Top defensive programming principles with examples

Explore the best defensive coding techniques and elevate your coding skills. Make your software secure and reliable.

Published on: 4/21/2024, **by:**  Piotr Bach  Wojciech Tengler

.NET Core # **Umbraco**

What is Defensive Programming?

Defensive programming is a **proactive strategy** for making software robust, secure, and reliable. It involves anticipating potential issues and implementing safeguards to prevent bugs, crashes, and vulnerabilities.

By adopting defensive programming techniques, you build applications that gracefully handle the unexpected.

💡 **Why it matters:** 85% of software bugs stem from edge cases developers didn't anticipate. Defensive coding reduces surprises and ensures your application performs as expected.

Top Defensive Programming Principles

Below are 20 practical defensive programming principles, illustrated with real-world .NET examples.

Each principle includes best practices, common pitfalls, and actionable insights to make your code bulletproof.

Validate Inputs to Prevent Errors

Always validate inputs to protect your application from bad data or misuse.

Scenario:

Imagine accepting user data for an API. Without validation, incorrect or malicious inputs can compromise your application.

Not Recommended:

```
public void SetUsername(string username)
{
    // No validation
    Console.WriteLine("Username set to " + username);
}
```

Recommended:

```
public void SetUsername(string username)
{
    if (string.IsNullOrWhiteSpace(username))
    {
        throw new ArgumentException("Username cannot be empty.");
    }

    Console.WriteLine("Username set to " + username);
}
```

💡**Key Takeaways:**

- Validate all inputs before processing.
- Use frameworks like **FluentValidation** to standardize input validation.

- Always sanitize user inputs to prevent injection attacks

Fail Fast to Spot Issues Early

Catch errors immediately to prevent them from propagating further into the system.

Scenario:

Imagine you're adding items to an inventory. If a null item is passed, the error might surface much later, causing confusion.

Not Recommended:

```
public void AddItem(Item item)
{
    // Assuming item is not null
    inventory.Add(item);
}
```

Recommended:

```
public void AddItem(Item item)
{
    if (item == null)
    {
        throw new ArgumentNullException(nameof(item), "Added item cannot be null.
    }
    inventory.Add(item);
}
```

💡Key Takeaways:

- Validate inputs early to stop errors at their source.
- Use clear exception messages to help diagnose issues quickly.

Use Assertions for Code Correctness

Assertions ensure that critical conditions hold true during execution.

Scenario:

You're calculating the area of a rectangle. Negative dimensions should trigger a warning during development.

Not Recommended:

```
public double CalculateArea(double width, double height)
{
    // No assertion
    return width * height;
}
```

Recommended:

```
public double CalculateArea(double width, double height)
{
    Debug.Assert(width > 0 && height > 0, "Width and height must be positive numb

    return width * height;
}
```

💡Key Takeaways:

- Use assertions to enforce assumptions during development.
- Avoid using assertions for runtime error handling.

Handle Exceptions Strategically

Well-designed exception handling ensures your application remains stable during errors.

Scenario:

Processing a file fails, and the application crashes without logging the error or notifying the user.

Not Recommended:

```
try
{
    ProcessFile("data.txt");
}
catch
{
    // Generic catch, does nothing
    Console.WriteLine("An error occurred.");
}
```

Recommended:

```
try
{
    ProcessFile("data.txt");
}
catch (IOException ex)
{
    Log.Error("Failed to process the file.", ex);
```

```
    NotifyUser("An error occurred. Please try again later.");
}
```

💡**Key Takeaways:**

- Catch specific exceptions for better diagnostics.
- Log failures and provide meaningful feedback to users.

Limit Variable Scope

Reducing variable scope makes code cleaner and avoids unintended side effects.

Scenario:

A loop counter is declared outside its loop, remaining accessible where it's no longer needed.

Not Recommended:

```
int i;
for (i = 0; i < users.Count; i++)
{
    Console.WriteLine(users[i]);
}
// i is still accessible here
```

Recommended:

```
for (int i = 0; i < users.Count; i++)
{
    Console.WriteLine(users[i]);
```

```
    // 'i' is limited to the scope of this loop
}
```

💡**Key Takeaways:**

- Declare variables in the smallest possible scope.
- Avoid global or unnecessarily broad variable declarations.

Default to Secure Settings

Design applications with security as the default, reducing the risk of vulnerabilities.

Scenario:

An API client is created with insecure defaults, allowing data to be transmitted without encryption.

Not Recommended:

```
public class ApiClient
{
    public bool UseEncryption = false;  // Insecure default

    public void SendData(string data)
    {
        if (UseEncryption)
        {
            data = Encrypt(data);
        }

        // Send data
    }
}
```

Recommended:

```
public class ApiClient
{
    public bool UseEncryption { get; set; } = true;  // Secure default

    public void SendData(string data)
    {
        if (UseEncryption)
        {
            data = Encrypt(data);
        }

        // Send data
    }
}
```

💡**Key Takeaways:**

- Set secure configurations as defaults.
- Allow flexibility for advanced users to override defaults when necessary.

