CppIndia Tech Talks

Let’s have a SOLID start

SOLID Principles Unleashed

26 Feb 2022

Honey Sukesan

Senior Software Developer

Good housekeeping

Some guidelines to facilitate a smooth virtual session

1

Please have your camera on. Seeing faces really support a connected discussion.

2

Please have your microphone on mute while you are not speaking.

3

Feel free to ask any questions by unmuting or post It in the meeting chat.

Question

Roadmap

Our agenda for this session

- SOLID principles - Why
- SOLID principles - What
- SOLID principles - How
- SOLID principles – Who & When
- SOLID principles – Which & What
- SOLID principles – Where & How

Why SOLID principles ?

Goals of SOLID principles

Good software systems begin with clean code.

SOLID principles are guidelines to clean code.

Its goal is the creation of **mid-level software** structures that:

✓ Tolerate change

✓ Are easy to understand

✓ Are the basis of components that can be used in many software systems

What are SOLID principles?

SOLID principles can be considered as Universal set of design principles.

They are applicable to Object oriented, Generic and Functional programming paradigms.

How SOLID principles help?

SOLID principles tell us how to arrange our functions and data structures into classes, and how those classes should be interconnected.

Who & When ?

Its origin

SOLID principles are a subset from many software design principles promoted by **Robert C. Martin** (Uncle Bob)

Michael Feathers suggested the idea of rearranging the order so that its first letters would spell the word **SOLID**.

SOLID principles

Single Responsibility Principle

Open - Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

Single Responsibility Principle (SRP)

High cohesion

> “A module should have one, and only one, reason to change.”
>
> - Robert C. Martin, Clean Architecture

Cohesion

“**Cohesion** is a measure of the strength of association of the elements inside a module. A highly cohesive module is a collection of statements and data items that should be treated as a whole because they are so closely related. Any attempt to divide them up would only result in increased coupling and decreased readability.”

- Tom DeMarco, Structured Analysis and System Specification

SRP violation - example

```
struct PhotoUploader {

  PhotoUploader (Camera& camera, CloudProvider& cloudProvider);

  Image getImage();

  Image resizeImage(int x, int y, Image image);

  void uploadImage(Image& image);

};
```

SRP – A better way

```
struct PhotoUploader {

  PhotoUploader (CloudProvider& cloudProvider);

  void uploadImage(Image& image);

};
```

SRP – More examples

The design of STL follows Single Responsibility Principle.

eg: Data structures, Algorithms

Single Responsibility Principle - takeaway

SRP is one of the simplest of the principles & one of the hardest to get right.

Prefer cohesive software entities.

Everything that does not strictly belong together, should be separated.

Open Closed Principle (OCP)

Simplifying addition / extension of functionality

> “A software artifact should be open for extension but closed for modification.”
>
> - Bertrand Meyer

OCP violation - example

Procedural Approach

```
//shape.h
enum ShapeType {circle, square};

struct Shape
{
    ShapeType itsType;
};
```

```
//square.h
struct Square : public Shape
{
  ShapeType itsType;
  double itsSide;
  Point itsTopLeft;
};

void DrawSquare(struct Square*);
```

```
//circle.h

struct Circle : public Shape
{
  ShapeType itsType;

  double itsRadius;

  Point itsCenter;
};

void DrawCircle(struct Circle*);
```

OCP violation - example

```
//draw_shapes.cpp
void DrawAllShapes(struct Shape *list[], int n)
{
  for (int i = 0; i < n; i++)
  {
    struct Shape* s = list[i];
    switch (s->itsType)
    {
      case square:
        DrawSquare((struct Square*)s);
      break;

      case circle:
        DrawCircle((struct Circle*)s);
      break;
    }
  }
}
```

OCP – A better way

Object oriented approach

```
//shape.h
class Shape
{
    public:
        virtual void Draw() const = 0;
};
```

```
//square.h
class Square : public Shape
{
    public:
        virtual void Draw() const;
};
```

```
//circle.h
class Circle : public Shape
{
    public:
        virtual void Draw() const;
};
```

