

MODULE-4

SOLID PRINCIPLE

The SOLID Principles are five principles of Object-Oriented class design. They are a set of rules and best practices to follow while designing a class structure.

These five principles help us understand the need for certain design patterns and software architecture in general.

The SOLID principles were first introduced by the famous Computer Scientist Robert J. Martin (Uncle Bob) in his paper in 2000. But the SOLID acronym was introduced later by Michael Feathers.

The SOLID Principles are as follows

S - The **Single Responsibility** Principle (**SRP**)

O - The **Open-Closed** Principle (**OCP**)

L - The **Liskov Substitution** Principle (**LSP**)

I - The **Interface Segregation** Principle (**ISP**)

D - The **Dependency Inversion** Principle (**DIP**)

The Single Responsibility Principle (SRP)

The Single Responsibility Principle states that **a class should do one thing and therefore it should have only a single reason to change.**

This means that if a class is a data container, like a Book class or a Student class, and it has some fields regarding that entity, it should change only when we change the data model.

Following the Single Responsibility Principle is important. First of all, because many different teams can work on the same project and edit the same class for different reasons, this could lead to incompatible modules.

Second, it makes version control easier. For example, say we have a persistence class that handles database operations, and we see a change in that file in the GitHub commits. By following the SRP, we will know that it is related to storage or database-related stuff.

Merge conflicts are another example. They appear when different teams change the same file. But if the SRP is followed, fewer conflicts will appear – files will have a single reason to change, and conflicts that do exist will be easier to resolve.

Example:

```
class Book {  
    String name;  
    String authorName;  
    int year;  
    int price;  
    String isbn;  
  
    public Book(String name, String authorName, int year, int price, String isbn) {  
        this.name = name;  
        this.authorName = authorName;  
        this.year = year;  
        this.price = price;  
        this.isbn = isbn;  
    }  
}
```

This is a simple book class with some fields.

Now let's create the invoice class which will contain the logic for creating the invoice and calculating the total price. For now, assume that our bookstore only sells books and nothing else.

```
public class Invoice {  
  
    private Book book;  
    private int quantity;  
    private double discountRate;  
    private double taxRate;  
    private double total;
```

```

public Invoice(Book book, int quantity, double discountRate, double taxRate) {
    this.book = book;
    this.quantity = quantity;
    this.discountRate = discountRate;
    this.taxRate = taxRate;
    this.total = this.calculateTotal();
}

public double calculateTotal() {
    double price = ((book.price - book.price * discountRate) * this.quantity);

    double priceWithTaxes = price * (1 + taxRate);

    return priceWithTaxes;
}

public void printInvoice() {
    System.out.println(quantity + "x" + book.name + " " + book.price +
"$");
    System.out.println("Discount Rate: " + discountRate);
    System.out.println("Tax Rate: " + taxRate);
    System.out.println("Total: " + total);
}

public void saveToFile(String filename) {
    // Creates a file with given name and writes the invoice
}
}

```

Here is our invoice class. It also contains some fields about invoicing and 3 methods:

- **calculateTotal** method, which calculates the total price,
- **printInvoice** method, that should print the invoice to console, and

- **saveToFile** method, responsible for writing the invoice to a file.

Our class violates the Single Responsibility Principle in multiple ways.

The first violation is the **printInvoice** method, which contains our printing logic. The SRP states that our class should only have a single reason to change, and that reason should be a change in the invoice calculation for our class.

But in this architecture, if we wanted to change the printing format, we would need to change the class. This is why we should not have printing logic mixed with business logic in the same class.

There is another method that violates the SRP in our class: the **saveToFile** method. It is also an extremely common mistake to mix persistence logic with business logic.

To fix this, We can create new classes for our printing and persistence logic so we will no longer need to modify the invoice class for those purposes.

We create 2 classes, **InvoicePrinter** and **InvoicePersistence**, and move the methods.

```
public class InvoicePrinter {
    private Invoice invoice;

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

    public void print() {
        System.out.println(invoice.quantity + "x " + invoice.book.name + " " +
invoice.book.price + "$");
        System.out.println("Discount Rate: " + invoice.discountRate);
        System.out.println("Tax Rate: " + invoice.taxRate);
        System.out.println("Total: " + invoice.total + "$");
    }
}
```

```
public class InvoicePersistence {  
    Invoice invoice;  
  
    public InvoicePersistence(Invoice invoice) {  
        this.invoice = invoice;  
    }  
  
    public void saveToFile(String filename) {  
        // Creates a file with given name and writes the invoice  
    }  
}
```

Now our class structure obeys the Single Responsibility Principle and every class is responsible for one aspect of our application.

Open-Closed Principle (OCP)

The Open-Closed Principle requires that **classes should be open for extension and closed to modification**.

