Software Qualimetry at Schneider Electric: a field background

By Hervé Dondey - Strategy & Innovation - Software Efficiency Team – **Schneider Electric**
and Christophe Peron – **SQuORING Technologies**

Abstract: This paper presents the Source Code Quality Indicators (SCQI) project led by the Strategy & Innovation corporate team to deploy Software Qualimetry within a large-scale multi-national organization such as Schneider Electric (SE)[1]. The related method (SCQI) was designed from a list of relevant use cases and relies on the main concepts of the SQALE [1] evaluation method. To support this method, SE has selected the SQuORE [2] platform thanks to its capability to allow large-scale deployment together with high versatility and adaptability to local needs. Feedback and lessons learned from initial deployments are now used to speed up the qualimetry process institutionalization within the whole company.

1. FOREWORD AND INTRODUCTION

Producing software products with the appropriate level of quality, under the time and resources constraints established in projects is definitively a key challenge for industries where software innovation impacts on business competitiveness increasingly.

Unfortunately, in the “Time-Money-Quality” devil’s triangle, software product quality often plays a subsidiary role from a business point of view where *functionality* always keeps a dominant position.

However, when they face operational failures and difficulty in maintaining or extending larger and larger source code, mature organizations start understanding that lack of quality is rather more expensive than the quality itself. And that prevention and early bug detection can be of high return on investment.

As a prerequisite, software quality must be specified in an explicit and measurable way. If not, it is worthless for project tradeoff against cost and time well-established indicators, useless as part of formal acceptance, and meaningless for post-mortem analysis and capitalization.

2. THE SOURCE CODE QUALITY INDICATORS (SCQI) METHOD

The Main Goals

The SCQI method aims to support the measurement activities of the internal quality of software product with an objective, impartial, reproducible and repeatable method. The key challenge is to combine and aggregate all measurement data available in a common approach in order to monitor software product quality with effective remediation plan for software developers.

Using a common quality model defined at corporate level and dedicated to source code first, each R&D centre shall be able to assess effective product quality level based on both static (e.g. complexity measurement, rule checking) and dynamic (e.g. test coverage) source code analysis results.

[1] **Schneider Electric’s profile: The global specialist in energy management**
As a global specialist in energy management with operations in more than 100 countries, Schneider Electric offers integrated solutions to make energy safe, reliable, efficient, productive and green across multiple market segments. The Group has leadership positions in energy and infrastructure, industrial processes, building automation, and data centres/networks, as well as a broad presence in residential applications. With 19.6 billion euros sales in 2010, the company's 130,000+ employees are committed to help individuals and organizations “Make the most of their energy.”
To reach this objective, around 2500 Software engineers (distributed in more than 60 R&D centres in 25 countries) are developing PC and embedded software / firmware (C/C++/C#/Java …), as well as PLCs applications, integrated within multi-level architecture solutions.

The SCQI method addresses three basic concepts of Software Qualimetry (see Fig. 1):

- the Software Product: it shall clearly state what work products generated by software processes are to be measured to assess the Quality Objectives;
- the Quality Model: It shall provide the breakdown of software product quality into characteristics, sub-characteristics and then establish the link with internal properties of the software product;
- the Analysis Model: it shall specify the rules and computation used for assessing the level of performance of the software product regarding the Quality Objectives.

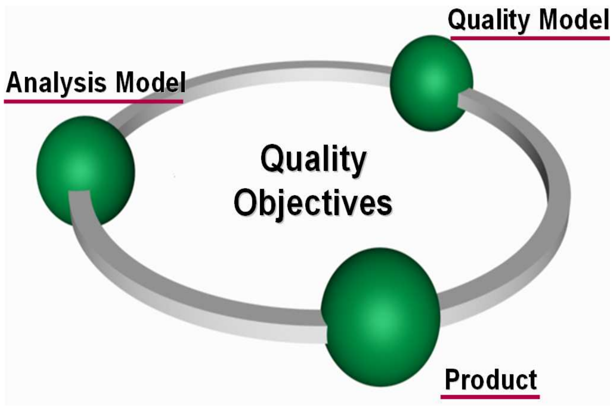


Fig. 1: Basic concepts of Software Qualimetry

The supporting process, infrastructure and tools have also been considered very early by the entities in charge of the SCQI project. As mentioned in the next chapter, they all are key conditions of success and clearly secure the return on investment.