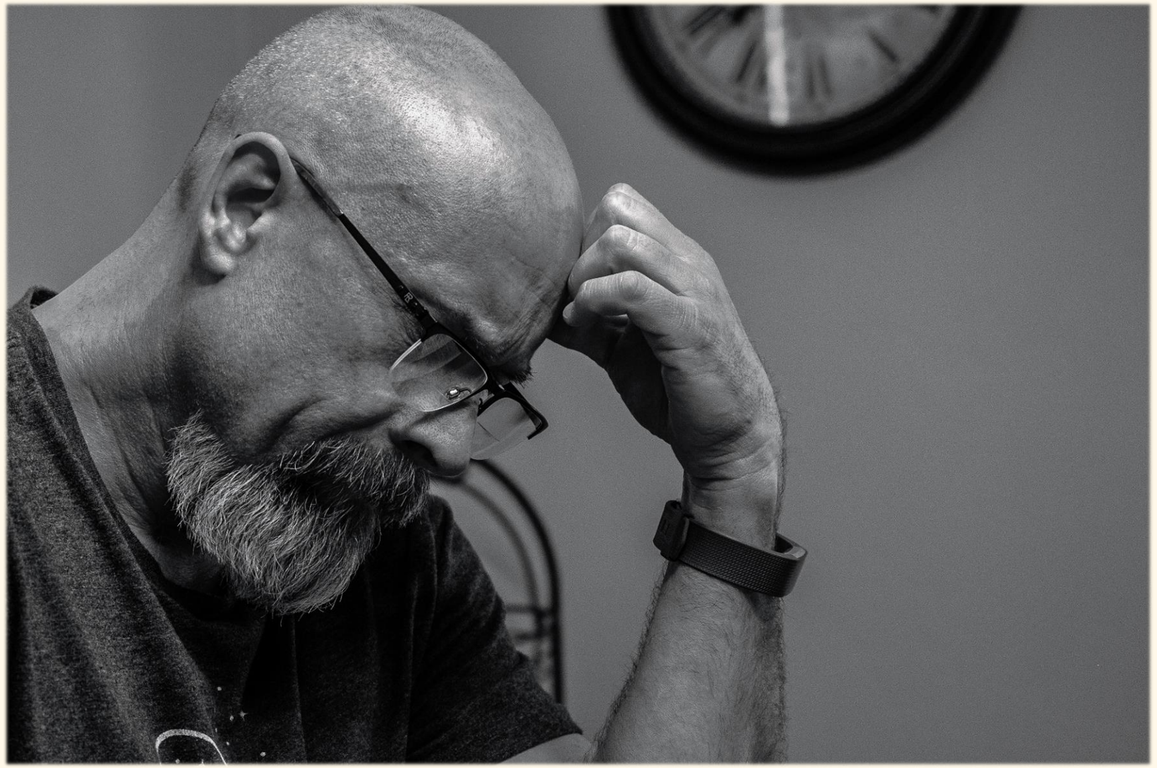
OCP – A better way

Object oriented approach

```cpp
//draw_shapes.cpp
void DrawAllShapes(std::vector<Shape*>& list)
{
    std::vector<Shape*>::iterator i;

    for (i = list.begin(); i != list.end(); i++)
        (*i)->Draw();
}
```

Food for thought!

Do you think the object-oriented approach can handle any type of requirement changes conforming to OCP?



OCP – The hard truth

In general, no matter how “closed” a module is, there will always be some kind of change against which it is not closed.

There is no model that is natural to all contexts!

Since closure cannot be complete, it must be strategic.

Open-closed principle - takeaway

Prefer software design that allows the addition of types or operations without the need to modify existing code

Conformance to OCP yields the greatest benefits claimed for object-oriented paradigm.

OCP is accomplished by identifying & applying abstraction only to those parts of the program that exhibit frequent change.

Liskov Substitution Principle (LSP)

Substitutability

> “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 behaviour of P is unchanged when o1 is substituted for o2, then S is a subtype of T.”
>
> - Barbara Liskov

Liskov Substitution Principle (LSP)

“Subtypes must be substitutable for their base types”

