**Design Patterns Theory:**

Design Pattern is a concept which is language agnostic. It can be implemented in any programming language.

**What is a Design Pattern?**

Why we create a software? (E.g. – Flight Booking, Shopping Mall, E-Commerce Web Site)

To perform some repetitive tasks.

To solve business problems. Every business has certain problems. To reduce manual efforts, to automate those efforts we create software.

What do business problems consist of?

Real Life Entities/Objects

Who will create a software?

Developer

How will you create a software?

By writing a “code” using “some programming language.”

You know everything about programming language. But you don’t know anything about entities.

E.g. – You don’t know role of invoice in shopping mall, you don’t know role of customer in banking system.

Unless you know how entities interact with each other, you cannot create a software.

The thought process of a developer must revolve around entities.

The “developer” has to use an “approach” oriented “towards real life entities”.

The language which is used to create the software should allow the developer to “map” real life entities into the software.

Such languages are known as “Object Oriented Languages”.

--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

When you start creating a software using programming language like C# or Java, you will come across

Software Design Problems.

These design problems are recurring.

E.g.-

How do you object of a class?

Using “new” keyword.

How do I iterate via collection of objects?

foreach loop/for loop

What if you want to customize the iteration?

People working in the IT industry have long back found solutions to such problems. These solutions have been tried out and tested.

These solutions will be used to solved these problems.

These solutions which have been tried, tested and they have proved to solved these kinds of recurring problems, these solutions are referred as design patterns.

**Design Pattern:** A tried and tested solutions to a commonly occurring or recurrent software problem.

**You cannot use any design pattern anywhere.**

**Every design pattern is designed to solve a particular problem.**

You first need to understand the problem, then pick up the design pattern, determine whether that design pattern solves your problem. If it does, then tweak that

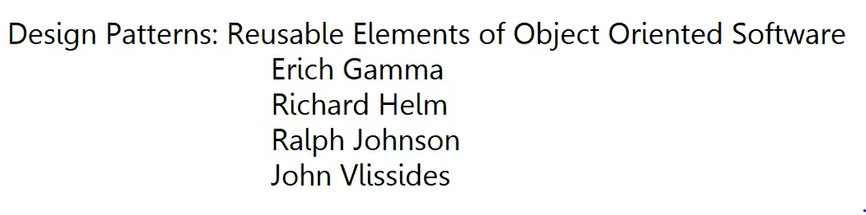
Design Pattern a little (minor modifications) and arrive at a solution.

**Why Design Patterns?**

Time Proven Solutions to recurrent software problems.

No need to reinvent the wheel.

­­­­­­­­­­----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------



Documented 23 Design Patterns.

Authors are popularly known as Gang of 4.

Patterns are popularly known as GoF patterns.

Available for download in PDF format. Perform Google Search.

Language bit difficult to understand.

GoF have Categorized Design Patterns in 3 Categories:

a) Creational Patterns

b) Structural Patterns

c) Behavioral Patterns

3-4 important patterns from each category will be covered.

Depending on the software problem, you have to brainstorm and decide what kind of problem it is and then decide which category of design pattern fits your problem.

Every pattern will have a name, every pattern will have an intent (problem that pattern solves), every pattern will have certain components (things involved in that pattern), every pattern will tell you how those components communicate with each other or what is the relationship between these components. You need to have good understanding of these things. Then you have to slowly map those patterns to your software problems. That is how you learn design patterns. Learning design patterns is a continuous learning process. You keep learning them as you implement them. You learn them with experience.

Only one book cannot give you 100% knowledge of design patterns. GoF book acts as a foundation.

---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Creational Patterns:**

Deal with Object Creation.

**Objective:**

To isolate or separate logic of object creation in client code.

a. Singleton

b. Factory

c. Abstract Factory

d. Prototype

e. Builder

a. **Singleton**

There should be only one instance of a class at any given point in time in your application.

Throughout lifetime of the app, there must be only one instance of some desired class.

Used in scenarios where resources need to be shared among multiple components of your application.

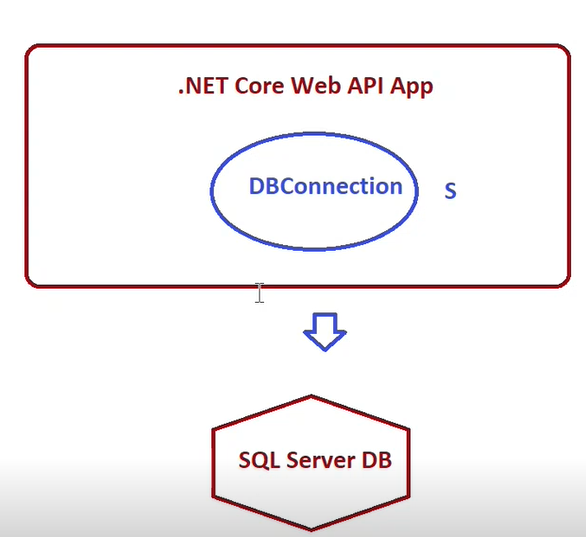
E.g.- One Printer is shared among all the employees working on that floor. Each floor will have one printer.

If Printer is a class, there will be only one instance of that printer for each floor.

Example 2:

Single instance of DBConnection throughout lifetime on an application.

Single Instance because it may be costly to create multiple instances. It occupies lot of resources. You cannot create a separate DBConnection instance for each request as it will occupy network bandwidth.



**Few Points for Singleton:**

a) 1 and only 1 instance

b) Logic for Singleton is within the object itself (E.g. – services.AddSingleton() method in ASP.NET Web API Core)

c) Private Constructor: Class which will be singleton should have a private parameter less constructor. External world (Client code) should not be able to create an

instance using new keyword. If class is not sealed and simply constructor is private then nested class can inherit from that class and outside

world can create instance of this nested class. Making class sealed is a genuine way to prevent the inheritance.

d) Usually a sealed class – Not mandatory. Better for performance reasons.

e) Lazy Loaded and Thread Safe

f) Specific way to obtain the instance 🡪 using static method or property

When you have a problem where client code should not know how to create an object and what object to create, that’s where creational patterns come into picture.

