بِسْمِ اللهِ الرَّحْمٰنِ الرَّحِيْمِ

CSE 4851: Design Patterns
Assignment on SOLID principles

Submitted by:
Ahmad Al Fayad Chowdhury
170041041
CSE

SOLID Principles

SOLID is an acronym coined by Michael Feathers for a subset of programming principles promoted by Robert C. Martin, widely known as Uncle Bob. They are centred on Object Oriented Programming that make the code more effective, readable, maintainable and flexible. They are especially useful for larger projects where the codebase can grow incredibly large and often difficult to manage without adherence to these principles.

SOLID stands for:

1. **S**ingle Responsibility Principle (SRP)
2. **O**pen Closed Principle (OCP)
3. **L**iskov Substitution Principle (LSP)
4. **I**nterface Segregation Principle (ISP)
5. **D**ependency Inversion Principle (DIP)

The following portions explore each of the above principles in brief.

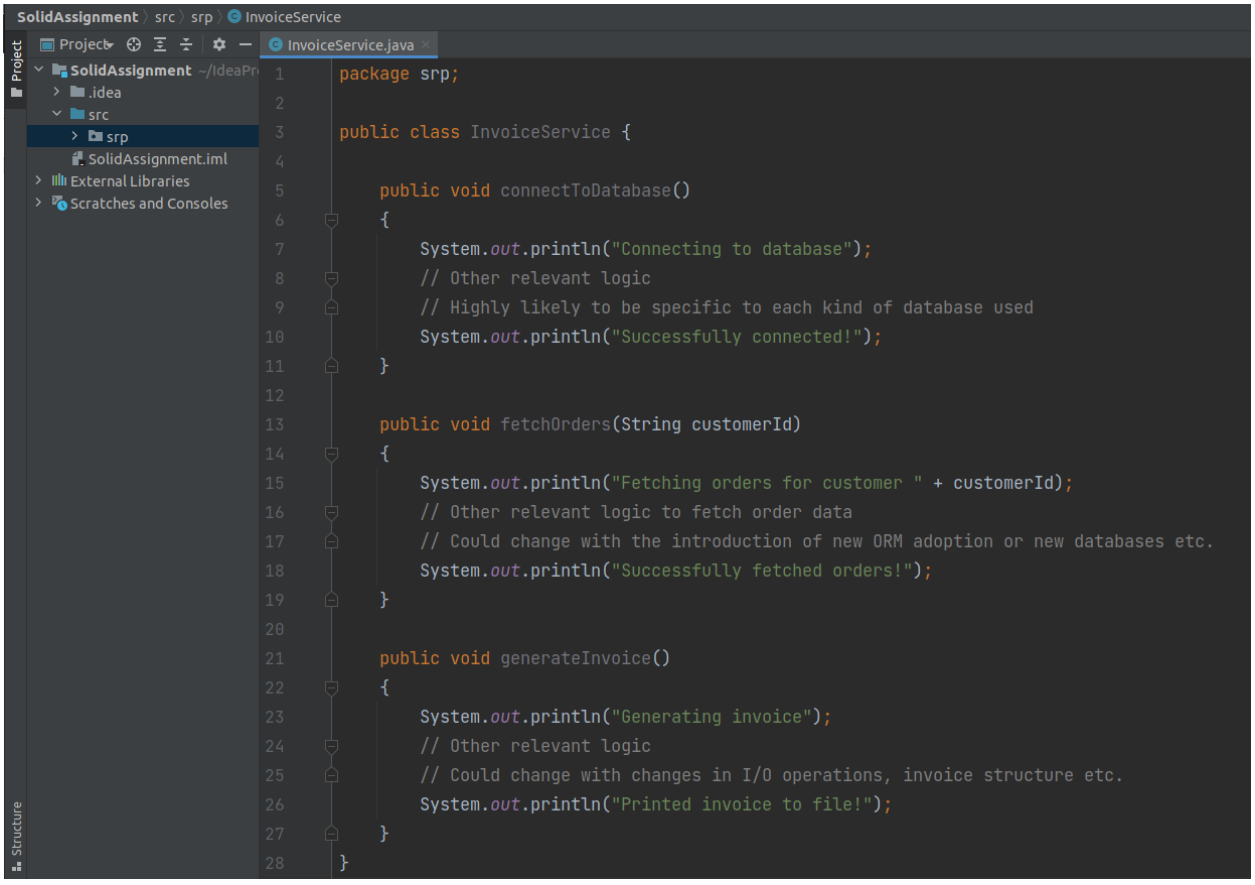
Single Responsibility Principle (SRP)

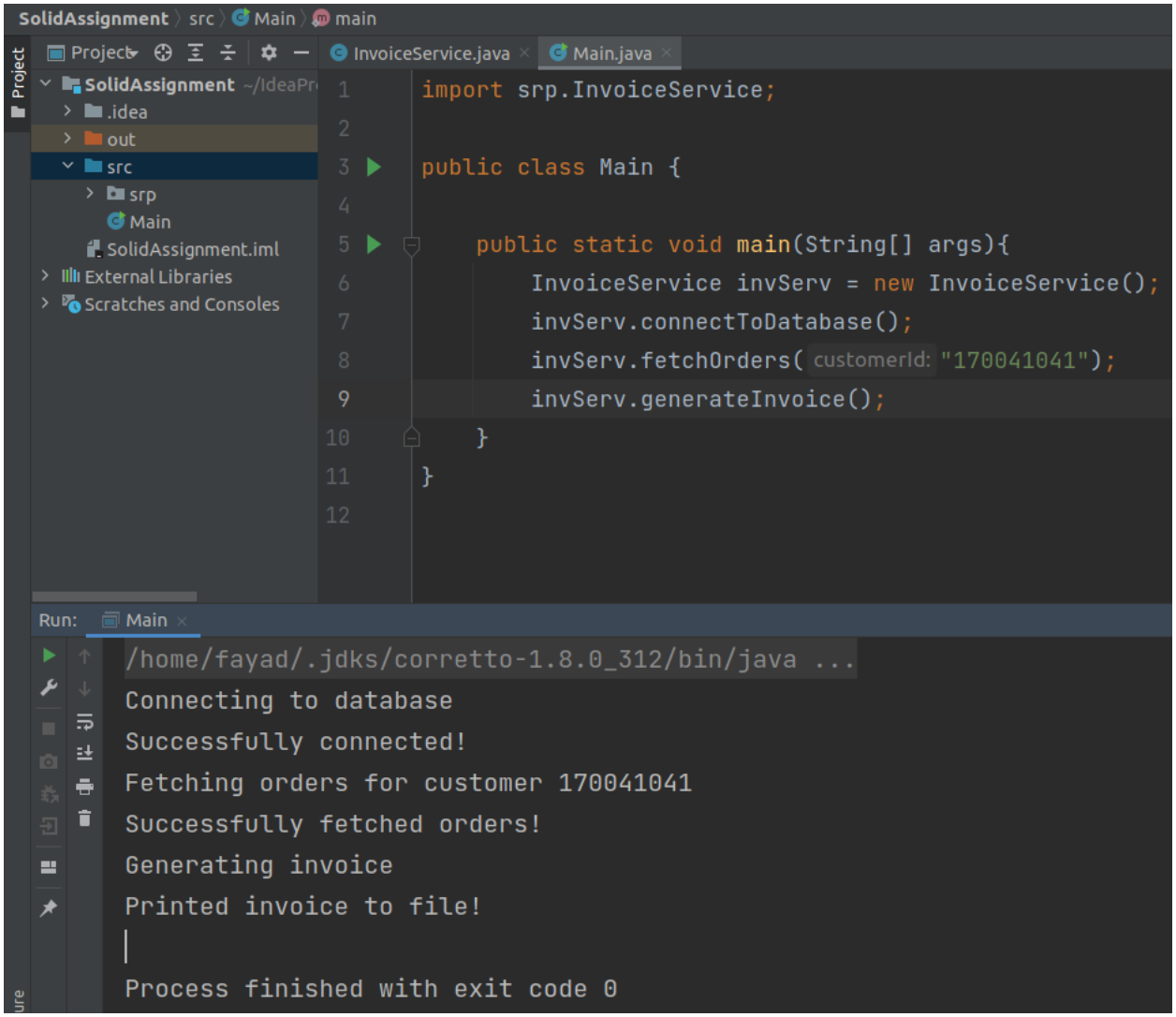
SRP states that each class in a program should have only one responsibility and, hence, only one reason to change. This leads us to the advantage of the classes being **decoupled**, **organized** and easier to test as units.

Considering the case of a module of software that generates invoices. Assume a single class is created and tasked with the responsibilities of:

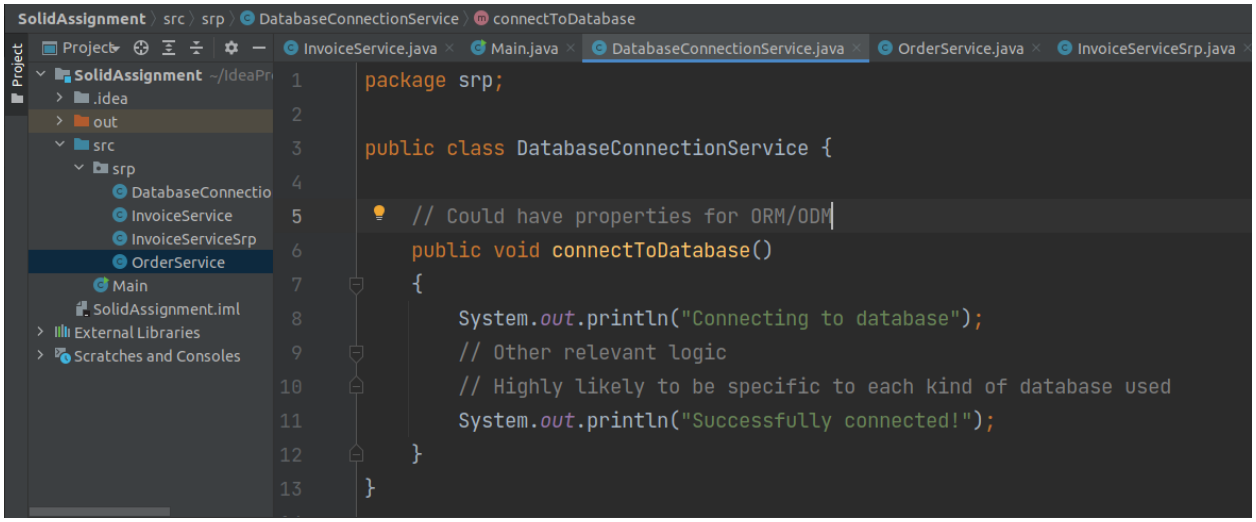
1. Connecting to the database
2. Fetching a customer's order data
3. Generating and printing invoices to a file or some other form of output

In such a scenario, **one** class handles **three** responsibilities and therefore there are at least 3 factors influencing the change of this class. This is a direct violation of SRP and makes this class highly subject to change and makes it very cluttered as well.