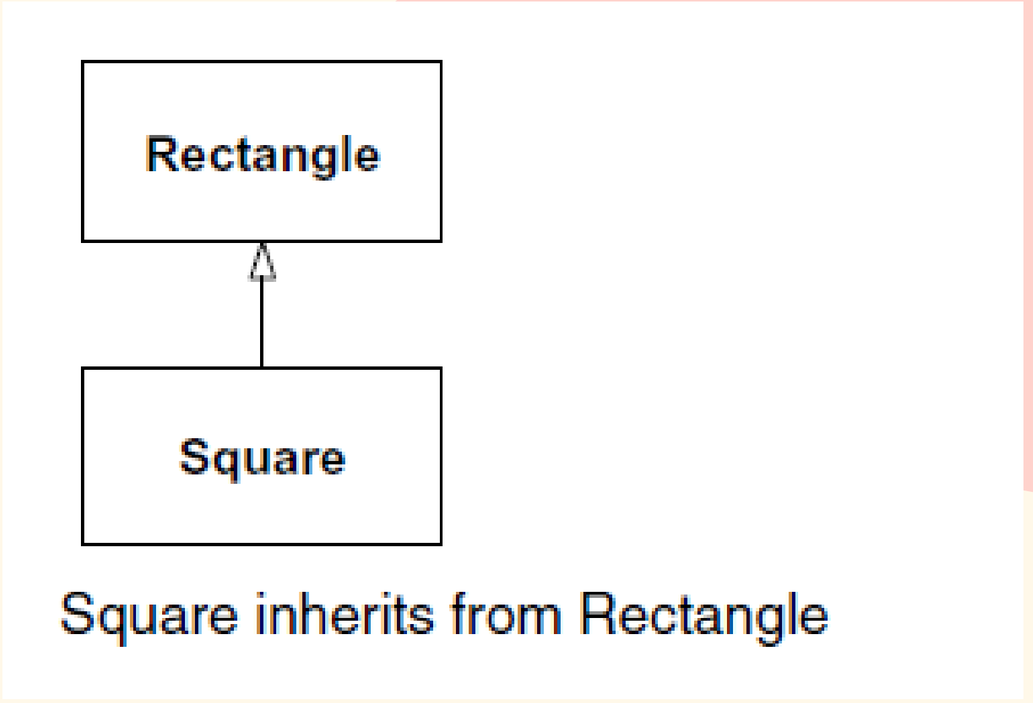
LSP violation - example

Square Rectangle problem

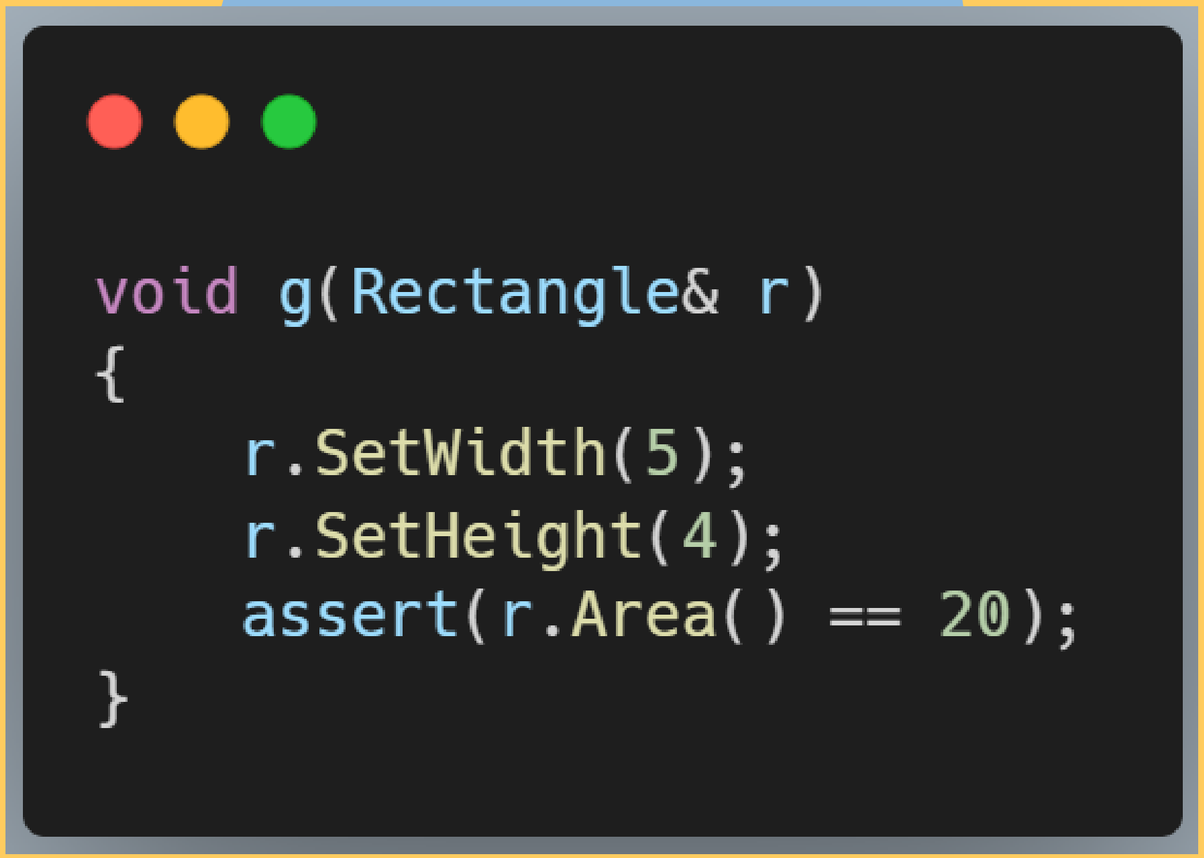
```
//rectangle.h
class Rectangle
{
    public:
        void SetWidth(double w) {itsWidth = w;}
        void SetHeight(double h) {itsHeight = w;}
        double GetHeight() const {return itsHeight;}
        double GetWidth() const {return itsWidth;}

    private:
        Point itsTopLeft;
        double itsWidth;
        double itsHeight;
};
```

