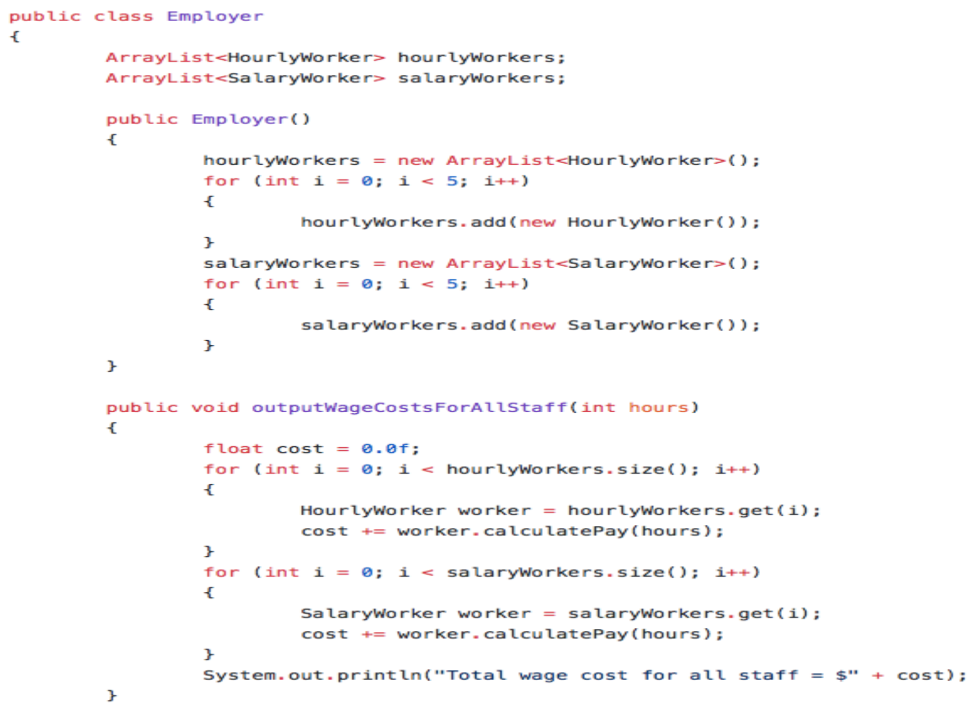
Assignment 2

Question 1

1. Which S.O.L.I.D. principle does the Employer class violate?

The Employer Class violates Dependency Inversion Principle (DIP) which states that a class must depend on abstraction rather than concretion.

2. Why does the code violate the principle?

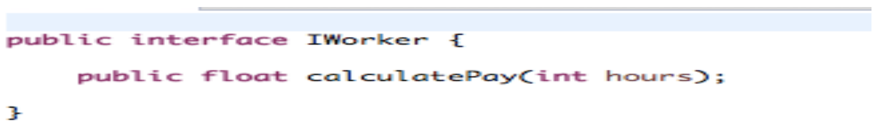


```java
public class Employer
{
        ArrayList<HourlyWorker> hourlyWorkers;
        ArrayList<SalaryWorker> salaryWorkers;

        public Employer()
        {
                hourlyWorkers = new ArrayList<HourlyWorker>();
                for (int i = 0; i < 5; i++)
                {
                        hourlyWorkers.add(new HourlyWorker());
                }
                salaryWorkers = new ArrayList<SalaryWorker>();
                for (int i = 0; i < 5; i++)
                {
                        salaryWorkers.add(new SalaryWorker());
                }
        }

        public void outputWageCostsForAllStaff(int hours)
        {
                float cost = 0.0f;
                for (int i = 0; i < hourlyWorkers.size(); i++)
                {
                        HourlyWorker worker = hourlyWorkers.get(i);
                        cost += worker.calculatePay(hours);
                }
                for (int i = 0; i < salaryWorkers.size(); i++)
                {
                        SalaryWorker worker = salaryWorkers.get(i);
                        cost += worker.calculatePay(hours);
                }
                System.out.println("Total wage cost for all staff = $" + cost);
        }
```

Employer class depend on other concrete classes such as HourlyWorker and SalaryWorker. Any change in these classes affect Employer class also. This is an example of tight coupling.

3. Write code that fixes the violation.

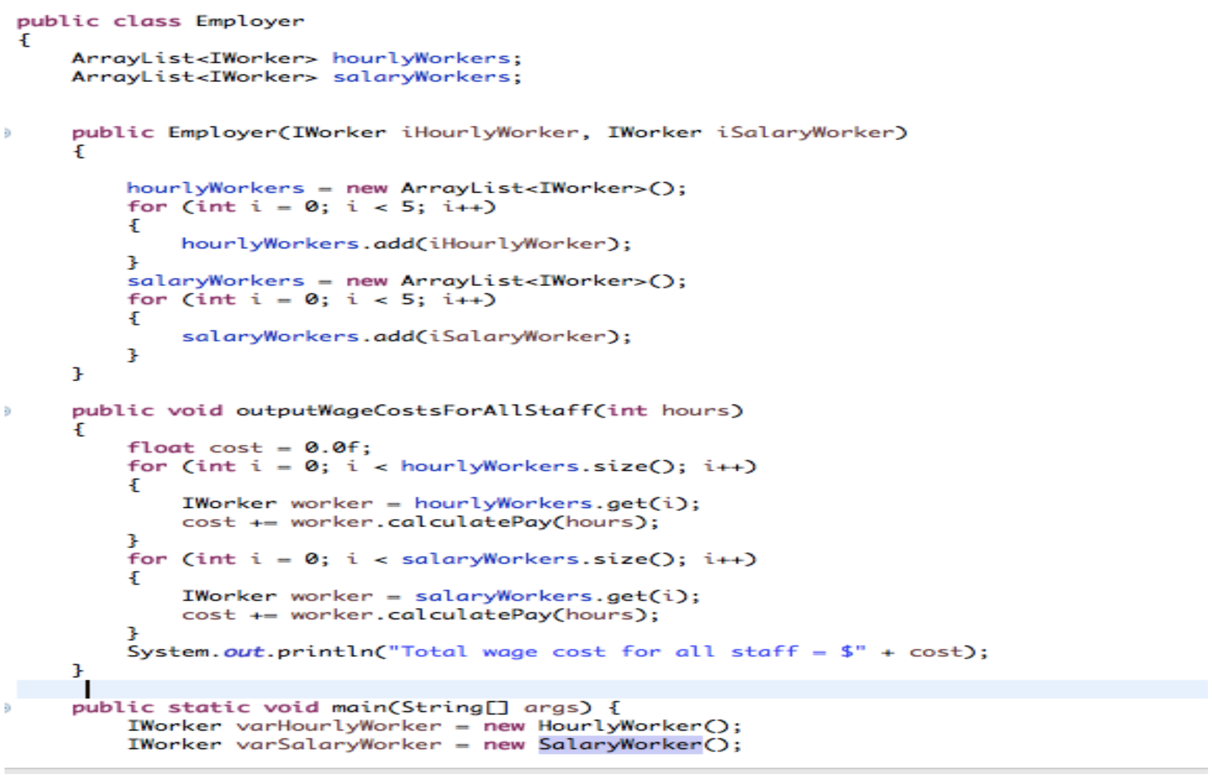
- An interface is created IWorker that establishes a contract on the implementing classes.



```java
public interface IWorker {
    public float calculatePay(int hours);
}
```

- Both the classes HourlyWorker and SalaryWorker implement IWorker interface
- By employing this technique, loose coupling is established.
- Employer class is dependent on abstraction rather than concretion.

