Managing Technical Debt with the SQALE Method

Jean-Louis Letouzey and Michel Ilkiewicz, Inspearit

// *The SQALE (software quality assessment based on life-cycle expectations) method provides guidance for managing the technical debt present in an application's source code.* //

옳지 않은 코드에 대한 모든 분은
debt에 대한 이자로 계산된다.

THE CONCEPT OF technical debt has already been researched extensively.[1–5] Remember, technical debt and code quality are two connected concepts. Specifically, Ward Cunningham said, "Every minute spent on not right code counts as interest on that debt."[1] (For more information on SQALE, see the "SQALE in Action" sidebar.)

In this article, we focus on the technical debt found in an application's source code and present the SQALE (software quality assessment based on life-cycle expectations) method, the goal of which is to estimate and manage technical debt. Although there are numerous methods for estimating source code's technical debt,[6,7] the SQALE method contains a precise model for estimating it. Moreover, it provides concrete support and priorities for planning refactoring or improvement.

SQALE를 이용한 debt를 추정하고, 관리한다.
많은 방법이 있는데, SQALE가 젤 정확함

What Technical Debt Management Means

There is, to our knowledge, no widely accepted definition of what "technical debt management" means. For our purposes, technical debt management includes (at a minimum)

Techinical Debt의 의미는 많음.
최소 다음을 포함함

- establishing and publishing a list of bad coding practices that create debt;
- establishing and publishing the estimation model that transforms noncompliance findings into the amount of technical debt;
- setting targets for debt and specifying what level and what kinds of debt are acceptable for the project or organization;
- monitoring this debt frequently enough to be able to react quickly;
- analyzing and understanding this debt to estimate potential impact and provide rationale for decisions;
- repaying debt if it exceeds the target and fixing nonconformities to return within acceptable limits (such repayment must take into account specific project constraints, including deadline, budget, and impact);
- using the technical debt as input for governance of application assets and analyzing an application's debt in correlation with other information such as business value or user-perceived quality; and
- institutionalizing the previous practices and putting in place tools and processes to produce the benefits of proactive technical debt management.

On that last point, institutionalization should cover not just development teams but also the entire hierarchy affected by the application portfolio.

Estimating Technical Debt

The project or organization must start by making a list of nonfunctional requirements that define the "right code." We call this the *quality model*. This definition will serve as a reference

0740-7459/12/$31.00 © 2012 IEEE

비기능 요구사항 - > quality model - > noncomplicance - > techinical debt - > 요구사항 위반 없이 debt도 없다.

to estimate the code's technical debt. Any noncompliance creates debt, and there's no debt without breach of at least one requirement. This is a contract for the development team, so its contents must be clear, verifiable, and not redundant. These requirements could cover implementation, naming, and presentation, but it's also important to include architectural and structural requirements.[8] Table 1 gives some requirements examples.

Using these requirements, the SQALE method asks projects or organizations to develop a debt-estimating model. For this, they must associate each requirement with a remediation function, which turns the number of noncompliances into a remediation cost. This can be a simple multiplication factor or a more complex function. It's 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 the remediation life cycle—equivalent to a development life cycle, but applied to a fix. For example, fixing badly indented lines will be done very quickly, often with the help of features included in the integrated development environment. There will be no impact on unit tests, and such a fix won't affect compiled code. Most requirements related to presentation will need the same remediation life cycle and will be associated with the same remediation function.

In contrast, removing redundant code (which can result from copying and pasting) requires a more complex life cycle. We would need to refactor classes, as well as create and debug new tests before delivering a new version of the code.

SQALE IN ACTION

Inspearit (formerly called DNV ITGS) developed the SQALE method to measure and manage as objectively as possible the quality of source code that projects deliver. The method was designed to be as generic as possible and is applicable to any kind of language and any development methodology. It's published under an open source license and is royalty free. The definition document of the method, the list of analysis tools that implement the method (there are four as of this writing), and additional information are available on the official website (www.sqale.org). The method is based on nine principles and four concepts. Without attempting to be exhaustive, we explain most of them in the main article with the objective of demonstrating and illustrating how they help manage technical debt.