Modification means changing the code of an existing class, and extension means adding new functionality.

So what this principle wants to say is: We should be able to add new functionality without touching the existing code for the class. This is because whenever we modify the existing code, we are taking the risk of creating potential bugs. So we should avoid touching the tested and reliable (mostly) production code if possible.

But how are we going to add new functionality without touching the class, you may ask. It is usually done with the help of interfaces and abstract classes.

Let's apply it to our Invoice application.

Let's say we want invoices to be saved to a database so that we can search them easily.

We create the database, connect to it, and we add a save method to

our **InvoicePersistence** class:

```

public class InvoicePersistence {
    Invoice invoice;

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

    public void saveToFile(String filename) {
        // Creates a file with given name and writes the invoice
    }

    public void saveToDatabase() {
        // Saves the invoice to database
    }
}

```

Unfortunately we did not design the classes to be easily extendable in the future. So in order to add this feature, we have to modify the **InvoicePersistence** class.

We have to change the type of **InvoicePersistence** to Interface and add a save method. Each persistence class will implement this save method.

```

interface InvoicePersistence {

    public void save(Invoice invoice);}

```

Each persistence class will implement this save method.

```

public class DatabasePersistence implements InvoicePersistence {

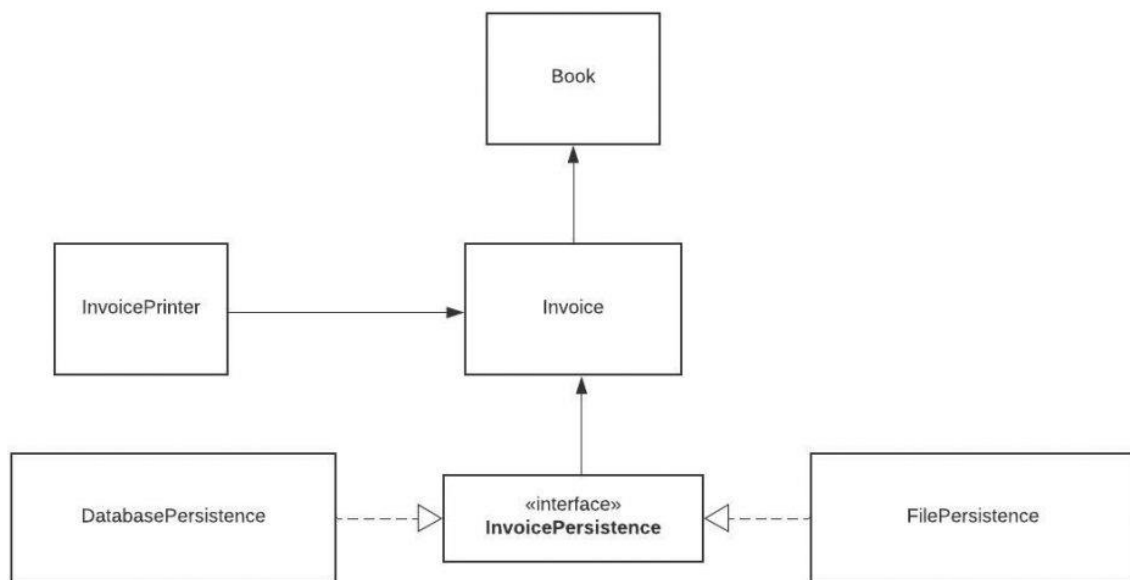
    @Override
    public void save(Invoice invoice) {
        // Save to DB
    }
}

```

```
}
```

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

So our class structure now looks like this:



Now our persistence logic is easily extendable.

Liskov Substitution Principle (LSP)

The Liskov Substitution Principle states that subclasses should be substitutable for their base classes.

This means that, given that class B is a subclass of class A, we should be able to pass an object of class B to any method that expects an object of class A and the method should not give any weird output in that case.

Liskov Substitution Principle.

*The principle defines that **objects** of a **superclass** shall be replaceable with objects of its **subclasses** without **breaking** the application.*

*That requires the **objects** of your **subclasses** to behave in the same way as the **objects** of your **superclass**.*

*It means that we must make sure that **new derived** classes are **extending** the base classes **without changing** their **behavior**.*

Amit Singh Rawat

Consider the following Example

```
class Rectangle{
    protected int width, height;
    public Rectangle(){
    }
    public Rectangle(int width, int height){
        this.width = width;
        this.height = height;
    }
    public void setWidth(int width){
        this.width = width;
    }

    public void setHeight(int height){
        this.height = height;
    }
    public int getWidth(){
        return width;
    }
    public int getHeight(){
        return height;
    }
    public int getArea(){
        return width * height;
    }
}

class Square extends Rectangle{
    public Square(){
    }

    public Square(int size){
        width = height = size;
    }

    @Override
    public void setWidth(int width){
        super.setWidth(width);
        super.setHeight(width);
    }

    @Override
    public void setHeight(int height){
        super.setHeight(height);
        super.setWidth(height);
    }
}
```

