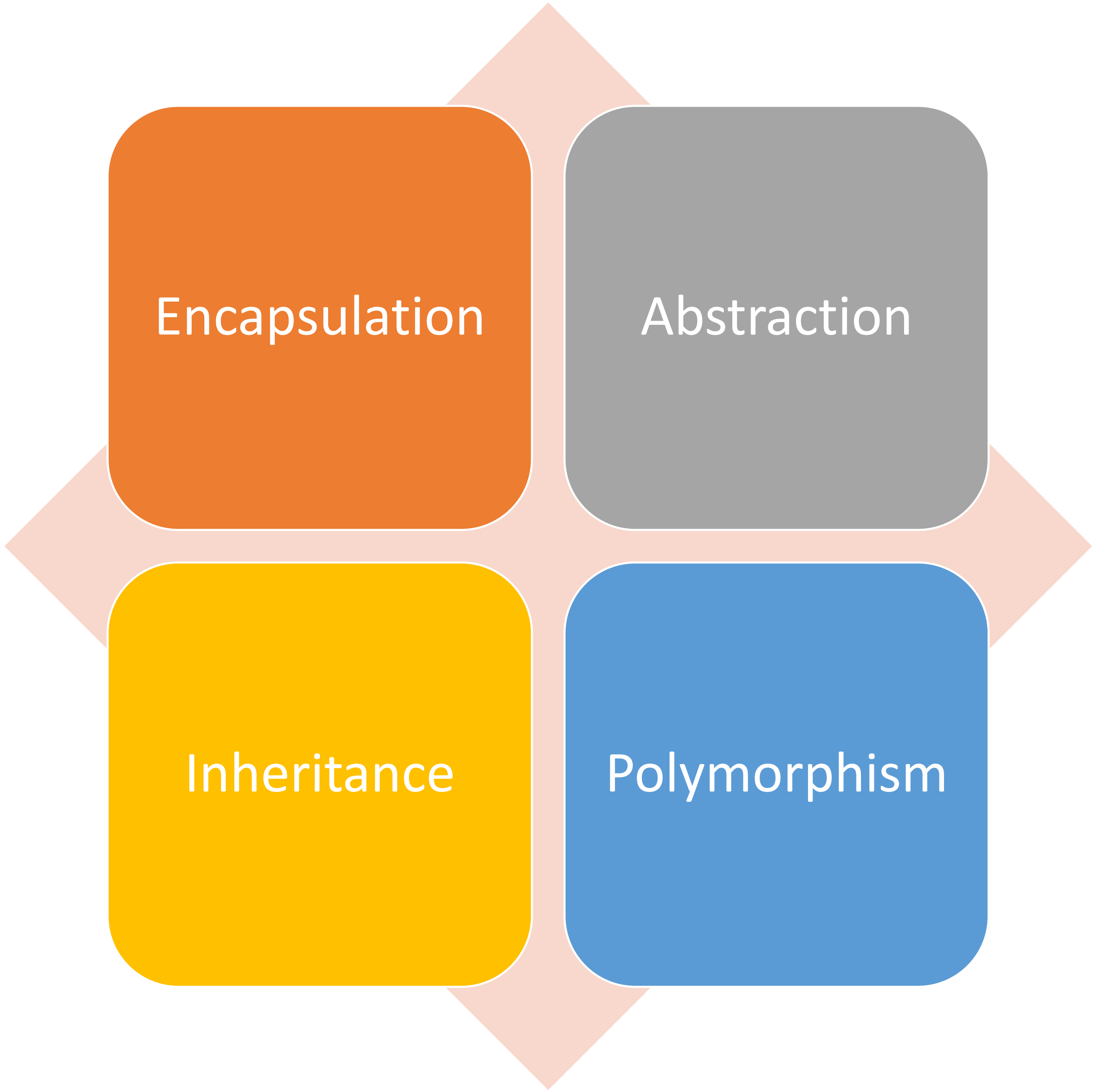
Fundamental Principles of Object Oriented Systems

A practical perspective!!!

Object Oriented Programming - OOPs



Source : https://www.codeproject.com/Articles/1166136/S-O-L-I-D-GRASP-And-Other-Basic-Principles-of-Obje#singleresponsibility-principle

Beyond OOPS - OOD

- OOD – Object Oriented Design Principles
- First five principles acronymed as S.O.L.I.D principles

- proposed by Robert C. Martin, popularly known as Uncle Bob.

- ➢ Helps programmer develop a software that is easy to extend, maintain.
- ➢ Helps in avoiding code smells, refactoring code, agile software development.

Beyond OOPS - OOD

- GRASP – General Responsibility Assignment Software Patterns
- GRASP with object-oriented programming classifies problems and their solutions together as patterns
- Other Principles coupled with GRASP and S.O.L.I.D help in keeping code simple, organizable, comprehensible, analyzable and reusable.

