

- Secrets
- ABAP
- Apex
- C
- C++
- CloudFormation
- COBOL
- C# 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... 🔍

Code Smell

Locales should be set for data types

Code Smell

Literals should not be passed as localized parameters

Code Smell

Operators should be overloaded consistently

Code Smell

Method signatures should not contain nested generic types

Code Smell

Enumeration members should not be named "Reserved"

Code Smell

"System.Uri" arguments should be used instead of strings

Code Smell

Collection properties should be readonly

Code Smell

Disposable types should declare finalizers

Code Smell

String URI overloads should call "System.Uri" overloads

Code Smell

URI properties should not be strings

Code Smell

URI return values should not be strings

Code Smell

Generic type parameters should be co/contravariant when possible

Analyze your code

Code Smell Major ? api-design

In the interests of making code as usable as possible, interfaces and delegates with generic parameters should use the `out` and `in` modifiers when possible to make the interfaces and delegates covariant and contravariant, respectively.

The `out` keyword can be used when the type parameter is used only as a return type in the interface or delegate. Doing so makes the parameter covariant, and allows interface and delegate instances created with a sub-type to be used as instances created with a base type. The most notable example of this is `IEnumerable<out T>`, which allows the assignment of an `IEnumerable<string>` instance to an `IEnumerable<object>` variable, for instance.

The `in` keyword can be used when the type parameter is used only as a method parameter in the interface or a parameter in the delegate. Doing so makes the parameter contravariant, and allows interface and delegate instances created with a base type to be used as instances created with a sub-type. I.e. this is the inversion of covariance. The most notable example of this is the `Action<in T>` delegate, which allows the assignment of an `Action<object>` instance to a `Action<string>` variable, for instance.

Noncompliant Code Example





```
interface IConsumer<T> // Noncompliant
{
    bool Eat(T fruit);
}
```

Compliant Solution

```
interface IConsumer<in T>
{
    bool Eat(T fruit);
}
```

Available In:

sonarlint | sonarcloud | sonarqube

URI Parameters should not be strings  Code Smell
Custom attributes should be marked with "System.AttributeUsageAttribute"  Code Smell
Assemblies should explicitly specify COM visibility  Code Smell
Assemblies should be marked as CLS compliant  Code Smell
"Generic.List" instances should not be