Managing Technical Debt with the SQALE Method

Jean-Louis Letouzey, Michel Ilkiewicz
inspearit
Arcueil, France
jean-louis.letouzey@inspearit.com
michel.ilkiewicz@inspearit.com

Keywords – Technical Debt; quality; source code; quality model; analysis model; SQALE.

INTRODUCTION

It's pretty easy to get a rough estimate of the Technical Debt of an application. What is more difficult is to manage the debt properly. The latest version (V1.0) of the SQALE method (Software Quality Assessment Based on Lifecycle Expectations) allows an accurate estimation of the debt. It also includes new indexes and indicators to manage the debt. We will show through examples how they allow to analyze the nature of this debt and to prioritize remediation actions depending on the project's objectives and constraints.

We will not dwell in detail on the concept of Technical Debt. The analogy initiated by Ward Cunningham (1) has been widely developed and is now the subject of numerous blogs, articles (2, 3, 4) and also a book (5).

What you should remember is that the Technical Debt is the result of poor code quality. Specifically, Ward Cunningham used the term "not right code" and said:

« Every minute spent on not right code counts as interest on that debt."

In the context of this article, we limit ourselves to the Technical Debt associated to the source code, whether its origin is voluntary or not. Specifically, if a project has set its definition of "right code", any violation of this definition creates debt and the amount of debt is equal to the remediation cost. The Technical Debt of a project, an application, a portfolio is equal to the remediation cost of all violations in the code.

WHAT TECHNICAL DEBT MANAGEMENT MEANS

If there is some consensus on the definition of Technical Debt associated to the source code, there is, to our knowledge, no accepted definition of what "Technical Debt Management" means. We will try to define it briefly here.

Technical Debt Management means at minimum:

- Establish and publish the list of bad coding practices that create debt.
- Establish and publish the estimation model that transforms the non-compliance findings into the amount of Technical Debt.
- Set targets for debt. Specify what level and what kind of debt is acceptable for the project or the organization.
- Monitor this debt over time sufficiently frequently to be able to react quickly.
- Analyze and understand this debt in order to provide rationale for decision. For example, analyze the debt according to its age or origin (i.e. identify the amount related to architecture issues compared to other issues like presentation or format). Also analyze it in terms of potential impact.
- Reimburse it. If debt has exceeded the target, fix non-conformities to return within acceptable limits. Such reimbursement must take into account specific constraints of the project (deadline, budget, impact).
- Use the Technical Debt as input for governance of application assets. Analyze the debt of an application in correlation with other information such as business value or quality perceived by users.
- Institutionalize the previous practices. Put in place tools and processes so that they produce the benefits of a proactive Technical Debt management. Institutionalization should cover development teams but also the entire hierarchy concerned by the application portfolio.

Characteristic	Requirement	Remediation Micro-cycle	Remediation function
Maintainability	There is no commented out block of instruction	Remove (no impact on compiled code)	2 mn per occurrence
Maintainability	Code indentation shall follow a consistent rule	Fix with help of the IDE feature	2mn per file, whatever the number of violations
Changeability	There is no cyclic dependency between packages	Refactor with IDE and write tests	1h per filr dependy to cut
Reliability	Exception handling shall not catch NPE	Rewrite code and associated test	40 mn per occurrence
Reliability	Code shall override both equals and hashcode	Write code and associated test	1h per occurrence
Reliability	There is no comparison between floating point	Rewrite code and associated test	40 mn per occurrence
Reliability	There is no iteration variables modified in the body of a loop	Rewrite code and associated test	40 mn per occurrence
Reliability	All files have unit testing with at least 70% code coverage	Write additional tests	20 mn per uncovered line to achieve 70%
Testability	There is no method with a cyclomatic complexity over 12	Refactor with IDE and write tests	1h per occurrence, if measure is <24, 2h if over
Testability	There is no cloned parts of 100 tokens or more	Refactor with IDE and write tests	20 mn per occurrence

TABLE 1. Some requirement samples, their mapping within a SQALE Quality Model and their associated remediation function.

