DESIGN PRINCIPLE

by

VIVEK DUTTA MISHRA

WHAT IS DESIGN PRINCIPLES

- BEST PRINCIPLES IN SOFTWARE DESIGN
- GOING BEYOND INHERITANCE AND POLYMOPHISM
- DO'S AND DONT'S OF DESIGN
- WHAT MAKES A DESIGN GREAT, WHAT DOESN'T

Hardware Vs Software – What's Different

Hardwares wear-tear and Rusts. Softwares Don't

changes are expensive!!! We need to swap our hardware for a newer one!!!

Really???

By the way why is Rusting Bad?

Hardware Vs Software – What's Different

You are right. Rusting force replacement. but How do you answer these?

Hardware Vs Software – What's Different

You are right. Rusting force replacement. but How do you answer these?
That's not rusting!!!

Hardware Vs Software – What's Different

That's not rusting!!!

You are right. Rusting force replacement. but How do you answer these?

But it has same effect. Right??? Need of change and expensive

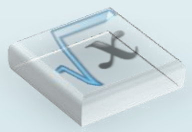
…and ddon't forget memory leak and performance loss in stress condition???

Hardware Vs Software – What's Different

Hardwares can't change. Softwares can

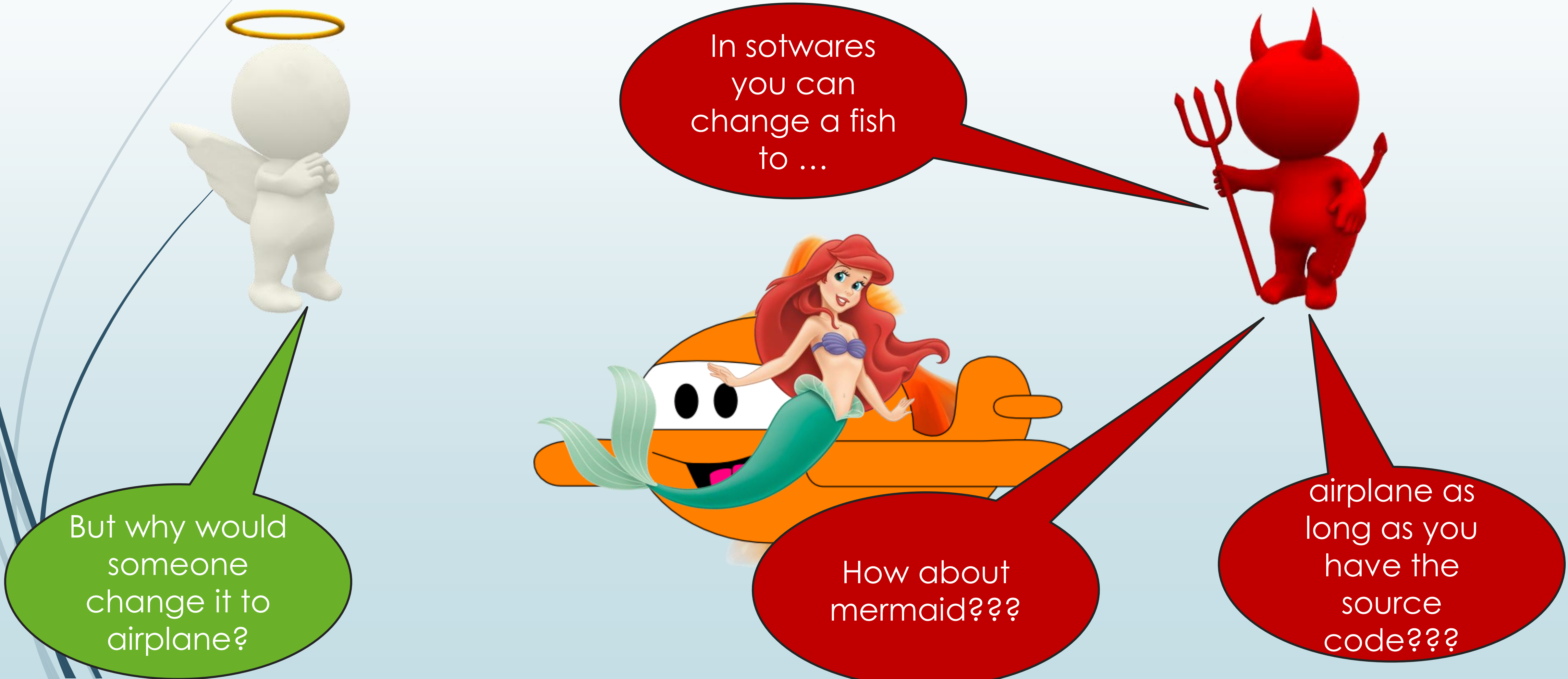
I Bought something

But something is missing √
I can't add it here. But if it was a software…



… You could have changed it. But is change really good? Absolutely good?

Hardware Vs Software – What's Different


In sotwares you can change a fish to …
How about mermaid???
airplane as long as you have the source code???
But why would someone change it to airplane?

Hardware Vs Software – What's Different

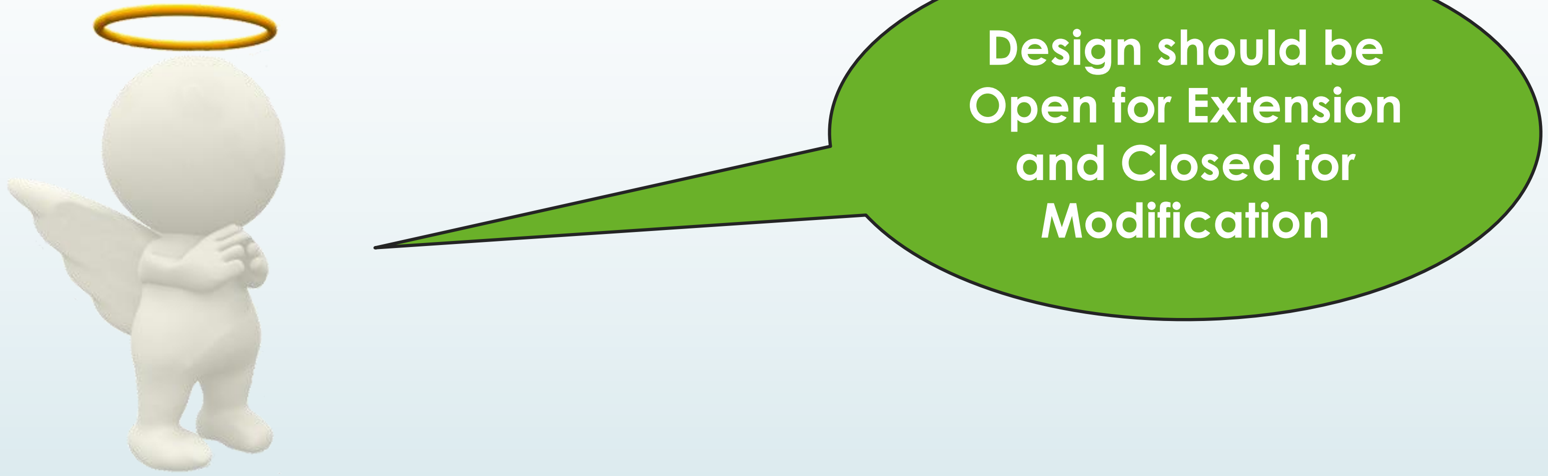
Repair may bring breaking changes

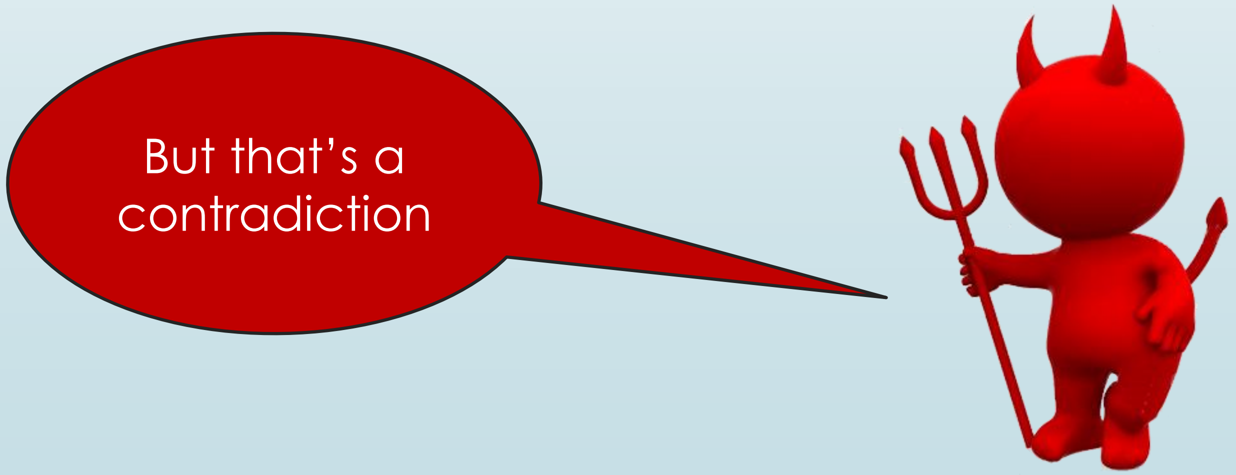
Hardware Vs Software – What’s Different

and not everyone likes same change

Don’t worry we have it covered.

Open Close Principle


Design should be Open for Extension and Closed for Modification



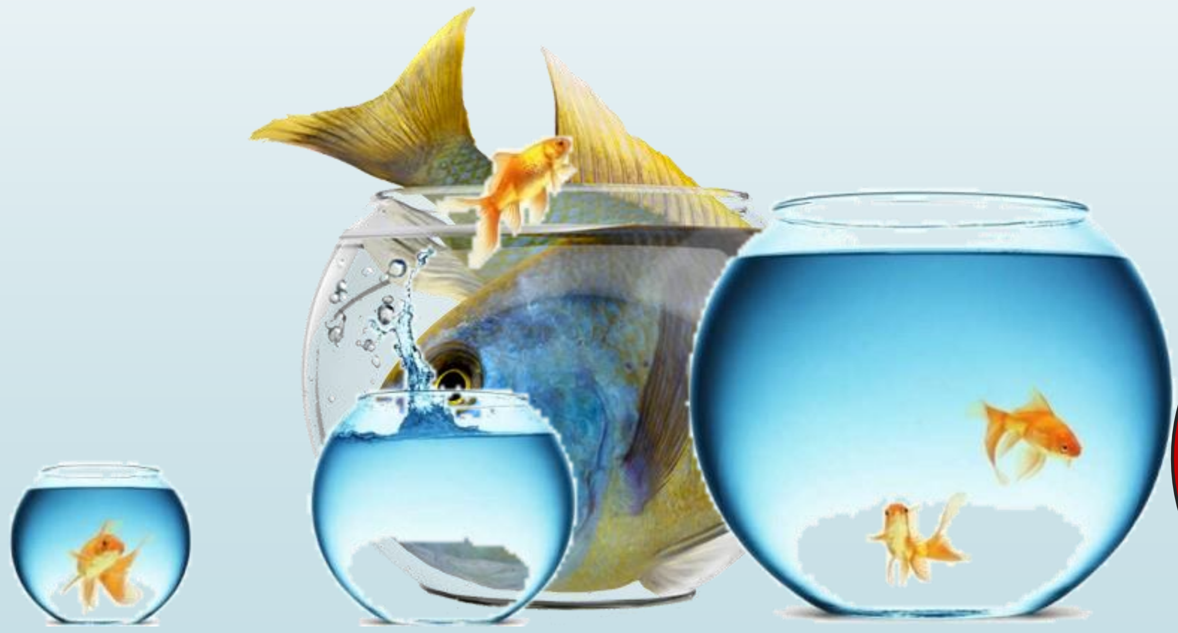
Open For Extension

Eventually Requirement changes
We should be ready **(open)** for change

Open For Extension

Eventually Requirement changes

We should be ready **(open)** for change



Easy!!!
we can always change the code. Lets create a flying Fish

Closed For Modification

That’s where the second part of the rule enters.

