# The SQALE Quality and Analysis Models for Assessing the Quality of Ada Source Code

Thierry Coq[1] and Jean-Pierre Rosen[2]

[1] DNV France, Paris, France thierry.coq@dnv.com, http://www.dnv.com
[2] Adalog, Issy-les-Moulineaux, France rosen@adalog.fr, http://www.adalog.com

**Abstract.** This article presents the quality and analysis model of the SQALE assessment method of software source code. It explains how an Ada quality model compliant to SQALE is implemented and the results of its application to selected software, and how the use of Ada reduces the quality debt unlike many other technologies.

## 1 Introduction

Det Norske Veritas (DNV) is a not-for-profit organization specialized in risk management. As such, we are conducting research in the measurement of software quality (qualimetry). We discovered that the analysis model, and more precisely the rules used for aggregating raw measures, is a key factor for the effective implementation of qualimetry. This paper introduces the analysis model of the SQALE (Software Quality Assessment Based on Lifecycle Expectations) method to assess the quality of software and in particular software source code. A detailed view of SQALE has been presented in our white paper [1]. The quality model of SQALE and its application to real-time or embedded software has been described in [2]. Its particular strength resides in its compliance to the representation clause [3]. The SQALE method is open source and freely available [4]. Several tool vendors provide implementations for various languages such as C, C++, Java and Cobol. We explain in this paper how the SQALE for Ada quality model is developed in compliance with the SQALE requirements, how the basic Ada metric and quality measurements tools can be used. Finally a few results of applying SQALE for Ada are described and analyzed, putting in evidence the small quality debt incurred in the selected projects.

## 2 The SQALE Analysis Model

Before describing in detail the implementation of the SQALE model for Ada, the quality model of the SQALE method will be briefly presented. The quality model of the SQALE method expresses the requirements applicable to the software and its source code over its life cycle. In the same manner that the activities linked to making the software and in particular its source code, follow a clear chronology the requirements applicable to the source code appear in a same

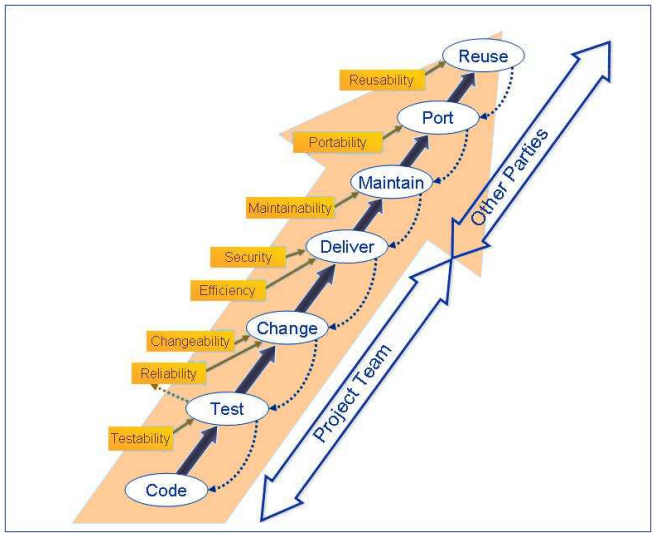



**Fig. 1.** Dependencies between Activities and Quality Characteristics

order. The approach and the structure of the SQALE quality model has been detailed elsewhere [1] and are summarized in figures 1 and 2.

The generic SQALE model is derived according to the implementation technologies (design and source code languages) and the tailoring needs of the project. As stated in [1], the quality model is a requirements model. The way it is built ensures the quality targets a total absence of non compliances. As written by Ph. Crosby [5], assessing a software source code is therefore similar to measuring the distance which separates it from its quality target. To measure this distance, the concept of remediation index has been defined and implemented in the SQALE analysis model. An index is associated to each component of the software source code (for example, a file, a module or a class). The index represents the remediation effort which would be necessary to correct the non compliances detected in the component, versus the model requirements. Since the remediation index represents a work effort, the consolidation of the indices is a simple addition of uniform information, which is compliant with the representation condition and a critical advantage of our model. A component index is computed by the addition of the indices of its elements.

