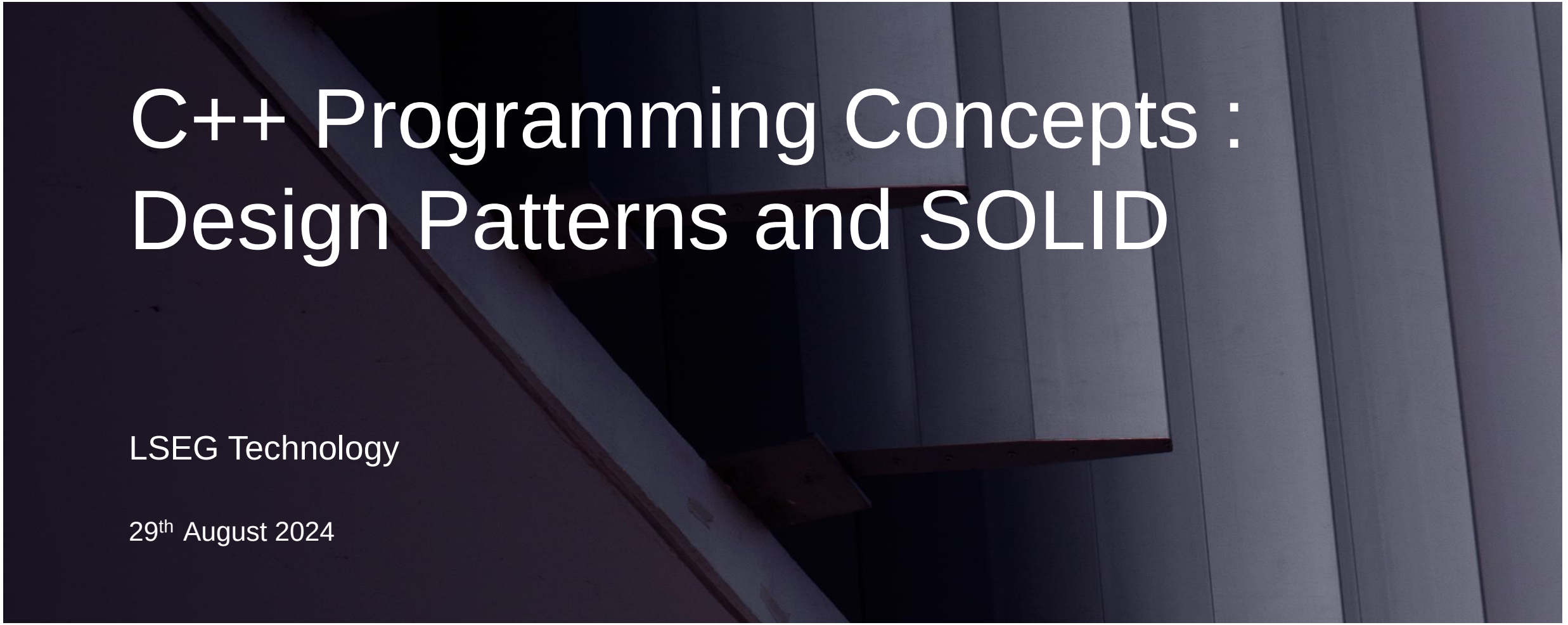


C++ Programming Concepts : Design Patterns and SOLID

LSEG Technology

29th August 2024

LSEG

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 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

S
O
L
I
D

```
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. …

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);
    }

};
```

