Application Architecture - Grab Fried Onion Rings And Throw Spear Into Onion Architecture And Domain Driven Design

HR Rony

5 May 2017 CPOL

Rate this: ★★★★★ 5.00 (18 votes)

This will cover how to use Domain Driven Design in your application according to the Onion Architecture. There are short descriptions about architecture Category / Style, N-Layer / N-Tier Architecture, Template Method Pattern and Facade Design Pattern.

Download project.zip - 50.6 KB

Software Architecture

Readers of the Software Testing Topics - Application Architect, Developers

What Will You Know From the Topics

I know that you already know N-Layer, N-Tier architecture. I will show you how to implement Onion Architecture using Domain Derive Design. Anyway, the following concepts will be covered in this topic:

- Architecture Category

- Architecture Style
- N-Layer Architecture
- N-Tier Architecture
- Onion Architecture
- Domain Driven Design
- Template Method Pattern
- Facade Design Pattern

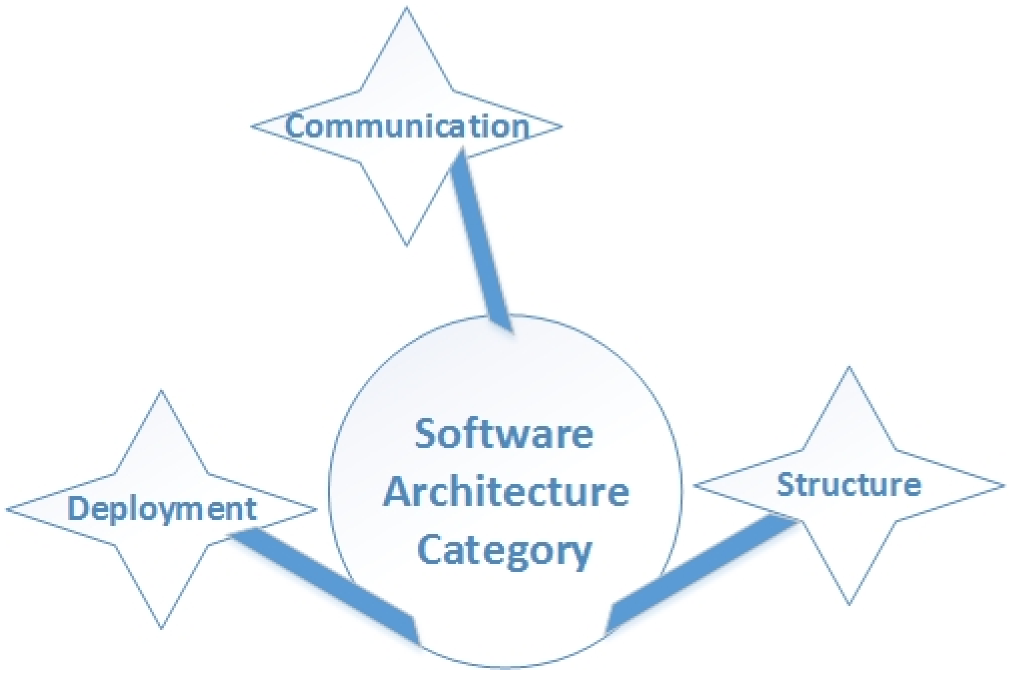
Let's Drilldown Basic Architecture Concept

Suppose you have finished a project; but, have you ever asked yourself what architectural style and design you have used for your components or application? The big difference between average work and excellent work is not only using the latest version of the technology but also choosing the right architectural style according to your requirements and scope. Architecture is the creation of a better software world. I know you already know that; so, no more words. Time to overview only few important terms first.

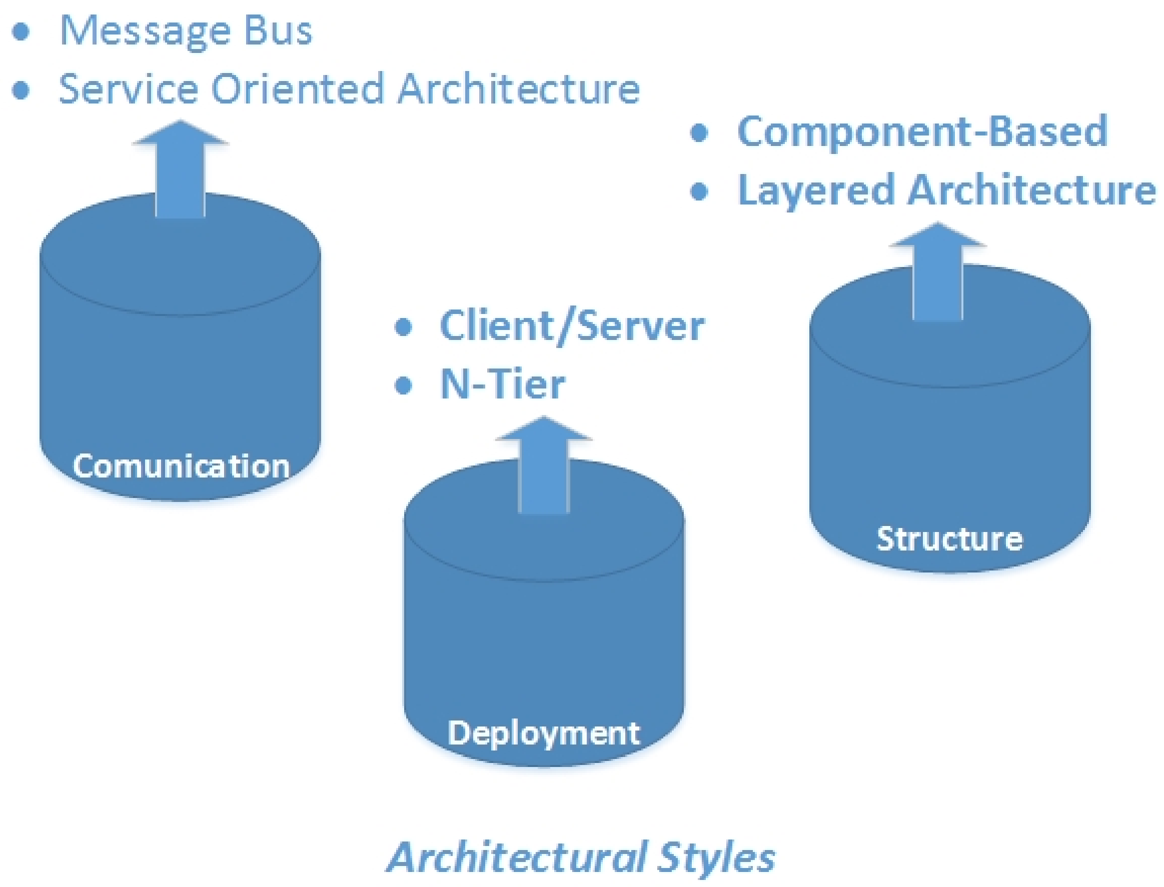
Architecture VS. Design

- **Architecture**: All architecture is design; it is a plan for the structure of the component.
- **Design**: All design is not architecture; it uses the architecture.

Architecture Category



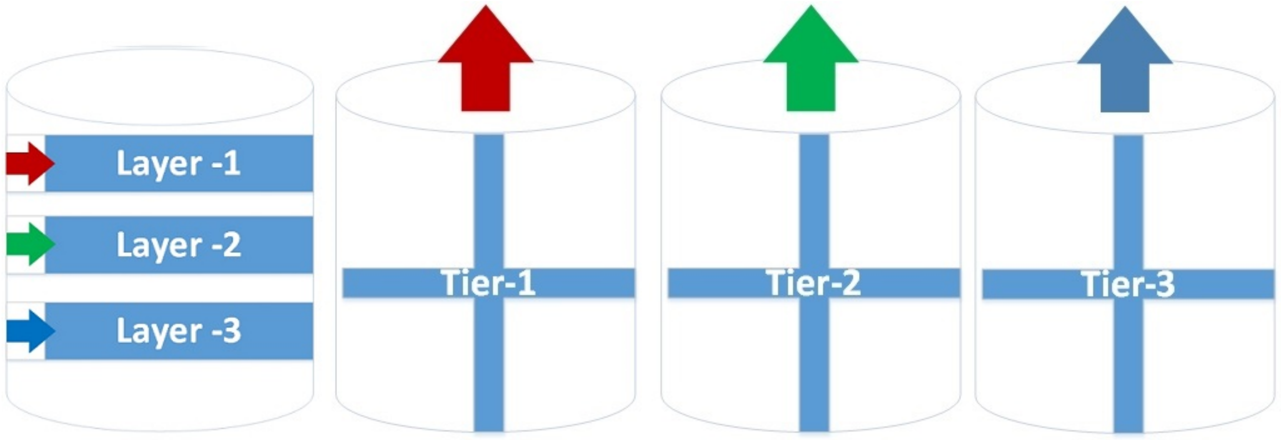
Architectural Styles



Basic Terms Of Architecture

N-Tier VS. N-Layer

- Tier indicates the physical separation of components. It could be different assemblies like DLL, EXE, etc. on the multiple servers or same server.
- Layers indicate the logical separation of the components such as Data-Layer, Business-Layer, Presentation-Layer, etc.



