

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ



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.

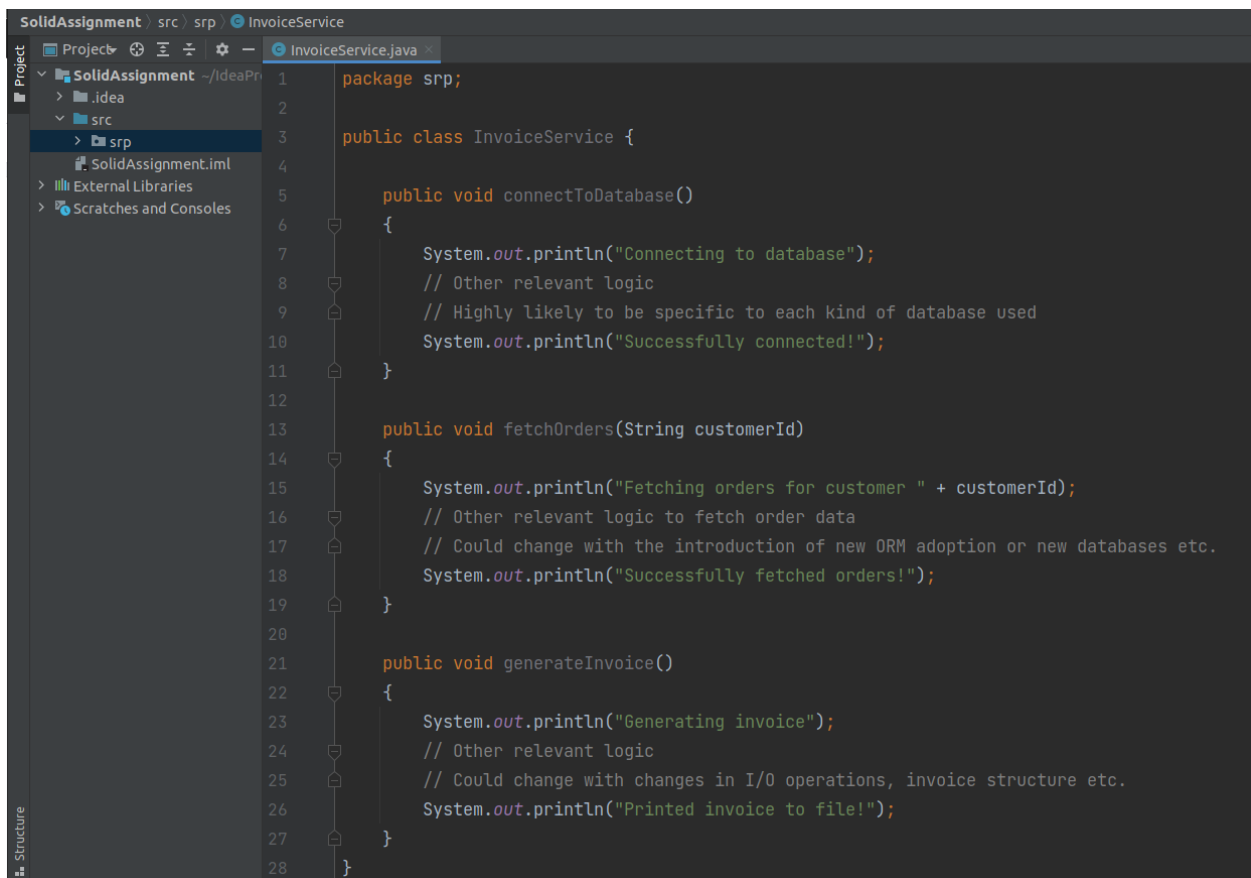
Single Responsibility Principle (SRP)

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.

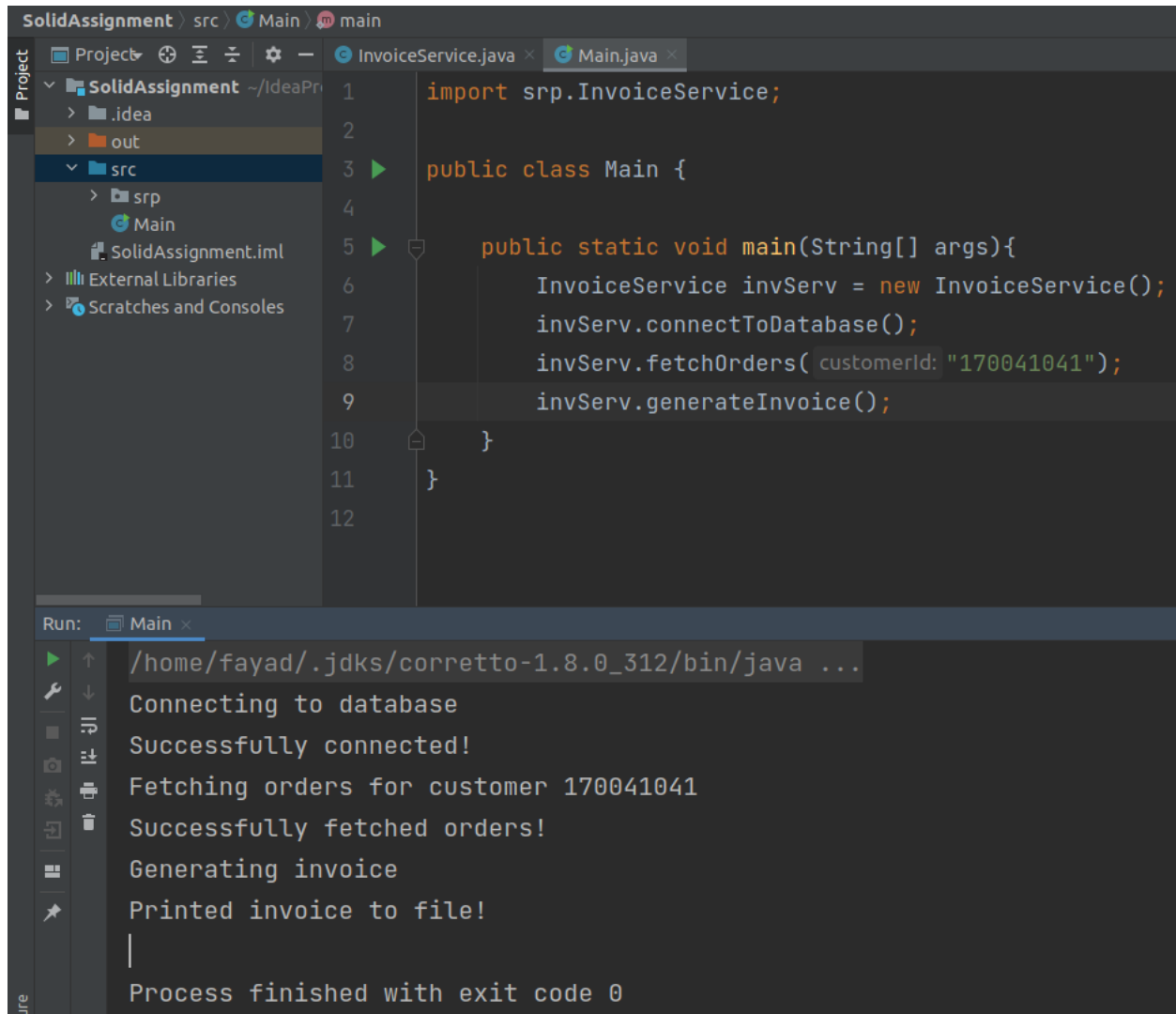
Considering the case of a module of software that generates invoices. Assume a single class is created and tasked with the responsibilities of:

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.



```
1 package srp;
2
3 public class InvoiceService {
4
5     public void connectToDatabase()
6     {
7         System.out.println("Connecting to database");
8         // Other relevant logic
9         // Highly likely to be specific to each kind of database used
10        System.out.println("Successfully connected!");
11    }
12
13    public void fetchOrders(String customerId)
14    {
15        System.out.println("Fetching orders for customer " + customerId);
16        // Other relevant logic to fetch order data
17        // Could change with the introduction of new ORM adoption or new databases etc.
18        System.out.println("Successfully fetched orders!");
19    }
20
21    public void generateInvoice()
22    {
23        System.out.println("Generating invoice");
24        // Other relevant logic
25        // Could change with changes in I/O operations, invoice structure etc.
26        System.out.println("Printed invoice to file!");
27    }
28 }
```



The screenshot shows an IDE window for a project named "SolidAssignment". The project structure on the left includes a "src" directory with a "Main" class. The main editor displays the code for "Main.java":

```
1 import srp.InvoiceService;
2
3 public class Main {
4
5     public static void main(String[] args){
6         InvoiceService invServ = new InvoiceService();
7         invServ.connectToDatabase();
8         invServ.fetchOrders( customerId: "170041041");
9         invServ.generateInvoice();
10    }
11 }
12
```

Below the code editor, the "Run" tab shows the execution output for "Main":

```
/home/fayad/.jdk/corretto-1.8.0_312/bin/java ...
Connecting to database
Successfully connected!
Fetching orders for customer 170041041
Successfully fetched orders!
Generating invoice
Printed invoice to file!
|
Process finished with exit code 0
```

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

```
1 package srp;
2
3 public class DatabaseConnectionService {
4
5     // Could have properties for ORM/ODM
6     public void connectToDatabase()
7     {
8         System.out.println("Connecting to database");
9         // Other relevant logic
10        // Highly likely to be specific to each kind of database used
11        System.out.println("Successfully connected!");
12    }
13 }
```