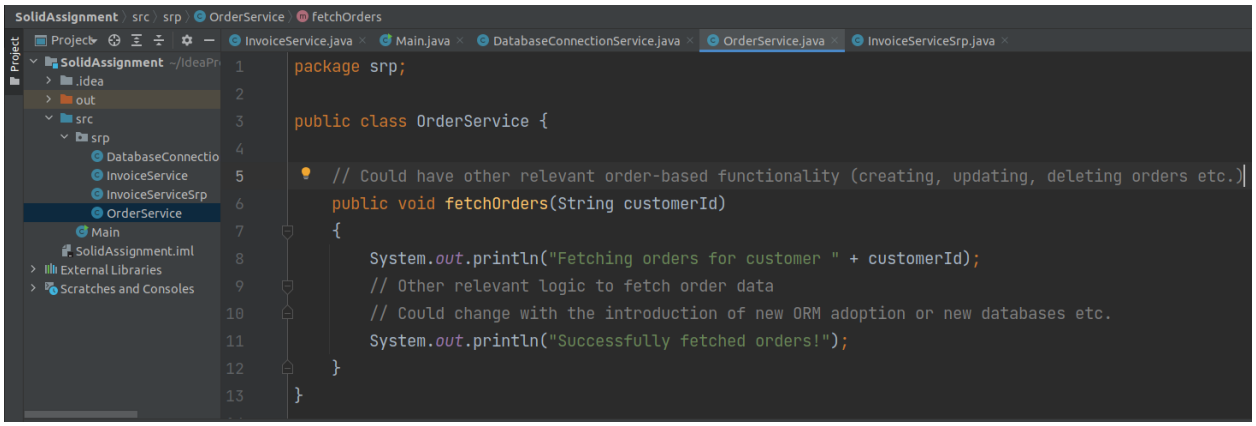




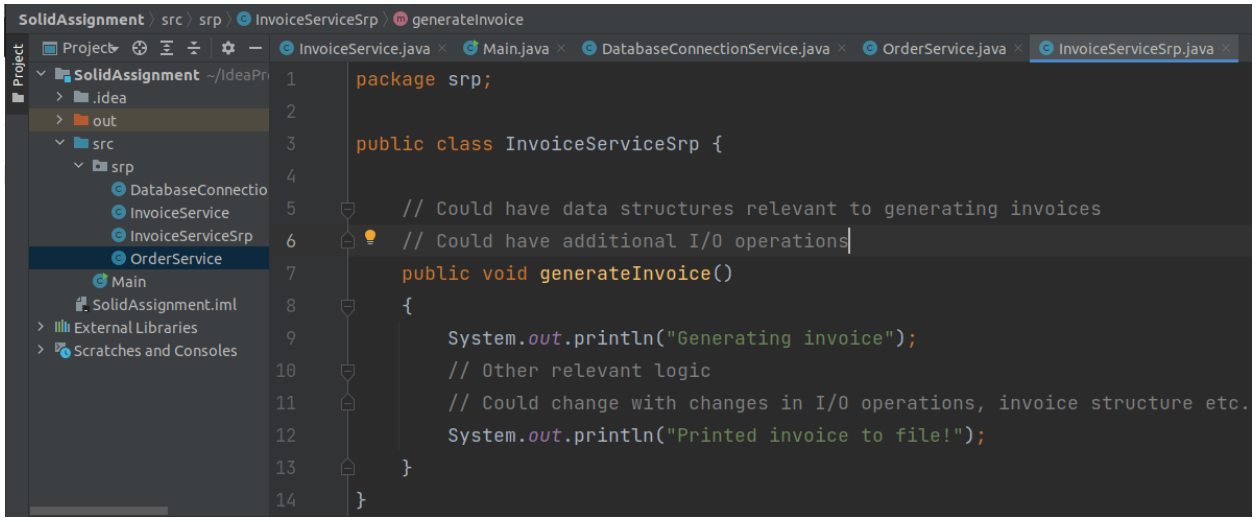
If instead we separate the responsibilities into three separate classes, we achieve adherence to SRP and make the module more **flexible** and **easily maintainable.**



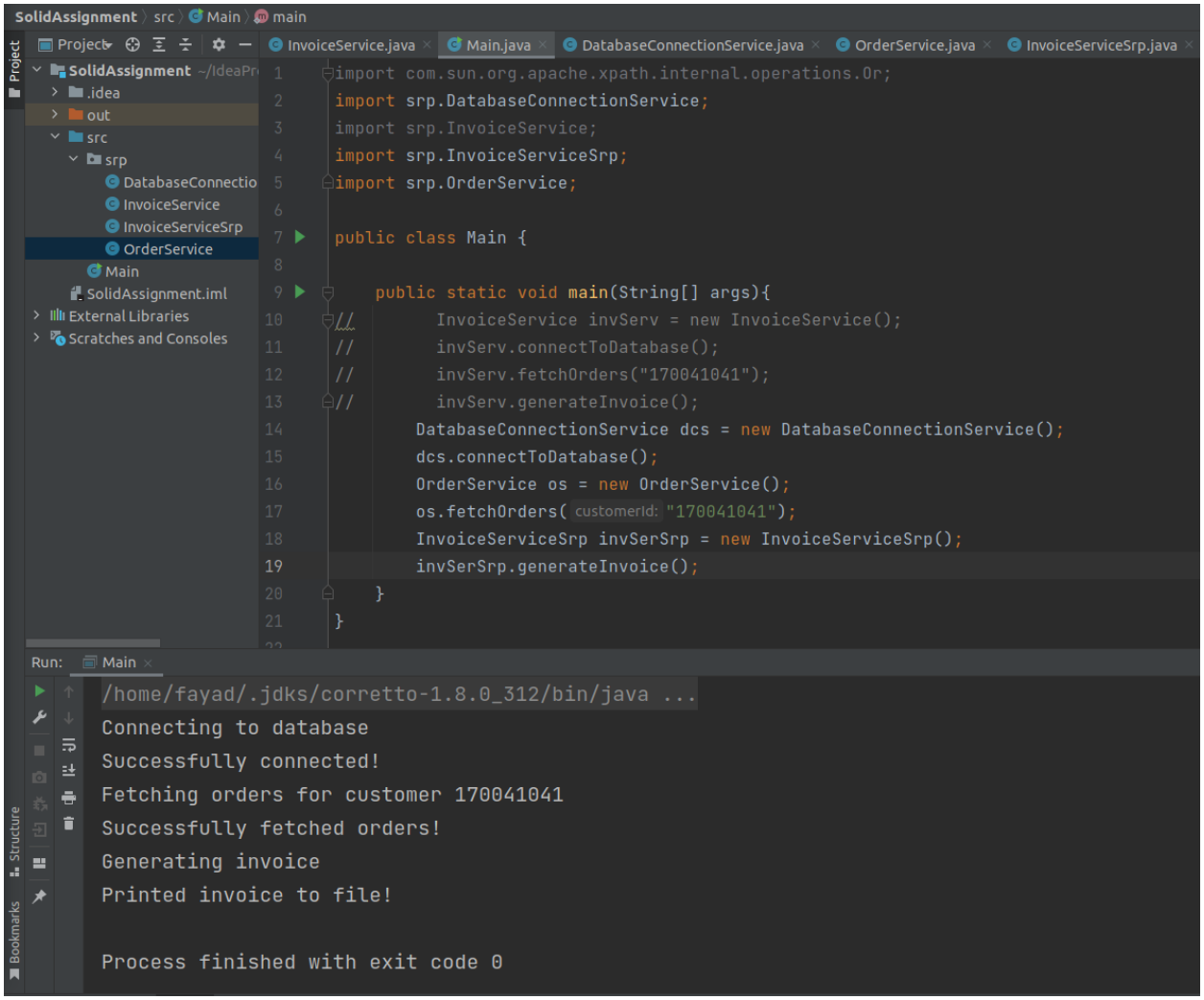
The DatabaseConnectionService is tasked with handling connections to the database.



The OrderService class is tasked with the responsibility of order functionality.



The InvoiceServiceSrp class is tasked with the generation of invoices as an output and other relevant functions.



Creating objects of the separate classes and running the functions accordingly results in the same output but, now that SRP has been achieved, it becomes easier to handle the submodules.

