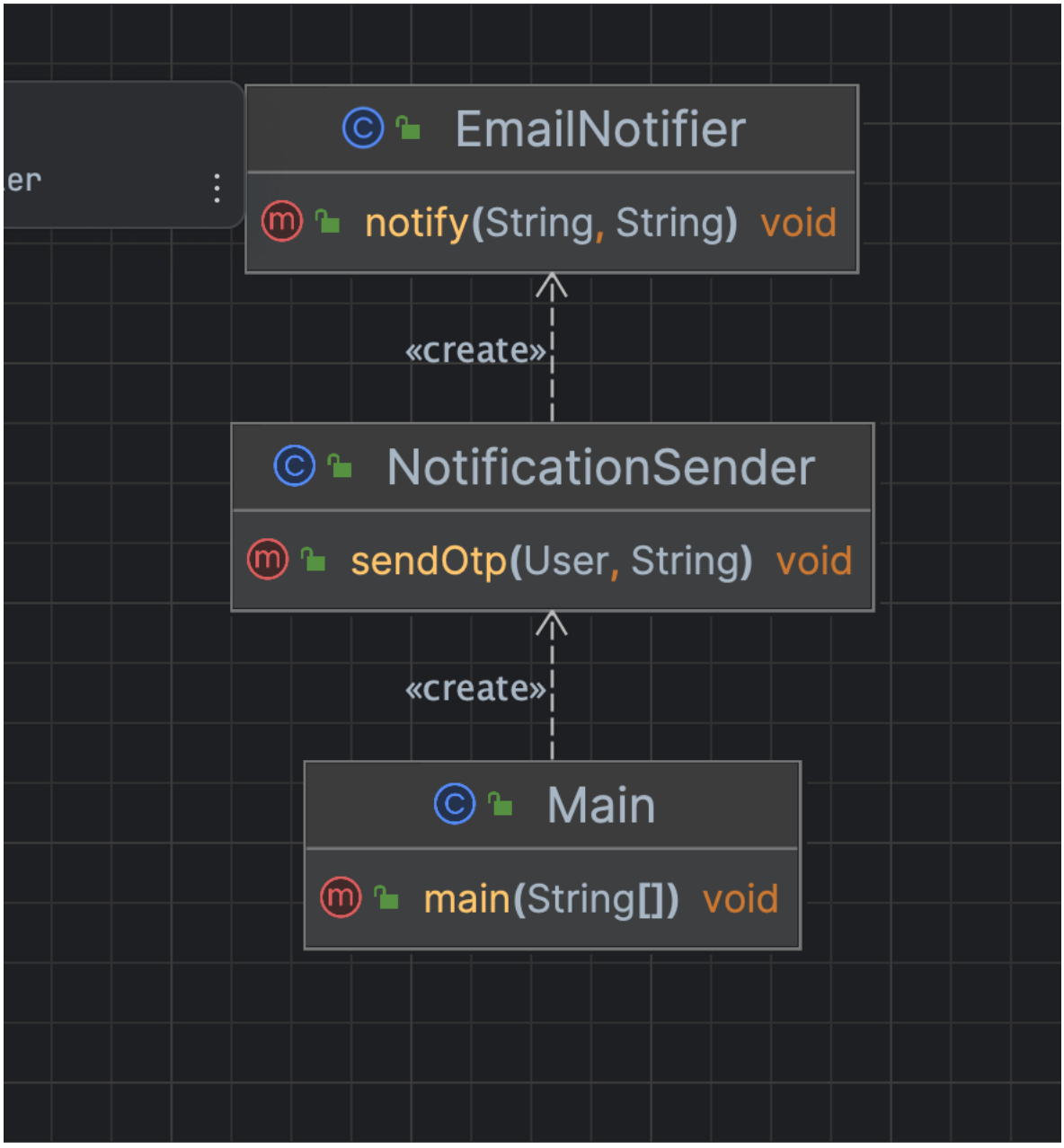

Figure 3 UML diagram of bad code of Open/Close Principle

```java
public class NotificationSender {
    1 usage   MANN MISTRY
    public void sendOtp(User user,String message){
        EmailNotifier emailNotifier=new EmailNotifier();
        emailNotifier.notify(user.getEmail(),message);
    }
}
```

Figure 4 Code of NotificationSender class

Here the sendOtp method under NotificationSender is tightly bound as it directly creates the object of EmailNotifier and utilizes it, now let's say if we want to add other medium to send notification such as SMS and Application notification then at that time we have to change the NotificationSender's sendOtp method, even though we want to just extend the functionality.