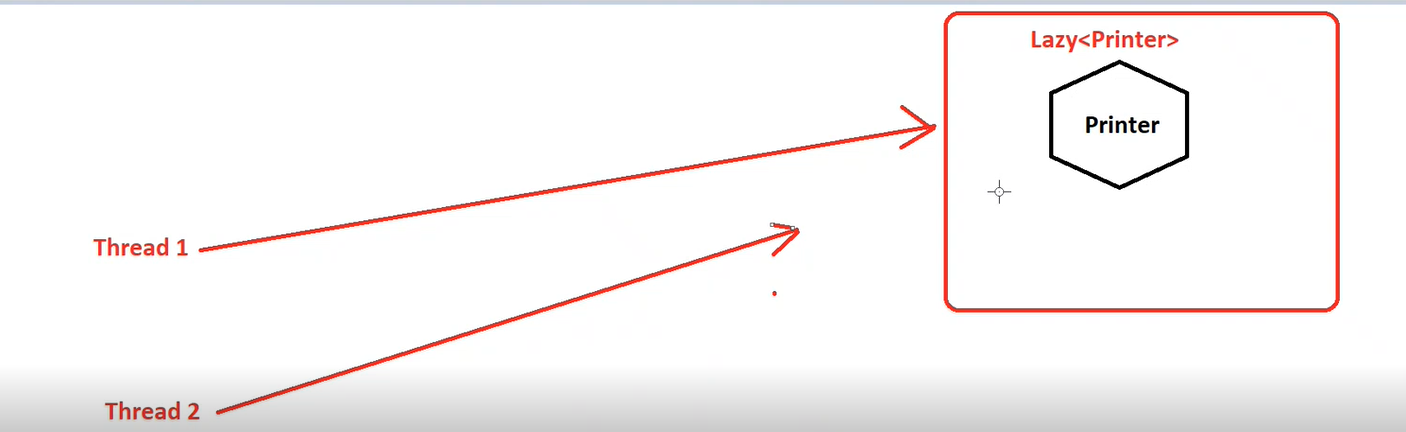
**Client Code:** Code written inside Program.cs file. If you are using some dll, you will create instance of a class in that dll in Program.cs file.

To make the class thread safe, .NET 4.0 has introduced Lazy class.

Lazy<T>

a. Allows lazy initialization

b. Is thread safe



Lazy has a built-in feature to ensure that at a time only one thread can access the code.

Functionalities in .NET Core use Design Patterns.

E.g. – foreach loop uses a design pattern known as Iterator Pattern.

b. **Factory Method:**

int x = Convert.ToInt32(“20”) : Convert is a factory class. ToInt32() is factory method. ToInt32() method is where data type conversion happens.

double d = Convert.ToDouble(“20.56”): ToDouble() is factory method

1) Create a method (factory method) for object creation.

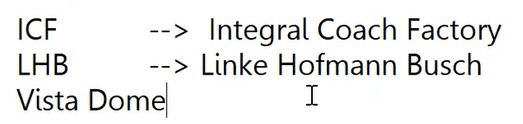
2) Invoke the factory method.

In programming, there are scenarios where you need to create instances of multiple classes. But which instance to create depends on the condition. You don’t need to

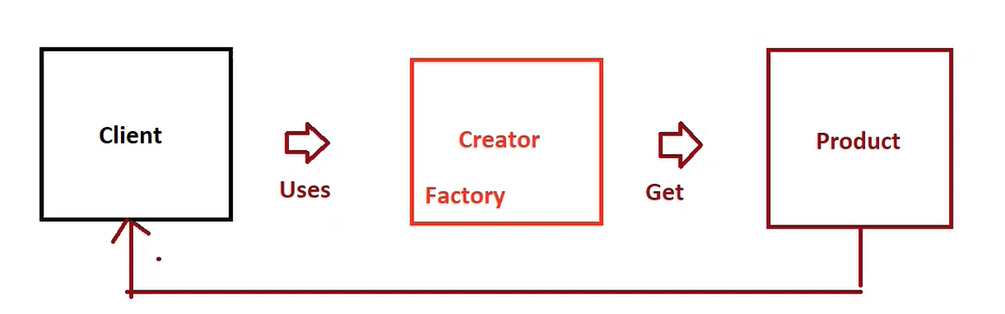
bother about process of creating instances. That’s where Factory pattern comes into picture.

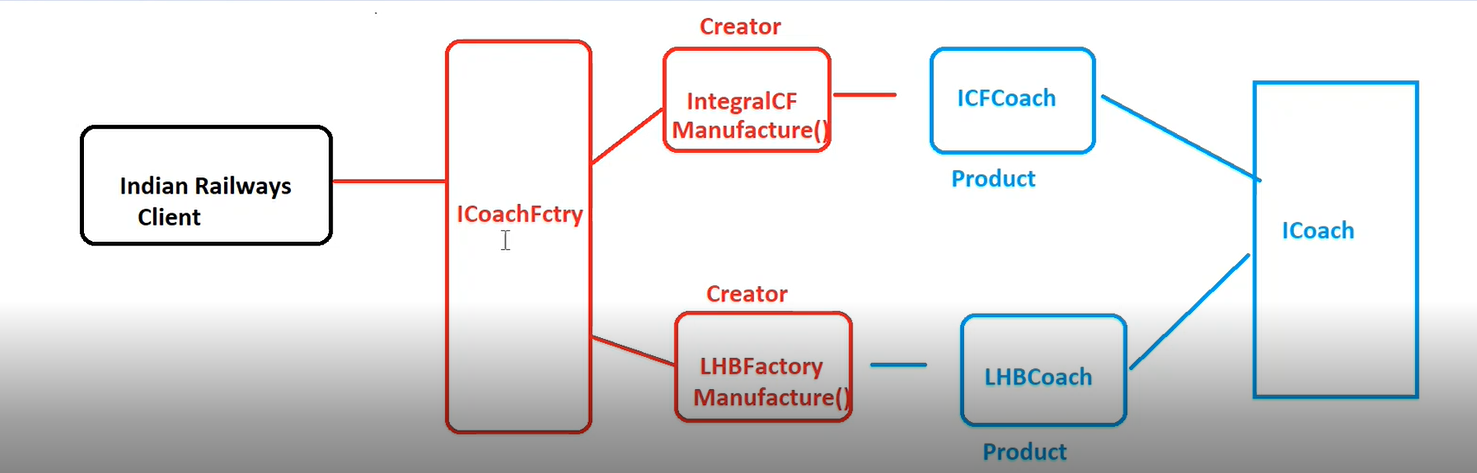
**Example:**

Indian Railways use 3 types of coaches:



3 players in factory pattern:





Indian Railways does not know which coaches to manufacture. It depends on demand. Whenever demand comes for particular coaches (E.g.- Vista Dome coaches), Indian Railways will simply get in touch with ICoachFactory to get the coaches.

c. **Abstract Factory:**

Extension of factory

Factory of factories where the main factory will give you abstraction of other factories.

