# Agile search

Team members:

1. Ahmed Heikel
2. Akram Muhammad
3. Mohamed Saied Hassan
4. Youssef Mohamed Abdullah

# What is Clean Architecture?

Clean Architecture is a way of organizing software so that the system becomes:

- Easy to maintain
- Easy to test
- Independent from frameworks
- Independent from databases
- Flexible and scalable

It was introduced by **Robert C. Martin (Uncle Bob)**.

The main idea is to separate business logic from external tools like UI, database, or frameworks.

## Layers of Clean Architecture

Clean Architecture has 5 main layers:

1. Entities (Domain Layer)
    - Contains the core business models (e.g., User, Product, Order)
    - Includes basic business rules.
    - Does NOT depend on any other layer

    Example:
    A User class should not know anything about how it is saved in the database

2. Use Cases (Application Layer)
    - Defines what the system should do.
    - Does not depend on UI or database.
    - Uses Entities to perform operations

    Example:
    If a user registers:
        - The Use Case validates the data.
        - Creates a User object.

    - Sends it to the infrastructure layer to save it.

3. Interface Adapters
    - Receives requests from the user (API call, HTTP request).
    - Validates and formats the input data
    - Calls the appropriate Use Case.
    - Sends back a response.

    Example:
    If someone sends invalid data (wrong date, missing fields),
    the controller stops it and returns an error.

4. Frameworks & Drivers (Outer Layer)

    This layer deals with external tools and technical details.

    - Database operations
    - External APIs
    - Email services
    - Payment gateways

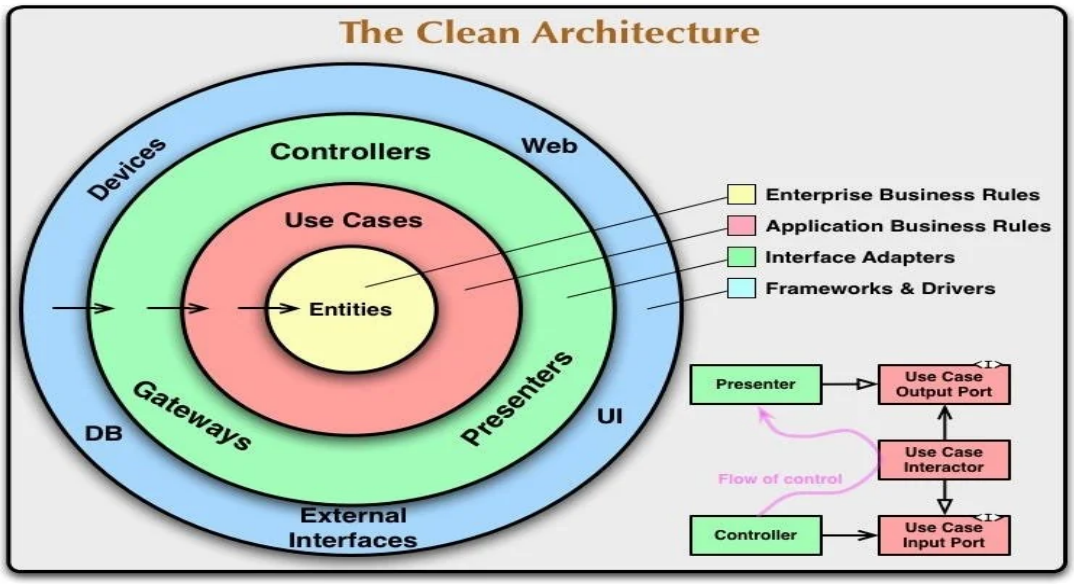
5. Outermost Layer (UI)

    This is the user interface:

    - Web pages
    - Mobile apps
    - Frontend applications

## When to Use Clean Architecture in Business

1. Large-Scale Applications




2. Long-Term Projects
3. Projects That May Change Technology (e.g., Change database or framework)
4. Systems That Require Strong Testing

## When NOT to Use It

1. Very small projects
2. Simple CRUD applications
3. Short-term freelance projects

## References

GeeksforGeeks. (2025). Retrieved from Complete Guide to Clean Architecture: https://www.geeksforgeeks.org/system-design/complete-guide-to-clean-architecture/

medium. (2025). *Complete Guide to Clean Architecture*. Retrieved from Complete Guide to Clean Architecture: https://medium.com/@rudrakshnanavaty/clean-architecture-7c1b3b4cb181

Continuous integration tools

What is Continuous integration

At the beginning the CI goals  is reduce integration problems and catch bugs early

So it's a tools to automate the process of integration code changes ,deploying the software ,building and testing

1- JenKis
it's open source automation server helping in building, testing, and deploying, facilitating continuous integration, and continuous delivery It supports version control tools, including AccuRev, CVS, Subversion, Git, Mercurial, Perforce, ClearCase, and RTC, and can execute Apache Ant, Apache Maven,

## When to Use Jenkins

1- team has dedicated DevOps/infrastructure engineers
2- full control over build environment

3- complex, custom pipelines

4- dealing with on-premises or air-gapped environments (banking, defense, healthcare)

**When NOT to Use Jenkins**

Small teams without DevOps expertise

Pricing

**Jenkins OSS Free**

**Self-Hosted Infra Variable ($50–$2,000+/mo)**

**Managed Hosting $40–$350+/mo**

**Enterprise Commercial Custom / $20,000+**

# GitHup:

GitHub is Version Control track changes to your code over time and be able to roll back to previous versions, it's a platform that helps you build and ship software with AI-powered coding assistance, CI/CD, cloud environments, and more.

