CODACY

Software metrics:

A practical guide for the curious developer

www.codacy.com

Index

Why software metrics are important

"If debugging is the process of removing software bugs, then programming must be the process of putting them in."
- Edsger Dijkstra

Is the code tested? Is it readable? How easy is it to add or modify code without introducing bugs? Is it more complex than it needs to be? How maintainable is it going to be? Is it resource efficient? Is it easily extensible? Is technical debt creeping up?

Software metrics are important because they help answer these questions. They provide objective, quantifiable measurements that support the myriad of decisions that developers make daily.

Yet software metrics are not without their flaws, far from it. First, no single software metric can claim to provide a complete and accurate picture. Second, most metrics fail as universal yardsticks: sensitivity to factors such as programming language or even the lack of consensus on the algorithms that underpin them, lead to confusion and frustrations. Consequently, most developers welcome them with about the same enthusiasm as someone expecting a root canal job.

But that's just throwing the baby out with the bathwater.

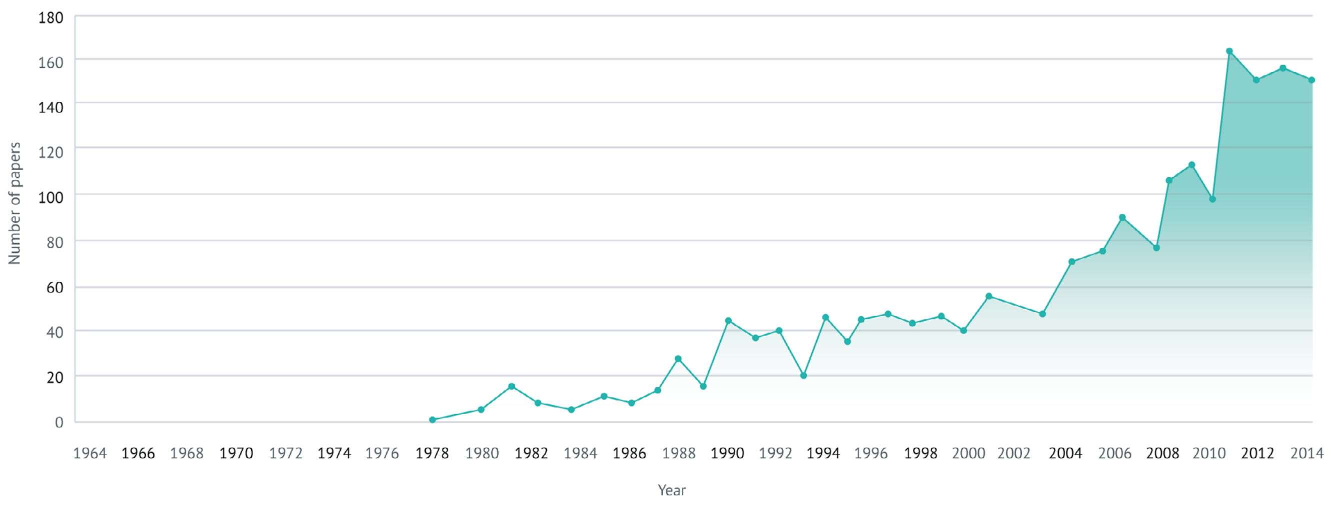
Of course, common sense activities, such as ongoing tests and code reviews are among the single best ways to ensure a healthy level of code quality. But what about visibility over the overall health of the codebase? Is there a way to automatically flag potential issues? Can qualitative judgement be complemented with quantitative information for the sake of better decision-making?

This is where software metrics come in.

Scope of this e-book

The history of software metrics is almost as old as the history of software engineering, but it wasn't until the 70's when the topic took off. A casual Google Scholar search shows that the first research papers surfaced in the late 70's - the volume of papers has since shot up.

Number of software metrics papers in Google Scholar



Here it's worth mentioning that software metrics can be divided in many categories. Functional quality metrics for example aim to assess how well a piece of software fulfils the mission is was designed for. But we won't be discussing these. Instead, we'll be looking at **structural quality metrics.**

Structural quality metrics are the ones that assess the quality of the code base - the inner structure of the software. We've hand-picked a selection of them because (1) they're among the most widely used, (2) they're easier to understand and therefore, more useful and (3) they play nicely within continuous testing, integration and delivery environments.

