# C++ Programming Concepts

LSEG Technology

12th November 2022

LSEG

# Agenda

- OOP Concepts
- SOLID Principles
- Design Patterns

# OOP

- Classes and objects
- Header and cpp files
- Members :  variables, functions, members could be other classes
- Structure and behavior
- Access specifiers
- Constructor, destructor : default, overloaded
- Copy constructor : deep/shallow copying
- Object ownership (deletion determines ownership)

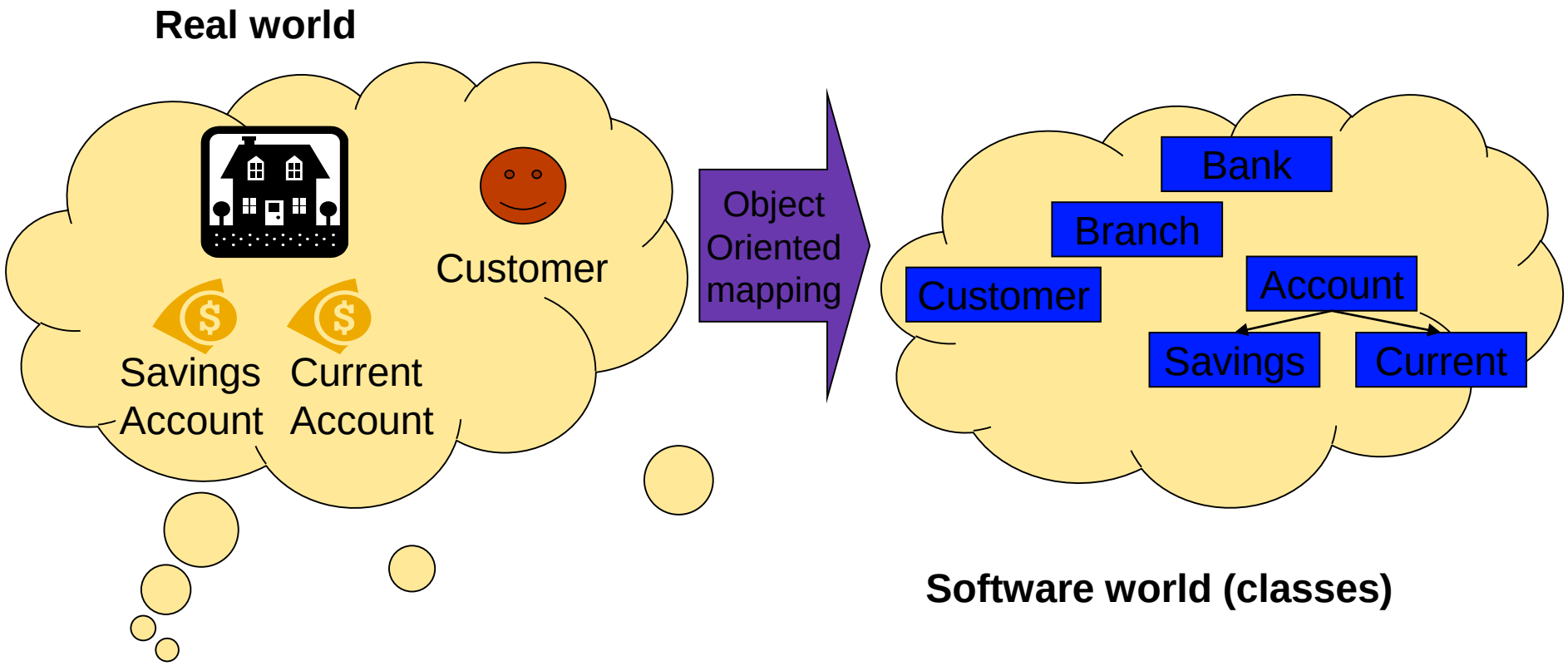
# Mapping real world to the software world

## Direct mapping reduces information loss

- Easy to analyze the real world and understand relationships

- Software implements these relationships with needed operations

# Mapping real world to the software world



A tool for creating new types

Easily be used as built in types (e.g. ComplexNumber class)

- With member functions like

`Add() or + operator support`

- With added functionality relevant to the new type
- Can use similar to integers (or any primitive type)

**Using integers (primitive data type)**

```
int iNum1 = 10;
int iNum2 = 20;
int iNum3 = iNum1+iNum2;
```

**Using ComplexNumber Class**

```
ComplexNumber mCN1 = 10;
ComplexNumber mCN2 = 20;
//using Add() member function
ComplexNumber mCN3 =
mCN1.Add(mCN2);
//OR overloading + operator
ComplexNumber mCN3 = mCN1+mCN2;
```

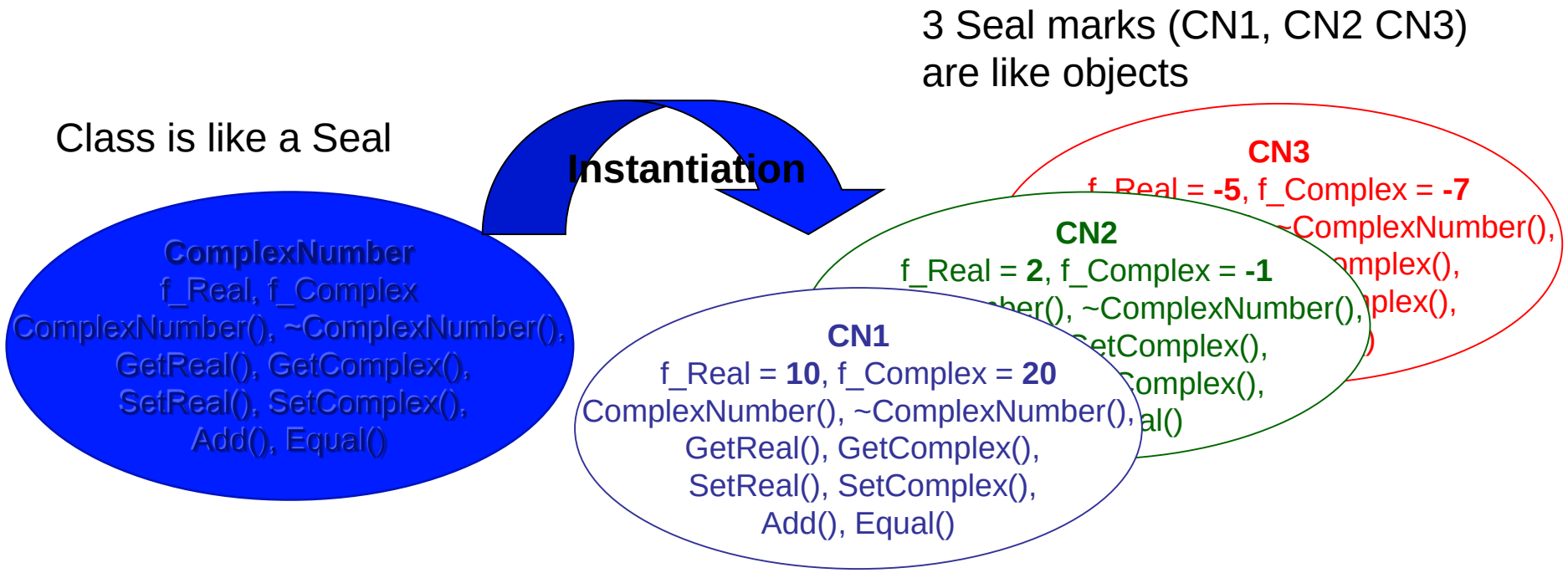
## A Class is a template.

- It does not exist physically
- It’s a custom made “type”
- Its just an definition

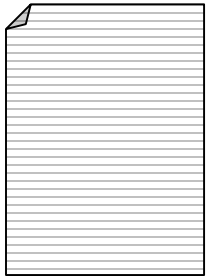
## An Instance of a class is an Object

- Object is created from a class
- Object is in memory
- Object has a state

# Class & Instance



# Class File Organization



Class Header file
(ComplexNumber.h)

- Class Declaration
- Can have more classes in a single header file.
- Declares the content of a class

Class Source file
(ComplexNumber.cpp)

- Class Definition or Implementation
- Can have more than one Definition files
- Members that are defined in the class declaration are implemented here.