When to Use GitHub:

1- When I need to track changes on code and rollback to the previous versions
2- When I need Collaboration Githup lets everyone works without overwriting on other's code
3- Safe storage
4- When I need open-source Project
5- Project management using tools like pull requests
6- Thousands of reusable community actions
7- need cross-platform builds (Linux, Windows, macOS runners available
8- Branching and Collaboration
9- CI, CD Atuomation for testing, deployment
10- Portfolio when I need to show my work to Clients

# 3- Azure DevOps:

it’s cloud based platform created by Microsoft have tools for software development from planning to deployment

Greate for Agile, has Git based version control system , Automate code building, testing, and deploying.

it includes:

1-Azure Boards

2- Azure Repos

3- Azure Pipelines

4- Azure Test Plans

5- Azure Artifacts

When to Use Azure DevOps:

1- working in a large enterprise
2- team uses Microsoft/Azure ecosystem
3- built in strong project management
4- security and grant access controls
5- if you have license from Microsoft
6- some thing that support any language ,cloud or platform

# TOP 10 CLEAN CODE

# 1. Meaningful Naming

- *Variables, functions, and classes should clearly describe their purpose.*
- *Good naming reduces the need for comments and improves readability.*

```
// Bad Example
int d = 5;

// Good Example
int daysUntilExpiration = 5;

// Bad Function Name
decimal Calc(decimal p) { return p * 0.1m; }

// Good Function Name
decimal CalculateOrderTotal(decimal price) { return price * 0.1m; }
```

# 2. Keep It Simple (KISS)

- Complex solutions often introduce unnecessary overhead.
- increase the risk of bugs.
- Simple code is easier to understand, debug, and maintain.

```
// Bad Example
if (true == true)
{
    Console.WriteLine("User is active");
}

// Good Example
bool isActive = true;
if (isActive)
    Console.WriteLine("User is active");
```

# 3. Small, Focused Functions (SRP)

- A function should do **one thing, and do it well**.
- Aim for short, focused functions — easier to test and reuse.

```
// Bad Example
void ProcessOrderBad(Order order)
{
    Validate(order);
    SaveToDatabase(order);
    SendEmail(order);
    Log(order);
}

// Good Example
void ProcessOrderGood(Order order)
{
    Validate(order);
    SaveOrder(order);
    NotifyCustomer(order);
}
```

# 4. DRY Principle (Don't Repeat Yourself)

- If you copy-paste code, that’s a red flag.
- Repeated code increases bugs and maintenance effort.
- Extract it into a function or utility.

```
// Bad Example
decimal total1 = 100 + 100 * 0.15m;
decimal total2 = 200 + 200 * 0.15m;

// Good Example
const decimal TaxRate = 0.15m;
decimal AddTax(decimal value)
{
    return value + value * TaxRate;
}
```

# 5. Avoid Magic Numbers/Strings

- Hard-coded values hide intent and fail silently.
- Don’t scatter mysterious numbers/strings in code.
- Use constants.

```
// Bad Example
int status = 404;
if (status == 404)
{
    Console.WriteLine("Not Found");
}

// Good Example
const int NotFoundStatusCode = 404;
if (status == NotFoundStatusCode)
{
    Console.WriteLine("Not Found");
}
```

# 6. Consistent Formatting

- Use a consistent style guide.
- For example: JavaScript → **Prettier / ESLint**
- Formatting tools remove debates and keep code uniform.

```
// Bad Example
public void Test(){Console.WriteLine("Hello");}

// Good Example
public void Test()
{
    Console.WriteLine("Hello");
}
```

# 7. Comment Responsibly

- Your code should explain *what* is happening
- Comments should explain why.
- If comments explain the code, refactor the code.

```
// Bad Example
int count = 0;
count++; // increment count

// Good Example
count++; // move to the next retry attempt
```

# 8. YAGNI (You Ain’t Gonna Need It)

- Don’t build features you don’t currently need.
- Implement features **only when the current requirement demands them**.

```
// Bad Example: Over-engineered for future features
public class OrderBad
{
    public decimal Amount { get; set; }
    public string DiscountCode { get; set; }  // Not needed yet
    public bool IsPremiumCustomer { get; set; } // Not needed
    public void ApplyVIPDiscount() { } // Not required now
}

// Good Example: Only implement current requirements
public class OrderGood
{
    public decimal Amount { get; set; }
}
```

# 9. Handle Errors

- Handle errors early and clearly.
- Don’t let issues hide deep in your program.

```
// Bad Example
void ProcessPaymentBad()
{
    _paymentGateway.Pay();
}

// Good Example
void ProcessPaymentGood()
{
    try
    {
        _paymentGateway.Pay();
    }
    catch (PaymentException ex)
    {
        _logger.LogError(ex, "Payment failed");
        throw;
    }
}
```

# 10. Refactor Regularly

- As requirements change, revisit and clean it up.
- Small, frequent refactors are better than giant rewrites.
- Replace long nested conditions with early returns.

```
// Bad Example
if (user != null)
{
    if (user.IsActive)
    {
        if (user.Role == "Admin")
        {
            GrantAccess();
        }
    }
}

// Good Example
if (user == null || !user.IsActive || user.Role != Roles.Admin)
    return;
GrantAccess();
```