Encapsulates set of factories having a common requirement without exposing the concrete factory classes.

Encapsulates number of other factories without exposing the concrete factory.

The product which you get back will have a family of products associated with it.

**E.g.-**

**SqlClientFactory** **OracleClientFactory**

SqlConnection - SqlConnectionFactory OracleConnection

SqlCommand - SqlCommandFactory OracleCommand

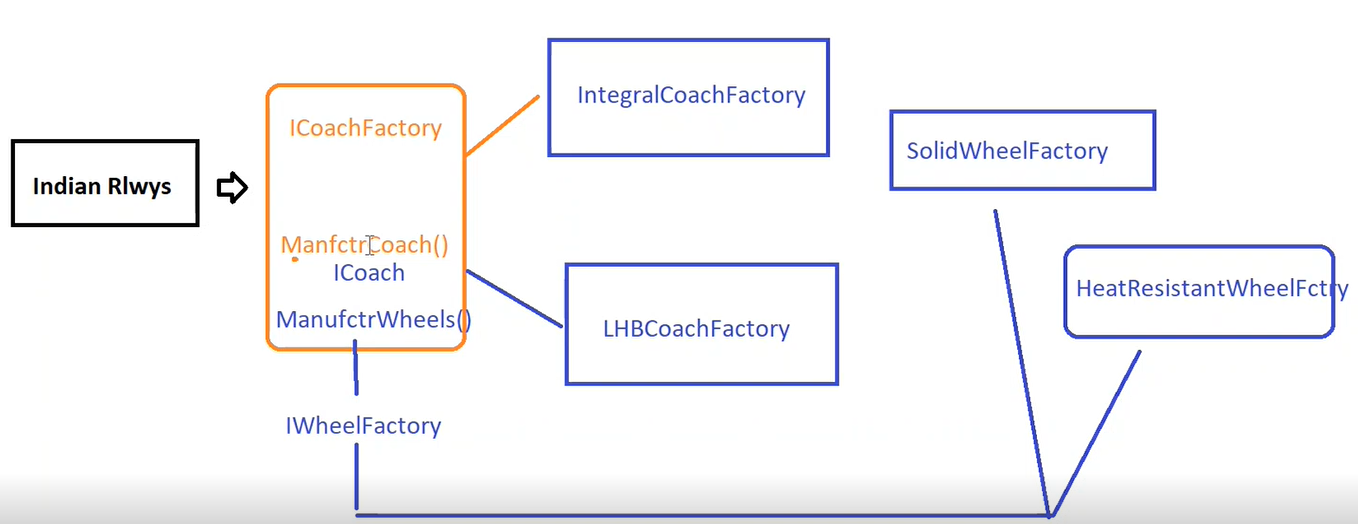
SqlParameter

SqlTransaction

When you ask SqlClientFactory to give you SqlClient, it comes with these combinations of objects.

**Indian Railways Example:**

CoachFactory will encapsulate WheelFactory as well and CoachFactory will decide which WheelFactory to use. This is where Abstract Factory pattern comes into picture.



d. **Prototype:**

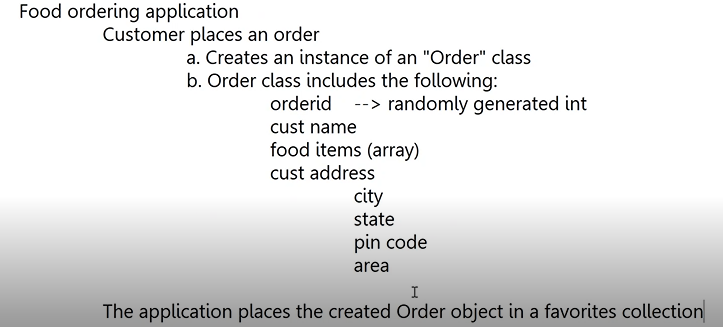
Create objects by cloning the existing ones.

Cloning logic is inside the class whose object needs to be cloned.

Template that specifies how cloning will be done.

Useful when you simply preview the changes. E.g.- Positioning mouse on Teams camera