A characteristic index is computed by the addition of the base indices of its sub-characteristics. A sub-characteristic index is computed by the addition of the indices of its control points. Base indices are computed by rules which comply with the following principles:

– A base index takes into account the unit remediation effort to correct the non-compliance. In practice, this effort mostly depends on the type of non-compliance. For example correcting a presentation defect (bad indentation,

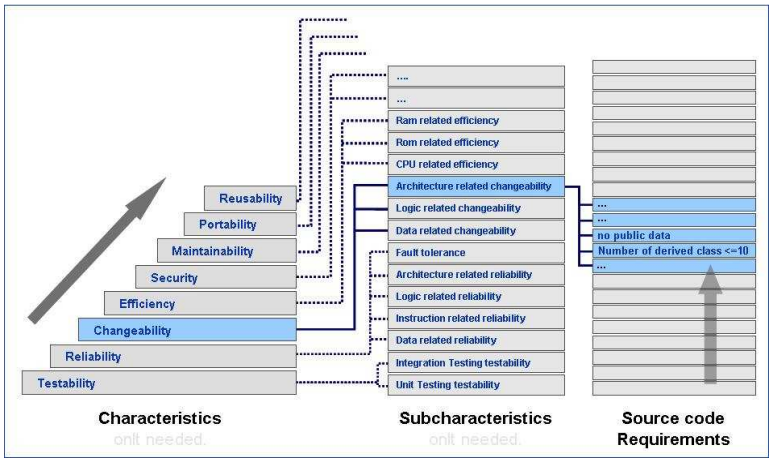



**Fig. 2.** Some details of Level 2 and 3 of the SQALE Quality Model

dead code) does not have the same unit effort cost as correcting an active code defect (which implies the creation and execution of new unit tests, possible new integration and regression tests).
– A base index also considers the number of non-compliances. For example, a file which has three methods which need to be broken down into smaller methods because of complexity will have an index three times as high as a file which has only one complex method, all other things being equal.

Base indices are aggregated either by the artifact where the non-compliance has been identified, or by the relevant (sub-)characteristic. In the end, the remediation indices provide a means to measure and compare non compliances of very different origins and very different types. Coding rule violation non-compliances, threshold violations for a metric or the presence of an antipattern non-compliance can be compared using their relative impact on the index.

The standard measure set of SQALE has more than 30 control points and the extended set more than 60. A few examples of base measures are explained in more detail, to show how each complies with the conditions explained above:

– A well-known issue contributing to reduced testability is an excessive cyclomatic complexity for a given operation (procedure or function) in the code. The default threshold for excessive complexity in SQALE is 15. Any operation having a V(G) over 15 will be counted as one violation, and the count is cumulative per class and file in order to apply the representation condition. This measure is mostly independent of the programming language, each language has an equivalent.
– A contributor to reliability measurement is the absence of dynamic memory allocation (for real-time software) or a balanced use of allocation and deal-

location instructions (malloc and freemem in C for example). Each violation of this rule increments the count by one, again for each class and file.
- A contributor changeability measurement can be obtained by computing the number of operations (methods) per class (excluding getters and setters) and checking it is beneath a threshold, fixed in SQALE at 30.
- A contributor to maintainability measurement is obtained by measuring the comment ratio, for each file. If it is below SQALEs default threshold of 25%, a violation is counted.
- Finally, for real-time software, the presence of dead code and commented-out code is also counted as a contributor against maintainability.

Of course, the various examples presented above have different remediation indices. The SQALE method uses the organizations remediation indices built using historical data from the projects. If it is not possible, a Delphi analysis [6] or AHP [7] may be used with the project and the SQALE experts to define expert-based remediation indices.

In the above examples, the thresholds provided are examples in a particular context, the SQALE method providing conservative defaults if needed. Many authors (f.e.: [8–12]) have proposed individual check points and thresholds that can be used in SQALE, provided the representation clause is satisfied [3].

The SQALE quality and analysis models have been used to perform many assessments of software source code, of various application domains and sizes. The same layered and generic quality model has been used to assess Cobol, Java, embedded Ada, C or C++ source code. For Java, C++ and Ada, the quality model contains object-oriented metrics to assess Testability, Changeability and Reusability. The quality model also provides control points to detect the absence of antipatterns such as those identified by Brown [13]. The indices are computed based on the average remediation efforts estimated by the development team. The index thresholds providing a rating in five levels (from "poor" to "excellent") are established by the application managers.

## 3 The SQALE for Ada quality model

Making a SQALE Ada quality model requires defining the requirements for the Ada language. Some requirements can be reused as-is from the SQALE default quality model, such as the maximum cyclomatic complexity for subprograms, or the absence of copy/paste (for 100 tokens). Other requirements such as the one for comments can be applied to Ada, but with a lower bound (and a maximum) due to the inherently more readable nature of the language.