Rectangle

```
setHeight(h)=> height=h;  
setWidth(w)=>width=w  
getarea()=> h*w;
```

Square extends Rectangle

```
setHeight(h) => width=h; height=h
```

```
setWidth(w) => width=w, height=w  
getarea() => h*h or w*w;
```

```
Rectangle r=new Rectangle()
```

```
Set(4,5) =>area=20
```

```
Square s=new square()
```

```
Set(4,4) =>area=16
```

```
Set(5,5) =>area=25
```

```
Rectangle square= new square()
```

```
Set(4,5)
```

```
Area=4*5=20 ← actual area
```

```
gerarea()=16 or 25 ← wrong
```

With Liskov Substitution Principle ,the code can be rewritten as follows.

```
public abstract class Shape  
{  
    abstract public int getArea();  
}
```

```
public class Rectangle : Shape  
{  
    public int Width { get; set; }  
    public int Height { get; set; }  
    public override int getArea()  
    {  
        return Height * Width;  
    }  
}
```

```
public class Square : Shape  
{  
    public int Length { get; set; }  
    public override int getArea()  
    {  
        return Length * Length;  
    }  
}
```

```
Shape rectangle = new Rectangle  
    { Height = 2, Width = 3 };  
Console.WriteLine("Area of rectangle : " +  
    rectangle.getArea());  
Output:  
Area of rectangle : 6
```

```
Shape square = new Square  
    { Length = 2 };  
Console.WriteLine("Area of square : " +  
    square.getArea());  
Output:  
Area of square : 4
```

Interface Segregation Principle (ISP)

Segregation means keeping things separated, and the Interface Segregation Principle is about separating the interfaces.

The principle states that many client-specific interfaces are better than one general-purpose interface. Clients should not be forced to implement a function they do not need.

This is a simple principle to understand and apply, so let's see an example.

```
public interface ParkingLot {  
  
    void parkCar();    // Decrease empty spot count by 1  
    void unparkCar(); // Increase empty spots by 1  
    void getCapacity(); // Returns car capacity  
    double calculateFee(Car car); // Returns the price based on number of hours  
    void doPayment(Car car);  
  
}  
  
class Car {  
  
}
```

We modeled a very simplified parking lot. It is the type of parking lot where you pay an hourly fee. Now consider that we want to implement a parking lot that is free.

```
public class FreeParking implements ParkingLot {  
  
    @Override  
    public void parkCar() {  
  
    }  
  
    @Override  
    public void unparkCar() {  
  
    }  
  
}
```

```

@Override
public void getCapacity() {

}

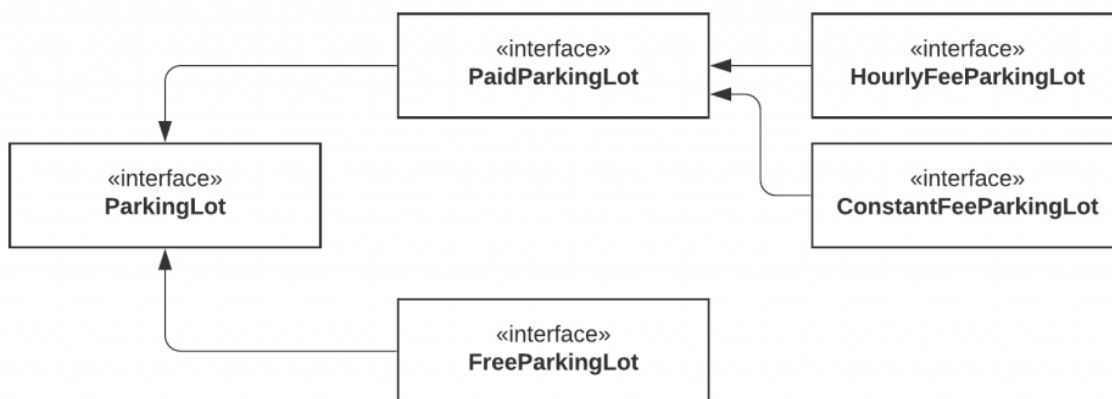
@Override
public double calculateFee(Car car) {
    return 0;
}

@Override
public void doPayment(Car car) {
    throw new Exception("Parking lot is free");
}
}

```

Our parking lot interface was composed of 2 things: Parking related logic (park car, unpark car, get capacity) and payment related logic.

But it is too specific. Because of that, our FreeParking class was forced to implement payment-related methods that are irrelevant. Let's separate or segregate the interfaces.