Change shouldn’t change existing code

Easy!!! we can always change the code. Lets create a flying Fish

Open Close Principle

- Requirements will change
- Our Design should be ready **(open)** to accommodate changes

- Changes are Expensive
- Change May Break Existing Design
- Each Change Triggers A Cycle Of
  - Compile
  - Test
  - Deploy
  - Distribute
- A Change May not be acceptable to all stake Holders

- That is why It should be **closed** for modification
- Our Design should be ready to accommodate changes
- It should be pro to change not prone to change

Open Close Principle

- Remember Every software is inherently **Open** (as long as you have source code)
- The Design Goal is → It should be **closed** to source level modification.
- Simple Thumb Rule.

- **Don't Mend It If It is Not Broken.**

- But How Do You Propose to Achieve It???

- There is No Single way to Achieve It
  - Or break it
- **Change should be additive**
  - Need new Feature → New Code
  - Modify Existing Feature → New Code
  - Delete Some Feature → New Code

Open Close Principle

- Can A Design be completely closed?

- No
- But often it is not even desirable
- The idea is to reduce the surface area of change
- A change shouldn't have rippling affect.

Open Close Principle Variation

- But I have heard a different variation of Open Close Principle
- ***A Design should be Open for Modification and Closed for Extension.***

- Can You Elaborate?

- Simple. If you have a bug, be **Open to Correction (Modification)**
- *Once the Bug is Fixed Don't* ***(closed)*** *add any thing new (****extension)***

- But Isn't the same thing I suggested?
- **Don't mend if it is not broken.**
- **Close a working code for future changes**

- How can contradictory statements mean the same?

Open Close Principle Variation

- Principles are proposed by different developers independently
- ***They may have used different phrases to mean the same thing.***

- Can You Elaborate?

- My OCP is applied on proven and tested components.
  - You are applying the principle before code is tested bug free.
- I suggested **system** should be **open/future proof.**
  - You apply the principle at source code level.

Be Open in a debate. we may be on the same side.

Disagreement is not a bad thing either

SINGLE RESPONSIBILITY PRINCIPLE (SRP)

- A COMPONENT (OBJECT, CLASS, METHOD) SHOULD HAVE A SINGLE REASON TO EXIST
  - SINGLE REASON TO CHANGE
  - CLOSED FOR ALL BUT ONE REASON
    - PRACTICALLY COMPLETELY CLOSED

- Opposite of Single Responsibility is GOD class

GOD CLASS

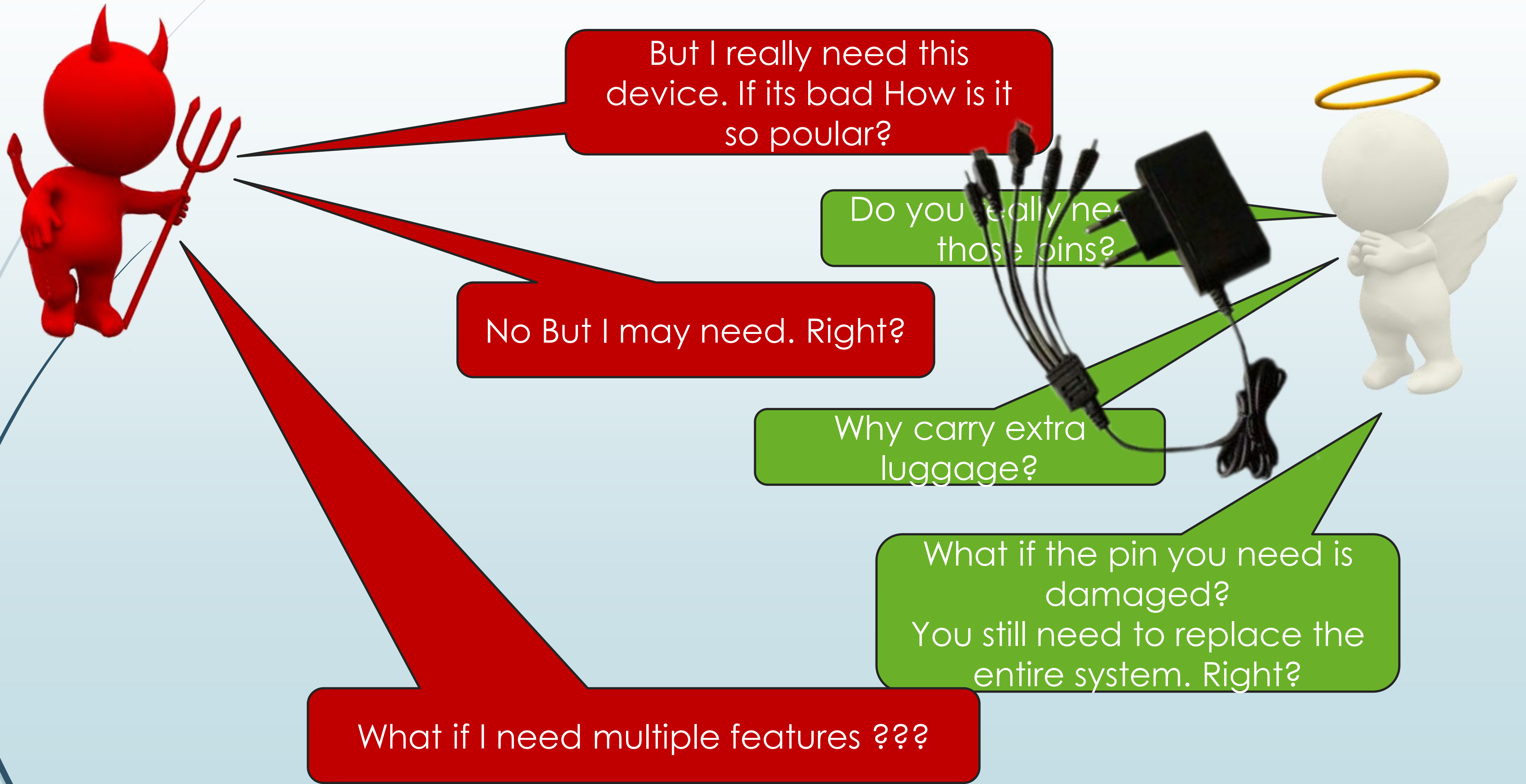
- ALTERNATIVE IS A GOD CLASS.
- A CLASS THAT KNOWS ALL AND DOES ALL.
- ANY CHANGE AND THE DESIGN NEED TO CHANGE

God or Demon???

MULTIPLE RESPONSIBILITY PROBLEM(SRP)

- IF YOU HAVE MORE RESPONSIBILITY
  - YOU ARE MORE HEAVY
  - YOU HAVE MORE REASON TO CHANGE
  - DIFFICULT TO MANAGE
  - MORE OFTEN NEED TO CHANGE
  - MAY NOT PERFORM YOUR RESPONSIBILITY PROPERLY.

Multiple Responsibility Problem


But I really need this device. If its bad How is it so poular?
Do you really need those pins?
No But I may need. Right?
Why carry extra luggage?
What if the pin you need is damaged? You still need to replace the entire system. Right?
What if I need multiple features ???

SINGLE RESPONSIBILITY PRINCIPLE (SRP)

SINGLE RESPONSIBILITY PRINCIPLE (SRP)

DOES SINGLE RESPONSBILITY MEANS A SINGLE METHOD PER CLASS

NO. IT CERTAINLY MEANS FEW METHODS ASSOCIATED WITH THE SAME RESPONSIBILITY

AND NOT TOO MANY METHODS

ASK THE QUESTION. IS THE RESPONSIBILITY RIGHT?

SINGLE RESPONSIBILITY PRINCIPLE (SRP)

A Printer should have Print(), Cancel() and Eject(). But Not Scan. Printers Don't Scan.

BUT PRINTERS DO SCAN. DON'T YOU KNOW?

Look closely. Its not a printer. Its a multi-function device… Its and assembly of printer and scanner.

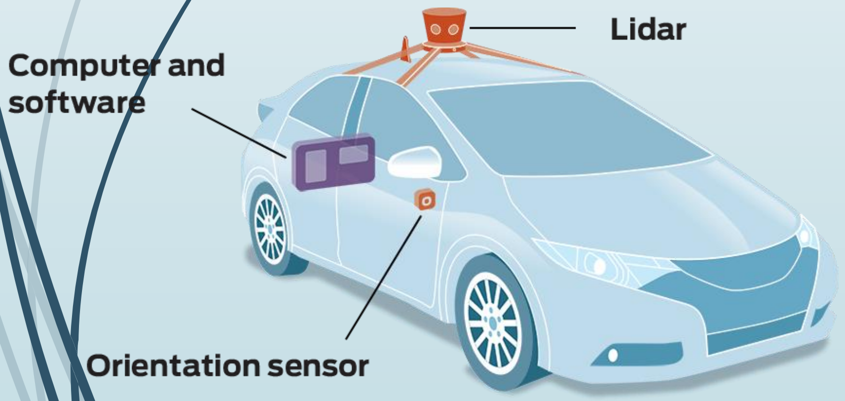
Printers don't scan.

SINGLE RESPONSIBILITY PRINCIPLE (SRP)

A CAR SHOULD HAVE START(), STOP(), MOVE(), TURN() BUT NOT DRIVE(). CARS DON'T DRIVE THEMSELVES.

HAVEN'T YOU HEARD ABOUT GOOGLE SELF DRIVEN CAR?

YOU MAY HAVE DRIVE() METHOD IN THE CLASS. BUT THEN NAME THE CLASS AS GOOGLE CAR.



NORMAL CARS DONT DRIVE THEMSELVES.

SRP – HOW?

1. USE MEANINGFUL NAMES FOR YOUR COMPONENTS
2. A CLASS SHOULDN'T HAVE TOO MANY METHODS
3. MOST OF THE METHODS SHOULD USE MOST OF THE CLASS MOST OF THE TIME.

SRP – USE MEANINGFUL NAMES

1. WITHOUT MEANINGFUL NAMES RESPONSIBILTY CANT BE ACERTAINTED
   - CAN PRINTER PRINT()?
   - CAN FISH FLY() ?
   - CAN FOOO DO ALPHA()?

SRP – USE MEANINGFUL NAMES

1. AVOID NAMES JOINED WITH AND/OR
   - CLEAR VIOLATION OF SRP
2. EXAMPLES 1 - IncomeAndServiceTaxCalculator()
   - Calculates Income Tax and Service Tax
   - Class changes if either of the two rules changes
3. Example 2 - InsertOrUpdate()

4. Example 3- CreateAndAdd()

   - Create and Add operation fused
   - Can't delay Add (Create Now Add Later)
   - Can't add existing
   - Any change in either aspect affects this code

SRP – USE MEANINGFUL NAMES

- Example 1 Refactored
  - ~~IncomeAndServiceTaxCalculator~~ to TaxCalculator
  - Logic is still the name
  - Becomes Harder to Identify violation
- Avoid abstract name for a concrete component
- Avoid too abstract name even for an abstract class
  - Calculator
    - What calculator -> Financial? Arithmetic?