Component-Based Vs. Tier/Layer Architectural Styles

Component-based" means "separation of concerns", whether those are:

- Tire-Component means separation/deployment
- Layer-Component means logical separation

Application Type

- Mobile Application
- Rich Client Application
- Rich Internet Application
- Service
- Web Application
- Hosted & Cloud Services
- Office Business Application
- Share Point Line Of Business Application

Key Points Before Choosing Application Architecture

This is the guide to get you started. Always, you have to consider these points like below:

- Application Type
- Deployment Strategy
- Appropriate Technologies
- Quality Attributes
- Cross-Cutting Concerns

Story Of Fried Onion Rings

I have very minimal knowledge about cooking. Most of the time, I buy food from the restaurant. Few months ago, I moved from one state to another state because of my job. One day, I was hanging out with one of my colleagues whose name was Christopher Robin. We had a long conversation. That's how he knew that I have very little knowledge about cooking.

So, once he offered to teach me - how to fry Onion Rings. Now I want to share the technique with you that I have learned from my colleague. So that you can start frying the onion-rings easily. Before start, you need to have an initial knowledge about:

- Onion Architecture
- Domain Driven Design(DDD)

Even-if you don't know that, then rock-a-bye, bye; don't worry about it.

How To Fry Onion Rings

Application Design

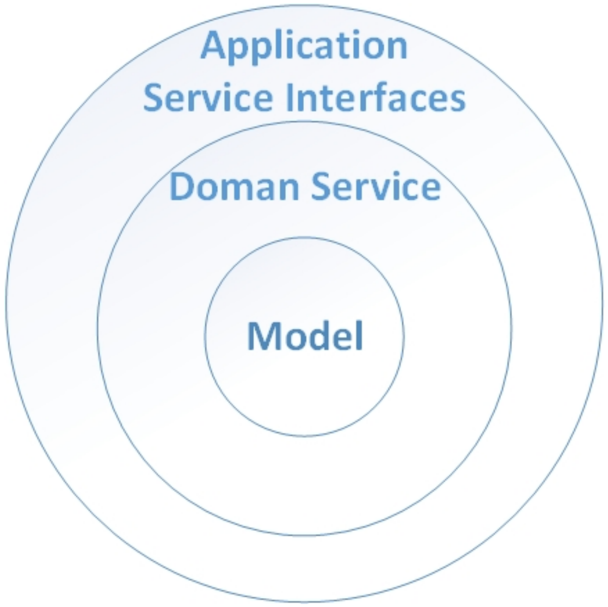
Although, in this design, I'm following the Domain Driven Design, I'm avoiding complexity just to keep it simple.

Domain Driven Design (DDD)

Focus Points Domain Core

- Domain Models
- Domain Services
- Application Services

Domain Core into the Onion Architecture

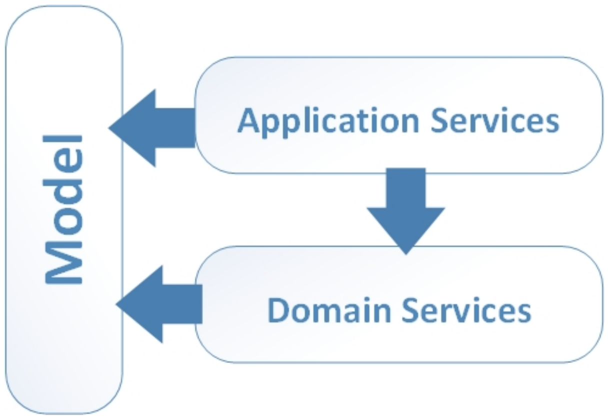


In this diagram, each of the circles of the onion means a layer. So, according to the Onion Architecture, your domain model layer (say, entity layer) should be placed into the core or center of the onion.

The very next layer of the onion core is the domain service layer. In this layer you can put all of the domain related logic services.

The outer layer of this diagram is known as the application services. In this layer we have to keep all of the service interface, abstract class and related classes so that other components (say, Data Access Layer) can implement their functionalities.

Domain Core In DDD



In this DDD diagram, the arrow symbol indicates the communication or the dependency of the layer. Application services and domain services are both have the dependencies to domain model. On the other hand, application services may use the domain services. These are known as the domain core.

Moral Points:

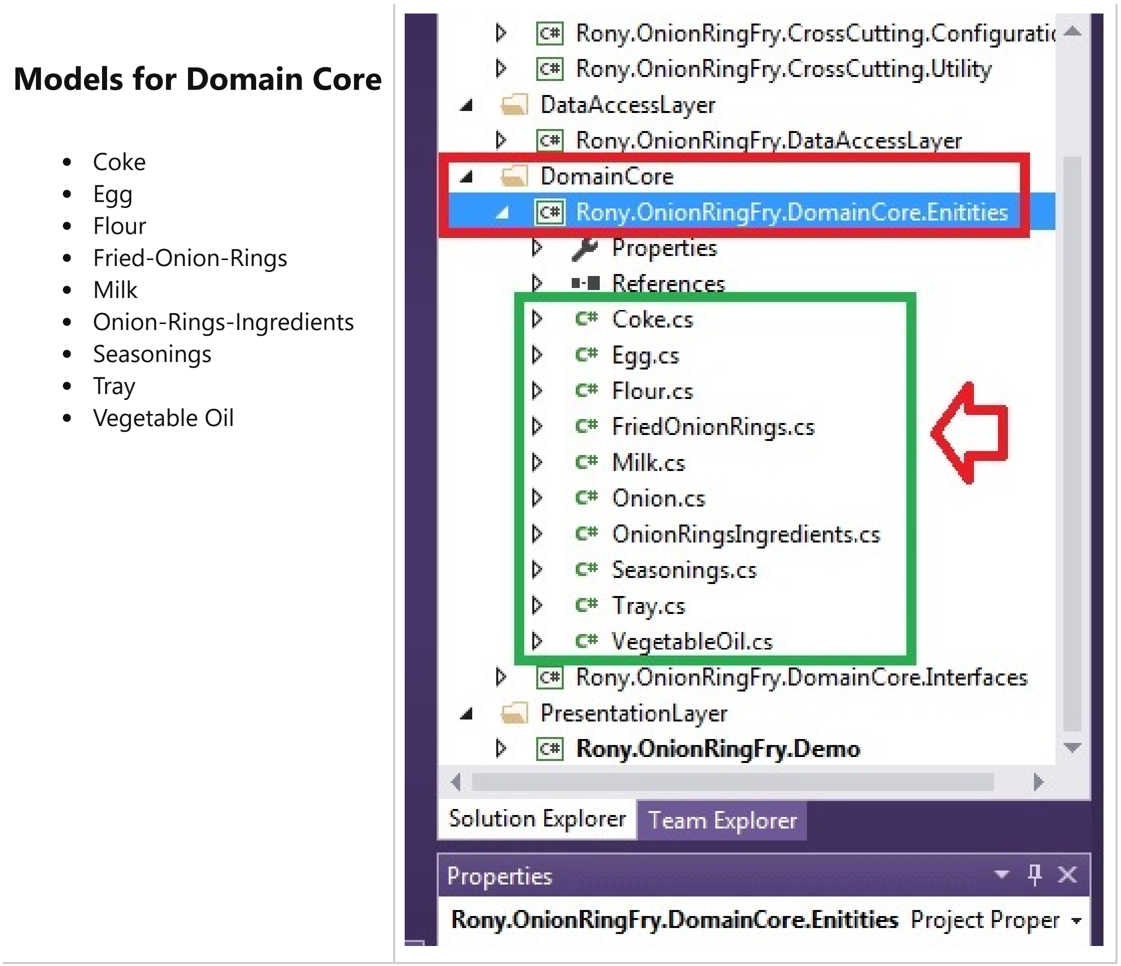
Architecture styles is telling you where you can place your application layers; but I'm not getting the direct answer from it that how I will implement the application. I mean how I will design my application.

Therefore, DDD will guide me about the design and how the layers of the application will communicate with each other's.