We've now separated the parking lot. With this new model, we can even go further and split the **PaidParkingLot** to support different types of payment.

Now our model is much more flexible, extendable, and the clients do not need to implement any irrelevant logic because we provide only parking-related functionality in the parking lot interface.

Dependency Inversion Principle (DIP)

The Dependency Inversion principle states that our classes should depend upon interfaces or abstract classes instead of These two principles are indeed related and we have applied this pattern before while we were discussing the Open-Closed Principle.

We want our classes to be open to extension, so we have reorganized our dependencies to depend on interfaces instead of concrete classes. Our PersistenceManager class depends on InvoicePersistence instead of the classes that implement that interface.

Martin's Package Metrics

In 1994 Robert Martin (known as “Uncle Bob”)proposed a group of object-oriented metrics that are popular until now. Those metrics, unlike other object-oriented ones don't represent the full set of attributes to assess individual object-oriented design, they only focus on the relationship between packages in the project.

The level of detail of Martin's metrics is still lower than the one of CK's metrics.

Martin's metrics include:

- **Efferent Coupling (Ce)**
- **Afferent Coupling (Ca)**
- **Instability (I)**
- **Abstractness (A)**
- **Normalized Distance from Main Sequence (D)**

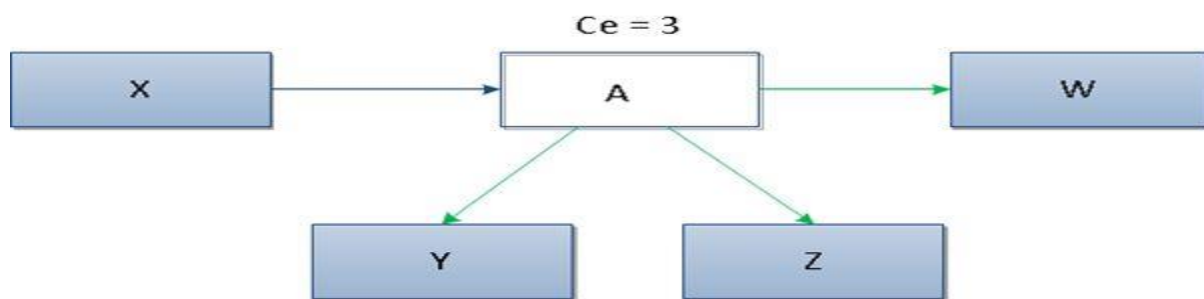
Efferent Coupling (Ce)

This metric is used to measure interrelationships between classes.

As defined, it is a number of classes in a given package, which depends on the classes in other packages. It enables us to measure the vulnerability of the package to changes in packages on which it depends.

Efferent = outgoing.

Efferent Coupling = Fan-out (*Efferent Coupling is also called as Fan-out*)



In the above picture, it can be seen that class A has outgoing dependencies to 3 other classes, that is why metric C_e for this class is 3.

The high value of the metric (i.e. $C_e > 20$) indicates instability of a package i.e. change in any of the numerous external classes can cause the need for changes to the package.

Note: Preferred values for the metric C_e are in the range of 0 to 20, higher values cause problems with care and development of code.

Afferent Coupling (C_a)

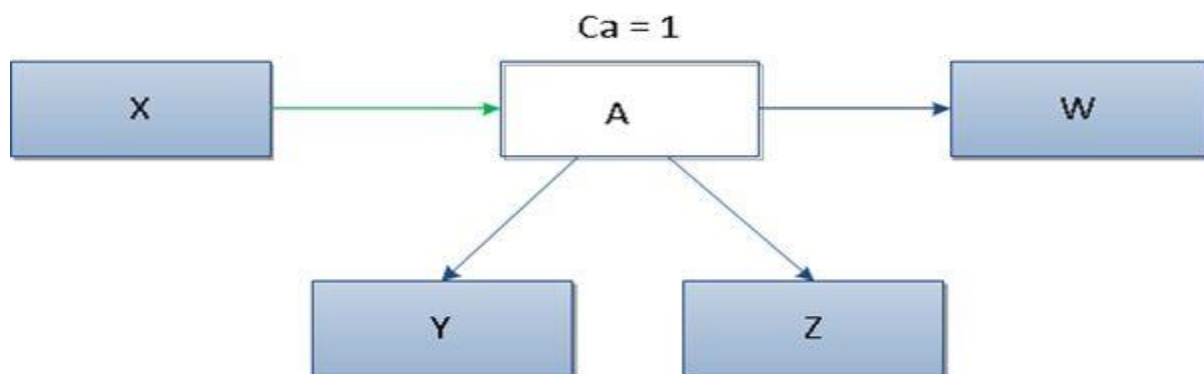
This metric is an addition to metric C_e and is used to measure another type of dependencies between packages, i.e. incoming dependencies.