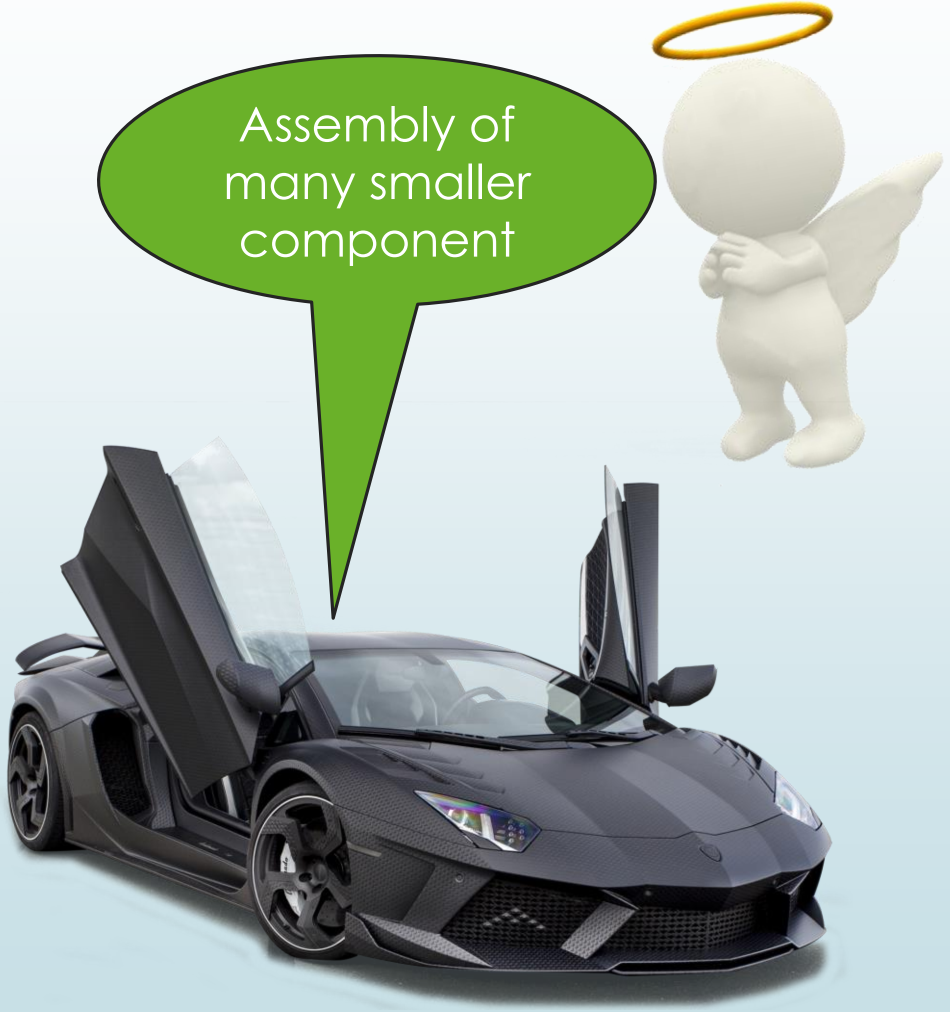
SRP – NOT TOO MANY METHODS

1. EXCLUDING GETTER/SETTER/PROPERTIES A CLASS SHOULDN'T HAVE MORE THAN AGREED (SAY 10) METHODS
   - REFACTOR IF REQUIRED

SRP – COHESIVE

1. MOST OF THE METHODS SHOULD ACCESS MOST OF THE FIELDS MOST OF THE TIME
   - AVOID MUTUALLY EXCLUSIVE
     - METHODS
     - PARAMETERS
     - CODE-BLOCK
2. Check for Parameters/Fields which are always null in a given context.

TOY CAR VS REAL CAR



SRP – FINAL WORDS

1. BUILD LARGE SYSTEMS USING SMALLER COMPONENTS WITH SPECIFIC RESPONSIBILITY
2. Remember the Largest Building is assembled of smaller units with specific responsibility

DONT REPEAT YOURSELF (DRY)

- WHAT
  - AVOID REDUNDANT CODE
  - PREFER REUSE
- WHY
  - EASY TO MANAGE
  - LESS CODE
  - EVOLVES STANDANDARD
    - MORE FUTURE PROOF

HOW TO DRY

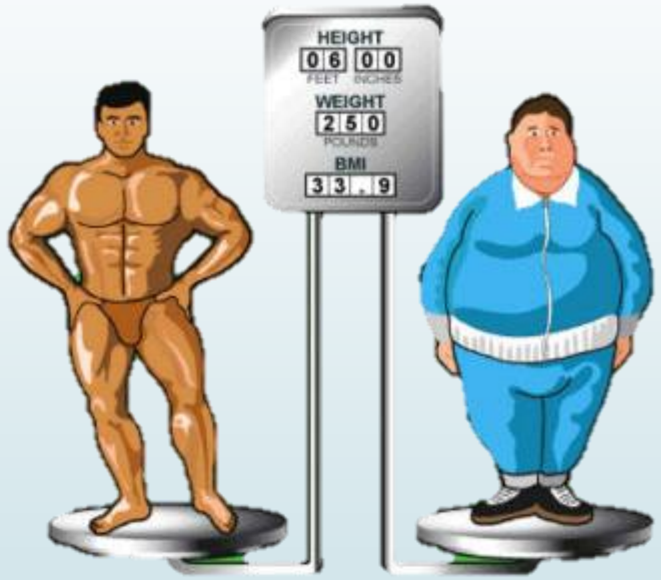
1. ENCAPSULATE WHATEVER REPEATS
    1. WHATEVER REPEATS IS GENERALLY PART OF SINGLE RESPONSIBILITY
    2. AVOIDS REDUNDANT CODE
    3. PROMOTES REUSABILTY
    4. PROMOTES SRP
2. ABSTRACT WHATEVER CHANGES
    1. WHATEVER CHANGES MUST BE A PART OF THE SAME RESPONSIBILTY
    2. PROMOTES SRP
    3. PROVIDES ABSTRACT SOCKET FOR REUSE
3. USE THE COMPONENT FROM STEP 1+2 AS
    1. ABSTRACT BASE
    2. COMPOSITION (PREFERRED)
    3. ASSEMBLES THE SOCKET AND THE PLUGGABLE COMPONENT

INTERFACE SEGREGATION PRINCIPLE (ISP)

- WHAT?
  - AVOID FAT INTERFACE
  - AN INTERFACE SHOULD HAVE ONLY AS MANY METHOD AS EVERY IMPLMENTOR WOULD LIKE TO IMPLEMENT.
  - AVOID MUTUALLY EXCLUSIVE/OPTION BEHAVIORS.
- WHY?
  - FAT INTERFACE → FAT CLASS
  - VIOLATES SRP
- HOW?
  - Break a Large Interface in smaller interfaces
  - Smaller Interfaces can extend each other or
  - Component can implement one or more interfaces
  - Eliminate unwanted interface element.

WHAT IS FAT INTERFACE?

- AN INTERFACE WITH UN-NECESSARY METHODS
- NOT EVERY LARGE INTERFACE IS FAT
  - THEY ALL MAY BE NEEDED
- TOO MANY METHODS IN MOST OF THE CASES WOULD MEAN FAT INTERFACE



FAT INTERFACE

```
public interface IStack{

    void Push(object o);
    Object Pop();
    bool IsEmpty();

    bool IsFull();

    Object Peek();
}
```

A dynamic stack may never be full

Peek() is not always required

INTERFACE SEGREGATION

```
public interface IBasicStack{
     void Push(object o);
    Object Pop();
    bool IsEmpty();
}
```

Core Functionality

```
public interface IStack : IBasicStack{
    bool IsFull()
}
```

Extended Functionality (More Popular Use Case)

```
public interface IPeekable{
    Object Peek();
}
```

Notice: This interface doesn't extend Stack Hierarchy.

IMPLEMENTING INTERFACES

```
public interface FixedStack: IStack{
       // Implements Push,Pop,IsEmpty,IsFull
}

public interface DynamicStack : IBasicStack,IPeekable{
      //implements Push,Pop,IsEmpty,Peek
}

public class SimpleQueue: IQueue,IPeekable{
      //implements Enqueu, Dqueue, Peek
}
```

Implements Push, Pop, IsEmpty, IsFull

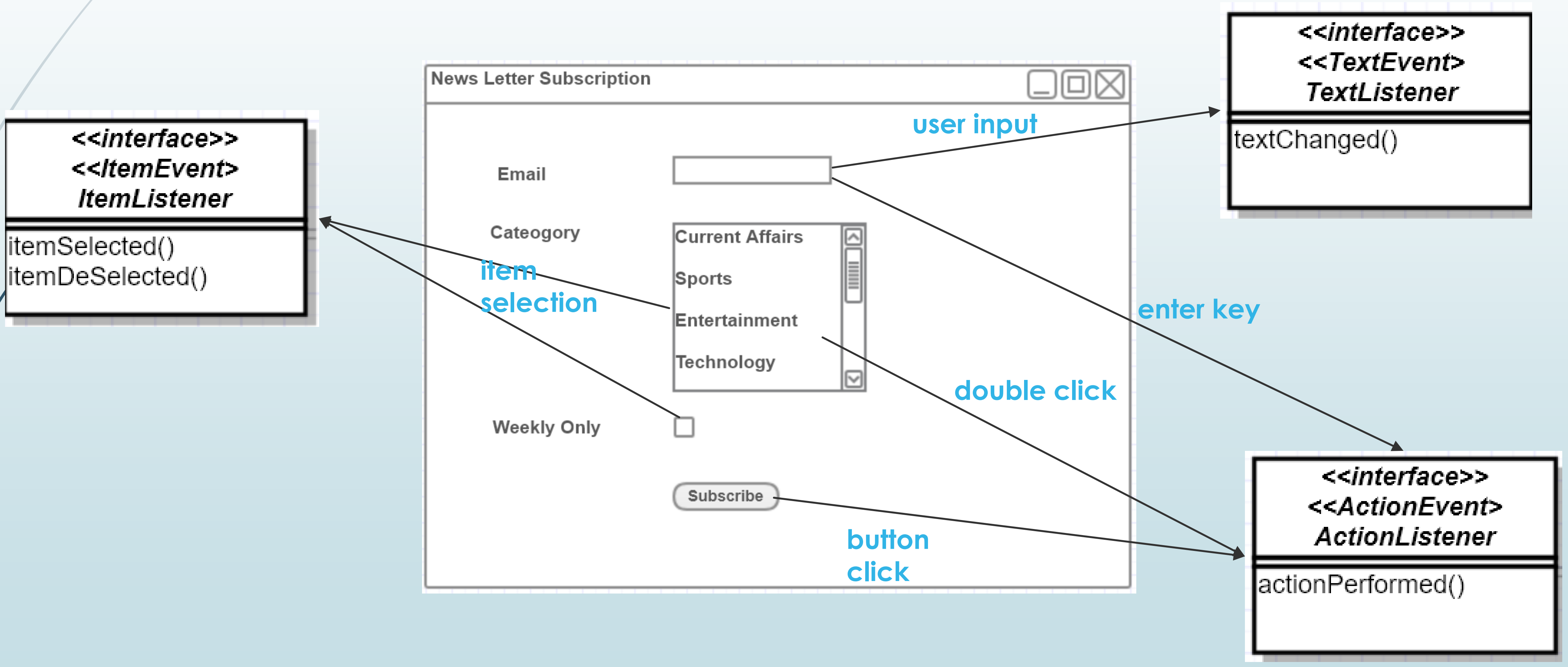
Implements Push, Pop, IsEmpty and Peek

Even a Queue can be Peekable!!!

