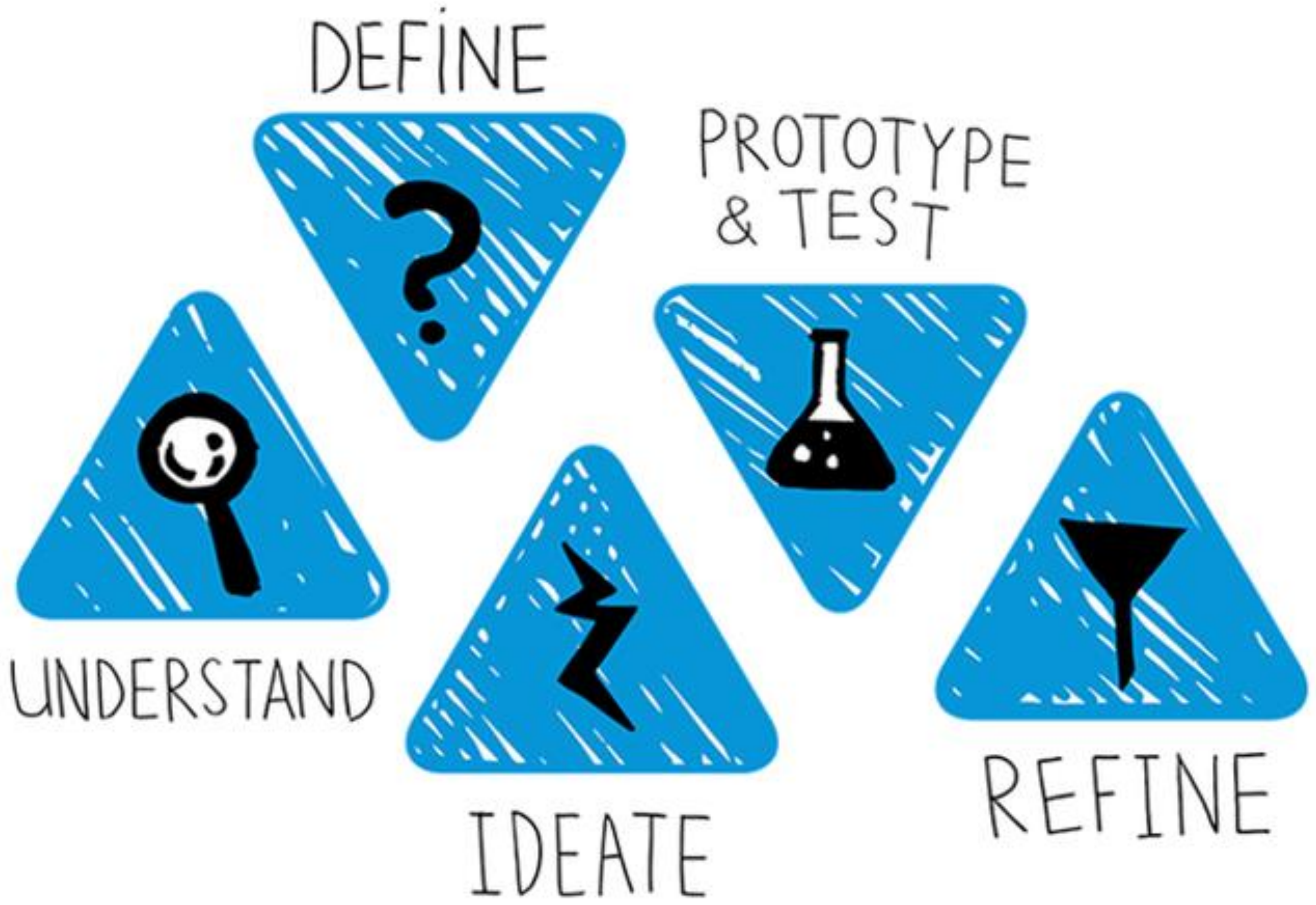
DESIGN PRINCIPLES

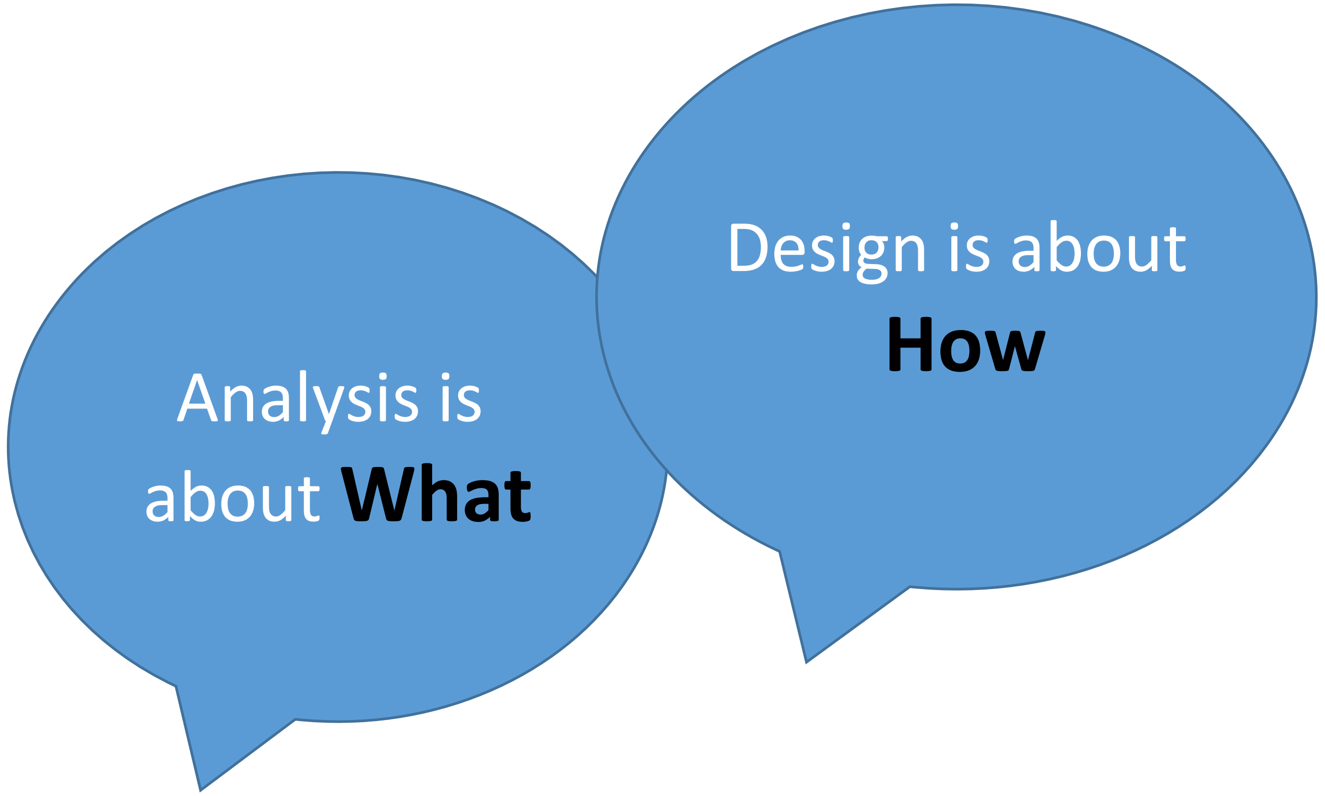


RAJA SEKHAR TADEPALLI

AGENDA

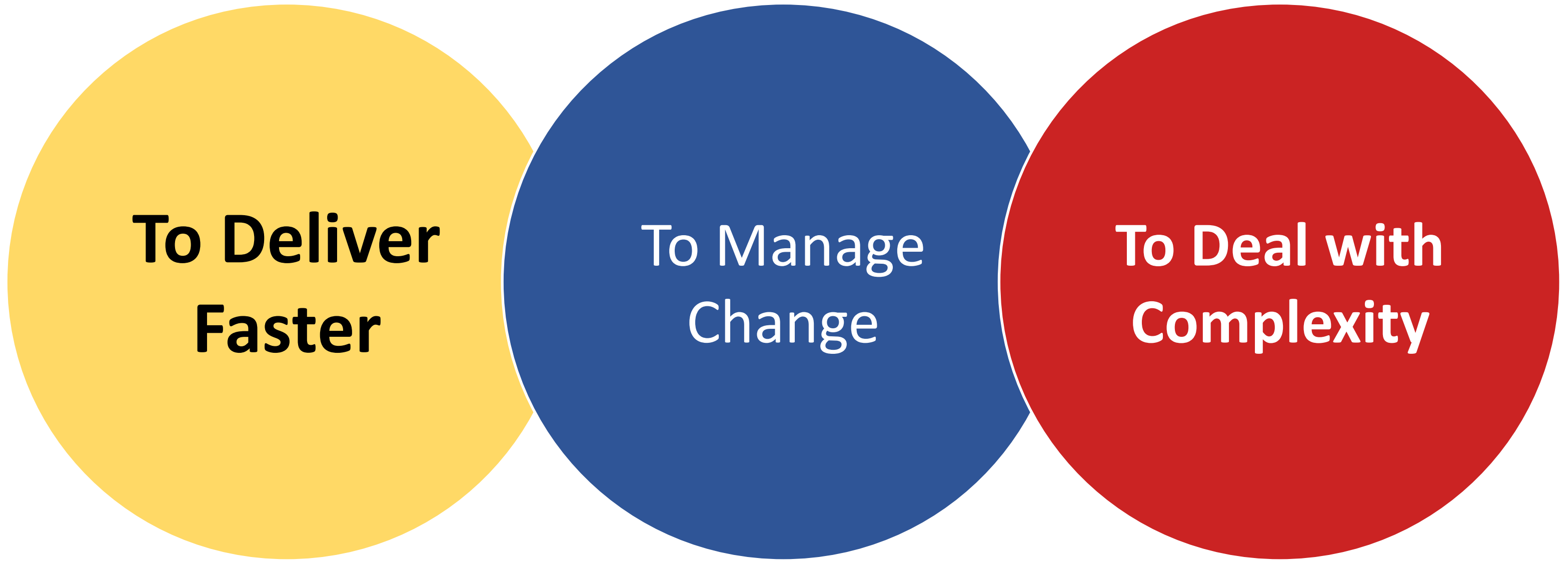
Design

- What's the meaning of design?
- What's the difference if compared to analysis?



Design

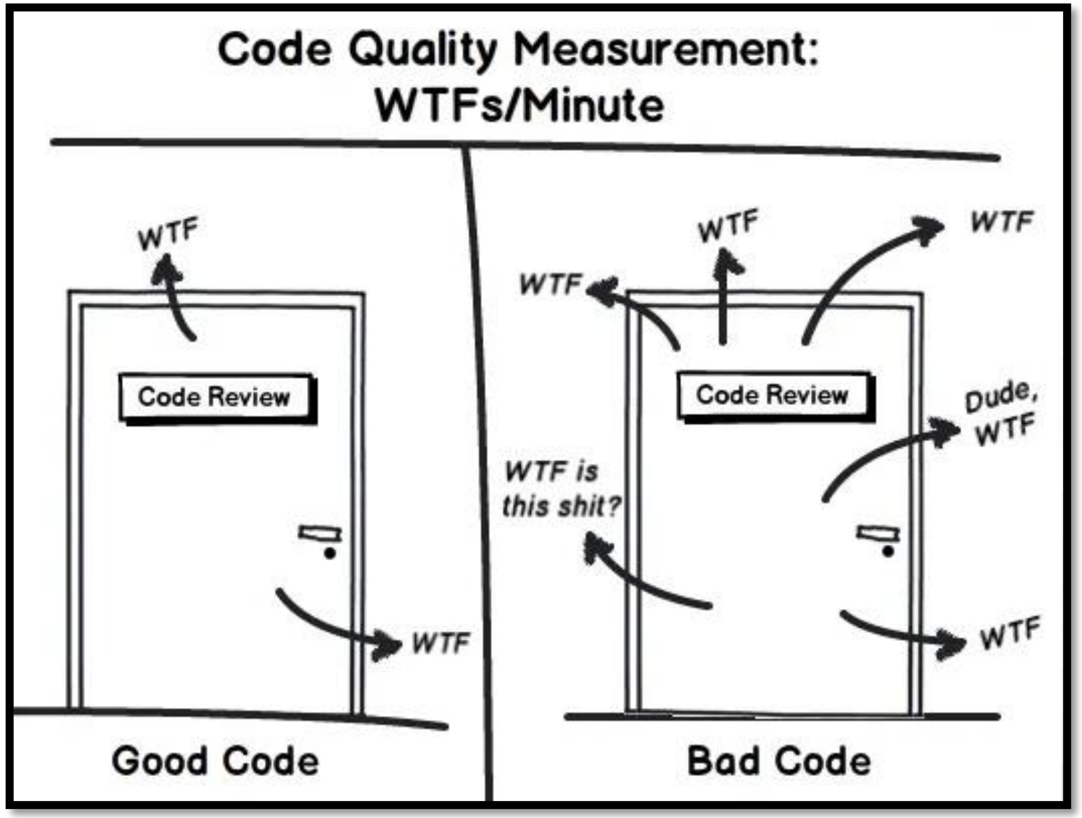
Why do we need (good) design?



Design

How do we know a design is bad?

- Cars have MPH (Miles per Hour) that measures the speed that they travel.
- The better the car the faster the MPH or speed.
- Developers have WTFPM (WTF per Minute) that measures the number of '**Works That Frustrate**' that the developer can read per minute, aka code quality.
- And just like cars, the better the developer the more WTFPM they can attain. Here is where they differ, however. With a car, the BETTER the road the more MPH it can attain. With a developer, the WORSE the code the more WTFPM can be obtained.



Design

Are there any ”symptoms” of bad design?

- **Rigidity**
  - inability to be to bend or be forced out of shape.
- **Fragility**
  - the quality of being easily broken or damaged.
- **Immobility**
  - Immobility refers to the inability to reuse software from other parts of the system..
- **Viscosity**
  - the state of being thick, sticky, and semi-fluid in consistency, due to internal friction.

Rigidity

- The impact of a change is unpredictable
- Every change causes a cascade of changes in dependent modules
- A nice ”two days” work become a kind of endless marathon
- Costs become unpredictable

Fragility

- the software tends to break in many places on every change
- the breakage occurs in areas with no conceptual relationship
- on every fix the software breaks in unexpected ways

Viscosity

- A hack is cheaper to implement than the solution within the design
- Preserving design moves are difficult to think and to implement
- it's much easier to do the wrong thing rather than the right one

Design

What's the reason why a design becomes rigid, fragile and viscous?

