INTRASOFT INTERNATIONAL

WG Software Developers Meetup

Clean code principles: What Is Clean Code? How can I write clean and maintainable code?

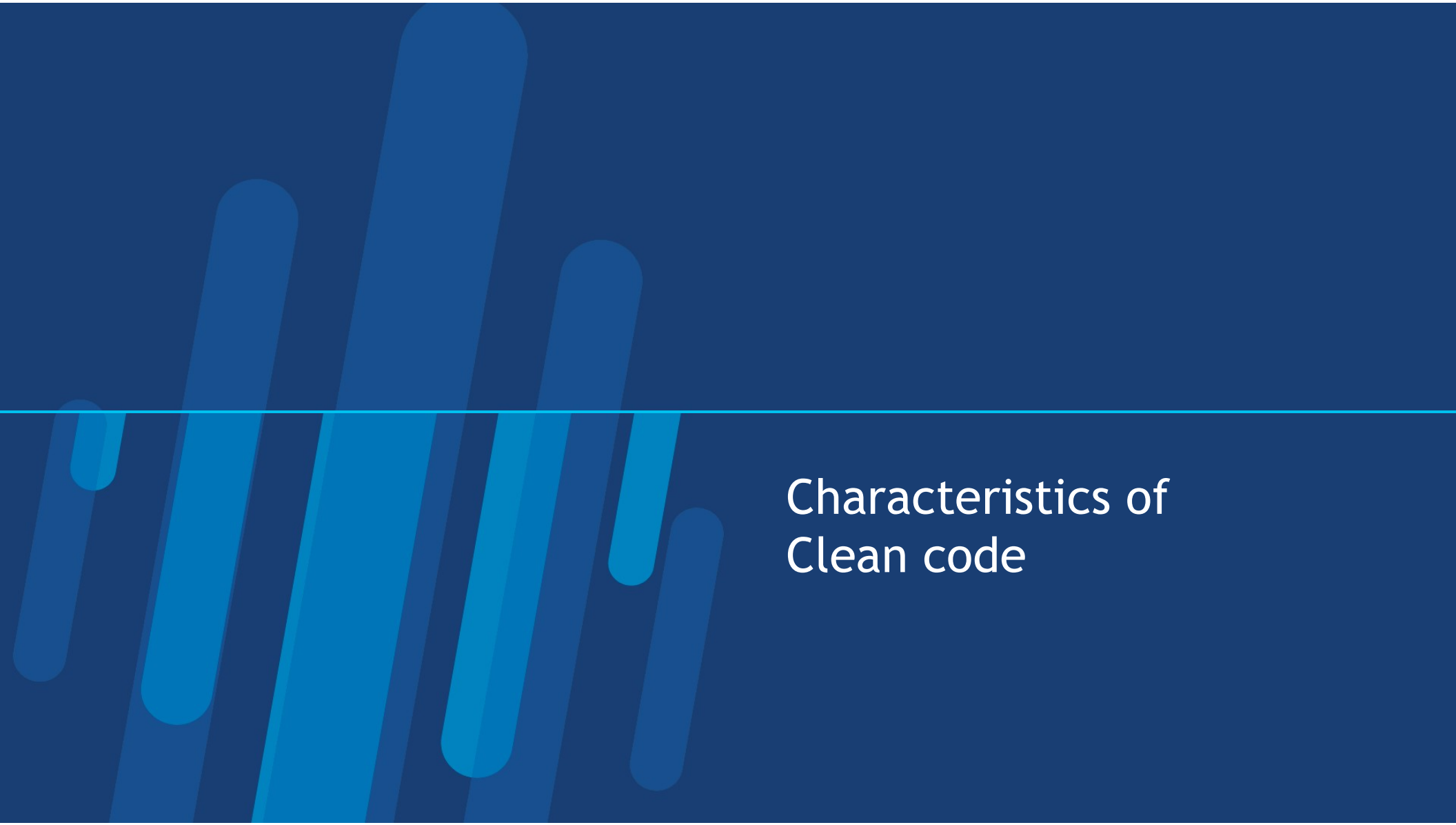
Nov 2019

Apostolos Stamatis
Senior SW Engineer

What is clean code and why do we need it?