Good code example:

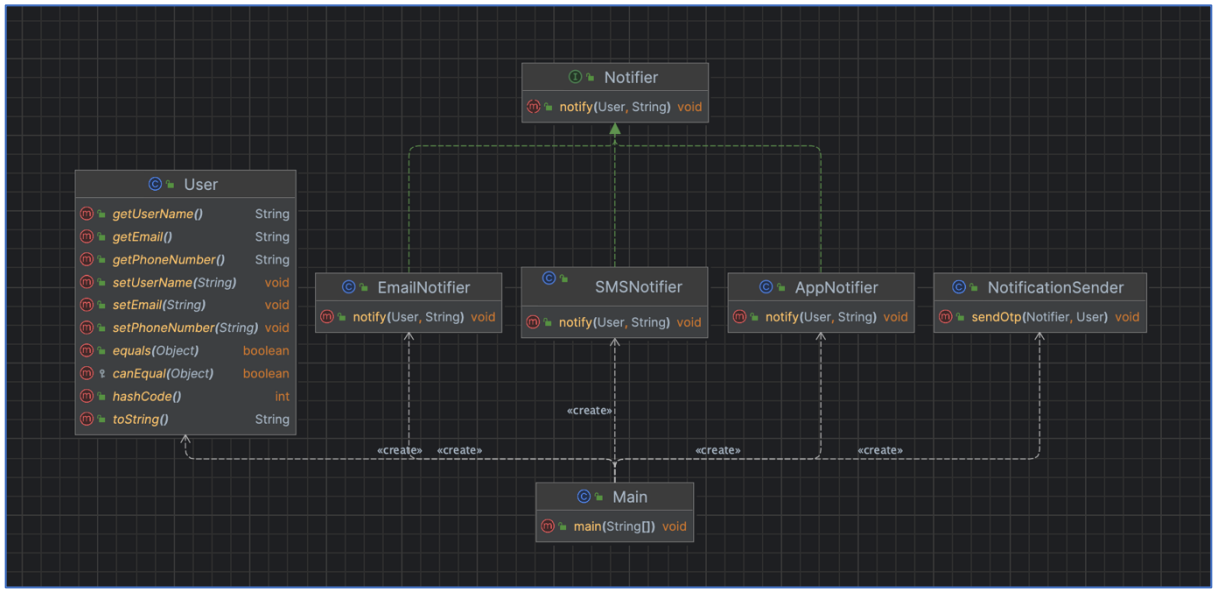

Figure 5 UML Diagram of Good Code of Open/Close Principle

```
2 usages   MANN MISTRY
public class NotificationSender {
    3 usages   MANN MISTRY
    public void sendOtp(Notifier notifier,User user){
        // generates otp
        Random random=new Random();
        String otp = String.format("%04d", random.nextInt( bound: 10000));
        notifier.notify(user,otp);
    }
}
```

Figure 6 NotificationSender classes new code

```
public static void main(String[] args) {
    User user=new User( userName: "mananmistry10", email: "mananmistry10@gmail.com", phoneNumber: "7828825538");
    AppNotifier appNotifier=new AppNotifier();
    EmailNotifier emailNotifier=new EmailNotifier();
    SMSNotifier smsNotifier=new SMSNotifier();

    NotificationSender notificationSender=new NotificationSender();
    notificationSender.sendOtp(appNotifier,user);
    notificationSender.sendOtp(emailNotifier,user);
    notificationSender.sendOtp(smsNotifier,user);

}
```

Figure 7 Main Method showing usage of multiple notifier which is being utilised in NotificationSender class

Here,Notifier is interface which will have a notify method, now whichever, new medium is added for notification will require a concreate class with implementation of notify method by implementing Notifier class, so that there won't be any changes required in NotificationSender class it just needs an implementation of Notifier interface.

That mean now onwards, NotificationSender is closed for modification and open for extension with the help of Notifier interface.

Liskov substitution principle: this principle states that the objects of a superclass should be replaceable with objects of its subclasses without breaking the application.

Bad code example:

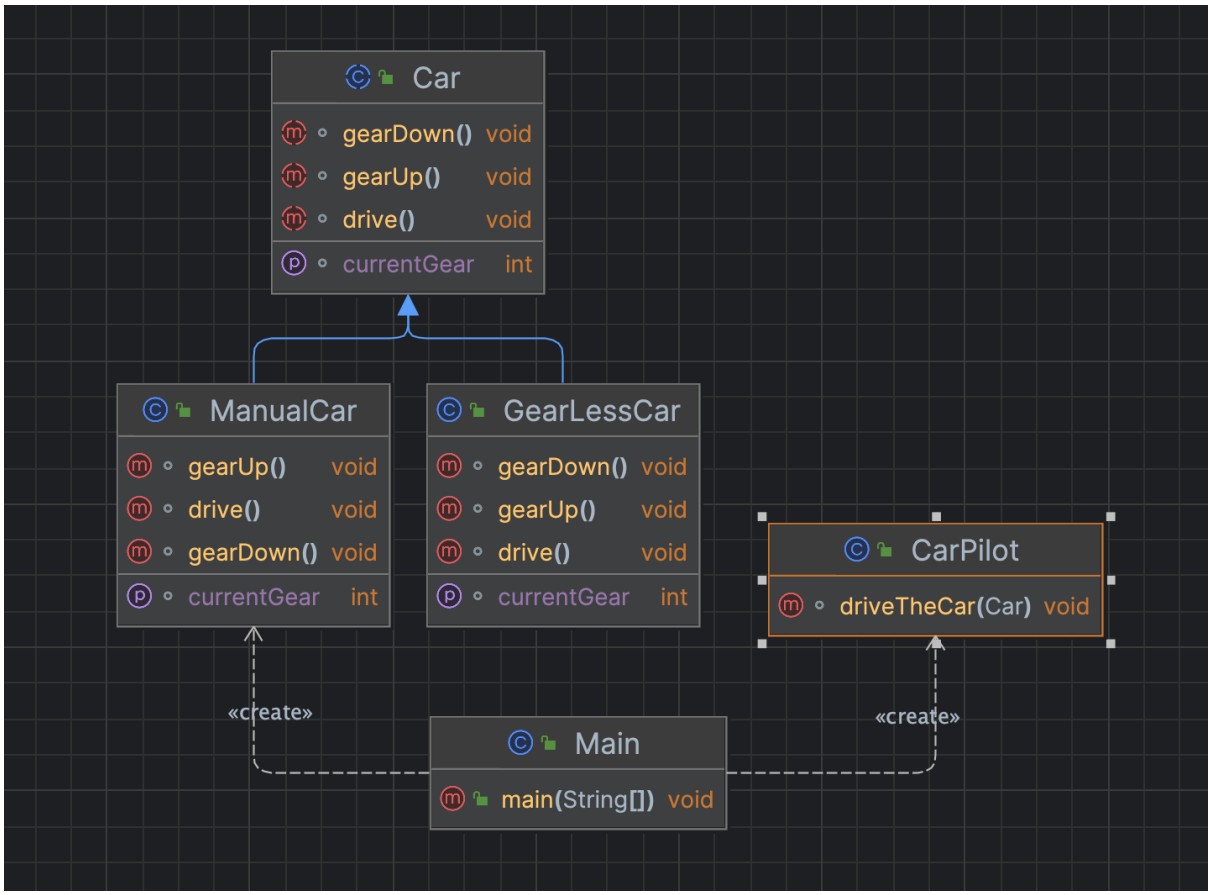

Figure 8 UML Diagram of Bad code example of Liskov substitution principle

Here , I have took the example of creating a CarPilot that will drive The Car whichever is given to it as a parameter of driveTheCar method. Car is a base abstract class which is being inherited by ManualCar and GrearLessCar, in CarPilot a method DriveTheCar takes the any kind of concreate implementation and starts driving it.

