EIU
TRƯỜNG ĐẠI HỌC
QUỐC TẾ
MIỀN ĐÔNG
EASTERN
INTERNATIONAL
UNIVERSITY

Coding Practice

Chapter 1: Clean Code

EIU

What is clean code and best practices?

What is Clean Code?

Code that is easy to read, understand, and maintain. Reflects care and craftsmanship of the developer.

Key Principles:

- Simplicity.
- Readability.
- Minimal dependencies.
- Maintenance.

Characteristics of Clean Code

- Readable: Like a well-written story.
- Simple: Avoids unnecessary complexity.
- Tested: Ensures reliability through unit and acceptance tests.
- Focused: Each piece does one thing well.
- Minimal Dependencies: Explicit and kept to a minimum.

Why Clean Code?

Clean code is essential for the long-term success and maintainability of any software project

Maintainability

- **Easier to understand**: Clean code is easier for other developers (and your future self) to read and comprehend.
- **Quicker updates**: Developers can identify the purpose of each component quickly, making updates or modifications less error-prone.

Debugging and Testing

- **Fewer bugs**: Cleaner, more organized code reduces logical errors and unexpected behaviors.
- **Simplified testing**: Well-structured code with clear functions and responsibilities is easier to write unit tests for.

Collaboration

- **Teamwork-friendly**: Clean code is easier to work on collaboratively since everyone can understand it without requiring extensive explanations.
- **Consistent style**: Adhering to clean coding principles ensures consistency, which helps team members quickly adapt to the codebase.

Why Clean Code?

Scalability

- **Handles complexity better**: Clean code adheres to good design principles (e.g., SOLID principles), ensuring the project scales gracefully as new features are added.

Cost Efficiency

- **Reduces technical debt**: Clean code minimizes the cost of future refactoring.
- **Speeds up onboarding**: New developers can get up to speed faster when the code is easy to follow.

Professionalism

- **Reflects care and expertise**: Writing clean code demonstrates professionalism and attention to detail, fostering trust among peers and stakeholders.

Best Practices for Clean Code

Meaningful Names

- **Variables, Functions, and Classes:** Use descriptive names that convey purpose. Avoid ambiguous or overly abbreviated names.
- **Naming Conventions:** Follow consistent naming conventions (e.g., camelCase for variables, PascalCase for classes).

Bad Example:

```
// 'd' does not clearly describe the meaning of the variable
int d;
```

Good Example:

```
// Variable names are clear and easy to understand
int daysSinceLastLogin;
```

Best Practices for Clean Code

Keep Functions Small

- Functions should perform a single task and be as small as possible. If a function does more than one thing, it should be split.

Bad

```
public void ProcessUserData(User user)
{
    // Check user
    if (user == null) throw new ArgumentNullException();

    // Perform complex processing logic
    // Check logic
    // Data processing logic
    // Notification sending logic
    // Logging
}
```

Good

```
public void ProcessUserData(User user)
    {
        ValidateUser(user);
        ProcessUserLogic(user);
        SendUserNotification(user);
        LogAction(user);
    }

private void ValidateUser(User user) { /* validation logic */ }
private void ProcessUserLogic(User user) { /* processing logic */ }
private void SendUserNotification(User user) { /* notification sending logic */ }
private void LogAction(User user) { /* logging logic */ }
```

Best Practices for Clean Code

Write Readable Code

- **Indentation:** Properly indent the code for better readability.
- **Whitespace:** Use blank lines to separate code logically (e.g., between different sections of a function or class).
- **Comments:** Use comments to explain *why* something is done, not *what* is done, unless the code is complex or non-obvious.
- **Avoid unnecessary comments:** Let the code be self-explanatory where possible.

Bad

```
int a = 1000; // '1000' has no clear meaning
if (a == 1000) { /* do something */ }
```

Good

```
const int MaxRetries = 3; // Clearly named constant
if (retries == MaxRetries) { /* do something */ }
```

Best Practices for Clean Code

Avoid Duplication (DRY Principle)

- Don’t Repeat Yourself. If the same logic or code is used in multiple places, extract it into a function or method.
- Use functions, methods, or classes to abstract repetitive logic.