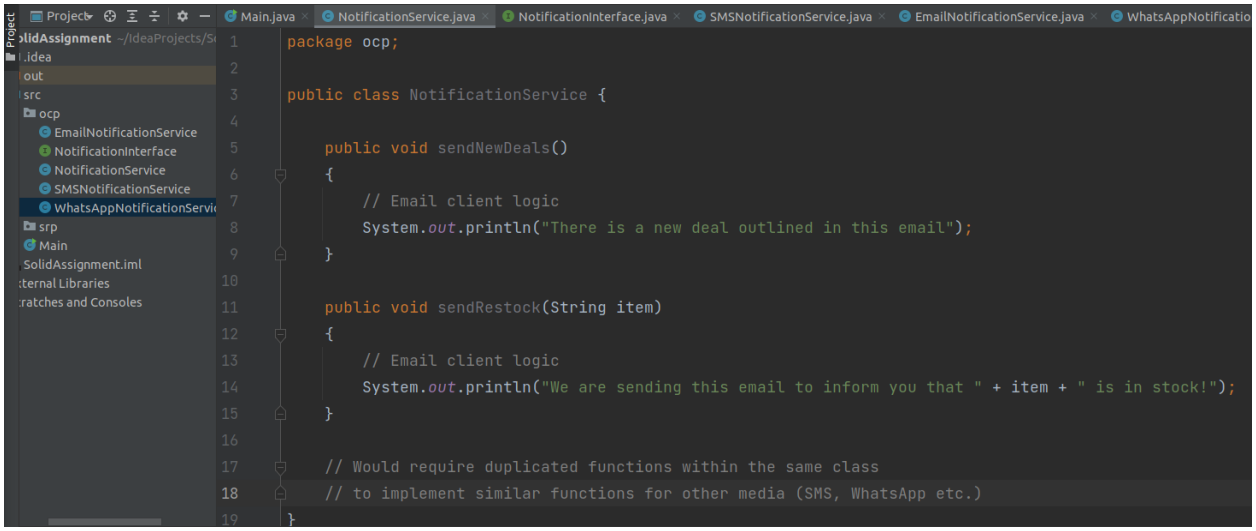
Open Closed Principle (OCP)

OCP states that classes should be **open for extension, but closed for modification.** This prevents us from modifying the existing code to add new functionality. Instead, we are encouraged to inherit or extend the existing class and add new functionality to the inheriting or extending class.

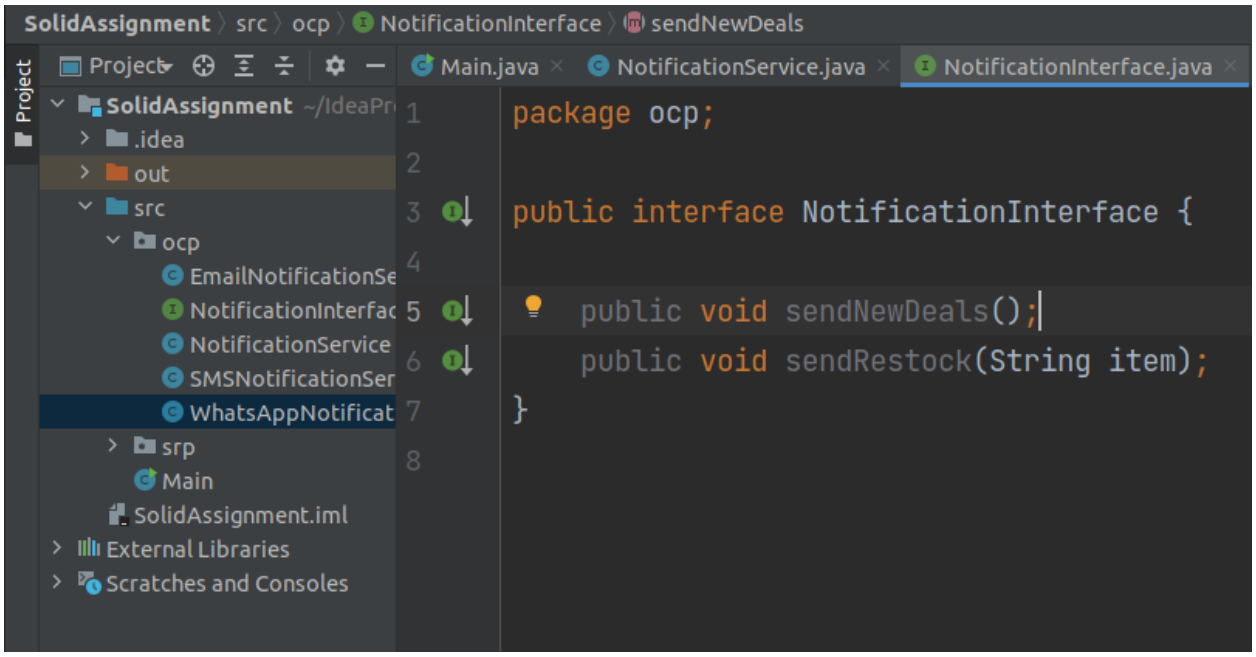
This is especially useful when there are **multiple developers working on a project**. If developer A has written some code for a particular function that developer B needs to change for his/her convenience, there may be i**ssues in dependencies** along the line. If instead there were a base class that both developers A and B could inherit from and add functionality, the base class and the children classes would be **easier to maintain.**

Consider the case of a notification submodule that needs to send customers notifications on new deals and whether a customer's desired items are in stock. Let us assume that the business initially relied on emails, but are now wishing to adopt SMS notifications and predict a shift towards WhatsApp notifications as well.

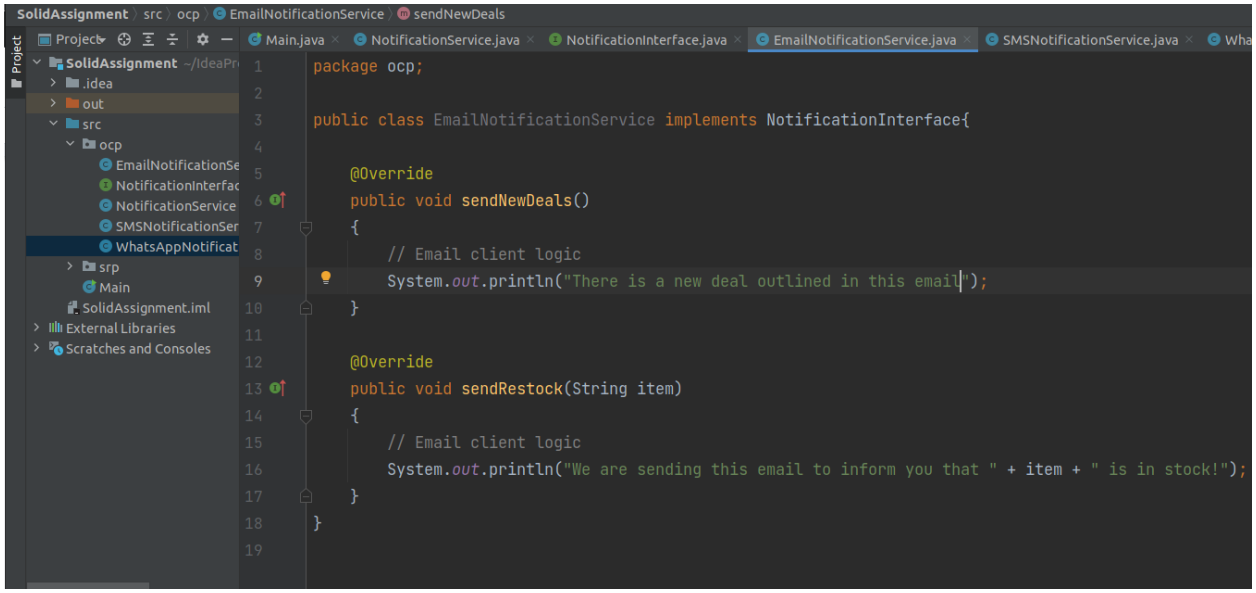
In the case that the initial NotificationService class was written keeping only email notification related functionalities in mind, it would require **duplicated functions** to implement the same functionalities for other media. This would lead to **cluttered code,** as shown below:



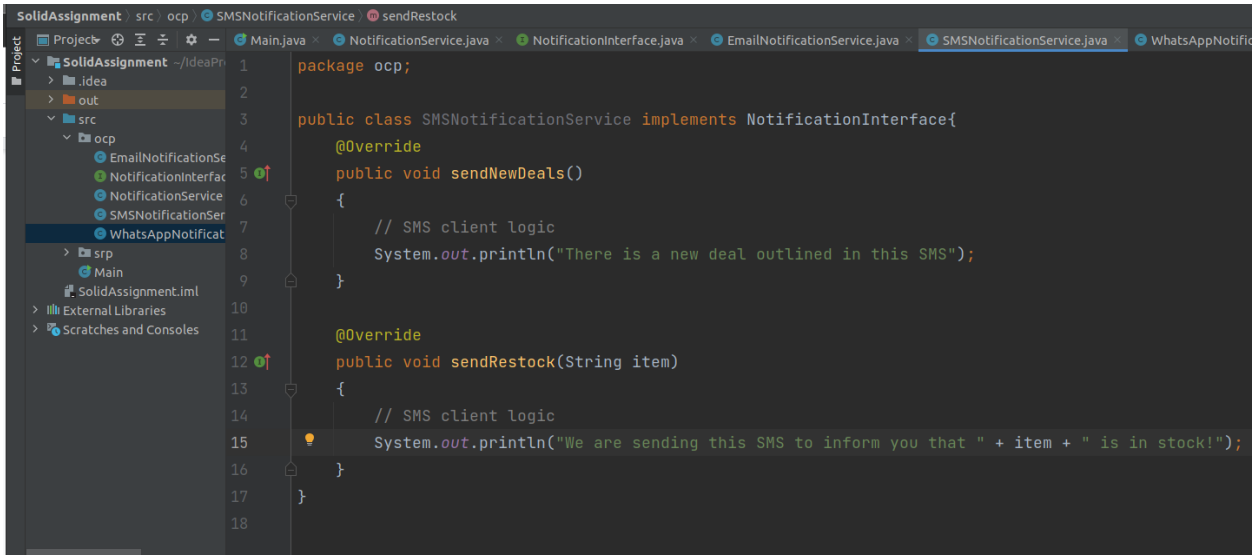
A neater way to write this module is with the help of a NotificationInterface interface that will have only the prototypes of the functions and classes implementing this interface will need to override these functions according to their own logic, as shown below:



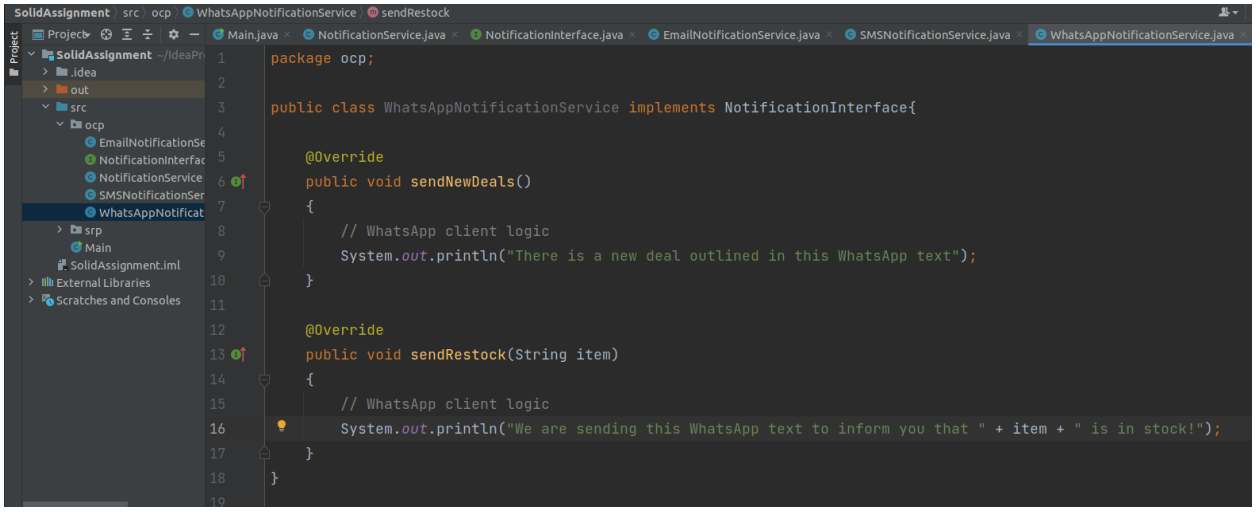
The NotificationInterface interface that will be implemented by the SMSNotificationService, EmailNotificationService and WhatsAppNotificationService classes as per their own logic.



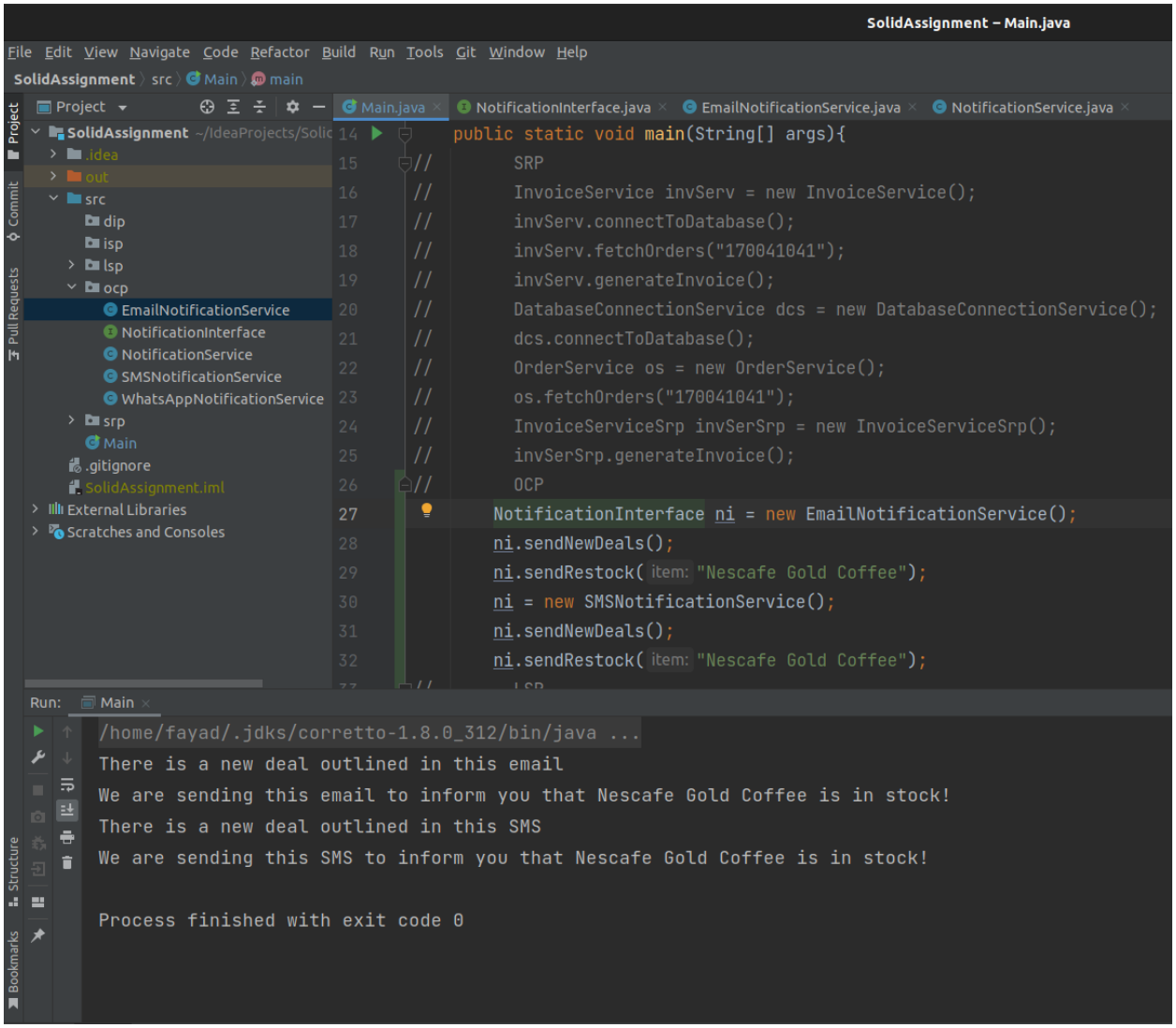
The EmailNotificationService class implements and overrides the functions in the NotificationInterface interface.



Similar implementation of the SMSNotificationService class.



Similar implementation of the WhatsAppNotificationService class.



Finally in the main call area, the function calls are shown.

This adherence to OCP makes it **easier to extend functionality** while keeping the **base code or logic unchanged.**

Liskov Substitution Principle (LSP)

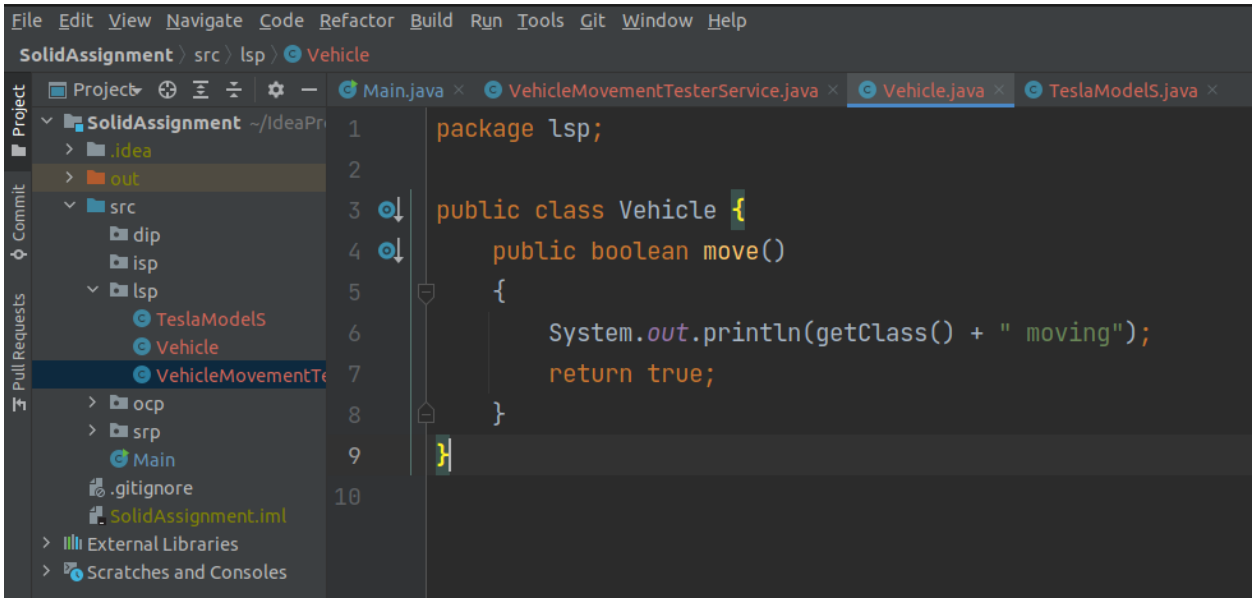
LSP states that **if a class A is a subtype of class B (inherits from class B), then it should be possible to replace instances of B with those of A without disrupting the behaviour of the program.** This implies that **derived classes must be completely substitutable for their base classes.**

LSP essentially implies that whatever the base or parent class does, the derived or children classes do as well; the children classes may extend the functionality too, but they are never meant to act very differently from the parent class.

