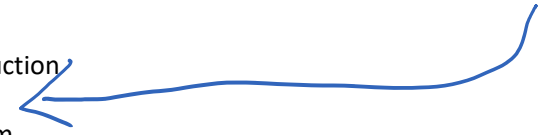
What is a Program

Wednesday, April 15, 2020 8:14 PM

Participants View

Is this an Object Oriented Definition?



A program is a colletion of instruction

(to whom?) the computer system

(why?) To perform a task

Popular Perception

Wednesday, April 15, 2020 8:06 PM

Abstraction

What?

hiding data.
hiding details.
hiding implementation.
showing only necessary information.

driver dont need to know the internal technical aspects of a car-abstraction ?

only useful information to be shown to end user.

Why?

End user may not require to know all minute details

How?

specifying in scope like private, public protected

is there a conflict here?

Encapsulation

What?

* grouping data and functions into a single object
* enclosing related information together
* wrapping data and function in a class
* prevent changes datA outside

Why?

* to protect information from other object
* data hiding

How?

* using class
* class with get and set property
* access specifier (private/protected)

Inheritance

What?

* reuse from parent
* derive properties from another class <-- what is another class?
* derived class inheriting the properties and behaviours from parent class
* hierarchy
* access properties of parent class by child class

Why?

* Avoid redundant code (code reuse)

How?

* language semantic

```
class X : Y{}
```

Polymorphism

What?

* behaving in different ways depending on the input received

who
functions

Why?

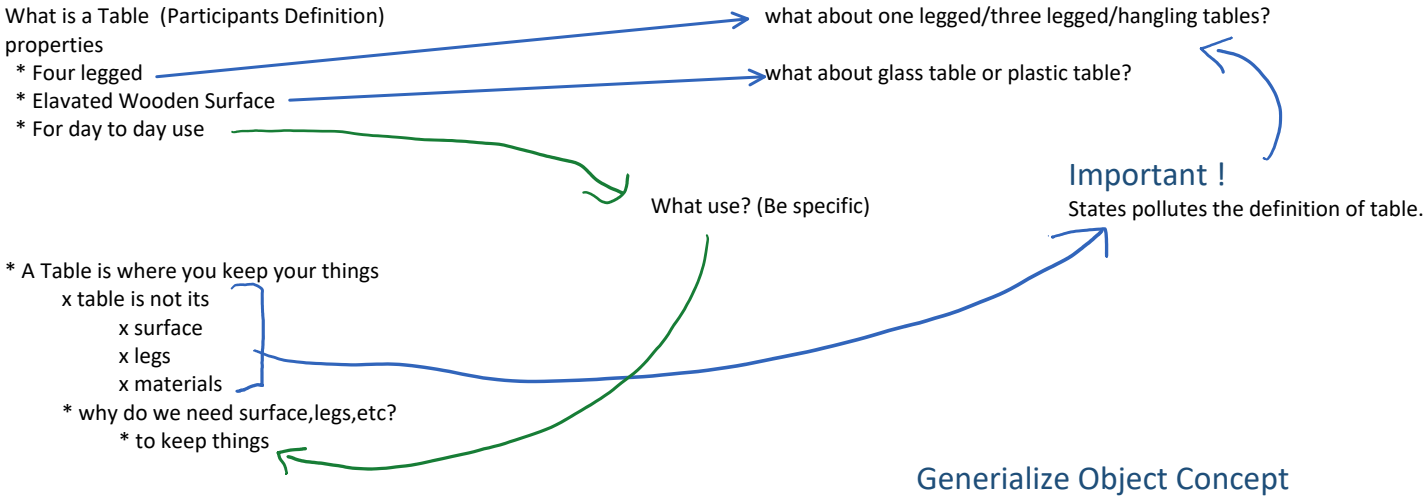
* Hide type/properties/methods
* features can be similar so we can specify them in base class?
* helps in abstraction---> you hide inner details

How?

* function and operator overloading
* method and overriding
* access modifier

What is an Object?

Wednesday, April 15, 2020 8:16 PM



Generalize Object Concept

* An Object is what object does.

* An object is a collection of Behaviors only
  * object is not a collection of states (Fields)

* why do we need state?
  * to implement behavior

What type of Programmer Are you?

Wednesday, April 15, 2020 8:21 PM

You are driving a bike on a busy day when your friend stops you asks a SFAQ (Silly Frequently Asked Question) --
Hey what are you doing? What would be your response?

A. I am burning the fuel
- engineering of driving, energy conversion, bike parts

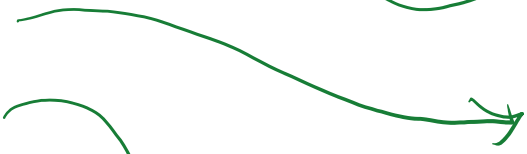
B. I am Driving
- following direction, following traffic rules etc etc

C. I am going to office

D. I am going to work
- Work from Home!

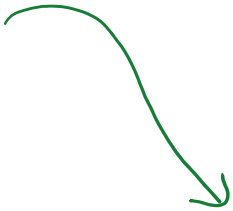
Assembly language thought process. Thinking in terms of memory, stack,heap and program as set of data and methods

Proceedural Thinking. In terms of instruction to achieve the task. Often in the details the real task loses the focus



Object Oriente Thinking

Focus on the business and defocus the implementation details. There are more than one proceedural approach to do whatever you want to do



Furhter Refinement of thinking. A more abstract generalized approach

Code Economics

Wednesday, April 15, 2020 8:27 PM

Everyline of code you write is either an expense or investment.

Code ---> Expense, Investment?