Bad

```
public void SaveUser(User user)
    {
        if (user.Name == null) throw new ArgumentNullException();
        if (user.Email == null) throw new ArgumentNullException();
        // Logic for saving users
    }

public void SaveOrder(Order order)
    {
        if (order.Name == null) throw new ArgumentNullException();
        if (order.Email == null) throw new ArgumentNullException();
        // Logic to save orders
    }
```

Good

```
public void SaveEntity(Entity entity)
    {
        ValidateEntity(entity);
        // Logic to save entities
    }

private void ValidateEntity(Entity entity)
    {
        if (entity.Name == null) throw new ArgumentNullException();
        if (entity.Email == null) throw new ArgumentNullException();
    }
```

Best Practices for Clean Code

Error Handling

- Properly handle exceptions or errors using try-catch blocks, and ensure you provide meaningful error messages.
- Avoid silent failure (e.g., swallowing exceptions without logging them).

Bad

```
try
    {
        // Handling logic
    }
catch (Exception)
    {
        // Handle the error without logging or reporting anything
    }
```

Good

```
try
    {
        // Handling logic
    }
catch (Exception)
    {
        // Handle the error without logging or reporting anything
    }


try
    {
        // Processing logic
    }
catch (Exception ex)
    {
        LogError(ex);
        throw new CustomException("An error occurred while processing your request", ex);
    }

private void LogError(Exception ex)
    {
       // Log error details for easy debugging
    }
```

Best Practices for Clean Code

Single Responsibility Principle

- A class or function should have one responsibility. If you find yourself adding multiple concerns to a single class, refactor it.

Bad

```
0 references
public class UserManager
    {
        0 references
        public void RegisterUser(User user) { /* User registration logic */ }
        0 references
        public void SendWelcomeEmail(User user) { /* Welcome email sending logic */ }
    }
```

Good

```
1 reference
public class UserManager
    {
        1 reference
        private readonly EmailService _emailService;

        0 references
        public UserManager(EmailService emailService)
            {
                _emailService = emailService;
            }

        0 references
        public void RegisterUser(User user) { /* User registration logic */ }
    }

2 references
public class EmailService
    {
        0 references
        public void SendWelcomeEmail(User user) { /* Logic for sending welcome email */ }
    }
```

Best Practices for Clean Code

Keep It Simple (KISS Principle)

- Strive for simplicity in your code. Avoid over-complicating logic or solutions that could be solved in simpler ways.

Bad

```
public bool IsUserEligibleForDiscount(User user)
{
    if (user.Age > 18 && user.Age < 65 && user.MembershipType == "Gold" || user.MembershipType == "Platinum" || user.MembershipType == "Silver")
    {
        return true;
    }
    return false;
}
```

Good

```
public bool IsUserEligibleForDiscount(User user)
{
    var eligibleMemberships = new[] { "Gold", "Platinum", "Silver" };
    return user.Age > 18 && user.Age < 65 && eligibleMemberships.Contains(user.MembershipType);
}
```

Best Practices for Clean Code

Use Version Control Properly

- Commit code often with meaningful commit messages. This helps to track changes, revert if needed, and collaborate with others.

Bad

```
Commit message: "Fixed stuff"
```

Good

```
Commit message: "Fix issue #123: Resolve null reference exception in UserService"
Commit message: "Add validation for email format in registration form"
Commit message: "Update README.md with setup instructions"
```

Best Practices for Clean Code

9. Refactor Regularly

- Refactor your code frequently to improve readability, performance, and maintainability.
- Use automated tests to ensure refactoring does not introduce bugs.

10. Unit Testing

- Write unit tests for your functions and methods. This ensures your code works as expected and helps to identify issues early.
- Keep tests independent, isolated, and focused on one unit of work.