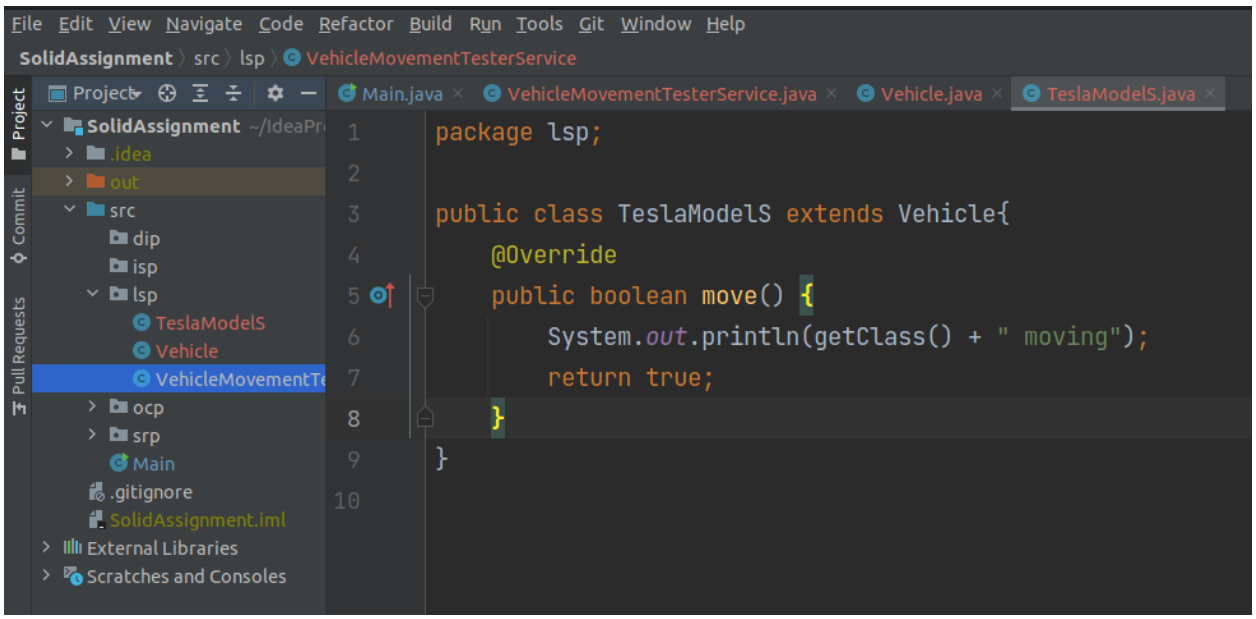
This principle is one of the **hardest to implement** since it requires foresight of how the end program will shape up. There has to be great care towards **ensuring that the behaviour of parent and children classes are consistent**.

Consider a parent class Vehicle that is a skeleton for all sorts of motor and electric vehicles. Further consider a derived or child class, TeslaModelS, which is a skeleton for all Tesla Model S vehicles. Both classes will have some function move() that will detail how a generic vehicle moves (represented by the Vehicle class) and how a Tesla Model S moves (represented by the TeslaModelS class). There will be a vehicular movement tester submodule, represented by the VehicleMovementTesterService class. Ideally the service should be designed in such a way that not only does it work for generic vehicles but also for very specific makes like the Tesla.

Designing the very basic code keeping LSP in mind, we produce something like the Vehicle and TeslaModelS classes shown below.



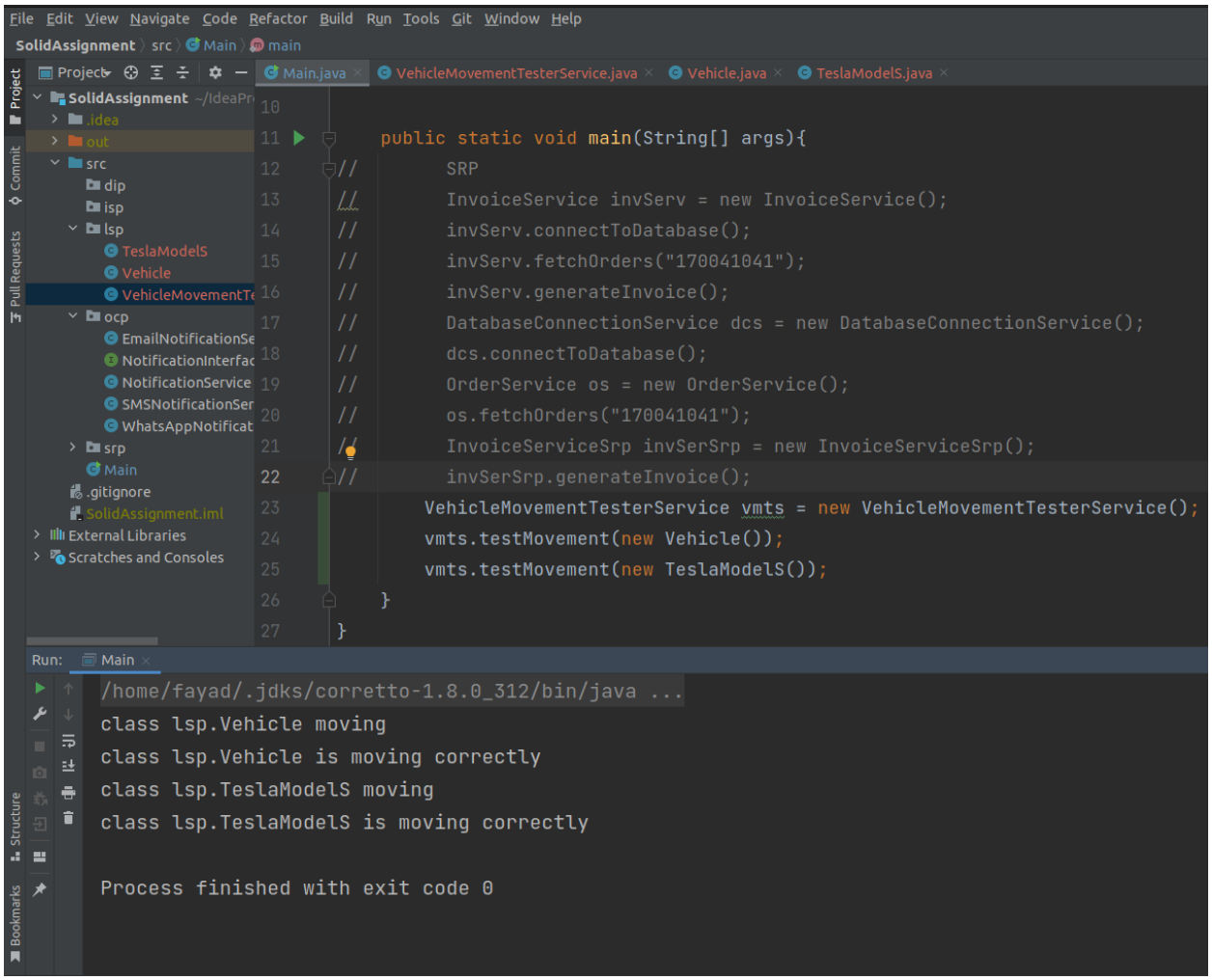
The base Vehicle class to represent generic vehicles.



The TeslaModelS class that extends the Vehicle class.

To fully implement LSP, the portions of the code or the subsystems that incorporate these affected classes need also to be designed accordingly. In this case, the VehicleMovementTesterService class has a testMovement() function that needs to accept the generic parent Vehicle class as shown below.

The testMovement() function checks to see whether the Vehicle v's move() function returns a true Boolean value or not.



In the main call area, it is observed that, because of the considerations made, this testMovement() function within the VehicleMovementTesterService class not only works for parent classes but also for derived or children classes.

Writing code keeping LSP in mind makes it easier to **model the behaviour of children classes** and also to **write unit tests** for them. This makes the entire program more flexible and maintainable, at least at testing time.

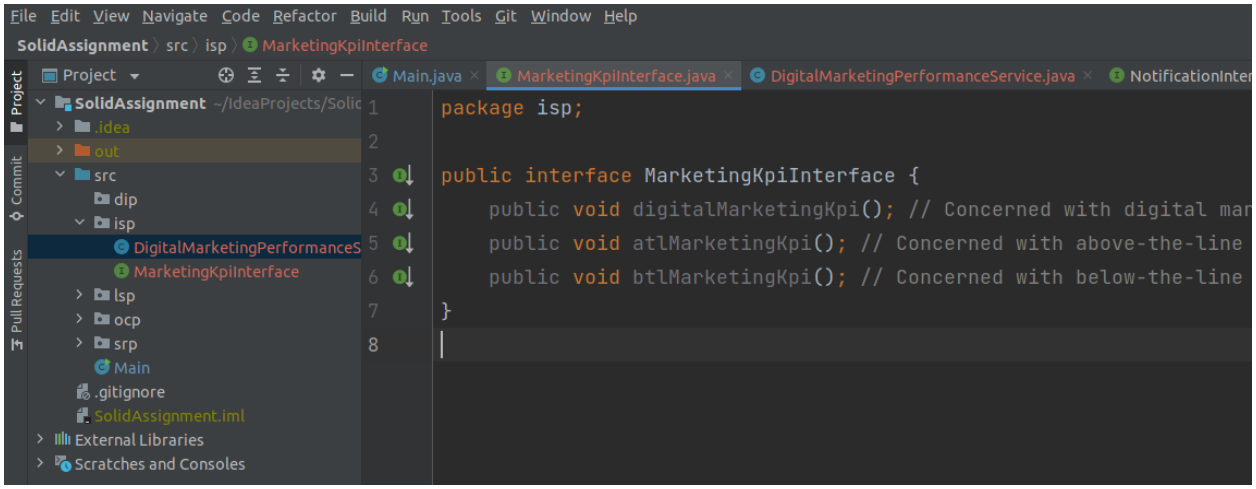
Interface Segregation Principle (ISP)

ISP states that **larger interfaces should be split into smaller interfaces such that the classes implementing these interfaces can be concerned only about the functionalities prototyped in these interfaces.**

