Technical Debt Indexes provided by tools: a preliminary discussion

Francesca Arcelli Fontana, Riccardo Roveda and Marco Zanoni
Università degli Studi di Milano - Bicocca, Milan, Italy
Email: {arcelli,riccardo.roveda,marco.zanoni}@disco.unimib.it

***Abstract*—In software maintenance and evolution, it is important to assess both code and architecture in order to identify issues to be solved to improve software quality. Different tools provide some kind of index giving us an overall evaluation of a project to be used when managing its technical debt. In this paper, we outline how the indexes, that we call in general Technical Debt Indexes, provided by five different tools are computed. We describe their principal features and differences, what aspects they are missing, and we outline if (and how) the indexes take into account architectural problems that could have a major impact on the architectural debt. We show that the indexes rely on different information sources and measure different quantities.**

I. Introduction

Many tools are available, addressing software quality and architecture assessment, e.g., through metrics computations, or code and design anomalies detection, i.e., code smells [8], architectural smells [13], [9] and antipatterns [1]. Detecting these anomalies is useful to identify problems to be solved through the right refactoring steps, but this does not provide us an indication of the overall quality assessment of a project. Hence, some tools offer some kind of Technical Quality Index, often called, e.g., Technical Debt/Severity, Deficit Index, that offers an evaluation of the overall quality of an analyzed project. In the paper, with the term Technical Debt Index (TDI), we refer to any kind of quality index computed by the tools. These indexes are derived in different ways and take into account different features. We focus our attention on the TDI provided by known tools, with the aim to understand what exactly each index takes into account, what the value of the index represents, and its completeness w.r.t. the information that can be exploited to estimate Technical Debt. With these aims, we try to answer the following questions:

- Q1 How are the quality indexes the tools provide exactly computed? Which features do they take into account?
- Q2 Which index does take more into account the architectural issues and in which way?
- Q3 Which are the features not provided or taken into account by the indexes?

We decided to consider in this paper five tools that provide some kind of TDI: CAST, inFusion, Sonargraph, SonarQube and Structure101. The choice of the tools is due, with respect to our knowledge, to the availability of tools providing this kind of Index and our experimentations of all of them on several OSS projects. For example, we experimented, with the aim to refactor or evaluate some architectural problems on existing systems, SonarQube, InFusion and Structure101 on 4 projects [2], Sonargraph and Structure 101 on 2 projects [4], and Sonargraph, SonarQube and inFusion on one project [3]. In each paper, we have considered different features with different aims, that allowed us to get, we hope, a good knowledge and experience on the different tools. We experimented also CAST on some OSS and academic projects.

Obviously, other tools are available, which are able to compute a huge number of metrics, also related to architectural issues, e.g., Massey Architecture Explorer computes an Antipatterns Score [7] and the Tangledness metric [16], Lattix provides Stability, Cyclicality, and Coupling metrics, and STAN supports different R. Martin's metrics [15]. These kinds of metrics are very interesting, but they do not represent (alone) a TDI trying to summarize the overall quality of a system, as those we consider in this paper.

The paper is organized as follows: in Sec. II, we describe how the TDIs are computed by the five tools; in Sec. III, we outline their main and missing features; finally, in Sec. IV we answer the posed questions and describe future developments.

II. Technical Debt Indexes

In this section, we describe how the TDI are computed by the tools we considered, i.e., CAST 7.3.2, inFusion v.1.8.5, Sonargraph v.8.8.0, SonarQube v.5.2, Structure101 v.4.2.10071. SonarQube is free and open source, while the other ones are commercial tools. We gathered this information through the use of the tools, their documentation, and by communicating with their technical support, when needed. The data we show are related to the versions reported above.

A. CAST

CAST[1] defines Technical Debt as the future costs attributable to known structural flaws in production code that need to be fixed, a cost that includes both principal and interest. CAST estimates the amount of principal in the Technical Debt (hereafter called TD-Principal or TDP) of an application based on detectable structural problems. TDP is a function of three variables: *1)* the number of must-fix problems in an application, *2)* the time required to fix each problem, *3)* and the cost for fixing a problem. Each detected problem can affect one or more *Health Factors*: Robustness, Performance Efficiency, Security, Transferability and Changeability. Scores

[1] http://www.castsoftware.com/

978-1-5090-3854-1/16 $31.00 © 2016 IEEE
DOI 10.1109/MTD.2016.11

assigned to each internal quality characteristic are aggregated from the component to the application level and reported on a scale from high to low risk, using an algorithm that weights the severity of each violation and its relevance to each Health Factor. CAST assumes that an IT organization would fix 100% of the high severity problems, 50% medium severity, and no more than 0% of low severity. Weights are assigned to severity levels using the following schema:

- Low severity = Weight of 1, 2 or 3
- Medium severity = Weight of 4, 5 or 6
- High severity = Weight of 7, 8 or 9

To keep the estimate of TDP conservative, the tool assumes that problems can be fixed in 1 hour. CAST sets the labor rate to an average of $75 per hour. The initial formula (and parameters) used to compute TDP is the following:

$$\begin{aligned} TDP = (75\tfrac{\$}{hr}) \times (1hr.) \times \\ ((\Sigma high\_severity\_violations) \times (1) + \\ (\Sigma medium\_severity\_violations) \times (0.5) + \\ (\Sigma low\_severity\_violations) \times (0)) \end{aligned}$$

B. inFusion

inFusion[2] (IF) supports the evaluation of software quality with a focus on code smells (CS) and architectural smells (AS), called *design flaws*, detected by evaluating different metrics. Design flaws are used as the basis for the computation of the *Quality Deficit Index* (QDI), which is reported as a global score or grouped by quality dimension (Complexity, Encapsulation, Coupling, Inheritance, Cohesion). The QDI, the detected design flaws and metric values can be browsed and filtered in different ways, e.g., by quality dimension, by package, or by focusing on single flaws or flawed entities.

The QDI evaluates the impact of all detected design flaws using three factors:

- *Influence* (I_{flaw_type}) This factor expresses how strongly a type of design flaw affects four criteria of good design [6]. It uses a three-level scale (high, medium, low) to characterize the negative influence of a design flaw on each of the four criteria. The assignment of high/medium/low to each design flaw is reported in [14]. I_{flaw_type} is computed as the weighted arithmetic mean of numerical values assigned to the three levels for each of the four criteria.
- *Granularity* (G_{flaw_type}) In general, a flaw that affects methods has a smaller impact on the overall quality than one that affects classes; consequently, it is assigned a weight to each design flaw according to the type of design entities that it affects: class $\rightarrow$ 3 and method $\rightarrow$ 1.
- *Severity* ($S_{flaw\_instance}$) The first two factors refer to design flaw types, which means that all instances of the same flaw weigh equally; since not all cases are equal, for each flaw a severity score is defined, based on its most critical symptoms, measured by one or more metrics. Severity scores span the range 1–10 (low–high).

Based on the three factors, inFusion computes the Flaw Impact Score (FIS) of a design flaw instance, and derives the QDI value, combining FIS with the size (KLOC) of the system, as follows:

$$FIS_{flaw\_instance} = I_{flaw\_type} \cdot G_{flaw\_type} \cdot S_{flaw\_instance}$$

$$QDI = \sum_{k \in flaw\_instances} FIS_k / KLOC$$

C. Sonargraph

Sonargraph (SG) is meant to support quality controllers, software developers, architects, and consultants. One of its main function related to architectural debt evaluation is the ability to detect deviations from a *defined architecture*, where the developer specifies which dependencies are allowed (or not) between the elements of the system. Moreover, the tool computes different metrics and detects code smells and violations to programming best practices. All detection results can be traced across different versions of the same software.

Sonargraph's Structural Debt [10] is quantified through two measures: *Structural Debt Index* (SDI) and *Structural Debt Cost* (SDC). The first measure is a score computed as the weighted sum of the type dependencies that would need to be cut to break all cyclic package dependencies. The second measure (SDC) is computed by multiplying the SDI by a constant time amount. The tool analyzes the dependency graph to find a good breakup set to disentangle cyclic nodes. The algorithm output is a set of links that need to be removed. The SDI metric is computed by multiplying the number of links to be removed by 10 and then adding the weight (number of dependencies) for each link. As for the SDC metric, the suggested time factor is between 6 and 10 minutes for each SDI point, as a starting point. Then it is possible to define programmers' hour costs, to obtain an estimation of the cost of correcting the issues in the system. Both measures can be computed at the level of system, project or build unit.

D. SonarQube

SonarQube (SQ) is a platform to manage code quality. Its main features are the ability to check large sets of coding rules and to gather software metrics. Examples of rules are the verification of coding constraints and the verification of the range of metric values. Currently, SonarQube's TDI computation does not take into account architectural or dependency information. Each rule is classified in five categories of increasing gravity: Info, Minor, Major, Critical, Blocker. Rule violations are reported as Issues, and can be browsed using different criteria, or shown in source code.

SonarQube implements the *SQALE* [12] model for the estimation of Technical Debt. We experimented with the free plugin integrated in the platform[3]. There are three values computed on the analyzed project, i.e., Technical Debt (TD), Technical Debt Ratio (TDR) and SQALE Rating (SR).