INTRASOFT INTERNATIONAL

- Clean code is code that is easy to understand and easy to change.
  - *"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."*—Martin Fowler
- Clean code always looks like it was written by someone who cares. You should care because code is (almost) never written just once and then forgotten.
- Most of the time you, or someone else, need to work on the code. And to be able to work on it efficiently you need to understand the code.


Characteristics of
Clean code

Characteristics of Clean code

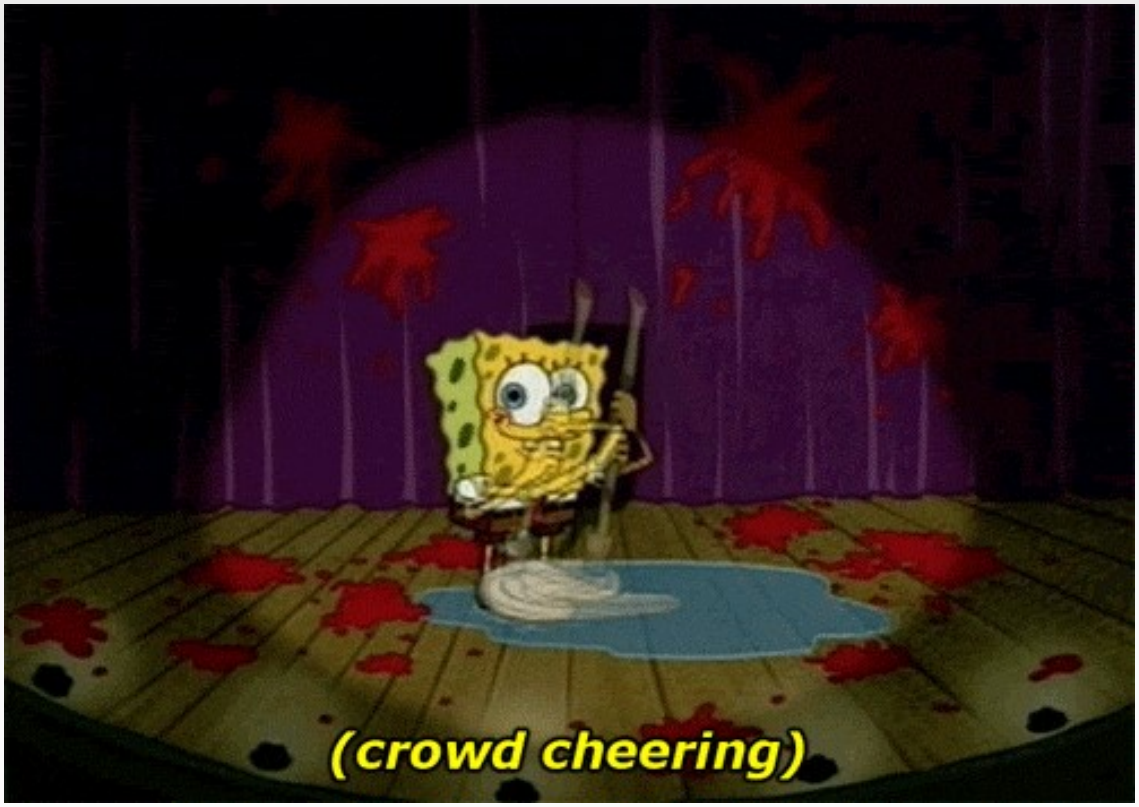
INTRASOFT INTERNATIONAL

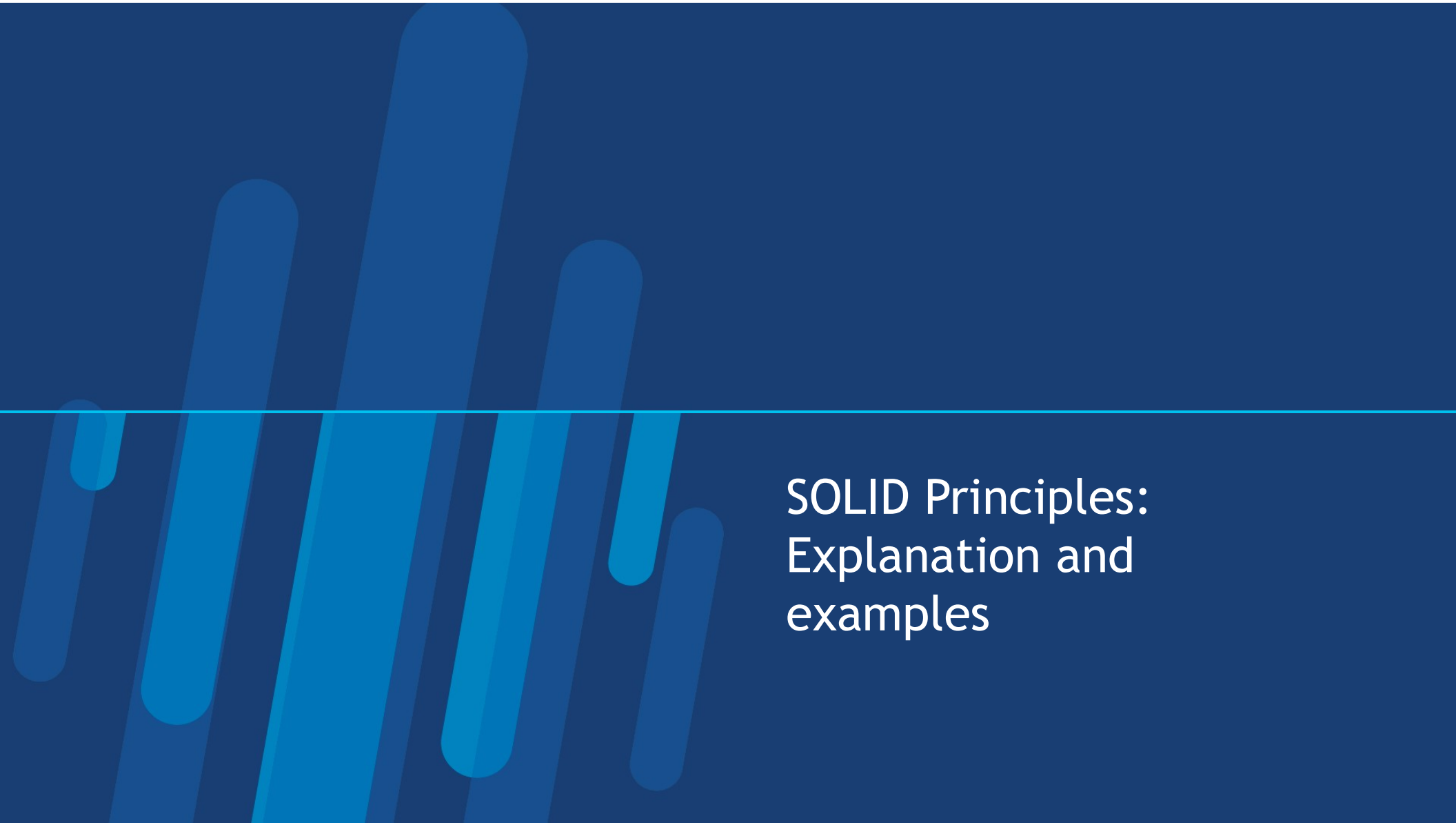
- ***Clean code is simple***. Perhaps not simple in algorithmic complexity, but certainly simple in implementation. Overly clever tricks and hacks are only fun for the author.
- ***Clean code is readable***.
  - If the naming conventions, structure, and flow used in a program are not designed with the reader in mind, then that reader will almost certainly fail to understand the original author's intent.
  - Conventions about how to write readable code helps to make code communal.
- ***Clean code is considerate***. Writing code that everyone understands, that the developer is confident is error-free and supported by clear documentation is being respectful of other team members.

Characteristics of Clean code

INTRASOFT INTERNATIONAL

- ***Clean code is tested***.
  - No one writes perfect, bug-free code on the first try. Even if it were possible to do so, there is no guarantee that perfect code won't break later.
  - Writing tested code means that future users can be confident they're interacting with something that works.
- ***Clean code is relentlessly refactored***.
  - Clean code should be in a constant state of refactoring.
  - With a good test suite to back up your code, you can refactor it as much as you like and never worry about breakage.
- ***Clean code is SOLID***. Good code is as much about good design as it is about cleanliness. Following the SOLID principles is one way to ensure that your code is acting the way it's supposed to, flexible, and maintainable.


(crowd cheering)


SOLID Principles:
Explanation and
examples

What is clean code and why do we need it?