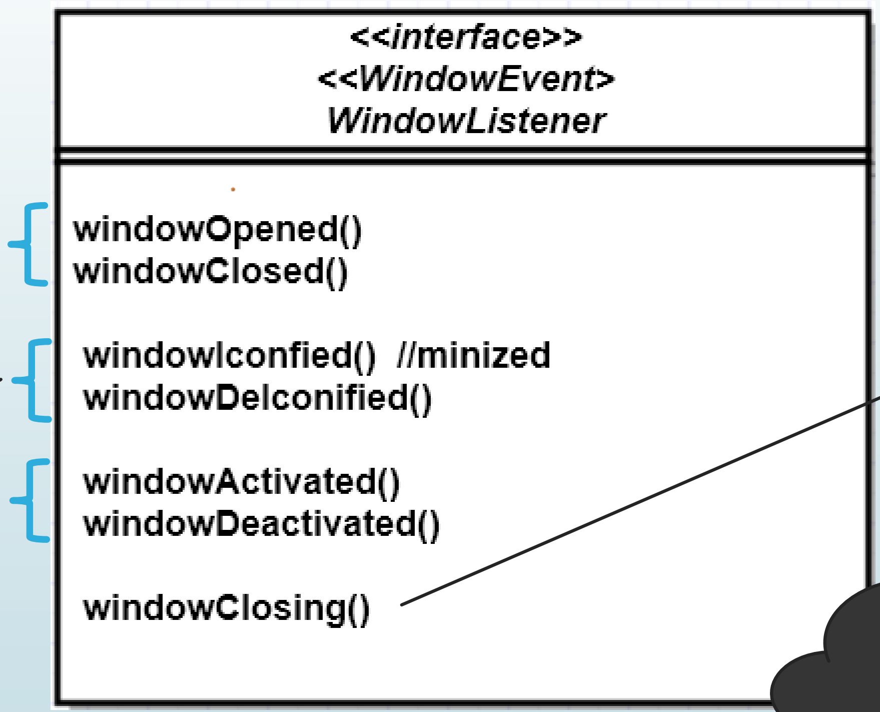
Understanding java.awt

- java.awt is the gui programming model
- GUI elements raises events like → ActionEvent, TextEvent, ItemEvent etc
- For each Event there is a Event Listener interface
- To Handle and Event you need to create class implementing associated listener.
- Attach the object to associate GUI
- Example: To Handle Text Change of TextBox
  1. Create **class MyTextHandler extends TextListner { … }**
  2. Use **textbox.addTextListener(new MyTextHandler());**

Event Handling (Great Design)



Fat Interface (7 Methods)



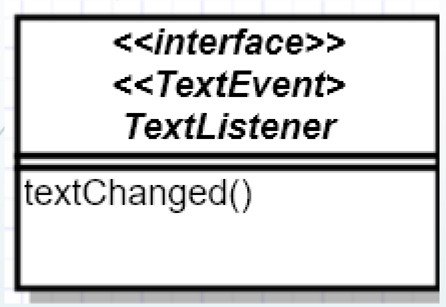
Nobody needs to implement all of them.

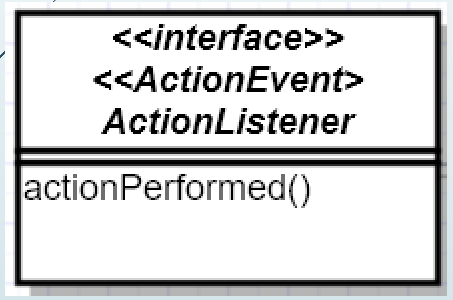
80% cases you need to need only this method

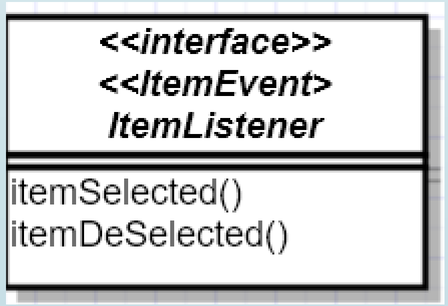
Generally Implemented in Pairs

You need or not you need to implement everything!!!

Whats wrong

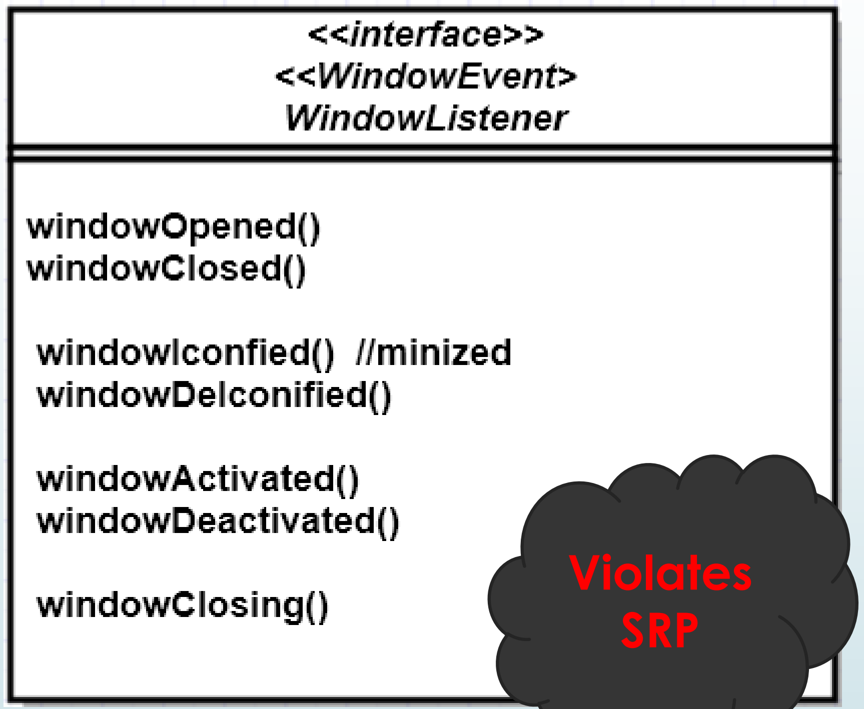






Abstraction of an **Event**

May be fired by different components



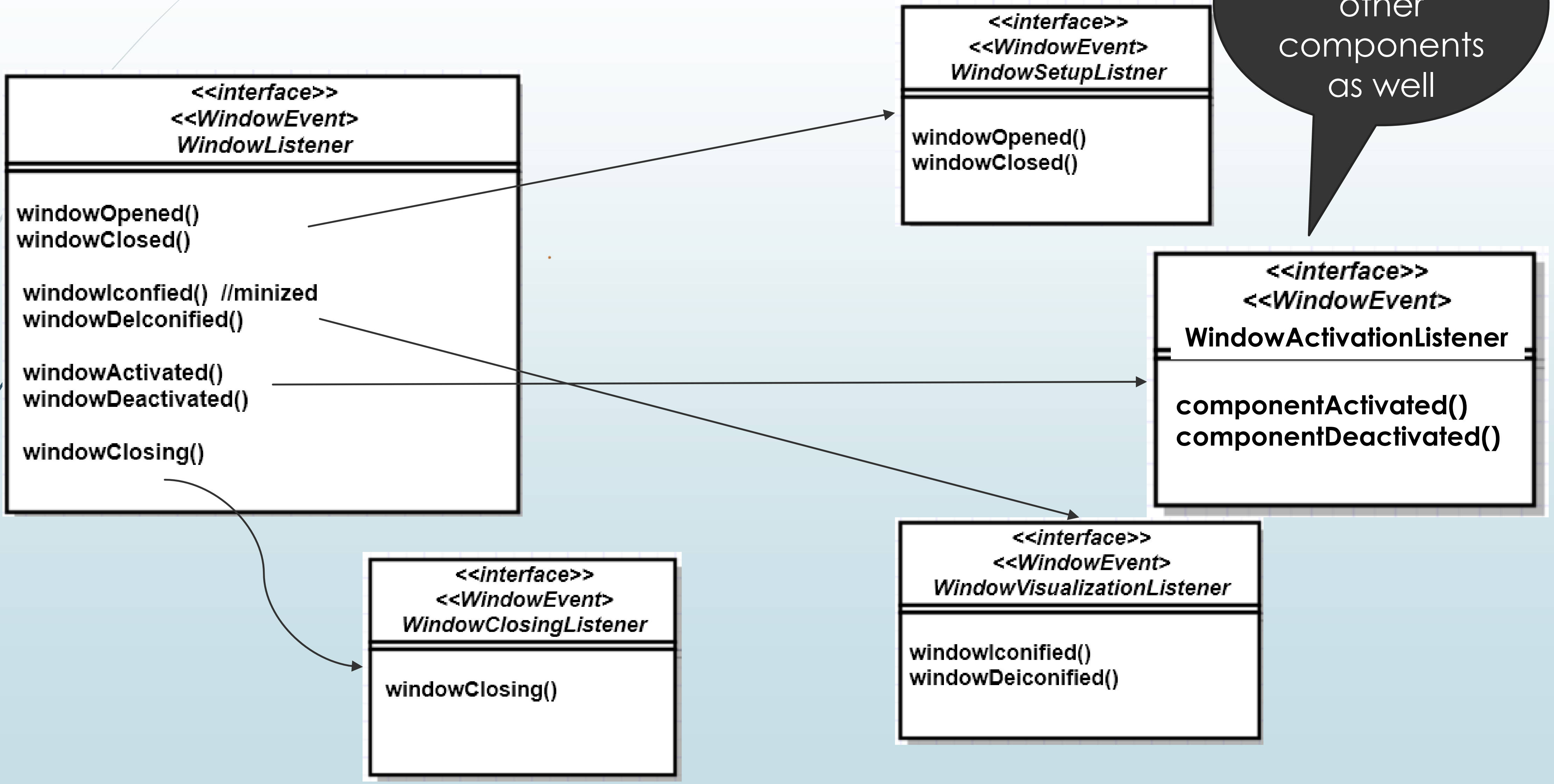
Violates SRP

Abstraction of **All Events** associated with Window

How many Interfaces??

- Should I create 7 interface instead of 1?
- A careful design may reveal 7 interface is not required
  - 4 will do
- But if there is only 2 choices 1 Interface or 7 interface
  - Always choose 7 interface
- Always prefer many smaller components than a few very large one
- Fear to create multiple component is one of prime reasons for bad design
- Smaller components are more likely to be Singly responsible
- More number of component doesn’t really mean more code.

