C++

26 March 2018 09:12

When?

- 1983

What was the original name of C++?

- C with Classes --> C With Objects --> new C --> D --> ++C --> C++
  - It was intended to be a next version of C (Not a new language)
  - A promise: Every valid C program will be a valid c++ program.
    - Complete backward compatible.
  - No breaking changes
    - C++ continue to support every (desirable/undesirable) C constructs
    - OOP is optional -> hybrid coding is most common (Multi-paradigm)

C++ originated without a true compiler/interpreter. What the used?

- C++ to C Language Translator
  - Compiler/interpreter needs to understand underlying platform (OS+Hardware)
  - You need a different compiler per platform - that takes time
  - C++ to C Translator was a stopgap solution
    - one translator can work on all platform where C compiler exists
  - Eventually a true compiler shall be created.
  - Many design decision in c++ is based on the fact that it would use a translator to C
    - It was a stopgap fix
    - It introduced a more permanent problem in the language
      - No Runtime scope
      - Name mangling etc

1991-92

C++ -- ---> OAK --> Java

-- simplified by removing undesirable features

-- globals
-- union/struct/typedef
-- macros

-- pointer arithmetic
-- memory deallocation/destructor
-- friend function
-- multiple inheritance
-- operator overloading
-- enum
-- templates

C++ to C Translation

26 March 2018 09:33

```
//C++ code
class Triangle{

    private:
        int s1,s2,s3;
    public:
        void set(int x,int y, int z){
            s1=x;
            s2=y;
            s3=z;
        }
        int perimeter(){
            return s1+s2+s3;
        }
};

void main(){
    Triangle t1,t2;

    t1.set(3,4,5);

    int p= t1.perimeter();

    double x= divide(7,2); //3
    double y= divide(7.0,2.0); //3.5

}


int divide(int x,int y){
    return x/y;
}

double divide(double x,double y){
    return x/y;
}
```

```
//C++ code Translated to C
class struct Triangle{

    private:
        int s1,s2,s3;
    public:
};
void Triangle_set(Triangle *this, int x,int y, int z){
    s1=x;
    s2=y;
    s3=z;
}
int Triangle_perimeter(Triangle* this){
    return s1+s2+s3;
}
};

void main(){
    Triangle t1,t2;

    Triangle_set(&t1,3,4,5); //t1.set(3,4,5);

    int p= Triangle_perimeter(&t1); //t1.perimeter();


    double x= divide_int_int(7,2); //3
    double y= divide_double_double(7.0,2.0); //3.5

}


int divide_int_int(int x,int y){
    return x/y;
}

double divide_double_double(double x,double y){
return x/y;
}
```

Translator would check for any violation of scope and fail in case it finds any.

In case there is no violation, if private and public keywords are removed.

No Runtime Scope

Actual function name Not available to Runtime

C++ Scope Rules

26 March 2018 09:52

```
class Shape{

public:
    virtual double area(){
        return 0;
    }
};

class Line : public Shape{

private:
    double area(){
        return -1;
    }

}

class Circle:public Shape{
    double r;
public:
    double area(){
        reutrn PI*r*r;
    }
};
```

```
void main(){

    Line * line=new Line();

    double x= line->area(); //compilation error: private

    Shape * shape= line;

    double y= shape->area(); // returns -1

}
```

Expected

There is no runtime scope in C++

In shape area is public so compiler cant stop this call. (It doesn't know you may be using Line at runtime.)

Using polymorphism it will call the latest area() function even if it is private (runtime doesn't know private or public)

Harry Potter

26 March 2018 15:01

```
class  MagicWand{

        Core core;
        int length;
        WoodType wood;
        Color color;
        Owner owner;

        public void CastSpell(MagicSpell spell){

        }

}

class MagicSpell{
        String name;
        boolean isLegal;

        public void effect(){

        }

}
```

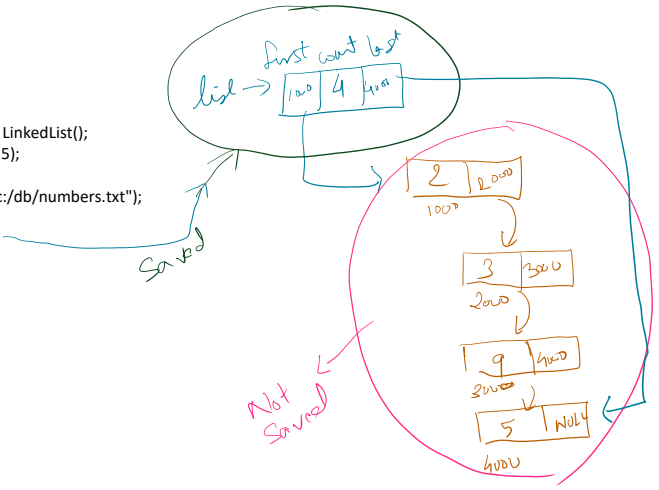
File Write Usecase 1

26 March 2018 10:54

```
void main(){

    Employee emp=new Employee(...);
    File file=new File ("c:/db/emp.db");

    file.Write("%s, %s, %s\n", emp.getName(), emp.getId(), emp.getPassword());

    file.Write(emp);

    emp.Write();

}
```

```
class File{
    public void open(){...}
    public abstract void Write(...);
    public void close(){...}
}

class Employee : File {

    public override void Write(){
        base.open(...);
        base.Write("%s, %s, %s\n", name, id, password);
        base.close();
    }

}
```

```
void main(){

    LinkedList list=new LinkedList();
    list.addMany(2,3,9,5);

    File file=new File("c:/db/numbers.txt");

    file.Write(list);

    list.Write();

}
```



Study of Responsibility - Defect in File Write

26 March 2018 12:18

Statement 1: File doesn't know what is to be written about Employee/List

- Possible solution
  - File may be given the knowledge About Employee and List
  - Should file have this knowledge?

Statement 2: File shouldn't know what is to be written about Employee/List

- File is not supposed to know Employee List
- Possible misinterpretation
  - It must be the Employee object to write its details.

- Possible Solution
  - Pass this responsibility to someone who knows Employee
  - It may be the Employee Object
  - It may be some other object that knows Employee Object

Statement 3: Double / overloaded ~~Single~~ Responsibility

- Solution of Statement1 leads to this problem

Remember:

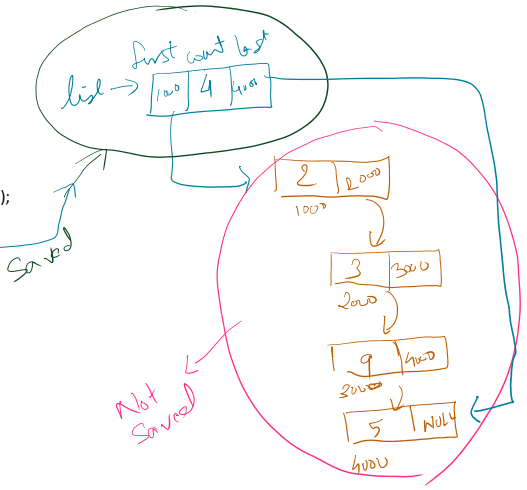
- **Knowledge is ownership**
  - **If you know something you own it.**
- **Ownership (Knowledge) is responsibility**
  - **If you know something you are responsible for it.**

File Write Usecase Refactored.

26 March 2018 10:54

```
void main(){

    Employee emp=new Employee(...);
    File file=new File ("c:/db/emp.db");

    file.Write("%s, %s, %s\n", emp.getName(), emp.getId(), emp.getPassword());

    file.Write(emp);

    emp.WriteTo(file);
    file.close();

}


class Employee  {

    public override void Write(File file){

        file.Write("%s, %s, %s\n", name, id, password);

    }

}


class EmployeeXmlWriter{

    public void Write(Employee emp, File file){
        ...
    }


}
```

```
void main(){

    LinkedList list=new LinkedList();
    list.addMany(2,3,9,5);

    File file=new File("c:/db/numbers.txt");

    file.Write(list);

    list.WriteTo(file);
    file.Close();

}
```



Serialization:

- **Its an object's ability to persist itself**
- **reverse is deserialization.**

Serialization V2:

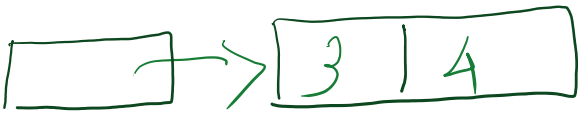
- **It's a process of persisting an object**
- **reverse is deserialization**