Since its publication under an open source license in August 2010, use of and support for the SQALE method has increased, and multiple vendors now implement it. The SQALE plugin of the Sonar tool produced the examples in the main text (www.sonarsource.com/products/plugins/governance/sqale), and many organizations worldwide use it to monitor technical debt on a daily basis. A thesis based on actual user data has validated the correlation of SQALE indicators and the perceived quality.[1]

Reference

1. J.H. Hegeman, "On the Quality of Quality Models," master's thesis, Formal Methods and Tools Dept., Univ. Twente, 2011.

The precision of technical debt estimations is directly linked to the care taken to define and validate remediation functions. Once we define the quality model and remediation functions, the calculation of the associated technical debt is simple. We run the code through the analysis tools and use remediation functions to work out remediation costs for each element. Technical debt is the sum of remediation costs for all noncompliances. In the SQALE method, this debt is called the SQALE quality index (SQI).

With the right tools, it's easy to monitor at every compilation or release the amount of the code's technical debt. We can also divide that amount by the code size (expressed, for example, in function points or KLOC) to obtain the debt density of the analyzed code. Debt densities are very useful when comparing teams and organizations, but they aren't sufficient for analyzing in detail the nature of the debt. They also don't tell us where to begin repayment.

Debt Analysis

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

Characteristics

The SQALE method identifies eight quality characteristics, as Figure 1 shows.[9,10] What you should remember is that testability is the foundation upon which all other characteristics rely. Testability is chronologically the

TABLE 1

Some requirement samples, their mappings within a SQALE quality model, and their associated remediation functions.*

Characteristic	Requirement	Remediation microcycle	Remediation function
Maintainability	There is no commented-out block of instruction	Remove (no impact on compiled code)	2 minutes per occurrence
	Code indentation shall follow a consistent rule	Fix with help of the integrated development environment (IDE) feature	2 minutes per file, regardless of the number of violations
Changeability	There is no cyclic dependency between packages	Refactor with IDE and write tests	1 hour per file dependency to cut
Reliability	Exception handling shall not catch null pointer exception	Rewrite code and associated test	40 minutes per occurrence
	Code shall override both equals and hash code	Write code and associated test	1 hour per occurrence
	There is no comparison between floating points	Rewrite code and associated test	40 minutes per occurrence
	No iteration variables are modified in the body of a loop	Rewrite code and associated test	40 minutes per occurrence
	All files have unit testing with at least 70% code coverage	Write additional tests	20 minutes per uncovered line to achieve 70%
Testability	There is no method with a cyclomatic complexity over 12	Refactor with IDE and write tests	1 hour per occurrence if measure is < 24; 2 hours if > 24
	There are no cloned parts of 100 tokens or more	Refactor with IDE and write tests	20 minutes per occurrence

** These remediation functions come from an organization that has instantiated the method within its context.*

first characteristic you need—it would be difficult to make an untestable component reliable.

We therefore must associate each requirement in the definition of right code to a quality characteristic that would be affected in the case of a requirement violation. If a requirement affects more than one characteristic, the method says to associate it to the lowest characteristic in the chronology. This lets us work out a debt index for each quality characteristic.

Pyramid

The SQALE method uses an indicator to represent the specific distribution of technical debt for each selected characteristic. This SQALE indicator, called the pyramid (for which we give two examples in Figure 2), can be read in two ways: through the analytic view—the distribution of debt by characteristic—or the consolidated view—the sum of debt for a given characteristic and all the characteristics that fall below it.

A Real-World Example

To show how to interpret the SQALE pyramid, let's take concrete examples from an organization that has adapted the SQALE method to its context and has therefore defined its own list of code quality requirements. It has also defined and calibrated the related remediation functions that apply to its context. This organization used these inputs to configure a static analysis tool that supports the SQALE method and to produce the two pyramid indicators.

In the analytic view, if the debt exceeds the objectives that were set for the project, we can use the graph to identify areas for training or coaching on the topics that are the cause of this debt (such as exception handling or a dangerous cast). These targeted actions should contain the evolution of debt and improve reliability of delivered code.

