

SMART TECHNICAL DEBT ESTIMATION

Table of contents

1. Introduction	3
2. Annual-Interest and Severity.....	5
3. Debt Settings	6
4. SQUALE Debt Ratio and Debt Rating	7
5. Prioritizing issues fix and the Breaking-Point metric.....	9
6. Browsing the Technical-Debt	11
7. Technical Debt and Quality Gate.....	16
8. Reasons why Technical Debt might be Zero or incomplete	17

1. Introduction

Nowadays, the technical-debt metaphor has been widely adopted by the software industry. It was coined by Ward Cunningham in 1992.

This reference article by Martin Fowler describes the technical-debt metaphor in great detail. To quote M.Fowler:

In this metaphor, doing things the quick and dirty way sets us up with a technical debt, which is similar to a financial debt. Like a financial debt, the technical debt incurs interest payments, which come in the form of the extra effort that we have to do in future development because of the quick and dirty design choice. We can choose to continue paying the interest, or we can pay down the principal by refactoring the quick and dirty design into the better design. Although it costs to pay down the principal, we gain by reduced interest payments in the future.

Using CppDepend, code rules can be written through C# LINQ queries. Applied on a code base a rule yields issues. A dedicated debt API is proposed to estimate both the technical-debt and the annual-interest of the issue through formulas written in C#. Both the technical-debt and annual-interest of an issue are measured in man-time.

The technical-debt is the estimated man-time that would take to fix the issue.

The annual-interest is the estimated man-time consumed per year if the issue is left unfixed. This provides an estimate of the business impact of the issue.

For example:

```
warnif count > 0
from m in Methods
where m.CyclomaticComplexity > 10
select new {
    m,
    m.CyclomaticComplexity,
    Debt = (3*(m.CyclomaticComplexity - 10)).ToMinutes().ToDebt(),
    AnnualInterest = (m.PercentageCoverage == 100 ? 10 : 120).ToMinutes().ToAnnualInterest()
}
```

In this example, the rule matches methods which are too complex, the complexity being measured through the [Cyclomatic Complexity](#) code metric. We can see that:

- The **technical-debt** is proportional to the complexity above a certain threshold.
- The **annual-interest** is 10 minutes per year if the method is 100% covered by tests, else it is 2 hours per year.

Leaving a complex method both unrefactored and uncovered by tests is an error-prone situation. At best such a situation impairs maintainability of the code, at worse it ends up in bugs at production time. The annual-interest estimates the average cost per year if the complex method is left unrefactored. This worsens if the method also is uncovered by tests. The word average is highlighted here due to the fact that, for example, out of 8 complex and untested methods maybe only one has a bug that will cost 2 days of man-work (2x8 hours) to be discovered, investigated, fixed and delivered.

Each rule in the set of default rules contains formulas to compute the technical-debt and the annual-interest for each issue. Rules and formulas can be created and customized to better match your teams' needs and habits since it is only raw C# which can be edited in Visual Studio. The main advantage is that **the technical-debt estimation is entirely transparent and easily customizable with CppDepend.**

2. Annual-Interest and Severity

The annual-interest is a measure of an issue severity. The severity and the annual-interest represent the same concept where the annual-interest is a continuous measure while the severity is a discrete measure.

CppDepend defines 5 levels of severity, and the severity of an issue is estimated through thresholds based on the annual-interest.

Low: An issue with a Low severity level represents a small improvement, a way to make the code look more elegant. Default Annual Interest threshold: zero or less than 2 man-minutes per year.

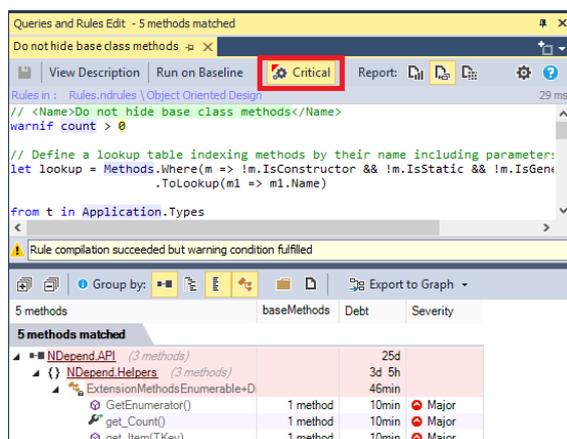
Medium: An issue with a Medium severity level represents a warning for an issue that, even if not fixed, won't have a significant impact on development. Default Annual Interest threshold: less than 20 man-minutes per year.

