**Clean Architecture কি?**

**Clean Architecture** হলো সফটওয়্যার ডিজাইনের একটা প্যাটার্ন, যা কোডকে এমনভাবে সাজায় যেনো সহজে রিড, টেস্ট এবং মেইনটেইন করা যায়।  
এর মূল উদ্দেশ্য হলো **বিভিন্ন লেয়ার (স্তর) আলাদা রাখা** এবং ডিপেন্ডেন্সি নিয়ন্ত্রণ করা যাতে ইউজার ইন্টারফেস, ব্যবসায়িক লজিক, ডাটাবেজ ইত্যাদি আলাদা থাকে।

**Clean Architecture এর লেয়ারগুলো:**

1. **Entities (ডোমেইন মডেল)**
   * ব্যবসায়িক নিয়ম এবং লজিক।
   * কোন বাইরের প্রযুক্তি নির্ভর নয়।
   * C# এ সাধারণত POCO (Plain Old CLR Object) ক্লাস।
2. **Use Cases (ইন্টারঅ্যাক্টর)**
   * এপ্লিকেশনের বিশেষ কাজের লজিক।
   * Entities কে ব্যবহার করে।
   * কোন বাইরের ডিপেন্ডেন্সি কম থাকাটা গুরুত্বপূর্ণ।
3. **Interface Adapters (প্রেজেন্টেশন / ডাটা কনভার্টার)**
   * ডাটা কে Use Case এবং Entity এর ফরম্যাটে কনভার্ট করে।
   * UI, ডাটাবেস, API ইত্যাদির ইন্টারফেস থাকে।
4. **Frameworks & Drivers (ডাটাবেস, UI, বাইরের সার্ভিস)**
   * বাস্তব জগতের বাইরের সফটওয়্যার, যেমন ডাটাবেস, ওয়েব সার্ভার ইত্যাদি।

**ডিপেন্ডেন্সি রুল (Dependency Rule):**

* বাইরের লেয়ার থেকে ভিতরের লেয়ারে ডিপেন্ডেন্সি থাকতে পারে, কিন্তু ভিতরের লেয়ার থেকে বাইরের লেয়ারে **ডিপেন্ডেন্সি যাবে না**।
* অর্থাৎ, বেসিক বিজনেস লজিক কোন বাইরের লাইব্রেরির উপর নির্ভরশীল হবে না।

**C# কোড উদাহরণ**

**1. Entity (ডোমেইন মডেল)**

// Business logic এর জন্য entity ক্লাস

public class Product

{

public int Id { get; private set; }

public string Name { get; private set; }

public decimal Price { get; private set; }

public Product(int id, string name, decimal price)

{

Id = id;

Name = name;

Price = price;

}

public void UpdatePrice(decimal newPrice)

{

if (newPrice <= 0)

throw new ArgumentException("Price must be positive");

Price = newPrice;

}

}

**2. Use Case (ইন্টারঅ্যাক্টর)**

public interface IProductRepository

{

Product GetById(int id);

void Save(Product product);

}

public class UpdateProductPriceUseCase

{

private readonly IProductRepository \_repository;

public UpdateProductPriceUseCase(IProductRepository repository)

{

\_repository = repository;

}

public void Execute(int productId, decimal newPrice)

{

var product = \_repository.GetById(productId);

if (product == null)

throw new Exception("Product not found");

product.UpdatePrice(newPrice);

\_repository.Save(product);

}

}

**3. Interface Adapter (Repository Implementation)**

// DbContext হলো EF Core এর অংশ (Framework Layer)

public class EfProductRepository : IProductRepository

{

private readonly AppDbContext \_context;

public EfProductRepository(AppDbContext context)

{

\_context = context;

}

public Product GetById(int id)

{

// EF Core entity কে domain entity তে কনভার্ট করতে হবে (এখানে সরাসরি ধরে নেওয়া হয়েছে)

var productEntity = \_context.Products.Find(id);

if (productEntity == null) return null;

return new Product(productEntity.Id, productEntity.Name, productEntity.Price);

}

public void Save(Product product)

{

var productEntity = \_context.Products.Find(product.Id);

if (productEntity != null)

{

productEntity.Price = product.Price;

\_context.SaveChanges();

}

}

}

**4. Framework Layer (ASP.NET Core Controller)**

[ApiController]