Expense Code --> main()/even-handler

Investment Code --> Reusable! Long Lasting Use!
printf()/Console.WriteLine

Animal Class

Wednesday, April 15, 2020 8:28 PM

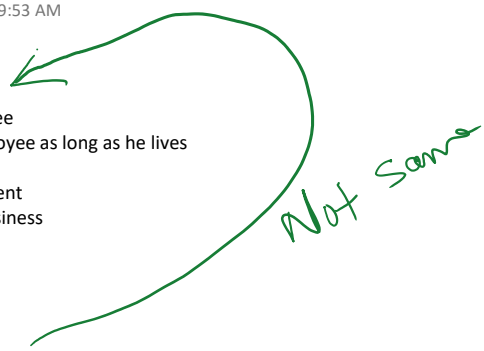
```
class Animal
{

	public void Move()
	{

		switch(animalType)
		{
			case AnimalType.Tiger:
				//move on land

			case AnimalType.Snake:
				//crawl
		}
	}


	public void Breed()
	{

		switch(animalType)
		{
			case AnimalType.Tiger:
				//child birth

			case AnimalType.Snake:
				//egg lay
		}
	}



}
```

Is A vs Has A

Thursday, April 16, 2020 9:53 AM

Gaurang is an Employee

- He is a born Employee
- Will remain an Employee as long as he lives
  - Forget
    - Retriement
    - Own Business



Gaurang Has an Employee

- Gaurang may have employed someone such as
  - Driver
  - HouseHelp

```
Employee gaurang=new Employee();

Console.WriteLine(gaurang.GetType()); //Employee

~~gaurang.SetType(); //~~unchangable
```

To change "is a" to "has a" you need to change your design

Gaurang **Has an Employeement**

- If you have an employement you can decide
  - Not to have an employement
  - Change the employement
  - **Have multiple employement**

Inhuman

Person is unqualified biological species with

- some behavior
- some skills
- some relationship

```
class Person{
        Behavior behavior
        List<Relation> relations;
        List<Skills> skills;
        …
}
```

```
//C# code

var gaurang= new ~~Human();~~Person();

gaurang.Behavior=new HumanBehavior();

gaurang.Employement= new LTTSEmployement();

//unemployed
gaurang.Employement=null;

//change employement
gaurang.Employement=new SelfEmployement()


//C# code for multiple employements
var gaurang=new ???();

gaurang.Employements.Add( new SoftwareDeveloper());
gaurang.Employements.Add(new YoutubeBlogger());



var vivek=new ???()

vivek.Role=new TrainerRole();
vivek.Work(); //works as trainer

vivek.Role=new Driver();
vivek.Work(); //works as Driver
```

Grammar of OO Design

Thursday, April 16, 2020 10:30 AM

Traditional/Academeic/Outdated Approach

Convert Noun to -----> class or object

Covert Verb to -------> class method

How to convert Adjective, Adverb, Preposition, Conjunction?

Modern Understanding of Object Oriented Design

Convert Everything to Object

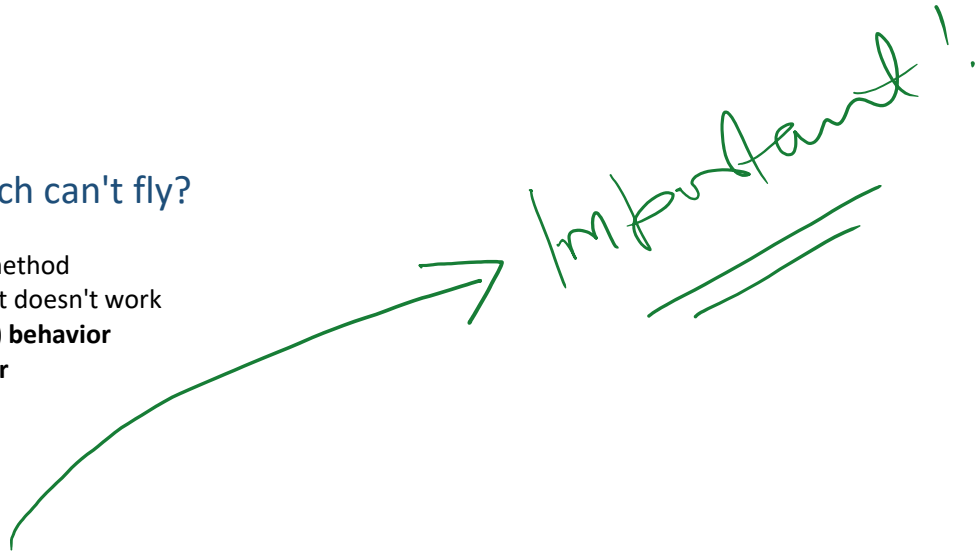
- You are an object
- Your states (age, height) is an object
- Your behaviors are object
- Your nature is object
- Your skills are object
- You relation to other object is also object

What is the difference

Thursday, April 16, 2020 11:22 AM

Humans can't fly vs Ostritch can't fly?

- Ostritch being a Bird has a fly method
  - But implmentation wise it doesn't work
  - It has **a non-working fly() behavior**
- Humans don't have **fly behavior**



What is True?

A. Square Don't have any orientation
B. Square have oreintation equal to None

Rectangle-Square Problems

Thursday, April 16, 2020 11:39 AM

1. Rectangle require 2 int to define, Square needs only 1
2. Rectangle has set method for Width/Height, Square's width, height can't be different
3. Rectangle has Orientation, Squares don't have Orientation

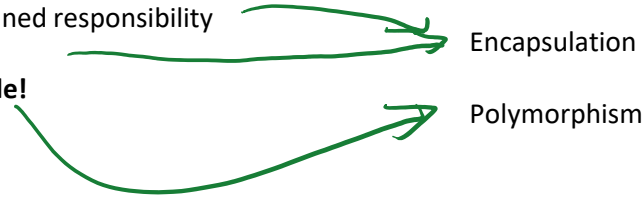
While Inheritance is easy, there is way to uninherit or partially inherit!