The purpose of this article is to provide a quick and easy overview of useful metrics, and how to interpret them. We therefore do not dwell too much on the theoretical underpinnings of each metrics, or the mathematics of it.

This is a cheat sheet for developers in a hurry - or their manager - who want a practical list of useful metrics and to get work done.

I. The Good

These are our favourite metrics: they are relatively easy to understand, and therefore easier to use. While not without their flaws, they help developers identify potential hotspots in the code, provide clues to reduce complexity and help with code decomposition and abstraction.

Cyclomatic complexity

"Complexity kills. It sucks the life out of developers, it makes products difficult to plan, build and test, it introduces security challenges, and it causes end-user and administrator frustration."

- Ray Ozzie

Brief description
Cyclomatic complexity is a measure of code complexity. In theory, the cyclomatic complexity formula is an equation with the number of edges, nodes and connected components of the control flow graph of the program as variables. In practice, it's near impossible for static analysis tools to measure this properly. Instead, they rely on proxies such as the number of times they encounter expressions such as IF, WHILE, FOR, FOREACH.

Possible range of values
1 to infinity.

Benchmark values
For method-level complexity, a maximum of 10 is the norm. As a rule of thumb:

Less than 10: Easy to maintain
Between 10 and 20: Harder to maintain
Over 20: Candidates for refactoring/redesign

Value for developers

Cyclomatic complexity is a simple way to flag bits of code that are too complex and therefore need to be reviewed. For example methods that have high cyclomatic complexity (i.e. a score over 20) are probably candidates for refactoring. On the other hand, low cyclomatic complexity is an indication of good decomposition of the problem into simple elements. All else being equal, code with low complexity is more readable and maintainable, easier to test and debug. Cyclomatic complexity is a useful -albeit imperfect- way to identify code smells and zero in on issues early in the development process.

Limitations

Like most metrics in this article, cyclomatic complexity is far from perfect. For starters, different tools measure cyclomatic complexity in different ways. Also, the implementation may vary between languages. Developers should therefore use caution when comparing the complexity of different programs.

Codacy advice

Cyclomatic complexity is particularly useful when applied to methods. It helps keep the discipline of refactoring code so that the methods only serve one specific function. Also, making sure that complexity is low at all times is a useful way to make sure that the code is properly decomposed.

Further reading

Cyclomatic complexity. Wikipedia. Link.

A complexity measure. Thomas J. McCabe. IEEE Transactions on Software Engineering, Vol. SE-2, No 4, December 1976. Link.

Code churn

"The first 90% of the code accounts for the first 90% of the development time. The remaining 10% of the code accounts for the other 90% of the development time."

- Tom Cargill

Brief description

Code churn is the number of lines added, modified or deleted to a file from one version to another.

Possible range of values

0 to infinity.

Benchmark values

Not applicable.

Value for developers

Code churn helps quantify the amount of change that is occurring in the project. Areas of the code that are subject to high levels of churn are less stable and may be sources of defects. When combined with ongoing code reviews, it is a useful metric to gage what the team is focusing on, whether they are testing their code enough, and identify potential bottlenecks.

Codacy advice

As a stand alone metric, code churn is moderately useful. There is some empirical evidence that relative churn (i.e. churned code divided by Lines of Code), is a better predictor of number of defects than absolute churn. An even more interesting information however is the churn vs. complexity scatterplot, which we discuss in the section below.

Further reading

Code Churn: A Measure for Estimating the Impact of Code Change. Munson and Elbaum (1998). Link.

Use of Relative Code Churn Measures to Predict System Defect Density. Nagappan and Ball (2005). Link.