It is defined as the number of classes in other packages that depend upon classes within the package is an indicator of the package's responsibility.

It enables us to measure the sensitivity of remaining packages to changes in the analysed package.

Afferent = incoming.

Afferent Coupling = Fan-in (*Afferent Coupling is also called as Fan-in*)



In the above picture, it can be seen that class A has only 1 incoming dependency (from class X), that is why the value for metrics C_a equals 1.

High values of metric Ca usually suggest high component stability. This is due to the fact that the class depends on many other classes. Therefore, it can't be modified significantly because, in this case, the probability of spreading such changes increases.

Note: Preferred values for the metric Ca are in the range of 0 to 500.

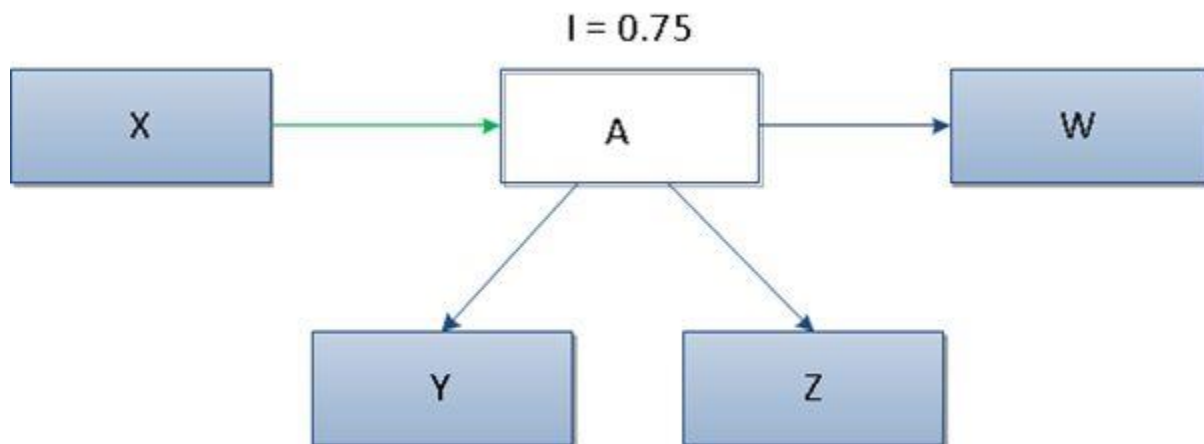
Instability (I)

This metric is used to measure the relative susceptibility of class to changes. According to the definition instability is the ration of outgoing dependencies to all package dependencies and it accepts value from 0 to 1.

The metric is defined according to the formula:

$$I = \frac{Ce}{Ce + Ca}$$

Where: Ce = outgoing dependencies and Ca = incoming dependencies



it can be seen that class A has 3 outgoing and 1 incoming dependencies, therefore according to the formula value of metric I will equal 0,75.

On the basis of value of metric I we can distinguish two types of components:

- **The ones having many outgoing dependencies and not many of incoming ones (*value of I is close to 1*), which are rather unstable due to the possibility of easy changes to these packages;**
- **The ones having many incoming dependencies and not many of outgoing ones (*value of I is close to 0*), therefore they are stable packages and more difficult in modifying due to their greater responsibility.**

Preferred values for the metric I should fall within the ranges of 0 to 0.3 or 0.7 to 1. Packages should be very stable or unstable, therefore we should avoid packages of intermediate stability.

Abstractness (A)

This metric is used to measure the degree of abstraction of the package and is somewhat similar to the instability.

Abstractness is defined as the number of abstract classes in the package to the number of all classes.

The metric is defined according to the formula:

$$A = \frac{T_{abstract}}{T_{abstract} + T_{concrete}}$$

Where: $T_{abstract}$ is the number of abstract classes in a package, $T_{concrete}$ is the number of concrete classes in a package ,