What is a software component

Thursday, April 16, 2020 12:58 PM

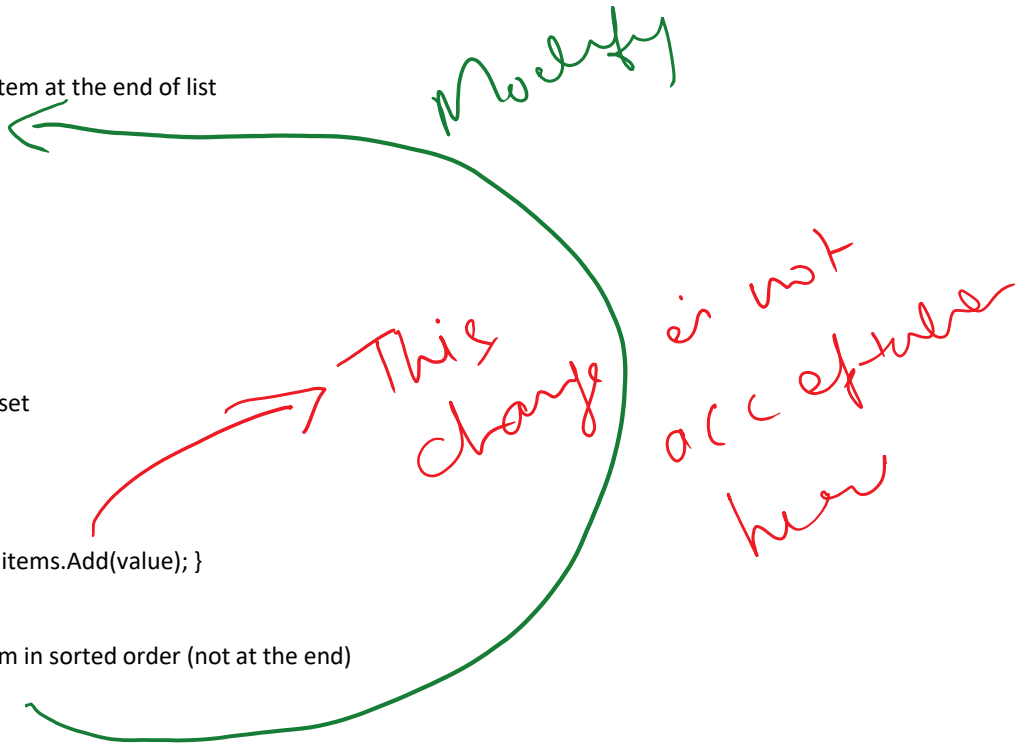
It has 3 qualities

1. It has a weldefined responsibility
2. It is reusable
3. It is **Replaceable!**



Friday, April 17, 2020 10:55 AM

```
class LinkedList
{
        void Add(Object item){ //adds item at the end of list

        }
        Object Remove(int pos){

        }

        int Size(){

        }
        Object this [](int index){} //get/set
}

class Stack{
        LinkedList items;
        public void Push(Object value){ items.Add(value); }
}

what if I need an LinkedList to add item in sorted order (not at the end)
//we can go and modify the add logic
```



Meaningful word

Friday, April 17, 2020 12:07 PM

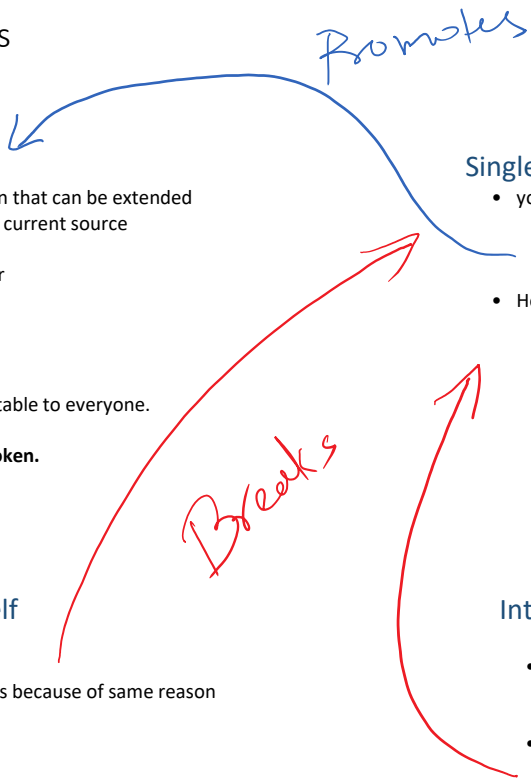
```
class Triangle{

        int Solve(){}
        int Calculate(){}
}
```

Together Triangle,Solve and Calculate doesn't make any sense

Meaningful means we can understand the responsibility of a method

Designed Principles

Monday, April 20, 2020 9:43 AM



Open Close Principle

- Create a Future Proof design that can be extended
  - but without changing current source

- changing source may trigger
  - compile
  - test
  - distribute
  - deploy cycle
- a change may not be acceptable to everyone.
- **changes should be additive**
- **don't mend it if it is not broken.**

Single Responsibility Principle

- your objects, methods, component should have a single responsibility
  - One reason to exist
  - One reason to change
    - A good single responsible component may never change!
- How
  - Meaningful names
    - avoid names joined with and/or
    - avoid abstract names for concerete class
    - most method should access most field most time
    - and too many methods