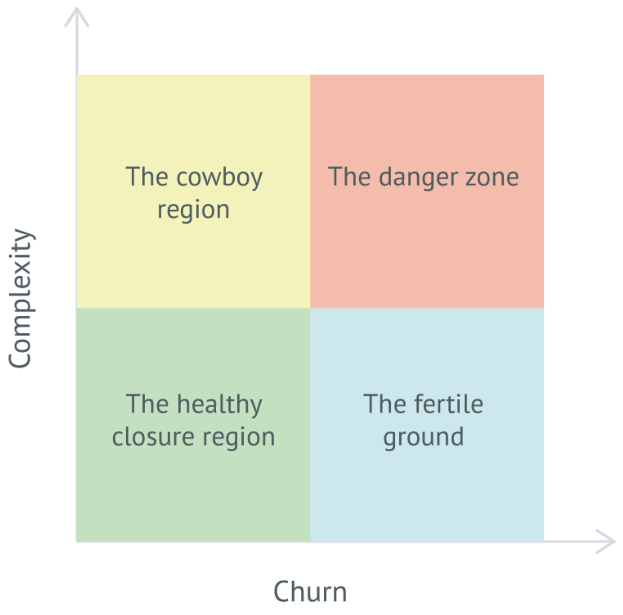
Churn vs. Complexity

"Any darn fool can make something complex; it takes a genius to make something simple."

- Pete Seeger

Brief description

Churn vs complexity - a concept first documented by Michael Feathers - is the scatterplot of churn on the x-axis and complexity on the y-axis. The scatterplot is then divided in four quadrants:



The danger zone are the areas of the code that are complex and modified a lot. The question then is whether complexity increases, which may reflect challenges with decomposition and abstraction, or whether it decreases, a possible indication of good progress.

The cowboy region are the areas of the code that are complex and not often modified. The questions then are: why wasn't the code refactored? Is the work unfinished? Was there a bottleneck? Did the team focus its efforts elsewhere?

The healthy closure region are the areas of the code that are neither complex nor modified a lot. This is usually an indication of mature, stable code.

The fertile ground is the area with little complexity and a lot of modifications. This happens for example with configuration files.

Value for developers

The churn vs complexity scatterplot provides a visual clue of the areas of the code that are being worked on and those need refactoring. Crucially, it helps understand how the team is approaching the project, and provides inputs for discussion on how best to manage the time and resources.

Codacy advice

As a stand alone metric, code churn is moderately useful. There is some empirical evidence that relative churn (i.e. churned code divided by Lines of Code), is a better predictor of number of defects than absolute churn. An even more interesting information however is the churn vs. complexity scatterplot, which we discuss in the section below.

Further reading

Getting Empirical about Refactoring. Michael Feathers (2011). Link.

Code coverage

"Program testing can be used to show the presence of bugs, but never to show their absence."

- Edsger Dijkstra

Brief description

Code coverage is the proportion of code covered by tests.

How it's measured

There are many ways to measure it: subroutine coverage, statement coverage, branch coverage, path coverage, etc. Also, the underlying tests may vary: unit tests, functional tests, etc. Depending on the role of the developer and how far down the development lifecycle the project is, different code coverage tests will apply.

Possible range of values
0 to 100%

Benchmark values
Developers should aim to reach 80% coverage. In practice however, the average is in the 60% - 80% range.

Value for developers
From an organizational perspective, the biggest benefit of having code coverage targets is that it foster a culture of continuous testing. From a management perspective, it is one of several metrics that help provide confidence in the quality of the code. From a developer perspective, code coverage helps identify areas of the code that need further testing early in the development lifecycle and degree of confidence when deploying code to production systems.

Limitations
Code coverage is only as good as the quality of the tests that drive it. Also, developers can game the system by writing unit tests that just increase coverage, at the cost of actually testing the application meaningfully.
Good code coverage policy should therefore go hand in hand with a clear definition of the testing policy (unit, functional, at what stage, etc.).

Codacy advice
Developers should aim to spend 10% approx of their time on unit tests until they reach the targeted code coverage threshold. While the majority of developers have minimum thresholds in the 60-80% range, we also recommend setting a threshold of 80% and the use of continuous integration to maintain the desired coverage level.

Further reading

Code coverage. Wikipedia. Link.