S
O
L
I
D

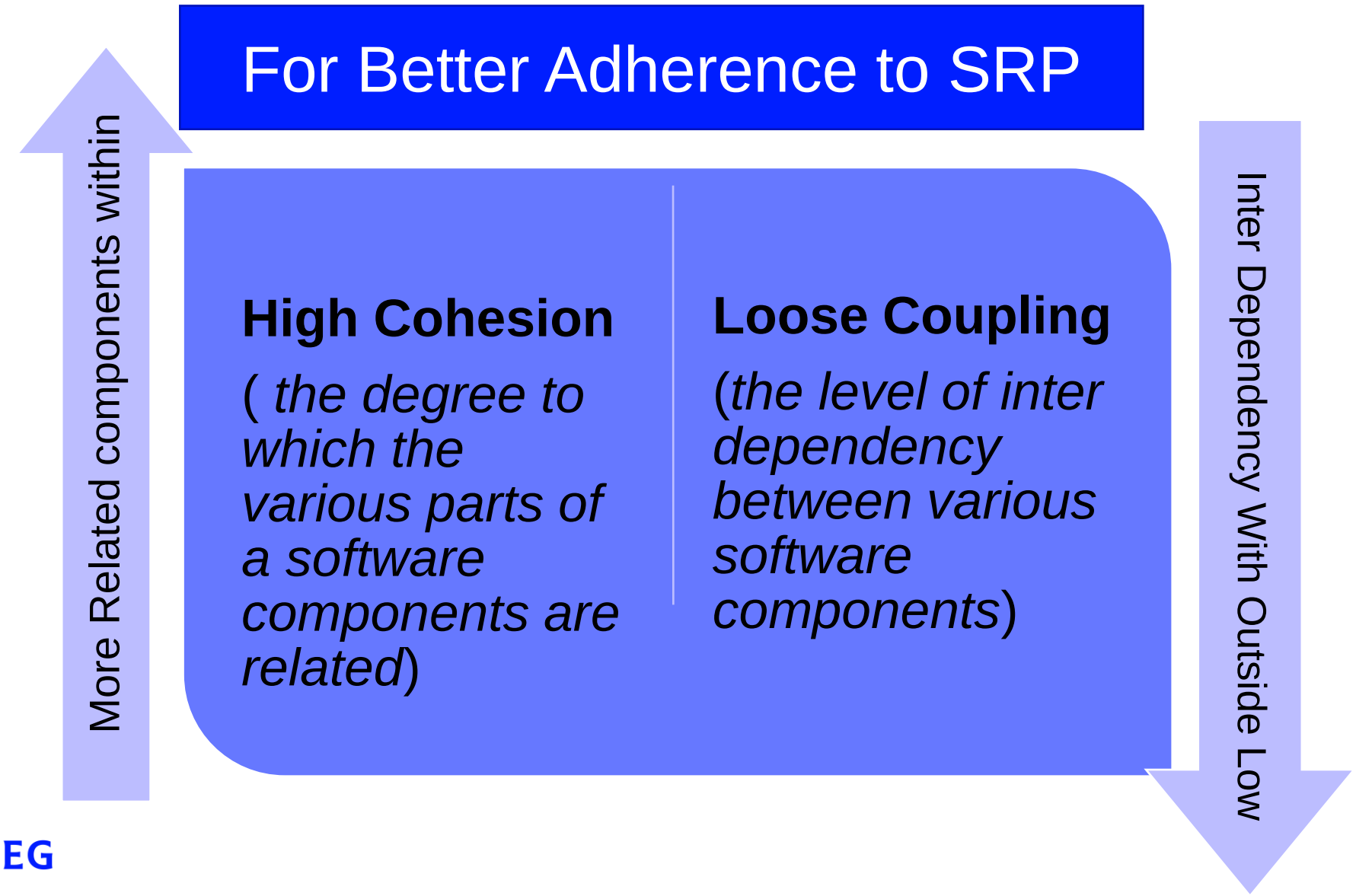
Classes have only 1 responsibility to each.

Student: keeps track of student information

DBConnection: handles DB queries

EmailSender: handle email related stuff

S - for Single Responsibility Principle (SRP)



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 interfaces 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.

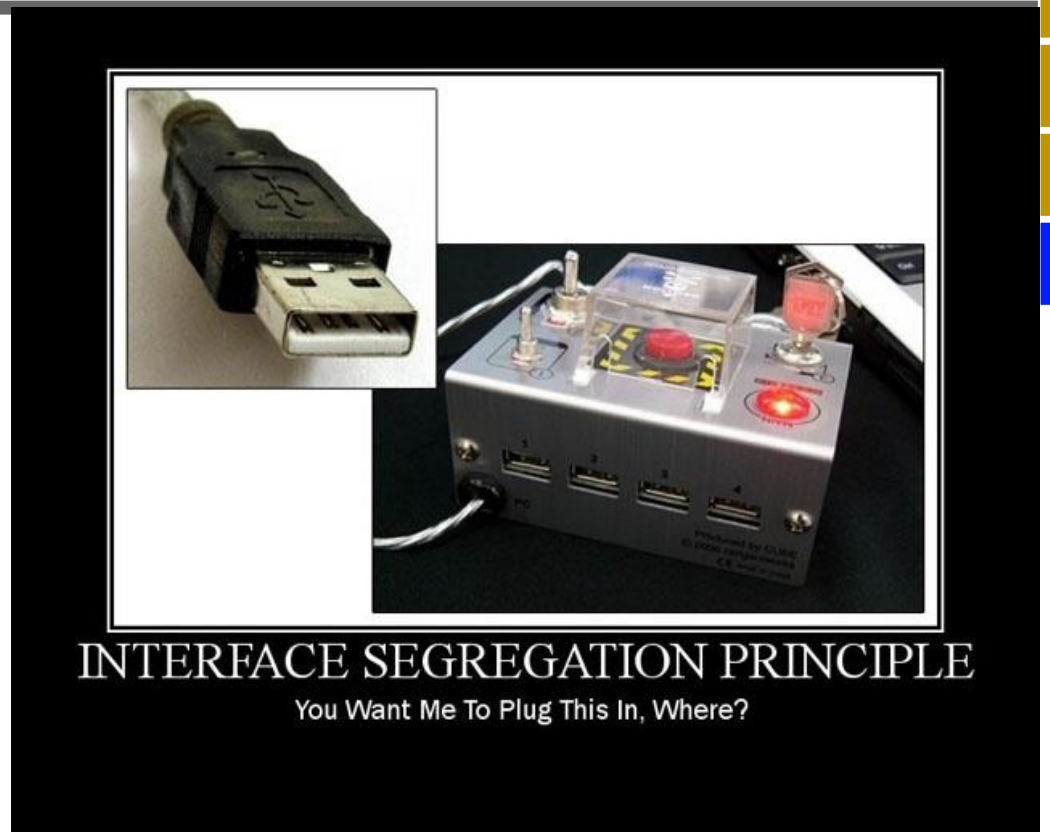


Image courtesy of Derick Bailey

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.

