Capgemini

Training

Collaboration. Commitment. Clarity.

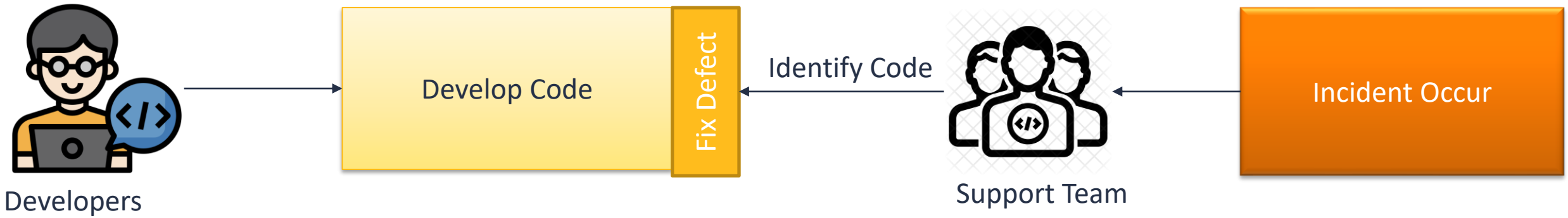
Coding Standard & Best Practices

People matter, results count.

Coding Standard

WHY

A coding standard gives a uniform appearance to the codes written by different engineers. It **improves readability, and maintainability of the code and it reduces complexity also**. It helps in code reuse and helps to detect error easily. It promotes sound programming practices and increases efficiency of the programmers



Best Practices

WHY

- ❑ Variable naming conventions
- ❑ Class and function naming conventions
- ❑ Clear and concise comments
- ❑ Indentations
- ❑ Portability
- ❑ Reusability and scalability
- ❑ Testing

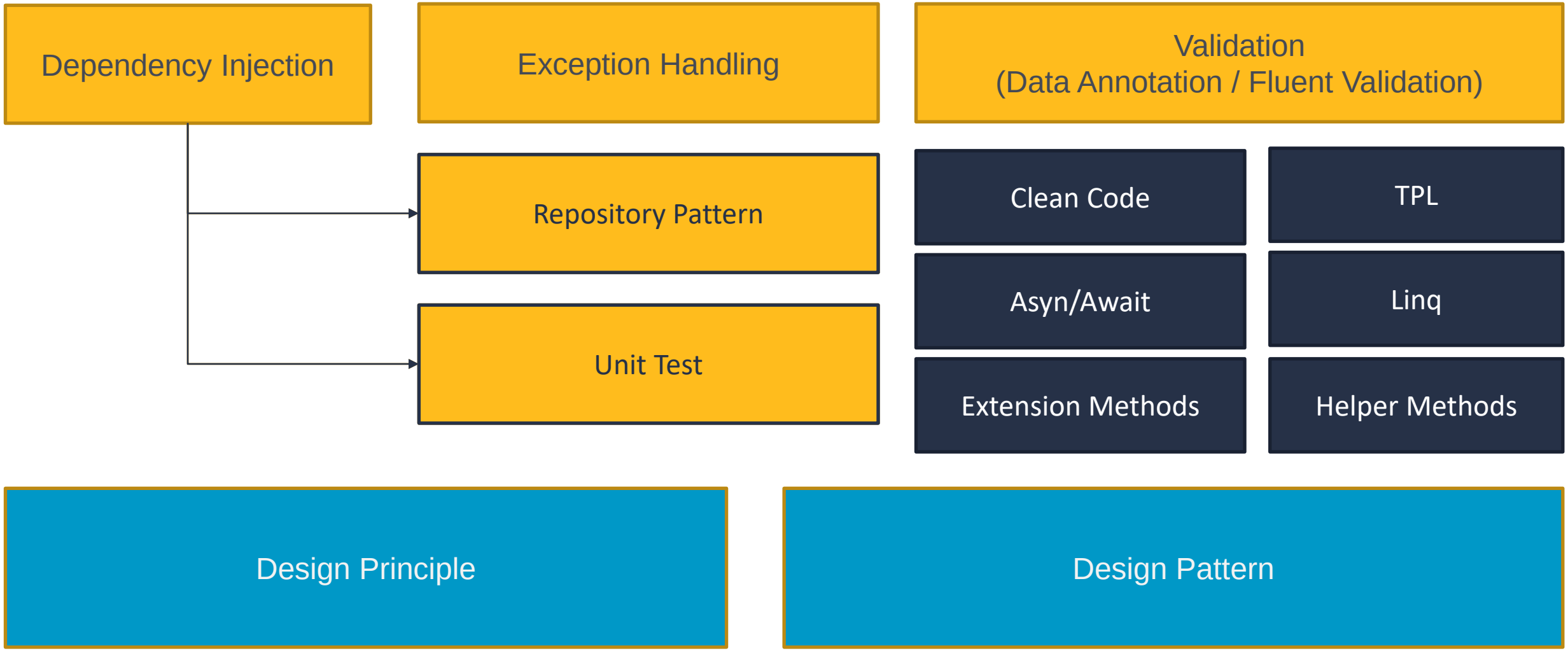
Ref: https://www.dofactory.com/csharp-coding-standards
Ref: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions

Coding Standard

Coding Standard – Good Code

Proper **indentation**	Meaningful **comments**	API **documentation** comments	Proper organization using **namespaces**
Good **naming** conventions	**Classes** that do **one job**	**Methods** that do **one thing**	Proper use of **exceptions**
Code that is **readable**	Code that is **loosely coupled**	High cohesion	Objects are cleanly **disposed** of
The right level of **abstraction**	**Encapsulation** and information hiding	**Object-oriented** code	**Design patterns**

Best Practices



Coding Standard – Example

Proper **indentation**

Bad Code

```
public void DoSomething()
{
for (var i = 0; i < 1000; i++)
{
var productCode = $"PRC000{i}";
//...implementation
}
}
```

Good Code

```
public void DoSomething()
{
   for (var i = 0; i < 1000; i++)
   {
      var productCode = $"PRC000{i}";
      //...implementation
   }
}
```

Coding Standard – Example

Meaningful **comments**

Bad Code

public int _value; // This is used for storing integer values.