Testing and code coverage, Paul Johnson. Link.

Code duplication

"I think one of the most valuable rules is avoid duplication. "Once and only once" is the Extreme Programming phrase."

- Martin Fowler

Brief description

Code duplication is a repetition of sets of instructions or their structure or semantics within the code base. It also happens when different code perform the same job, even though they look different. Code duplication is in violation of the 'DRY' (Don't Repeat Yourself) principle.

How it's measured

There are different types of duplication (Data, Type, Algorithm), at different levels (module vs. class vs.method) and with different formats (character-for-character, token-for-token, functionality). There are therefore many ways to measure code duplication.

Possible range of values

0 to 100%

Benchmark value

0

Value for developers

Duplicated code makes for harder to read and maintain code. It may also reflect bad code quality, poor development processes or sloppy design. But more importantly, it can reflect the failure to decompose or abstract the code adequately. Making sure that the codebase has as little as possible code duplication often leads to more elegant and higher code quality.

Limitations

Code duplication is one of those easy to understand, yet hard to measure metrics. Static analysis tools can help, however developers will have to configure the tool to minimize false positives.

Codacy advice

Code duplication is an easy way to assess code quality and identify parts of the code that need to be reviewed. Developer team should really aim for zero duplication. In the words of Martin Fowler: "*Look at some program and see if there's some duplication. Then, without really thinking about what it is you're trying to achieve, just pigheadedly try to remove that duplication. Time and time again, I've found that by simply removing duplication I accidentally stumble onto a really nice elegant pattern.*"

Further reading

Duplicate code. Wikipedia. Link.

Book: Refactoring. Improving the Design of Existing Code. Martin Fowler. Link.

Lack of Cohesion Of Methods (LCOM)

Brief description

Cohesion is an important concept in object-oriented programming. LCOM in particular is used to measure the cohesion of each class of the system. There are several ways to compute it, but to understand the concept, let's look at the Henderson-Sellers formula:

LCOM HS = (memberCount – sumFieldUsage / fieldCount)/(memberCount - 1)

Where:

memberCount = number of methods in class (including getter/setter methods and constructors).

fieldCount = number of instance fields in the class.

sumFieldUsage = the sum of the number of methods accessing a particular field.

Value for developers

The basic idea of LCOM is straightforward: if all the methods of a class use all its instance field, then the value of LCOM is 0 and the class is cohesive. On the other hand, classes with low cohesion (i.e. values of 1 or greater with more than 10 members and more than 10 fields) are potentially in breach of the single responsibility principle. In that case, developers should look at subdividing the class into subclasses with greater cohesion.

Further reading

Calculating Lack of Cohesion of Methods with Roslyn. Link.

Single responsibility principle. Wikipedia. Link.

Cohesion. Wikipedia. Link.

A validation of object-oriented design metrics as quality indicators. Basili et al. 1995. Link.

Relational cohesion

This metric assesses the strength of relationships within a module. The formula is:

$$H = (R + 1) / N.$$

Where

R is the number of references inside the module

N is the number of types within the module

So if a module has 5 classes and each class references 3 classes within the same module, the value of the metric is (5 x 3 + 1) / 5, which equals 3. Ideally the value should be between 1.5 and 4.0. Too high a value is an indication of complexity. Too low a value indicates loose relationship.

Further reading

Relational cohesion, nDepend Metrics. Link.

II. The Bad

With software development, there's always a probability that a modification to a specific area of the code with ripple through the entire code base. These metrics provide clues as to the role a package plays in the architecture of the programme. There are primarily useful for software architects but developers may find them useful too.