Considering Source Code as a Vital Software Product

There is no surprise here. In the SCQI method, the software product under evaluation is the source code. Definitively not a new idea would suggest Halstead [3]. But, even if some work products such as requirements, design or tests may impact the quality objectives more significantly, the source code remains vital. Indeed, if some software comes without requirements or test cases, none comes without pieces of code… and even millions of lines. Source code also impacts some quality characteristics significantly and mostly remains a "human-made" product, so potentially highly faulty. At last, compared with other work products, the source code is quite simple to measure thanks to static analyzers. These are many reasons to monitor and control the source code thus increasing software developer productivity by early defect detection and less code to rework.

Establishing Source Code Quality Model as a Corporate Standard

The SE Source Code Quality Model (SCQM) is derived from the ISO/IEC 9126 International Standard [4], enhanced with input from Dromey [5]. As shown in fig. 2, it consists of two upper level main characteristics: *Maintainability* and *Re-usability*, each refined in four attributes and one to three sub-attributes per attribute. This breakdown of software product quality is now considered a corporate standard for SE. It means that each SE entity should have to consider and report source code quality performance through this commonly acknowledged quality model.

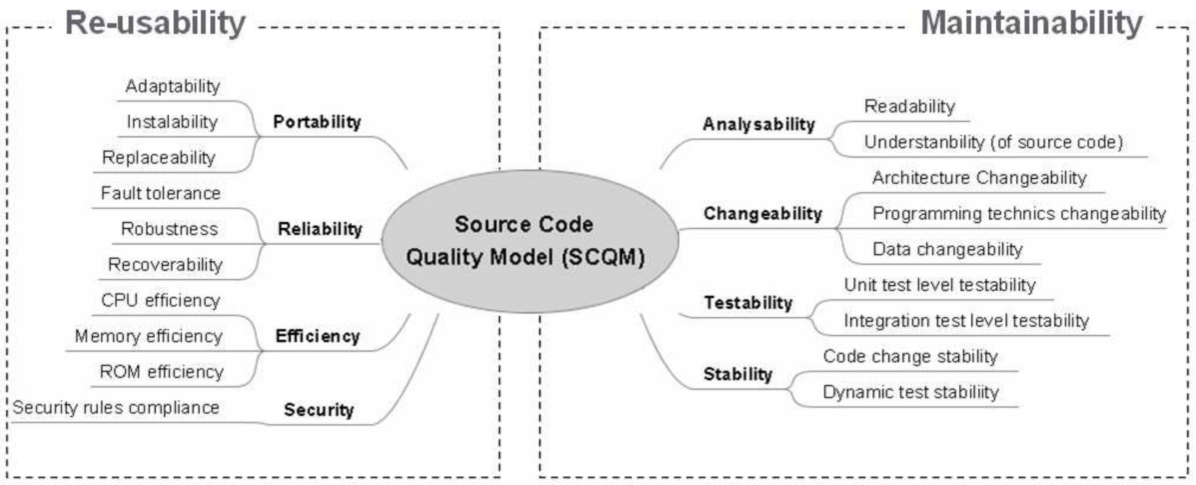


Fig. 2: The Source Code Quality Model, Schneider Electric - Corporate standard part.

At the lower level, the sub-attributes are not intended to be sub-divided and should allow being evaluated by measures and/or rules which are considered as measurable internal properties of the source code. These internal properties are selected from the Source Code Quality Indicators handbook common to all Schneider Electric entities where each property is clearly specified using:

- an internal property name,
- a type: “Measure” or ”Rule”,
- the artifact scope: Application, Package, File, Class and Method/Function, …,
- the applicable thresholds (for measure only) depending of the artifact scope,
- a severity level: i.e. “High”, “Mild”, “Low” to be used by the Analysis Model.

Tailoring the Quality Model to the Needs and Constraints

The diversity of technical contexts (e.g. PC and embedded software, firmware) and technologies in use (C, C++, C#, Java) within the SE R&D centres is managed by tailoring the Quality Model at the level of the internal properties. At that level, each R&D centre could select, add or remove measures and rules, set up specific thresholds and adjust severity levels for each internal property.

Usual taloring allows getting a balanced distribution of product quality rating between “poor” and “good” artifacts: roughly a quarter of ”poor”, a half of “fair” and a quarter of “good”.

Fig. 3 shows in yellow the tailored internal property checks done for a R&D centre producing PC software written in C#.