Best Practices for Clean Code

11. Avoid Magic Numbers and Strings

- Replace magic numbers (literal constants) or hardcoded strings with named constants or enums.

```
Bad Example:
public double CalculatePrice(double price)
{
    return price * 1.25; // What is 1.25? Magic number
}


Good Example:
public double CalculatePrice(double price)
{
    const double TaxRate = 1.25;
    return price * TaxRate;
}
```

12. Consistent Formatting

- Use consistent code formatting tools like linters or formatters to ensure code consistency across your project.

EIU

Naming conventions and styles in C# Programming Language

Best Practices for Clean Code

C# Coding Standards and Naming Conventions

Object Name	Notation	Length	Plural	Prefix	Suffix	Abbreviation	Char Mask	Underscores
Namespace name	PascalCase	128	Yes	Yes	No	No	[A-z][0-9]	No
Class name	PascalCase	128	No	No	Yes	No	[A-z][0-9]	No
Constructor name	PascalCase	128	No	No	Yes	No	[A-z][0-9]	No
Method name	PascalCase	128	Yes	No	No	No	[A-z][0-9]	No
Method arguments	camelCase	128	Yes	No	No	Yes	[A-z][0-9]	No
Local variables	camelCase	50	Yes	No	No	Yes	[A-z][0-9]	No
Constants name	PascalCase	50	No	No	No	No	[A-z][0-9]	No
Field name Public	PascalCase	50	Yes	No	No	Yes	[A-z][0-9]	No
Field name Private	_camelCase	50	Yes	No	No	Yes	_[A-z][0-9]	Yes
Properties name	PascalCase	50	Yes	No	No	Yes	[A-z][0-9]	No
Delegate name	PascalCase	128	No	No	Yes	Yes	[A-z]	No
Enum type name	PascalCase	128	Yes	No	No	No	[A-z]	No

Best Practices for Clean Code

Do use PascalCasing for class names and method names:

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

Best Practices for Clean Code

Do use camelCasing for method arguments and local variables:

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

Best Practices for Clean Code

Do not use Hungarian notation or any other type identification in identifiers

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

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

Best Practices for Clean Code

Do not use Screaming Caps for constants or readonly variables:

```
// Correct
0 references
public const string ShippingType = "DropShip";
// Avoid
0 references
public const string SHIPPINGTYPE = "DropShip";
```

Best Practices for Clean Code

Use meaningful names for variables. The following example uses binhduongCustomers for customers who are located in Binh Duong:

```
var binhduongCustomers = from customer in customers
  where customer.City == "Bình Dương"
  select customer.Name;
```

Best Practices for Clean Code

Avoid using 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;
```

Best Practices for Clean Code

Do use PascalCasing or camelCasing (Depending on the identifier type) for abbreviations 3 characters or more (2 chars are both uppercase when PascalCasing is appropriate or inside the identifier).:

```
HtmlHelper htmlHelper;
FtpTransfer ftpTransfer, fastFtpTransfer;
UIControl uiControl, nextUIControl;
```

Best Practices for Clean Code

Do not use Underscores in identifiers.
Exception: you can prefix private fields with an underscore:

```
1    // Correct
     0 references
2    public DateTime clientAppointment;
     0 references
3    public TimeSpan timeLeft;
4    // Avoid
     0 references
5    public DateTime client_Appointment;
     0 references
6    public TimeSpan time_Left;
7    // Exception (Class field)
     0 references
8    private DateTime _registrationDate;
```

Best Practices for Clean Code

Do use predefined type names (C# aliases) like int, float, string for local, parameter and member declarations. Do use .NET Framework names like Int32, Single, String when accessing the type's static members like Int32.TryParse or String.Join.

```
1   // Correct
2   string firstName;
3   int lastIndex;
4   bool isSaved;
5   string commaSeparatedNames = String.Join(", ", names);
6   int index = Int32.Parse(input);
7   // Avoid
8   String firstName;
9   Int32 lastIndex;
10  Boolean isSaved;
11  string commaSeparatedNames = string.Join(", ", names);
12  int index = int.Parse(input);
13
```

Best Practices for Clean Code

Do use implicit type var for local variable declarations. Exception: primitive types (int, string, double, etc) use predefined names.