High: An issue with a High severity level should be fixed quickly, but can wait until the next scheduled interval. Default Annual Interest threshold: less than 2 man-hours per year.

Critical: An issue with a Critical severity level should not move to production. It still can for business imperative needs purposes, but at worth it must be fixed during the next iterations. Default Annual Interest threshold: less than 10 man-hours per year.

Blocker: An issue with a Blocker severity level cannot move to production, it must be fixed. Default Annual Interest threshold: more than 10 man-hours per year.

Notice that the notion of critical issue is different from the notion of critical rule. The severity of an issue is not related to its rule being critical or not. A rule can be tagged as critical to enforce some constraint on it, like for example a quality gate that fails upon critical rule violation can be written.



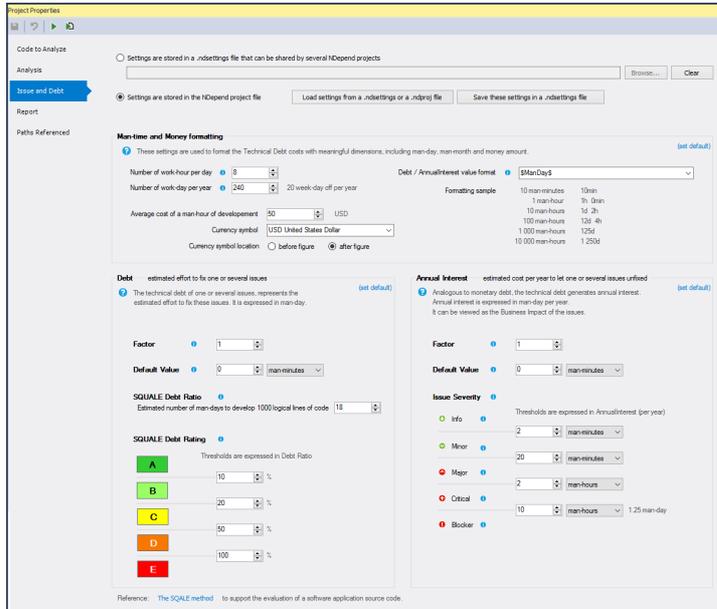
The screenshot shows the CppDepend interface. At the top, a window titled "Queries and Rules Edit - 5 methods matched" displays a rule configuration. The rule is named "Do not hide base class methods" and is marked as "Critical". The rule text includes a warning condition and a lookup table for methods. Below the rule editor, a message states "Rule compilation succeeded but warning condition fulfilled".

At the bottom, a table displays the results of the rule. The table has columns for "baseMethods", "Debt", and "Severity". The results are as follows:

baseMethods	Debt	Severity
5 methods		
5 methods matched		
NDepend API (3 methods)	25d	
NDepend.Helpers (3 methods)	3d 5h	
ExtensionMethodsEnumerable+D	46min	
GetEnumerator()	1 method 10min	Major
get_Count()	1 method 10min	Major
get_Item(TKey)	1 method 10min	Major

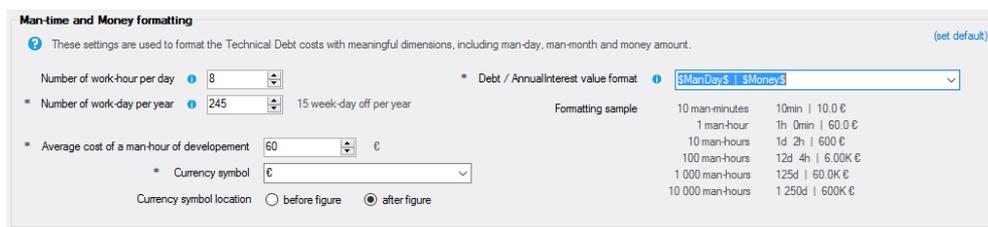
3. Debt Settings

Technical-debt computation and results can be fine-tuned through the settings in the panel *CppDepend > Project Properties > Issues and Debt*.



You can see:

- Thresholds relative to issues severity and annual-interest which were explained in the previous section.
- Thresholds relative to SQALE debt-rating explained in the next section
- Two multiplicative factors that can be applied to all technical-debt and annual-interest estimated values. By default these factors are set to 1.
- To make sure that debt estimations are shown through meaningful man-time measures, settings concerning the number of work-hours per day or number of work-days per year can be adjusted.
- There are also settings to choose how debt values are formatted and to convert man-time debt values into money cost debt values.