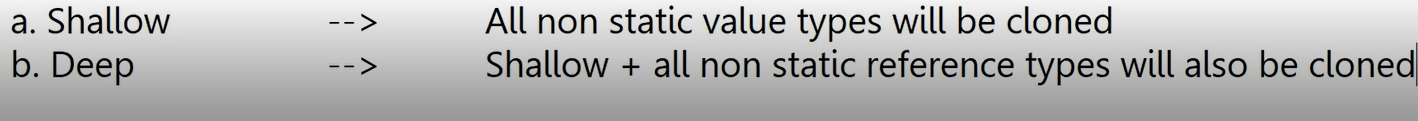
**Example:**

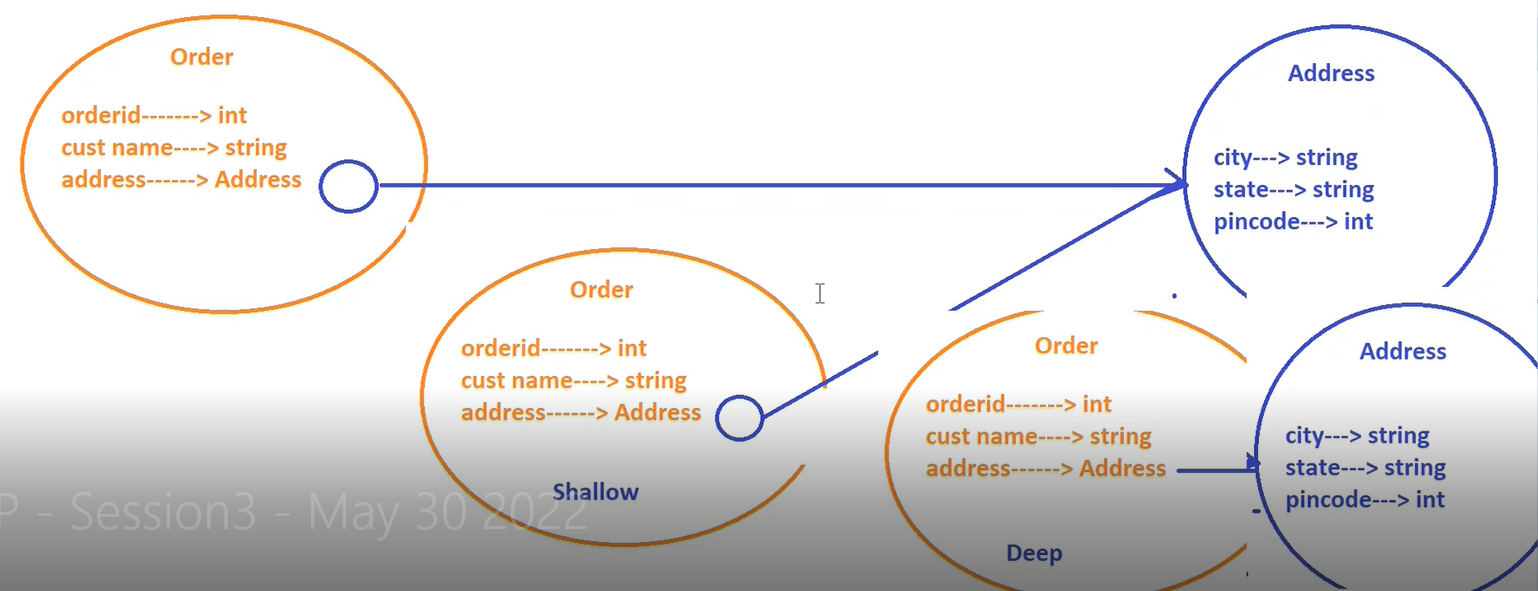


The application will allow user to make changes to this order next time.

Cloning improves the performance. It is easier to keep track of object references rather than keeping track of individual properties of an object.

Used in pure and impure pipes in Angular.





Prototype is template that specifies how cloning will be done. It gives methods which indicate how cloning will be done.

this.MemberwiseClone() does Shallow Cloning.

Prototype pattern does not mandate Shallow or Deep Cloning. It is up to you.

.NET Core/.NET: IClonable interface 🡪 Clone();

Type of cloning logic (Shallow/Deep) can be written inside Clone() method

e. **Builder**

-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Structural Patterns:**

Focus is on relationship between classes.

How objects are composed to form complex structures.

a. Adapter

b. Bridge

c. Composite

d. Decorator

e. Proxy

f. Fly Weight

g. Facade

If you have complex structure that contains number of objects, What will be the relationship between those objects? This is the focus of structural patterns.

a. **Adapter:**

**Objective:** Deal with incompatible interfaces between client and service provider.

Adapter should know about both the interfaces i.e. interface of the service provider and interface of the client.

Converts interface of one class into another which client/service provider is expecting.

Our application may use number of third-party libraries. But interface of third-party libraries may not be compatible with our application.

To use those third-party libraries as it is, we use adapter. Let the adapter handle the compatibility issues of the third-party libraries with the service providers.

**Service Providers:** Sockets

**Applications:** Plugs

Adapter does not perform CRUD operations. Behind the scenes, it is calling the appropriate methods of a class.

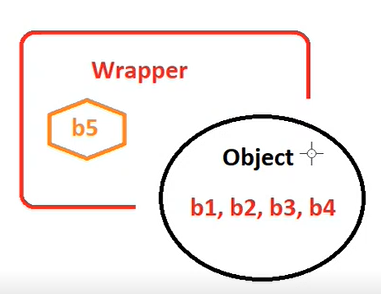
Adapter just acts as a bridge between service provider’s interface and client interface.

**Decorator:**

**Objective:** To add additional behavior to an object at runtime without modifying the code.

Use Decorator Pattern when you want to add Dynamic Behavior to your object.

Also called as Wrapper. When you use the wrapper, wrapper will internally use the object.



Wrapper is like Diwali Lights. Once we remove those lights. The house is still there.

Wrapper makes unit testing much easier. You can make changes to wrapper without modifying the object.

Example of Wrapper: Attributes in .NET. E.g. – [Serializable]

Demo:

Convert() method is calling some external APIs and those external APIs are taking some time to respond. Once the response the received, Convert() method is saying “Currency Conversion Successful”.

Client wants to know how much time Convert() method took to complete. You need to modify the code.

Client may want to add some logging to the Convert() method. You again need to modify the code.

You cannot modify the code every time. That is where Decorator pattern comes into picture.

CurrencyConverterService is like a gift.

DurationDecorator is like a gift wrapper.

Two primary components in decorator:

1) The class which is to be decorated

2) Actual decorator which internally uses the object on which it is internally acting as a wrapper.

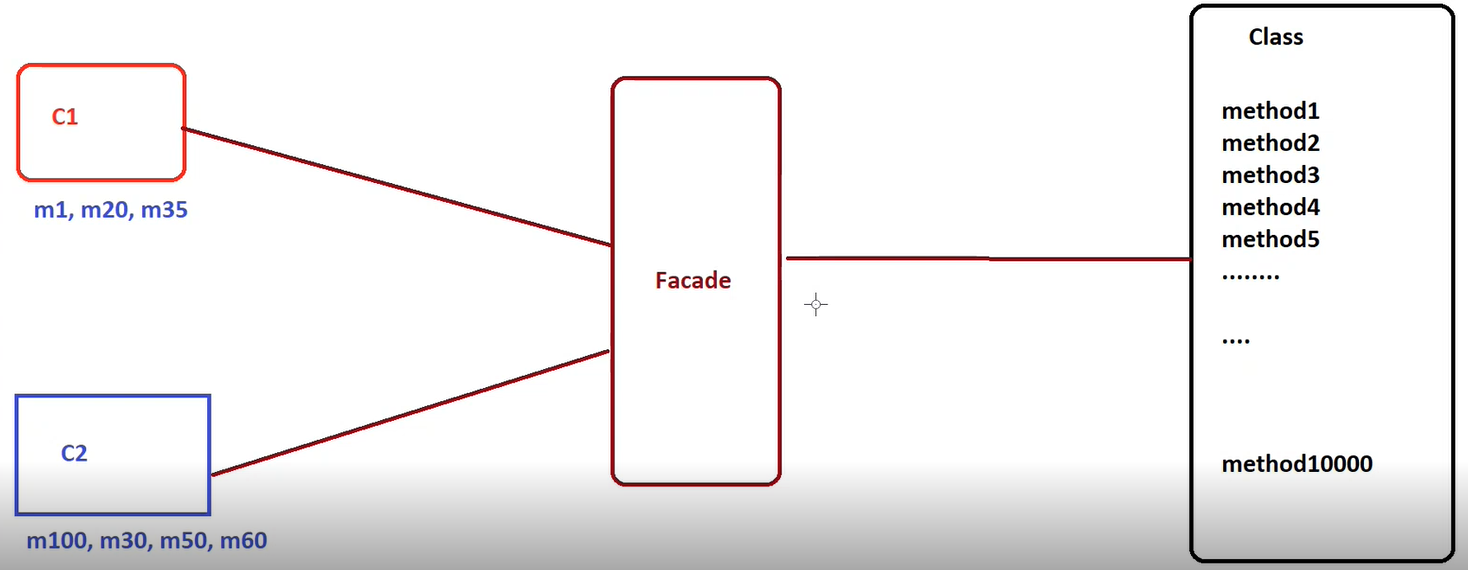
The wrapper will have code that calculates time taken for currency conversion. We will not change currency converter code.