INTRASOFT INTERNATIONAL

- SOLID is an acronym for 5 important design principles when doing OOP (Object Oriented Programming).
- Though they apply to any object-oriented design, the SOLID principles can also form a core philosophy for methodologies such as agile development.
- The theory of SOLID principles was introduced by Robert C. Martin (also known as “uncle Bob”) in his 2000 paper Design Principles and Design Patterns, although the SOLID acronym was introduced later by Michael Feathers.

SOLID Principles

INTRASOFT INTERNATIONAL

- 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)

S — Single responsibility principle (SRP)

S — Single responsibility principle (SRP)

INTRASOFT INTERNATIONAL

- **State**: “Every module or class should have responsibility over a single part of the functionality provided by the software”.
- **Meaning**: Each software component (class, method or module) should be responsible for doing only one thing, and it should do that one thing really well.
- **Benefits**: Single responsibility components result in code that is easer to understand, maintain and unit test.

- **Violation by**: Writing software components with multiple responsibilities.
- **Problem**: More responsibilities make component changes more frequent, result in compatibility issues, components make harder to understand, increase the chance of introducing bugs.

S — Single responsibility principle (SRP)

INTRASOFT INTERNATIONAL

Dirty code example

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

Clean code example

```
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 occured: ", error);
      File.WriteAllText("\LocalErrors.txt", error);
    }
}
```

O — Open/closed principle (OCP)

O — Open/closed principle (OCP)

INTRASOFT INTERNATIONAL

- **State**: “Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.”
- **Meaning**: Design software entities in such a way that we then can easily add features, without having to modify, recompile and redeploy them.
- **Benefits:** This results in well-designed code and reduced risk of breaking code.

- **Violation by**: Writing software components that need to modify when we need to introduce a change or a new feature.
- **Problem**: Modifying existing components may result in different behavior not expected by clients of the components, need to recompile & redeploy code that wastes time, introduces high chance of new bugs.

O — Open/closed principle (OCP)

INTRASOFT INTERNATIONAL

Dirty code example

```
1   class Post
2   {
3       void CreatePost(Database db, string postMessage)
4       {
5           if (postMessage.StartsWith("#"))
6           {
7               db.AddAsTag(postMessage);
8           }
9           else
10          {
11              db.Add(postMessage);
12          }
13      }
14  }
```

Clean code example

```
1   class Post
2   {
3       void CreatePost(Database db, string postMessage)
4       {
5           db.Add(postMessage);
6       }
7   }
8
9   class TagPost : Post
10  {
11      override void CreatePost(Database db, string postMessage)
12      {
13          db.AddAsTag(postMessage);
14      }
15  }
```

L — Liskov substitution principle (LSP)

L — Liskov substitution principle (LSP)

INTRASOFT INTERNATIONAL

- **State**: “If S is a subtype of T, then objects of type T may be replaced with objects of type S, without breaking the program.”
- **Meaning**: Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
- **Benefits**: This results in well-designed code that easily adopts new features and different implementations without affecting the existing code.

- **Violation by**: Applying OO techniques like Inheritance and Polymorphism in a way that derived classes break ‘contracts’ of their parent classes.
- **Problem**: Modifying existing components may result in different behavior not expected by clients of the components, need to recompile & redeploy code that wastes time, introduces high chance of new bugs.

L — Liskov substitution principle (LSP) | code example

INTRASOFT INTERNATIONAL

Dirty code example

```
using System;

namespace LSP
{
    #region Violating LSP
    5 references
    public interface IAnimal
    {
        3 references
        void Eat();
    }
    3 references
    public class Dog : IAnimal
    {
        3 references
        public void Eat()
        {
            Console.WriteLine("Eating dog food...");
        }

        1 reference
        public void Bark()
        {
            Console.WriteLine("Woof");
        }
    }
    3 references
    public class Cat : IAnimal
    {
        3 references
        public void Eat()
        {
            Console.WriteLine("Eating cat food...");
        }

        1 reference
        public void Meow()
        {
            Console.WriteLine("Meouw");
        }
    }
```