SBI Use Case 1

26 March 2018 12:52

Rectangle r1= new Rectangle(3,4);

r1



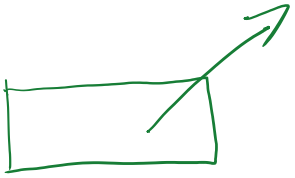
Rectangle r2=new Rectangle(1,1);

r2

Rectangle r3=new Rectangle(3,4);
Rectangle r4= r3;

r3

r4



SBI Use Case 2

26 March 2018 12:52

Rectangle r1= new Rectangle(3,4);

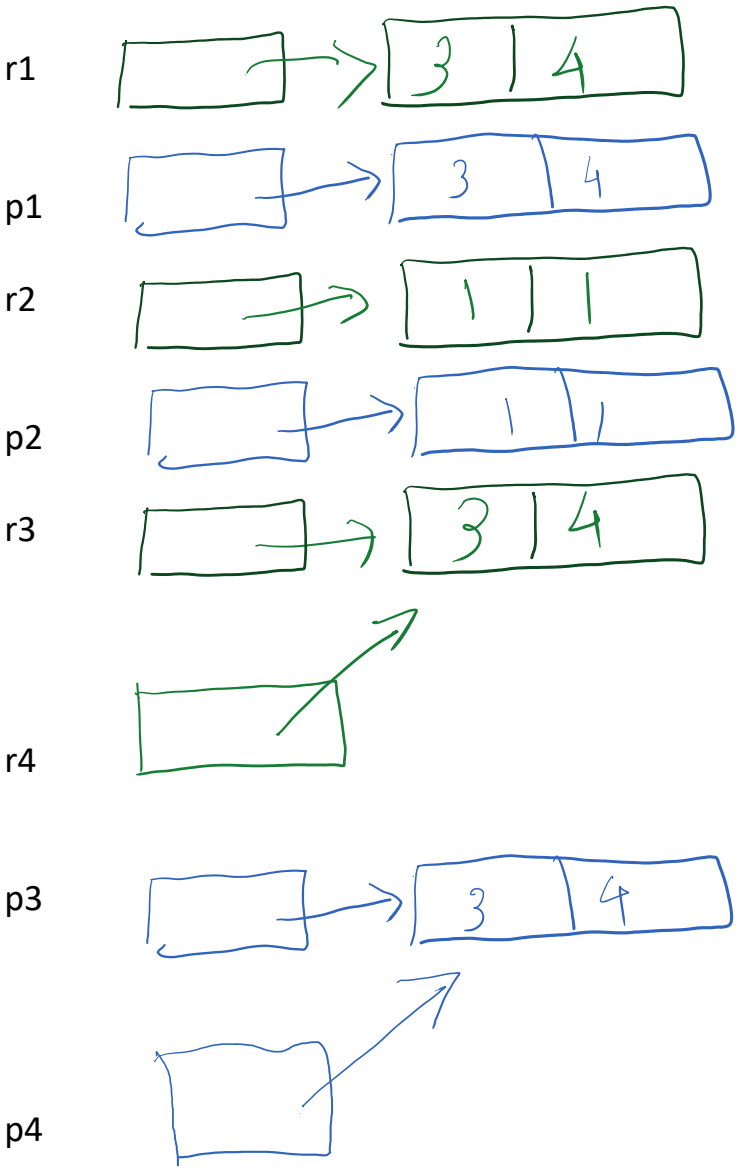
Point p1=new Point(3,4);

Rectangle r2=new Rectangle(1,1);

Point p2=new Point(1,1);

Rectangle r3=new Rectangle(3,4);
Rectangle r4= r3;

Point p3=new Point(1,1);
Point p4=p3;



SBI Use Case 2

26 March 2018 12:52

Rectangle r1= new Rectangle(3,4);

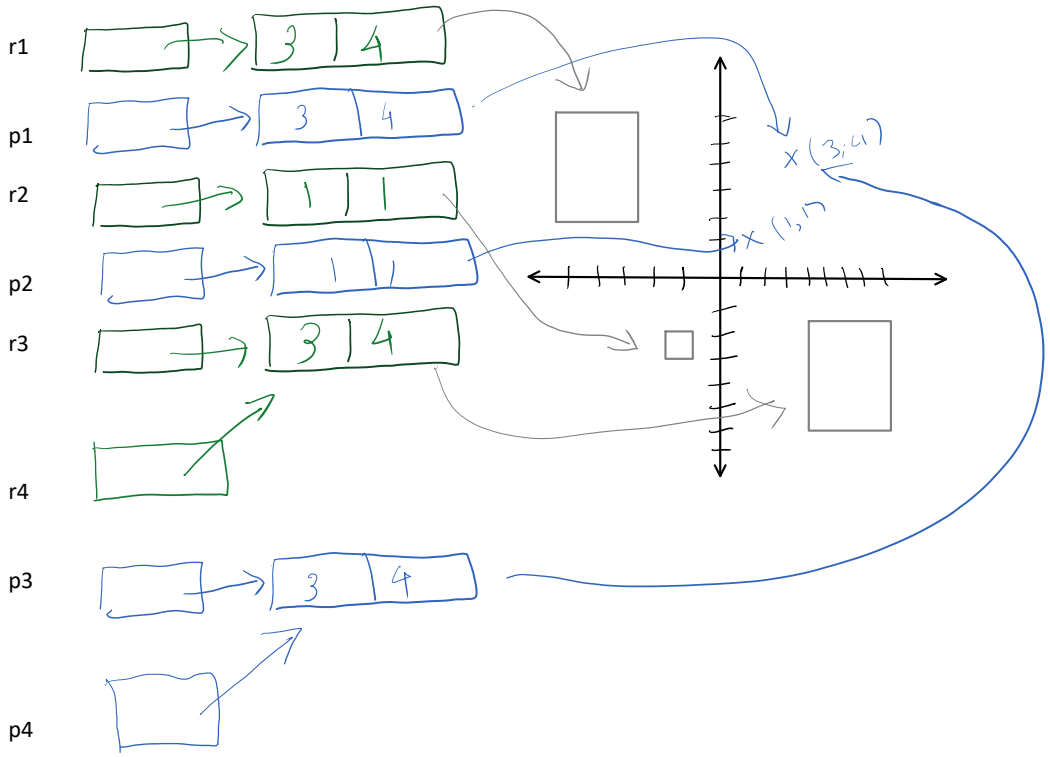
Point p1=new Point(3,4);

Rectangle r2=new Rectangle(1,1);

Point p2=new Point(1,1);

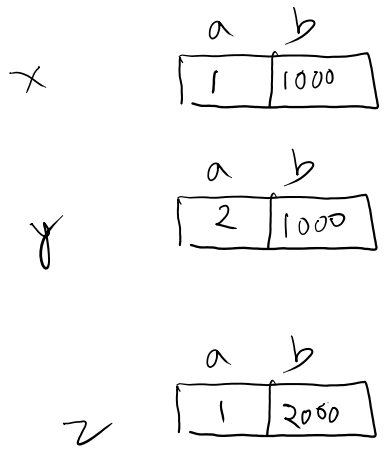
Rectangle r3=new Rectangle(3,4);
Rectangle r4= r3;

Point p3=new Point(1,1);
Point p4=p3;



SBI Use Case 4

26 March 2018 13:01



//15:29 hrs
BankAccount x=new BankAccount(1); //fetch details for db

//15:30 hrs
BankAccount y=new BankAccount(2);

//15:31 hrs
BankAccount z=new BankAccount(1);

28 March 2018 09:12

```
class File
{
    public void Write(Employee emp){

    }

    public void Write(LinkedList list){

    }

}
```

Same

```
class EmployeeWriter
{
    public void Write(Employee emp){

    }
}

class LinkedListWriter
{
    public void Write(LinkedList list){

    }

}
```

Crow-Parrot Relationship

27 March 2018 08:57

```
class Crow{

    public virtual Color getColor(){return Color.black;}

    public void fly(){…}

    public Egg layEgg(){…}

    …

}

class Parrot : Crow{
    public override Color getColor(){return Color.green;}
}
```

```
[Test]
public void ensureCrowsAreBlack(){
    Crow crow=new Parrot();

    Assert.AreEqual( Color.black, crow.getColor());
}
```

Assertion Failed:
expected black found green

```
[Test]
public void ensureParrotsAreNotCrows(){
    Parrot parrot=new Parrot();
    /* c++
    Crow * crow= dynamic_cast<Crow*>(parrot);
    ASSERT(crow!=NULL);
    */
    Assert.False( parrot is Crow);
}
```

Assertion Failed:
(Parrot is Crow)

```
[Test]
public void parrotBabiesAreParrot(){
    Parrot mother=new Parrot();
    Object baby= mother
                .layEgg()
                .hatch();

    Assert.Equals( mother.GetType().Name, baby.GetType().Name)
}
```