Improper Dependencies Between Modules

Design

So, what are the characteristics of a good design?

High Cohesion

Low Coupling

https://enterprisecraftsmanship.com/posts/cohesion-coupling-difference/

Cohesion & Coupling

- When a software program is modularized, its tasks are divided into several modules based on some characteristics.
- As we know, modules are set of instructions put together in order to achieve some tasks.
- They are though, considered as single entity but may refer to each other to work together.
- **There are measures by which the quality of a design of modules and their interaction among them can be measured.**
- These measures are called coupling and cohesion.

Cohesion

- Cohesion is a measure that defines the degree of intra-dependability within elements of a module. The greater the cohesion, the better is the program design.

There are seven types of cohesion, namely –

- **Co-incidental cohesion -** It is unplanned and random cohesion, which might be the result of breaking the program into smaller modules for the sake of modularization. Because it is unplanned, it may serve confusion to the programmers and is generally not-accepted.
- **Logical cohesion -** When logically categorized elements are put together into a module, it is called logical cohesion.
- **Temporal Cohesion -** When elements of module are organized such that they are processed at a similar point in time, it is called temporal cohesion.
- **Procedural cohesion -** . When elements of module are grouped together, which are executed sequentially in order to perform a task, it is called procedural cohesion

Cohesion

- **Communicational cohesion -** When elements of module are grouped together, which are executed sequentially and work on same data (information), it is called communicational cohesion.

- **Sequential cohesion -** When elements of module are grouped because the output of one element serves as input to another and so on, it is called sequential cohesion.

- **Functional cohesion -** It is considered to be the highest degree of cohesion, and it is highly expected. Elements of module in functional cohesion are grouped because they all contribute to a single well-defined function. It can also be reused.

Coupling

- Coupling is a measure that defines the level of inter-dependability among modules of a program. It tells at what level the modules interfere and interact with each other. The lower the coupling, the better the program.

There are five levels of coupling, namely –

- **Content coupling -** When a module can directly access or modify or refer to the content of another module, it is called content level coupling.
- **Common coupling-** When multiple modules have read and write access to some global data, it is called common or global coupling.
- **Control coupling-** Two modules are called control-coupled if one of them decides the function of the other module or changes its flow of execution.

Coupling

- **Stamp coupling-** When multiple modules share common data structure and work on different part of it, it is called stamp coupling.

- **Data coupling-** Data coupling is when two modules interact with each other by means of passing data (as parameter). If a module passes data structure as parameter, then the receiving module should use all its components.

Ideally, no coupling is considered to be the best.

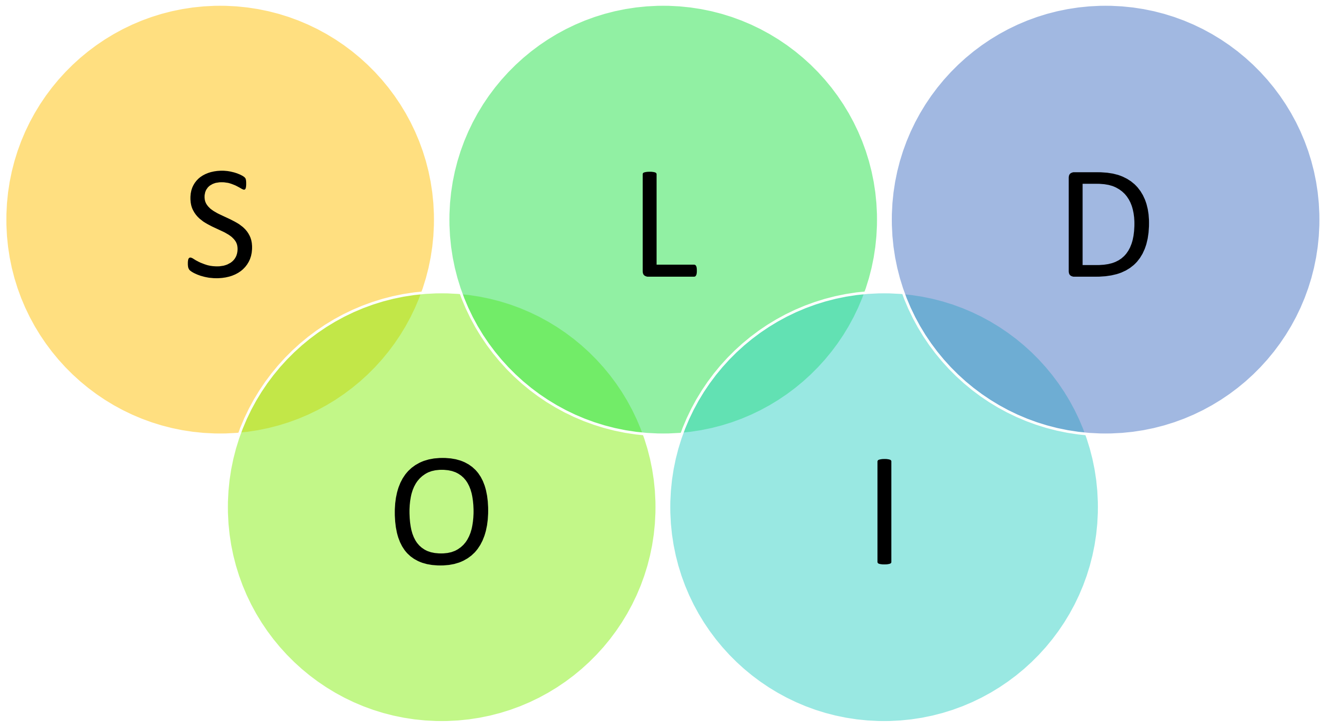
Design

How can we achieve a good design?

- SOLID Principles
- KISS Principles
- DRY Principles
- YAGNI Principles

BUILDING SOFTWARE ON A SOLID FOUNDATION

RAJA SEKHAR TADEPALLI



Single Responsibility Principle

Definition: ***A class should have one, and only one, reason to change.***

The Benefits :

- The class is easier to understand
- The class is easier to maintain
- The class is more reusable

Single Responsibility Principle

Now ask yourself, “which one is better for whittling?”

Single Responsibility Principle

Too many responsibilities on a single thing can caused problems.

Open Closed Principle

Definition: **Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.**

Bertrand Meyer is generally credited for having originated the definition of open/closed principle in his book Object-Oriented Software Construction

The application or software should be flexible to change.