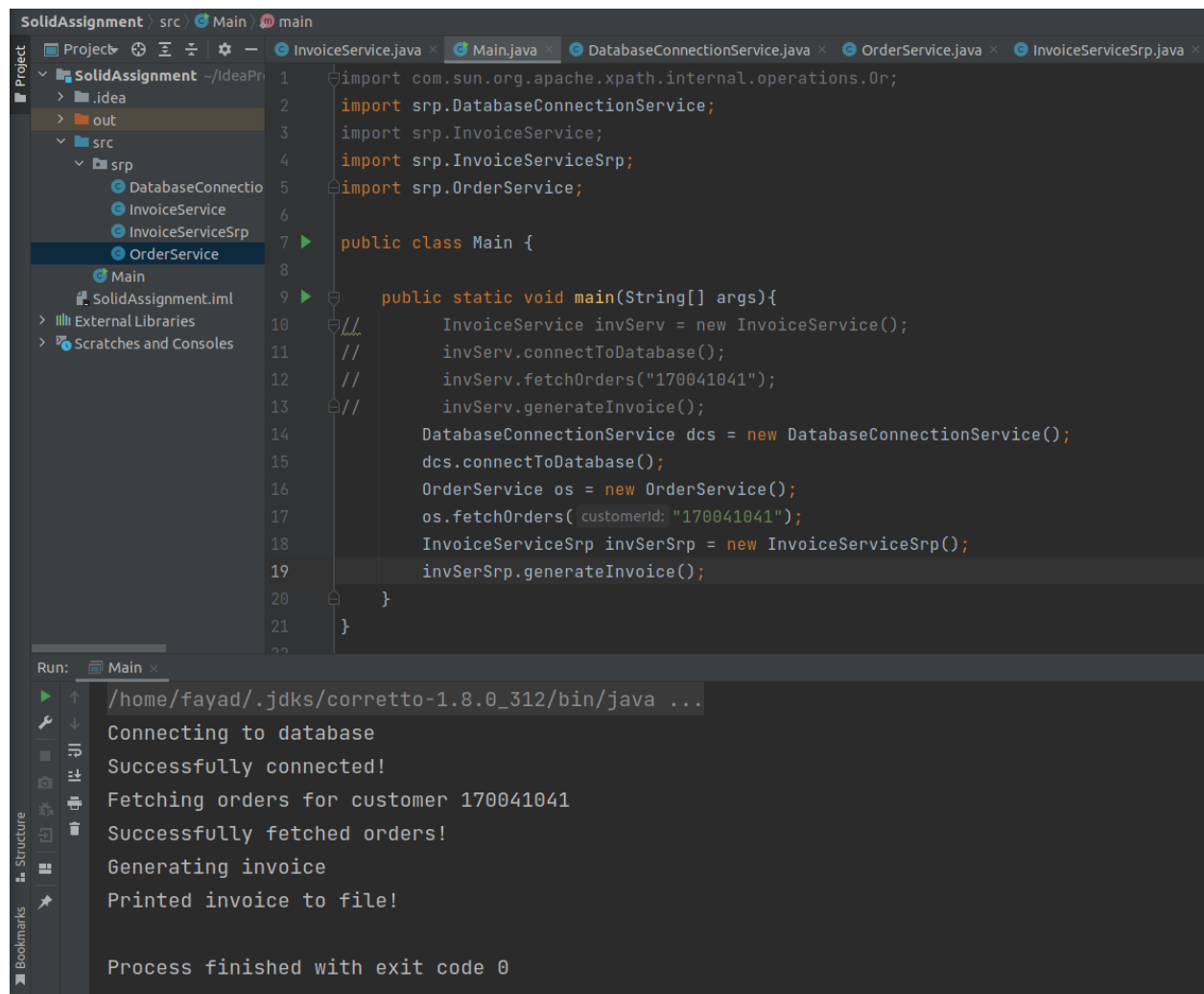
The DatabaseConnectionService is tasked with handling connections to the database.

```
1 package srp;
2
3 public class OrderService {
4
5     // Could have other relevant order-based functionality (creating, updating, deleting orders etc.)
6     public void fetchOrders(String customerId)
7     {
8         System.out.println("Fetching orders for customer " + customerId);
9         // Other relevant logic to fetch order data
10        // Could change with the introduction of new ORM adoption or new databases etc.
11        System.out.println("Successfully fetched orders!");
12    }
13 }
```

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

```
1 package srp;
2
3 public class InvoiceServiceSrp {
4
5     // Could have data structures relevant to generating invoices
6     // Could have additional I/O operations
7     public void generateInvoice()
8     {
9         System.out.println("Generating invoice");
10        // Other relevant logic
11        // Could change with changes in I/O operations, invoice structure etc.
12        System.out.println("Printed invoice to file!");
13    }
14 }
```

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



The screenshot displays an IDE window for a project named 'SolidAssignment'. The 'Project' view on the left shows a directory structure with 'src' containing 'srp' (with sub-packages 'DatabaseConnection', 'InvoiceService', 'InvoiceServiceSrp', and 'OrderService') and a 'Main' class. The 'Main.java' file is open in the editor, showing the following code:

```
1 import com.sun.org.apache.xpath.internal.operations.Or;
2 import srp.DatabaseConnectionService;
3 import srp.InvoiceService;
4 import srp.InvoiceServiceSrp;
5 import srp.OrderService;
6
7 public class Main {
8
9     public static void main(String[] args){
10         // InvoiceService invServ = new InvoiceService();
11         // invServ.connectToDatabase();
12         // invServ.fetchOrders("170041041");
13         // invServ.generateInvoice();
14         DatabaseConnectionService dcs = new DatabaseConnectionService();
15         dcs.connectToDatabase();
16         OrderService os = new OrderService();
17         os.fetchOrders( customerid: "170041041");
18         InvoiceServiceSrp invSerSrp = new InvoiceServiceSrp();
19         invSerSrp.generateInvoice();
20     }
21 }
```

Below the code editor, the 'Run' window shows the execution output for 'Main':

```
/home/fayad/.jdk/corretto-1.8.0_312/bin/java ...
Connecting to database
Successfully connected!
Fetching orders for customer 170041041
Successfully fetched orders!
Generating invoice
Printed invoice to file!

Process finished with exit code 0
```

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.

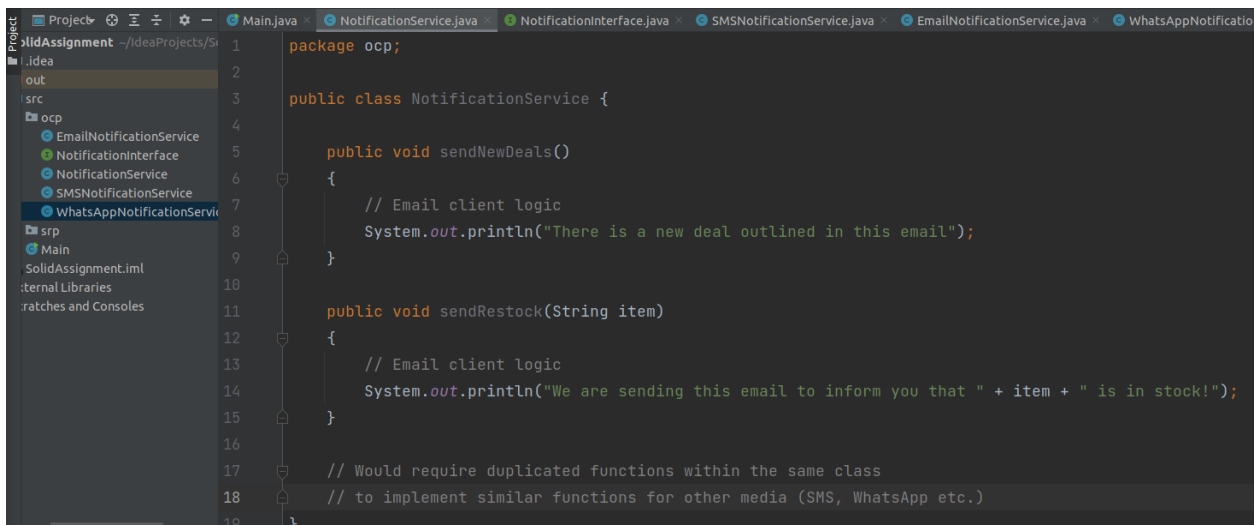
Open Closed Principle (OCP)