# Classes vs. functions and Data

e.g. Complex number implementation using ‘C’ language

```
typedef struct CN_Data
{
    float f_Real;
    float f_Complex;
}ComplexNumber;
```

***What is a struct?***

**Data**

```
ComplexNumber Add(ComplexNumber mCN1, ComplexNumber mCN2)
{
    ComplexNumber mResult;
    mResult.f_Real = mCN1.f_Real + mCN2.f_Real;
    mResult.f_Complex = mCN1.f_Complex + mCN1.f_Complex;
    return mResult;
}
```

**Function**

```
//usage
ComplexNumber mCN1, mCN2, mCN3;
mCN1.f_Real = 10;
mCN1.f_Complex = 20;
mCN2.f_Real = 0;
mCN2.f_Complex = -10;
mCN3 = Add(mCN1, mCN2);
```

# Class Syntax – Header file

## Class Header file (ComplexNumber.h)

```
class ComplexNumber
{
private:
        float f_Real;
        float f_Complex;
public:
        ComplexNumber();      //Constructor
        ~ComplexNumber();     //Destructor
        //Getters
        float GetReal() const;       //Get method for Real part
        float GetComplex() const;    //Get method for Complex part
        //Setters
        void SetReal(const float fReal);    //Set Real part
        void SetComplex(const float fComp);//Set Complex part
        //Operations
        void Add(ComplexNumber mCN);         //Add mCN to this
        bool Equal(ComplexNumber mCN);       //Check for equality
        //...more...
};
```

# Class Implementation (ComplexNumber.cpp)

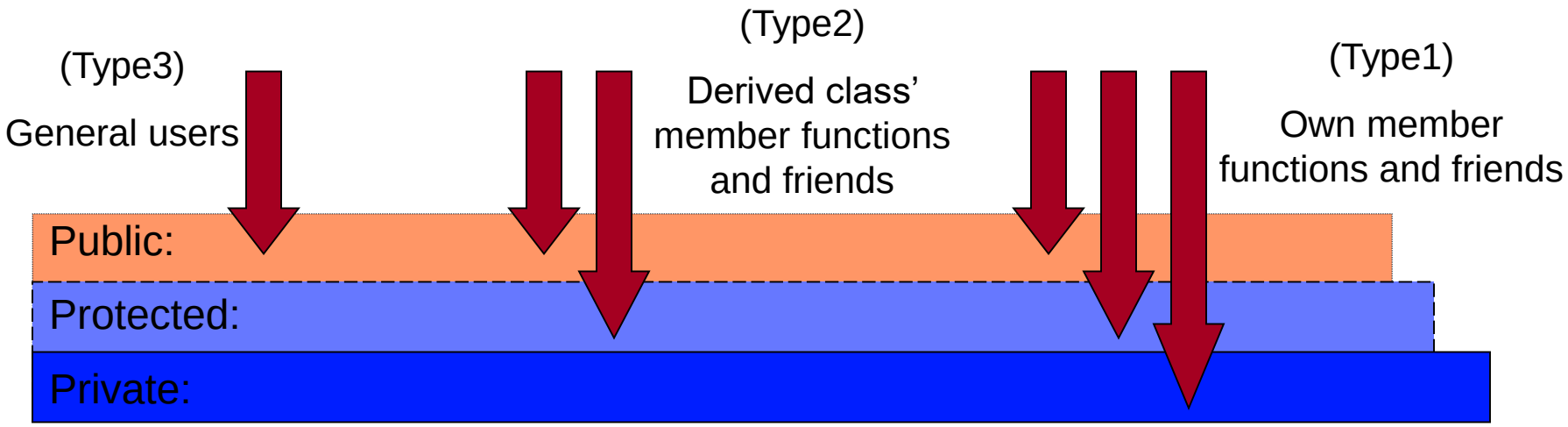
```
#include ”ComplexNumber.h”

ComplexNumber::ComplexNumber() //Constructor
{
        f_Real = 0.0; f_Complex = 0.0;
}
//...more...
float ComplexNumber::GetReal() const //Get method for Real part
{
        return f_Real;
}
//...more...
void ComplexNumber::SetReal(const float fReal) //Set Real part
{
        f_Real = fReal;
}
//...more...
void ComplexNumber::Add(ComplexNumber mCN) //Add mCN to this
{
        f_Real = f_Real + mCN.GetReal();
        f_Complex = f_Complex + mCN.GetComplex();
}
//...more...
```

**Private -** can only be used by member functions and friends of that class in which it is declared

**Protected -** can only be used by member functions, friends of the class AND member functions and friends of derived classes

**Public -** publicly known interface to other objects can be used by any function



# Creating Class Objects

- E.g. create a object of class ComplexNumber in function scope.

```
void function()
{
    ComplexNumber mCN; //mCN object in the function scope
    //...
}
```

- E.g. create objects of class ComplexNumber in Static, Global & Namespace levels

```
class X
{
    //...
    static ComplexNumber m_CNStatic; //static member of class X
};

ComplexNumber g_CNGlobal;         //global variable
ComplexNumber X::m_CNStatic;      //definition of static member of X

Namespace Z
{
    ComplexNumber g_CNNameSpace; //variable global to namespace Z
}
```

# Creating Class Objects

- **E.g. create a object of class ComplexNumber in free store.**
- `new` will allocate memory for one object

```
ComplexNumber *pCN1 = new ComplexNumber; //pCN1 is a pointer to a object allocated in the heap. Call to default constructor
ComplexNumber *pCN2 = new ComplexNumber(10, 20); //pCN2 is a pointer to a object allocated in the heap. Call to custom constructor
```

- `new[]` will allocate memory for an array of objects

```
ComplexNumber *pCN2 = new ComplexNumber[iCount]; /*pCN2 is a pointer to an array of iCount number of object allocated in the heap. */
```

- **E.g. delete a object of class ComplexNumber in free store.**
- Call to `delete` to destruct a object

```
delete pCN1; //delete object which is pointed by pCN1 pointer
pCN1 = NULL; //to prevent accessing deleted object
```

- Call `delete[]` will destruct all the objects in the array

```
delete [] pCN2; //delete array of objects which is pointed by pCN2 pointer
pCN2 = NULL; //to prevent accessing deleted object
```

# OOP (Recap)

C++ has added object orientation to the C programming language

OO aims to model the real world in the programming domain

A Class is a template. It does not exist; It's a custom made "type"

Objet is an Instance of a class. Object exists in the memory. Object has properties, behaviors and state.

In C++, a class is organized to two files in general.

Class header contain the class declaration. Its the promise. It talks about what are its member variables are and which member operations are permitted at which level of access. Class source file is the way the promise is actually implemented.

# OOP Concepts

- Abstraction
  - Representing only the needed information in program.
- Encapsulation
  - A mechanism to preserve the state of objects
- Inheritance
  - A generalization/specialization mechanism
  - Defines an “is a” relationship
- Polymorphism
  - A mechanism to separate interface and implementation

# Abstraction

Abstraction is in our minds.
We (our mind) extract only the required details.
It’s an extremely powerful technique to deal with the complexity.
When we can’t master the entirety of a complex object, we select only the essential details of the object and we create an idealized model of the object.
A class represents a real world entity; class contains only the essential details matching to the problem domain.