contract common to all animals

specific to dog

specific to cat

```
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            List<IAnimal> animals = new List<IAnimal>();
            animals.Add(new Dog());
            animals.Add(new Cat());

            foreach (IAnimal animal in animals)
            {
                animal.Eat();

                if (animal.GetType() == typeof(Dog))
                {
                    ((Dog)animal).Bark();
                }
                else if (animal.GetType() == typeof(Cat))
                {
                    ((Cat)animal).Meow();
                }
                else
                {
                    throw new Exception("Animal type not supported");
                }
            }
        }
    }
    #endregion
```

checking subtypes and down-casting is needed for the code to work

L — Liskov substitution principle (LSP) | code example

INTRASOFT INTERNATIONAL

Clean code example

```
68      #region Applying LSP
        5 references
69      public interface IAnimal
70      {
            3 references
71          void Eat();
            3 references
72          void MakeNoise();
73      }
        1 reference
74      public class Dog : IAnimal
75      {
            3 references
76          public void Eat()
77          {
78              Console.WriteLine("Eating dog food...");
79          }
80
            3 references
81          public void MakeNoise()
82          {
83              Console.WriteLine("Woof");
84          }
85      }
        1 reference
86      public class Cat : IAnimal
87      {
            3 references
88          public void Eat()
89          {
90              Console.WriteLine("Eating cat food...");
91          }
92
            3 references
93          public void MakeNoise()
94          {
95              Console.WriteLine("Meouw");
96          }
97      }
```

contract common to all animals

```
        0 references
99      class Program
100     {
            0 references
101         static void Main(string[] args)
102         {
103             List<IAnimal> animals = new List<IAnimal>();
104             animals.Add(new Dog());
105             animals.Add(new Cat());
106
107             foreach (IAnimal animal in animals)
108             {
109                 animal.Eat();
110                 animal.MakeNoise();
111             }
112         }
113     }
114     #endregion
```

no need for checking subtypes and sub-casting

I — Interface segregation principle (ISP)

I — Interface segregation principle (ISP)

INTRASOFT INTERNATIONAL

- **State**: “No client should be forced to depend on methods it does not use.”
- **Meaning**: Interfaces should serve a well-defined purpose and expose only purpose related specific functions.
- **Benefits**: This results in well-designed code with clear separation of responsibilities.

- **Violation by**: Writing generic interfaces with multiple functions that do not need to be exposed to all clients.
- **Problem**: Clients need to implement functions they do not need, adding additional unnecessary complexity, risking the change to break compatibility with clients and to introduce new bugs.

I — Interface segregation principle (ISP) | code example

INTRASOFT INTERNATIONAL

Dirty code example

```
#region Violating ISP
2 references
public interface IToy
{
    2 references
    void SetPrice(int price);
    2 references
    void Fly();
    2 references
    void Talk();
}

0 references
public class Doll : IToy
{
    int _price;
    2 references
    public void SetPrice(int price)
    {
        this._price = price;
    }
    2 references
    public void Fly()
    {
        throw new Exception("Not allowed operation");
    }
    2 references
    public void Talk()
    {
        Console.WriteLine("Doll toy is talking...");
    }
}

0 references
public class Plane : IToy
{
    int _price;
    2 references
    public void SetPrice(int price)
    {
        this._price = price;
    }
    2 references
    public void Fly()
    {
        Console.WriteLine("Plane toy is flying...");
    }
    2 references
    public void Talk()
    {
        throw new Exception("Not allowed operation");
    }
}
```

contract common to all toys

Doll toy does not need the Fly method

Plane toy does not need the Talk method

Clean code example

```
#region Applying ISP
1 reference
public interface IFly
{
    1 reference
    void Fly();
}
1 reference
public interface ITalk
{
    1 reference
    void Talk();
}
2 references
public interface IToy
{
    2 references
    void SetPrice(int price);
}

0 references
public class Doll : IToy, ITalk
{
    int _price;
    String _color;
    2 references
    public void SetPrice(int price)
    {
        this._price = price;
    }
    1 reference
    public void Talk()
    {
        Console.WriteLine("Doll toy is talking...");
    }
}

0 references
public class Plane : IToy, IFly
{
    int _price;
    String _color;
    2 references
    public void SetPrice(int price)
    {
        this._price = price;
    }
    1 reference
    public void Fly()
    {
        Console.WriteLine("Plane toy is flying...");
    }
}
#endregion
```

contracts splitted

Doll toy implements only the methods needed

Plane toy implements only the methods needed

D - Dependency inversion principle (DIP)

D - Dependency inversion principle (DIP)

INTRASOFT INTERNATIONAL

- **State**: “High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.”
- **Meaning**: Components like classes should not depend on each other directly, they should rather depend on abstractions (like interfaces or abstract classes).
- **Benefits**: This results in well-designed code that supports the generic goals of high readability and maintainability.