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 **issues 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:



```
package ocp;

public class NotificationService {

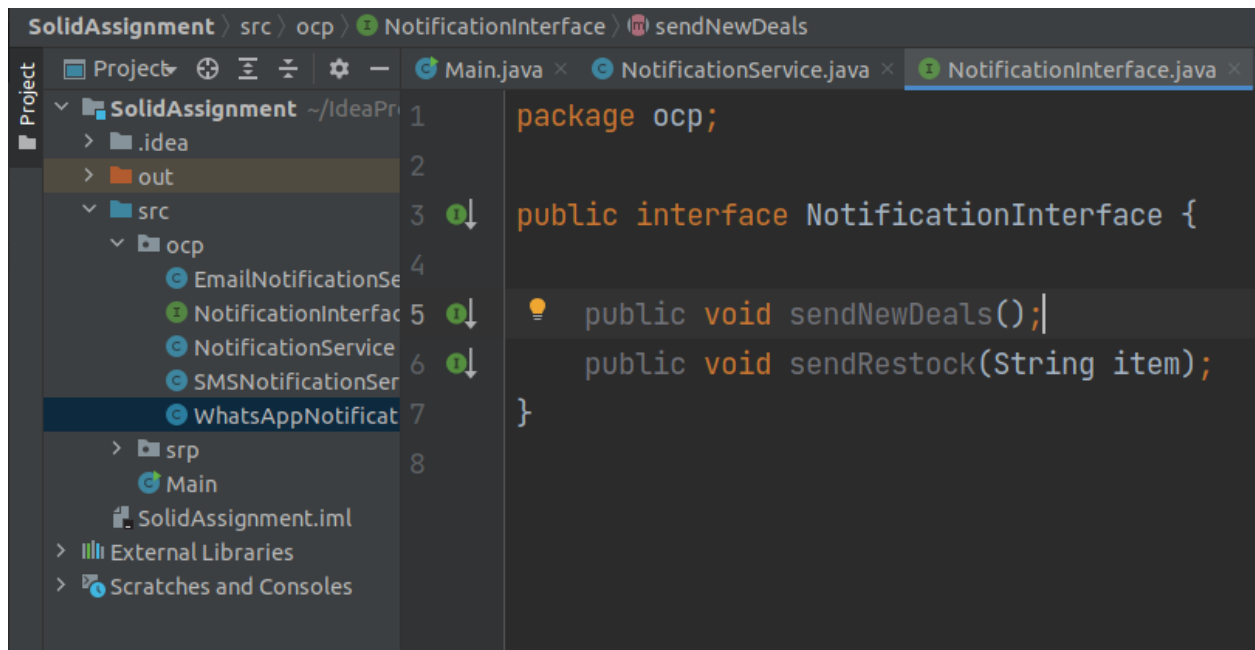
    public void sendNewDeals()
    {
        // Email client logic
        System.out.println("There is a new deal outlined in this email");
    }

    public void sendRestock(String item)
    {
        // Email client logic
        System.out.println("We are sending this email to inform you that " + item + " is in stock!");
    }

    // Would require duplicated functions within the same class
    // to implement similar functions for other media (SMS, WhatsApp etc.)
}
```

The screenshot shows an IDE with a project structure on the left and a code editor on the right. The project structure includes a package 'ocp' with classes 'EmailNotificationService', 'NotificationInterface', 'NotificationService', 'SMSNotificationService', and 'WhatsAppNotificationService'. The code editor displays the 'NotificationService' class, which has two methods: 'sendNewDeals()' and 'sendRestock(String item)'. Both methods use 'Email client logic' for sending notifications, as indicated by the comments. A comment at the bottom of the class states: '// Would require duplicated functions within the same class // to implement similar functions for other media (SMS, WhatsApp etc.)'.

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:

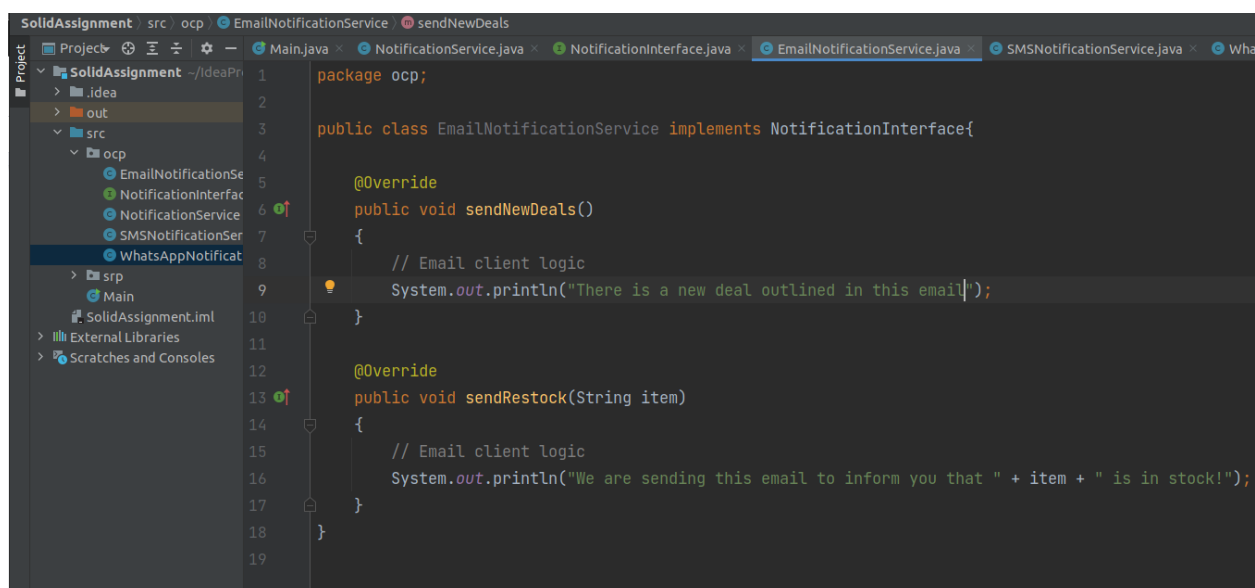


```
package ocp;

public interface NotificationInterface {

    public void sendNewDeals();
    public void sendRestock(String item);
}
```

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



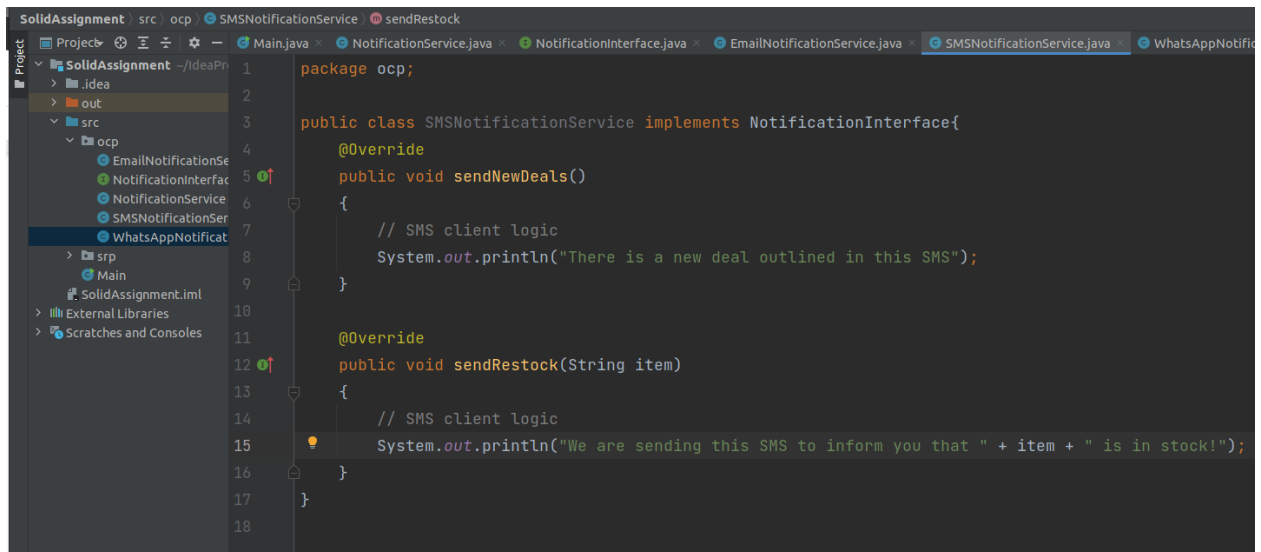
```
package ocp;

public class EmailNotificationService implements NotificationInterface{

    @Override
    public void sendNewDeals()
    {
        // Email client logic
        System.out.println("There is a new deal outlined in this email!");
    }

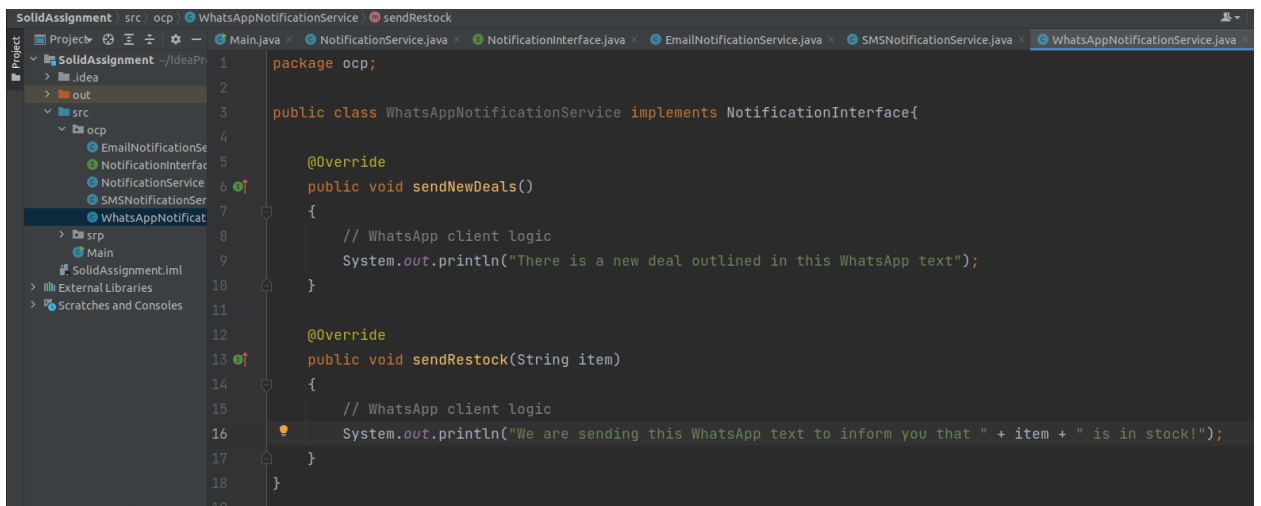
    @Override
    public void sendRestock(String item)
    {
        // Email client logic
        System.out.println("We are sending this email to inform you that " + item + " is in stock!");
    }
}
```


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



```
1 package ocp;
2
3 public class SMSNotificationService implements NotificationInterface{
4     @Override
5     public void sendNewDeals()
6     {
7         // SMS client logic
8         System.out.println("There is a new deal outlined in this SMS");
9     }
10
11     @Override
12     public void sendRestock(String item)
13     {
14         // SMS client logic
15         System.out.println("We are sending this SMS to inform you that " + item + " is in stock!");
16     }
17 }
18
```

Similar implementation of the SMSNotificationService class.



```
1 package ocp;
2
3 public class WhatsAppNotificationService implements NotificationInterface{
4     @Override
5     public void sendNewDeals()
6     {
7         // WhatsApp client logic
8         System.out.println("There is a new deal outlined in this WhatsApp text");
9     }
10
11     @Override
12     public void sendRestock(String item)
13     {
14         // WhatsApp client logic
15         System.out.println("We are sending this WhatsApp text to inform you that " + item + " is in stock!");
16     }
17 }
18
```

Similar implementation of the WhatsAppNotificationService class.