Don't Repeat Yourself

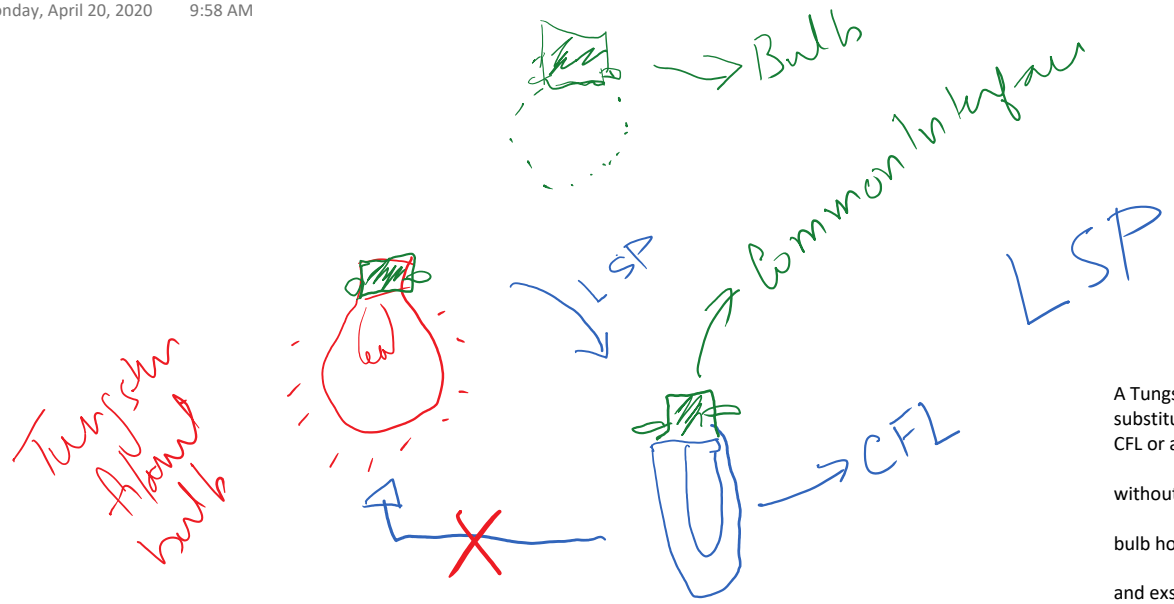
- Avoid Redundant code
- Avoid code that changes because of same reason

Interface Segregation Principle

- Avoid fat interface
  - interface shouldn't have optional or mutually exclusive methods!
  - There shouldn't be methods client may not need.
- Fat Interface means Fat class
  - violates SRP

A Bulb Replacement

Monday, April 20, 2020 9:58 AM



A Tungston filament bulb can be substituted (replaced) by a CFL or an LED bulb

without changing the

bulb holder (client)

and exsting

tungston filament buib (component)

CFL doesn't inherit Tungstson filament bulb

What does it inherit?

A CFL or LED is not a substitution for Tungston Filament Bulb,

They are substitution for a bulb

BankAccount and ATM

Monday, April 20, 2020 10:11 AM

```
class LocalBankAccount : BankAccount{

        public void Withdraw(int amount, string password)
        {

                ...
                throw new SqlException();
        }

}
```

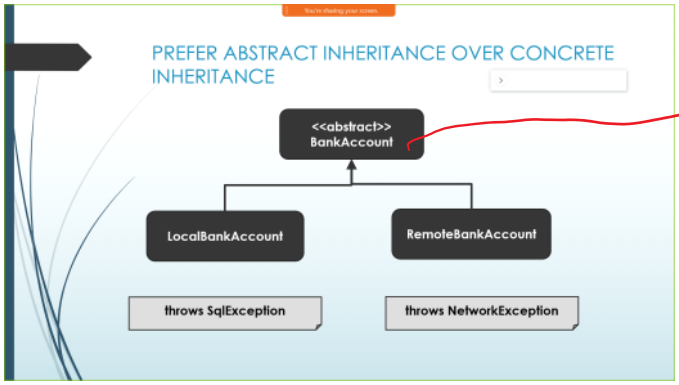
```
// A future change

class LocalBankAccount  : LocalBankAccount{

        public void Withdraw(int amount, string password)
        {

                ...
                throw new NetworkException();
        }

}
```

```
class ATM //client
{

        public withdrawOption(){

                try{
                 account.Withdraw(amount,password);
                dispenseCash(amount);

                } catch(SqlException e){
                  printErrorInfo();
                }


        }
```

designed to handle this exception

Not designed to handle This exception

ATM (client) will crash



What exception does the blass class document?

Can I know what exception will be thrown by sub classes in future?

Can I know all the exceptions that the future implementation shall throw

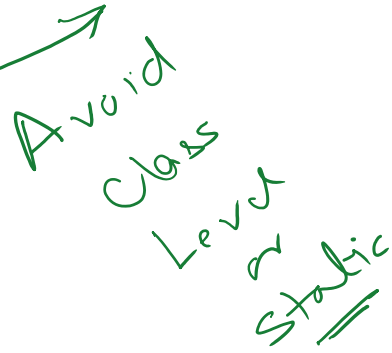
- Exceptions depends on implmentation

static vs non static

Tuesday, April 21, 2020 9:52 AM

```
class BankAccount{

        int accountNumber;
        int amount;
        string pass;

        public void Withdraw(int amount, string pass){

                …
                this.amount-=amount;
                …
        }

        public void static Transfer(BankAccount src, int amount, string password,
        BankAccount target){

                this.amount+=amount; //there is no this!
                …
                src.amount-=amount;
                …
                target.amount+=amount;

        }

}
```

```
var a1=new BankAccount(1,50000,"pass");
var a2=new BankAccount(2,50000,"pass");

a1.Withdraw(1000,"pass");

BankAccount.Transfer(a1,1000,"pass",a2);
```



constructor - static or non static?

