Understand Metrics

Table of Contents

1. All Methods
2. Auto-Implemented Properties
3. Average Blank Lines
4. Average Blank Lines (Includes Inactive)
5. Average Code Lines
6. Average Code Lines (Includes Inactive)
7. Average Comment Lines
8. Average Comment Lines (Includes Inactive)
9. Average Cyclomatic Complexity
10. Average Essential Complexity
11. Average Lines
12. Average Modified Cyclomatic Complexity
13. Average Strict Cyclomatic Complexity
14. Average Strict Modified Cyclomatic Complexity
15. Average Strict Modified Essential Complexity
16. Base Classes
17. Blank Lines
18. Blank Lines (HTML)
19. Blank Lines (Includes Inactive)
20. Blank Lines (JavaScript)
21. Blank Lines (PHP)
22. Changed Lines
23. Class Methods
24. Class Variables
25. Classes
26. Code Files
27. Code Lines
28. Code Lines (Includes Inactive)
29. Code Lines (JavaScript)
30. Code Lines (PHP)
31. CodeCheck Violation Density by Code Lines
32. CodeCheck Violation Density by Lines
33. CodeCheck Violation Types
34. CodeCheck Violations
35. Comment Lines
36. Comment Lines (HTML)
37. Comment Lines (Includes Inactive)
38. Comment Lines (JavaScript)
39. Comment Lines (PHP)
40. Comment to Code Ratio

41. Const Methods
42. Coupled Classes
43. Coupled Classes Modified
44. Coupled Packages
45. Cyclomatic Complexity
46. Declarative Code Lines
47. Declarative Statements
48. Declarative Statements (JavaScript)
49. Declarative Statements (PHP)
50. Default Methods
51. Derived Classes
52. Empty Statements
53. Essential Complexity
54. Executable Code Lines
55. Executable Statements
56. Executable Statements (JavaScript)
57. Executable Statements (PHP)
58. Executable Units
59. Files
60. Friend Methods
61. Functions
62. Header Files
63. Inactive Lines
64. Inputs
65. Instance Methods
66. Instance Variables
67. Internal Instance Variables
68. Internal Methods
69. Knots
70. Lines
71. Lines (HTML)
72. Lines (JavaScript)
73. Lines (PHP)
74. Max Cyclomatic Complexity
75. Max Essential Complexity
76. Max Essential Knots
77. Max Inheritance Tree
78. Max Modified Cyclomatic Complexity
79. Max Nesting
80. Max Strict Cyclomatic Complexity
81. Max Strict Modified Cyclomatic Complexity
82. Max Strict Modified Essential Complexity
83. Methods
84. Min Essential Knots
85. Modified Cyclomatic Complexity

86. Modules
87. New Lines
88. Outputs
89. Paths
90. Paths Log(x)
91. Percent Changed
92. Percent Lack Of Cohesion
93. Percent Lack Of Cohesion Modified
94. Preprocessor Lines
95. Private Instance Variables
96. Private Methods
97. Program Units
98. Properties
99. Protected Instance Variables
100. Protected Internal Instance Variables
101. Protected Internal Methods
102. Protected Methods
103. Public Instance Variables
104. Public Methods
105. Published Methods
106. Removed Lines
107. Semicolons
108. Statements
109. Strict Cyclomatic Complexity
110. Strict Modified Cyclomatic Complexity
111. Strict Modified Essential Complexity
112. Strict Private Methods
113. Strict Protected Methods
114. Subprograms
115. Sum Cyclomatic Complexity
116. Sum Essential Complexity
117. Sum Modified Cyclomatic Complexity
118. Sum Strict Cyclomatic Complexity
119. Sum Strict Modified Cyclomatic Complexity
120. Sum Strict Modified Essential Complexity

All Methods

API ID: CountDeclMethodAll
Also Known As: Number of Methods (NM), Response for a Class (RFC)
Languages: Basic, C#, C++, Java, Pascal, Python, Web
Targets: Classes

Number of methods, including inherited ones.

For example:

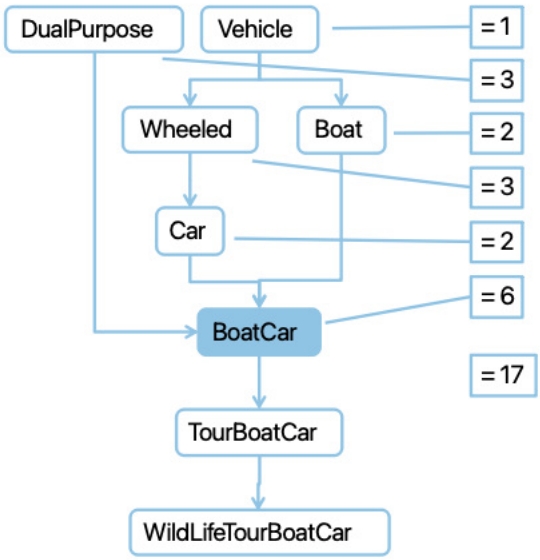
```
1 class Vehicle {
2 public:
3   Vehicle() {}
4 };
5
6 class Wheeled : public Vehicle {
7 public:
8   Wheeled(int wheels) : mWheels(wheels) {}
9   int wheels() { return mWheels; }
10   void setWheels(int wheels) { mWheels = wheels; }
11 private:
12   int mWheels;
13 };
14
15 class Car : public Wheeled {
16 public:
17   Car() : Wheeled(4) {}
18   Car(int wheels) : Wheeled(wheels) {}
19 };
20
21 class Boat : public Vehicle {
22 public:
23   Boat() {}
24   ~Boat() {}
25 };
26
27 class DualPurpose {
28 public:
29   DualPurpose() {}
30   char* purpose1() { return nullptr; }
31   char* purpose2() { return nullptr; }
32 };
33
34 class BoatCar: private Car, public Boat, protected DualPurpose {
35 public:
36   BoatCar() : Car(4), Boat(), mInWater(false), mColor("Blue") {}
37   virtual int passengers() const { return 4; }
38
39   static int numRegistered() { return sRegistered; }
40
41   bool mInWater;
42
43 protected:
44
45   void toggleInWater(bool inWater) { mInWater = inWater; }
46   char * mColor;
47   friend void init() {}
48
49   static int sRegistered;
50   static double calcSpeed(double distance, double time) {
```

```
51      return distance/time;
52    }
53
54  private:
55    int mMaxPassengers;
56    void travel() {}
57  };
58
59  class TourBoatCar : public BoatCar {};
60  class WildLifeTourBoatCar: public TourBoatCar {};
```

BoatCar writes 6 methods of its own and multiply inherits from Car (which itself derives from Wheeled, which derives from Vehicle), Boat (which also derives from Vehicle), and DualPurpose. Counting the transitive union of all of those methods — and only counting Vehicle's single constructor once even though it's reachable through both the Car/Wheeled chain and the Boat chain — gives CountDeclMethodAll = 17 for BoatCar. TourBoatCar and WildLifeTourBoatCar add no methods of their own, so they inherit everything BoatCar has unchanged: both also show CountDeclMethodAll = 17.

See Methods to count only methods declared locally in the class itself.

Also known as Chidamber & Kemerer - Response for a Class (RFC); Lorenz & Kidd - Number of Methods (NM).



Targets By Language:

- **Basic:** Class, Struct

- **C#:** Class, Struct
- **C++:** Class, Struct, Union
- **Java:** Class, Interface
- **Pascal:** Class, Interface
- **Python:** Class
- **Web:** PHP Class, PHP Interface

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Auto-Implemented Properties

API ID: CountDeclPropertyAuto
Languages: C#
Targets: Architectures, Classes, Project

Number of auto-implemented properties.

An auto-implemented property (C#'s { get; set; } shorthand) has its backing field generated automatically by the compiler, rather than declared and managed explicitly.

For example:

```
1 namespace MetricsDemo
2 {
3     public class PropertyDemo
4     {
5         public int X { get; set; }
6
7         private int y;
8         public int Y
9         {
10             get { return y; }
11             set { y = value; }
12         }
13     }
14 }
```

X is auto-implemented, giving CountDeclPropertyAuto = 1. Y has an explicit backing field (y) and hand-written accessor bodies, so it doesn't count here even though it's still a property.

See Properties for the total including explicitly-implemented properties like Y.

Targets By Language:

- **C#:** Project, Class, Struct

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated class metrics" option. Enabling this option can be time consuming.
- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Average Blank Lines

API ID: AvgCountLineBlank
Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, Web
Targets: Classes, Files, Modules, Packages

Average number of blank lines for all nested functions or methods.

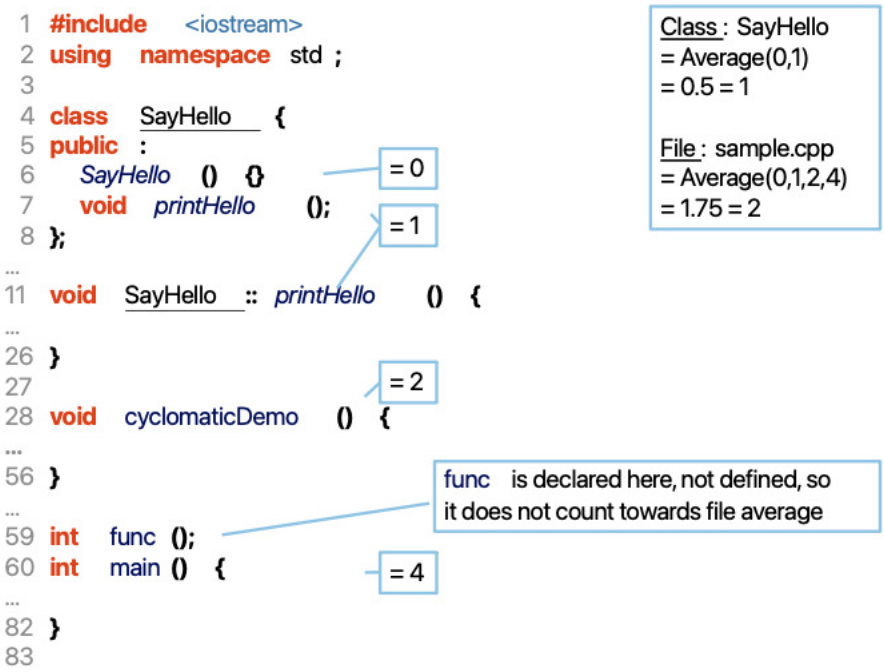
Only functions and methods that are actually defined (not just declared) directly inside the container count towards the average, and the result rounds to the nearest whole number.

For example:

```
1 class SayHello {
2 public:
3   SayHello() {}
4   void printHello();
5 };
6
7 void SayHello::printHello() {
8   ...
9 }
10
11 void cyclomaticDemo() {
12   ...
13 }
14
15 int func();
16 int main() {
17   ...
18 }
```

SayHello() has 0 blank lines and SayHello::printHello() has 1, so class SayHello's Average Blank Lines is Average(0, 1) = 0.5, which rounds to 1. The file also includes cyclomaticDemo() (2 blank lines) and main() (4 blank lines): Average(0, 1, 2, 4) = 1.75, which rounds to 2. func() is declared here but never defined, so it contributes nothing to either average.

To also count blank lines inside inactive regions before averaging, use Average Blank Lines (Includes Inactive). See Blank Lines for the per-function count this averages.



Targets By Language:

- **Ada:** File, Package
- **Basic:** File, Module, Class, Struct
- **C#:** File, Class, Struct
- **C++:** File, Class, Struct, Union
- **Fortran:** File
- **Java:** File, Class, Interface
- **Jovial:** File
- **Pascal:** File, Class, Interface
- **Python:** File, Class
- **Rust:** File
- **Web:** File, PHP Class, PHP Interface

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Include inactive lines" option for languages that have inactive lines.

Average Blank Lines (Includes Inactive)

API ID: AvgCountLineBlankWithInactive
Languages: C++
Targets: Classes, Files

Average number of blank lines for all nested functions or methods, including inactive regions.

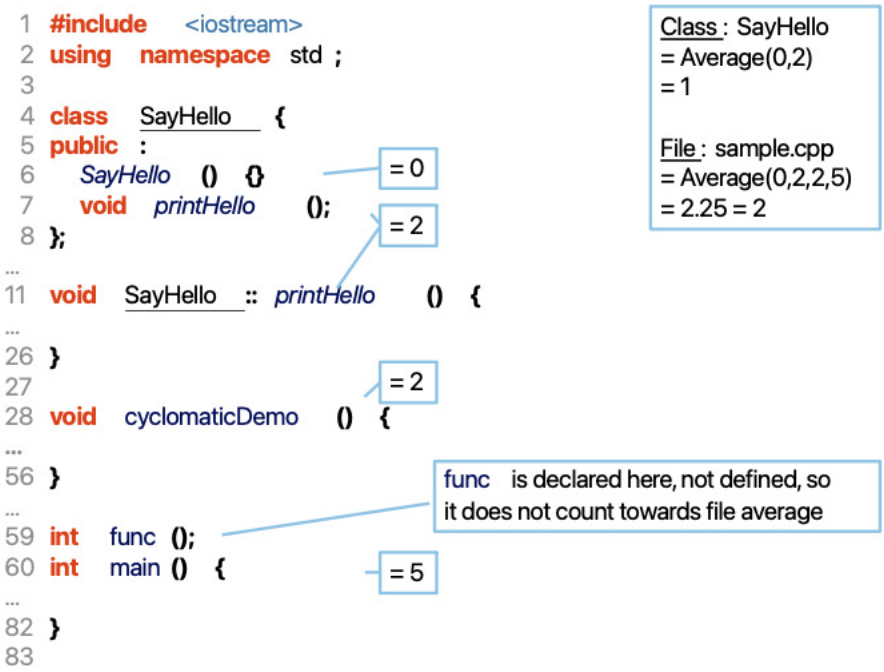
Unlike Average Blank Lines, this counts each function's blank lines inside inactive #if/#ifdef regions too, before averaging. Only functions and methods that are actually defined (not just declared) directly inside the container count towards the average, and the result rounds to the nearest whole number.

For example:

```
1 class SayHello {
2 public:
3   SayHello() {}
4   void printHello();
5 };
6
7 void SayHello::printHello() {
8   ...
9 }
10
11 void cyclomaticDemo() {
12   ...
13 }
14
15 int func();
16 int main() {
17   ...
18 }
```

SayHello() has 0 blank lines (including inactive) and SayHello::printHello() has 2 (one of its blank lines falls inside an inactive #ifdef block), so class SayHello's average is Average(0, 2) = 1. The file also includes cyclomaticDemo() (2) and main() (5, which also has an inactive blank line): Average(0, 2, 2, 5) = 2.25, which rounds to 2.

See Blank Lines (Includes Inactive) for the per-function count this averages.



Targets By Language:

- **C++:** File, Class, Struct, Union

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Lines and Statements" with the "Include inactive lines" option.

Average Code Lines

API ID: AvgCountLineCode
Also Known As: Average Method Size (AMS)
Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, Web
Targets: Classes, Files, Modules, Packages

Average number of lines containing source code for all nested functions or methods.

Only functions and methods that are actually defined (not just declared) directly inside the container count towards the average, and the result rounds to the nearest whole number.

For example:

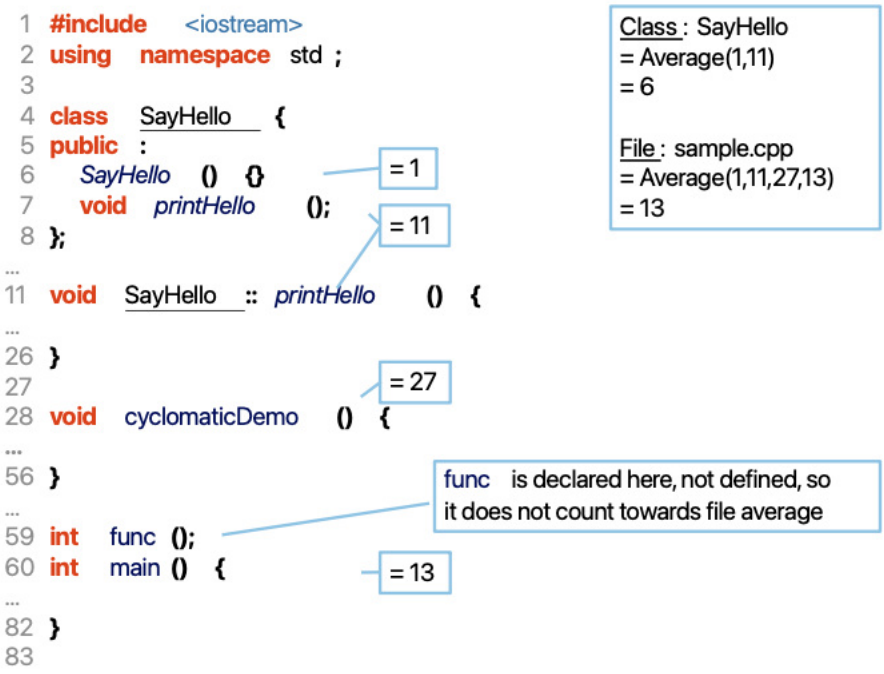
```
1 class SayHello {
```

```
2 public:
3   SayHello() {}
4   void printHello();
5 };
6
7 void SayHello::printHello() {
8   ...
9 }
10
11 void cyclomaticDemo() {
12   ...
13 }
14
15 int func();
16 int main() {
17   ...
18 }
```

SayHello() has 1 code line and SayHello::printHello() has 11, so class SayHello's Average Code Lines is Average(1, 11) = 6. The file also includes cyclomaticDemo() (27 code lines) and main() (14 code lines): Average(1, 11, 27, 14) = 13.25, which rounds to 13. func() is declared here but never defined, so it contributes nothing to either average.

See Code Lines for the per-function count this averages, and Average Code Lines (Includes Inactive) to also count inactive code lines before averaging.

Also known as Lorenz & Kidd - Average Method Size (AMS).



Targets By Language:

- **Ada:** File, Package
- **Basic:** File, Module, Class, Struct
- **C#:** File, Class, Struct
- **C++:** File, Class, Struct, Union
- **Fortran:** File
- **Java:** File, Class, Interface
- **Jovial:** File
- **Pascal:** File, Class, Interface
- **Python:** File, Class
- **Rust:** File
- **Web:** File, PHP Class, PHP Interface

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Include inactive lines" option for languages that have inactive lines.

Average Code Lines (Includes Inactive)

API ID: AvgCountLineCodeWithInactive
Languages: C++
Targets: Classes, Files

Average number of lines containing source code for all nested functions or methods, including inactive regions.

Unlike Average Code Lines, this counts each function's code lines inside inactive #if/#ifdef regions (and the preprocessor conditional lines guarding them) too, before averaging. Only functions and methods that are actually defined (not just declared) directly inside the container count towards the average, and the result rounds to the nearest whole number.

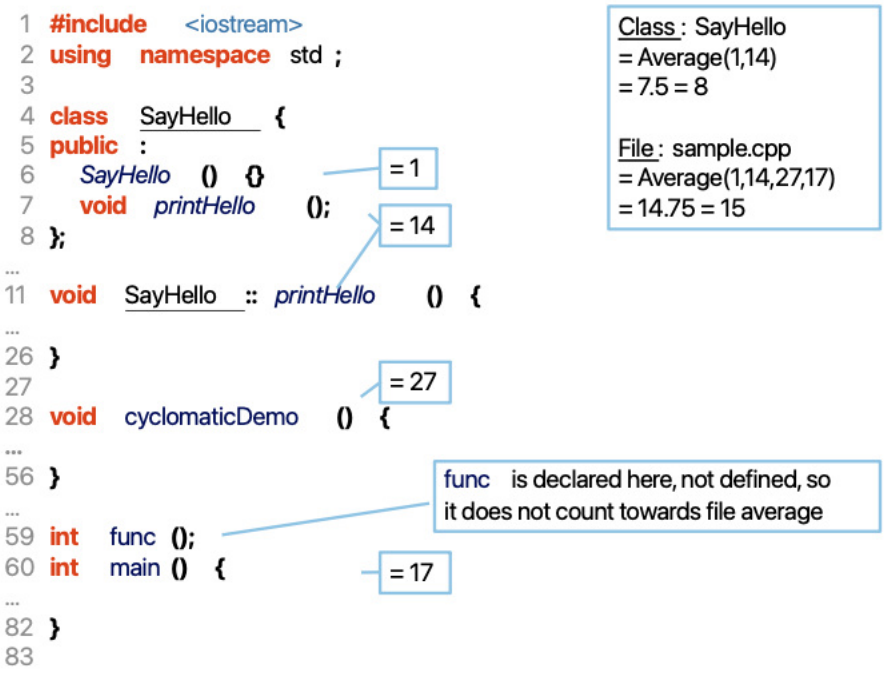
For example:

```
1 class SayHello {
2 public:
3   SayHello() {}
4   void printHello();
5 };
6
7 void SayHello::printHello() {
8   ...
```

```
9 }
10
11 void cyclomaticDemo() {
12   ...
13 }
14
15 int func();
16 int main() {
17   ...
18 }
```

SayHello() has 1 code line (including inactive) and SayHello::printHello() has 14 (its inactive block adds 1 inactive code line plus 2 preprocessor lines), so class SayHello's average is Average(1, 14) = 7.5, which rounds to 8. The file also includes cyclomaticDemo() (27) and main() (18, which has its own inactive block): Average(1, 14, 27, 18) = 15.

See Code Lines (Includes Inactive) for the per-function count this averages.



Targets By Language:

- **C++:** File, Class, Struct, Union

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Lines and

Statements" with the "Include inactive lines" option.

Average Comment Lines

API ID: AvgCountLineComment
Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, Web
Targets: Classes, Files, Modules, Packages

Average number of lines containing comment for all nested functions or methods.

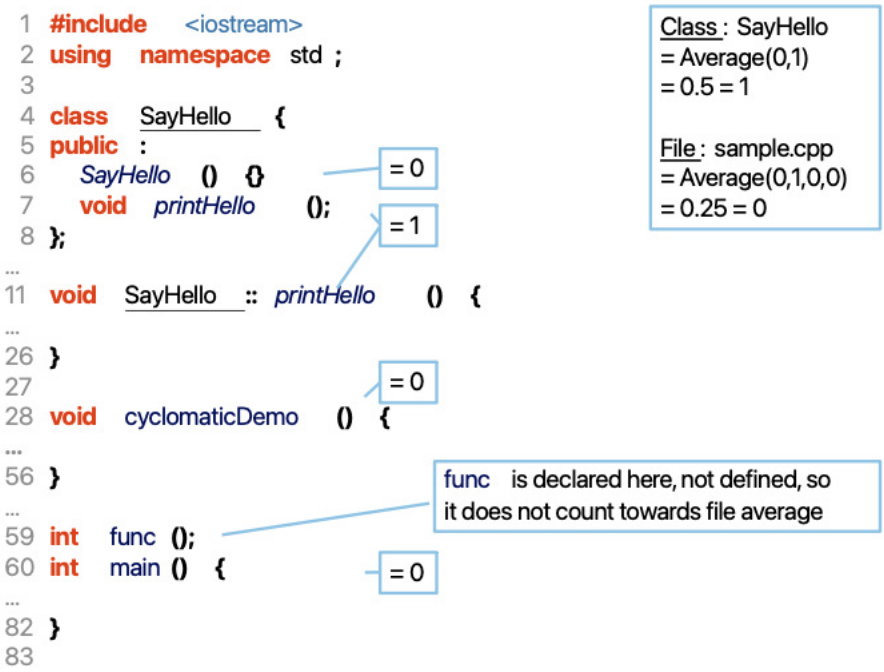
Only functions and methods that are actually defined (not just declared) directly inside the container count towards the average, and the result rounds to the nearest whole number.

For example:

```
1 class SayHello {
2 public:
3   SayHello() {}
4   void printHello();
5 };
6
7 void SayHello::printHello() {
8   ...
9 }
10
11 void cyclomaticDemo() {
12   ...
13 }
14
15 int func();
16 int main() {
17   ...
18 }
```

SayHello() has 0 comment lines and SayHello::printHello() has 1, so class SayHello's Average Comment Lines is Average(0, 1) = 0.5, which rounds to 1. The file also includes cyclomaticDemo() and main(), neither of which has any comments: Average(0, 1, 0, 0) = 0.25, which rounds to 0. func() is declared here but never defined, so it contributes nothing to either average.

To also count comments inside inactive regions before averaging, use Average Comment Lines (Includes Inactive). See Comment Lines for the per-function count this averages.



Targets By Language:

- **Ada:** File, Package
- **Basic:** File, Module, Class, Struct
- **C#:** File, Class, Struct
- **C++:** File, Class, Struct, Union
- **Fortran:** File
- **Java:** File, Class, Interface
- **Jovial:** File
- **Pascal:** File, Class, Interface
- **Python:** File, Class
- **Rust:** File
- **Web:** File, PHP Class, PHP Interface

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Include inactive lines" option for languages that have inactive lines.

Average Comment Lines (Includes Inactive)

API ID: AvgCountLineCommentWithInactive
Languages: C++
Targets: Classes, Files

Average number of lines containing comment for all nested functions or methods, including inactive regions.

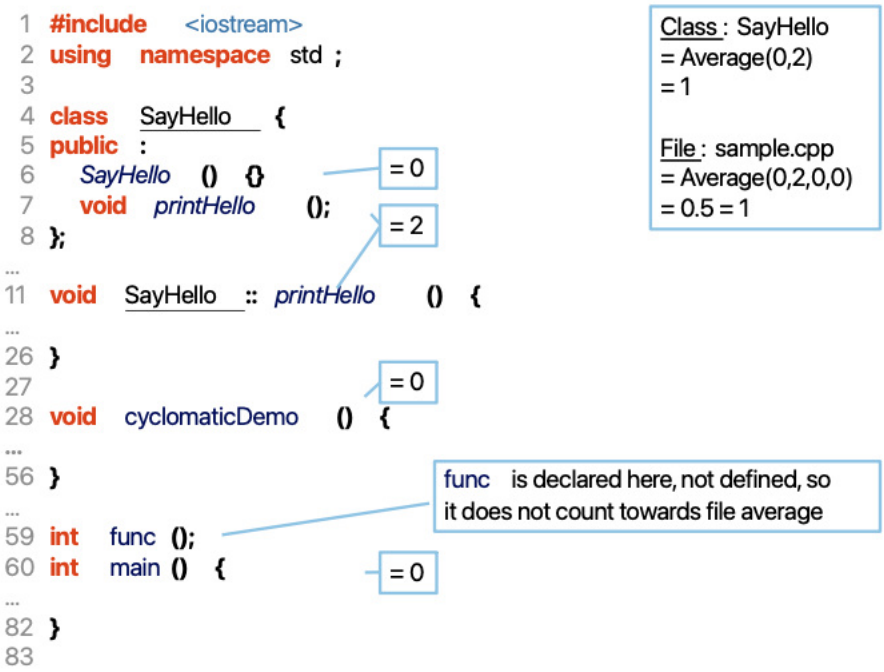
Unlike Average Comment Lines, this counts each function's comment lines inside inactive #if/#ifdef regions too, before averaging. Only functions and methods that are actually defined (not just declared) directly inside the container count towards the average, and the result rounds to the nearest whole number.

For example:

```
1 class SayHello {
2 public:
3   SayHello() {}
4   void printHello();
5 };
6
7 void SayHello::printHello() {
8   ...
9 }
10
11 void cyclomaticDemo() {
12   ...
13 }
14
15 int func();
16 int main() {
17   ...
18 }
```

SayHello() has 0 comment lines (including inactive) and SayHello::printHello() has 2 (its // Inactive comment falls inside an inactive #ifdef block), so class SayHello's average is Average(0, 2) = 1. The file also includes cyclomaticDemo() and main(), neither of which has any comments: Average(0, 2, 0, 0) = 0.5, which rounds to 1.

See Comment Lines (Includes Inactive) for the per-function count this averages.



Targets By Language:

- **C++:** File, Class, Struct, Union

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Lines and Statements" with the "Include inactive lines" option.

Average Cyclomatic Complexity

API ID: AvgCyclomatic
Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, VHDL, Web
Targets: Architectures, Classes, Files, Modules, Packages, Project

Average cyclomatic complexity for all nested functions or methods.

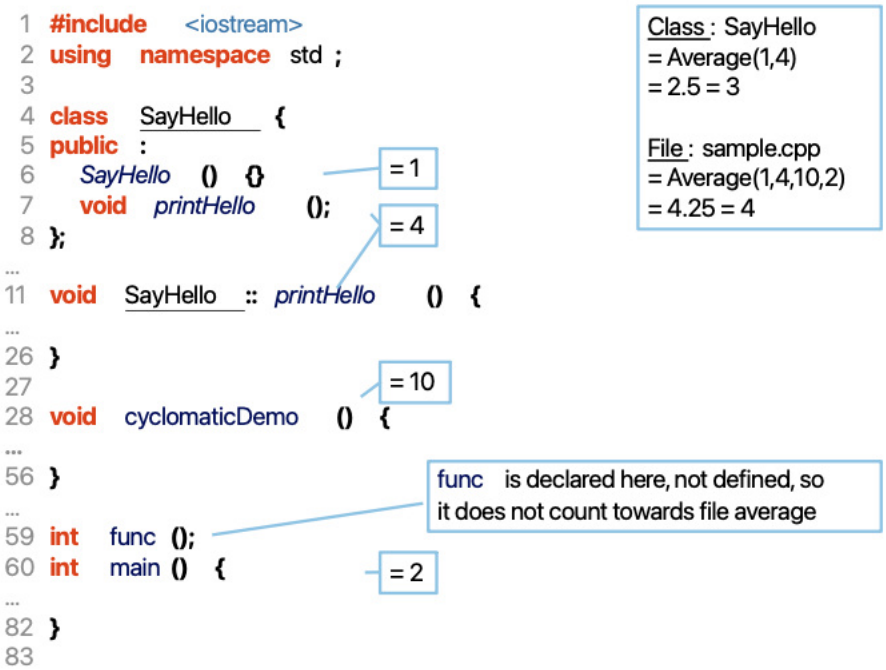
For example:

```
1 class SayHello {
2 public:
3   SayHello() {}
4   void printHello();
5 };
```

```
6
7 void SayHello::printHello() {
8   switch (i) {
9     ...
10   }
11 }
12
13 void cyclomaticDemo() {
14   ...
15 }
16
17 int func();           // declared here, not defined - excluded from aggreg
18
19 int main() {
20   ...
21 }
```

SayHello(), printHello(), cyclomaticDemo(), and main() have Cyclomatic Complexity of 1, 4, 10, and 2. func() is only declared, not defined, so it isn't counted. For the SayHello class, the average is (1 + 4) / 2 = 2.5, rounded to 3. For the whole file (sample.cpp), the average is (1 + 4 + 10 + 2) / 4 = 4.25, rounded to 4.

See Sum and Max Cyclomatic Complexity for the other aggregations.



Targets By Language:

- **Ada:** Project, File, Package
- **Basic:** Project, File, Module, Class, Struct
- **C#:** Project, File, Class, Struct

- **C++:** Project, File, Class, Struct, Union
- **Fortran:** Project, File
- **Java:** Project, File, Class, Interface
- **Jovial:** Project, File
- **Pascal:** Project, File, Class, Interface, Compunit
- **Python:** Project, File, Class
- **Rust:** Project, File
- **VHDL:** Project, File, Architecture
- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated function metrics" option.
- The cyclomatic metric used is configured from "Project Configuration/Metrics/Complexity". This metric results from having "Count each decision in a multi-way-decision structure" on and having "Count logical conjuctions (OR) and logical and (AND)" off. Checking "Show all available cyclomatic complexity metric variants" will also enable the metric.

Average Essential Complexity

API ID: AvgEssential
Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Python, Rust, Web
Targets: Architectures, Classes, Files, Modules, Packages, Project

Average Essential complexity for all nested functions or methods.

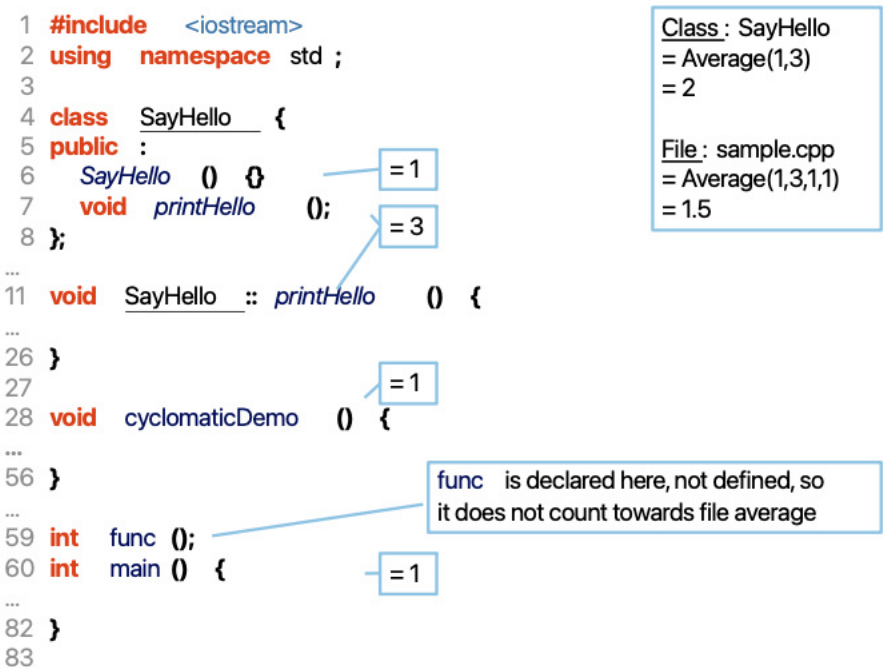
For example:

```
1 class SayHello {
2 public:
3   SayHello() {}
4   void printHello();
5 };
6
7 void SayHello::printHello() {
8   switch (i) {
9     ...
10   }
11 }
12
13 void cyclomaticDemo() {
14   ...
15 }
16
```

```
17 int func();          // declared here, not defined - excluded from aggreg
18
19 int main() {
20   ...
21 }
```

SayHello(), printHello(), cyclomaticDemo(), and main() have Essential Complexity of 1, 3, 1, and 1. func() is only declared, not defined, so it isn't counted. For the SayHello class, the average is (1 + 3) / 2 = 2, exactly. For the whole file (sample.cpp), the average is (1 + 3 + 1 + 1) / 4 = 1.5, rounded to 2.

See Sum and Max Essential Complexity for the other aggregations.



Targets By Language:

- **Ada:** Project, File, Package
- **Basic:** Project, File, Module, Class, Struct
- **C#:** Project, File, Class, Struct
- **C++:** Project, File, Class, Struct, Union
- **Fortran:** Project, File
- **Java:** Project, File, Class, Interface
- **Jovial:** Project, File
- **Python:** Project, File, Class
- **Rust:** Project, File
- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated function metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Complexity" with the "Essential" option.

Average Lines

API ID: AvgCountLine
Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, Web
Targets: Classes, Files, Modules, Packages

Average number of lines for all nested functions or methods.

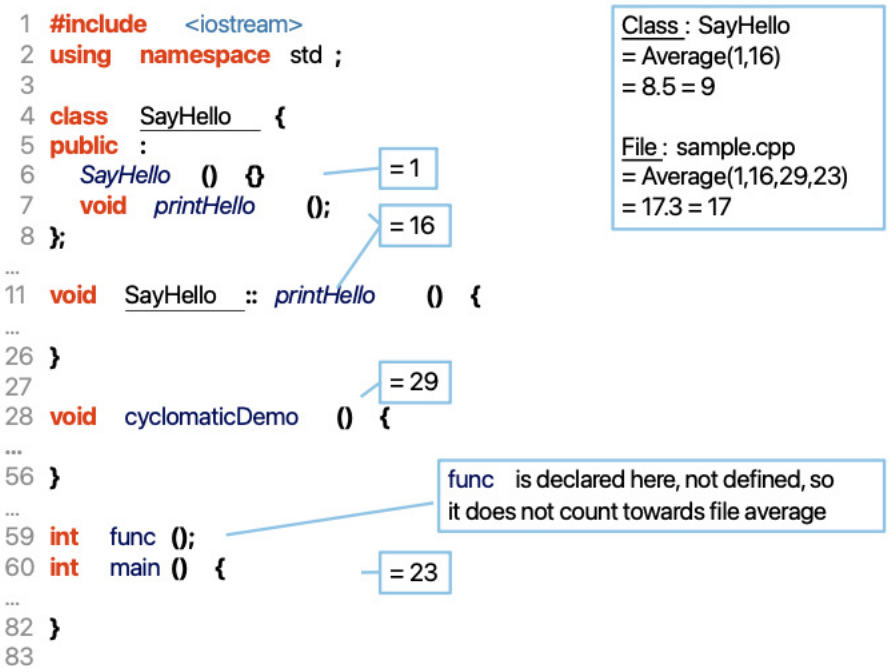
Only functions and methods that are actually defined (not just declared) directly inside the container count towards the average, and the result rounds to the nearest whole number.

For example:

```
1 class SayHello {
2 public:
3   SayHello() {}
4   void printHello();
5 };
6
7 void SayHello::printHello() {
8   ...
9 }
10
11 void cyclomaticDemo() {
12   ...
13 }
14
15 int func();
16 int main() {
17   ...
18 }
```

SayHello() is 1 line and SayHello::printHello() is 16, so class SayHello's Average Lines is Average(1, 16) = 8.5, which rounds to 9. The file also includes cyclomaticDemo() (29 lines) and main() (23 lines): Average(1, 16, 29, 23) = 17.3, which rounds to 17. func() is declared here but never defined, so it contributes nothing to either average.

See Lines for the per-function count this averages, and Average Code Lines, Average Comment Lines, and Average Blank Lines for the other line-type averages.



Targets By Language:

- **Ada:** File, Package
- **Basic:** File, Module, Class, Struct
- **C#:** File, Class, Struct
- **C++:** File, Class, Struct, Union
- **Fortran:** File
- **Java:** File, Class, Interface
- **Jovial:** File
- **Pascal:** File, Class, Interface
- **Python:** File, Class
- **Rust:** File
- **Web:** File, PHP Class, PHP Interface

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.

Average Modified Cyclomatic Complexity

API ID: AvgCyclomaticModified
Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, VHDL, Web
Targets: Architectures, Classes, Files, Modules, Packages, Project

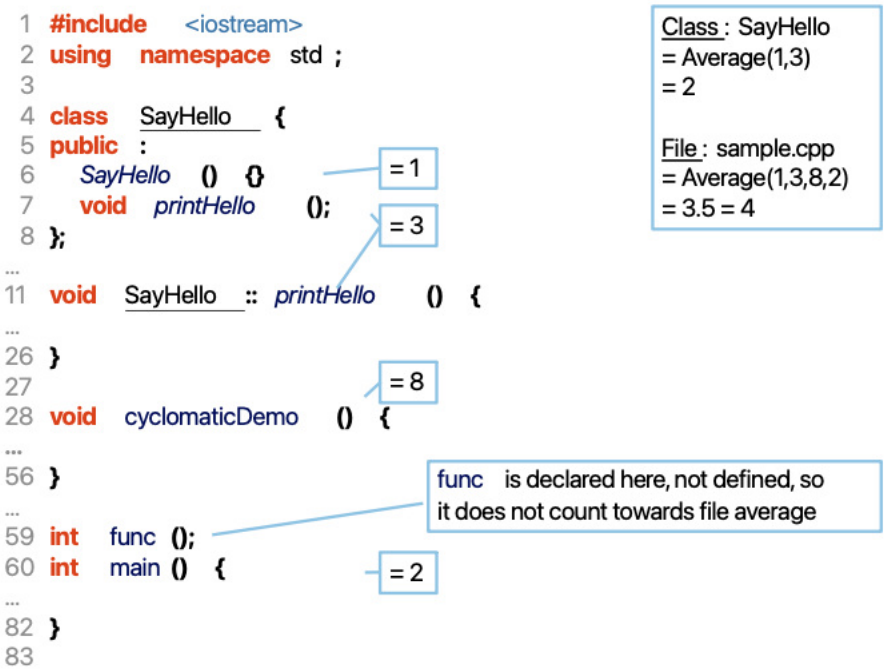
Average modified cyclomatic complexity for all nested functions or methods.

For example:

```
1 class SayHello {
2 public:
3   SayHello() {}
4   void printHello();
5 };
6
7 void SayHello::printHello() {
8   switch (i) {
9     ...
10   }
11 }
12
13 void cyclomaticDemo() {
14   ...
15 }
16
17 int func();          // declared here, not defined - excluded from aggreg
18
19 int main() {
20   ...
21 }
```

SayHello(), printHello(), cyclomaticDemo(), and main() have Modified Cyclomatic Complexity of 1, 3, 8, and 2. func() is only declared, not defined, so it isn't counted. For the SayHello class, the average is (1 + 3) / 2 = 2, exactly. For the whole file (sample.cpp), the average is (1 + 3 + 8 + 2) / 4 = 3.5, rounded to 4.