The analytic view of the pyramid indicator from Project A in Figure 2 shows that the debt related to reliability in this example is 18.2 days. The graph also tells us that violations for a total of 7.1 days are linked to code maintainability. Because this concerns only maintainability by third parties, which, in our case, isn't an immediate concern, we can ignore this part of the

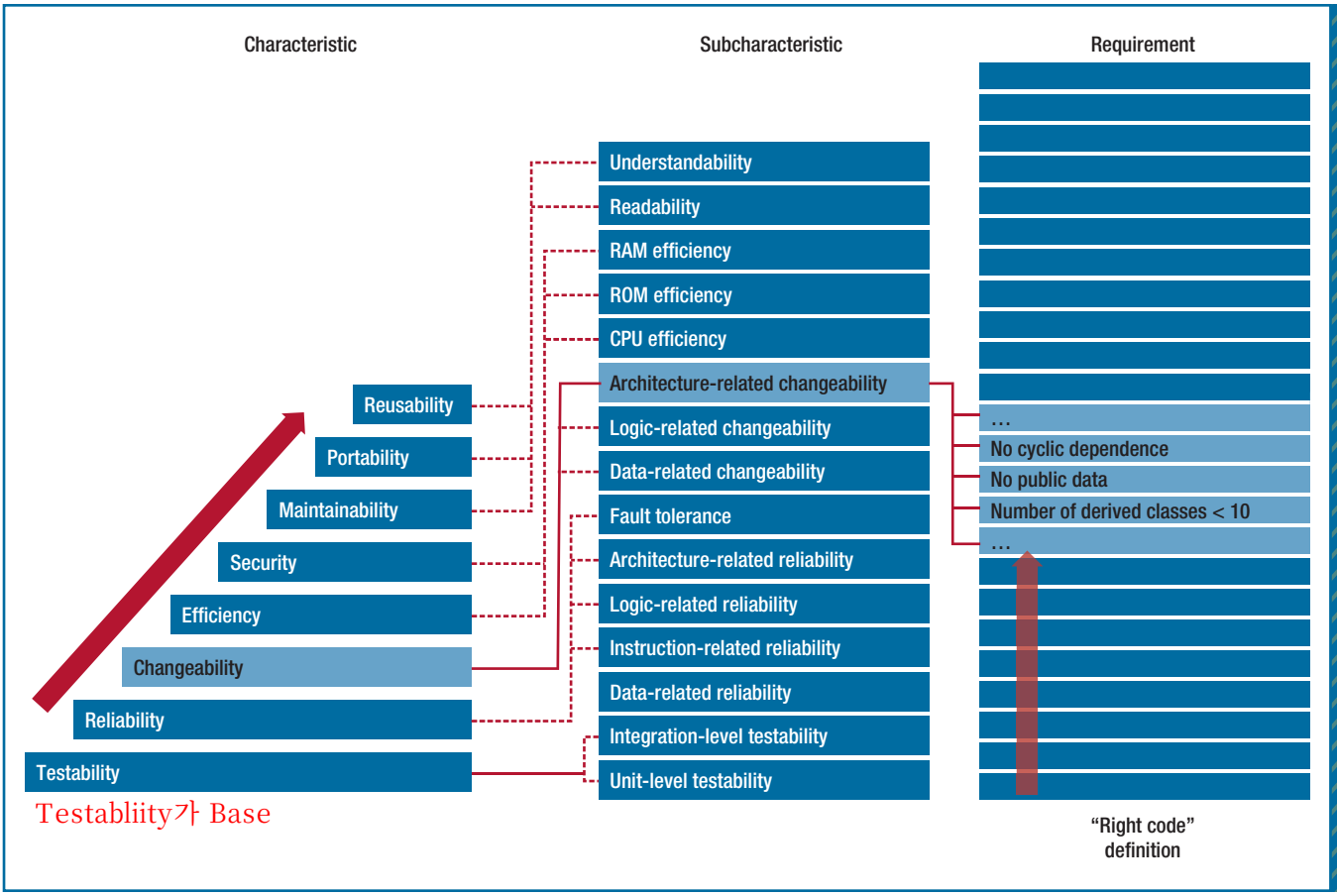

FIGURE 1. The chronological organization of "right code" requirements. Testability is at the bottom of the list because it's the foundation on which all other quality characteristics rely.

agile에서는 testability, reliability, changeability가 좋아야 됨
안그러믄, 생산성 떨어짐

debt and delay without risk remediation of the related violations.

Let's now look at how to interpret the consolidated view of the pyramid, as shown in the right columns of Figure 2. 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, developers will slow their productivity—their code isn't agile enough. With a little experience, an organization can identify the threshold beyond which it shouldn't move the maintenance of an application to agile mode.