Tuesday, April 21, 2020 9:52 AM

```
class BankAccount{

        int accountNumber;
        int amount;
        string pass;


        public BankAccount(int accountNumber, int amount, String password)
        {
                this.accountNumber=accountNumber;
                …
        }


        public void Withdraw(int amount, string pass){

                …
                this.amount-=amount;
                …
        }

        public void static Transfer(BankAccount src, int amount, string password,
        BankAccount target){

                this.amount+=amount; //there is no this!
                …
                src.amount-=amount;
                …
                target.amount+=amount;

        }

}
```

```
var a1=new BankAccount(1,50000,"pass");
var a2=new BankAccount(2,50000,"pass");

a1.Withdraw(1000,"pass");
BankAccount.Transfer(a1,1000,"pass",a2);
```

```
// can't invoke using class reference
//var x= BankAccount.BankAccount(1,50000,"pass");

//No class or object reference is used
BankAccount a1= new BankAccount(1,50000,"pass");

//can't use object reference
var a2= a1.BankAccount(2,50000,"pass");
```

It's a special function,
it is the creator of an object, not
the part of the object

neither static (object level)
nor class level (non-static)

class defines the object

why should creator defintion be present in object?

At the top you need either a constructor or a static factory

Tuesday, April 21, 2020 11:41 AM

```
RBI rbi= goi.GetRBI();  //-> where will it end?
Bank icici= rbi.GetBank("ICICI");
BankBranch iciciManyataBranch= icici.GetBranch("ManyataTechPark,BLR");
BankAccount account =  iciciManyataBranch.OpenAccount(…); //a factory
```

Proxy: Composition Vs Inheritance

Wednesday, April 22, 2020 11:59 AM

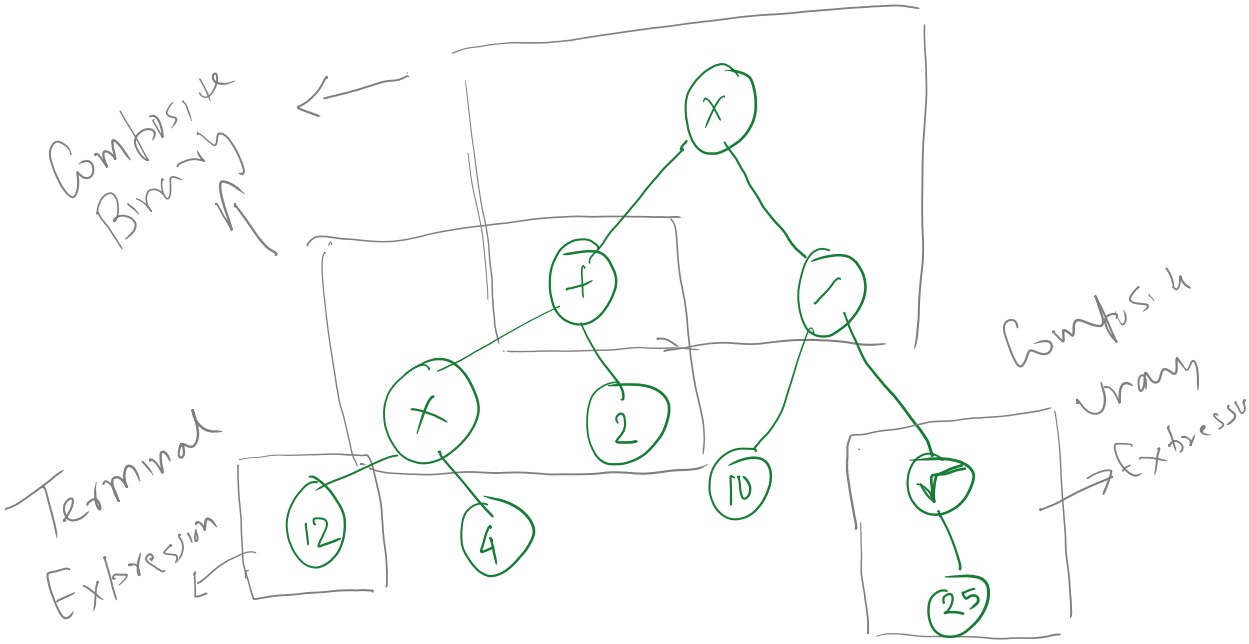
```
public interface ISearchEngine{
        IResult Search(Iquery q);
}

public class BookSearchEngine : ISearchEngine{
    …
}
```

```
//Composition Proxy


class AuthenticatedSearchEngine : ISearchEngine{
        ISearchEngine target;
        public AuthenticatedSearchEngine(ISearchEngine t){target=t;}

        public IResult Search(IQuery q){
                if(!HttpContext.IsAuthenticated)
                        throw new SearchFailedException();
                else
                        return target.Search(q);
        }

}
```

```
//Inheritance Proxy


class AuthenticatedSearchEngine : BookSearchEngine{

        public IResult Search(IQuery q){
                if(!HttpContext.IsAuthenticated)
                        throw new SearchFailedException();
                else
                        return base.Search(q);
        }

}
```

Composite -- Expression Solver

Wednesday, April 22, 2020 12:58 PM




```
public interface IExpression {

        double Solve();

}

//Terminal Expression
public class ValueExpression : Iexpression{

        public double Value{get;set;}

        double Solve(){ return Value; }
}
```

```
//CompositeExpression
public class BinaryOperator : IExpression{

        public  IExpression Left{get;set;}
        public  IExpression Right{get;set;}

        protected  abstract SolveInternal(double x,double y);

        public double Solve()
        {
                var l= Left.Solve();
                var r= Right.Solve();

                return SolveInternal(l,r);
        }

}


public void PlusOperator : BinaryOperator{

    protected override SolveInternal(double x,double y){
        return x+y;
    }
}
```