- The modified code files are present in Question1 folder



```
public class Employer
{
    ArrayList<IWorker> hourlyWorkers;
    ArrayList<IWorker> salaryWorkers;

    public Employer(IWorker iHourlyWorker, IWorker iSalaryWorker)
    {
        hourlyWorkers = new ArrayList<IWorker>();
        for (int i = 0; i < 5; i++)
        {
            hourlyWorkers.add(iHourlyWorker);
        }
        salaryWorkers = new ArrayList<IWorker>();
        for (int i = 0; i < 5; i++)
        {
            salaryWorkers.add(iSalaryWorker);
        }
    }

    public void outputWageCostsForAllStaff(int hours)
    {
        float cost = 0.0f;
        for (int i = 0; i < hourlyWorkers.size(); i++)
        {
            IWorker worker = hourlyWorkers.get(i);
            cost += worker.calculatePay(hours);
        }
        for (int i = 0; i < salaryWorkers.size(); i++)
        {
            IWorker worker = salaryWorkers.get(i);
            cost += worker.calculatePay(hours);
        }
        System.out.println("Total wage cost for all staff = $" + cost);
    }

    public static void main(String[] args) {
        IWorker varHourlyWorker = new HourlyWorker();
        IWorker varSalaryWorker = new SalaryWorker();
```

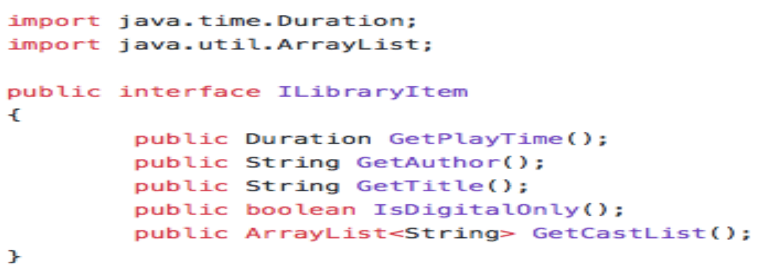
Question 2

1. Which S.O.L.I.D. principle does the following code violate?

The question violates Interface Segregation Principle (ISP). which states to use many client specific interfaces than a general purpose interface.

2. Why does the code violate the principle?

- ILibrayItem interface consists of several methods such as GetPlayTime(), GetAuthor() and IsDigitalOnly() that are common to both DVD and Book classes.



```
import java.time.Duration;
import java.util.ArrayList;

public interface ILibraryItem
{
        public Duration GetPlayTime();
        public String GetAuthor();
        public String GetTitle();
        public boolean IsDigitalOnly();
        public ArrayList<String> GetCastList();
}
```

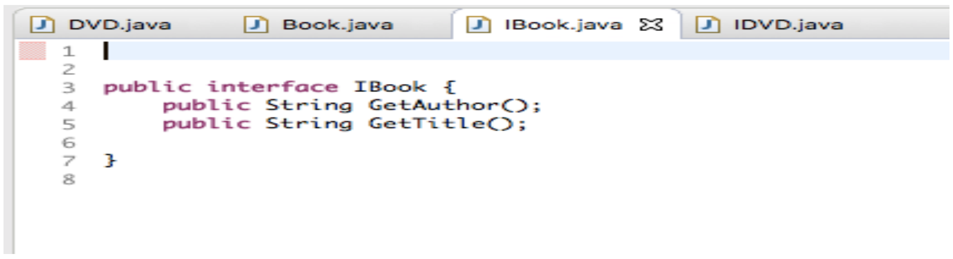
- The common interface forces the implementing classes to define irrelevant methods.

```
public class Book implements ILibraryItem
{
        public Duration GetPlayTime()
        {
                return Duration.ZERO;
        }

        public String GetAuthor()
        {
                return "Hemingway";
        }

        public String GetTitle()
        {
                return "For Whom The Bell Tolls";
        }

        public boolean IsDigitalOnly()
        {
                return false;
        }

        public ArrayList<String> GetCastList()
        {
                return null;
        }
}
```

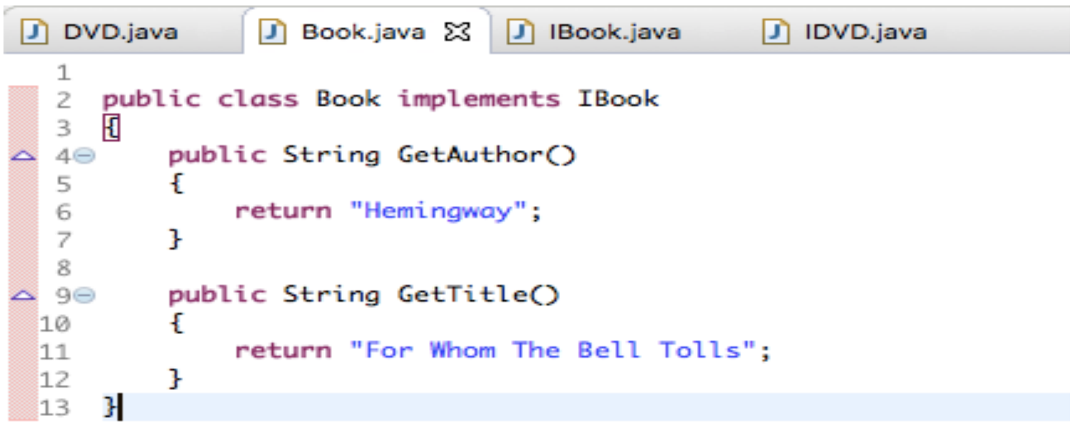
- This is an example of overhead complexity.

3. Write code that fixes the violation.

- Two interfaces namely, IDVD and IBook are created with segregated method implementations.
- IDVD is implemented by DVD class and IBook interface is implemented by Book class.
- The modified code files are present in Question2 folder.



```

public interface IBook {
    public String GetAuthor();
    public String GetTitle();

}

```



```java

public class Book implements IBook
{
    public String GetAuthor()
    {
        return "Hemingway";
    }

    public String GetTitle()
    {
        return "For Whom The Bell Tolls";
    }
}
```