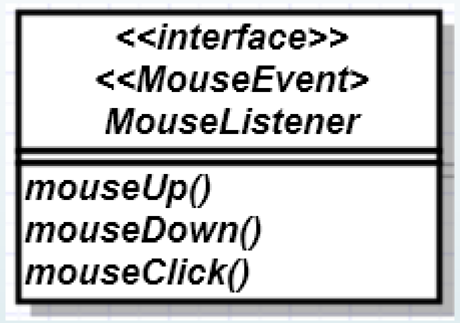
Applying Interface Segregation

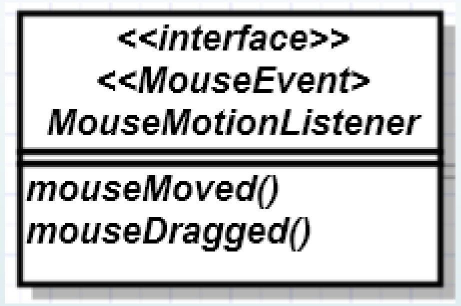


This action applies to other components as well

Bad Segregation

MouseEvent is segregated in two interfaces





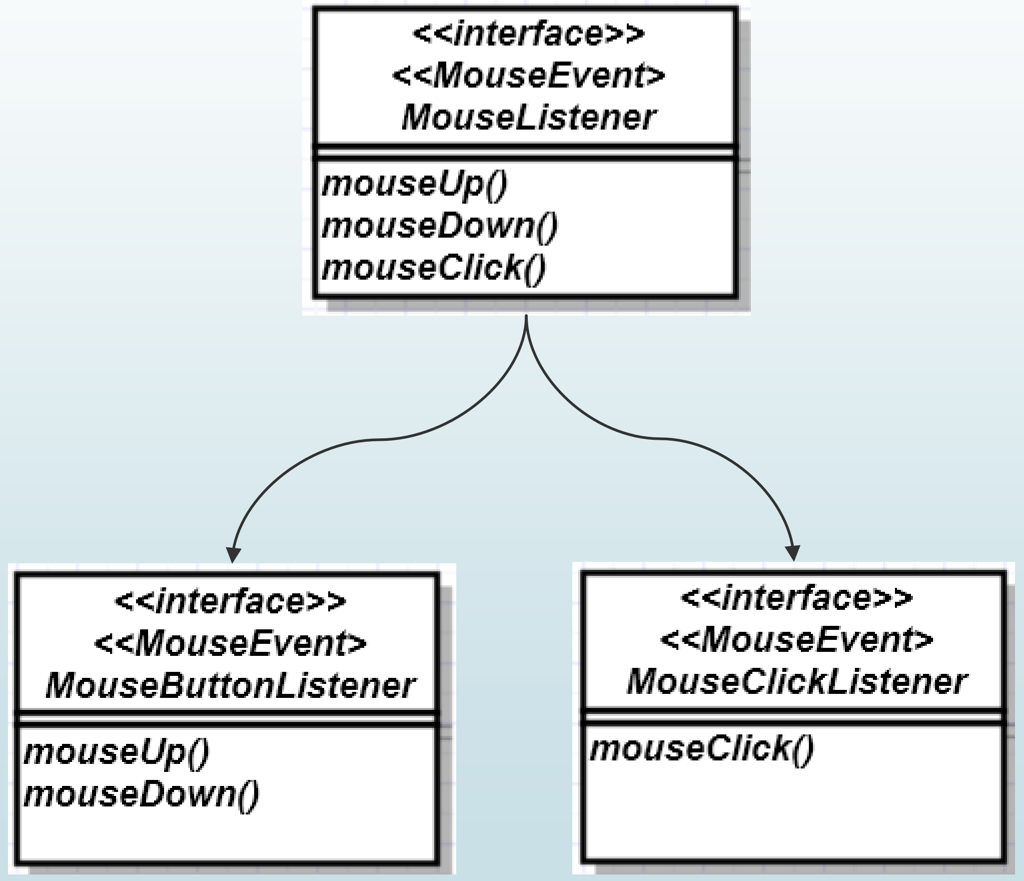
But is it a good segregation?

- **MouseListener has 3 methods**
- **mouseUp() and mouseDown() is always used together**
- **You either use mouseUp()/mouseDown() or mouseClick()**

Interface Segregation - MouseListener

Apply Interface Segregation

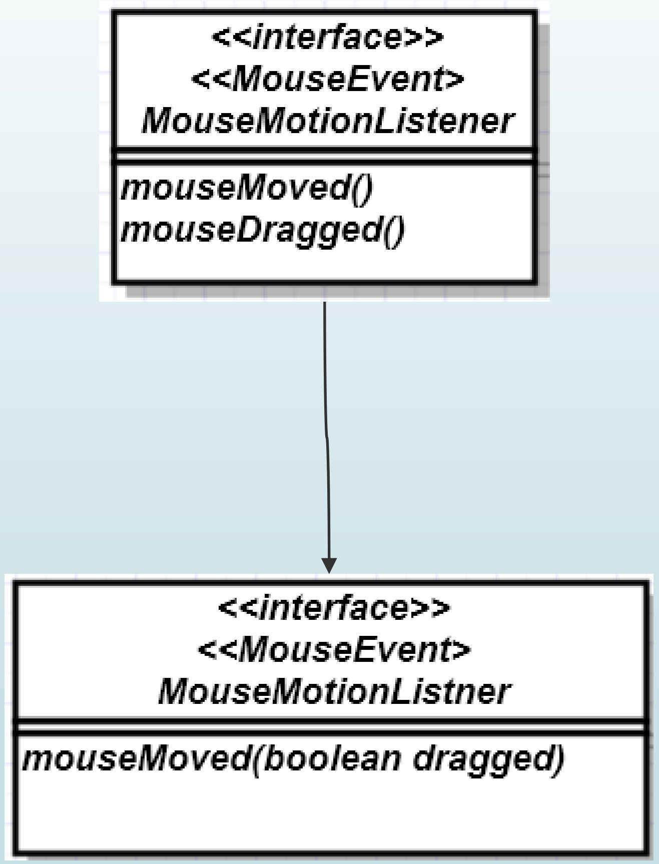
Now User can decide what they need



MouseMotionListener

- **MouseMoved() and mouseDragged() are mutually exclusive**
- **But do I really need to break in two interface?**
- **Do I really need both methods?**

Idea is to reduce un-needed interface, that can be also done by removing methods



Liskov's Substitution Principle (LSP)

- Named after Barbara Liskov, the proposer.
- What?
  - A base component can be replaced by derived component without breaking the client
  - If a client can use base component, it can also use the derived component.
- Why?
  - A change can be introduced as a derived component
  - Design is now **open to extension**
    - By Creating a New Component
  - Its **closed for modification** of
    - Existing Client
    - Existing Components

LSP Is Difficult to Break

- By Design Language Like Java/C# makes violating LSP Difficult
  - You can’t override a public method of base class as private
  - You can’t hide it.
- In C++ although a method can be overridden in private
  - It is still polymorphically callable.
- If a method is available in base class; It is available in derived class.

LSP Is Can Still be Broken

- LSP can be broken if derived class object throws unexpected exception not documented at the base level.
- Client doesn't know about the exception; It hasn't Handled It
- Substituting base component with derived component will cause client to crash.

LSP Consideration

Q. Does LSP Advocate concrete class to concrete class inheritance?

A. No.

- LSP doesn’t decide the design of base class (abstract or concrete).
- LSP recommends design of derived component
  - Shouldn’t introduce breaking changes.

LSP Consideration

Q. Is LSP against throwing Exception?

A. No

- LSP is not against throwing Exception
- LSP is against throwing unexpected exception
- If the Exception is Documented as Base level its is can be thrown
  - Client would be ready to handle such exception.

- Example:
  - Birds Fly but Few of them can't (E.g. Ostrich)
  - If Bird class documents that Fly() may throw **CantFlyException**
    - It is perfectly find to throw that exception.

BankAccount Use Case

- Consider a BankAccount Design from Pre Internet Era.
- The BankAccount is access local (No Network) Data

```
class LocalBankAccount{

    public void Withdraw(...){

        if(...)
            throw new SqlException();

    }

}
```

Client Will be designed to handle SQLException

BankAccount Extended Use Case

- Next Generation Banking is Internet Based.

```
class NetworkBankAccount : LocalBankAccount{

    public void Withdraw(...){

        if(...)
            throw new NetworkException();

    }

}
```

Client isn't designed to handle NetworkException?

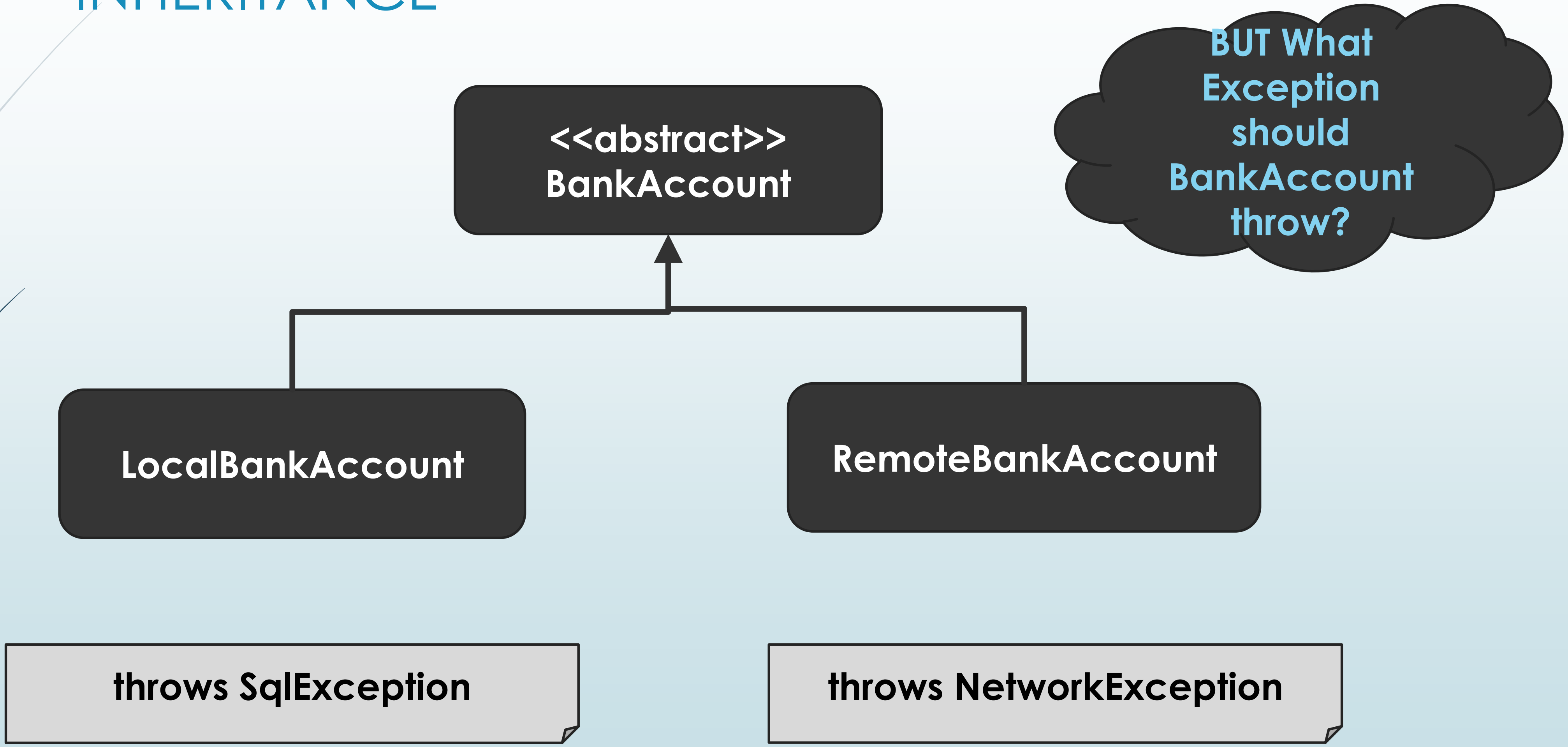
What went wrong?

