VECTOR >

Software Analytics Handbook

Squore 20.0.12

Table of Contents

Preface

Foreword

This edition of the Software Analytics Handbook was released by Vector Informatik GmbH.

It is part of the user documentation of the Squore software product edited and distributed by Vector Informatik GmbH.

For information on how to use and configure Squore, the full suite of manuals includes:

User Manual	Target Audience
Squore Installation Checklist	New users before their first installation
Squore Installation and Administration Guide	IT personnel and Squore administrators
Squore Getting Started Guide	End users, new users wanting to discover Squore features
Squore Command Line Interface	Continuous Integration Managers
Squore Configuration Guide	Squore configuration maintainers, Quality Assurance personnel
Squore Eclipse Plugin Guide	Eclipse IDE users
Squore Reference Manual	End Users, Squore configuration maintainers
Squore API Guide	End Users, Continuous Integration Managers
Squore Software Analytics Handbook	End Users, Quality Assurance personnel

> You can also use the online help from any page when using the Squore web interface by clicking **? > Help**.

Licence

No part of this publication may be reproduced, transmitted, stored in a retrieval system, nor translated into any human or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of the copyright owner, Vector Informatik GmbH. Vector Informatik GmbH reserves the right to revise this publication and to make changes from time to time without obligation to notify authorised users of such changes. Consult Vector Informatik GmbH to determine whether any such changes have been made. The terms and conditions governing the licensing of Vector Informatik GmbH software consist solely of those set forth in the written contracts between Vector Informatik GmbH and its customers. All third-party products are trademarks or registered trademarks of their respective companies.

Warranty

Vector Informatik GmbH makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Vector Informatik GmbH shall not be liable for errors contained herein nor for incidental or consequential damages in connection with the furnishing, performance or use of this material.

This edition of the Software Analytics Handbook applies to Squore 20.0.12 and to all subsequent releases and modifications until otherwise indicated in new editions.

Responsabilities

Approval of this version of the document and any further updates are the responsibility of Vector Informatik GmbH.

Contacting Vector Informatik GmbH Product Support

If the information provided in this manual is erroneous or inaccurate, or if you encounter problems during your installation, contact Vector Informatik GmbH Product Support: https://portal.vector.com/

You will need a valid customer account to submit a support request. You can create an account on the support website if you do not have one already.

For any communication:

- **support@vector.com**
- **Vector Informatik GmbH Product Support**

    Vector Informatik GmbH - Holderäckerstr. 36 / 70499 Stuttgart - Germany

Getting the Latest Version of this Manual

The version of this manual included in your Squore installation may have been updated. If you would like to check for updated user guides, consult the Vector Informatik GmbH documentation site to consult or download the latest Squore manuals at https://support.squoring.com/documentation/latest. Manuals are constantly updated and published as soon as they are available.
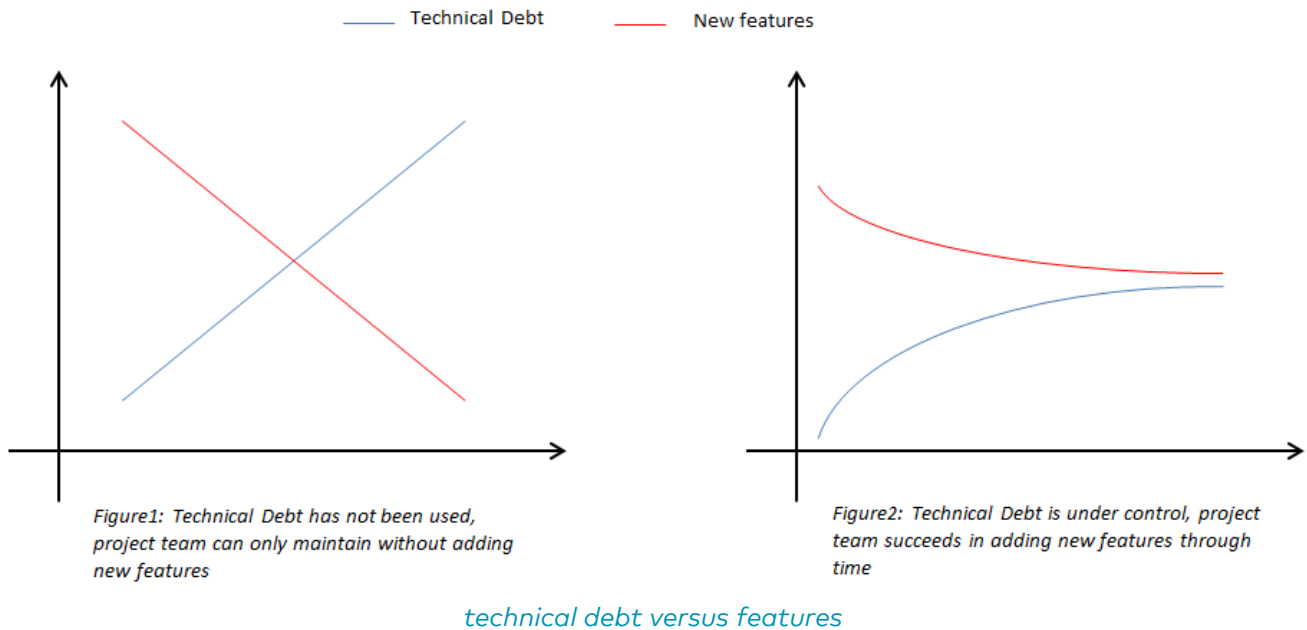
Chapter 1. Technical Debt

Concept

Technical Debt is used to address the non-quality of a software project. It is evaluated in time unit (man day), or actual cost (monetary value), based on the remediation cost of all default which have been found in the project.

Technical Debt represents the effort to fix all quality issues in the project.

Utility

Using Technical Debt helps anticipate future issues the development team will face. The more debt a project has, the more difficult the development will be. High debt often reveals a low ratio of "new features", as team spends its time in understanding and fixing current issues.

Figure1: Technical Debt has not been used, project team can only maintain without adding new features

Figure2: Technical Debt is under control, project team succeeds in adding new features through time

technical debt versus features

Technical Debt = Maintainability?

No, Technical Debt concept is not limited to Maintainability, and can be extended to other ISO quality characteristics: maintainability, portability, reliability, security, efficiency.

However, quality issues other than Maintainability are most of the time addressed during development because they are part of the delivery requirements (ie: no reliability issue shall be detected in the delivered product...).

Maintainability is often considered as rather "Nice to Have" than "Mandatory". As a consequence, technical debt is (wrongly) associated to Maintainability.

Computation

Technical Debt computation is based on quality issues provided by Static Code Analysis. Each issue is associated to a remediation cost, mapped to a time effort.