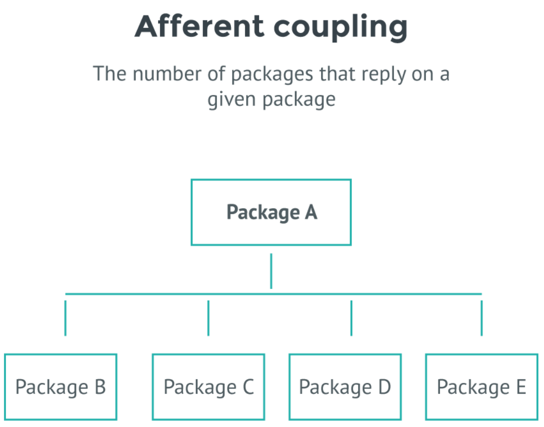
Afferent & efferent coupling

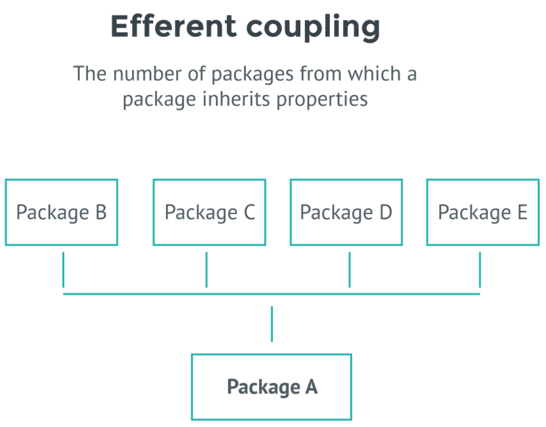
"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

- C.A.R. Hoare

Brief description

Afferent coupling is the number of packages that depend on a given package. Efferent coupling is the number of packages that a given package depends on.




Efferent coupling
The number of packages from which a package inherits properties
Package B
Package C
Package D
Package E
Package A

Possible range of values
For afferent coupling: 0 to infinity.
For efferent coupling: 0 to infinity.

Benchmark values
For afferent coupling: Max 500
For efferent coupling: Max 20.

Value for developers
The higher the **afferent coupling** of a package, the more important this package is in your code base. Having packages with high afferent coupling may reflect a low level of code duplication, which is good. However, making changes to them increases the likelihood of something breaking somewhere. Any change to these package should therefore be thoroughly tested. On the other hand, a value of zero may indicate dead code, although you may need to execute the code for confirmation.

Packages with high **efferent coupling** are probably trying to do too much and are more likely to break if any of the packages it depend on change. To avoid this problem, a package should have a single responsibility and few parents; consider decomposing a package with high efferent coupling into smaller, simpler packages with single responsibility.

Codacy advice
Afferent and efferent coupling provide flags for those bits of code that can propagate defects across a system, those that need extra testing before implementing any modification, those that are potentially useless, and those that should be decomposed into simpler packages. We therefore recommend developers keep an eye on it, along with the instability, abstractness, and distance from the main sequence metrics, which we discuss in the next sections.

Further reading
Software package metrics Wikipedia article. Link.
Law of Demeter Wikipedia article. Link.

Instability

"If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization."

- Gerald Weinberg

Brief description

Package instability is a measure of its resistance to change. The more a package is stable, the harder it is to change it. Instability is computed by dividing efferent coupling to total coupling. In other words:

I = Ce / (Ce + Ca),

Where

I is the degree of instability,

Ce is efferent coupling and

Ca is afferent coupling.

Possible range of values

A value of 1 means the package is highly unstable, and a value of 0 means it is stable.

Benchmark value

For stable packages: between 0.0 and 0.3

For unstable packages: 0.7 to 1.0.

Value for developers

This metric, when used in conjunction with the afferent and efferent coupling metrics, is an excellent indicator of the role played by a package.

A low value (i.e. high afferent coupling relative efferent coupling) is an indication of stability: it depends on few other packages, and many packages depend on it. You therefore want this package to remain stable, since making changes will impact the rest of the application.

A medium value (i.e. similar values for efferent and afferent coupling) indicates that this is a package through which bugs will ripple.

A high value (i.e. high efferent coupling relative to afferent coupling) indicates instability: it depends on many other package and few packages depend on it. Making changes is therefore easier and will have a lesser impact on the rest of the application. On the other hand, when changes are made to other packages, you'll want to check whether it is affected or not.

Codacy advice
Competitive pressures dictate that code should be easy to modify, with features constantly being added, removed, or optimized. Packages should therefore be very stable (with a value between 0 to 0.3) or unstable (with a value from 0.7 to 1). Instability is a data point that developers can use to ensure good software design, decomposition of the problem and levels of abstractions.