or
$$A = \frac{N_a}{N_c}$$

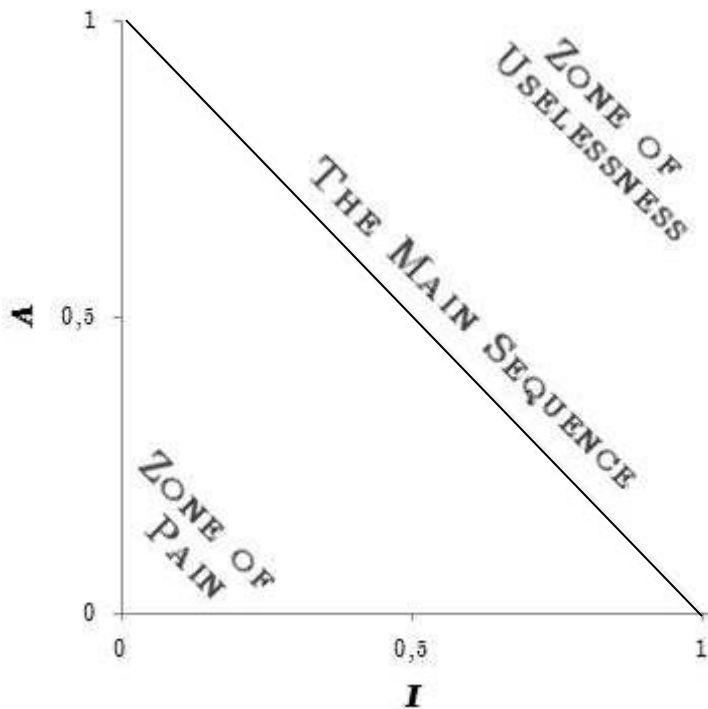
Where, **N_c** is the number of classes in the package and **N_a** is the number of abstract classes or interfaces in the package.

Preferred values for the metric **A** should take extreme values close to 0 or 1.

Packages that are stable (metric **I** close to 0), which means they are dependent at a very low level on other packages, *should also be abstract* (metric **A** close to 1).

In turn, the very unstable packages (metric **I** close to 1) should consist of concrete classes (A metric close to 0).

Additionally, it is worth mentioning that combining abstractness and stability enabled Martin to formulate the existence of main sequence as shown below



Every stable package should be abstract. Stable packages are hard to change because of their responsibilities, but they should also be abstract so they can be extended. On the other hand, an instable package should be concrete so their code can be easily changed.

With these principles the relationship between **I** and **A** is defined by **$I + A = 1$**

In the ideal case, all packages lie in the points (0,1) or (1,0). Stable and abstract packages should be at the upper left corner at (0,1). Instable and concrete packages should be at the lower right corner at (1,0).

The coordinates (0,0) and (1,1) represent bad design. In the **Zone of Pain**, which is the area near lower left corner at (0,0), packages are rigid. They are difficult to extend because of their lack of abstractness and hard to change because of their responsibilities.

The area near the upper right corner at (1,1) is called **Zone of Uselessness**. Packages in this area are abstract but they do not have clients. Clearly, these kinds of packages are useless.

The ideal position for packages is as far as possible from these zones. The **Main Sequence** is a line that connects (1,0) to (0,1) and packages on it are in the right proportions of abstractness and stability. The Martin's metric for a package is the distance from the main sequence:

$$D = \frac{|A + I - 1|}{\sqrt{2}}$$
$$D' = |A + I - 1| .$$

Where D is the distance from the main sequence and it ranges between $[0, \frac{1}{\sqrt{2}}]$. D' is a normalized distance and its range is $[0,1]$.

Normalized Distance from Main Sequence (D)

This metric is used to measure the balance between stability and abstractness and is calculated using the following formula:

$$D = \frac{|A + I - 1|}{\sqrt{2}}$$
$$D' = |A + I - 1| .$$

Where: A- abstractness, I – instability

Object Oriented Metrics(O-O Metrics)

Object Oriented Metrics have been developed for various aspects such as Complexity ,Coupling, Cohesion, Inheritance, Polymorphism and Information hiding.

Two popular O-O metrics are

- **Chidamber and Kemerer Metrics (CK Metrics)**
- **Metrics for Object Oriented Design (MOOD)**

Chidamber and Kemerer Metrics (CK Metrics)

CK suite of metrics claim that these measures can aid in understanding design complexity, in detecting design flaws and in predicting certain project outcomes and external software qualities such as reusability, software defects, testing, and maintenance effort.

CK suite covers all aspects of object oriented (reusability, encapsulation and polymorphism) and CK suite proves to be useful in predicting class fault proneness

The Chidamber & Kemerer metrics suite originally consists of 6 metrics calculated for each class:

- **Weighted Methods per Class (WMC)**
- **Depth of Inheritance Tree (DIT)**
- **Number of Childrens (NOC)**
- **Coupling Between Object Classes (CBO)**
- **The Response for a Class (RFC)**
- **Lack of Cohesion Methods (LCOM)**

Weighted Methods per Class (WMC) :

WMC is simply the method count for a class.

WMC = number of methods defined in class

It is defined as the sum of the complexities of all methods of a class.

A high WMC has been found to lead to more faults. Classes with many methods are likely to be more application specific, limiting the possibility of reuse.

WMC is a predictor of how much time and effort is required to develop and maintain the class.

A large number of methods also means a greater potential impact on derived classes, since the derived classes inherit (some of) the methods of the base class.