4. SQALE Debt Ratio and Debt Rating

The SQALE method (commonly pronounced “scale”) is a standardized way to assess the technical-debt. CppDepend implements the Debt Ratio and the Debt Rating that are part of the SQALE method.

The Debt Ratio on a code base, or on a code element, is expressed in percentage of the estimated technical-debt, compared to the estimated effort it would take to rewrite the code element from scratch. The estimated effort it would take to rewrite the code element from scratch is inferred from the code element size in lines of code, and from the debt setting named Estimated number of man-days to develop 1.000 logical lines of code (see the screenshot in the previous section about debt settings).

The value of the Estimated number of man-days to develop 1.000 logical lines of code setting is just an estimation so in the short-term it is meaningless. After a few man-months or even man-years of development this value is typically stable enough to rely on for estimation purposes. This estimated setting also needs to take into account the cost of writing unit-tests. The default value is 18 man-days which represents an average of 55 new logical lines of code, 100% covered by unit-tests, written per day, per developer.

The Debt Rating of a code base or of a code element is inferred from thresholds applied on the Debt Ratio. The Debt Rating is in the range **A**, **B**, **C**, **D**, **E**. The four thresholds are customizable in the debt settings panel (see the screenshot in the previous section about debt settings). The default thresholds are:

- [0, 5% [of Debt Ratio leads to an **A** debt rating.
- [5%, 10% [of Debt Ratio leads to a **B** debt rating.
- [10%, 20% [of Debt Ratio leads to a **C** debt rating.
- [20%, 50% [of Debt Ratio leads to a **D** debt rating.
- 50% or more of Debt Ratio lead to an **E** debt rating.

The code base Debt Rating and Debt Ratio values are shown in the Dashboard. In the section **Browsing the Technical-Debt** we'll show that simple C# code queries can display the Debt Ratio and Rating values for any code element.

Project Name: **NHibernate 4.0.4** v4.0.0.4000
 Analysis Date: **Today 15:22** most recent
 Analyzed by NDepend v2017.1.0.8903

Baseline: Project Name: **NHibernate 3.4** v3.4.0.0
 Analysis Date: **Today 15:21** most recent
 Analyzed by NDepend v2017.1.0.8903

Choose Baseline 1mn 2mn 1d 2d 3d 4d 5d 7d 14d 30d 60d 87d any define

Lines of Code
73 905 ↘ -732
 15 386 (NotMyCode) ↗ +67
 Estimated Dev Effort 1 940d ↘ -12.37d

Debt
19.01% ↗ from 18.14%
 Rating **C** 174d effort to reach **B**
 Debt 368d ↗ +14d 5h
 Annual Interest 221d ↗ +69d
 Breaking Point 20m ↘ -8m
 Explore Debt ▾

Quality Gates

Fail	6	+1
Warn	2	
Pass	3	

Types
2 558 ↗ +8
 2 Assemblies no diff
 100 Namespaces no diff
 22 277 Methods no diff
 6 285 Fields ↘ -71
 1 829 Source Files ↘ -2
 2 254 Third-Party Elements ↗ +52

Coverage
76.25% ↗ from 75.25%
 56 349 Lines of Code Covered ↗ +187
 17 556 Lines of Code Not Covered ↘ -919
 0 Lines of Code Uncoverable no diff

Rules

Critical	13	
Violated	103	
Ok	40	

Comment
34.85% ↗ from 34.81%
 39 530 Lines of Comment ↘ -316

Method Complexity
 192 Max no diff
 1.97 Average ↘ -0.0068

Issues

All	19 098	+3 111	-1 890
Blocker	0		
Critical	11	+5	-2
Major	2 416	+1 578	-93
Minor	14 364	+1 355	-1 640
Info	2 307	+173	-155

Group issues by rules ▾

5. Prioritizing issues fix and the Breaking-Point metric

The Breaking-Point of an issue or of a set of issues, is the time point from now to when the estimated cost-to-fix the issue(s) will reach the estimated cost to leave the issue(s) unfixed.

The breaking point is the debt divided by the annual-interest. For example if the estimated cost-to-fix the debt is equal to 10 man-days and the estimated annual-interest is equal to 2 man-days per year, then the breaking point is equal to 5 years from now.