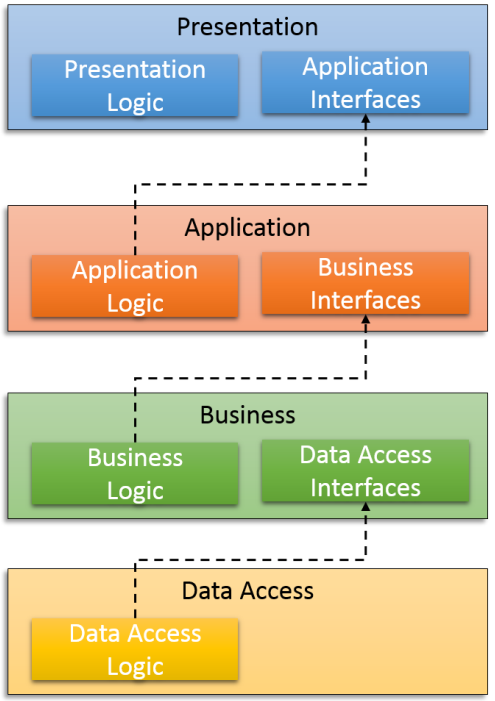
B. Abstractions should not depend upon details. Details should depend upon abstractions.

<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)



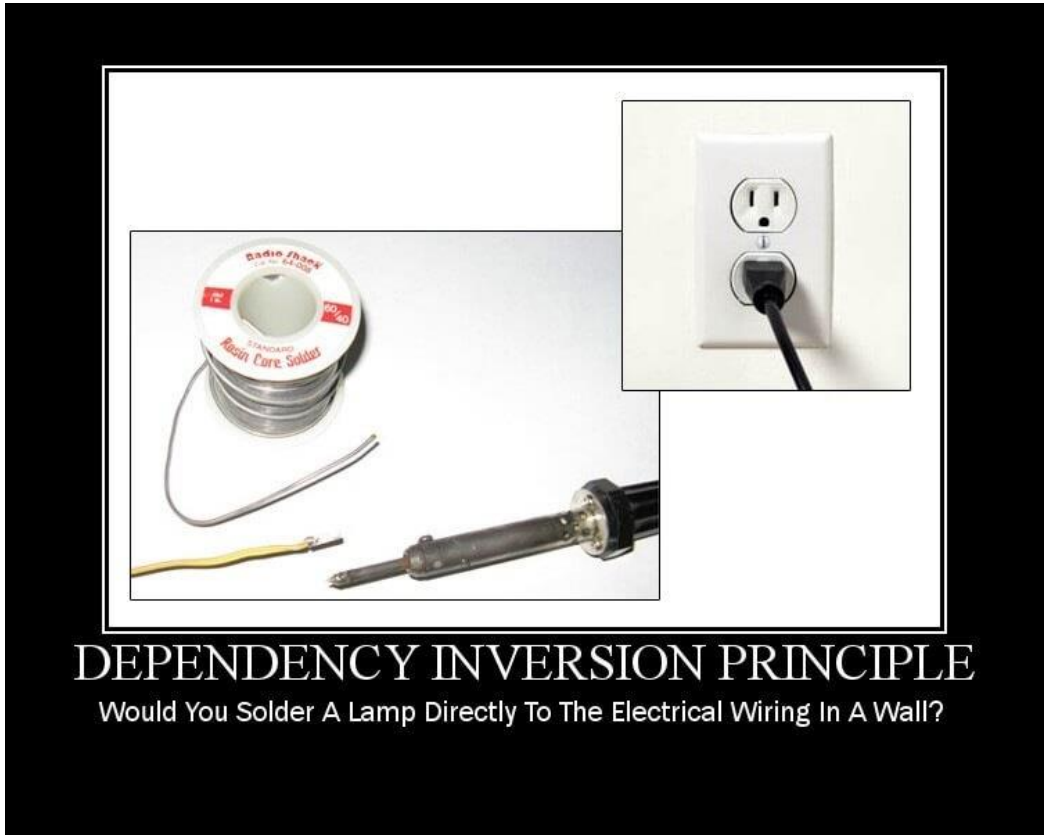


Image courtesy of Derick Bailey

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

THANK YOU