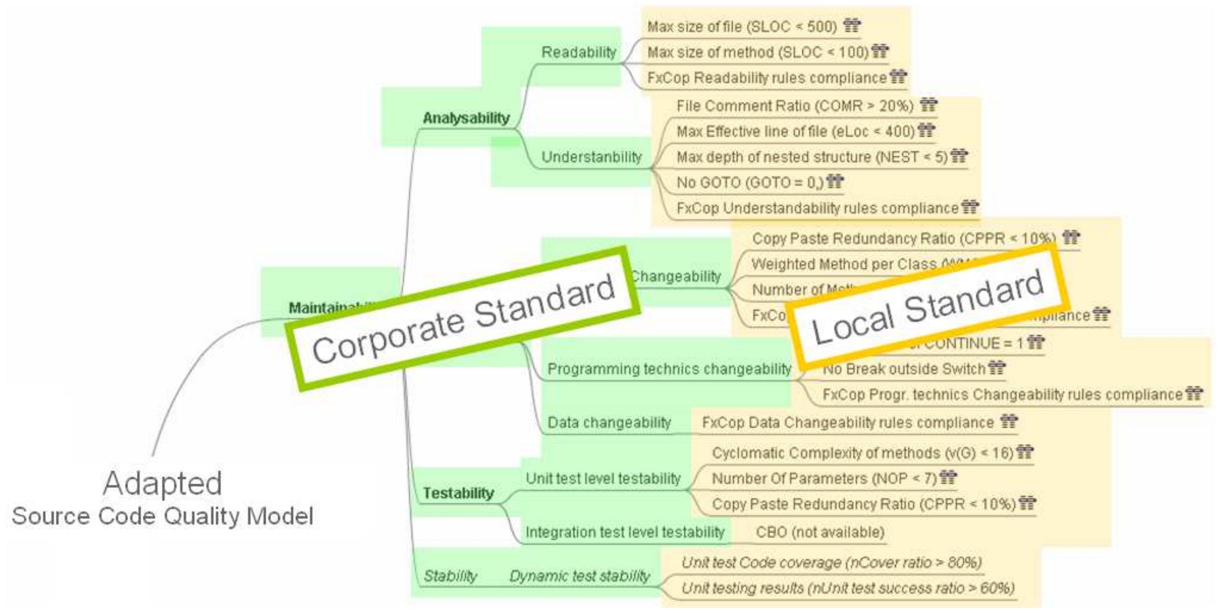

Readability
Max size of file (SLOC < 500)
Max size of method (SLOC < 100)
FxCop Readability rules compliance
Analysability
File Comment Ratio (COMR > 20%)
Max Effective line of file (eLoc < 400)
Understanbility
Max depth of nested structure (NEST < 5)
No GOTO (GOTO = 0,)
FxCop Understandability rules compliance
Copy Paste Redundancy Ratio (CPPR < 10%)
Weighted Method per Class
Changeability
Number of
Maintainability
Corporate Standard
Local Standard
CONTINUE = 1
Programming technics changeability
No Break outside Switch
FxCop Progr. technics Changeability rules compliance
Data changeability
FxCop Data Changeability rules compliance
Cyclomatic Complexity of methods (v(G) < 16)
Unit test level testability
Number Of Parameters (NOP < 7)
Testability
Copy Paste Redundancy Ratio (CPPR < 10%)
Integration test level testability
CBO (not available)
Adapted
Source Code Quality Model
Unit test Code coverage (nCover ratio > 80%)
Stability
Dynamic test stability
Unit testing results (nUnit test success ratio > 60%)

Fig. 3: The Maintainability part of the Quality Model tailored for PC software written in C#

Specifying a Relevant Analysis Model

Once established, the quality model serves as a framework for specifying all necessary indicators to evaluate the quality of source code.

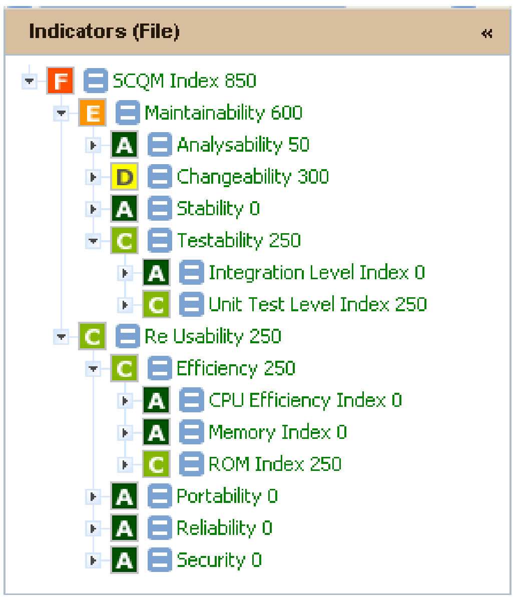
As defined in the ISO/IEC 15939 standard [6], “*an indicator is a measure that provides an estimate or evaluation of specified attributes derived from a model with respect to define information needs*” where a “model” is “*an algorithm or calculation combining one or more base and/or derived measures with associated criteria*”.