Practice ⇕	Occ. ⇕	Delta ⇕	Data Provider ⇕	ISO Characteristic ▾	Nature ▾	Remediation Cost ▾	Severity ▾
▸ Assignment in Boolean	7	+7	Squan Sources	Maintainability	Non Conformity	Low	Minor
▸ Missing compound if	40	+26	Squan Sources	Maintainability	Non Conformity	Tiny	Minor
▸ Missing compound statement	15	+10	Squan Sources	Maintainability	Non Conformity	Tiny	Minor
▸ Avoid Duplicated Blocks in Function	6	+6	Squan Sources	Maintainability	Cloning	Low	Major
▸ IO Functions shall not be used	1	0	Squan Sources	Maintainability	Risky Construction	Medium	Major
▸ 'atof, atoi or atol' shall not be used	1	0	Squan Sources	Maintainability	Risky Construction	Medium	Major
▸ Common realloc mistake: 'varname' nulled but not freed upon failure	5	0	Cppcheck	Performance efficiency	Risky Construction	Medium	Blocking
▸ Dynamic Memory Allocation shall not be used	2	0	Squan Sources	Performance efficiency	Risky Construction	Medium	Major
▸ Missing final else	1	+1	Squan Sources	Reliability	Non Conformity	Low	Minor

Each issue is mapped to a ISO characteristic

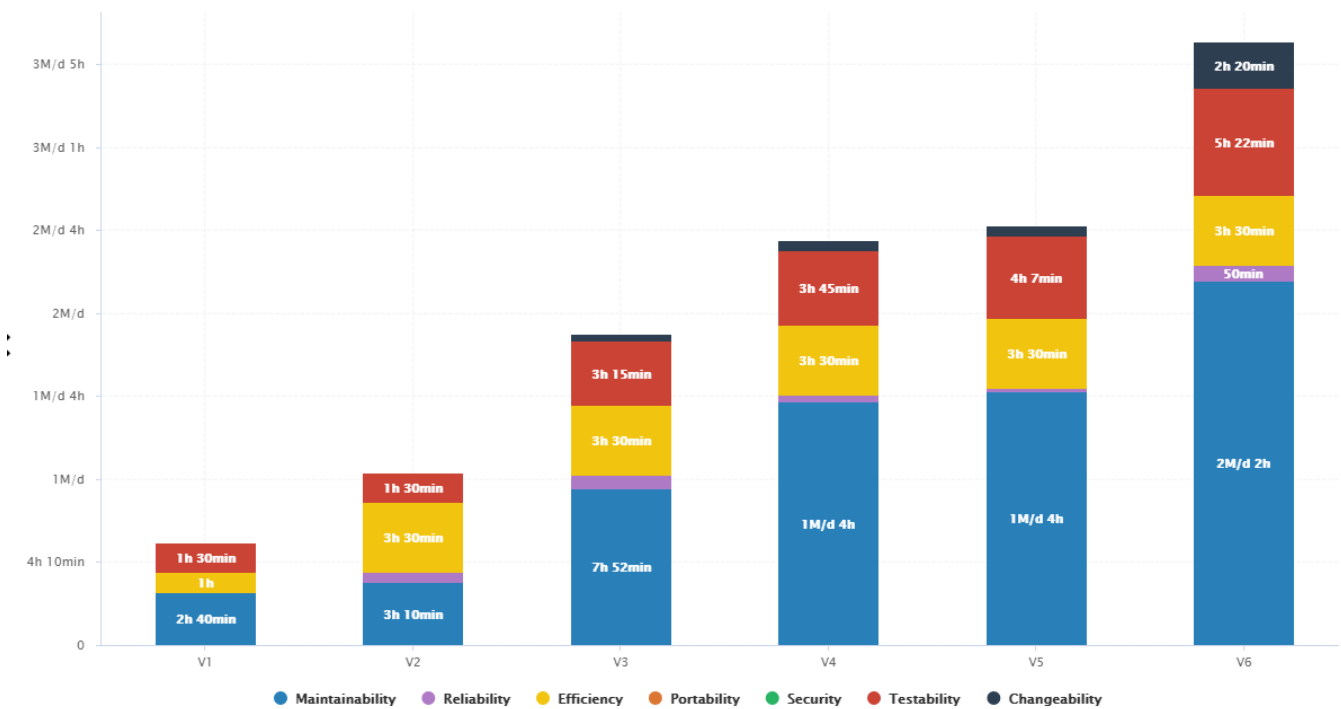Remediation Cost is converted into time effort

Technical Debt computation

Formula:

sum_(All_issues)=(#violations * remediation cost)

Remediation Cost:

Remediation Cost	Minutes
Low	10
Medium	30
High	60
Huge	480

ℹ Squore uses the ISO mapping to provide Technical Debt trend according quality breakdown.

*Technical Debt trend according to ISO characteristics*

Technical Debt is based on the findings provided by the static analyzers. Its value may vary depending on the analyzer which is used (in case they do not generate the same amount of issues). Activating a maximum number of programming rules during static code analysis increases the confidence level in the technical debt results.

How to address Technical Debt with Squore

Find the right project

At project portfolio level, Squore provides a Technical Debt chart highlighting the distribution of all projects according their technical debt density (=average of technical debt per 1000 lines of code). Interpretation:

- Projects in the top area have a higher Technical Debt Density meaning they are more difficult to maintain,
- Projects in the right area are bigger (wrt. Line counting),
- The bubble size is proportional to the "number of programing rules which have been activated during static analysis". A larger bubble means a higher confidence in technical debt value.
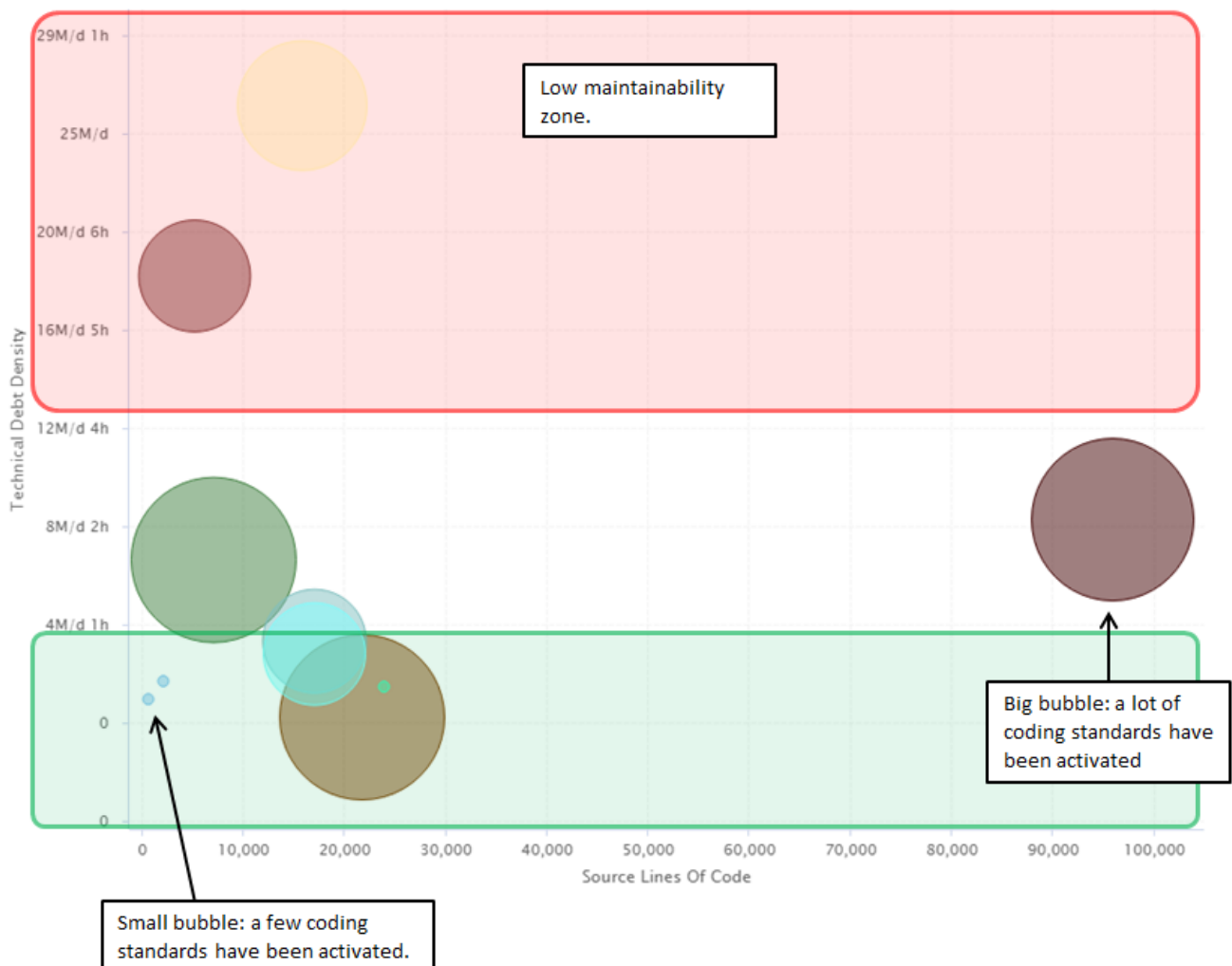

Technical Debt at portfolio level

Find the right function

Inside a project, Squore provides a Technical Debt chart highlighting the distribution of all modules (i.e functions) according their technical debt and their violations density (density of issues weighted by severity). Interpretation:

- Modules in the top area have a higher Technical Debt meaning they are more difficult to maintain,
- Modules in the right area are bigger (wrt. Line counting),
- The bubble color shows the severity of issues inside the components.
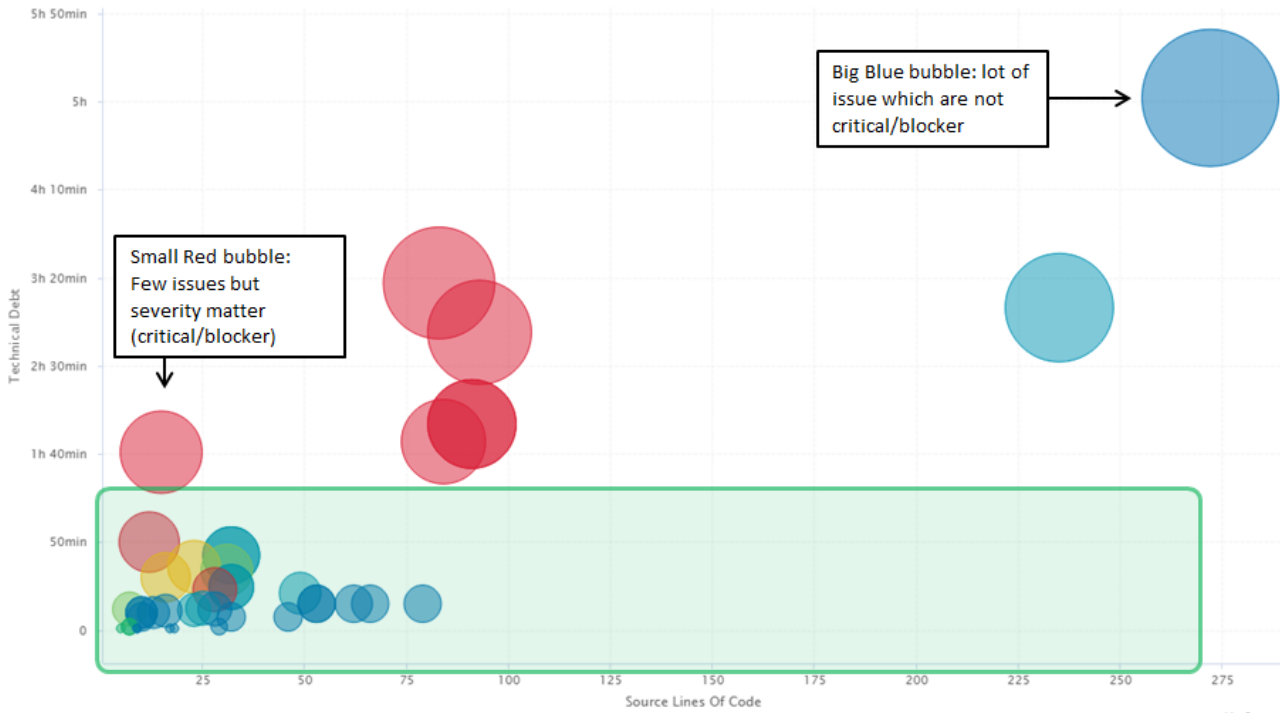
Technical Debt - Module distribution

Chapter 2. Test Strategy

Concept

The Test Strategy intends to help development team increase the reliability confidence of their software project. It is more and more difficult carry out exhaustive testing activities, as the size and complexity of the code grow. With the Test Strategy you can:

1. Reduce the scope of the code which needs to be tested

2. Define code coverage expectations for those component which need to be tested

As a result, Squore provides the Code Coverage Compliance KPI which represents the ratio of components which comply with this Test Strategy.