[Route("api/products")]

public class ProductsController : ControllerBase

{

private readonly UpdateProductPriceUseCase \_updatePriceUseCase;

public ProductsController(UpdateProductPriceUseCase updatePriceUseCase)

{

\_updatePriceUseCase = updatePriceUseCase;

}

[HttpPut("{id}/price")]

public IActionResult UpdatePrice(int id, [FromBody] decimal newPrice)

{

try

{

\_updatePriceUseCase.Execute(id, newPrice);

return Ok();

}

catch (Exception ex)

{

return BadRequest(ex.Message);

}

}

}

**সারাংশ**

* **Entities**: শুধু ব্যবসায়িক তথ্য ও নিয়ম।
* **Use Cases**: এপ্লিকেশনের কাজের লজিক।
* **Interface Adapters**: ডাটা ট্রান্সফার ও কনভার্শন।
* **Framework**: বাইরের সিস্টেম (ডাটাবেস, UI)।

এভাবেই Clean Architecture কোডকে ছোট ছোট অংশে ভাগ করে কাজ সহজ ও মেইনটেইনযোগ্য করে তোলে।

**Clean Architecture এর বিস্তারিত ব্যাখ্যা (বাংলায়)**

**১. কেন Clean Architecture দরকার?**

প্রকৃত জীবনের সফটওয়্যার সিস্টেম বড় এবং জটিল হয়ে যায়। যখন তুমি নতুন ফিচার যোগ করো, বাগ ঠিক করো বা অন্য কোনো পরিবর্তন করো, তখন যদি কোড ঠিকভাবে ভাগ করা না থাকে, তাহলে:

* **কোড বুঝতে কষ্ট হয়**
* **বাগ হয় বেশি**
* **কোড রিপিটেশন বেশি হয়**
* **পরিবর্তন করতে গিয়ে অন্য জায়গায় সমস্যা তৈরি হয়**

Clean Architecture এসব সমস্যা কমাতে সাহায্য করে।

**২. Clean Architecture এর মূল উদ্দেশ্য**

* **সিস্টেমের বিভিন্ন অংশকে আলাদা রাখা (Separation of Concerns)**
* **অন্তর্নিহিত ব্যবসায়িক নিয়ম এবং লজিককে বাইরের ইমপ্লিমেন্টেশন থেকে আলাদা রাখা**
* **ডিপেন্ডেন্সি ইনভার্শন প্রিন্সিপল (Dependency Inversion Principle) মানা** — অর্থাৎ ভিতরের লেয়ার বাইরের উপর নির্ভর করে না, বাইরের লেয়ার ভিতরের উপর নির্ভর করে।
* **টেস্টিং সহজ করা**
* **মেইনটেইনযোগ্যতা এবং রিইউজেবিলিটি বৃদ্ধি করা**

**৩. Clean Architecture এর লেয়ার সমূহ**

**৩.১ Entities (ডোমেইন মডেল)**

* **ব্যবসায়িক নিয়ম ও ডেটার মডেল।**
* টেকনোলজি-নিরপেক্ষ (Pure Business Logic)।
* অন্য কোনো লাইব্রেরি বা ফ্রেমওয়ার্কের উপর নির্ভর নয়।
* **উদাহরণ:** Product, Customer, Order ইত্যাদি ক্লাস।

**৩.২ Use Cases (ইন্টারঅ্যাক্টর)**

* **ব্যবসায়িক লজিকের কাজ বা প্রসেস।**
* Entities কে ব্যবহার করে কাজ সম্পাদন করে।
* **Business Rules কে বাস্তবায়ন করে।**
* বাইরের যেকোনো ইমপ্লিমেন্টেশন থেকে সম্পূর্ণ আলাদা।
* এক ধরনের সার্ভিস বা ইন্টারফেস যা Use Case বাস্তবায়ন করে।

**৩.৩ Interface Adapters (প্রেজেন্টেশন ও ডাটা কনভার্টার)**

* ইউজার ইন্টারফেস, ডাটাবেস, ওয়েব API থেকে আসা ডাটা এবং Use Case এর মধ্যে ডাটা কনভার্সন করে।
* উদাহরণ: Repository ইমপ্লিমেন্টেশন, Controller, Presenter, ViewModel ইত্যাদি।
* বাইরের ফ্রেমওয়ার্ক বা টুল থেকে ডোমেইন মডেলে বা Use Case-এ ডাটা পাঠানোর মাধ্যম।