- Wrong Relationship:
- NetworkBankAccount **is not a type of** LocalBankAccount

```
class RemoteBankAccount : LocalBankAccount{

    public void Withdraw(...){

        if(...)
            throw new NetworkException();

    }

}
```

First we Need the Right Design !!! What is the Right Design?

PREFER ABSTRACT INHERITANCE OVER CONCRETE INHERITANCE



LSP Consideration

Q. Is it possible to know all exception that derived class component will throw?

A. No.

Q. Then how can we document Exception at Base component level

A. In Question Lies the Answer

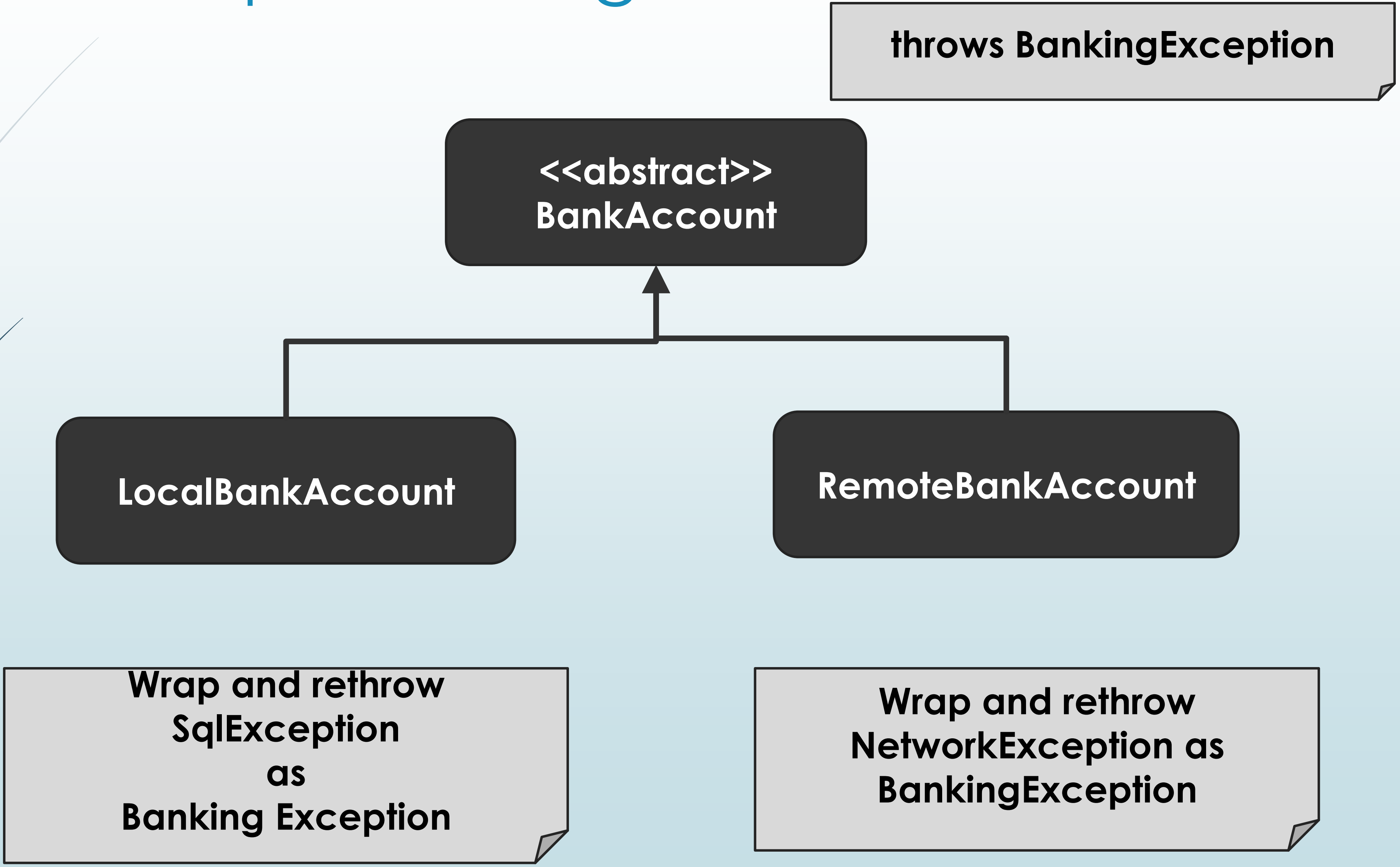
- Why should base class document exception it doesn't know about?
- Why should BankAccount class know about SqlException or NetworkException?

Q. What is the solution?

A. Two Step Solution

1. Define Business Layer Exception
2. Wrap Implemenation Exception in Business Exceptoin

Exception Design



Code Snippet

```
class RemoteBankAccount : BankAccount{

    public void Withdraw(...){

      try{

        if(...)

            throw new NetworkException();

      }catch(NetworkException ex){

        throw new BankingException(ex);

      }

    }

}
```

DEPENDENCY

- WHAT IS DEPENDENCY?
  - KNOWLEDGE IS DEPENDNECY
  - IF YOUR DEPENDENCY CHANGES, IT MAY INDUCE CHANGE IN YOU
    - IT MAY EVEN INDUCE CHANGE IN YOUR DEPENDENTS
  - CHANGE BREAKS OCP

```
class  ComponentX  :  ComponentA
{
    ComponentB bComp;

    public ComponentX( int a, int b)  : base(a)
    {
        bComp=new ComponentB(b);
    }

    public void JobA(){
        base.DoJob();
    }

    public void JobB(){
        bComp.Execute();
    }

    public void JobC(ComponentC cComp){
        cComp.Work();
    }

}
```

ComponentX depends on base ComponentA at 2 Points

Any Change in ComponentA may induce change in ComponentX or its clients

```
class  ComponentX  :  ComponentA
{
    ComponentB bComp;



    public void JobA(){
        base.DoJob();
    }


}
```

Q. What Happens if DoJob() takes new Parameter?

A. Job Need to Take a New Parameter and Pass to DoJob(). This will introduce change in
   - ComponentX
   - Clients of ComponentX

Q. What Happens if DoJob() throws an Exception?

A. DoJob() Need to add try-catch block. This will introduce change in
   - ComponentX

```
class  ComponentX  :  ComponentA
{
    ComponentB bComp;

    public ComponentX( int a, int b) : base(a)
    {
    ...
    }

}
```

Q. What Happens if constructor takes new Parameter?

A. ComponentX Constructor Need to Take a New Parameter and Pass to base class. This will introduce change in
   - ComponentX
   - Clients of ComponentX

Q. What Happens if ComponentA constructor throws an Exception?

A. CANT HANDLE EXCEPTION OF BASE CLASS. MUST PASS TO THE CLIENT.

```
class  ComponentX  :  ComponentA
{
    ComponentB bComp;

    public ComponentX( int a, int b)  : base(a)
    {
        bComp=new ComponentB(b);
    }

    public void JobA(){
        base.DoJob();
    }

    public void JobB(){
        bComp.Execute();
    }

    public void JobC(ComponentC cComp){
        cComp.Work();
    }

}
```

ComponentX depends on encapsulated ComponentB at 2 Points

Any Change in ComponentB may induce change in ComponentX or its clients

Almost as bad as inheritance dependency

```
class  ComponentX  :  ComponentA
{
    ComponentB bComp;


    public void JobB(){
        bComp.Execute();
    }


}
```

Q. What Happens if Execute() throws an Exception?

A. JobB() Need to add try-catch block. This will introduce change in
   - ComponentX

Q. What Happens if Execute() takes new Parameter?

A. JobB() Need to Take a New Parameter and Pass to Execute(). This will introduce change in
   - ComponentX
   - Clients of ComponentX

Same problems as in case of Inheritance

```
class  ComponentX  :  ComponentA
{
    ComponentB bComp;

    public ComponentX( int a, int b)  : base(a)
    {
        bComp=new ComponentB(b);
    }


}
```

Q. What Happens if ComponentB constructor takes new Parameter?

A. ComponentX Constructor Need to Take a New Parameter and Pass to ComponentB constructor. This will introduce change in
   - ComponentX
   - Clients of ComponentX

Q. What Happens if ComponentB constructor throws an Exception?

A. ComponentX Constructor can define the try-catch block.
   - ComponentX

Better than Inheritance

```
class  ComponentX  :  ComponentA
{
    ComponentB bComp;

    public ComponentX( int a, int b)  : base(a)
    {
        bComp=new ComponentB(b);
    }

    public void JobA(){
        base.DoJob();
    }

    public void JobB(){
        bComp.Execute();
    }

    public void JobC(ComponentC cComp){
        cComp.Work();
    }

}
```

ComponentX depends on associated ComponentC

This dependenchy is better than other dependencies. Why?

No constructor dependnecy

What have we Learnt?

- All Dependencies are bad
- Change in Dependencies may induce change in you
  - Sometimes those dependencies may even induce change in your dependents.
- The more number of points, you depend, more is the probability of change.
- Constructor dependencies causes more problem.

Dependency Management

- There are several ways to Manage Dependencies

1. Reduce the Dependency points
2. Invert the Dependency
3. Stable Dependency

Dependency Reduction

```
class  ComponentX  :  ComponentA
{
    ComponentB bComp;

    public ComponentX( int a,   ~~int b~~   ComponentB bComp   ) : base(a)
    {
        ~~this.bComp=new ComponentB(b);~~

        this.bComp=bComp;

    }

}
```

Dependency on Encapsulated Constructor is removed.

