Contents

1 SOLID

Playlist The term SOLID is an acronym for five design principles intended to make software designs more understandable and maintainable. These 5 are just subset of the many principles promoted by Robert C. Martin.

1.1 Single Responsibility Principle

Robert C. Martin expresses this principle as "A class should have only one reason to change."

It means, Every module or class should be responsible over a single part of the functionality provided by the software and that responsibility should be entirely encapsulated by the class.

1.2 Liskov Substitution Principle

Introduced by Barbara Liskov. It states that "Objects in a program should be replacable with instances of their subtypes without altering the correctness of the program."

It means,

- If a program is using a Base class, then the reference to the base class(the object variable) can be replaced with a derived class without affecting the functionality of the program module.

1.3 Open/Close Priciple

It states "Software entities should be open for extension and closed for modification". It means,

- The design of the software should be done in a way that adding new functionality will result in minimum modification of existing code.

1.4 Interface Segregation Principle

It states, "Many client-specific interfaces are better than one general-purpose interface".

It means,

- We should not enforce clients to implement interfaces that they don't use. Instead of creating one big interface, we can break it down to smaller interfaces.

1.5 Dependency Inversion Principle

Abstractions should not depend on details whereas the details should depend on abstraction. It means,

- High level modules should not depend on low level modules.

1.6 If we don't follow SOLID principles

- We may end up with `tight coupling` of the code with many modules in our application.
- Tight coupling causes adding new feature or fixing bug time consuming and more costly. It can even introduce new bugs and unknown issues. The more you interact an existing code, the more likely it is for a new unknown issue to pop up.
- Duplication of code.
- Create new bygs when solving a bug.

1.7 If we follow SOLID principles,

- Achieve reduction of complexity of code.
- Increase readibiliy
- Extensibility
- Maintainance
- Reduce errors
- Reduce Tight Coupling

2 Single Responsibility Principle

A class should have only one reason to change.

- Every class should have responsibility over one part of the functionalities performed by the software and it should encapsulate that part.
- In SRP, `classes become smaller and cleaner.`

2.1 Why use it?

- Maintainability, documentation
- Testability
- Update
- Fix bugs
- Understanding
- `Parallel development`

2.2 Example

Login and Registration and related errors

```
interface IUSer{
    bool Login(Srting username, String password);
    bool Register(Srting username, String password, String email);
    void LogError(String error);
    bool SendEmail(String emailContent);
}
```

It seems like a clean and perfect piece of code, right? Actually no. You see, a USER object should not have BOTH login and register or sendEmail.

```
interface IRegister{
    bool Register(Srting username, String password, String email);
}
interface ILogin {
    bool Login(Srting username, String password);
}
interface ILogger{
    void LogError(String error);
}
interface IVerification {
    bool SendEmail(String emailContent);
}


interface IUSer implement ILogin, IRegister, ILogger, IVerification{
}
```

3 Interface Segregation Principle

"No client should be forced to depend on methods it does not use"

It means, we should split a big interface into multiple small interface.

3.1 History

The Xerox company created a new printer system that could perform a variety of tasks such as stapling, faxing along with the regular printing tasks.

But soon, they encountered trouble. There was a lot of overhead and the TIME and cost of maintainance become very high. They wondered why.

The problem was - they had one big Job interface that would implement print job, stapler job, fax job etc. The stapler job do not need to know about fax job so all of these `overhead` was unnecessary, Not to mention bug fixing became hard.

So then, Robert C Martin proposed to split that big interface into multiple small interface and came up with this principle - Interface Segregation Principle.

`In the example from SRP,` we have already implemented ISP princniple! Because we have split one big interface into multiple small interface.

3.2 Example

```
interface IPrintTasks{
  bool printContent(String content);
  bool faxContent(String content);
  bool photocopyContent(String content);
  bool staplerContent(String content);
}

class AnaloguePrinter implements IPrintTasks{}
```

Analogue printer do not need all of these methods because it can only print but with this structure, its forced to print all of these.