SOLID Principles

S - Single-responsiblity principle

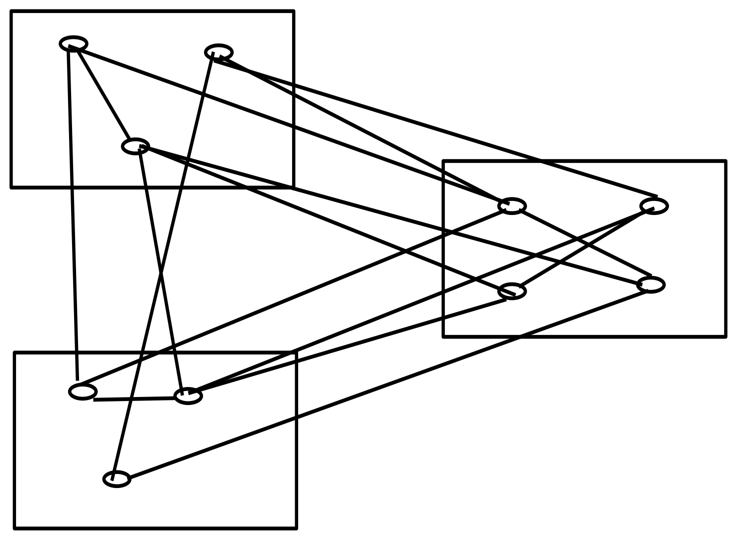
O - Open-closed principle

L - Liskov substitution principle

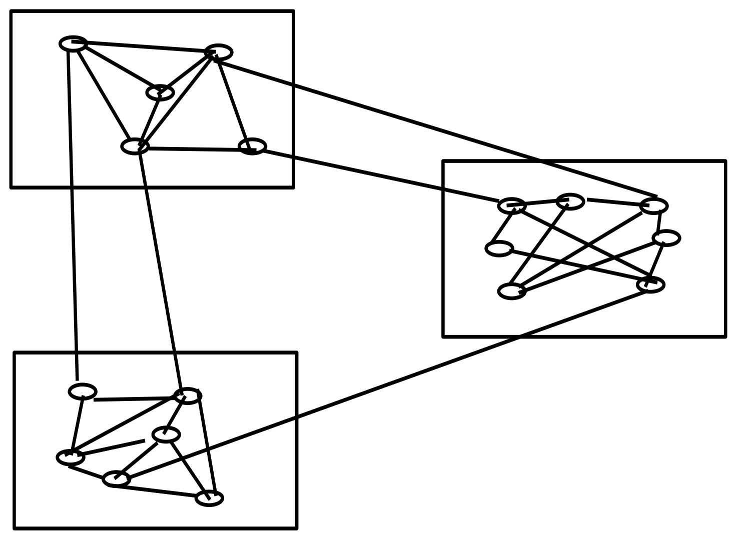
I - Interface segregation principle

D - Dependency Inversion Principle

A fundamental principle


(a)

high coupling, low cohesion


(b)

low coupling, high cohesion

GRASP - Assigns Seven types of roles to classes and objects

- High Cohesion
- Low Coupling
- Polymorphism
- Information Expert
- Creator
- Pure Fabrication
- Controller
- Indirection

Other Principles of OOD

- Program to an Interface, Not to an Implementation
- Hollywood Principle
- Favor Composition Over Inheritance
- Don't Repeat Yourself

"A class should have only one responsibility"

```
Class Simulation{
Public LoadSimulationFile()
Public Simulate()
Public ConvertParams()
}
```

Single Responsibility Principle (SRP)

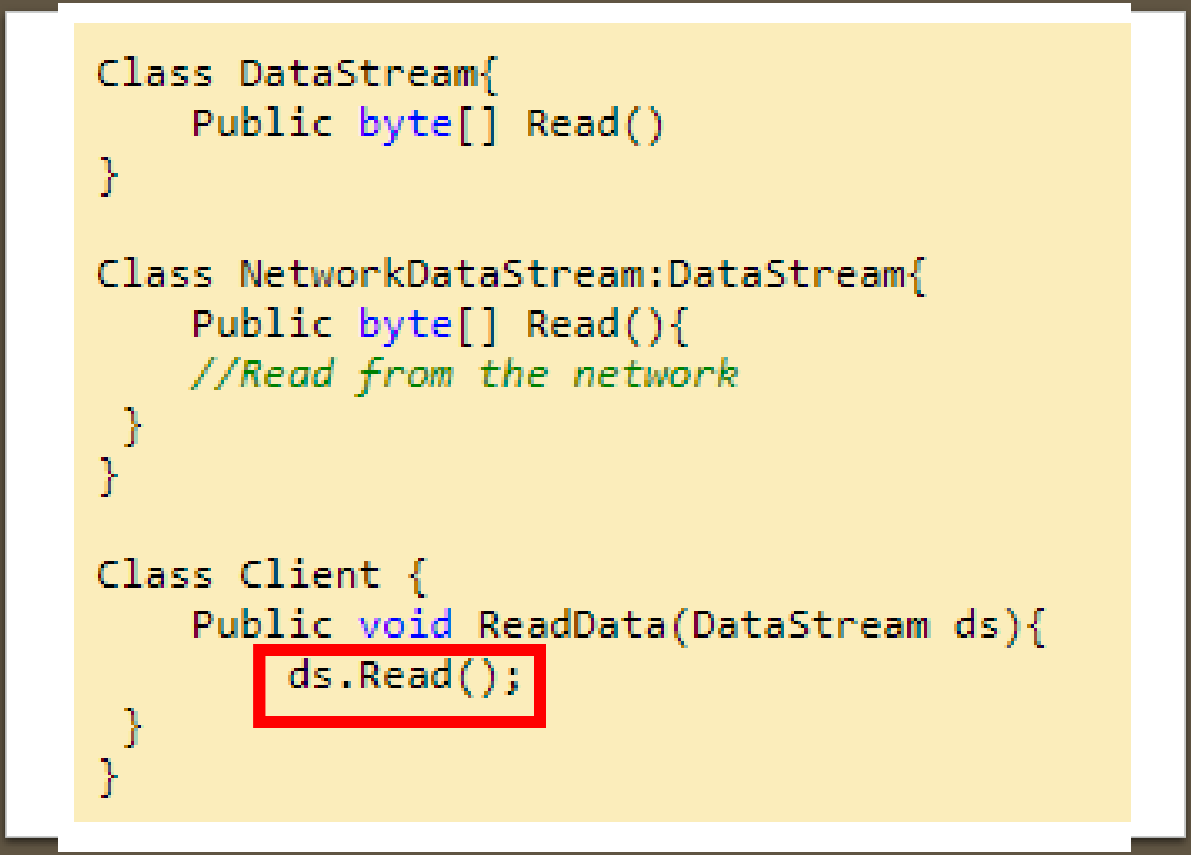
Single Responsibilty Principle (SRP)

- Responsibility == Reason to Change
- More than one reason to change class => SRP not followed
- Hard to be followed...
- Hence, depend on GRASP??

