

S.O.L.I.D

5 PRINCIPLES FOR CLEAN CODE

S

Single Responsibility

O

Open/Closed

L

Liskov Substitution

I

Interface Segregation

D

Dependency Inversion


ORIGIN

The SOLID principles control the relationships and operations between classes in object orientated languages

First developed by Robert C. Martin aka 'Uncle Bob' in a 2000 essay, "Design Principles and Design Patterns,"

The actual 'SOLID' acronym was coined later by Michael Feathers.

They describe the way classes in OOD relate to one another , about the dependencies between those classes and the motivations for creating those dependencies

Several white lines of varying lengths and orientations are positioned on the right side of the slide, extending from the middle to the bottom right corner.

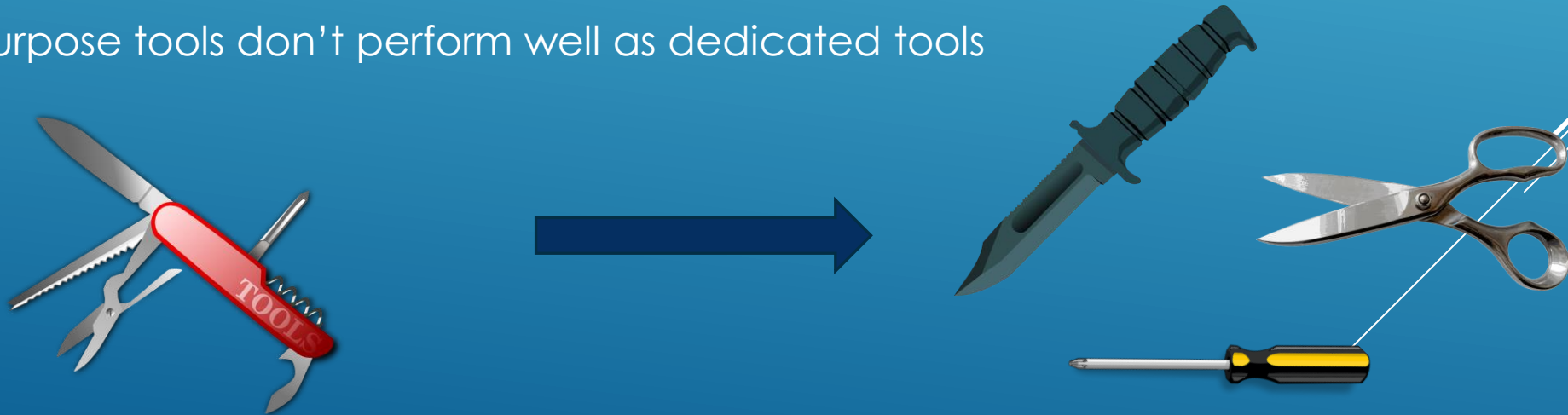
SINGLE RESPONSIBILITY

"Each software module should only have one reason to change."

The individual classes and methods define WHAT the application does, and HOW it does it.

Can often improve the design of software by separating out the WHAT from the HOW.

Multipurpose tools don't perform well as dedicated tools



SINGLE RESPONSIBILITY

What is a responsibility?

Decision the code is making about the specific implementation details of some part of what the application does e.g:



Persistence



Logging



Validation

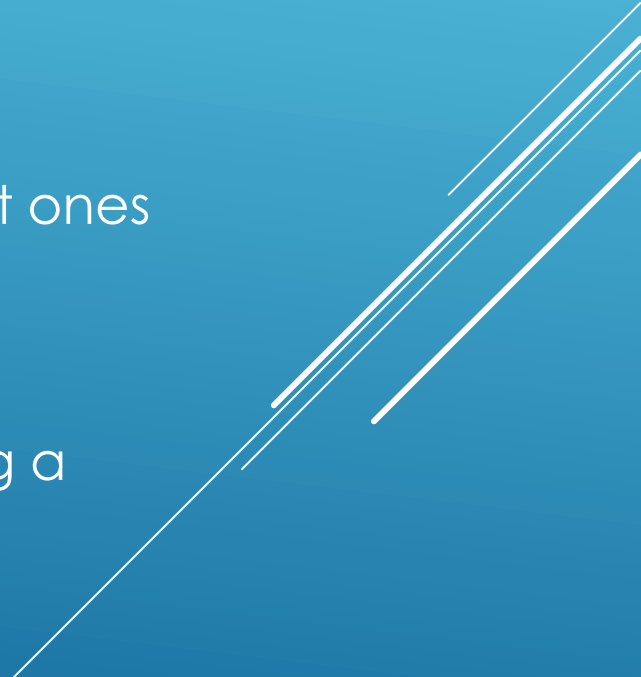


Business Logic

Responsibilities can change at different times for different reasons

SINGLE RESPONSIBILITY

Advantages

- Helps reduce tight COUPLING
(when two (or more) details are bound together in a way that's difficult to change)
 - Improves COHESION
(grouping class elements that belong together – separating out ones that don't)
 - Helps adhere to the 'Separation of Concerns' principle
(Programs should be separated into distinct sections addressing a separate concern or set of information)
- 
- A series of white diagonal lines of varying lengths and thicknesses are positioned on the right side of the slide, extending from the middle towards the bottom right corner.

SINGLE RESPONSIBILITY

Example violation:

```
class User
{
    void CreatePost(Database db, string postMessage)
    {
        try
        {
            db.Add(postMessage);
        }
        catch (Exception ex)
        {
            db.LogError("An error occurred: ", ex.ToString());
            File.WriteAllText(@"\LocalErrors.txt", ex.ToString());
        }
    }
}
```



```
class Post
{
    private ErrorLogger errorLogger = new ErrorLogger();

    void CreatePost(Database db, string postMessage)
    {
        try
        {
            db.Add(postMessage);
        }
        catch (Exception ex)
        {
            errorLogger.log(ex.ToString())
        }
    }
}

class ErrorLogger
{
    void log(string error)
    {
        db.LogError("An error occurred: ", error);
        File.WriteAllText(@"\LocalErrors.txt", error);
    }
}
```


OPEN/CLOSED

“Software entities should be open for extension, but closed for modification”

It should be possible to change the behaviour of a method without editing its source code

Open for extension:

New behaviour can be added in future

Closed for modification:

Unnecessary to change the source or binary code

Code that is closed for extension has fixed behaviour

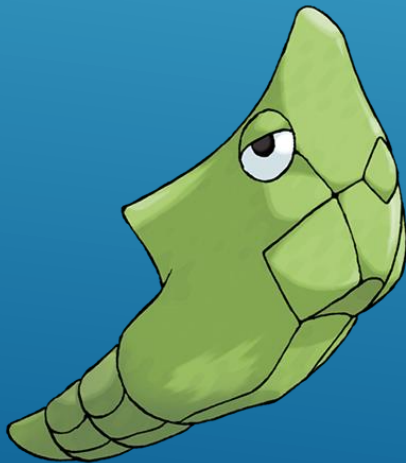
The only way to change the behaviour of code closed to extension is to **MODIFY** the code itself

OPEN/CLOSED

Need to balance abstraction/extensibility and concreteness

Extreme concreteness

Does exactly one thing, one way. The only way to change it's behaviour, is to change it's code



Extreme abstraction/extensibility

Can do anything, doesn't do anything itself. All its functionality is passed into it.



OPEN/CLOSED

Advantages

- Less likely to introduce bugs in code that isn't touched/redeployed
- Can build a new class for new features, which enables:
 - easier testing
 - SRP adherence
 - nothing will depend on it
 - add behaviour without touching existing code
- Often results in simpler code – fewer conditionals

OPEN/CLOSED

Example violation

```
class Post
{
    void CreatePost(Database db, string postMessage)
    {
        if (postMessage.StartsWith("#"))
        {
            db.AddAsTag(postMessage);
        }
        else
        {
            db.Add(postMessage);
        }
    }
}
```



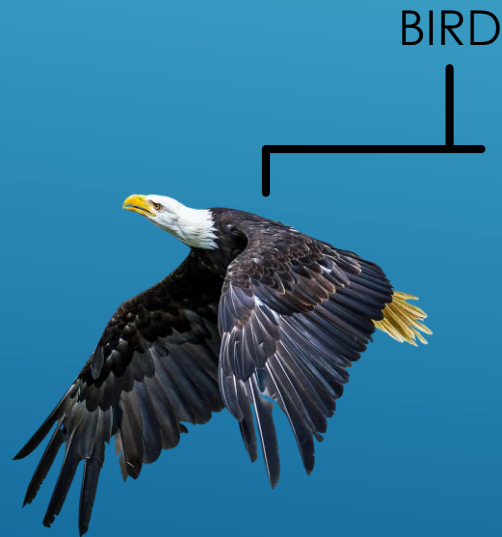
```
class Post
{
    void CreatePost(Database db, string postMessage)
    {
        db.Add(postMessage);
    }
}

class TagPost : Post
{
    override void CreatePost(Database db, string postMessage)
    {
        db.AddAsTag(postMessage);
    }
}
```