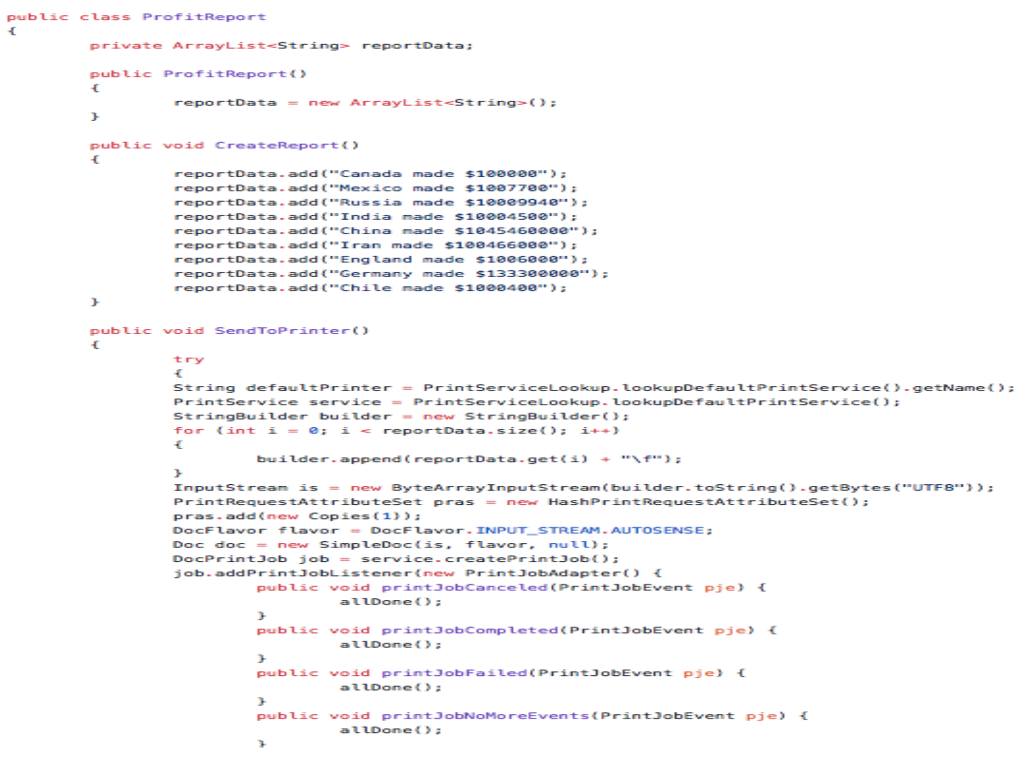
Question 3

1. Which S.O.L.I.D. principle does the Profit Report class violate?

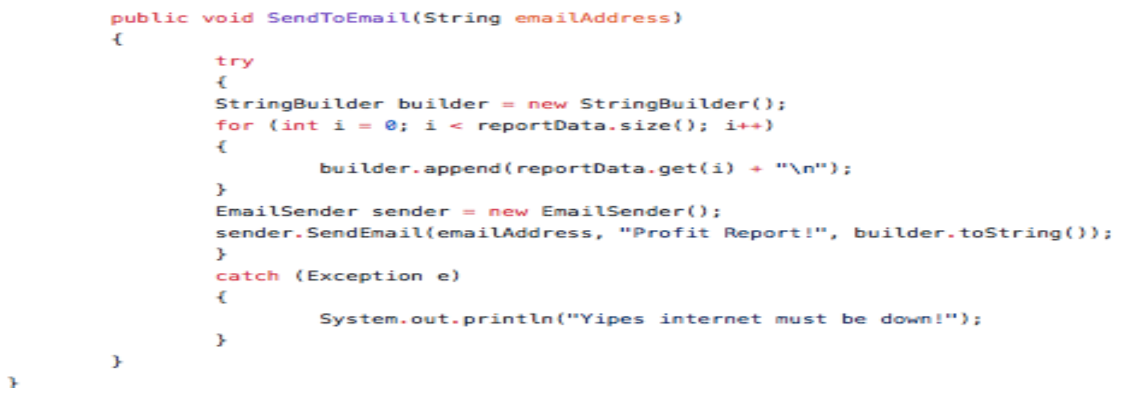
The question violates Single Responsibility Principle (SRP). which states that a class must take responsibility over a single function of the system. A class must be changed for a single reason only.

2. Why does the code violate the principle?

- Profit Report class consists of three major responsibilities such as report generation, printer and email functionalities.



```java
public class ProfitReport
{
        private ArrayList<String> reportData;

        public ProfitReport()
        {
                reportData = new ArrayList<String>();
        }

        public void CreateReport()
        {
                reportData.add("Canada made $100000");
                reportData.add("Mexico made $1007700");
                reportData.add("Russia made $10009940");
                reportData.add("India made $10004500");
                reportData.add("China made $1045460000");
                reportData.add("Iran made $100466000");
                reportData.add("England made $1006000");
                reportData.add("Germany made $133300000");
                reportData.add("Chile made $1000400");
        }

        public void SendToPrinter()
        {
                try
                {
                String defaultPrinter = PrintServiceLookup.lookupDefaultPrintService().getName();
                PrintService service = PrintServiceLookup.lookupDefaultPrintService();
                StringBuilder builder = new StringBuilder();
                for (int i = 0; i < reportData.size(); i++)
                {
                        builder.append(reportData.get(i) + "\f");
                }
                InputStream is = new ByteArrayInputStream(builder.toString().getBytes("UTF8"));
                PrintRequestAttributeSet pras = new HashPrintRequestAttributeSet();
                pras.add(new Copies(1));
                DocFlavor flavor = DocFlavor.INPUT_STREAM.AUTOSENSE;
                Doc doc = new SimpleDoc(is, flavor, null);
                DocPrintJob job = service.createPrintJob();
                job.addPrintJobListener(new PrintJobAdapter() {
                        public void printJobCanceled(PrintJobEvent pje) {
                                allDone();
                        }
                        public void printJobCompleted(PrintJobEvent pje) {
                                allDone();
                        }
                        public void printJobFailed(PrintJobEvent pje) {
                                allDone();
                        }
                        public void printJobNoMoreEvents(PrintJobEvent pje) {
                                allDone();
                        }
```

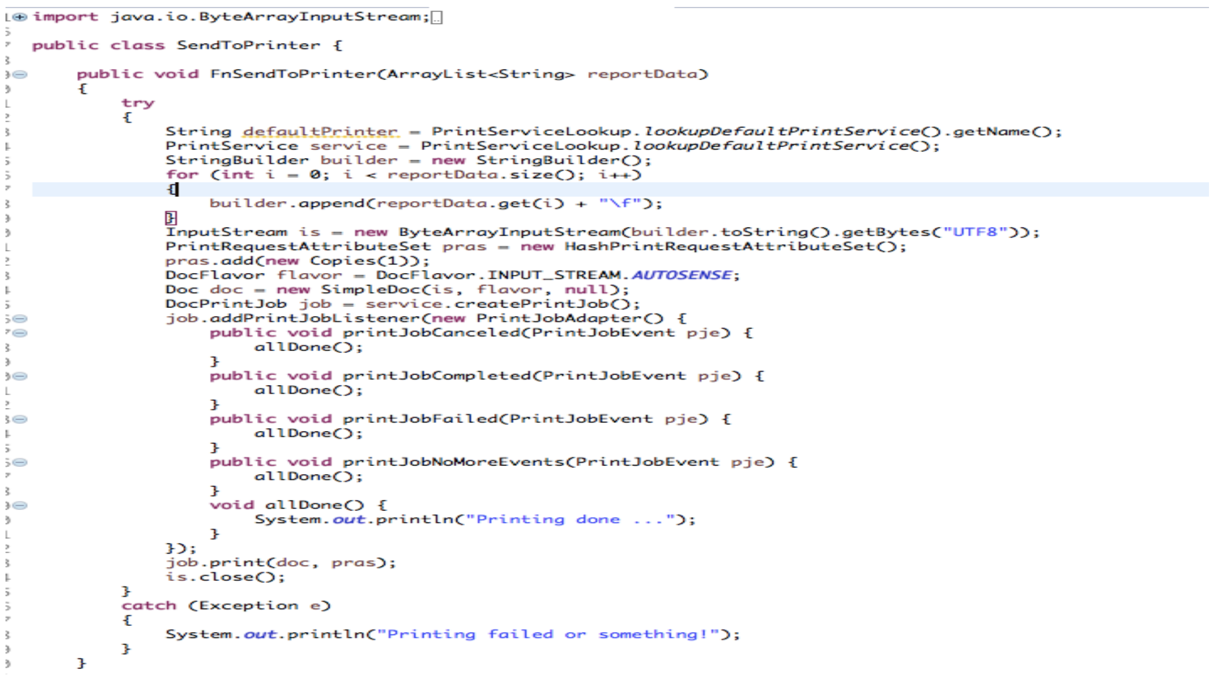


```
        public void SendToEmail(String emailAddress)
        {
                try
                {
                StringBuilder builder = new StringBuilder();
                for (int i = 0; i < reportData.size(); i++)
                {
                        builder.append(reportData.get(i) + "\n");
                }
                EmailSender sender = new EmailSender();
                sender.SendEmail(emailAddress, "Profit Report!", builder.toString());
                }
                catch (Exception e)
                {
                        System.out.println("Yipes internet must be down!");
                }
        }
}
```

- Since the class has three functionalities, any change in these functionalities require to change this base class. This affects other functionalities available in that class.
- In an ideal architecture, a class must change only for one reason.