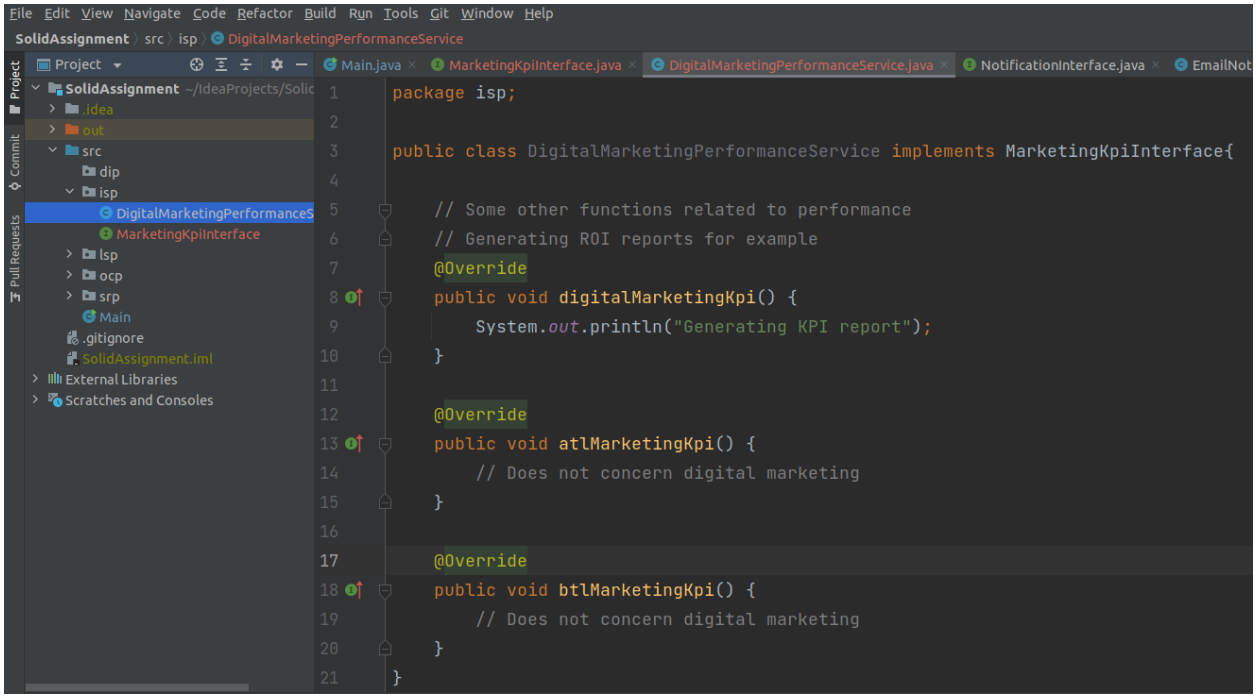
ISP basically implies that **a client should never be forced to depend on methods that they do not need**. By way of this principle, **“fat” interfaces** (those containing far too many method prototypes) are broken down into **“thinner” interfaces** (those with far fewer method prototypes that are very specific to the interface’s purpose as per business logic).

This makes the codebase more **readable** and **maintainable,** especially for large projects.

Consider the case of a subsystem that generates reports and graphics for different kinds of digital, ATL, BTL etc. marketing strategies. If we do not separate the KPI analysis functions for each of the above in separate interfaces, we are forced to implement all of the functions in each of the performance analysis classes (which themselves may have other functions like generating graphs or targets or analysing ROIs and other metrics for example). This makes the classes very clunky and populated with unnecessary code as shown below.

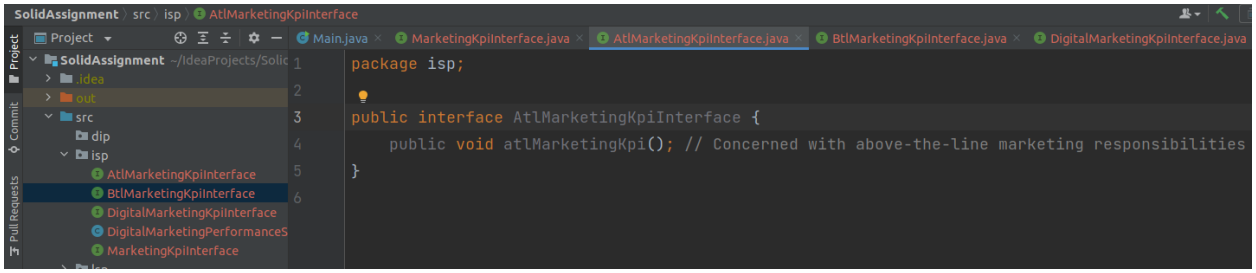


The MarketingKpiInterface that has all different KPI calculation function prototypes in itself

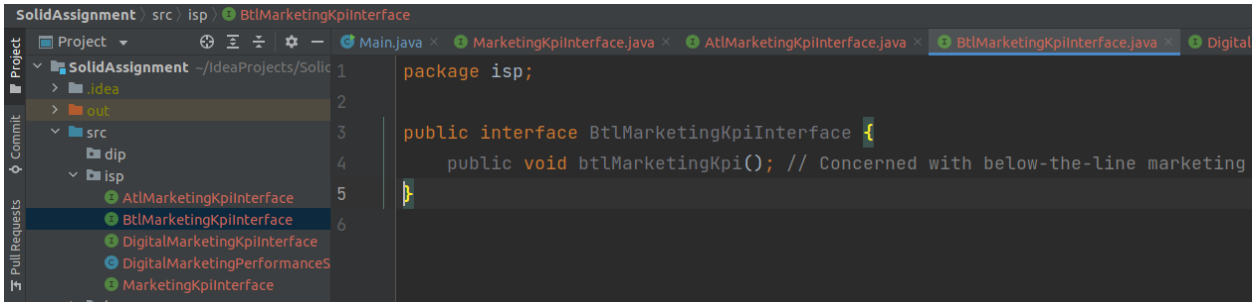


A DigitalMarketingPerformanceService class that is forced to implement the ATL and BTL KPI functions as well because of the current implementation of the “fat” MarketingKpiInterface.

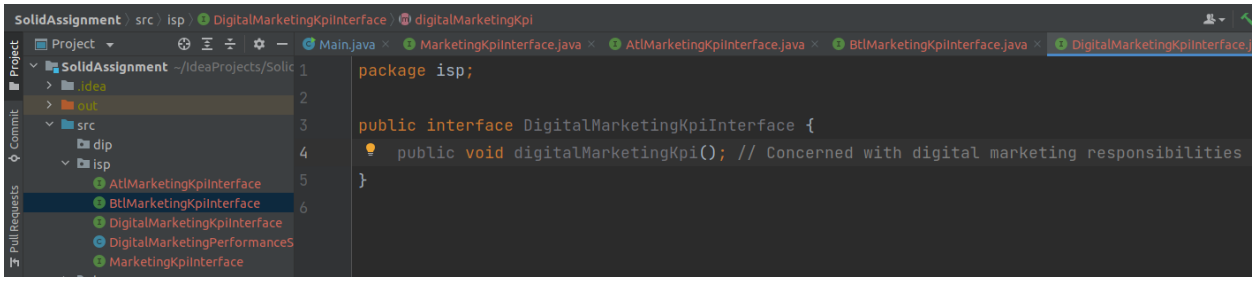
Keeping in mind ISP, we separate the KPI calculation functions into 3 separate interfaces as shown below.



The AtlMarketingKpiInterface with the prototype of the ATL marketing KPI calculation function.

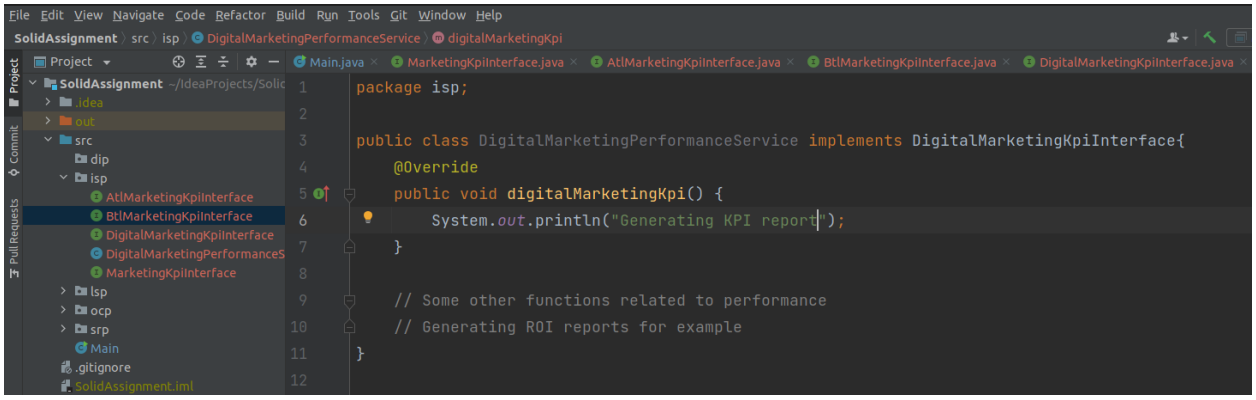


A similar implementation of BtlMarketingKpiInterface.



A similar implementation of DigitalMarketingKpiInterface.

This allows the DigitalMarketingPerformanceService to implement the DigitalMarketingKpiInterface and not have to write functions for ATL and BTL KPI calculations as shown below



The DigitalMarketingPerformanceService class is now much **leaner** and **more readable**.