There is also another scenario. Like if we try to add new feature for our new printer machine, all old machine also have to implement that!

```
interface IPrintTasks{
  bool printContent(String content);
  bool faxContent(String content);
  bool photocopyContent(String content);
```

```
  bool staplerContent(String content);

  bool someNewFeature(String content);
}

class AnaloguePrinter implements IPrintTasks{}
```

So instead, we should do this.

```
interface IPrintContent{
    bool printContent(String content);
}

interface IFaxContent{
    bool faxContent(String content);
}
interface IPhotocopyContent{
    bool photocopyContent(String content);
}

interface IStaplerContent{
    bool staplerContent(String content);
}

interface ISomeNewFeature{
  bool someNewFeature(String content);
}
```

4 Open Closed Principle

Software entities such as classes, modules, functions etc should be open for extension but closed for modification.

Its considered as the `most important principle` by Robert C. Martin. `Making often-updated features abstract do the trick!`

- Any new feature should be implemented by adding new classes, methods and attributes instead of changing existing ones.

4.1 How to implement new feature

- Create a new class that inherits from old class.

- Use abstract interface to access original class.

4.2 Example

```
class Employee {
  Id, Name, salary;
  int getBonus(){
    return salary*10%;
  }
}
```

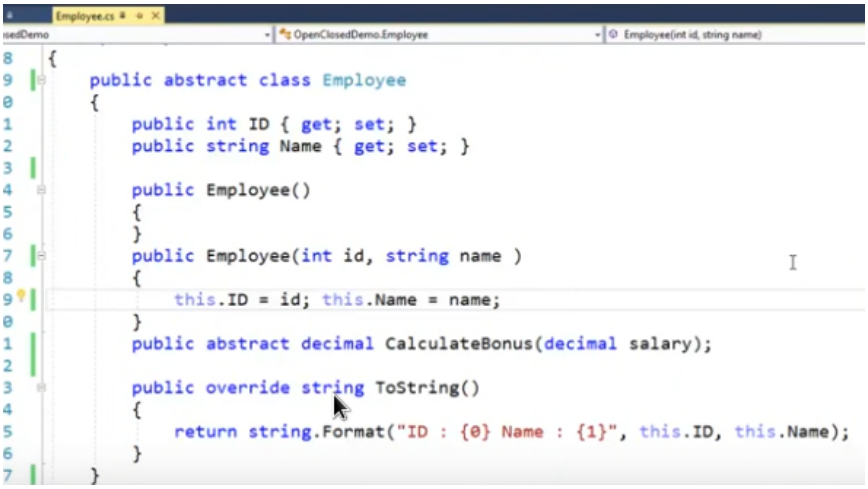
Now lets say, our company has interns or temrporary workers as well. We wish to give them 5% bonus while keeping the bonus for permanent workers same(10%)

```
class Employee {
  Id, Name, salary;
  int getBonus(String emp_type){
    if(emp_type=="Permanent")
    return salary*10%;
    else return salary*5%;
  }
}
```

Now, to reflect that update

- Add emp_{type} parameter in every module this function is used. Very costly and hard to test!
- Edit this code segment (which is ok)

Instead, if we had made this function abstract, then we would not need to do all this.



5 Liskov's Substitution Principle

- Derieved types must be completely substitutable with base type.
- Introduced by Barbara Liskov.
- Polymorphic variables are lowkey LSP's example. Lowkey because the actual subclasses may contain new exceptions.

Its an extension of open close principle.

5.1 Implementation guideline

- No new exceptions can be thrown by the subtype
- New derived classes just extend `without replacing old funcionality of the class.`
- Clients should know which specific subtype they are calling.

Tips: Use interface to spearate the feature that may generate exception.