The screenshot displays an IDE window titled "SolidAssignment - Main.java". The left sidebar shows the project structure with folders like "src", "out", "dip", "isp", "lsp", and "ocp". The "EmailNotificationService" class is selected under the "src" folder. The main editor shows the "Main.java" file with the following code:

```
14 public static void main(String[] args){
15     //
16     //     InvoiceService invServ = new InvoiceService();
17     //     invServ.connectToDatabase();
18     //     invServ.fetchOrders("170041041");
19     //     invServ.generateInvoice();
20     //     DatabaseConnectionService dcs = new DatabaseConnectionService();
21     //     dcs.connectToDatabase();
22     //     OrderService os = new OrderService();
23     //     os.fetchOrders("170041041");
24     //     InvoiceServiceSrp invSerSrp = new InvoiceServiceSrp();
25     //     invSerSrp.generateInvoice();
26     //     OCP
27     NotificationInterface ni = new EmailNotificationService();
28     ni.sendNewDeals();
29     ni.sendRestock( item: "Nescafe Gold Coffee");
30     ni = new SMSNotificationService();
31     ni.sendNewDeals();
32     ni.sendRestock( item: "Nescafe Gold Coffee");
33     //     LSP
```

The bottom panel shows the output of the program:

```
Run: Main x
/home/fayad/.jdk/corretto-1.8.0_312/bin/java ...
There is a new deal outlined in this email
We are sending this email to inform you that Nescafe Gold Coffee is in stock!
There is a new deal outlined in this SMS
We are sending this SMS to inform you that Nescafe Gold Coffee is in stock!

Process finished with exit code 0
```

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)

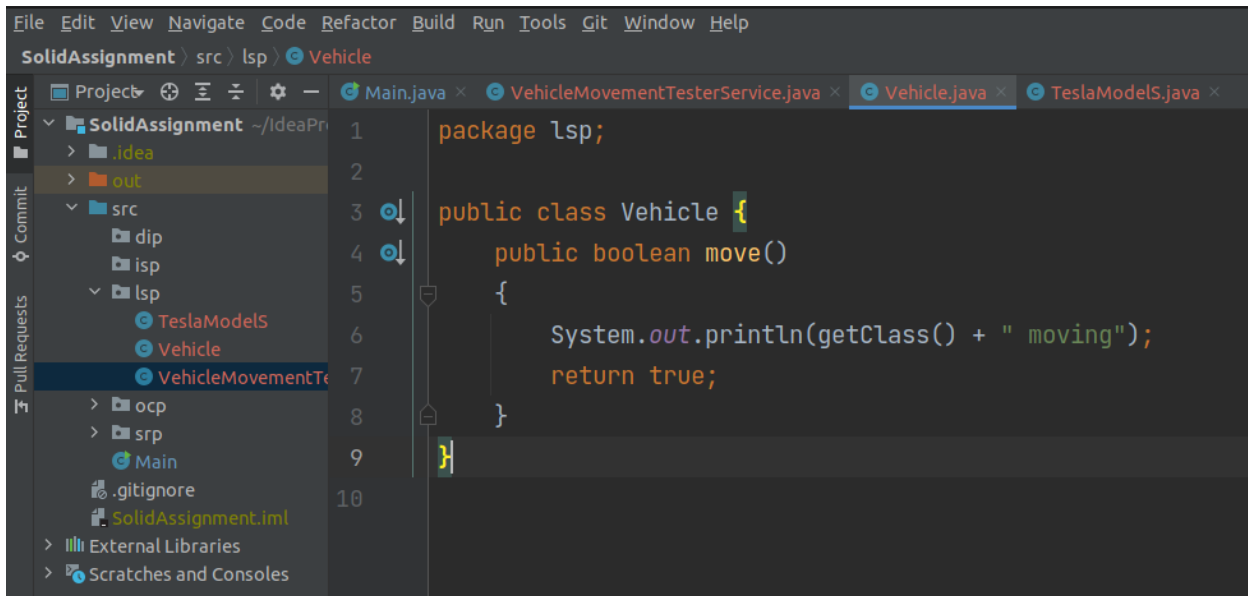
LSP states that **if a class A is a subtype of class B (inherits from class B), then it should be possible to replace instances of B with those of A without disrupting the behaviour of the program.** This implies that **derived classes must be completely substitutable for their base classes.**

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.**

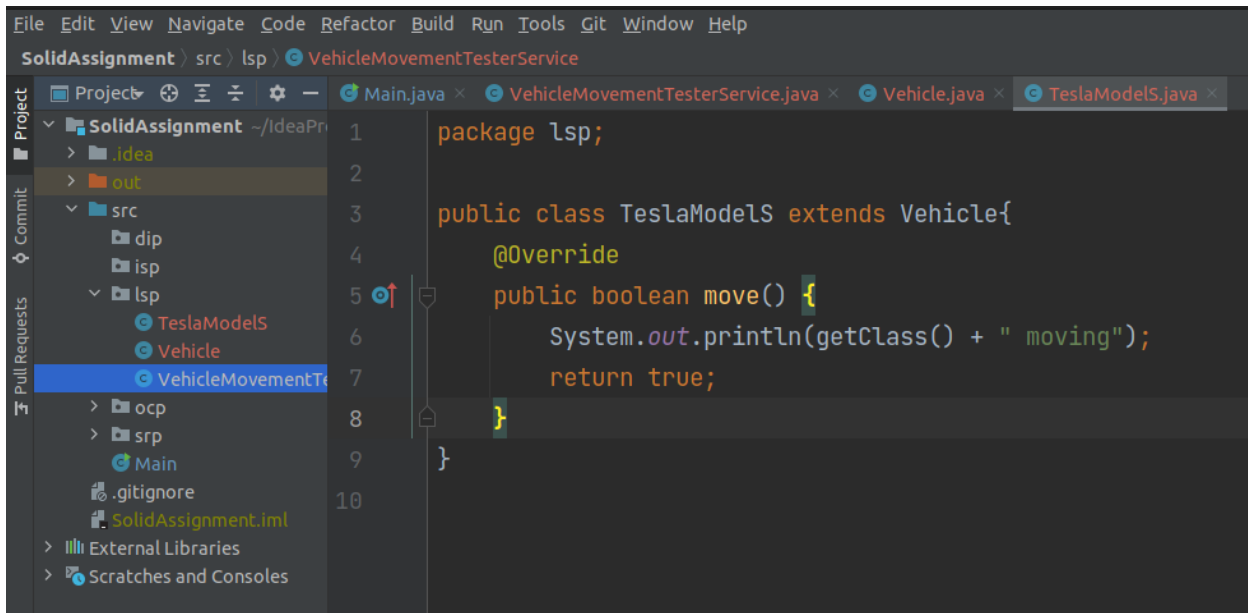
Consider a parent class Vehicle that is a skeleton for all sorts of motor and electric vehicles. Further consider a derived or child class, TeslaModelS, which is a skeleton for all Tesla Model S vehicles. Both classes will have some function move() that will detail how a generic vehicle moves (represented 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.



```
File Edit View Navigate Code Refactor Build Run Tools Git Window Help
SolidAssignment > src > lsp > Vehicle
Project: SolidAssignment ~/IdeaPr
  > .idea
  > out
  > src
    > dip
    > isp
    > lsp
      TeslaModelS
      Vehicle
      VehicleMovementTe
    > ocp
    > srp
    Main
  .gitignore
  SolidAssignment.iml
  External Libraries
  Scratches and Consoles
1 package lsp;
2
3 public class Vehicle {
4     public boolean move()
5     {
6         System.out.println(getClass() + " moving");
7         return true;
8     }
9 }
10
```