Assertion Failed:
expected Parrot found Crow

Parent-Child relationship

27 March 2018 09:20

```
Father rdm=new Father("RDM");

Son vdm=new Son("VDM");
```

*Inheritance in real world is an **object to object** relationship between objects of (generally) same type (class) and **is not same as***

*Inheritance in programming which is a **class to class** relationship and doesn't represent parent-child relationship.*

*Inheritance in Programming is meant to represent **Is A Type of Relationship** and shouldn't be used for any other purpose.*

*Don't Inherit if you don't have **Is A Type of Relationship***
Don't Inherit for

- *Reuse*
- *Has A*
- *Is Similar To / Is Like A*
- *Works together etc.*

```
Person rdm=new Person("RDM");

Person vdm=new Person("VDM");


//inheriting the bank balance
vdm
    .BankAccount
    .Deposit(
            rdm
                .BankAccount
                .Withdraw(amount)
            );

vdm.DNA= rdm.DNA;
```

Computer HardDisk

27 March 2018 09:36

```
class HardDisk{

    public int GetSpeed(){
        /*
        read and write 3 sets of
        data at different position
        and average it */
        return kbps;
    }

}

class Computer : HardDisk{


}


void main(){
    Computer computer=new Computer();

    Print( "speed of my computer is "+computer.GetSpeed());
}
```

```
class Computer {
    HardDisk hdd;
    CPU cpu;
    public double GetSpeed(){ return cpu.GetSpeed(); }
    public int GetHddSpeed(){ return hdd.GetSpeed(); }

}


void main(){
    Computer computer=new Computer();

    Print("speed of my computer is "+computer.GetSpeed());

}
```

```
class ParkerPen
{
    public void UseInHand(Hand hand){
        print("writing");
    }

    public void UseInPocket(Pocket pocket)public void
    UseInPocket(Pocket pocket){
        print("status");
}
```

```
class ParkerPen
{
    public void Use(Hand hand){
        print("writing");
    }

    public void Use(Pocket pocket){
        print("status");
    }
}
```

```
void main()
{
  ParkerPen pen=new ParkerPen();

  Object context= GetHandOrPocket(); //uses random algorithm

     Hand * h= dynamic_cast<Hand*>(context);
     Pocket*p=dynmaic_cast<Pocket*>(context);

     if(h!=NULL)
          pen.Use(h);
     else if(p!=NULL)
          pen.Use(p);

     if( context is Hand)
          pen.Use((Hand)context);
     else
          pen.Use((Pocket) context);

}
```

Laws of OO Design

27 March 2018 10:01

Law #1 (Law of Encapsulation)

- Prefer **Has A(Encapsulation)** over **Is A(Inheritance)**
  - Has is a
    - dynamic
    - scalable
    - runtime relationship
  - **Is a**
    - is static
    - non-scalable
    - design time
    - class to class relationship

- Generally we have a tendency to confuse a "Has A" relationship into "Is a relationship"
  - We say He is a Doctor rather than He Has a Profession of Doctor
  - We say He is a Father rather than He has a relation of being Father to a particular Person
  - He is a Driver rather than He has the skill of a Driver

Law #2 of OO (Law of Inheritance)

- Prefer abstract inheritance over concerete inheritance
- Why?
  - Inheritance is easy
    - Un inheritance in impossible
    - Partial iheritance is impossible
  - You may not want everything from base class
  - **There is no true "Is A type of " relationship between two concrete classes**
    - **Most of those relationship actually turns out to be "Is Like A"**
- **When you inherit an abstract you do it only for relationship and not for reuse.**

Software Design Principle

27 March 2018 15:40

1. Open Close Principle (OCP)

Feature Level (System)

- What?
  - Your design should be **open** for extension (addition, modification, deletion of features)
    - A software code in inherently open for extension (by nature)
    - You can convert a Cow to an Airplane as long as you have the source code.
  - your design should be **closed** for modification.

Source code Level

- Explaination
  - Your system (design) should be pro to change not prone to change
  - System should be ready for New features (changes) without changing existing source code
  - A change should be additivie
    - To add new feature write new code
    - To modify existing feature write new code
    - To delete existing feature write new code
  - **Don't mend it if it is not broken.**

- Why?
  - A change may introduce a bug
    - every change needs a compile-test-distribute-deploye cycle
      - changes are expensive
    - A change may not be acceptable to all stake holders
      - Implementation may be a matter of preference rather than performance.
- How?
  - I Don't know.
- Note:
  - 100% OCP is not feasible.
    - Its not even desirable.

2. Single Responsibility Principle (SRP)

What & Why

- A component (object, method, component) should have a single responsibility
  - A single reason to exist
    - A single reason to change
    - Closed for all but one reason (Theoretically)
    - Practically such elements are most often completely closed.
  - Single Responsibility doesn't mean a single function per object
    - It certainly means not too many methods.
    - It means few strongly related states and behaviors
- **How?**
  1. Use Meaningful names for objects, classes, behaviors etc.
     - **Avoid names joined with And/Or**
       - CreateAndAdd <--- creates an Object and adds to the database
       - InsertOrUpdate
       - IncomeAndServiceTaxCalculator
       - Acceptable when the component is gathering functionality rather than creating it.
         - core responsibility of the CreateAndAdd function should be co-oridnate between Create and Add and **not** define how to create and how to add.
     - **Avoid abstract names for a concrete class.**
       - TaxCalculator <--- what Tax?
     - **Avoid too abstract name even for an abstract class**
       - Calculator
  2. Shouldn't have too many methods
     - excluding getter/setter/property your class shouldn't have more than a agreed number of methods (generally mentioned in design guildelines) (say 10)
  3. Most methods should access most fields most of the time.
     - avoid optional elements
     - avoid mutually exclusive elements

- avoid elements that can be always null in a given scenario.

3. Don't Repeat Yourself (DRY)
   - What?
     - Avoid redundant design
     - Prefer Reuse
   - How?
     1. Encapsulate whatever repeats (Create a SRP component)
     2. Abstract whatever changes together (What changes together must be part of same responsibility)
     3. Use components created in 1+2 as either
        1) abstract base class
        2) encapsulated component (PREFERRED)

4. Interface Segregation Principle (ISP)
   - What?
     - Avoid FAT interfaces
     - FAT interface --> Fat Class --> violates SRP
     - Avoid Interface with
       - Mututally exclusive functions
       - Optional functions
     - An interface should contain only as many methods as every implementor would like to implement.
   - How?
     - Break a FAT interface into smaller interfaces
     - May delete un-wanted redundant methods
     - An interface may extend another interface
     - A class may implement one or more interface.

5. Liskov's Substitution Principle (LSP)
- What?
  - A base component can be substituted (replaced) with a derived component without breaking the client
  - If client can use a base object it can also use derived object without risking any break.
- Why is it important?
  - A change can be introduced as a derived object, without breaking the client or the existing component
  - open for extension (new derived class) closed for modification of (existing component and client)
- How?
  - By default C# and Java like language makes it difficult to break LSP
    - A method public in base class will remain public in derived class and client can call it in the regular faishon.
    - LSP can however be broken
      - By component throws exceptions unexpected/unknown to client
        - client has not planned or handled them.
        - client code need to change to handle the exception.

Q. Is LSP violating Law #2 of OO Design (prefer abstract inheritance) by recommending to derived from an existing base?

A. No. LSP doesn't dictate the design of base class (abstract or concrete), it dictates the design of derived class. LSP recommends:
   a. ***Do not introduce breaking changes in a component.***

Q. Is LSP against throwing exception?

A. *No. LSP is against throwing* ***unexpected exceptions.*** *not documented at the base class level. You can throw all exceptions that are documented at the base level.*
   a. *Its ok for fly() of Ostritech to throw CanNotFlyException() provided it is documented for the Bird class*
   b. *client using Bird will write a try-catch for fly()*

Q. ***Is it possible to be aware of all the future exception at the base (abstract) level?***

A. We can handle it using
   a. A base class should define its own BusinessLayer Exception

b. *Any implementation layer exception should be wrapped and rethrown as BusinessLayerException*

What is Dependency

What?
- knowledge is dependency

Why is Dependnecy a Problem?
- If you know something you depend on it
  - If your dependnecy changes, it may induce change in you
    - May even induce a change in your dependents
  - vioates OCP

Solution.
1. ***Dependnecy Reduction.***
2. ***Dependnecy Inversion***
3. ***Stable Dependnecy***

6. Dependency Inversion Princple (DIP) a.k.a Dependnecy Abstraction.

- What?
  - Instead of a component (client) depending on another concrete component, both should depend on a common abstraction
  - Client should **use/know/depend on** abstract component
  - Component should implement the abstract component.