The pyramid also gives the organization the order in which the remediation must be done if the debt isn't very high or if there is enough time to repay it all. If organizations don't follow the order suggested by the pyramid, they could waste time. They might correct reliability problems or maintainability within code portions that have a testability debt; these will need to be refactored or deleted because they're too complex or redundant.

Optimized Payback

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

In the same way that the SQALE method requires associating each requirement with a remediation function, it also requires association with a nonremediation function. It uses this to quantify all resulting costs of the delivery of one or more nonconformities, such as

- costs to locate and fix a bug resulting from the delivery of a nonconformity (possibly incurring income losses),
- costs of additional maintenance resources, and

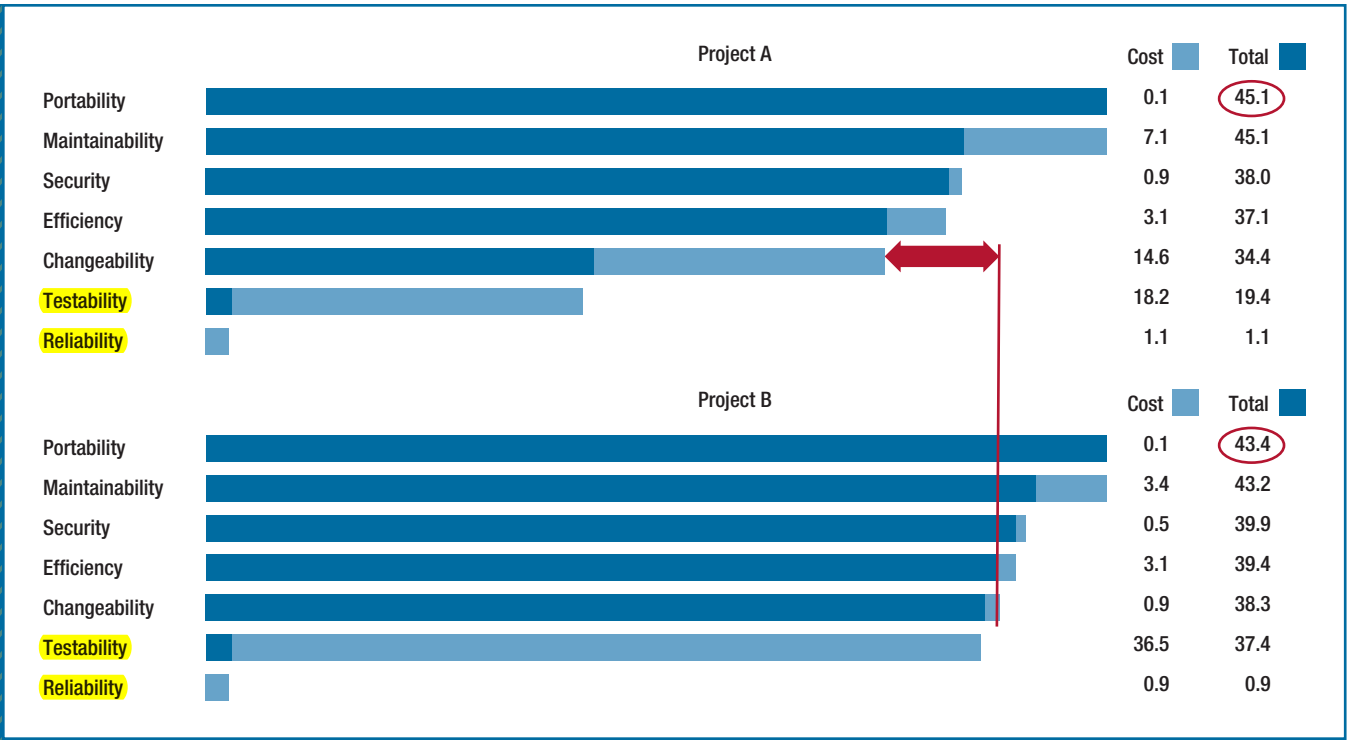

FIGURE 2. Samples of the software quality assessment based on life-cycle expectations (SQALE) indicator called the pyramid. These two projects have a similar amount of technical debt but different distribution profiles.

- costs of additional noncompliance-related resources (such as CPU or memory).

In other words, the nonremediation function estimates the penalty that the product owner (or someone who represents the organization) might claim as compensation for accepting violations. This should cover all real or potential damage that could result from low quality in the delivered code. If the compensation amount is less than this, the product owner shouldn't accept delivery.