See Sum and Max Modified Cyclomatic Complexity for the other aggregations.



Targets By Language:

- **Ada:** Project, File, Package
- **Basic:** Project, File, Module, Class, Struct
- **C#:** Project, File, Class, Struct
- **C++:** Project, File, Class, Struct, Union
- **Fortran:** Project, File
- **Java:** Project, File, Class, Interface
- **Jovial:** Project, File
- **Pascal:** Project, File, Class, Interface, Compunit
- **Python:** Project, File, Class
- **Rust:** Project, File
- **VHDL:** Project, File, Architecture
- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated function metrics" option.
- The cyclomatic metric used is configured from "Project Configuration/Metrics/Complexity". This metric results from having "Count each decision in a multi-way-decision structure" off and having "Count logical conjuctions (OR) and logical and (AND)" off. Checking "Show all available cyclomatic complexity metric variants" will also enable the metric.

Average Strict Cyclomatic Complexity

API ID: AvgCyclomaticStrict
Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, Web
Targets: Architectures, Classes, Files, Modules, Packages, Project

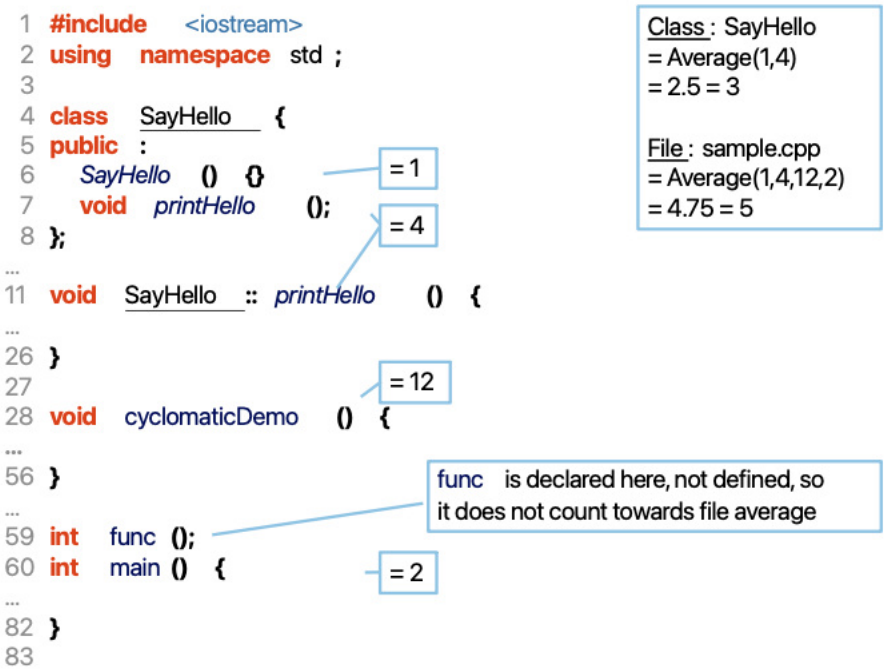
Average strict cyclomatic complexity for all nested functions or methods.

For example:

```
1 class SayHello {
2 public:
3   SayHello() {}
4   void printHello();
5 };
6
7 void SayHello::printHello() {
8   switch (i) {
9     ...
10   }
11 }
12
13 void cyclomaticDemo() {
14   ...
15 }
16
17 int func();          // declared here, not defined - excluded from aggreg
18
19 int main() {
20   ...
21 }
```

SayHello(), printHello(), cyclomaticDemo(), and main() have Strict Cyclomatic Complexity of 1, 4, 12, and 2. func() is only declared, not defined, so it isn't counted. For the SayHello class, the average is (1 + 4) / 2 = 2.5, rounded to 3. For the whole file (sample.cpp), the average is (1 + 4 + 12 + 2) / 4 = 4.75, rounded to 5.

See Sum and Max Strict Cyclomatic Complexity for the other aggregations.



Targets By Language:

- **Ada:** Project, File, Package
- **Basic:** Project, File, Module, Class, Struct
- **C#:** Project, File, Class, Struct
- **C++:** Project, File, Class, Struct, Union
- **Fortran:** Project, File
- **Java:** Project, File, Class, Interface
- **Jovial:** Project, File
- **Pascal:** Project, File, Class, Interface, Compunit
- **Python:** Project, File, Class
- **Rust:** Project, File
- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated function metrics" option.
- The cyclomatic metric used is configured from "Project Configuration/Metrics/Complexity". This metric results from having "Count each decision in a multi-way-decision structure" on and having "Count logical conjuctions (OR) and logical and (AND)" on. Checking "Show all available cyclomatic complexity metric variants" will also enable the metric.

Average Strict Modified Cyclomatic Complexity

API ID: AvgCyclomaticStrictModified
Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, Web
Targets: Architectures, Classes, Files, Modules, Packages, Project

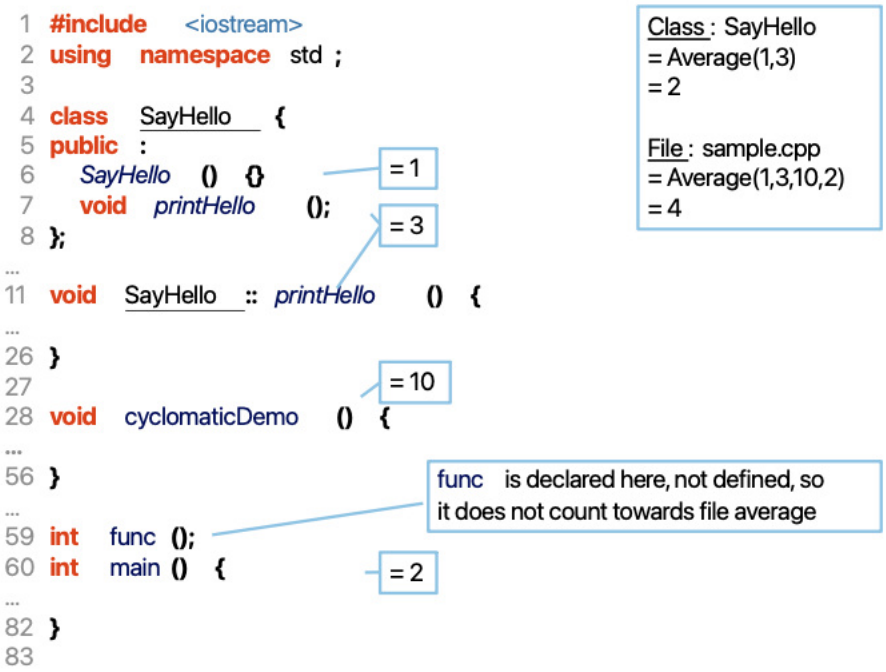
Average strict modified cyclomatic complexity for all nested functions or methods.

For example:

```
1 class SayHello {
2 public:
3   SayHello() {}
4   void printHello();
5 };
6
7 void SayHello::printHello() {
8   switch (i) {
9     ...
10   }
11 }
12
13 void cyclomaticDemo() {
14   ...
15 }
16
17 int func();          // declared here, not defined - excluded from aggreg
18
19 int main() {
20   ...
21 }
```

SayHello(), printHello(), cyclomaticDemo(), and main() have Strict Modified Cyclomatic Complexity of 1, 3, 10, and 2. func() is only declared, not defined, so it isn't counted. For the SayHello class, the average is (1 + 3) / 2 = 2, exactly. For the whole file (sample.cpp), the average is (1 + 3 + 10 + 2) / 4 = 4, exactly.

See Sum and Max Strict Modified Cyclomatic Complexity for the other aggregations.



Targets By Language:

- **Ada:** Project, File, Package
- **Basic:** Project, File, Module, Class, Struct
- **C#:** Project, File, Class, Struct
- **C++:** Project, File, Class, Struct, Union
- **Fortran:** Project, File
- **Java:** Project, File, Class, Interface
- **Jovial:** Project, File
- **Pascal:** Project, File, Class, Interface, Compunit
- **Python:** Project, File, Class
- **Rust:** Project, File
- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated function metrics" option.
- The cyclomatic metric used is configured from "Project Configuration/Metrics/Complexity". This metric results from having "Count each decision in a multi-way-decision structure" off and having "Count logical conjuctions (OR) and logical and (AND)" on. Checking "Show all available cyclomatic complexity metric variants" will also enable the metric.

Average Strict Modified Essential Complexity

API ID: AvgEssentialStrictModified
Languages: Ada
Targets: Architectures, Files, Packages, Project

Average strict modified essential complexity for all nested functions or methods.

For example:

```
1 package Essential_Demo is
2    procedure Simple_Proc (X : in out Integer);
3    function Strict_Func (A, B : Boolean; N : Integer) return Integer;
4    function Case_Func (Day : Integer) return Integer;
5    function Mixed_Func (A, B : Boolean; Day : Integer) return Integer;
6 end Essential_Demo;
```

Simple_Proc, Strict_Func, Case_Func, and Mixed_Func have Strict Modified Essential Complexity of 1, 5, 2, and 7. For the Essential_Demo package, the average is (1 + 5 + 2 + 7) / 4 = 3.75, rounded to 4.

See Sum and Max Strict Modified Essential Complexity for the other aggregations.

Targets By Language:

- **Ada:** Project, File, Package

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated function metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Complexity" with the "Essential" option.

Base Classes

API ID: CountClassBase
Also Known As: IFANIN
Languages: Basic, C#, C++, Java, Pascal, Python, Web
Targets: Classes

Number of immediate base classes.

Every class listed directly in this class's inheritance list counts, regardless of whether it's inherited publicly, privately, or protectedly.

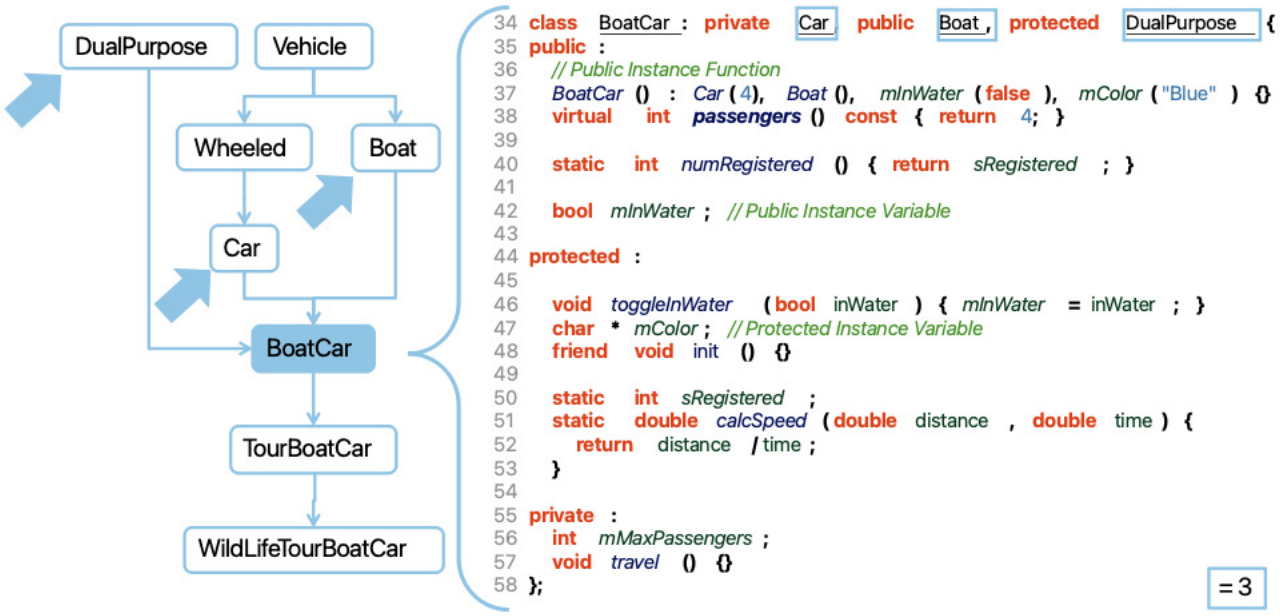
For example:

```
1 class Vehicle {
2 public:
3   Vehicle() {}
4 };
5
6 class Wheeled : public Vehicle {
7   ...
8 };
9
10 class Car : public Wheeled {
11   ...
12 };
13
14 class Boat : public Vehicle {
15   ...
16 };
17
18 class DualPurpose {
19   ...
20 };
21
22 class BoatCar: private Car, public Boat, protected DualPurpose {
23   ...
24 };
```

BoatCar lists three direct base classes on its inheritance list — Car (private), Boat (public), and DualPurpose (protected) — so CountClassBase = 3. The access level used for each base doesn't change whether it's counted.

See Derived Classes for the reverse relationship, and Max Inheritance Tree for how deep the full inheritance chain runs.

Also known as IFANIN.



Targets By Language:

- **Basic:** Class, Struct
- **C#:** Class, Struct
- **C++:** Class, Struct, Union
- **Java:** Class, Interface
- **Pascal:** Class, Interface
- **Python:** Class
- **Web:** PHP Class, PHP Interface

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show inheritance metrics" option.

Blank Lines

API ID: CountLineBlank
Also Known As: Blank Lines of Code (BLOC)
Languages: Ada, Assembly, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, VHDL, Web
Targets: Architectures, Classes, Files, Functions, Modules, Packages, Project, Subprograms

Number of blank lines.

A line only counts if it is active.

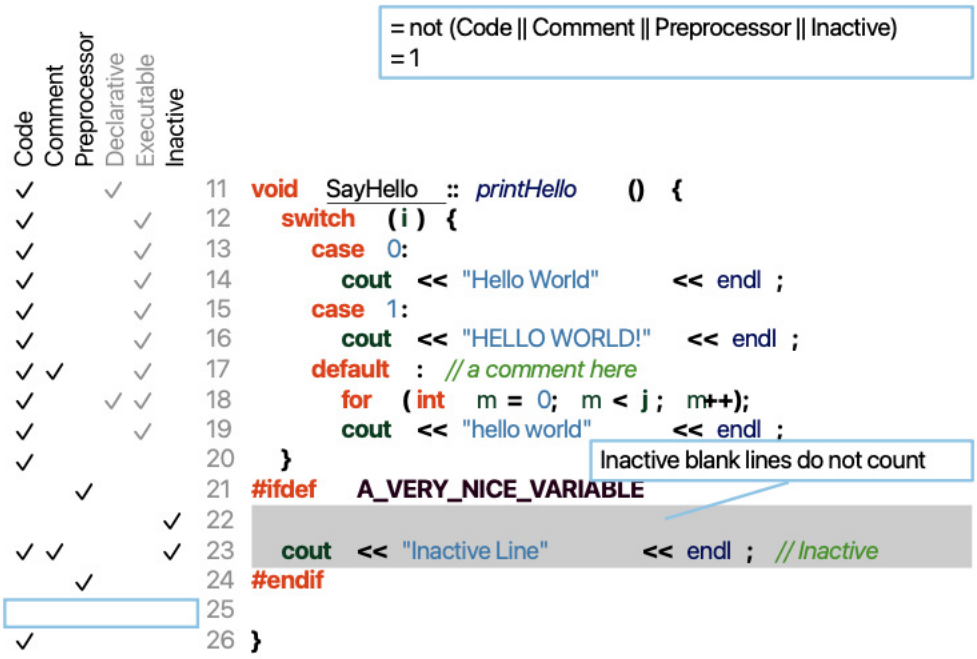
For example:

```
1 void SayHello::printHello() {
2   switch (i) {
3     case 0:
4       cout << "Hello World" << endl;
5     case 1:
6       cout << "HELLO WORLD!" << endl;
7     default: // a comment here
8       for (int m = 0; m < j; m++);
9       cout << "hello world" << endl;
10   }
11 #ifdef A_VERY_NICE_VARIABLE
12
13   cout << "Inactive Line" << endl; // Inactive
14 #endif
15
16 }
```

Line 25, the blank line just before the closing brace, is the only blank active line. The blank line inside the #ifdef A_VERY_NICE_VARIABLE block (line 22) doesn't count because it's inactive, giving CountLineBlank = 1.

To also count blank lines inside inactive regions, use Blank Lines (Includes Inactive). For the per-language breakdown in web files, see Blank Lines (HTML), Blank Lines (JavaScript), and Blank Lines (PHP).

Also known as Blank Lines of Code (BLOC).



Targets By Language:

- **Ada:** Project, File, Type, Entry, Function, Package, Procedure, Protected, Task
- **Basic:** Project, File, Method, Module, Class
- **C#:** Project, File, Class, Method
- **C++:** Project, File, Class, Struct, Union, Function
- **Fortran:** Project, File, Module, Block Data, Function, Program, Subroutine
- **Java:** Project, File, Class, Interface, Method
- **Jovial:** Project, File, Module, Subroutine
- **Pascal:** Project, File, Class, Interface, Compunit, Function, Procedure
- **Python:** Project, File, Class, Function
- **Rust:** Project, File, Function
- **VHDL:** Project, File, Procedure, Function, Process, Architecture
- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Include inactive lines" option for languages that have inactive lines.
- For projects and architectures, this metric is enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show default summary metrics" option.

Blank Lines (HTML)

API ID: CountLineBlankHtml
Languages: Web
Targets: Architectures, Files, Project

Number of blank HTML lines.

Only counts blank lines outside embedded sub-language blocks; a blank line inside a <script> block counts towards Blank Lines (JavaScript) instead.

For example:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Welcome</title>
5
6   <!-- Greeting script: tracks visits and confirms with the user -->
7   <script>
8     // Number of times the page has greeted someone
9     var greetingCount = 0;
10
11     function greetUser(name) {
12       greetingCount++;
13       return "Hello, " + name + "!";
14     }
15
16     let message = greetUser("Ada");
17     console.log(message);
18
19     if (greetingCount > 0) {
20       confirm(message);
21     }
22   </script>
23 </head>
24 <body>
25   <h1>Welcome to the site</h1>
26   <p>Have a look around.</p>
27 </body>
28 </html>
```

Line 5, the blank line inside <head> before the script's comment, is the only blank HTML line. The three blank lines inside the <script> block (10, 15, 18) count towards Blank Lines (JavaScript) instead, giving CountLineBlankHtml = 1.

See Blank Lines (JavaScript) for blank lines inside embedded scripts, and Blank Lines for the language-independent metric this breaks down.

Targets By Language:

- **Web:** Project, File

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Lines and Statements" with the "Breakdown counts by web language" option.

Blank Lines (Includes Inactive)

API ID: CountLineBlankWithInactive
Languages: C++
Targets: Architectures, Classes, Files, Functions, Project

Number of blank lines, including inactive regions.

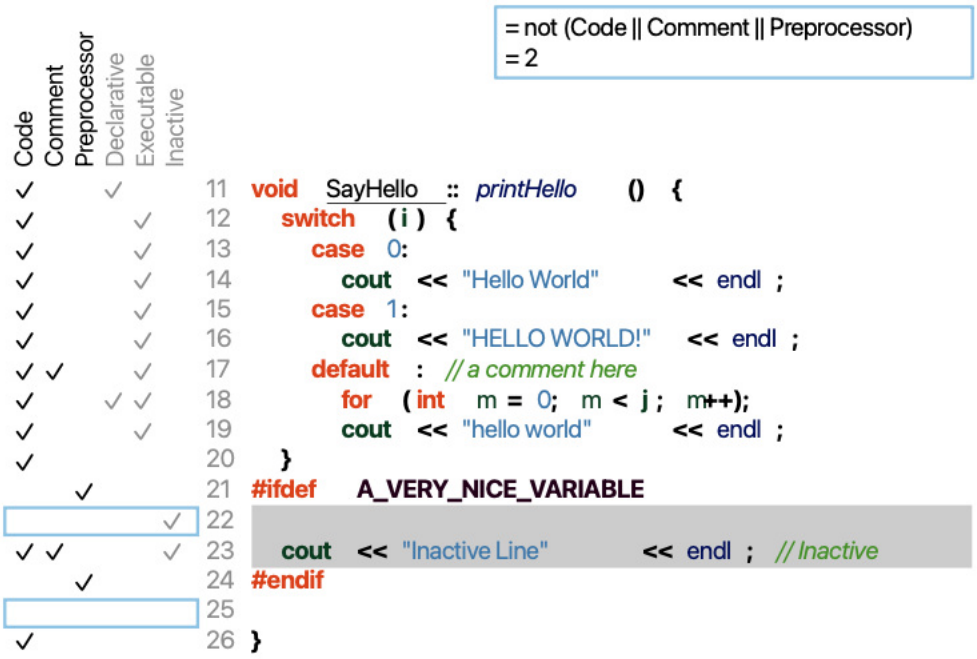
Unlike Blank Lines, this also counts blank lines that fall inside an inactive region.

For example:

```
1 void SayHello::printHello() {
2   switch (i) {
3     case 0:
4       cout << "Hello World" << endl;
5     case 1:
6       cout << "HELLO WORLD!" << endl;
7     default: // a comment here
8       for (int m = 0; m < j; m++);
9       cout << "hello world" << endl;
10   }
11 #ifdef A_VERY_NICE_VARIABLE
12
13   cout << "Inactive Line" << endl; // Inactive
14 #endif
15
16 }
```

Both the blank line just before the closing brace (line 25) and the blank line inside the #ifdef A_VERY_NICE_VARIABLE block (line 22) count, for CountLineBlankWithInactive = 2.

See Blank Lines to exclude inactive blank lines.



Targets By Language:

- **C++:** Project, File, Class, Struct, Union, Function

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Lines and Statements" with the "Include inactive lines" option.

Blank Lines (JavaScript)

API ID: CountLineBlankJavascript
Languages: Web
Targets: Architectures, Files, Project

Number of blank JavaScript lines.

Only counts blank lines inside an embedded sub-language block; see Blank Lines (HTML) for blank lines in the surrounding page.

For example:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
```

```
4    <title>Welcome</title>
5
6    <!-- Greeting script: tracks visits and confirms with the user -->
7    <script>
8      // Number of times the page has greeted someone
9      var greetingCount = 0;
10
11      function greetUser(name) {
12        greetingCount++;
13        return "Hello, " + name + "!";
14      }
15
16      let message = greetUser("Ada");
17      console.log(message);
18
19      if (greetingCount > 0) {
20        confirm(message);
21      }
22    </script>
23  </head>
24  <body>
25    <h1>Welcome to the site</h1>
26    <p>Have a look around.</p>
27  </body>
28  </html>
```

Lines 10, 15, and 18, separating the script's declarations and statements, are the only blank JavaScript lines, giving CountLineBlankJavascript = 3.

See Blank Lines (HTML) for blank lines outside embedded scripts, and Blank Lines for the language-independent metric this breaks down.

Targets By Language:

- **Web:** Project, File

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Lines and Statements" with the "Breakdown counts by web language" option.

Blank Lines (PHP)

API ID: CountLineBlankPhp
Languages: Web
Targets: Architectures, Classes, Files, Project

Number of blank PHP lines.

For example:

```
<?php
// Compute a visitor count for this session
const SITE_NAME = "Understand Demo";

function nextVisit($count) {
  return $count + 1;
}

$visits = 1;
$visits = nextVisit($visits);
echo "You are visitor number " . $visits . " to " . SITE_NAME;
```

Lines 4 and 8, separating SITE_NAME's declaration, the nextVisit() function, and the visitor-count logic, are the only blank lines, giving CountLineBlankPhp = 2.

See Blank Lines (HTML)/Blank Lines (JavaScript) for the HTML/JavaScript equivalents, and Blank Lines for the language-independent metric this breaks down.

Targets By Language:

- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Lines and Statements" with the "Breakdown counts by web language" option.

Changed Lines

API ID: CountLineChanged
Languages: Ada, Assembly, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, VHDL, Web
Targets: Classes, Files, Functions, Modules, Packages, Subprograms

Number of changed lines, compared to the comparison database.

Understand does a textual diff of the entity's contents and processes each changed region, meaning a run of adjacent added and deleted lines bounded by unchanged

lines. For each changed region, the changed lines is the number of lines in common, regardless of line contents. So, it's the minimum of the added and deleted line counts in each changed region.

Note that because this is specific to the entity's contents, things like reordering every function in a file while leaving function contents unchanged means that each individual function will be unchanged, even though the file changed. Also, an entity's lines are counted the same way they are for the Lines metric, so a C++ Class considers all the defined method bodies even when the definition is outside the class definition.

For example, diffing a file's current version against the comparison database's version:

```
--- sample.cpp (comparison database)
+++ sample.cpp (current)
@@ -1,15 +1,16 @@
 #include <iostream>

-// Utility functions
 int square(int x) {
-  return x * x;
+  return x * x * x;
 }

-int unused() {
-  return 0;
+// Compute a cube
+int cube(int x) {
+  return x * x * x;
 }

 int main() {
   std::cout << square(4) << std::endl;
+  std::cout << cube(3) << std::endl;
   return 0;
 }
```

The sample above has 4 changed regions. The first is the deletion of // Utility functions, so added is 0 and deleted is 1, and the changed lines is the smaller value: 0. The second is the change to the return statement with 1 line added and 1 line deleted. So changed lines is 1. The third change is the cube() function with 3 lines added and 2 lines deleted, giving 2 lines changed from the smaller (deleted) count. The final change is the insertion of std::cout << cube(3) << std::endl; with 1 added

line and 0 deleted lines, so changed lines is 0. So, the complete CountLineChanged is 0 + 1 + 2 + 0 = 3.

See New Lines and Removed Lines for the lines a region doesn't have in common, and Percent Changed for a single score combining all three.

Class Methods

API ID: CountDeclClassMethod
Also Known As: Number of Class Methods (NCM)
Languages: Basic, C#, C++, Java, Pascal, Web
Targets: Architectures, Classes, Files, Project

Number of class methods.

A "class method" is a static member function — one that belongs to the class as a whole rather than to any particular object, and so can be called without an instance.

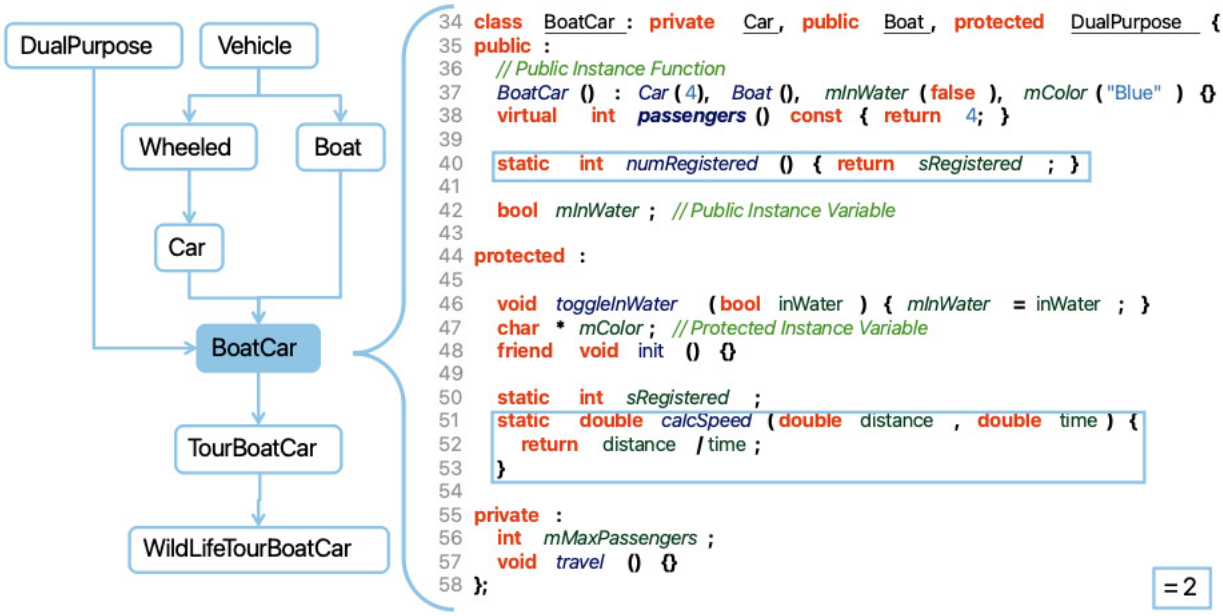
For example:

```
1 class BoatCar: private Car, public Boat, protected DualPurpose {
2 public:
3   // Public Instance Function
4   BoatCar() : Car(4), Boat(), mInWater(false), mColor("Blue") {}
5   virtual int passengers() const { return 4; }
6
7   static int numRegistered() { return sRegistered; }
8
9   bool mInWater; // Public Instance Variable
10
11 protected:
12
13   void toggleInWater(bool inWater) { mInWater = inWater; }
14   char * mColor; // Protected Instance Variable
15   friend void init() {}
16
17   static int sRegistered;
18   static double calcSpeed(double distance, double time) {
19     return distance/time;
20   }
21
22 private:
23   int mMaxPassengers;
24   void travel() {}
25 };
```

BoatCar declares two static methods, numRegistered() and calcSpeed(...), so CountDeclClassMethod = 2.

See Instance Methods for methods that require an object to call, and Methods for the combined total.

Also known as Lorenz & Kidd - Number of Class Methods (NCM).



Targets By Language:

- **Basic:** Project, Class, Struct
- **C#:** Project, Class, Struct
- **C++:** Project, Class, Struct, Union
- **Java:** Project, File, Class, Interface
- **Pascal:** Project, Class, Interface
- **Web:** Project, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated class metrics" option. Enabling this option can be time consuming.
- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Class Variables

API ID: CountDeclClassVariable
Also Known As: Number of Class Variables (NCV)

Languages: Basic, C#, C++, Java, Pascal, Web
Targets: Architectures, Classes, Files, Project

Number of class variables.

A "class variable" is a static data member — a single value shared by every instance of the class, rather than a separate copy per object.

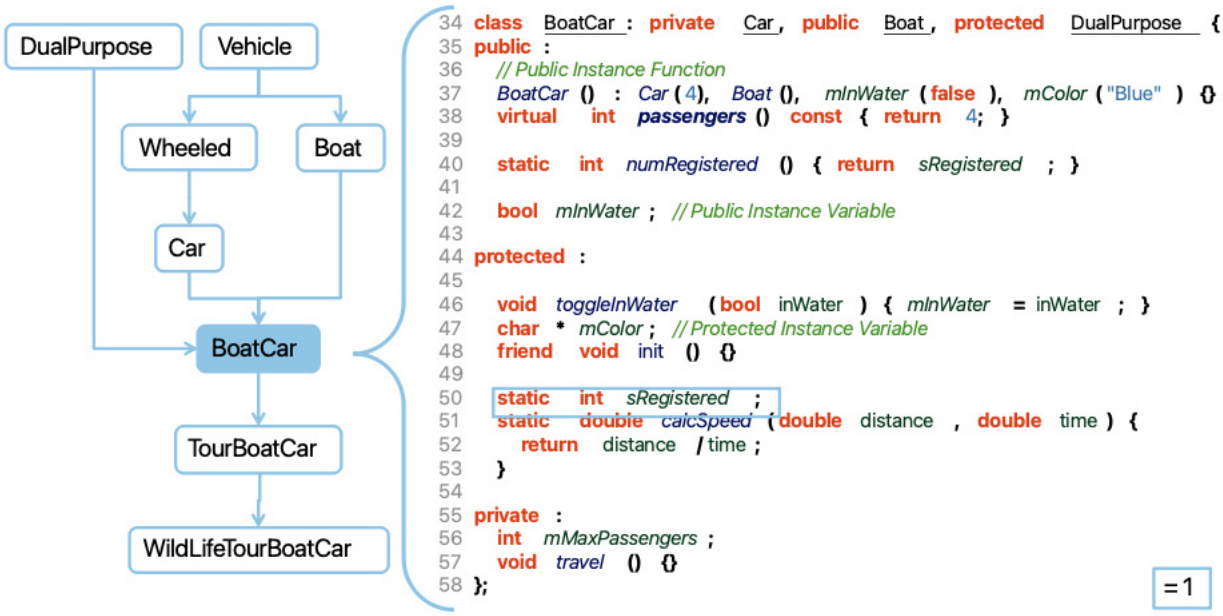
For example:

```
1 class BoatCar: private Car, public Boat, protected DualPurpose {
2 public:
3   // Public Instance Function
4   BoatCar() : Car(4), Boat(), mInWater(false), mColor("Blue") {}
5   virtual int passengers() const { return 4; }
6
7   static int numRegistered() { return sRegistered; }
8
9   bool mInWater; // Public Instance Variable
10
11 protected:
12
13   void toggleInWater(bool inWater) { mInWater = inWater; }
14   char * mColor; // Protected Instance Variable
15   friend void init() {}
16
17   static int sRegistered;
18   static double calcSpeed(double distance, double time) {
19     return distance/time;
20   }
21
22 private:
23   int mMaxPassengers;
24   void travel() {}
25 };
```

BoatCar declares one static data member, sRegistered, so CountDeclClassVariable = 1.

See Instance Variables for its per-object counterpart.

Also known as Lorenz & Kidd - Number of Class Variables (NCV).



Targets By Language:

- **Basic:** Project, Class, Struct
- **C#:** Project, Class, Struct
- **C++:** Project, Class, Struct, Union
- **Java:** Project, File, Class, Interface
- **Pascal:** Project, Class, Interface
- **Web:** Project, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated class metrics" option. Enabling this option can be time consuming.
- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Classes

API ID: CountDeclClass
Languages: Basic, C#, C++, Java, Pascal, Python, Web
Targets: Architectures, Files, Project

Number of classes.

Exactly what qualifies as a class for this count depends on the language and, for C+

+, the parse mode.

For example:

```
1 // class_macro.h
2 #define DECLARE_TAG_CLASS(ClassName) \
3   class ClassName { \
4   public: \
5     int tag = 0; \
6   };
7
8 // classes.cpp
9 #include "class_macro.h"
10
11 struct Point {
12   int x;
13   int y;
14 };
15
16 union Value {
17   int asInt;
18   float asFloat;
19 };
20
21 class Shape {
22 public:
23   virtual double area() const = 0;
24 };
25
26 enum class Color { Red, Green, Blue };
27
28 DECLARE_TAG_CLASS(MacroClass)
```

In C/C++, classes, structs, and unions all contribute to the count, but enums do not. So classes.cpp above has CountDeclClass = 4: the struct Point, the union Value, the class Shape, and the class MacroClass generated by the DECLARE_TAG_CLASS macro (the preprocessor expands it into a real class declaration before the parser ever sees it). The scoped enum Color doesn't count.

Note that C++'s fuzzy parsing mode doesn't support enum class syntax, so it mistakes Color for a class and reports CountDeclClass = 5 for the same file — one too many.

See Functions for the equivalent count of function declarations.

Targets By Language:

- **Basic:** Project, File
- **C#:** Project, File
- **C++:** Project, File
- **Java:** Project, File

- **Pascal:** Project, File
- **Python:** Project, File
- **Web:** Project, File

Configuration:

- For projects and architectures, this metric is enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated file metrics" option.
- For projects and architectures, this metric is enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show default summary metrics" option.

Code Files

API ID: CountDeclFileCode
Languages: C++
Targets: Architectures, Project

Number of code files.

Normally a .c/.cpp file, though it's based on how the parser identifies the file, not just its extension.

For example, this directory:

```
test/c/
  cohesion.cpp
  csvExportBigNumbers.cpp
  friend.cpp
  io.cpp
  knots.cpp
  paths.cpp
  pond.cpp
  sample.cpp
  vehicles.cpp
  classes.cpp
  class_macro.h
```

has 10 .cpp code files (everything except class_macro.h), so CountDeclFileCode = 10.

See Header Files for the complementary count, and Files for the total across all languages.

Targets By Language:

- **C++:** Project

Configuration:

- For projects and architectures, this metric is enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated file metrics" option.

Code Lines

API ID: CountLineCode
Also Known As: Lines of Code (LOC), Source Lines of Code (SLOC)
Languages: Ada, Assembly, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, VHDL, Web
Targets: Architectures, Classes, Files, Functions, Modules, Packages, Project, Subprograms

Number of lines containing source code.

A line only counts if it is active. Note that a line can contain source and a comment and thus count towards multiple metrics. For classes this is the sum of CountLineCode for the member functions of the class.

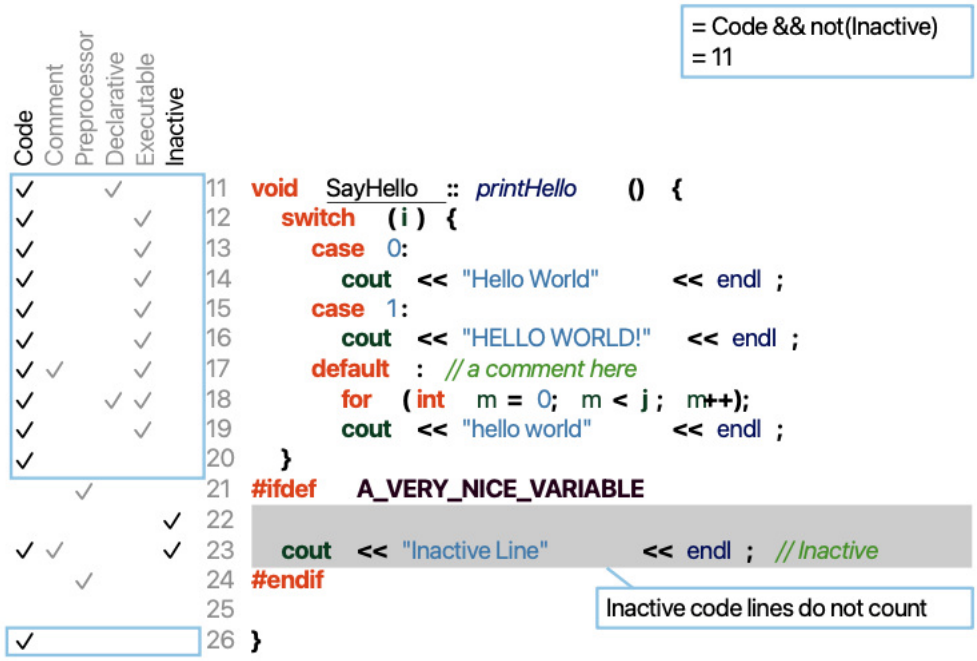
For example:

```
1 void SayHello::printHello() {
2   switch (i) {
3     case 0:
4       cout << "Hello World" << endl;
5     case 1:
6       cout << "HELLO WORLD!" << endl;
7     default: // a comment here
8       for (int m = 0; m < j; m++);
9       cout << "hello world" << endl;
10   }
11 #ifdef A_VERY_NICE_VARIABLE
12
13   cout << "Inactive Line" << endl; // Inactive
14 #endif
15
16 }
```

Lines 11-20 and line 26 (12 lines) all contain code. The cout << "Inactive Line" statement inside the #ifdef A_VERY_NICE_VARIABLE block also looks like code, but since A_VERY_NICE_VARIABLE isn't defined it's inactive and doesn't count, leaving CountLineCode = 11.

To also count code lines inside inactive regions, use Code Lines (Includes Inactive). For the per-language breakdown of code lines in web files, see Code Lines (JavaScript) and Code Lines (PHP).

Also known as Lines of Code (LOC); Source Lines of Code (SLOC).



Targets By Language:

- **Ada:** Project, File, Type, Entry, Function, Package, Procedure, Protected, Task
- **Basic:** Project, File, Method, Module, Class
- **C#:** Project, File, Class, Method
- **C++:** Project, File, Class, Struct, Union, Function
- **Fortran:** Project, File, Module, Block Data, Function, Program, Subroutine
- **Java:** Project, File, Class, Interface, Method
- **Jovial:** Project, File, Module, Subroutine
- **Pascal:** Project, File, Class, Interface, Compunit, Function, Procedure
- **Python:** Project, File, Class, Function
- **Rust:** Project, File, Function
- **VHDL:** Project, File, Procedure, Function, Process, Architecture
- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Include inactive lines" option for languages that have

inactive lines.
- For projects and architectures, this metric is enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show default summary metrics" option.

Code Lines (Includes Inactive)

API ID: CountLineCodeWithInactive
Languages: C++
Targets: Architectures, Classes, Files, Functions, Project

Number of lines containing source code, including inactive regions.