...
int value = GetDataValue(); // This sometimes causes a divide by zero error. Don't know why!
...

Good Code

Avoid misleading comment..
Method or variable should be able the meaningful name

// TODO:

developers can be notified and work on it

Coding Standard – Example

API **documentation** comments

Bad Code

Good Code

```
/// <summary>
/// Create a new <see cref="KustoCode"/> instance from the text and globals. Does not perform
/// semantic analysis.
/// </summary>
/// <param name="text">The code text</param>
/// <param name="globals">
///   The globals to use for parsing and semantic analysis. Defaults to <see cref="GlobalState.Default"/>
/// </param>.
 public static KustoCode Parse(string text, GlobalState globals = null) { ... }
```

Coding Standard – Example

Proper organization using **namespaces**

Bad Code

```
namespace MyProject.TextFileMonitor
{
    + public class Program { ... }
    + public class DateTime { ... }
    + public class FileMonitorService { ... }
    + public class Cryptography { ... }
}
```

This can make finding the right code pretty hard or impossible, especially in large code bases

Good Code

Name	Description
CompanyName.IO.FileSystem	The namespace contains classes that define file and directory operations.
CompanyName.Converters	The namespace contains classes for performing various conversion operations.
CompanyName.IO.Streams	The namespace contains types for managing stream input and output.

Coding Standard – Example

Good **naming** conventions

PascalCase

camelCase

PascalCase - Example

Use pascal casing ("PascalCasing") when naming a **class**, **record**, or **struct**.

```
public class DataService
{
}
```

```
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```

```
public struct ValueCoordinate
{
}
```

Ref Link: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions

Coding Standard – Example

Good **naming** conventions

PascalCase

camelCase

PascalCase - Example

When naming public members of types, such as **fields**, **properties**, **events**, **methods**, and local **functions**, use pascal casing.

```
public class ExampleEvents
{
    // A public field, these should be used sparingly
    public bool IsValid;

    // An init-only property
    public IWorkerQueue WorkerQueue { get; init; }

    // An event
    public event Action EventProcessing;

    // Method
    public void StartEventProcessing()
    {
        // Local function
        static int CountQueueItems() => WorkerQueue.Count;
        // ...
    }
}
```

Coding Standard – Example

Good **naming** conventions

PascalCase

camelCase

camelCase - Example

Use camel casing ("camelCasing") when naming **private** or **internal** fields, and prefix them with _

```
public class DataService
{
    private IWorkerQueue _workerQueue;
}
```

Coding Standard – Example

Good **naming** conventions

PascalCase

camelCase

camelCase - Example

When working with **static fields** that are private or internal, use the s_ prefix and for thread static use t_.

```
public class DataService
{
    private static IWorkerQueue s_workerQueue;

    [ThreadStatic]
    private static TimeSpan t_timeSpan;
}
```

Coding Standard – Example

Good **naming** conventions

PascalCase

camelCase

camelCase - Example

When writing **method parameters**, use camel casing.

```
public T SomeMethod<T>(int someNumber, bool
isValid)
{
}
```

Coding Standards – Best Practices

Class Names

use PascalCasing for class names and method names.

```
public class ClientActivity
{
   public void ClearStatistics()
   {
      //...
   }
   public void CalculateStatistics()
   {
      //...
   }
}
```

Variable Names

use camelCasing for local variables and method arguments

```
public class UserLog
{
   public void Add(LogEvent logEvent)
   {
      int itemCount = logEvent.Items.Count;
      // ...
   }
}
```

Constants

use Screaming Caps for constants or readonly variables

```
// Correct
public static const string ShippingType = "DropShip";

// Avoid
public static const string SHIPPINGTYPE = "DropShip";
```

Identifiers

use Hungarian notation or any other type identification in identifiers

```
// Correct
int counter;
string name;

// Avoid
int iCounter;
string strName;
```

Coding Standards – Best Practices

Abbreviations

Exceptions: abbreviations commonly used as names, such as Id, Xml, Ftp, Uri

```
// Correct
UserGroup userGroup;
Assignment employeeAssignment;

// Avoid
UserGroup usrGrp;
Assignment empAssignment;

// Exceptions
CustomerId customerId;
XmlDocument xmlDocument;
FtpHelper ftpHelper;
UriPart uriPart;
```

Type Names

use predefined type names instead of system type names like Int16, Single, UInt64, etc

```
// Correct
string firstName;
int lastIndex;
bool isSaved;
```

```
// Avoid
String firstName;
Int32 lastIndex;
Boolean isSaved;
```

No Underscores

Exception: you can prefix private static variables with an underscore.

```
// Correct
public DateTime clientAppointment;
public TimeSpan timeLeft;

// Avoid
public DateTime client_Appointment;
public TimeSpan time_Left;

// Exception
private DateTime _registrationDate;
```

Noun Class Names

use noun or noun phrases to name a class

```
public class Employee
{
}
public class BusinessLocation
{
}
public class DocumentCollection
{
}
```

Coding Standards – Best Practices

Interfaces

prefix interfaces with the letter I. Interface names are noun (phrases) or adjectives

```
public interface IShape
{
}
public interface IShapeCollection
{
}
public interface IGroupable
{
}
```

Namespaces

organize namespaces with a clearly defined structure

```
// Examples
namespace RMG.SOMS.UI
namespace RMG.SOMS.Backend
namespace RMG.SOMS.Core
namespace RMG.SOMS.Model
namespace RMG.SOMS.Validation
```

File Names

name source files according to their main classes. Exception: file names with partial classes reflect their source or purpose, e.g. designer, generated, etc.