Notice that a breaking point which is lower than a year means that during the next 12 months, it is estimated that it would be cheaper to fix the debt than not to fix it.

Notice also that a breaking point is not measured through man-time like debt or annual-interest (a man-month or a man-year), but rather through regular duration (months or years). Breaking point values are typed with TimeSpan.

When it comes to prioritizing issues to fix first, the issue severity is an important parameter. As a reminder: the severity is the discrete measure of the annual-interest. Hence the higher the annual-interest, the more important it is to fix.

However, given a certain severity level, not all issues are equal. Some will demand more effort to fix. This is estimated through the technical-debt measure. Hence, to estimate the Return On Investment (ROI) of an issue fix, it makes sense to estimate the debt divided by the annual-interest. This estimation is the breaking-point for which the lower the value, the higher the ROI.

Let's specify that in the set of default rules, issues that are relative to new problems since the baseline, such as API breaking changes, code elements quality getting even worse, new code elements not tested... are issues which produce a higher annual-interest and thus a higher severity than the other issues. This complies with the best practice to fix recently introduced issues first.

Queries and Rules Edit - 18 422 issues matched

Issues to Fix Priority

View Description | Run on Baseline | Critical | Report: [Icons] | 54 ms

Rules in: Rules.ndrules | Hot Spots

```

// <Name>Issues to Fix Priority</Name>
From i In Issues
where i.BreakingPoint > TimeSpan.Zero
orderby i.BreakingPoint.TotalMinutes ascending
select new { i,
    Debt = i.Debt,
    AnnualInterest = i.AnnualInterest,
    BreakingPoint = i.BreakingPoint,
    CodeElement = i.CodeElement
}

```

18 422 issues

	Debt	Annual Interest	Breaking Point	CodeElement
18 422 issues matched				
■ NHibernate (17 199 issues)	367d	217d	618d	
■ Mark assemblies with ComVisible	5min	2min 0s	912d	■ NHibernate
{} NHibernate.Cfg (413 issues)	13d 7h	8d 1h	619d	
⚡ SettingsFactory (20 issues)	1d 1h	1d 6h	234d	
● Avoid namespaces mutually dependent	15min	5h 42min	15d	⚡ SettingsFactory
● Avoid namespaces mutually dependent	15min	1h 42min	53d	⚡ SettingsFactory
● Avoid namespaces mutually dependent	15min	1h 37min	55d	⚡ SettingsFactory
● Avoid namespaces mutually dependent	15min	55min	99d	⚡ SettingsFactory
● Avoid namespaces mutually dependent	20min	47min	154d	⚡ SettingsFactory
● Avoid namespaces mutually dependent	25min	33min	270d	⚡ SettingsFactory
● Avoid namespaces mutually dependent	30min	33min	329d	⚡ SettingsFactory
⚡ CreateBatcherFactory(IDictionary<String,Stri	29min	35min	299d	
● Avoid namespaces mutually dependent	15min	33min	162d	⚡ CreateBatcherFactory(IDictionary<S
● Code should be tested	14min	2min 0s	2 611d	⚡ CreateBatcherFactory(IDictionary<S
⚡ BuildSettings(IDictionary<String,String>) (5	5h 56min	2h 5min	1 035d	
● Avoid namespaces mutually dependent	15min	33min	164d	⚡ BuildSettings(IDictionary<String,Stri
● Avoid making large methods even large	15min	26min	210d	⚡ BuildSettings(IDictionary<String,Stri
● Avoid methods potentially poorly commen	13min	8min	547d	⚡ BuildSettings(IDictionary<String,Stri
● Avoid methods with too many local variab	2h 50min	45min	1 367d	⚡ BuildSettings(IDictionary<String,Stri
● Code should be tested	2h 22min	11min	4 374d	⚡ BuildSettings(IDictionary<String,Stri
⚡ log (1 issue)	3min 0s	2min 0s	547d	
● Static fields naming convention	3min 0s	2min 0s	547d	⚡ log
⚡ CreateCacheProvider(IDictionary<String,Strin	6min	2min 0s	1 176d	
● Code should be tested	6min	2min 0s	1 176d	⚡ CreateCacheProvider(IDictionary<S
⚡ CreateQueryTranslatorFactory(IDictionary<S	6min	2min 0s	1 176d	
● Code should be tested	6min	2min 0s	1 176d	⚡ CreateQueryTranslatorFactory(IDic

6. Browsing the Technical-Debt

In the introduction we saw that code rules are implemented through C# LINQ queries and we also saw that the debt and annual-interest estimations are inferred from formulas embedded in these LINQ queries.