Designing Domain Core Of DDD

Identify And Create Domain Model

This central layer of the onion architecture defines the domain model and all of the outer layers will use these domain models. So, identify the following models for fried onion rings:



Models for Domain Core

- Coke
- Egg
- Flour
- Fried-Onion-Rings
- Milk
- Onion-Rings-Ingredients
- Seasonings
- Tray
- Vegetable Oil

```
  ▷ Rony.OnionRingFry.CrossCutting.Configurati
  ▷ Rony.OnionRingFry.CrossCutting.Utility
◢ DataAccessLayer
  ▷ Rony.OnionRingFry.DataAccessLayer
◢ DomainCore
  ◢ Rony.OnionRingFry.DomainCore.Enitities
    ▷ Properties
    ▷ References
    ▷ Coke.cs
    ▷ Egg.cs
    ▷ Flour.cs
    ▷ FriedOnionRings.cs
    ▷ Milk.cs
    ▷ Onion.cs
    ▷ OnionRingsIngredients.cs
    ▷ Seasonings.cs
    ▷ Tray.cs
    ▷ VegetableOil.cs
  ▷ Rony.OnionRingFry.DomainCore.Interfaces
◢ PresentationLayer
  ▷ Rony.OnionRingFry.Demo
```

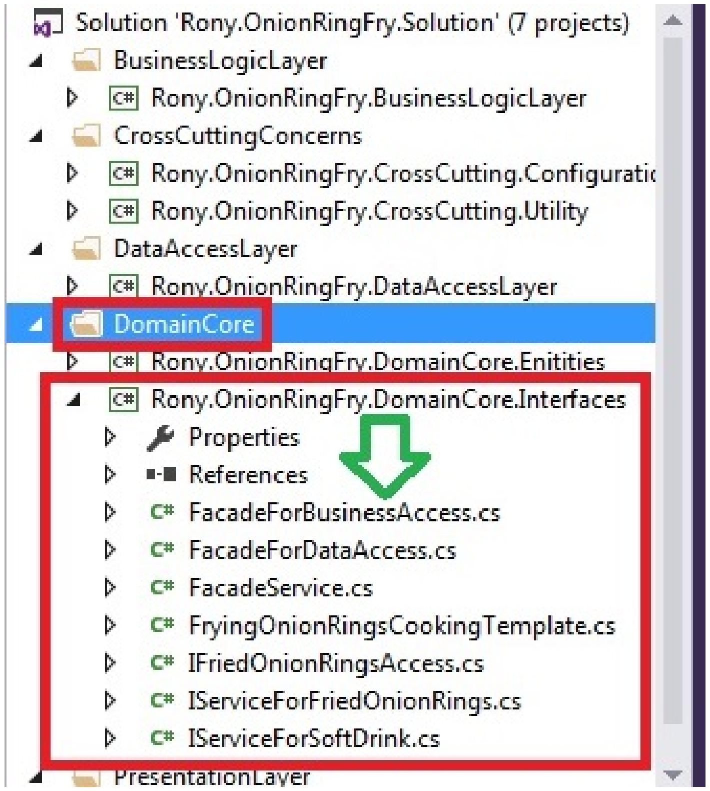
Rony.OnionRingFry.DomainCore.Enitities Project Proper

Identify And Create Domain Services

In this example, I'm not using any domain service and repository.

Identify And Create Application Service Interfaces

This layer defines the Interfaces of the services so that the outer layers can implement these interfaces. In this application service layer, I am placing all of the interfaces of the services.

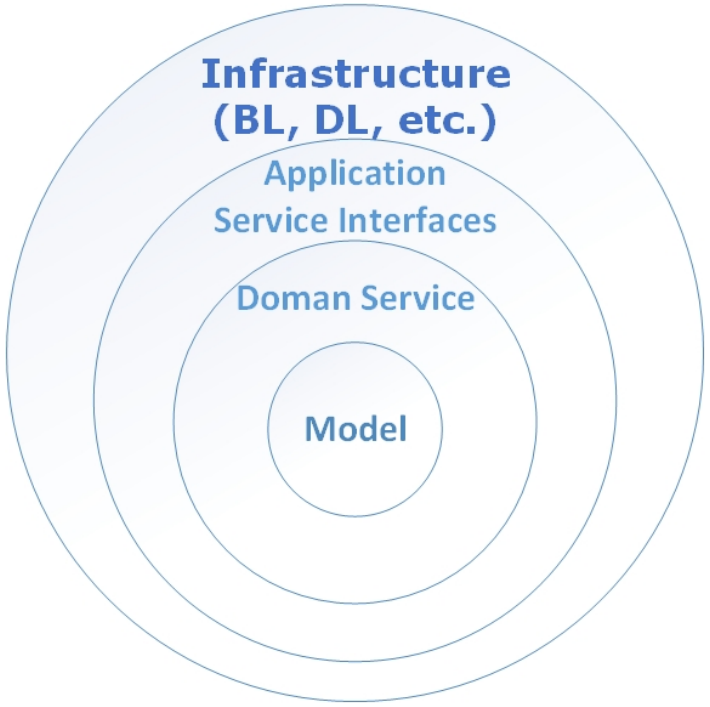


```
Solution 'Rony.OnionRingFry.Solution' (7 projects)
  BusinessLogicLayer
    Rony.OnionRingFry.BusinessLogicLayer
  CrossCuttingConcerns
    Rony.OnionRingFry.CrossCutting.Configuratic
    Rony.OnionRingFry.CrossCutting.Utility
  DataAccessLayer
    Rony.OnionRingFry.DataAccessLayer
  DomainCore
    Rony.OnionRingFry.DomainCore.Enitities
    Rony.OnionRingFry.DomainCore.Interfaces
      Properties
      References
      FacadeForBusinessAccess.cs
      FacadeForDataAccess.cs
      FacadeService.cs
      FryingOnionRingsCookingTemplate.cs
      IFriedOnionRingsAccess.cs
      IServiceForFriedOnionRings.cs
      IServiceForSoftDrink.cs
  PresentationLayer
```

Focus Points Infrastructure

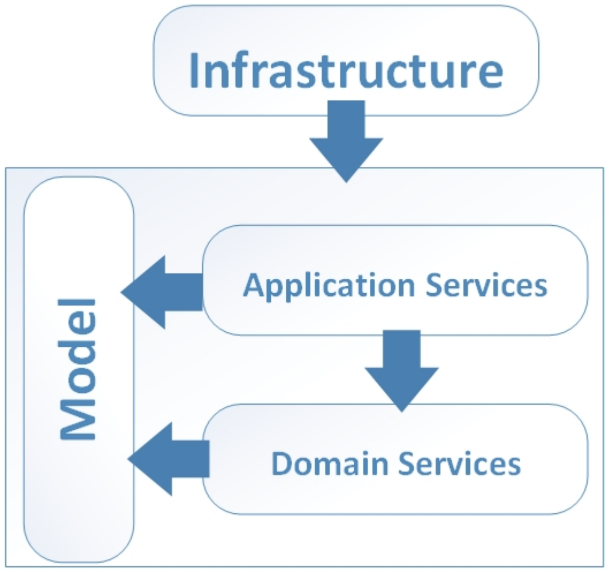
- Business Layer
- Data Access Layer
- External Service

Infrastructure Into Onion Architecture



The Data access layer, Business Layer and Services etc. are known as infrastructure layer according to this onion architecture.

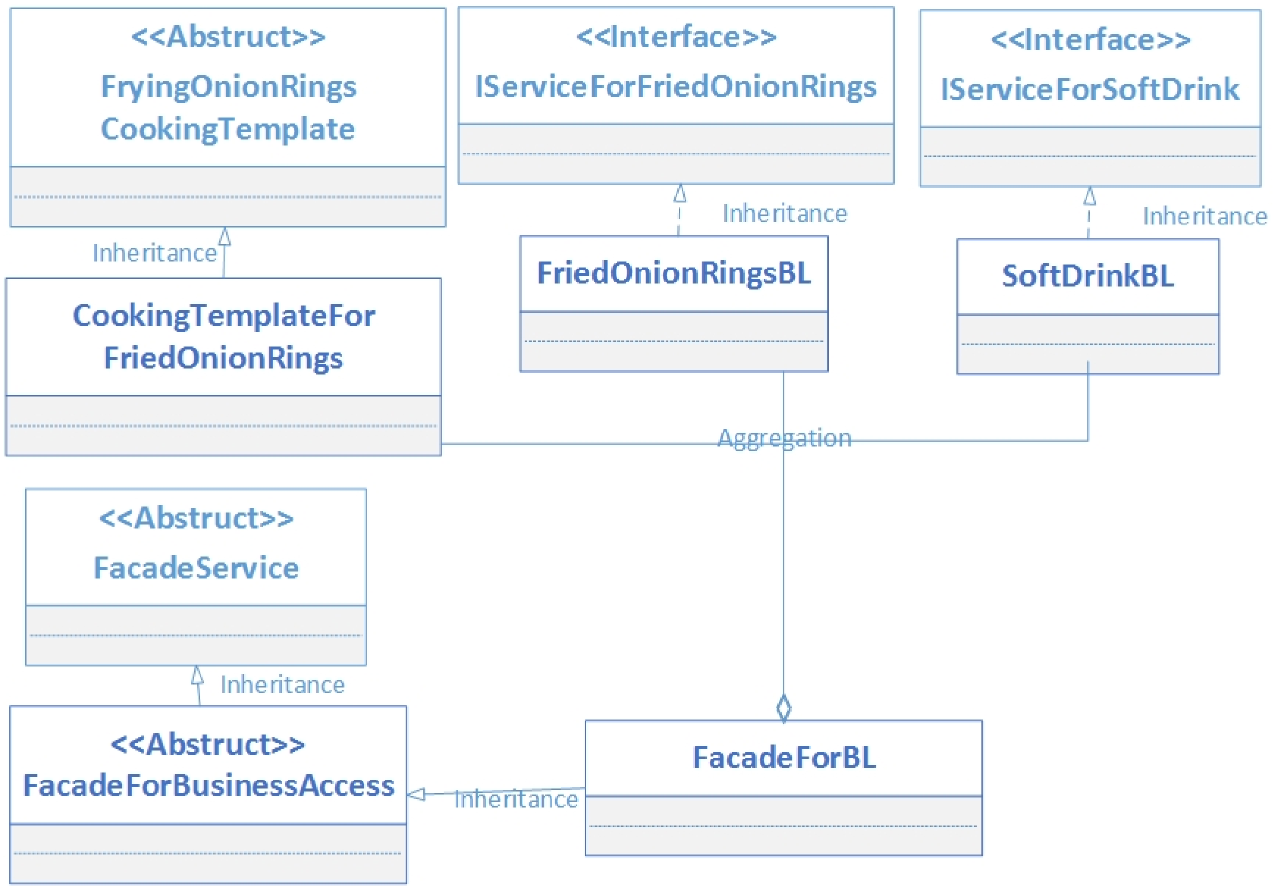
Infrastructure Into DDD



Infrastructure Design

It could be your external Service (WCF, restful service), Business Layer (BL), Data Access Layer (DAL), etc. These are known as the infrastructure. This is just the outer layer of the domain core.