```
// Located in Task.cs
public partial class Task
{
   //…
}
// Located in Task.generated.cs
public partial class Task
{
   //…
}
```

Curly Brackets

vertically align curly brackets.

```
// Correct
class Program
{
   static void Main(string[] args)
   {
   }
}
```

Coding Standards – Best Practices

Member Variables

declare all member variables at the top of a class, with static variables at the very top.

```
// Correct
public class Account
{
    public static string BankName;
    public static decimal Reserves;

    public string Number {get; set;}
    public DateTime DateOpened {get; set;}
    public DateTime DateClosed {get; set;}
    public decimal Balance {get; set;}

    // Constructor
    public Account() {   }
}
```

Enum Types

explicitly specify a type of an enum or values of enums (except bit fields)

```
// Don't
public enum Direction : long
{
    North = 1,
    East = 2,
    South = 3,
    West = 4
}
```

```
// Correct
public enum Direction
{
    North,
    East,
    South,
    West
}
```

Enums

use singular names for enums. Exception: bit field enums.

```
// Correct
public enum Color
{
    Red,
    Green,
    Blue,
    Yellow,
    Magenta,
    Cyan
}
```

```
// Exception
[Flags]
public enum Dockings
{
    None = 0,
    Top = 1,
    Right = 2,
    Bottom = 4,
    Left = 8
}
```

Enum Suffix

suffix enum names with Enum

```
// Don't
public enum CoinEnum
{
    Penny,
    Nickel,
    Dime,
    Quarter,
    Dollar
}
```

```
// Correct
public enum Coin
{
    Penny,
    Nickel,
    Dime,
    Quarter,
    Dollar
}
```

Coding Standard – Example

check for null or empty conditions

Bad Code

```
var employeeName=”testing”;
if(employeeName!=null && employeeName!=””)
{
//..
}
```

Good Code

```
var employeeName=”testing”;
if(!string.IsNullOrEmpty(employeeName))
{
      //..
}
```

object initialization

Bad Code

```
Test test=new Test();
test.id=1;
test.name=”value”;
```

Good Code

```
var test=new Test
{
      Id=1;
      Name=”value”;
};
```

Coding Standard – Example

null conditional operator

Bad Code

```
var employeeName="";
Session["Name"]="test";
If(Session["Name"]!=null)
{
      employeeName=Session["Name"].ToString();
}
else
{
      employeeName="";
}
```

Good Code

```
var employeeName="";
Session["Name"]="test";
employeeName=Session["Name"]?.ToString() ?? ""
```

Avoid extra braces

Bad Code

```
var count=10;
if(count>0)
{
count++;
```

Good Code

```
If(count>0) count++;
```

Coding Standard – Example

string interpolation

Bad Code

```
Test test=new Test();
var details = test.Name + ",you are welcome, Your Id
is " + test.Id + "_emp");
```

Bad Code

```
Test test=new Test();
var details = string.Format("{0}, you are welcome,
Your Id is {1}", test.Name , test.Id + "_emp");
```

Good Code

```
Test test=new Test();
var details = $"{test.Name}, you are welcome, Your Id is
{test.Id}_emp";
```

Coding Standard – Example

switch case

Ok Code

```
int productSwitch = 1;

If(productSwitch == 1)
{
        Console.WriteLine(“Product 1”);
}
else if(productSwitch == 2)
{
        Console.WriteLine(“Product 2”);
}
else
{
        Console.WriteLine(“Product case”);
}
```

Good Code

```
int productSwitch = 1;

switch (productSwitch)
{
        case 1:
                Console.WriteLine(“Product 1”);
                break;
        case 2:
                Console.WriteLine(“Product 2”);
                break;
        default:
                Console.WriteLine(“Product case”);
                break;
}
```

Better Code

```
int productSwitch = 1;

var message = productSwitch switch
{
        1 => Console.WriteLine(“Product 1”),
        2 => Console.WriteLine(“Product 2”)
};
```

Coding Standard – Example

Classes that do **one job**

Bad Code

```
public class DbAndFileManager
{
 #region Database Operations
        public void OpenDatabaseConnection() { throw new
         NotImplementedException(); }
        public void CloseDatabaseConnection() { throw new
         NotImplementedException(); }
 #endregion

 #region File Operations
        public string ReadText(string filename) { throw new
         NotImplementedException(); }
        public void WriteText(string filename, string text) {
        throw new
         NotImplementedException(); }
 #endregion
}
```

Good Code

A good class should only do one job

Single Responsibility Principle

```
public class Student
{
   public int StudentId { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }

   public void Save()
   {
      Logger.Log("Starting Save()");
      _studentRepo.Save(this);
      Logger.Log("End Save()");
   }
}

public class StudentRepository()
{
   Public bool Save(Student std)
   {
      Logger.log("Starting Save()");
      //update existing student to db
      Logger.log("Ending Saving()");
   }
}
```