Figure 9 code of CarPilot class

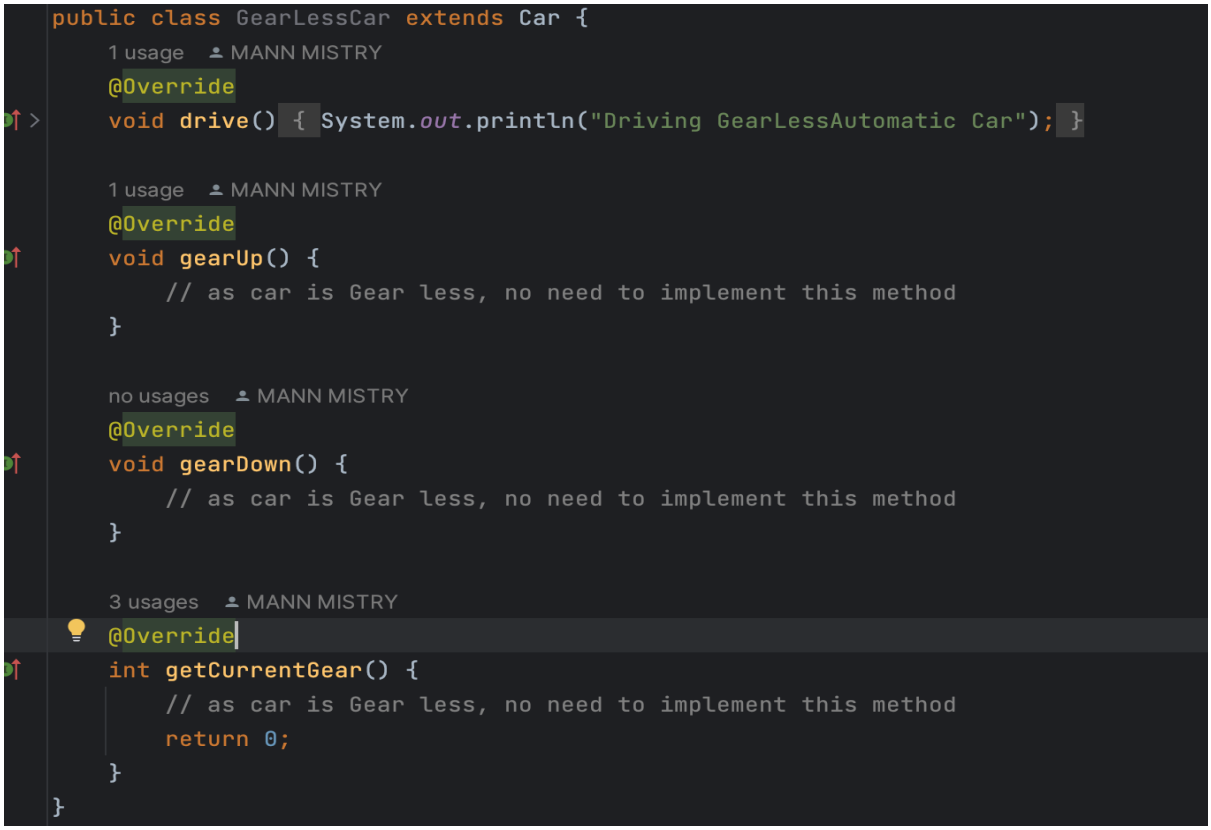

Figure 10 code of GearLessCar class

The car pilot uses the getCurrentGrear method to check the gear and increments to drive the car. But here the problem occurs when there are some methods,which are not having implementation as automatic car don't require managing the gears, due to which dummy method of base class needs to be implemented.

Good code example:

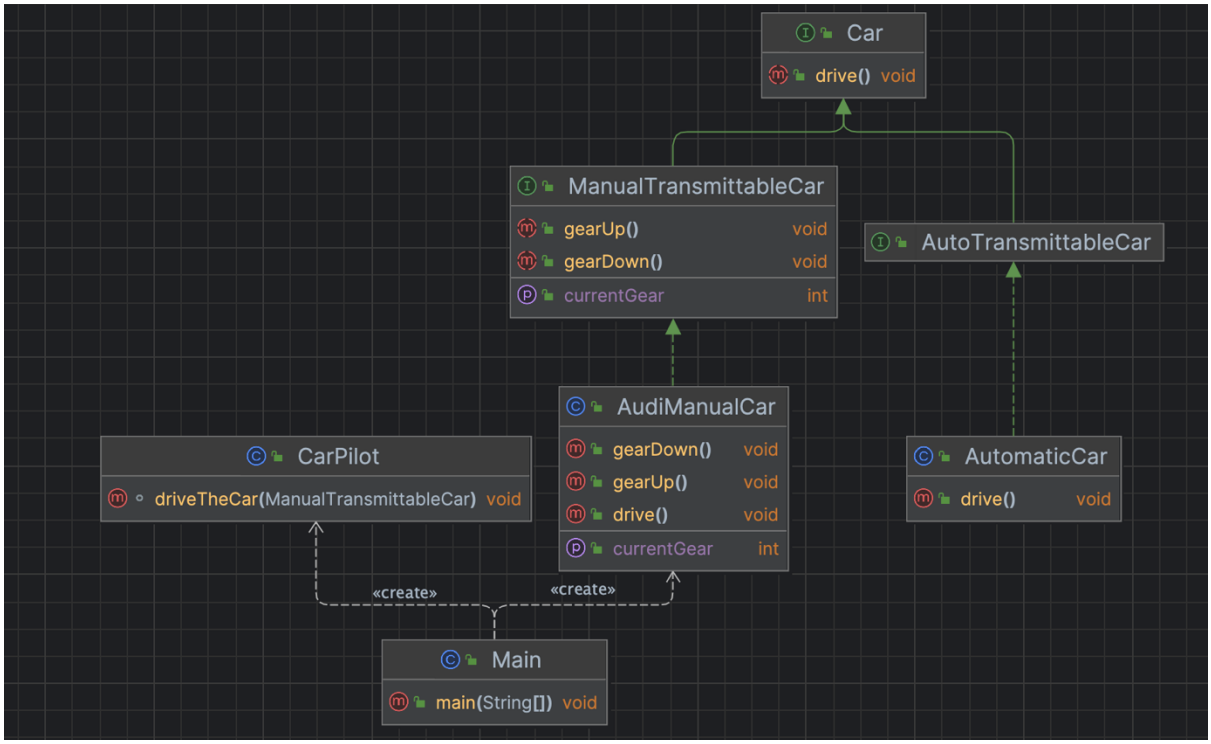

Figure 11 UML Diagram of good code example of liskov substitution

Now to resolve the issue, the methods which are related to manual transmitter car can be added to a newer interface as ManualTransmittableCar and CarPilot will utilize the object of ManualTransmittableCar type. Another interface has also been created as AutoTransmittableCar which extends the drive method from Car interface and it is used as marker for whichever AutomaticCar is created.

Interface segregation:

The interface segregation principle (ISP) states that no code should be forced to depend on methods it does not use[2].

Bad code example:

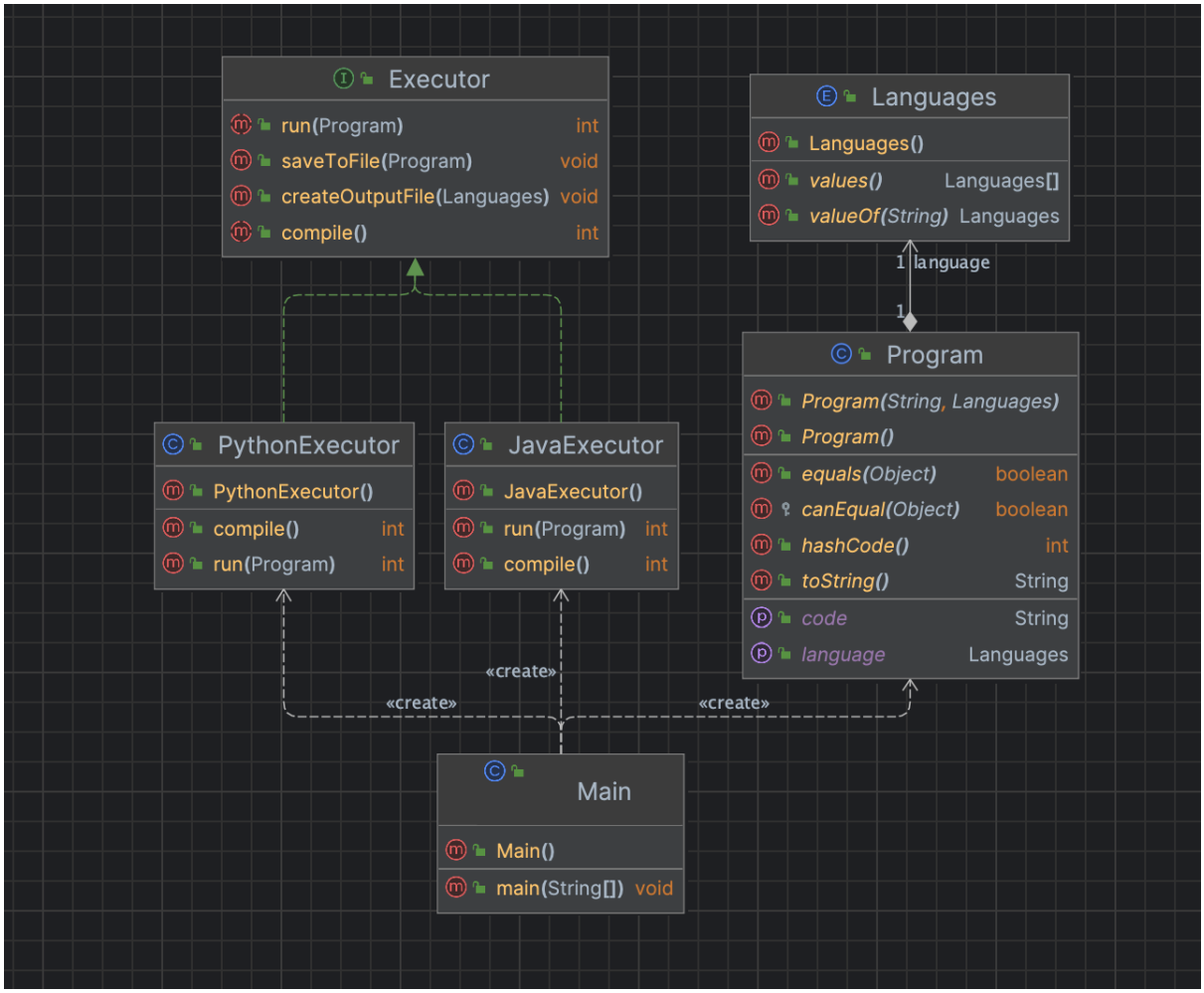

Figure 12 UML Diagram of Bad code example of Interface segregation principle

Here I have provided the example of creating a code executor, the base class is Executor which provides the basic methods to work with the code, and it is being extended by concreate implementation of Python and Java code Executor classes.