Business Layer Design



Look at the class diagram of the business layer (BL). I am implementing the required functionality of the BL from the application services. So, BL inherits the interfaces and abstract classes from the application services of the domain core and it also uses the domain models. In this layer, I have used two patterns and these are:

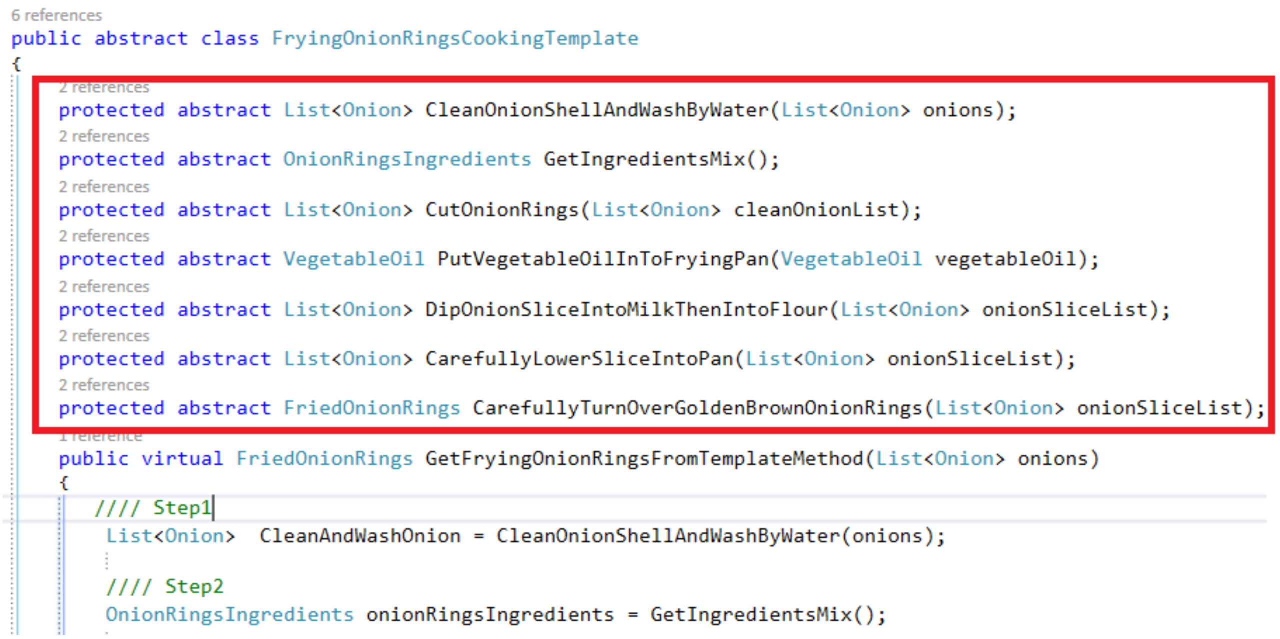
- Template Method Pattern
- Facade Pattern

Template Method Pattern

Template encapsulates the sequence steps of an algorithm.

In this example, I am going to show you how to fry the onion rings. So, if you want to fry this then we have to follow some of the steps and techniques. These are sequential and you have to maintain the order. You can't fry the full onion first and then can't clean the shell of the onion. Look at the below steps:

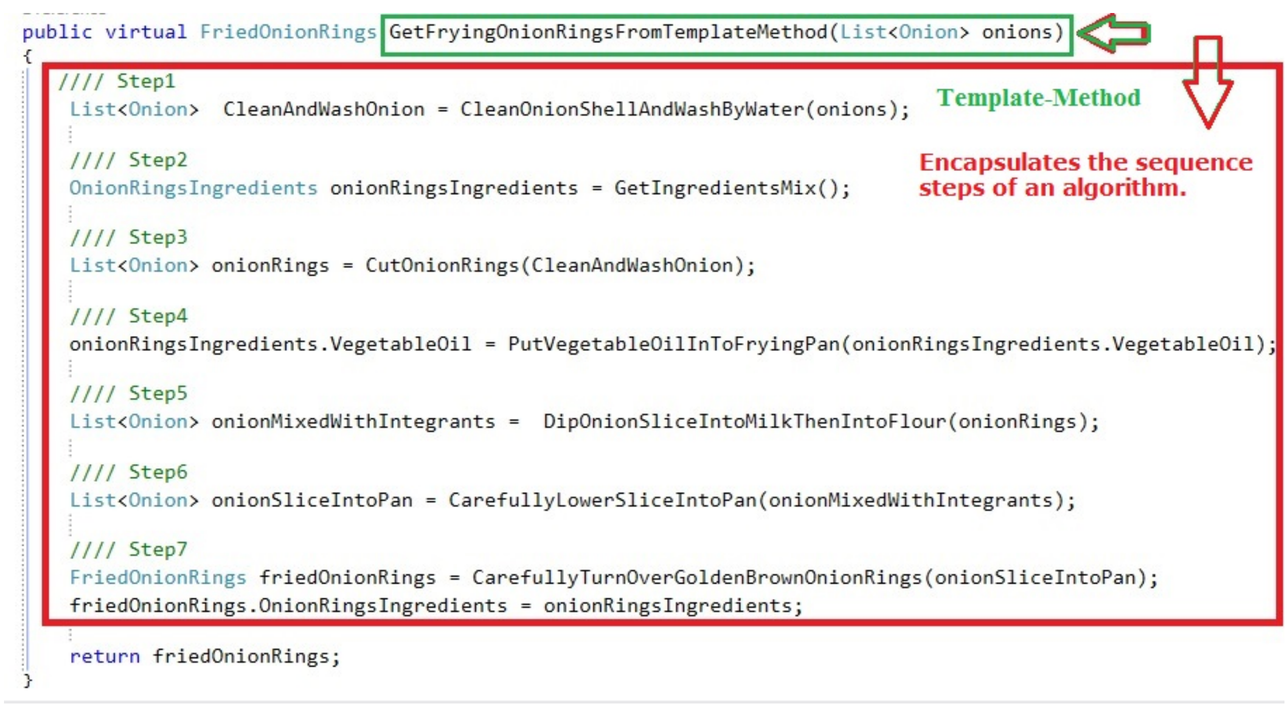
- Step 1: Clean and wash onion
- Step 2: Cut onion slices
- Step 3: Get ingredients
- Step 4: Put vegetable-oil into the frying pan
- Step 5: Dip onion-slice into milk then into flour
- Step 6: Carefully lower slice into Pan
- Step 7: Carefully turn over golden-brown onion-rings



```
6 references
public abstract class FryingOnionRingsCookingTemplate
{
    2 references
    protected abstract List<Onion> CleanOnionShellAndWashByWater(List<Onion> onions);
    2 references
    protected abstract OnionRingsIngredients GetIngredientsMix();
    2 references
    protected abstract List<Onion> CutOnionRings(List<Onion> cleanOnionList);
    2 references
    protected abstract VegetableOil PutVegetableOilInToFryingPan(VegetableOil vegetableOil);
    2 references
    protected abstract List<Onion> DipOnionSliceIntoMilkThenIntoFlour(List<Onion> onionSliceList);
    2 references
    protected abstract List<Onion> CarefullyLowerSliceIntoPan(List<Onion> onionSliceList);
    2 references
    protected abstract FriedOnionRings CarefullyTurnOverGoldenBrownOnionRings(List<Onion> onionSliceList);
    1 reference
    public virtual FriedOnionRings GetFryingOnionRingsFromTemplateMethod(List<Onion> onions)
    {
       //// Step1
        List<Onion>  CleanAndWashOnion = CleanOnionShellAndWashByWater(onions);

        //// Step2
        OnionRingsIngredients onionRingsIngredients = GetIngredientsMix();
```

We can think of each of these steps as separate operations. I mean if we have 7 steps, then it means we need to implement 7 methods to complete the process and you can't ignore the order of the steps.