```
Public class Logger
{
  Public static void Log(string message)
   {
      Console.WriteLine(message);
   }
}
```

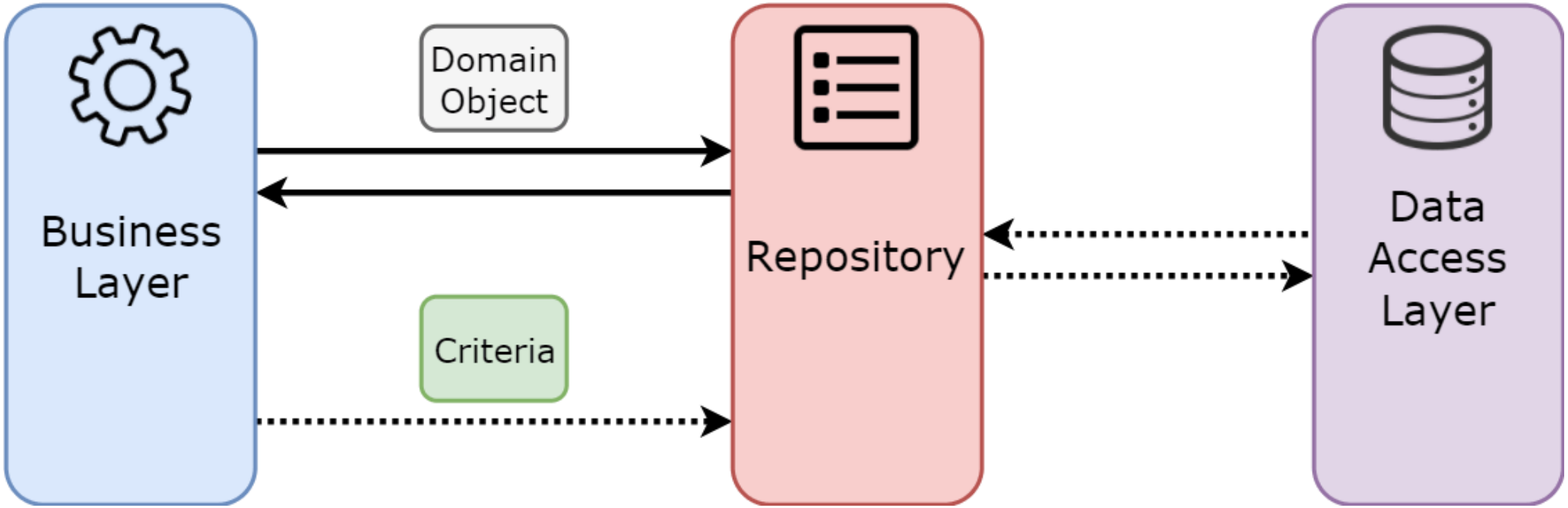
```
Public class EmailManager
{
Public static void SendEmail(string
recEmailed, string senderEmailId, string
subject, string message)
   {
     // smtp code here
   }
}
```

Best Practices

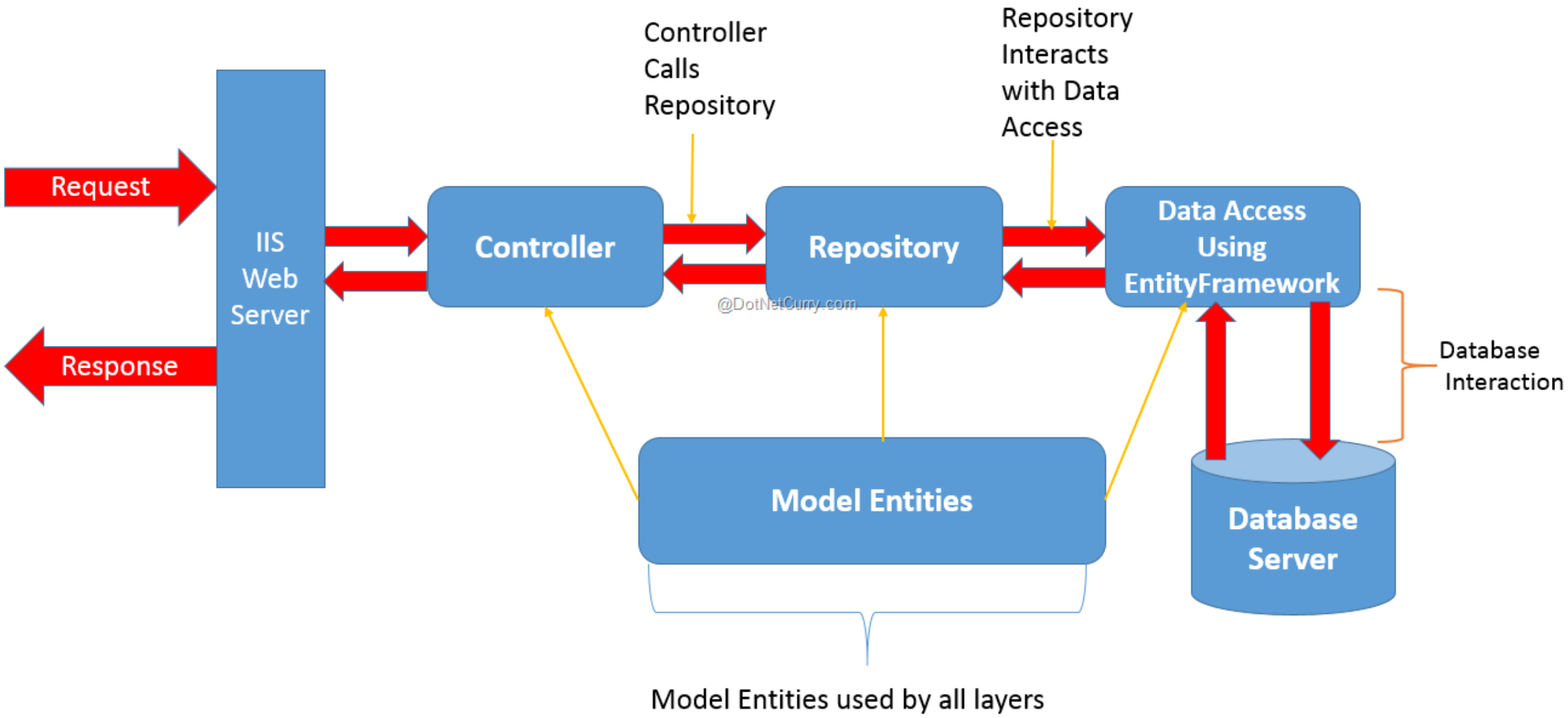
Repository Pattern

What is a Repository Design Pattern?

By definition, the Repository Design Pattern in C# mediates between the domain and the data mapping layers using a collection-like interface for accessing the domain objects. Repository Design Pattern separates the data access logic and maps it to the entities in the business logic. It works with the domain entities and performs data access logic. In the Repository pattern, the domain entities, the data access logic, and the business logic talk to each other using interfaces. It hides the details of data access from the business logic.



Repository Pattern



AutoMapper

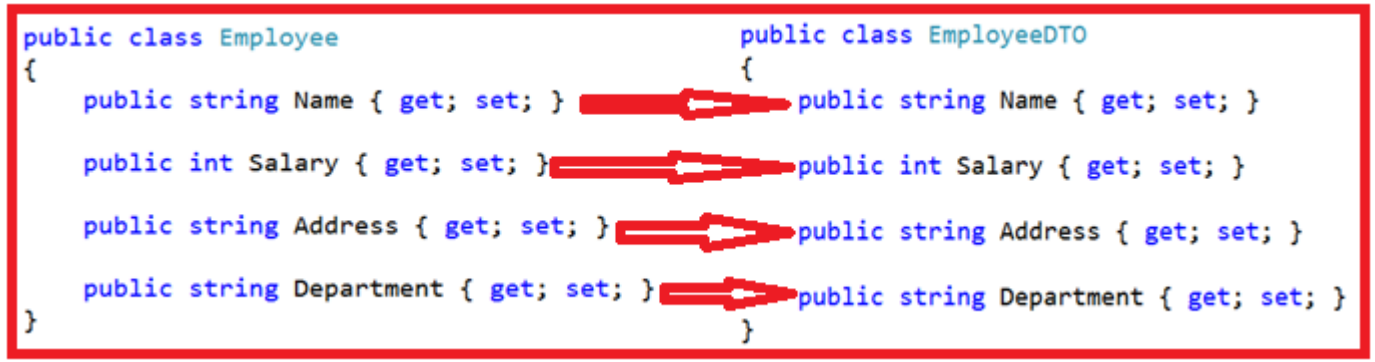
Capgemini Training

AutoMapper in C# is **a library used to map data from one object to another**. It acts as a mapper between two objects and transforms one object type into another. It converts the input object of one type to the output object of another type until the latter type follows or maintains the conventions of AutoMapper.