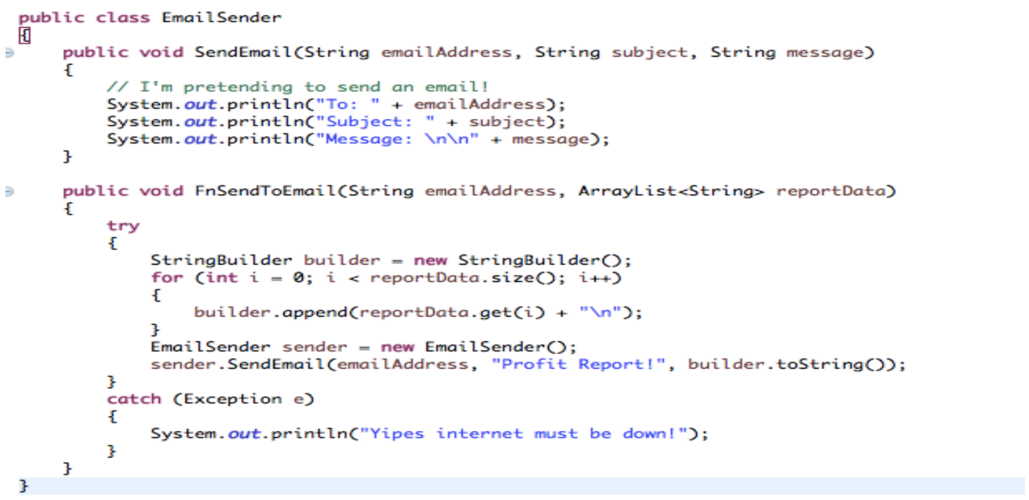
3. Write code that fixes the violation.

- A new classes SendToPrinter is generated.
- SendToPrinter consists of printer functionality which is removed from ProfitReport class. ReportData is sent as parameter to the function.



```
import java.io.ByteArrayInputStream;

public class SendToPrinter {

    public void FnSendToPrinter(ArrayList<String> reportData)
    {
        try
        {
            String defaultPrinter = PrintServiceLookup.lookupDefaultPrintService().getName();
            PrintService service = PrintServiceLookup.lookupDefaultPrintService();
            StringBuilder builder = new StringBuilder();
            for (int i = 0; i < reportData.size(); i++)
            {
                builder.append(reportData.get(i) + "\f");
            }
            InputStream is = new ByteArrayInputStream(builder.toString().getBytes("UTF8"));
            PrintRequestAttributeSet pras = new HashPrintRequestAttributeSet();
            pras.add(new Copies(1));
            DocFlavor flavor = DocFlavor.INPUT_STREAM.AUTOSENSE;
            Doc doc = new SimpleDoc(is, flavor, null);
            DocPrintJob job = service.createPrintJob();
            job.addPrintJobListener(new PrintJobAdapter() {
                public void printJobCanceled(PrintJobEvent pje) {
                    allDone();
                }
                public void printJobCompleted(PrintJobEvent pje) {
                    allDone();
                }
                public void printJobFailed(PrintJobEvent pje) {
                    allDone();
                }
                public void printJobNoMoreEvents(PrintJobEvent pje) {
                    allDone();
                }
                void allDone() {
                    System.out.println("Printing done ...");
                }
            });
            job.print(doc, pras);
            is.close();
        }
        catch (Exception e)
        {
            System.out.println("Printing failed or something!");
        }
    }
```

- Similarly, EmailSender class takes up additional function to send email, that is removed from ProfitReport class. ReportData is sent as parameter to the function.



```java
public class EmailSender
{
    public void SendEmail(String emailAddress, String subject, String message)
    {
        // I'm pretending to send an email!
        System.out.println("To: " + emailAddress);
        System.out.println("Subject: " + subject);
        System.out.println("Message: \n\n" + message);
    }

    public void FnSendToEmail(String emailAddress, ArrayList<String> reportData)
    {
        try
        {
            StringBuilder builder = new StringBuilder();
            for (int i = 0; i < reportData.size(); i++)
            {
                builder.append(reportData.get(i) + "\n");
            }
            EmailSender sender = new EmailSender();
            sender.SendEmail(emailAddress, "Profit Report!", builder.toString());
        }
        catch (Exception e)
        {
            System.out.println("Yipes internet must be down!");
        }
    }
}
```

- ProfitReport class is now responsible for the single responsibility of report generation.
- The modified code files are present in Question3 folder.

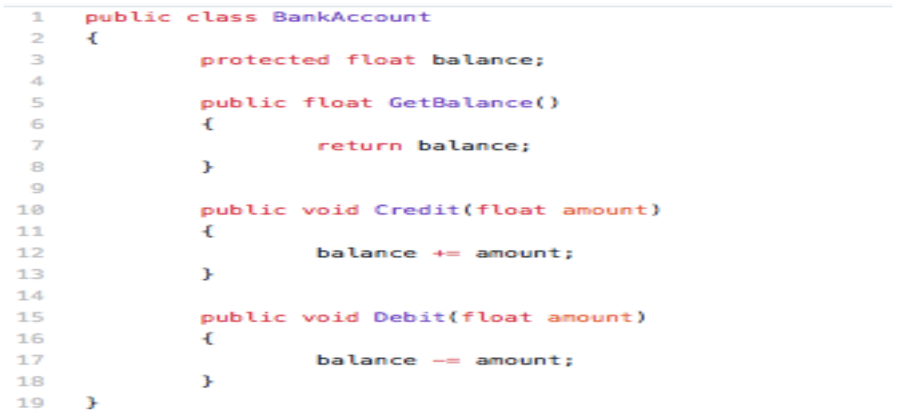
Question 4

1. Which S.O.L.I.D. principle does the following code violate?

The question violates Liskov Substitution Principle (LSP). which states that the child class must not break the parent class's type definition and behaviour.