# Encapsulation

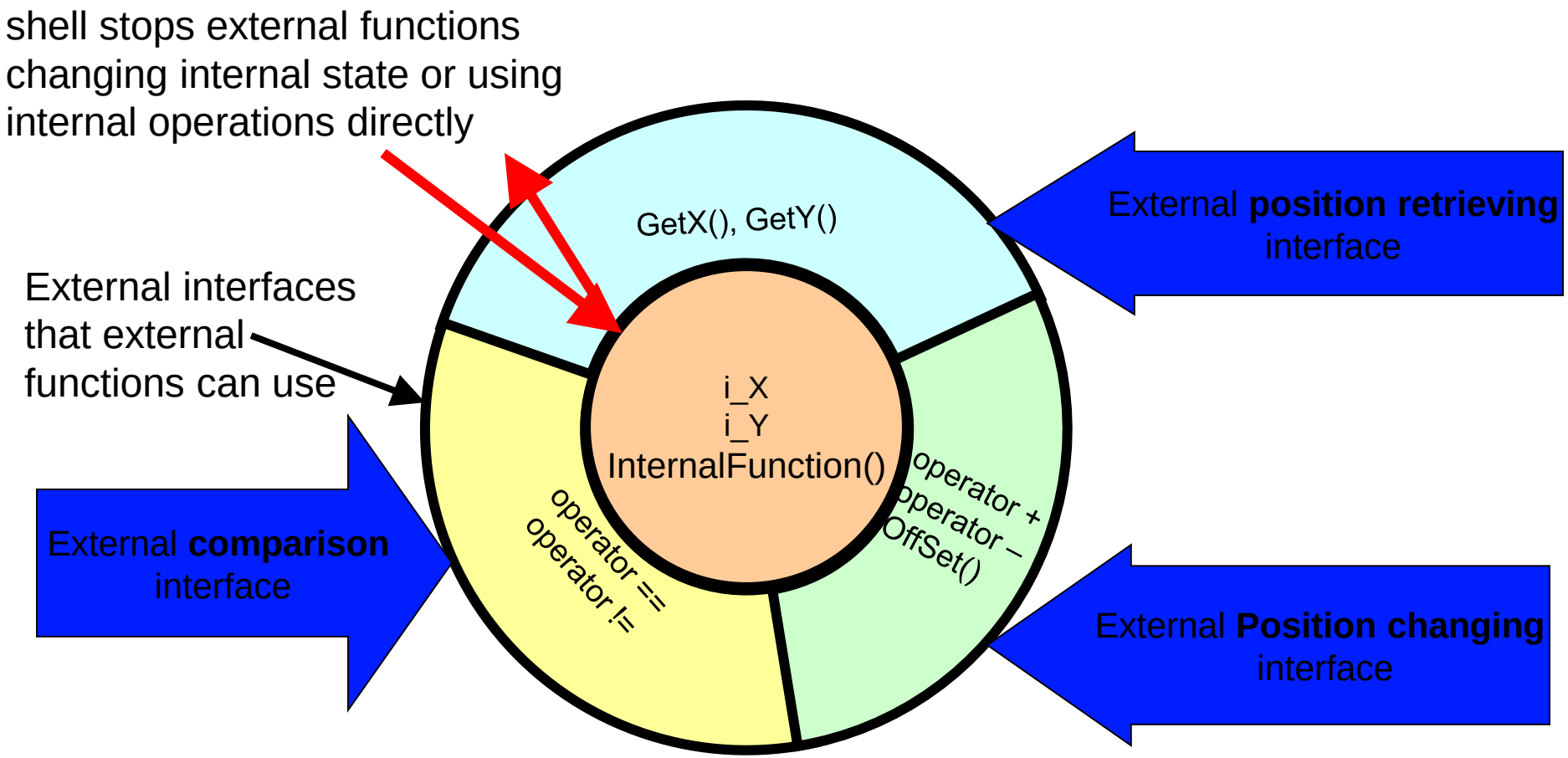
```
class Point{
private:
    int i_X, i_Y;
    void InternalFunction();
public:
    //External constructor and destructor
    Point (int iX, iY);
    ~Point ();
    //External Position changing interface
    Point operator+(Point mRHS);
    Point operator-(Point mRHS);
    Point Offset(int xOffset, int yOffset);
    //External comparison interface
    bool operator==(Point mRHS);
    bool operator!=(Point mRHS);
    //External position retrieving interface
    int GetX();
    int GetY();
};
```

**Internal representation & operations are hidden from external world**

**Operation implementation is hidden** from external world

**Only operational interface is visible**

# Encapsulation



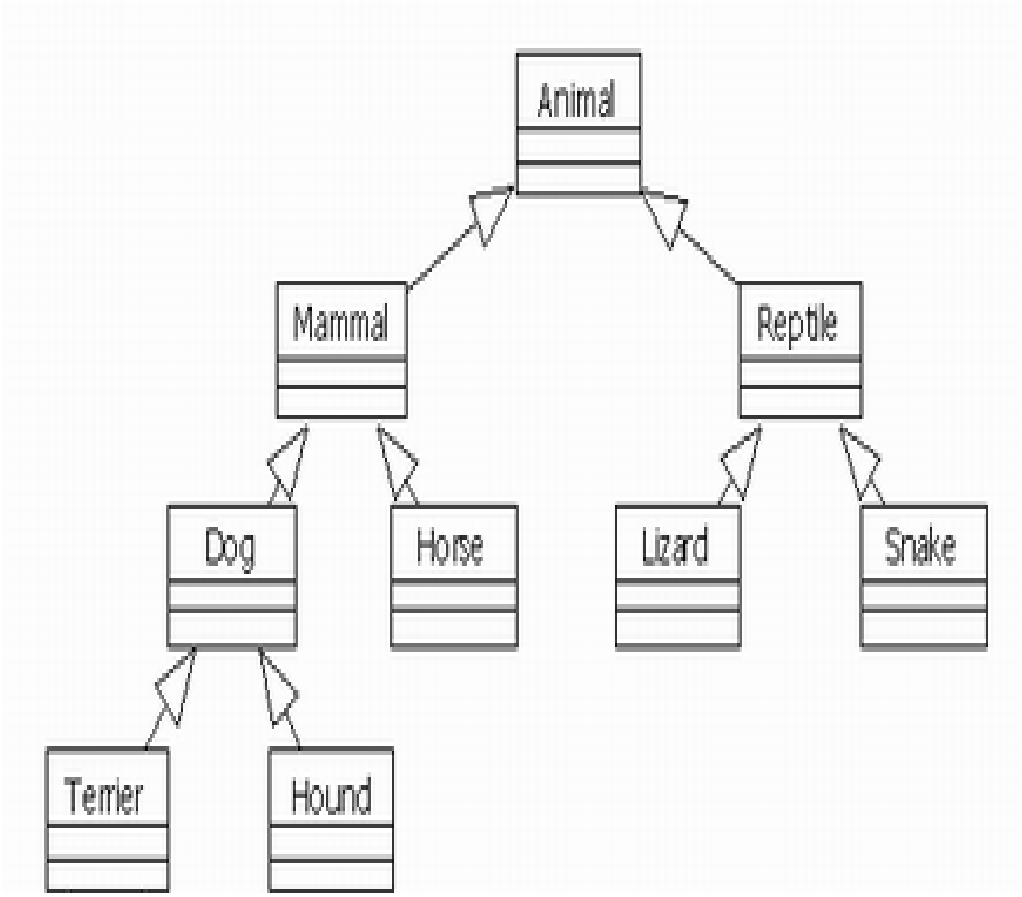
# Inheritance

Entities in the real world often form conceptual hierarchies.

Inheritance is used in OO programming for modelling such conceptual hierarchies in programming domain.

Inheritance is also known as the **“is-a”** relationship between the classes.

Eg: Dog is a mammal



Hierarchical organization based on the behavior of classes
The combination of both

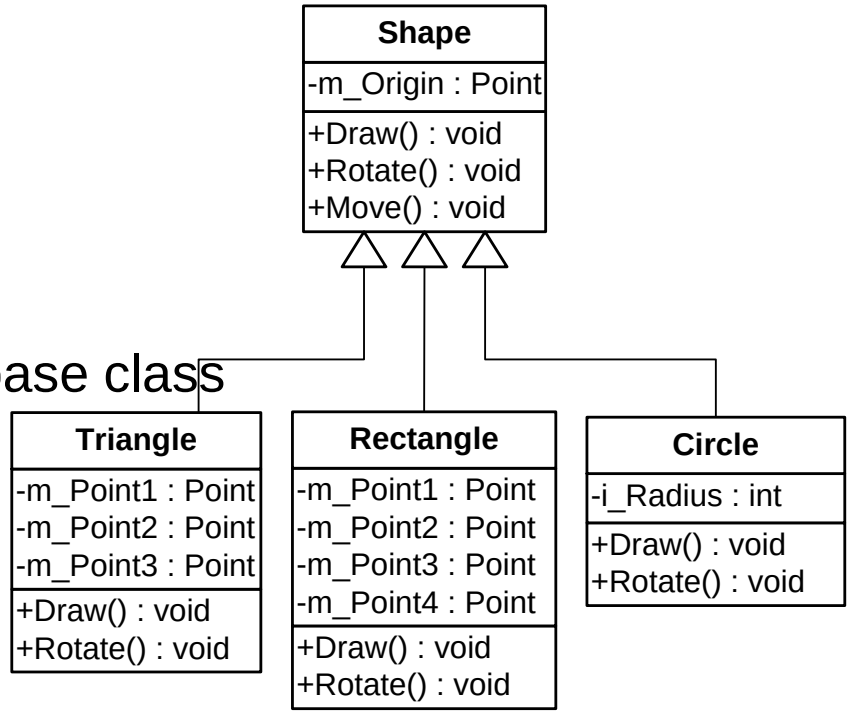
- **Specialization** – division in to various classes
  - E.g. Triangle, Rectangle and Circle are various shape classes which draw a Triangle, Rectangle or a Circle respectively on a workspace
- **Generalization** – abstractions of various classes
  - E.g. Shapes can be drawn on a workspace. A shape can be Triangle, Rectangle or Circle. Shape class provides common functionality to any shape. Each Triangle, Rectangle and Circle provides more specific functionality