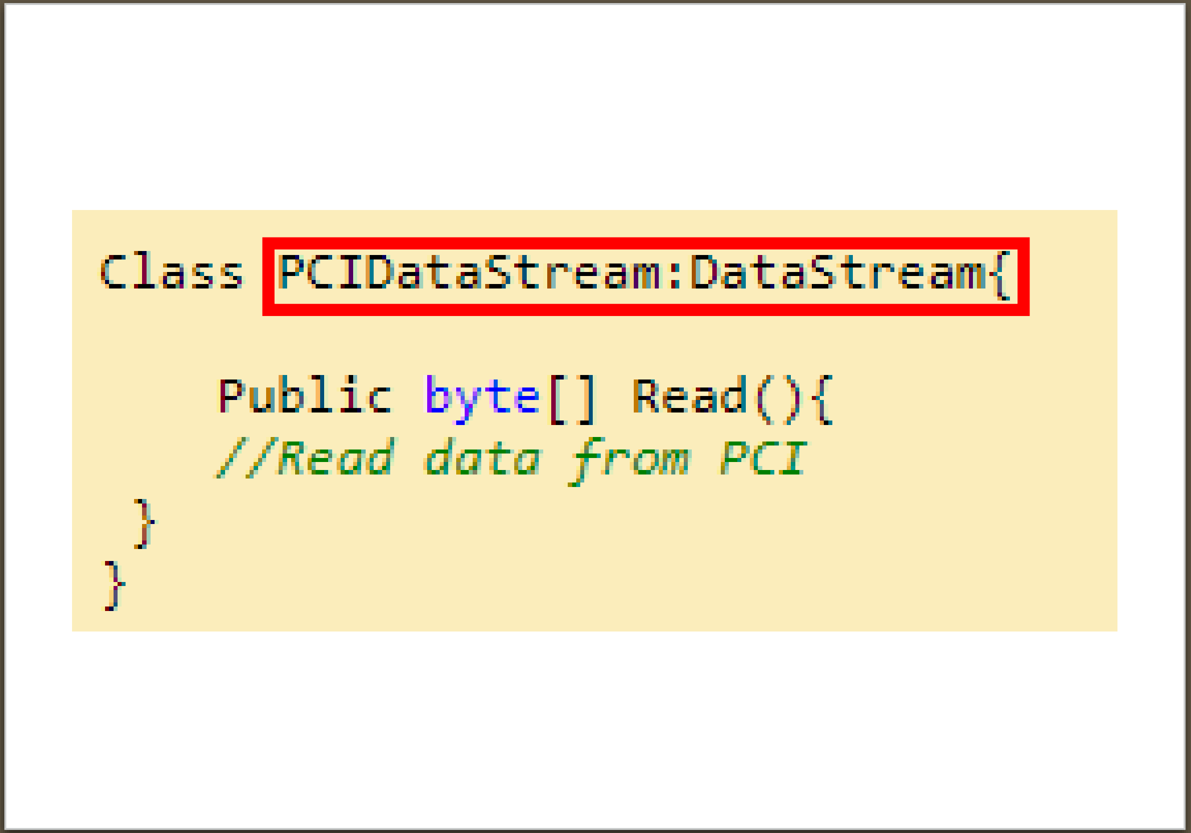
Open-Closed Principle

"A software module (it can be a class or method) should be open for extension but closed for modification."

- Inheritance
- Composition



```
Class DataStream{
    Public byte[] Read()
}

Class NetworkDataStream:DataStream{
    Public byte[] Read(){
    //Read from the network
 }
}

Class Client {
    Public void ReadData(DataStream ds){
        ds.Read();
 }
}
```



```
Class PCIDataStream:DataStream{

    Public byte[] Read(){
    //Read data from PCI
 }
}
```

Ex: Open-Closed Principle using Inheritance

- Strategically apply this principle, when you suspect a piece of code might change

What is Liskov Substitution Principle?

- “if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., an object of type T may be substituted with any object of a subtype S) without altering any of the desirable properties of that program!”
- Rectangle and Square example!
- Stack and Queue example!

- “The base class would not include properties for Height and Width, instead allowing the dimensions of different shapes to be handled by their implementations.”

Liskov Substitution Principle

"Derived classes must be substitutable for their base classes."

Example of data Acquisition Device Abstraction:

```
Public Interface IDevice{
    Void Open();
    Void Read();
    Void Close();
}
```

```
public class PCIDevice:IDevice {
   public void Open(){
   // Device specific opening logic
   }
   public void Read(){
   // Reading logic specific to this device
   }
   public void Close(){
   // Device specific closing logic.
   }
}

public class NetWorkDevice:IDevice{

 public void Open(){
 // Device specific opening logic
 }
 public void Read(){
 // Reading logic specific to this device
 }
 public void Close(){
 // Device specific closing logic.
 }
}
```

```
public class USBDevice:IDevice{
 public void Open(){
    // Device specific opening logic
 }
 public void Read(){
    // Reading logic specific to this device<br>
 }
public void Close(){
   // Device specific closing logic.
 }
 public void Refresh(){
   // specific only to USB interface Device
 }
}

//Client code...

Public void Acquire(IDevice aDevice){
        aDevice.Open();
        // Identify if the object passed here is USBDevice class Object.
    if(aDevice.GetType() == typeof(USBDevice)){
    USBDevice aUsbDevice = (USBDevice) aDevice;
    aUsbDevice.Refresh();
    }

        // remaining code….
}
```

Update Interface as :

```
Public Interface IDevice{
     Void Open();
     Void Refresh();
     Void Read();
     Void Close();
}
```

Client of IDevice:

```
Public void Acquire(IDevice aDevice)
     {
     aDevice.open();
     aDevice.refresh();
     aDevice.acquire()
     //Remaining code...
}
```