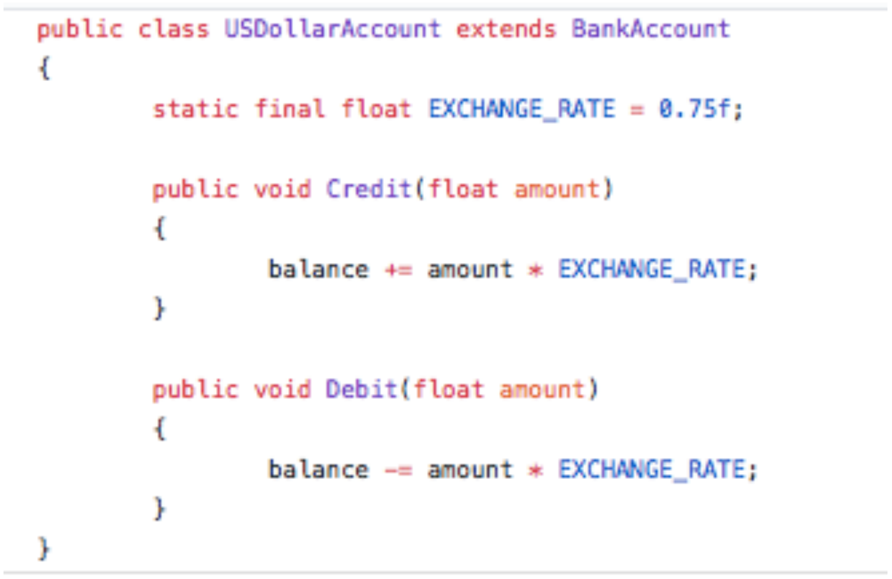
2. Why does the code violate the principle?

- BankAccount class consists of methods such as Credit(), Debit() that perform addition and subtraction functionalitites.



```java
public class BankAccount
{
        protected float balance;

        public float GetBalance()
        {
                return balance;
        }

        public void Credit(float amount)
        {
                balance += amount;
        }

        public void Debit(float amount)
        {
                balance -= amount;
        }
}
```

- USDollarAccount class extends BankAccount class and it changes the Credit() and Debit() methods with additional functionalities..

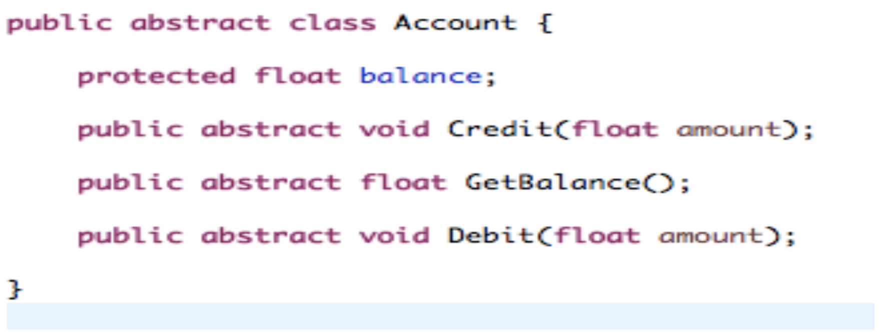


```
public class USDollarAccount extends BankAccount
{
        static final float EXCHANGE_RATE = 0.75f;

        public void Credit(float amount)
        {
                balance += amount * EXCHANGE_RATE;
        }

        public void Debit(float amount)
        {
                balance -= amount * EXCHANGE_RATE;
        }
}
```

- Therefore, it is not possible to substitute the methods of derived class with the base class because of the behavioural change.

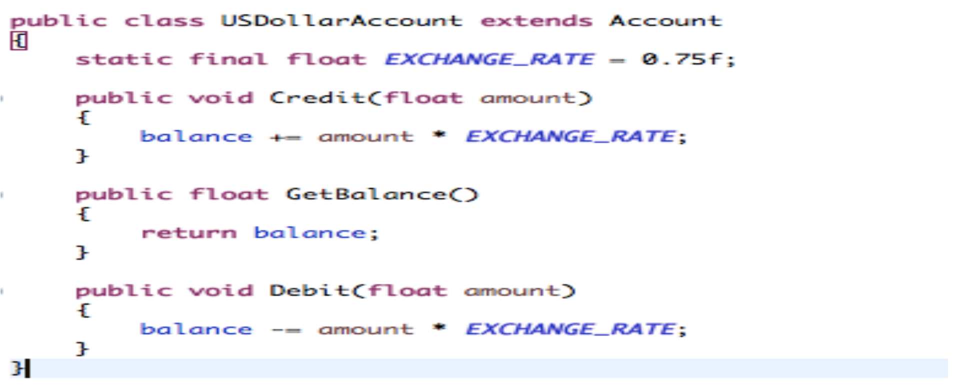
3. Write code that fixes the violation.

- Though interfaces could be used as solution, it was intended to preserve inheritance principle. Therefore, a new abstract class Account is created with all the abstract methods.



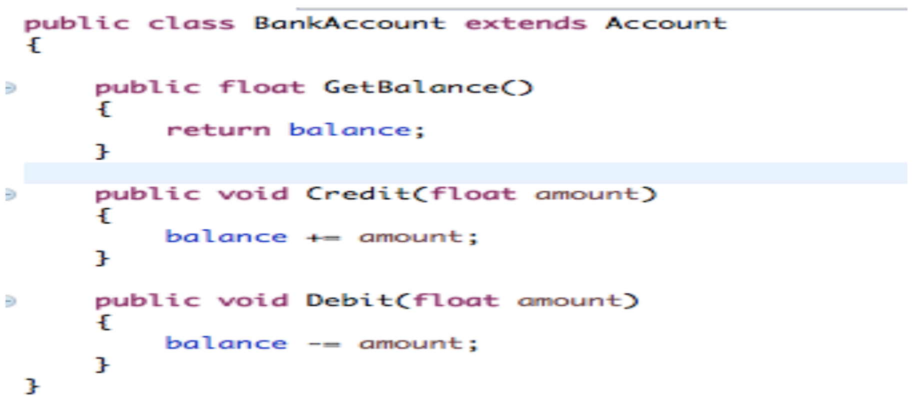
```
public abstract class Account {

    protected float balance;

    public abstract void Credit(float amount);

    public abstract float GetBalance();

    public abstract void Debit(float amount);

}
```

- USDollarAccount extends this abstract class and define their corresponding methods.



```
public class USDollarAccount extends Account
{
    static final float EXCHANGE_RATE = 0.75f;

    public void Credit(float amount)
    {
        balance += amount * EXCHANGE_RATE;
    }

    public float GetBalance()
    {
        return balance;
    }

    public void Debit(float amount)
    {
        balance -= amount * EXCHANGE_RATE;
    }
}
```

- Similarly, BankAccount extends this abstract class and define their corresponding methods.



```
public class BankAccount extends Account
{
    public float GetBalance()
    {
        return balance;
    }

    public void Credit(float amount)
    {
        balance += amount;
    }

    public void Debit(float amount)
    {
        balance -= amount;
    }
}
```

- By this way, loose coupling is established.
- The modified code files are present in Question4 folder.