In Decorator pattern, both original class and decorator will implement the same interface. Otherwise there is no way of connecting them together.

.NET uses Attributes, Java uses Data Annotations, Python uses @. All these are examples of decorators.

**Façade:**

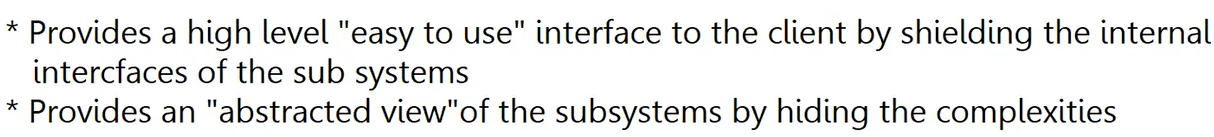




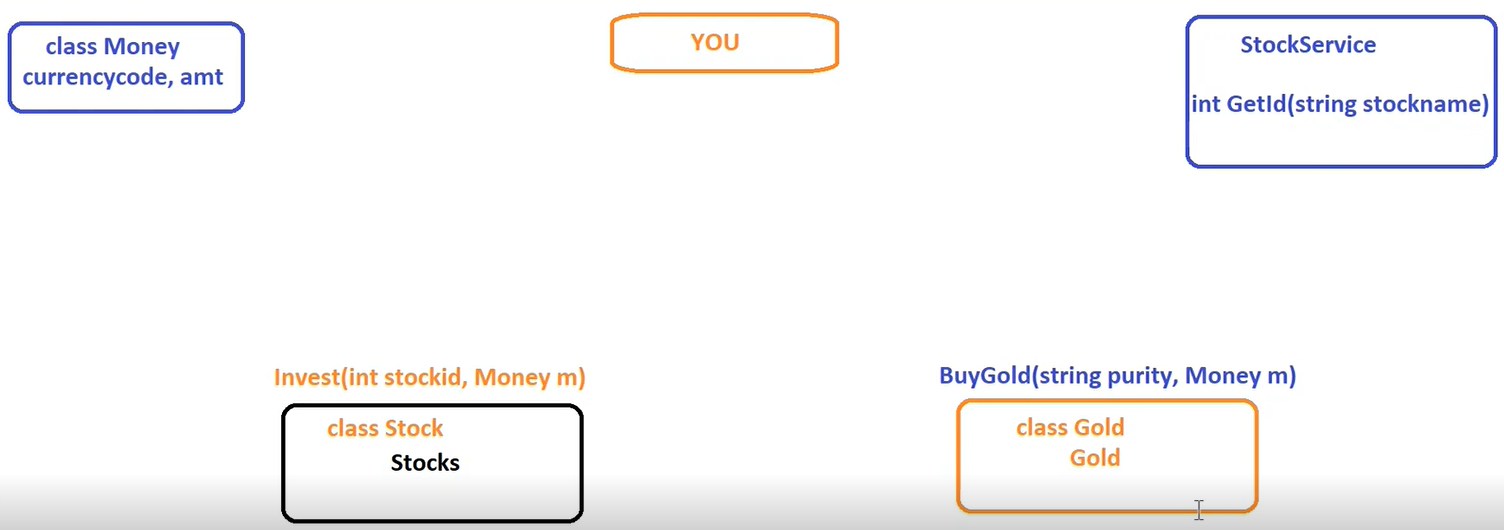
C1 and C2 want to use only specific operations of a system but they need to know the entire system.

Façade will hide the complexities (in this case going through other methods to call a particular method).

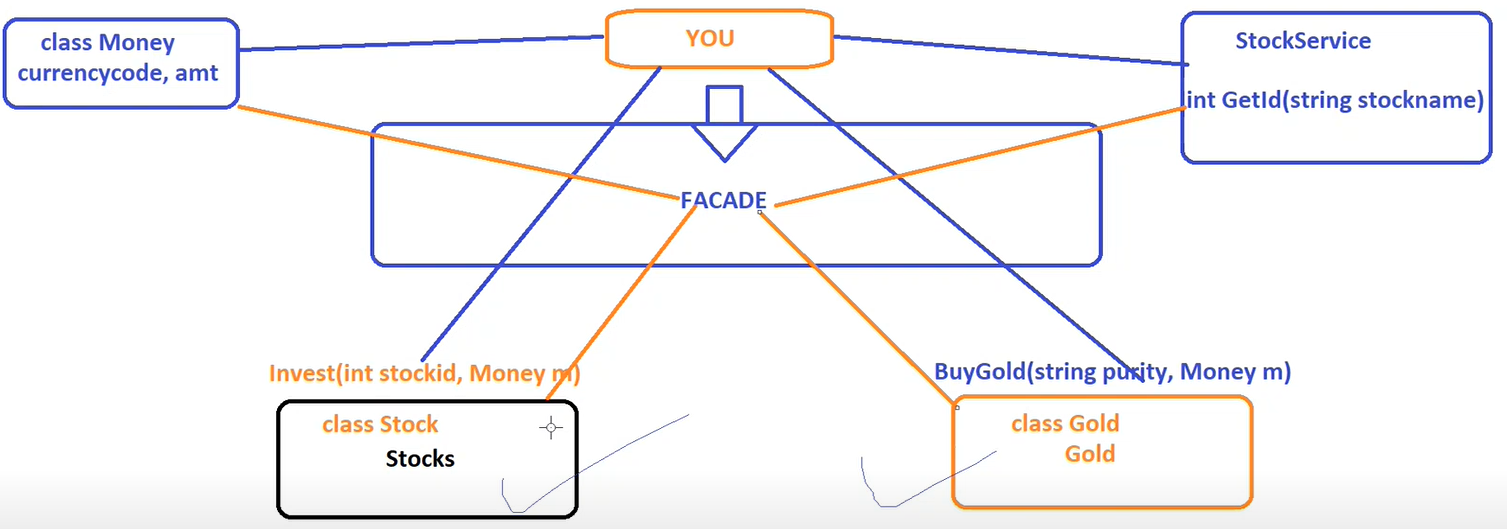
Façade will manage the interactions internally.