Other requirements are not applicable as they are enforced by the compiler. The most notable of those is the requirement for a directed acyclic hierarchical dependency graph between units of compilation: cyclic dependencies between packages are prohibited by the language.

Some requirements related to object-oriented concepts are more difficult to analyze in the dual nature of the Ada language and have to be adapted to the language. For example, the stability requirement is computed on the efferent and

afferent coupling of packages, not objects. The final list of selected requirements, for our SQALE Ada Quality Model covers most characteristics and subcharacteristics of the SQALE model.

A drawback of the model is the lack of efficiency requirements. SQALE requirements in other languages require the absence of certain statements or library functions likely to cause inefficiencies. A first analysis has not uncovered the equivalent in the Ada language. Other efficiency requirements need to be set up, related to the two sub-characteristics: CPU performance, memory (RAM) performance. One requirement was identified related to the absence of dead code in the source, and linked to the memory (ROM) performance subcharacteristic. The final SQALE Ada Quality Model is presented in table 1. The reusability

**Table 1.** The SQALE for Ada Default Quality Model

| No | Characteristic | Sub-Characteristic | Generic Requirement Description | Ada Requirement |
|---|---|---|---|---|
| 1 | Testability | Unit testability | Acceptable number of parameters in a call (NOP) | NOP $\leq$ 5 |
| 2 | Testability | Unit testability | Acceptable number of test paths in a module (V(G)) | V(G) $\leq$ 15 |
| 3 | Testability | Unit testability | Tolerable number of test paths in a module (v(G)) | V(G) $\leq$ 60 |
| 4 | Testability | Unit testability | Acceptable number of different called modules from a module (FANOUT) | Efferent coupling $\leq$ 20 |
| 5 | Testability | Unit testability | Acceptable duplication within a module (CPRR100) | Number of CPRR100 violations |
| 6 | Testability | Unit testability | All code paths within a module are reachable | All code is reachable |
| 7 | Testability | Unit testability | All modules are reachable | All modules are reachable |
| 8 | Testability | Unit testability | No module calling itself recursively | No recursion |
| 9 | Testability | Integration testability | Acceptable coupling between objects (CBO) | CBO $\leq$ 7 |
| 10 | Testability | Integration testability | No public data within classes | No directly accessed globals, all public (tagged) types are private. |
| 11 | Testability | Integration testability | Acceptable number of direct declared required files | With count < 50 |
| 16 | Reliability | Data reliability | All types are safely converted | No unchecked conversions |
| 17 | Reliability | Data reliability | No use of unitialized variables | No use of unitialized variables |
| 19 | Reliability | Logic reliability | One single point of exit per module | No multiple exits |
| 25 | Reliability | Statement reliability | Reproducible floating point computations | No equality comparison between reals |
| 27 | Reliability | Statement reliability | No ambiguous statement execution order | No operator precedence order ambiguity |
| 28 | Reliability | Synchronization related reliability | Shared resources are used in protected scope | No shared variables used in several contexts |
| 34 | Reliability | Architecture reliability | Standardized error and exception handling | No exception propagates to other languages |
| 38 | Changeability | Architecture changeability | No different elements with the same name | No local hiding |
| 39 | Changeability | Architecture changeability | Acceptable number of class methods (NOM) | NOM (public) $\leq$ 60 (for a package)? |
| 42 | Changeability | Data changeability | No explicit constants directly used in the code (except 0,1...) | No literals in expressions or statements |
| 43 | Changeability | Data changeability | All objects are declared at smallest scope | No unnecessary use or with clause, no reduceable scope |
| 45 | Efficiency | RAM efficiency | No unused variable, parameter or constant in code | No unused variable, parameter or constant in code |
| 48 | Efficiency | ROM efficiency | All statements are useful | No simplifiable statements |
| 51 | Maintainability | Readability | Acceptable File size | LSLOC < 2000 |
| 58 | Maintainability | Readability | Capitalization rules are followed for code elements. | Casing |
| 59 | Maintainability | Readability | Rules for identifying types, variables and other code elements are followed. | Check the project's naming rules are applied. (To be adjusted to the project) |
| 60 | Maintainability | Understandability | Acceptable minimum level of comments | Comments density $\geq$10% (needs to take into account verbose FOSS headers) |
| 61 | Maintainability | Understandability | Acceptable maximum level of comments | Comments density $\leq$35% (needs to take into account verbose FOSS headers) |
| 64 | Maintainability | Understandability | No unstructured statements (goto, break outside a switch...) (eV(G)) | eV(G) $\leq$1 |
| 66 | Reusability | Stability | The SDP (Stability Dependency Principle) is applied | The less stable package is not used by the more stable package[3] |

requirement is based on the stability principle, where the dependency graph and the stability are computed to determine the actual reusability. If a less stable package is reused by a more stable one, then it is a violation of the model (for both packages). The stability is computed as the ratio of using packages over the total of the using and used packages. If a package does not use any other packages, it has a stability of one.