Further reading
Software package metrics Wikipedia article. Link.

Design Principles and Design Patterns. Object Mentor. Robert C. Martin (2000). Link.

Abstractness

"The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise."

- Edsger Dijkstra

Brief description
To quote wikipedia, abstractness is "the ratio of the number of abstract classes (and interfaces) in the analyzed package to the total number of classes in the analyzed package."

Possible range of values
Between 0 (indicating a completely concrete package) and 1 (indicating a completely abstract package).

Benchmark values
n/a

Codacy advice
Abstractness is, in itself, not very useful. When combined with the Instability metric however, developers can calculate the distance from the main sequence, which we discuss in the next section.

Further reading
Stack overflow: What is the difference between a concrete class and an abstract class? Link.

OO Design Quality Metrics. An Analysis of Dependencies. Robert C. Martin (1994). Link.

Distance from main sequence (DMS)

"Good design adds value faster than it adds cost."

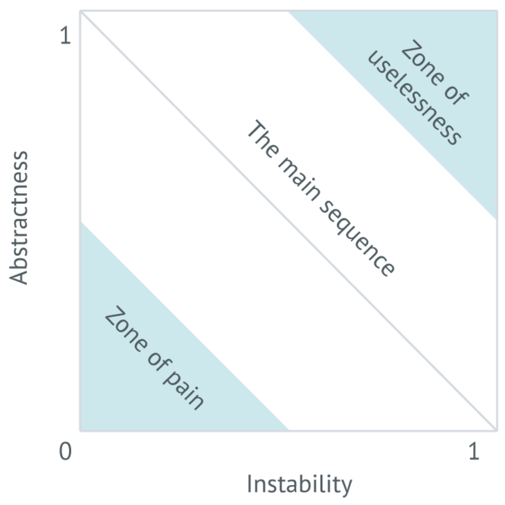
- Thomas C. Gale

Brief description
The distance from the main sequence is the balance between the abstraction and instability of a package. The more abstract a package is, the more stable it should be. Conversely, the more concrete a package is, the more unstable it should be.

How it's measured
Formally, it is the perpendicular normalized distance of a package from the idealized line

$$A + I = 1,$$

which is called the main sequence (see illustration below). The metric ranges from 0.0 (on the main sequence) to 1.0 (very far from the main sequence).



Possible range of values
0 to 1

Benchmark values
Max. 0.7

Codacy advice
Distance from the main sequence is interesting when used in combination with the afferent and efferent coupling, and instability metrics. Taken together they can highlight potential weaknesses in the architecture of a software: any package that has a value over 0.1 should be re-examined. Values above 0.7 is a flag for potential problems.

Further reading
Design Principles and Design Patterns. Object Mentor. Robert C. Martin (2000). Link.

III. The Ugly

These metrics are at best moderately useful. Individual developers may want to keep an eye on them and review code that lie several standard deviations away from the average.

Density of comments

"Everything is complicated if no one explains it to you."

- Fredrik Backman

Brief description
The Density of Comments (DC) is the ratio of comment lines (CLOC) to all lines (LOC). The formula is:

DC = CLOC / LOC

Possible range of values
0 to 100%

Benchmark values
20% to 40%

Value for developers
All developers know that good code is usually well commented code. What "well commented code" actually means is a controversial topic. Density of comment says nothing about the quality of the comments. Also, the calculation of the density of comments depends directly on SLOC, which is flawed, so DC cannot be used as an objective assessment of code quality. When DC is below 20% however, it might be worth reviewing the code to make sure that it is perfectly understandable and does what it is supposed to do.

Codacy advice
Density of comments is a source of ongoing debate at Codacy, and as I write these lines some of my colleagues are preparing the tar and feathers - understandably so. Ideally, the code should be so clear that it is self-explanatory and no comments are required. However, there are times when it makes sense to clarify what the code should be doing. Rather than setting targets however, it's best to have regular code reviews and foster the sharing of commenting best practices within a company.