Settings

To Be Tested

Squore applies an algorithm to include/exclude modules which should be integrated in the Test Strategy. Parameters are:

- Cyclomatic Complexity (VG)
- Nesting Level (LEVL)
- Number of non-cyclic paths (NPAT)
- Vocabulary Frequency (VOCF)
- Code Stability Index (SI)

Using these parameters, the most complex and unstable code are identified. Increasing the test on this code will increase reliability and reduce the risk of delivery. Consequences:

- Modules which are not part of the "to be tested" list will be "ignored" in the Code Coverage Compliance KPI
- Modules which shall be tested will be evaluated according their coverage (Statement, Branch and MCDC) with regards to their safety level (ASIL, SIL …)

Coverage Thresholds

The code coverage thresholds can be tuned according the type of coverage and the safety level. Here are the default settings:

Level	Statement	Branch	MC/DC
Level A	80%	50%	0%
Level B	100%	80%	50%
Level C	100%	100%	80%
Level D	100%	100%	100%

Coverage thresholds

How to address the Test Strategy with Squore

How to setup the parameters

Parameters are available at project creation time. The "To be tested" list parameters are available in the "Test Strategy" section.

Test Strategy - settings

Notes:

- Each of the 'TO BE TESTED' parameters can be disabled (by setting the value '-1')
- It is possible to disable the impact of test strategy on the Code Coverage Compliance KPI. "Code Coverage thresholds" are available in the "Test Coverage Thresholds" section.

Test Strategy - Coverage Thresholds

Note: it is possible to disable a dedicated type of coverage (Statement and/or Branch and/or MCDC). For instance if the team just wants to assess the statement coverage only, Branch and MCDC can be disabled.

How to setup the Safety Level (=Critical Factor)

Squore allows defining the critical factor for every component in the project.

Using the GUI:

- Select an artefact
- Open the Form tab
- Define the critical factor

The critical factor will spread to all "children artefacts" (i.e. all modules will automatically inherit the value of the file). In addition, it is possible to overload an inherited value.
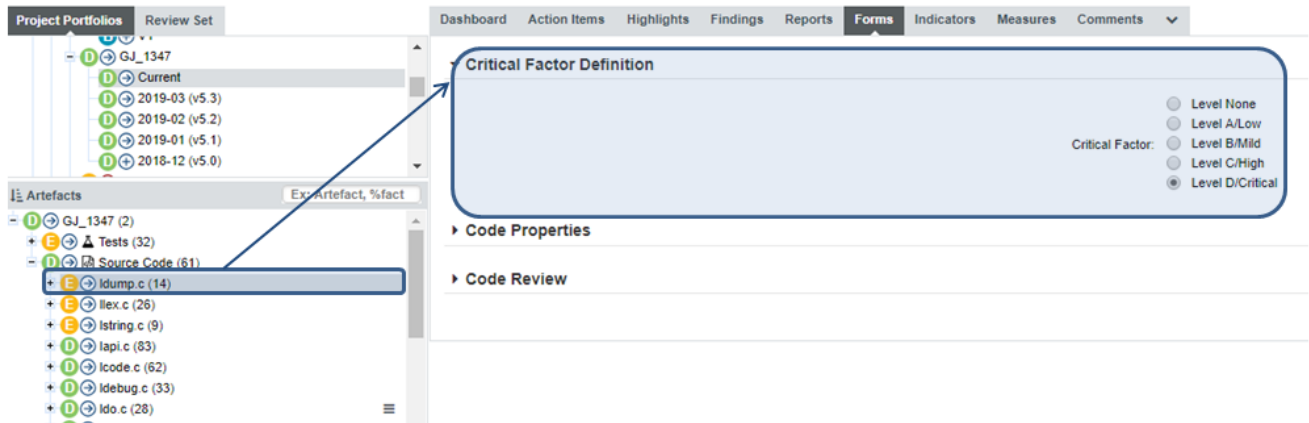
Test Strategy - Safety Level definition

Using a Text/Csv file import:

It is possible to provide a csv file during the project creation which contains the Critical Factor information. Create a csv file as follow (1=level A, 2=level B, 3=Level C, 4=level D):

	A	B
1	Path	CRITICAL_FACTOR
2	core/newutil.c	1
3	core/util.c	1
4	core/util.h	1
5	core/write.c	2
6	core/write.h	2
7	core/control.c	4
8	core/base.c	4
9	core/base.h	4
10	apps/robot.c	2
11	apps/player.c	3
12	apps/player.h	3
13	apps/machine.c	2
14	apps/machine.h	2
15	apps/score.c	3
16	apps/score.h	3
17	apps/master.c	3
18	apps/master.h	3
19		
20		

Critical_Factor_and_code_Sta

Test Strategy - sample of "safety level" csv file

Feed this file to the "csv tag import" data provider:
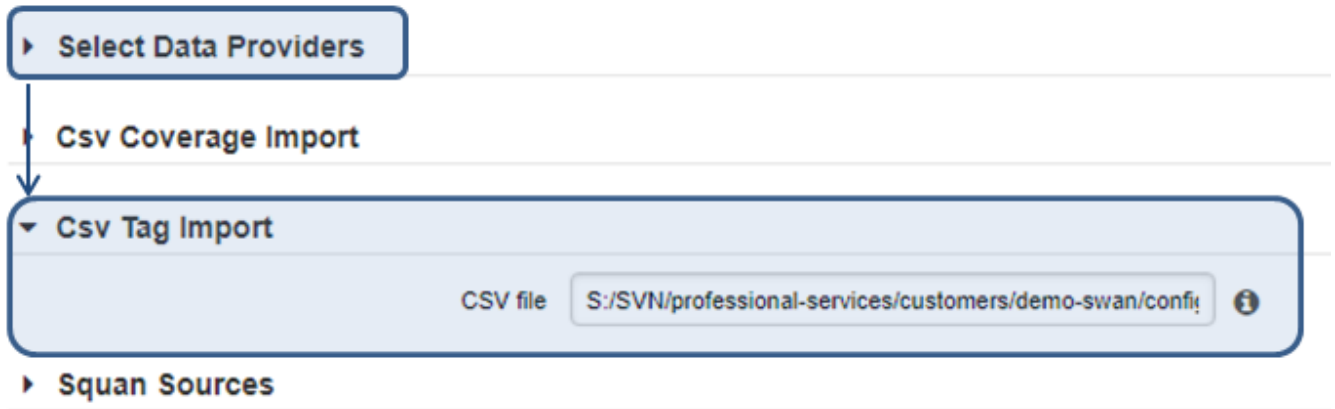
Test Strategy - Safety Level Data Provider

The Code Coverage Compliance Treemap

The treemap highlights components according to:

- Their code size (=size of the treemap zone)
- Their code coverage compliance (=color of the treemap zone)
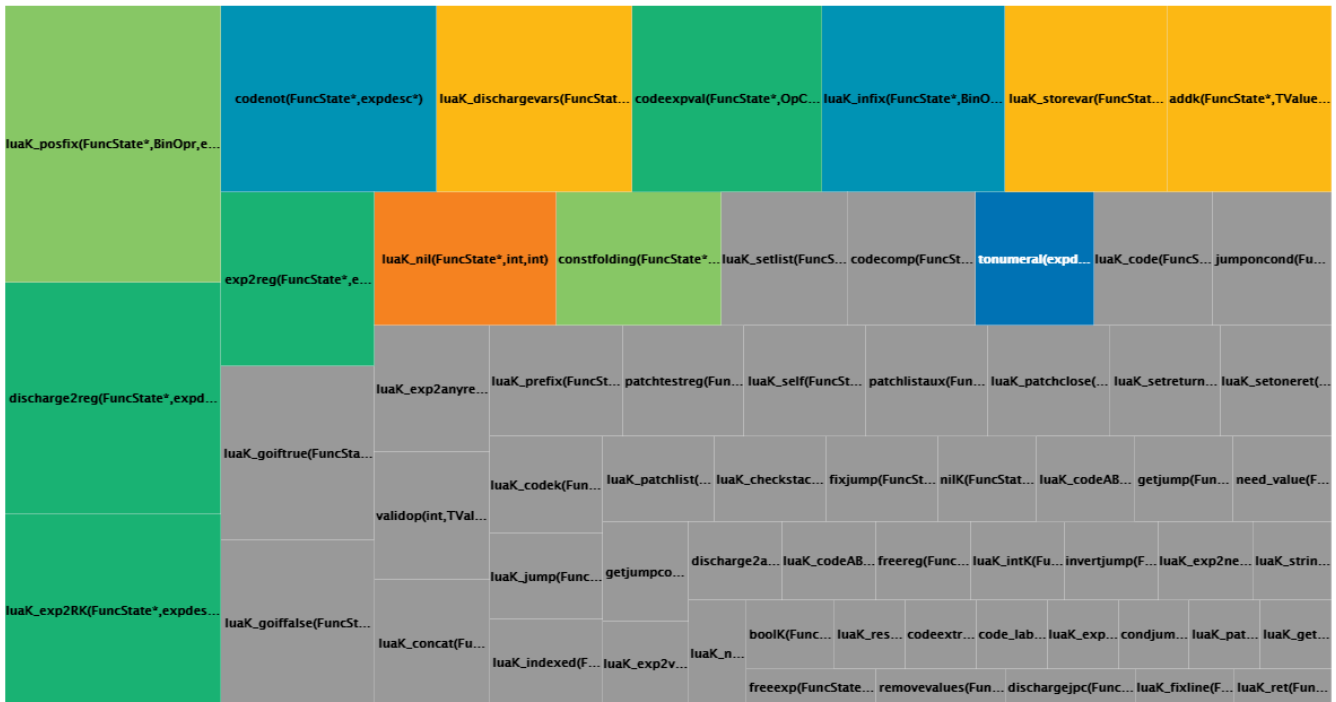
Test Strategy - Coverage Compliance Treemap

Interpretation:

- Grey zone means the module is excluded from the Test Strategy
- The color code indicates how well the component is tested regarding to expectations defined in the "Test Coverage Thresholds" form

- ■ Coverage Objectives are reached: 100%
- ■ 90% <= Coverage Objectives < 100%
- ■ 70% <= Coverage Objectives < 90%
- ■ 50% <= Coverage Objectives < 70%
- ■ 30% <= Coverage Objectives < 50%
- ■ 10% <= Coverage Objectives < 30%
- ■ 0% <= Coverage Objectives < 10%
- ■ Module is out of the Test Strategy (=not To be tested)

Test Strategy - Coverage Compliance scale

Note: Safety level is implicitly taken into account in this evaluation. Whatever the safety level is, Squore highlights the distance to the coverage objectives.

Chapter 3. Rule Compliance

Concept

Rule Compliance indicates how well the project development team follows the coding guidelines. A rule is not compliant if at least one unjustified violation is detected. If a violation is relaxed/justified using the "relaxation feature", it will not impact rule compliance. For this KPI, Squore only takes into account rules which nature is categorized as "Non Conformity".

Practice ⬍	Occ. ⬍	Delta ⬍	Data Provider ⬍	ISO Characteristic ▾	Nature ▾	Remediation Cost ▾	Severity ▾
▸ Assignment in Boolean	7	+7	Squan Sources	Maintainability	Non Conformity	Low	Minor
▸ Missing compound if	40	+26	Squan Sources	Maintainability	Non Conformity	Tiny	Minor
▸ Missing compound statement	15	+10	Squan Sources	Maintainability	Non Conformity	Tiny	Minor
▸ Avoid Duplicated Blocks in Function	6	+6	Squan Sources	Maintainability	Cloning	Low	Major
▸ IO Functions shall not be used	1	0	Squan Sources	Maintainability	Risky Construction	Medium	Major
▸ 'atof, atoi or atol' shall not be used	1	0	Squan Sources	Maintainability	Risky Construction	Medium	Major
▸ Common realloc mistake: 'varname' nulled but not freed upon failure	5	0	Cppcheck	Performance efficiency	Risky Construction	Medium	Blocking
▸ Dynamic Memory Allocation shall not be used	2	0	Squan Sources	Performance efficiency	Risky Construction	Medium	Major
▸ Missing final else	1	+1	Squan Sources	Reliability	Non Conformity	Low	Minor

Total: 78 (+50) in 9 rules

Rule Compliance - Findings

Formula

Rule Compliance = (#violated rules / #Verified Rules)

Where: * "Violated rules" is the number of rules for which there is at least 1 violation * "Verified Rules" is the number of rules which were checked during static code analysis

Side effect of external data provider

The list of "verified rules" depends on the static code analysis and thus, depends on the external tools results which are imported in the Squore project with Data providers.

Example:

1. Using 'Squore Analyzer' Data provider only: Total verified rules is 19 (for C language)
2. Using 'Squore Analyzer' + 'PCLint' Data providers: Total verified rules is 152.

Rule Compliance may drastically vary, depending on the static code analyzers used on the project.

Note: Squore dynamically create the ruleset based on the selected data providers. For instance, MISRA Checker will not be counted in the "verified rules" if no MISRA analyzer are called.

Overall Rating has been improved although the code didn't change

Rule Compliance has been affected by the import of PC Lint.