To ensure a repeatable, reproducible, objective and impartial evaluation, the SCQI Analysis Model has been based on the SQALE method [1] and the popular concept of the Technical Debt [7].

The Technical Debt can be defined as the cost of remediating / refactoring the software components to remove (intentional or unintentional) defects or to comply with requirements. In such a case, a coding standard, programming rule or internal property is considered a *Maintainability, Reliability* or *Portability* requirement.

The SQALE method defines rules for aggregating the remediation costs, either in the Quality Model tree structure, or in the hierarchy of the source code artifacts.

As illustrated on Fig. 4, this simply means that the Technical Debt at a quality characteristic level: e.g. *Maintainability* is the sum of the Technical Debts of all associated sub-characteristics: i.e. *Analysability, Changeability, Stability, Testability*. And that the Technical Debt of a source code artifact (e.g. a source file) is the sum of the Technical Debts of the embedded artifacts (e.g. the functions) added to the artifact intrinsic Technical Debt.



Indicators (File)

- F SCQM Index 850
  - E Maintainability 600
    - A Analysability 50
    - D Changeability 300
    - A Stability 0
    - C Testability 250
      - A Integration Level Index 0
      - C Unit Test Level Index 250
  - C Re Usability 250
    - C Efficiency 250
      - A CPU Efficiency Index 0
      - A Memory Index 0
      - C ROM Index 250
    - A Portability 0
    - A Reliability 0
    - A Security 0

Fig.4: Breakdown of the SCQM Index per software characteristic

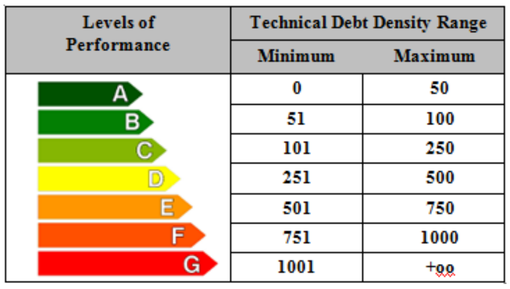
In order to provide the development teams with a comparative evaluation, the SCQI Analysis Model specifies a set of Key Performance Indicators (KPI).

As recommended in [2] but too often omitted, a KPI shall provide a level of performance, first of all: "*the degree to which the needs are satisfied, represented by a specific set of values for the quality characteristics*" [2]. This implies associating a scale to a base or derived measure and performing a rating.

The main KPI used in the SCQI Analysis Model is based on the Technical Debt Density: i.e. the average Technical Debt per thousand of source lines of code (KLOC). The lower is the density, the better the source code.

In addition, for *Learnability* purpose, the standard European Community energy label (see fig. 5) has been chosen to provide a commonly understood graphical summary of the level of performance of source code regarding the expected requirements to all stakeholders.

The table besides provides the key scale associated to the Technical Debt Density. As an example, a software project or folder containing between 2 and 5 major non conformities per KLOC would be rated "C".



Levels of Performance	Technical Debt Density Range	
	Minimum	Maximum
A	0	50
B	51	100
C	101	250
D	251	500
E	501	750
F	751	1000
G	1001	+∞

Fig.5: The SCQI Technical Debt Density Scale

Considering the ability to tailoring both the internal properties of the Quality Model and the KPI scales, the SCQI Analysis Model has been designed to be customizable enough to fit the various levels of maturity of the R&D centres within SE, the criticality of software products and the type of software development processes and technologies.

Integrating Qualimetry into the Software Development Life Cycle

The deployment process applied by the SCQI project includes integrating source code qualimetry into the Software Development Life Cycle (SDLC).

Fig. 6 explains how applying qualimetry activities all along the SDLC when using a waterfall development process. The equivalent for an Agile development process such as Scrum is currently under definition by SE teams. Three main strategies are highlighted:

1. Carry out source code quality assessment at the end of the development just before starting maintenance … so, with a risk of being too late for effective project's benefits,
2. Perform source code quality assessment on intermediate milestones : e.g. "Quality stage gate",
3. Perform as soon as and as frequently as possible quality checks.

The latter would comply with an Agile development process.