5.2 Example

Lets add a new type of employee. They get no bonus.

```
abstract class Employee{
    String name;
   String salary;
   String id;

   abstract double getBonus();
    abstract public GetMinimumSalary();
 }

 class PermanentEmployee extends Employee{
   double getBonus(){
     return super.getSalary()*10%;
   }
 }

 class InternEmployee extends Employee {
   double getBonus(){
     return super.getSalary()*5%;
   }
 }

 class ContractedEmployee extends Employee {
   double getBonus(){
     throw new NoBonusException;
   }
 }

 public class Main{
   public static void main(String[] args){
     Employee em1 = new InternEmployee();
     Employee em1 = new ContractedEmployee();

     em1.getBonus();
     em2.getBonus();
   }
 }
```

The above code will generate exception at `emp2.getBonus()` and even when doing `Employee emp2 = new ContractedEmployee()`. It hppenned because we have added a new exception in one of the subclass. We can try manually

adding try-catch but its not feasible if we have used this class a lot in our modules. And thus, we need a new design.

```
interface IEmployee{
   abstract public GetMinimumSalary();
}
interface IBonus{
  double getBonus();
}

abstract class Employee implements IEmployee{
  String name;
  String salary;
  String id;
}

class PermanentEmployee extends Employee, IBonus{
  double getBonus(){
    return super.getSalary()*10%;
  }
}

class InternEmployee extends Employee {
  double getBonus(){
    return super.getSalary()*5%;
  }
}

class ContractedEmployee extends Employee {
  double getBonus(){
    throw new NoBonusException;
  }
}

public class Main{
  public static void main(String[] args){
    Employee em1 = new InternEmployee();
    Employee em1 = new ContractedEmployee();

  }
```

```
}
```

The above code will work flawlessly and people will be notified when they try to use getSalary() method on ContractedEmployee.

6 Dependency Inversion Principle

Video

- High level modules should not depend on low level modules. Both should depend on abstractions.
- Abstrations should not depend on details, rather details should depend on abstractions.

Our store app, a high level module uses the low level module PaypalAPI for money exchange. Now if we wish to add new payment methods, we have to change the higher level module. Not to mention, update in lower level module PaypalAPI will also affect our coupled Store Module. But most importantly, `We can not use this package kinda independently elsewhere` as it will always have carry around the low level module.

Due to all of these problems in maintainance, we add a interface or class/processor in-between the high level and low level class as a wrapper.

Store——Paypal Store——Paypal, SonaliBank??? Store—PaymentProcessor—-paypal,SonaliBank

6.1 Example

The original problem. As you can see, to add new payment method, we have to maually edit all the `buy_rice, buy_cloth` methods.

```
class PaypalAPI {
  bool makePayment(int amount){//blah}
}

public class Store{
    PaypalAPI paypal = new PaypalAPI();
    bool buyRice(){
paypal.makePayment(500);
    }
    bool buyCloth(String transaction_method){
```

```
      if(transaction_method=="Paypal") paypal.makePayment(500);
      else if(transaction_method=="SonaliBank") sb.makePayment(500);
    }
}
```

We make the payment processor between high level module and low level module to make high level module independent of low level module and rather, dependent on abstraction.

```
abstract class PaymentProcessor {
  int amount;
  double transaction_percentage;
  double tax=0.2;
  PaymentProcessor(int amount){
    this.amount=amount;
  }
  abstract bool makePayment(int amount);
}

class Paypal extends PaymentProcessor {
  super.transaction_percentage=0.5;

  bool makePaymet(int amount){
    amount-=amount+amount*transaction_percetage;
  }
}

class Paypal extends PaymentProcessor {
  super.transaction_percentage=0.6;

  bool makePaymet(int amount){
    amount-=amount+amount*transaction_percetage;
  }
}


public class Store {
  PaymentProcessor paymentProcessor = new PaymentPRocessor();
  bool buy_bike(int amount){
    logger(date);
    paymentProcessor.makePayment(100);
```

```
  }
  bool buy_Rice(int amount){
    logger(date);
    paymentProcessor.makePayment(100);
  }

}
```

7 Modularity & Coupling & Cohesion

Video

7.1 Modularity

Modularity is the principle of keeping separate the various unrelated aspects of a system, so that each aspect can be studied in isolation.