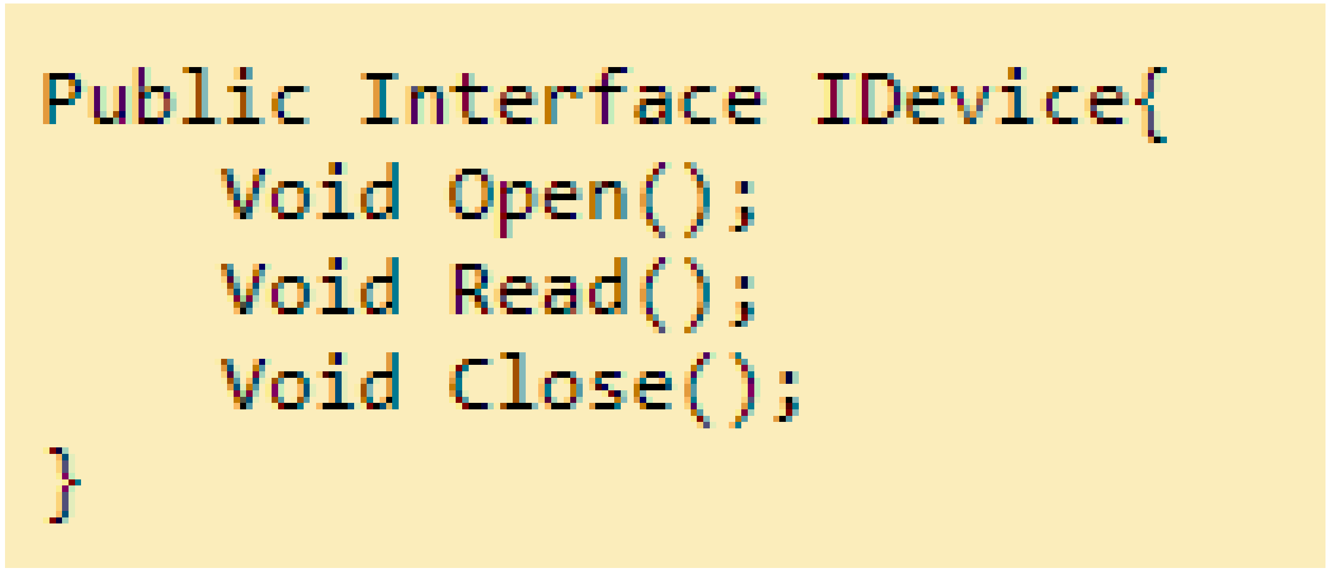
Behavioral Subtyping

- A type S is a behavioral subtype of a type T should depend only on the *specification* (i.e. the *documentation*) of type T;
- the *implementation* of type T does not really matter!
- “A type S is a behavioral subtype of a type T if each behavior allowed by the specification of S is also allowed by the specification of T. This requires, in particular, that for each method M of T, the specification of M in S is *stronger* than the one in T.”

Interference Segregation Principle (ISP)

"Clients should not be forced to depend upon the interfaces that they do not use."

Previous Example:

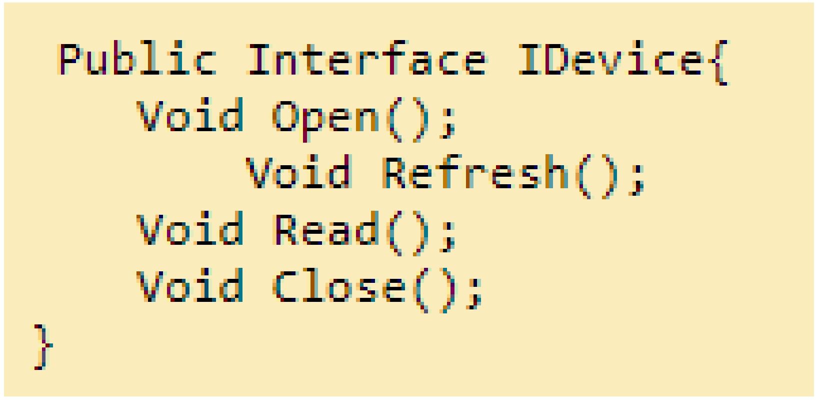


```
Public Interface IDevice{
    Void Open();
    Void Read();
    Void Close();
}
```

If USBDevice needs to implement Refresh ( ) functionality.......

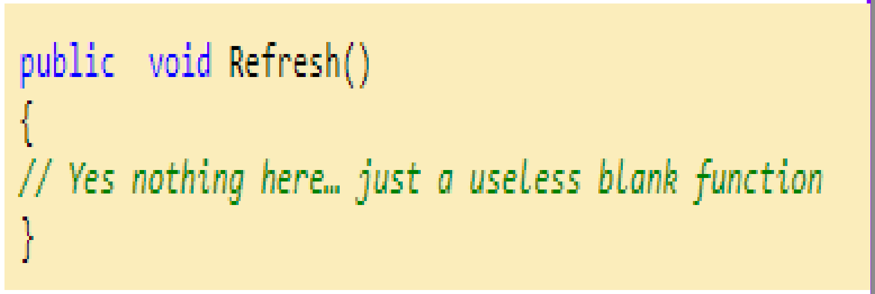
Avoid Fat Interfaces!

Update Interface as below:



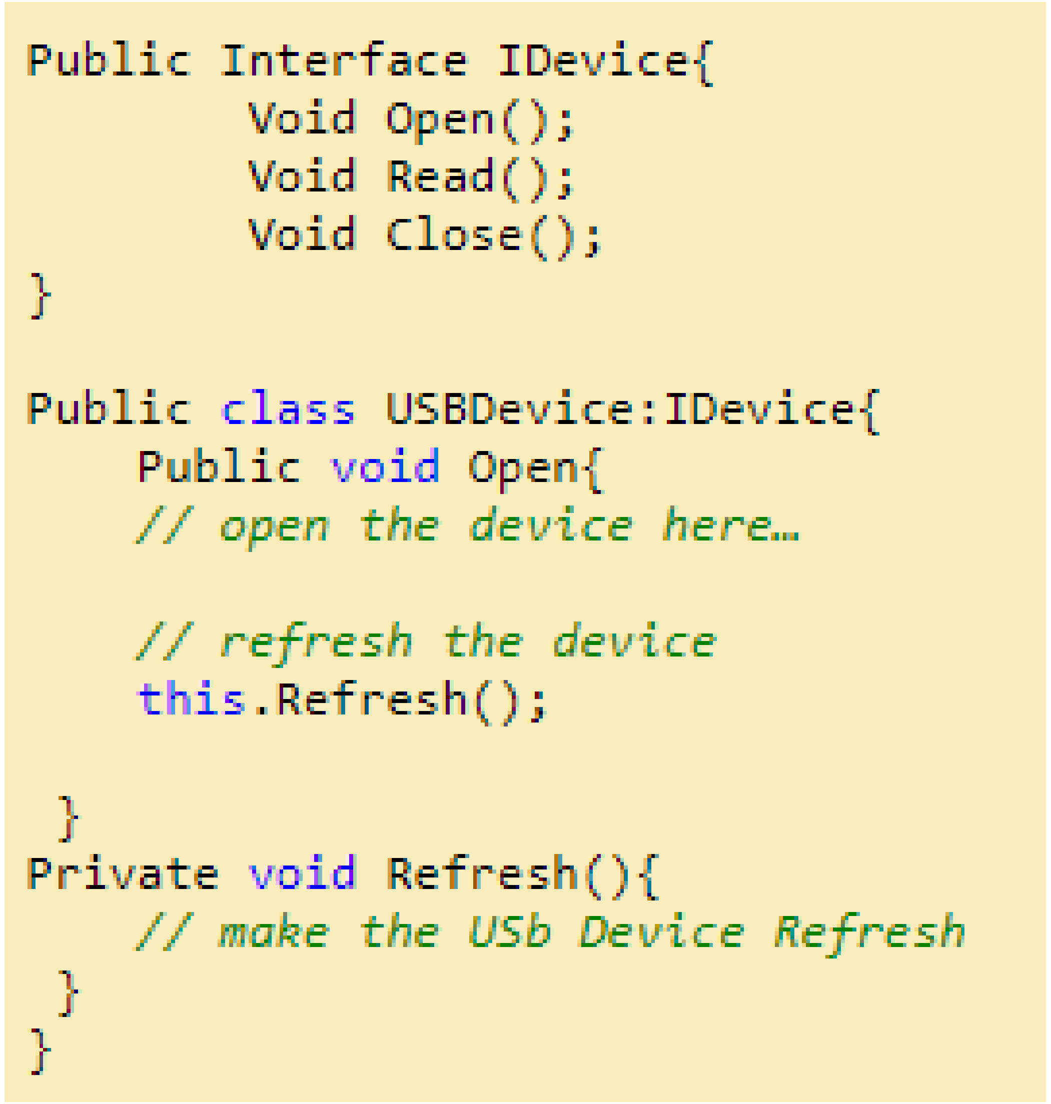
```
 Public Interface IDevice{
    Void Open();
         Void Refresh();
    Void Read();
    Void Close();
}
```