Here one thing where it violates the Interface segregation principle is that Python is interpreted language which doesn't required to be compiled.

Good code example:

The problem can be solved by introducing two other interfaces such as CompiledLanguageExecutor and Interpretable executor. Now JavaExecutor will implement CompiledLanguageExecutor whereas, Python will implement Interpretable interface which will be used as a marker interface.

Due to segregating this two interfaces the problem of forcing to provide implementation of compile method to Pythons executor have been resolved.

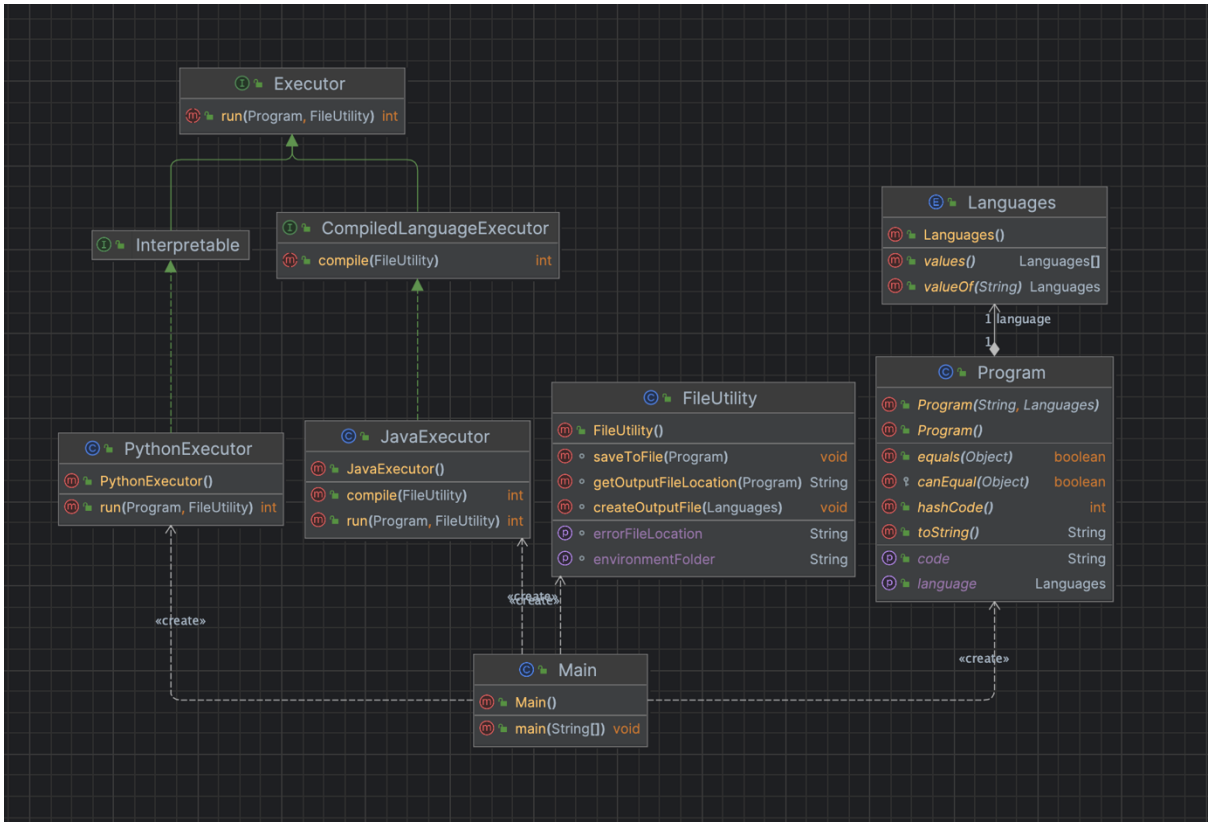

Figure 13 UML diagram of good code example of Interface Segregation

Dependency Inversion:

The principle states two points:

A. High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g., interfaces) [3].

B. Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions [3].