ESTIMATING TECHNICAL DEBT

The SQALE method was developed by inspearit (at the time called DNV ITGS) to measure and manage as objectively as possible the quality of source code delivered by projects. This method is published under an open source license and is royalty free.

As it is based on the concept of Technical Debt, it benefits from the success of the concept:

- It has been the subject of a thesis (6).
- It is implemented by multiple tool vendors. Examples in this article were produced using the SQALE plugin of the Sonar tool (7).
- It is used by many organizations worldwide to monitor Technical Debt on a daily basis.

The definition Document of the method, the list of available tools and complementary information are available on the official website (8).

The method is based on nine principles and four concepts. Without attempting to be exhaustive, we will explain most of them in this article with the objective of demonstrating and illustrating through examples how they help manage Technical Debt.

Let's see what the SQALE method requests for identifying and estimating the Technical Debt of an application or a portfolio.

The project or the organization must start by making a list of non-functional requirements which are the definition of "right code". In the method, it is called the Quality Model. This definition will serve as a reference to estimate the Technical Debt of the code. Any non-compliance creates debt and, on the opposite, there is no debt without breach of at least one of the requirements. This is a contract for the development team. Its contents must be clear, verifiable and non-redundant.

These requirements may cover implementation, naming and presentation. As suggested by (9), it is also important to include architectural and structural requirements. Table 1 gives some examples of requirements.

Using these requirements, the SQALE method requests the project or the organization to develop a model for estimating the debt. For this, they must associate each requirement to a remediation function. It turns the number of non-compliances into a remediation cost. This can be a simple multiplication factor or a more complex function. It is important to have a remediation function for each requirement because the remediation cost varies widely depending on the nature of activities to be performed during remediation.

Indeed, the remediation workload is highly dependent on what we call "remediation lifecycle".

For example, fixing badly indented lines will be done very quickly, often with the help of features included in the IDE. There will be no impact on unit tests and it won't affect the compiled code. The "remediation lifecycle" is simple. Most requirements related to presentation will need the same "remediation lifecycle" and will be associated with the same remediation function.

In contrast, removing redundant code (resulting from copy / paste) will require a more complex lifecycle. One will have to refactor classes, probably create and debug new tests before delivering a new version of the code.

The precision of Technical debt estimations is directly linked to the care taken to define and validate the remediation functions.

Table 1 gives examples of requirements and associated remediation functions.

Once we have defined the Quality Model and remediation functions, the calculation of the Technical Debt is simple. We run the code through the analysis tools and use remediation functions to work out remediation costs for each element in the scope of the analysis. Technical Debt is the sum of remediation costs for all non-compliances. In the SQALE method this debt is called SQALE Quality Index (SQI).

If one has the right tools, it is easy to monitor at every compilation or release the amount of Technical Debt of the code. We can also divide that amount by the code size (expressed for example in function points or thousands lines of code) to obtain the density of the analyzed code debt. Debt densities are very useful to compare teams or organizations. However these figures are not sufficient to analyze in detail the nature of the debt. They do not tell you where to begin reimbursement either.

ANALYSING THE DEBT

The SQALE method defines additional indexes and indicators to analyze and understand this debt. For this, the method organizes and groups requirements according to a specific chronology.