7. Stable Dependnecy Principle

- What?
  - A Component must always depend on another component that is more stable than itself.
  - A Component is stable if it is less likely to change.

- Why?
  - Any component typically has 2 reasons change:
    1. A change in the core feature (requirement) of the current component itself.
    2. One of its dependency has changed inducing change in it.
  - A stable component is less likely to change
    - Is less likely to induce change in its dependents.

- How to Identify a stable component
  - A Component is stable if
    1. It has fewer dependnecy
       - less reason to change because of others.
       - can still change because of its own core reason.
    2. It has more dependents
       - Fear of breaking dependents is big deterent for this component
       - Doesn't change even if it has its own core reason to change
    3. It is abstract
       - No personal (Core) reason to change
       - No dependnecy or stable dependeny on another abstraction.

8. Common Closure Principle

- Components of a package (assembly/dll/compilation unit) should be closed (and open) for same change.
- They should exist for same purpose
- They may need to change for same purpose.
- When one changes, others too should change.
- A change should
  - Impact Minimum number package (best case only one)

        - Impact Maximum classes within the package
            - The tells us that class within the package are **really realated.**

9. Common Reuse Principle
    - Components of a package should be used together.
    - When you use one of them, you use most of them.
        - Tells us that those components are really related.
10. ADP

ISP - STACK

28 March 2018 09:34

```
interface IStack {

    void Push(Object value)
    Object Pop();
    bool IsEmpty();

    bool IsFull();

    Object Peek();

}
```

A dynamic stack may never be full and doesn't require IsFull()

Peek() is not always required.

```
//Refactored using ISP

interface IBasicStack{
    void Push(Object value);
    Object Pop();
    bool IsEmpty();
}

interface IStack : IBasicStack{
    bool IsFull();
}

interface IPeekable  //doesn't extend IStack or IBasicStack
{
    Object peek();
}
```

```
//implementing classes

class FixedStack : IStack{
    //Push, Pop, IsEmpty, IsFull
}


class DynamicStack : IBasicStack, IPeekable{
    //Push, Pop, IsEmpty, Peek()

}

class SimpleQueue : IQueue, IPeekable{
    //Enqueue, Dqueue, Peek()
}
```

Multiple Interfaces Implemented

Peek() not more exclusive to Stack

LocalBankAccount

28 March 2018 11:46

```
//document: throws SQLException
class LocalBankAccount {

    public void Withdraw(double amount){


        //make a SQLConnection
        if(…)
            throw new SQLException();

    }

}
```

```
//client
class ATM
{
    void doWithdraw(){

        try{
        account.Withdraw();
        dispenseCash();

        }
        catch(SQLException ex){
            //print error
            //don't dispense cash
        }

    }
}
```

NetBankAccount

28 March 2018 11:46

```
//document: throws SQLException
class LocalBankAccount {

    public void Withdraw(double amount){


        //make a SQLConnection
        if(...)
            throw new SQLException();

    }

}

class NetBankAccount : LocalBankAccount(){

    public override void Withdraw(double amount)
    {
        //connect to internet
        if(...)
            throw new NoNetworkException();
        //call base.withdraw
        base.withdraw();

    }


}
```

```
//client
class ATM
{
    void doWithdraw(){

        try{
        account.Withdraw();
        dispenseCash();

        }
        catch(SQLException ex){
            //print error
            //don't dispense cash
        }

    }
}
```

NetBankAccount2

28 March 2018 11:46

SqlException or NetworkException in Banking
is not related to Banking (they are related to implementation)

```
//document: throws BankingException
abstract class BankAccount
{
    abstract void Withdraw();
}


//document: throws SQLException
class LocalBankAccount :BankAccount{

    public void Withdraw(double amount){

        try{
        //make a SQLConnection
        //may throw SQLException ex
        }
        catch(SQLException ex){
            throw new BankingException (ex);
        }
    }

}

class NetBankAccount : BankAccount{

    public override void Withdraw(double amount)
    {
        try{
        //make a SQLConnection
        //may throw NetworkException ex
        }
        catch(NetworkException ex){
            throw new BankingException (ex);
        }

    }


}
```

```
//client
class ATM
{
    void doWithdraw(){

        try{
        account.Withdraw();
        dispenseCash();

        }
        catch(BankingException ex){
            //print error
            //don't dispense cash
        }

    }
}
```

What is Dependency

28 March 2018 13:04

```
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();
    }

}
```

No try catch allowed

ComponentX depends on its **base class** ComponentA.
This is a **two point dependnecy** (constructor, DoJob())

what happens when:

Q. base class constructor takes a new parameter?
A. ComponentX constructor need to take a new parameter and pass to constructor
  a. even the client of ComponentX need to change

Q. base class constructor throws some exception?
A. **Nothing can be done. You need to pass it to your client**
  a. **Client of ComponentX will change**
Q. base class DoJob() takes a new parameter?
A. ComponentX JobA() need to take a new parameter and pass to DoJob
  a. even the client of ComponentX need to change
Q. base class DoJob() throws some exception?
A. ComponentX need to introduce try-catch

ComponentX also depends on assoicated component ComponentC, However it is better than the other two because it is a **one point dependency**

ComponentX doesn't depend on constructor of ComponentC and is not affected by any change in the constructor ComponentC

ComponentX also depends on its **encapsulated component** ComponentB.
This is a **two point dependnecy** (constructor, Execute())

what happens when:

Q. ComponentB constructor takes a new parameter?
A. ComponentX constructor need to take a new parameter and pass to constructor
  a. even the client of ComponentX need to change

Q. ComponentB constructor throws some exception?
R. ComponentX constructor need write a try-catch block
  - **Client of ComponentX need not change**

A.
Q. ComponentB class Execute() takes a new parameter?
A. ComponentX JobB() need to take a new parameter and pass to DoJob
  a. even the client of ComponentX need to change
Q. base class Execute() throws some exception?
A. ComponentX need to introduce try-catch

Takeaway:

1. **Dependencies are Bad**
2. **Change in dependency may induce change in the dependent component or their dependents.**
3. **More points of dependency more likely to change**
4. **We should try to reduce amount of dependency**

A composition dependency has scope of improvement

Dependency Reduction

28 March 2018 13:04

```
class  ComponentX  :  ComponentA
{
    ComponentB bComp;

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

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

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

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

}
```

ComponentX no longer depends on the constructor of encapsulated component B and is now uneffected by any change in the same.

We can live without knowing the constructor of encapsulated component.

Unfortunately there is no way to avoid dependency on the base class constructor. a dervied class constructor always calls (not optional) the base class constructor.

Q. How can I ensure that ComponentX will not change because of ComponentA constructor

A. Only if ComponentX can ensure its constructor never changes.

Q. Can I ensure that constructor of base class never takes a new parameter?

A. Only if base class has nothing initialize

Q. Can I ensure that constructor of base class never throws a new exception?

R. Only if base class constructor does nothing

Only if base class is abstract

Dependency Inversion

28 March 2018 13:04

```
class  ComponentX  :  ~~ComponentA~~ AbstractA
{
    ~~ComponentB bComp;~~
    AbstractB bComp;
                                    AbstractB bComp
    public ComponentX( int a, ~~ComponentB bComp~~) : base(a)
    {

        this.bComp=bComp;
    }

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

    public void JobB(){
        bComp.Execute();
    }
                        AbstractC
    public void JobC(~~ComponentC~~ cComp){
        cComp.Work();
    }

}
```

Why abstract dependency is good?

- Abstraction are less likely change.
- They don't have **logic** so no logical reason change

Q. Can abstraction gurantee parameters wont change?
A. Yes

Q. Can abstraction gurantee no exception will be thrown by the implementation?
A. No. But
   a. LSP recommends throwing known exceptions only.
   b. abstraction also includes documented exception which client must already been handling.