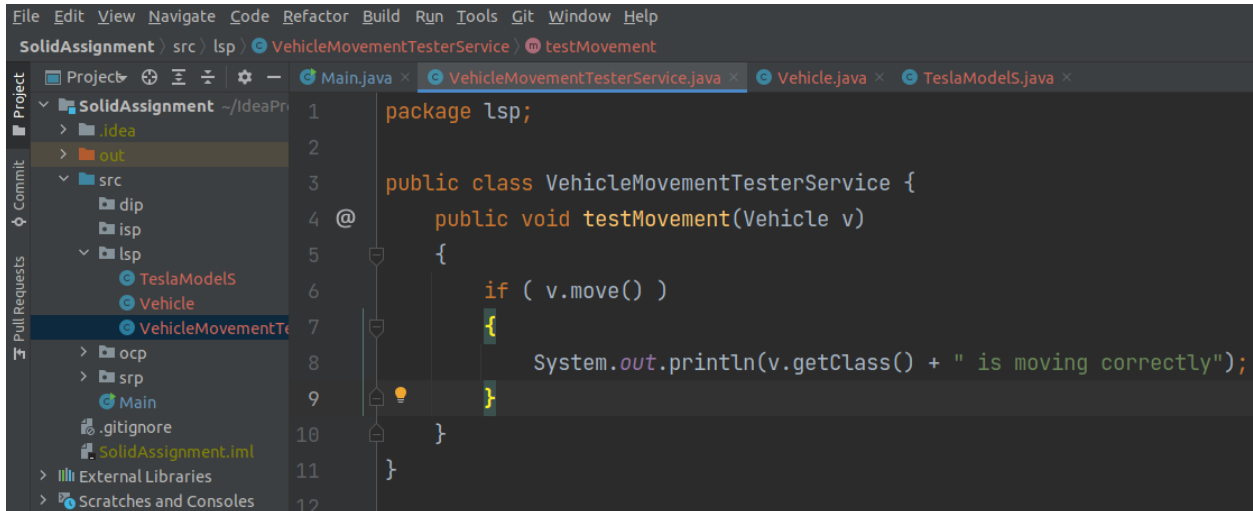
The base Vehicle class to represent generic vehicles.



```
File Edit View Navigate Code Refactor Build Run Tools Git Window Help
SolidAssignment > src > lsp > VehicleMovementTesterService
Project: SolidAssignment ~/IdeaPr
  > .idea
  > out
  > src
    > dip
    > isp
    > lsp
      TeslaModelS
      Vehicle
      VehicleMovementTe
    > ocp
    > srp
    Main
  .gitignore
  SolidAssignment.iml
  External Libraries
  Scratches and Consoles
1 package lsp;
2
3 public class TeslaModelS extends Vehicle{
4     @Override
5     public boolean move() {
6         System.out.println(getClass() + " moving");
7         return true;
8     }
9 }
10
```

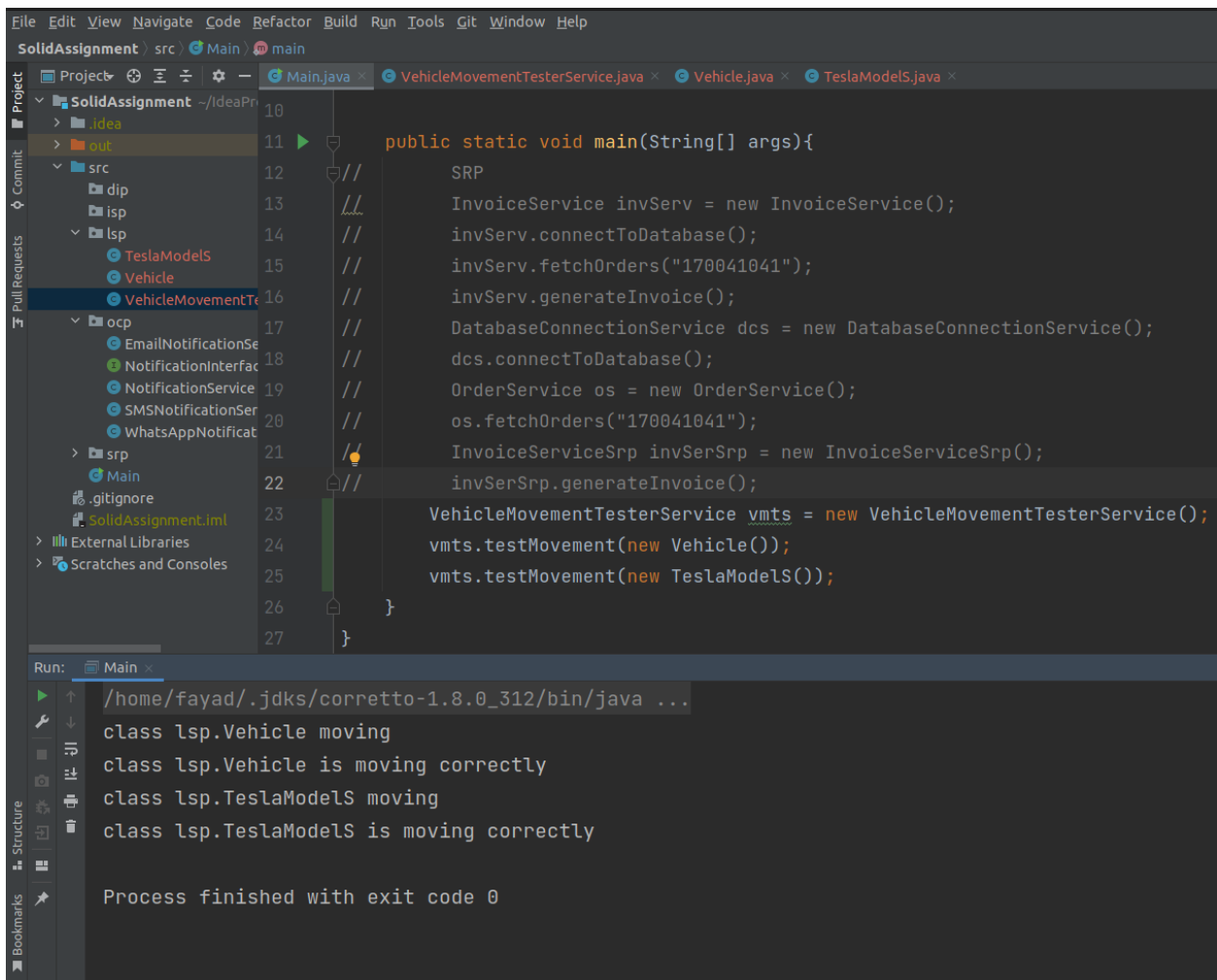
The TeslaModelS class that extends the Vehicle class.

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.



```
1 package lsp;
2
3 public class VehicleMovementTesterService {
4     @
5     public void testMovement(Vehicle v)
6     {
7         if ( v.move() )
8         {
9             System.out.println(v.getClass() + " is moving correctly");
10        }
11    }
12 }
```

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



```
10
11 public static void main(String[] args){
12     // SRP
13     // InvoiceService invServ = new InvoiceService();
14     // invServ.connectToDatabase();
15     // invServ.fetchOrders("170041041");
16     // invServ.generateInvoice();
17     // DatabaseConnectionService dcs = new DatabaseConnectionService();
18     // dcs.connectToDatabase();
19     // OrderService os = new OrderService();
20     // os.fetchOrders("170041041");
21     // InvoiceServiceSrp invSerSrp = new InvoiceServiceSrp();
22     // invSerSrp.generateInvoice();
23     VehicleMovementTesterService vmts = new VehicleMovementTesterService();
24     vmts.testMovement(new Vehicle());
25     vmts.testMovement(new TeslaModels());
26 }
27 }
```

Run: Main

```
/home/fayad/.jdk/corretto-1.8.0_312/bin/java ...
class lsp.Vehicle moving
class lsp.Vehicle is moving correctly
class lsp.TeslaModels moving
class lsp.TeslaModels is moving correctly

Process finished with exit code 0
```

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.

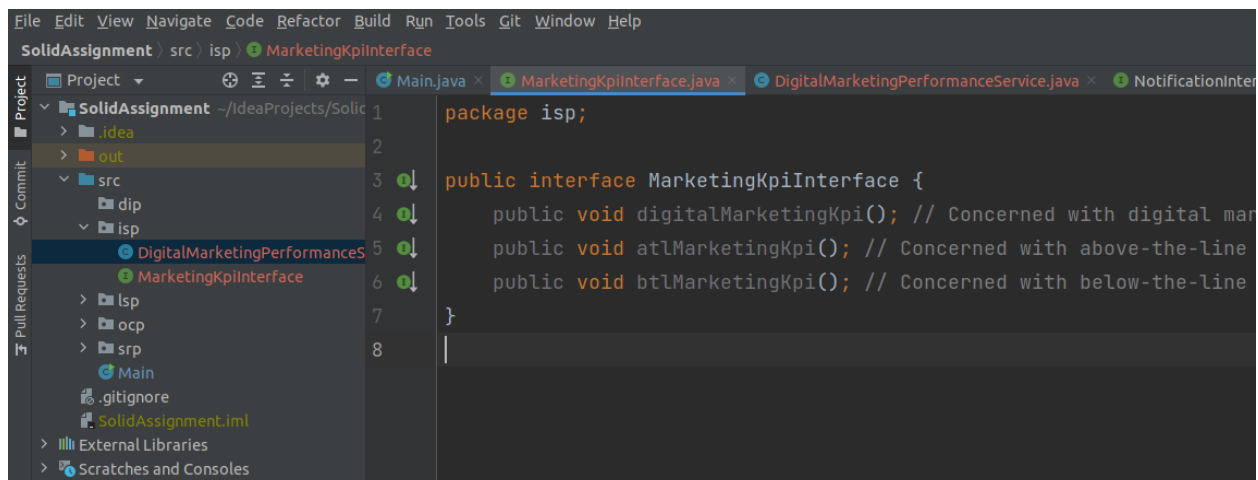
Interface Segregation Principle (ISP)

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).