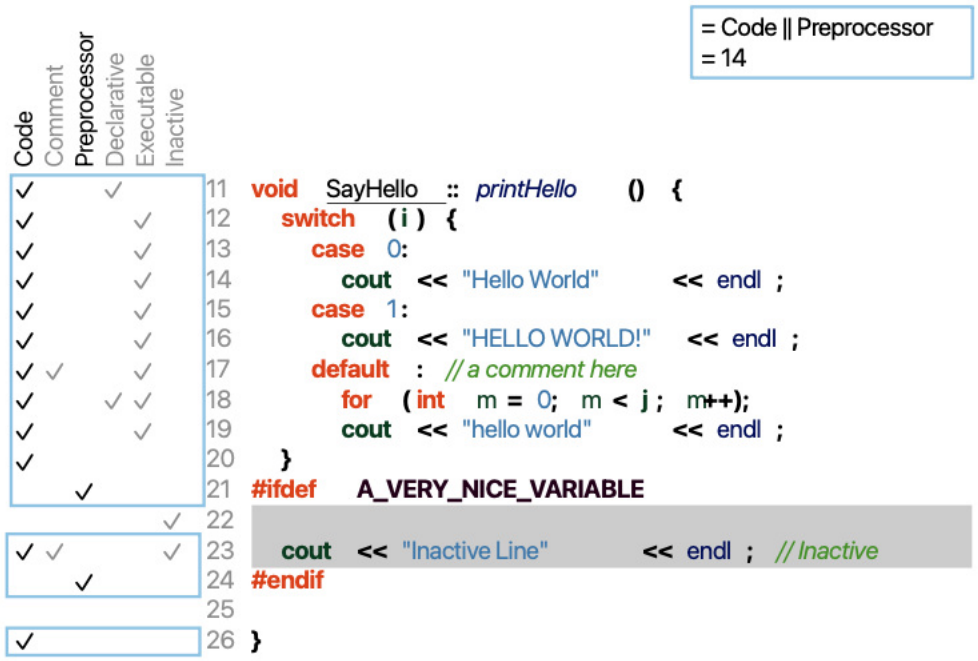
Unlike Code Lines, this counts a code line whether or not it falls inside an inactive #if/#ifdef region, and also counts the preprocessor conditional directive lines (#ifdef, #endif, etc.) that guard those regions.

For example:

```
1 void SayHello::printHello() {
2   switch (i) {
3     case 0:
4       cout << "Hello World" << endl;
5     case 1:
6       cout << "HELLO WORLD!" << endl;
7     default: // a comment here
8       for (int m = 0; m < j; m++);
9       cout << "hello world" << endl;
10   }
11 #ifdef A_VERY_NICE_VARIABLE
12
13   cout << "Inactive Line" << endl; // Inactive
14 #endif
15
16 }
```

The 12 code lines (11-20, 23, 26 — including the inactive cout << "Inactive Line" statement) plus the 2 preprocessor lines (#ifdef A_VERY_NICE_VARIABLE and #endif) give CountLineCodeWithInactive = 14.

See Code Lines to exclude inactive code.



Targets By Language:

- **C++:** Project, File, Class, Struct, Union, Function

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Lines and Statements" with the "Include inactive lines" option.

Code Lines (JavaScript)

API ID: CountLineCodeJavascript
Languages: Web
Targets: Architectures, Files, Project

Number of JavaScript lines containing source code.

For example:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Welcome</title>
5
6   <!-- Greeting script: tracks visits and confirms with the user -->
7   <script>
```

```
8     // Number of times the page has greeted someone
9     var greetingCount = 0;
10
11     function greetUser(name) {
12       greetingCount++;
13       return "Hello, " + name + "!";
14     }
15
16     let message = greetUser("Ada");
17     console.log(message);
18
19     if (greetingCount > 0) {
20       confirm(message);
21     }
22   </script>
23 </head>
24 <body>
25   <h1>Welcome to the site</h1>
26   <p>Have a look around.</p>
27 </body>
28 </html>
```

Of the 14 JavaScript lines, 10 contain code — every line except the comment (8) and the three blank lines (10, 15, 18) — giving CountLineCodeJavascript = 10.

See Code Lines for the combined count across all languages, and Code Lines (PHP) for the PHP equivalent.

Targets By Language:

- **Web:** Project, File

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Lines and Statements" with the "Breakdown counts by web language" option.

Code Lines (PHP)

API ID: CountLineCodePhp
Languages: Web
Targets: Architectures, Classes, Files, Project

Number of PHP lines containing source code.

For example:

```
<?php
// Compute a visitor count for this session
const SITE_NAME = "Understand Demo";

function nextVisit($count) {
  return $count + 1;
}

$visits = 1;
$visits = nextVisit($visits);
echo "You are visitor number " . $visits . " to " . SITE_NAME;
```

Of the 11 total lines, 8 contain code — every line except the two blank lines (4, 8) and the comment (2), including the opening <?php tag itself (line 1) — giving CountLineCodePhp = 8.

See Code Lines for the combined count across all languages, and Code Lines (JavaScript) for the JavaScript equivalent.

Targets By Language:

- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Lines and Statements" with the "Breakdown counts by web language" option.

CodeCheck Violation Density by Code Lines

API ID: CCViolDensityCode
Languages: Any
Targets: Architectures, Files, Functions, Project

The density of CodeCheck Violations by Lines of Code

This metric is calculated as Number of CodeCheck Violations divided by Code Lines. See also CodeCheck Violation Density by Lines.

CodeCheck Violation Density by Lines

API ID: CCViolDensityLine
Languages: Any
Targets: Architectures, Files, Functions, Project

The density of CodeCheck Violations by Line

This metric is calculated as Number of CodeCheck Violations divided by Lines. See also CodeCheck Violation Density by Code Lines.

CodeCheck Violation Types

API ID: CountCCViolType
Languages: Any
Targets: Architectures, Files, Functions, Project

The number of distinct violation types (distinct check ids)

For example, if the violation browser for a file showed these violations:

- non-void function does not return a value
- Remove unused global object `name`
- Remove unused local object dummy
- 'header.h' file not found.

The violation check ids are CPP_WARN_RETURN_TYPE, STI_UNUSED, STI_UNUSED, and UND_ERROR. UND_ERROR is an analysis error and not part of CodeCheck, so the value of this metric is 2 for CPP_WARN_RETURN_TYPE and STI_UNUSED.

See also CountAnalysisError for a count of analysis errors and CountAnalysisWarning for a count of analysis warnings.

CodeCheck Violations

API ID: CountCCViol
Languages: Any
Targets: Architectures, Files, Functions, Project

The number of CodeCheck Violations.

For example, if the violation browser for a file showed these violations:

- non-void function does not return a value
- Remove unused global object `name`
- Remove unused local object dummy

- 'header.h' file not found.

Then the value of this metric would be 3 because the fourth violation is an analysis error and this metric only counts CodeCheck violations.

See also CountAnalysisError for a count of analysis errors and CountAnalysisWarning for a count of analysis warnings.

Comment Lines

API ID: CountLineComment
Also Known As: Comment Lines of Code (CLOC)
Languages: Ada, Assembly, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, VHDL, Web
Targets: Architectures, Classes, Files, Functions, Modules, Packages, Project, Subprograms

Number of lines containing a comment.

A line only counts if it is active. This can overlap with other code counting metrics — for instance int j = 1; // comment has a comment, is a source line, is an executable source line, and is a declarative source line.

For example:

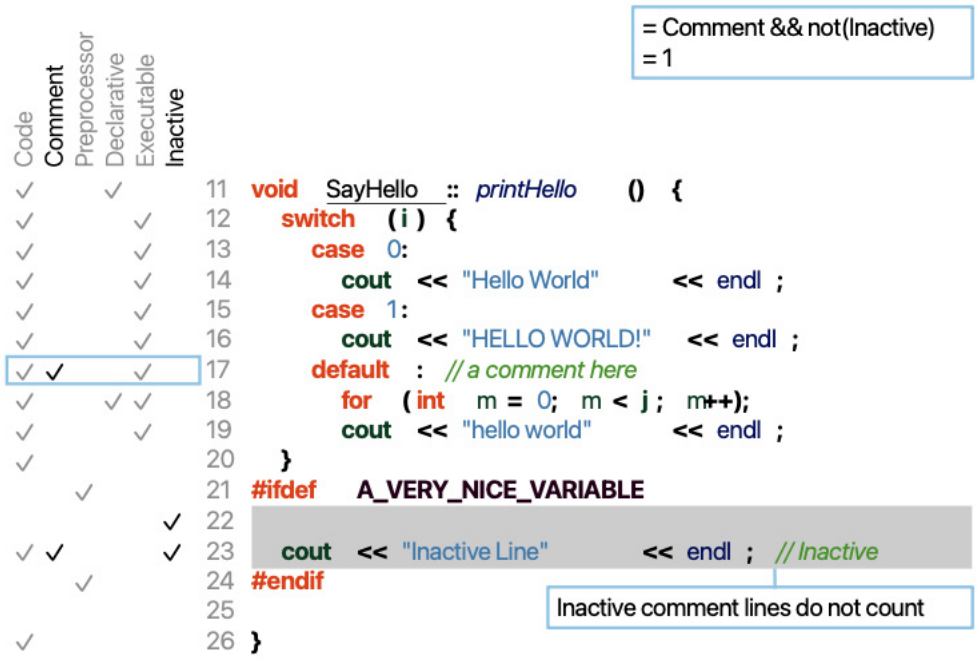
```
1 void SayHello::printHello() {
2   switch (i) {
3     case 0:
4       cout << "Hello World" << endl;
5     case 1:
6       cout << "HELLO WORLD!" << endl;
7     default: // a comment here
8       for (int m = 0; m < j; m++);
9       cout << "hello world" << endl;
10   }
11 #ifdef A_VERY_NICE_VARIABLE
12
13   cout << "Inactive Line" << endl; // Inactive
14 #endif
15
16 }
```

The default: // a comment here line (17) is the only active comment line. The // Inactive comment on the cout line inside the #ifdef block doesn't count because that line is inactive, giving CountLineComment = 1.

To also count comments inside inactive regions, use Comment Lines (Includes Inactive). For the per-language breakdown in web files, see Comment Lines (HTML),

Comment Lines (JavaScript), and Comment Lines (PHP).

Also known as Comment Lines of Code (CLOC).



Targets By Language:

- **Ada:** Project, File, Type, Entry, Function, Package, Procedure, Protected, Task
- **Basic:** Project, File, Method, Module, Class
- **C#:** Project, File, Class, Method
- **C++:** Project, File, Class, Struct, Union, Function
- **Fortran:** Project, File, Module, Block Data, Function, Program, Subroutine
- **Java:** Project, File, Class, Interface, Method
- **Jovial:** Project, File, Module, Subroutine
- **Pascal:** Project, File, Class, Interface, Compunit, Function, Procedure
- **Python:** Project, File, Class, Function
- **Rust:** Project, File, Function
- **VHDL:** Project, File, Procedure, Function, Process, Architecture
- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Include inactive lines" option for languages that have inactive lines.
- For projects and architectures, this metric is enabled from "Project

Configuration/Metrics/Projects and Architectures" with the "Show default summary metrics" option.

Comment Lines (HTML)

API ID: CountLineCommentHtml
Languages: Web
Targets: Architectures, Files, Project

Number of HTML lines containing a comment.

Only counts comments outside embedded sub-language blocks; a comment inside a <script> block counts towards Comment Lines (JavaScript) instead.

For example:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Welcome</title>
5
6   <!-- Greeting script: tracks visits and confirms with the user -->
7   <script>
8     // Number of times the page has greeted someone
9     var greetingCount = 0;
10
11     function greetUser(name) {
12       greetingCount++;
13       return "Hello, " + name + "!";
14     }
15
16     let message = greetUser("Ada");
17     console.log(message);
18
19     if (greetingCount > 0) {
20       confirm(message);
21     }
22   </script>
23 </head>
24 <body>
25   <h1>Welcome to the site</h1>
26   <p>Have a look around.</p>
27 </body>
28 </html>
```

The <!-- Greeting script: ... --> comment (line 6) is the only HTML comment. The // Number of times the page has greeted someone comment inside the <script> block (line 8) counts towards Comment Lines (JavaScript) instead, giving CountLineCommentHtml = 1.

See Comment Lines (JavaScript) for comments inside embedded scripts, and Comment Lines for the language-independent metric this breaks down.

Targets By Language:

- **Web:** Project, File

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Lines and Statements" with the "Breakdown counts by web language" option.

Comment Lines (Includes Inactive)

API ID: CountLineCommentWithInactive
Languages: C++
Targets: Architectures, Classes, Files, Functions, Project

Number of lines containing a comment, including inactive regions.

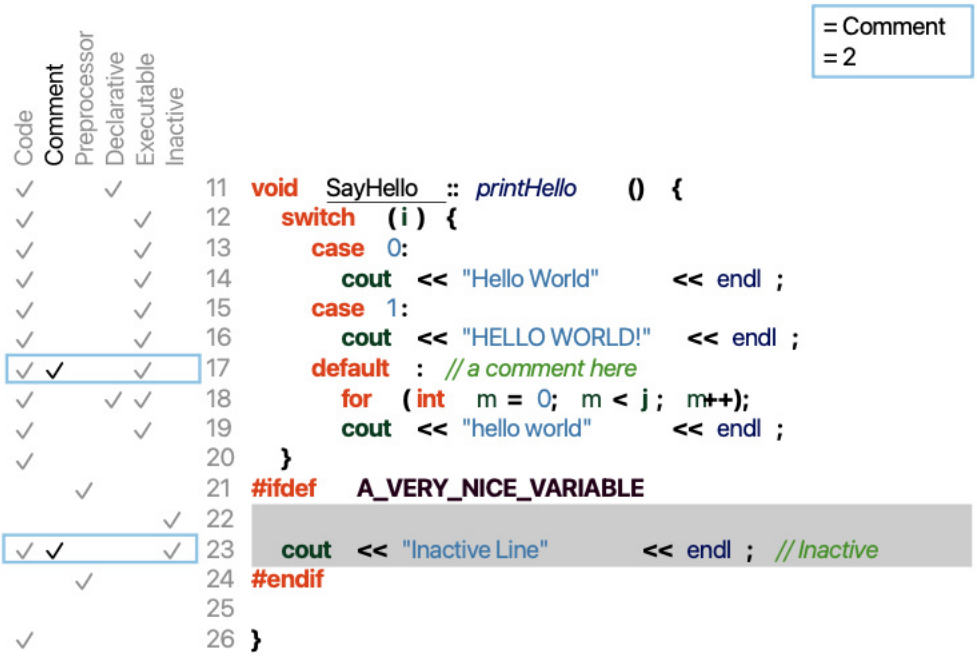
Unlike Comment Lines, this counts a comment line whether or not it falls inside an inactive region.

For example:

```
1 void SayHello::printHello() {
2   switch (i) {
3     case 0:
4       cout << "Hello World" << endl;
5     case 1:
6       cout << "HELLO WORLD!" << endl;
7     default: // a comment here
8       for (int m = 0; m < j; m++);
9       cout << "hello world" << endl;
10   }
11 #ifdef A_VERY_NICE_VARIABLE
12
13   cout << "Inactive Line" << endl; // Inactive
14 #endif
15
16 }
```

Both the default: // a comment here line and the // Inactive comment on the cout line inside the #ifdef block count, for CountLineCommentWithInactive = 2.

See Comment Lines to exclude inactive comments.



Targets By Language:

- **C++:** Project, File, Class, Struct, Union, Function

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Lines and Statements" with the "Include inactive lines" option.

Comment Lines (JavaScript)

API ID: CountLineCommentJavascript
Languages: Web
Targets: Architectures, Files, Project

Number of JavaScript lines containing a comment.

For example:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Welcome</title>
5
```

```
6   <!-- Greeting script: tracks visits and confirms with the user -->
7   <script>
8     // Number of times the page has greeted someone
9     var greetingCount = 0;
10
11     function greetUser(name) {
12       greetingCount++;
13       return "Hello, " + name + "!";
14     }
15
16     let message = greetUser("Ada");
17     console.log(message);
18
19     if (greetingCount > 0) {
20       confirm(message);
21     }
22   </script>
23 </head>
24 <body>
25   <h1>Welcome to the site</h1>
26   <p>Have a look around.</p>
27 </body>
28 </html>
```

The // Number of times the page has greeted someone comment (line 8) is the only JavaScript comment, giving CountLineCommentJavascript = 1.

See Comment Lines (HTML) for comments outside embedded scripts, and Comment Lines for the language-independent metric this breaks down.

Targets By Language:

- **Web:** Project, File

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Lines and Statements" with the "Breakdown counts by web language" option.

Comment Lines (PHP)

API ID: CountLineCommentPhp
Languages: Web
Targets: Architectures, Classes, Files, Project

Number of PHP lines containing a comment.

For example:

```
<?php
// Compute a visitor count for this session
const SITE_NAME = "Understand Demo";

function nextVisit($count) {
  return $count + 1;
}

$visits = 1;
$visits = nextVisit($visits);
echo "You are visitor number " . $visits . " to " . SITE_NAME;
```

The // Compute a visitor count for this session comment (line 2) is the only comment line, giving CountLineCommentPhp = 1.

See Comment Lines (HTML)/Comment Lines (JavaScript) for the HTML/JavaScript equivalents, and Comment Lines for the language-independent metric this breaks down.

Targets By Language:

- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Lines and Statements" with the "Breakdown counts by web language" option.

Comment to Code Ratio

API ID: RatioCommentToCode
Languages: Ada, Assembly, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, VHDL, Web
Targets: Architectures, Classes, Files, Functions, Modules, Packages, Project, Subprograms

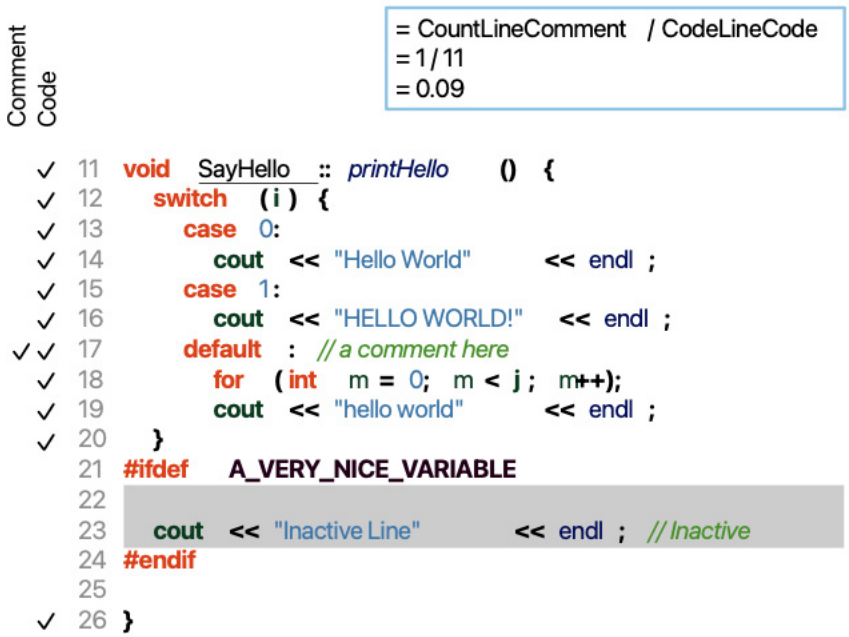
Ratio of comment lines to code lines.

Calculated as Comment Lines / Code Lines. Note that because some lines are both code and comment, this could easily yield a ratio higher than 1.

For example:

```
1 void SayHello::printHello() {
2   switch (i) {
3     case 0:
4       cout << "Hello World" << endl;
5     case 1:
6       cout << "HELLO WORLD!" << endl;
7     default: // a comment here
8       for (int m = 0; m < j; m++);
9       cout << "hello world" << endl;
10   }
11 #ifdef A_VERY_NICE_VARIABLE
12
13   cout << "Inactive Line" << endl; // Inactive
14 #endif
15
16 }
```

printHello() has 1 comment line (default: // a comment here) and 11 code lines, for a Comment to Code Ratio of 1 / 11 = 0.09.



Targets By Language:

- **Ada:** Project, File, Type, Entry, Function, Package, Procedure, Protected, Task
- **Basic:** Project, File, Method, Module, Class
- **C#:** Project, File, Class, Method
- **C++:** Project, File, Class, Struct, Union, Function
- **Fortran:** Project, File, Block Data, Function, Program, Subroutine

- **Java:** Project, File, Class, Interface, Method
- **Jovial:** Project, File, Module, Subroutine
- **Pascal:** Project, File, Class, Interface, Compunit, Function, Procedure
- **Python:** Project, File, Class, Function
- **Rust:** Project, File, Function
- **VHDL:** Project, File, Procedure, Function, Architecture
- **Web:** Project, File

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- For projects and architectures, this metric is enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show default summary metrics" option.

Const Methods

API ID: CountDeclMethodConst
Languages: C++
Targets: Architectures, Classes, Project

Number of local const methods.

A const method promises not to modify the object it's called on.

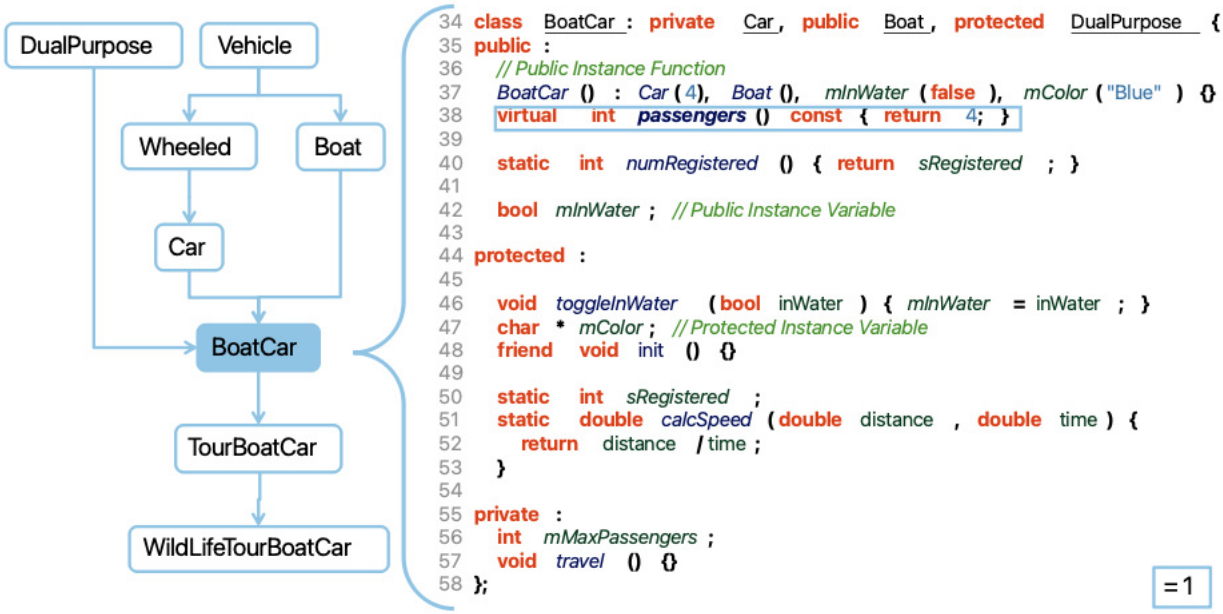
For example:

```
1 class BoatCar: private Car, public Boat, protected DualPurpose {
2 public:
3   // Public Instance Function
4   BoatCar() : Car(4), Boat(), mInWater(false), mColor("Blue") {}
5   virtual int passengers() const { return 4; }
6
7   static int numRegistered() { return sRegistered; }
8
9   bool mInWater; // Public Instance Variable
10
11 protected:
12
13   void toggleInWater(bool inWater) { mInWater = inWater; }
14   char * mColor; // Protected Instance Variable
15   friend void init() {}
16
17   static int sRegistered;
18   static double calcSpeed(double distance, double time) {
19     return distance/time;
20   }
```

```
21
22 private:
23   int mMaxPassengers;
24   void travel() {}
25 };
```

passengers() const is the only const method BoatCar declares, so CountDeclMethodConst = 1.

See Methods for the total across all local methods.



Targets By Language:

- **C++:** Project, Class, Struct, Union

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated class metrics" option. Enabling this option can be time consuming.
- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Coupled Classes

API ID: CountClassCoupled
Also Known As: Coupling Between Objects (CBO)

Languages: Basic, C#, C++, Java, Pascal, Python
Targets: Classes

Number of other classes coupled to.

The Coupling Between Object Classes (CBO) measure for a class is a count of the number of other classes to which it is coupled. Base classes and nested classes are not counted. Class A is coupled to class B if class A uses a type, data, or member from class B. This metric is also referred to as Efferent Coupling (Ce). Any number of couplings to a given class counts as 1 towards the metric total.

For example:

```
1 class Amphibian {
2 public:
3   void eatenbybird() { ... }
4 };
5
6 class Toad: public Amphibian {
7
8 };
9
10 class Water {
11 public:
12   static int temp() { return 60; }
13 };
14
15 class Fly {
16 public:
17   void getEaten() {}
18 };
19
20 class Frog: public Amphibian {
21 public:
22   bool swimming() {
23     if (Water::temp() > 50)
24       return 1;
25     return 0;
26   }
27   void eatFly() {
28     Fly edible;
29     edible.getEaten();
30   }
31 private:
32   class HopCalculator {
33   public:
34     int calculateHops() { return 1;}
35   };
36
37   Toad mCousin;
```

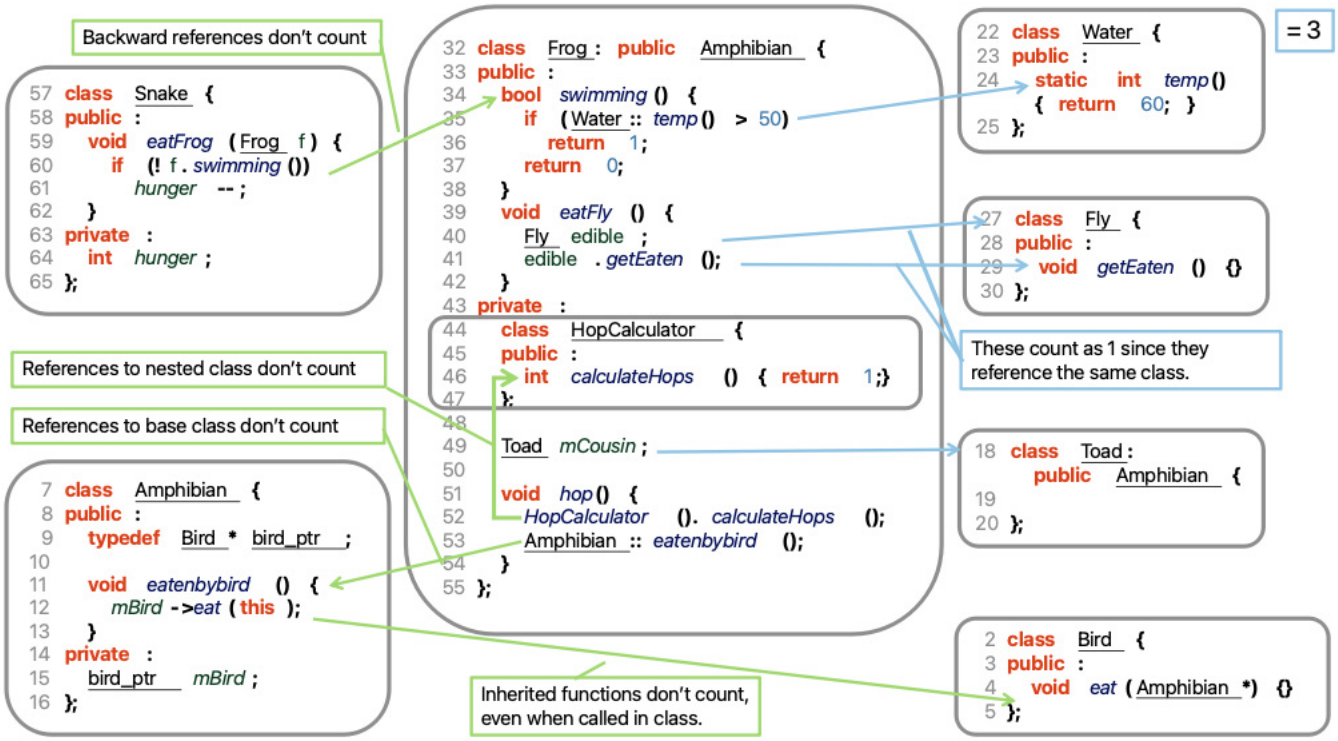
```
38
39    void hop() {
40      HopCalculator().calculateHops();
41      Amphibian::eatenbybird();
42    }
43 };
```

Frog is coupled to three other classes: Water (via the call to Water::temp()), Fly (via the local Fly edible; and the call to edible.getEaten()), and Toad (via the member Toad mCousin;). That gives CountClassCoupled = 3. Frog's own base class Amphibian is also used, through the call to Amphibian::eatenbybird(), but base classes are not counted, per the rule above. The nested class HopCalculator, declared and used inside Frog itself, is likewise excluded, per the "nested classes are not counted" rule.

See Coupled Packages for the package-level counterpart, and Coupled Classes Modified to exclude coupling to standard-library classes. If you're calculating this for every class in a large C++ project (for example, for a metrics export), consider Coupled Classes (Batch) instead, which caches every class's value on first use rather than recalculating per class. See the Coupled Classes Metric interactive report to see exactly which classes a given class is coupled to.

Chidamber & Kemerer suggest that:
1) Excessive coupling between object classes is detrimental to modular design and prevents reuse.
2) Inter-object class couples should be kept to a minimum.
3) The higher the inter-object class coupling, the more rigorous testing needs to be.

Also known as Chidamber & Kemerer - Coupling Between Objects (CBO).



Targets By Language:

- **Basic:** Class, Struct
- **C#:** Class, Struct
- **C++:** Class, Struct, Union
- **Java:** Class, Interface
- **Pascal:** Class, Interface
- **Python:** Class

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show coupling and cohesion metrics" option.

Coupled Classes Modified

API ID: CountClassCoupledModified
Languages: Basic, C#, Java, Pascal, Python
Targets: Classes

Number of other non-standard classes coupled to.

Same as Coupled Classes (Chidamber & Kemerer's CBO), except coupling to standard-library classes doesn't count — only coupling to other user-defined classes does.

For example:

```
1 class Helper {
2
3   public int value;
4
5   public void doWork() {
6     value++;
7   }
8
9 }
10
11 class Coupler {
12
13   public void useHelper() {
14     Helper h = new Helper();
15     h.doWork();
16
17     java.util.ArrayList<String> list = new java.util.ArrayList<String>();
18     list.add("test");
19     System.out.println(list.size());
20   }
21
22 }
```

Coupler uses a type, data, or member from five different classes: the user-defined Helper, plus four standard-library ones (java.util.ArrayList, String, System, and java.io.PrintStream, the last used implicitly via System.out), giving CountClassCoupled = 5. Only Helper is non-standard, so CountClassCoupledModified = 1.

See Coupled Classes to also count coupling to standard-library classes.

Targets By Language:

- **Basic:** Class, Struct
- **C#:** Class, Struct
- **Java:** Class, Interface
- **Pascal:** Class, Interface
- **Python:** Class

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show coupling and cohesion metrics" option.

Coupled Packages

API ID: CountPackageCoupled
Languages: Ada
Targets: Packages

Number of other packages coupled to.

Package A is coupled to package B if package A depends on (Ada's with) something declared in package B. Only the depending package counts the coupling — the package being depended on doesn't count as coupled to its dependents.

For example:

```
1 with Essential_Demo;
2
3 package Essential_Helper is
4
5    procedure Use_Essential (X : in out Integer);
6
7 end Essential_Helper;
```

Essential_Helper depends on Essential_Demo (calling Essential_Demo.Simple_Proc), giving Essential_Helper's CountPackageCoupled = 1. Essential_Demo itself has no dependencies of its own, so its CountPackageCoupled = 0, even though another package depends on it.

This is the package-level counterpart to Coupled Classes (Chidamber & Kemerer's CBO), which measures the same kind of coupling between classes.

Targets By Language:

- **Ada:** Package

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show coupling and cohesion metrics" option.

Cyclomatic Complexity

API ID: Cyclomatic
Also Known As: CC
Languages: Ada, Assembly, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, VHDL, Web
Targets: Files, Functions, Modules, Packages, Subprograms

McCabe Cyclomatic Complexity, the number of decision points + 1.

McCabe Cyclomatic complexity as per the original NIST paper on the subject. The cyclomatic complexity of any structured program with only one entrance point and

one exit point is equal to the number of decision points contained in that program plus one. Understand counts the keywords for decision points (FOR, WHILE, etc) and then adds 1. For a switch statement, each 'case' is counted as 1. For languages with macros, the expanded macro text is also included in the calculation.

For example:

```
1 void cyclomaticDemo() {
2   bool a = true, b = true, c = true;
3
4   if (a || (b && c)) {
5     while (a ? b : c) {
6       for (int i = 0; i < 10; i++) {
7         switch (i) {
8           case 1:
9           case 2:
10            cout << i << endl;
11            break;
12          case 5:
13            break;
14          default:
15            cout << i << endl;
16        }
17      }
18    }
19  } else {
20    try {
21      do {
22        cout << a << b << c << endl;
23      } while (a);
24    }
25    catch (...) {
26    }
27  }
28 }
```

if, while, the ?: conditional, for, do, and catch are always decision points, for 6 total. Each case in the switch statement adds another (3 total: case 1, case 2, case 5), and the &&/|| in the if condition are not counted. That's 9 decision points, for a Cyclomatic Complexity of 10.

To count the entire switch statement as a single decision instead of one per case, use Modified Cyclomatic Complexity. To also count && and || operators, use Strict Modified Cyclomatic Complexity. To count those operators without changing how the switch is handled, use Strict Cyclomatic Complexity.

See Sum Cyclomatic Complexity, Max Cyclomatic Complexity, and Average Cyclomatic Complexity for aggregated values, such as for an architecture or the project.

Also known as McCabe - Cyclomatic Complexity (CC).



Targets By Language:

- **Ada:** Type, Entry, Function, Package, Procedure, Protected, Task
- **Basic:** Method
- **C#:** Method
- **C++:** Function
- **Fortran:** Module, Block Data, Function, Program, Subroutine
- **Java:** Method
- **Jovial:** Subroutine
- **Pascal:** Compunit, Function, Procedure
- **Python:** File, Function
- **Rust:** Function
- **VHDL:** Procedure, Function, Process
- **Web:** File

Configuration:

- The cyclomatic metric used is configured from "Project Configuration/ Metrics/Complexity". This metric results from having "Count each decision in a multi-way-decision structure" on and having "Count logical conjuctions (OR) and logical and (AND)" off. Checking "Show all available cyclomatic complexity metric variants" will also enable the metric.

Declarative Code Lines

API ID: CountLineCodeDecl
Languages: Ada, Assembly, C#, C++, Fortran, Java, Pascal, Python
Targets: Architectures, Classes, Files, Functions, Modules, Packages, Project, Subprograms

Number of lines containing declarative source code.

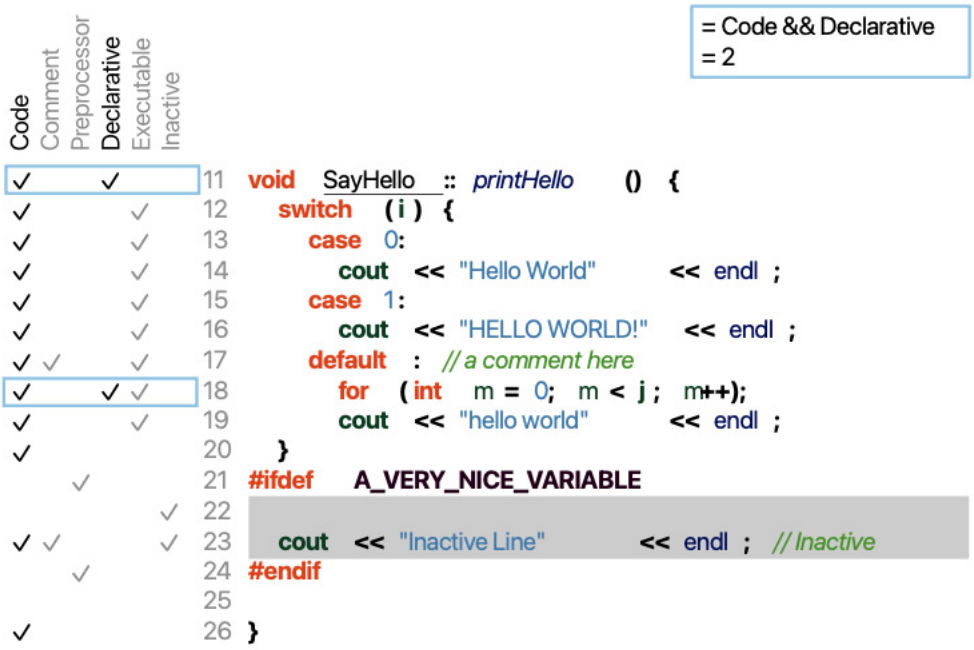
Note that a line can be declarative and executable — int i = 0;, for instance.

For example:

```
1 void SayHello::printHello() {
2   switch (i) {
3     case 0:
4       cout << "Hello World" << endl;
5     case 1:
6       cout << "HELLO WORLD!" << endl;
7     default: // a comment here
8       for (int m = 0; m < j; m++);
9       cout << "hello world" << endl;
10   }
11 #ifdef A_VERY_NICE_VARIABLE
12
13   cout << "Inactive Line" << endl; // Inactive
14 #endif
15
16 }
```

The function signature (line 11, declaring printHello) and the for loop's initializer (line 18, declaring m) are the only declarative code lines, giving CountLineCodeDecl = 2.

See Executable Code Lines for the complementary breakdown, and Code Lines for the total.



Targets By Language:

- **Ada:** Project, File, Type, Entry, Function, Package, Procedure, Protected, Task
- **C#:** Project, File, Class, Method
- **C++:** Project, File, Class, Struct, Union, Function
- **Fortran:** Project, File, Module, Block Data, Function, Program, Subroutine
- **Java:** Project, File, Class, Interface, Method
- **Pascal:** Project, File, Class, Interface, Compunit, Function, Procedure
- **Python:** Project, File, Class, Function

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.

Declarative Statements

API ID: CountStmtDecl
Languages: Ada, Assembly, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, VHDL, Web
Targets: Architectures, Classes, Files, Functions, Modules, Packages, Project, Subprograms

Number of declarative statements.

A statement that both declares a variable and initializes it with a function call counts

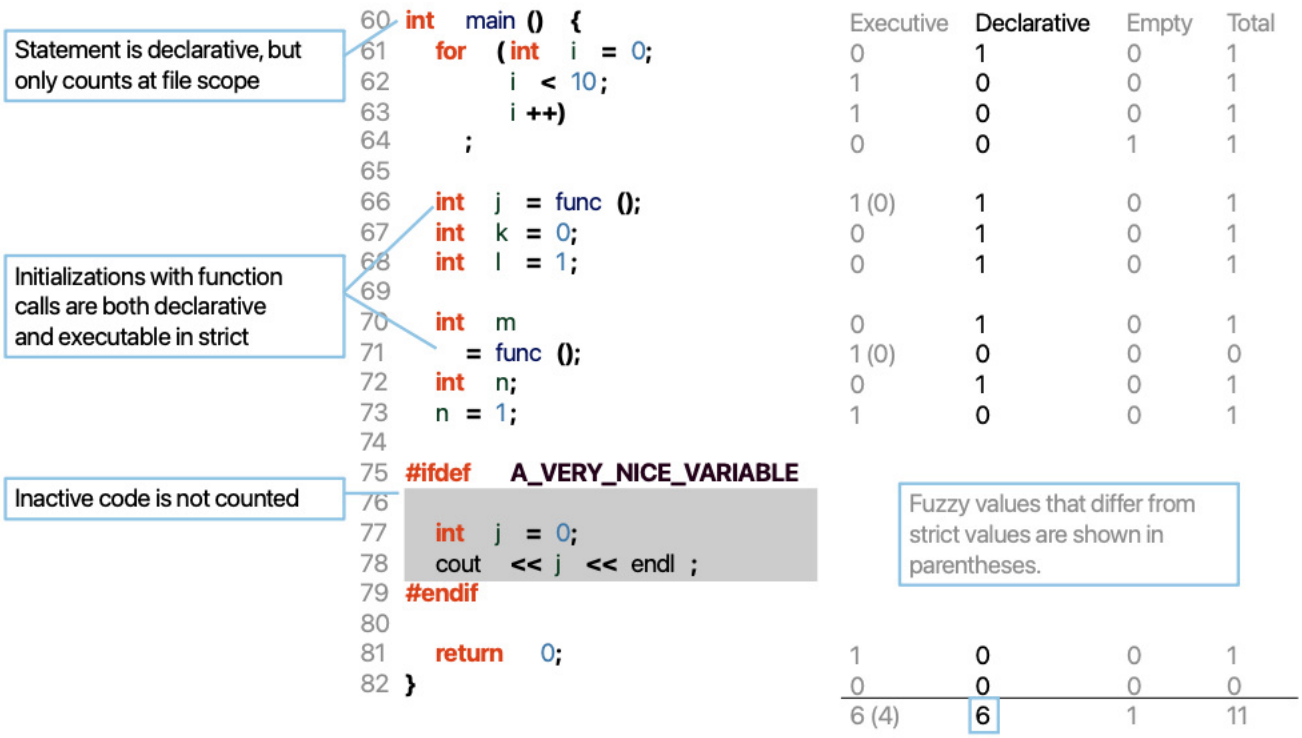
as both declarative and executable.

For example:

```
1 int main() {
2   for (int i = 0;
3        i < 10;
4        i++)
5     ;
6
7   int j = func();
8   int k = 0;
9   int l = 1;
10
11   int m
12     = func();
13   int n;
14   n = 1;
15
16 #ifdef A_VERY_NICE_VARIABLE
17
18   int j = 0;
19   cout << j << endl;
20 #endif
21
22   return 0;
23 }
```

The six variable declarations — int i (the for loop's own initializer), int j = func();, int k = 0;, int l = 1;, int m = func();, and int n; — are all declarative, for CountStmtDecl = 6. int j = func(); and int m = func(); also count as executable, since each calls func().

See Executable Statements for the complementary breakdown, and Statements for the total. For the per-language breakdown in web files, see Declarative Statements (JavaScript) and Declarative Statements (PHP).



Targets By Language:

- **Ada:** Project, File, Type, Entry, Function, Package, Procedure, Protected, Task
- **Basic:** Project, File, Method, Module, Class
- **C#:** Project, File, Class, Method
- **C++:** Project, File, Class, Struct, Union, Function
- **Fortran:** Project, File, Module, Block Data, Function, Program, Subroutine
- **Java:** Project, File, Class, Interface, Method
- **Jovial:** Project, File, Module, Subroutine
- **Pascal:** Project, File, Class, Interface, Compunit, Function, Procedure
- **Python:** Project, File, Class, Function
- **VHDL:** Project, File
- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show statement count metrics" option.
- For projects and architectures, this metric is enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show default summary metrics" option.

Declarative Statements (JavaScript)

API ID: CountStmtDeclJavascript
Languages: Web
Targets: Architectures, Files, Project

Number of JavaScript declarative statements.

For example:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Welcome</title>
5
6   <!-- Greeting script: tracks visits and confirms with the user -->
7   <script>
8     // Number of times the page has greeted someone
9     var greetingCount = 0;
10
11     function greetUser(name) {
12       greetingCount++;
13       return "Hello, " + name + "!";
14     }
15
16     let message = greetUser("Ada");
17     console.log(message);
18
19     if (greetingCount > 0) {
20       confirm(message);
21     }
22   </script>
23 </head>
24 <body>
25   <h1>Welcome to the site</h1>
26   <p>Have a look around.</p>
27 </body>
28 </html>
```

var greetingCount = 0; (9), function greetUser(name) { ... } (11), and let message = greetUser("Ada"); (16) are the three declarations, giving CountStmtDeclJavascript = 3. Note that line 16 also counts as an executable statement, since it calls greetUser() as well as declaring message.