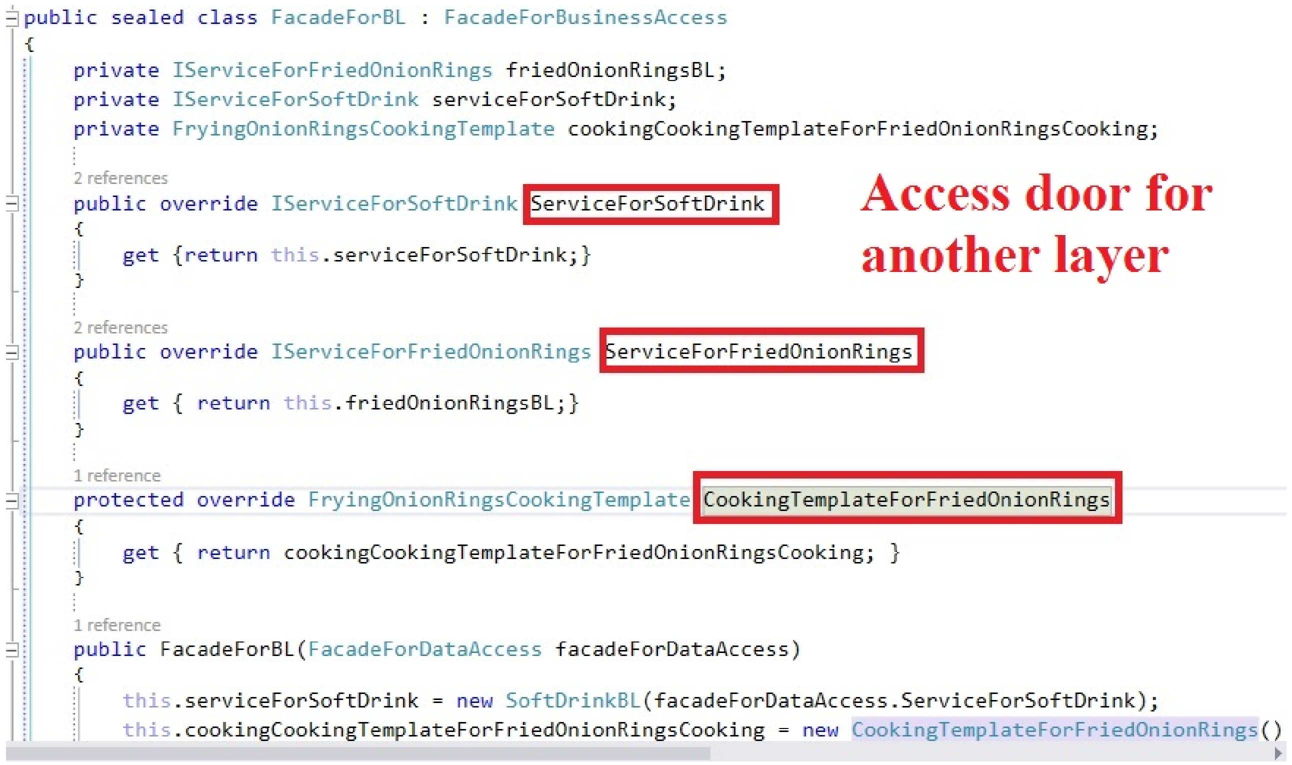
Template method design pattern gives you the facility to implement all of these 7 steps. Because here, you can create the `abstract` methods for each of the steps and finally it has a template method to call all of these steps sequentially.



```
public virtual FriedOnionRings GetFryingOnionRingsFromTemplateMethod(List<Onion> onions)
{
   //// Step1
    List<Onion>  CleanAndWashOnion = CleanOnionShellAndWashByWater(onions);

    //// Step2
    OnionRingsIngredients onionRingsIngredients = GetIngredientsMix();

    //// Step3
    List<Onion> onionRings = CutOnionRings(CleanAndWashOnion);

    //// Step4
    onionRingsIngredients.VegetableOil = PutVegetableOilInToFryingPan(onionRingsIngredients.VegetableOil);

    //// Step5
    List<Onion> onionMixedWithIntegrants =  DipOnionSliceIntoMilkThenIntoFlour(onionRings);

    //// Step6
    List<Onion> onionSliceIntoPan = CarefullyLowerSliceIntoPan(onionMixedWithIntegrants);

    //// Step7
    FriedOnionRings friedOnionRings = CarefullyTurnOverGoldenBrownOnionRings(onionSliceIntoPan);
    friedOnionRings.OnionRingsIngredients = onionRingsIngredients;

    return friedOnionRings;
}
```

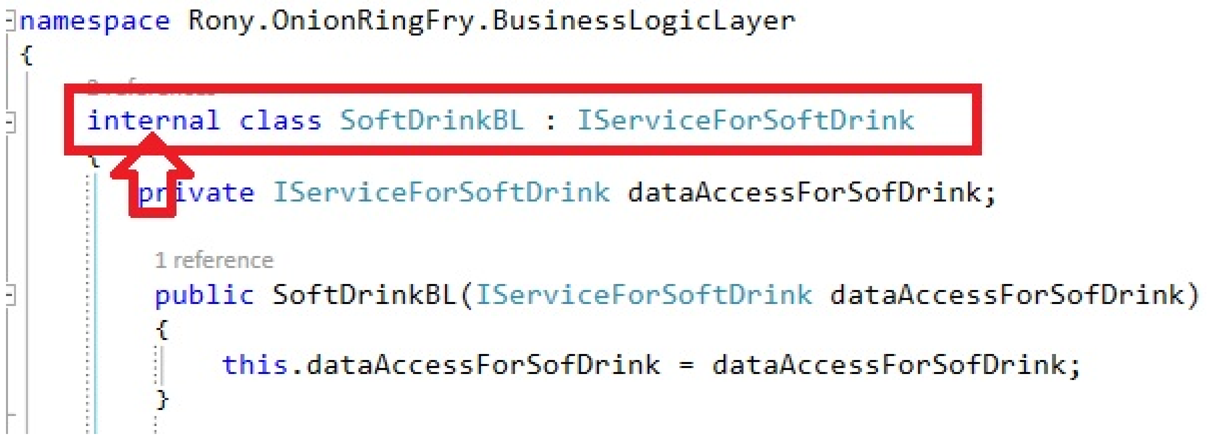
Facade Pattern

Facade is usually used to mask the complexity of an entire subsystem of the layer. You can say that this is the API for other layers to communicate with it. So, another layer doesn't need to know anything about this layer except the facade class.

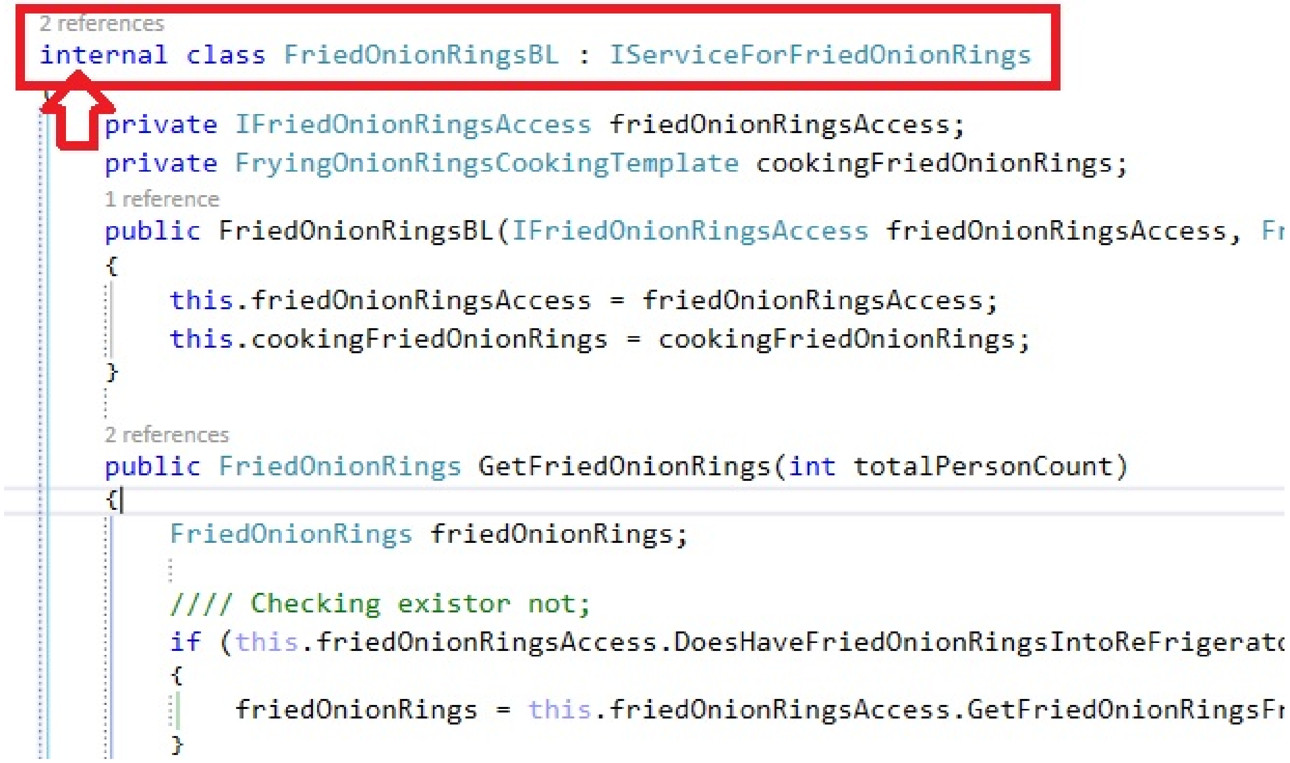


```
public sealed class FacadeForBL : FacadeForBusinessAccess
{
    private IServiceForFriedOnionRings friedOnionRingsBL;
    private IServiceForSoftDrink serviceForSoftDrink;
    private FryingOnionRingsCookingTemplate cookingCookingTemplateForFriedOnionRingsCooking;

    2 references
    public override IServiceForSoftDrink ServiceForSoftDrink
    {
        get {return this.serviceForSoftDrink;}
    }

    2 references
    public override IServiceForFriedOnionRings ServiceForFriedOnionRings
    {
        get { return this.friedOnionRingsBL;}
    }

    1 reference
    protected override FryingOnionRingsCookingTemplate CookingTemplateForFriedOnionRings
    {
        get { return cookingCookingTemplateForFriedOnionRingsCooking; }
    }

    1 reference
    public FacadeForBL(FacadeForDataAccess facadeForDataAccess)
    {
        this.serviceForSoftDrink = new SoftDrinkBL(facadeForDataAccess.ServiceForSoftDrink);
        this.cookingCookingTemplateForFriedOnionRingsCooking = new CookingTemplateForFriedOnionRings()
```