```
public class Employee
{
    public string Name { get; set; }
    public int Salary { get; set; }
    public string Address { get; set; }
    public string Department { get; set; }
}
```

Trnasfer or Copy

```
public class EmployeeDTO
{
    public string Name { get; set; }
    public int Salary { get; set; }
    public string Address { get; set; }
    public string Department { get; set; }
}
```




```
//Initialize the mapper
var config = new MapperConfiguration(cfg =>
        cfg.CreateMap<TSource, TDestination>()
    );

//Using automapper
var mapper = new Mapper(config);
var empDTO1 = mapper.Map<TDestination>(TSourceObject);
//OR
var empDTO2 = mapper.Map<TSource, TDestination>(TSourceObject);
```

Ref: https://dotnettutorials.net/lesson/automapper-in-c-sharp/

Exception Handling

Exceptions allow an application to transfer control from one part of the code to another. When an exception is thrown, the current flow of the code is interrupted and handed back to a parent try catch block. C# exception handling is done with the follow keywords: try, catch, finally, and throw

try – A try block is used to encapsulate a region of code. If any code throws an exception within that try block, the exception will be handled by the corresponding catch.
catch – When an exception occurs, the Catch block of code is executed. This is where you are able to handle the exception, log it, or ignore it.
finally – The finally block allows you to execute certain code if an exception is thrown or not. For example, disposing of an object that must be disposed of.
throw – The throw keyword is used to actually create a new exception that is the bubbled up to a try catch finally block.

Ref: https://dev.to/bytehide/5-good-practices-for-error-handling-in-c-4391

Exception Handling - Example


```
WebClient wc = null;
try
{
  wc = new WebClient(); //downloading a web page
  var resultData = wc.DownloadString("http://google.com");
}
catch (ArgumentNullException ex)
{
  //code specifically for a ArgumentNullException
}
catch (WebException ex)
{
  //code specifically for a WebException
}
catch (Exception ex)
{
  //code for any other type of exception
}
finally
{
  //call this if exception occurs or not
  //in this example, dispose the WebClient
  wc?.Dispose();
}
```

```
try
{
  //do something
}
catch (Exception ex)
{
  //LOG IT!!!
  Log.Error(string.Format("Excellent description goes here
about the exception. Happened for client {0}",
_clientContext.ClientId), ex);
  throw; //can rethrow the error to allow it to bubble up,
or not, and ignore it.
}
```

Exception Handling - Example

Bad Code