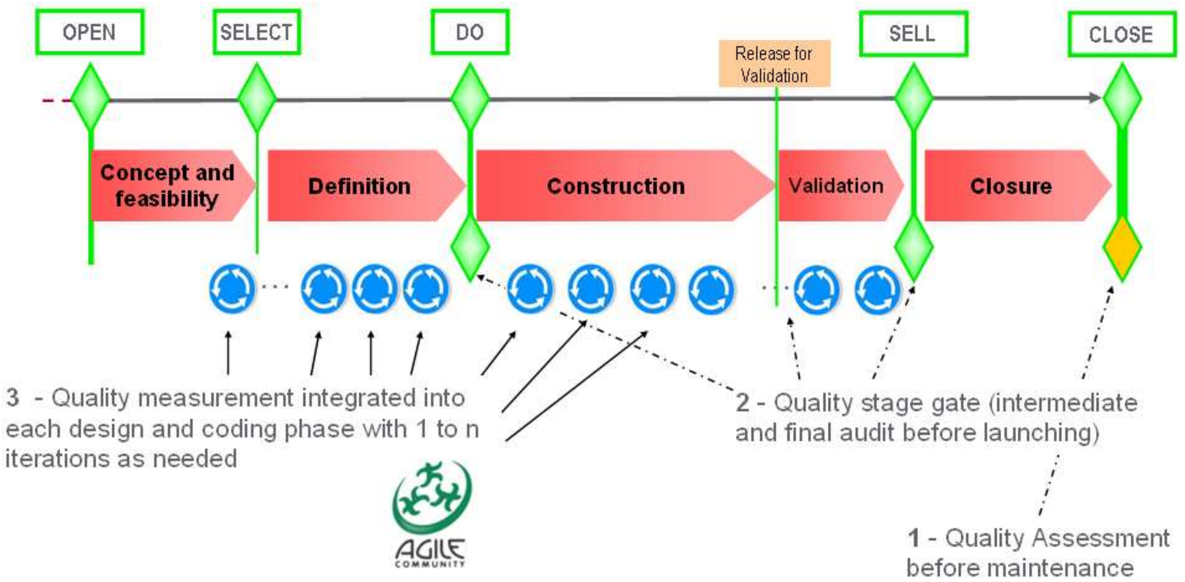

OPEN
SELECT
DO
Release for Validation
SELL
CLOSE
Concept and feasibility
Definition
Construction
Validation
Closure
3 - Quality measurement integrated into each design and coding phase with 1 to n iterations as needed
2 - Quality stage gate (intermediate and final audit before launching)
1 - Quality Assessment before maintenance
AGILE COMMUNITY

Fig. 6: Three main strategies to carry-out qualimetry activities in software development process life cycle.

The main use cases (or purpose of evaluation) supported by this process are:

- internal quality audit prior to product launching : e.g. a "GO/NOGO" decision,
- internal quality audit prior to open source component selection,
- quality assessment prior sub-contracted component acceptance ,
- quality assessment before stage gate or project milestone; Release for integration (at component level), Release for validation (at application level),
- internal quality check of intermediate builds in the continuous integration process
- quality assessment to estimate workload before maintenance

At last, the detailed SCQI verification process is naturally derived from the standard software product evaluation process specified in the part 5 of the ISO/IEC 14598 International standard [8].

Selecting the Supporting Toolset: the SQuORE Platform

It has been identified very early that automating the SCQI process and integrating it smoothly within the developer environment would be a key condition for a successful deployment.

The infrastructure required to support the SCQI process has been setting up based on Continuous Integration environments such as Cruise Control [9] or Jenkins [10].

The SQuORE software platform edited by SQuORING Technologies [2] has been selected to:

- collect and store measurement data into a centralized database,
- aggregate base and derived measures to provide Key Performance Indicators,
- provide rating of software artifacts according to the SCQI Analysis Model,