Requirement Classification

Unfortunately, it's difficult to model and accurately estimate the full financial consequences of a violation. Instead, we can implement a simple but equally powerful solution. We classify the requirements into categories (such as blocking, critical, or major) and associate an identical symbolic cost or penalty for each class. What is important in doing this is that these amounts represent the relative importance of each category. Table 2 gives examples of such nonremediation factors.

Because the nonremediation costs are established on a ratio scale rather than an ordinal scale, we can aggregate the measures via addition and comply with the measurement theory, specifically with the representation clause (one of the requirements needed to obtain an objective measurement system[11]).

The SQALE Business Impact Index

The SQALE method defines an index that sums all the nonremediation costs associated with a given scope. This SQALE business impact index quantifies the business impact of the findings made on the code and represents the business perspective of nonquality.

An Example

We saw earlier that when we had the necessary budget, remediation order was given directly by the SQALE pyramid. Now imagine that we're in the case of Project A from Figure 2. Suppose the agreed technical debt goal is no more than five days. The project needs 40.1 working days to return to target, but let's suppose that there are only 10 working days available before the imposed delivery date. In this case, we would need to compromise and make an optimal use of those 10 days.

We establish the remediation priority by taking into account the business impact of nonconformities. We select priority actions giving the highest return—that is, giving the best nonremediation cost to remediation cost ratio.

For this, SQALE defines the debt map graph on which a file, component, or application is represented on two

TABLE 2

Sample of nonremediation factors issued from a specific context.

Category	Description	Sample type	Nonremediation factor
Blocking	Could result in a bug	Division by zero	5,000
High	Will 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	Will have a low impact on the maintainance cost	Naming convention	15
Report	Has very low impact—it's just a remediation cost report	Presentation issue	2

axes: the technical debt and the business impact (see Figure 3). We start with the top left quadrant and select items with the highest slope for available budget (as shown by the shaded area).

Deploying the SQALE Method

장점중에 하나

Because the SQALE method is open source and royalty free, some organizations have built their own solution by loading results of different analysis tools to a business intelligence tool, but most organizations already use available SQALE-compatible tools. When the deployment scope is small (fewer than 50 developers), using COTS tools with default SQALE models allows a very quick implementation and immediate results. When the scope is larger (more than 100 developers), deployment becomes a transverse project.

We helped six large financial and industrial organizations on such projects, which allowed us to form several recommendations for this context. When all these recommendations are followed, the technical debt becomes very visible. We found that this creates virtuous effects among developers—they start challenging their peers and other projects. This triggers quick improvement in the quality of the code produced.

Share the Concept

Managers understand and appreciate the concept of technical debt. They want to integrate this information in their performance indicators. But to do this, all projects in the scope must use the same right code definition and consistent remediation and nonremediation 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 this definition to a number of requirements between 50 and 100. Similarly, it's important to involve experts to identify the remediation life cycle and associated remediation functions.