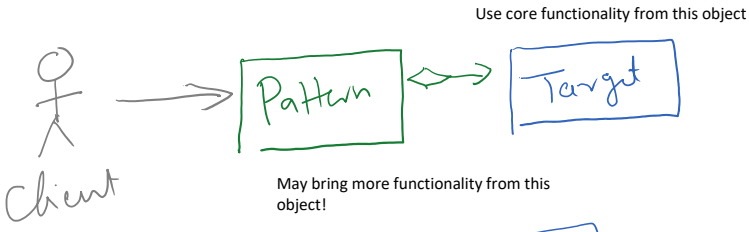
```
var expr= new MultipicationOperator(){

        Left = new PlusOperator(){

                Left= new MultiplicationOperator(){
                        Left=new ValueExpression(12),
                        Right=new New ValueExpression(4)
                },
                Right=new ValueExpression(2)
        },
        Right= new DivideOperator(){
                Left=new ValueExpression(10),
                Right=new SquareRootOperator(
                        new ValueExpression(25))

        }

};


var result =expr.solve()
```

structural patterns

Thursday, April 23, 2020 9:42 AM

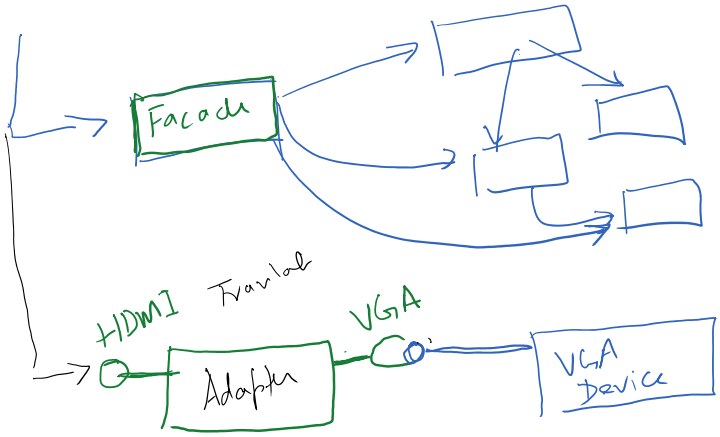
What is Common

- Wrapper Design Patterns
  - Client calls the pattern object.
  - Patterns object reuses the functionality from some other (target) object
  - The pattern and underlying target works toghether as a unit
- **They represent Encapsulate To Reuse!**
- **Multiple objects combine together to work as a single Unit.**



Façade

- Pattern wraps complicated interactions between component
- Provides a simplified way to interact with a compex system.
- Reduce points of dependency
- Provides a new simpler method call and hides a larges number of component or steps



Adapter

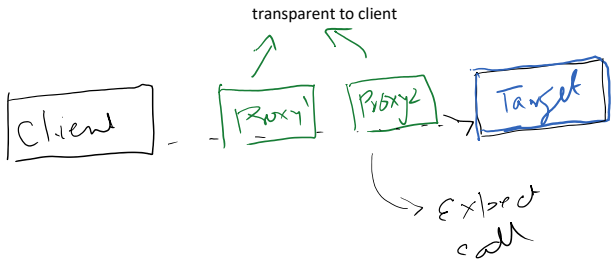
- Translates interface
- Makes an existing object accessible under new interface
- Makes two objects not designed to work with each other work
- **Doesn't add new functionality. Just collects the functionality**
- Late Design Solution
- Makes Functionaly usable for different interfaced object work.
- **Example**
- **[Laptop with HDMI Output] <---(HDMI VGA Adapter) ---> (Projector with VGA Input)**

eg. projector

can be used with
laptop without VGA but with
HDMI socket
This is not a projector. just converter

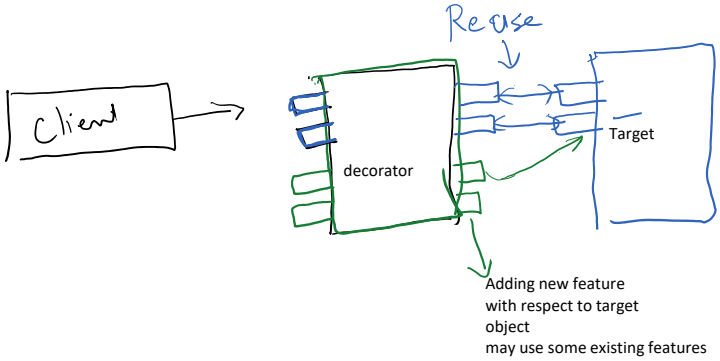
Proxy

- Doesn't Translates interface
- Doesn't Add New Functionality
- It Controls Current Functionality
- **It eliminates** **Inherit to override usecase**
- May or may not call the target
- May perform optional activities before and/or after the target
- One of the most widely used patterns
- Implements same interface as actual target
- Often transparent to client (client may not know there is a proxy)
- Can be chained.
- Comoon uses
  - Webservice Proxy
  - Aspect Oriented Programming
  - Filters
  - Firewalls
  - Caching
  - Logging
  - Unit Test Mocking



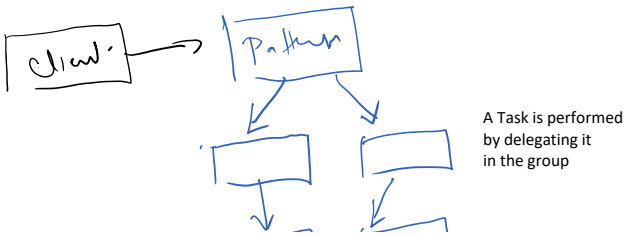
Decorator

- Proxy+
- Adds new functionality to existing object
- **It eliminates** **Inherit to add new functionality usecase**
- They expose all the current functionaity of the current object
- can add new functionality
- Multiple Decorators can be applied
-
- **Not** transparent to client
- Client knows proxy to use additional behaviors
- Creates Onion Layer Structure