Rule Compliance - Rating Impact of Data Providers

The reason comes from the "verified rules" which drastically changed:

Rule Compliance - Rating with Squore only

Rule Compliance - Rating with Squore and PCLint

Settings (Ruleset template)

After selection of the data providers, Squore provides an interface to tune the ruleset in order to enable/disable specific rules.

	Active ⇕	Name ▲	Id ⇕	Data Provider ⇕	ISO Characteristic ⇕
		return			
☐	⬤	'class::operator=' should return 'class &'	OPERATOREQ	CPPCHECK	Reliability
☐	⬯*	'operator=' should return reference to self	OPERATOREQRETREFTHIS	CPPCHECK	Reliability
☐	⬤	Consecutive return break continue goto or throw statements are unnecessary.	DUPLICATEBREAK	CPPCHECK	Reliability
☐	⬯*	HIS Metrics: RETURN (Number of Return) shall be uniq	R_HIS_RETURN	SQUORE	
☐	⬤	IDMS Return Code	R_IDMSRETURNCODE	SQUORE	Reliability
☐	⬤	Return of the address of an auto-variable	RETURNADDRESSOFAUTOVARIABLE	CPPCHECK	Reliability
☐	⬤	Return the address of function parameter 'parameter'	RETURNADDRESSOFFUNCTIONPARAMETER	CPPCHECK	Reliability
☐	⬤	Statements following return break continue goto or throw will never be executed.	UNREACHABLECODE	CPPCHECK	Reliability
☐	⬤	Suspicious checking of string::find() return value.	STLIFSTRFIND	CPPCHECK	Reliability

Rule Compliance - Ruleset Template

It is possible to create a template to share this ruleset configuration between projects.

Relaxing a violation

Relaxation inside Squore GUI

It is possible to relax a violation or a group of violations within Squore.

Rule Compliance - Findings Relaxation

Note: Make sure you have selected the "Current" version of the project. Indeed, it is not possible to modify/update a baselined squore version.

Relaxation imported from external tools

Squore can manage justifications which have been provided outside Squore environment. This mechanism is handled by the data provider, which detects justifications from the data it analyzes, and injects them into Squore.

Chapter 4. Violations Density

Concept

The Violations Density intends to highlight the part of the code with the most violated rules (of "Non Conformity" or "Risky Construction" nature). The density takes into account the severity of the issues which are detected: **the density is weighted by the findings severity.** This allows to highlights either:

- Code with blocker/critical issues
- Code with a lot of minor/major issues

In both cases, remediation shall be performed to improve code quality.

Formula

Violation Density = (sum(issue * severity)) / code size)

Where:

- Code size is expressed in KEloc (Effective Lines /1000)
- Severity weights are:

Severity	Weight
Minor	0.1
Major	5
Critical	20
Blocker	100

The rating is based on the following scale:
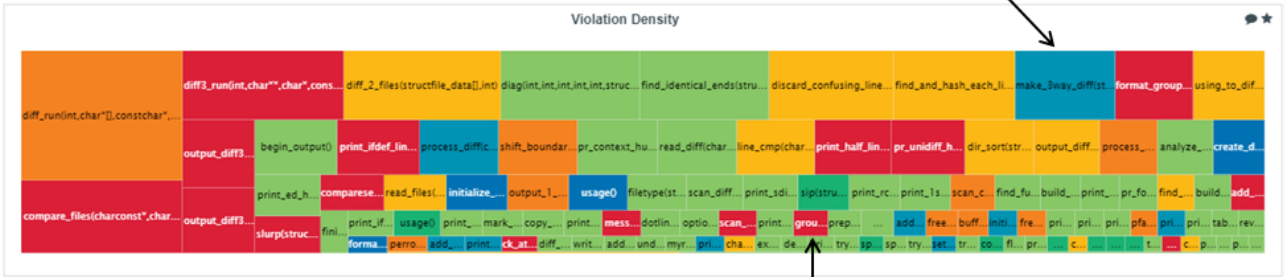

Violations Density - Scale

A component which is rated "G" has in average the equivalent of 10 blocker issues every 1000 lines.

The Violation Density Treemap

The treemap highlights the component according to:

- Their code size (=size of the treemap zone)
- Their violation density (=color of the treemap zone)

Violations Density - Treemap

Chapter 5. Cloning and Duplication

Concept

Cloning and Duplication KPI intends to focus on the code that should be reworked. It basically addresses architecture design and implementation. The KPI is computed thanks to Square Analyzer metrics. These metrics highlight duplicated block and similar algorithms.

Algorithmic Cloning Vs Duplication

Duplication is directly computed from the source code textual analysis. Cloning on the other hand is related to the source code algorithm analysis.

The analyzer can generate both metrics and findings related to cloning and duplication. The quality model embeds 2 indicators: **Inner Cloning** and **Outer Cloning**.

- **Inner (or Internal)** Cloning is related to cloning within the artefact ⇒ **is the artefact well designed?**
- **Outer (or External)** Cloning is related to cloning outside the artefact ⇒ **is the artefact cloned with another artefact?**
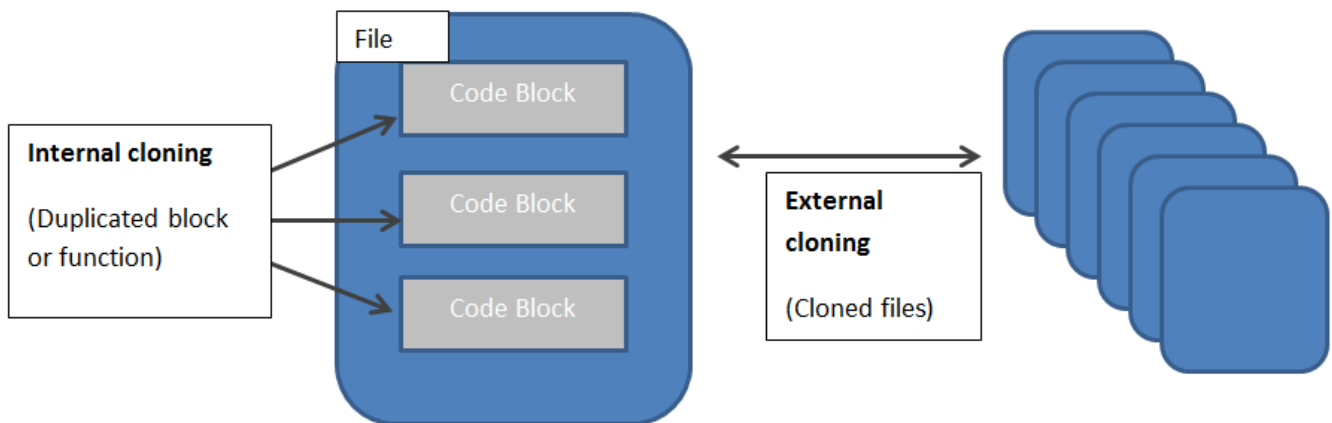
Cloning - Overview

Metrics

The analyzer produces the following metrics:

- ICC = Inner Cloned Code = Number of duplicated line of code within the artefact
- CC = Cloned Code = Number of duplicated line of code outside the artefact

From these information the Inner and Outer cloning ratio can be computed

- ICCR = 100 * (ICC/LC)
- CCR = 100 * (CC/LC)

Findings

In order to help find cloning and or duplication instances, Square generates findings that are reachable from the findings tab.