Dependency on base constructor can't be removed

Base class constructor

Q. Can we Ensure Our class won’t change because of base class constructor?

A. Only if

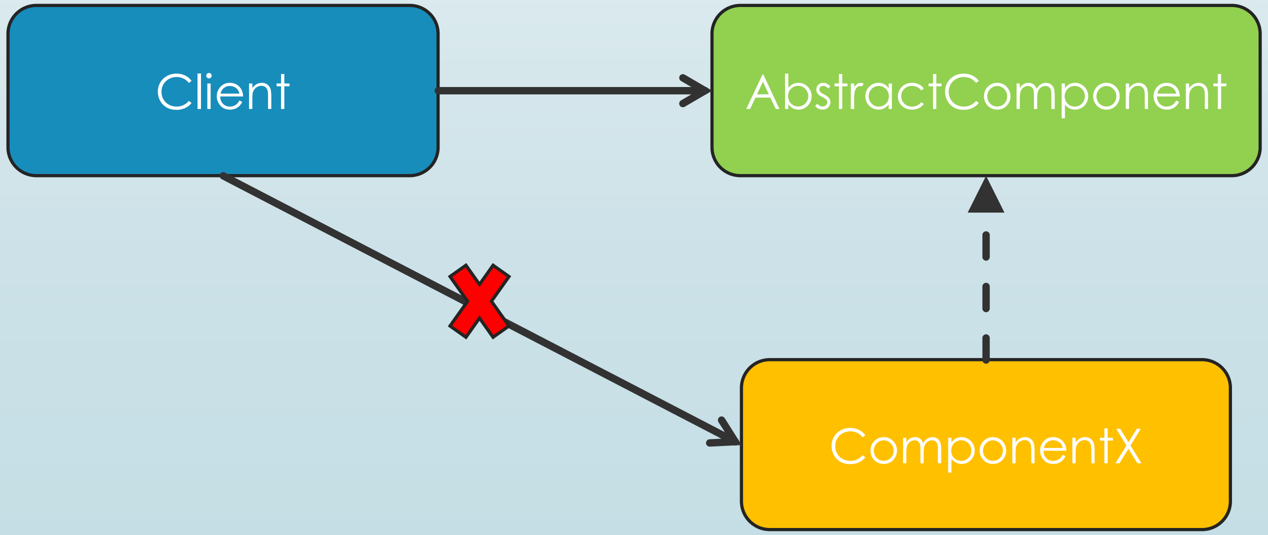
1. Base class constructor Never Change is Parameter
   - Only If Base class constructor takes NO Parameter
2. Base class constructor Never Throws an Exception
   - Only If Base class does nothing

Q. When will these condition be fulfilled?

A. Only If the Base class is Abstract

DEPENDENCY INVERSION

- WHAT?
  - INSTEAD OF A CONCRETE COMPONENT (CLIENT) DEPENDING ON ANOTHER CONCRETE COMPONENT; BOTH SHOULD DEPEND ON A COMMON ABSTRACTION.
  - COMPONENT SHOULD IMPLEMENT ABSTRACTION
  - CLIENT SHOULD USE ABSTRACTION



DEPENDENCY INVERSION

```
class  ComponentX  :  AbstractA
{

    AbstractB bComp;

    public ComponentX(  AbstractB bComp) ~~: base(a)~~
    {
        this.bComp=bComp;
    }
    public void JobA(){
        base.DoJob();
    }

    public void JobB(){
        bComp.Execute();
    }

    public void JobC(AbstractC cComp){
        cComp.Work();
    }

}
```

Inherit from an Abstract Class

Compose Abstract Component

Use Abstract Component

Computer HardDisk Use Case

```
class HardDisk{
    int capacity;
    public HardDisk(int capacity){
        this.capacity=capacity;
    }

    public void Write(…){…}
    public byte[] Read(…){…}
}
```

Case 1: Computer Inherits HardDisk

```
class Computer  : HardDisk
{
      public Computer(int capacity):base(capacity){
      }

      public void Save(){
            base.Write(…);
      }
}

void main(){
        Computer c1=new Computer(512);
        c1.Save(…);

        //what if HardDisk crashes

}
```

Bad Relationship. Computer is Not a HardDisk

class to class relationship. Class computer knows class hardDisk.

If HardDisk crashes computer is throw-away.

Case 2: Computer Has HardDisk

```
class Computer ~~: HardDisk~~
{
    HardDisk hdd;
   public Computer(int capacity)~~:base(capacity)~~{
          hdd=new HardDisk(hdd);
   }

   public void Save(){
        base.Write(…);
   }
}

void main(){
     Computer c1=new Computer(512);
     c1.Save(…);

     //what if HardDisk crashes

}
```

Good Relationship. Computer Has a HardDisk

Bad Responsibility Computer creating HardDisk

If HardDisk crashes computer is **still a** throw-away.

Welded code. No replacement socket

still. class to class relationship . Class computer knows class hardDisk.

Case 2.1: Dependency Injection

```
class Computer
{
      HardDisk hdd;
       public Computer(HardDisk hdd)
                this.hdd=hdd;
       }
        public void SetHardDisk(HardDisk hdd){this.hdd=hdd;}

       public void Save() {   hdd.Write(…);   }
}

void main(){
        Computer c1=new Computer(new HarDisk(512));
        c1.Save(…);

        //what if HardDisk crashes
        c1.SetHardDisk(new HardDisk(1024));
}
```

computer not creating HardDisk

extension socket

dependency injection

If computer crashes…

still. class to class relationship. Class computer knows class hardDisk.

Where we stand

- My Computer Has A HardDisk
  - It doesn't create the HardDisk
- HardDisk can be replaced
  - We defined Extension Sockets
    - Constructor
    - Setter
- Process of Supplying (connecting) Dependency to Extension socket is known as **Dependnecy Injection.**

Types of Dependency Injection

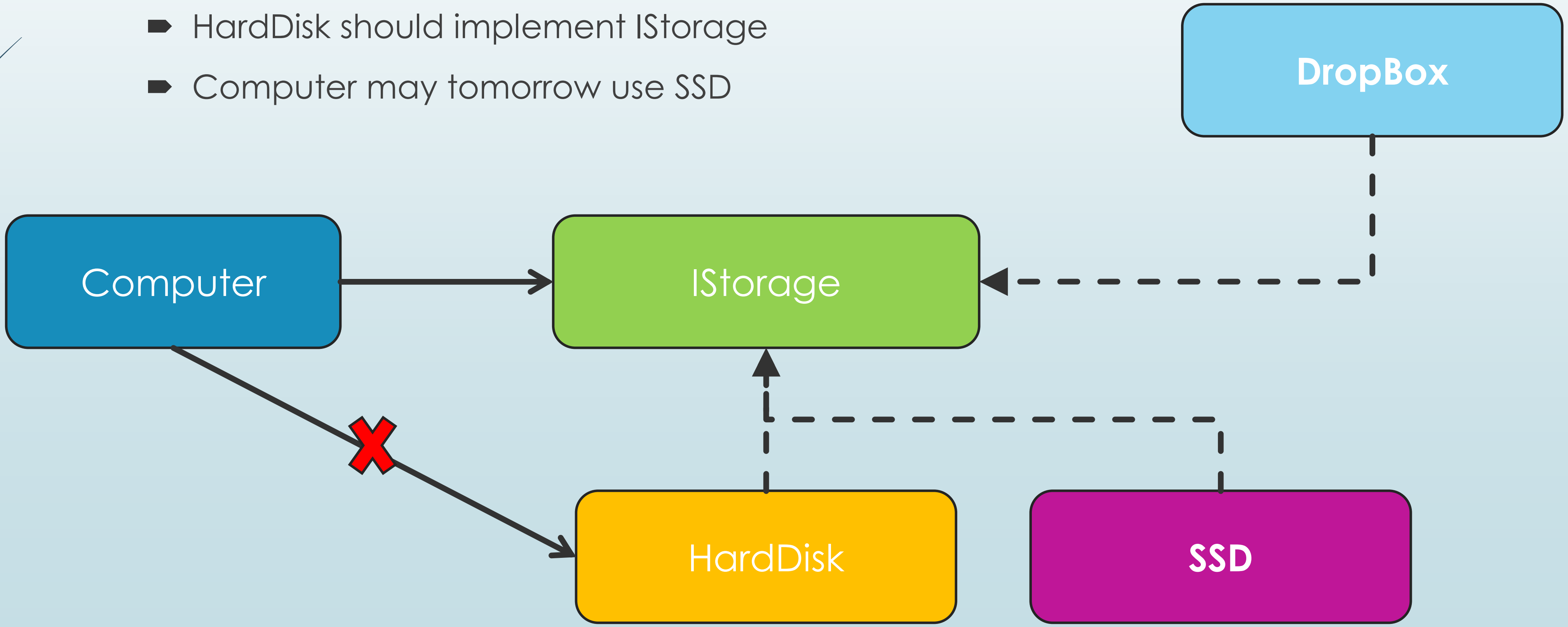
- Constructor Based
  - Supply Dependency as Constructor parameter
  - Generally used for compulsory dependencies
- Setter Based
  - Supplied using Setter
  - May be used for optional dependency
  - May be used for changing dependency
- Method Based
  - Directly supplied to method needing it
  - Generally specific to a particular method

Design Limitation

- Class Computer knows Class HardDisk
- It’s a class to class relationship
- HardDisk can be replaced with HardDisk but not with SSD or MicrosSD

DEPENDENCY INVERSION

- WHAT?
  - Computer Instead of Using HardDisk can use IStorage
  - HardDisk should implement IStorage
  - Computer may tomorrow use SSD



HardDisk is a Storage

```
interface IStorage{
    void Write(…){…}
    byte[] Read(…){…}
}
class HardDisk: IStorage{…}
class SSD: IStorage{…}
class DropBox: IStorage{…}
```

Case 3: Dependency Inversion

This is Object Oriented Design

```
class Computer
{
     IStorage storage;
     public Computer(IStorage storage)
              this.storage=storage
     }
      public void SetStorage(IStroage s){storage=s;}

     public void Save() {   storage.Write(…);   }
}

void main(){
       Computer c1=new Computer(new HardDisk(512));
       Computer c2=new Computer(new SSD(256));
       c1.Save(…);

       //what if HardDisk crashes
       c1.SetStorage(new SSD(512);
}
```

Class Computer doesn't know Class hardDisk. No Class to Class relationship

Object c1 knows HardDisk.
Object c2 knows SSD

Object c1 now knows object SSD

Default (Opinionated) Dependency

- Computer still comes with a default Storage. Why?
- For Convenience
- For People who may not know what they need
- It is a Recommendation (Opinion)
- Is it not violating Dependency Inversion
  - No
  - We Still have option to override default choice.

Case 3.1: Opinionated Dependency

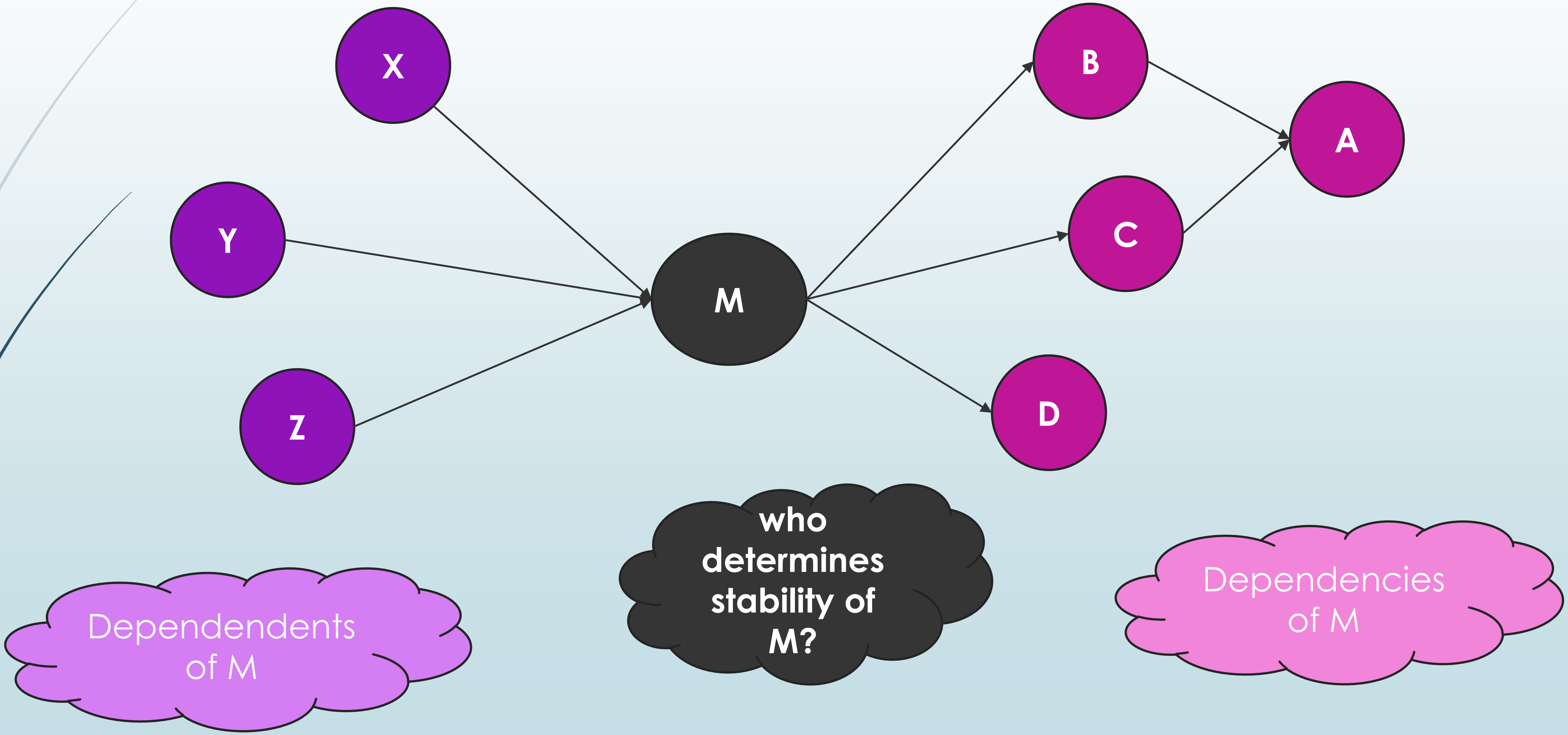
```
class Computer
{
      IStorage storage;
       public Computer(IStorage storage)
                this.storage=storage
       }
        public void SetStorage(IStroage s){storage=s;}

       public Computer() { storage=new HardDisk(1024); }
}

void main(){
        Computer c1=new Computer(new HardDisk(512));
        Computer c2=new Computer(new SSD(256));

        Computer c3=new Computer();
}
```