If you look at the other classes in this layer except the facade class, then we will find that I have used internal access modifier so that nobody can create any instance outside of this layer.



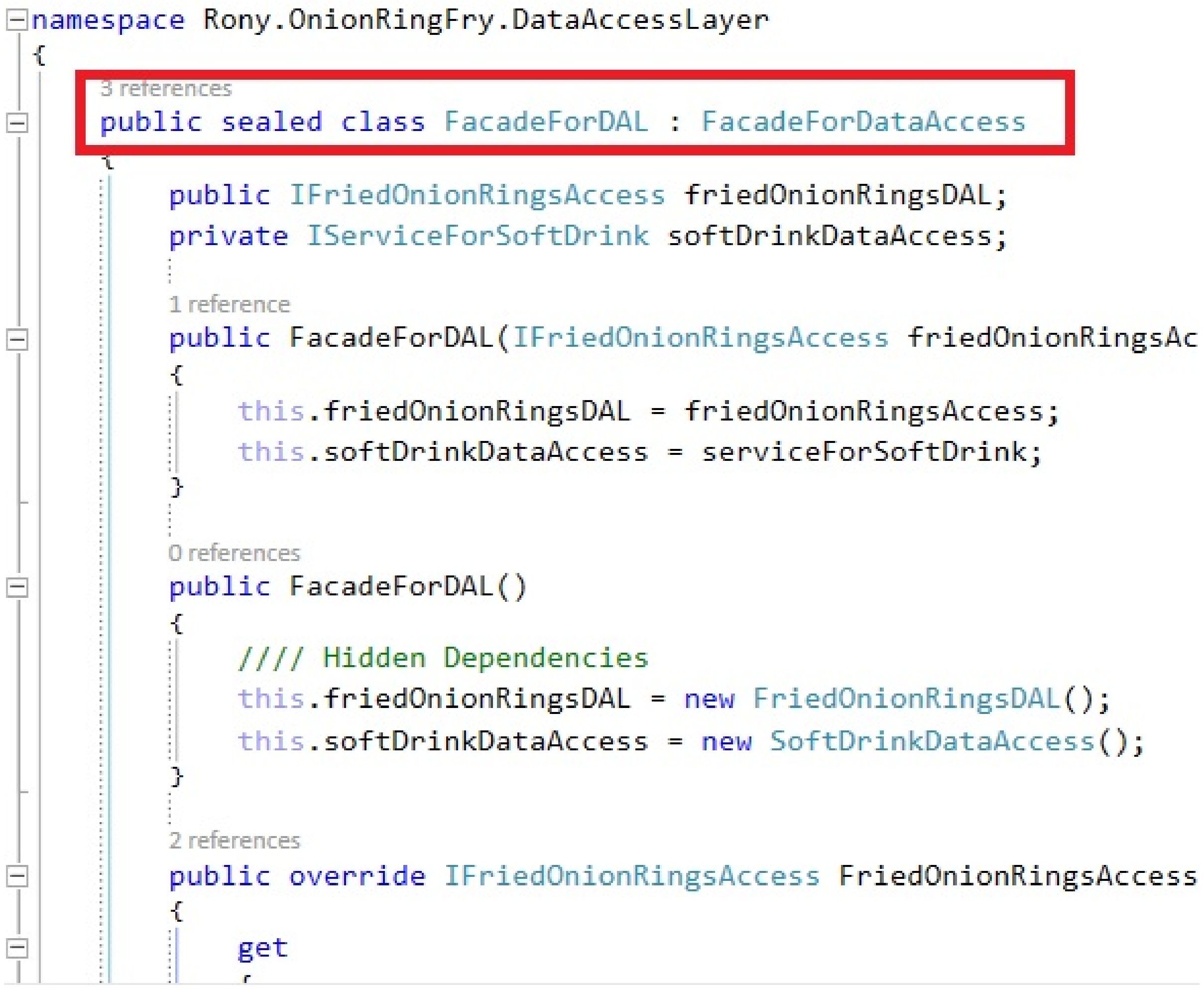
```
namespace Rony.OnionRingFry.BusinessLogicLayer
{
    internal class SoftDrinkBL : IServiceForSoftDrink
    {
        private IServiceForSoftDrink dataAccessForSofDrink;

        1 reference
        public SoftDrinkBL(IServiceForSoftDrink dataAccessForSofDrink)
        {
            this.dataAccessForSofDrink = dataAccessForSofDrink;
        }
```



```
2 references
internal class FriedOnionRingsBL : IServiceForFriedOnionRings
{
    private IFriedOnionRingsAccess friedOnionRingsAccess;
    private FryingOnionRingsCookingTemplate cookingFriedOnionRings;
    1 reference
    public FriedOnionRingsBL(IFriedOnionRingsAccess friedOnionRingsAccess, Fr
    {
        this.friedOnionRingsAccess = friedOnionRingsAccess;
        this.cookingFriedOnionRings = cookingFriedOnionRings;
    }

    2 references
    public FriedOnionRings GetFriedOnionRings(int totalPersonCount)
    {
        FriedOnionRings friedOnionRings;

        //// Checking existor not;
        if (this.friedOnionRingsAccess.DoesHaveFriedOnionRingsIntoReFrigerat
        {
            friedOnionRings = this.friedOnionRingsAccess.GetFriedOnionRingsFr
        }
```

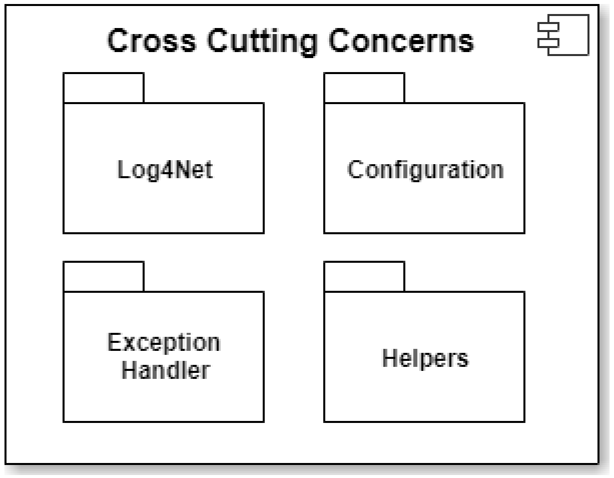
Data Access Layer Design

Similar to BL, I have implemented the required functionality of the data access layer from the application services. So, DL inherits the interfaces and `abstract` classes from the application Services of the domain core and it also uses the domain models. This layer will be accessible to another layer via the facade class.



```
namespace Rony.OnionRingFry.DataAccessLayer
{
    3 references
    public sealed class FacadeForDAL : FacadeForDataAccess
    {
        public IFriedOnionRingsAccess friedOnionRingsDAL;
        private IServiceForSoftDrink softDrinkDataAccess;

        1 reference
        public FacadeForDAL(IFriedOnionRingsAccess friedOnionRingsAc
        {
            this.friedOnionRingsDAL = friedOnionRingsAccess;
            this.softDrinkDataAccess = serviceForSoftDrink;
        }

        0 references
        public FacadeForDAL()
        {
            //// Hidden Dependencies
            this.friedOnionRingsDAL = new FriedOnionRingsDAL();
            this.softDrinkDataAccess = new SoftDrinkDataAccess();
        }

        2 references
        public override IFriedOnionRingsAccess FriedOnionRingsAccess
        {
            get
```