- R_NOCFTC - No Algorithmic Cloning
- R_NOCC - No Code Cloning

- R_NORS - No Repeated Substrings (Block duplication)
- R_NOCAC - Consider refactorization (Artifacts contains too many clones)

Example:

Practice ⬍	Occ. ⬍	Delta ⬍	Data Provider ⬍	Remediation Cost ⬍	Severity ⬍	SCQM Impact ⬍	Nature ⬍
▶ Cloned Files	6	+6	Squan Sources	Heavy	Major	Via Quality Rule	Risky Construction
▶ Cloned Classes	6	+6	Squan Sources	Heavy	Major	No	Risky Construction
▶ Factorizable Packages	1	+1	Squan Sources	Heavy	Major	No	Risky Construction
▶ Factorizable Classes	3	+3	Squan Sources	High	High	No	Risky Construction
▶ Factorizable Files	2	+2	Squan Sources	High	High	No	Risky Construction
▶ Factorizable Blocks in Function	5	+5	Squan Sources	Medium	Medium	Via Quality Rule	Risky Construction
▶ Cloned Functions	8	+8	Squan Sources	Medium	Medium	No	Risky Construction
▶ Cloned Algorithmic	2	+2	Squan Sources	Medium	Medium	No	Risky Construction

Cloning - Findings

Chapter 6. Test Gap Analysis

Concept

The Test Gap Analysis highlights the GAP between the code changes and the Test results. Squore combines Test Results and stability of the code (SI) to classify modules in different categories:

- ■ Code remains unchanged + Test results are compliant
- ■ Code has changed + Test results are compliant + Test execution is up to date
- ■ Code has changed + Test results are compliant + Test execution is NOT up to date
- ■ Code remains unchanged + Test results are NOT compliant
- ■ Code has changed + Test results are NOT compliant

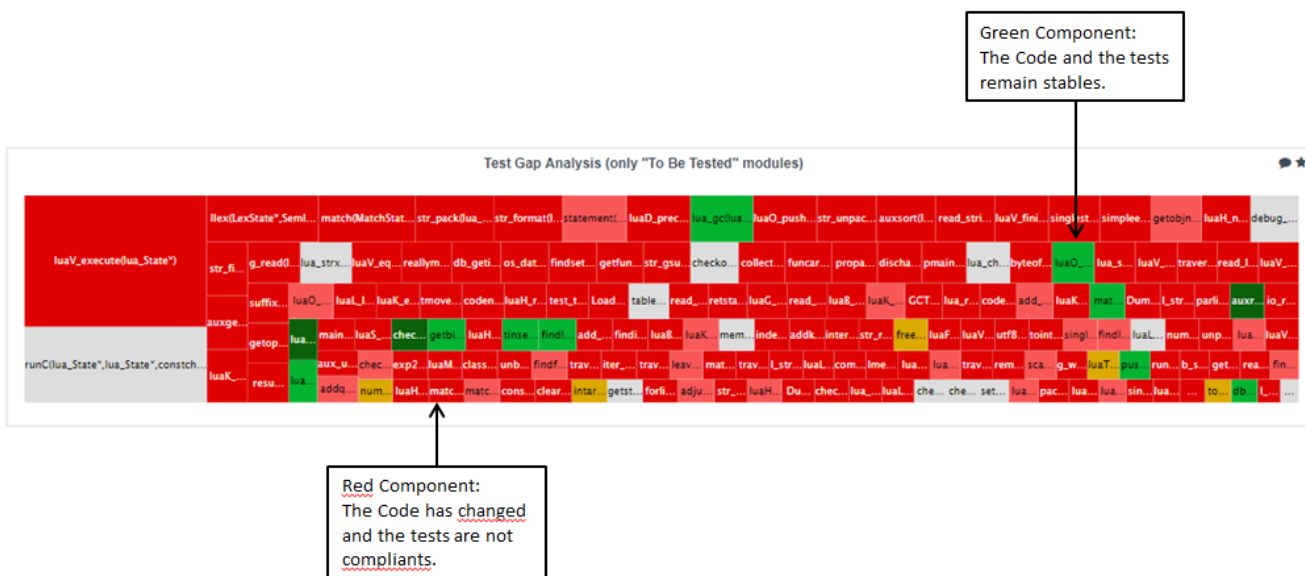*Test Gap Analysis - Legend*

How to address the test gap with Squore

Squore classifies modules according different criteria:

- Code is stable (based on source code changes: added/removed/modifed lines)
- Associated Test is Passed
- Code coverage complies with the test strategy
- Associated Test is up to date

As a result, Squore provides a Test Gap Treemap.

Test Gap Analysis - Treemap

Chapter 7. Self Descriptivness compliance

Concept

The "Self Descriptivness" KPI intends to highlight components in the code which are not well documented. The KPI analyzes the comment on different criteria:

- Comment Quantity (evaluate the comment size regarding the module complexity)
- Comment Quality (detection of "commented-out" source code)
- Comment Style (check for documentation programming rules violations)
- Comment Header (look for header comment)

Settings

During the project creation, it is possible to enable/disable the criteria in the "self descriptivness" section.

Self Descriptivness - settings

How to track comment compliance with Squore

Squore provides a dedicated "highlight" which lists the modules and their compliance regarding self descriptiveness criteria.

Self Descriptivness - Highlights

Chapter 8. Complexity

Concept

The Complexity KPI intends to assess the project's risky components regarding their complexity metrics. The indicator takes into account several criteria.

Function Metrics	Standard Reference
Comment Density COMF	>20%
Number of Paths PATH	<=80
Cyclomatic Complexity VG	<=15
Number of Parameters PARAM	<=5
Statement STMT	<=50
Nesting Level LEVL	<=5
Number of return points RETURN	<=1
Vocab Frequency VOCF	<=4

Classe Metrics	Standard Reference
Number Of Methods (NOM)	<=25
Weighted Metrics per Class (WMC)	<=60
Depth of Inheritance Tree (DIT)	<=2
Number of Children (NOC)	<=2
Multiple Inheritance (MII)	<=1
Number of Attributes (DATA)	<=7
Number of Public Attributes (APBL)	<=0
Number of Statements (STAT)	<=100

Complexity - Criteria

The Complexity indicator aggregates these metrics.