This C# LINQ queries scheme goes further and can be used to browse and explore the technical-debt. The domain Issues is an enumerable of all issues found in the code base. Obviously, queries that rely on this domain are executed after all rules have been executed.

For example, when clicking a number of issues on the dashboard, like new major issues since baseline in the example below, a code query is generated to list relevant issues. Notice that the issues can be grouped per rules or per code elements. In the screenshot below issues are grouped per rule.

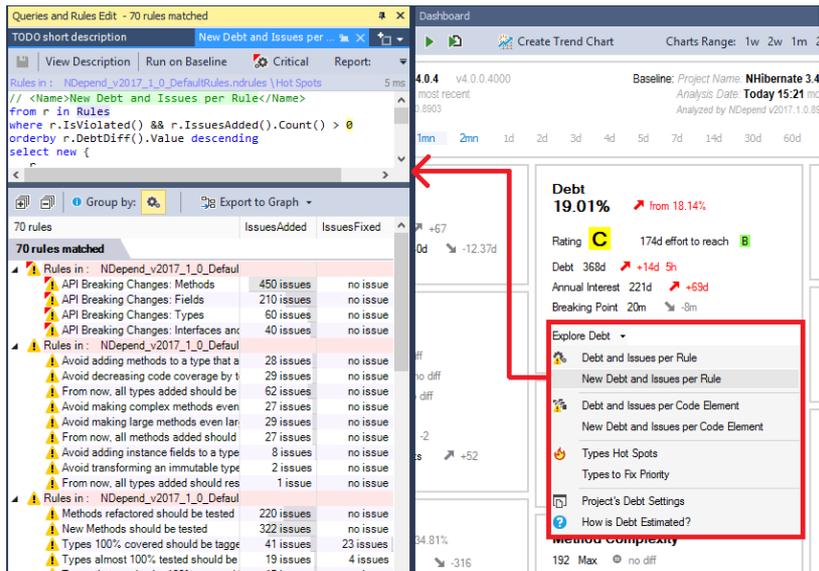
The screenshot displays the CppDepend dashboard for a project named 'NHibernate 3.4'. The dashboard includes several key metrics:

- Debt:** 9.01% (up from 18.14%)
- Coverage:** 76.25% (up from 75.25%)
- Quality Gates:** 6 Fail, 2 Warn, 3 Pass
- Rules:** 13 Critical, 103 Violated, 40 Ok
- Issues:** 19,098 total (111 new, -1,890 old)

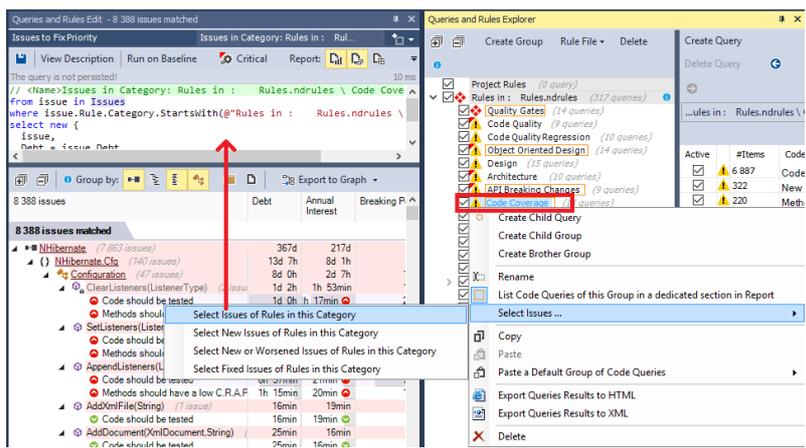
The 'Issues' section is expanded to show a list of issues grouped by rule. A red box highlights the 'Group issues by rules' dropdown menu. The list of issues includes:

Rule	Count	Change
All	19,098	+111 -1,890
Blocker	0	
Critical	11	+5 -2
Major	2,415	+1,578 -83
Minor	14,364	+1,355 -1,640
Info	2,307	+173 -155

Notice the Explore Debt menu on the Dashboard that generate some queries on the rules, issues and code elements to explore in-depth the technical debt.