# Inheritance

**Super class (parent class / base class)**
- **More common class**
- Shape class

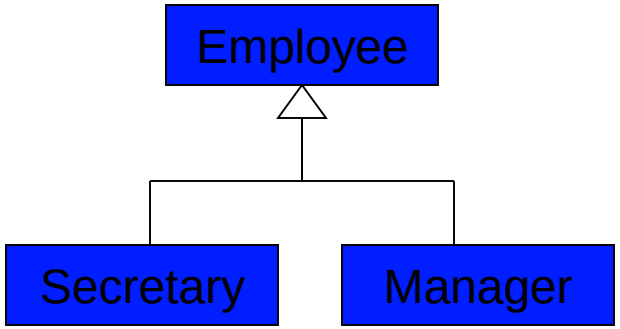
**Sub class (child class / derived class)**
- **More specific class**
- Triangle
- Rectangle
- Circle

Sub class is said to be **derived from** the base class



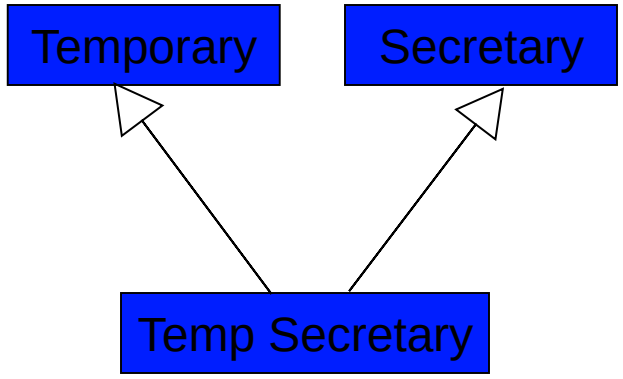
## Single Inheritance

- Sub class have only one base class
- Java and C++ provides at language level




## Multiple Inheritance

- Sub class can have more than one base class
- Java does NOT provide at language level
- C++ provides at language level



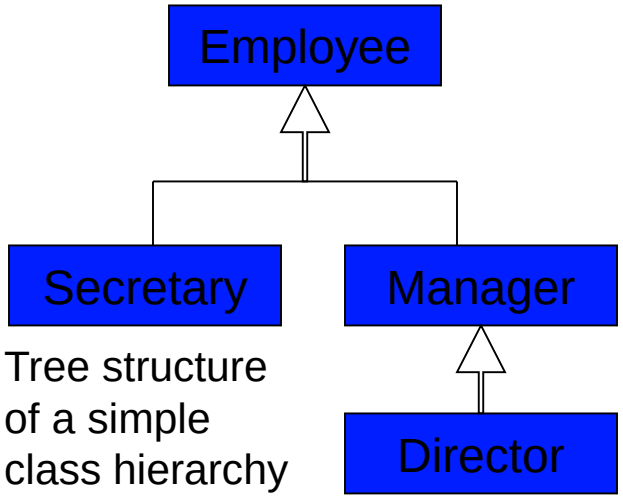
# Class hierarchies

```
class Employee{/**/};
class Secretary{/**/};
class Manager : public Employee{/**/};
class Director : public Manager{/**/};
```

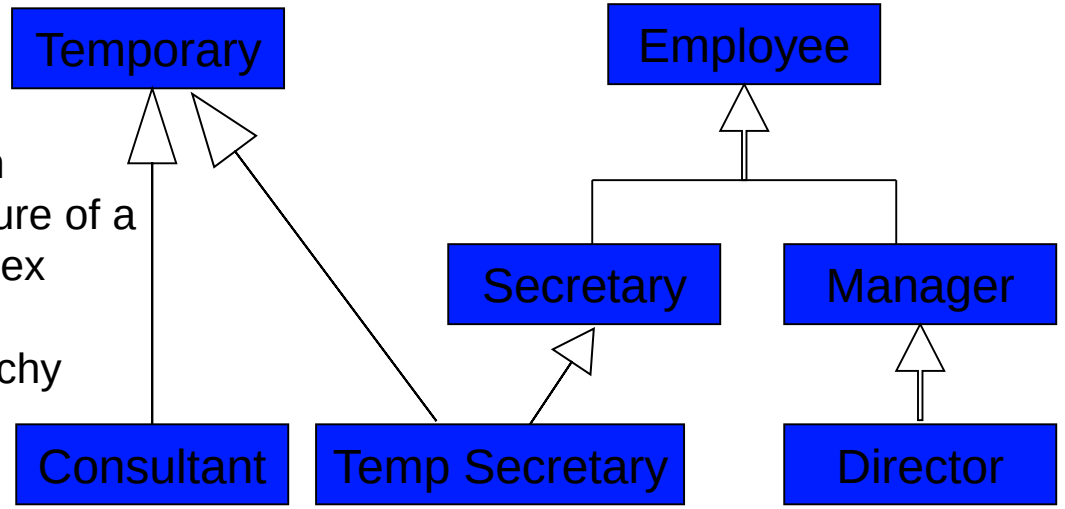
A set of related classes is called a class hierarchy

Derived classes can itself be a base class

Most often a tree, but can be a more general graph




Tree structure of a simple class hierarchy




Graph structure of a complex class hierarchy

# Polymorphism

One of the key features of class inheritance is that a **pointer to a derived class** is type-compatible with a **pointer to its base class**. Polymorphism is the art of taking advantage of this simple but powerful and versatile feature.

Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

Poly + Morphism => ability to have multiple forms of the same thing

In other words, some code or operations or objects behave differently in different contexts

Polymorphism is generally used in the forms of Overriding, Overloading and Dynamic Binding (late/lazy binding)

# Overloading

One flavor of polymorphism.

An exiting operator or function is made to operate on new data type

Eg: Overloading in the “+” operator

```
2 + 3 <-- integer addition
3.14 + 7.0 <-- floating point addition
“foo” + “bar” <-- string concatenation
```

# Overriding

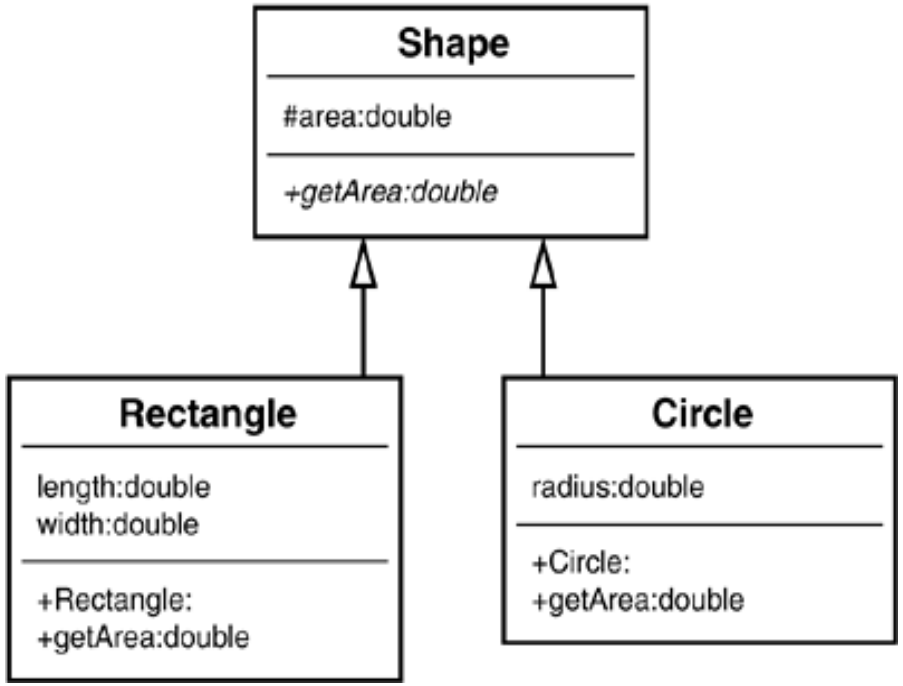
Another flavor of polymorphism.

Presents when a base class's function is redefined in a subclass with a new/specific implementation.

.

Eg : Overriding the "getArea" function for different types of Shapes

Same function "getArea" is called. Implementation is decided by the type of object its executed on.