For example, the person who is making thr GUI do not need to know whether its SQL server or what the connectDB() method is doing. He only needs to know that it conects database to the gui.

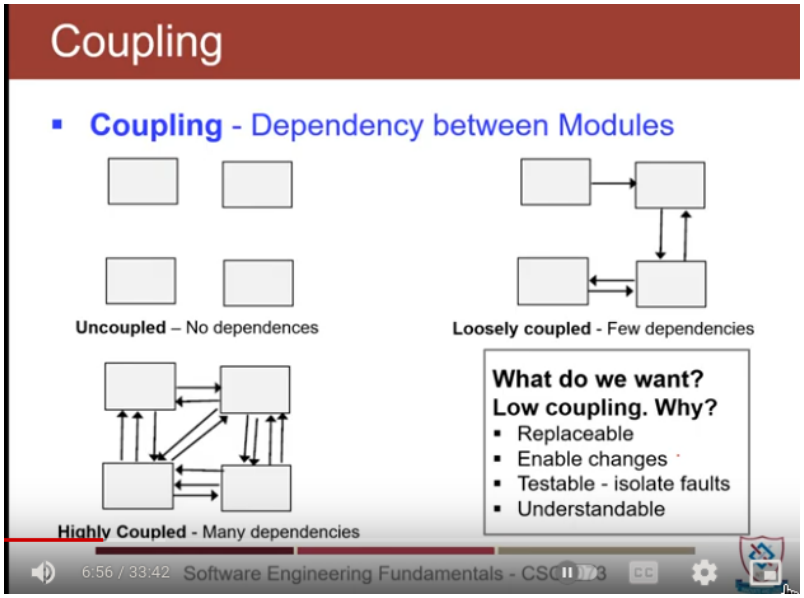
If we use modularity in our software design,we can:

- A person only eeds to understand his module to start working. Thus, it makes it simple for a team to work.
- Easier to locate faults.
- Easier to add new features.

7.2 Coupling

Couping is the dependency between modules.

1. Its impossible to reduce coupling count to zero. Because at least one module must use all the other modules. Like MAIN modules should use GUI, DB, NETWORK modules and thus, the coupling is 3. So zero coupling is impossible.
2. Loosely coupled: Each module has few dependency modules.
3. Highly Coupled: Each module uses Many many modules .



In the highly coupled picture, if we were to edit, update or understand the third Module, we have to change or understand all the other modules! Thus, highly coupled structure makes maintainence tedious.