Unlike tool-based quality models, the SQALE Ada Quality model is based on defining the objectives and requirements first, then finding or building the tools needed to implement it, creating the tools check points from the requirements.

In addition to the quality model, an analysis model was defined for the Ada language. Each requirement was assigned a remediation factor, based on the estimated work units required to correct the defect.

The remediation factors are defined by the following table, and mapped to the quality model. In addition, in order to compute index densities, the size of

**Table 2.** The SQALE Ada remediation factors

| Non-Compliance Type Name | Description | Remediation Factor | Sample |
|---|---|---|---|
| Type0 | Undefined | 0 | Not applicable |
| Type1 | Fixable by automated tool, no risk | 0.01 | Change in capitalization |
| Type2 | Manual remediation, but no impact on compilation | 0.1 | Add comments |
| Type3 | Local impact, need only unit testing | 1 | Replace an instruction by another |
| Type4 | Medium impact, need integration testing | 5 | Split a big function in two |
| Type5 | Large impact, need a complete validation | 20 | Architectural change |

the packages (in source lines of code) was used as a rough estimate of the number of work units to produce the package from scratch, if it were entirely rewritten. For example, a typical package with 15 lines of specification and 250 lines of body

**Table 3.** The effort scale per source line for each type of package

| Package Type | Work Unit per Line |
|---|---|
| Package Specification (.ads) | 1 |
| Package Body (.adb) | 0.1 |

rates 15 + 25 or 40 work units in this model. The justification for this choice is

based on the authors experience of using Ada as a specification language where the structure of the packages is as important as the implementation.

This number can then be compared with the indices obtained, either in total or for a given characteristic, and the file rated using the rating scale described in table 4. For example, a package with 25 work units and a remediation index of 30 would be rated as "E", very bad (rating of 1.2 in the interval $]1, +\infty[$) . The same with a remediation index of 7 would be rated as a "C", medium (rating of 0.28 in the interval ]0.1, 0.3]). Of course, where available in organizations, a better estimation model may be used to assign more precise remediation and work effort factors in the analysis model.

Once the quality and the analysis models are defined, the tools implementing these models may be selected and implemented where missing.

**Table 4.** The SQALE Ada rating thresholds

| Class Name | Class Letter | Rating Interval | Color |
|---|---|---|---|
| Excellent | A | [0, 0.03] | green |
| Good | B | ]0.03, 0.1] | light green |
| Medium | C | ]0.1, 0.3] | yellow |
| Bad | D | ]0.3, 1] | orange |
| Very Bad | E | ]1, $+\infty$[ | red |

## 4 Implementing SQALE for Ada quality model

The first step of the method consists in building the non-compliance table from the source code. The choice of an appropriate set of tools for this is fundamental to the method, since the significance of the results depends strongly on the accuracy and reliability of the measurement tool. For example, simple text processing tools like Unix's "grep", are too sensitive to presentation issues to be used [15].

The generic SQALE quality model [4] identifies more than 66 points of measurements, and no single tool is able to measure all of the derived checkpoints. Developing a custom tool for SQALE is not feasible, given the limited time and budget allotted to the Ada implementation. However, using a limited number of tools and some "glue" processing, we were able to implement the method for Ada. These checkpoints are close to programming rules: they are places in the source code where undesirable features are used (goto, multiple loop exits), or where a certain limit is exceeded (number of parameters, cyclomatic complexity).

The requirements are designed independently of any programming language. An interesting property of Ada is that, among the 66 requirements, 17, which correspond to features that are best avoided, are actually forbidden by the language definition (and thus automatically enforced).