See Executable Statements (JavaScript) for the complementary breakdown, and Declarative Statements for the combined count across all languages.

Targets By Language:

- **Web:** Project, File

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show statement count metrics" option.

- This metric can be enabled from "Project Configuration/Metrics/Lines and Statements" with the "Breakdown counts by web language" option.

Declarative Statements (PHP)

API ID: CountStmtDeclPhp
Languages: Web
Targets: Architectures, Classes, Files, Project

Number of PHP declarative statements.

For example:

```
<?php
// Compute a visitor count for this session
const SITE_NAME = "Understand Demo";

function nextVisit($count) {
  return $count + 1;
}

$visits = 1;
$visits = nextVisit($visits);
echo "You are visitor number " . $visits . " to " . SITE_NAME;
```

const SITE_NAME = "Understand Demo"; (3) and function nextVisit($count) { ... } (5) are the two declarations, giving CountStmtDeclPhp = 2.

See Executable Statements (PHP) for the complementary breakdown, and Declarative Statements for the combined count across all languages.

Targets By Language:

- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show statement count metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Lines and Statements" with the "Breakdown counts by web language" option.

Default Methods

API ID: CountDeclMethodDefault
Languages: Java
Targets: Architectures, Classes, Files, Project

Number of local default methods.

A default method (also called package-private) has no explicit public, private, or protected modifier in Java, making it accessible only from other classes in the same package.

For example:

```
1 public class sample {
2
3   public void publicMethod() {
4   }
5
6   void defaultMethod() {
7   }
8
9   private void privateMethod() {
10   }
11
12 }
```

defaultMethod() has no access modifier, giving CountDeclMethodDefault = 1. publicMethod() and privateMethod() have explicit modifiers, so neither counts here.

See Public Methods, Protected Methods, and Private Methods for the access levels shared with C++, C#, and Java, and Internal Methods/Protected Internal Methods for C#'s additional access levels.

Targets By Language:

- **Java:** Project, File, Class, Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated class metrics" option. Enabling this option can be time consuming.
- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Derived Classes

API ID: CountClassDerived
Also Known As: Number of Children (NOC)

Languages: Basic, C#, C++, Java, Pascal, Python, Web
Targets: Classes

Number of immediate subclasses.

(i.e. the number of classes one level down the inheritance tree from this class).

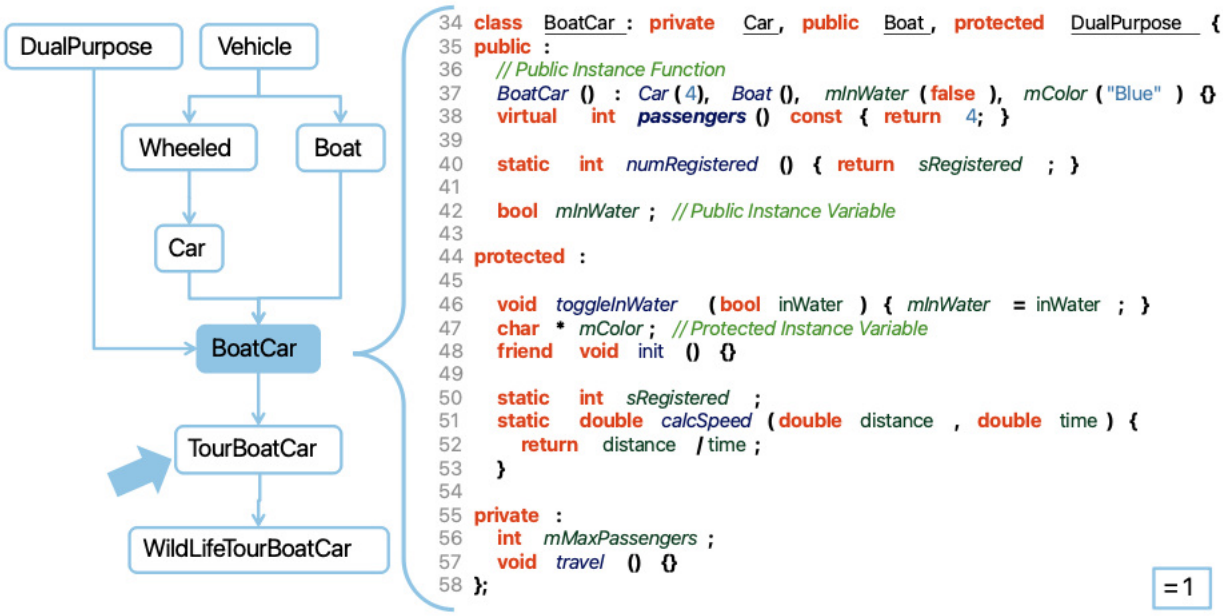
For example:

```
1 class Vehicle {
2 public:
3   Vehicle() {}
4 };
5
6 class Wheeled : public Vehicle {
7   ...
8 };
9
10 class Car : public Wheeled {
11   ...
12 };
13
14 class Boat : public Vehicle {
15   ...
16 };
17
18 class DualPurpose {
19   ...
20 };
21
22 class BoatCar: private Car, public Boat, protected DualPurpose {
23   ...
24 };
25
26 class TourBoatCar : public BoatCar {};
27 class WildLifeTourBoatCar: public TourBoatCar {};
```

Only TourBoatCar derives directly from BoatCar, so CountClassDerived = 1 for BoatCar. WildLifeTourBoatCar derives from TourBoatCar, not directly from BoatCar, so it doesn't count here even though it's a "grandchild" further down the same tree.

See Base Classes for the reverse relationship.

Also known as Chidamber & Kemerer - Number of Children (NOC).



Targets By Language:

- **Basic:** Class, Struct
- **C#:** Class, Struct
- **C++:** Class, Struct, Union
- **Java:** Class, Interface
- **Pascal:** Class, Interface
- **Python:** Class
- **Web:** PHP Class, PHP Interface

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show inheritance metrics" option.

Empty Statements

API ID: CountStmtEmpty
Languages: C++
Targets: Architectures, Classes, Files, Functions, Project

Number of empty statements.

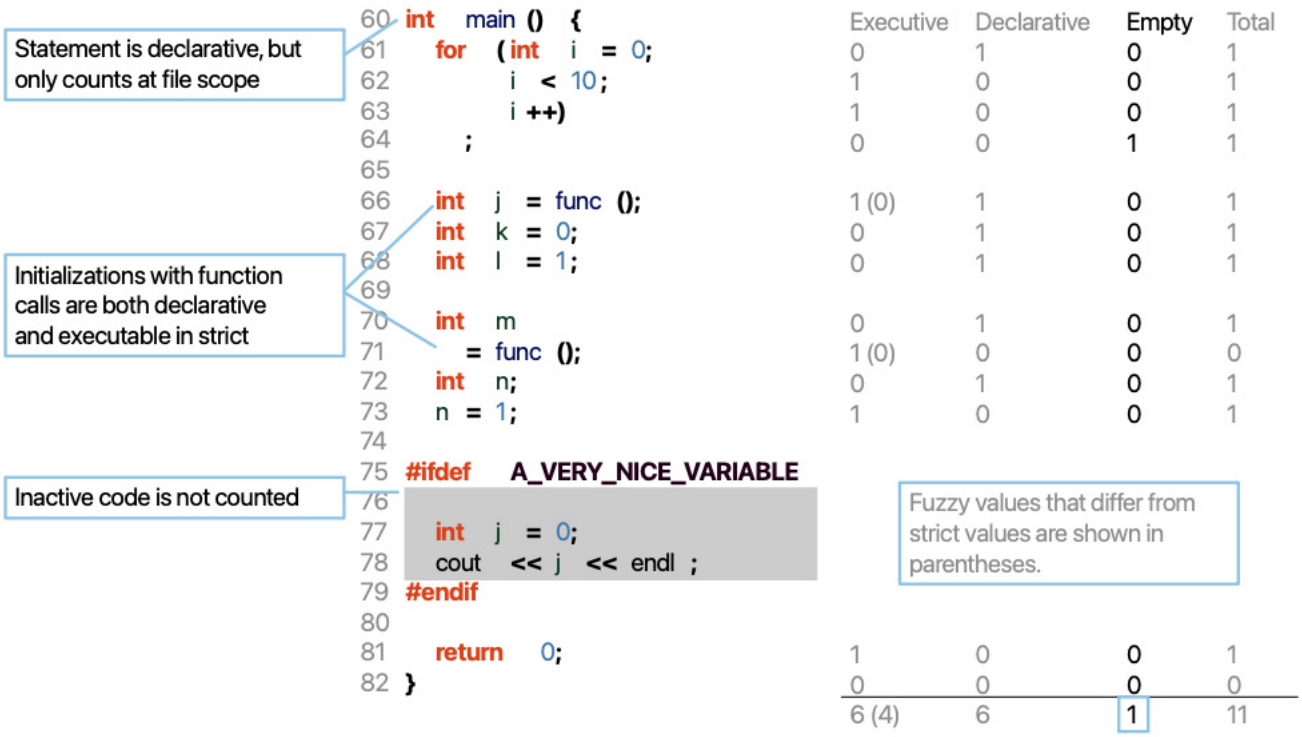
An empty statement is a lone ; with no expression, most often the trailing semicolon of an otherwise-empty loop body.

For example:

```
1 int main() {
2   for (int i = 0;
3        i < 10;
4        i++)
5     ;
6
7   int j = func();
8   int k = 0;
9   int l = 1;
10
11   int m
12     = func();
13   int n;
14   n = 1;
15
16 #ifdef A_VERY_NICE_VARIABLE
17
18   int j = 0;
19   cout << j << endl;
20 #endif
21
22   return 0;
23 }
```

The for loop's body is just ; (line 64) — a placeholder, since the loop's real work happens in its header — for CountStmtEmpty = 1.

See Statements for the total statement count this is part of.



Targets By Language:

- **C++:** Project, File, Class, Struct, Union, Function

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show statement count metrics" option.

Essential Complexity

API ID: Essential
Also Known As: ev(G)
Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, Web
Targets: Files, Functions, Modules, Packages, Subprograms

The number of decision points + 1 after control graph reduction.

Essential complexity is the cyclomatic complexity after iteratively replacing all well structured control structures with a single statement. Structures such as if-then-else and while loops are considered well structured. Understand calculates the essential complexity by removing all the structured subgraphs from the control graph and then calculating the complexity. A graph that has only the regular single entry/single exit loops or branches will be reducible to a graph with complexity one. Any branches into or out of a loop or decision will make the graph non-reducible and will have Essential Complexity > 2. (You never get 2 since a graph with complexity 2 is always reducible to a graph with complexity 1)

For example:

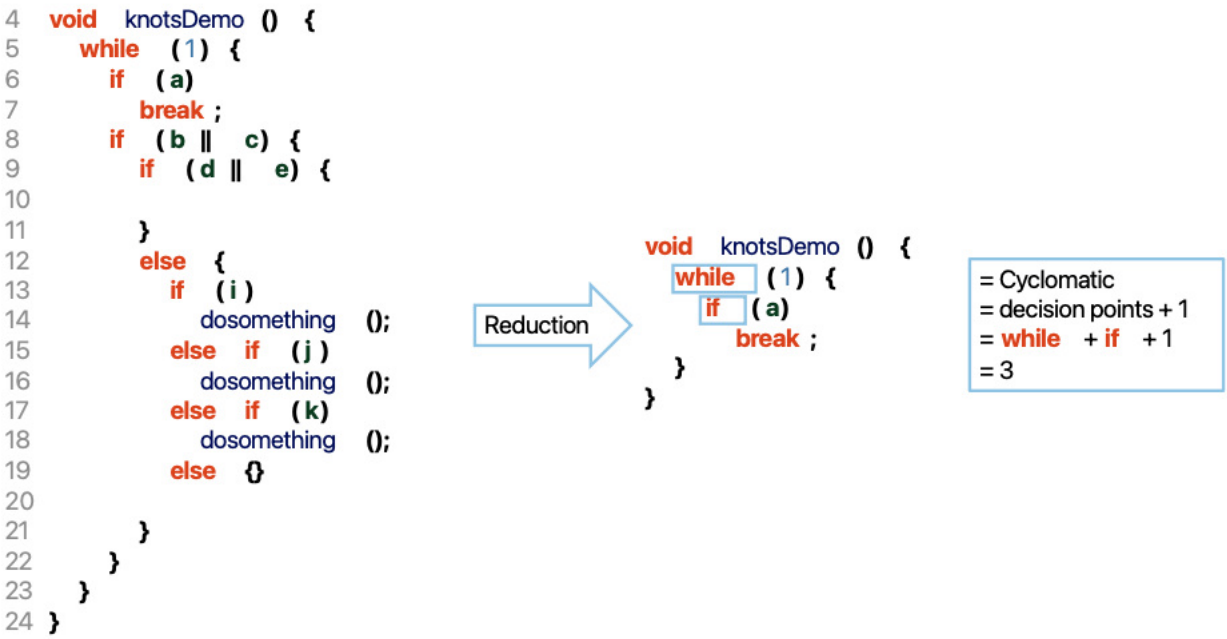
```
1 void knotsDemo() {
2   while (1) {
3     if (a)
4       break;
5     if (b || c) {
6       if (d || e) {
7       }
8       else {
9         if (i)
10           dosomething();
11        else if (j)
12          dosomething();
13        else if (k)
14          dosomething();
15        else {}
16      }
17    }
18  }
```

```
19 }
```

The if (b || c) { ... } block (with its nested if/else if/else chain) is a single-entry, single-exit sequence: every path through it reconverges before the loop repeats, so it reduces away to a single placeholder statement. The outer while (1) { if (a) break; } doesn't reduce: break is a branch out of the loop that reaches the same exit as the loop's own condition check, so that exit has more than one way in. What's left after reduction is just while (1) { if (a) break; } — 2 decision points (while, if), for an Essential Complexity of 3.

See Sum, Max, and Average Essential Complexity for aggregated values, such as for an architecture or the project.

Also known as McCabe - ev(G).



Targets By Language:

- **Ada:** Type, Entry, Function, Package, Procedure, Protected, Task
- **Basic:** Method
- **C#:** Method
- **C++:** Function
- **Fortran:** Module, Block Data, Function, Program, Subroutine
- **Java:** Method
- **Jovial:** Subroutine
- **Pascal:** Compunit, Function, Procedure
- **Python:** File, Function
- **Rust:** File, Function

- **Web:** File

Configuration:

- This metric can be enabled from "Project Configuration/Metrics/Complexity" with the "Essential" option.

Executable Code Lines

API ID: CountLineCodeExe
Languages: Ada, Assembly, C#, C++, Fortran, Java, Pascal, Python
Targets: Architectures, Classes, Files, Functions, Modules, Packages, Project, Subprograms

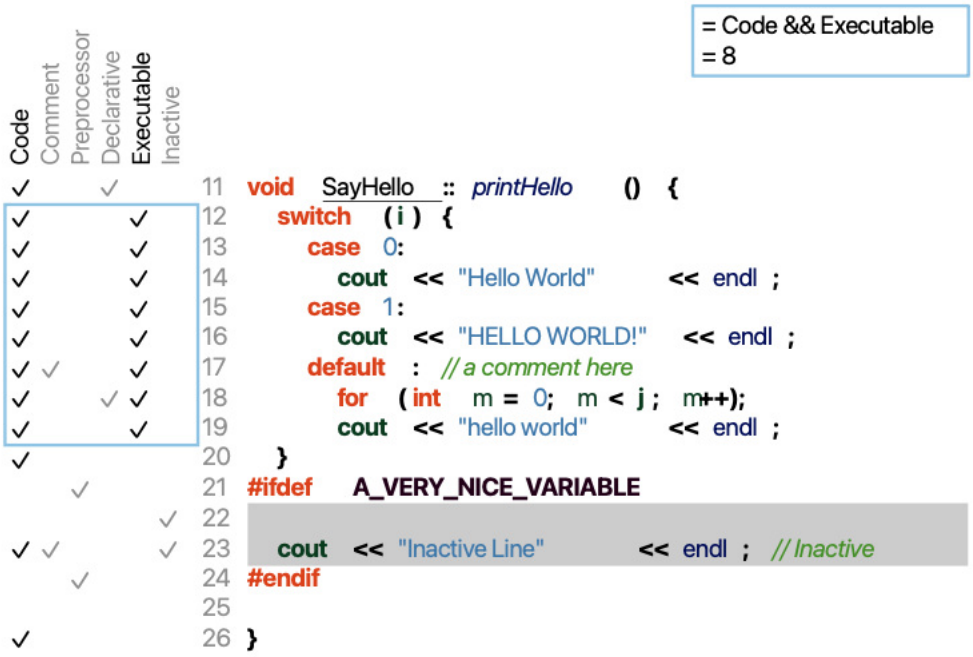
Number of lines containing executable source code.

For example:

```
1 void SayHello::printHello() {
2   switch (i) {
3     case 0:
4       cout << "Hello World" << endl;
5     case 1:
6       cout << "HELLO WORLD!" << endl;
7     default: // a comment here
8       for (int m = 0; m < j; m++);
9       cout << "hello world" << endl;
10   }
11 #ifdef A_VERY_NICE_VARIABLE
12
13   cout << "Inactive Line" << endl; // Inactive
14 #endif
15
16 }
```

Every code line except the function signature is executable: the switch, both cases, the default, the for loop, and the three cout statements (lines 12-19), for CountLineCodeExe = 8.

See Declarative Code Lines for the complementary breakdown, and Code Lines for the total.



Targets By Language:

- **Ada:** Project, File, Type, Entry, Function, Package, Procedure, Protected, Task
- **C#:** Project, File, Class, Method
- **C++:** Project, File, Class, Struct, Union, Function
- **Fortran:** Project, File, Module, Block Data, Function, Program, Subroutine
- **Java:** Project, File, Class, Interface, Method
- **Pascal:** Project, File, Class, Interface, Compunit, Function, Procedure
- **Python:** Project, File, Class, Function

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.

Executable Statements

API ID: CountStmtExe
Languages: Ada, Assembly, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, VHDL, Web
Targets: Architectures, Classes, Files, Functions, Modules, Packages, Project, Subprograms

Number of executable statements.

A statement that both declares a variable and initializes it with a function call counts

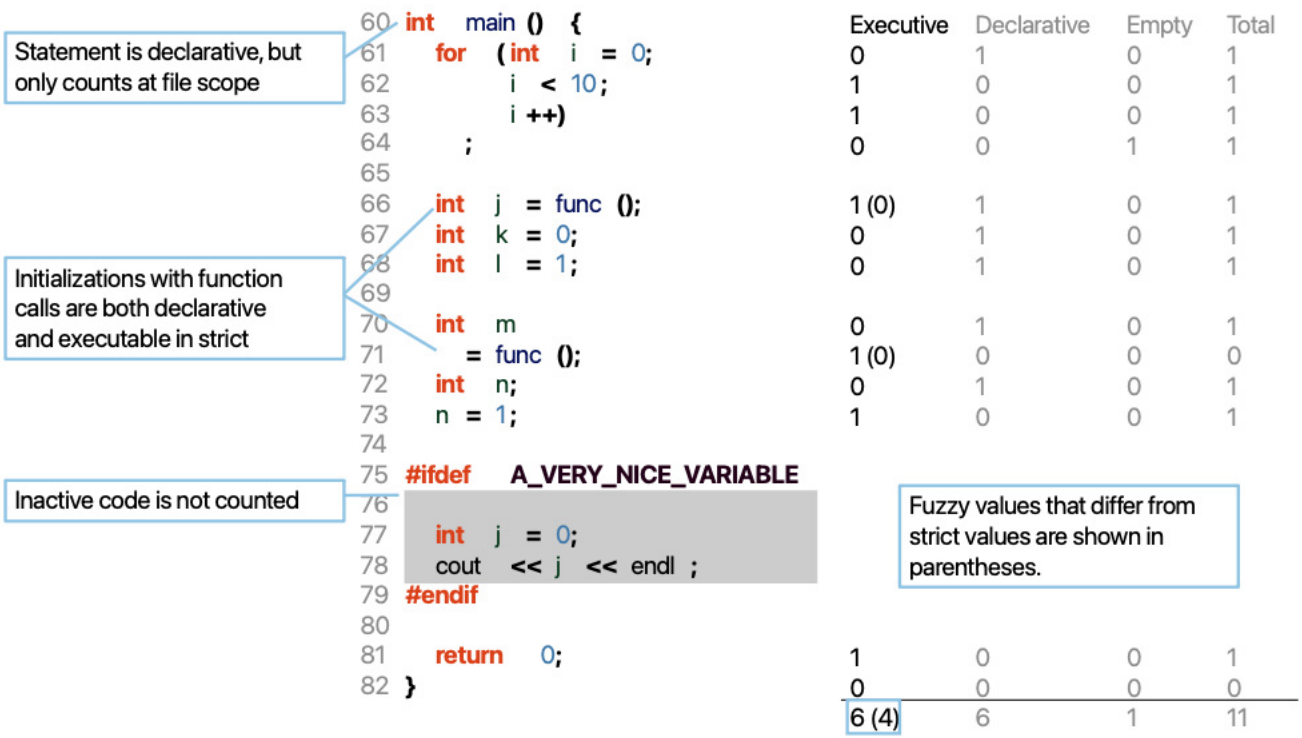
as both executable and declarative.

For example:

```
1 int main() {
2   for (int i = 0;
3        i < 10;
4        i++)
5     ;
6
7   int j = func();
8   int k = 0;
9   int l = 1;
10
11   int m
12     = func();
13   int n;
14   n = 1;
15
16 #ifdef A_VERY_NICE_VARIABLE
17
18   int j = 0;
19   cout << j << endl;
20 #endif
21
22   return 0;
23 }
```

The for loop's condition (i < 10) and increment (i++) clauses, the two initializations that call func() (int j = func(); and int m = func();), n = 1;, and return 0; are all executable, for CountStmtExe = 6.

See Declarative Statements for the complementary breakdown, and Statements for the total. For the per-language breakdown in web files, see Executable Statements (JavaScript) and Executable Statements (PHP).



Targets By Language:

- **Ada:** Project, File, Type, Entry, Function, Package, Procedure, Protected, Task
- **Basic:** Project, File, Method, Module, Class
- **C#:** Project, File, Class, Method
- **C++:** Project, File, Class, Struct, Union, Function
- **Fortran:** Project, File, Module, Block Data, Function, Program, Subroutine
- **Java:** Project, File, Class, Interface, Method
- **Jovial:** Project, File, Subroutine
- **Pascal:** Project, File, Class, Interface, Compunit, Function, Procedure
- **Python:** Project, File, Class, Function
- **VHDL:** Project, File
- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show statement count metrics" option.
- For projects and architectures, this metric is enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show default summary metrics" option.

Executable Statements (JavaScript)

API ID: CountStmtExeJavascript
Languages: Web
Targets: Architectures, Files, Project

Number of JavaScript executable statements.

For example:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Welcome</title>
5
6   <!-- Greeting script: tracks visits and confirms with the user -->
7   <script>
8     // Number of times the page has greeted someone
9     var greetingCount = 0;
10
11     function greetUser(name) {
12       greetingCount++;
13       return "Hello, " + name + "!";
14     }
15
16     let message = greetUser("Ada");
17     console.log(message);
18
19     if (greetingCount > 0) {
20       confirm(message);
21     }
22   </script>
23 </head>
24 <body>
25   <h1>Welcome to the site</h1>
26   <p>Have a look around.</p>
27 </body>
28 </html>
```

greetingCount++; (12), the return statement (13), let message = greetUser("Ada"); (16, which also declares message), console.log(message); (17), the if statement (19), and confirm(message); (20) are executable, giving CountStmtExeJavascript = 6.

See Declarative Statements (JavaScript) for the complementary breakdown, and Executable Statements for the combined count across all languages.

Targets By Language:

- **Web:** Project, File

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show statement count metrics" option.

- This metric can be enabled from "Project Configuration/Metrics/Lines and Statements" with the "Breakdown counts by web language" option.

Executable Statements (PHP)

API ID: CountStmtExePhp
Languages: Web
Targets: Architectures, Classes, Files, Project

Number of PHP executable statements.

For example:

```
<?php
// Compute a visitor count for this session
const SITE_NAME = "Understand Demo";

function nextVisit($count) {
  return $count + 1;
}

$visits = 1;
$visits = nextVisit($visits);
echo "You are visitor number " . $visits . " to " . SITE_NAME;
```

The return statement (6) and the three assignment/output lines ($visits = 1; on 9, $visits = nextVisit($visits); on 10, and the echo statement on 11) are executable, giving CountStmtExePhp = 4.

See Declarative Statements (PHP) for the complementary breakdown, and Executable Statements for the combined count across all languages.

Targets By Language:

- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show statement count metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Lines and Statements" with the "Breakdown counts by web language" option.

Executable Units

API ID: CountDeclExecutableUnit
Languages: Ada, Basic, C#, Fortran, Java, Pascal, Python, Web
Targets: Architectures, Files, Project

Number of program units with executable code.

Counts local methods, functions, and similar callable units that actually have an implementation — not bare declarations, and not abstract/external/forward-declared ones.

For example:

```
1 public class sample {
2
3   public void publicMethod() {
4   }
5
6   void defaultMethod() {
7   }
8
9   private void privateMethod() {
10   }
11
12 }
13
14 class Helper {
15
16   public int value;
17
18   public void doWork() {
19     value++;
20   }
21
22 }
23
24 class Coupler {
25
26   public void useHelper() {
27     Helper h = new Helper();
28     h.doWork();
29
30     java.util.ArrayList<String> list = new java.util.ArrayList<String>();
31     list.add("test");
32     System.out.println(list.size());
33   }
34
35 }
```

sample.java declares 5 concrete methods across its 3 classes — publicMethod, defaultMethod, and privateMethod (in sample), doWork (in Helper), and useHelper (in Coupler) — so CountDeclExecutableUnit = 5 for this file.

Targets By Language:

- **Ada:** Project, File
- **Basic:** Project, File
- **C#:** Project, File
- **Fortran:** Project, File
- **Java:** Project, File
- **Pascal:** Project, File
- **Python:** Project, File
- **Web:** Project, File

Configuration:

- For projects and architectures, this metric is enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated file metrics" option.

Files

API ID: CountDeclFile
Languages: Ada, Assembly, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, VHDL, Web
Targets: Architectures, Project

Number of files.

Counts every file in the project or architecture, regardless of language.

For example, this directory:

```
test/c/
  cohesion.cpp
  csvExportBigNumbers.cpp
  friend.cpp
  io.cpp
  knots.cpp
  paths.cpp
  pond.cpp
  sample.cpp
  vehicles.cpp
  classes.cpp
```

class_macro.h

has 11 files, so CountDeclFile = 11 for a project or architecture covering it.

See Code Files and Header Files for the C/C++-specific breakdown by file type.

Targets By Language:

- **Ada:** Project
- **Basic:** Project
- **C#:** Project
- **C++:** Project
- **Fortran:** Project
- **Java:** Project
- **Jovial:** Project
- **Pascal:** Project
- **Python:** Project
- **Rust:** Project
- **VHDL:** Project
- **Web:** Project

Configuration:

- For projects and architectures, this metric is enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated file metrics" option.
- For projects and architectures, this metric is enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show default summary metrics" option.

Friend Methods

API ID: CountDeclMethodFriend
Also Known As: Number of Friend Methods (NFM), Number of Friends (NF)
Languages: C++
Targets: Classes

Number of local (not inherited) friend methods.

The number of friend functions plus the Methods of any friend classes.

For example:

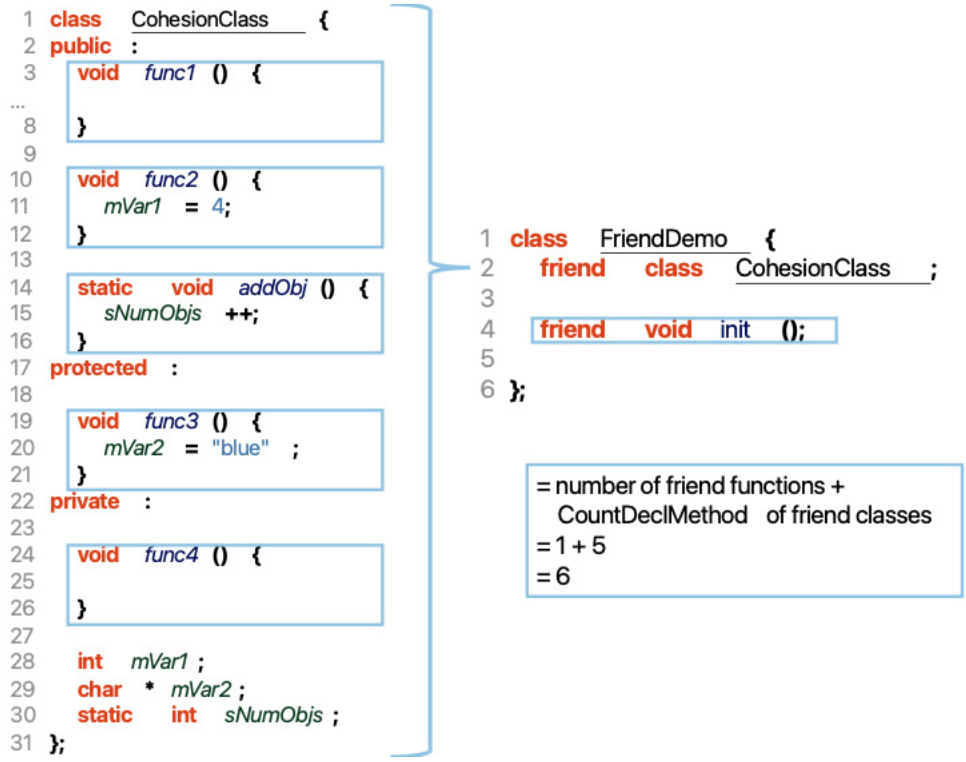
```
1 class FriendDemo {
2   friend class CohesionClass;
3
```

```
4   friend void init();
5
6 };
7
8 class CohesionClass {
9 public:
10   void func1() {
11     for (int i = 0; i < mVar1; i++) {
12       mVar2 = nullptr;
13     }
14     mVar1 = 3;
15   }
16
17   void func2() {
18     mVar1 = 4;
19   }
20
21   static void addObj() {
22     sNumObjs ++;
23   }
24 protected:
25
26   void func3() {
27     mVar2 = "blue";
28   }
29 private:
30
31   void func4() {
32
33   }
34
35   int mVar1;
36   char * mVar2;
37   static int sNumObjs;
38 };
```

FriendDemo declares one friend function, init(), and one friend class, CohesionClass. CohesionClass itself has 5 local methods (func1, func2, func3, func4, addObj), i.e. CountDeclMethod = 5 for CohesionClass. So FriendDemo's CountDeclMethodFriend = 1 + 5 = 6 — the friend function plus all of the friend class's own methods, not just the ones that actually use FriendDemo.

See Methods for a class's own local method count, which is what gets added in for each friend class, and the Friend References interactive report to see how, if at all, a given friend actually uses that access.

Also known as Lorenz & Kidd - Number of Friends (NF); Number of Friend Methods (NFM).



Targets By Language:

- **C++:** Class, Struct, Union

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Functions

API ID: CountDeclFunction
Languages: Assembly, C#, C++, Java, Python, Rust, Web
Targets: Architectures, Files, Project

Number of functions.

Combines free functions and member functions (methods) into one count, and only counts ones that are actually defined, not just declared.

For example:

```
1 class SayHello {
2 public:
3   SayHello() {}
4   void printHello();
5 };
6
```

```
7 int i,j;
8 void SayHello::printHello() {
9   switch (i) {
10     case 0:
11       cout << "Hello World" << endl;
12     case 1:
13       cout << "HELLO WORLD!" << endl;
14     default: // a comment here
15       for (int m = 0; m < j; m++);
16       cout << "hello world" << endl;
17   }
18 #ifdef A_VERY_NICE_VARIABLE
19
20   cout << "Inactive Line" << endl; // Inactive
21 #endif
22
23 }
24
25 void cyclomaticDemo() {
26   bool a = true, b = true, c = true;
27
28   if (a || (b && c)) {
29     while (a ? b : c) {
30       for (int i = 0; i < 10; i++) {
31         switch(i) {
32           case 1:
33           case 2:
34             cout<<i<<endl;
35             break;
36           case 5:
37             break;
38           default:
39             cout <<i<<endl;
40         }
41       }
42     }
43   } else {
44     try {
45       do {
46         cout << a << b << c << endl;
47       } while(a);
48     }
49     catch(...){
50
51     }
52   }
53 }
54
55
56 int func();
57 int main() {
58   for (int i = 0;
59        i < 10;
```

```
60         i++)
61     ;
62
63   int j = func();
64   int k = 0;
65   int l = 1;
66
67   int m
68     = func();
69   int n;
70   n = 1;
71
72 #ifdef A_VERY_NICE_VARIABLE
73
74   int j = 0;
75   cout << j << endl;
76 #endif
77
78   return 0;
79 }
```

sample.cpp defines 4 functions: the SayHello() constructor and SayHello::printHello() (methods), plus the free functions cyclomaticDemo() and main() — so CountDeclFunction = 4. func() is only declared, never defined in this file, so it doesn't count.

See Classes for the equivalent count of class-like declarations, and Subprograms for the same concept in Ada, Basic, Fortran, Jovial, and Pascal, which use that term instead.

Targets By Language:

- **C#:** Project, File
- **C++:** Project, File
- **Java:** Project, File
- **Python:** Project, File
- **Rust:** Project, File
- **Web:** Project, File

Configuration:

- For projects and architectures, this metric is enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated file metrics" option.
- For projects and architectures, this metric is enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show default summary metrics" option.

Header Files

API ID: CountDeclFileHeader
Languages: C++
Targets: Architectures, Project

Number of header files.

Normally a .h/.hpp file, though it's based on how the parser identifies the file, not just its extension.

For example, this directory:

```
test/c/
  cohesion.cpp
  csvExportBigNumbers.cpp
  friend.cpp
  io.cpp
  knots.cpp
  paths.cpp
  pond.cpp
  sample.cpp
  vehicles.cpp
  classes.cpp
  class_macro.h
```

has 1 header file, class_macro.h, so CountDeclFileHeader = 1.

See Code Files for the complementary count, and Files for the total across all languages.

Targets By Language:

- **C++:** Project

Configuration:

- For projects and architectures, this metric is enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated file metrics" option.

Inactive Lines

API ID: CountLineInactive
Languages: C#, C++, Pascal
Targets: Architectures, Classes, Files, Functions, Modules, Project, Subprograms

Number of inactive lines.

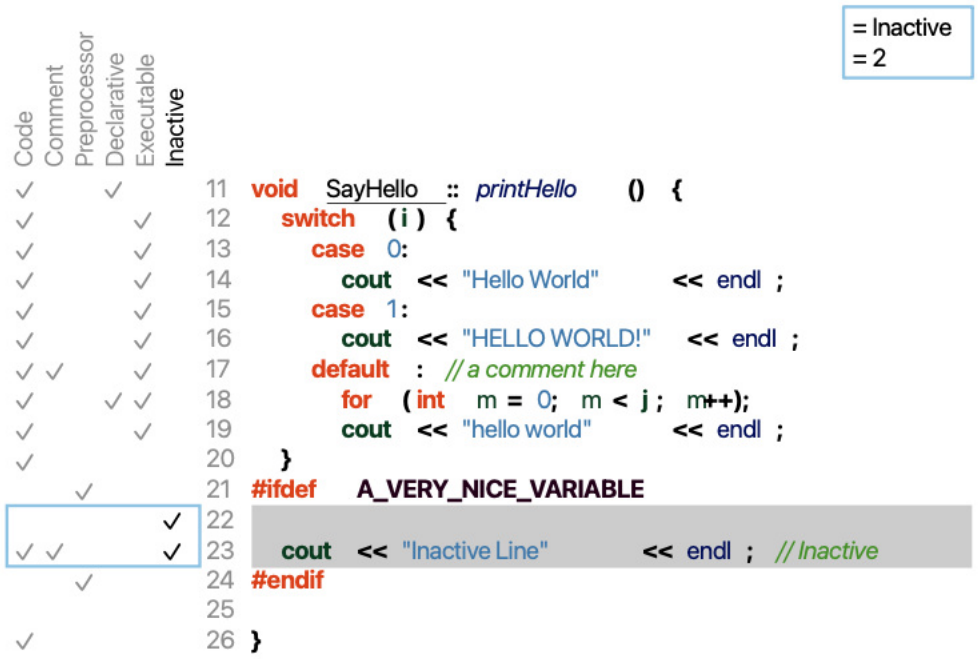
This is the number of lines that are inactive from the view of the preprocessor. In other words, they are on the FALSE side of a #if or #ifdef preprocessor directive.

For example:

```
1 void SayHello::printHello() {
2   switch (i) {
3     case 0:
4       cout << "Hello World" << endl;
5     case 1:
6       cout << "HELLO WORLD!" << endl;
7     default: // a comment here
8       for (int m = 0; m < j; m++);
9       cout << "hello world" << endl;
10   }
11 #ifdef A_VERY_NICE_VARIABLE
12
13   cout << "Inactive Line" << endl; // Inactive
14 #endif
15
16 }
```

The blank line and the cout << "Inactive Line" statement between #ifdef A_VERY_NICE_VARIABLE and #endif (lines 22 and 23) are both inactive, since A_VERY_NICE_VARIABLE isn't defined, giving CountLineInactive = 2. The #ifdef and #endif directive lines themselves are always considered active, so they count towards Preprocessor Lines instead of here.

See Code Lines, Comment Lines, and Blank Lines (Includes Inactive) to count what's inside inactive regions by category.



Targets By Language:

- **C#:** Project, File, Class, Method
- **C++:** Project, File, Class, Struct, Union, Function
- **Pascal:** Project, File, Class, Interface, Compunit, Function, Procedure

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- For projects and architectures, this metric is enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show default summary metrics" option.

Inputs

API ID: CountInput
Also Known As: FANIN (Infomational fan-in)
Languages: C#, C++, Fortran, Java
Targets: Functions, Subprograms

Number of calling subprograms plus global variables read.

The number of inputs a function uses plus the number of unique subprograms calling the function. Inputs include parameters and global variables that are used in the function, so Functions calledby + Parameters read + Global Variables read.

Recursive function calls and local variables that are not class static variables are not included. Of the two general approaches to calculating FANIN (informational versus structural) ours is the informational approach.

For example:

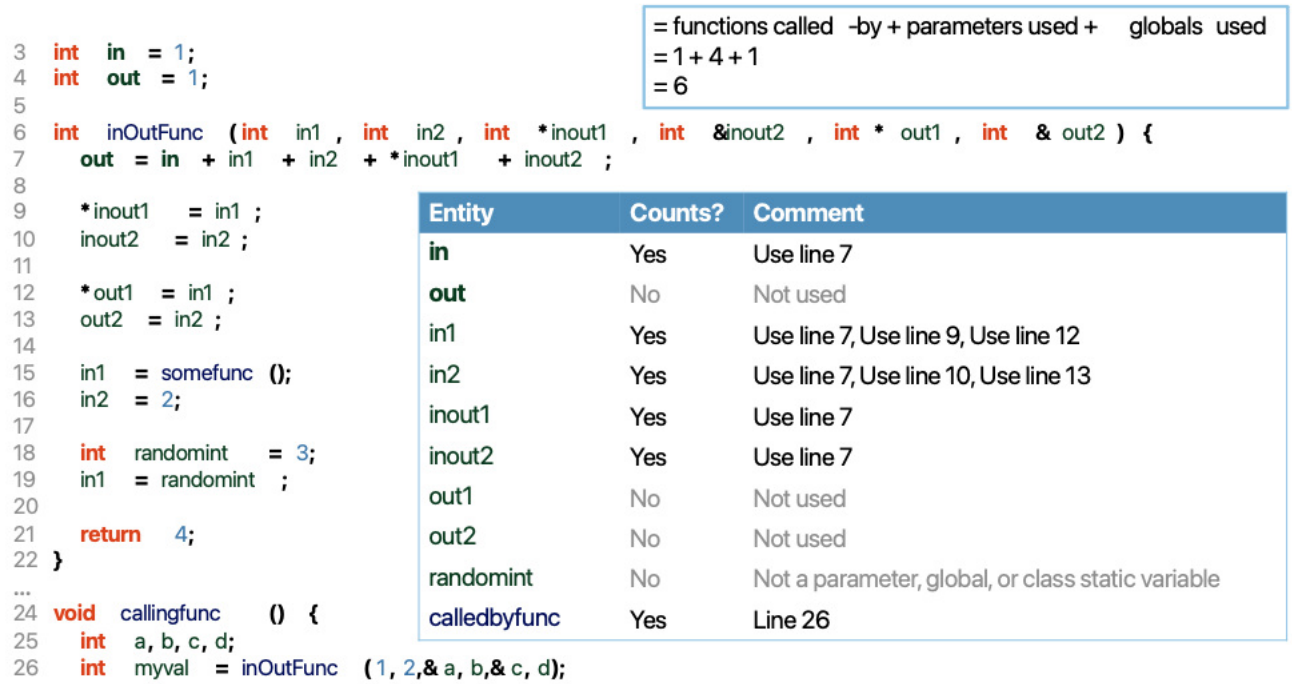
```
1 int somefunc();
2
3 int in = 1;
4 int out = 1;
5
6 int inOutFunc(int in1, int in2, int *inout1, int &inout2, int* out1, int &
7   out = in + in1 + in2 + *inout1 + inout2;
8
9   *inout1 = in1;
10   inout2 = in2;
11
12   *out1 = in1;
13   out2 = in2;
14
15   in1 = somefunc();
16   in2 = 2;
17
18   int randomint = 3;
19   in1 = randomint;
20
21   return 4;
22 }
23
24 void callingfunc() {
25   int a,b,c,d;
26   int myval = inOutFunc(1,2,&a,b,&c,d);
27 }
```

inOutFunc reads the global in and the parameters in1, in2, inout1, and inout2, all used together in the expression that sets out. out, out1, and out2 are only ever written, never read, so they don't count, and randomint is a local variable rather than a parameter, global, or class static variable, so it's excluded even though it's used. That's 1 global + 4 parameters read, plus 1 calling function (callingfunc), for an Inputs count of 6.

See Outputs for the related fan-out metric, and the Inputs Metric interactive report to list the references contributing to it for a given function.

Also known as FANIN (Infomational fan-in).

Introduced by Henry, Sallie, and Dennis Kafura in "Software Structure Metrics Based on Information Flow." IEEE Transactions on Software Engineering, vol. SE-7, no. 5, 1981, pp. 510-518.