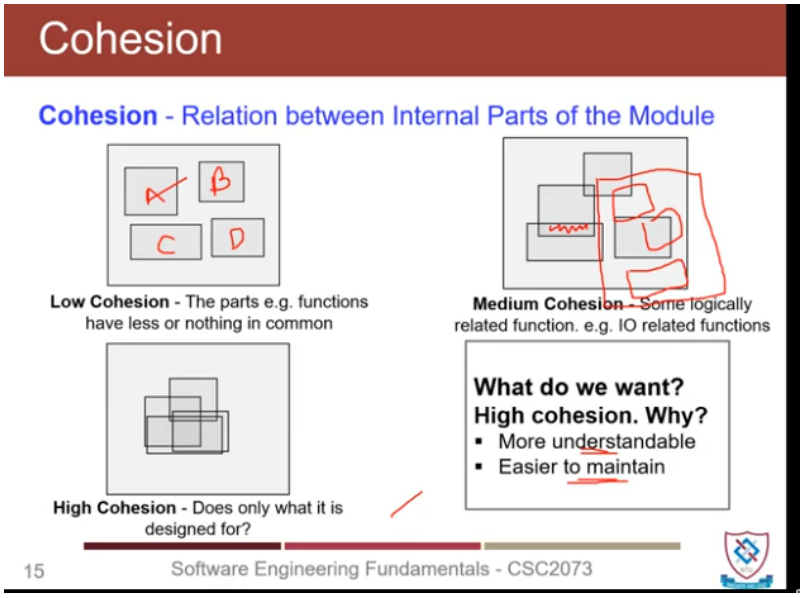
7.2.1 Why low coupling is beneficial

1. Easier to maintain.
2. Easier to resolve bug.
3. Easier to introduce new feature.
4. Easy to understand for new workers.
5. `Testable` because we can isolate faults.

7.3 Cohesion

Relation between internal parts of a module.

1. Low Cohesion: Some functions that don'e use each other.
2. Medium Cohesion: Some function use each other, some remain independent.
3. High Cohesion: Every function use each other and thus, better.



If we have student class, then it should only contain student information. It should not contain teacher's information. It reduces cohesion because the teacher-related functions won't be used by most functions of that module. And most importantly, Single responsibility principle states that a class should perform only one responsibility. If a class has a low cohesive function such as teacher's information, its likely that SRP is not being followed.

So in a good design, there should be high cohesion. Low cohesion is an indication of not following SRP.

7.3.1 Why high cohesion is better?

1. More understandable
2. Easier to maintain

7.3.2 Example

```
class Customer {
   int total_price;
   Date date;
   int product_weight;
   String name;
```

```
   String address;
   float discount;

   //getter-setter for all of them

}
```

It seems like a good one, no? Well, its not. Look below.

```
class Order{
  int total_price;
  int total_weight;
  float discount;
  int date;
}

class Customrt{
  String name;
  String address;
}
```

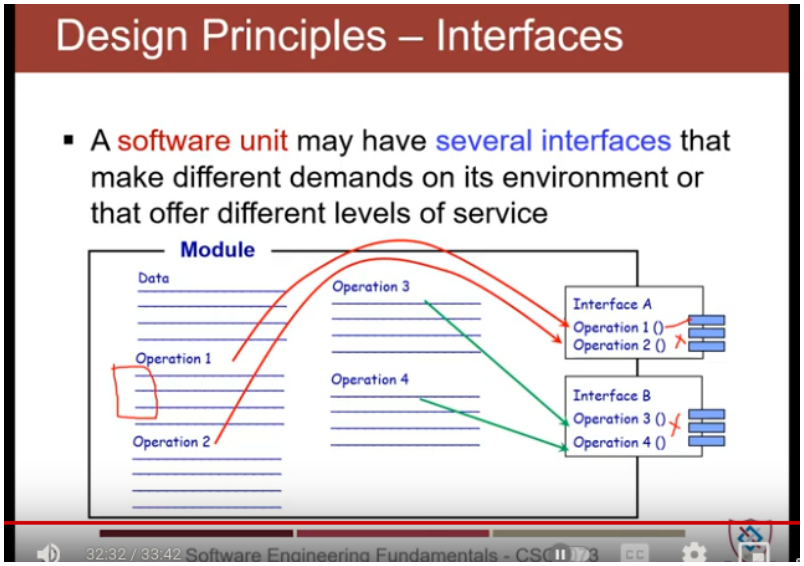
It follows SRP and its highly cohesive as all functions will use both of these classes frequently.

7.4 Interface

Interface is a software unit that provides some service. We define the service in the interface.

A software unit may have or may use several interfaces.

- Interfaces do not provide implementation. Its much like a gui. It hides implementation, it just provides the service.


Design Principles – Interfaces
A software unit may have several interfaces that make different demands on its environment or that offer different levels of service
Module
Data
Operation 1
Operation 2
Operation 3
Operation 4
Interface A
Operation 1 ()
Operation 2 ()
Interface B
Operation 3 ()
Operation 4 ()
32:32 / 33:42 Software Engineering Fundamentals - CSC 3