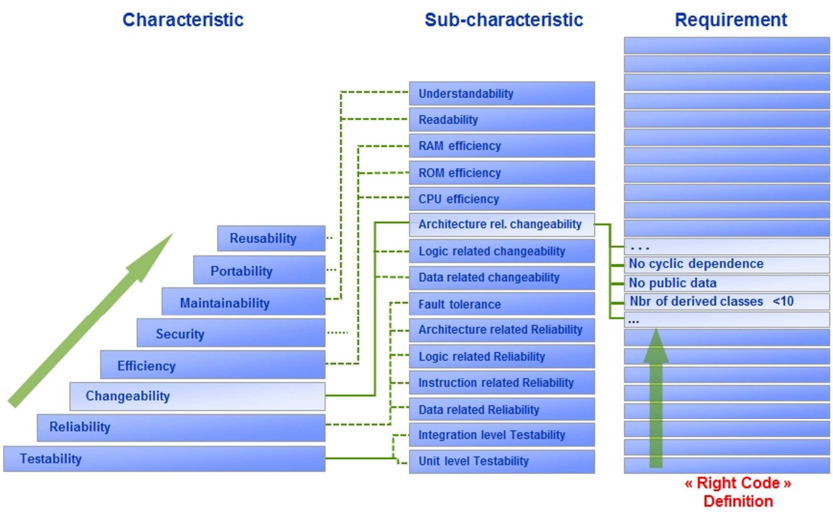

FIGURE 1. The organization of "right code" requirements in a chronological order.

The SQALE method identifies eight quality characteristics as shown in Figure 1. Their choice and the order in which they are organized have already been explained (10). What you should remember is that the testability is the foundation upon which all other characteristics rely. Testability is chronologically the first characteristic you need. For example, it will be very difficult to make an untestable component reliable.

We therefore need to associate each requirement in the definition of "right code" to a quality characteristic: the one that would be impacted in the case of a requirement violation. If a requirement impacts more than one characteristic, the method tells you to associate it to the lowest characteristic in the chronology. This allows to work out a debt index for each quality characteristic.

The method uses an indicator to represent the specific distribution of Technical Debt for each retained characteristic. This SQALE indicator called Pyramid (for which we give two examples in Figure 2) can be read in two ways:

- The first way is the analytic view
- The second way is the consolidated view

As a concrete example, we will look at the pyramid from project A in Figure 2 and see all the information it gives.

Let's start with the analytic view, that is to say, the distribution of debt by characteristic.

In the example shown, the debt related to reliability is 18.2 days. If it exceeds the objectives set, the graph can identify and initiate training or coaching on one or more topics that are the cause of this debt (i.e. exception handling, dangerous "cast"...). These targeted actions should contain the evolution of debt and improve reliability of delivered code.

The graph also tells us that violations, for a total of 7.1 days, are linked to code maintainability. Since this concerns only maintainability by third parties, which in our case is not an immediate concern, we can ignore this part of the debt and delay without risk remediation of the related violations.

Let's see now how to use the consolidated view of the pyramid, that is to say, when for a given characteristic we add the debt of all lower characteristic levels. This is shown by numbers in the right columns within Figure 2.

Take the example of the consolidated changeability of Project A which is 19.4 days.

Agile projects generate a large number of change cycles to the code. The necessary quality characteristics to support these developments are testability, reliability and changeability. If the debt of the code for these three characteristics is too high, then developers will be slowed down in their productivity. Their code is not "agile" enough.

In Figure 2, the two projects have comparable Technical Debts but different distribution profiles. Project A code is more "agile" than project B.

With a little experience, an organization can establish the threshold beyond which it is not recommended to move the maintenance of an application to agile mode.

Finally, the pyramid also gives us the order in which the remediation must be done. If the debt is not very high, or if there is enough time to repay all the debt, the pyramid gives the natural remediation order.

You should follow the Pyramid chronology and start fixing testability issues.

If you do not follow the order suggested by the Pyramid, you could waste time. You might correct reliability problems or maintainability within code portions that have a testability debt and will need to be refactored or deleted because they are too complex or redundant and so lose your previous work.

Overall, to summarize, the SQALE Pyramid provides technical rationale for decisions. It represents the "technical perspective" of the debt.

OPTIMIZED PAY BACK

Unfortunately, in many cases, we do not have enough time to repay the entire debt, or even enough time to bring it down to the acceptable limits defined for the project. The SQALE method defines a different perspective and other indexes and indicators to address this issue.