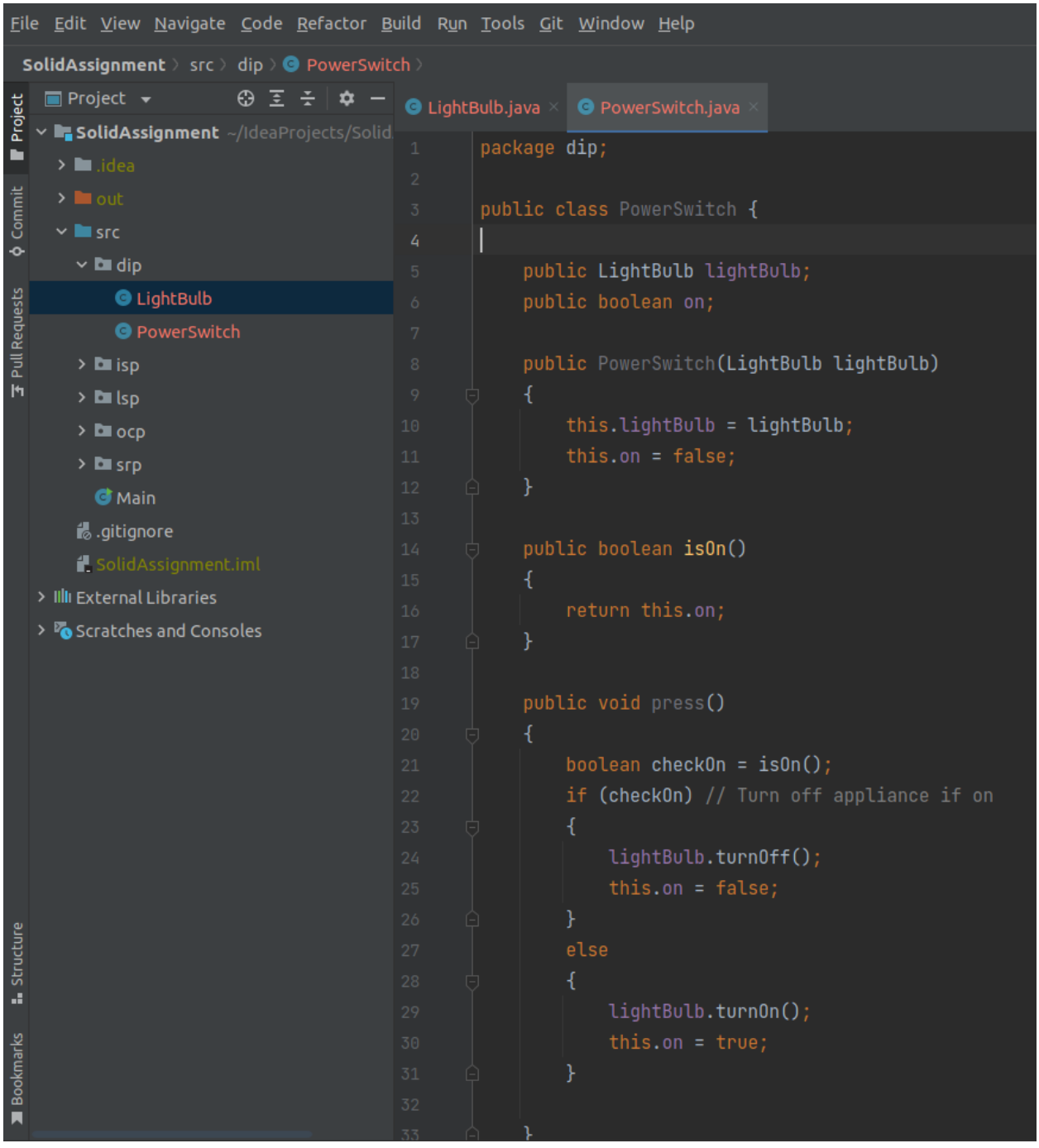
Dependency Inversion Principle (DIP)

DIP states two things:

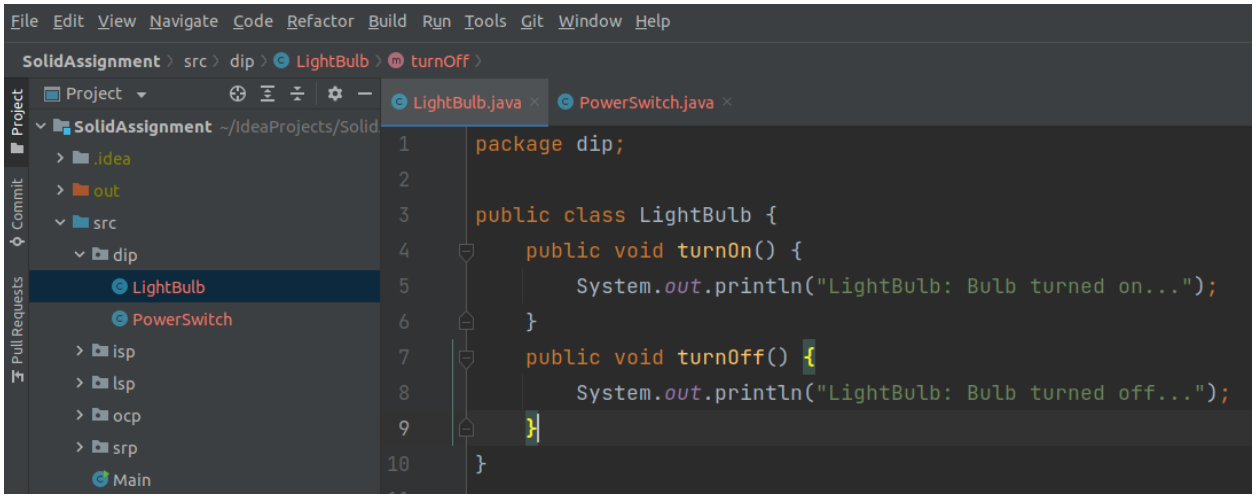
1. **High level modules will not depend on low level modules; both will depend on abstractions**
2. **Abstractions will not depend on details; details will depend on abstractions**

These two sub-principles combined allow us to model our codebases such that they are very **flexible** and **robust,** by **removing tight coupling.**

Consider the case of a power switch and an electrical appliance, a light bulb for example. A bad way to program this would be to make the power switch specifically for the light bulb and adapt the functionalities of turning the switch on and off specifically for both the light bulb and the specific make of the power switch. In the event of a new kind of switch (a remote switch for example) and/or a new kind of appliance (a fan for example), the code would have to be drastically modified or rewritten almost entirely. An example of this implementation is shown below.



The PowerSwitch class that is highly specific to the LightBulb class representing the light bulb appliance. Implementing this functionality for a fan to be used with a remote switch becomes incredibly difficult here.

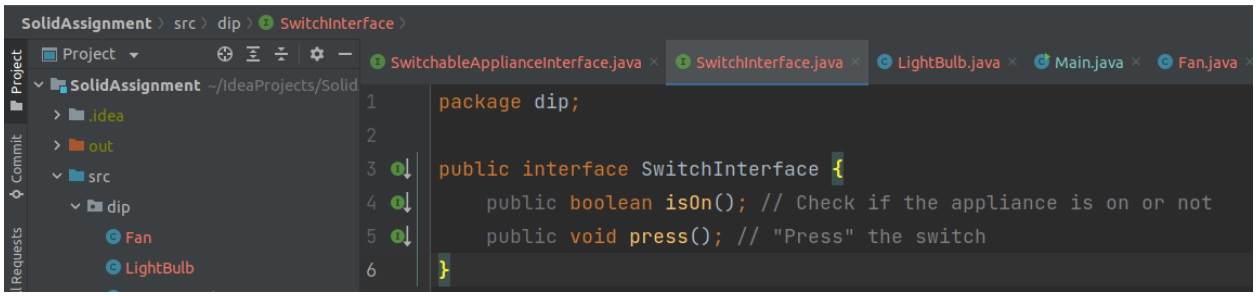


The LightBulb class implementation with very specific on/off functionalities.

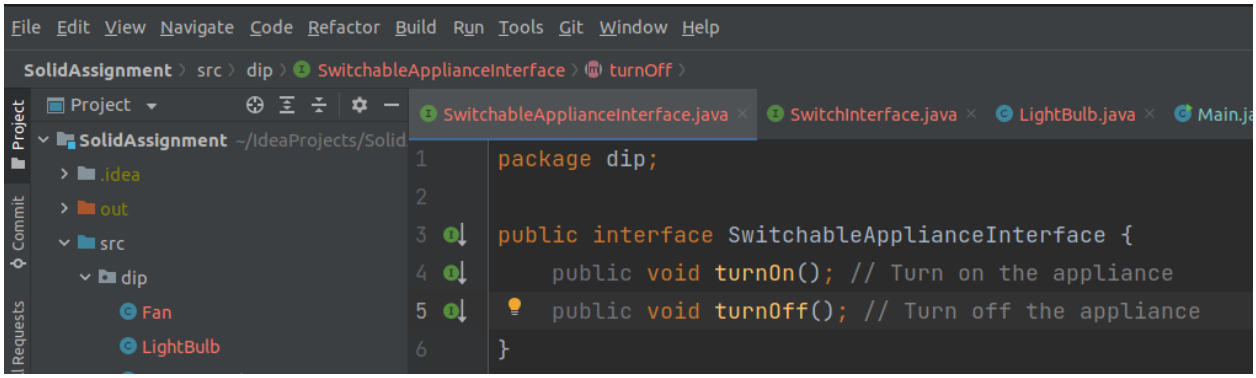
A good way to fix this and make the code more open to addition (in line with OCP) is by incorporating interfaces:

1. A SwitchableApplianceInterface that the electrical appliances will implement. This will contain the on/off function prototypes that each electrical appliance's class will override.
2. A SwitchInterface that each kind of switch will implement. This will contain the prototypes of the press function and the check function that each switch's class will override.