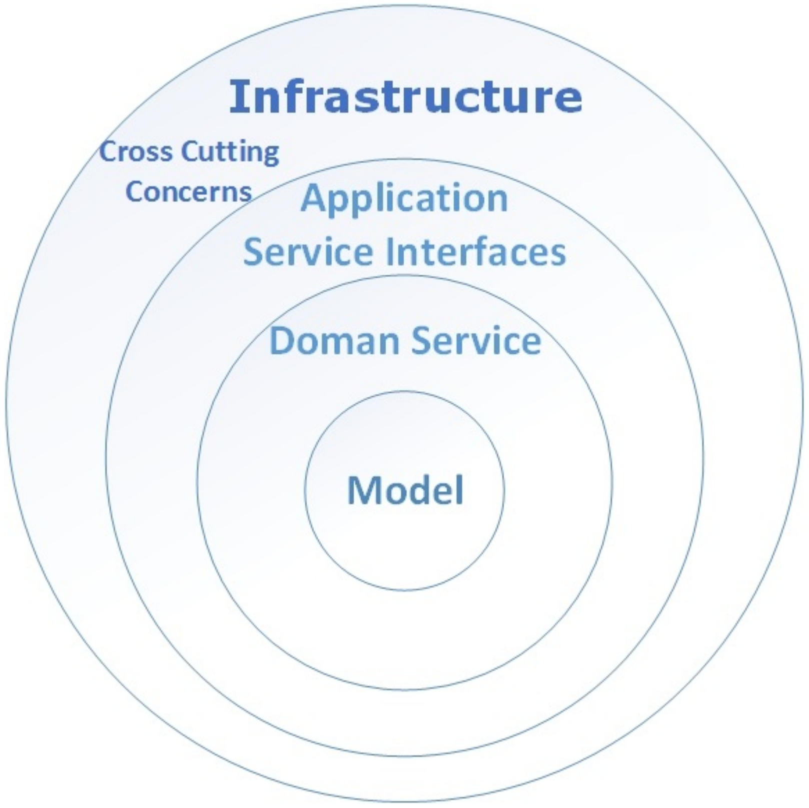
Cross Cutting Concerns



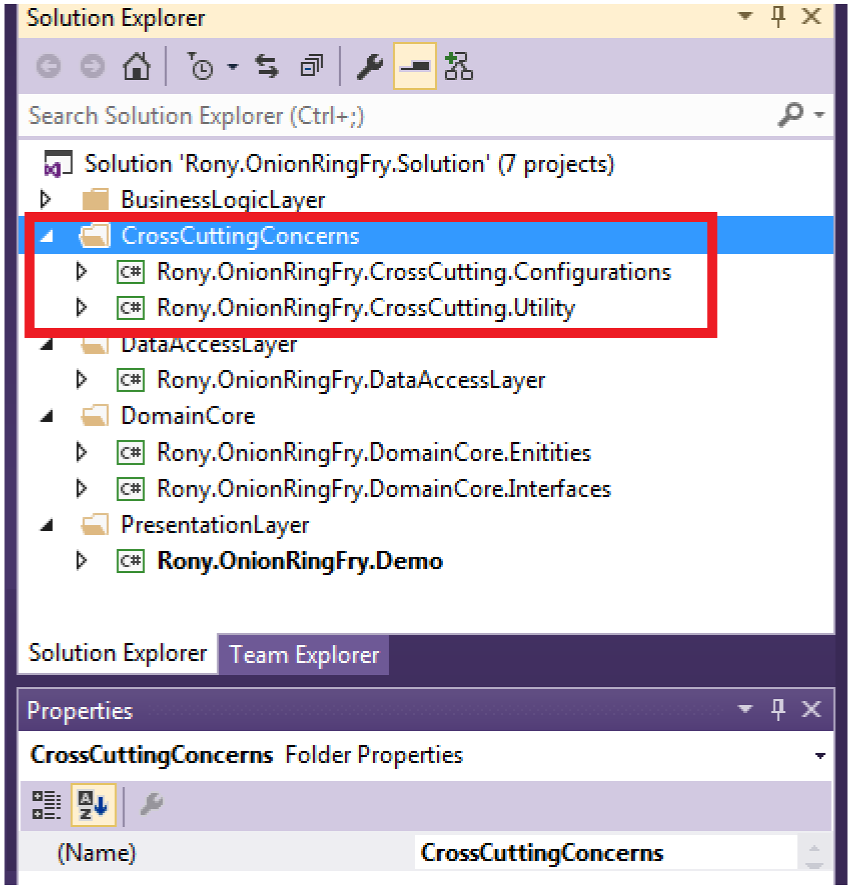
In this component diagram, each of the package folder like Log4Net, Configuration, Exception Handling, and Common Utility or Helpers etc. are known as cross cutting concerns. Generally, any components (BL, DL, PL) can access the crosscutting layers. It is common for all of the layers.

In this component diagram I have placed them into packages of a component; but you can design them as a separate layer or components. It depends on the size of the application and if you want to re-use them.

Cross-Cutting Concerns Into Onion Architecture



In the Onion Architecture crosscutting concern is placed to the last outer layer.



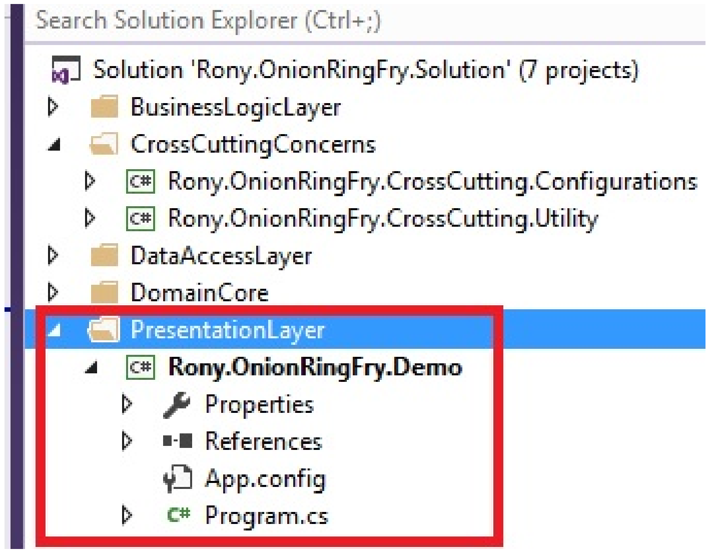
```
Solution 'Rony.OnionRingFry.Solution' (7 projects)
  BusinessLogicLayer
  CrossCuttingConcerns
    Rony.OnionRingFry.CrossCutting.Configurations
    Rony.OnionRingFry.CrossCutting.Utility
  DataAccessLayer
    Rony.OnionRingFry.DataAccessLayer
  DomainCore
    Rony.OnionRingFry.DomainCore.Enitities
    Rony.OnionRingFry.DomainCore.Interfaces
  PresentationLayer
    Rony.OnionRingFry.Demo
```

Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution Explorer
Team Explorer
Properties
CrossCuttingConcerns Folder Properties
(Name) CrossCuttingConcerns

Note that - in this application I have created the Configuration and Utility structure into the Cross Cutting project folder. But I did not implement anything there. Even-if I did not handle the exceptions or logging stuff. Because my goal to show you the structures of the application. So, I avoid the coding related stuff to keep it light.

Presentation Layer

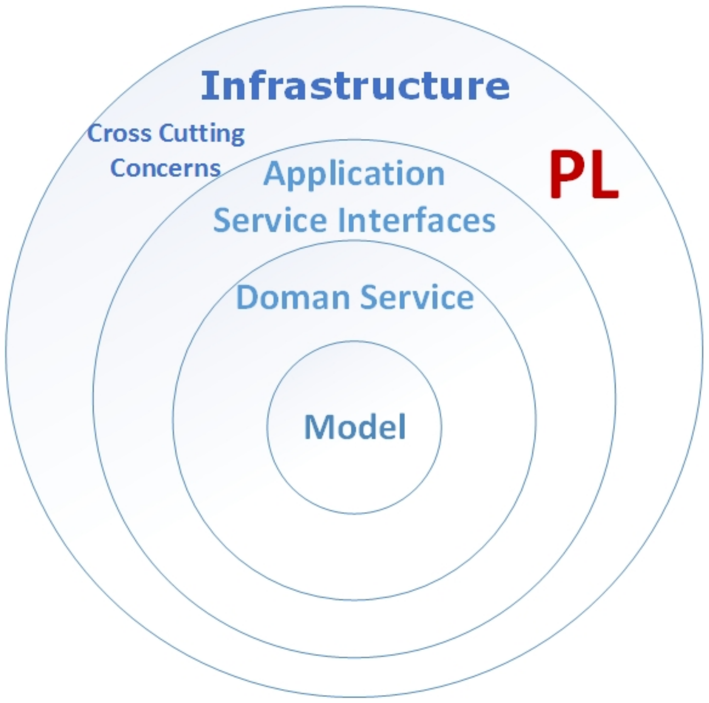
Presentation layer (PL) or user interface will be placed on the outer layer of the onion architecture like infrastructure.