The OCP states that the behaviors of the **system can be extended without having to modify its existing implementation.**

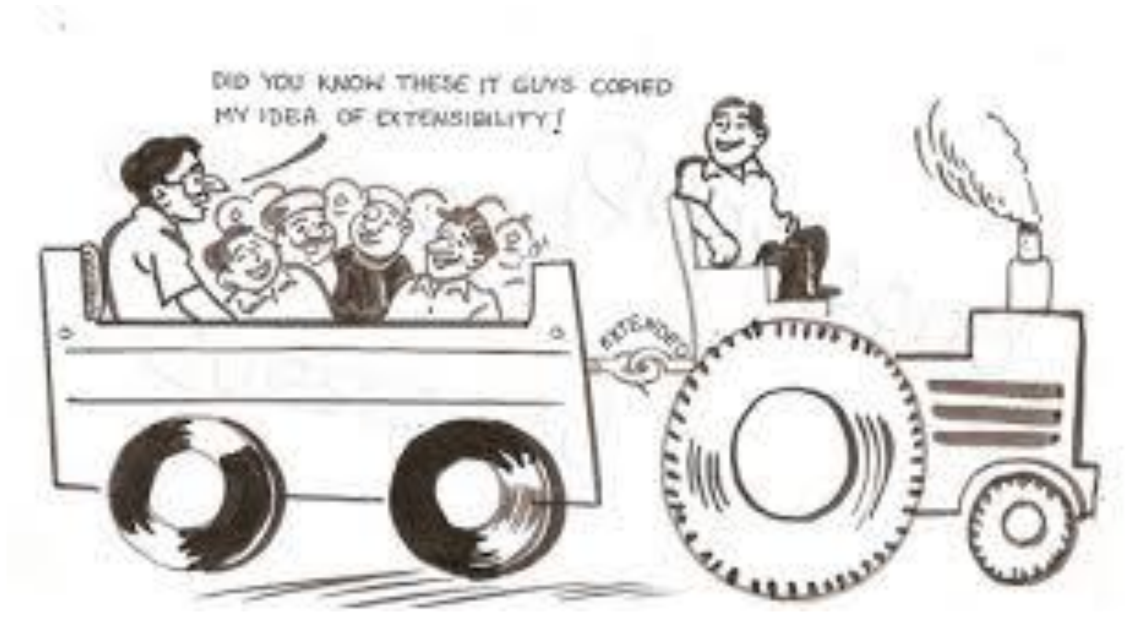
i.e. **New features should be implemented using the new code, but not by changing existing code.** The main benefit of adhering to OCP is that it potentially streamlines code maintenance and reduces the risk of breaking the existing implementation.

Original Code (Violates the Open Closed Principle)

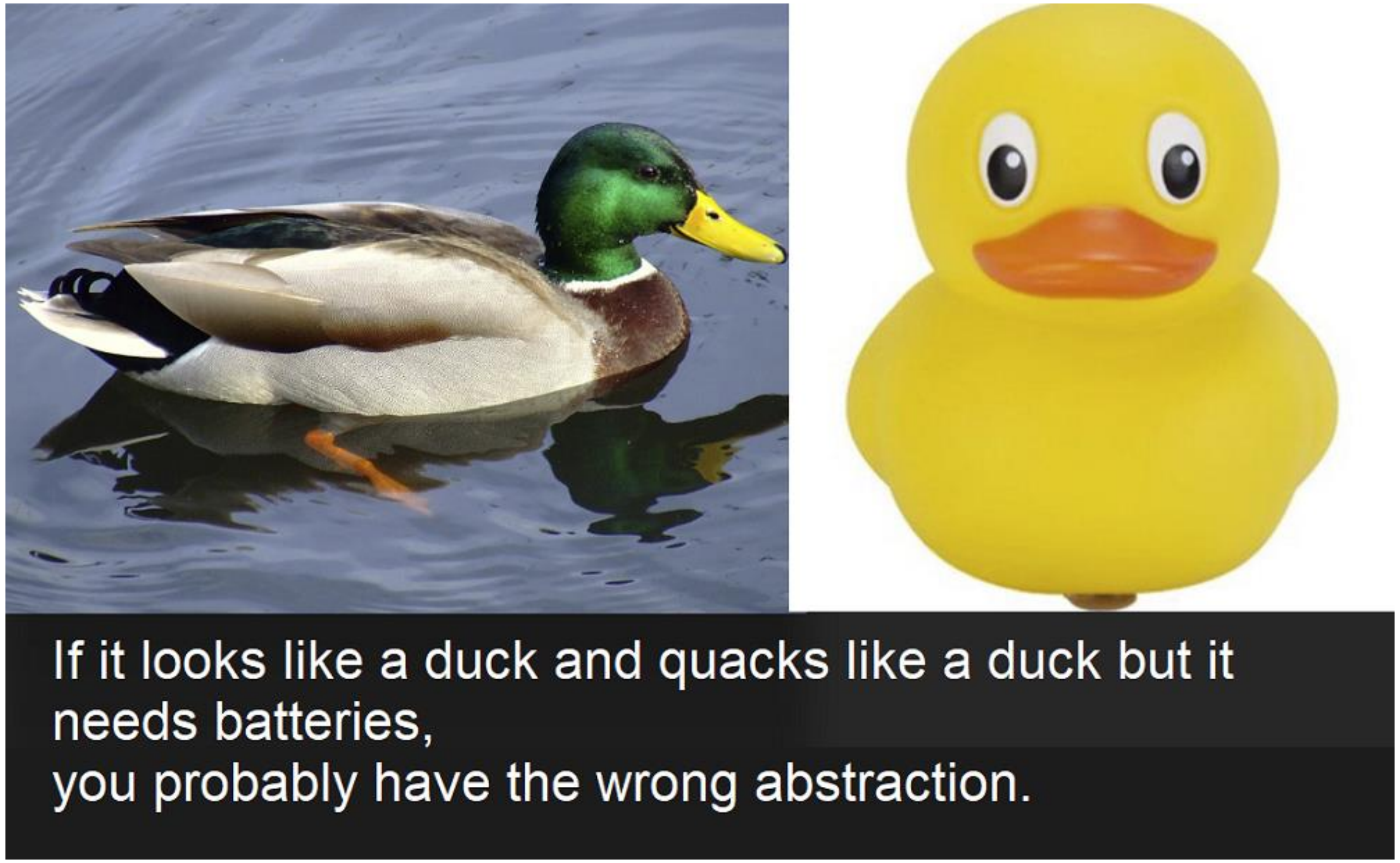
Several disadvantages if we modify the Existing Code/ original Code:

- The code **needs to be unit tested** whenever a new resource type is added.
- Adding a new resource type introduces **considerable risk in the design** as almost all aspects of resource allocation have to be modified.
- Developer adding a new functionality has to understand the inner workings for existing code.

Open Closed Principle



Liskov Substitution Principle


If it looks like a duck and quacks like a duck but it needs batteries,
you probably have the wrong abstraction.