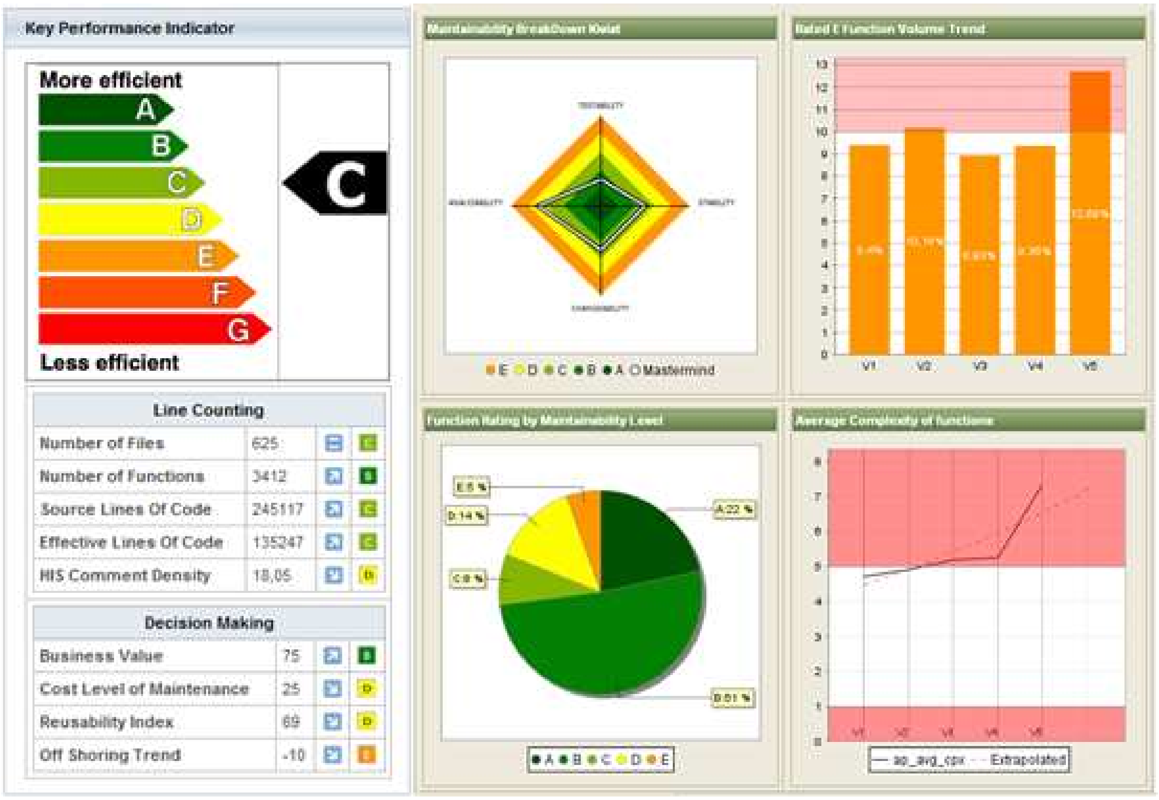
- support trend analysis from milestone to milestone allowing classic Statistical Process Control [11] for process monitoring and improvement,
- ensure immediate and up-to-date access to the data and indicators to all stakeholders.

Beyond pure performances, an intuitive and user friendly web-based interface was also a key requirement for SE to ease the adoption of the tools by the development teams.

Using a double drill-down, the SQuORE platform supports the two main use cases for developers quite simply:

- locating risky constructions and complex components to be reworked in priority,
- identifying artifacts whose level of performance has decreased since the previous version and may have fallen to a unsatisfactory level.

The Fig. 7 below provides a screenshot from the SCQI customization into the SQuORE platform:



Line Counting			
Number of Files	625		C
Number of Functions	3412		B
Source Lines Of Code	245117		C
Effective Lines Of Code	135247		C
HIS Comment Density	18,05		D

Decision Making			
Business Value	75		B
Cost Level of Maintenance	25		D
Reusability Index	69		D
Off Shoring Trend	-10		E

Fig.7: A Screenshot from a SCQI dashboard into the SQuORE product including the product level of performance (e.g. a Rated "C" artifact) and some associated charts

The SQuORE Platform is delivered with versatile parsers for C, C++, C# and Java code. They provide usual but efficient static analysis techniques:

- Complexity and coupling source code metrics such as specified in the HIS standard [12]: e.g. Cyclomatic Complexity [13],
- Control flow analysis (see example in Fig. 8),
- Programming rule checking,
- Code cloning detection.

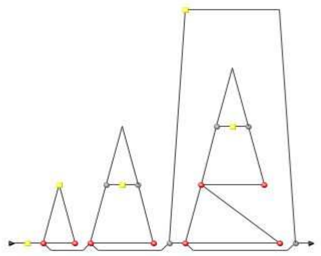


Fig. 8: A control flow graph of a C function

It allows Schneider Electric R&D centres to quickly get up to speed with a basic SCQI Analysis Model with no other measurement and analysis tool required.

However, the SQuORE platform is mainly used to interoperate with additional third party tools or "Data Providers" as defined in [6] using standard or specific connectors according to the format of data to be imported into the SQuORE data base.

Usual data providers for the SQuORE platform are:
- Rule checking tools such as CheckStyle[14], CodeSonar [15], Coverity [16], FxCop [17], Klocwork, [18] or Polyspace[19],
- Test coverage measurement tools such as gcov [20] or NCover [21].

In the context of the SQCI project, such *interoperability* was also a key criterion for selecting the supporting toolset. Indeed, it allows the SCQI rating to be more accurate and comprehensive by aggregating results from tools that are already well integrated in the various and eclectic development environments available in SE.

3. FEEDBACK FROM THE FIELD

This section presents some key feedback and lessons learned from the various deployments of the SCQI method that have been performed all over SE since 2010 by the Software Efficiency team.

Establish the scope of the evaluation clearly

It may seem obvious one stated, but it shall always be reminded to the developers that the evaluation perimeter shall only include the relevant product components.