Sanitize Data to Prevent Injection Attacks

Always clean user input to prevent malicious attacks like SQL injection or XSS.

Scenario:

A greeting message directly incorporates unvalidated user input, allowing script injection.

Not Recommended:

```
public string CreateGreeting(string name)
{
    // Direct use of user input
```

```
    return $"Hello, {name}!";
}
```

Recommended:

```
public string CreateGreeting(string name)
{
    name = name.Replace("<script>", "");  // Basic sanitization example

    return $"Hello, {name}!";
}
```

💡**Key Takeaways:**

- Always sanitize user input before processing.
- Use specialized libraries for input validation and sanitization where possible.

Avoid Magic Numbers

Replace hard-coded values with constants to make your code more understandable and maintainable.

Scenario:

A hard-coded user type value makes the code harder to read and update.

Not Recommended:

```
if (userType == 1) // Magic number
{
```

```
    // Grant admin access
}
```

Recommended:

```
const int AdminUserType = 1;
if (userType == AdminUserType)
{
    // Grant admin access
}
```

💡**Key Takeaways:**

- Use named constants for better readability and maintainability.
- Avoid hard-coding values in multiple places to prevent errors during updates.

Encapsulate Data

Encapsulation protects object integrity by controlling how data is accessed and modified.

Scenario:

A public field exposes sensitive data directly, leading to potential misuse.

Not Recommended:

```
public class Account
{
    public decimal Balance; // Publicly accessible
}
```

Recommended:

```
public class Account
{
    private decimal balance;

    public decimal GetBalance()
    {
        return balance;
    }
}
```

💡**Key Takeaways:**

- Use private fields with controlled access via methods or properties.
- Encapsulation ensures data consistency and reduces unintended changes.

Use Immutable Objects to Avoid Unwanted Changes

Immutable objects help eliminate bugs caused by unexpected state changes, making your code safer and more predictable.

Scenario:

A mutable object allows direct modification of its properties, leading to unpredictable application behavior.

Not Recommended:

```
public class UserInfo
{
    public string Username { get; set; }
```

```
    public DateTime BirthDate { get; set; }
}
```

Recommended:

```
public class UserInfo
{
    public string Username { get; }
    public DateTime BirthDate { get; }

    public UserInfo(string username, DateTime birthDate)
    {
        Username = username;
        BirthDate = birthDate;
    }
}
```

💡**Key Takeaways:**

- Use immutable objects for better consistency.
- Design constructors to enforce valid states at the time of object creation.

Always Check Return Values to Avoid Errors

Never assume a method call will succeed - always verify its result.

Scenario:

An unchecked method call might fail silently, leading to undefined behavior later in your application.

Not Recommended:

```
int number;
int.TryParse(input, out number);
// Uses number without checking if the parsing was successful
```

Recommended:

```
if (int.TryParse(input, out int number))
{
    Console.WriteLine($"Parsed number: {number}");
}
else
{
    Console.WriteLine("Invalid number entered.");
}
```

💡**Key Takeaways:**

- Always handle method return values, especially for parsing and data retrieval.
- Use conditional logic to deal with failure scenarios gracefully.

Practice the Principle of Least Privilege

Grant users and processes only the permissions they need to perform specific tasks.

Scenario:

A method unnecessarily requires elevated privileges, increasing the risk of abuse.

Not Recommended:

```
public void DeleteUser(int userId)
{
    // Requires higher privileges
    // Code that deletes user
}
```

Recommended:

```
public void DeleteUser(int userId)
{
    if (UserHasPermission("DeleteUser"))
    {
        // Code that deletes user
    }
    else
    {
        throw new UnauthorizedAccessException("User does not have delete permissi
    }
}
```

💡**Key Takeaways:**

- Minimize the permissions required for each operation.
- Always validate user roles and permissions before executing sensitive actions.

Implement Effective Logging and Monitoring

Use detailed logging to track application behavior and diagnose issues.

Scenario:

A payment process fails, but without proper logging, the root cause remains unclear.

Not Recommended:

```
public void ProcessPayment(decimal amount)
{
    try
    {
        // No logging of payment process steps
        Console.WriteLine("Processing payment.");
    }
    catch (Exception ex)
    {
        Console.WriteLine("Payment failed.");
    }
}
```

Recommended:

```
public void ProcessPayment(decimal amount)
{
    try
    {
        Log.Info($"Starting payment processing for amount: {amount}");
        // Payment processing logic
        Log.Info("Payment processed successfully.");
    }
    catch (Exception ex)
    {
        Log.Error("Payment processing failed.", ex);
    }
}
```

💡**Key Takeaways:**

- Use structured logging tools like **Serilog** or **NLog**.

- Log at different levels: Info for routine operations and Error for critical failures.