Computer-HardDisk Design (Inheritance)

28 March 2018 13:42

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

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

HardDisk failure is same as whole Computer failure because Computer is a HardDisk

```
//Bad Relationship: Computer is not a HardDisk
// Class to Class relationship : Class Computer knows Class HardDisk
// Strong Dependency
class Computer  : HardDisk
{
    public Computer(int capacity):base(capacity){
    }

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

void main(){
    Computer c1=new Computer(512); //512 gb

    c1.Save(); //works

    //what happens if HardDisk fails???

}
```

Computer-HardDisk Design (Composition)

28 March 2018 13:42

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

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

HardDisk failure is same as whole Computer failure because Computer is a HardDisk

```
//Right relationship: computer has HardDisk
// Class to Class relationship : Class Computer knows Class HardDisk
// Strong Dependency
class Computer ~~: HardDisk~~
{
    HardDisk hdd;
    public Computer(int capacity)~~:base(capacity)~~{

        hdd=new HardDisk(capacity);
    }

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

void main(){
    Computer c1=new Computer(512); //512 gb

    c1.Save(); //works

    //what happens if HardDisk fails???

}
```

Computer-HardDisk Design (Dependency Injection)

28 March 2018 13:42

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

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

```
//Right relationship: computer has HardDisk
// Class to Class relationship : Class Computer knows Class HardDisk
//Replacable Dependency
class Computer
{
    HardDisk hdd;
    public Computer(int capacity HardDisk hdd){
        this.hdd=hdd;
        hdd=new HardDisk(capacity);
    }

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



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

void main(){
    Computer c1=new Computer(new HardDisk(512)); //512 gb

    c1.Save(); //works

    //what happens if HardDisk fails???
    //No problem
    c1.SetHardDisk(new HardDisk(1024)); //get a better hard disk

}
```

constructor based

Dependency Injection:
Supplying dependnecy (need) to a component

setter based/property based

If a HardDisk fails it can be replaced without needing to replace the whole computer.

However a computer's HardDisk can be replaced by another HardDisk only not with SSD or MicroSD?

Computer-HardDisk Design (Dependency Inversion)

28 March 2018 13:42

```
interface IStorage{
    void Write(…);
    byte [] Read(…);
}

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

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

class SSD : IStorage{}

class Dropbox: Istorage{}
```

object c1 knows object HardDisk

object c2 knows object SSD

object c1 can switch to object HardDisk

no information present at class level.

This is Object Oriented Design

```
//Right relationship: computer has HardDisk
// No Class to Class relationship : Class Computer doesn't knows Class HardDisk
//Replacable Dependency
class Computer
{
    ~~HardDisk hdd;~~
    IStorage storage;
    public Computer(~~HardDisk hdd~~IStorage storge){
        ~~this.hdd=hdd;~~
        this.storage=storage
    }

    ~~public void SetHardDisk(HardDisk hdd){~~
        ~~this.hdd=hdd;~~
    ~~}~~

    public void SetStorage(IStorage storage){
        this.storage=storage;
    }


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

void main(){
    Computer c1=new Computer(new HardDisk(512)); //512 gb

    Computer c2=new Computer(new SSD(256));

    c1.Save(); //works

    //what happens if HardDisk fails???
    //No problem
    c1.SetStorage(new SSD(512)); //swaps a HardDisk with SSD

}
```

constructor based

Dependency Injection:
Supplying dependnecy (need) to a component

setter based/property based

Computer-HardDisk Design (Default Dependency)

28 March 2018 13:42

```
interface IStorage{
    void Write(…);
    byte [] Read(…);
}

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

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

class SSD : IStorage{}

class Dropbox: Istorage{}
```

Default/Fail Safe/Opinionated Dependency

Opinionated dependency is a recommendation of what you should be using to get a decent design.

Opinions are optional (often very useful) and you can always ignore by going to explicit decision.

```
//Right relationship: computer has HardDisk
// No Class to Class relationship : Class Computer doesn't knows Class HardDisk
//Replacable Dependency
class Computer
{

    IStorage storage;
    public Computer(IStorage storge){

        this.storage=storage
    }

    public Computer(){
        this.storage=new HardDisk(512);
    }

    public void SetStorage(IStorage storage){
        this.storage=storage;
    }

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

void main(){
    Computer c1=new Computer(new HardDisk(512)); //512 gb

    Computer c2=new Computer(new SSD(256));

    Computer c3=new Computer(); //you get a 512 gb HardDisk

    c3.SaveExternal(new PenDrive());
}
```

constructor based

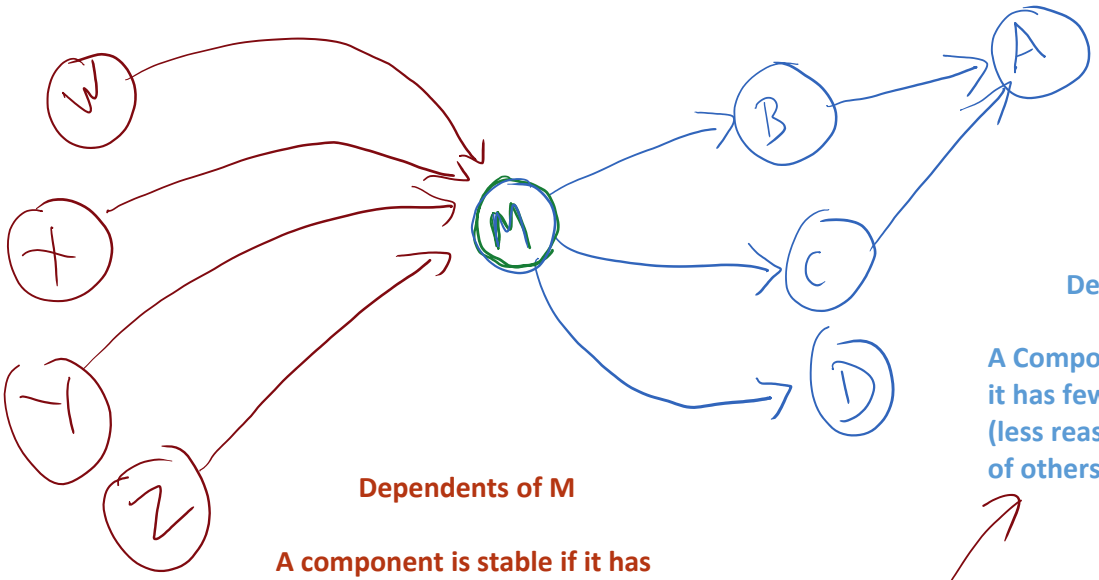
Dependency Injection:
Supplying dependnecy (need) to a component

setter based/property based

method based
dependnecy injection

Stable Dependnecy

28 March 2018 14:35



Dependencies of M

A Component is more stable if it has few dependency (less reason to change because of others)

Dependents of M

A component is stable if it has more dependents.

- **A very good reason for not changing.**

greater reason than

Design Principles Summary

28 March 2018 14:46

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

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

SOLID Design Principle

Constructor Concerns

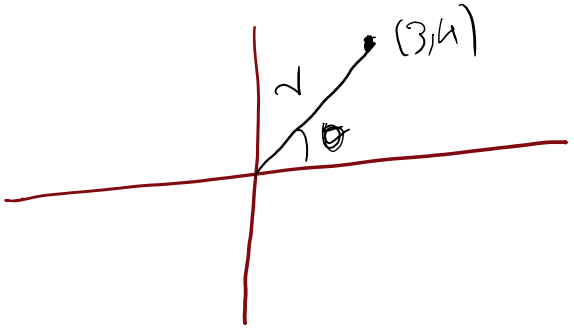
28 March 2018 15:08

1. A constructor is always used with **new** keyword and there is no **old/existing** keyword
   - there is a difference between needing an object and needing a new object
   - constructor fails to differentiate
   - constructor fails to reuse existing object

   var p1= new Point(3,4) ; //it's a new object
   var p2= new Point(1,1); //it's again a new object

   var p3=new Point(3,4); //Is it a new object??

2. Constructor has a completely meaningless name.
   - constructor is the **creator** of the object **not a part** of the object
   - constructor name is same as that of class
     - Class represents Object
     - A name valid for object may not be valid for the **creator** of the object
   - can't choose.



3. Constructors are non-polymorphic
   - A polymorphic code is resolved dynamically
   - I may want to dynamically decide which object to create based on the context.
   - Constructors always create the same object.
   - No virtual or abstract constructor

4. Constructor always creates.
   - The cant return existing objects

5. Constructors are anti-dependnecy inversion
    - constructor connects you to concrete implemenation (Same name)
    - Dependency inversion means not connected to implementation.