Liskov Substitution Principle

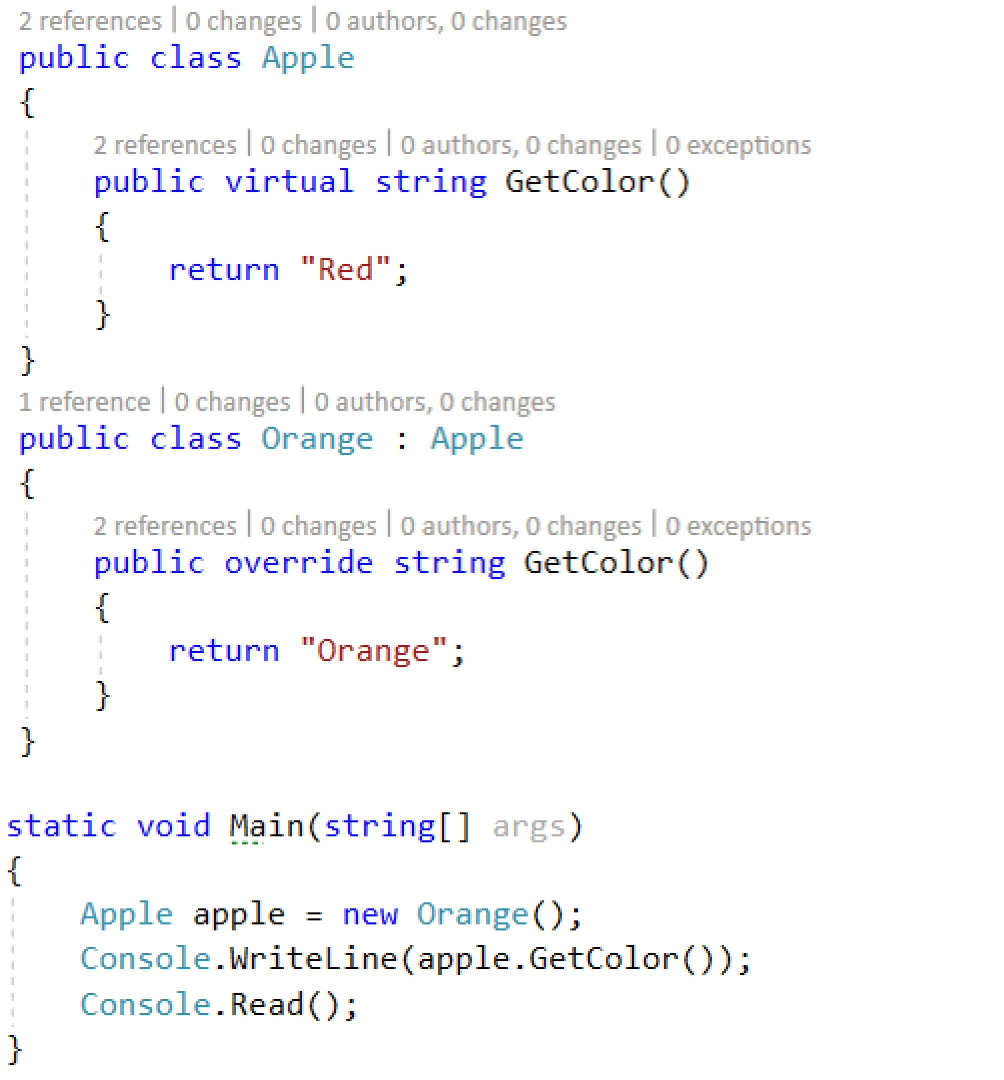
- The **Liskov substitution principle** (**LSP**) is a particular definition of a subtyping relation, called (**strong**) **behavioral subtyping**, that was initially introduced by Barbara Liskov in a 1987 conference keynote address titled *Data abstraction and hierarchy*

Definition :

Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program”

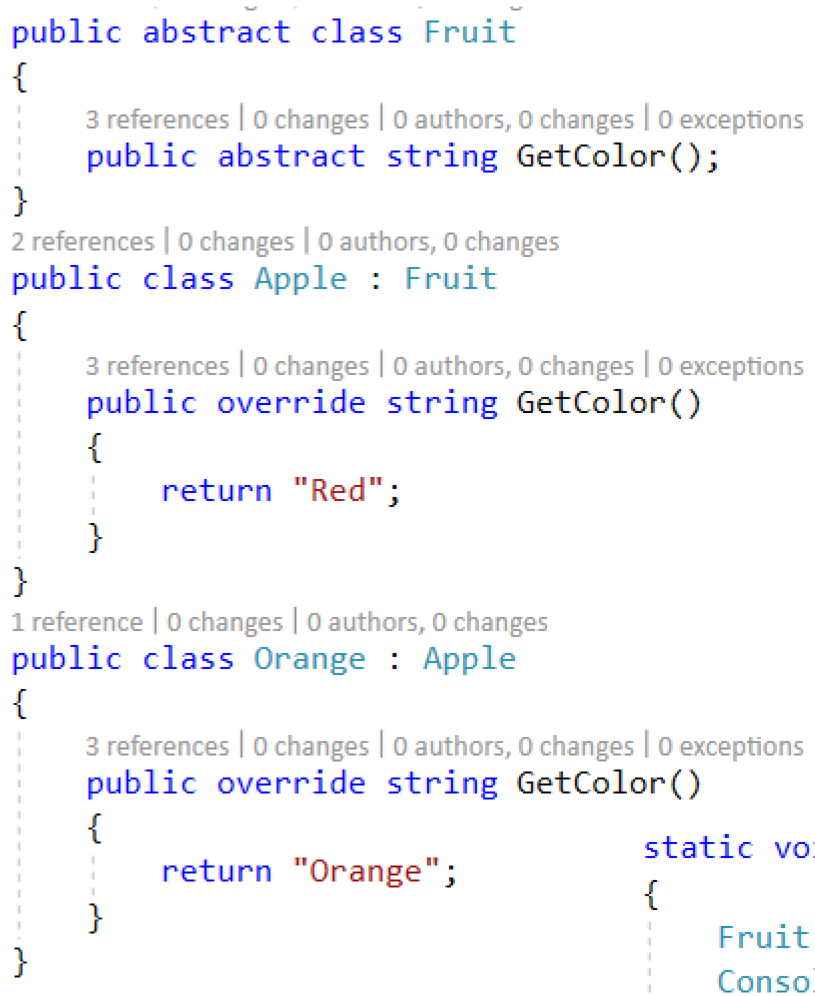
- It means that we must make sure that new derived classes are extending the base classes without changing their behavior.

Liskov Substitution Principle

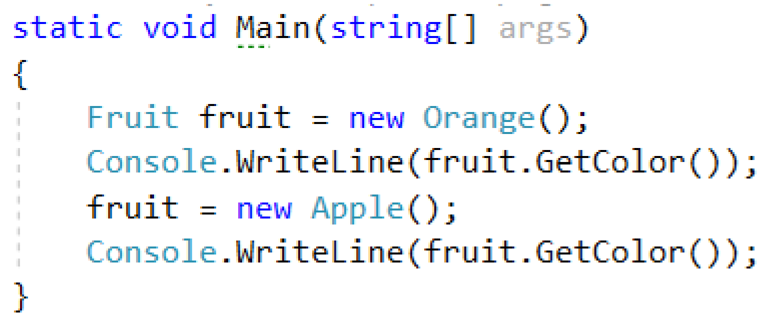


```
2 references | 0 changes | 0 authors, 0 changes
public class Apple
{
    2 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public virtual string GetColor()
    {
        return "Red";
    }
}
1 reference | 0 changes | 0 authors, 0 changes
public class Orange : Apple
{
    2 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public override string GetColor()
    {
        return "Orange";
    }
}

static void Main(string[] args)
{
    Apple apple = new Orange();
    Console.WriteLine(apple.GetColor());
    Console.Read();
}
```



```
public abstract class Fruit
{
    3 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public abstract string GetColor();
}
2 references | 0 changes | 0 authors, 0 changes
public class Apple : Fruit
{
    3 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public override string GetColor()
    {
        return "Red";
    }
}
1 reference | 0 changes | 0 authors, 0 changes
public class Orange : Apple
{
    3 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public override string GetColor()
    {
        return "Orange";
    }
}
```



```
static void Main(string[] args)
{
    Fruit fruit = new Orange();
    Console.WriteLine(fruit.GetColor());
    fruit = new Apple();
    Console.WriteLine(fruit.GetColor());
}
```

Liskov Substitution Principle

Advantages:

- Code Re-Usability
- Reduced coupling
- Easier maintenance

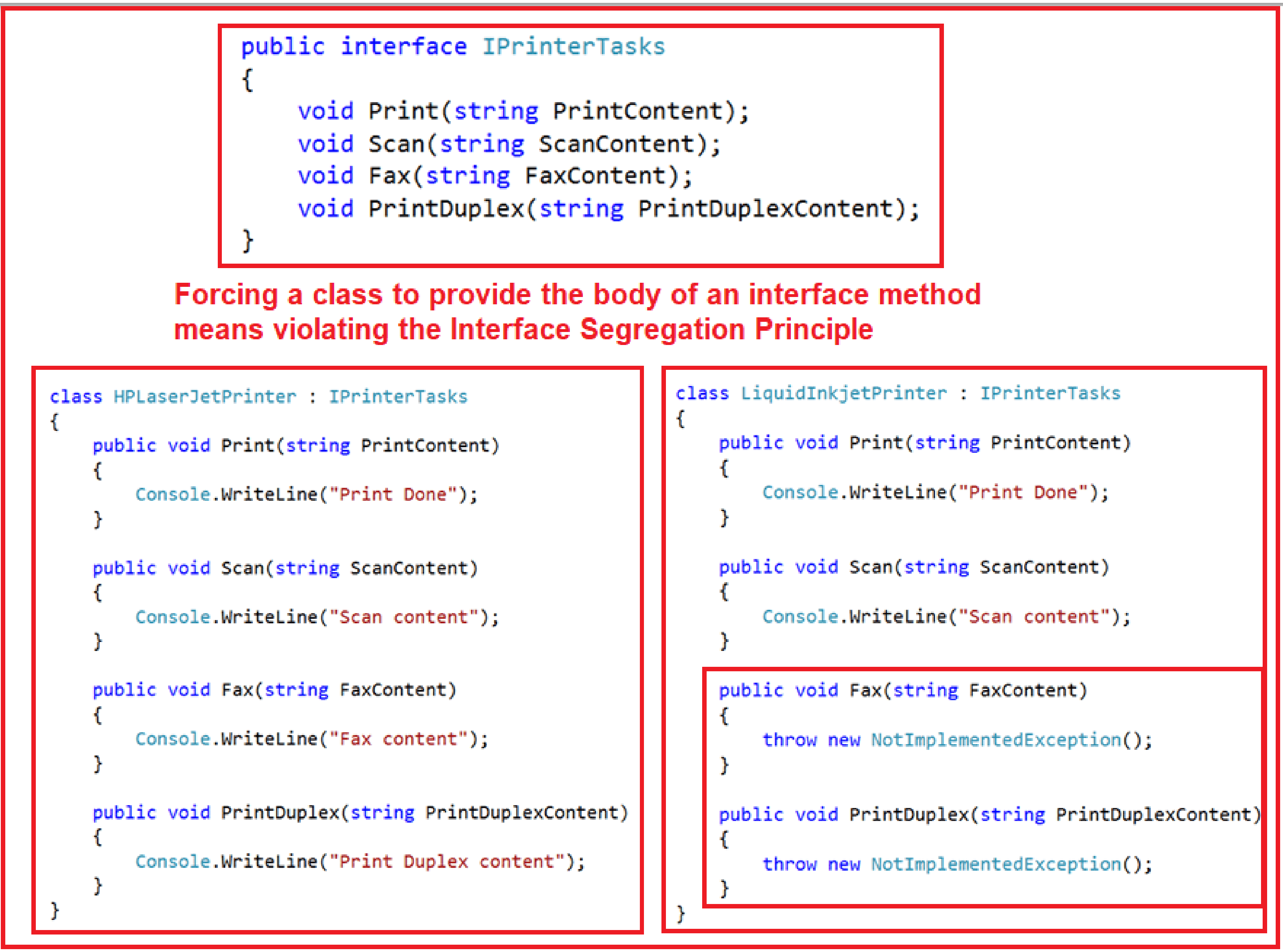
Interface-segregation principle (ISP)

Definition:

Clients should not be **forced** to implement any methods they don’t use. Rather than one fat interface, numerous little interfaces are preferred based on groups of methods with each interface serving one submodule

interface-segregation principle (ISP)

interface-segregation principle (ISP)



```
public interface IPrinterTasks
{
    void Print(string PrintContent);
    void Scan(string ScanContent);
    void Fax(string FaxContent);
    void PrintDuplex(string PrintDuplexContent);
}
```

Forcing a class to provide the body of an interface method means violating the Interface Segregation Principle

```
class HPLaserJetPrinter : IPrinterTasks
{
    public void Print(string PrintContent)
    {
        Console.WriteLine("Print Done");
    }

    public void Scan(string ScanContent)
    {
        Console.WriteLine("Scan content");
    }

    public void Fax(string FaxContent)
    {
        Console.WriteLine("Fax content");
    }

    public void PrintDuplex(string PrintDuplexContent)
    {
        Console.WriteLine("Print Duplex content");
    }
}
```

```
class LiquidInkjetPrinter : IPrinterTasks
{
    public void Print(string PrintContent)
    {
        Console.WriteLine("Print Done");
    }

    public void Scan(string ScanContent)
    {
        Console.WriteLine("Scan content");
    }

    public void Fax(string FaxContent)
    {
        throw new NotImplementedException();
    }

    public void PrintDuplex(string PrintDuplexContent)
    {
        throw new NotImplementedException();
    }
}
```

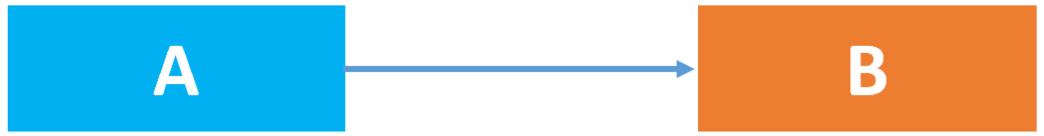
interface-segregation principle (ISP)

Benefits:

- Increases the readability
- Maintainability of our code

Dependency Inversion

- What is dependency?



- Dependency can be understood with the help of a simple example. Suppose Class A needs Class B to do its job, Hence, Class B is a dependency of Class A.
- This principle is about dependencies among components. The definition of DIP is given by Robert C. Martin is as follows:

Definition:

- **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.**

Dependency Injection

- Dependency Injection (DI) is a software design pattern that allows us to develop loosely coupled code.
- DI is a great way to reduce tight coupling between software components.
- DI also enables us to better manage future changes and other complexity in our software.
- The purpose of DI is to make code maintainable.

Types of Dependency Injection

Constructor Injection

- This is a widely used way to implement DI.
- Dependency Injection is done by supplying the **DEPENDENCY** through the **class's constructor** when creating the instance of that class.
- Injected component can be used anywhere within the class.
- Recommended to use when the injected dependency, you are using across the class methods.
- It addresses the most common scenario where a class requires one or more dependencies.

Types of Dependency Injection

Property/Setter Injection

- Recommended using when a class has optional dependencies, or where the implementations may need to be swapped.

- Different logger implementations could be used in this way.

- Does not require the creation of a new object or modifying the existing one. Without changing the object state, it could work.