- The number of methods and the complexity of methods involved is a predictor of how much time and effort is required to develop and maintain the class.
- The larger the number of methods in a class the greater the potential impact on children, since children will inherit all the methods defined in the class.
- Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse.

Note: Search for high WMC values to spot classes that could be restructured into several smaller classes.

Depth of Inheritance Tree (DIT) :

It is defined as the maximum length from the node to the root of the tree.

DIT = maximum inheritance path from the class to the root class

The deeper a class is in the hierarchy, the more methods and variables it is likely to inherit, making it more complex. Deep trees as such indicate greater design complexity. Inheritance is a tool to manage complexity, really, not to not increase it. A high DIT has been found to increase faults.

- The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior.
- Deeper trees constitute greater design complexity, since more methods and classes are involved.
- The deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods.

Number of Childrens (NOC):

It is defined as the number of immediate subclasses or children

NOC = number of immediate sub-classes of a class

NOC equals the number of immediate child classes derived from a base class.

High NOC has been found to indicate fewer faults. This may be due to high reuse, which is desirable.

- The greater the number of children, the greater the reuse, since inheritance is a form of reuse.
- The greater the number of children, the greater the likelihood of improper abstraction of the parent class. If a class has a large number of children, it may be a case of misuse of sub classing.
- The number of children gives an idea of the potential influence a class has on the design. If a class has a large number of children, it may require more testing of the methods in that class.

Note: A class with a high NOC and a high WMC indicates complexity at the top of the class hierarchy. The class is potentially influencing a large number of descendant classes. This can be a sign of poor design. A redesign may be required.

Coupling Between Object Classes (CBO):

It is defined as the count of the classes to which a class is coupled.

Two classes are coupled when methods declared in one class use methods or instance variables of the other class.

High CBO is undesirable. Excessive coupling between object classes is detrimental to modular design and prevents reuse. The more independent a class is, it is easier to reuse it in another application.

In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult.

A high coupling has been found to indicate fault-proneness. Rigorous testing is thus needed.

- Excessive coupling between object classes is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application.
- In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult.
- A measure of coupling is useful to determine how complex the testing of various parts of a design is likely to be. The higher the inter-object class coupling, the more rigorous the testing needs to be.

Note: Good design = High Cohesion and Low coupling

The Response for a Class (RFC):

The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class.

It is defined as number of methods in the set of all methods that can be invoked in response to a message sent to an object of a class.

Since RFC specifically includes methods called from outside the class, it is also a measure of the potential communication between the class and other classes.

A large RFC has been found to indicate more faults. Classes with a high RFC are more complex and harder to understand. Testing and debugging is complicated.

- If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding on the part of the tester.
- The larger the number of methods that can be invoked from a class, the greater the complexity of the class.
- A worst case value for possible responses will assist in appropriate allocation of testing time.

Lack of Cohesion Methods (LCOM) :

It is defined as the number of different methods within a class that reference a given instance variable.

- A highly cohesive module should stand alone; high cohesion indicates good class subdivision.
- High cohesion implies simplicity and high reusability.
- Cohesiveness of methods within a class is desirable, since it promotes encapsulation. As a drawback, a highly cohesive class has high coupling between the methods of class, which in turn indicates high testing effort for that class.
- Lack of cohesion implies classes should probably be split into two or more subclasses.
- Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

Note: Good design = High Cohesion and Low coupling

Metrics for Object Oriented Design (MOOD)

The **MOOD (Metrics for Object oriented Design)** set of metrics refers to a basic structural mechanism of the OO paradigm as

➤ *Encapsulation*

- Method Hiding Factor(MHF) and
- Attribute Hiding Factor(AHF)

➤ *Inheritance*

- *Method Inheritance Factor* (MIF) and
- *Attribute Inheritance Factor* (AIF)

➤ *Polymorphism*

- Polymorphism Factor (PF)

➤ *Message-passing*

- *Coupling Factor*(COF).

Method Hiding Factor (MHF):

MHF is defined as the ratio of the sum of the invisibilities of all methods defined in all classes to the total number of methods defined in the system under consideration.

In other words, MHF is the ratio of hidden methods – protected or private methods to total number of methods.

$$MHF = \frac{\sum_{i=1}^n M_h(C_i)}{\sum_{i=1}^n M_d(C_i)}$$

Where $M_h(C_i)$ = hidden methods in class C_i

$M_d(C_i)$ = total number of methods defined in class C_i

n = total number of classes

The number of methods defined in the class is the sum of the number of methods visible in the class and number of methods hidden in the class.

Note:

- ✓ MHF is proposed to measure the encapsulation (the relative amount of information hidden).
- ✓ As MHF increases, the defect density and the effort spent to fix is expected to decrease. Inherited methods are not considered.