When considering the source code, a developer starting applying the SCQI method usually swiftly selects all source files (e.g. all "*.c" file) in a given folder or from the configuration management system as part of the project and get them automatically analyzed by SQuORE. In such a case, the selection could contain generated code, test drivers and stubs, re-useable packages, deliverable components, even useless or obsolete source files.

At the end of the process, the developer sometimes complains about the fact that the remediation plan does not only focus on the effective code he/she develops or maintains but also includes change proposals on non relevant pieces of code … when he/she explicitly provided the selection of the files to be analyzed.

Indeed, evaluating the level of *Maintainability* of generated source code is totally meaningless assuming that nobody will ever open the corresponding file to locate and fix a bug.

So, to avoid wasting time understanding findings or diagnoses on non relevant pieces of code, a strict definition of the product scope: i.e. a list of software artifacts is required to keep the project team members focused on the actual valuable part of the source code to define an effective remediation plan. If some source files do not need to be maintained or re-used as is, don't assess them!

Make your own "Technical Debt"

The Technical Debt concept as just the "cost of remediation" clearly misses the level of criticality of the defects/non conformities found.

Towards the goal of providing the developers with an optimized remediation plan, the highest Technical Debt does not necessarily mean the worst or most critical artifact or the one to be corrected first. Indeed, a costly remediation may not be of great benefit regarding a cheaper one but clearly a bug.

Considering the 4 following change proposals being part of a remediation plan:
a) completing function comment headers for *Analysability* purpose,
b) specifying a default clause at the end of a switch structure for *Fault Tolerance*,
c) factorizing cloned code for *Changeability* and *Testability*,
d) adding a missing break statement ending a case block of a switch structure to avoid unintentionally falling through the next case,

the last one has definitively the lowest cost of remediation but the highest benefit for the *Maturity* of the application as it clearly removes a latent defect with no impact regarding *Stability*. So, the action d) should be placed with the highest priority in the optimized remediation plan.

Therefore, the Technical Debt finally in use in the SCQI Analysis Model looks much more a quantified "penalty" linked to a remediation priority level than a pure remediation cost. Such a Technical Debt has been established from a trade-off between remediation costs, expected benefits, and *Stability* i.e. the risk of side effects when modifying the code.

Deliver an easy-to-access solution "out-of-the-box"

In many organizations including SE, software qualimetry is often perceived as painful and time-consuming by project managers or software developers. So, arguing they are busy and under pressure, it is an easy game for them to bypass or postpone the source code verification; a usual human self-defence mechanism when facing changes or new responsibilities.

To take apart from this, the SCQI project has set up a shared infrastructure based on continuous integration environments to automate data collection and analysis. Then, only a web-browser is needed to get access to the results on the intranet. All stakeholders can easily benefit from up-to-date data and indicators.

Ensure simple use cases for the end-users

When process improvement is tool-based, the learning curve and the training process should not be underestimated.

Facing a new method and a new tool, beginners may get lost in front of the amount of information and the multiple browsing capabilities (e.g. drilling down the artifact hierarchy or the KPI tree) and then forget their initial goal: i.e. building up an optimized remediation plan using the SCQI method.

To minimize this risk, user training materials shall provide the end-users with very few simple use cases where they learn how getting started and access to the heart of the matter in a few mouse clicks.

In addition, the SQuORE decision model has been configured to automatically generate a limited selection of the key findings so called "defect reports". For beginners, setting up a remediation plan can be just limited to sorting the proposed defect reports by priority and export the most relevant ones into the project change tracking system or the product backlog in case of an Agile process.

Extend the scope of quality evaluation when possible

When initiated, the SCQI were restricted to source code. However, mature projects have expressed their need for considering complementary work products such as requirements, design models, test related artifacts, very early. Indeed, test coverage measurement as well as data from change management would clearly help evaluating *Reliability* more accurately but also the performance of the related software processes [22], [23].

In addition, the SQuORE platform integrates measures from configuration management such as the Stability Index [12]: i.e. the percentage of unchanged code since the previous version. It delivers additional information for optimizing the remediation plan. Indeed, there may be no need of proposing remediation actions on a software component that has not been changed since many versions or even years.

Optimize standardized "configurations"

From all previous deployments, several typical contexts and configurations have been identified related to:

- the level of maturity of the organization in software,
- the type of technology / languages and,
- in-place tool chain and assessment purposes.

The table below shows some of these typical configurations. They can be packaged and optimized to speed up the deployment process by delivering "ready-to-use" configurations to the R&D centres.