LISKOV SUBSTITUTION

“Subtypes must be substitutable for their base types”

IS-A relationship:



LSP states

The 'IS-A' relationship is insufficient, and should be replaced with 'IS-SUBSTITUTABLE-FOR'

LISKOV SUBSTITUTION

Advantages

Helps us properly use inheritance

Fewer bugs



LSKOV SUBSTITUTION

The square/rectangle example

```
public class Rectangle
{
    public virtual int Height { get; set; }
    public virtual int Width { get; set; }
}
```

```
public class AreaCalculator
{
    public static int CalculateArea(Rectangle r)
    {
        return r.Height * r.Width;
    }
}
```

```
public class Square : Rectangle
{
    private int _height;
    public override int Height
    {
        get { return _height; }
        set
        {
            _width = value;
            _height = value;
        }
    }
    // Width implemented similarly
}
```

```
Rectangle myRect = new Square();
myRect.Width = 4;
myRect.Height = 5;

Assert.Equal(20, AreaCalculator.CalculateArea(myRect));

// Actual Result: 25
```

LSKOV SUBSTITUTION

The square/rectangle solutions

```
public class Rectangle
{
    public int Height { get; set; }
    public int Width { get; set; }
    public bool IsSquare => Height == Width;
}
```

```
public class Rectangle
{
    public int Height { get; set; }
    public int Width { get; set; }
}

public class Square
{
    public int Side { get; set; }
}
```


LISKOV SUBSTITUTION

Example violation

```
class Post
{
    void CreatePost(Database db, string postMessage)
    {
        db.Add(postMessage);
    }
}

class TagPost : Post
{
    override void CreatePost(Database db, string postMessage)
    {
        db.AddAsTag(postMessage);
    }
}

class MentionPost : Post
{
    void CreateMentionPost(Database db, string postMessage)
    {
        string user = postMessage.parseUser();

        db.NotifyUser(user);
        db.OverrideExistingMention(user, postMessage);
        base.CreatePost(db, postMessage);
    }
}
```



```
class MentionPost : Post
{
    override void CreatePost(Database db, string postMessage)
    {
        string user = postMessage.parseUser();

        NotifyUser(user);
        OverrideExistingMention(user, postMessage);
        base.CreatePost(db, postMessage);
    }

    private void NotifyUser(string user)
    {
        db.NotifyUser(user);
    }

    private void OverrideExistingMention(string user, string postMessage)
    {
        db.OverrideExistingMention(_user, postMessage);
    }
}
```

INTERFACE SEGREGATION

“Clients (calling code) should not be forced to depend on methods they do not use”

Small cohesive interfaces are preferential to large fat ones.

Whats an
interface in terms
of the ISP?

Whatever can be accessed by calling code working
with an instance of that type

Interface
segregation
principle



Liskov
substitution
principle

Large interfaces are harder to
fully implement, more likely to
only be partially implemented
and not substitutable for their
base type

INTERFACE SEGREGATION

Advantages

Encourages loose coupling

Easier to change or swap out individual implementations

More resilient code – less likely for things to break

Enables easier testing



INTERFACE SEGREGATION

Example violation

```
interface IPost
{
    void CreatePost();
}

interface IPostNew
{
    void CreatePost();
    void ReadPost();
}
```



```
interface IPostCreate
{
    void CreatePost();
}

interface IPostRead
{
    void ReadPost();
}
```

Try to avoid adding additional functionality to an existing interface by adding new methods

Create new interface and let class inherit multiple interfaces if needed

DEPENDENCY INVERSION

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

High-level

- More abstract
- Business rules
- More process orientated than detail orientated

Further from input/output
e.g. forms, buttons, files,
databases etc.

Low-level

Concerned with where
input/output – where is the
information coming from and in
what format?

Plumbing code – to connect
business logic to external aspects

DEPENDENCY INVERSION

Abstractions should not depend on details – details should depend on abstractions

Interfaces

- No implementation code
- need to provide the implementations within the class that implements interface

Abstract base classes

- has one or more abstract embers
- Abstract methods within, must be implemented by child classes

Separation of concerns – keep plumbing code separate from high level business logic

A way to decouple software modules