Bad code example:

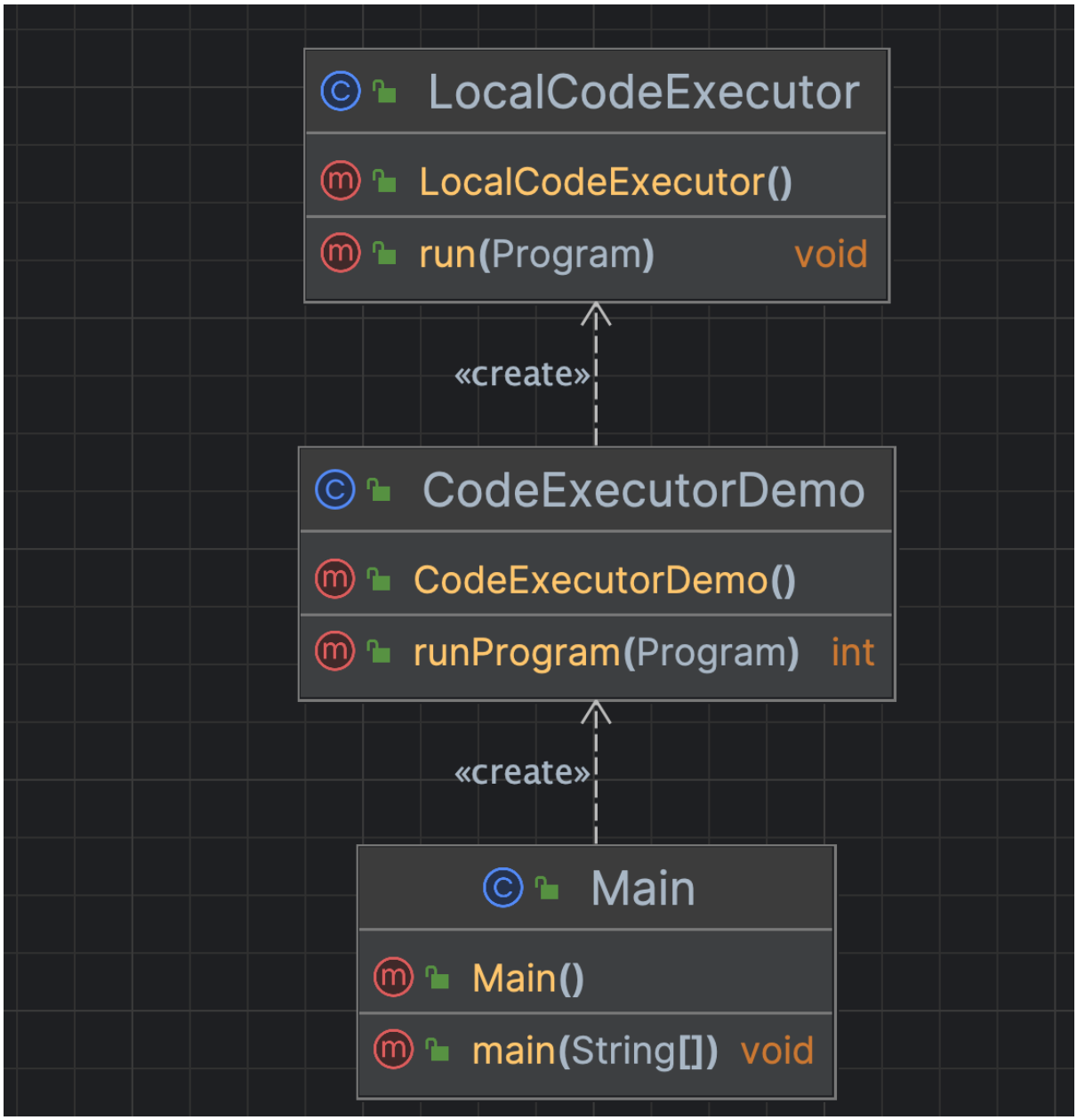

Figure 14 UML Diagram of bad code example of Dependeny Inversion Principle

Here from UML diagram, the LocalCodeExecutor is a concreate class which executes the code locally on the same system, and CodeExecutorDemo class directly creates the object of

LocalCodeExecutor class to run the program. Now here the problem occurred, as High level module CodeExecutorDemo is directly depended upon the concrete implementation of LocalCodeExecutor class, not only that but here concrete implementation is also not being depended on abstraction, as it directly utilized the LocalCodeExecutor.