```
try
{
   // Do something..
}
catch (Exception ex)
{
   // Any action something like roll-back or logging etc.
   throw ex;
}
```

Good Code

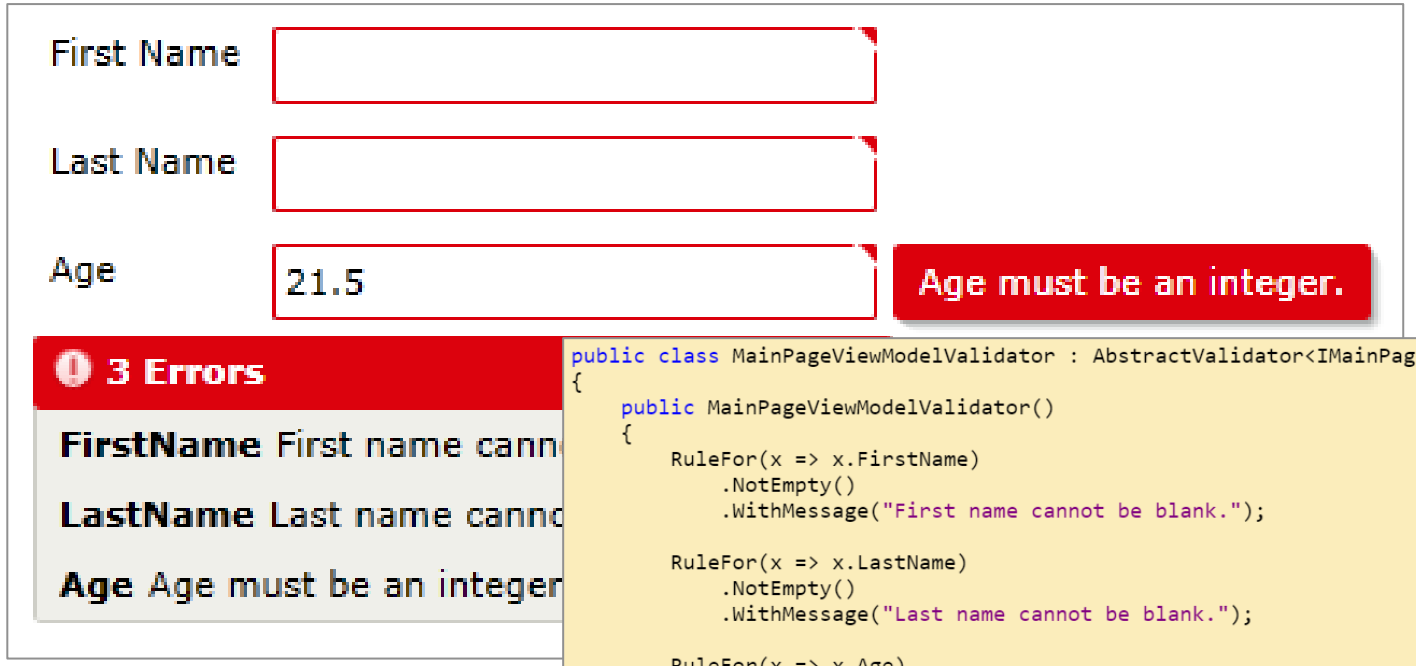
```
try
{
   // Do something..
}
catch (Exception ex)
{
   // Any action something like roll-back or logging etc.
   throw;
}
```

We can see in the good way to do it I have simply used throw. In this way, the **original** exception stack would be **conserved**. Otherwise, with throw ex, it would be overwritten with the line of code where this statement was called.

Fluent Validation

Capgemini Training

Validation is a process to validate and check the data inserted by the user in the view. ASP.NET MVC provides various mechanisms for the validation like Remote Validation, Validation using Data Annotations, Fluent Validation and Custom Validation. In this article, we will read about Fluent Validation. Fluent Validation contains .NET libraries and the validation is performed using the Lambda expression. Use Fluent Validation when you want to create some advanced and complex validation for the user data. Let's start this session.



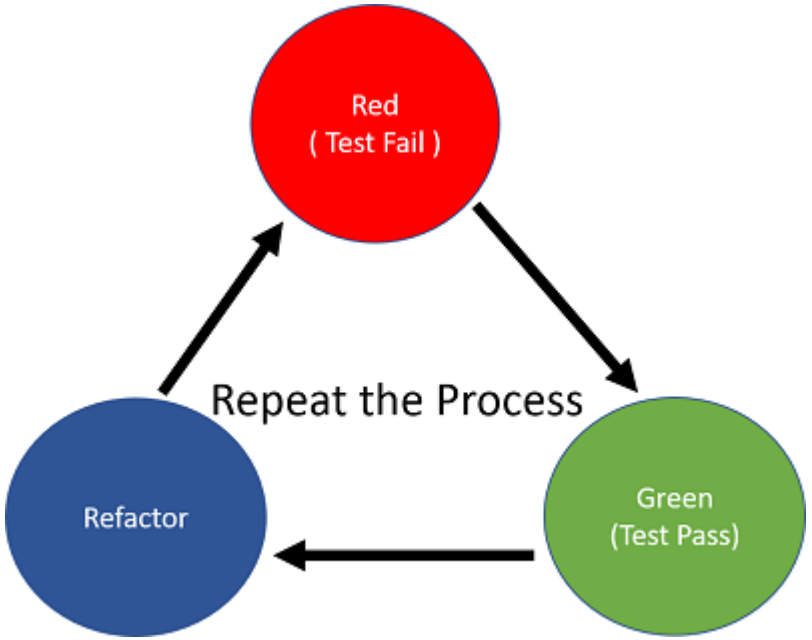

```
public class MainPageViewModelValidator : AbstractValidator<IMainPageViewModel>
{
    public MainPageViewModelValidator()
    {
        RuleFor(x => x.FirstName)
            .NotEmpty()
            .WithMessage("First name cannot be blank.");

        RuleFor(x => x.LastName)
            .NotEmpty()
            .WithMessage("Last name cannot be blank.");

        RuleFor(x => x.Age)
            .NotEmpty()
            .WithMessage("Age is required.");