In the same way the method requires that you associate a remediation function, it also requires you to associate a non-remediation function. This is used to quantify all resulting costs of the delivery of one or more

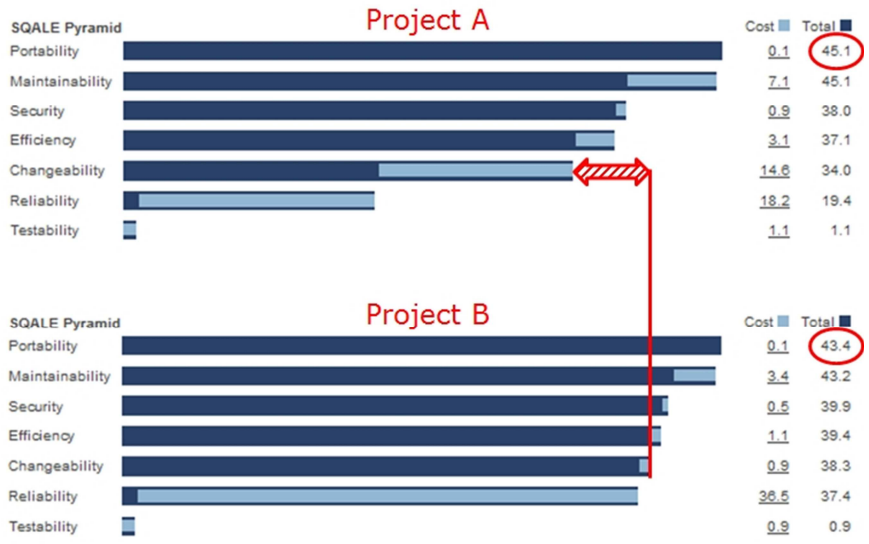

FIGURE 2. Samples of SQALE Pyramid. These 2 projects have a similar amount of Technical Debt but different distribution profiles.

non-conformities. These include for example:

- Cost to locate and fix a bug resulting from the delivery of a non-conformity. Possible income losses incurred.
- Cost of additional maintenance resources.
- Cost of additional resources (CPU, memory) caused by a non-compliance

In other words, the non-remediation function estimates the penalty that the Product Owner (or someone who represents the Business) may claim as compensation for accepting violations. This will cover all real or potential damage that could result from non-quality.

If the compensation amount is less than this, he should not accept delivery.

As in practice it is difficult to estimate and accurately model the full financial consequences of a violation, we can implement a simple but just as powerful solution.

It is possible to classify the requirements into categories such as "blocking", "critical", "major" and associate an identical symbolic cost or penalty for each class. What is important is that these amounts represent the relative importance of these different categories. Table 2 gives examples of such non-remediation factors.

Because the non-remediation costs are not established on an ordinal scale but on a ratio scale, we have shown (10) that we can aggregate the measures by addition and comply with the measurement theory and the representation clause.

The SQALE method defines an index that sums all the non-remediation costs associated with a given perimeter. This is the SBII (SQALE Business Impact Index). This index quantifies the business impact of the findings made on the code. It represents the business perspective of non-quality.

Let's see how we use this perspective and this index to optimize the debt repayment.

We saw that when we had the necessary budget, remediation order was given directly by the SQALE pyramid.

Now imagine that we are in the case of project A shown in Figure 1. Suppose the agreed limit in the "Definition of Done" is 5 days. We can see that 40.1 working days are needed to return to target. Suppose that unfortunately there are only 10 working days available before the imposed delivery date. In this case, we will have to compromise and make an optimal use of these 10 days.

The remediation priority will be established by taking into account the business impact of non-conformities. We will select priority actions giving the highest return, that is to say, having the best ratio Non-remediation cost / remediation cost.

For this, SQALE defines the Debt Map graph on which an item (either a file, a component or an application) is represented on two axes, the Technical Debt and the Business Impact. An example is given in Figure 4. We will start with the top left quadrant and select items with highest slope as far as available budget (as shown by red line).