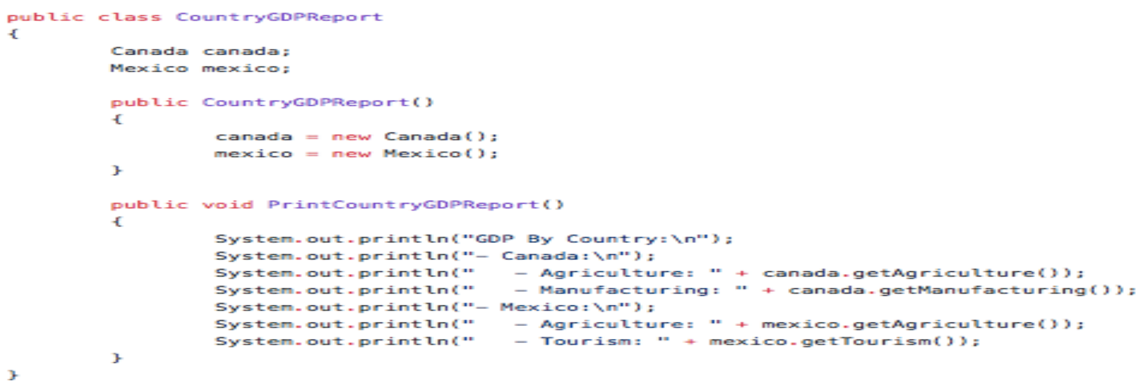
Question 5

1. Which S.O.L.I.D. principle does the following code violate?

The Employer Class violates Open Closed Principle (OCP) which states that a class must be closed for modification and open for extension.

2. Why does the code violate the principle?

- CountryGDPReport class generates GDP report for countries. In the case of adding more countries, the class needs to modified which is against OCP.

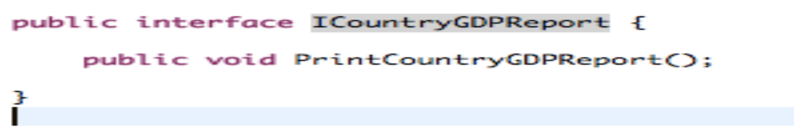


```
public class CountryGDPReport
{
        Canada canada;
        Mexico mexico;

        public CountryGDPReport()
        {
                canada = new Canada();
                mexico = new Mexico();
        }

        public void PrintCountryGDPReport()
        {
                System.out.println("GDP By Country:\n");
                System.out.println("- Canada:\n");
                System.out.println("    - Agriculture: " + canada.getAgriculture());
                System.out.println("    - Manufacturing: " + canada.getManufacturing());
                System.out.println("- Mexico:\n");
                System.out.println("    - Agriculture: " + mexico.getAgriculture());
                System.out.println("    - Tourism: " + mexico.getTourism());
        }
}
```

- The classes are vulnerable to code change complexities.

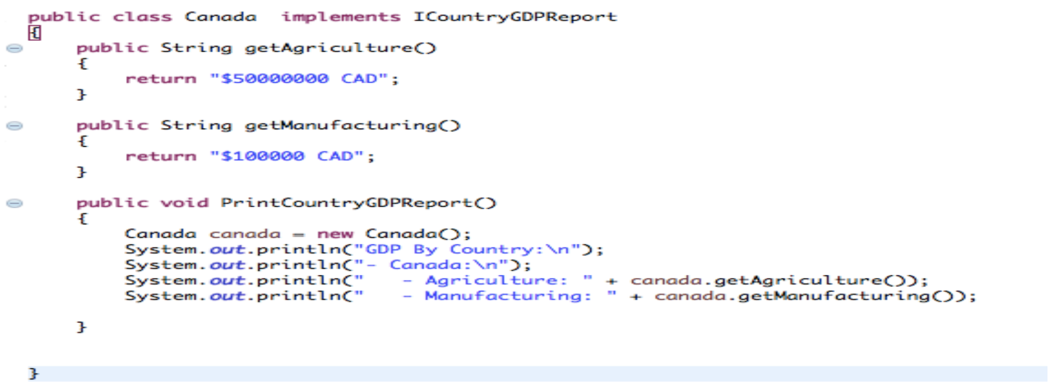
3. Write code that fixes the violation.

- A new interface ICountryGDPReport is created.

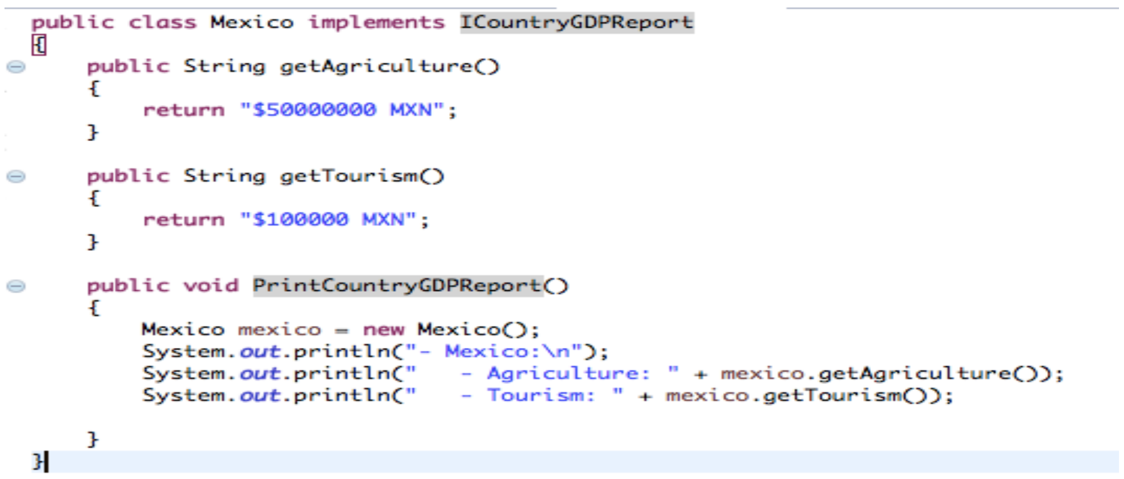


```
public interface ICountryGDPReport {
    public void PrintCountryGDPReport();
}
```

- The interfaces are implemented by the respective classes.



```
public class Canada  implements ICountryGDPReport
{
    public String getAgriculture()
    {
        return "$50000000 CAD";
    }

    public String getManufacturing()
    {
        return "$100000 CAD";
    }

    public void PrintCountryGDPReport()
    {
        Canada canada = new Canada();
        System.out.println("GDP By Country:\n");
        System.out.println("- Canada:\n");
        System.out.println("    - Agriculture: " + canada.getAgriculture());
        System.out.println("    - Manufacturing: " + canada.getManufacturing());

    }

}
```



```
public class Mexico implements ICountryGDPReport
{
    public String getAgriculture()
    {
        return "$50000000 MXN";
    }

    public String getTourism()
    {
        return "$100000 MXN";
    }

    public void PrintCountryGDPReport()
    {
        Mexico mexico = new Mexico();
        System.out.println("- Mexico:\n");
        System.out.println("    - Agriculture: " + mexico.getAgriculture());
        System.out.println("    - Tourism: " + mexico.getTourism());

    }
}
```

- By employing this technique, loose coupling is established.
- Employer class is dependent on abstraction rather than concretion.
- The modified code files are present in Question5 folder