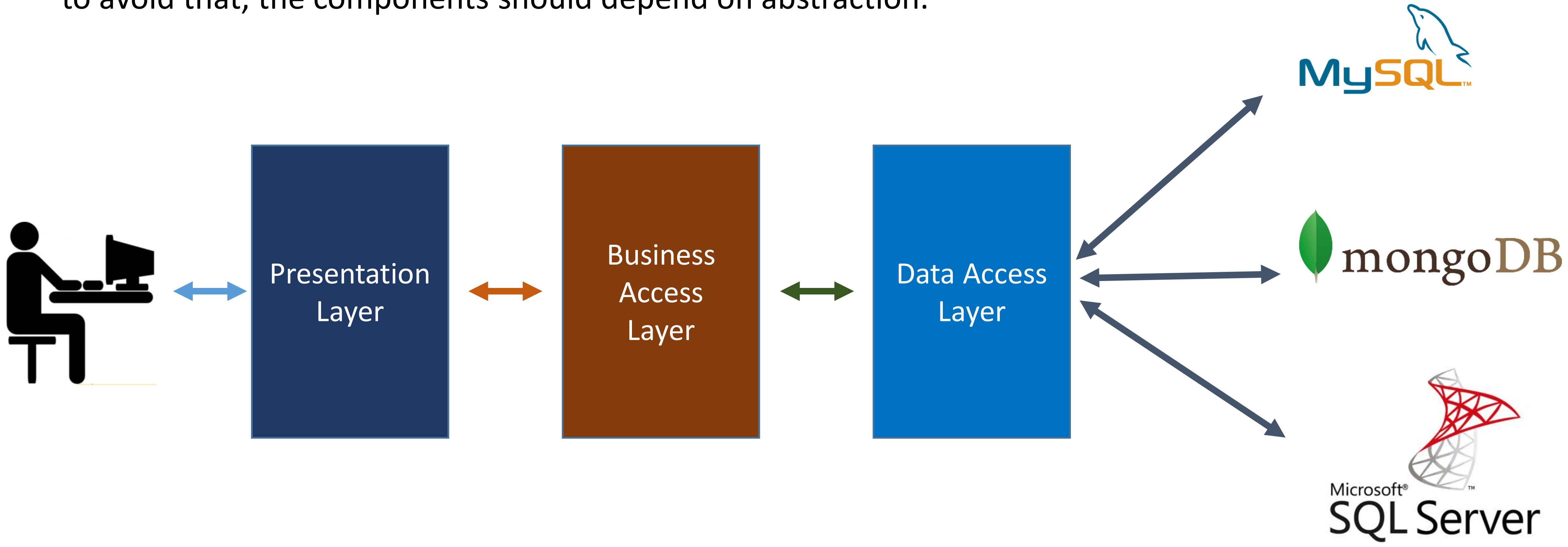
Types of Dependency Injection

Method Injection

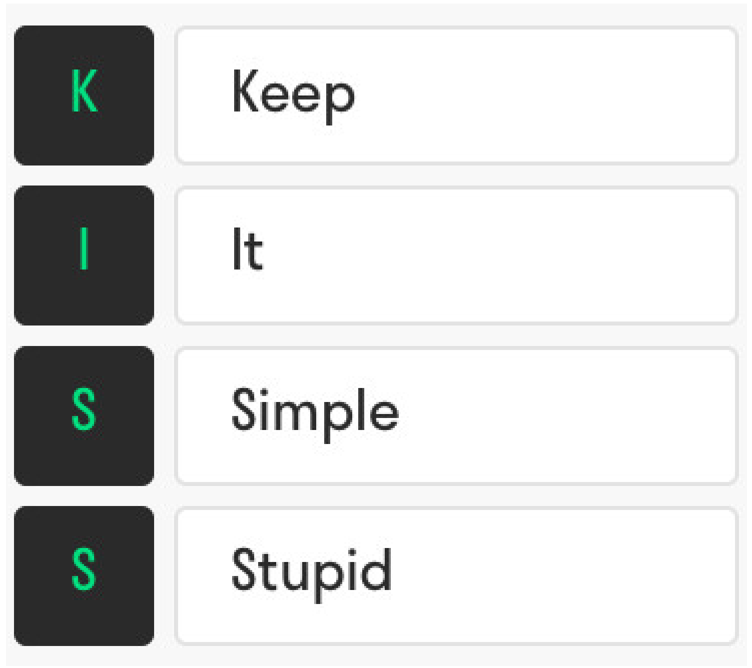
- Inject the dependency into a single method and generally for the use of that method.
- It could be useful, where the **whole class does not need the dependency**, **only one method having that dependency.**
- This is the way is rarely used.

Dependency Inversion

- the principle says that there should not be a tight coupling among components of software and to avoid that, the components should depend on abstraction.