DEPLOYING THE SQALE METHOD

As the SQALE method is open source and royalty free, some organizations have built their own solution by loading results of different analysis tools in a Business Intelligence tool. But most organizations use available SQALE

NC Type	Description	Type Sample	Non-Remediation Factor
Blocking	Will or may result in a bug	Division by zero	5000
High	Wil have a high/direct impact on the maintainance cost	Copy and paste	250
Medium	Will have a medium/potential impact on the maintainance cost	Complex logic	50
Low	Wil have a low impact on the maintainance cost	Naming convention	15
Report	Very low impact, it is just a remediation cost report	Presentation issue	2

TABLE 2. Sample of non-remediation factors.

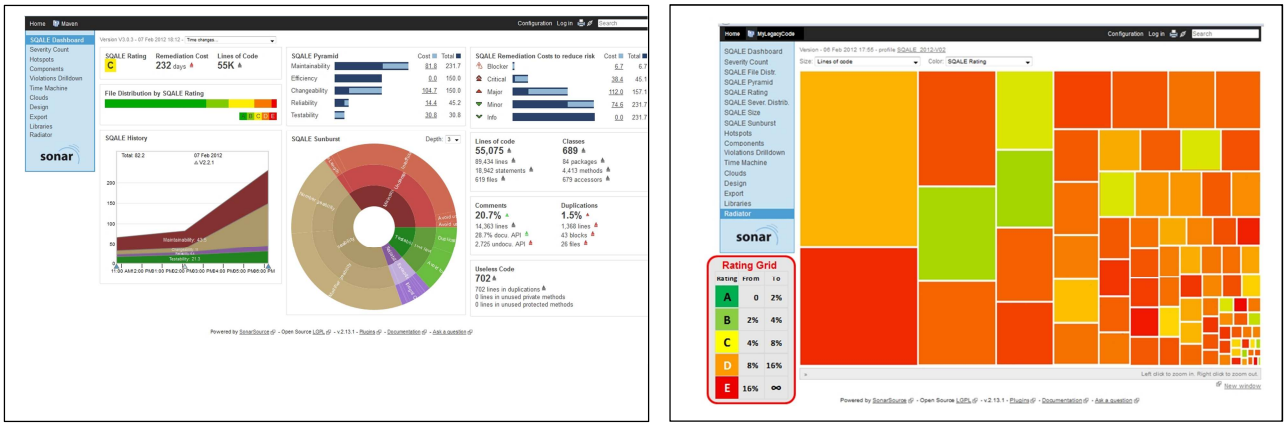

FIGURE 3. 2 sample Dashboards allowing the management of Technical Debt

compatible tools. When the perimeter is small (less than 50 developers), using the SQALE default settings from the tools allows a very quick implementation and immediate results.

When the perimeter is more important (that is to say 100 developers and more), deployment becomes a transverse project. We helped 6 large organizations on such projects and provide here a summary of our findings and our recommendations in this type of context.

1°) Managers and upper management understand and appreciate the concept of Technical Debt. They want to integrate this information in their performance indicators. But to do this, it is necessary that all projects in the scope use the same "right code" definition and consistent remediation and non-remediation functions. This should be established independently of the location and language used.

By facilitating workshops with experts from different units, you can achieve a general consensus on the content of the definition of "right code."

In our experience, we recommend limiting the models to a number of requirements between 50 and 100. Similarly, it is important to involve experts to identify the remediation lifecycle and associated remediation functions.

2°) Pay specific attention to the process aspect. It is desirable to define and communicate the implementation and management process for Technical Debt. This process will typically answer questions such as:

- Who decides the Technical Debt goals for new projects?
- Is a project allowed to remove or add specific requirements in the definition of "right code"?
- What are the Technical Debt management rules for the legacy code?
- What are the implications for subcontracted projects?

3°) Test on a pilot before full deployment. This helps to check and validate the Quality models and, above all, to calibrate the remediation and non-remediation functions on representative projects.

4°) Automate the production of the debt indicators (if possible within the continuous integration flow) and make sure it is produced at least daily with no additional workload for users.

5°) Allocate time for training and coaching the different stakeholders in Technical Debt and the SQALE method according to their profile. As an example:

- A one-day training for experts participating in workshops
- A 45 minutes awareness session for top managers.