Our main checking tool is Adalog's AdaControl [14]. The choice of AdaControl was motivated by several reasons:

– Since it is an ASIS [21] tool, its analysis on the language is based on the same technology as the compiler, thus increasing the confidence that the tool processes the language correctly.
– It has a rich set of rules. Out of the remaining 49 requirements, 22 had checkpoints that were provided right out of the box.
– It can output its results in CSV format, making them directly loadable in a spreadsheet program for further analysis.
– Moreover, since AdaControl is free software and easily extendable [15–17], more checkpoints can be added at will.

AdaControl is oriented towards finding occurrences of various constructs, more than actually measuring mathematical or statistical properties of the source. For rules that were of this second kind (number of paths, cyclomatic complexity, fan-out), we used AdaCore's Gnatmetric tool. Gnatmetric is also ASIS based and free software.

PMD-CPD[18] is also used to compute the copy/paste non compliances. In addition, a little post-processing was used to compute some of the complex checkpoints, glue the results together and build the indices. Table 5 summarizes the tool set used.

**Table 5.** The SQALE Ada tool set

| Tool | Usage |
|---|---|
| AdaControl | Most check-points |
| Gnatmetric | Volumetry, comments |
| PMD-CPD | Copy/Paste detection |
| Specific tooling | Stability, dependencies, Index assembly |

## 5 Some results of SQALE for Ada

Two open-source projects are used to present SQALE for Ada: AdaControl[14] and Ada Web Server (AWS)[19, 20]. The data presented here are for illustration purposes and does not constitute an endorsement or rejection of either project.

### 5.1 SQALE for Ada applied to AdaControl

Naturally we used AdaControl as an example of applying SQALE to an Ada project. AdaControl is free software: the source code is readily available, and our results can be published, unlike most industrial applications of SQALE which are performed on confidential software.

In this analysis, we analyzed the AdaControl as a whole, and computed the indices for the three parts: AdaControl itself, the ASIS framework, and the GNAT packages used. Showing the indices for the 3 different parts allows us to

show how each part has unique properties. Each SQALE analysis first provides size measurements as in figure 3:

| Volumetry | GNAT | ASIS | AdaControl | Total |
|---|---|---|---|---|
| Total number of lines | 74 550 | 99 248 | 56 282 | 230 080 |
| Total number of source lines | 38 860 | 48 105 | 36 810 | 123 775 |
| Total number of source files | 64 | 131 | 183 | 378 |
| Sum of cyclomatic complexity (v(G)) | 5 523 | 7 079 | 4 766 | 17 368 |
| Total number of methods or functions | - | - | - | - |

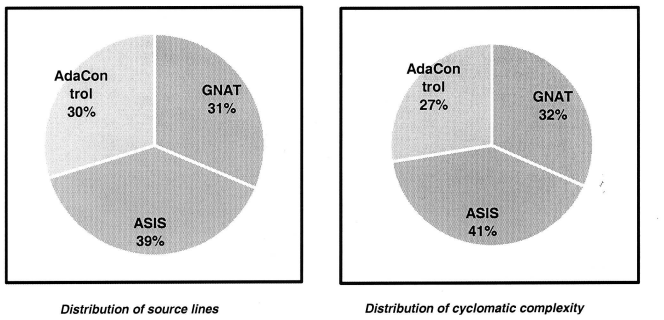



**Fig. 3.** Volume indicators for the AdaControl software

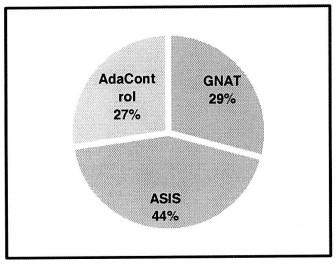
The total line code is around 230 KLoc, evenly distributed between the three parts of the application. This is a rather low range of the application size for SQALE, which can target the 0.1  40 million lines of code range. It is an ideal example, however to demonstrate the usefulness of SQALE.

Once the indices and the densities have been computed, it is possible to use the aggregated indices and compare the index densities, as in figure 4. The total quality index for the AdaControl application is 2930 work units, where the GNAT library takes 1474, the ASIS library 1109 and the AdaControl specific part 347. The absolute numbers are difficult to compare, so the index densities are computed over the sum of the lines of code, resulting in a quality index density of 40, 20 and 10 for each KLoc of code for the GNAT library, the ASIS library and the AdaControl specific part, respectively. For an industrial analysis, where the parts could be delivered by different teams, this figure could be used to benchmark the quality results of each team, against expected quality targets.