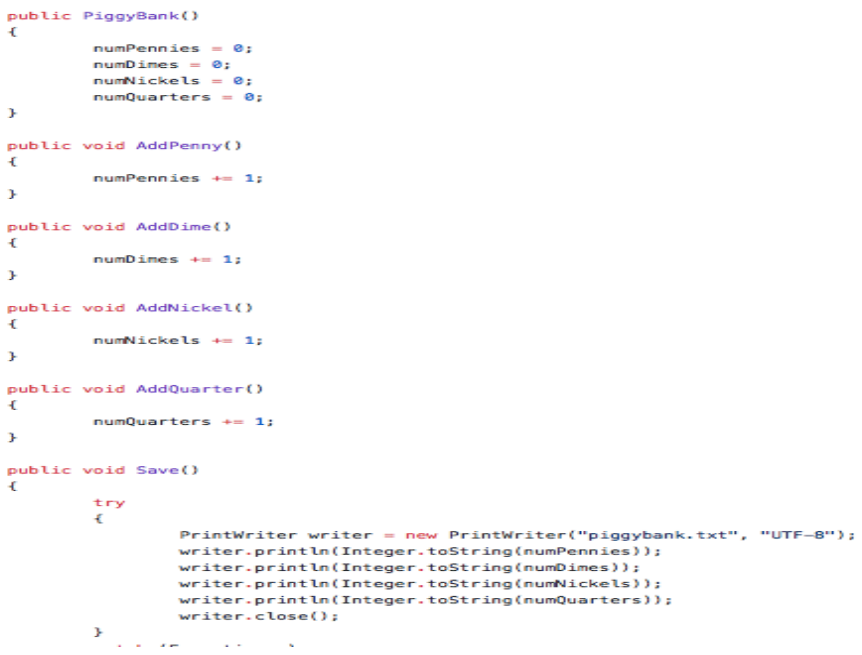
Question 6

1. Which S.O.L.I.D. principle does the Piggybank class violate?

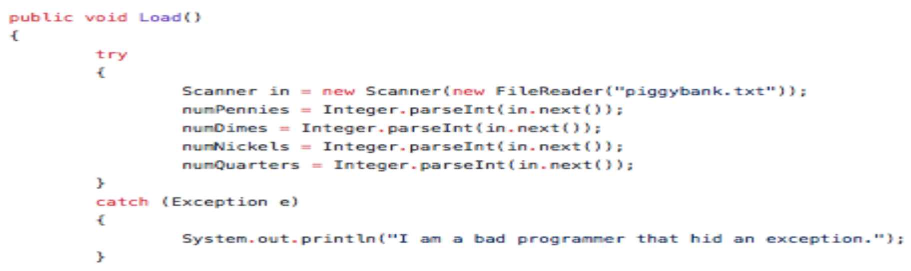
The question violates Single Responsibility Principle (SRP). which states that a class must take responsibility over a single function of the system. A class must be changed for a single reason only.

2. Why does the code violate the principle?

- Piggybank class consists of three major responsibilities such as loading, printing and adding functionalities.



```
public PiggyBank()
{
        numPennies = 0;
        numDimes = 0;
        numNickels = 0;
        numQuarters = 0;
}

public void AddPenny()
{
        numPennies += 1;
}

public void AddDime()
{
        numDimes += 1;
}

public void AddNickel()
{
        numNickels += 1;
}

public void AddQuarter()
{
        numQuarters += 1;
}

public void Save()
{
        try
        {
                PrintWriter writer = new PrintWriter("piggybank.txt", "UTF-8");
                writer.println(Integer.toString(numPennies));
                writer.println(Integer.toString(numDimes));
                writer.println(Integer.toString(numNickels));
                writer.println(Integer.toString(numQuarters));
                writer.close();
        }
```

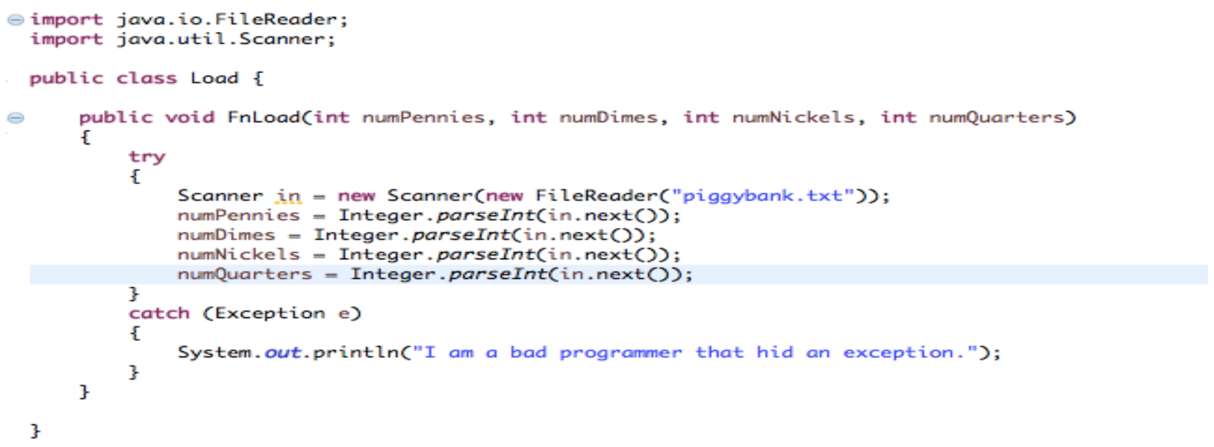


```
public void Load()
{
        try
        {
                Scanner in = new Scanner(new FileReader("piggybank.txt"));
                numPennies = Integer.parseInt(in.next());
                numDimes = Integer.parseInt(in.next());
                numNickels = Integer.parseInt(in.next());
                numQuarters = Integer.parseInt(in.next());
        }
        catch (Exception e)
        {
                System.out.println("I am a bad programmer that hid an exception.");
        }
```

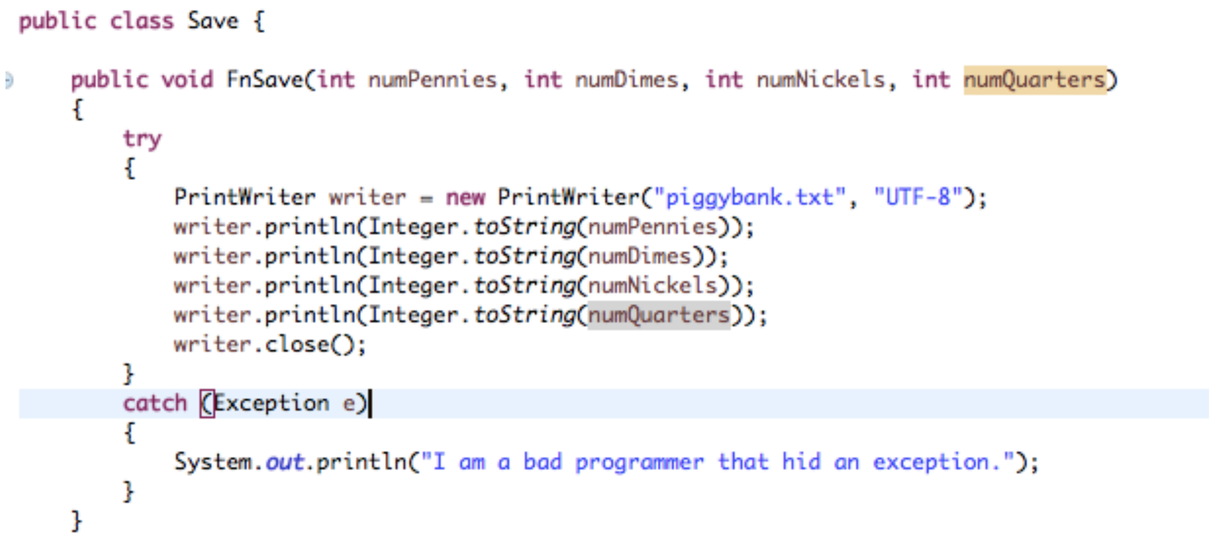
- Since the class has three functionalities, any change in these functionalities require to change this base class. This affects other functionalities available in that class.
- In an ideal architecture, a class must change only for one reason.