A Class or Function is considered as complex if at least **half of these metrics** do not respect the expected threshold. At project level, Squore provides a Complexity indicator based on the Volume and Distribution of the overall complexity.
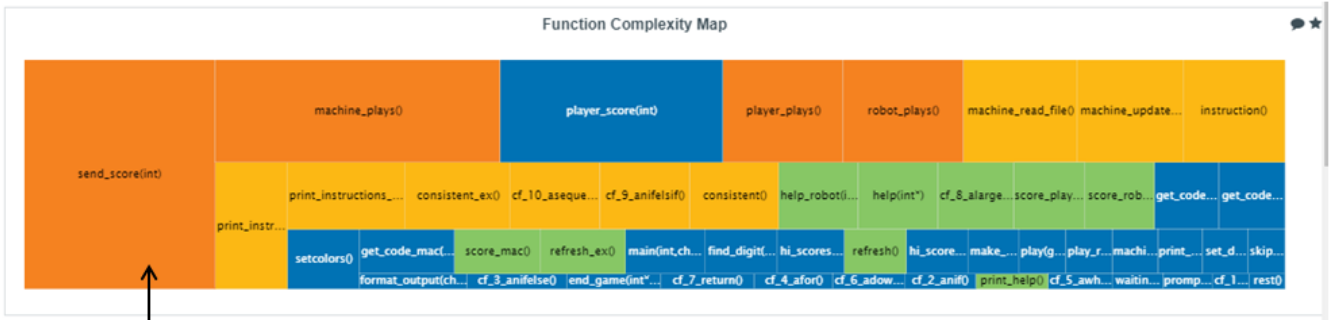

Complexity - Rating

Formula

Distribution = (#Number of Complex "Function|Classes" / #Total "Function|Classes")

Volume = (#Size of Complex Function / #Size all Functions)
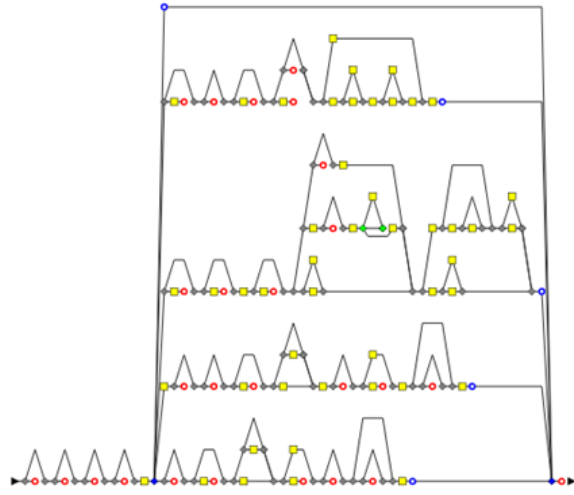
How to track the complexity with Squore

Squore provide a treemap which distribute the modules according to their complexity.

Orange Module: complexity is relatively high

Complexity - Treemap

Squore also provides "complexity highlights" which sort modules by complexity.

- List of all modules, sorted by complexity:

Rating	Artefact	Complexity Distribution	Cyclomatic Complexity	Maximum Nested Structures	Non-Cyclic Paths	Executable Statements	Number of Parameters	Distinct Operands	Vocabulary Frequency
E	robot_plays()	F	20	4	1,392	61	0	36	5.63 a
E	player_plays()	F	20	4	1,392	61	0	36	5.63 a
C	send_score(int)	F	52	4	17,056	205	1	107	5.85 a
D	machine_plays()	F	82	6	999,999	205	0	46	17.87 a
C	print_instructions_fr()	E	1	0	1	63	0	7	27.36 c
C	print_instructions_gb()	E	1	0	1	76	0	6	36.5 c
C	cf_9_anifelsif()	E	11	10	11	37	0	7	7.65 c
C	cf_10_asequenceofif()	E	11	1	1,024	37	0	7	7.12 c
D	machine_read_file()	E	15	4	77	67	0	28	7.7 a
D	machine_update_scores(int)	E	15	3	151	65	1	27	8.06 a
C	consistent()	E	19	5	496	35	0	23	7.93 c
B	consistent_ex()	E	19	5	496	37	0	23	7.98 c
E	instruction()	E	20	3	26	62	0	9	9.93 c
A	print_help()	D	1	0	1	7	0	4	4.38 c
B	refresh()	D	8	2	27	17	0	20	4.14 c

Complexity - Highlights: all modules

- List of all "unstable" modules, sorted by complexity.

Rating	Artefact	Stable within Monitoring Period	Complexity Distribution	Cyclomatic Complexity	Maximum Nested Structures	Non-Cyclic Paths	Executable Statements	Number of Parameters	Distinct Operands	Vocabulary Frequency
E	player_plays()	No	F	20	4	1,392	61	0	36	5.63
E	robot_plays()	No	F	20	4	1,392	61	0	36	5.63
D	machine_plays()	No	F	82	6	999,999	205	0	46	17.87

Complexity - Highlights: "unstable" modules

Squore combines the complexity and the Stability of the component over the Monitoring Period.

Chapter 9. Monitoring Period

Concept

The Monitoring Period defines the timeframe which is used to evaluate stability of components. It is defined in "number of days" or "number of Squore versions".

Squore analyses the history of component Stability (SI) and determines if the component has changed during this monitoring period. This is very useful in different cases:

- Test Strategy: in order to detect if how long components have been stable
- Unstable Complexity highlight (in order to list only components which have changed in the monitoring period)
- Unstable Code Coverage highlight (in order to only list components which have changed during the monitoring period)
- Test Gap Analysis (in order to detect which components are unstable)

Settings

During the project creation, , it is possible to change the Monitoring Period parameters:

```
▾ Monitoring Period

        Monitoring Period Unit  ◉ NUMBER OF DAYS      ○ NUMBER OF ANALYSES                          ⓘ
  Monitoring Period Lenght (Days)  [                                              50.0 ]  ⓘ
Monitoring Period Lenght (Version)  [                                               3.0 ]  ⓘ
```

Monitoring Period - settings

- Define the monitoring period unit The time period can be defined as a number of days meaning the "n" days before the last Squore analysis or as a number of "n" squore versions regarding the last Squore analysis. By default, the "number of days" unit is used.

  > ⓘ Choice of the unit may depend on the development process maturity and the level of automation of the Squore analysis (ie, daily analysis in continuous integration vs. manual trigger analysis from the GUI).

- Define the monitoring period The monitoring period duration can be set, depending on the monitoring period unit

Index