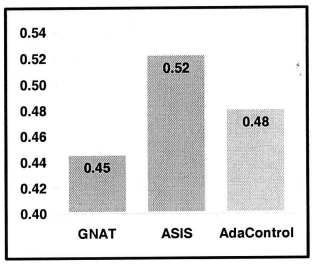
There is another way to look at the results, especially from a project managers point of view, which has a limited budget for the improvement of the quality of the software. Which defects should be corrected first? Figures 5 and 6 present a vivid picture demonstrating where the major benefits could be obtained.

The GNAT library actually has a limited testability issue and the ASIS framework has a higher index, as well as a higher index density. Improving the ASIS framework would increase the overall quality of this application and is a major result of the SQALE analysis.

| Code Type | GNAT | ASIS | AdaControl |
|---|---|---|---|
| Sum of Indexes | 2 469 | 3 709 | 2 297 |
| Σ Index/ Σ V(G) | 0.45 | 0.52 | 0.48 |



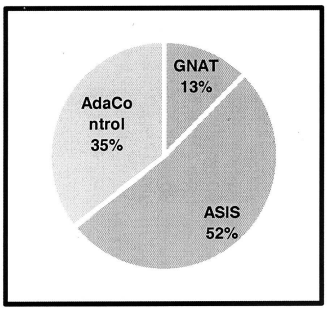

Repartition of Index




Comparison of index density (by v(G))

**Fig. 4.** The indices and index densities for the three parts of AdaControl

| TESTABILITY | GNAT | ASIS | AdaControl |
|---|---|---|---|
| Unit level Testability | 175 | 725 | 460 |
| Integration level Testability | - | - | 34 |
| TOTAL | 175 | 725 | 494 |




Repartition of Index

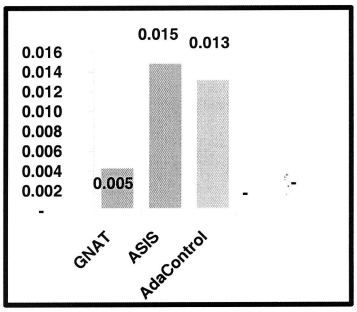



**Fig. 5.** The testability indices and index densities

| RELIABILITY | GNAT | ASIS | AdaControl |
|---|---|---|---|
| Data related Robustness | - | - | - |
| Instruction related Robustness | - | - | - |
| Logic related Robustness | 1 062 | 276 | 19 |
| Architecture related Robustness | - | - | - |
| Fault Tolerance | - | - | - |
| Exception Handling | - | - | - |
| TOTAL | 1 062 | 276 | 19 |

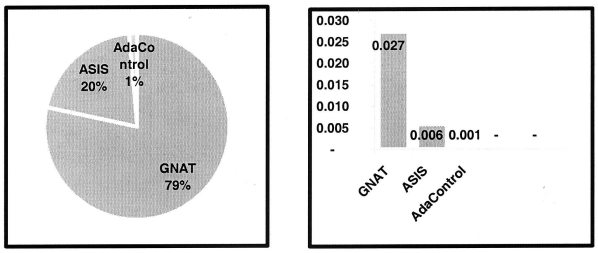



**Fig. 6.** The reliability indices and index densities

The reliability indices computed by the SQALE method are mostly linked to the GNAT library, and within the indices, related to the use of global unprotected data. If some quality budget remains, it might be useful for the maintainers to review and protect the global variables published by the GNAT library. Fully half of the indices have been analyzed by just reviewing the first two characteristics of the SQALE model. A careful project manager may not need to look further

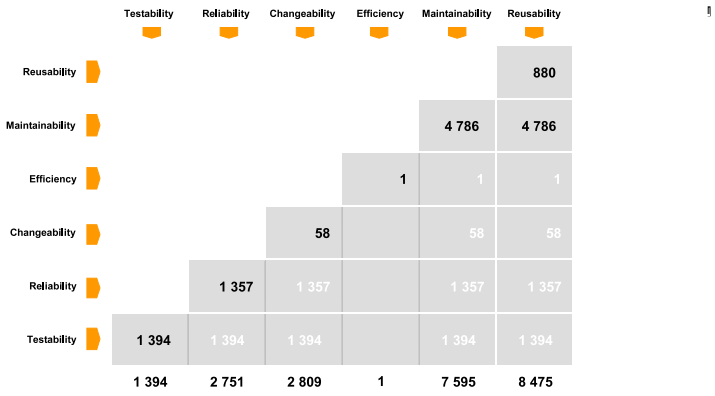



**Fig. 7.** The SQALE pyramid for AdaControl

for quality increases until these issues have been solved. The SQALE pyramid

provides the same indication in a clear picture: 1731 work units out 2930 are assigned to the testability and reliability characteristics. The actual amount of files to be modified is extremely low, mostly below 10%. In addition, since the copy/paste requirement is in the testability characteristic, the low values we see here indicate a low copy/paste problem, which is often not the case in industrial SQALE analyses.