6°) Organize an annual review and maintenance of models.

When all these recommendations are followed, the Technical Debt becomes very visible. We found that it creates virtuous effects among developers. They start challenging

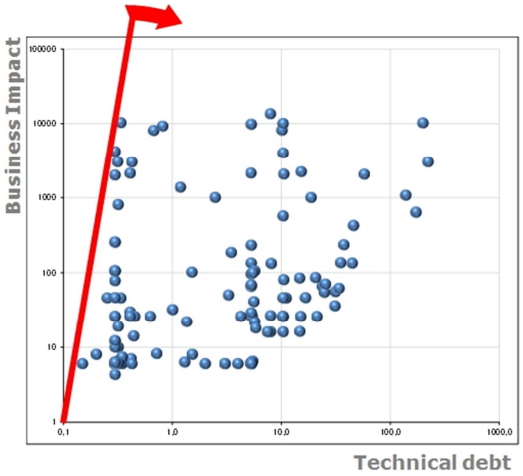

FIGURE 4. A SQALE Debt Map sample. This graph provides remediation priority when the remediation budget needs to be optimized.

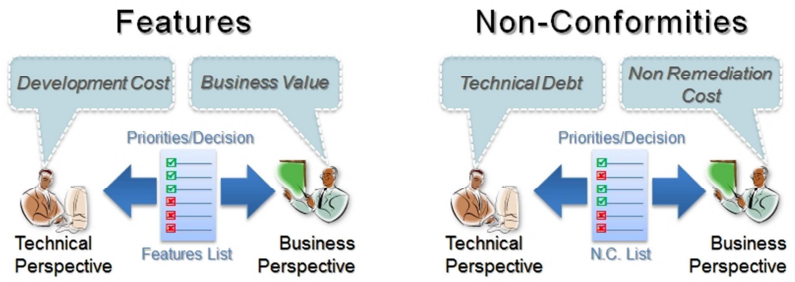

FIGURE 5: Comparison between prioritization of remediation and prioritization of features.

their peers and other projects. This triggers quick improvement in the quality of the code produced.

CONCLUSION

As we have shown, the SQALE method analyzes the nature and impact of Technical Debt from two perspectives, the technical perspective (with the remediation costs and the characteristic distribution) and the business perspective (with non-remediation costs).

Combining the two perspectives is powerful and provides prioritization logic. In fact, as shown in Figure 5, this logic is similar to the one recommended by the Agile community. It uses the cost of development and business value to prioritize the implementation of user stories.

With this enhancement, the new version of the method covers the weaknesses reported by some users. Today, what the user community is still expecting and will welcome is a standardized definition of "right code".

ACKNOWLEDGEMENTS

We thank the reviewers who provided constructive comments and suggestions for improving this article and especially Paul Bricknell.

REFERENCES

[1] W. Cunningham, "The WyCash portfolio management system", ACM SIGPLAN OOPS Messenger, vol. 4(2), pp. 29–30, 1993.

[2] M. Fowler, "Technical Debt Quadrant, 2009. http://martinfowler.com/bliki/TechnicalDebtQuadrant.html

[3] S. McConnell, "What is Technical Debt? Two Basic Kinds". http://forums.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx

[4] I. Gat, "Technical Debt", IT Journal, vol. 23, October 2010.

[5] C. Sterling, "Managing Software Debt"; Addison-Wesley, 2010.

[6] . J. H. Hegeman, 'On the Quality of Quality Models", Master thesis, University of Twente, July 2011.

[7] Sonarsource, the Sonar company, SQALE plugin overview. http://www.sonarsource.com/plugins/plugin-sqale/overview/

[8] J.-L. Letouzey, The SQALE Method – Definition Document, Version 1.0, January 2012. http://www.sqale.org/

[9] N. Brown, M. Gonzalez, Ph. Kruchten, R. Nord, I. Ozkaya, "Managing Structural Technical Debt", OOPSLA 2011, Portland, 2011.

[10] 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.