- **Violation by**: Writing classes that contain other classes via composition.
- **Problem**: Tight coupling among classes results in more complex code, harder to understand and maintain.

- **Apply DIP**: use a design pattern known as a dependency inversion pattern, most often solved by using dependency injection. Typically, dependency injection is used simply by ‘injecting’ any dependencies of a class through the class constructor as an input parameter.

D - Dependency inversion principle (DIP) | code example

INTRASOFT INTERNATIONAL

Dirty code example

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

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

Clean code example

```
class Post
{
    private Logger _logger;

    public Post(Logger injectedLogger)
    {
        _logger = injectedLogger;
    }

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

DRY or “Don’t Repeat Yourself”

DRY or “Don’t Repeat Yourself”

INTRASOFT INTERNATIONAL

- **State**: "Every piece of knowledge or logic must have a single, unambiguous representation within a system."
- **Meaning**: Software development aimed at reducing repetition of information.

- **Violation by**: Writing the same code or logic in many places.
- **Problem**: Makes code difficult to maintain, possible updates result in making multiple changes, increases the chance of bugs.

- **Apply DRY**: Divide your code and logic into smaller reusable units and use that code.
- **Benefits**: It saves time and effort, it is easy to maintain, and also reduces the chances of bugs.

KISS or “Keep it Simple, Stupid”

KISS or “Keep it Simple, Stupid”

INTRASOFT INTERNATIONAL

- **State**: “A simple solution is better than a complex one, even if the solution look stupid.”
- **Meaning**: Keep the code simple and clear, making it easy to understand.

- **Violation by**: Writing complex and messy code.
- **Problem**: Makes code difficult to maintain and understand, increases the chance of bugs.

- **Apply KISS**: Try making the simplest implementation using the decomposition technique (breaking the code into simpler and cleaner parts).
- **Benefits**: It makes the code easier to understand, easier to maintain and also reduces the chances of bugs.

Tips On Clean Code

Tips On Clean Code

INTRASOFT INTERNATIONAL

- Use meaningful names
- Use convention
- Try not to produce unused code
- Using libraries/APIs to not reinvent the wheel
- Do not write comments and variable names in your native language
- Do not write classes with too many lines of code
- Write unit tests
- Run test coverage to see how much of your code you're testing
- Do not use comments to comment out unused code
- Do not use magic numbers
- Avoid in-line comments, put comments in the method doc

Clean Code Examples

INTRASOFT INTERNATIONAL

Dirty Code Example	Clean Code Example
int d; // elapsed time in days	int elapsedTimeInDays;
apikey = 123456;	API_KEY = 123456;

if (today == 7) {
return 'It is holiday';
}
	const int SUNDAY = 7;
	if (today == SUNDAY) {
	return 'It is holiday';
	}

class Model01(object){
get_ageiymd();//get age in year, month, days
get_lmdiymd();//get last modified in year,month,days
}
	class Person (object){
	get_age_year_month_days ();
	get_last_modified_year_month_days (); }

for (int j = 0; j < 34; j++) {
s += (t[j] * 4) / 5;
}
	int realDaysPerIdealDay = 4;
	const int WORK_DAYS_PER_WEEK = 5;
	const int NUMBER_OF_TASKS = 34;
	int sum = 0;
	for (int = 0; j < NUMBER_OF_TASKS; j++) {
	int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
	int realTaskWeeks=(realTaskDays/WORK_DAYS_PER_WEEK);
	sum += realTaskWeeks;
	}

Conclusion

Conclusion

INTRASOFT INTERNATIONAL

- It takes a lot of practice and effort to write clean code. However it is definitely worth the extra effort.
- Writing clean code is integral to the success of the project.
- Perfect code is not always feasible but it is important to try and write code as cleanly as possible. Programming is an art. Let other programmers enjoy your code.

Any questions?

INTRASOFT INTERNATIONAL

Thank you for your attention

Follow us