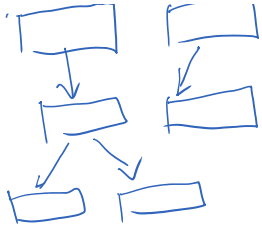


Composite

- Composites Patterns is composite of multiple object
- Pattern may encapsulate more than one target
- It's a Recursive Tree kind of design pattern.
- Each encapsulated object may encapsulate even more sub target
- An action performed on the outer most object is delegate to all chidlren
- Comoon uses
  - Expression Tree
  - Organization Tree
  - GUI Tree



- Expression Tree
- Organization Tree
- GUI Tree



by delegating it in the group

Tank

Thursday, April 23, 2020 10:35 AM

```
abstract class Tank
{
	public abstract void Attack();
	public abstract void Defend();
	public abstract void Move();
}

abstract class AggresiveTank: Tank
{
	public abstract void Attack(){  Console.WriteLine("Fire") }
	public abstract void Defend(){  Console.WriteLine("Cover Fire") }
	public abstract void Move(){  Console.WriteLine("Move Towards Enemey") }
}

abstract class DefensiveTank: Tank
{
	public abstract void Attack(){  Console.WriteLine("Wait") }
	public abstract void Defend(){  Console.WriteLine("Hide") }
	public abstract void Move(){  Console.WriteLine("Move Away Enemey") }
}

abstract class GurillaTank: Tank
{
	public abstract void Attack(){  Console.WriteLine("Fire+Move") }
	public abstract void Defend(){  Console.WriteLine("Duck") }
	public abstract void Move(){  Console.WriteLine("Hapazard Move") }
}
```

Cohesion -- Good and Bad

Thursday, April 23, 2020 11:36 AM

Cohesion

What?

- Unitiy
- Sticking Together/Being Together
- Encapsulation
- Dependency
- **Is any to things put together is Cohesion? -> YES**
- **Is every cohesion good? -> NO**

Shopping Basket That Contains
- 1 KG Tomoato
- A tooth Paste
- A Shoe Police
- A Teashirt

Is this a cohesion? --> YES

Why are they together? --> coincidence

Coincidential Cohesion

Programming Scenario --
- A **utils.dll or namespace** containing classes like
  - Date
  - Random
  - LinkedList
  - StringBuilder

Bathroom Rack That contains
- Toothpaste
- Showerjel
- Shampoo

Why are they together? --> They are used at same point of time

Temporal Cohesion

Programming Scenario --

- Everytime you open a database connection, you log the information
- We can put Logger and Connection classes together

Your Medine Shopping Cart Includes
- Desprine
- Paracitamole
- Painkiller ointment

Why are they together? --> They are used at same point of time

Hierarchial Cohesion

classes from same hierarchy stays together

Programming Scenario --

- Collection Classes in **Collection namespace --> LinkedList, Set, Dictionary**
- **Math class** contains all Math functions

Objects from the same hierarchy are very often competitors and mutually exclusive
They work in place of each other and not together

- **you may use either a LinkedList or Dcitionary to store a particular information**
  - **you will not use both of them**
- **How many Math functions you use together generally?**

Your Sport Kit containing
- A Tennis Bat
- A Tennis Ball
- A Tennis Shoe

Notice
- The are not similar
- They don't belong to same hierarchy
- They are not made of same material
- They don't have the same job

Why are they together? --> They work together to perform the responsibility
- **You can't use a tennis bat without ball and shoes**
- **You can't use ball with bat and shoes**

Functional Cohesion

Objects which together work as a responsibility is a Functional Cohesion

Programming Scenario --

- Stack class with methods Push-Pop-IsEmpty
  - **If you push something you will certainingly like to pop it out at somepoint**
  - **You are not pushing to black hole**
  - **You can't pop if haven't pushed**
  - **Push change stack from IsEmpty to not IsEmpty**
  - **Pop may change stack from !IsEmpty to IsEmpty**

Functional Cohesion is the only desirable cohesion

- They defien **Single Responsibility**
- **All Structureal Design Patterns are excellent example for Functional Cohesion**

Cohesion and Coupling

Thursday, April 23, 2020 11:59 AM

Coupling

- Being connected.
- Coupling => Bad Cohesion

Decoupling

- Getting Rid of Bad Cohesion

Cohesion

- Study of how well the parts of a system (internal) work well together to perform aresponsibility

Decoupling

- How independent is One system from another (external independence)

Design Goal --> High Cohesion (Strong bound to implement single responsibility) Low Coupling (High Independce) and

- Single Reponsibility Principle
- Law #1 of Object Oriented Design
  - Encapsulate to Reuse
- All Structural Pattern Represents this idea very properly
  - Adapter and Adaptee works together
  - Proxy chain can configure different behavaiors to work together with target
  - Decorator adds new functionality to the target

- Dependency Inversion Principle
- Law#2 of Object Oriented Design
  - Prefer Abstract Inheritance
- All Beahvioral Design Patterns try to achieve Decoupling