Right-clicking a *Rules category*, like the **Code Coverage** category here, shows menus to query issues in this category:



Some default debt and issues queries can be found in the **Hot Spots group**. For example the query Types Hot Spots lists the types with most debt first.

Queries and Rules Edit - 105 rules matched

Issues to Fix Priority Debt and Issues per Rule

View Description Run on Baseline Critical Report

```

Rules in : Rules.ndrules \ Hot Spots
// <Name>Debt and Issues per Rule</Name>
from r in Rules
where r.IsViolated()
orderby r.Debt().Value descending
select new {
    r,
    Issues = r.Issues(),
    Debt = r.Debt(),
    AnnualInterest = r.AnnualInterest(),
    BreakingPoint = r.BreakingPoint(),
    Category = r.Category
}

```

Group by: Export to Graph

105 rules	Issues	Debt	Annual Interest	Breaking Point
105 rules matched				
Rules in : Rules.ndrules \ Code Coverage (9 rules)		141d	70d	731d
Code should be tested	6 887 issues	128d	40d	1 146d
Types 100% covered should be tagged with Fu	552 issues	3d 3h	0min 0s	00:00:00
Methods should have a low C.R.A.P score	76 issues	2d 4h	3d 1h	298d
Methods refactored should be tested	220 issues	2d 1h	9d 1h	86d
New Methods should be tested	322 issues	2d 0h	13d 3h	55d
Assemblies Namespaces and Types should be	239 issues	1d 7h	0min 0s	00:00:00
Types almost 100% tested should be 100% tes	73 issues	3h 28min	3d 0h	52d
Types that used to be 100% covered by tests s	15 issues	40min	5h 0min	49d
Namespaces almost 100% tested should be 10	4 issues	28min	1h 20min	128d
Rules in : Rules.ndrules \ Architecture (4 rules)		49d	37d	488d
Avoid namespaces mutually dependent	1 304 issues	48d	36d	481d
Avoid namespaces dependency cycles	2 issues	4h 0min	40min	2 190d
Namespaces with poor cohesion (RelationalCo	13 issues	2h 10min	0min 0s	00:00:00
Assemblies with poor cohesion (RelationalCoh	2 issues	20min	0min 0s	00:00:00
Rules in : Rules.ndrules \ Code Quality (9 rules)		103d	22d	1 712d
Avoid types with too many methods	129 issues	33d	4d 3h	2 778d
Avoid methods too big, too complex	74 issues	24d	1d 5h	5 226d
Avoid methods with too many parameters	75 issues	11d 5h	1d 1h	3 446d
Avoid types too big	40 issues	9d 1h	1d 1h	2 943d
Avoid types with too many fields	36 issues	7d 5h	6h 51min	3 294d
Avoid types with poor cohesion	62 issues	7d 0h	2d 3h	1 079d
Avoid methods with too many local variables	25 issues	5d 1h	7h 22min	2 061d
Avoid methods potentially poorly commented	288 issues	2d 0h	7d 2h	104d
Avoid methods with too many overloads	499 issues	2d 0h	2d 0h	365d
Rules in : Rules.ndrules \ Object Oriented Design		30d	18d 4h	598d
Avoid interfaces too big	67 issues	12d 7h	3d 2h	1 443d
Overrides of Method() should call base Method	730 issues	7d 4h	3d 0h	912d
Sum	19 947	387d	236d	73 24d

The baseline plays a major role when it comes to exploring the issues set because new or fixed issues since the baseline assess the quality of recent work.

Per default the baseline is the historic analysis result closest to 30 days ago and per default, a historic analysis result is persisted at most every day.

Because when assessing recent work quality, one will certainly want to juggle between yesterday, last week and last month baselines, the CppDepend dashboard allows you to apply a temporary baseline with a single click. The debt and issues set is then recomputed within a few seconds accordingly.

Choose Baseline 1d 2d 3d 4d 5d 6d 17d 30d 58d 86d 111d any

And since assessing issues and debt since the baseline is important as we just saw, all Hot Spots default queries come with a since baseline version. For example here is a query to assess New Debt and Issues per Rule since the baseline.