```
1    var stream = File.Create(path);
2    var customers = new Dictionary();
3    // Exceptions
4    int index = 100;
5    string timeSheet;
6    bool isCompleted;
```

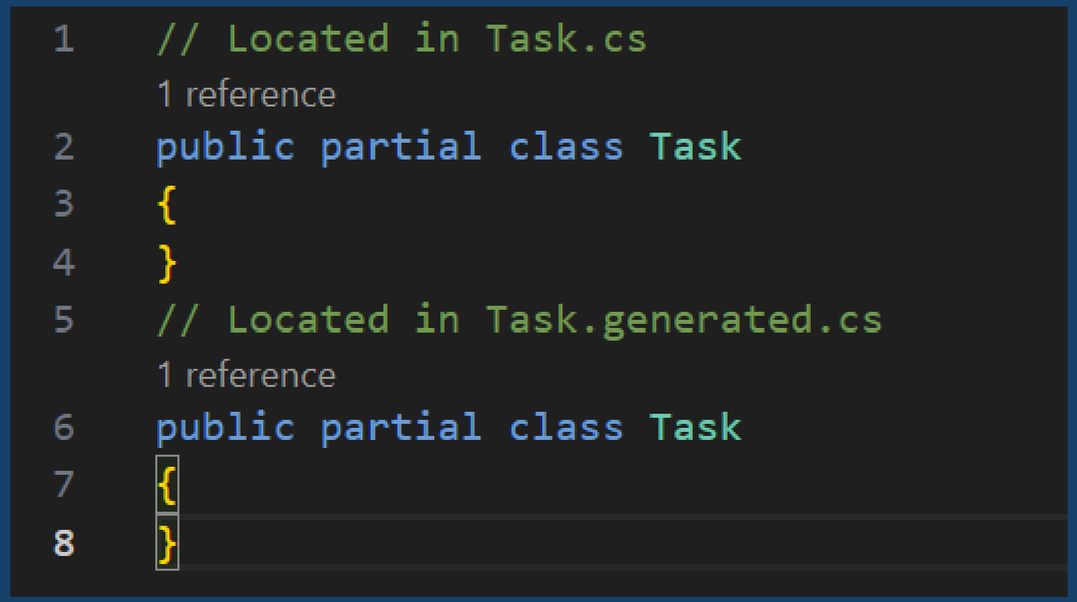
Best Practices for Clean Code

Do use noun or noun phrases to name a class.

```
0 references
1   public class Employee
2   {
3   }
    0 references
4   public class BusinessLocation
5   {
6   }
    0 references
7   public class DocumentCollection
8   {
9   }
```

Best Practices for Clean Code

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



Best Practices for Clean Code

Do organize namespaces with a clearly defined structure:

```
// Examples
namespace Company.Technology.Feature.Subnamespace
{
}
namespace Company.Product.Module.SubModule
{
}
namespace Product.Module.Component
{
}
namespace Product.Layer.Module.Group
{
}
```

Best Practices for Clean Code

Do vertically align curly brackets:

```
1   // Correct
    0 references
2   class Program
3   {
        0 references
4       static void Main(string[] args)
5       {
6           //...
7       }
8   }
9
```

Best Practices for Clean Code

Do 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()
  {
    // ...
  }
}
```

Best Practices for Clean Code

Do 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
}
```

EIU
TRƯỜNG ĐẠI HỌC
QUỐC TẾ
MIỀN ĐÔNG
EASTERN
INTERNATIONAL
UNIVERSITY

Start your future at EIU

Q&A

EIU
TRƯỜNG ĐẠI HỌC
QUỐC TẾ
MIỀN ĐÔNG
EASTERN
INTERNATIONAL
UNIVERSITY

Start your future at EIU

THANK YOU