Unnecessary implementation of Refresh( )



```
public  void Refresh()
{
// Yes nothing here… just a useless blank function
}
```

Move Refresh( ) to USB device class:

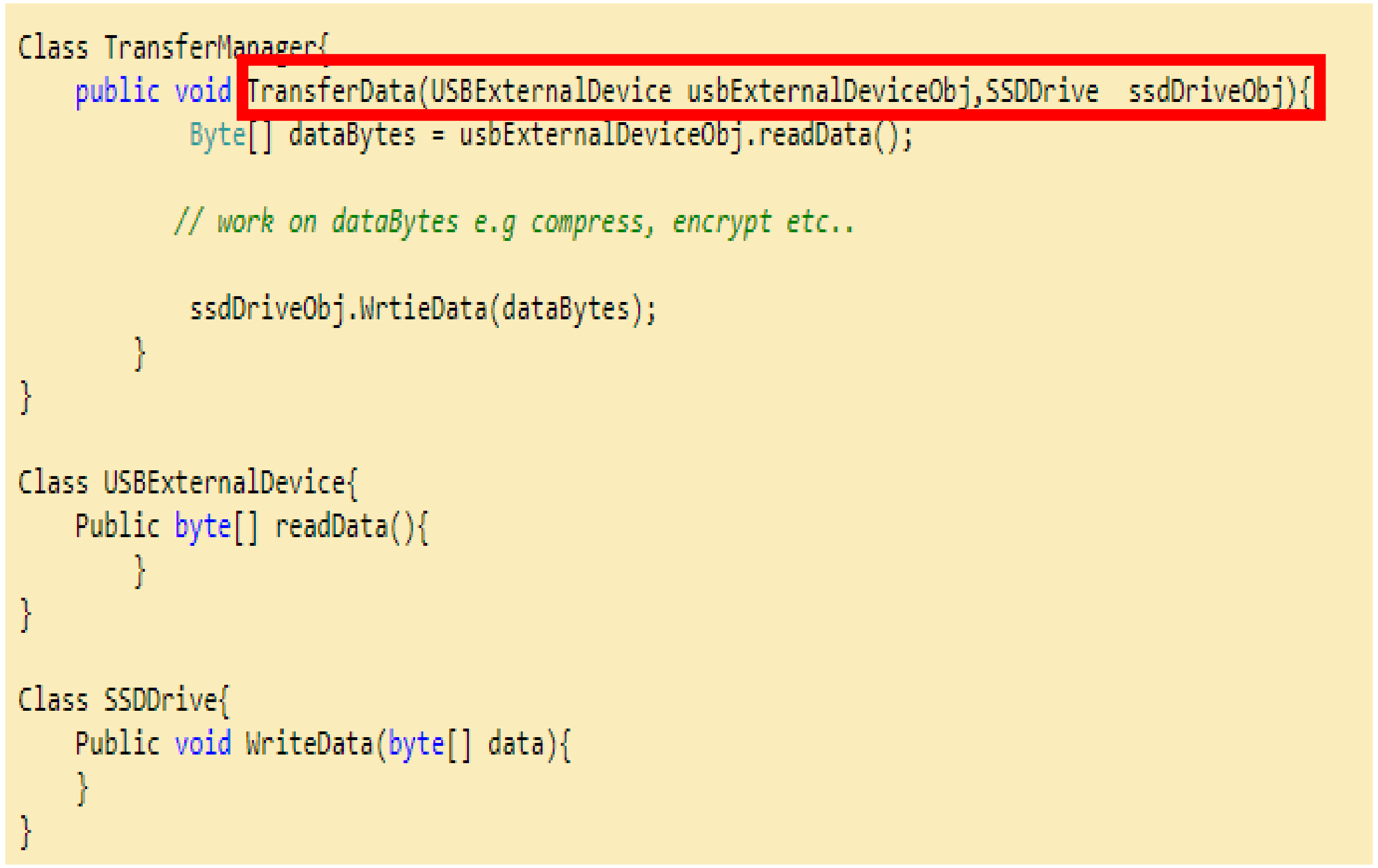


```
Public Interface IDevice{
        Void Open();
        Void Read();
        Void Close();
}

Public class USBDevice:IDevice{
    Public void Open{
    // open the device here…

    // refresh the device
    this.Refresh();

 }
Private void Refresh(){
    // make the USb Device Refresh
 }
}
```

Dependency Inversion Principle (DIP)

"Program to an interface, not to an implementation"

"High-level modules should not depend upon low-level modules. Both should depend upon abstractions."



```
Class TransferManager{
    public void TransferData(USBExternalDevice usbExternalDeviceObj,SSDDrive  ssdDriveObj){
            Byte[] dataBytes = usbExternalDeviceObj.readData();

           // work on dataBytes e.g compress, encrypt etc..

            ssdDriveObj.WrtieData(dataBytes);
        }
}

Class USBExternalDevice{
    Public byte[] readData(){
        }
}

Class SSDDrive{
    Public void WriteData(byte[] data){
    }
}
```

TransferData( ) using low level modules – USBExternalDevice, SSDDrive

If required to add other external devices, will have to change the higher level module. Higher level module will be dependent upon the lower level module and that dependency will make the code difficult to change.