6. **Constructors are not optional.**
    - They cant be deleted.
    - ***They can be hidden***

UserManagementUI

28 March 2018 15:10

```
class UserUI
{
    public void Register(){
        //this is a new user
        User user=new User(name,email,password,
                        hintQuestion,hintAnswer);
                    //insert into users…
    }

    public void Login(){
        //is this also a new user???
        User user=new User(email,password);
                    //select * from users where …
    }

}
```

```
class UserUI
{
    public void Register(){

        User user=UserManager
                    .CreateUser(name,email,password,
                        hintQuestion,hintAnswer);

    }

    public void Login(){

        User user=UserManager
                    .FetchUser(email,password);

    }

}
```

Static Vs NonStatic

28 March 2018 15:15

Static	Non Static
1. same memory for all instance	Different memory for each object
2. belongs to the class (Class Level)	Belongs to Objects (Instance level)
3. Life of Application	Life ends with object
4. no **this reference**	**this reference**
5. Accessible using class reference	**Accessible using object reference**
6. No object needed	**Object Needed.**

UserManagementUI - Extended Requirement

28 March 2018 15:10

```
class UserUI
{
    public void Register(){

                    InvactiveUser
        User user=new ~~User~~(name,email,password,
                        hintQuestion,hintAnswer);
                    //insert into users…
    }

    public void Login(){
        ~~User user=new User(email,password);~~
        //which user to create depends on userType
        column in database

        SqlConnection con=…
        SqlCommand cmd="select * from users";
        SqlDataReader reader=…

        if(reader.Read()){
            switch(reader["userType")
            {
                case "ADMIN": user=new Admin(…);
                case "CUSTOMER": user=new
                Customer(…);
                …
            }
        }
    }

}
```

abstract

```
class UserUI
{
    public void Register(){

        User user=UserManager
                .CreateUser(name,email,password,
                    hintQuestion,hintAnswer);

    }

    public void Login(){

        User user=UserManager
                .FetchUser(email,password);

    }

}
```

CreateUser function (Business Layer)
will internally change. But call of this function from
UI layer will remain unaffected.

Originally CreateUser was returning a object of concrete class User

Now User is an abstract class. CreateUser is returning an object of
concerete class InactiveUser

but client need not change InactiveUser is a type of User

We have written a
Data access code in
presentation tier

Overlap of
responsibility
(presentation tier
shouldn't talk to
data tier directly)

A very similar logic will be written in
UserManager.FetchUser function. But FetchUser is
a business layer object and they are expected to
intreract with data tier.

We have a change (big change) in business layer.
That change should affect business layer and not
presentation layer.

BankAccount

29 March 2018 09:46

```
class BankAccount
{
    String name;
    int accountNumber;
    double balance;
    String password;
    static double rate;

    public void Deposit(double amount){

    }

    public void CreditInterest(){
        balance+=(balance*rate)/1200;
    }
    public static void SetRate(double r){rate=r;}
}



void main()
{
    RBI rbi= RBI. GetInstance();
    //RBI rbi=  Govt.GetRBI();                //new RBI()
    Bank icici=   rbi.GetBank("ICICI");     //new Bank("ICICI");
    BankAccount vivek= icici.OpenAccount(…); //factory method
}
```

```
class BankAccount
{
    String name;
    int accountNumber;
    double balance;
    String password;
    public void Deposit(double amount){

    }
    public void CreditInterest(){
        balance+=(balance*rate)/1200;
    }
}

class Bank
{
    double rate;
    public void SetRate(double r){rate=r;}
    List<BankAccount> accounts;

    public BankAccount OpenAccount(){
        BankAccount newAccount=new BankAccount(…);
        accounts.Add(newAccount);
        return newAccount;
    }

    public void Deposit(int accountNumber, int amount){
        GetAccountById(accountNumber)
                .Deposit(amount);
    }

}
```

UI API

29 March 2018 11:17

```
void POC(){

    SteelForm     form     = new SteelForm();
    SteelButton   button   = new SteelButton();
    SteelTextBox textBox = new SteelTextBox();

    form.Add(button);
    form.Add(textBox);

    form.Draw();
}
```

switching from Steel to Rubber needs a change at 6 places in POC and 1000 places in actual code.

We need to change all reference to **implementation classes.**

Real World

May have 500 objects of 20 different types
like checkbox, progress bar etc

UI API (DI)

29 March 2018 11:17

In a Dependency Inverted Design

```
void POC(){

    AbstractForm     form     = new SteelForm();
    AbstractButton   button   = new SteelButton();
    AbstractTextBox textBox = new SteelTextBox();

    form.Add(button);
    form.Add(textBox);

    form.Draw();
}
```

switching from Steel to Rubber needs a change at 3 places in POC and 500 places in actual code.

changes reduced to half

We need to change all reference to **constructors.**

Real World

May have 500 objects of 20 different types
like checkbox, progress bar etc

UI API (Factory Method)

29 March 2018 11:17

In a Dependency Inverted Design

```
void POC(){

    AbstractFormFactory     ff     = new SteelFormFactory();
    AbstractButtonFactory  bf   = new SteelButtonFactory();
    AbstractTextBoxFactory tf = new SteelTextBoxFactory();


    AbstractForm     form     = ff. CreateForm();  new SteelForm();
    AbstractButton   button   = bf. CreateButton();  new SteelButton();
    AbstractTextBox textBox = tf.CreateTextBox(); new SteelTextBox();

    form.Add(button);
    form.Add(textBox);

    form.Draw();
}
```

switching from Steel to Rubber still needs a change at 3 places in POC

but only 20 places in actual code. One Button Factory can create all Buttons

What still changes is - **constructors.**

What is need is not to create a single object but a family of related objects which work together. (used together and replaced together)

Real World

May have 500 objects of 20 different types
like checkbox, progress bar etc

UI API (Abstract Factory) a.k.a Toolkit Pattern

29 March 2018 11:17

In a Dependency Inverted Design

```
void POC(){

    AbstractFormFactory     ff     = new SteelFormFactory();
    AbstractButtonFactory  bf   = new SteelButtonFactory();
    AbstractTextBoxFactory tf = new SteelTextBoxFactory();

    AbstractUIFactory     ui     = new SteelUIFactory();

    AbstractForm     form     = ui. CreateForm();
    AbstractButton   button   = ui. CreateButton();
    AbstractTextBox textBox = ui.CreateTextBox();

    form.Add(button);
    form.Add(textBox);

    form.Draw();
}
```

switching from Steel to Rubber still needs a change at 1 places in POC

And only 1 places in actual code. One Button Factory can create all Buttons

What still changes is - **constructors.**

Idea is to create a list of associated objects rather than just one object.

Real World

May have 500 objects of 20 different types
like checkbox, progress bar etc

UI API (Assignment)

29 March 2018 11:17

```
void main(){

    AbstractUIFactory     ui     = new SteelUIFactory();

    AbstractForm     form     = ui. CreateForm();
    AbstractButton   button   = ui. CreateButton();
    AbstractTextBox textBox = ui.CreateTextBox();

    form.Add(button);
    form.Add(textBox);

    form.Draw();
}
```

expected output:

```
SteelForm Drawn
    SteelButton drawn
    SteelTextBox drawn
```

Assignment Requirement

1. You may create Gliffy Diagram or Program or Notepad psudo code (No paper sketch/text)
2. Define at least three different objects (button,textbox form) --> plan for 20.
3. Create at least 2 sets (steel, rubber etc)
4. define add method to add
   a. Button
   b. text box
   c. … real world will have 20
5. when you draw the form
   a. form also draws textbox and button
6. Create proper packaging (assembly and dll)
7. Show classes, abstractions and methods

C++ Factory Provider

29 March 2018 14:07

```
//steel.dll

class SteelButton : public UIComponent{};

class SteelTextBox : public UIComponent{};
…
class SteelFactory :public UIFactory{};

UIFactory * createFactory(){ return new SteelFactory; }

//Rubber.dll

class RubberButton : public UIComponent{};

class RubberTextBox : public UIComponent{};
…
class RubberFactory :public UIFactory{};

UIFactory * createFactory(){ return new RubberFactory; }


//core.dll

typdef UIFactory * (*CreatorFunction)();

UIFactory * GetFactory( char * factoryDllPath){

    HINSTANCE lib= LoadLibrary(factoryDllPath);
    if(!lib)
      return new DefaultFactory();

    CreatorFunction  creator=(CreatorFunction) GetProcAddress( lib, "createFactory");

    return creator();

}
```

Singleton

29 March 2018 14:17

- What?
  - A particular class should have a single instance in the context
- Why?
  1. Object is conceptually singleton
     - A context can't have two object of this type
     - E.g.
       - Chairman object in an Organization
       - Sun in a SolarSystem
       - RBI in the country
     - There are very few **real Singleton** in a domain
  2. Convinience Singleton
     - Object is not conceptually Singleton.
     - But making multiple copies may lead to bad design.
     - Object created Singleton for optimization purpose
     - Generally such objects shares a lot of data resource
     - Often a mediator between various components of your system.
  3. Zero Weight Singleton
     - Object contains no state
     - It need not be singleton
     - There is no difference between two object of a class
       - why to create two objects???

Singleton V1

29 March 2018 14:15

```
class Singleton
{
    private Singleton(){}

    private static Singleton instance=null;

    public static Singleton GetInstance(){

        if(instance==null)
        {
            instance=new Singleton();
        }
        return instance;
    }

}
```

Not Thread Safe

If mutiple Threads try to access GetInstance() when the same is not yet created.