The screenshot shows a Visual Studio window titled "Queries and Rules Edit - 70 rules matched". The main pane displays a LINQ query:

```

Rules in : Rules.ndrules \ Hot Spots
// <Name>New Debt and Issues per Rule</Name>
from r in Rules
where r.IsViolated() && r.IssuesAdded().Count() > 0
orderby r.DebtDiff().Value descending
select new {
    r,
    IssuesAdded = r.IssuesAdded(),
    IssuesFixed = r.IssuesFixed(),
    Issues = r.Issues(),
    Debt = r.Debt(),
    DebtDiff = r.DebtDiff(),
    Category = r.Category
}

```

Below the query, a table shows the results for 70 rules. The table has columns: IssuesAdded, IssuesFixed, Issues, Debt, DebtDiff, and Category. One rule is highlighted in blue:

Rule	IssuesAdded	IssuesFixed	Issues	Debt	DebtDiff	Category
Types almost 100% tested should be 100% tes	19 issues	4 issues	73 issues	3h 28min	46min	Rules in :

A second window shows a detailed view of 19 issues. The table has columns: Debt and Annual Interest. Issues are listed with their parent elements and associated debt/interest values.

Issue	Debt	Annual Interest
NHibernate (19 issues)	367d	217d
NHibernate.Event.Default (1 issue)	3d 6h	2d 0h
DefaultLoadEventListener (1 issue)	1h 28min	59min
Types almost 100% tested should be 100	8min	20min
NHibernate.Loader.Hql (1 issue)	5h 21min	4h 41min
QueryLoader (1 issue)	5h 9min	4h 21min
Types almost 100% tested should be 100	5min	20min
NHibernate.Util (2 issues)	8d 3h	10d 2h
ArrayHelper (1 issue)	4h 33min	1d 0h
Types almost 100% tested should be 100	4min 50s	20min
CollectionPrinter (1 issue)	35min	1h 22min
Types almost 100% tested should be 100	2min 0s	20min
NHibernate.Engine.Loading (2 issues)	2h 42min	2h 8min
CollectionLoadContext (1 issue)	1h 18min	1h 3min
Types almost 100% tested should be 100	6min	20min

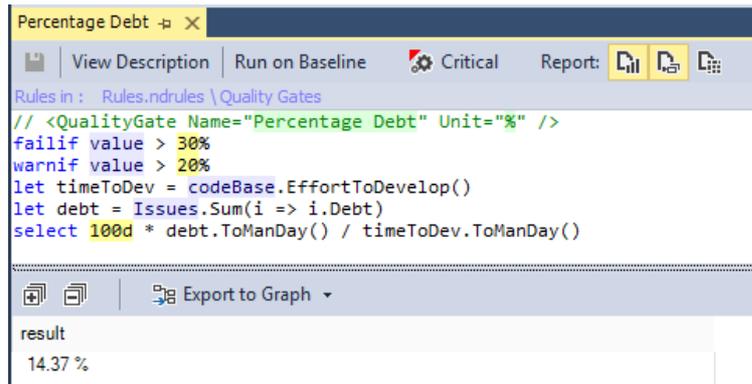
Let's mention a subtlety when it comes to debt and issues querying. Types contain methods and fields, namespaces contain types and assemblies contain namespaces. Hence types, namespaces and assemblies are code element parents.

All issues-related ICodeElement extension methods like `elem.Debt()`, `elem.AnnualInterest()`, `elem.Issues()`, have a version prefixed with `All` that returns the debt and issues for the code element parent and all its child elements. Hence:

- `elem.AllIssues()` returns an enumerable of issues in the code element parent and issues on its child code elements. Sometime in the product we use the terminology cumulated issues of a code element parent like an assembly, a namespace or a type.
- `elem.AllDebt()` returns the estimated summed debt for the code element parent and its child code elements.
- `elem.AllAnnualInterest()` returns the estimated summed annual-interest for the code element parent and its child code elements.
- `elem.AllBreakingPoint()` returns the estimated breaking-point for the code element parent and its child code elements.

7. Technical Debt and Quality Gate

You'll find default quality gates relative to technical debt and issues, including Percentage of Debt, New Debt since Baseline or New Blocker / Critical / Major Issues. Quality gates relative to absolute technical debt value are disabled by default because the proper thresholds can only be defined in the context of a particular project.



The screenshot shows a web-based interface for configuring a quality gate. At the top, there's a title bar 'Percentage Debt' with a close button. Below it are navigation buttons: 'View Description', 'Run on Baseline', 'Critical' (with a red flag icon), and 'Report:' (with three icons). The main area displays the rule configuration in a code editor. The code is as follows:

```
// <QualityGate Name="Percentage Debt" Unit="%" />
failif value > 30%
warnif value > 20%
let timeToDev = codeBase.EffortToDevelop()
let debt = Issues.Sum(i => i.Debt)
select 100d * debt.ToManDay() / timeToDev.ToManDay()
```

Below the code editor, there's a section for the result. It shows a 'result' label and a value of '14.37%'. There are also icons for 'Export to Graph'.