LSP violation - example

```cpp
class Square : public Rectangle
{
    public:
        virtual void SetWidth(double w);
        virtual void SetHeight(double h);
};

void Square::SetWidth(double w)
{
    Rectangle::SetWidth(w);
    Rectangle::SetHeight(w);
}

void Square::SetHeight(double h)
{
    Rectangle::SetHeight(h);
    Rectangle::SetWidth(h);
}
```



LSP - Real problem



```
void g(Rectangle& r)
{
    r.SetWidth(5);
    r.SetHeight(4);
    assert(r.Area() == 20);
}
```

LSP – Another example

Templates

The Liskov Substitution Principle (LSP)

```
namespace std {

template< typename InputIt, typename OutputIt >
OutputIt copy( InputIt first, InputIt last, OutputIt dest )
{
   for( ; first != last; ++first, ++dest ) {
      *dest = *first;
   }

   return dest;
}

} // namespace std
```

Liskov substitution principle - takeaway

LSP is all about substitutability.
The term “Is A” is too broad to act as a definition for subtype.

Make sure inheritance is about behavior, not about data.
Make sure the expectations of the base class is adhered to.

Interface Segregation Principle (ISP)

Low coupling

> “Clients should not be forced to depend on methods that they do not use.”
>
> - Robert C. Martin

ISP violation - example

```
//bird.h
class Bird {
    public:
        virtual void fly() = 0;
        virtual void eat() = 0;
        virtual void run() = 0;
        virtual void tweet() = 0;
};
```

ISP – violation example

```cpp
//sparrow.cpp
class Sparrow : public Bird {

    public:

    virtual void fly() override {
    //...
    }
    virtual void eat() override {
    //...
    }
    virtual void run() override {
    //...
    }
    virtual void tweet() override {
    //...
    }
};
```

```cpp
//penguin.cpp
class Penguin : public Bird {

    public:
    virtual void fly() override {
    // ???
    }
    //...
};
```

ISP example - Better alternative

```
class Lifeform {
    public:
        virtual void eat() = 0;
        virtual void move() = 0;
    };

class Flyable {
    public:
        virtual void fly() = 0;
};

class Audible {
    public:
        virtual void makeSound() = 0;
};
```

```
class Sparrow : public Lifeform, public Flyable, public Audible {
//...
};

class Penguin : public Lifeform, public Audible {
//...
};
```

Interface segregation principle - takeaway

Make sure interfaces are not bloated with member functions that sub-classes are unable to implement in a meaningful way.

Make sure interfaces don’t induce unnecessary dependencies.

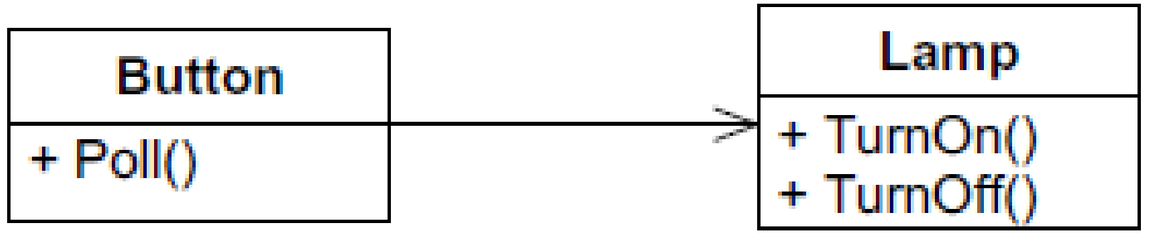
Dependency-Inversion Principle (DIP)

Loose coupling

> “a. High-level modules should not depend on low-level modules.
> Both should depend on abstractions.
>
> b. Abstractions should not depend on details.
> Details should depend on abstractions.”
>
> - Robert C. Martin