Make both High level and Low level modules depend upon abstractions.

```
Class USBExternalDevice implements IExternalDevice{
    Public byte[] readData(){
    }
}

Class SSDDrive implements IInternalDevice{
    Public void WriteData(byte[] data){
    }
}

Class TransferManager implements ITransferManager{
    public void Transfer(IExternalDevice externalDeviceObj, IInternalDevice internalDeviceObj){
           Byte[] dataBytes = externalDeviceObj.readData();

           // work on dataBytes e.g compress, encrypt etc..

           internalDeviceObj.WrtieData(dataBytes);
        }
}

Interface IExternalDevice{
        Public byte[] readData();
}

Interface IInternalDevice{
    Public void WriteData(byte[] data);
}

Interface ITransferManager {
    public void Transfer(IExternalDevice usbExternalDeviceObj,SSDDrive  IInternalDevice);
}
```

High Cohesion, Low Coupling

- High Cohesion :

Subjective scale, not objective scale of responsibilities

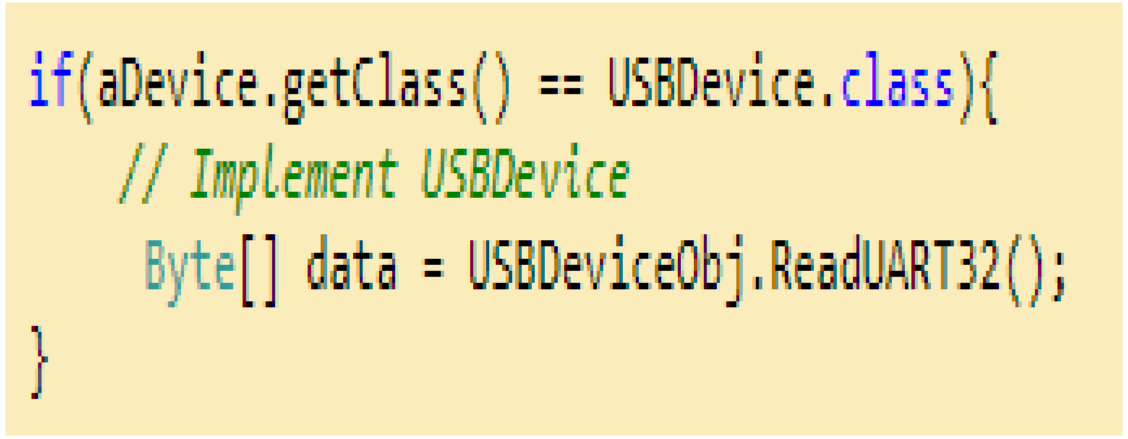
High Coupling
=> Single responsibility

- Low Coupling :

Assign a responsibility so that the dependency between the classes remains low.

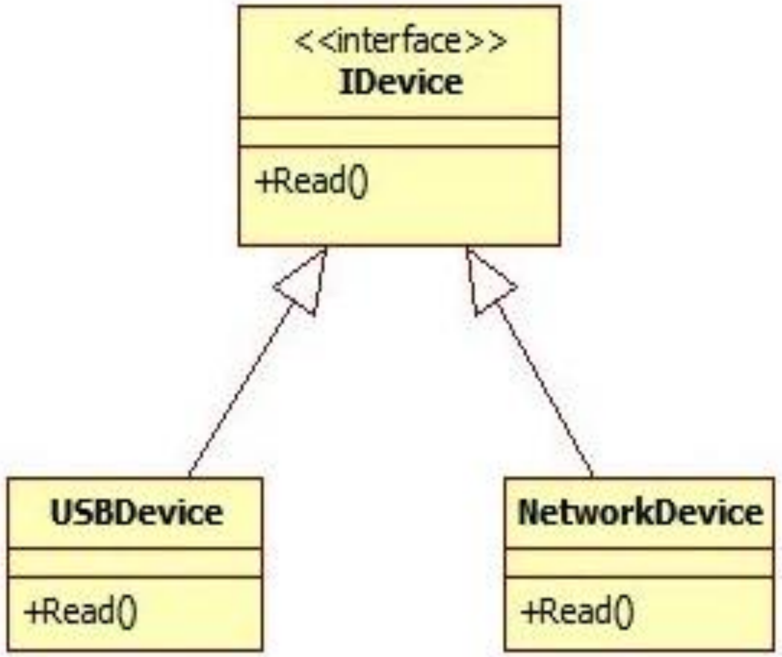
Polymorphism – Is it type safe?

- Restricts use of Run-time type identification(RTTI)
- getClass(), instanceOf() in Java



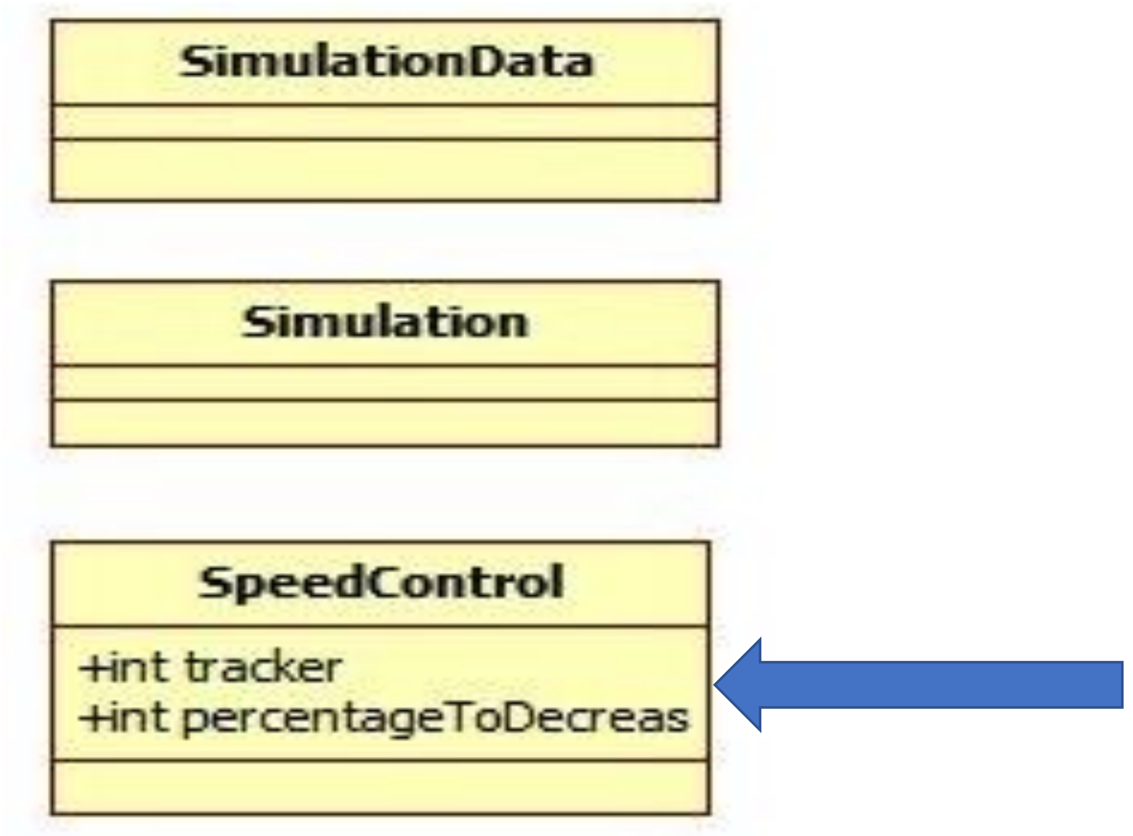
```
if(aDevice.getClass() == USBDevice.class){
    // Implement USBDevice
     Byte[] data = USBDeviceObj.ReadUART32();
}
```