```
3  int  in  = 1;
4  int  out  = 1;
5
6  int  inOutFunc  (int  in1 , int  in2 , int  *inout1  , int  &inout2  , int * out1 , int  & out2 ) {
7    out  = in  + in1  + in2  + *inout1  + inout2  ;
8
9    *inout1  = in1 ;
10   inout2  = in2 ;
11
12   *out1  = in1 ;
13   out2  = in2 ;
14
15   in1  = somefunc  ();
16   in2  = 2;
17
18   int  randomint  = 3;
19   in1  = randomint  ;
20
21   return  4;
22 }
...
24 void  callingfunc  () {
25   int  a, b, c, d;
26   int  myval  = inOutFunc  (1, 2,& a, b,& c, d);
```

= functions called -by + parameters used + globals used
= 1 + 4 + 1
= 6

Entity	Counts?	Comment
in	Yes	Use line 7
out	No	Not used
in1	Yes	Use line 7, Use line 9, Use line 12
in2	Yes	Use line 7, Use line 10, Use line 13
inout1	Yes	Use line 7
inout2	Yes	Use line 7
out1	No	Not used
out2	No	Not used
randomint	No	Not a parameter, global, or class static variable
calledbyfunc	Yes	Line 26

Targets By Language:

- **C#:** Method
- **C++:** Function
- **Fortran:** Function, Program, Subroutine
- **Java:** Method

Instance Methods

API ID: CountDeclInstanceMethod
Also Known As: Number of Instance Methods (NIM)
Languages: Basic, C#, C++, Java, Pascal, Python, Web
Targets: Architectures, Classes, Files, Project

Number of instance methods.

Methods defined in a class that are only accessable through an object of that class.

For example:

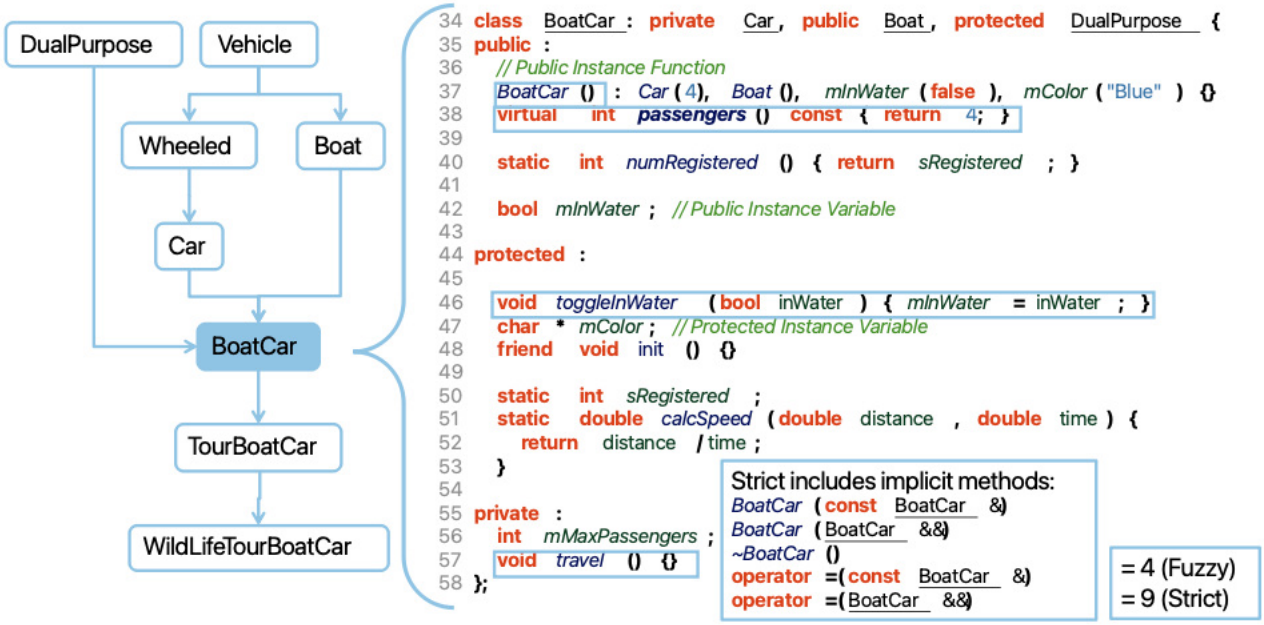
```
1 class BoatCar: private Car, public Boat, protected DualPurpose {
2 public:
3   // Public Instance Function
4   BoatCar() : Car(4), Boat(), mInWater(false), mColor("Blue") {}
5   virtual int passengers() const { return 4; }
6
```

```
7    static int numRegistered() { return sRegistered; }
8
9    bool mInWater; // Public Instance Variable
10
11 protected:
12
13    void toggleInWater(bool inWater) { mInWater = inWater; }
14    char * mColor; // Protected Instance Variable
15    friend void init() {}
16
17    static int sRegistered;
18    static double calcSpeed(double distance, double time) {
19      return distance/time;
20    }
21
22 private:
23    int mMaxPassengers;
24    void travel() {}
25 };
```

BoatCar explicitly writes 4 instance methods: the BoatCar() constructor, passengers(), toggleInWater(bool), and travel(). Understand also counts the compiler-generated special member functions that C++ synthesizes automatically whenever they aren't explicitly declared: the copy constructor BoatCar(const BoatCar&), move constructor BoatCar(BoatCar&&), destructor ~BoatCar(), copy assignment operator=(const BoatCar&), and move assignment operator=(BoatCar&&). That's 4 explicit plus 5 implicit, for CountDeclInstanceMethod = 9.

See Class Methods for static methods, and Methods for the count of only what's actually written in the source.

Also known as Lorenz & Kidd - Number of Instance Methods (NIM).



Targets By Language:

- **Basic:** Project, Class, Struct
- **C#:** Project, Class, Struct
- **C++:** Project, Class, Struct, Union
- **Java:** Project, File, Class, Interface
- **Pascal:** Project, Class, Interface
- **Python:** Project, Class
- **Web:** Project, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated class metrics" option. Enabling this option can be time consuming.
- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Instance Variables

API ID: CountDeclInstanceVariable
Also Known As: Number of Instance Variables (NIV)
Languages: Basic, C#, C++, Java, Pascal, Python, Web
Targets: Architectures, Classes, Files, Project

Number of instance variables.

Variables defined in a class that are only accessible through an object of that class.

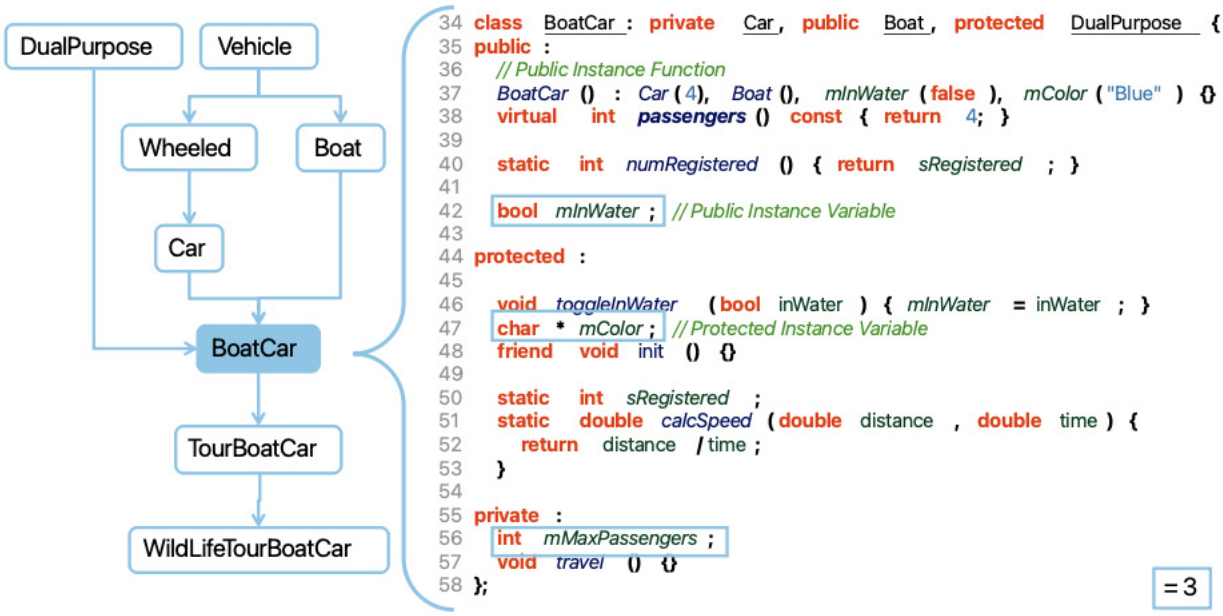
For example:

```
1 class BoatCar: private Car, public Boat, protected DualPurpose {
2 public:
3   // Public Instance Function
4   BoatCar() : Car(4), Boat(), mInWater(false), mColor("Blue") {}
5   virtual int passengers() const { return 4; }
6
7   static int numRegistered() { return sRegistered; }
8
9   bool mInWater; // Public Instance Variable
10
11 protected:
12
13   void toggleInWater(bool inWater) { mInWater = inWater; }
14   char * mColor; // Protected Instance Variable
15   friend void init() {}
16
17   static int sRegistered;
18   static double calcSpeed(double distance, double time) {
19     return distance/time;
20   }
21
22 private:
23   int mMaxPassengers;
24   void travel() {}
25 };
```

BoatCar declares three instance variables across all access levels — mInWater (public), mColor (protected), and mMaxPassengers (private) — so CountDeclInstanceVariable = 3.

See Public Instance Variables, Protected Instance Variables, and Private Instance Variables for the breakdown by access level, and Class Variables for the static counterpart. C# also has additional access levels: see Internal Instance Variables and Protected Internal Instance Variables.

Also known as Lorenz & Kidd - Number of Instance Variables (NIV).



Targets By Language:

- **Basic:** Project, Class, Struct
- **C#:** Project, Class, Struct
- **C++:** Project, Class, Struct, Union
- **Java:** Project, File, Class, Interface
- **Pascal:** Project, Class, Interface
- **Python:** Project, Class
- **Web:** Project, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated class metrics" option. Enabling this option can be time consuming.
- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Internal Instance Variables

API ID: CountDeclInstanceVariableInternal
Languages: C#
Targets: Architectures, Classes, Project

Number of internal instance variables.

An internal instance variable (C#'s internal access modifier) is accessible from anywhere in the same assembly, but not from other assemblies.

For example:

```
1 namespace MetricsDemo
2 {
3     public class AccessModifiers
4     {
5         internal int InternalField;
6         protected internal int ProtectedInternalField;
7 
8         internal void InternalMethod()
9         {
10             InternalField++;
11         }
12 
13         protected internal void ProtectedInternalMethod()
14         {
15             ProtectedInternalField++;
16         }
17     }
18 }
```

InternalField is the only internal instance variable, giving CountDeclInstanceVariableInternal = 1. ProtectedInternalField is protected internal — a different, broader access level (accessible from the same assembly OR from derived classes in other assemblies) — so it doesn't count here.

See Instance Variables for the total across all access levels, and Protected Internal Instance Variables for the related access level. Public Instance Variables, Protected Instance Variables, and Private Instance Variables cover the access levels shared with C++ and Java.

Targets By Language:

- **C#:** Project, Class, Struct

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated class metrics" option. Enabling this option can be time consuming.
- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Internal Methods

API ID: CountDeclMethodInternal

Languages: C#
Targets: Architectures, Classes, Project

Number of local internal methods.

An internal method (C#'s internal access modifier) is accessible from anywhere in the same assembly, but not from other assemblies.

For example:

```
1 namespace MetricsDemo
2 {
3     public class AccessModifiers
4     {
5         internal int InternalField;
6         protected internal int ProtectedInternalField;
7
8         internal void InternalMethod()
9         {
10            InternalField++;
11         }
12
13         protected internal void ProtectedInternalMethod()
14         {
15            ProtectedInternalField++;
16         }
17     }
18 }
```

InternalMethod() is the only internal method, giving CountDeclMethodInternal = 1. ProtectedInternalMethod() is protected internal — a different, broader access level — so it doesn't count here.

See Methods for the total across all access levels, and Protected Internal Methods for the related access level. Public Methods, Protected Methods, and Private Methods cover the access levels shared with C++ and Java, and Default Methods covers Java's package-private methods.

Targets By Language:

- **C#:** Project, Class, Struct

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated class metrics" option. Enabling this option can be time consuming.
- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Knots

API ID: Knots
Languages: Ada, C#, C++, Java, Rust
Targets: Functions, Packages, Subprograms

Measure of overlapping jumps.

If a piece of code has arrowed lines indicating where every jump in the flow of control occurs, a knot is defined as where two such lines cross each other. The number of knots is proportional to the complexity of the control flow.

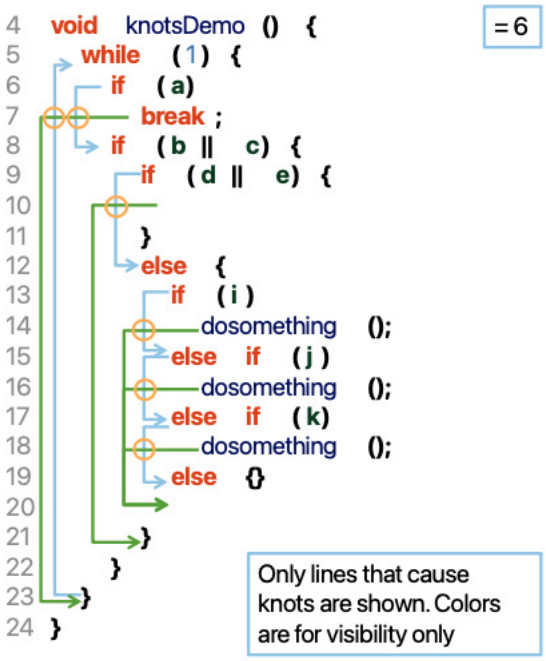
For example:

```
1 void knotsDemo() {
2   while (1) {
3     if (a)
4       break;
5     if (b || c) {
6       if (d || e) {
7       }
8       else {
9         if (i)
10           dosomething();
11        else if (j)
12          dosomething();
13        else if (k)
14          dosomething();
15        else {}
16      }
17    }
18  }
19 }
```

Each if, while, and break is a jump: draw an arrow from each one to every line it can transfer control to, and count how many pairs of those arrows cross. For this function, that's 6 knots.

Unlike Essential Complexity, which measures unstructured control flow by reducing the graph and taking the complexity of what's left, Knots measures it directly by counting crossings in the raw (unreduced) control-flow graph. See Max and Min Essential Knots for the knot count within the reduced graph.

Introduced by Woodward, M.R., M.A. Hennell, and D. Hedley in "A Measure of Control Flow Complexity in Program Text." IEEE Transactions on Software Engineering, vol. SE-5, no. 1, 1979, pp. 45-50.



Targets By Language:

- **Ada:** Type, Entry, Function, Package, Procedure, Protected, Task
- **C#:** Method
- **C++:** Function
- **Java:** Method
- **Rust:** Function

Configuration:

- This metric can be enabled from "Project Configuration/Metrics/Complexity" with the "Knots" option.

Lines

API ID: CountLine
Also Known As: Number of Lines (NL)
Languages: Ada, Assembly, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, VHDL, Web
Targets: Architectures, Classes, Files, Functions, Modules, Packages, Project, Subprograms

Number of physical lines.

Calculated as End_Line - Start_Line + 1, so it counts every line in the entity's extent — code, comments, blank lines, preprocessor directives, and inactive regions alike.

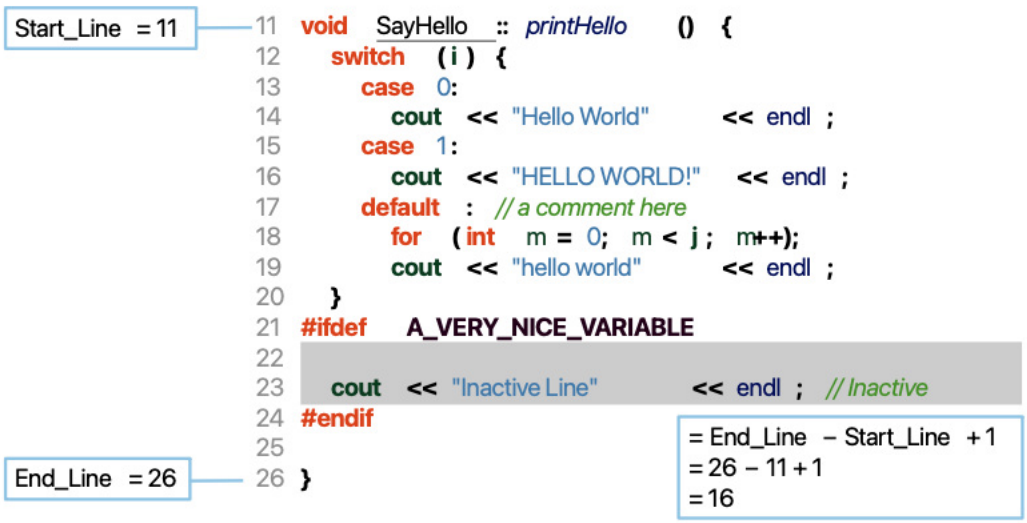
For example:

```
1 void SayHello::printHello() {
2   switch (i) {
3     case 0:
4       cout << "Hello World" << endl;
5     case 1:
6       cout << "HELLO WORLD!" << endl;
7     default: // a comment here
8       for (int m = 0; m < j; m++);
9       cout << "hello world" << endl;
10   }
11 #ifdef A_VERY_NICE_VARIABLE
12
13   cout << "Inactive Line" << endl; // Inactive
14 #endif
15
16 }
```

printHello() starts at line 11 and ends at line 26, for CountLine = 26 - 11 + 1 = 16.

See Code Lines, Comment Lines, Blank Lines, Preprocessor Lines, and Inactive Lines for a breakdown by line type. For the per-language breakdown in web files, see Lines (HTML), Lines (JavaScript), and Lines (PHP).

Also known as Number of Lines (NL).



Targets By Language:

- **Ada:** Project, File, Type, Entry, Function, Package, Procedure, Protected, Task
- **Basic:** Project, File, Method, Module, Class
- **C#:** Project, File, Class, Method
- **C++:** Project, File, Class, Struct, Union, Function
- **Fortran:** Project, File, Module, Block Data, Function, Program, Subroutine
- **Java:** Project, File, Class, Interface, Method
- **Jovial:** Project, File, Module, Subroutine
- **Pascal:** Project, File, Class, Interface, Compunit, Function, Procedure
- **Python:** Project, File, Class, Function
- **Rust:** Project, File, Function
- **VHDL:** Project, File, Procedure, Function, Process, Architecture
- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- For projects and architectures, this metric is enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show default summary metrics" option.

Lines (HTML)

API ID: CountLineHtml
Languages: Web
Targets: Architectures, Files, Project

Number of all HTML lines.

Counts every line of an HTML file that isn't inside an embedded sub-language block, including the tag lines that open and close those blocks (such as <script>/</script>).

For example:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Welcome</title>
5
6   <!-- Greeting script: tracks visits and confirms with the user -->
7   <script>
8     // Number of times the page has greeted someone
9     var greetingCount = 0;
10
11     function greetUser(name) {
12       greetingCount++;
```

```
13        return "Hello, " + name + "!";
14      }
15
16      let message = greetUser("Ada");
17      console.log(message);
18
19      if (greetingCount > 0) {
20        confirm(message);
21      }
22    </script>
23  </head>
24  <body>
25    <h1>Welcome to the site</h1>
26    <p>Have a look around.</p>
27  </body>
28  </html>
```

sample.html is 28 lines total. The <script> block's contents (lines 8-21) are JavaScript, but the <script> (7) and </script> (22) tag lines themselves still count as HTML, along with the rest of the page (1-6, 23-28), giving CountLineHtml = 14.

See Lines (JavaScript) for the embedded script's contents, and Blank Lines (HTML)/ Comment Lines (HTML) for the HTML breakdown. See Lines for the language-independent total across the whole file.

Targets By Language:

- **Web:** Project, File

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Lines and Statements" with the "Breakdown counts by web language" option.

Lines (JavaScript)

API ID: CountLineJavascript
Languages: Web
Targets: Architectures, Files, Project

Number of all JavaScript lines.

Counts the lines inside an embedded <script> block (or a linked .js file), not including the <script>/</script> tag lines themselves, which count as HTML.

For example:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Welcome</title>
5
6   <!-- Greeting script: tracks visits and confirms with the user -->
7   <script>
8     // Number of times the page has greeted someone
9     var greetingCount = 0;
10
11     function greetUser(name) {
12       greetingCount++;
13       return "Hello, " + name + "!";
14     }
15
16     let message = greetUser("Ada");
17     console.log(message);
18
19     if (greetingCount > 0) {
20       confirm(message);
21     }
22   </script>
23 </head>
24 <body>
25   <h1>Welcome to the site</h1>
26   <p>Have a look around.</p>
27 </body>
28 </html>
```

Lines 8-21, the full body of the <script> block, are JavaScript, giving CountLineJavascript = 14. The <script> (7) and </script> (22) tag lines themselves count towards CountLineHtml instead.

See Lines (HTML) for the surrounding page, and Code Lines (JavaScript), Comment Lines (JavaScript), and Blank Lines (JavaScript) for the JavaScript breakdown. See Lines for the language-independent total across the whole file.

Targets By Language:

- **Web:** Project, File

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Lines and Statements" with the "Breakdown counts by web language" option.

Lines (PHP)

API ID: CountLinePhp
Languages: Web
Targets: Architectures, Classes, Files, Project

Number of all PHP lines.

Every line of a .php file counts here, from the opening <?php tag onward. Unlike an HTML file's embedded <script> block (see Lines (JavaScript)), markup mixed into a PHP file isn't broken out into a separate HTML or JavaScript count — it's all PHP.

For example:

```
<?php
// Compute a visitor count for this session
const SITE_NAME = "Understand Demo";

function nextVisit($count) {
  return $count + 1;
}

$visits = 1;
$visits = nextVisit($visits);
echo "You are visitor number " . $visits . " to " . SITE_NAME;
```

sample.php is 11 lines total, all of which count as PHP (including the opening <?php tag on line 1), giving CountLinePhp = 11.

See Blank Lines (PHP) and Comment Lines (PHP) for the PHP breakdown, and Lines (HTML)/Lines (JavaScript) for the equivalents on an HTML file with an embedded script. See Lines for the language-independent total across the whole file.

Targets By Language:

- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Lines and Statements" with the "Breakdown counts by web language" option.

Max Cyclomatic Complexity

API ID: MaxCyclomatic
Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, Web
Targets: Architectures, Classes, Files, Modules, Packages, Project

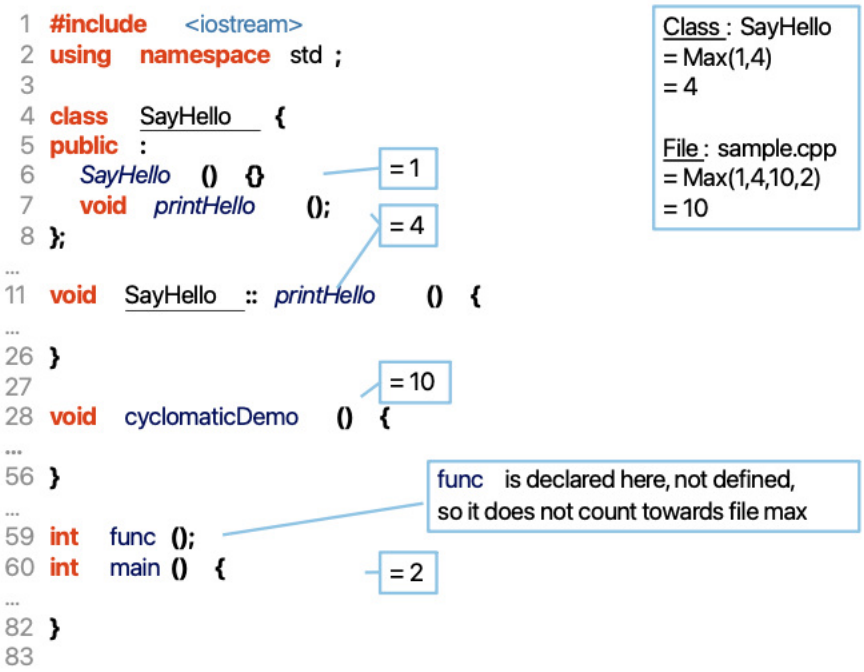
Maximum cyclomatic complexity of all nested functions or methods.

For example:

```
1 class SayHello {
2 public:
3   SayHello() {}
4   void printHello();
5 };
6
7 void SayHello::printHello() {
8   switch (i) {
9     ...
10   }
11 }
12
13 void cyclomaticDemo() {
14   ...
15 }
16
17 int func();          // declared here, not defined - excluded from aggreg
18
19 int main() {
20   ...
21 }
```

SayHello(), printHello(), cyclomaticDemo(), and main() have Cyclomatic Complexity of 1, 4, 10, and 2. func() is only declared, not defined, so it isn't counted. For the SayHello class, the maximum is 4. For the whole file (sample.cpp), the maximum is 10.

See Sum and Average Cyclomatic Complexity for the other aggregations.



Targets By Language:

- **Ada:** Project, File, Package
- **Basic:** Project, File, Module, Class, Struct
- **C#:** Project, File, Class, Struct
- **C++:** Project, File, Class, Struct, Union
- **Fortran:** Project, File
- **Java:** Project, File, Class, Interface
- **Jovial:** Project, File
- **Pascal:** Project, File, Class, Interface, Compunit
- **Python:** Project, File, Class
- **Rust:** Project, File
- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated function metrics" option.
- The cyclomatic metric used is configured from "Project Configuration/Metrics/Complexity". This metric results from having "Count each decision in a multi-way-decision structure" on and having "Count logical conjuctions (OR) and logical and (AND)" off. Checking "Show all available cyclomatic complexity metric variants" will also enable the metric.

Max Essential Complexity

API ID: MaxEssential
Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, Web
Targets: Architectures, Classes, Files, Modules, Packages, Project

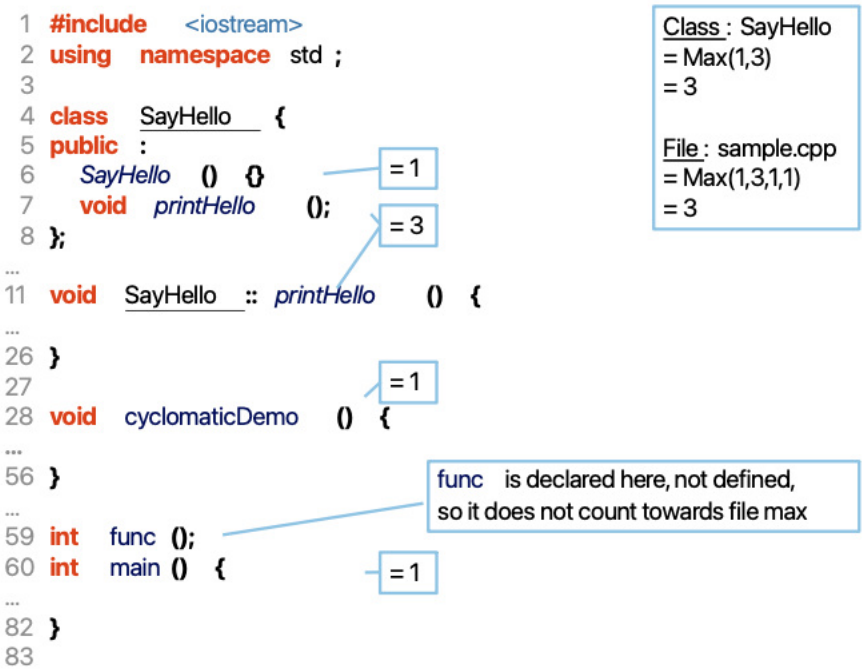
Maximum essential complexity of all nested functions or methods.

For example:

```
1 class SayHello {
2 public:
3   SayHello() {}
4   void printHello();
5 };
6
7 void SayHello::printHello() {
8   switch (i) {
9     ...
10   }
11 }
12
13 void cyclomaticDemo() {
14   ...
15 }
16
17 int func();          // declared here, not defined - excluded from aggreg
18
19 int main() {
20   ...
21 }
```

SayHello(), printHello(), cyclomaticDemo(), and main() have Essential Complexity of 1, 3, 1, and 1. func() is only declared, not defined, so it isn't counted. For the SayHello class, the maximum is 3. For the whole file (sample.cpp), the maximum is 3.

See Sum and Average Essential Complexity for the other aggregations.



Targets By Language:

- **Ada:** Project, File, Package
- **Basic:** Project, File, Module, Class, Struct
- **C#:** Project, File, Class, Struct
- **C++:** Project, File, Class, Struct, Union
- **Fortran:** Project, File
- **Java:** Project, File, Class, Interface
- **Jovial:** Project, File
- **Pascal:** Project, File, Class, Interface, Compunit
- **Python:** Project, File, Class
- **Rust:** Project, File
- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated function metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Complexity" with the "Essential" option.

Max Essential Knots

API ID: MaxEssentialKnots
Languages: Ada, C#, C++, Java, Rust
Targets: Functions, Packages, Subprograms

Maximum Knots after structured programming constructs have been removed.

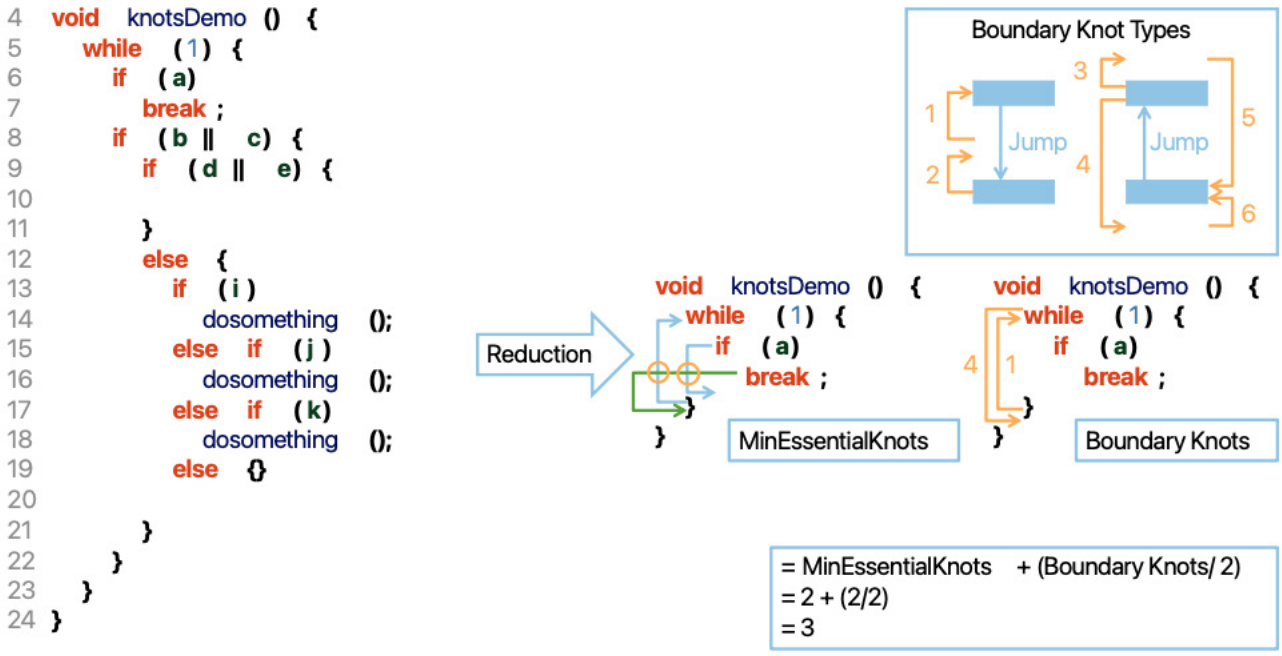
The reduction Essential Complexity uses can hide crossings that existed inside a collapsed region — a jump whose original path passed through part of what's now a single placeholder statement. Max Essential Knots estimates those as "boundary knots": for each jump remaining in the reduced graph, it checks whether the collapsed region it crosses could have hidden such a crossing. Since each potential crossing is detected from both ends of the jump, the raw boundary knot count is halved before being added to Min Essential Knots: MaxEssentialKnots = MinEssentialKnots + BoundaryKnots / 2, capped at the Knots count of the original, unreduced graph.

For example:

```
1 void knotsDemo() {
2   while (1) {
3     if (a)
4       break;
5     if (b || c) {
6       if (d || e) {
7       }
8       else {
9         if (i)
10           dosomething();
11         else if (j)
12           dosomething();
13         else if (k)
14           dosomething();
15         else {}
16       }
17     }
18   }
19 }
```

For knotsDemo, the reduced graph (while (1) { if (a) break; }) has a Min Essential Knots of 2. The while loop's own jump back to its condition spans over where the collapsed if (b || c) { ... } region used to be, so it's flagged as a possible hidden crossing: a boundary knot (raw count of 2, halved since it's detected from both ends of the jump) contributing 2 / 2 = 1, for a Max Essential Knots of 2 + 1 = 3.

See Min Essential Knots for the knot count without that estimate, and Knots for the count on the unreduced graph.



Targets By Language:

- **Ada:** Type, Entry, Function, Package, Procedure, Protected, Task
- **C#:** Method
- **C++:** Function
- **Java:** Method
- **Rust:** Function

Configuration:

- This metric can be enabled from "Project Configuration/Metrics/Complexity" with the "Knots" option.

Max Inheritance Tree

API ID: MaxInheritanceTree
Also Known As: Depth of Inheritance Tree (DIT)
Languages: C#, C++, Java, Pascal, Python, Web
Targets: Classes

Maximum depth of class in inheritance tree.

The depth of a class within the inheritance hierarchy is the maximum number of nodes from the class node to the root of the inheritance tree. The root node has a DIT of 0. The deeper within the hierarchy, the more methods the class can inherit, increasing its complexity. For multiple inheritance, the depth is 1 plus the maximum

depth among all of the class's direct base classes, not the sum.

For example:

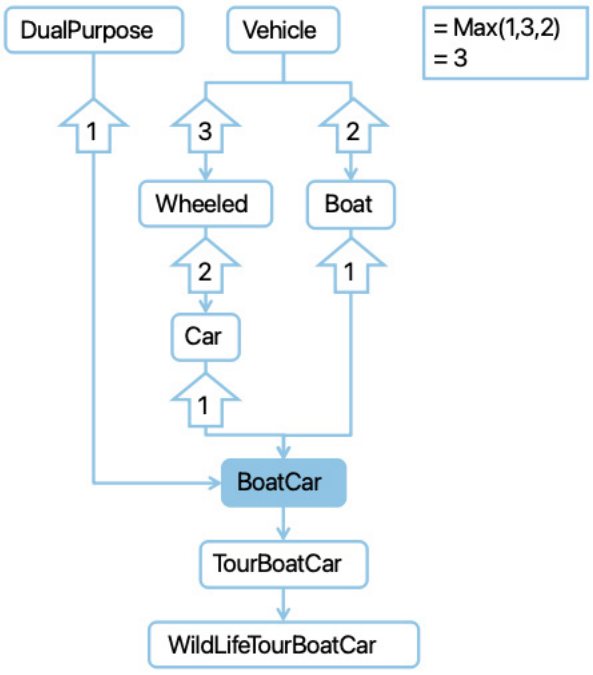
```
1 class Vehicle {
2 public:
3   Vehicle() {}
4 };
5
6 class Wheeled : public Vehicle {
7 public:
8   Wheeled(int wheels) : mWheels(wheels) {}
9   int wheels() { return mWheels; }
10   void setWheels(int wheels) { mWheels = wheels; }
11 private:
12   int mWheels;
13 };
14
15 class Car : public Wheeled {
16 public:
17   Car() : Wheeled(4) {}
18   Car(int wheels) : Wheeled(wheels) {}
19 };
20
21 class Boat : public Vehicle {
22 public:
23   Boat() {}
24   ~Boat() {}
25 };
26
27 class DualPurpose {
28 public:
29   DualPurpose() {}
30   char* purpose1() { return nullptr; }
31   char* purpose2() { return nullptr; }
32 };
33
34 class BoatCar: private Car, public Boat, protected DualPurpose {
35 public:
36   BoatCar() : Car(4), Boat(), mInWater(false), mColor("Blue") {}
37   virtual int passengers() const { return 4; }
38
39   static int numRegistered() { return sRegistered; }
40
41   bool mInWater;
42
43 protected:
44
45   void toggleInWater(bool inWater) { mInWater = inWater; }
46   char * mColor;
47   friend void init() {}
48
```