**Without Façade:**



**With Façade:**

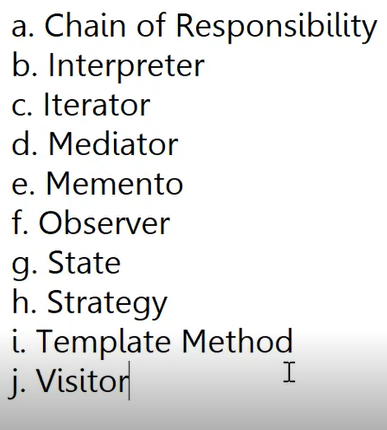


**Behavioral Patterns:**

**Objective:**

How objects communicate with each other.

How objects interact with each other in loosely coupled manner.



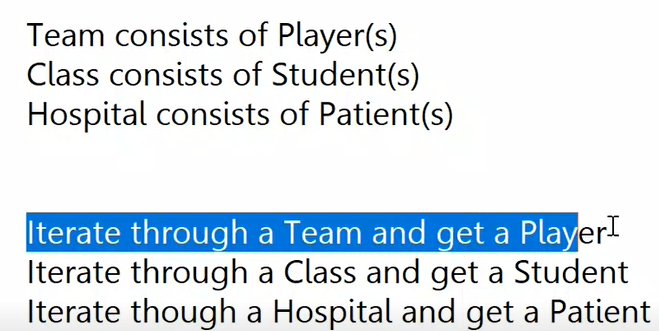
a. **Iterator:**

Allows your code to iterate/access individual elements of an aggregate object sequentially.

Abstracts/hides internals of the aggregate object from the clients.

**Example:** foreach loop can be used to iterate via List, Stack, Queue and Dictionary. Internals of each of these data structure are different.

Foreach loop: IEnumerable (worth iterating) and IEnumerator (actual logic of iteration). IEnumerable is implemented for standardization.



Team, Class, Hospital : Aggregate Object (which contains the collection internally).

Player, Student, Patient : Elements

How internally team stores players is shielded from the client.

Player[] players = new Player[11];

Team India = new Team(players);

foreach(Player p in India)

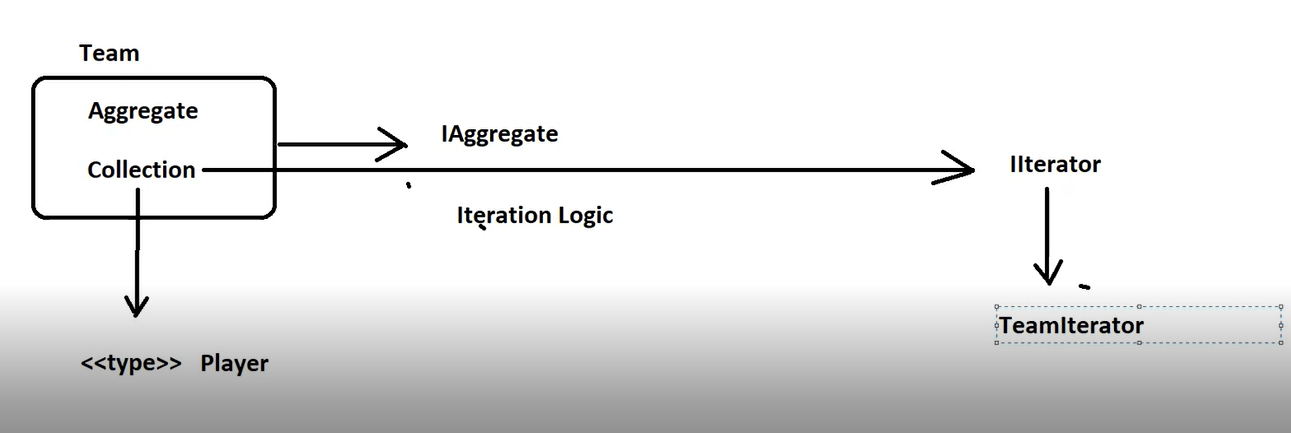
{

Console.WriteLine(p.Name);

}

foreach loop will give an error saying “Player does not implement IEnumerable…”.

This is where Iterator pattern comes into picture. Iterator pattern is useful when you have a class that contains a collection and you need to use that class as if it were a collection.



**Observer Pattern:**

Defines 1 to many dependencies between objects such that when the state of an object changes, all the dependent objects are notified.

Called as “Pub/Sub (Publisher/Subscriber)” pattern.

Frequently used pattern in Asynchronous Communication.

Observable in Angular implements Observer pattern.

.NET implements this pattern using Events and Delegates.