**৩.৪ Frameworks & Drivers (External Infrastructure)**

* ডাটাবেস, ওয়েব ফ্রেমওয়ার্ক, UI, বাইরের সার্ভিস, ইত্যাদি।
* সিস্টেমের বাইরের অংশ।
* সাধারণত এখানে ফ্রেমওয়ার্ক, ORM (যেমন EF Core), UI লাইব্রেরি থাকে।

**৪. ডিপেন্ডেন্সি ইনভার্শন রুল**

* **কোডের ডিপেন্ডেন্সি সবসময় ভিতরের লেয়ারের দিকে থাকা উচিত।**
* অর্থাৎ, বাইরের লেয়ার যেমন ডাটাবেস বা UI, ভিতরের লেয়ারের (ব্যবসায়িক লজিক) ইন্টারফেসে ডিপেন্ড করবে।
* ভিতরের লেয়ার বাইরের কোন ক্লাস বা লাইব্রেরির ওপর নির্ভর করবে না।

**৫. ডোমেইন ড্রিভেন ডিজাইন (DDD) এর সাথে Clean Architecture**

Clean Architecture প্রায়শই Domain-Driven Design (DDD) এর সাথে ব্যবহার করা হয়, যেখানে:

* **Entity, Value Object, Aggregate Root** ডোমেইন মডেল হিসেবে থাকে।
* Use Cases বা Application Services ব্যবসায়িক কাজের প্রবাহ নিয়ন্ত্রণ করে।
* Repository Interface ডোমেইনের অংশ কিন্তু Implementation বাইরের লেয়ারে।

**৬. C# প্রোজেক্টে Clean Architecture ফোল্ডার স্ট্রাকচার**

/src

/Domain --> Entities, Interfaces, Business Rules

/Application --> Use Cases, DTOs, Services

/Infrastructure --> DB Context, Repository Implementations, External Services

/WebApi --> Controllers, UI, API related stuff

**৭. ডিপেন্ডেন্সি ইনজেকশন (DI) দিয়ে Clean Architecture বাস্তবায়ন**

Use Case গুলো Interface এর ওপর নির্ভর করবে, Repository ইমপ্লিমেন্টেশন DI দিয়ে Inject করা হবে।

// Program.cs এ DI কনফিগারেশন

services.AddScoped<IProductRepository, EfProductRepository>();

services.AddScoped<UpdateProductPriceUseCase>();

**৮. আরো উন্নত ধারণা**

**৮.১ Boundary Interface**

Use Case এবং Repository ইন্টারফেস দ্বারা লেয়ার গুলো আলাদা রাখা হয়।

**৮.২ Dependency Rule Violations**

অধিকাংশ সমস্যা হয় যখন ভিতরের লেয়ার বাইরের লেয়ারের ক্লাস বা লাইব্রেরির উপর নির্ভর করে।

**৮.৩ টেস্টিং সহজ হয়**

বিজনেস লজিক সম্পূর্ণ আলাদা হওয়ায় Unit Test লেখা সহজ হয়।

**৯. Clean Architecture এর সুবিধা**

* মডিউলার ও স্কেলেবল কোড
* সহজে টেস্টিং করা যায়
* পরিবর্তন সহজ (যেমন UI বদলানো বা ডাটাবেস পরিবর্তন)
* উন্নত রিইউজেবিলিটি
* ডিপেন্ডেন্সি কম থাকে

**১০. উপসংহার**

Clean Architecture এর মূল কথা হলো **ব্যবসায়িক লজিককে সম্পূর্ণ স্বাধীন রাখা** এবং বাইরের ডিপেন্ডেন্সিগুলোকে লেয়ার অনুযায়ী আলাদা রাখা। এতে করে সফটওয়্যার সহজে বোঝা যায়, রক্ষণাবেক্ষণ ও পরিবর্তন করা যায়।

**তোমার জন্য একটা ছোট টিপস:**

Clean Architecture একটি গাইডলাইন, সেটাকে একদম কঠোর নিয়ম মনে করার দরকার নেই। প্রয়োজন অনুযায়ী নিজের প্রোজেক্টে অ্যাডাপ্ট করো।

তুমি কি Clean Architecture নিয়ে কোনো নির্দিষ্ট অংশে আরও বিস্তারিত জানতে চাও?  
যেমন:

* Use Case লেখা
* Repository ইমপ্লিমেন্টেশন
* DI কিভাবে কাজ করে
* বা কোডের একটা পূর্ণাঙ্গ উদাহরণ?