Car Vs Google Car

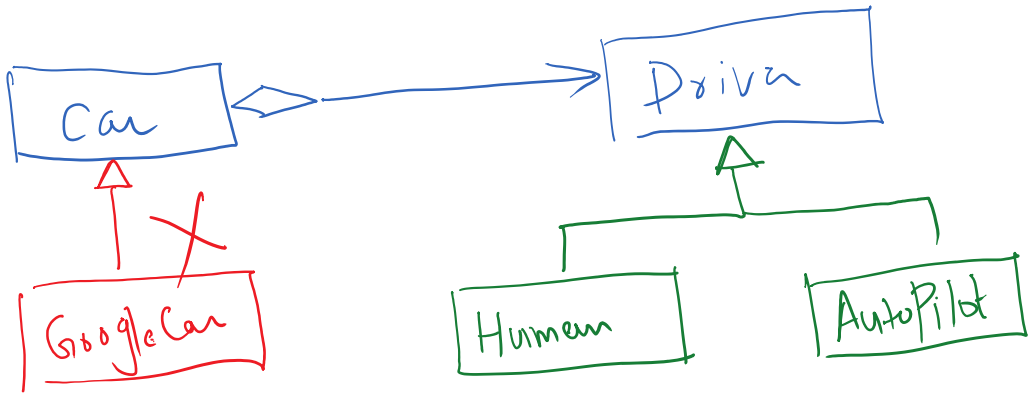
Friday, April 24, 2020 12:08 PM

```
public class Car
{
        Driver driver;

        public void Start(){…}
        public void Stop(){…}
        public void Trun(){…}
        public void Speed(){…}

}
```

```
public class GoogleCar :  Car
{

        public void Drive(){…}
}

public class TeslaCar :  Car
{

        public void Drive(){…}
}
```

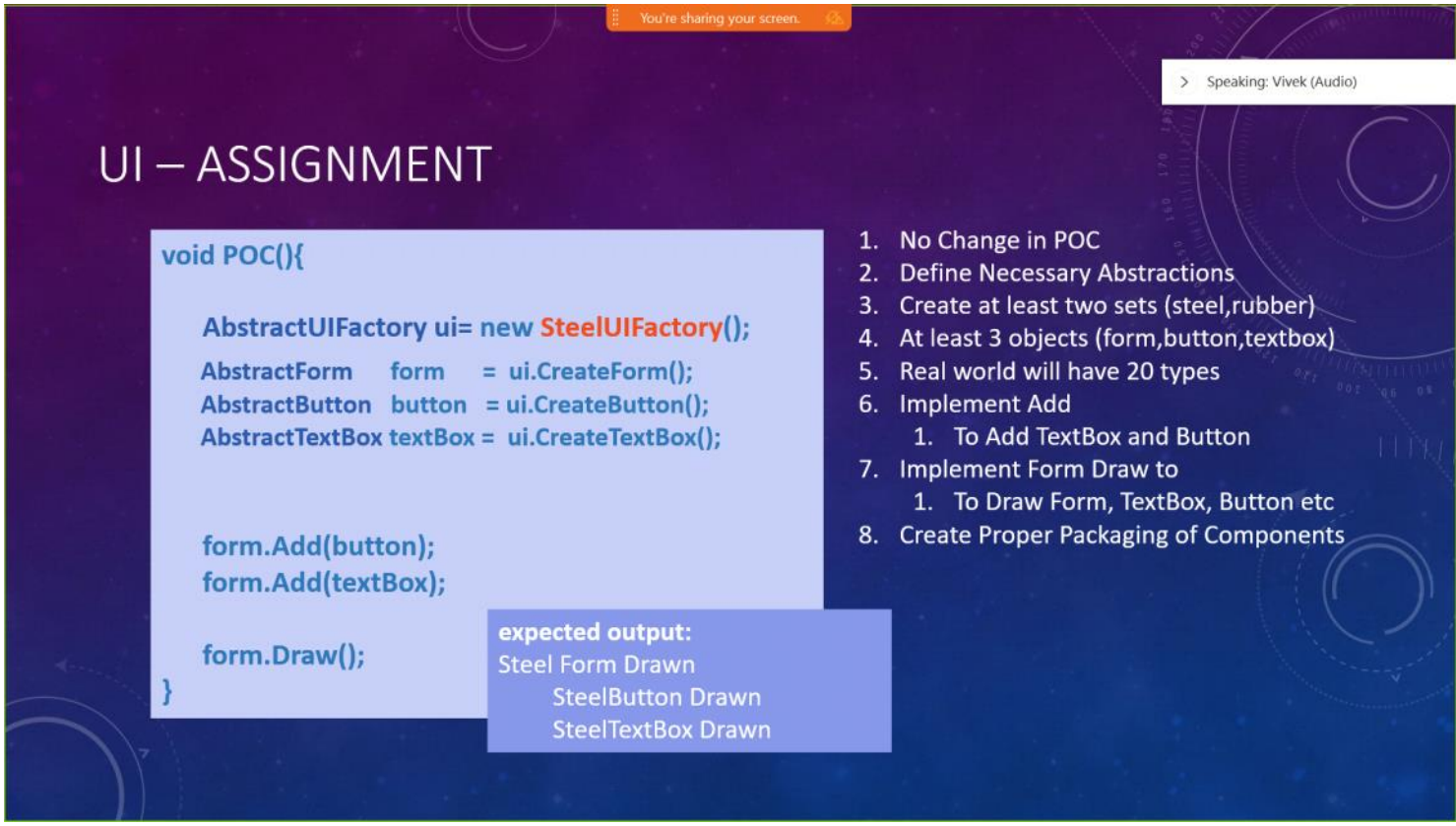
```
public interface Driver
{
        void Drive();
}

public class HumanDriver: Driver
{

}

public class AutoPilotDriver : Driver
{

}
```

```
var hondaCity=new Car( new HumanDriver());

var googleCar=new Car(new AutoPilotDriver());
```



The Abstract Factory Assignment

Tuesday, April 21, 2020 12:02 PM




```
static 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();
}
```

when Draw() is called

Expected Output

```
Steel Form Drawn
    Steel Button Drawn
    SteelTextBox Drawn
```

1. No Change in POC (Main)
2. Define Necessary Abstractions
3. Create at least two sets (steel,rubber)
4. At least 3 types of objects (form,button,textbox)
5. Real world will have 20 types <-- remember while coding
6. Implement Add
   - To Add TextBox and Button
7. Implement Form Draw to
   - To Draw Form, TextBox, Button etc
8. Create Proper Packaging of Components