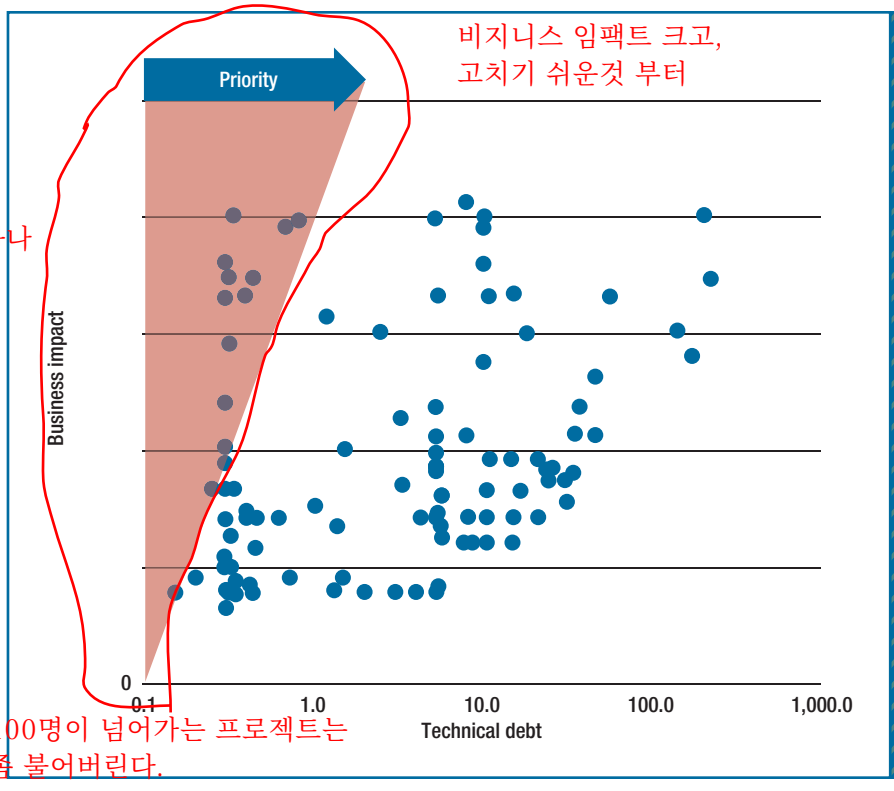

FIGURE 3. A SQALE debt map example. This graph provides remediation priority when the remediation budget needs to be optimized.

단점

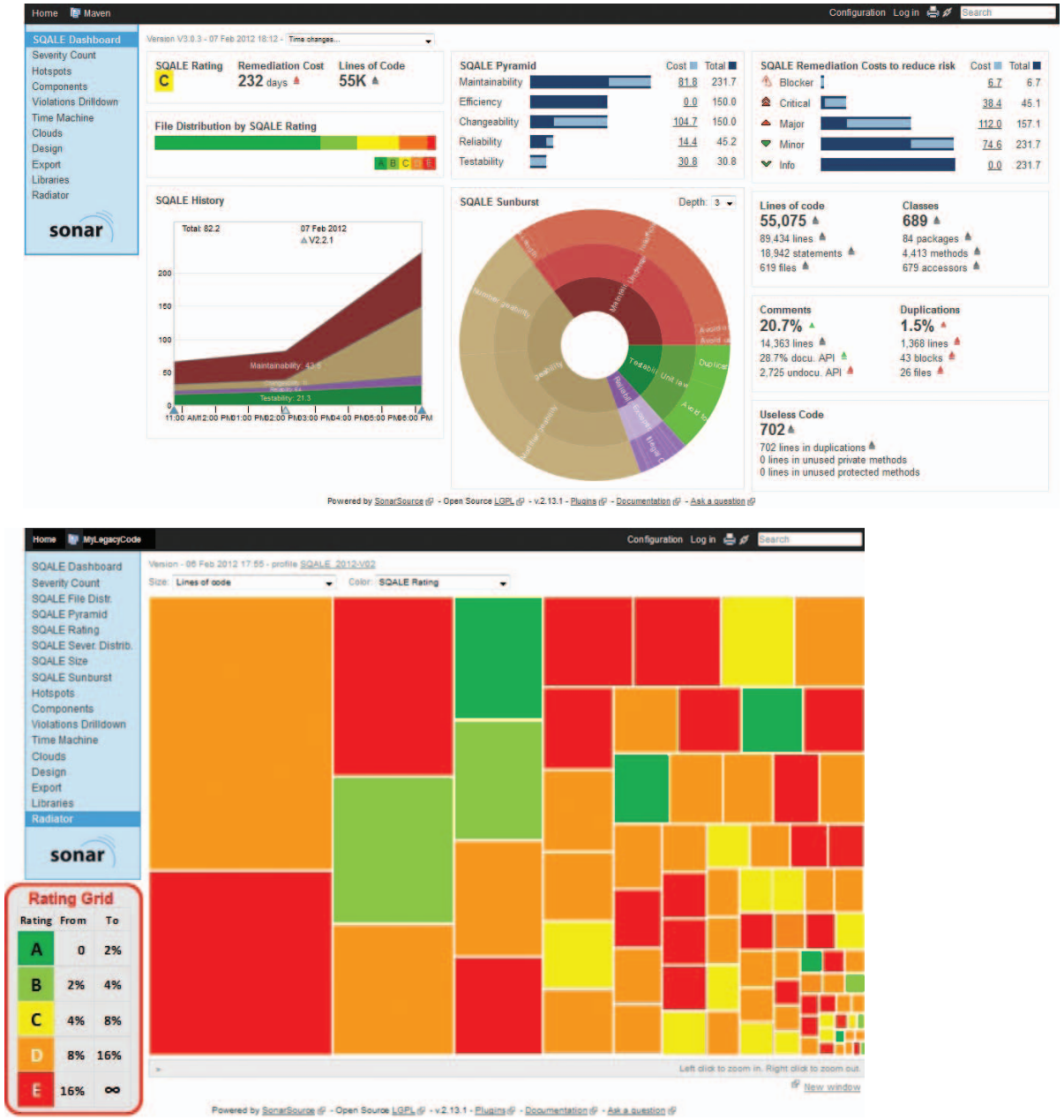

FIGURE 4. Two example dashboards allowing the management of technical debt.