They each find instance == null and they each create it

finally only one will stored.

But singleton is broken

Severity: Severe (if object holds resource)

Frequency: Very Rare.

Singleton V2

29 March 2018 14:15

```
class Singleton
{
    private Singleton(){}

    private static Singleton instance=null;
    private static Object _lock=new Object();

    public static Singleton GetInstance(){

        lock(_lock)
        {
            if(instance==null)
            {
                instance=new Singleton();
            }
        }
        return instance;
    }

}
```

Thread Safe

mutliple thread may reach lock together only one gets inside at a time. ensure signleton is not broek

Problem: Performance

Thread locks are generally time and resource hungry.

Severity: Severe
Frequency: Very High

Singleton V3 (double checked locked)

29 March 2018 14:15

```
class Singleton
{
    private Singleton(){}

    private static Singleton instance=null;
    private static Object _lock=new Object();

    public static Singleton GetInstance(){

        if(instance==null)
        {
            lock(_lock)
            {
                if(instance==null)
                {
                    instance=new Singleton();
                }
            }
        }
        return instance;
    }

}
```

Thread Safe + High Performance

outer if statement avoids un-necessary lock if the object has already been created.

if object is not created multiple threads may enter outer if

lock will alow only one to actually enter at a time.

first thread will create the object

inner if will avoid additional object creation.

Double Checked Lock

is now considered as a design pattern in itself (NOT GOF)

It is a concurrency pattern to avoid un-necessary lock.

Singleton V4

29 March 2018 14:15

```
class Singleton
{
    private Singleton(){}

    private static Singleton instance=new Singleton();


    public static Singleton GetInstance(){

        return instance;
    }

}
```

A heavy weight object should be Lazy Initialized (As Late as possible - preferrably just before there use)

Thread Safe + High Performance

static initialization is ensured Thread Safe at language level.

Problem

Its eagerly initialzied. Object is created as soon as class is loaded in the memory and you may not be needing it immediately.

It may allocate a lot resource which may or may not be used.

Double Checked Lock

is now considered as a design pattern in itself (NOT GOF)

It is a concurrency pattern to avoid un-necessary lock.

Singleton V4.1

29 March 2018 14:15

```
class Singleton
{
    private Singleton(){}

    public static readonly Singleton Instance=new Singleton();



}
```

A heavy weight object should be Lazy Initialized (As Late as possible - preferrably just before there use)

Double Checked Lock

is now considered as a design pattern in itself (NOT GOF)

It is a concurrency pattern to avoid un-necessary lock.

ThreadPool

29 March 2018 14:43

```
public class ThreadPool
{
    List<Thread> threads;
    List<Task> tasks;

    void Init()
    {
        int max_threads=GetMaxThreads(); //say 10
        for(in i=0;i<max_threads;i++)
        {
            Thread t=new Thread( PoolTask);
            t.Start();
            threads.add(t);
        }
    }

    void PoolTask()
    {
        while(true)
        {
           if( tasks.Count>0){
                Task task= tasks.remove(0); //get and remove first task
                task.Execute();
            }else
                Thread.sleep(500);
        }
    }

    public void AddTask(Task task) {tasks.Add(task); }

    public int GetPendingTaskCount(){…}
    public int GetCompletedTaskCount(){…}



}



 void main()
 {
     ThreadPool pool1=new ThreadPool(); //10 threads
```

10 threads busy trying to process 50000 jobs

```
{
    ThreadPool pool1=new ThreadPool(); //10 threads

    ThreadPool pool2=new ThreadPool();

    foreach(File file in FileSystem.GetAllFiles()) //50000 jobs
        pool1.Add( new FileSearchTask(file));

}
```

10 threads busy trying to process 50000 jobs

10 threads sitting idle doing nothing

ThreadPool (Singleton)

29 March 2018 14:43

```
public class ThreadPool : ITaskExecutor
{
    List<Thread> threads;
    List<Task> tasks;

    void Init()
    {
        int max_threads=GetMaxThreads(); //say 10
        for(in i=0;i<max_threads;i++)
        {
            Thread t=new Thread( PoolTask);
            t.Start();
            threads.add(t);
        }
    }

    void PoolTask()
    {
        while(true)
        {
           if( tasks.Count>0){
                Task task= tasks.remove(0); //get and remove first task
                task.Execute();
            }else
                Thread.sleep(500);
        }
    }

    public void AddTask(Task task) {tasks.Add(task); }

    public int GetPendingTaskCount(){…}
    public int GetCompletedTaskCount(){…}

    private ThreadPool(){ Init(); }
    private static ThreadPool _instance=null;

    public static ThreadPool GetInstance() { /* version 3 impl */}

}
```

```
public interface ITaskExecutor
{
    void Add(Task task);
}

public SequentialTaskExecutor : ITaskExecutor
{
    //single thread task executor
}

public RealTimeTaskExecutor : ITaskExecutor
{
    //each task executes on new thread
}
```

```
void main()
{
    ThreadPool pool1=ThreadPool.GetInstance();

    ThreadPool pool2=ThreadPool.GetInstance();

    foreach(File file in FileSystem.GetAllFiles()) //50000 jobs
        ThreadPool.GetInstance().Add( new FileSearchTask(file));

}
```

A Single ThreadPool

ThreadPool (Static)

29 March 2018 14:43

```
public class ThreadPool
{
    static List<Thread> threads;
    static List<Task> tasks;

    static void Init()
    {
        int max_threads=GetMaxThreads(); //say 10
        for(in i=0;i<max_threads;i++)
        {
            Thread t=new Thread( PoolTask);
            t.Start();
            threads.add(t);
        }
    }

    static void PoolTask()
    {
        while(true)
        {
           if( tasks.Count>0){
                Task task= tasks.remove(0); //get and remove first task
                task.Execute();
            }else
                Thread.sleep(500);
        }
    }

    public static void AddTask(Task task) {tasks.Add(task); }

    public static int GetPendingTaskCount(){…}
    public static int GetCompletedTaskCount(){…}

}
```

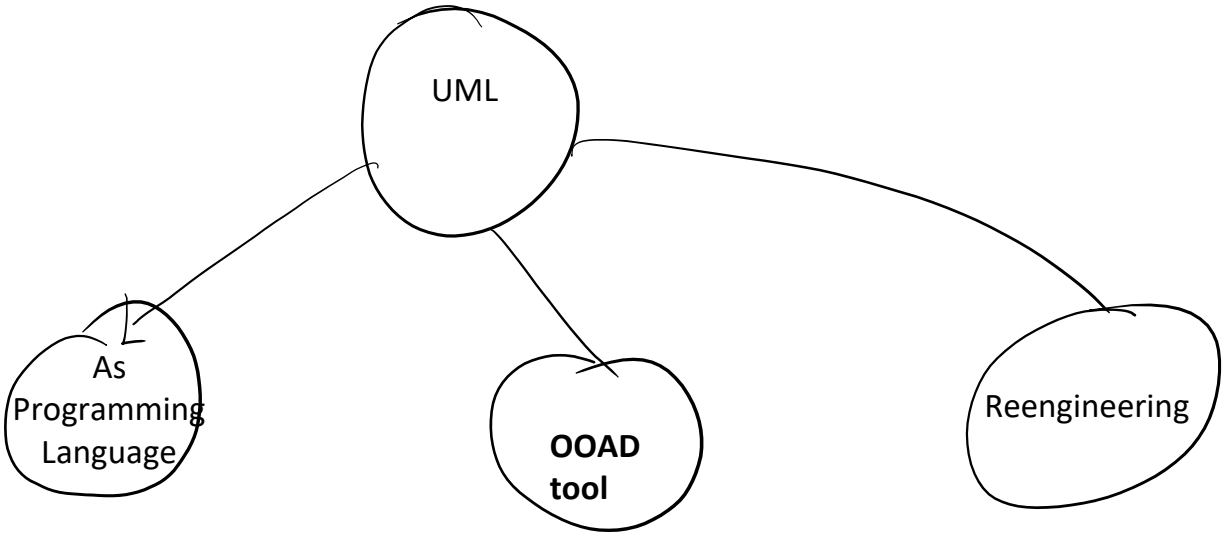
```
void main()
{

    foreach(File file in FileSystem.GetAllFiles()) //50000 jobs
        ThreadPool.Add( new FileSearchTask(file));

}
```

No Instance Required.