        RuleFor(x => x.Age)
            .Must(a => a.IsInt32())
            .When(x => !x.Age.IsNullOrEmpty())
            .WithMessage("Age must be an integer.");
    }
}
```

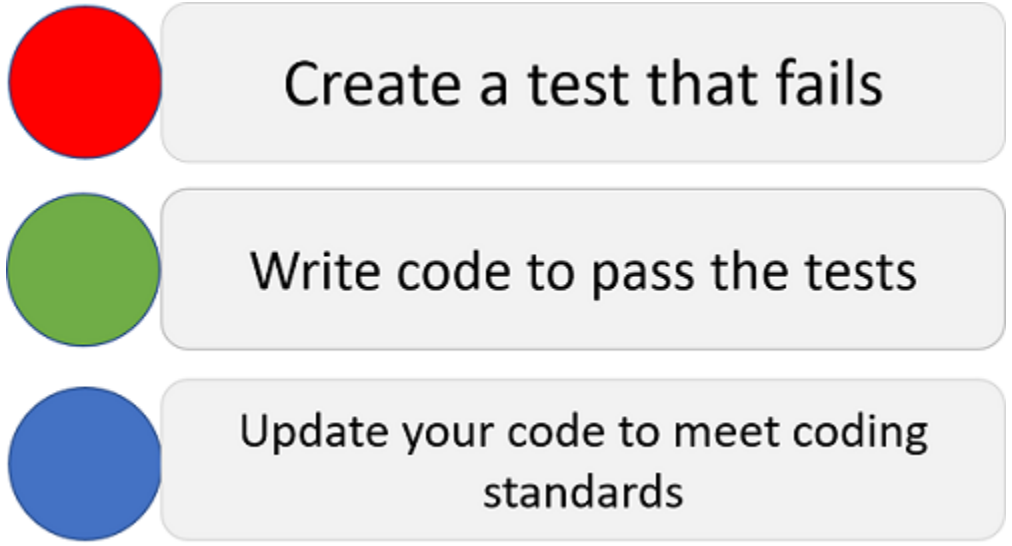
Data Annotation:

```
public class SampleModel : BaseModel
{
    [Required]
    [Range(1, 100)]
    public int? Id { get; set; }

    [Required]
    [StringLength(10)]
    [RegularExpression("w+")]
    public string Name { get; set; }

    [Required]
    [StringLength(500, MinimumLength = 10)]
    public string Description { get; set; }
```

Unit Test

Unit Tests basically test individual parts (also called as Unit) of code (mostly methods) and make it work as **expected** by programmer. A Unit Test is a code written by any programmer which test small pieces of functionality of big programs. Performing unit tests is always designed to be simple, A "UNIT" in this sense is the smallest component of the large code part that makes sense to test, mainly a method out of many methods of some class. Generally the tests cases are written in the form of functions that will evaluate and determine whether a returned value after performing Unit Test is equals to the value you were expecting when you wrote the function.





Unit Test - Example


```
namespace Calculator.Test
{
   [TestClass]
   public class CalculatorTest
   {
      [TestMethod]
      public void Test_Divide()
      {
         // Arrange
         int expected = 10;
         int numerator = 100;
         int denominator = 10;

         // Act
         int actual=  Calculators.Divides.divide(numerator, denominator);

         // Asset
         Assert.AreEqual(expected, actual);
      }
   }
}
```

```
[TestMethod]
    [ExpectedException(typeof (DivideByZeroException))]
    public void Divide_DenominatorIsZero_ThrowDivideByZeroException()
    {
      // Arrange

      int numerator = 100;
      int denominator = 0;
      // Act
      try
      {

     Calculators.Divides.divide(numerator, denominator);
      }catch(Exception ex)
      {
         Assert.AreEqual("denominator is zero", ex.Message);
         throw;
      }

    }
```

Best Practices (Design Principles)

Design Principles (SOLID)

SOLID

DRY

KISS

- In object-oriented programming, **SOLID** is an acronym for the five design principles introduced by Robert C. Martin. These principles are used to design software applications maintainable and testable.

Ref: https://www.tutorialsteacher.com/csharp/solid-principles

Design Principles (DRY)

SOLID | DRY | KISS

- The DRY principle states that every piece of knowledge must have a single, unambiguous, authoritative representation within a system.
- The implication from this principle is often deemed to be avoiding code duplication.

Ref: https://enterprisecraftsmanship.com/posts/dry-revisited/#:~:text=The%20DRY%20principle%20states%20that,to%20be%20avoiding%20code%20duplication.
https://www.c-sharpcorner.com/article/software-design-principles-dry-kiss-yagni/

Design Principles (DRY) - Example


```
public class Product
{
   /* Other members */
   public string Name { get; set; }

   public override string ToString()
   {
      return Name;
   }
}

public class Customer
{
   /* Other members */
   public string Name { get; set; }

   public override string ToString()
   {
      return Name;
   }
}
```

Possible Solution