Further reading
The Fine Art of Commenting. Bernhard Spuida, 2002. Link.

Code as documentation. Martin Fowler. 2005. Link.

Source line of code (SLOC)

"Measuring programming progress by lines of code is like measuring aircraft building progress by weight."
- Bill Gates

Brief description
Source line of code (SLOC) - also known as lines of code (LOC) - is a measure of the number of lines of the program's source code.

How it's measured
There are at least two different measures of SLOC:

The "physical" SLOC. That's the total number of lines in the source code files.

The "logical" SLOC. That's the number of lines that will be executed.

Possible range of values
1 to infinity

Benchmark values
Not applicable.

Value for developers
Intuitively, all developers understand that the higher the SLOC, the larger the number of potential bugs, the higher the maintenance costs, the efforts to develop new features, and so on. The problem is that it's not possible to standardize or benchmark this metric in any meaningful way: a change in SLOC in itself says nothing about code quality, or developer productivity.
To make matter worse, and despite the simplicity of its definition, there is no consensus on how to measure SLOC. So different static analysis tools will provide different values. Developers therefore cannot use it to make comparisons or draw conclusions.

Codacy advice
Although we're not big fans of SLOC as a standalone metric, if you have to use it, then use the "logical" SLOC. Another practice which we find useful is to standardize SLOC by measuring SLOC per file (or per class). When the SLOC per file is far from the average, this is often a flag that something has to be reviewed.

Further reading
Source lines of code. Wikipedia. Link.

Number of parameters

Simply the number of parameters of a method. There are no clear rules of thumb. According to code complete, you should limit the number of parameters to 7 approx. However, the best approach is to review the code to make sure that the code to identify refactoring opportunities.

Number of variables

This is the number of variables declared in a method. The thing to do is to review the methods that have high numbers of variables, and consider dividing them.

Number of overloads

This is simply the number of overloads of a method. The higher the number of overloads, the less readable the code, and the more opportunities for defects.

Bonus: The best metric of all

Profanities per hour expressed by the developer
It’s a classic and it’s so true. Simply put, good code should have low complexity, be straightforward to read and understand. Crucially, it should be a pleasure to read and work with.

Conclusion

We have seen that no single software metric can be used to gage project quality accurately. Nor do they provide hard and fast answers to follow blindly.

Instead, software metrics provide additional inputs for meaningful conversations during tests and code reviews. Using a combination of them sharpens the developer intuition and contributes towards making software engineering a more predictable affair.

About Codacy

Codacy is a code review and code quality automation platform used by tens of thousands of open source users and industry leaders, including Adobe, Paypal, Deliveroo, Schneider Electric, Schlumberger, and many others.

With Codacy, developers save up to 50% of the time spent on code reviews: The codebase is continuously analyzed to help developers address potential problems early on in the development cycle. Codacy also provides a range of code metrics to help teams track the evolution of their codebase and reduce technical debt throughout your sprints.

Integration with your workflow is easy, thanks to the integration with a wide range of developer tools (GitHub, BitBucket, GitLab, and more).

Codacy is free for public, open source projects, and has a paid plan for private repositories. Enterprise plans are available for organizations who want to host Codacy on their own servers.

You can learn more about Codacy here. You can also sign up for a free trial or request a demo.