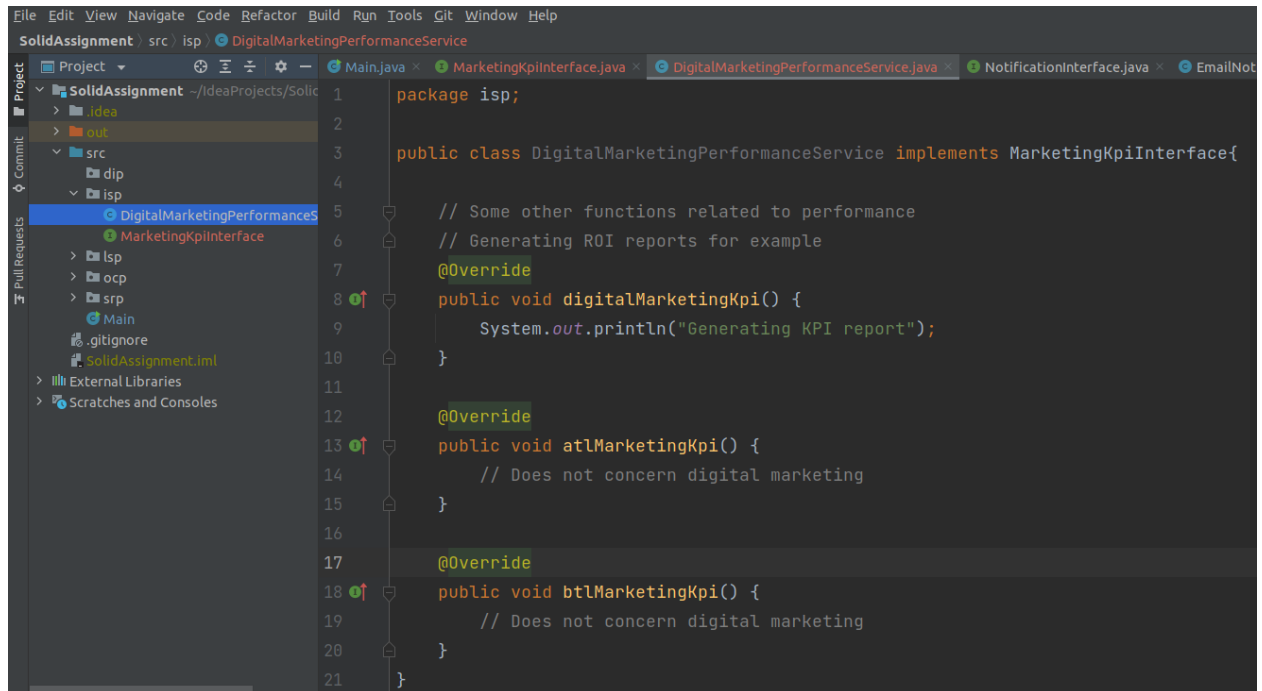
This makes the codebase more **readable** and **maintainable**, especially for large projects.

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.



```
File Edit View Navigate Code Refactor Build Run Tools Git Window Help
SolidAssignment | src \ isp \ MarketingKpiInterface
Project
  SolidAssignment ~\IdeaProjects\SolidAssignment
    .idea
    out
    src
      dip
      isp
        DigitalMarketingPerformanceService.java
        MarketingKpiInterface.java
      isp
      ocp
      srp
      Main
      .gitignore
      SolidAssignment.iml
    External Libraries
    Scratches and Consoles
Commit
Pull Requests
1 package isp;
2
3
4 public interface MarketingKpiInterface {
5     public void digitalMarketingKpi(); // Concerned with digital mar
6     public void atlMarketingKpi(); // Concerned with above-the-line
7     public void btlMarketingKpi(); // Concerned with below-the-line
8 }
```

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

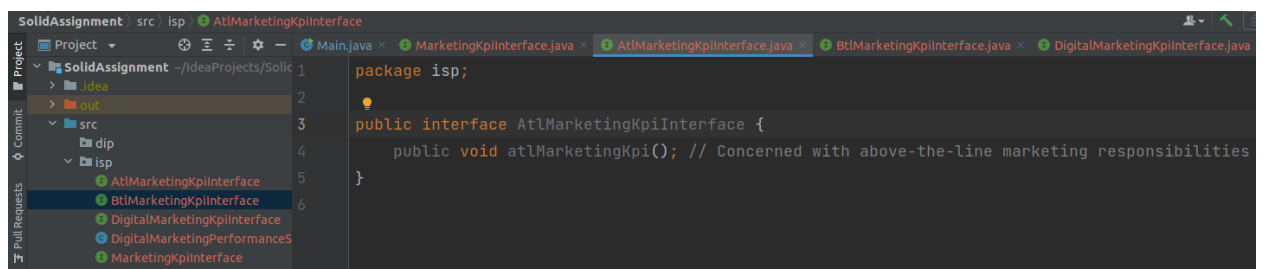


The screenshot shows an IDE window with the project 'SolidAssignment' and the file 'DigitalMarketingPerformanceService.java' open. The code defines a class that implements the 'MarketingKpiInterface'. It includes comments about performance-related functions and overrides for 'digitalMarketingKpi()', 'atlMarketingKpi()', and 'btlMarketingKpi()'. The 'atlMarketingKpi()' and 'btlMarketingKpi()' methods are marked as 'Does not concern digital marketing'.

```
1 package isp;
2
3 public class DigitalMarketingPerformanceService implements MarketingKpiInterface{
4
5     // Some other functions related to performance
6     // Generating ROI reports for example
7     @Override
8     public void digitalMarketingKpi() {
9         System.out.println("Generating KPI report");
10    }
11
12    @Override
13    public void atlMarketingKpi() {
14        // Does not concern digital marketing
15    }
16
17    @Override
18    public void btlMarketingKpi() {
19        // Does not concern digital marketing
20    }
21 }
```

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.

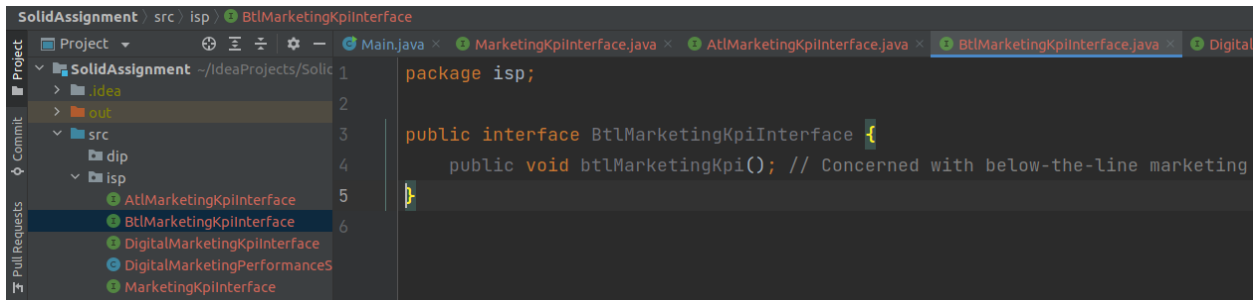
Keeping in mind ISP, we separate the KPI calculation functions into 3 separate interfaces as shown below.



The screenshot shows an IDE window with the project 'SolidAssignment' and the file 'AtlMarketingKpiInterface.java' open. The code defines a public interface 'AtlMarketingKpiInterface' with a single method 'atlMarketingKpi()' and a comment indicating it is concerned with above-the-line marketing responsibilities.

```
1 package isp;
2
3 public interface AtlMarketingKpiInterface {
4     public void atlMarketingKpi(); // Concerned with above-the-line marketing responsibilities
5 }
6
```

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



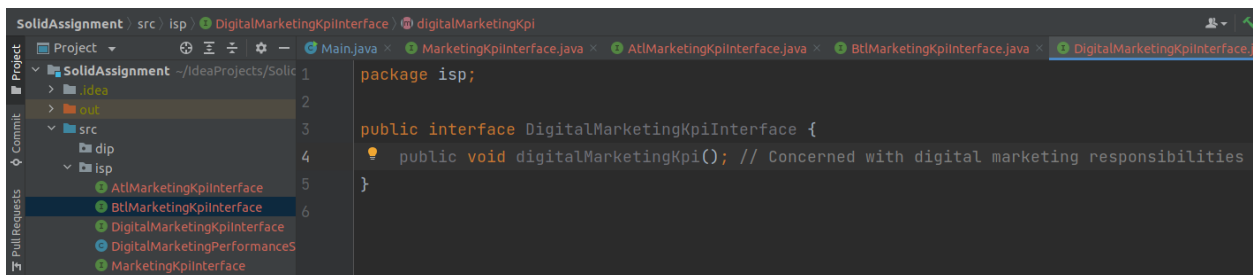
```
package isp;

public interface BtlMarketingKpiInterface {

    public void btlMarketingKpi(); // Concerned with below-the-line marketing

}
```

A similar implementation of BtlMarketingKpiInterface.



```
package isp;

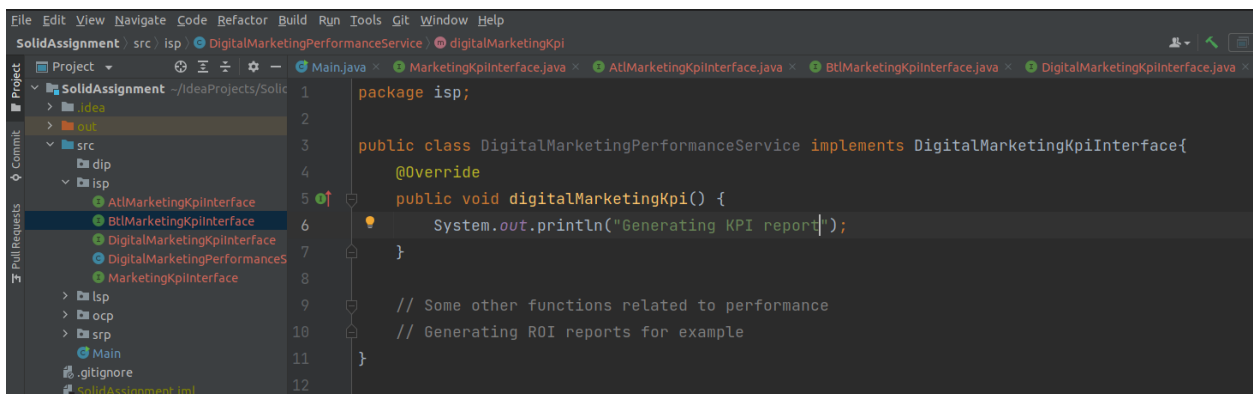
public interface DigitalMarketingKpiInterface {

    public void digitalMarketingKpi(); // Concerned with digital marketing responsibilities

}
```

