

- Secrets
- ABAP
- Apex
- C
- C++
- CloudFormation
- COBOL
- C#**
- CSS
- Flex
- Go
- HTML
- Java
- JavaScript
- Kotlin
- Objective C
- PHP
- PL/I
- PL/SQL
- Python
- RPG
- Ruby
- Scala
- Swift
- Terraform
- Text
- TypeScript
- T-SQL
- VB.NET
- VB6
- XML



## C# static code analysis

Unique rules to find Bugs, Vulnerabilities, Security Hotspots, and Code Smells in your C# code

All rules 409

Vulnerability 34

Bug 76

Security Hotspot 28

Code Smell 271

Quick Fix 52

Tags

Search by name...



checked for direct equality

Code Smell

"for" loop increment clauses should modify the loops' counters

Code Smell

"switch" statements should not be nested

Code Smell

Methods and properties should not be too complex

Code Smell

Control flow statements "if", "switch", "for", "foreach", "while", "do" and "try" should not be nested too deeply

Code Smell

"switch/Select" statements should contain a "default/Case Else" clauses

Code Smell

"if ... else if" constructs should end with "else" clauses

Code Smell

Control structures should use curly braces

Code Smell

Expressions should not be too complex

Code Smell

ASP.NET HTTP request validation feature should not be disabled

Vulnerability

Serialization constructors should be secured

Vulnerability

Calculations should not overflow

### "IDisposable" should be implemented correctly

Analyze your code

Code Smell Major pitfall

The `IDisposable` interface is a mechanism to release unmanaged resources, if not implemented correctly this could result in resource leaks or more severe bugs.

This rule raises an issue when the recommended dispose pattern, as defined by Microsoft, is not adhered to. See the **Compliant Solution** section for examples.

Satisfying the rule's conditions will enable potential derived classes to correctly dispose the members of your class:





- sealed classes are not checked.
- If a base class implements `IDisposable` your class should not have `IDisposable` in the list of its interfaces. In such cases it is recommended to override the base class's protected virtual `void Dispose(bool)` method or its equivalent.
- The class should not implement `IDisposable` explicitly, e.g. the `Dispose()` method should be public.
- The class should contain protected virtual `void Dispose(bool)` method. This method allows the derived classes to correctly dispose the resources of this class.
- The content of the `Dispose()` method should be invocation of `Dispose(true)` followed by `GC.SuppressFinalize(this)`
- If the class has a finalizer, i.e. a destructor, the only code in its body should be a single invocation of `Dispose(false)`.
- If the class inherits from a class that implements `IDisposable` it must call the `Dispose`, or `Dispose(bool)` method of the base class from within its own implementation of `Dispose` or `Dispose(bool)`, respectively. This ensures that all resources from the base class are properly released.

#### Noncompliant Code Example

```
public class Foo1 : IDisposable // Noncompliant - provide pr
{
    public void Dispose() // Noncompliant - should contain o
    {
        // Cleanup
    }
}

public class Foo2 : IDisposable
{
    void IDisposable.Dispose() // Noncompliant - Dispose() s
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    public virtual void Dispose() // Noncompliant - Dispose(
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
}
```

 Bug
<b>Floating point numbers should not be tested for equality</b>  Bug
<b>Increment (++) and decrement (--) operators should not be used in a method call or mixed with other operators in an expression</b>  Code Smell
<b>Use a testable date/time provider.</b>  Code Smell
<b>Property names should not match get methods</b>

```
public class Foo3 : IDisposable
{
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        // Cleanup
    }

    ~Foo3() // Noncompliant - Modify Foo.~Foo() so that it c
    {
        // Cleanup
    }
}{code}
```

Compliant Solution

```
// Sealed class
public sealed class Foo1 : IDisposable
{
    public void Dispose()
    {
        // Cleanup
    }
}

// Simple implementation
public class Foo2 : IDisposable
{
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        // Cleanup
    }
}

// Implementation with a finalizer
public class Foo3 : IDisposable
{
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        // Cleanup
    }

    ~Foo3()
    {
        Dispose(false);
    }
}

// Base disposable class
public class Foo4 : DisposableBase
{
    protected override void Dispose(bool disposing)
    {
        // Cleanup
        // Do not forget to call base
        base.Dispose(disposing);
    }
}
```

## See

### Refer to

- [MSDN](#) for complete documentation on the dispose pattern.
- [Stephen Cleary](#) for excellent Q&A about IDisposable
- [Pragma Geek](#) for additional usages of IDisposable, beyond releasing resources.
- [IDisposable documentation](#)

Available In:

**sonarlint**  | **sonarcloud**  | **sonarqube** 