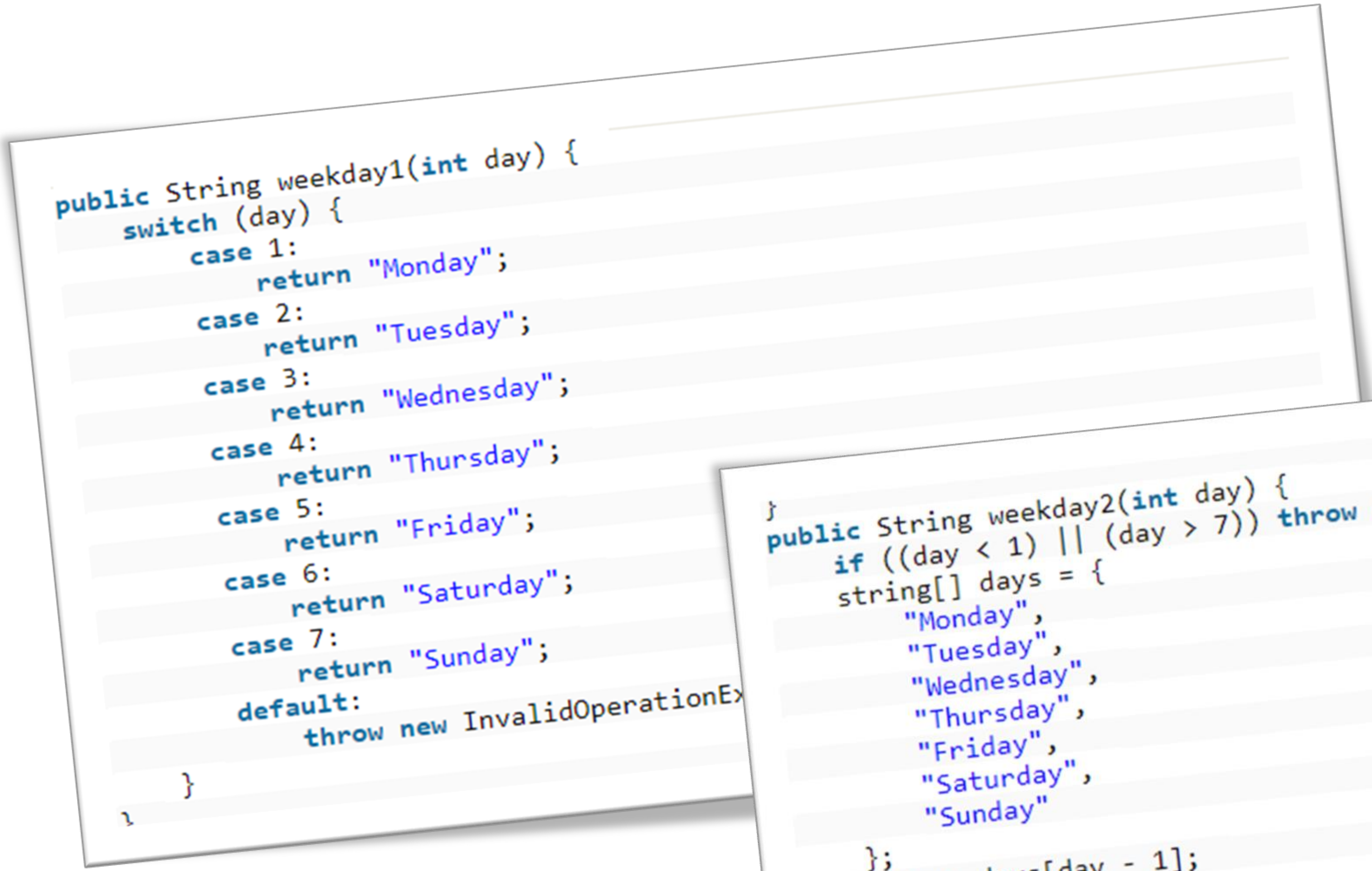


KISS

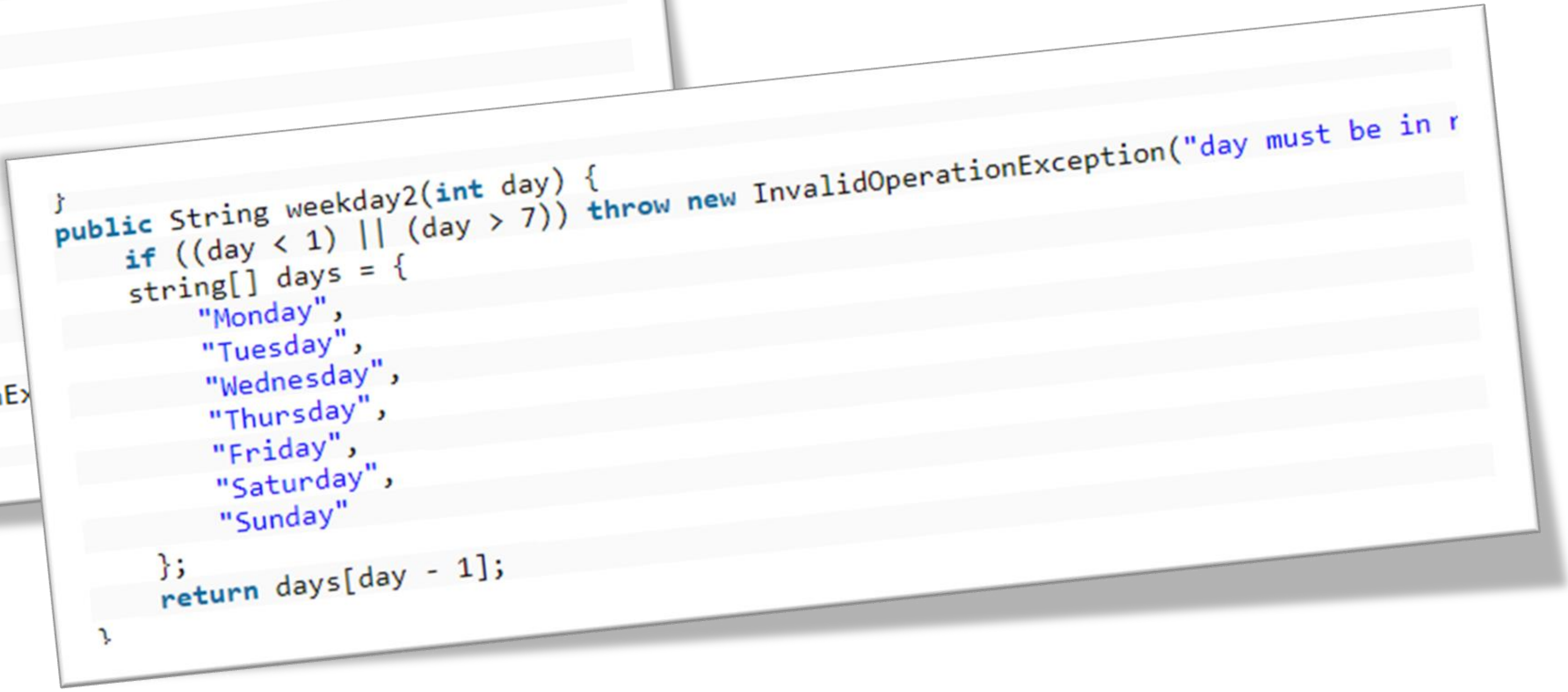


- KISS is a design principle noted by the U.S. Navy in 1960.
- Most systems work best if they are kept simple rather than made complicated; therefore, simplicity should be a key goal in design, and unnecessary complexity should be avoided.
- Keep your methods small, each method should never be more than 40-50 lines.
- Each method should only solve one small problem, not many use cases. If you have a lot of conditions in the method, break these out into smaller methods. It will not only be easier to read and maintain but also can find bugs a lot faster.

KISS



```
public String weekday1(int day) {
    switch (day) {
        case 1:
            return "Monday";
        case 2:
            return "Tuesday";
        case 3:
            return "Wednesday";
        case 4:
            return "Thursday";
        case 5:
            return "Friday";
        case 6:
            return "Saturday";
        case 7:
            return "Sunday";
        default:
            throw new InvalidOperationEx
    }
}
```



```
}
public String weekday2(int day) {
    if ((day < 1) || (day > 7)) throw new InvalidOperationException("day must be in r
    string[] days = {
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday",
        "Sunday"
    };
    return days[day - 1];
}
```

Now, you have to decide which one is clearer.

DRY (DON’T REPEAT YOURSELF)

- The DRY principle states that every piece of knowledge must have a single, unambiguous, authoritative representation within a system.
- This principle states that each small pieces of knowledge (code) may only occur exactly once in the entire system. This helps us to write scalable, maintainable and reusable code.

DRY (DON'T REPEAT YOURSELF)

```
public class Product
{
    /* Other members */
    public string Name { get; set; }

    public override string ToString()
    {
        return Name;
    }
}

public class Customer
{
    /* Other members */
    public string Name { get; set; }

    public override string ToString()
    {
        return Name;
    }
}
```

```
public class NamedEntity
{
    public string Name { get; set; }

    public override string ToString()
    {
        return Name;
    }
}

public class Product : NamedEntity
{
    /* Other members */
}

public class Customer : NamedEntity
{
    /* Other members */
}
```

DRY (DON'T REPEAT YOURSELF)

- Enterprise Libraries
- Helper Class
- Every Piece Of Code Is Unique In The Libraries And Helper Classes.

YAGNI (You ain't gonna need it)

- This principle states that always implement things when you actually need them never implements things before you need them.
- When the DRY principle is applied successfully, a modification of any single element of a system does not require a change in other logically unrelated elements.
- Additionally, elements that are logically related all change predictably and uniformly and are thus kept in sync.

TASK

- Implement Calculator Application with Add, Divide, Multiply and Subtract Methods
- Implement SOLID, KISS, DRY, YAGNI principles

URL to Submit the Task -

https://autocode.lab.epam.com/student/group/221

LEICESTER ARMS
Superdry
GENT STREET W.1
T-SHIRT GALLERY OPEN
superdry
ORLEBAR BROWN
H&M

Q & A

Superdry
GENT STREET W1
Superdry
T-SHIRT GALLERY OPEN
LEICESTER ARMS
ORLEBAR BROWN
H&M

THANK YOU!