# Dynamic Binding

Static Binding Vs. Dynamic Binding

Static Binding : The compiler uses the type of the pointer to perform the binding at compile time.

Dynamic Binding: The decision is made at run-time based upon the type of the actual object.

It is required to use the keyword “virtual” at the function declaration for signaling the compiler that the function may be overridden in a child class.

Dynamic Binding is another flavour of polymorphism used in OO.

# Exercise

- Write a program to print area and circumference of a triangle, square and a rectangle. Use inheritance and polymorphism.

- First, understand the requirement
- Then, have a simple design on a paper
- Next, code it !
- Finally, Test it.

- Think : Are there any classes which cannot instantiate objects. If so, why they are required ?

# Virtual Functions

A function in the base class **declared with the keyword virtual**.
This tells the compiler that we don't want static linkage for this function. Instead, use another implementation of this function which **can be** available in the sub class.
Therefore, the compiler have to decide which function to call at the runtime, not at the compile time.
This sort of operation is referred to as dynamic linkage, or late binding.
This allows us to change the function implementation to be used based on the kind of object for which it is called.

# Exercise (result)

```cpp
#include <cstdlib>
#include <iostream>

using namespace std;

class Shape
{
public:
        Shape(){}
        ~Shape(){}

        virtual void getName()
        {
                cout << "I'm a Shape !" << endl;
        }
};

class ClosedShape : public Shape
{
public:
        float getArea();

        ClosedShape(){}
        ~ClosedShape(){}

        virtual void getName()
        {
                cout << "I'm a ClosedShape !" << endl;
        }
};

class Ellipse : public ClosedShape
{
public:
        Ellipse(int width, int height)
        {
            _width = width;
            _height = height;
        }

        float getArea()
        {
            return 22 / 7 * ((_width + _height) / 2 ) * ((_width + _height) / 2 );
        }

        virtual void getName()
        {
                cout << "I'm an Ellipse !" << endl;
        }

protected:
            int _width;
            int _height;

private:

};

class Circle : public Ellipse
{
public:
        Circle(int diameter) : Ellipse(diameter, diameter)
        {
        }

        virtual void getName()
        {
                cout << "I'm a Circle !" << endl;
        }
};

int main(int argc, char *argv[])
{
    Shape* pShape1 = new Shape();
    pShape1->getName();

    Shape* pShape = new Circle(4);
    pShape->getName();

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

# Interfaces in C++ (Abstract Classes)

- Abstract classes are classes with pure virtual functions
- Abstract classes cannot be instantiated
- Abstract classes with all pure virtual functions are called Interfaces (like Java interfaces)

```
1   class ClosedShape : public Shape
2   {
3   public:
4       virtual float getArea() = 0;    //< Pure virtual method
5
6       ClosedShape() { };
7       ~ClosedShape() { };
8
9       virtual void getName()
10      {
11          cout << "I'm a ClosedShape !" << endl;
12      }
13  };
```

# SOLID Principles

**Software always change**
Software should be written such that it supports changes.

For a good Object Oriented Design of a Software, it should be easy to

- Understand
- Maintain
- Extend

Many Principles which Guide us to design quality software.

- SOLID
- GRASP
- DRY

- SOLID:
  - first introduced by Uncle Bob (Robert C.Martin) in his 2000 paper *Design Principles and Design Patterns*.
  - acronym was introduced later, around 2004, by Michael Feathers.

# What is SOLID?

- S - Single Responsibility Principle (SRP)
- O – Open Closed Principle (OCP)
- L – Liskov Substitution Principle (LSP)
- I – Interface Segregation Principle (ISP)
- D – Dependency Inversion Principle (DIP)

# What is SOLID?

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

- **Each Software Component should have only one reason to change**

O – Open Closed Principle (OCP)

L – Liskov Substitution Principle (LSP)

I – Interface Segregation Principle (ISP)

D – Dependency Inversion Principle (DIP)

# S - for Single Responsibility Principle (SRP)

S
O
L
I
D

***Each Software Component should have One & Only One Responsibility.***

*OR, As Uncle Bob Says:*

***Each Software Component should have One & Only One Reason To Change.***

More responsibilities/ reasons to change → More Frequent Changes → More Possibility of Bugs → Not easy to maintain


*Image courtesy of Derick Bailey*

# S - for Single Responsibility Principle

```
class  Student
{
private:
    int id;
    string name;
    int age;
    string email;
    int marks[9];

    string dburl;
    string emailpassword;
public:
    Student(int id, string name, int age, string email): id(id), name(name), age(age), email(email)
    {

    }
    string getName() { return name; }
    int getId() { return id;  }
    int getAge() { return age; }

    void save() {
        cout << "Save called" << endl;
        //create db connection
        //code to save to DB
    }

    void sendEmail(string toemail, string content)
    {
        cout << "Send email called" << endl;
        //create smtp connection
        //send email
    }
};
```

What are the reasons for changes to this class??

1. Changing student information
2. Changing database backend
3. Changing email sending options
4. …

S
O
L
I
D

LSEG

# S - for Single Responsibility Principle

```
class  Student
{
private:
    int id;
    string name;
    int age;

    int marks[9];

    DBConnection dbconnection;
    EmailSender emailSender;


public:
    Student(int id, string name, int age, string email): id(id), name(name), age(age)
    {

    }
    string getName() { return name; }
    int getId() { return id;  }
    int getAge() { return age; }

    void save() {
        dbconnection.save();
    }

    void sendEmail(string toemail, string content)
    {
        emailSender.sendEmail(toemail, content);
    }

};
```

Classes have only 1 responsibility to rach.

Student: keeps track of student information

DBConnection: handles DB queries

EmailSender: handle email related stuff

S
O
L
I
D

# S - for Single Responsibility Principle (SRP)

S
O
L
I
D

## For Better Adherence to SRP

More Related components within

**High Cohesion**

( *the degree to which the various parts of a software components are related*)

**Loose Coupling**

(*the level of inter dependency between various software components*)

Inter Dependency With Outside Low

# S - for Single Responsibility Principle - Tips

S
O
L
I
D

- Prevent Antipattern of God Object (1 Class doing all): (the opposite of SRP)
- Always try  High Cohesion and Loose Coupling.
- But, Prevent Needless Complexity:
  - Group responsibilities/reasons to change in a related way.
  - Don’t try to create classes/separate modules for simplest levels.
- Always separate business logic and persistent logic.
- Use Facade, DAO or Proxy patterns to separate responsibilities.

# What is SOLID?

S - Single Responsibility Principle (SRP)

- Each Software Component should have only one reason to change

**O – Open Closed Principle (OCP)**

- **Software Components should be closed for modification, but open for extension**

L – Liskov Substitution Principle (LSP)

I – Interface Segregation Principle (ISP)

D – Dependency Inversion Principle (DIP)

# O - for Open Closed Principle (OCP)

S
O
L
I
D

***Software Entities (classes, modules, functions, etc.) should be OPEN for EXTENSION, but CLOSED for MODIFICATION.***

To add New features, we should not modify existing code.

Abstraction is the Key.

Model the behavior through an abstraction (interfaces/abstract classes)
Use concrete classes for extension

Testing is easy as we don't touch already available code

The idea was 1st given by Bertrand Meyer, in 1988.


*Image courtesy of Derick Bailey*

# O - for Open Closed Principle

S
O
L
I
D

```
enum ShapeType
{
    Circle,
    Square,
    Pentagon
};
class ShapeDrawer
{
    void drawCircle() { cout << "Draw Circle" << endl; }
    void drawSquare(){ cout << "Draw Square" << endl; }
    void drawPentagon(){ cout << "Draw Pentagon" << endl; }
public:
    void draw(ShapeType s)
    {
        if (s == Circle)
        {
            drawCircle();
        }
        else if (s = Square)
        {
            drawSquare();
        }
        else if (s = Pentagon)
        {
            drawPentagon();
        }
    }
};

int main()
{
    ShapeDrawer shapedrawer;
    shapedrawer.draw(Circle);
    shapedrawer.draw(Pentagon);
```

# O - for Open Closed Principle

S
O
L
I
D