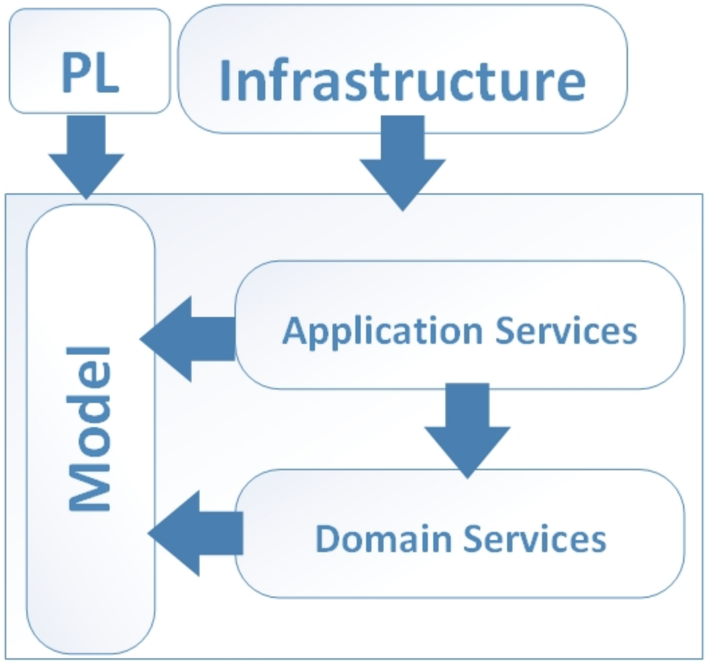
Search Solution Explorer (Ctrl+;)

```
Solution 'Rony.OnionRingFry.Solution' (7 projects)
  BusinessLogicLayer
  CrossCuttingConcerns
    Rony.OnionRingFry.CrossCutting.Configurations
    Rony.OnionRingFry.CrossCutting.Utility
  DataAccessLayer
  DomainCore
  PresentationLayer
    Rony.OnionRingFry.Demo
      Properties
      References
      App.config
      Program.cs
```

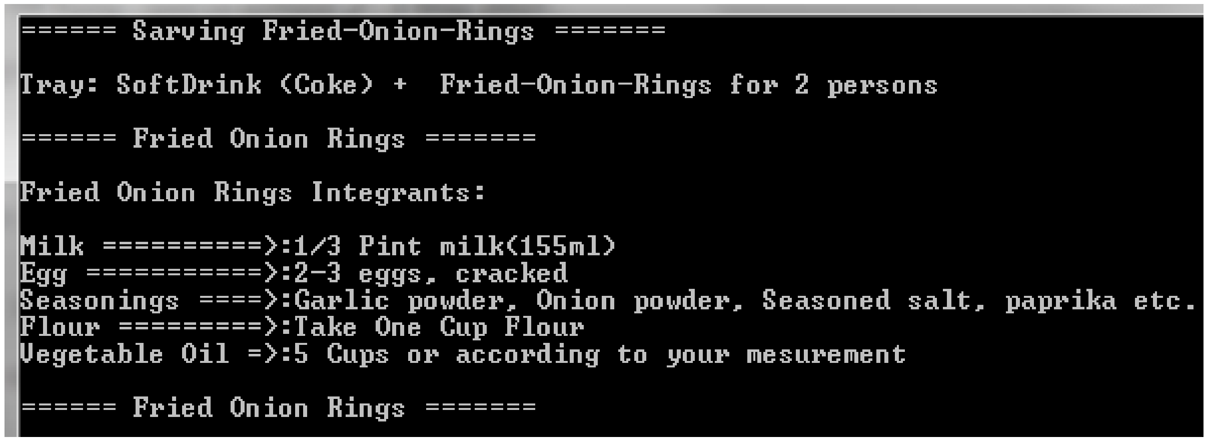
PL Into Onion Architecture



PL Into DDD

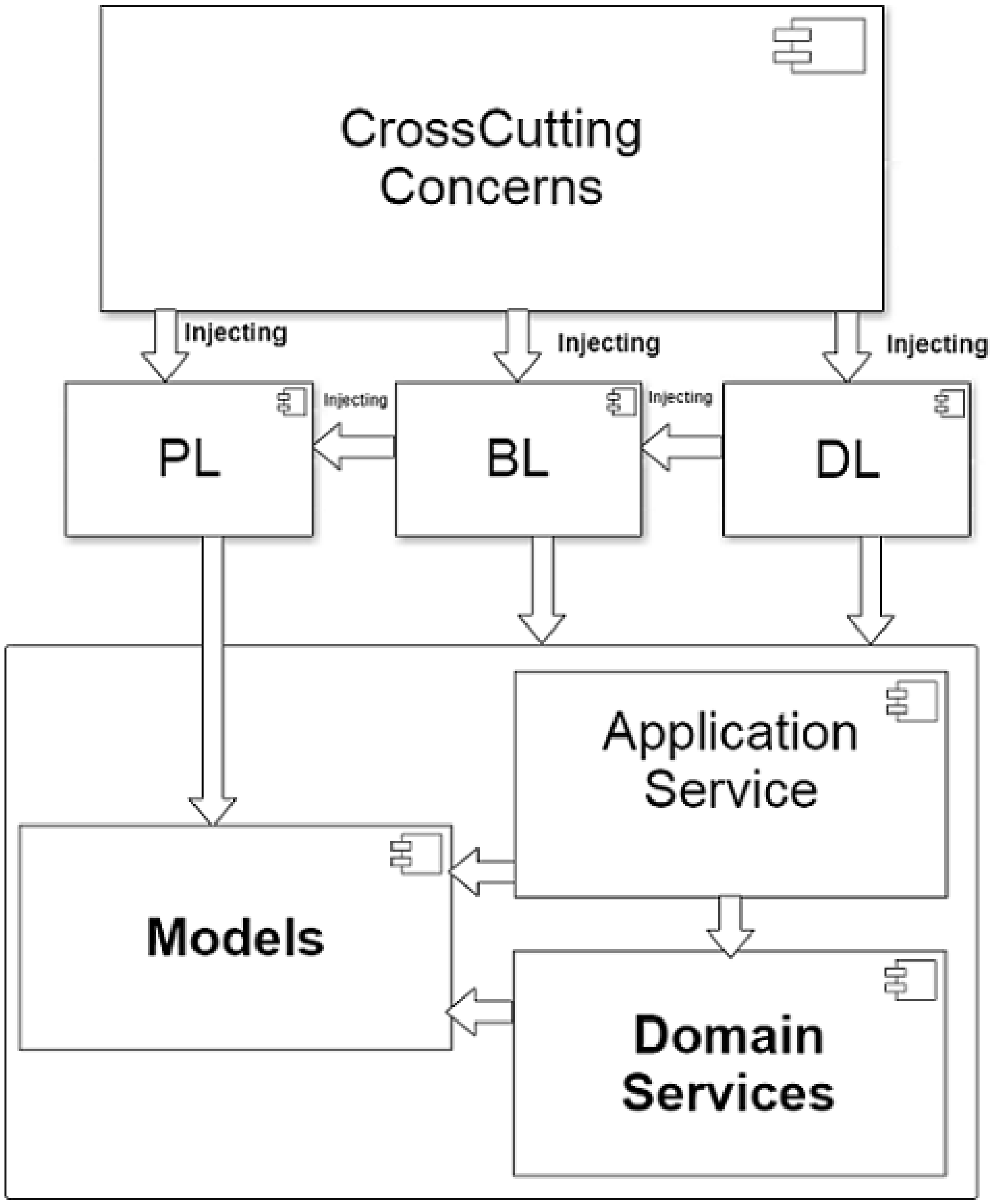


If you run the project (find the attachment), then you will see the output of the fried-onion-rings from the PL.



```
====== Sarving Fried-Onion-Rings =======

Tray: SoftDrink (Coke) +  Fried-Onion-Rings for 2 persons

====== Fried Onion Rings =======

Fried Onion Rings Integrants:

Milk ==========>:1/3 Pint milk(155ml)
Egg ===========>:2-3 eggs, cracked
Seasonings ====>:Garlic powder, Onion powder, Seasoned salt, paprika etc.
Flour =========>:Take One Cup Flour
Vegetable Oil =>:5 Cups or according to your mesurement

====== Fried Onion Rings =======
```

Whole Picture Of The Application



Finally, I can say that I'm not telling you that if you use DDD for your application, then it will be excellent. My main goal was to introduce you with the architectural style and design. That's why, I have escaped the principles, advantage, disadvantage and some other coding stuff.

If you start to implement any components of your application, then at-least think that if you follow any architectural style and design, then your software will be healthy. Nobody likes to stay overnight in the hospital with his/her application. So, the right design can help you to build better applications, improve endurance, and speed recovery.

License

Share

About the Author

HR Rony

Engineer
United States

Lazy software engineer in web/mobile application development, architecture design, and automated unit testing. Don't like study; but love to learn. Don't believe in hard working, that's why, spend most of the time to think and work as minimum as I can! Don't compromise if something goes against my philosophy. Don't believe that majority always right!

Comments and Discussions

You must Sign In to use this message board.

Search Comments

Spacing Relaxed ▼ Layout Normal ▼ Per page 25 ▼ Update

-- There are no messages in this forum --

Permalink
Advertise
Privacy
Cookies
Terms of Use

Layout: fixed | fluid