```
public class NamedEntity
{
   public string Name { get; set; }

   public override string ToString()
   {
      return Name;
   }
}

public class Product : NamedEntity
{
   /* Other members */
}

public class Customer : NamedEntity
{
   /* Other members */
}
```

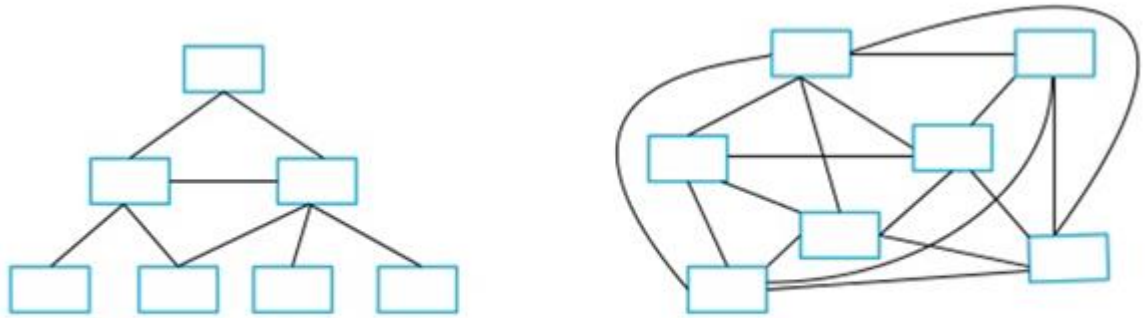
Design Principles (DRY)

SOLID

DRY

KISS

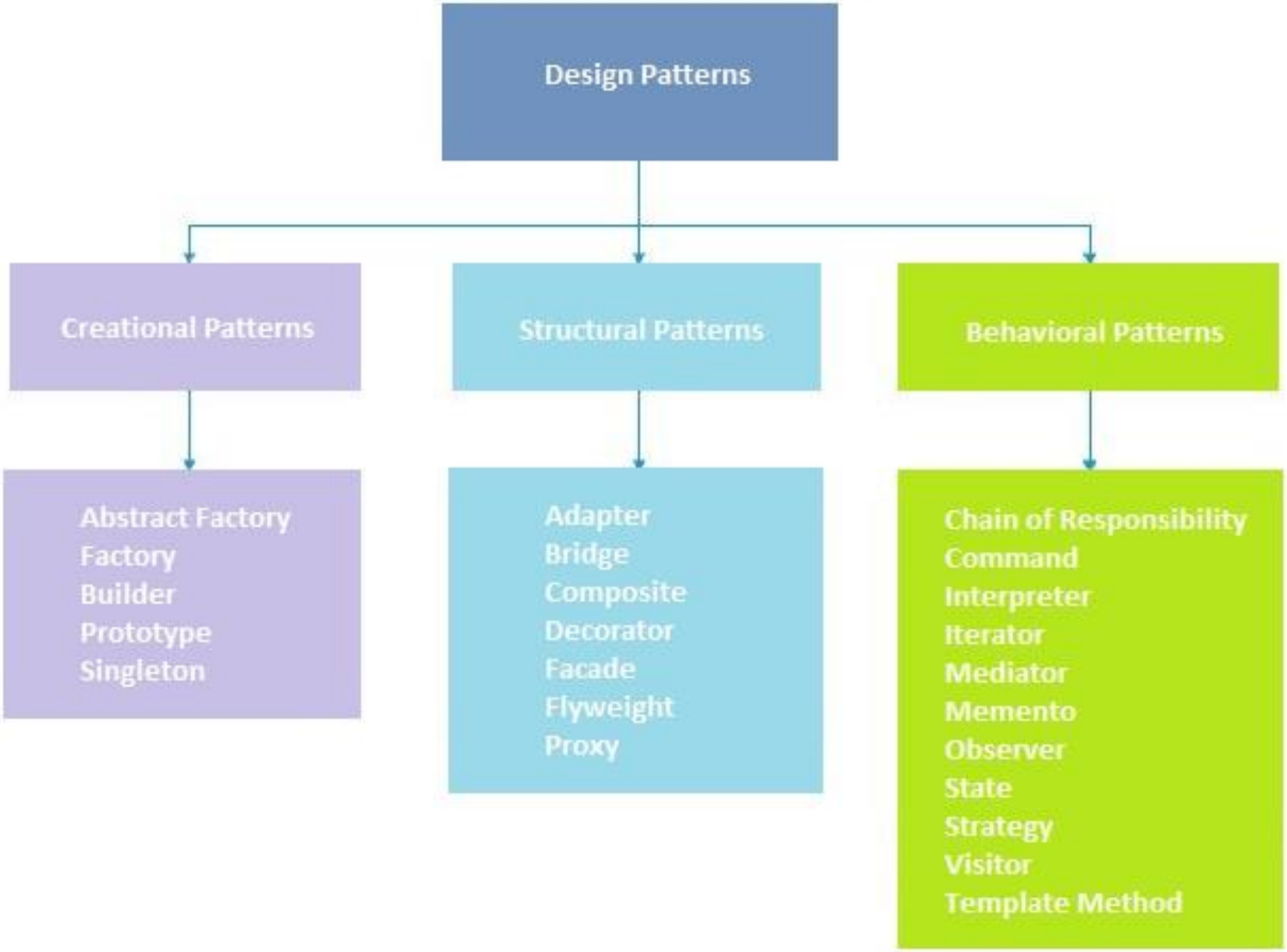
- As described in the heading, KISS design pattern here stands for KEEP IT SIMPLE STUPID.
- This principle simply indicates that the simplest solution or path should be taken in a situation. This principle can be applied to any scenario, including many business activities, such as planning, management, and development.



Try to keep the process as simple as possible

Best Practices (Design Pattern)

Design Pattern



Design patterns provide general solutions or a flexible way to solve common design problems. This article provides an introduction of design patterns and how design patterns are implemented in C# and .NET.

Reference Link

Reference Link

- https://learn.microsoft.com/en-us/aspnet/core/fundamentals/best-practices?view=aspnetcore-7.0
- https://github.com/justinamiller/Coding-Standards
- https://github.com/thangchung/clean-code-dotnet
- https://github.com/ryanmcdermott/clean-code-javascript
- https://github.com/brminnick/AsyncAwaitBestPractices
- https://github.com/ktutak1337/Clean-Architecture-Template
- https://www.dofactory.com/csharp-coding-standards
- https://www.tutorialsteacher.com/csharp/single-responsibility-principle
- https://dev.to/bytehide/5-good-practices-for-error-handling-in-c-4391
- https://www.c-sharpcorner.com/article/c-sharp-string-object-impact-on-performance/
- https://www.c-sharpcorner.com/article/a-basic-introduction-of-unit-test-for-beginners/
- https://app.pluralsight.com/course-player?clipId=2eafe99a-6c15-490a-83ae-763ccc5c7c6e

About Capgemini

Capgemini is a global leader in partnering with companies to transform and manage their business by harnessing the power of technology. The Group is guided everyday by its purpose of unleashing human energy through technology for an inclusive and sustainable future. It is a responsible and diverse organization of 270,000 team members in nearly 50 countries. With its strong 50 year heritage and deep industry expertise, Capgemini is trusted by its clients to address the entire breadth of their business needs, from strategy and design to operations, fuelled by the fast evolving and innovative world of cloud, data, AI, connectivity, software, digital engineering and platforms. The Group reported in 2020 global revenues of €16 billion.

Get the Future You Want | www.capgemini.com