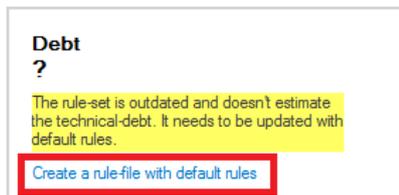
The same way Issues and Rules are predefined as queryable domains that provide an enumerable of issues or rules, the domain QualityGates is an enumerable of quality gates. The default query below estimates the quality gates trend since the baseline. Notice that quality gates that rely on the baseline (like New Debt since Baseline) have neither a value nor a status defined on the baseline.

8.Reasons why Technical Debt might be Zero or incomplete

My technical-debt estimation shows zero or ?:

If the technical debt is zero or ?, you are likely analyzing a project created with an older version of CppDepend (v6 or lower). The rules-set of previous CppDepend versions didn't have debt formulas, and hence per default issues with no debt formulas have a zero debt.

In the Dashboard > Debt panel you should see a link named Create a rule-file with default rules.



Clicking this link will automatically create a rule-file that contains all new default rules, the ones with debt estimation formulas. Once done, it is recommended to replace the actual project rules with the rules that estimate the technical debt. To do so, drag&drop can be used from the Queries and Rules Explorer panel (both for rules and for group of rules). Notice that debt formulas provoke rule compilation errors when read by lower versions of {0} (v6 and lower). If you plan to use this project from CppDepend v6 or lower, please clone it first.

For customized rules, we recommend to modify their source code to write custom debt estimation formulas.

Finally, please note that the default rules file will be created in the same directory than the project file and will be attached to the project with a relative file path. This path can be edited from the CppDepend Project Properties > Paths Referenced.

My technical-debt estimation is incomplete because no code coverage data provided:

Code not tested, or partially tested by unit-tests, represents a large source of technical-debt. Actually each line of code left uncovered by tests contributes to the technical debt. This is why the Debt section of the dashboard shows a warning message when code coverage files import is not setup in the CppDepend project.

Debt
11.83% no diff

Rating **C** 6d 3h effort to reach **B**

Debt 41d no diff

The technical-debt is incomplete because no coverage data specified.

Explore Debt ▾

Coverage
 N/A because no coverage data specified

[Import Code Coverage Data](#)

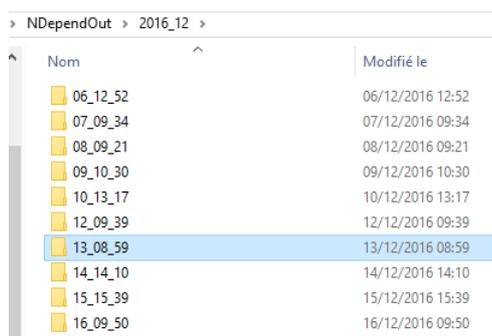
Code coverage data not available on the baseline:

When code coverage is available in the current analysis result but is not available in the baseline analysis result, rules related to code coverage don't produce issues. Indeed, in this situation coverage issues cannot be estimated on baseline and all coverage issues would then appear as new issues.

Often this situation appears when a project has been created and the first analysis result obtained doesn't contain coverage data. In the CppDepend project the default baseline setting is to choose the baseline analysis result closer to obtained 30 days ago, so this problem might persist for a month.

Typically to fix this situation, we advise to get rid of history analysis result(s) that don't have code coverage data. To do so you need to open the folder that contains History Analysis Result defined in CppDepend Project Properties > Analysis > Baseline for Comparison > Historic Analysis Results (per default set to the project output folder). Then identify the folder that contains the history analysis result to remove and just delete the folder.

For example in the screenshot below, the selected folder represents the History Analysis Result obtained on the 13th of December 2016, 8:59 AM.



We understand that this manual folder tweak is not the optimal way to solve such situation. If you'd like us to provide a UI that would list History Analysis Results, that would show which ones doesn't have coverage data (or others flaws like source code not resolved), and that would let remove them.

We could also provide a filter at analysis time that would not persist an analysis result as history if it doesn't satisfy certain criteria (like coverage data available...).