[2]https://www.intooitus.com/, its evolution at http://www.aireviewer.com

[3]http://docs.sonarqube.org/display/SONARQUBE52/Technical+Debt

TD represents the time needed to fix all the Issues (i.e., rules violations) detected by SonarQube. A remediation cost (resolution time) is assigned to each issue, and TD is computed as the sum of all the single remediation costs. The TD value associated to single issues can be customized only by purchasing the commercial version of the SQALE plugin. As a result of an aggregation, TD can be associated to any project element (project, subproject, file) or any *quality characteristic* (e.g., maintainability, security). The TDR is a derived index, obtained as: $TDR = technical\_debt/estimated\_development\_cost$. $estimated\_development\_cost$ represents the estimated time needed to rewrite the project from scratch. In SonarQube, this is set to $LOC * 30minutes$. Please note that in the formula reported above, the two variables are time measures, and need to be in the same unity of measure. The TDR is produced to allow better comparability among different projects. Finally, the SQALE Rating is a letter from A (best) to E (worst), obtained by applying thresholds to the TDR value: A=0–0.1, B=0.11–0.2, C=0.21–0.5, D=0.51–1.

E. Structure101

Structure101[4] (S101) is a tool specialized in architectural evaluation and provides an automatically reconstructed view of the software architecture with different kinds of refactoring support. Structure101 shows a *Structural over-Complexity (SoC)* view to estimate the percentage of the system involved in architectural issues. The view displays two measures: %Tangle and %Fat, i.e., the percentage of metrics Fat and Tangle in the system. The Tangle metric is the count of tangles in packages or classes; specifically, a tangle is a dependencies subgraph containing one or more connected cycles. The Fat metric measures the complexity of the system. At the method level, it is computed using Cyclomatic Complexity (McCabe's metrics), while at any other level it corresponds to the number of inter-dependencies existing among items (e.g., classes, packages). The *Excessive Structural Complexity*, in Structure101, is called *XS* ("excess") and is computed on every item: methods, classes, packages, components. XS represents an estimation of the portion of the LOC of an item that are "affected" by Fat or Tangle.

III. Discussion on the main TDI features

In this section, we discuss the most relevant features and differences we found in the indexes provided by the tools, to summarize information useful to answer our questions.

In Tab. I (to answer Q1, Q2, Q3), we show the different categories of input information used by the tools to compute their indexes. Tools may also support the extraction of some information that is not used to compute a TDI: we consider only the information used in the index computation. For example, past versions of SQ were able to detect some architectural issues, but not used in the index computation, while it uses the wider range of code level information. The other tools exploit some architectural information in their TDIs.

[4]http://structure101.com/products/

Table I
Input information of Technical Debt Indexes used by tools

Information category	CAST	IF	SG	SQ	S101
Architectural Smells, e.g., [13], [9]	yes	yes	yes	no	yes
Code Smells [8], [11]	no	yes	no	yes	no
Architecture/Design Metrics, e.g., [15]	yes	no	no	no	yes
Code Metrics, e.g., [5], [11]	yes	no	no	yes	yes
Architectural Violations$^{\alpha}$	yes	no	yes	no	no
Coding Rule Violations$^{\beta}$	yes	no	no	yes	no

α: deviations from a reference architecture, i.e., unallowed dependencies
β: detected bad coding practices or excessive values of single metrics (some tools, e.g., SQ, internally refer to the latter as "smells")

Table II
Output of Technical Debt Indexes provided by tools

	CAST	inFusion	Sonargraph		SonarQube			Structure101
TDI name →	TDP	QDI	SDI	SDC	TD	TDR	SR	XS
Resolution cost	yes	no	no	yes	yes	yes	yes	no
Keeping cost	no	yes	yes	no	no	no	no	yes
Unity Measure	US$	-	-	US$	Time	-	Rank	LOC

In Tab. II (respect to Q1 and Q3), we characterize the information provided by the different TDIs, regarding both what the measures address and how they represent it (output information). From the table, we can see that indexes do not always provide estimation of both the costs of correcting the system (Resolution cost, i.e., the TD principal) and of keeping it unchanged (Keeping cost, i.e., the TD interest). These two aspects of TD are highly relevant during estimation, and this should be seen as an important improvement opportunity for tool vendors. A particular issue can be found in SonarQube, which implements the SQALE method for TD estimation, where the costs of keeping an issue are captured by the SQALE Business Impact Index (SBII). This index is not available in the free version of SonarQube, while it may exist in the commercial one. Anyway, the SonarQube classification of issues using the Info–Blocker range can be seen as a non-quantitative suggestion of the non-remediation costs. A similar choice has been done in CAST, by using a three-value severity (Low, Medium, High) to classify the detected problems. In both cases, the severity of each issue/problem has been defined by the authors of the tool, and can be customized by advanced users. In Sonargraph, the SDI is computed as a score, proportional to the set of issues considered in the index computation. This makes the SDI proportional to the estimated quality of the system, i.e., an estimation of Keeping costs. SDC is the money cost of fixing the system, i.e., a Resolution cost, but its value is computed linearly from the SDI, making the distinction between them not very clear from this point of view. Finally, different indexes provide different kinds of estimation. SonarQube's TD computes its value directly in terms of time needed to fix the reported issues. Then, it provides derived indexes with the aim of making comparisons among different projects easier, since the absolute TD value