### 5.2 SQALE for Ada applied to Ada Web Server

Ada Web Server (AWS) is a framework to provide complete web based applications. It can be embedded in an Ada application to provide web services. See [19] for more information. AWS itself reuses other libraries available within the Ada community, such as XMLAda, a SSL library. It contains a well-defined "templates parser" module to separate web design from the code. For our analysis, we decided to rate each part separately.

The overall size of the framework and its parts is shown in figure 8 below. With 132 KLOC, this framework is smaller than the AdaControl application. The 7000 cyclomatic complexity sum is consistent with the size. The "templates parser" module is a small component of AWS, while XMLAda is roughly equivalent to the rest of the code base. Figure 9 below shows the SQALE pyra-

| Volumetry | AWS | XMLAda | SSL | Templates | Total |
|---|---|---|---|---|---|
| Total number of lines | 69 112 | 51 498 | 709 | 11 275 | 132 594 |
| Total number of source lines | 37 773 | 37 286 | 459 | 7 059 | 82 577 |
| Total number of source files | 275 | 196 | 2 | 21 | 494 |
| Sum of cyclomatic complexity (v(G)) | 3 461 | 3 410 | - | 509 | 7 380 |
| Total number of methods or functions | - | - | - | - | - |

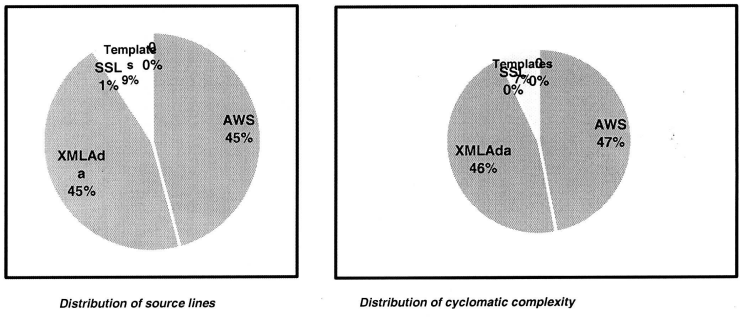



**Fig. 8.** application of SQALE to another Ada framework: Ada Web Server (AWS)

mid for AWS. Testability and reliability indices are quite correct, whereas some work might still be useful on the maintainability and reusability characteristics. The final quotation shows little need for improving the AWS packages for testability, while maintainability might be an issue for some of the packages.

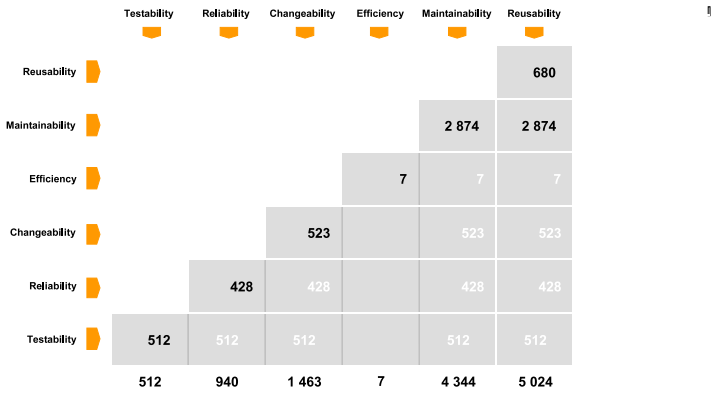



**Fig. 9.** The SQALE pyramid for AWS

Most of the remediation in the maintainability characteristic is related to the requirement "eV(G) $\leq 1$" and needs more investigation. Again, as for AdaControl and contrary to common industrial practice, there is little need for copy/paste refactoring.

## 6 Future work

The validity of the SQALE model is based on its focusing on the quality requirements first, then drilling down into the sub-characteristics and how the requirements are implemented by the tools as check points. As described above, the efficiency characteristic is lacking requirements for Ada. Additional research can therefore identify requirements, or if that proves too difficult, identify why in the Ada language such an endeavor is difficult. Additional reusability requirements would also be very useful. Once SQALE starts to be used, the Ada community will be able to review the various SQALE requirements and fine-tune them for its specific needs.