Refactor as



Information Expert

"Assign a responsibility to the class which has the information necessary to fulfill that responsibility."



Assign Responsibility whether to display a next frame or not to SpeedControl

Creator

- Who should create an instance of a class?

According to Larman, a class “B” should be given the responsibility to create another class “A” if any of the following conditions are `true`.

- B contains A
- B aggregates A
- B has the initializing data for A
- B records A
- B closely uses A

Pure Fabrication

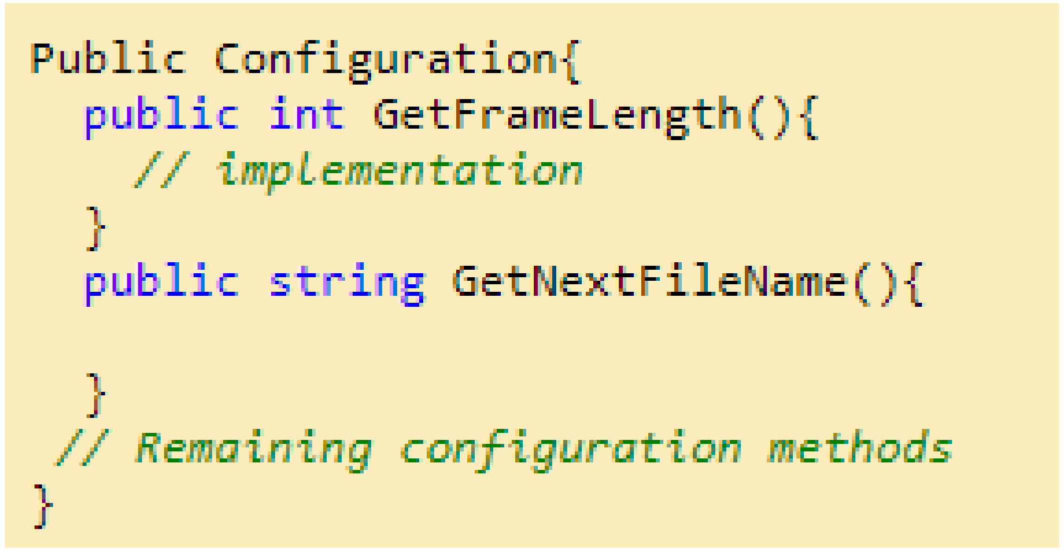
- Identification of classes in your problem domain – Object Oriented Approach
- Banking Example :

Classes - `Account`, `Branch`, `Cash`, `Check`, `Transaction`,….

To store information about customers, introduce another class which does not represent any domain concept.

Ex: PersistenceProvider – purpose is to handle data storage functions

- To decouple the domain code from the configuration, specific classes can be added.



```
Public Configuration{
  public int GetFrameLength(){
    // implementation
  }
  public string GetNextFileName(){

  }
 // Remaining configuration methods
}
```

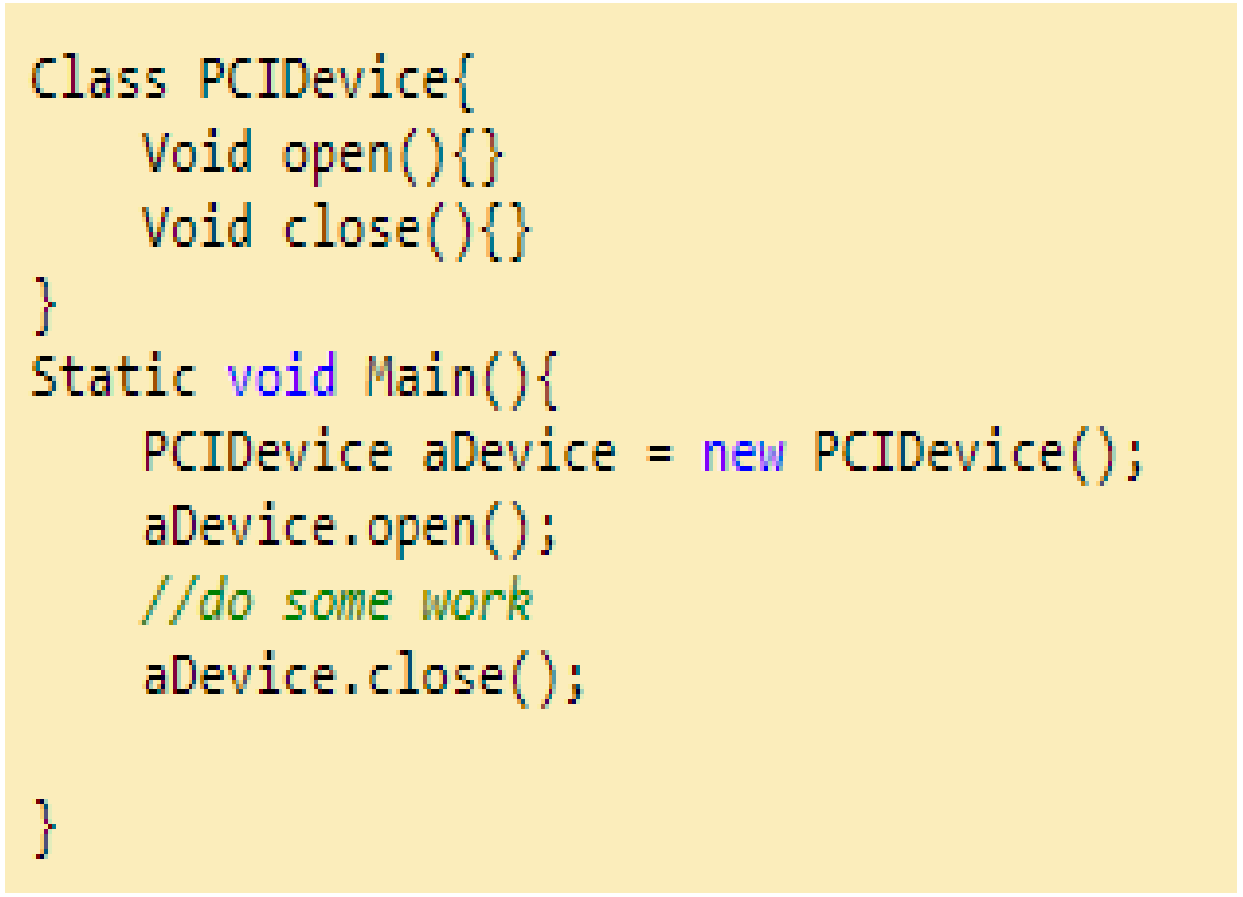
- If any domain object wants to read a certain configuration setting, it will ask the `Configuration` class object. Therefore, main code is decoupled from the configuration code.