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



```
package isp;

public class DigitalMarketingPerformanceService implements DigitalMarketingKpiInterface{

    @Override
    public void digitalMarketingKpi() {

        System.out.println("Generating KPI report");

    }

    // Some other functions related to performance
    // Generating ROI reports for example

}
```

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.

```
File Edit View Navigate Code Refactor Build Run Tools Git Window Help

SolidAssignment > src > dip > PowerSwitch >

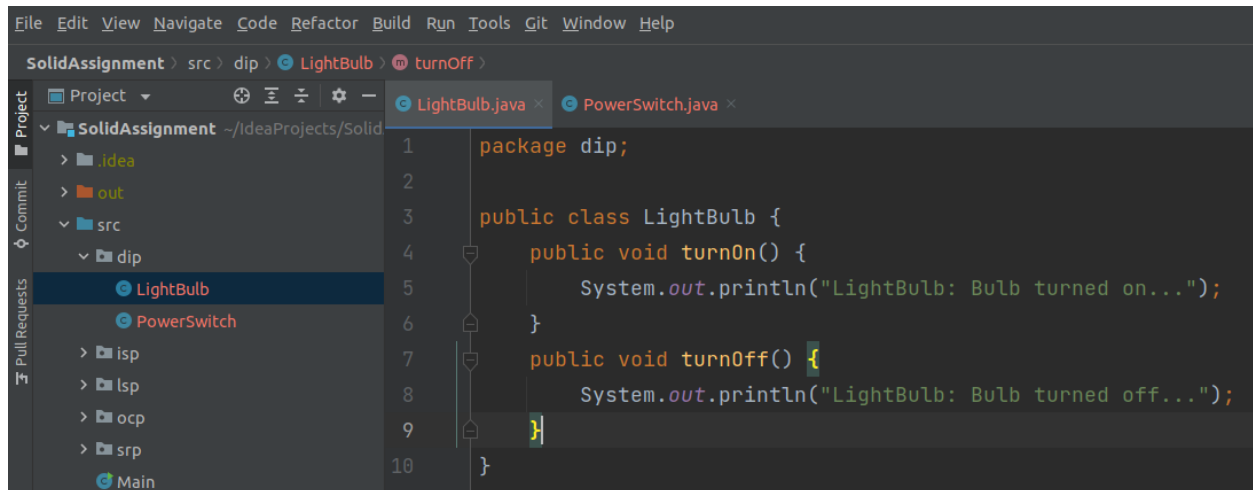
Project
  SolidAssignment ~/IdeaProjects/SolidAssignment
    .idea
    out
    src
      dip
        LightBulb
        PowerSwitch
      isp
      lsp
      ocp
      srp
      Main
      .gitignore
      SolidAssignment.iml
    External Libraries
    Scratches and Consoles

Pull Requests

Structure
  Bookmarks

1 package dip;
2
3 public class PowerSwitch {
4
5     public LightBulb lightBulb;
6     public boolean on;
7
8     public PowerSwitch(LightBulb lightBulb)
9     {
10         this.lightBulb = lightBulb;
11         this.on = false;
12     }
13
14     public boolean isOn()
15     {
16         return this.on;
17     }
18
19     public void press()
20     {
21         boolean checkOn = isOn();
22         if (checkOn) // Turn off appliance if on
23         {
24             lightBulb.turnOff();
25             this.on = false;
26         }
27         else
28         {
29             lightBulb.turnOn();
30             this.on = true;
31         }
32     }
33 }
```

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.



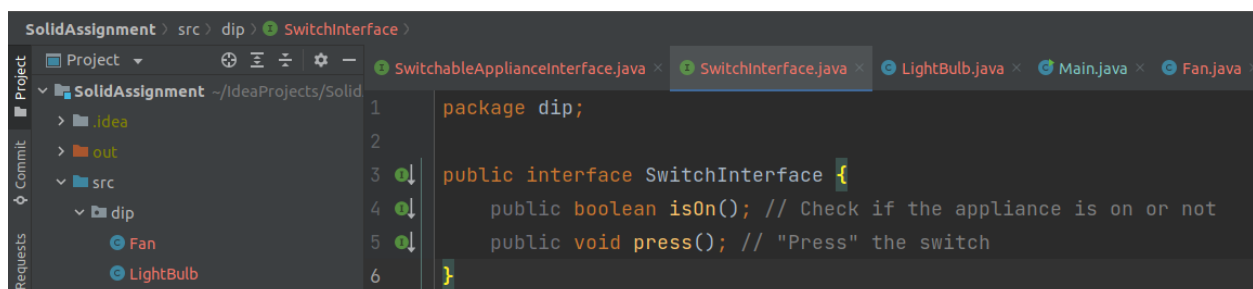
```
1 package dip;
2
3 public class LightBulb {
4     public void turnOn() {
5         System.out.println("LightBulb: Bulb turned on...");
6     }
7     public void turnOff() {
8         System.out.println("LightBulb: Bulb turned off...");
9     }
10 }
```

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:

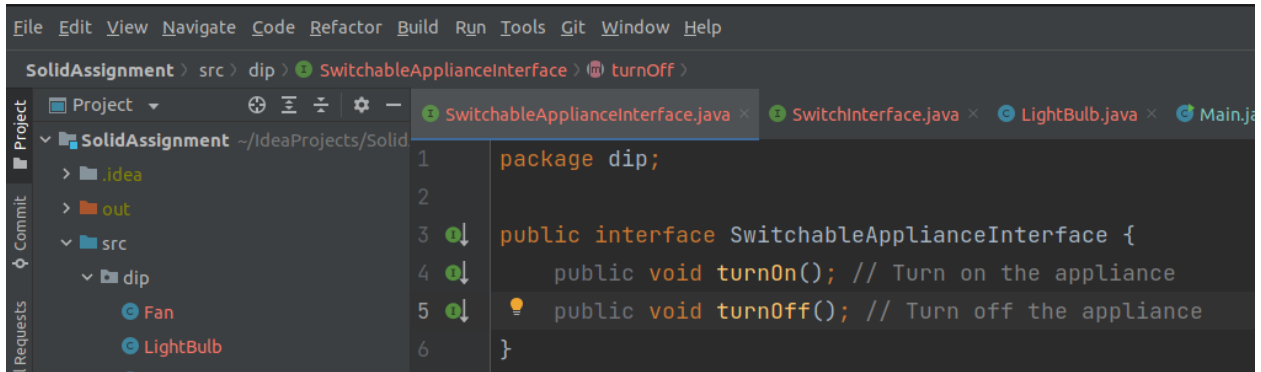
1. A SwitchableApplianceInterface that the electrical appliances will implement. This will contain the on/off function prototypes that each electrical appliance's class will override.
2. A SwitchInterface that each kind of switch will implement. This will contain the prototypes of the press function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

This is demonstrated below.



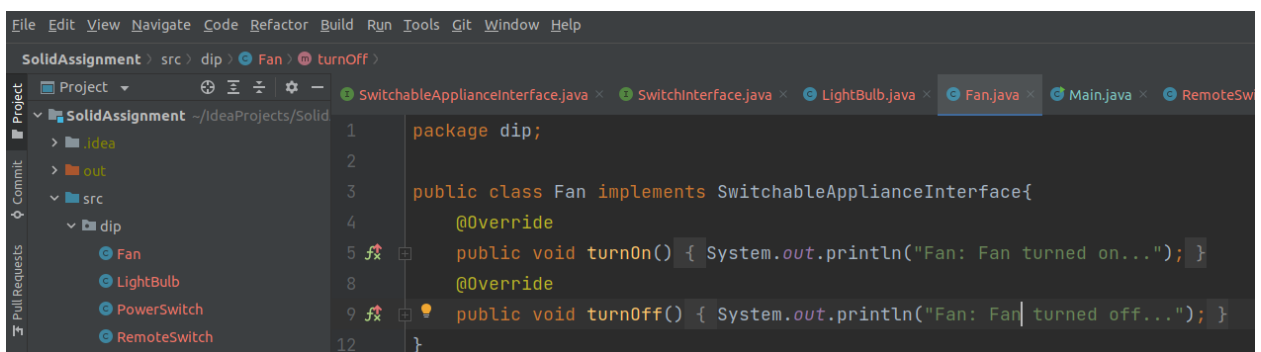
```
1 package dip;
2
3 public interface SwitchInterface {
4     public boolean isOn(); // Check if the appliance is on or not
5     public void press(); // "Press" the switch
6 }
```

The SwitchInterface as outlined above.