Attribute Hiding Factor (AHF)

AHF is defined as the ratio of the sum of the invisibilities of all attributes defined in all classes to the total number of attributes defined in the system under consideration.

In other words, AHF is the ratio of hidden attributes –protected or private- to total attributes.

$$AHF = \frac{\sum_{i=1}^n A_h(C_i)}{\sum_{i=1}^n A_d(C_i)}$$

Where $A_h(C_i)$ = hidden attributes in class C_i

$A_d(C_i)$ = total number of attributes defines in class C_i

n = total number of classes

The number of attributes defined in the class is the sum of the number of attributes visible in the class and number of attributes hidden in the class

Note:

- ✓ AHF is also proposed to measure the encapsulation (amount of information hidden).
- ✓ Ideally this metric should be always 100%. Systems as a rule should try to hide nearly all instance data. Design guidelines suggest that public attributes should not be used because are generally considered to violate the rules of OO

Method Inheritance Factor (MIF)

MIF is defined as the ratio of the sum of the inherited methods in all classes of the system under consideration to the total number of available methods (locally defined plus inherited) for all classes.

MIF measure directly the number of inherited methods as a proportion of the total number of methods.

$$MIF = \frac{\sum_{i=1}^n M_i(C_i)}{\sum_{i=1}^n M_a(C_i)}$$

Where $M_i(C_i)$ = the number of methods inherited in C_i

$M_a(C_i)$ = the number of methods that can be invoked in C_i

n = total number of classes

The number of methods that can be invoked with a class is the sum of the number of methods declared in the class and number of methods inherited by the class.

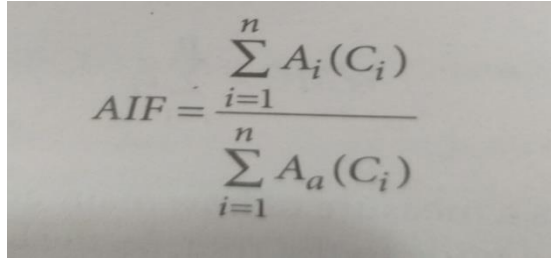
Note:

- ✓ MIF is a measure of inheritance, and consequently as a means of measure the level of reuse.
- ✓ It is also a aid to the assessment of testing needed.

Attribute Inheritance Factor (AIF)

AIF is defined as the ratio of the sum of inherited attributes in all classes of the system under consideration to the total number of available attributes (locally defined plus inherited) for all classes.

AIF measure directly the number of inherited attributes as a proportion of the total number of attributes.


$$AIF = \frac{\sum_{i=1}^n A_i(C_i)}{\sum_{i=1}^n A_a(C_i)}$$

Where $A_i(C_i)$ = the number of attributes inherited in C_i

$A_a(C_i)$ = the number of attributes that can be invoked in C_i

n = total number of classes

The number of attributes that defined in a class is the sum of the number of attributes declared in the class and the number of inherited attributes.

Note:

- ✓ AIF as a way to express the level of reuse in a system.

Polymorphism Factor (PF)

PF is defined as the ratio of the actual number of possible different polymorphic situation for class C_i to the maximum number of possible distinct polymorphic situations for class C_i .

PF is the number of methods that redefine inherited methods, divided by the maximum number of possible distinct polymorphic situations.

$$PF = \frac{\sum_i M_o(C_i)}{\sum_i [M_n(C_i) * DC(C_i)]}$$

Where

$M_o(C_i)$ = the number of overriding methods in class C_i

$M_n(C_i)$ = the number of new methods declared in class C_i

$DC(C_i)$ = the number of descendants count in C_i

n = total number of classes

Note:

- ✓ PF is a measure of polymorphism potential.
- ✓ Polymorphism arises from inheritance. In some cases, overriding methods reduce complexity, so increasing understandability and maintainability.
- ✓ PF is an indirect measure of the dynamic binding in a system.

Coupling Factor (CF)

CF is defined as the ratio of the maximum possible number of couplings in the system to the actual number of couplings excluding the coupling due to inheritance.

That is, this metrics counts the number of inter-class communications.

$$CF = \frac{\sum_i \sum_j is_client(C_i, C_j)}{(n^2 - n)}$$

Where $is_client(C_i, C_j) = 1$ if class i has relation with class j , Otherwise it is zero

$n^2 - n$ = actual number of couplings in the system where n is the total number of classes.

The relationship might be that class i calls a method in class j or has a reference to class j or to an attribute in class j . This relationship can't be an inheritance.

Note:

- ✓ Coupling is viewed as a measure of increasing complexity, reducing both encapsulation and potential reuse and limiting understandability and maintainability.
- ✓ As coupling among classes increases, the defect density and lack of maintainability increase.