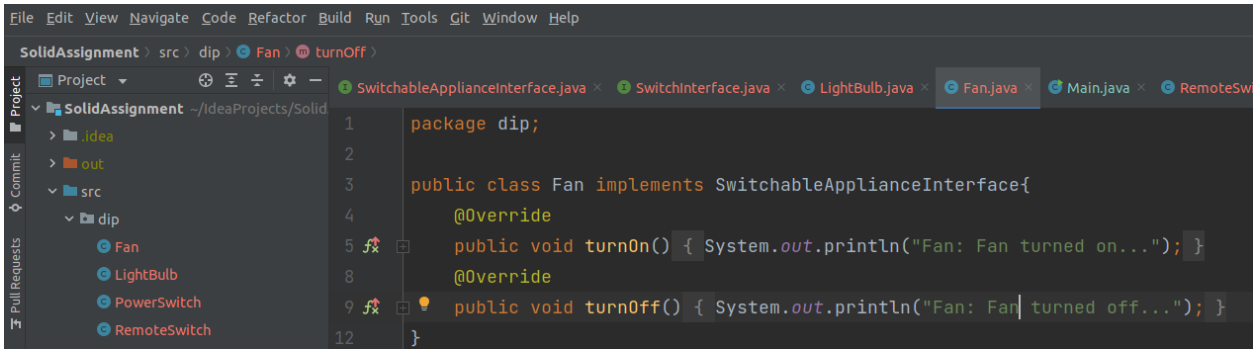
This is demonstrated below.



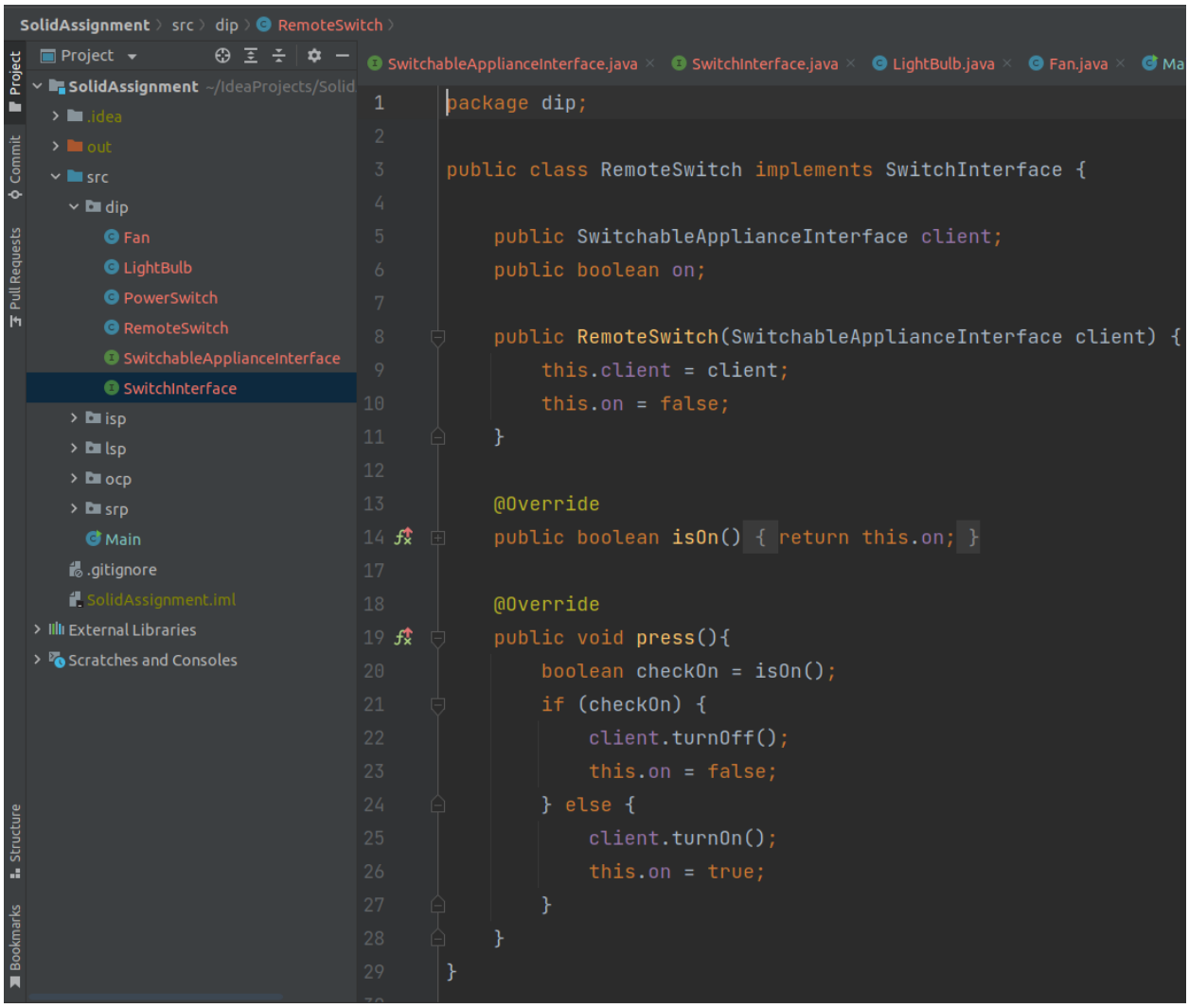
The SwitchInterface as outlined above.



The SwitchableApplianceInterface as outlined above.

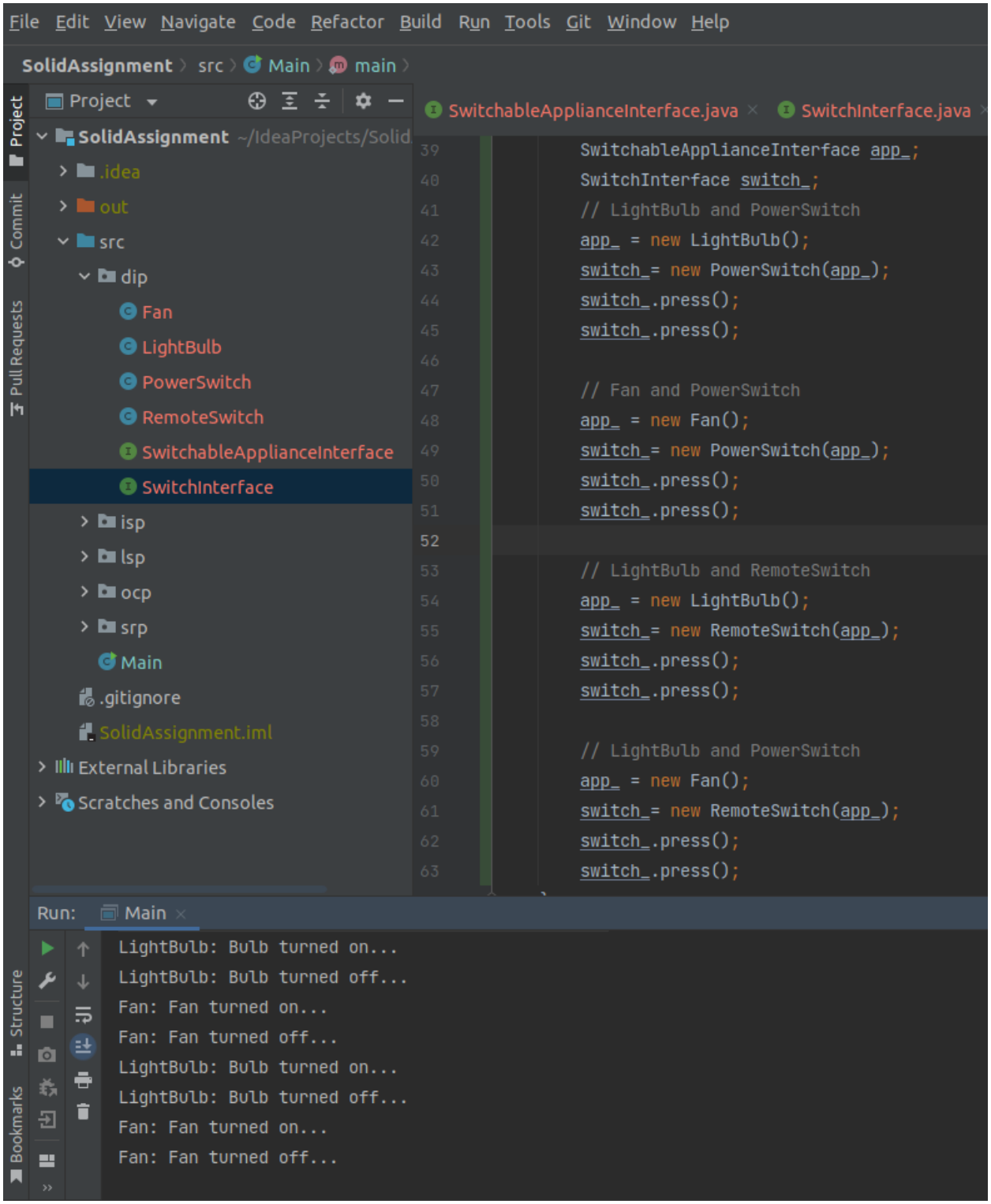


A Fan class representing a fan and implementing the SwitchableApplianceInterface as outlined above. The LightBulb class is also modified similarly to implement the SwitchableApplianceInterface.



A RemoteSwitch class implementing the SwitchInterface and relying on and overriding the isOn() and press() functions defined in the interface. The PowerSwitch class is modified similarly.

The main call area also relies heavily on the interfaces as shown below and can easily adapt to classes implementing the interfaces as shown below.



The same interface object is used to point to different classes implementing the interface. This makes the code much more **maintainable**, **readable** and **flexible**. It also becomes much easier to write newer electrical appliance classes and switch classes.

Conclusion

As can be seen from the multiple examples listed, following the SOLID principles is a “solid” first step to ensuring that the codebase for large projects becomes easier to handle. For small projects, however, a counter-argument may be posed by the large number of files needed to implement even a simple submodule. But, these principles are a great starting point to writing organised, maintainable, readable and flexible code.

The code for this assignment as well as this PDF is available at the following GitHub link: https://github.com/fayadchowdhury/solid-principles-java