하기전에 아래 것들부터 정해놔라

Pay Attention to the Process

We recommend defining and communicating the implementation and management process for technical debt. This process typically answers 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?

We recommend deploying the process

internally before extending it to subcontracted activities.

Test

Test on a pilot before full deployment. This helps check and validate the quality models and, above all, to calibrate the remediation and nonremediation functions on representative projects.

Define

Define standardized indicators and dashboards (like in the examples in Figure 4) common to the entire organization. Such dashboards establish a visual language and allow fast and transparent communication between people and entities.

Automate

Automate the measurement process as much as possible. Make sure dashboards are produced at least daily with no additional workload for users.

Budget Time

Allocate time for training and coaching the different stakeholders in technical debt and the SQALE method according to their profile—for example, a one-day training for experts participating in workshops and a 45-minute awareness session for top managers.

Review

Organize an annual review and maintenance of models.

Today, users expect and welcome business-specific definitions of "right code" and associated remediation functions. Delivering this is possible if early adopters of the method are willing to share their experience and feedback. One possible way to achieve this is to launch a SQALE user group.

Because the method is currently limited to the debt present in the source code, it would also be interesting to study how the method could be extended to support other types of technical debt, including the nonvisible part of the architectural debt in the code. Another possibility would be to include it as a component of a more generic method framework dedicated to managing all kinds of technical debt.

Acknowledgments

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

References

1. W. Cunningham, "The WyCash Portfolio Management System," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, 1993, pp. 29–30.
2. M. Fowler, "Technical Debt Quadrant," blog, 14 Oct. 2009; http://martinfowler.com/bliki/TechnicalDebtQuadrant.html.
3. S. McConnell, "Technical Debt," blog, 1 Nov. 2007; http://blogs.construx.com/tblogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx.
4. I. Gat, "Technical Debt," *Cutter IT J.*, Oct. 2010, pp. 3–6.
5. C. Sterling, *Managing Software Debt*; Addison-Wesley, 2010.
6. A. Nugroho, J.Visser, and T. Kuipers, "An Empirical Model of Technical Debt and Interest," *Proc. 2nd Int'l Workshop Managing Technical Debt*, ACM, 2011, pp. 1–8.
7. B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the Size, Cost, and Types of Technical Debt," *Proc. 3rd Int'l Workshop Managing Technical Debt*, IEEE CS, 2012, pp. 49–53.
8. "Technical Debt Evaluation (SQALE)," SonarSource, 2012; www.sonarsource.com/plugins/plugin-sqale/overview.
9. R. Nord et al., "In Search of a Metric for Managing Architectural Technical Debt," to be published in *Proc. Joint l0th Working IEEE/IFIP Conf. Software Architecture and 6th European Conf. Software Architecture* (WICSA/ECSA 2012), IEEE CS, 2012.
10. *ISO/IEC Std. 25010, Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (Square)—System and Software Quality Models*, Int'l Org. for Standardization, 2011.
11. J.-L. Letouzey and T. Coq, "The SQALE Analysis Model: An Analysis Model Compliant with the Representation Condition for Assessing the Quality of Software Source Code," *Proc. 2nd Int'l Conf. Advances in System Testing and Validation Lifecycle* (VALID 10), IEEE, 2010, pp. 43–48.

ABOUT THE AUTHORS

JEAN-LOUIS LETOUZEY is a consultant at Inspearit. His research interests include code quality, static analysis, and change management. Letouzey received a master's degree in engineering from the Centrale Graduate School of Lille. Contact him at jean-louis.letouzey@inspearit.com.

MICHEL ILKIEWICZ is a senior consultant at Inspearit. His research interests include process improvement, code quality, and system safety. Ilkiewicz received a master's degree in software, systems and networks from the Supélec Graduate School of Paris. Contact him at michel.ilkiewicz@inspearit.com.