```
49    static int sRegistered;
50    static double calcSpeed(double distance, double time) {
51      return distance/time;
52    }
53
54  private:
55    int mMaxPassengers;
56    void travel() {}
57  };
58
59  class TourBoatCar : public BoatCar {};
60  class WildLifeTourBoatCar: public TourBoatCar {};
```

Vehicle is a root, so MaxInheritanceTree = 0. Wheeled and Boat derive directly from Vehicle, so each is 1. Car derives from Wheeled, so it's 2. DualPurpose has no base class, so it's also 0. BoatCar multiply inherits Car (depth 2), Boat (depth 1), and DualPurpose (depth 0); taking the maximum of those and adding 1 gives BoatCar = 3. That continues down the chain: TourBoatCar = 4 and WildLifeTourBoatCar = 5.

See Base Classes for the number of direct base classes this depth is computed from.

Also known as Chidamber & Kemerer - Depth of Inheritance Tree (DIT).



Targets By Language:

- **C#:** Class, Struct
- **C++:** Class, Struct, Union

- **Java:** Class, Interface
- **Pascal:** Class, Interface
- **Python:** Class
- **Web:** PHP Class, PHP Interface

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show inheritance metrics" option.

Max Modified Cyclomatic Complexity

API ID: MaxCyclomaticModified
Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, Web
Targets: Architectures, Classes, Files, Modules, Packages, Project

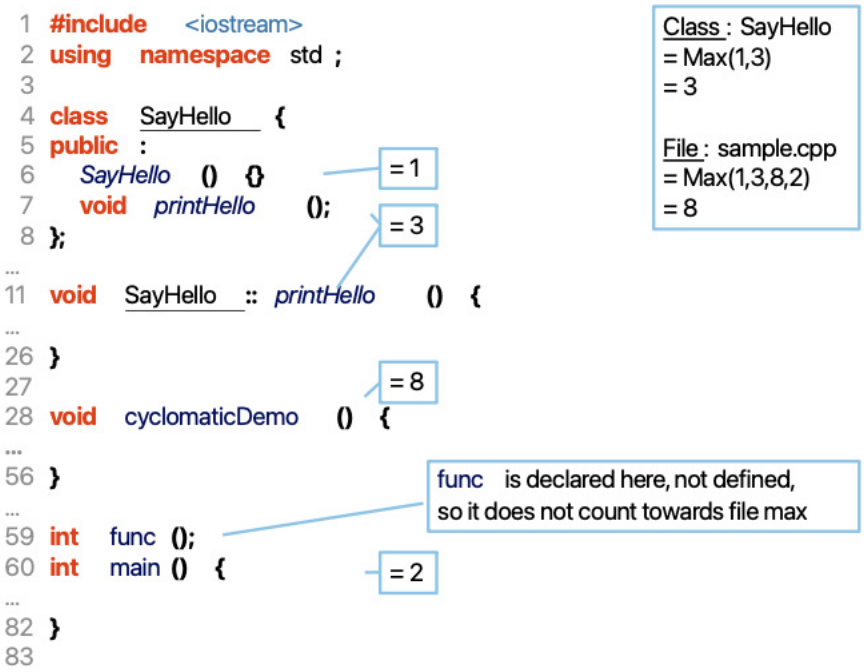
Maximum modified cyclomatic complexity of nested functions or methods.

For example:

```
1 class SayHello {
2 public:
3   SayHello() {}
4   void printHello();
5 };
6
7 void SayHello::printHello() {
8   switch (i) {
9     ...
10   }
11 }
12
13 void cyclomaticDemo() {
14   ...
15 }
16
17 int func();          // declared here, not defined - excluded from aggreg
18
19 int main() {
20   ...
21 }
```

SayHello(), printHello(), cyclomaticDemo(), and main() have Modified Cyclomatic Complexity of 1, 3, 8, and 2. func() is only declared, not defined, so it isn't counted. For the SayHello class, the maximum is 3. For the whole file (sample.cpp), the maximum is 8.

See Sum and Average Modified Cyclomatic Complexity for the other aggregations.



Targets By Language:

- **Ada:** Project, File, Package
- **Basic:** Project, File, Module, Class, Struct
- **C#:** Project, File, Class, Struct
- **C++:** Project, File, Class, Struct, Union
- **Fortran:** Project, File
- **Java:** Project, File, Class, Interface
- **Jovial:** Project, File
- **Pascal:** Project, File, Class, Interface, Compunit
- **Python:** Project, File, Class
- **Rust:** Project, File
- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated function metrics" option.
- The cyclomatic metric used is configured from "Project Configuration/Metrics/Complexity". This metric results from having "Count each decision in a multi-way-decision structure" off and having "Count logical conjuctions (OR) and logical and (AND)" off. Checking "Show all available cyclomatic complexity metric variants" will also enable the metric.

Max Nesting

API ID: MaxNesting
Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, Web
Targets: Architectures, Classes, Files, Functions, Modules, Packages, Project, Subprograms

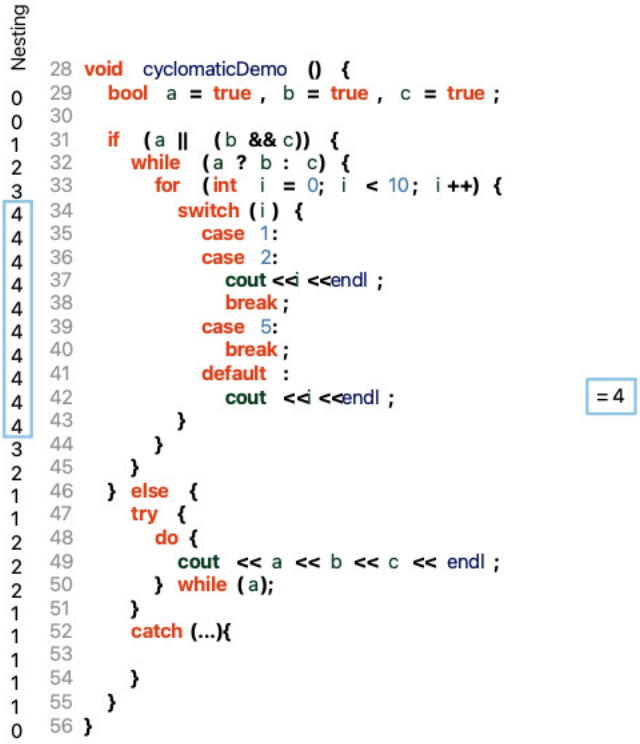
Maximum nesting level of control constructs (if, while, for, switch, etc.) in the function.

For example:

```
1 void cyclomaticDemo() {
2   bool a = true, b = true, c = true;
3
4   if (a || (b && c)) {
5     while (a ? b : c) {
6       for (int i = 0; i < 10; i++) {
7         switch (i) {
8           case 1:
9           case 2:
10             cout << i << endl;
11             break;
12          case 5:
13            break;
14          default:
15            cout << i << endl;
16        }
17      }
18    }
19  } else {
20    try {
21      do {
22        cout << a << b << c << endl;
23      } while (a);
24    }
25    catch (...) {
26    }
27  }
28 }
```

Nesting starts at 0 for the function's top-level statements and increases by 1 for each control construct nested inside another. The if is nesting level 1; the while inside it is level 2; the for inside that is level 3; and the switch inside that is level 4 — the deepest point in the function. The else branch's try/do only reaches nesting level 2. The maximum across the whole function is 4.

See Cyclomatic Complexity, which uses the same example.



Targets By Language:

- **Ada:** Project, File, Type, Entry, Function, Package, Procedure, Protected, Task
- **Basic:** Project, File, Method, Module, Class, Struct
- **C#:** Project, File, Class, Struct, Method
- **C++:** Project, File, Class, Struct, Union, Function
- **Fortran:** Project, File, Module, Block Data, Function, Program, Subroutine
- **Java:** Project, File, Class, Interface, Method
- **Jovial:** Project, File, Subroutine
- **Pascal:** Project, File, Class, Interface, Compunit, Function, Procedure
- **Python:** Project, File, Class, Function
- **Rust:** Project, File, Function
- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated function metrics" option.
- This metric can be disabled from "Project Configuration/Metrics/Complexity" with the "Maximum Nesting" option.

Max Strict Cyclomatic Complexity

API ID: MaxCyclomaticStrict
Languages: Ada, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, Web
Targets: Architectures, Classes, Files, Modules, Packages, Project

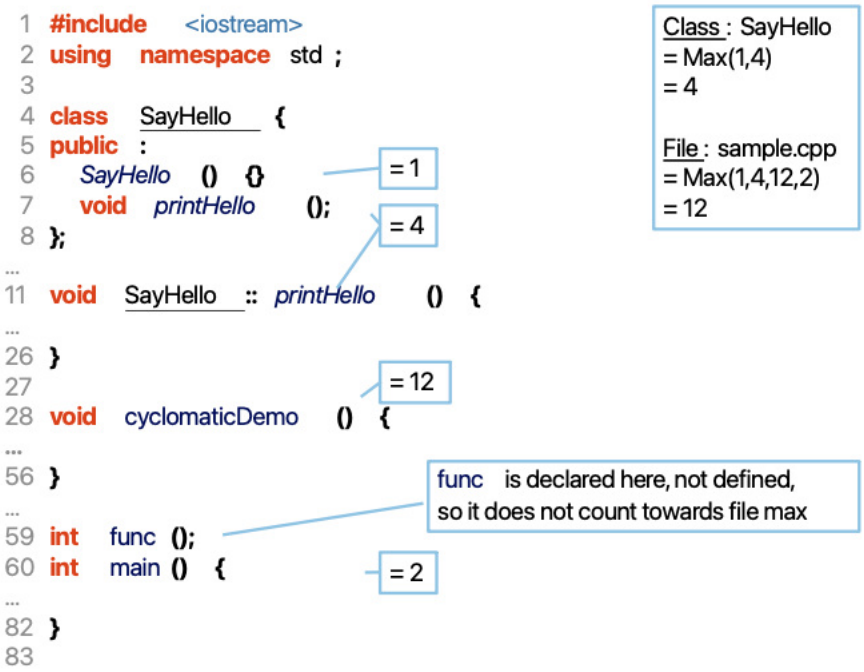
Maximum strict cyclomatic complexity of nested functions or methods.

For example:

```
1 class SayHello {
2 public:
3   SayHello() {}
4   void printHello();
5 };
6
7 void SayHello::printHello() {
8   switch (i) {
9     ...
10   }
11 }
12
13 void cyclomaticDemo() {
14   ...
15 }
16
17 int func();          // declared here, not defined - excluded from aggreg
18
19 int main() {
20   ...
21 }
```

SayHello(), printHello(), cyclomaticDemo(), and main() have Strict Cyclomatic Complexity of 1, 4, 12, and 2. func() is only declared, not defined, so it isn't counted. For the SayHello class, the maximum is 4. For the whole file (sample.cpp), the maximum is 12.

See Sum and Average Strict Cyclomatic Complexity for the other aggregations.



Targets By Language:

- **Ada:** Project, File, Package
- **C#:** Project, File, Class, Struct
- **C++:** Project, File, Class, Struct, Union
- **Fortran:** Project, File
- **Java:** Project, File, Class, Interface
- **Jovial:** Project, File
- **Pascal:** Project, File, Class, Interface, Compunit
- **Python:** Project, File, Class
- **Rust:** Project, File
- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated function metrics" option.
- The cyclomatic metric used is configured from "Project Configuration/Metrics/Complexity". This metric results from having "Count each decision in a multi-way-decision structure" on and having "Count logical conjuctions (OR) and logical and (AND)" on. Checking "Show all available cyclomatic complexity metric variants" will also enable the metric.

Max Strict Modified Cyclomatic Complexity

API ID: MaxCyclomaticStrictModified

Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, Web
Targets: Architectures, Classes, Files, Modules, Packages, Project

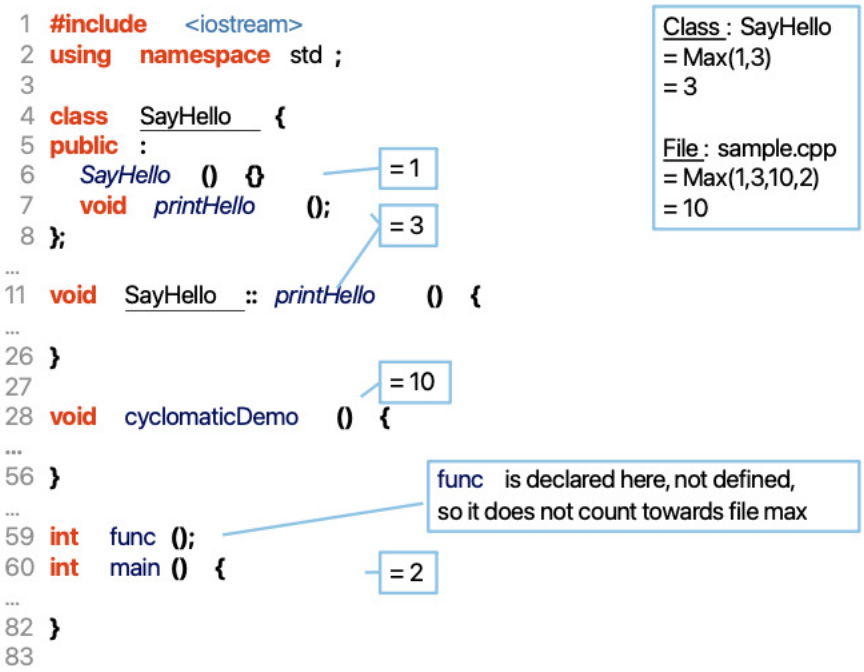
Maximum strict modified cyclomatic complexity of nested functions or methods.

For example:

```
1 class SayHello {
2 public:
3   SayHello() {}
4   void printHello();
5 };
6
7 void SayHello::printHello() {
8   switch (i) {
9     ...
10   }
11 }
12
13 void cyclomaticDemo() {
14   ...
15 }
16
17 int func();          // declared here, not defined - excluded from aggreg
18
19 int main() {
20   ...
21 }
```

SayHello(), printHello(), cyclomaticDemo(), and main() have Strict Modified Cyclomatic Complexity of 1, 3, 10, and 2. func() is only declared, not defined, so it isn't counted. For the SayHello class, the maximum is 3. For the whole file (sample.cpp), the maximum is 10.

See Sum and Average Strict Modified Cyclomatic Complexity for the other aggregations.



Targets By Language:

- **Ada:** Project, File, Package
- **Basic:** Project, File, Module, Class, Struct
- **C#:** Project, File, Class, Struct
- **C++:** Project, File, Class, Struct, Union
- **Fortran:** Project, File
- **Java:** Project, File, Class, Interface
- **Jovial:** Project, File
- **Pascal:** Project, File, Class, Interface, Compunit
- **Python:** Project, File, Class
- **Rust:** Project, File
- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated function metrics" option.
- The cyclomatic metric used is configured from "Project Configuration/Metrics/Complexity". This metric results from having "Count each decision in a multi-way-decision structure" off and having "Count logical conjuctions (OR) and logical and (AND)" on. Checking "Show all available cyclomatic complexity metric variants" will also enable the metric.

Max Strict Modified Essential Complexity

API ID: MaxEssentialStrictModified
Languages: Ada
Targets: Architectures, Files, Packages, Project

Maximum strict modified essential complexity of all nested functions or methods.

For example:

```
1 package Essential_Demo is
2    procedure Simple_Proc (X : in out Integer);
3    function Strict_Func (A, B : Boolean; N : Integer) return Integer;
4    function Case_Func (Day : Integer) return Integer;
5    function Mixed_Func (A, B : Boolean; Day : Integer) return Integer;
6 end Essential_Demo;
```

Simple_Proc, Strict_Func, Case_Func, and Mixed_Func have Strict Modified Essential Complexity of 1, 5, 2, and 7. For the Essential_Demo package, the maximum is 7.

See Sum and Average Strict Modified Essential Complexity for the other aggregations.

Targets By Language:

- **Ada:** Project, File, Package

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated function metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Complexity" with the "Essential" option.

Methods

API ID: CountDeclMethod
Also Known As: Weighted Methods per Class (WMC)
Languages: Basic, C#, C++, Java, Pascal, Python, Web
Targets: Architectures, Classes, Files, Modules, Project

Number of local (not inherited) methods.

Counts only methods actually written in the source — it combines both instance and static methods, across all access levels, but does not include compiler-generated implicit special members or anything inherited from a base class.

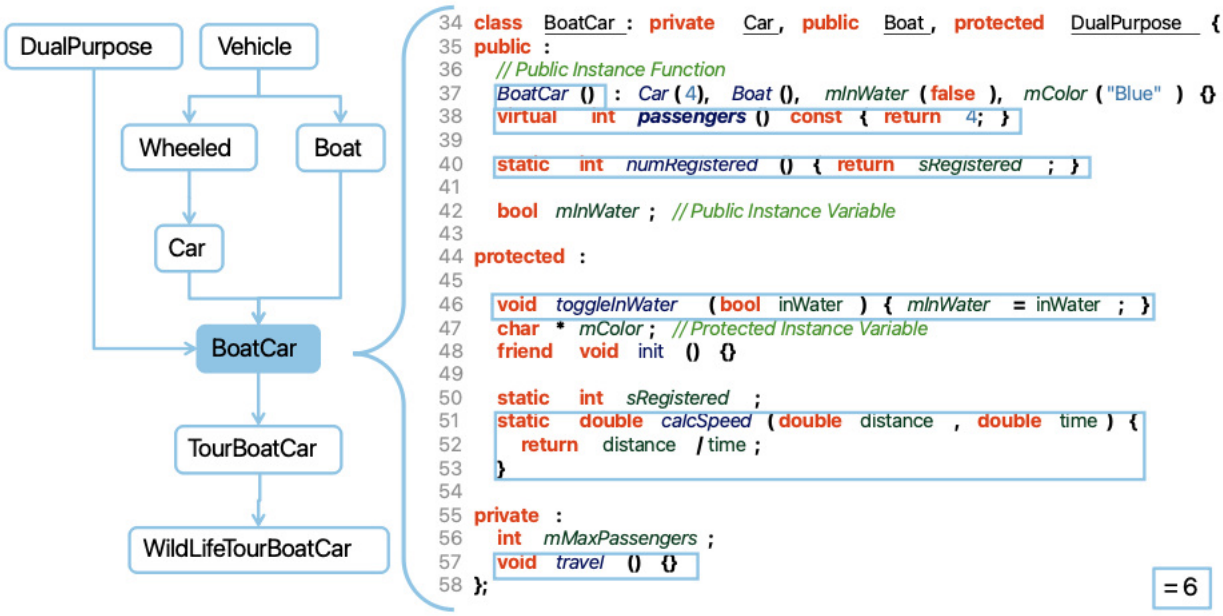
For example:

```
1 class BoatCar: private Car, public Boat, protected DualPurpose {
2 public:
3   // Public Instance Function
4   BoatCar() : Car(4), Boat(), mInWater(false), mColor("Blue") {}
5   virtual int passengers() const { return 4; }
6
7   static int numRegistered() { return sRegistered; }
8
9   bool mInWater; // Public Instance Variable
10
11 protected:
12
13   void toggleInWater(bool inWater) { mInWater = inWater; }
14   char * mColor; // Protected Instance Variable
15   friend void init() {}
16
17   static int sRegistered;
18   static double calcSpeed(double distance, double time) {
19     return distance/time;
20   }
21
22 private:
23   int mMaxPassengers;
24   void travel() {}
25 };
```

BoatCar explicitly writes 6 methods total: the constructor, passengers(), toggleInWater(bool), and travel() (instance), plus numRegistered() and calcSpeed(...) (static). So CountDeclMethod = 6. That splits cleanly by access level: 3 public (BoatCar(), passengers(), numRegistered()) + 2 protected (toggleInWater(bool), calcSpeed(...)) + 1 private (travel()) = 6, so CountDeclMethod = CountDeclMethodPublic + CountDeclMethodProtected + CountDeclMethodPrivate for this class.

See All Methods to also count methods inherited from base classes. For just the instance or class subset, see Instance Methods (which also counts the compiler-generated implicit special members) in contrast to Class Methods (the static ones). Other languages have access levels C++ doesn't: see Default Methods for Java's package-private methods, and Internal Methods/Protected Internal Methods for C#'s additional access levels.

Also known as Chidamber & Kemerer - Weighted Methods per Class (WMC).



Targets By Language:

- **Basic:** Project, Module, Class, Struct
- **C#:** Project, Class, Struct
- **C++:** Project, Class, Struct, Union
- **Java:** Project, File, Class, Interface
- **Pascal:** Project, Class, Interface
- **Python:** Project, Class
- **Web:** Project, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated class metrics" option. Enabling this option can be time consuming.
- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Min Essential Knots

API ID: MinEssentialKnots
Languages: Ada, C#, C++, Java, Rust
Targets: Functions, Packages, Subprograms

Minimum Knots after structured programming constructs have been removed.

Knots and Essential Complexity both start by removing well-structured (single-entry, single-exit) subgraphs from the control-flow graph. Min Essential Knots is just the Knots count of what's left after that reduction, instead of the raw (unreduced) graph.

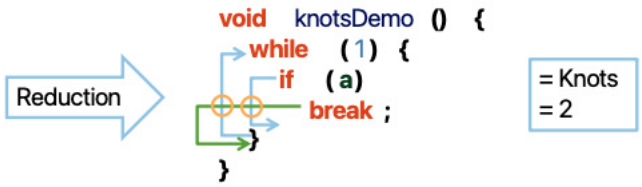
For example:

```
1 void knotsDemo() {
2   while (1) {
3     if (a)
4       break;
5     if (b || c) {
6       if (d || e) {
7       }
8       else {
9         if (i)
10           dosomething();
11        else if (j)
12          dosomething();
13        else if (k)
14          dosomething();
15        else {}
16      }
17    }
18  }
19 }
```

Reducing knotsDemo collapses the well-structured if (b || c) { ... } chain away, leaving just while (1) { if (a) break; }. That reduced graph has 2 knots (the loop's back edge crossing the break's forward jump), so Min Essential Knots is 2 — much lower than the 6 knots in the unreduced function.

See Max Essential Knots, which also accounts for crossings that reduction hides.

```
4  void  knotsDemo ()  {
5    while  (1) {
6      if  (a)
7        break ;
8      if  (b ||  c) {
9        if  (d ||  e) {
10
11       }
12       else  {
13         if  (i)
14           dosomething  ();
15         else  if  (j)
16           dosomething  ();
17         else  if  (k)
18           dosomething  ();
19         else  {}
20
21       }
22     }
23   }
24 }
```



Targets By Language:

- **Ada:** Type, Entry, Function, Package, Procedure, Protected, Task
- **C#:** Method
- **C++:** Function
- **Java:** Method
- **Rust:** Function

Configuration:

- This metric can be enabled from "Project Configuration/Metrics/Complexity" with the "Knots" option.

Modified Cyclomatic Complexity

API ID: CyclomaticModified
Also Known As: CC3
Languages: Ada, Assembly, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, VHDL, Web
Targets: Files, Functions, Modules, Packages, Subprograms

Modified McCabe Cyclomatic complexity.

The Cyclomatic Complexity except that each decision in a multi-decision structure (switch in C/Java, Case in Ada, computed Goto and arithmetic if in FORTRAN) statement is not counted and instead the entire multi-way decision structure counts

as 1.

For example:

```
1 void cyclomaticDemo() {
2   bool a = true, b = true, c = true;
3
4   if (a || (b && c)) {
5     while (a ? b : c) {
6       for (int i = 0; i < 10; i++) {
7         switch (i) {
8           case 1:
9           case 2:
10            cout << i << endl;
11            break;
12          case 5:
13            break;
14          default:
15            cout << i << endl;
16        }
17      }
18    }
19  } else {
20    try {
21      do {
22        cout << a << b << c << endl;
23      } while (a);
24    }
25    catch (...) {
26    }
27  }
28 }
```

if, while, the ?: conditional, for, do, and catch are always decision points, for 6 total. The switch statement adds one more regardless of how many cases it has, and the &&/|| in the if condition are not counted. That's 7 decision points, for a Modified Cyclomatic Complexity of 8.

To count each case in the switch statement as its own decision instead, use Cyclomatic Complexity. To also count && and || operators, use Strict Cyclomatic Complexity. To count those operators without changing how the switch is handled, use Strict Modified Cyclomatic Complexity.

See Sum Modified Cyclomatic Complexity, Max Modified Cyclomatic Complexity, and Average Modified Cyclomatic Complexity for aggregated values, such as for an architecture or the project.

Also known as McCabe - Modified Cyclomatic Complexity (CC3).



Targets By Language:

- **Ada:** Type, Entry, Function, Package, Procedure, Protected, Task
- **Basic:** Method
- **C#:** Method
- **C++:** Function
- **Fortran:** Module, Block Data, Function, Program, Subroutine
- **Java:** Method
- **Jovial:** Subroutine
- **Pascal:** Compunit, Function, Procedure
- **Python:** File, Function
- **Rust:** Function
- **VHDL:** Procedure, Function, Process
- **Web:** File

Configuration:

- The cyclomatic metric used is configured from "Project Configuration/ Metrics/Complexity". This metric results from having "Count each decision in a multi-way-decision structure" off and having "Count logical conjuctions (OR) and logical and (AND)" off. Checking "Show all available cyclomatic complexity metric variants" will also enable the metric.

Modules

API ID: CountDeclModule
Languages: Fortran, Jovial, Pascal

Targets: Architectures, Files, Functions, Modules, Project, Subprograms

Number of modules.

For example:

```
1 MODULE sample_mod
2   IMPLICIT NONE
3 CONTAINS
4   SUBROUTINE mod_internal_sub(x)
5     INTEGER, INTENT(IN) :: x
6     PRINT *, x
7   END SUBROUTINE mod_internal_sub
8 END MODULE sample_mod
```

sample.f90 declares one MODULE, sample_mod, so CountDeclModule = 1 for this file.

See Program Units for the combined count across modules, block data units, and subprograms.

Targets By Language:

- **Fortran:** Project, File, Module, Block Data, Function, Program, Subroutine
- **Jovial:** Project, File
- **Pascal:** Project, File

Configuration:

- For projects and architectures, this metric is enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated file metrics" option.

New Lines

API ID: CountLineNew
Languages: Ada, Assembly, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, VHDL, Web
Targets: Classes, Files, Functions, Modules, Packages, Subprograms

Number of new lines, compared to the comparison database.

Understand does a textual diff of the entity's contents and processes each changed region, meaning a run of adjacent added and deleted lines bounded by unchanged lines. For each changed region, if lines added is greater than lines deleted, then added - deleted is added to the new line count. So, insertions (deleted=0) are new

lines, and also added lines in a changed region that exceed the number of lines deleted.

Note that because this is specific to the entity's contents, things like reordering every function in a file while leaving function contents unchanged means that each individual function will be unchanged, even though the file changed. Also, an entity's lines are counted the same way they are for the Lines metric, so a C++ Class considers all the defined method bodies even when the definition is outside the class definition.

For example, diffing a file's current version against the comparison database's version:

```
--- sample.cpp (comparison database)
+++ sample.cpp (current)
@@ -1,15 +1,16 @@
 #include <iostream>

-// Utility functions
 int square(int x) {
-  return x * x;
+  return x * x * x;
 }

-int unused() {
-  return 0;
+// Compute a cube
+int cube(int x) {
+  return x * x * x;
 }

 int main() {
   std::cout << square(4) << std::endl;
+  std::cout << cube(3) << std::endl;
   return 0;
 }
```

The added std::cout << cube(3) << std::endl; is a new line from a simple insertion (no removed lines). There's also a new line from the cube function because the new region is 3 lines while the old region was only 2, so the difference counts as a new line. That gives CountLineNew = 2.

See Changed Lines for the lines a region has in common, Removed Lines for the reverse of this metric, and Percent Changed for a single score combining all three.

Outputs

API ID: CountOutput
Also Known As: FANOUT (Infomational fan-out)
Languages: C#, C++, Fortran, Java
Targets: Functions, Subprograms

Number of called subprograms plus global variables set.

The number of outputs that are SET. These can be parameters or global variables. So Functions calls + Parameters set/modify + Global Variables set/modify. A non-void return value adds one to the count. Recursive function calls, local variables that are not class static variables, and parameters that are pass by value are not included. Of the two general approachs to calculating FANOUT (informational versus structural) ours is the informational approach.

For example:

```
1 int somefunc();
2
3 int in = 1;
4 int out = 1;
5
6 int inOutFunc(int in1, int in2, int *inout1, int &inout2, int* out1, int &
7   out = in + in1 + in2 + *inout1 + inout2;
8
9   *inout1 = in1;
10   inout2 = in2;
11
12   *out1 = in1;
13   out2 = in2;
14
15   in1 = somefunc();
16   in2 = 2;
17
18   int randomint = 3;
19   in1 = randomint;
20
21   return 4;
22 }
23
24 void callingfunc() {
25   int a,b,c,d;
26   int myval = inOutFunc(1,2,&a,b,&c,d);
27 }
```

inOutFunc sets the global out and the parameters inout1, inout2, out1, and out2 (each modified through a pointer or reference). in1 and in2 are also reassigned later in the function, but since they're passed by value, those changes don't escape the

function, so they don't count. inOutFunc also calls somefunc() (a non-recursive call) and has a non-void (int) return type, which adds one more. That's 1 global + 4 parameters set + 1 function called + 1 for the non-void return, for an Outputs count of 7.

See Inputs for the related fan-in metric, and the Outputs Metric interactive report to list the references contributing to it for a given function.

Also known as FANOUT (Infomational fan-out).

Introduced by Henry, Sallie, and Dennis Kafura in "Software Structure Metrics Based on Information Flow." IEEE Transactions on Software Engineering, vol. SE-7, no. 5, 1981, pp. 510-518.

```
3  int  in  = 1;
4  int  out  = 1;
5
6  int  inOutFunc  (int  in1 , int  in2 , int  *inout1  , int  &inout2  , int * out1 , int  & out2 ) {
7     out  = in  + in1  + in2  + *inout1  + inout2  ;
8
9     *inout1  = in1 ;
10    inout2  = in2 ;
11
12    *out1  = in1 ;
13    out2  = in2 ;
14
15    in1  = somefunc  ();
16    in2  = 2;
17
18    int  randomint  = 3;
19    in1  = randomint  ;
20
21    return  4;
22 }
```

= functions called + parameters set + globals set + non -void return type
= 1 + 4 + 1 + 1
= 7

Entity	Counts?	Comment
in	No	Not set
out	Yes	Set line 7
in1	No	Pass by value does not count
in2	No	Pass by value does not count
inout1	Yes	Set line 9
inout2	Yes	Set line 10
out1	Yes	Set line 12
out2	Yes	Set line 13
randomint	No	Not a parameter, global, or class static variable
somefunc	Yes	Non-recursive function call, line 15

Targets By Language:

- **C#:** Method
- **C++:** Function
- **Fortran:** Function, Program, Subroutine
- **Java:** Method

Paths

API ID: CountPath
Also Known As: NPATH
Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, Web

Targets: Files, Functions, Modules, Packages, Subprograms

Number of unique paths through a body of code, not counting abnormal exits or gotos.

For example:

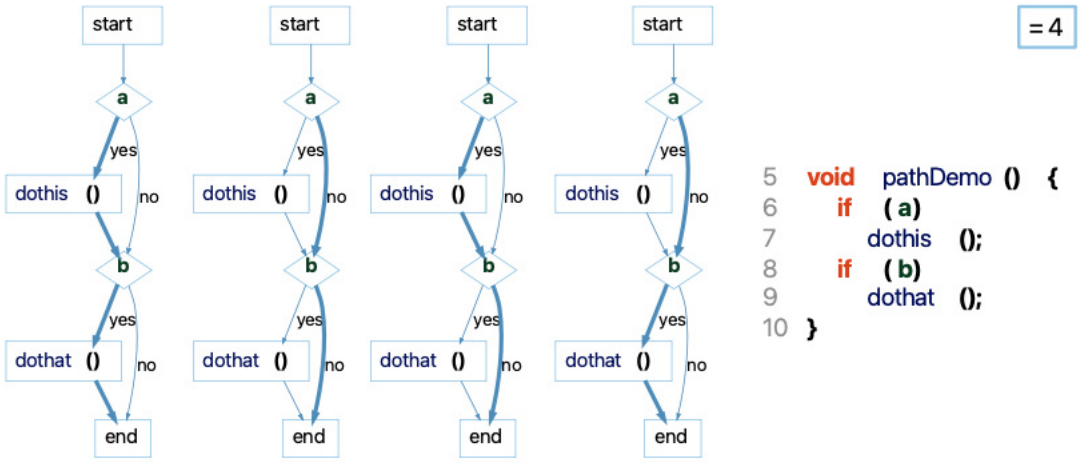
```
1 void pathDemo() {
2   if (a)
3     dothis();
4   if (b)
5     dothat();
6 }
```

The two if statements are independent, and each has 2 outcomes (taken or not), so there are 2 × 2 = 4 possible paths through the function: (a, b), (a, not b), (not a, b), and (not a, not b). That gives pathDemo a Paths count of 4.

Path counts are capped at 999,999,999 — functions with more paths than that all report the same maximum value. See Paths Log(x) for a version of this value that isn't capped the same way, and scales better for functions with many paths.

Also known as NPATH.

Introduced by Nejmeh, Brian A. "NPATH: A Measure of Execution Path Complexity and Its Applications." Communications of the ACM, vol. 31, no. 2, 1988, pp. 188-200.



Targets By Language:

- **Ada:** Type, Entry, Function, Package, Procedure, Protected, Task
- **Basic:** Method
- **C#:** Method
- **C++:** Function
- **Fortran:** Module, Block Data, Function, Program, Subroutine
- **Java:** Method
- **Jovial:** Subroutine
- **Pascal:** Compunit, Function, Procedure
- **Python:** File, Function
- **Rust:** File, Function
- **Web:** File

Configuration:

- This metric can be enabled from "Project Configuration/Metrics/Complexity" with the "Paths" option.

Paths Log(x)

API ID: CountPathLog
Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, Web
Targets: Files, Functions, Modules, Packages, Subprograms

The base 10 logarithm Log(x) of the number of unique paths though a body of code, not counting abnormal exits or gotos, rounded to an integer value.

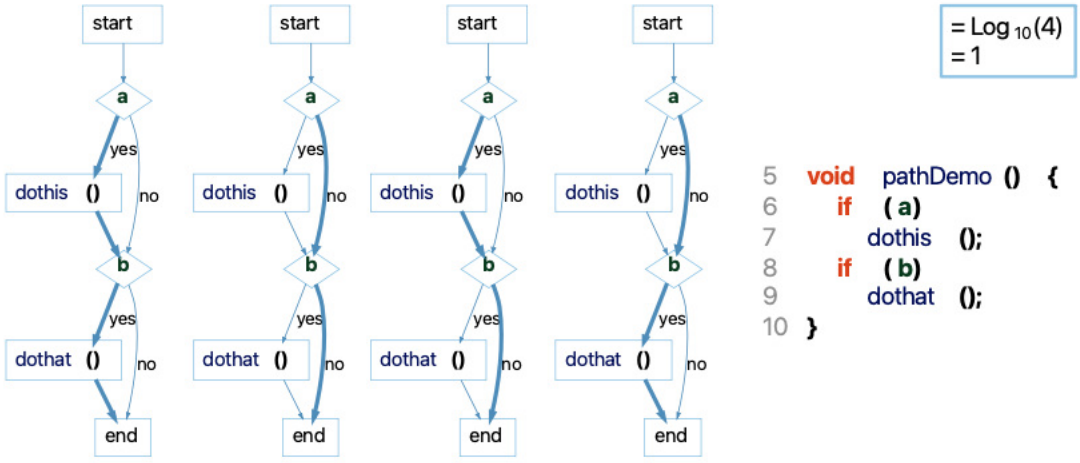
For example:

```
1 void pathDemo() {
2   if (a)
3     dothis();
4   if (b)
5     dothat();
6 }
```

pathDemo has 4 paths. Log10(4) is about 0.6, which rounds to 1, so its Paths Log(x) is 1.

Unlike Paths, this isn't capped at 999,999,999 paths, so it keeps giving a meaningful (if compressed) signal for functions whose true path count is far larger than that.

See Paths for the unrounded path count.



Targets By Language:

- **Ada:** Type, Entry, Function, Package, Procedure, Protected, Task
- **Basic:** Method
- **C#:** Method
- **C++:** Function
- **Fortran:** Module, Block Data, Function, Program, Subroutine
- **Java:** Method
- **Jovial:** Subroutine
- **Pascal:** Compunit, Function, Procedure
- **Python:** File, Function
- **Rust:** File, Function
- **Web:** File

Configuration:

- This metric can be enabled from "Project Configuration/Metrics/Complexity" with the "Paths" option.

Percent Changed

API ID: PercentChanged
Languages: Ada, Assembly, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, VHDL, Web
Targets: Classes, Files, Functions, Modules, Packages, Subprograms

Percentage of lines that differ from the comparison database.

(New + Removed + 2 × Changed) / (Current Lines + Comparison Lines) × 100

The formula weights changed lines double since they show up on both sides of the diff.

For example, diffing a file's current version against the comparison database's version:

```
--- sample.cpp (comparison database)
+++ sample.cpp (current)
@@ -1,15 +1,16 @@
 #include <iostream>

-// Utility functions
 int square(int x) {
-  return x * x;
+  return x * x * x;
 }

-int unused() {
-  return 0;
+// Compute a cube
+int cube(int x) {
+  return x * x * x;
 }

 int main() {
   std::cout << square(4) << std::endl;
+  std::cout << cube(3) << std::endl;
   return 0;
 }
```

gives New Lines = 2, Removed Lines = 1, and Changed Lines = 3, across 16 lines in the current version and 15 in the comparison database, so:

PercentChanged = (2 + 1 + 2 × 3) / (16 + 15) × 100 = 29.03%

Percent Lack Of Cohesion

API ID: PercentLackOfCohesion
Also Known As: LCOM2, Lack of Cohesion in Methods (LCOM/LOCM)
Languages: Ada, Basic, C#, C++, Java, Pascal

Targets: Classes, Packages

100% minus the average cohesion for class data members, or for package entities, 100% minus the average cohesion of the packaged classes.

For a class, this calculates what percentage of (data member, method) pairs actually interact: for every data member (instance or static) and every local, non-implicit method (instance or static), the pair counts if the method reads, writes, calls, or otherwise modifies that member. The percentage of pairs that interact, out of all possible pairs, is the class's cohesion; subtracting that from 100% gives the lack of cohesion. A lower percentage means higher cohesion between class data and methods.

For example:

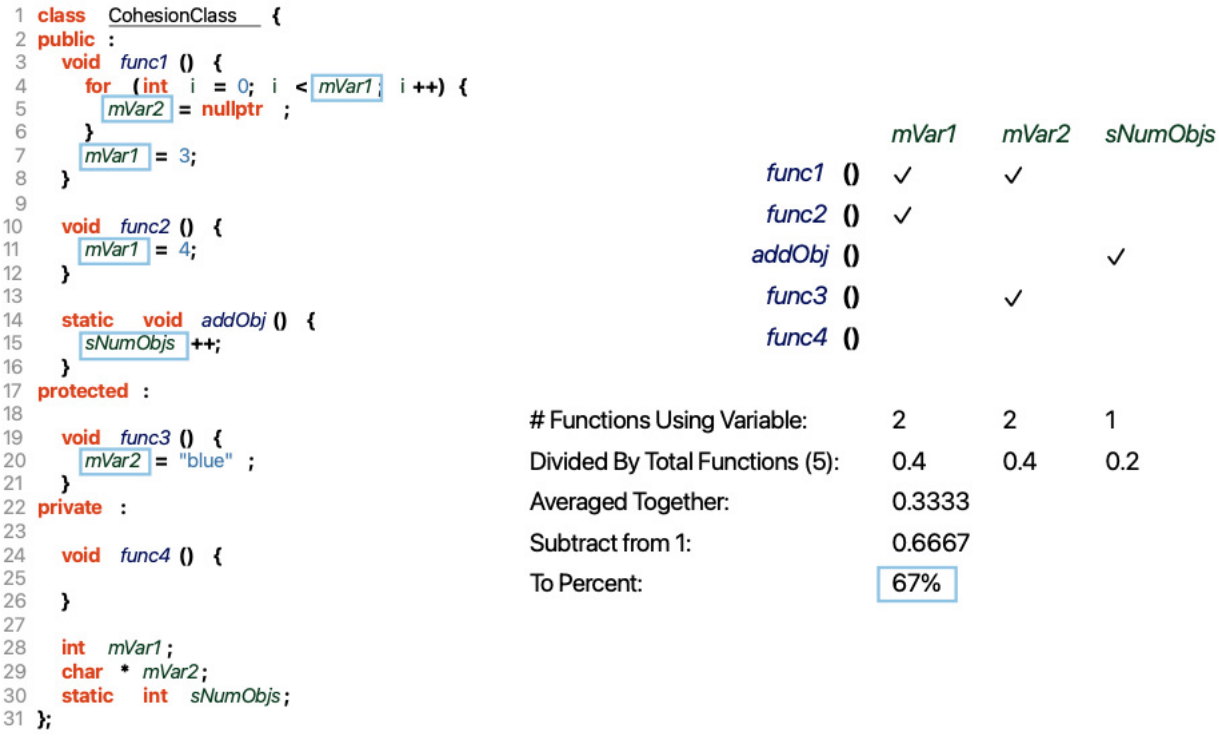
```
1 class CohesionClass {
2 public:
3   void func1() {
4     for (int i = 0; i < mVar1; i++) {
5       mVar2 = nullptr;
6     }
7     mVar1 = 3;
8   }
9
10   void func2() {
11     mVar1 = 4;
12   }
13
14   static void addObj() {
15     sNumObjs ++;
16   }
17 protected:
18
19   void func3() {
20     mVar2 = "blue";
21   }
22 private:
23
24   void func4() {
25
26   }
27
28   int mVar1;
29   char * mVar2;
30   static int sNumObjs;
31 };
32
33 int CohesionClass::sNumObjs = 3;
```

CohesionClass has 3 data members (mVar1, mVar2, sNumObjs) and 5 local methods

(func1, func2, func3, func4, addObj), for 3 × 5 = 15 possible pairs. mVar1 is touched by func1 (read in the for-loop condition, then written) and func2 (written): 2 methods. mVar2 is touched by func1 and func3: 2 methods. sNumObjs is touched by addObj: 1 method. func4 touches nothing. That's 2 + 2 + 1 = 5 interacting pairs out of 15 possible, or 33% cohesion, so PercentLackOfCohesion = 100% - 33% = 67.

See Percent Lack Of Cohesion Modified to avoid penalizing methods that touch data only through the class's own accessor methods.

Also known as Chidamber & Kemerer - Lack of Cohesion in Methods (LCOM/LOCM); LCOM2.



Targets By Language:

- **Ada:** Package
- **Basic:** Class, Struct
- **C#:** Class, Struct
- **C++:** Class, Struct, Union
- **Java:** Class, Interface
- **Pascal:** Class, Interface

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show coupling and cohesion metrics" option.

Percent Lack Of Cohesion Modified

API ID: PercentLackOfCohesionModified
Languages: Basic, C#, Java, Pascal
Targets: Classes

100% minus the average cohesion for class data members, modified for accessor methods.

Same as Percent Lack Of Cohesion, except a method that only touches an instance variable indirectly — by calling one of the class's own accessor methods rather than referencing the variable directly — is still credited with touching it, instead of being treated as disconnected from the class's data.

For example:

```
1 namespace MetricsDemo
2 {
3     public class CohesionDemo
4     {
5         private int a;
6         private int b;
7 
8         public int GetA() { return a; }
9         public void SetA(int val) { a = val; }
10 
11         public int GetB() { return b; }
12         public void SetB(int val) { b = val; }
13 
14         public void Combine()
15         {
16             SetA(GetA() + GetB());
17         }
18     }
19 }
```

GetA()/SetA() and GetB()/SetB() each touch one instance variable directly. Combine() never references a or b directly — it only calls the accessors. The base Percent Lack Of Cohesion doesn't follow that call to credit Combine() with touching a/b, giving a lack-of-cohesion score of 60. PercentLackOfCohesionModified follows the calls and recognizes Combine() as using both variables (through the accessors it calls), raising the measured cohesion and giving a lower score of 40.

Targets By Language:

- **Basic:** Class, Struct
- **C#:** Class, Struct
- **Java:** Class, Interface

- **Pascal:** Class, Interface

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show coupling and cohesion metrics" option.

Preprocessor Lines

API ID: CountLinePreprocessor
Languages: Ada, C#, C++
Targets: Architectures, Classes, Files, Functions, Packages, Project, Subprograms

Number of preprocessor lines.

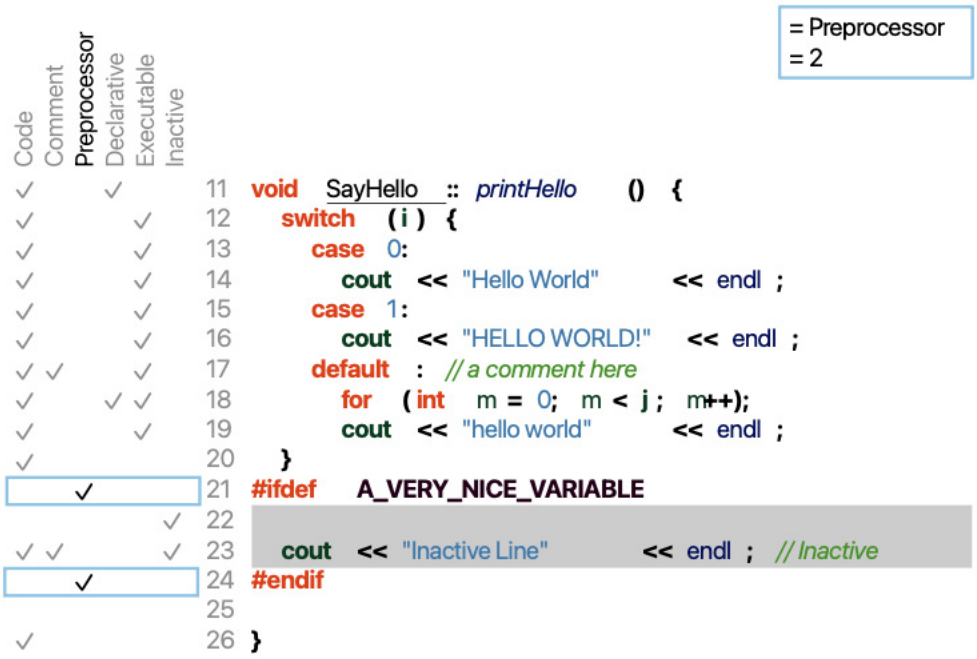
Lines containing a preprocessor directive, such as #include, #define, #if/#ifdef, and #endif. These lines are always considered active, regardless of whether the code they guard is active.

For example:

```
1 void SayHello::printHello() {
2   switch (i) {
3     case 0:
4       cout << "Hello World" << endl;
5     case 1:
6       cout << "HELLO WORLD!" << endl;
7     default: // a comment here
8       for (int m = 0; m < j; m++);
9       cout << "hello world" << endl;
10   }
11 #ifdef A_VERY_NICE_VARIABLE
12
13   cout << "Inactive Line" << endl; // Inactive
14 #endif
15
16 }
```

#ifdef A_VERY_NICE_VARIABLE and #endif (lines 21 and 24) are the only preprocessor lines, giving CountLinePreprocessor = 2.

See Inactive Lines for the lines on the false side of a directive like these.



Targets By Language:

- **Ada:** Project, File, Type, Entry, Function, Package, Procedure, Protected, Task
- **C#:** Project, File, Class, Method
- **C++:** Project, File, Class, Struct, Union, Function

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show line count metrics" option.
- For projects and architectures, this metric is enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show default summary metrics" option.

Private Instance Variables

API ID: CountDeclInstanceVariablePrivate
Languages: C#, C++, Web
Targets: Architectures, Classes, Project

Number of private instance variables.

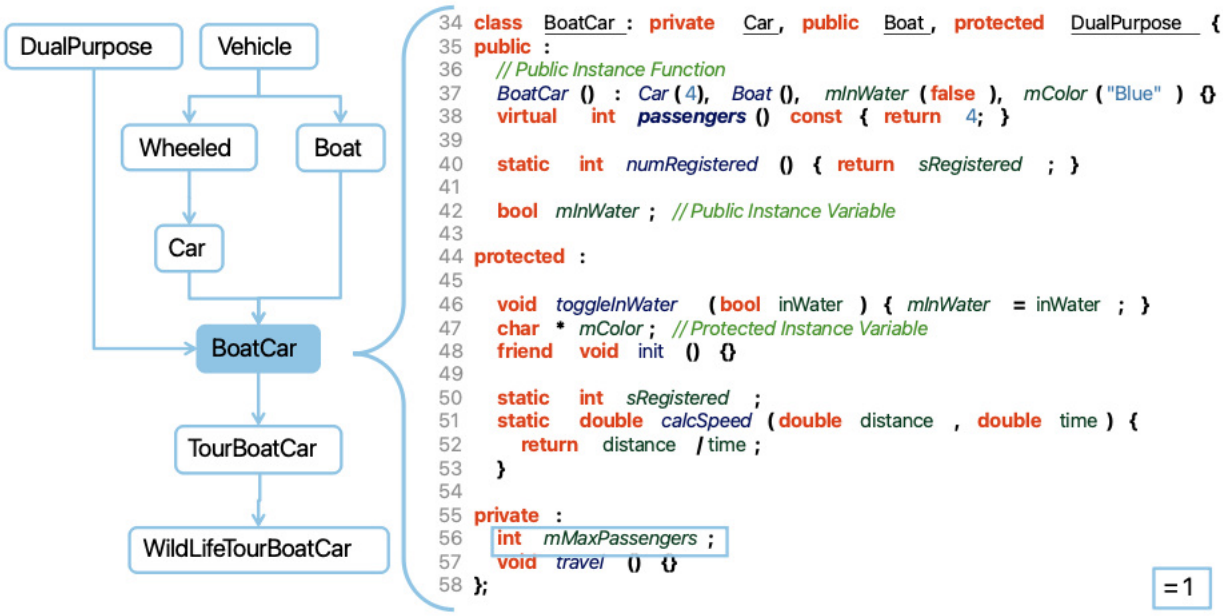
For example:

```
1 class BoatCar: private Car, public Boat, protected DualPurpose {
2 public:
3   // Public Instance Function
```

```
4   BoatCar() : Car(4), Boat(), mInWater(false), mColor("Blue") {}
5   virtual int passengers() const { return 4; }
6
7   static int numRegistered() { return sRegistered; }
8
9   bool mInWater; // Public Instance Variable
10
11 protected:
12
13   void toggleInWater(bool inWater) { mInWater = inWater; }
14   char * mColor; // Protected Instance Variable
15   friend void init() {}
16
17   static int sRegistered;
18   static double calcSpeed(double distance, double time) {
19     return distance/time;
20   }
21
22 private:
23   int mMaxPassengers;
24   void travel() {}
25 };
```

BoatCar declares one instance variable in its private: section, mMaxPassengers, so CountDeclInstanceVariablePrivate = 1.

See Instance Variables for the total across all access levels, and Protected Instance Variables / Public Instance Variables for the other two. C# also has additional access levels: see Internal Instance Variables/Protected Internal Instance Variables.



Targets By Language:

- **C#:** Project, Class, Struct
- **C++:** Project, Class, Struct, Union
- **Web:** Project, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated class metrics" option. Enabling this option can be time consuming.
- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Private Methods

API ID: CountDeclMethodPrivate
Also Known As: Number Private Methods (NPRM)
Languages: Basic, C#, C++, Java, Pascal, Web
Targets: Architectures, Classes, Files, Modules, Project

Number of local (not inherited) private methods.

For example:

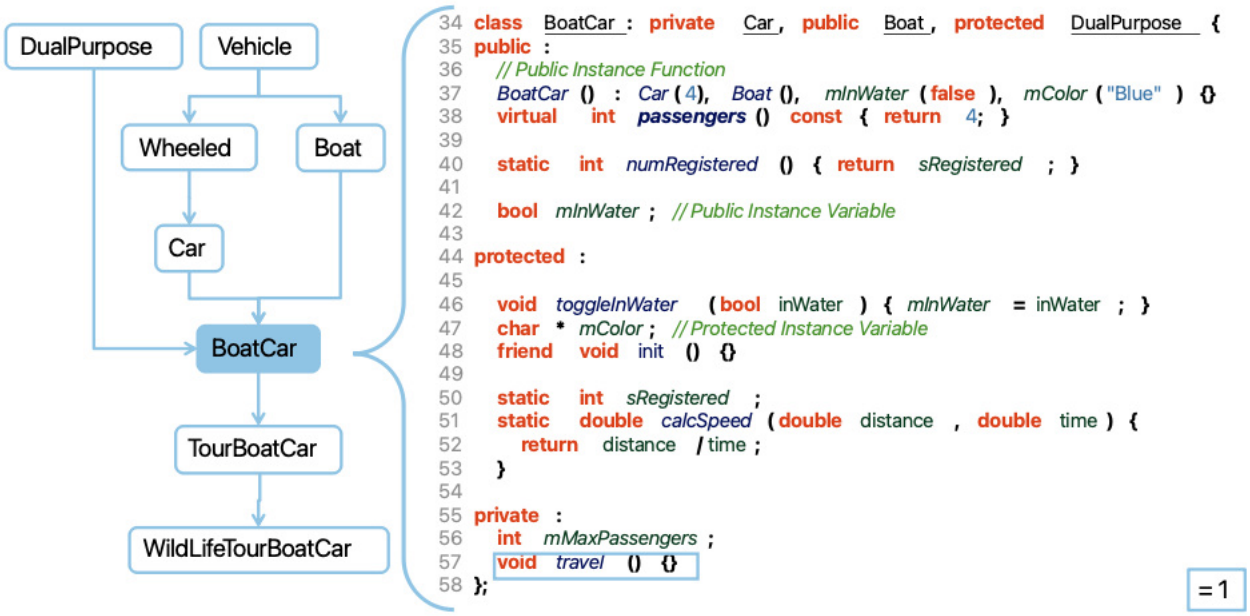
```
1 class BoatCar: private Car, public Boat, protected DualPurpose {
2 public:
```

```
3   // Public Instance Function
4   BoatCar() : Car(4), Boat(), mInWater(false), mColor("Blue") {}
5   virtual int passengers() const { return 4; }
6
7   static int numRegistered() { return sRegistered; }
8
9   bool mInWater; // Public Instance Variable
10
11 protected:
12
13   void toggleInWater(bool inWater) { mInWater = inWater; }
14   char * mColor; // Protected Instance Variable
15   friend void init() {}
16
17   static int sRegistered;
18   static double calcSpeed(double distance, double time) {
19     return distance/time;
20   }
21
22 private:
23   int mMaxPassengers;
24   void travel() {}
25 };
```

BoatCar's private: section declares one method, travel(), so CountDeclMethodPrivate = 1.

See Methods for the total, and Public Methods / Protected Methods for the other two access levels. Some languages have additional access levels: see Default Methods for Java's package-private methods, and Internal Methods/Protected Internal Methods for C#'s additional access levels, and Strict Private Methods for Object Pascal's narrower variant.

Also known as Number Private Methods (NPRM).



Targets By Language:

- **Basic:** Project, Module, Class, Struct
- **C#:** Project, Class, Struct
- **C++:** Project, Class, Struct, Union
- **Java:** Project, File, Class, Interface
- **Pascal:** Project, Class, Interface
- **Web:** Project, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated class metrics" option. Enabling this option can be time consuming.
- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Program Units

API ID: CountDeclProgUnit
Languages: Fortran
Targets: Architectures, Files, Project

Number of non-nested modules, block data units, and subprograms.

Only counts program units declared directly in the file — a subprogram nested

inside a module's own CONTAINS section doesn't add to this total, even though the module itself does.

For example:

```
1 MODULE sample_mod
2   IMPLICIT NONE
3 CONTAINS
4   SUBROUTINE mod_internal_sub(x)
5     INTEGER, INTENT(IN) :: x
6     PRINT *, x
7   END SUBROUTINE mod_internal_sub
8 END MODULE sample_mod
9
10
11 SUBROUTINE ext_sub(x)
12   INTEGER, INTENT(IN) :: x
13   PRINT *, x
14 END SUBROUTINE ext_sub
15
16
17 FUNCTION ext_func(x) RESULT(y)
18   INTEGER, INTENT(IN) :: x
19   INTEGER :: y
20   y = x + 1
21 END FUNCTION ext_func
22
23
24 PROGRAM sample
25   USE sample_mod
26   IMPLICIT NONE
27   INTEGER :: ext_func
28   CALL mod_internal_sub(1)
29   CALL ext_sub(2)
30   PRINT *, ext_func(3)
31 END PROGRAM sample
```

sample.f90 declares four non-nested program units: the module sample_mod, the external subroutine ext_sub, the external function ext_func, and the main program sample — so CountDeclProgUnit = 4. mod_internal_sub, nested inside sample_mod's CONTAINS section, doesn't add to this count, though it does count toward Subprograms (which, for Fortran, counts subroutines and functions wherever they're declared, nested or not, but not the enclosing module itself).

See Modules for just the module count.

Targets By Language:

- **Fortran:** Project, File

Configuration:

- For projects and architectures, this metric is enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated file metrics" option.

Properties

API ID: CountDeclProperty
Languages: C#, Pascal
Targets: Architectures, Classes, Project

Number of properties.

Counts every property regardless of how its accessors are implemented — both auto-implemented properties and ones with an explicit backing field and accessor bodies.

For example:

```
1 namespace MetricsDemo
2 {
3     public class PropertyDemo
4     {
5         public int X { get; set; }
6
7         private int y;
8         public int Y
9         {
10             get { return y; }
11             set { y = value; }
12         }
13     }
14 }
```

Both X (auto-implemented) and Y (explicit backing field y, with hand-written get/set bodies) count, giving CountDeclProperty = 2.

See Auto-Implemented Properties for just the auto-implemented subset. Properties aren't C#-only — Object Pascal's property Name: Type read Getter write Setter; syntax counts here too.

Targets By Language:

- **C#:** Project, Class, Struct
- **Pascal:** Project, Class, Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project

Configuration/Metrics/Projects and Architectures" with the "Show aggregated class metrics" option. Enabling this option can be time consuming.
- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Protected Instance Variables

API ID: CountDeclInstanceVariableProtected
Languages: C#, C++, Web
Targets: Architectures, Classes, Project

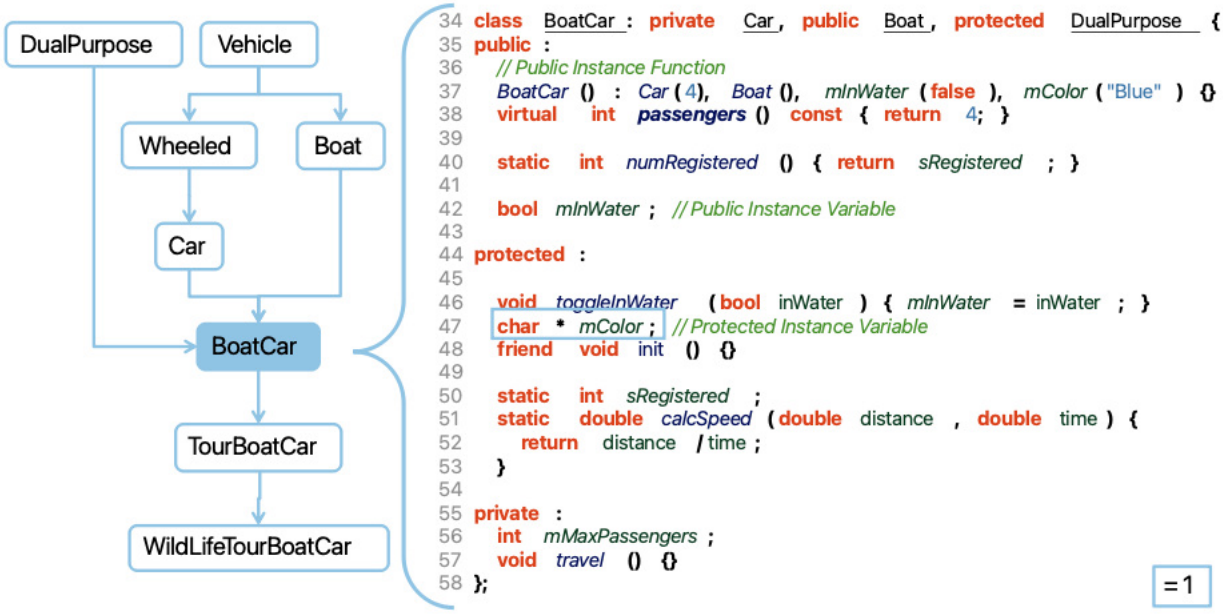
Number of protected instance variables.

For example:

```
1 class BoatCar: private Car, public Boat, protected DualPurpose {
2 public:
3   // Public Instance Function
4   BoatCar() : Car(4), Boat(), mInWater(false), mColor("Blue") {}
5   virtual int passengers() const { return 4; }
6
7   static int numRegistered() { return sRegistered; }
8
9   bool mInWater; // Public Instance Variable
10
11 protected:
12
13   void toggleInWater(bool inWater) { mInWater = inWater; }
14   char * mColor; // Protected Instance Variable
15   friend void init() {}
16
17   static int sRegistered;
18   static double calcSpeed(double distance, double time) {
19     return distance/time;
20   }
21
22 private:
23   int mMaxPassengers;
24   void travel() {}
25 };
```

BoatCar declares one instance variable in its protected: section, mColor, so CountDeclInstanceVariableProtected = 1.

See Instance Variables for the total across all access levels, and Private Instance Variables / Public Instance Variables for the other two. C# also has additional access levels: see Internal Instance Variables/Protected Internal Instance Variables.



Targets By Language:

- **C#:** Project, Class, Struct
- **C++:** Project, Class, Struct, Union
- **Web:** Project, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated class metrics" option. Enabling this option can be time consuming.
- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Protected Internal Instance Variables

API ID: CountDeclInstanceVariableProtectedInternal
Languages: C#
Targets: Architectures, Classes, Project

Number of protected internal instance variables.

A protected internal instance variable (C#'s protected internal access modifier) is accessible from anywhere in the same assembly, or from derived classes in other assemblies.

For example:

```
1 namespace MetricsDemo
2 {
3     public class AccessModifiers
4     {
5         internal int InternalField;
6         protected internal int ProtectedInternalField;
7 
8         internal void InternalMethod()
9         {
10             InternalField++;
11         }
12 
13         protected internal void ProtectedInternalMethod()
14         {
15             ProtectedInternalField++;
16         }
17     }
18 }
```

ProtectedInternalField is the only protected internal instance variable, giving CountDeclInstanceVariableProtectedInternal = 1. InternalField is plain internal — a narrower access level (same assembly only, no exception for derived classes elsewhere) — so it doesn't count here.

See Instance Variables for the total across all access levels, and Internal Instance Variables for the related access level. Public Instance Variables, Protected Instance Variables, and Private Instance Variables cover the access levels shared with C++ and Java.

Targets By Language:

- **C#:** Project, Class, Struct

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated class metrics" option. Enabling this option can be time consuming.
- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Protected Internal Methods

API ID: CountDeclMethodProtectedInternal
Languages: C#
Targets: Architectures, Classes, Project

Number of local protected internal methods.

A protected internal method (C#'s protected internal access modifier) is accessible from anywhere in the same assembly, or from derived classes in other assemblies.

For example:

```
1 namespace MetricsDemo
2 {
3     public class AccessModifiers
4     {
5         internal int InternalField;
6         protected internal int ProtectedInternalField;
7 
8         internal void InternalMethod()
9         {
10             InternalField++;
11         }
12 
13         protected internal void ProtectedInternalMethod()
14         {
15             ProtectedInternalField++;
16         }
17     }
18 }
```

ProtectedInternalMethod() is the only protected internal method, giving CountDeclMethodProtectedInternal = 1. InternalMethod() is plain internal — a narrower access level — so it doesn't count here.

See Methods for the total across all access levels, and Internal Methods for the related access level. Public Methods, Protected Methods, and Private Methods cover the access levels shared with C++ and Java, and Default Methods covers Java's package-private methods.

Targets By Language:

- **C#:** Project, Class, Struct

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated class metrics" option. Enabling this option can be time consuming.
- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Protected Methods

API ID: CountDeclMethodProtected
Languages: Basic, C#, C++, Java, Pascal, Web

Targets: Architectures, Classes, Files, Modules, Project

Number of local protected methods.

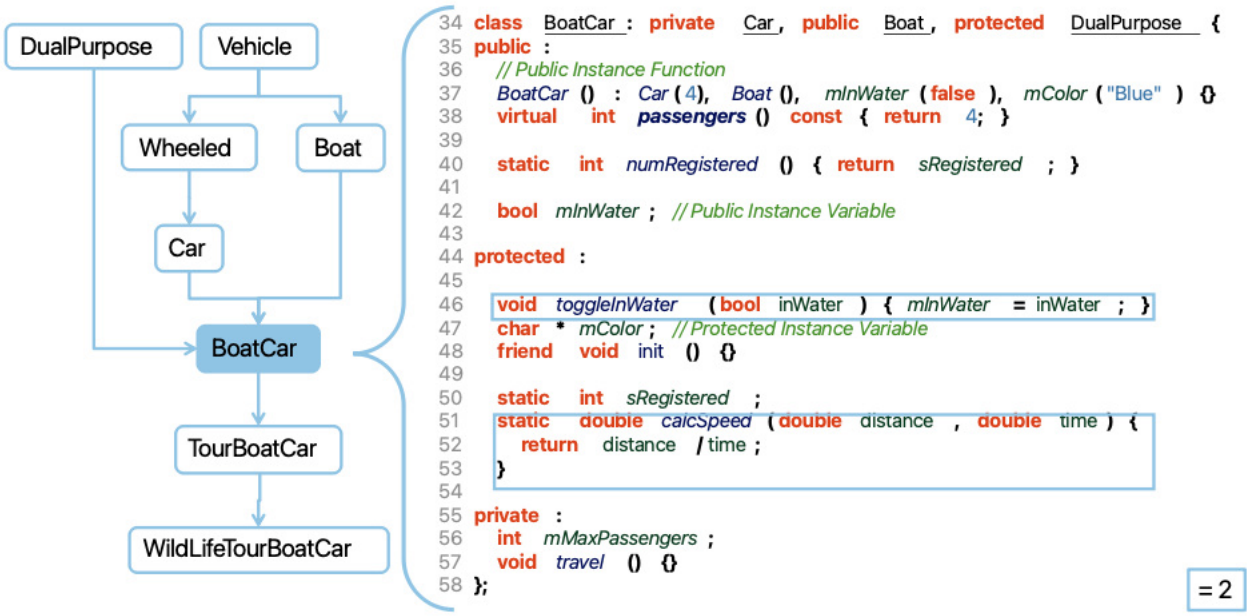
Counts protected methods regardless of whether they're instance or static methods.

For example:

```
1 class BoatCar: private Car, public Boat, protected DualPurpose {
2 public:
3   // Public Instance Function
4   BoatCar() : Car(4), Boat(), mInWater(false), mColor("Blue") {}
5   virtual int passengers() const { return 4; }
6
7   static int numRegistered() { return sRegistered; }
8
9   bool mInWater; // Public Instance Variable
10
11 protected:
12
13   void toggleInWater(bool inWater) { mInWater = inWater; }
14   char * mColor; // Protected Instance Variable
15   friend void init() {}
16
17   static int sRegistered;
18   static double calcSpeed(double distance, double time) {
19     return distance/time;
20   }
21
22 private:
23   int mMaxPassengers;
24   void travel() {}
25 };
```

BoatCar's protected: section declares two methods — toggleInWater(bool) (an instance method) and calcSpeed(double, double) (a static method) — so CountDeclMethodProtected = 2.

See Methods for the total, and Public Methods / Private Methods for the other two access levels. Some languages have additional access levels: see Default Methods for Java's package-private methods, and Internal Methods/Protected Internal Methods for C#'s additional access levels, and Strict Protected Methods for Object Pascal's narrower variant.



Targets By Language:

- **Basic:** Project, Module, Class, Struct
- **C#:** Project, Class, Struct
- **C++:** Project, Class, Struct, Union
- **Java:** Project, File, Class, Interface
- **Pascal:** Project, Class, Interface
- **Web:** Project, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated class metrics" option. Enabling this option can be time consuming.
- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Public Instance Variables

API ID: CountDeclInstanceVariablePublic
Languages: C#, C++, Web
Targets: Architectures, Classes, Project

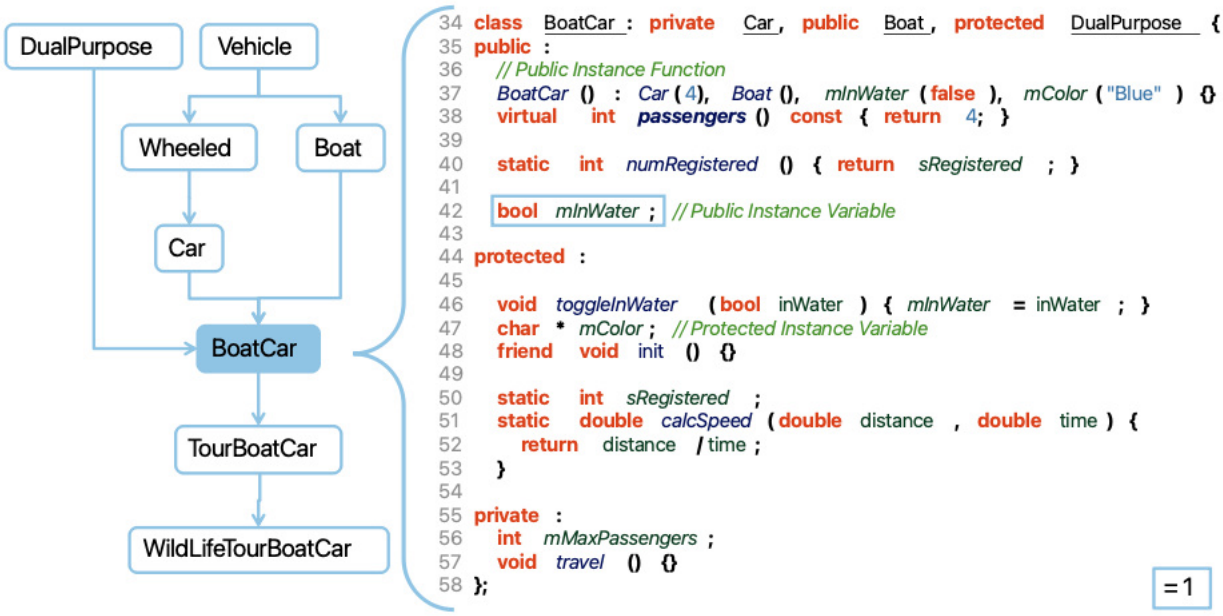
Number of public instance variables.

For example:

```
1 class BoatCar: private Car, public Boat, protected DualPurpose {
2 public:
3   // Public Instance Function
4   BoatCar() : Car(4), Boat(), mInWater(false), mColor("Blue") {}
5   virtual int passengers() const { return 4; }
6
7   static int numRegistered() { return sRegistered; }
8
9   bool mInWater; // Public Instance Variable
10
11 protected:
12
13   void toggleInWater(bool inWater) { mInWater = inWater; }
14   char * mColor; // Protected Instance Variable
15   friend void init() {}
16
17   static int sRegistered;
18   static double calcSpeed(double distance, double time) {
19     return distance/time;
20   }
21
22 private:
23   int mMaxPassengers;
24   void travel() {}
25 };
```

BoatCar declares one instance variable in its public: section, mInWater, so CountDeclInstanceVariablePublic = 1.

See Instance Variables for the total across all access levels, and Protected Instance Variables / Private Instance Variables for the other two. C# also has additional access levels: see Internal Instance Variables/Protected Internal Instance Variables.



Targets By Language:

- **C#:** Project, Class, Struct
- **C++:** Project, Class, Struct, Union
- **Web:** Project, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated class metrics" option. Enabling this option can be time consuming.
- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Public Methods

API ID: CountDeclMethodPublic
Also Known As: Number of Public Methods (PM, NPM), Public Instance Methods (PIM)
Languages: Basic, C#, C++, Java, Pascal, Web
Targets: Architectures, Classes, Files, Modules, Project

Number of local (not inherited) public methods.

For example:

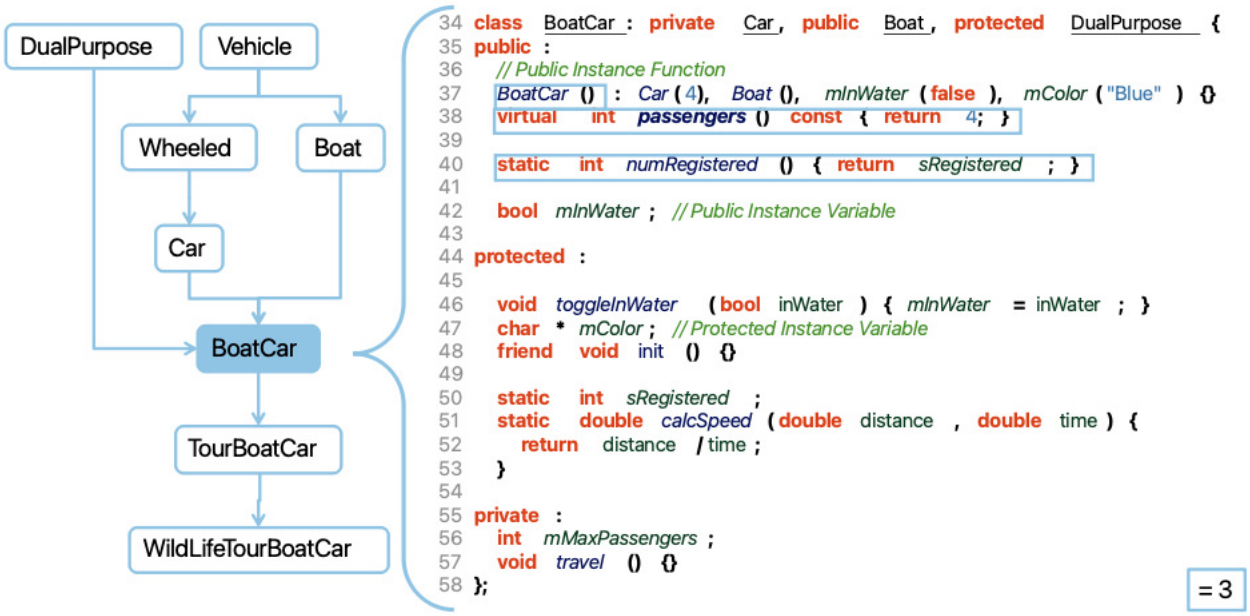
```
1 class BoatCar: private Car, public Boat, protected DualPurpose {
```

```
2 public:
3   // Public Instance Function
4   BoatCar() : Car(4), Boat(), mInWater(false), mColor("Blue") {}
5   virtual int passengers() const { return 4; }
6
7   static int numRegistered() { return sRegistered; }
8
9   bool mInWater; // Public Instance Variable
10
11 protected:
12
13   void toggleInWater(bool inWater) { mInWater = inWater; }
14   char * mColor; // Protected Instance Variable
15   friend void init() {}
16
17   static int sRegistered;
18   static double calcSpeed(double distance, double time) {
19     return distance/time;
20   }
21
22 private:
23   int mMaxPassengers;
24   void travel() {}
25 };
```

BoatCar's public: section declares three methods — the BoatCar() constructor, passengers() const, and the static numRegistered() — so CountDeclMethodPublic = 3.

See Methods for the total, and Protected Methods / Private Methods for the other two access levels. Some languages have additional access levels: see Default Methods for Java's package-private methods, and Internal Methods/Protected Internal Methods for C#'s additional access levels, and Published Methods for Object Pascal's public-with-RTTI level.

Also known as Lorenz & Kidd - Public Instance Methods (PIM); Number of Public Methods (PM, NPM).



Targets By Language:

- **Basic:** Project, Module, Class, Struct
- **C#:** Project, Class, Struct
- **C++:** Project, Class, Struct, Union
- **Java:** Project, File, Class, Interface
- **Pascal:** Project, Class, Interface
- **Web:** Project, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated class metrics" option. Enabling this option can be time consuming.
- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Published Methods

API ID: CountDeclMethodPublished
Languages: Pascal
Targets: Architectures, Classes, Project

Number of local published methods.

Object Pascal's published visibility is accessible from anywhere, just like public, but

published members also get run-time type information (RTTI) generated for them, which the IDE's Object Inspector and form-streaming system use to discover and persist them at design time. Properties are published most often, but methods — such as event handlers — can be published too.

For example:

```
unit sample;

interface

type
  TShape = class(TPersistent)
  strict private
    FArea: Double;
    procedure CalcArea;
  strict protected
    procedure Resize;
  published
    procedure NotifyChanged;
  public
    property Area: Double read FArea write FArea;
  end;

implementation

procedure TShape.CalcArea;
begin
  FArea := 0;
end;

procedure TShape.Resize;
begin
end;

procedure TShape.NotifyChanged;
begin
end;

end.
```

NotifyChanged is declared published, giving CountDeclMethodPublished = 1.

See Public Methods for the plain (non-published) counterpart, and Strict Private

Methods/Strict Protected Methods for Object Pascal's other visibility levels.

Targets By Language:

- **Pascal:** Project, Class, Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated class metrics" option. Enabling this option can be time consuming.
- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Removed Lines

API ID: CountLineRemoved
Languages: Ada, Assembly, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, VHDL, Web
Targets: Classes, Files, Functions, Modules, Packages, Subprograms

Number of removed lines, compared to the comparison database.

Understand does a textual diff of the entity's contents and processes each changed region, meaning a run of adjacent added and deleted lines bounded by unchanged lines. For each changed region, if lines deleted is greater than lines added, then deleted - added is added to the removed line count. So, deletions (added=0) are removed lines, and also deleted lines in a changed region that exceed the number of lines added.

Note that because this is specific to the entity's contents, things like reordering every function in a file while leaving function contents unchanged means that each individual function will be unchanged, even though the file changed. Also, an entity's lines are counted the same way they are for the Lines metric, so a C++ Class considers all the defined method bodies even when the definition is outside the class definition.

For example, diffing a file's current version against the comparison database's version:

```
--- sample.cpp (comparison database)
+++ sample.cpp (current)
@@ -1,15 +1,16 @@
 #include <iostream>
```

```
-// Utility functions
 int square(int x) {
-  return x * x;
+  return x * x * x;
 }

-int unused() {
-  return 0;
+// Compute a cube
+int cube(int x) {
+  return x * x * x;
 }

 int main() {
   std::cout << square(4) << std::endl;
+  std::cout << cube(3) << std::endl;
   return 0;
 }
```

The // Utility functions comment is a removed line from a simple deletion (no added lines). All other changed regions have at least as many lines added as there were lines deleted, so CountLineRemoved = 1.

See Changed Lines for the lines a region has in common, New Lines for the reverse of this metric, and Percent Changed for a single score combining all three.

Semicolons

API ID: CountSemicolon
Languages: Ada, C#, C++, Java
Targets: Architectures, Classes, Files, Functions, Packages, Project, Subprograms

Number of semicolons.

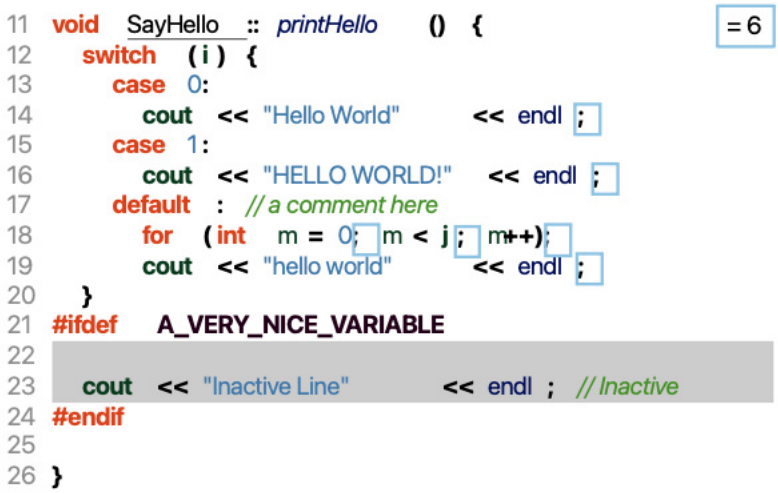
This is a simpler token count than Statements: since each clause of a for loop's header counts as its own statement, it's actually a single semicolon that can end multiple statements at once — cyclomaticDemo()'s bool a = true, b = true, c = true; declares three variables (three declarative statements) with only one semicolon. Other lines, like a block or an if without a trailing statement, have none.

For example:

```
1 void SayHello::printHello() {
2   switch (i) {
3     case 0:
```

```
4       cout << "Hello World" << endl;
5     case 1:
6       cout << "HELLO WORLD!" << endl;
7     default: // a comment here
8       for (int m = 0; m < j; m++);
9       cout << "hello world" << endl;
10   }
11 #ifdef A_VERY_NICE_VARIABLE
12
13   cout << "Inactive Line" << endl; // Inactive
14 #endif
15
16 }
```

printHello() has 6 semicolons: one ending each of the three cout statements (lines 14, 16, 19), and three inside the for loop's header on line 18 (m = 0;, m < j;, and the empty body's own ;). The semicolon on the inactive cout line inside the #ifdef block (line 23) doesn't count.



Targets By Language:

- **Ada:** Project, File, Type, Entry, Function, Package, Procedure, Protected, Task
- **C#:** Project, File, Class, Method
- **C++:** Project, File, Function
- **Java:** Project, File, Class, Interface, Method

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show statement count metrics" option.

Statements

API ID: CountStmt
Languages: Ada, Assembly, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, VHDL, Web
Targets: Architectures, Classes, Files, Functions, Modules, Packages, Project, Subprograms

Number of statements.

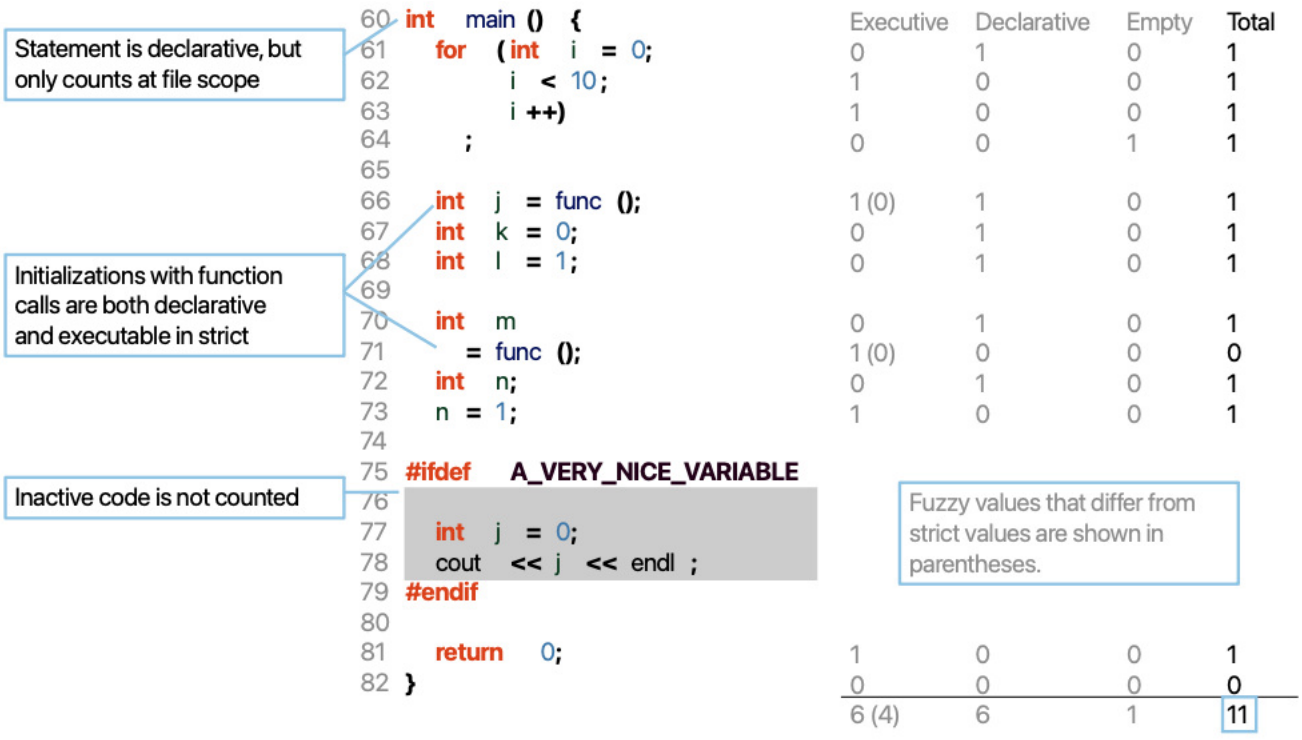
A single physical line can contain more than one statement, and each clause of a for loop's header (initialization, condition, and increment) counts as its own statement, in addition to its body.

For example:

```
1 int main() {
2   for (int i = 0;
3        i < 10;
4        i++)
5     ;
6
7   int j = func();
8   int k = 0;
9   int l = 1;
10
11   int m
12     = func();
13   int n;
14   n = 1;
15
16 #ifdef A_VERY_NICE_VARIABLE
17
18   int j = 0;
19   cout << j << endl;
20 #endif
21
22   return 0;
23 }
```

main() contains 11 statements: the for loop's three header clauses (int i = 0, i < 10, i++) plus its empty body (;), the six variable declarations and assignments (int j = func();, int k = 0;, int l = 1;, int m = func();, int n;, n = 1;), and the final return 0;, for CountStmt = 11. The two statements inside the inactive #ifdef block don't count.

See Declarative Statements, Executable Statements, and Empty Statements for a breakdown by statement type, and Semicolons for the related count of ; tokens.



Targets By Language:

- **Ada:** Project, File, Type, Entry, Function, Package, Procedure, Protected, Task
- **Basic:** Project, File, Method, Module, Class
- **C#:** Project, File, Class, Method
- **C++:** Project, File, Class, Struct, Union, Function
- **Fortran:** Project, File, Block Data, Function, Program, Subroutine
- **Java:** Project, File, Class, Interface, Method
- **Jovial:** Project, File, Module, Subroutine
- **Pascal:** Project, File, Class, Interface, Compunit, Function, Procedure
- **Python:** Project, File, Class, Function
- **Rust:** Project, File, Function
- **VHDL:** Project, File
- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- This metric can be disabled from "Project Configuration/Metrics/Lines and Statements" with the "Show statement count metrics" option.

Strict Cyclomatic Complexity

API ID: CyclomaticStrict

Also Known As: CC2
Languages: Ada, Assembly, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, VHDL, Web
Targets: Files, Functions, Modules, Packages, Subprograms

Strict McCabe Cyclomatic Complexity.

The Cyclomatic Complexity with logical conjunction and logical and in conditional expressions also adding 1 to the complexity for each of their occurrences. i.e. The statement 'if (a && b || c)' would have a cyclomatic complexity of 1 but a strict cyclomatic complexity of 3

For example:

```
1 void cyclomaticDemo() {
2   bool a = true, b = true, c = true;
3
4   if (a || (b && c)) {
5     while (a ? b : c) {
6       for (int i = 0; i < 10; i++) {
7         switch (i) {
8           case 1:
9           case 2:
10            cout << i << endl;
11            break;
12          case 5:
13            break;
14          default:
15            cout << i << endl;
16        }
17      }
18    }
19  } else {
20    try {
21      do {
22        cout << a << b << c << endl;
23      } while (a);
24    }
25    catch (...) {
26    }
27  }
28 }
```

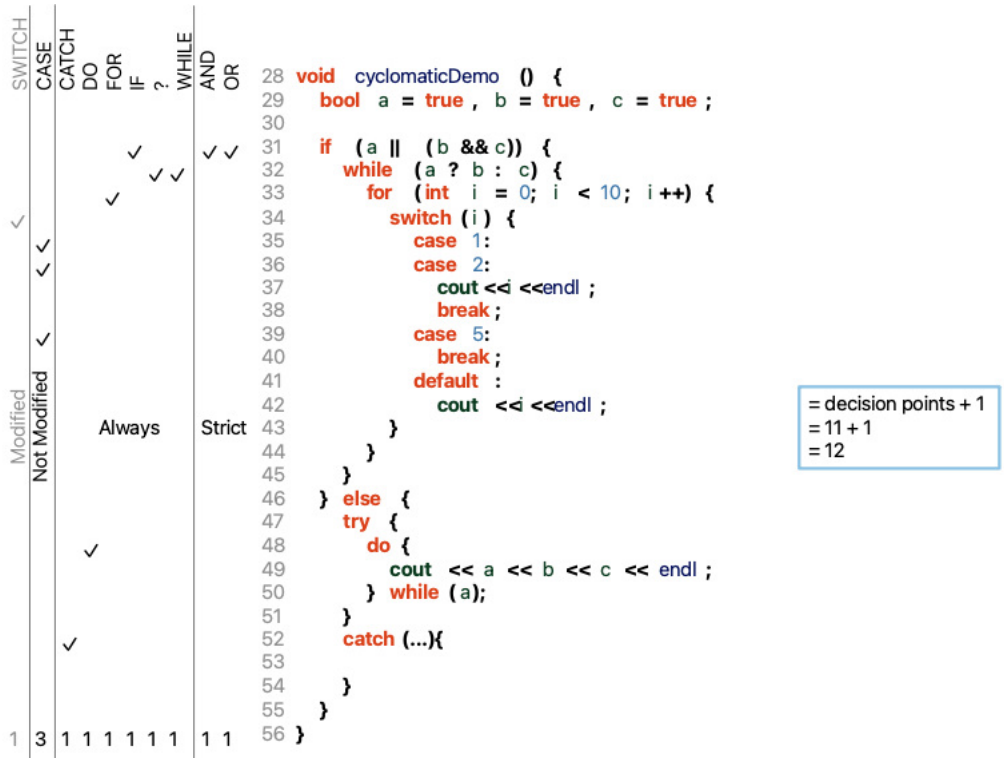
if, while, the ?: conditional, for, do, and catch are always decision points, for 6 total. Each case in the switch statement adds another (3 total), and the && and || in the if condition each add one as well (2 total). That's 11 decision points, for a Strict Cyclomatic Complexity of 12.

To stop counting && and || operators, use Cyclomatic Complexity. To also count the

entire switch statement as a single decision, use Strict Modified Cyclomatic Complexity. To change only how the switch is handled, use Modified Cyclomatic Complexity.

See Sum Strict Cyclomatic Complexity, Max Strict Cyclomatic Complexity, and Average Strict Cyclomatic Complexity for aggregated values, such as for an architecture or the project.

Also known as McCabe - Strict Cyclomatic Complexity (CC2).



Targets By Language:

- **Ada:** Type, Entry, Function, Package, Procedure, Protected, Task
- **Basic:** Method
- **C#:** Method
- **C++:** Function
- **Fortran:** Module, Block Data, Function, Program, Subroutine
- **Java:** Method
- **Jovial:** Subroutine
- **Pascal:** Compunit, Function, Procedure
- **Python:** File, Function
- **Rust:** Function
- **VHDL:** Procedure, Function, Process
- **Web:** File

Configuration:

- The cyclomatic metric used is configured from "Project Configuration/ Metrics/Complexity". This metric results from having "Count each decision in a multi-way-decision structure" on and having "Count logical conjuctions (OR) and logical and (AND)" on. Checking "Show all available cyclomatic complexity metric variants" will also enable the metric.

Strict Modified Cyclomatic Complexity

API ID: CyclomaticStrictModified
Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, Web
Targets: Files, Functions, Modules, Packages, Subprograms

Cyclomatic Complexity with the following conditions.

(1) Logical operators (AND, OR) in conditional expressions add one (1) to the complexity for each occurrence.
(2) In multi-decision structures (switch in C/Java, Case in Ada, computed Goto, and arithmetic if in FORTRAN) the decision points are not counted, instead, the entire multi-way decision structure counts as 1

For example:

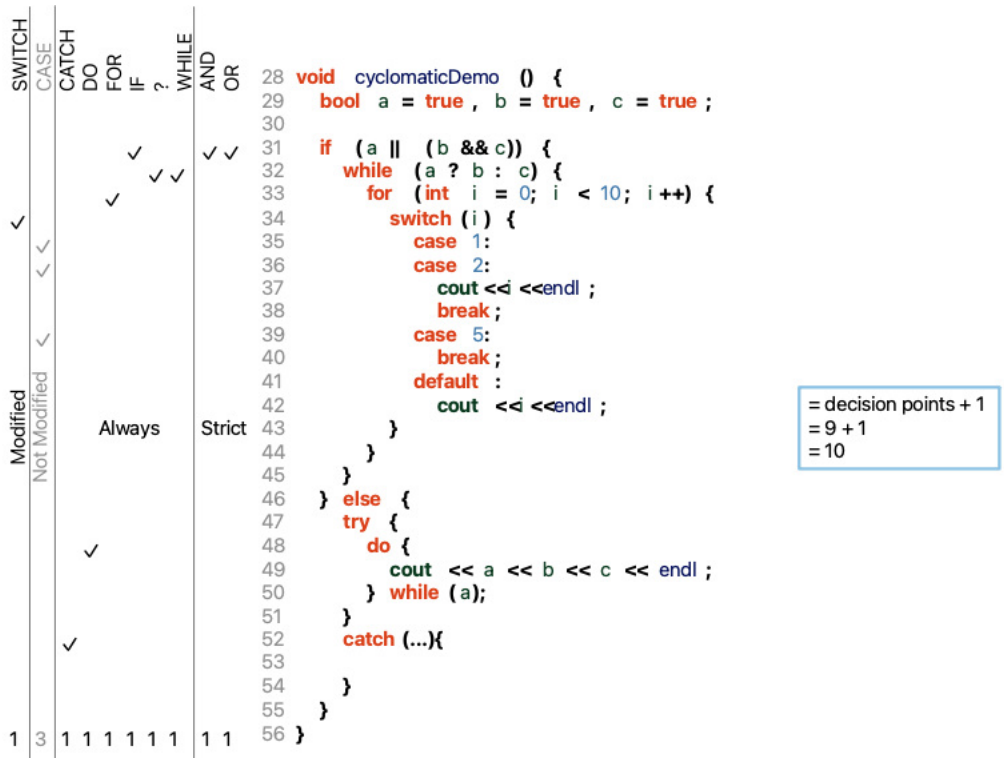
```
1 void cyclomaticDemo() {
2   bool a = true, b = true, c = true;
3
4   if (a || (b && c)) {
5     while (a ? b : c) {
6       for (int i = 0; i < 10; i++) {
7         switch (i) {
8           case 1:
9           case 2:
10             cout << i << endl;
11             break;
12           case 5:
13             break;
14           default:
15             cout << i << endl;
16         }
17       }
18     }
19   } else {
20     try {
21       do {
22         cout << a << b << c << endl;
23       } while (a);
24     }
25     catch (...) {
26     }
27   }
```

```
28 }
```

if, while, the ?: conditional, for, do, and catch are always decision points, for 6 total. The switch statement adds one more regardless of how many cases it has, and the && and || in the if condition each add one as well (2 total). That's 9 decision points, for a Strict Modified Cyclomatic Complexity of 10.

To count each case in the switch statement as its own decision instead, use Strict Cyclomatic Complexity. To also stop counting && and || operators, use Cyclomatic Complexity. To change only how the switch is handled, use Modified Cyclomatic Complexity.

See Sum Strict Modified Cyclomatic Complexity, Max Strict Modified Cyclomatic Complexity, and Average Strict Modified Cyclomatic Complexity for aggregated values, such as for an architecture or the project.



Targets By Language:

- **Ada:** Type, Entry, Function, Package, Procedure, Protected, Task
- **Basic:** Method
- **C#:** Method
- **C++:** Function
- **Fortran:** Module, Block Data, Function, Program, Subroutine
- **Java:** Method
- **Jovial:** Subroutine
- **Pascal:** Compunit, Function, Procedure
- **Python:** File, Function

- **Rust:** Function
- **Web:** File

Configuration:

- The cyclomatic metric used is configured from "Project Configuration/ Metrics/Complexity". This metric results from having "Count each decision in a multi-way-decision structure" off and having "Count logical conjuctions (OR) and logical and (AND)" on. Checking "Show all available cyclomatic complexity metric variants" will also enable the metric.

Strict Modified Essential Complexity

API ID: EssentialStrictModified
Languages: Ada
Targets: Packages, Subprograms

Cyclomatic Complexity after graph reduction, except that a decision using a short circuit operator (and then/or else) is treated as unstructured — so it isn't reduced away — while a case statement that survives reduction still only adds one, no matter how many structured paths run through it.

This is Ada-only: and then/or else are Ada's short-circuit boolean operators. Like Essential Complexity, it's computed after reducing well-structured subgraphs out of the control-flow graph, but a decision using and then/or else isn't reduced away, and a case that survives reduction still counts as a single decision point (as in Modified Cyclomatic Complexity) regardless of how many when branches it has.

For example:

```
1 procedure Simple_Proc (X : in out Integer) is
2 begin
3   if X < 0 then
4     X := 0;
5   end if;
6
7   for I in 1 .. X loop
8     X := X - 1;
9   end loop;
10 end Simple_Proc;
11
12 function Case_Func (Day : Integer) return Integer is
13   Result : Integer := 0;
14 begin
15   case Day is
16     when 1 =>
17       return 10;
18     when 2 =>
```

```
19        Result := 20;
20      when 3 =>
21        Result := 30;
22      when others =>
23        Result := 0;
24    end case;
25    return Result;
26 end Case_Func;
```

Simple_Proc has no short-circuit operators or case statement, so it reduces the same way for both metrics: Essential Complexity and Strict Modified Essential Complexity are both 1.

Case_Func's case statement doesn't reduce away, since one branch (when 1) returns early while the others fall through to the final return. Counted normally, that's 3 decision points (when 1, when 2, when 3; when others doesn't count), for an Essential Complexity of 4. Modified counting collapses the whole case to a single decision point instead, for a Strict Modified Essential Complexity of 2.

A decision using and then/or else doesn't reduce away either, and an elsif clause is judged on its own condition. For example, a chain of if A and then B / elsif B or else N > 10 has an Essential Complexity of 1 (fully reduced under plain rules), but under strict modified counting neither clause reduces, so the if, the elsif, and both short circuit operators all remain, for a Strict Modified Essential Complexity of 5.

See Sum, Max, and Average Strict Modified Essential Complexity for aggregated values across a package.

Targets By Language:

- **Ada:** Type, Entry, Function, Package, Procedure, Protected, Task

Configuration:

- This metric can be enabled from "Project Configuration/Metrics/Complexity" with the "Essential" option.

Strict Private Methods

API ID: CountDeclMethodStrictPrivate
Languages: Pascal
Targets: Architectures, Classes, Project

Number of local strict private methods.

Object Pascal's strict private is narrower than plain private: a plain private member is visible to any code in the same unit (file), while a strict private member is only

visible within the declaring class itself.

For example:

```
unit sample;

interface

type
  TShape = class(TPersistent)
  strict private
    FArea: Double;
    procedure CalcArea;
  strict protected
    procedure Resize;
  published
    procedure NotifyChanged;
  public
    property Area: Double read FArea write FArea;
  end;

implementation

procedure TShape.CalcArea;
begin
  FArea := 0;
end;

procedure TShape.Resize;
begin
end;

procedure TShape.NotifyChanged;
begin
end;

end.
```

CalcArea is declared strict private, giving CountDeclMethodStrictPrivate = 1.

See Private Methods for the plain (non-strict) access level, Strict Protected Methods for the other strict access level, and Published Methods for Object Pascal's public-with-RTTI level.

Targets By Language:

- **Pascal:** Project, Class, Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated class metrics" option. Enabling this option can be time consuming.
- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Strict Protected Methods

API ID: CountDeclMethodStrictProtected
Languages: Pascal
Targets: Architectures, Classes, Project

Number of local strict protected methods.

Object Pascal's strict protected is narrower than plain protected: a plain protected member is visible to descendant classes as well as to any code in the same unit (file), while a strict protected member is visible only to the declaring class and its descendants, not to unrelated code elsewhere in the unit.

For example:

```
unit sample;

interface

type
  TShape = class(TPersistent)
  strict private
    FArea: Double;
    procedure CalcArea;
  strict protected
    procedure Resize;
  published
    procedure NotifyChanged;
  public
    property Area: Double read FArea write FArea;
  end;

implementation
```

```
procedure TShape.CalcArea;
begin
  FArea := 0;
end;

procedure TShape.Resize;
begin
end;

procedure TShape.NotifyChanged;
begin
end;

end.
```

Resize is declared strict protected, giving CountDeclMethodStrictProtected = 1.

See Protected Methods for the plain (non-strict) access level, Strict Private Methods for the other strict access level, and Published Methods for Object Pascal's public-with-RTTI level.

Targets By Language:

- **Pascal:** Project, Class, Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated class metrics" option. Enabling this option can be time consuming.
- This metric can be disabled from "Project Configuration/Metrics/Object Orientated" with the "Show method and variable count metrics" option.

Subprograms

API ID: CountDeclSubprogram
Languages: Ada, Basic, Fortran, Jovial, Pascal
Targets: Architectures, Files, Functions, Modules, Packages, Project, Subprograms

Number of subprograms.

Counts local procedures, functions, and similar callable routines — the term "subprogram" comes from Ada, Fortran, and Pascal, which use it directly, but the metric also spans Basic and Jovial.

For example:

```
1 package Essential_Demo is
2
3    -- Well-structured if/loop only; reduces fully during essential-complex
4    -- graph reduction.
5    procedure Simple_Proc (X : in out Integer);
6
7    -- If/elsif using "and then"/"or else"; demonstrates the "Strict" short
8    -- circuit counting.
9    function Strict_Func (A, B : Boolean; N : Integer) return Integer;
10
11    -- Case statement where one branch returns early and the others fall
12    -- through, so the case stays in the reduced graph; demonstrates the
13    -- "Modified" case counting.
14    function Case_Func (Day : Integer) return Integer;
15
16    -- Combines a loop, short circuit operators, and a case statement with
17    -- mix of early return and fall-through branches; the most complex
18    -- subprogram in the package.
19    function Mixed_Func (A, B : Boolean; Day : Integer) return Integer;
20
21 end Essential_Demo;
```

Essential_Demo declares 4 subprograms — Simple_Proc, Strict_Func, Case_Func, and Mixed_Func — so CountDeclSubprogram = 4 for this package.

See Functions for the equivalent concept in C, C#, Java, Python, Rust, and Web, which use that term instead.

Targets By Language:

- **Ada:** Project, File, Package
- **Basic:** Project, File
- **Fortran:** Project, File, Module, Block Data, Function, Program, Subroutine
- **Jovial:** Project, File, Module, Subroutine
- **Pascal:** Project, File, Compunit, Function, Procedure

Configuration:

- For projects and architectures, this metric is enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated file metrics" option.
- For projects and architectures, this metric is enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show default summary metrics" option.

Sum Cyclomatic Complexity

API ID: SumCyclomatic
Also Known As: Weighted Methods per Class (WMC)
Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, Web
Targets: Architectures, Classes, Files, Functions, Modules, Packages, Project, Subprograms

Sum of cyclomatic complexity of all nested functions or methods.

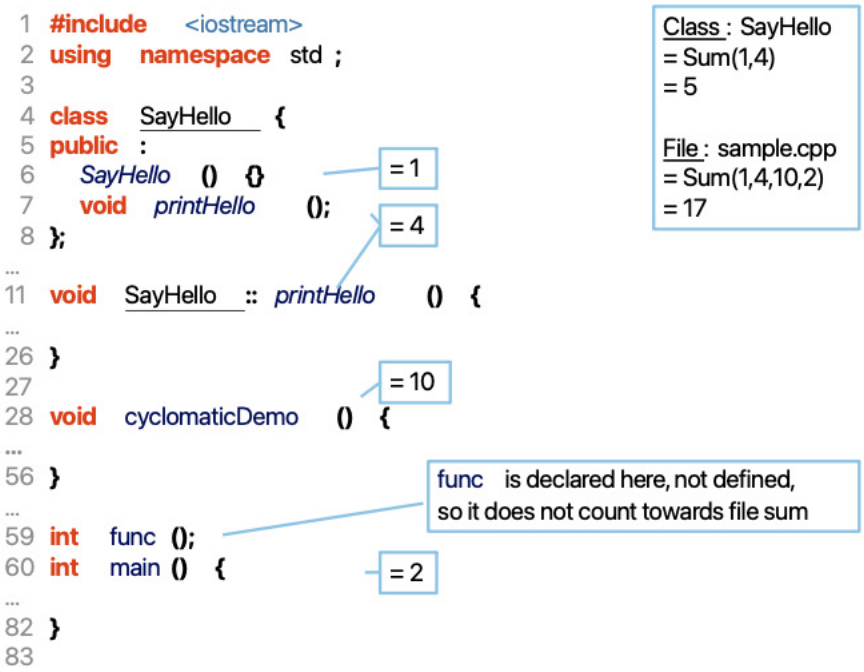
For example:

```
1 class SayHello {
2 public:
3   SayHello() {}
4   void printHello();
5 };
6
7 void SayHello::printHello() {
8   switch (i) {
9     ...
10   }
11 }
12
13 void cyclomaticDemo() {
14   ...
15 }
16
17 int func();          // declared here, not defined - excluded from aggreg
18
19 int main() {
20   ...
21 }
```

SayHello(), printHello(), cyclomaticDemo(), and main() have Cyclomatic Complexity of 1, 4, 10, and 2. func() is only declared, not defined, so it isn't counted. For the SayHello class, the sum is 1 + 4 = 5. For the whole file (sample.cpp), the sum is 1 + 4 + 10 + 2 = 17.

See Max and Average Cyclomatic Complexity for the other aggregations.

Also known as Chidamber & Kemerer - Weighted Methods per Class (WMC).



Targets By Language:

- **Ada:** Project, File, Type, Entry, Function, Package, Procedure, Protected, Task
- **Basic:** Project, File, Method, Module, Class, Struct
- **C#:** Project, File, Class, Struct, Method
- **C++:** Project, File, Class, Struct, Union
- **Fortran:** Project, File, Module, Function, Program, Subroutine
- **Java:** Project, File, Class, Interface, Method
- **Jovial:** Project, File
- **Pascal:** Project, File, Class, Interface, Compunit, Function, Procedure
- **Python:** Project, File, Class
- **Rust:** Project, File
- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated function metrics" option.
- The cyclomatic metric used is configured from "Project Configuration/Metrics/Complexity". This metric results from having "Count each decision in a multi-way-decision structure" on and having "Count logical conjuctions (OR) and logical and (AND)" off. Checking "Show all available cyclomatic complexity metric variants" will also enable the metric.

Sum Essential Complexity

API ID: SumEssential
Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, Web
Targets: Architectures, Classes, Files, Functions, Modules, Packages, Project, Subprograms

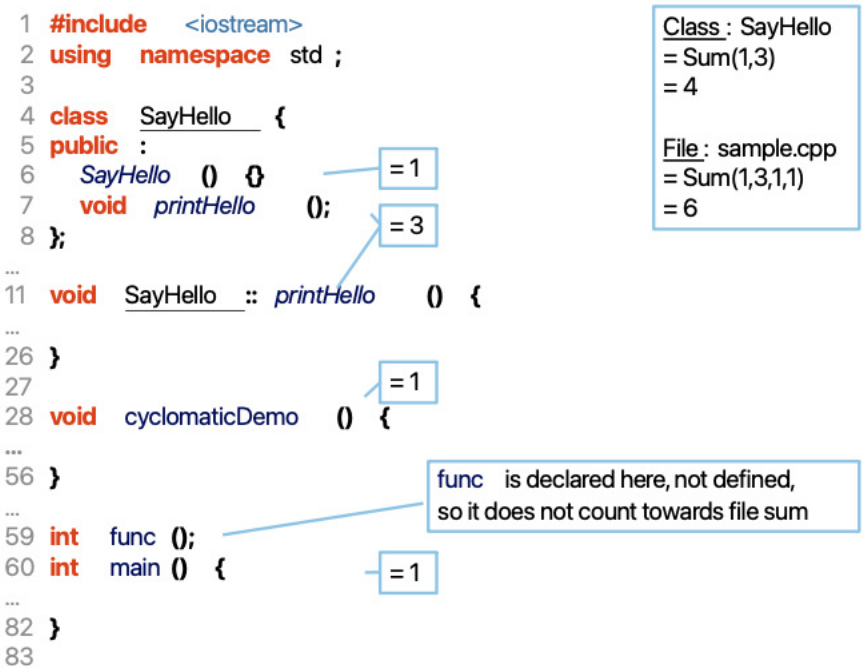
Sum of essential complexity of all nested functions or methods.

For example:

```
1 class SayHello {
2 public:
3   SayHello() {}
4   void printHello();
5 };
6
7 void SayHello::printHello() {
8   switch (i) {
9     ...
10   }
11 }
12
13 void cyclomaticDemo() {
14   ...
15 }
16
17 int func();          // declared here, not defined - excluded from aggreg
18
19 int main() {
20   ...
21 }
```

SayHello(), printHello(), cyclomaticDemo(), and main() have Essential Complexity of 1, 3, 1, and 1. func() is only declared, not defined, so it isn't counted. For the SayHello class, the sum is 1 + 3 = 4. For the whole file (sample.cpp), the sum is 1 + 3 + 1 + 1 = 6.

See Max and Average Essential Complexity for the other aggregations.



Targets By Language:

- **Ada:** Project, File, Type, Entry, Function, Package, Procedure, Protected, Task
- **Basic:** Project, File, Method, Module, Class, Struct
- **C#:** Project, File, Class, Struct, Method
- **C++:** Project, File, Class, Struct, Union
- **Fortran:** Project, File, Module, Function, Program, Subroutine
- **Java:** Project, File, Class, Interface, Method
- **Jovial:** Project, File
- **Pascal:** Project, File, Class, Interface, Compunit, Function, Procedure
- **Python:** Project, File, Class
- **Rust:** Project, File
- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated function metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Complexity" with the "Essential" option.

Sum Modified Cyclomatic Complexity

API ID: SumCyclomaticModified
Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, Web
Targets: Architectures, Classes, Files, Functions, Modules, Packages, Project,

Subprograms

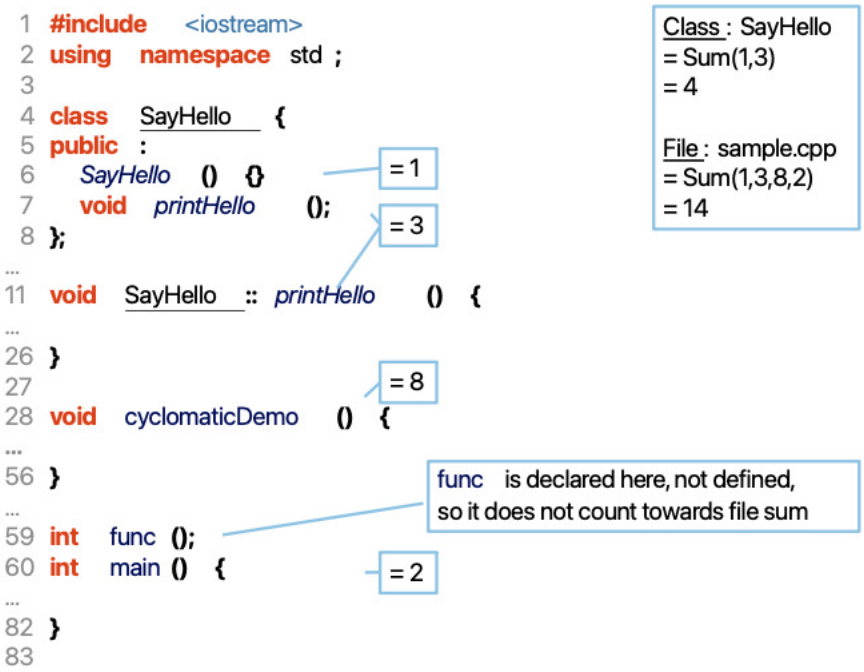
Sum of modified cyclomatic complexity of all nested functions or methods.

For example:

```
1 class SayHello {
2 public:
3   SayHello() {}
4   void printHello();
5 };
6
7 void SayHello::printHello() {
8   switch (i) {
9     ...
10   }
11 }
12
13 void cyclomaticDemo() {
14   ...
15 }
16
17 int func();          // declared here, not defined - excluded from aggreg
18
19 int main() {
20   ...
21 }
```

SayHello(), printHello(), cyclomaticDemo(), and main() have Modified Cyclomatic Complexity of 1, 3, 8, and 2. func() is only declared, not defined, so it isn't counted. For the SayHello class, the sum is 1 + 3 = 4. For the whole file (sample.cpp), the sum is 1 + 3 + 8 + 2 = 14.

See Max and Average Modified Cyclomatic Complexity for the other aggregations.



Targets By Language:

- **Ada:** Project, File, Type, Entry, Function, Package, Procedure, Protected, Task
- **Basic:** Project, File, Method, Module, Class, Struct
- **C#:** Project, File, Class, Struct, Method
- **C++:** Project, File, Class, Struct, Union
- **Fortran:** Project, File, Module, Function, Program, Subroutine
- **Java:** Project, File, Class, Interface, Method
- **Jovial:** Project, File
- **Pascal:** Project, File, Class, Interface, Compunit, Function, Procedure
- **Python:** Project, File, Class
- **Rust:** Project, File
- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated function metrics" option.
- The cyclomatic metric used is configured from "Project Configuration/Metrics/Complexity". This metric results from having "Count each decision in a multi-way-decision structure" off and having "Count logical conjuctions (OR) and logical and (AND)" off. Checking "Show all available cyclomatic complexity metric variants" will also enable the metric.

Sum Strict Cyclomatic Complexity

API ID: SumCyclomaticStrict
Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, Web
Targets: Architectures, Classes, Files, Functions, Modules, Packages, Project, Subprograms

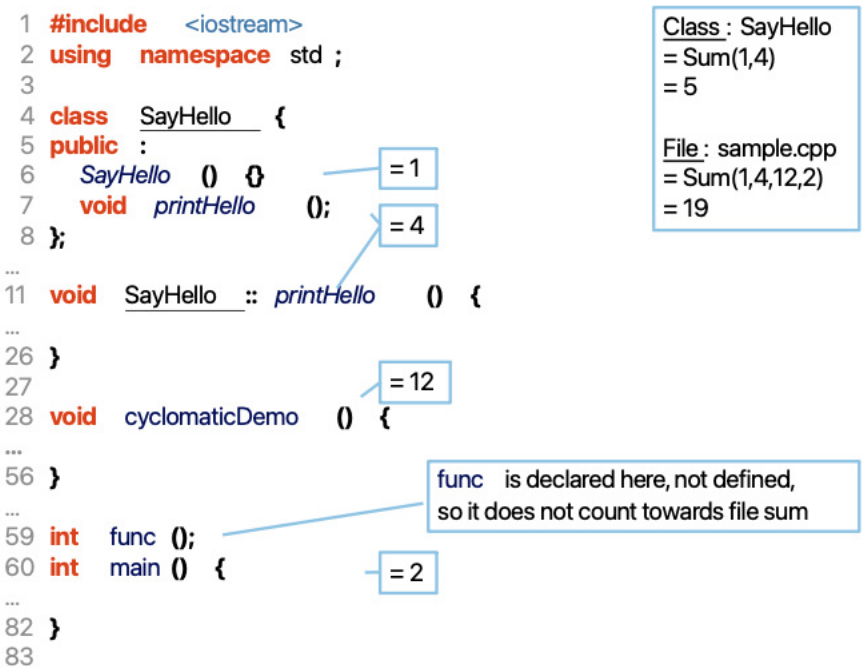
Sum of strict cyclomatic complexity of all nested functions or methods.

For example:

```
1 class SayHello {
2 public:
3   SayHello() {}
4   void printHello();
5 };
6
7 void SayHello::printHello() {
8   switch (i) {
9     ...
10   }
11 }
12
13 void cyclomaticDemo() {
14   ...
15 }
16
17 int func();          // declared here, not defined - excluded from aggreg
18
19 int main() {
20   ...
21 }
```

SayHello(), printHello(), cyclomaticDemo(), and main() have Strict Cyclomatic Complexity of 1, 4, 12, and 2. func() is only declared, not defined, so it isn't counted. For the SayHello class, the sum is 1 + 4 = 5. For the whole file (sample.cpp), the sum is 1 + 4 + 12 + 2 = 19.

See Max and Average Strict Cyclomatic Complexity for the other aggregations.



Targets By Language:

- **Ada:** Project, File, Type, Entry, Function, Package, Procedure, Protected, Task
- **Basic:** Project, File, Method, Module, Class, Struct
- **C#:** Project, File, Class, Struct, Method
- **C++:** Project, File, Class, Struct, Union
- **Fortran:** Project, File, Module, Function, Program, Subroutine
- **Java:** Project, File, Class, Interface, Method
- **Jovial:** Project, File
- **Pascal:** Project, File, Class, Interface, Compunit, Function, Procedure
- **Python:** Project, File, Class
- **Rust:** Project, File
- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated function metrics" option.
- The cyclomatic metric used is configured from "Project Configuration/Metrics/Complexity". This metric results from having "Count each decision in a multi-way-decision structure" on and having "Count logical conjuctions (OR) and logical and (AND)" on. Checking "Show all available cyclomatic complexity metric variants" will also enable the metric.

Sum Strict Modified Cyclomatic Complexity

API ID: SumCyclomaticStrictModified
Languages: Ada, Basic, C#, C++, Fortran, Java, Jovial, Pascal, Python, Rust, Web
Targets: Architectures, Classes, Files, Functions, Modules, Packages, Project, Subprograms

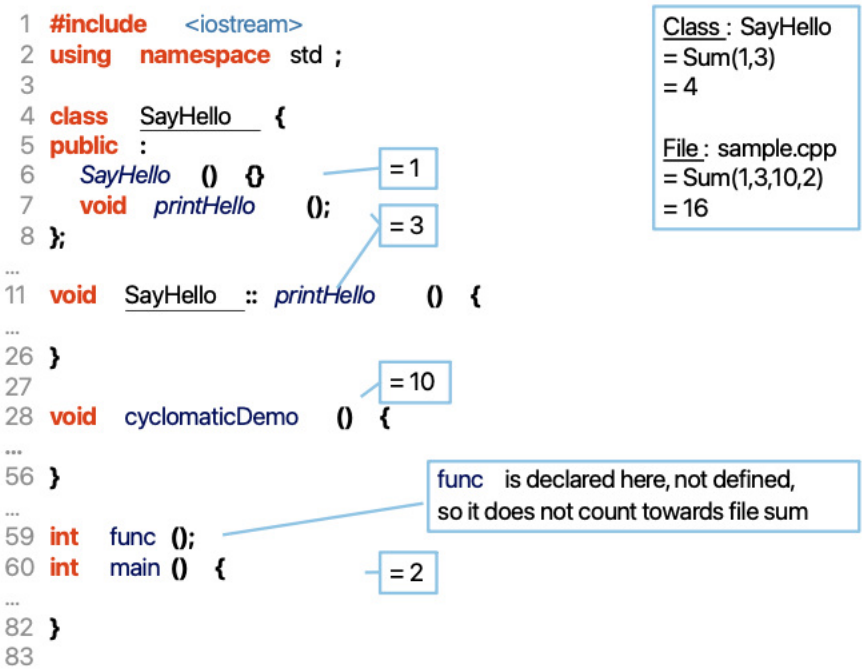
Sum of strict modified cyclomatic complexity of all nested functions or methods.

For example:

```
1 class SayHello {
2 public:
3   SayHello() {}
4   void printHello();
5 };
6
7 void SayHello::printHello() {
8   switch (i) {
9     ...
10   }
11 }
12
13 void cyclomaticDemo() {
14   ...
15 }
16
17 int func();          // declared here, not defined - excluded from aggreg
18
19 int main() {
20   ...
21 }
```

SayHello(), printHello(), cyclomaticDemo(), and main() have Strict Modified Cyclomatic Complexity of 1, 3, 10, and 2. func() is only declared, not defined, so it isn't counted. For the SayHello class, the sum is 1 + 3 = 4. For the whole file (sample.cpp), the sum is 1 + 3 + 10 + 2 = 16.

See Max and Average Strict Modified Cyclomatic Complexity for the other aggregations.



Targets By Language:

- **Ada:** Project, File, Type, Entry, Function, Package, Procedure, Protected, Task
- **Basic:** Project, File, Method, Module, Class, Struct
- **C#:** Project, File, Class, Struct, Method
- **C++:** Project, File, Class, Struct, Union
- **Fortran:** Project, File, Module, Function, Program, Subroutine
- **Java:** Project, File, Class, Interface, Method
- **Jovial:** Project, File
- **Pascal:** Project, File, Class, Interface, Compunit, Function, Procedure
- **Python:** Project, File, Class
- **Rust:** Project, File
- **Web:** Project, File, PHP Class, PHP Interface

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated function metrics" option.
- The cyclomatic metric used is configured from "Project Configuration/Metrics/Complexity". This metric results from having "Count each decision in a multi-way-decision structure" off and having "Count logical conjuctions (OR) and logical and (AND)" on. Checking "Show all available cyclomatic complexity metric variants" will also enable the metric.

Sum Strict Modified Essential Complexity

API ID: SumEssentialStrictModified
Languages: Ada
Targets: Architectures, Files, Packages, Project, Subprograms

Sum of strict modified essential complexity of all nested functions or methods.

For example:

```
1 package Essential_Demo is
2    procedure Simple_Proc (X : in out Integer);
3    function Strict_Func (A, B : Boolean; N : Integer) return Integer;
4    function Case_Func (Day : Integer) return Integer;
5    function Mixed_Func (A, B : Boolean; Day : Integer) return Integer;
6 end Essential_Demo;
```

Simple_Proc, Strict_Func, Case_Func, and Mixed_Func have Strict Modified Essential Complexity of 1, 5, 2, and 7. For the Essential_Demo package, the sum is 1 + 5 + 2 + 7 = 15, as it is for the file containing their implementations (essential.adb). A subprogram that nests other subprograms sums their values the same way.

See Max and Average Strict Modified Essential Complexity for the other aggregations.

Targets By Language:

- **Ada:** Project, File, Type, Entry, Function, Package, Procedure, Protected, Task

Configuration:

- For projects and architectures, this metric must also be enabled from "Project Configuration/Metrics/Projects and Architectures" with the "Show aggregated function metrics" option.
- This metric can be enabled from "Project Configuration/Metrics/Complexity" with the "Essential" option.