UML (Unified Model Language)

29 March 2018 10:15

represents a system diagramatically.
(it is set of 10 different type of diagrams)

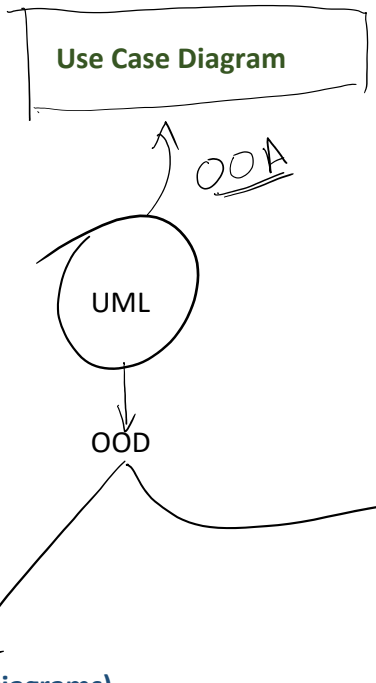


- create UML as replacement of coding
- No complete implemenation.
- partial implementations exist as Micorsoft WF
  - converts 3 diagrams to code
- Tools exist to convert uml diagram to basic source code.

OOAD --> UML
UML --> OOAD

- Refactoring
- Understanding the code base
- tools exists to convert source code to UML diagrams.

29 March 2018 10:21



Structural (Static Diagrams)

- **Class Diagram**
- **Package Diagram**
- **Component Diagram**
- **Deployment Diagram**

Behavioral (Dynamic Diagrams)

- **Object Diagram**
- **Sequence Daigram**
- **Activitiy Diagram**
- **State Machine Diagram**
- **Collaboration Diagram**

Structural Patterns

29 March 2018 15:16

Common Traits

- Wrapper Objects
  - Wraps some target (existing) object to gather the functionality
  - Wrapper and Target are cohesive
  - Encapsulate (Wrap) to Reuse
  - Lightweight/Inexpensive.

Façade

- Wraps the target to simpliy the interaction
- Reduces dependency between client and actual target
  - Reduces reason to change
- Optional
  - You may still use the actual target

Adapter

- Wraps to translate the interface
- Doesn't Add new Behavior
- Translates Interface
- Dynamically adds new interface to exisiting object
- Often late design solution
  - Allows two incompatible systems to work together
- Great Refactoring Design

Next Pattern

- Wraps the target with the same interface
- Doesn't add new behavior (same interface)
- Doesn't translate the interface
- Often Transparent
  - Client may not know there is a wrapper (layer) between it and the actual target

Messanger API

29 March 2018 15:16

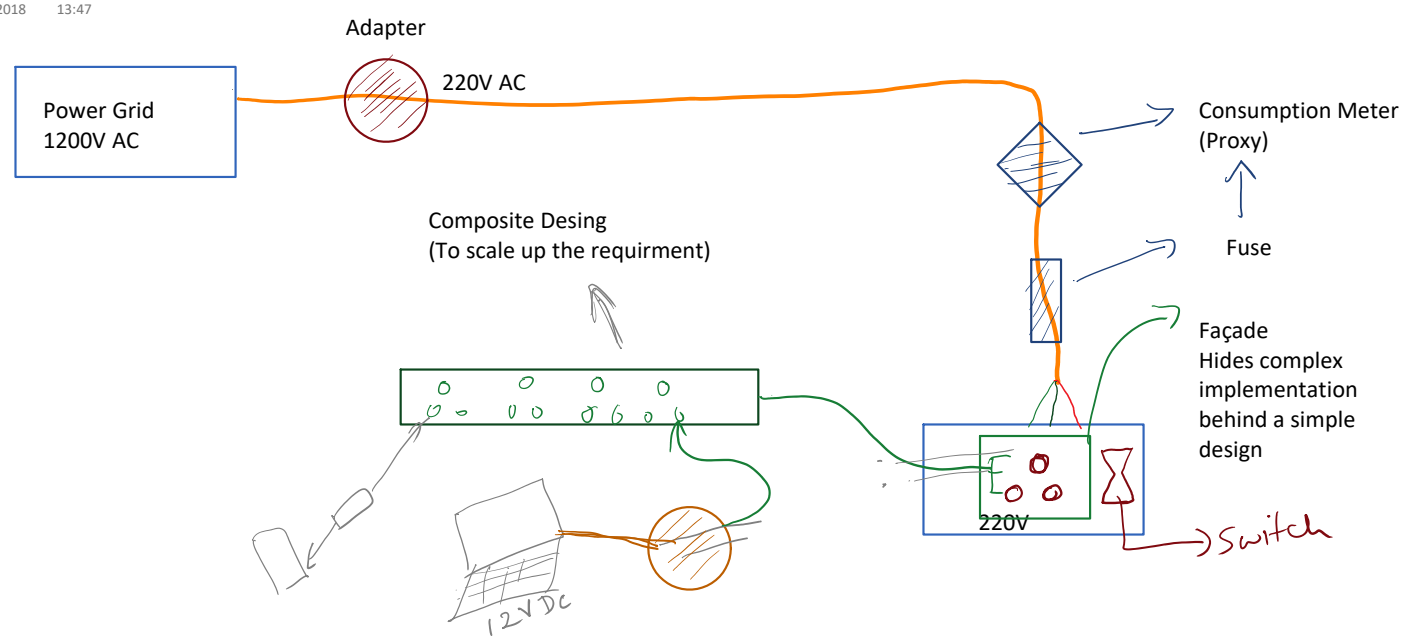
```
class UserRegistrationSystem{

    public void SendActivationLink()
    {
        Messanger m=new Messanger();

        m.Protocol=Smtp.Instance;
        m.Security=new UserNamePassword( "admin","admin");
        m.From=new Email("admin@web.com");
        m.To =new Email( newUser.Email);
        m.Subject="Activation Link";
        m.Body= new PlainText( activationText);

        m.Send();


    }
}
```

Messanger API(Façade)

29 March 2018 15:16

```
class UserRegistrationSystem{

    public void SendActivationLink()
    {
        new SimpleMailer().SendMail(newUser.Email,"activation
        link", activationText);
    }

}

class SimpleMailer{

    public void SendMail(String to, String subject, String body){

        Messanger m=new Messanger();

        m.Protocol=Smtp.Instance;
        m.Security=new UserNamePassword( "admin","admin");
        m.From=new Email("admin@web.com");
        m.To =new Email(to);
        m.Subject=subject;
        m.Body= new PlainText( body);

        m.Send();


    }
}
```

30 March 2018 13:47



TankWar

30 March 2018 14:09

Tank	Move	Attack	Defend
AggresiveTank	Runs to Enemy	Fire	Cover Fires
Defensive	Runs away from Enemey	Wait	Hides
Gurilla	Hapazard	Fires+ Move	Ducks

```
public double Average()
{
   double sum = 0;
   int count = 0;
   for (Node n = first; n != null; n = n.next)
   {
      sum += double.Parse(n.item.ToString());
      count++;
   }

   return sum / count;
}
```

encapsulate
whatever repeats

```
public R Execute<R>(ITask<T,R> x)
 {
    x.Init();
    for (Node n = first; n != null; n = n.next)
    {
      x.Process(n.item);
    }

    return x.Close();
 }
```

inject as dependency
(parameter)

```
interface ITask<T,R>
{
     void Init();
     void Process( T item);
     R Close();
}
```

abstract
whatever
changes

Decoupling

30 March 2018 16:27

- Getting rid of Bad Cohesion
- Separating unwanted connection (cohesion)

How?

- Apply DRY
  - Encapsulate whatever repeats (internal structure of service h object)
  - Abstract whatever changes (behavior client expects)
    - Referred as Callback
  - Client should pass callback as an object.

Key Players

- Service
  - Any piece of work that client needs to execute
    - Should be generic
    - Every service may need some specific knowledge from the client

  - Call back
    - An object that contains specific customized requirement of the client
  - Client
    - Uses service
    - Customizes service by passing it a callback object
  - Service+Callback => functionality

Strategy Design Pattern

- callback to specialize a generic algorithm
- A decision maker
- service layer performs the core job
- callback helps in making some client specific decision.
- callback is generally a very light weight object
- Condition

- **callback to ask client a question**

Command Pattern

- callback performs the corejob.
- service layer is just a facilitator
- service executes the callback at the right time
- Example:
  - ThreadStart
  - ThreadPool
  - SechduledTaskExecutor

Visitor

- callback to add new behavior the service layer.
- adds new behavior independent of the structural model
- visitor is a command to add new behavior

Observer

- Multiple callback (Subscribers) connected to Service (Publisher)
- Publisher may send upate notification to the client via callback object.
- Observer is a callback to inform client
- Publisher-subscriber model
- all event driven design is an observer
- A strong misconception
  - callback -> observer.