Use Parameterized Queries to Secure Your Database

Parameterized queries protect your database from SQL injection attacks.

Scenario:

An attacker injects SQL into a query string, bypassing authentication and accessing sensitive data.

Not Recommended:

```
public void AddUser(string username, string password)
{
    var query = $"INSERT INTO Users (Username, Password) VALUES ('{username}', '{
    // Unsafe query construction
}
```

Recommended:

```
public void AddUser(string username, string password)
{
    using (var connection = new SqlConnection(connectionString))
    {
        var command = new SqlCommand("INSERT INTO Users (Username, Password) VALU
        command.Parameters.AddWithValue("@Username", username);
        command.Parameters.AddWithValue("@Password", password);
        connection.Open();
        command.ExecuteNonQuery();
    }
}
```

💡**Key Takeaways:**

- Never concatenate user input into SQL queries.
- Always validate user input even when using parameterized queries.

Regularly Update Your Dependencies to Mitigate Vulnerabilities

Stay current with library and framework updates to prevent security risks.

Scenario:

An outdated library introduces a vulnerability that attackers exploit to compromise your system.

Not Recommended:

Ignoring update alerts or skipping dependency reviews.

Recommended:

Review dependencies periodically and update to the latest stable versions. To simplify the process, use tools like Dependabot or NuGet Package Manager.

💡**Key Takeaways:**

- Keep a regular update schedule for libraries and frameworks.
- Test thoroughly after updates to prevent regressions or compatibility issues.

Utilize Strong Typing to Prevent Bugs

Strong typing reduces runtime errors by enforcing type safety during compilation.

Scenario:

Incorrect type casting causes a runtime error that’s hard to trace.

Not Recommended:

```
object data = "123";
int number = (int)data; // Runtime error if unchecked
```

Recommended:

```
if (int.TryParse("123", out int number))
{
    Console.WriteLine($"Successfully parsed number: {number}");
}
else
{
    Console.WriteLine("Failed to parse the number.");
}
```

💡**Key Takeaways:**

- Use strong typing and safe parsing methods like TryParse.
- Avoid relying on loosely typed variables like object.

Opt for Non-executable Stacks to Enhance Security

Prevent buffer overflow attacks by disabling code execution in stack memory.

Scenario:

An application allows executable stacks, making it vulnerable to injected code execution.

Not Recommended:

Allowing executable stacks in environments where security is critical.

Recommended:

Enable non-executable stacks in the operating system and hosting environments:

1. Use Data Execution Prevention (DEP) in Windows.

2. Enable the NX (No-eXecute) bit on supported processors.

3. Configure Address Space Layout Randomization (ASLR) for added protection.

💡**Key Takeaways:**

- Enforce non-executable stacks to mitigate buffer overflow attacks.
- Regularly audit system settings for compliance with best practices.

Conduct Thorough Code Reviews for Quality Assurance

Code reviews ensure that potential bugs, security vulnerabilities, and inefficiencies are identified before changes are merged into the main codebase.

Scenario:

Skipping code reviews under tight deadlines can lead to unchecked vulnerabilities or poorly written code making it to production.

Not Recommended:

Skipping code reviews due to tight deadlines or lack of resources.

Recommended:

Require at least one peer review for every change before merging it into the main branch. Use tools like GitHub or GitLab for effective collaboration during reviews.

💡**Key Takeaways:**

- Ensure at least one other developer reviews every change.
- Use code review tools to enforce standards and catch errors early.
- Code reviews improve quality and facilitate knowledge sharing among team members.

Minimize the Use of Global Variables to Reduce Side Effects

Global variables make code harder to maintain and debug due to unintended interactions between different parts of your application.

Scenario:

A global variable is modified in one part of the program, causing unexpected behavior elsewhere.

Not Recommended:

```
public static string CurrentUser; // Global variable
```

Recommended:

```
public class Session
{
    public string CurrentUser { get; private set; } // Encapsulated within a clas
}
```

💡Key Takeaways:

- Avoid global state wherever possible.
- Use dependency injection or encapsulate variables within classes to ensure modularity and testability.

Build Redundancy and Implement Fail-Safe Measures for Reliability

Incorporate redundancy and fail-safes to ensure your application continues functioning even during failures.

Scenario:

Your backup process relies on a single execution attempt. If it fails, no data is backed up.

Not Recommended:

```
public void PerformBackup()
{
    // Single point of failure in backup strategy
    BackupData();
}
```

If BackupData fails, the entire backup process stops without any fallback or retry.

Recommended:

```
public void BackupData()
{
    try
    {
        PerformBackup();
    }
    catch
    {
        PerformBackup();  // Try again if the first attempt fails
```

```
    }
}
```

💡**Key Takeaways:**

- **Retry Logic:** Attempt multiple retries for transient failures before giving up.
- **Fail-Safe Alerts:** Notify administrators or log critical errors if all retries fail.
- **Improved Reliability:** Redundancy ensures a higher chance of successful recovery from failures.