```
class Shape
{
public:
    virtual void draw() = 0;
};

class Circle : public Shape
{
public:
    virtual void draw()
    {
        cout << "Draw Circle" << endl;
    }
};

class Square : public Shape
{
public:
    virtual void draw()
    {
        cout << "Draw Square" << endl;
    }
};

int main()
{
    Shape* s = new Circle();
    s->draw();
    s = new Square();
    s->draw();
```

# O - for Open Closed Principle - Tips

S
O
L
I
D

- Always use abstractions.
- If a single change in 1 program, causes a set of changes in other modules, OCP is violated.
- Functions with if blocks or switch statements checking type of sub class objects, are violating this rule.
- No program can be 100% closed. Always check the probability of different changes and apply OCP for most frequent changes.
- Heuristics related to OCP in OOD:
  - Make all member variables private
  - Don't use global variables
  - Prevent RTTI (run time type identification) including dynamic_cast and static_cast that violate OCP.
- Don’t follow OCP blindly. If to fix a simple bug in current code, don’t try OCP. But, may be to change code for a bug that is due to bad design.

# What is SOLID?

S - Single Responsibility Principle (SRP)

- Each Software Component should have only one reason to change

O – Open Closed Principle (OCP)

- Software Components should be closed for modification, but open for extension

**L – Liskov Substitution Principle (LSP)**

- **Objects should be replaceable with their subtypes without affecting the correctness of the program.**

I – Interface Segregation Principle (ISP)

D – Dependency Inversion Principle (DIP)

# L – Liskov Substitution Principle (LSP)

S
O
L
I
D

By Barbara Liskov in 1988 originally as

"*What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.*"

**Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.**

In simple words,

- ***If any subclass does nothing or do not have a sufficient overriding of any method in the super class, then it violates this principle.***
- ***All methods in super class must have a meaning at its sub class.***

**Inheritance is not merely a IS-A relationship. We should consider behaviors we Need.**


*Image courtesy of Derick Bailey*

# L – Liskov Substitution Principle

S
O
L
I
D

```
class Bird
{

public:
    void fly()
    {
        cout << "Bird flies" << endl;
    }
};

class Parrot: public Bird
{
 //this is okay
};

class Ostrich : public Bird
{
//this is wrong. Ostrich cant fly :(.
};

int main()
{
    Bird* b = new Parrot();
    b->fly();
    b = new Ostrich();
    b->fly();
}
```

# L – Liskov Substitution Principle

S
O
L
I
D

```
class Bird
{

public:

};

class FlyingBird : public Bird
{
public:
    void fly()
    {
        cout << "Bird flies" << endl;
    }
};

class Parrot: public FlyingBird
{
 //this is okay
};

class Ostrich : public Bird
{
//this is wrong. Ostrich cant fly :(.
};

int main()
{
    FlyingBird* b = new Parrot();
    b->fly();
    Bird* nb = new Ostrich();
    //nb->fly();
}
```

# L – Liskov Substitution Principle - Tips

S
O
L
I
D

- Prevent matching the real world (ISA relationship) always. Always think about what is the responsibility of the class and what functions and behaviors you have in a class in creating subclasses.
- How to solve the issue?
  - Break the hierarchy into more granular level.
  - Restructure code such that related class do the related functionality.

# What is SOLID?

S - Single Responsibility Principle (SRP)

- Each Software Component should have only one reason to change

O – Open Closed Principle (OCP)

- Software Components should be closed for modification, but open for extension

L – Liskov Substitution Principle (LSP)

- Objects should be replaceable with their subtypes without affecting the correctness of the program.

**I – Interface Segregation Principle (ISP)**

- **No Client should be forced to depend on methods it does not use**

D – Dependency Inversion Principle (DIP)

# I – Interface Segregation Principle (ISP)

S
O
L
I
D

***Clients should not be forced to depend upon interfaces that they do not use.***

Separate "fat interfaces" into abstract base classes that break unwanted coupling between clients.

Fat interfaces :

- a lot of method definitions in it.
- client who use it have to override all of them whether or not he need them

Use thin or small interfaces so that their reusability is high.

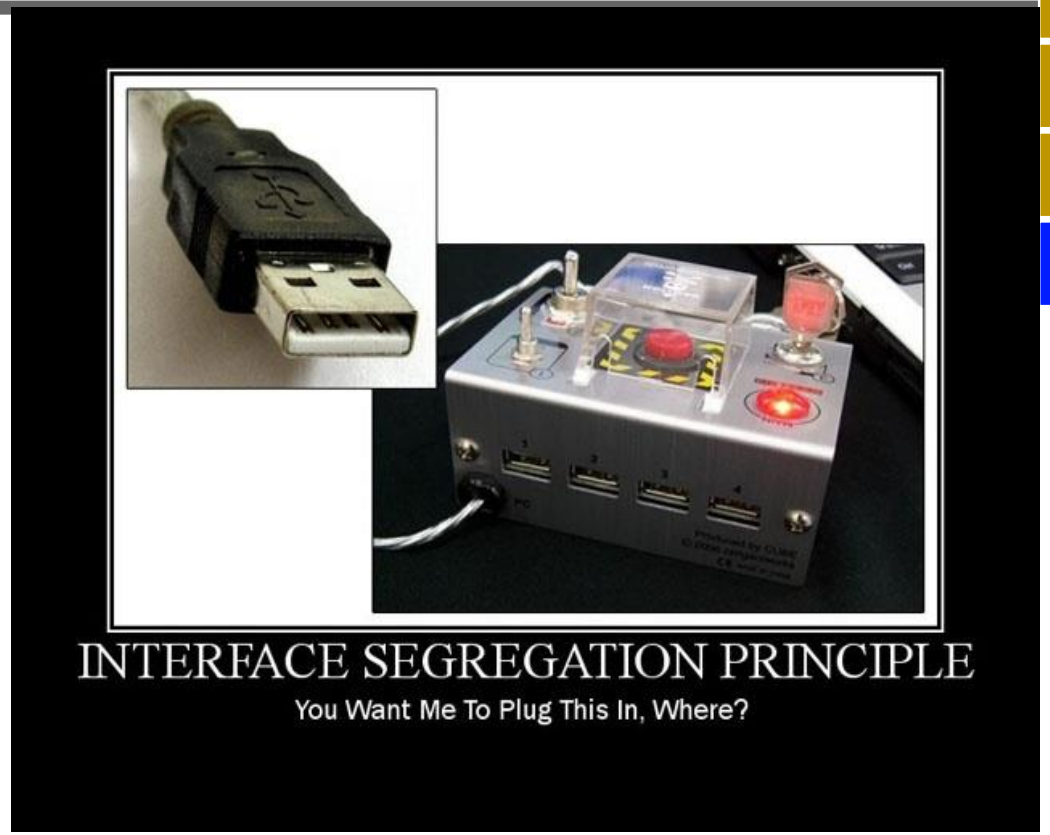



*Image courtesy of Derick Bailey*

# I – Interface Segregation Principle (ISP)

S
O
L
I
D

```
class  MultiFunctionMachine
{
public:
    virtual void photocopy() = 0;
    virtual void scan() = 0;
    virtual void print() = 0;
};

class MultiFunctionPhotocopyMachine : public MultiFunctionMachine
{
public:
    virtual void photocopy()
    {
        cout << "MultiFunction Photocopy Success" << endl;
    }
    virtual void scan()
    {
        cout << "MultiFunction Scan Success" << endl;
    }
    virtual void print()
    {
        cout << "MultiFunction Print Success" << endl;
    }
};

class Printer : public MultiFunctionMachine
{
public:
    virtual void photocopy()
    {
        cout << "Printer : Photocopy Not Implemented" << endl;
    }
    virtual void scan()
```

# I – Interface Segregation Principle

S
O
L
I
D