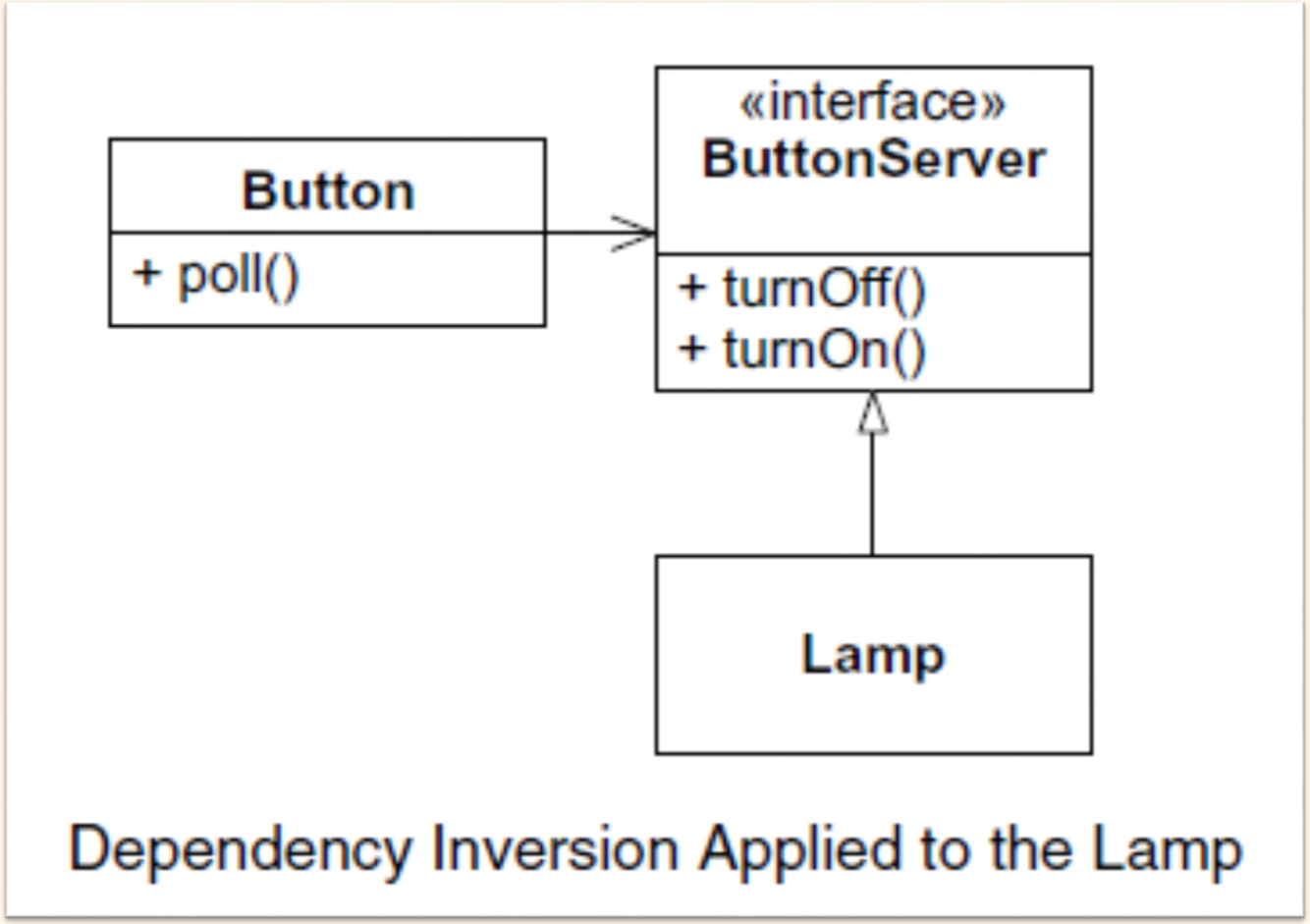
DIP violation - example

```cpp
//button.cpp
class Button
{
    private:
        Lamp itsLamp;

    public:
        void poll()
        {
            if (/*some condition*/)
                itsLamp.turnOn();
        }
};
```



Naive Model of a Button and a Lamp

DIP example – A better approach



Dependency Inversion Applied to the Lamp

Dependency inversion principle - takeaway

The hallmark of good object-oriented design is dependency inversion.

No matter what language a program is written :
If its dependencies are inverted, it has an OO design.
If its dependencies are not inverted, it has a procedural design.

Prefer to depend on abstractions instead of concrete types.

Your key takeaways

3- minute activity

SOLID principles - recap

Summary

SOLID principles enable agile software development.

SOLID principles are not just a set of object-oriented guidelines.

Use SOLID principles to reduce coupling and facilitate change.

Any questions ?

Thank you for listening!

Feel free to send in any questions to CppIndia Discord channel.

You can reach me at honey.sambhu@gmail.com

Honey Sukesan | LinkedIn

https://twitter.com/HSukesan

honey-speaks-tech (Honey Sukesan) (github.com)