3. Write code that fixes the violation.

- Two new classes Load and Save are generated.
- Load class consists of loading functionality while Save class consists of saving functionality.



```java
import java.io.FileReader;
import java.util.Scanner;

public class Load {

    public void FnLoad(int numPennies, int numDimes, int numNickels, int numQuarters)
    {
        try
        {
            Scanner in = new Scanner(new FileReader("piggybank.txt"));
            numPennies = Integer.parseInt(in.next());
            numDimes = Integer.parseInt(in.next());
            numNickels = Integer.parseInt(in.next());
            numQuarters = Integer.parseInt(in.next());
        }
        catch (Exception e)
        {
            System.out.println("I am a bad programmer that hid an exception.");
        }
    }

}
```



```java
public class Save {

    public void FnSave(int numPennies, int numDimes, int numNickels, int numQuarters)
    {
        try
        {
            PrintWriter writer = new PrintWriter("piggybank.txt", "UTF-8");
            writer.println(Integer.toString(numPennies));
            writer.println(Integer.toString(numDimes));
            writer.println(Integer.toString(numNickels));
            writer.println(Integer.toString(numQuarters));
            writer.close();
        }
        catch (Exception e)
        {
            System.out.println("I am a bad programmer that hid an exception.");
        }
    }
```

- Piggybank class is now responsible for the single responsibility of report generation.
- The modified code files are present in Question6 folder.

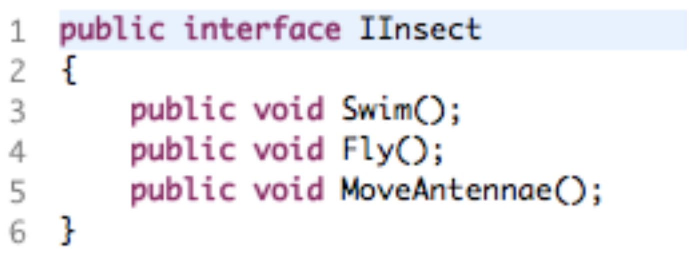
Question 7

1. Which S.O.L.I.D. principle does the following code violate?

The question violates Interface Segregation Principle (ISP). which states to use many client specific interfaces than a general purpose interface.

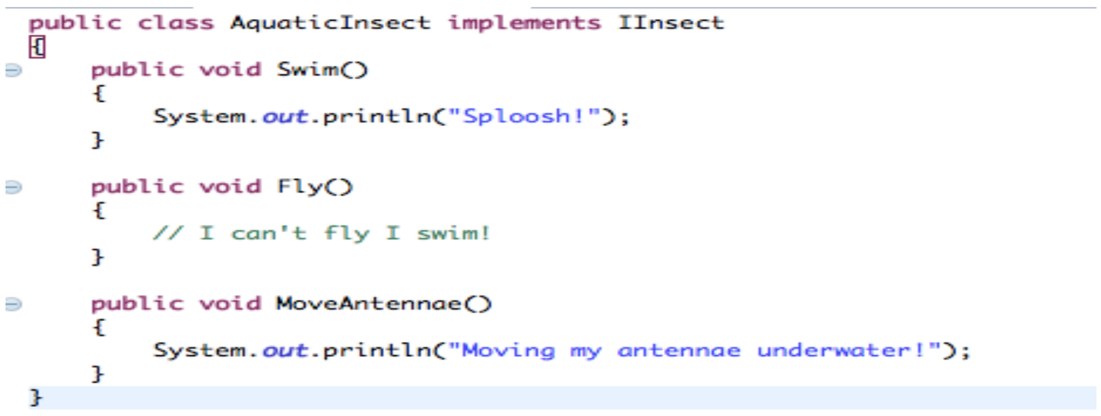
2. Why does the code violate the principle?

- Interface IInsect class consists of methods such as Swim(), Fly() and MoveAntennae().



```
public interface IInsect
{
    public void Swim();
    public void Fly();
    public void MoveAntennae();
}
```

- AquaticInsect class has implementations for all interface methods except one.



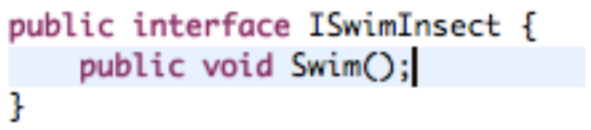
```
public class AquaticInsect implements IInsect
{
    public void Swim()
    {
        System.out.println("Sploosh!");
    }

    public void Fly()
    {
        // I can't fly I swim!
    }

    public void MoveAntennae()
    {
        System.out.println("Moving my antennae underwater!");
    }
}
```

.

- The Fly() method in AquaticInsect class does not have a proper definition. The common interface forces the implementing classes to define irrelevant methods
- Therefore, this case violates Interface Segregation Principle.

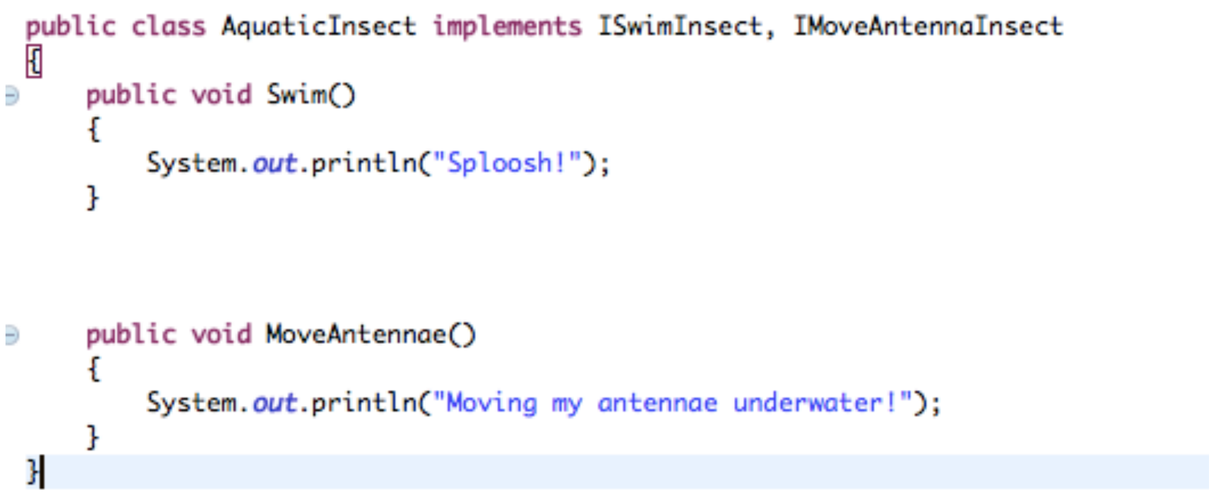
3. Write code that fixes the violation.

- This violation could be solved by multiple interfaces.
- Three interfaces IFlyInsect, ISwimInsect and IMoveAntennaInsect are created with appropriate method contracts.



```
public interface ISwimInsect {
    public void Swim();
}
```

- Since AquaticInsect class has proper implementations for Swim() and MoveAntenna() methods, it implements ISwimInsect and IMoveAntennaInsect interfaces.



```java
public class AquaticInsect implements ISwimInsect, IMoveAntennaInsect
{
    public void Swim()
    {
        System.out.println("Sploosh!");
    }


    public void MoveAntennae()
    {
        System.out.println("Moving my antennae underwater!");
    }
}
```

- The modified code files are present in Question7 folder.