Project Maturity	Data Providers	Configuration and Training
Low Level: No code review or analysis yet in place. No continuous integration	**All technologies:** CPD (Copy/paste detection) SQuORE (Source code metrics)	- Corporate standard Quality Model - SQuORE deployed as a static code analyzer and qualimetry dashboard - Training/coaching provided only to super-users (senior software developer)
Medium Level: Code review and/or one or more static code analysers running at different steps during development. No continuous integration	**All technologies:** CPD (Copy/paste detection) SQuORE (Source code metrics) **C/C++:** Klocwork, Coverity, CodeSonar **C# :** FxCop **Java:** Checkstyle	- Quality Model adapted to the local technical rules set - Rule Checking results integrated into SQuORE - Qualimetry dashboard shared within the team from the SQuORE server - Training/coaching provided to some key users (i.e. senior software developers and quality engineers)
High Level: Code review supported by static code analyzers running all along the software development lifecycle. Continuous integration including static code analysis, unit testing and code coverage measurement	**All technologies:** CPD (Copy/paste detection) SQuORE (Source code metrics) **C/C++:** Klocwork, Coverity, CodeSonar**,** Gcov **C#:** FxCop, Nunit, Ncover **Java:** Checkstyle, Junit, FindBugs	- Quality Model adapted to the local technical rules set - All results integrated in automatic tool chain - Qualimetry dashboard shared within team in development loop - Training/coaching provided to all team members.

4. CONCLUSION AND PERSPECTIVES

The SCQI project has mainly justified its return on investment from a higher productivity of the software development and testing teams. Indeed, a better quality of the source code clearly leads to less rework, less testing and review.

The gain on investment can be established and demonstrated from:
- a lower rate of bugs per kilo line of code due to early defect detection,
- a higher rate of change requests implemented per Man/Day due to less code to maintain and a code easier to dive in.

As of today, it is too early to evaluate the gain on investment in such a quantified way. However, leveling off the Technical Debt for existing projects would be considered as a first success ensuring no regression in *Maintainability* of the legacy code. This is the first goal assigned to the project managers.

On another hand, the SCQI are just spreading within SE. And this takes various ways:

Deploying on more and more R&D centres: The ultimate goal is for all software projects within SE to use the SCQI. In the next quarter, several new deployments are already planned all around the world: e.g. China, Australia, Canada; clearly, a first fulfillment for the SCQI as the deployment is funded by the R&D centres and initiated on a voluntary basis.

Adding new data providers: All along the future deployments, the SCQI project will face new field configurations leading to more and more third party tools interoperating with the SQuORE platform. At last, the SCQI should be more and more accurate and future implementations easier and faster.

Including architectural data: Evaluating some internal properties on the architecture or general design of the application misses information and measures. The SQCI supporting toolset will provide such capabilities soon.

Towards a "Capitalization Database": The SQuORE platform includes statistical features e.g. histograms, correlation matrices. Assuming the collection of the relevant data, such data analysis features will be used for return of experience studies or post-mortem analyses:

- demonstrate the ROI of the SCQI as stated below,
- identify correlation between "Quality-In-Use" variables under monitoring e.g. the number of bugs, and potential explanatory "product and processes attributes" variables e.g. code complexity, test coverage.

Such studies should lead to effective tuning of the SCQM and forecasting capabilities for Project Monitoring and Control.

REFERENCES

1. More information on sqale.org
2. More information on squoring.com
3. Halstead (1977) - Elements of software science – Elsevier
4. ISO/IEC 9126-1: 2001 – Software Engineering – Product quality – Part 1: Quality model
5. Dromey (1994) – A model for software product – Software Quality Institute – Griffith University Nathan, Brisbane
6. ISO/IEC 15939:2002- Software Engineering – Software measurement process
7. Cunninghan (1992). – The WyCash Portfolio Management System.
8. ISO/IEC 14598:1999- Information technology – Software product evaluation
9. More information on cruisecontrol.sourceforge.net
10. More information on jenkins-ci.org
11. Zhang, Kim (2010) – Monitoring Software Quality Evolution for Defects – IEEE Software
12. HIS Source Code Metrics (2005) – Hersteller Inititative Software AK Softwaretest
13. McCabe (1976) - A Complexity Measure – IEEE Transactions on Software Engineering – Vol. SE-2
14. More information on checkstyle.sourceforge.net
15. More information on grammatech.com
16. More information on coverity.com
17. More information on microsoft.com
18. More information on klocwork.com
19. More information on mathworks.com
20. More information on gcov-eclipse.sourceforge.net
21. More information on ncover.com
22. IEEE Std 928.2 (1988) - Software Maturity Index
23. Graves, Karr, Marron, Siy (1998) - Predicting Fault Incidence using software change history – NISS Tech. report 80