```
class  IPhotocopy
{
public:
    virtual void photocopy() = 0;
};
class IPrint
{
public:
    virtual void print() = 0;
};
class IScan
{
public:
    virtual void scan() = 0;
};

class MultiFunctionPhotocopyMachine : public IPhotocopy,IPrint, IScan
{
public:

    virtual void photocopy() override
    {
        cout << "MultiFunction Photocopy Success" << endl;
    }
    virtual void scan() override
    {
        cout << "MultiFunction Scan Success" << endl;
    }
    virtual void print() override
    {
        cout << "MultiFunction Print Success" << endl;
    }
```

# I – Interface Segregation Principle (ISP) - Tips

S
O
L
I
D

- Check for unimplemented methods in concrete classes. ISP violated.

  : Break interfaces to avoid unused methods.
- Check for FAT interfaces
- Check for interfaces with Low Cohesion : unrelated methods
- If interfaces follow SRP→ ISP too is preserved.

# What is SOLID?

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

- Each Software Component should have only one reason to change

**O – Open Closed Principle (OCP)**

- Software Components should be closed for modification, but open for extension

**L – Liskov Substitution Principle (LSP)**

- Objects should be replaceable with their subtypes without affecting the correctness of the program.

**I – Interface Segregation Principle (ISP)**

- No Client should be forced to depend on methods it does not use

**D – Dependency Inversion Principle (DIP)**

- **High Level Modules should not depend on low level modules. Both should depend on abstractions.**
- **Abstractions should not depend upon details. Details should depend upon abstractions**.

# D – Dependency Inversion Principle (DIP)

S
O
L
I
D

***A. High level modules should not depend upon low level modules. Both should depend upon abstractions.***

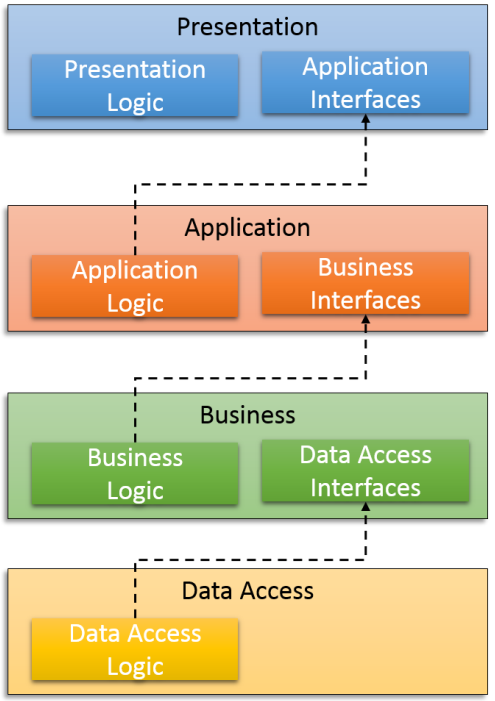
***B. Abstractions should not depend upon details. Details should depend upon abstractions.***

<u>High level modules:</u>

More closer to business logic

<u>Low level modules:</u>

More closer to low level implementation (databases/HW interfaces etc)



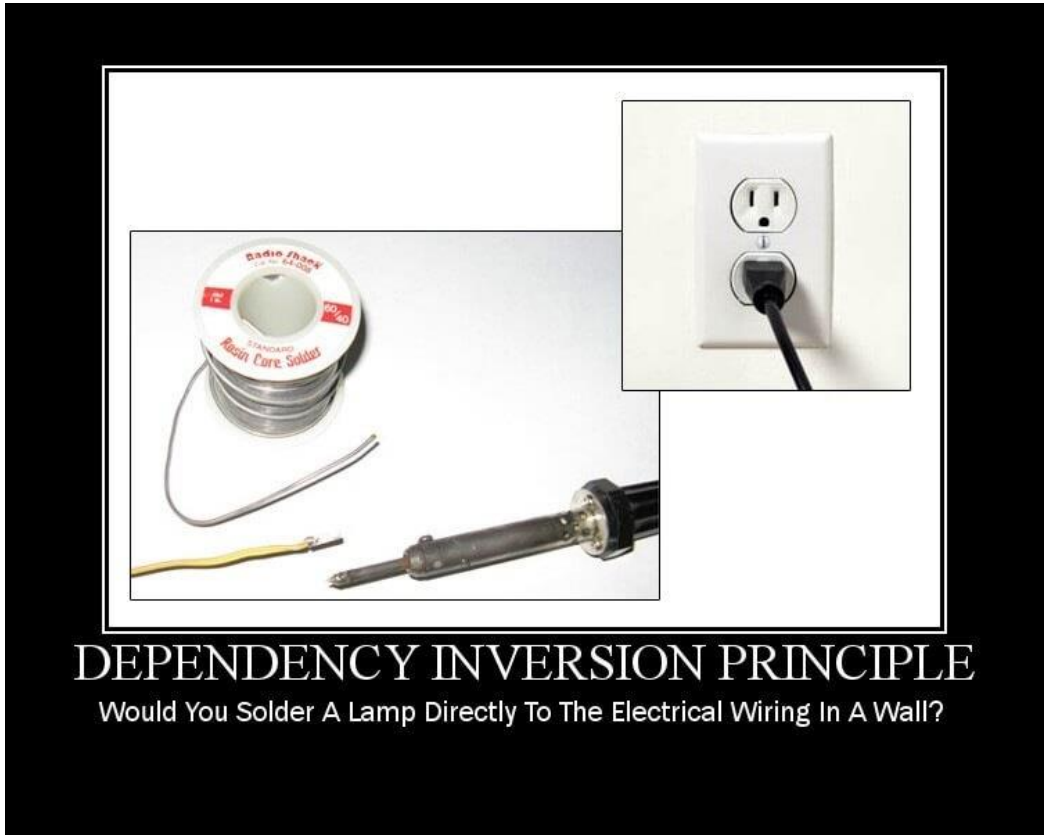




*Image courtesy of Derick Bailey*

LSEG

# D – Dependency Inversion Principle (DIP)

S
O
L
I
D

```
class CustomerDataAccess
{
public:
    string getCustomerName(int id) {
        return "Customer Name Derived from DB";
    }
};

class CustomerBusinessLogic
{
    CustomerDataAccess dataAccess;
public:
    string getCustomerName(int id)
    {
        return dataAccess.getCustomerName(id);
    }
};


int main()
{
    CustomerBusinessLogic customer;
    string name = customer.getCustomerName(1);
    cout << name << endl;

}
```

# D – Dependency Inversion Principle (DIP)

S
O
L
I
D

```
class ICustomerDataAccess
{
public:
    virtual string getCustomerName(int id) = 0;
};

class CustomerDataAccess : public ICustomerDataAccess
{
public:
    virtual string getCustomerName(int id) {
        return "Customer Name Derived from DB";
    }
};


class CustomerBusinessLogic
{
    ICustomerDataAccess* dataAccess = new CustomerDataAccess();
public:
    string getCustomerName(int id)
    {
        return dataAccess->getCustomerName(id);
    }
};


int main()
{
    CustomerBusinessLogic customer;
    string name = customer.getCustomerName(1);
    cout << name << endl;

}
```

LSEG

# What is SOLID?

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

- Each Software Component should have only one reason to change

**O – Open Closed Principle (OCP)**

- Software Components should be closed for modification, but open for extension

**L – Liskov Substitution Principle (LSP)**

- Objects should be replaceable with their subtypes without affecting the correctness of the program.

**I – Interface Segregation Principle (ISP)**

- No Client should be forced to depend on methods it does not use

**D – Dependency Inversion Principle (DIP)**

- High Level Modules should not depend on low level modules. Both should depend on abstractions.
- Abstractions should not depend upon details. Details should depend upon abstractions.

# Conclusion

- If OOP is like grammar, OOD is actually writing an essay with the grammar.
- **SOLID principles helps to design quality software which easy to maintain, extend and understand.**
- In addition to SOLID, there are several other principles that are used in OOP such as
  - GRASP : 9 fundamental principles in object design and responsibility assignment
    - Information expert.
    - Creator.
    - Low coupling.
    - Protected variations.
    - Indirection.
    - Polymorphism.
    - High cohesion.
    - Pure fabrication.
  - DRY : Don’t Repeat Yourself
    - Every piece of knowledge must have a single, unambiguous, authoritative representation within a system

# References

- Bob Martin SOLID Principles of Object Oriented and Agile Design
- Robert Martin SOLID report series (S, O ,L ,I ,D)
- Solid with Motivational Posters by DerickBailey
- How I explained OOD to my wife

# Design Patterns

- What are design patterns
- Benefits of design patterns
- Types of design patterns
  - Creational Design Patterns
  - Structural Design Patterns
  - Behavior Design Patterns

# **Singleton** (Creational)

- Only one instance of an object is created

```cpp
#include <stdio.h>
#include <iostream>
#include <string>

class Student
{
public:
    Student(const std::string& name) : m_name(name)
    {
    }
    ~Student() {}
private:
    const std::string m_name;
};

class StudentRegistry
{
public:
    static StudentRegistry& get() { return m_reg; }

    Student* find(const std::string name) { return nullptr; /*return student fond*/ }
    void add(Student* student) {}

private:
    StudentRegistry() {}
    ~StudentRegistry() {}

    static StudentRegistry m_reg;
};

StudentRegistry StudentRegistry::m_reg;

int main()
{
    auto& reg = StudentRegistry::get();
    auto student = reg.find("Student1");

    std::cin.get();
}
```

LSEG

# **Factory** (Creational)

- Creates objects without specifying the exact class to create