Default Dependency If user choose not to decide.

Stable Dependency Principle (SDP)

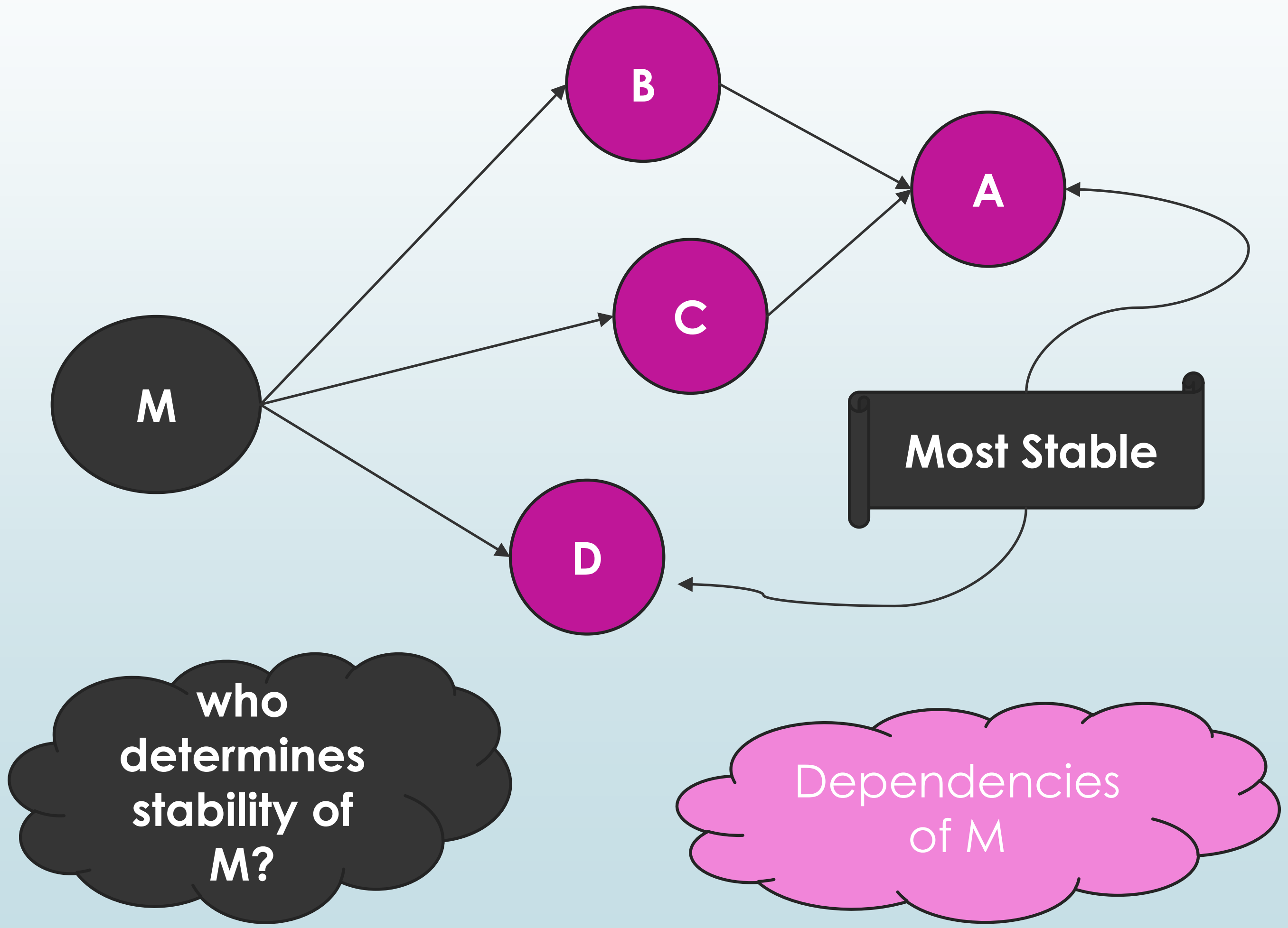
- What?
  - A component must depend on a component more stable than itself
  - It is acceptable to depend on a concrete component as long it is a stable component.
- Why?
  - A component may change due to 2 reasons
    1. A change in the core responsibility of the component itself
    2. A change induced by change in one of its dependency.
  - Stable Dependency Ensures that you don't change because of others (Point 2)
  - Stable Dependency makes you Stable

How to Identify a Stable Dependency

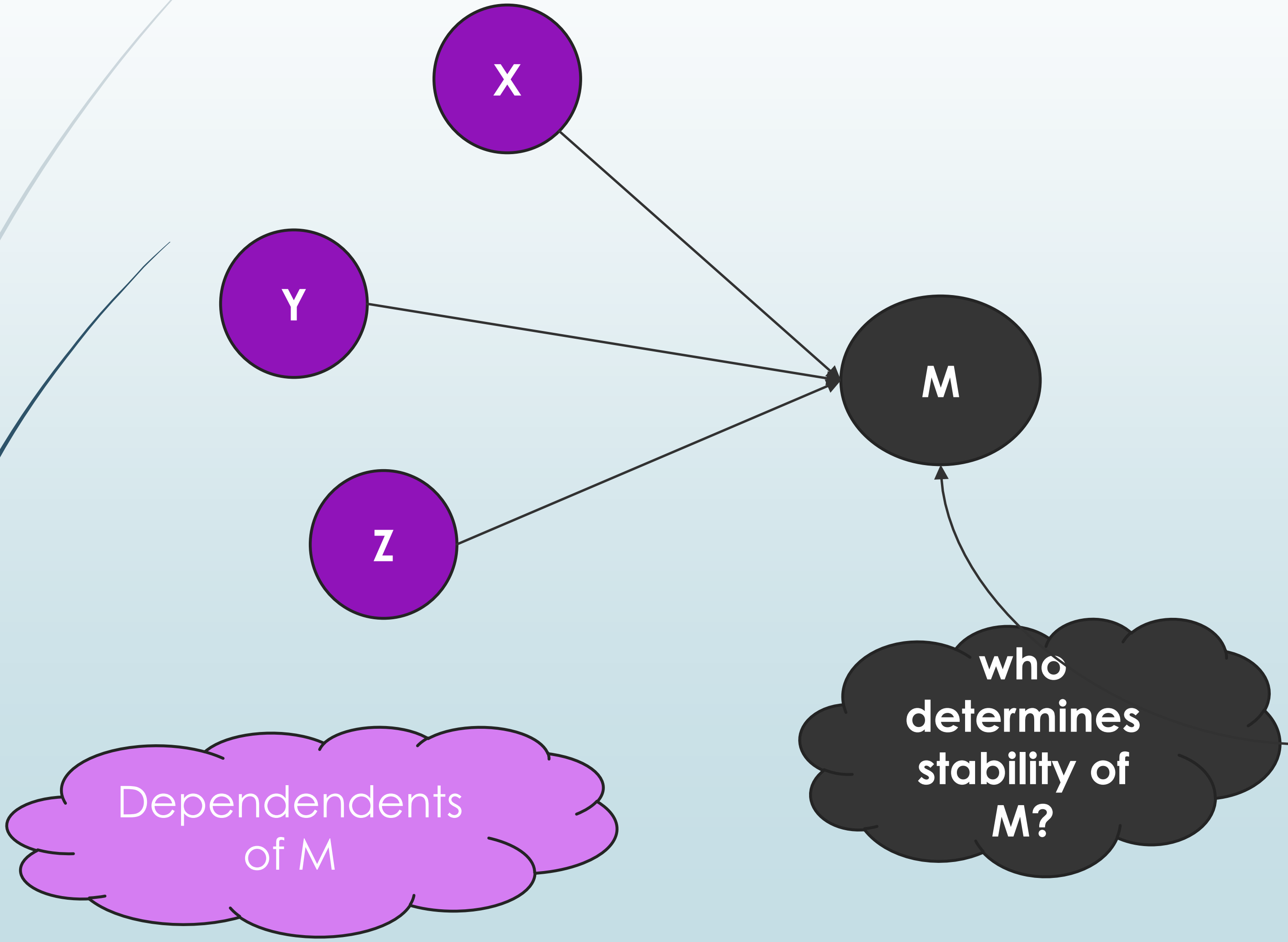


Rule 1: Component Dependencies

- **A component is More Stable if It has Fewer Dependency**
- **Less Reason To Change Because of Others.**



Rule 2: Dependents



- A component is more stable if it has more dependents
- A very great Reason Not to Change.
- If They change all dependents break

Such Components shall not change even if there is a real need of change

STABLE DEPENDENCY PRINCIPLE

- A COMPONENT IS STABLE IF

1. IT HAS FEW DEPENDENTS
   - COMPONENT WILL CHANGE ONLY FOR ITS INTERNAL RESPONSIBILITY
2. IT HAS MORE DEPENDENTS
   - COMPONENT WONT CHANGE EVEN FOR ITS INTERNAL REASONS
   - ITS RESPONSIBLE FOR ITS DEPENDENTS

SHOULD MORE DEPENDENTS MAKE YOU STABLE?

- ITS NOT ALWAYS A GREAT IDEA
- ITS ESSENTIALLY A BACKWARD COMPATIBILTY IDEA
- GOOD OR BAD IT'S A OFTEN FOLLOWED PRACTICE

- JAVA DEPRECATED MANY FEATURES, BUT NEVER REMOVED ANY. WHY?

Summary

- S R P
- O C P
- L S P
- I S P
- D I P

SOLID Design Principles

- D R Y
- S D P
- A D P
- C C P
- C R P