Cheat sheet

Metric	Description	Recommendation	Value for developers
Cyclomatic complexity	A measure of control flow complexity, often measured by the number of expressions such as IF, WHILE, FOR, FOREACH.	For modules: - Less than 10: Easy to maintain - Between 10 and 20: Harder to maintain - Over 20: Candidates for refactoring/redesign	★ ★ ★
Code churn	The number of lines added, modified or deleted to a file from one version to another.	Use relative churn (i.e. churned code divided by Lines of Code) instead and churn vs. complexity.	★
Churn vs complexity	The scatterplot of churn on the x-axis and complexity on the y-axis.	Provides a visual clue of the areas of the code that are being worked on and those need refactoring and how the team is approaching the project.	★ ★
Code coverage	The proportion of code covered by tests.	Target 80% coverage ratio and foster a culture of continuous testing. Code coverage is only as good as the underlying tests.	★ ★ ★
Code duplication	A repetition of sets of instructions or their structure or semantics within the code base.	Developer team should really aim for zero duplication. Evidence suggests that efforts to eliminate duplication results in better code.	★ ★ ★
Lack of Cohesion Of Methods (LCOM)	A measure of the cohesion of each class of the system.	Classes with low cohesion (i.e. values of 1 or greater with more than 10 members and more than 10 fields) are potentially in breach of the single responsibility principle.	★ ★ ★
Relational cohesion	Assesses the strength of relationships within a module.	The value should be between 1.5 and 4.0.	★ ★ ★
Afferent & efferent coupling	Afferent coupling is the number of packages that depend on a given package. Efferent coupling is the number of packages that a given package depends on.	Afferent and efferent coupling provide flags for those bits of code that can propagate defects across a system, those that need extra testing before implementing any modification, those that are potentially useless, and those that should be decomposed into simpler packages.	★

Metric	Description	Recommendation	Value for developers
Instability	Package instability is a proxy for the level of resistance to change.	Packages should be very stable (with a value between 0 to 0.3) or unstable (with a value from 0.7 to 1).	★
Abstractness	The ratio of the number of abstract classes (and interfaces) to the total number of classes within a given package.	Abstractness is mainly useful to calculate the distance from the main sequence.	★
Distance from main sequence (DMS)	The balance between the abstraction and instability of a package.	Highlight potential weaknesses in the architecture of a software: any package that has a value over 0.1 should be re-examined. Values above 0.7 is a flag for potential problems.	★
Source line of code (SLOC) per file	A measure of the number of lines of the program's source code.	Review the code that lies several standard deviations away from the average.	★ ★
Number of parameters	The number of parameters of a method		
Number of variables	The number of variables declared in a method.		
Number of overloads	The number of overloads of a method.		
Density of comments	The ratio of comment lines to all lines	Rather than setting targets it's best to have regular code reviews and foster the sharing of commenting best practices.	★

Further reading

20% of code review comments are about style and best practices. Link.

6 ways to formalize and enforce code reviews. Link.

A complexity measure. Thomas J. McCabe. IEEE Transactions on Software Engineering, Vol. SE-2, No 4, December 1976. Link.

A validation of object-oriented design metrics as quality indicators. Basili et al. 1995. Link.

Calculating Lack of Cohesion of Methods with Roslyn. Link.

Code complete. Steve McConnell. Link.

Code Review vs. Testing. Link.

Code Review Etiquette. Link.

Cyclomatic complexity. Wikipedia. Link.

Code as documentation. Martin Fowler. 2005. Link.

Code Churn: A Measure for Estimating the Impact of Code Change. Munson and Elbaum (1998). Link.

Code coverage. Wikipedia. Link.

Cohesion. Wikipedia. Link.

Design Principles and Design Patterns. Object Mentor. Robert C. Martin (2000). Link.

Developing with Code Reviews. Link.

Duplicate code. Wikipedia. Link.

Getting Empirical about Refactoring. Michael Feathers (2011). Link.

How to prioritize your technical debt? Link.

Law of Demeter. Wikipedia. Link.

OO Design Quality Metrics. An Analysis of Dependencies. Robert C. Martin (1994). Link.

Refactoring. Improving the Design of Existing Code. Book by Martin Fowler. Link.

Relational cohesion, nDepend Metrics. Link.

Single responsibility principle. Wikipedia. Link.

Software package metrics Wikipedia article. Link.

Source lines of code. Wikipedia. Link.

Testing and code coverage, Paul Johnson. Link.

The Fine Art of Commenting. Bernhard Spuida, 2002. Link.

The ultimate guide to code reviews. Codacy Ebook. Link.

Top 10 ways to be a faster code reviewer. Link.

Use of Relative Code Churn Measures to Predict System Defect Density. Nagappan and Ball (2005). Link.

What is the difference between a concrete class and an abstract class? Stack overflow. Link.

What your mother didn’t tell you about Code Coverage. Link.