Indirection

- Give the responsibility of interaction to an intermediate object so that the coupling among different components remains low.
- This lets objects interact in a manner that the bond among them remains weak.

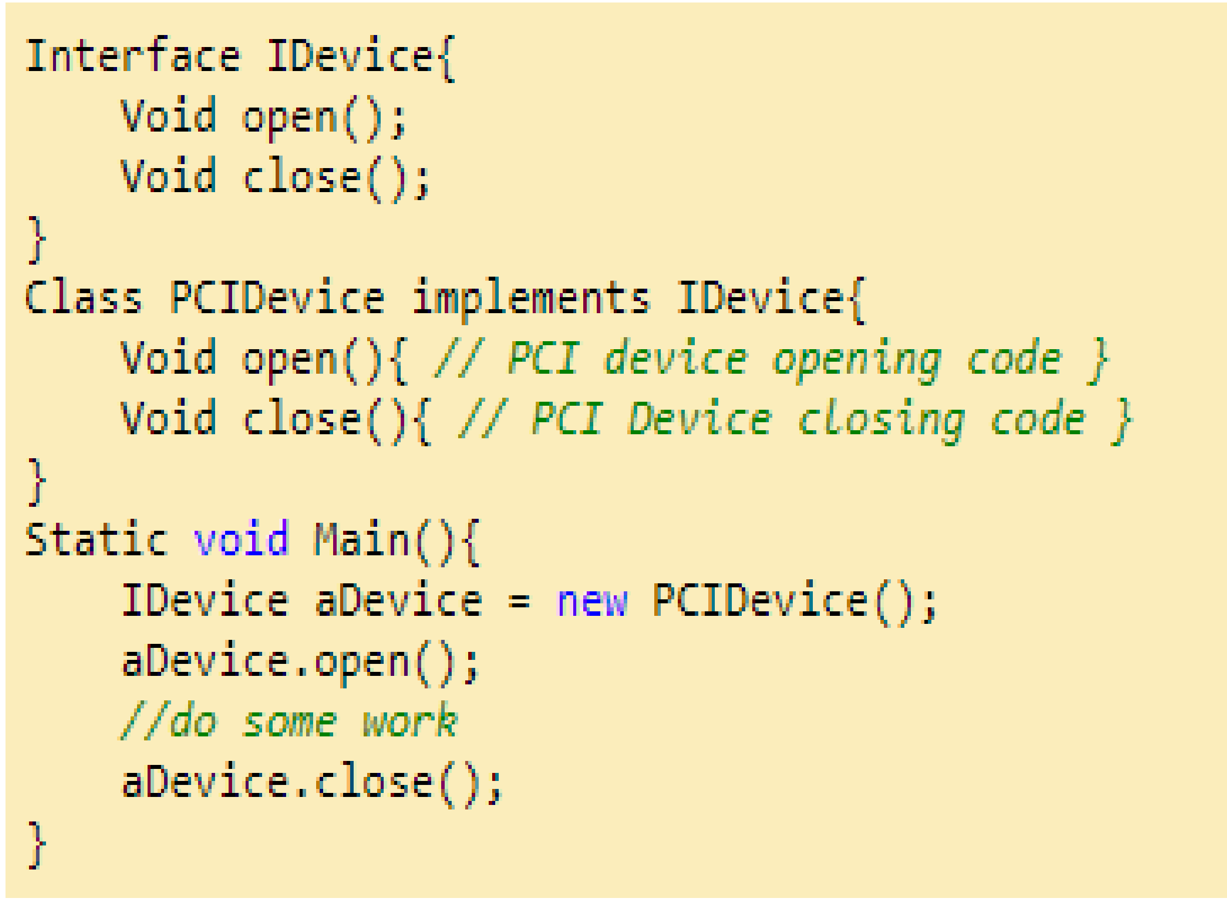
Program to an interface, not to an implementation

Violates Principle



```
Class PCIDevice{
    Void open(){}
    Void close(){}
}
Static void Main(){
    PCIDevice aDevice = new PCIDevice();
    aDevice.open();
    //do some work
    aDevice.close();


}
```

Follows Principle



```
Interface IDevice{
    Void open();
    Void close();
}
Class PCIDevice implements IDevice{
    Void open(){ // PCI device opening code }
    Void close(){ // PCI Device closing code }
}
Static void Main(){
    IDevice aDevice = new PCIDevice();
    aDevice.open();
    //do some work
    aDevice.close();
}
```

How to extend functionality of existing code in OO systems? Reusable code?

Favor Composition over Inheritance

- Helpful feature of the composition is that behavior can be set at run-time
- Using inheritance, behavior can be set only at compile time – Class Explosion
- This feature reduces the total number of classes and ultimately reduces maintainability issues
- *is-a* Vs *has-a* relationship
- What happens to internal details?

Don't Repeat Yourself (DRY)

- Don't try to write the same code again and again.
- Combine frequently written lines of code in a function and call when required.

Beware!

- Context plays a major role in deciding principles!
  - Apply relevant principles!!!
- Principles could be conflicting – Tradeoffs