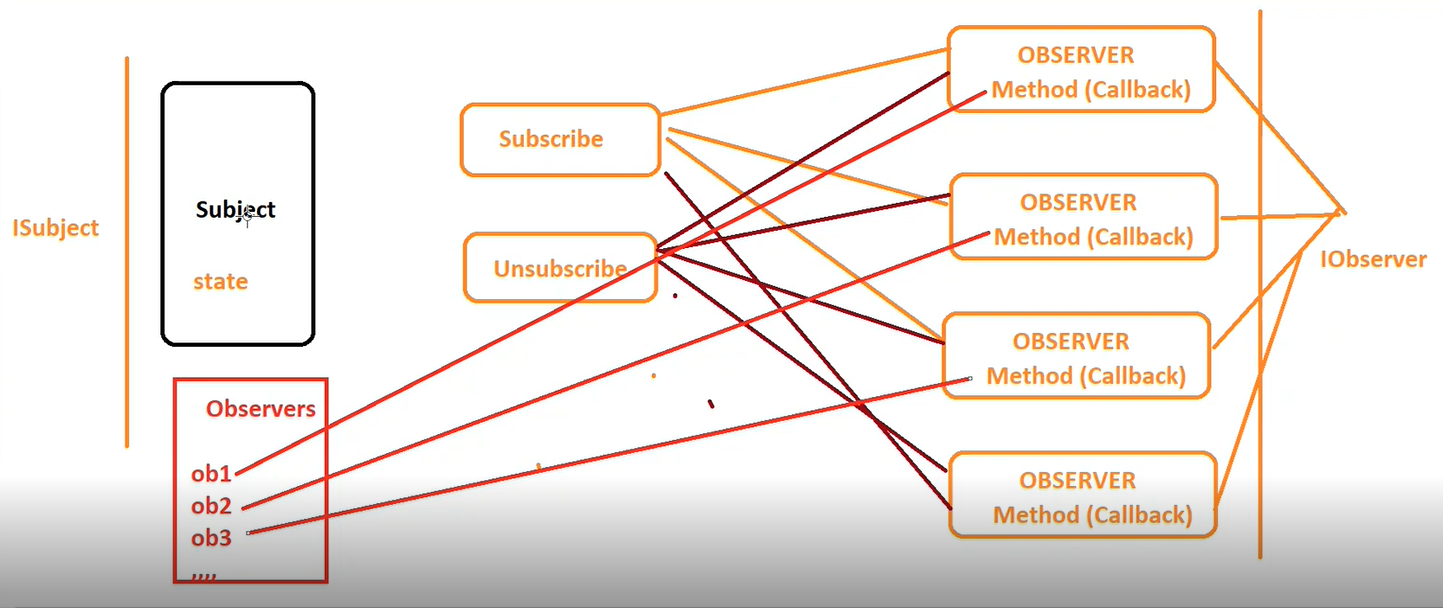
Subject is an object which is capable of raising notifications. Subject is an object whose state changes.

Observers have a method which gets invoked by the Subject when the State changes.

Subjects will provide ways to Observers to receive notifications (Subscribe()).

Subject keeps track of Observers. Subject maintains the list of Observers.

Once the State changes, all the Observers are notified and the Callback() method is called.

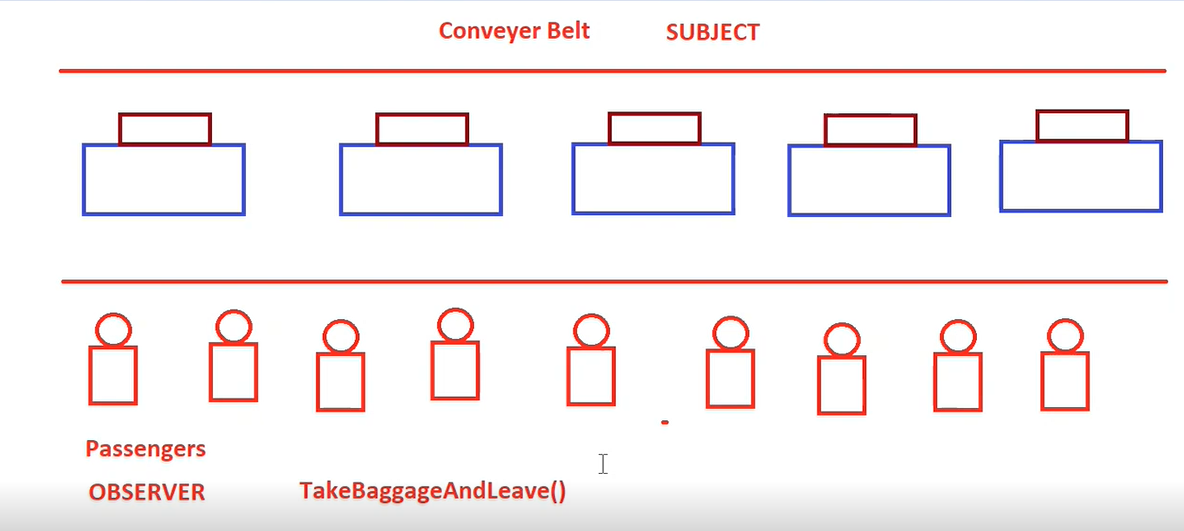


**Benefit:**

Subject need not bother as to who are the Observers.

The Observers have one single point of contact which is Subscribe() to receive notification from that Subject. They don’ have to directly talk to Subject. This is loosely coupled.

**Example:**



Subject may or may not keep track of its observers. That’s where unicasting and multicasting comes into picture.

Observable is Unicast.

Subject is also an Observable. But Subject is Multicast.

Observables requires an Observer. Observer will have certain callbacks. Whatever callbacks observer specifies, the subject will invoke those callbacks whenever change happens.

It will simply iterate over the collection of observers and call their callbacks. Subject needs not know the internal structure of an observer as long as it’s implementing the standardized interface.

**Memento Pattern:**

Sometimes also known a Token Pattern.

Allows you to save state of an object external to the object and restore the state as and when needed.

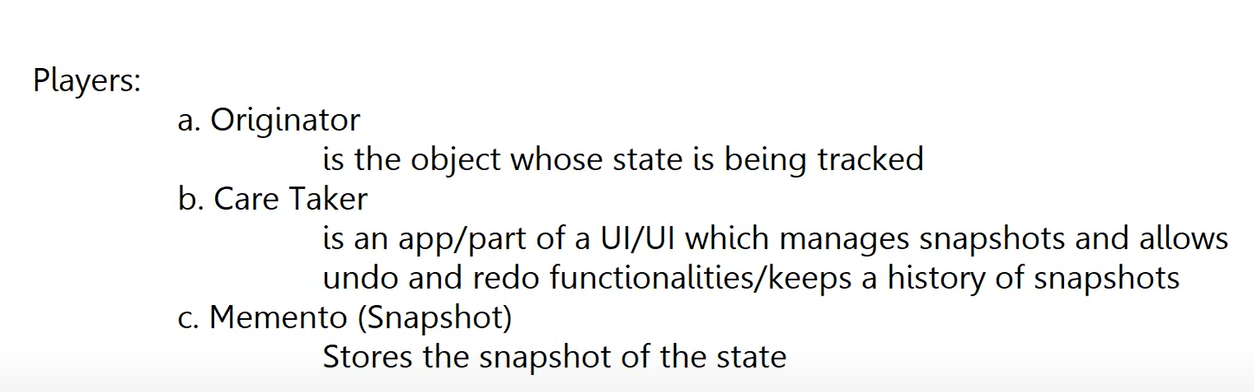
A way to create a snapshot of an object’s state and restore back to a specific snapshot as and when required.