Building a SQALE quality model for Ada has proved surprisingly easy, especially compared to other languages. SQALE has always been intended as a method for checking software quality, not only source code quality. SQALE for Ada might be the right quality model to extend SQALE and start using requirements for other software artifacts such as requirements or design models, test cases and test results (using coverage and dynamic analysis tools). Particularly, being able to measure design at an early stage may result in SQALE being used as early predictor of final quality.

## 7 Conclusion

This paper proposes a quality model and an analysis model for measuring the quality of applications using the Ada language, based on the SQALE method. It also describes a set of tools for implementing the checkpoints and computing the resulting indices, index densities and ratings according to the SQALE method. These indices are computed to estimate the remaining technical debt, or work effort remaining in the application from quality non-compliances.

Two examples of the application of the SQALE method are described, to the AdaControl tool itself, and to the Ada Web Server framework. Both demonstrate the value of measuring the software, pinpointing testability, reliability and maintainability issues that, once corrected, will raise the quality of the software by more than half.

Finally, since the indices of the SQALE method are independent of the target language, once computed, the results provided demonstrate the low quality debt remaining in an Ada application, quality debt which can be comparably estimated in other applications in other languages. SQALE for Ada can be one of the benchmark tools to help promote the use of Ada.

## References


1. J-L Letouzey, Th. Coq, The SQALE Models for assessing the quality of software source code, DNV Paris, white paper, September 2009
2. J.-L. Letouzey, Th. Coq, The SQALE Models for Assessing the Quality of Real Time Source Code, ERTSS 2010, Toulouse, September 2010
3. J.-L. Letouzey, Th. Coq, The SQALE Analysis Model - An Analysis Model Compliant with the Representation Condition for Assessing the Quality of Software Source Code, VALID 2010, Nice, August 2010
4. http://www.sqale.org.
5. P.B. Crosby, Quality is free: the art of making quality certain, ISBN 0-07-014512-1, McGraw-Hill, New-York, 1979
6. Harold A. Linstone, Murray Turoff, The Delphi Method: Techniques and Applications, Adison-Wesley, Reading, Mass, 1975.
7. Thomas L. Saaty, Fundamentals of Decision Making and Priority Theory, RWS Publications, Pittsburgh, 2001
8. J. A. McCall, P. K. Richards and G.F. Walters, Factors in Software Quality, The National Technical Information Service, No. Vol 1, 2 and 3, 1977
9. B. W. Boehm, J. R Brown, H. Kaspar,., M. Lipow,., G. McLeod,., and M. Merrit, Characteristics of Software Quality, North Holland, 1978
10. Th. McCabe, A. H. Watson, Structured Testing: A Testing Methodology using the Cyclomatic Complexity Metric, National Institute of Standards and Technology, Special Publication 500-235, 1996
11. S. R. Chidamber, C.F. Kemerer, A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering, Vol 20, N 6, PP 476-493, June 1994
12. N.E. Fenton, S. L. Pfleeger, Software Metrics: A rigourous Practical Approach, second edition, ISBN 053495425-1,PWS Publishing Company, Boston, 1997
13. Brown et al, Anti patterns : refactoring software, architectures and projects in crisis, ISBN 978-0-471-19713, John Wiley, 1998

14. http://www.adalog.fr/adacontrol2.htm
15. J-P. Rosen, "AdaControl: a free ASIS based tool", presentation at FOSDEM, Brussels, Belgium, February 2006.
16. J-P. Rosen, "On the benefits for industrials of sponsoring free software development", Ada User Journal, Volume 26, n 4, december 2005.
17. M. Jemli and J-P. Rosen, "A Methodology for Avoiding Known Compiler Problems Using Static Analysis", proceedings of the ACM SIGAda Annual International Conference (SIGAda 2010), ACM Press, ACM order number 825100, Fairfax, USA, October 24-28, 2010.
18. PMD-CPD site: http://pmd.sourceforge.net/cpd.html
19. Ada Web Server site: http://libre.adacore.com/aws/
20. J-P. Rosen, "Developing a Web server in Ada with AWS", Ada User Journal, Volume 25, n3, September 2004.
21. ISO/IEC 15291:1999. Information technology Programming languages Ada Semantic Interface Specification (ASIS)