```cpp
#include <stdio.h>
#include <iostream>
#include <list>
#include <memory>

class Employee
{
public:
    Employee() {}
    virtual ~Employee() {}

    virtual void calculateSalary() = 0;
};

class FullTimeEmployee : public Employee
{
public:
    FullTimeEmployee() {}
    virtual ~FullTimeEmployee() {}

    void calculateSalary() override /*final also available*/
    {
        std::cout << "FullTimeEmployee - Salary calculation logic" << std::endl;
    }
};

class PartTimeEmployee : public Employee
{
public:
    PartTimeEmployee() {}
    virtual ~PartTimeEmployee() {}

    void calculateSalary() override /*final also available*/
    {
        std::cout << "PartTimeEmployee - Salary calculation logic" << std::endl;
    }
};
```

```cpp
class BaseEmployeeFactory
{
public:
    virtual std::unique_ptr<Employee> createEmployee(std::string type) = 0;
};

class EmployeeFactory : public BaseEmployeeFactory
{
public:
    std::unique_ptr<Employee> createEmployee(std::string type) override
    {
        if (type == "FullTime")
        {
            return std::make_unique<FullTimeEmployee>();
        }
        else if (type == "PartTime")
        {
            return std::make_unique<PartTimeEmployee>();
        }
        else
        {
            std::cerr << "Unknown type" << std::endl;
            return nullptr;
        }
    }
};

int main()
{
    std::list<std::unique_ptr<Employee>> employees;
    std::unique_ptr<BaseEmployeeFactory> factory = std::make_unique<EmployeeFactory>();
    employees.push_back(std::move(factory->createEmployee("FullTime")));
    employees.push_back(std::move(factory->createEmployee("PartTime")));

    for (auto& employee : employees)
    {
        employee->calculateSalary();
    }

    std::cin.get();
}
```

```
FullTimeEmployee - Salary calculation logic
PartTimeEmployee - Salary calculation logic
```

# **Adapter** (Structural)

- Allows for two incompatible classes to work together
- Wrapping an interface around one of the existing classes

```cpp
#include <stdio.h>
#include <iostream>
#include <memory>
#include <string>

struct Message
{
};

class Encoder
{
public:
    Encoder() {}
    virtual ~Encoder() {}
    virtual const std::string encode(Message& msg) = 0;
};

class JsonEncoder : public Encoder
{
public:
    JsonEncoder() {}
    virtual ~JsonEncoder() {}
    const std::string encode(Message& msg) override
    {
        std::cout << "JsonEncoder::encode" << std::endl;
        return "JsonEncoder - encoded buffer";
    }
};

class OldXMLEncoder
{
public:
    OldXMLEncoder() {}
    virtual ~OldXMLEncoder() {}
    const void encode(Message& msg, std::string& encoded)
    {
        std::cout << "OldXMLEncoder::encode" << std::endl;
        encoded = "OldXMLEncoder - encoded buffer";
    }
};
```

```cpp

class XMLEncoder : public Encoder
{
public:
    XMLEncoder() {}
    virtual ~XMLEncoder() {}

    const std::string encode(Message& msg) override
    {
        std::cout << "XMLEncoder::encode" << std::endl;
        std::string encodedBuffer;
        m_oldEncoder.encode(msg, encodedBuffer);
        return encodedBuffer;
    }

private:
    OldXMLEncoder m_oldEncoder;
};


int main()
{
    Message msg;
    std::unique_ptr<Encoder> e1 = std::make_unique<JsonEncoder>();
    std::cout << e1->encode(msg) << std::endl;

    std::cout << std::endl;
    std::unique_ptr<Encoder> e2 = std::make_unique<XMLEncoder>();
    std::cout << e2->encode(msg) << std::endl;

    std::cin.get();
}
```

```
JsonEncoder::encode
JsonEncoder - encoded buffer

XMLEncoder::encode
OldXMLEncoder::encode
OldXMLEncoder - encoded buffer
```

# **Visitor** (Behavior)

- Separates an algorithm from an object structure

```cpp
#include <stdio.h>
#include <iostream>
#include <memory>

class FullTimeEmployee;
class PartTimeEmployee;

class EmployeeVisitor
{
public:
    EmployeeVisitor() {}
    virtual ~EmployeeVisitor() {}

    virtual void visit(FullTimeEmployee& employee) = 0;
    virtual void visit(PartTimeEmployee& employee) = 0;
};

class Employee
{
public:
    Employee() {}
    virtual ~Employee() {}

    virtual float getRate() = 0;
    virtual float getTotalHours() = 0;

    virtual void accept(EmployeeVisitor& visitor) = 0;
};

class FullTimeEmployee : public Employee
{
public:
    FullTimeEmployee() {}
    virtual ~FullTimeEmployee() {}

    float getRate() override { return 350.0; }
    float getTotalHours() override { return 150.0; }
    void accept(EmployeeVisitor& visitor) override { visitor.visit(*this); }
    float getAllowance() { return 10000.0; }
};

class PartTimeEmployee : public Employee
{
public:
    PartTimeEmployee() {}
    virtual ~PartTimeEmployee() {}
    float getRate() override { return 250.0; }
    float getTotalHours() override { return 100.0; }
    void accept(EmployeeVisitor& visitor) override { visitor.visit(*this); }
};

class EmployeeVisitorImpl : public EmployeeVisitor
{
public:
    EmployeeVisitorImpl() {}
    virtual ~EmployeeVisitorImpl() {}

    void visit(FullTimeEmployee& employee) override
    {
        float salary = (employee.getRate() * employee.getTotalHours()) + employee.getAllowance();
        std::cout << "Visiting FullTimeEmployee, Salary: " << salary << std::endl;
    }

    void visit(PartTimeEmployee& employee) override
    {
        float salary = (employee.getRate() * employee.getTotalHours());
        std::cout << "Visiting PartTimeEmployee, Salary: " << salary << std::endl;
    }
};

int main()
{
    std::unique_ptr<Employee> e1 = std::make_unique<FullTimeEmployee>();
    std::unique_ptr<Employee> e2 = std::make_unique<PartTimeEmployee>();
    std::unique_ptr<EmployeeVisitor> visitor = std::make_unique<EmployeeVisitorImpl>();
    e1->accept(*visitor);
    e2->accept(*visitor);

    std::cin.get();
}
```

```
Visiting FullTimeEmployee, Salary: 62500
Visiting PartTimeEmployee, Salary: 25000
```

# **Observer** (Behavior)

- Is a publish/subscribe pattern

```cpp
#include <stdio.h>
#include <iostream>
#include <memory>
#include <list>
#include <string>

class Socket;
class SocketObserver
{
public:
    SocketObserver() {}
    virtual ~SocketObserver() {}
    virtual void update(Socket& socket, const std::string data) = 0;
    //Instead of update can use meaningful name like onData, onRecvData, etc ...
};

class Socket
{
public:
    Socket() {}
    virtual ~Socket() {}
    void onData(const std::string data)
    {
        notifyObservers(data);
    }

    void registerObserver(std::shared_ptr<SocketObserver> observer)
    {
        m_observers.push_back(observer);
    }

private:
    void notifyObservers(const std::string data)
    {
        for (auto observer : m_observers)
        {
            observer->update(*this, data);
        }
    }
    std::list<std::shared_ptr<SocketObserver>> m_observers;
};

class WindowObserver : public SocketObserver
{
public:
    WindowObserver() {}
    virtual ~WindowObserver() {}
    void update(Socket& socket, const std::string data) override
    {
        std::cout << "Data received by WindowObserver. Data:" << data << std::endl;
    }
};

class ContainerObserver : public SocketObserver
{
public:
    ContainerObserver() {}
    virtual ~ContainerObserver() {}
    void update(Socket& socket, const std::string data) override
    {
        std::cout << "Data received by ContainerObserver. Data:" << data << std::endl;
    }
};

int main()
{
    std::unique_ptr<Socket> socket = std::make_unique<Socket>();
    std::shared_ptr<SocketObserver> observer1 = std::make_shared<WindowObserver>();
    std::shared_ptr<SocketObserver> observer2 = std::make_shared<ContainerObserver>();
    socket->registerObserver(observer1);
    socket->registerObserver(observer2);

    socket->onData("Test data");

    std::cin.get();
}
```

```
Data received by WindowObserver. Data:Test data
Data received by ContainerObserver. Data:Test data
```

# THANK YOU