Good code example:

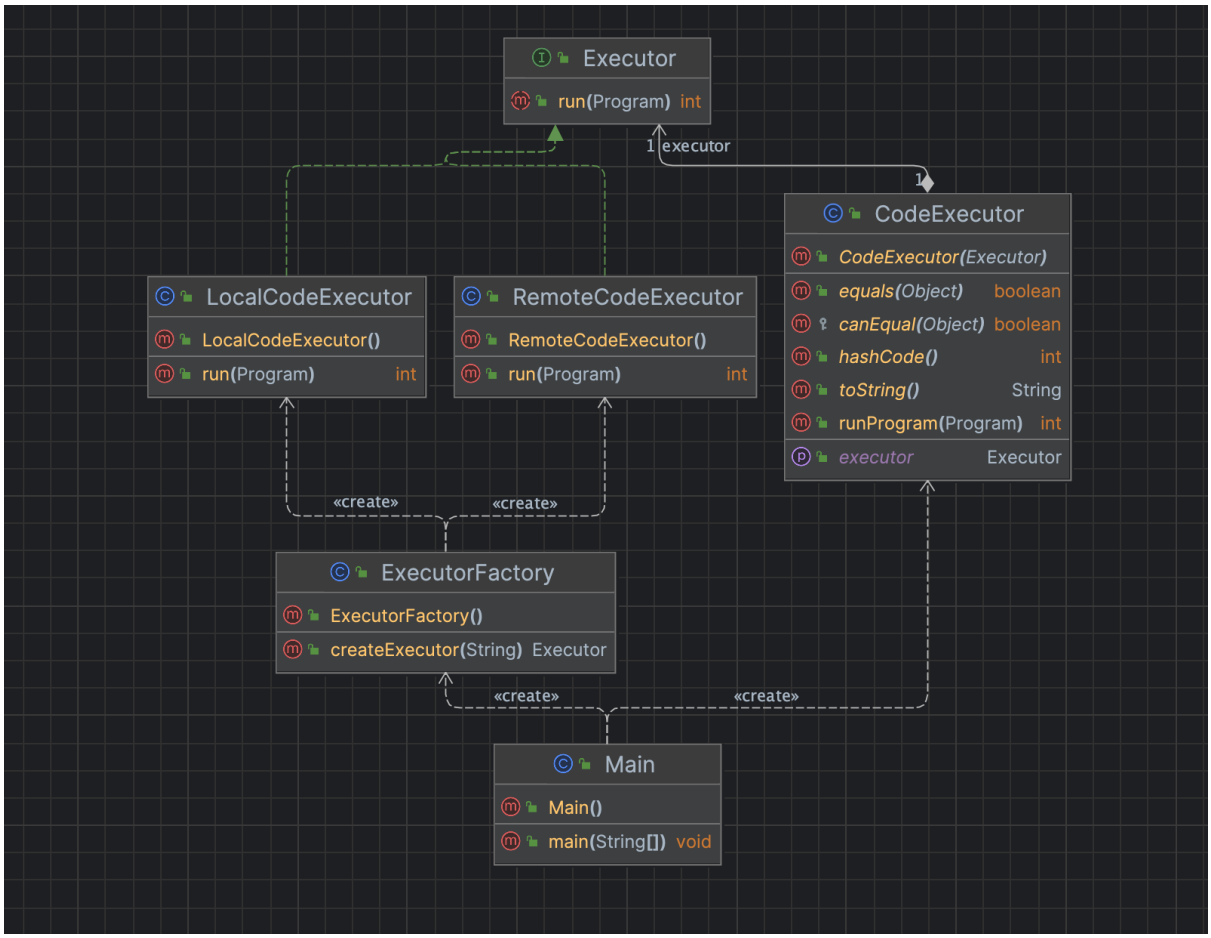

Figure 15 UML diagram of Good Code example of Dependency Inversion Principle

Figure 16 Code of Main method showing Dependency Injection example

Here, the executor interface has been introduced and it has two concrete implementation which is RemoteCodeExecutor and LocalCodeExecutor, among those will be created using the Executor Factory and will be injected to CodeExecutor using constructor based Dependency injection concept. It will help them to execute the code on which ever platform they want to use, whether it can be local system or it can be RemoteExecutor which can be Isolated from other devices to execute any code which can be malicious.

References:

[1] Wikipedia contributors. (2020, April 17). Single-responsibility principle. In *Wikipedia, The Free Encyclopedia*. Retrieved 01:48, July 09, 2023, from https://en.wikipedia.org/w/index.php?title=Single-responsibility_principle&oldid=951463512

[2] Martin, Robert (2002). Agile Software Development: Principles, Patterns, and Practices. Pearson Education.

[3] *Martin, Robert C. (2003). Agile Software Development, Principles, Patterns, and Practices. Prentice Hall. pp. 127–131. ISBN 978-0135974445.*