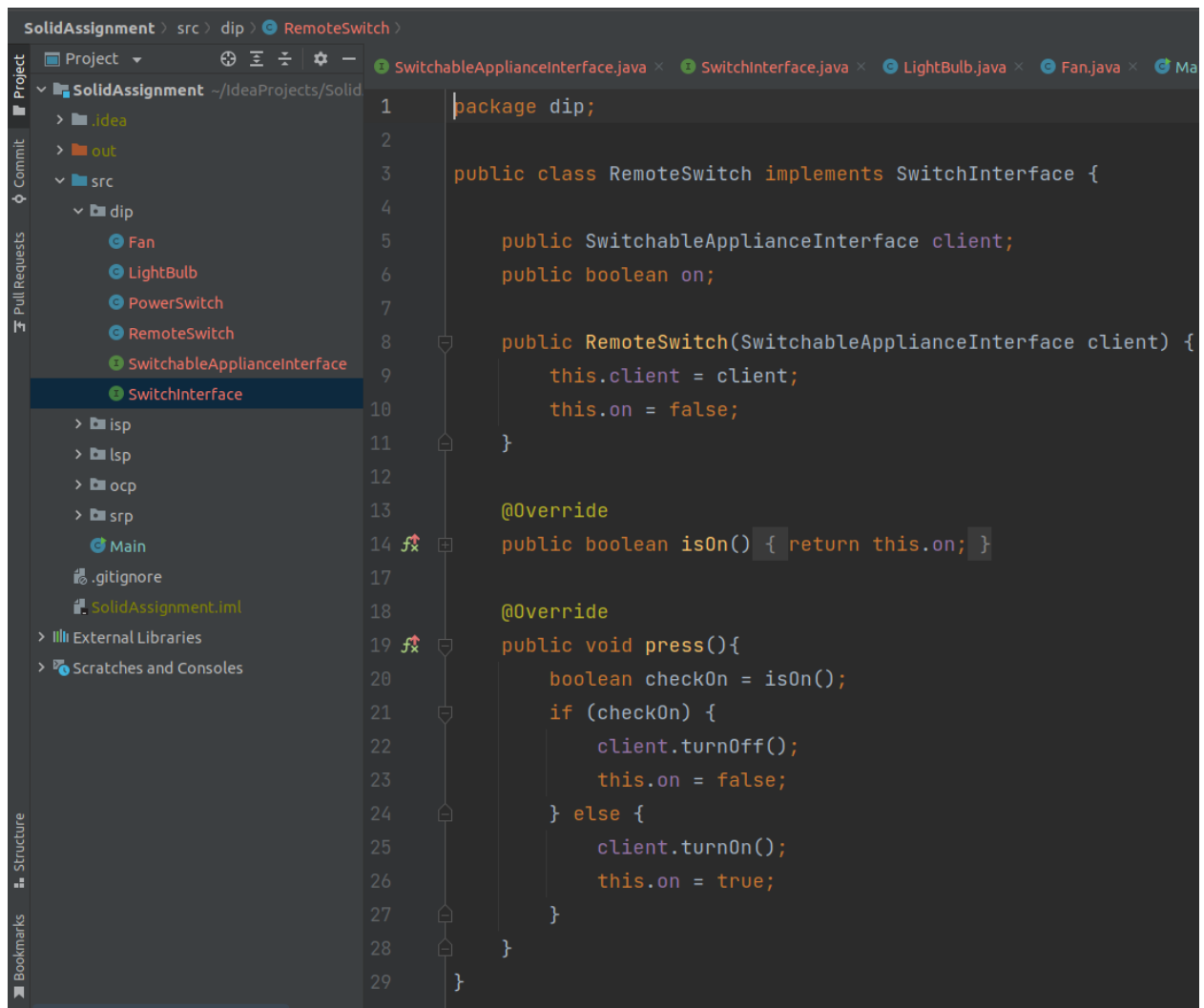
```
File Edit View Navigate Code Refactor Build Run Tools Git Window Help
SolidAssignment > src > dip > SwitchableApplianceInterface > turnOff >
Project
  SolidAssignment -/IdeaProjects/SolidAssignment
    .idea
    out
    src
      dip
        Fan
        LightBulb
SwitchableApplianceInterface.java x SwitchInterface.java x LightBulb.java x Main.java
1 package dip;
2
3 public interface SwitchableApplianceInterface {
4     public void turnOn(); // Turn on the appliance
5     public void turnOff(); // Turn off the appliance
6 }
```

The SwitchableApplianceInterface as outlined above.



```
File Edit View Navigate Code Refactor Build Run Tools Git Window Help
SolidAssignment > src > dip > Fan > turnOff >
Project
  SolidAssignment -/IdeaProjects/SolidAssignment
    .idea
    out
    src
      dip
        Fan
        LightBulb
        PowerSwitch
        RemoteSwitch
SwitchableApplianceInterface.java x SwitchInterface.java x LightBulb.java x Fan.java x Main.java x RemoteSwitch.java
1 package dip;
2
3 public class Fan implements SwitchableApplianceInterface{
4     @Override
5     public void turnOn() { System.out.println("Fan: Fan turned on..."); }
6
7     @Override
8     public void turnOff() { System.out.println("Fan: Fan turned off..."); }
9 }
10
11
12 }
```

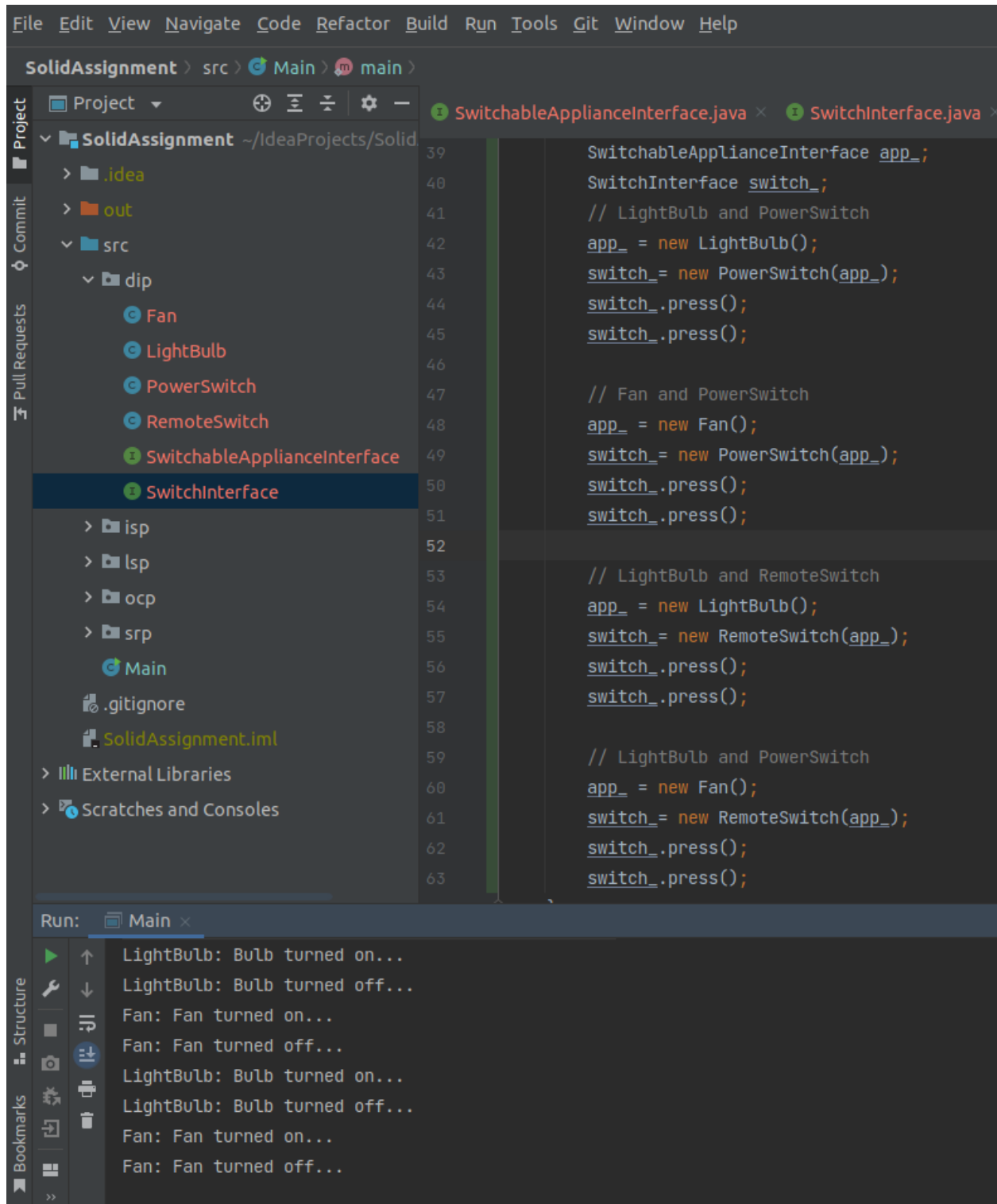
A Fan class representing a fan and implementing the SwitchableApplianceInterface as outlined above. The LightBulb class is also modified similarly to implement the SwitchableApplianceInterface.

The image is a screenshot of an IDE window. On the left, the 'Project' view shows a directory structure for 'SolidAssignment'. The 'src' directory is expanded, showing subdirectories 'dip', 'isp', 'lsp', 'ocp', and 'srp'. The 'dip' directory is selected, and its contents are listed: 'Fan', 'LightBulb', 'PowerSwitch', 'RemoteSwitch', 'SwitchableApplianceInterface', and 'SwitchInterface'. The 'SwitchInterface' is highlighted. The main editor area shows the code for 'RemoteSwitch.java'. The code is as follows:

```
1 package dip;
2
3 public class RemoteSwitch implements SwitchInterface {
4
5     public SwitchableApplianceInterface client;
6     public boolean on;
7
8     public RemoteSwitch(SwitchableApplianceInterface client) {
9         this.client = client;
10        this.on = false;
11    }
12
13    @Override
14    public boolean isOn() { return this.on; }
15
16
17    @Override
18    public void press(){
19        boolean checkOn = isOn();
20        if (checkOn) {
21            client.turnOff();
22            this.on = false;
23        } else {
24            client.turnOn();
25            this.on = true;
26        }
27    }
28 }
29 }
```

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>