

Single Responsibility Principle

A class should have only one reason to change

```
class Invoice {  
    private Marker marker; ✓  
    private int quantity;  
  
    public Invoice(Marker marker, int quantity) {  
        this.marker = marker;  
        this.quantity = quantity;  
    }  
  
    public int calculateTotal() {  
        int price = ((marker.price) * this.quantity);  
        return price;  
    }  
  
    public void printInvoice() {  
        //print the Invoice  
    }  
  
    public void saveToDB() {  
        // Save the data into DB  
    }  
}
```

Handwritten annotations: A circled '1' with a bracket next to the `calculateTotal()` method. A circled '2' next to the `printInvoice()` method. A circled '3' next to the `saveToDB()` method.

```
class Invoice {  
    private Marker marker;  
    private int quantity;  
  
    public Invoice(Marker marker, int quantity) {  
        this.marker = marker;  
        this.quantity = quantity;  
    }  
  
    public int calculateTotal() {  
        int price = ((marker.price) * this.quantity);  
        return price;  
    }  
}
```

```
class InvoicePrinter {  
    private Invoice invoice;  
  
    public InvoicePrinter(Invoice invoice) {  
        this.invoice = invoice;  
    }  
  
    public void print() {  
        //print the invoice  
    }  
}
```

```
class InvoiceDao {  
    Invoice invoice;  
  
    public InvoiceDao(Invoice invoice) {  
        this.invoice = invoice;  
    }  
  
    public void saveToDB() {  
        // Save into the DB  
    }  
}
```

Open Close Principle


Open for modification close for extension

```
class InvoiceDao {
    Invoice invoice;

    public InvoiceDao(Invoice invoice) {
        this.invoice = invoice;
    }

    public void saveToDB() {
        // Save Invoice into DB
    }

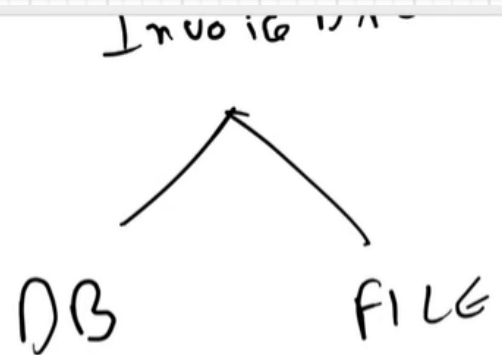
    public void saveToFile(String filename) {
        // Save Invoice in the File with the given name
    }
}
```



```
interface InvoiceDao {
    public void save(Invoice invoice);
}

class DatabaseInvoiceDao implements InvoiceDao {
    @Override
    public void save(Invoice invoice) {
        // Save to DB
    }
}

class FileInvoiceDao implements InvoiceDao {
    @Override
    public void save(Invoice invoice) {
        // Save to file
    }
}
```



Liskov Substitution Principle

If class B is a subtype of class A, then we should be able to replace object A with B without breaking the behavior of the program. Subclass should extend the capability of the parent class not narrow it down.

Interface Segregation Principle

Several specific client interfaces are better than having one generic client interface

```
private Interface IMachine{  
    public task A();  
    public task B();  
    public task C();  
    public task D();  
}
```

```
// segregating the interfaces  
private Interface IGenericFeature{  
    public task A();  
    public task B();  
    public task C();  
}  
  
private Interface IAdvancedFeature{  
    public task D();  
}
```

```
private class MachineA implements IMachine{  
    public task A() {};  
    public task B() {};  
    public task C() {};  
    public task D() {};  
}  
  
// we are forcing all the classes to implement all the methods  
private class MachineB implements IMachine{  
    public task A() {};  
    public task B() {};  
    public task C() {};  
    public task D() {};// cannot support this method  
}
```

Dependency Inversion Principle

High level modules should not depend on low level modules, it should depend on abstraction (interfaces rather than concrete classes)

```
public class Project {
    private BackendDev backendDev = new BackendDev();
    private FrontendDev frontendDev = new FrontEndDev();

    public void implement() {
        backendDev.writeInJava();
        frontendDev.writeInJS();
    }
}
```

```
public class BackendDev {
    public void writeInJava() {
    }
}
```

```
public class FrontendDev {
    public void writeInJS() {
    }
}
```

```
public class Project {
    private Developer developer;

    public Project( Developer developer) {
        this.developer=developer;
    }

    private void implement() {
    }
}
```

```
private Interface Developer{
    public void develop();
}
```

```
public class BackendDev implements Developer {
    @Override
    public void develop() {
        writeInJava();
    }

    private void writeInJava() {
    }
}
```

```
public class FrontendDev implements Developer {
    @Override
    public void develop() {
        writeInJS();
    }

    private void writeInJS() {
    }
}
```

Here Project class can take in backend dev or frontend dev.