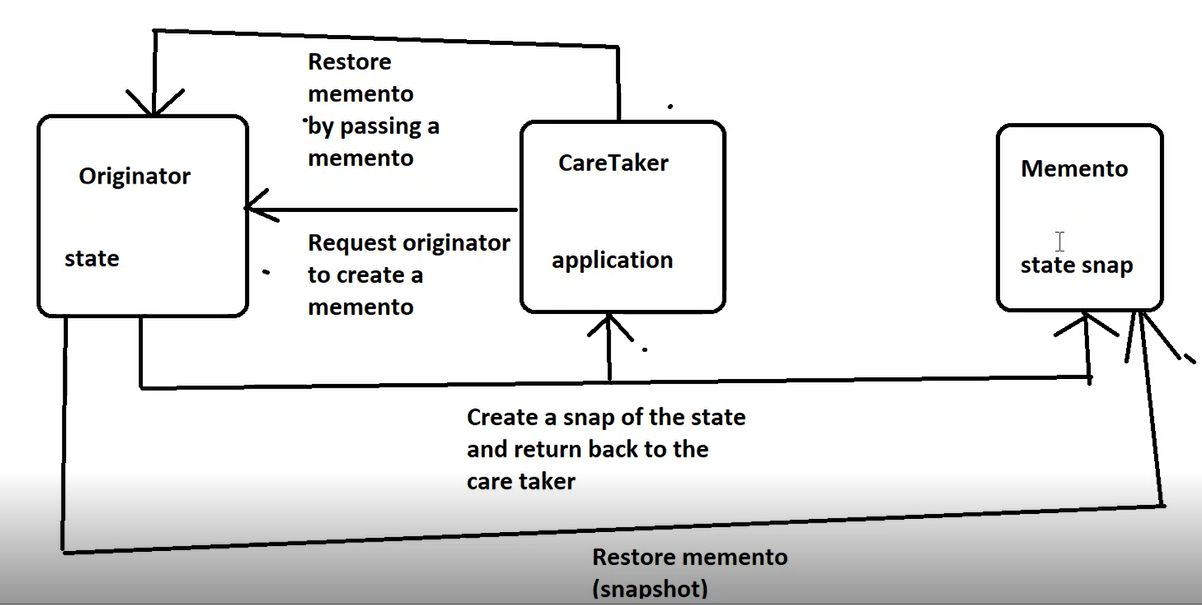
Used in scenarios where you have lot of undo and redo functionality.

E.g. – MS Word keeps track of what you are typing.

Outlook can store emails in Drafts folder which can be modified later.

Checkpoints in games.





Caretaker: Outlook

Caretaker will send the request to Originator to create a Memento.

E.g. – User clicks on Save as Draft button in Outlook.

Originator should have the methods to create a Memento and restore back the Memento.

Outlook will ask email to create a snapshot.

Originator will create a snapshot and return it back to Caretaker.

Caretaker can decide what it wants to do with the state. Outlook can decide whether it should be kept in memory or written to some file.

User makes a request to restore back the saved copy of the email.

Caretaker will send the request to the Originator to restore the Memento by passing the Memento. Caretaker will have list of snapshots.

Now the Originator will get in touch with Memento to restore that snapshot.

This is how the interaction works.

Caretaker cannot directly access the Memento. So, the encapsulation is not broken. So, if this Memento has got some internal state, it is protected from accidental access.

Only the Originator can access the Memento. Originator will access the Memento only when there is a request from Caretaker. So, communication happens but happens in a loosely coupled way.

**Example:** The "Outlook” application is like a “Caretaker”.

The “Snapshot of the Email” is the “Memento”.

The “Email” is the “Originator”.

The Outlook may have methods to create the email, to save the email, to restore back the email.

Behind the scenes, when you say save email as draft, it sends a request to email and email actually creates a memento. The actual state is with the originator.

Originator will take a snapshot of that state and it will create a Memento and return it back to the Caretaker.

So, the snapshot of that email is taken by the object itself and that object is returned back to the caretaker.

**Demo:**

**Email class:** **Originator**

Originator (Email) has got a method SaveEmail(). SaveEmail() returns a snapshot of the email. It creates the instance of EmailMemento and passes the state.

Originator should have a method to create a snapshot and return it back. It’s originator who should decide what snapshot to create. Caretaker should not do that.

Caretaker will request the originator please create the snapshot and give it to me.

RestoreSavedEmail(): Takes a snapshot and restores email from that snapshot.

**EmailMemento: Snapshot of the state.** Memento itself is a type. So, you can decide what values you want to put in a snapshot.

**OutlookApp: Caretaker**

Caretaker has got a stack for Undo/Redo functionality.

It maintains a stack of EmailMemento.

Current Email is the email on which app is working currently.

Outlook has got a method CreateEmail() which returns an email.

Outlook has got a method known as SaveAsDraft() which accepts email to save. Outlook does not save the email. Outlook calls SaveEmail() methods which returns a Memento (snapshot). Outlook is saving that snapshot and putting it inside the stack. So, Outlook is maintaining the stack of all the emails which have been saved as draft. Outlook directly is not taking a snapshot. It is asking the Originator to take a snapshot and then return back that snapshot. Snapshot is saved by Outlook and then it says “Email saved as draft.”

Same way when it is time to restore the email, Outlook has got a method known as RestoreSavedEmail(). RestoredSavedEmail() method of Email class accepts an EmailMemento and that’s what Outlook is passing. It pops the topmost EmailMemento from that stack and that is passed to RestoreSavedEmail() method. RestoreSavedEmail() method populates the current email.

Outlook has got a method CurrentEmail() which displays the current email.

**Program.cs:**

1st, 2nd and 3rd emails are saved as drafts.

SaveAsDraft() will create snapshot of the 1st email (e1) and save it in that stack. Same process is followed for remaining emails.

RestoredSavedEmail() will call RestoredSavedEmail() of Email class passing the latest snapshot. This is like undoing the previous email. This is like a simple undo functionality.