will be higher in larger projects, but its "density" should be kept under control. All the other indexes start instead from computing a score based on the count or size/severity of the issues detected in the analyzed project. Then, in CAST and Sonargraph there is a mechanism that weights the effort associated to the computed score to express it in terms of time and costs. inFusion and Structure101 do not provide this mechanism. inFusion expresses its index using an abstract number. A peculiarity of Structure101 is to express its XS metric in terms of LOC affected by some detected issues. This choice allows relating the XS value to the size of the system, and to have an idea of how much the detected issues are widespread in the system.

All the Indexes share a common rationale, i.e., they start from the evaluation of specific quality indicators, e.g., architectural violations, code smells, and weight their relative severity, aggregating the outcome to provide an overall evaluation of the entire project or of its different parts. Anyway (always w.r.t Q3), the association of resolution times to issues is arbitrary. Tools allow customizing the effort associated to single issues (SQ) or to their scores (SG, CAST), but these values are arbitrary and there is no established guideline on how to set them. This is probably the reason why the other tools, other than SQ, did not associate effort measures to their indexes, and prefer to keep them as indexes to be used mainly when comparing the quality of a system in different revisions.

IV. Concluding Remarks

In this paper, we described the different TDIs provided by five tools, and outlined their differences, with a particular focus on architecture-related issues. In the following, we provide answers to the questions posed in Sec. I.

Q1 - In Sec. II, we described how the indexes are computed by the considered tools. They mainly take into account metrics, smells, coding rules violations and architecture violations, as shown in Tab. I. The estimations they provide are different in terms of unity of measure (e.g., time, cost, abstract numbers) and TD target (remediation costs and keeping costs), as shown in Tab. II.

Q2 - Given the discussion on the indexes provided in Sec. III, we can see that: *1)* Sonargraph, Structure101 and CAST use the largest share of architectural information in their indexes; *2)* SonarQube does not use architectural information when computing its index.

Q3 - First, we can observe that no tool uses all the information that can be exploited, at both code and architectural level, and no tool provides both Keeping and Resolution costs. Hence, tools could try to fill the gaps in their estimation models by re-using some knowledge exploited by other tools. Another observation is that tools are conservative in their estimation features, by relying only on static analysis and without exploiting historical information about the analyzed projects. In particular, most tools allow showing the history of their analyses on different revisions of the same project, but none of them uses the underlying changes in the software itself to spot issues relevant to Technical Debt estimation. For example, the detection of evolutionary coupling can be used to discover unseen dependencies.

In most cases, the available Indexes are not directly useful when evaluating a single project. The provided measures cannot be interpreted with the aim to understand the overall quality of the analyzed project on a global scale. As a consequence, we think that these Indexes are in particular useful on a relative scale, in the case a single team evaluates an entire portfolio of applications. In this case, the Index can be used, e.g., to rank new projects w.r.t. the old/existing ones.

In future work, we are interested in verifying how much architectural issues affect the overall quality, with the aim of giving different relevance to architecture and design issues w.r.t. coding ones in a TDI. We plan to investigate the role of code and architectural smells in TD, since they are associated to known solutions, that can speed up their resolution process. We would like also to work on the definition of a new TDI, with a focus on code and architectural debt, and experiment it on a large dataset of projects. In the TDI computation we would like to consider: 1) Code and Architectural smells detection; 2) Code and architecture/design metrics; 3) History of a system, including code changes and lifespan of smells; 4) Identification of problems more critical than others, to weight the collected analysis elements (e.g., metrics, smells, issues) according to their relevance in existing (past) projects.

References

[1] Arcelli Fontana, F., Maggioni, S.: Metrics and antipatterns for software quality evaluation. In: Proc. SEW 2011. IEEE, Ireland (Jun 2011)

[2] Arcelli Fontana, F., Roveda, R., Vittori, S., Metelli, A., Saldarini, S., Mazzei, F.: On evaluating the impact of the refactoring of architectural problems on software quality. In: RefTest 2016 — XP Conf. Workshop. ACM, Scotland (May 2016)

[3] Arcelli Fontana, F., Roveda, R., Zanoni, M.: Tool support for evaluating architectural debt of an existing system: An experience report. In: Proc. 31st Symp. Applied Computing (SAC 2016). ACM, Italy (Apr 2016)

[4] Arcelli Fontana, F., Roveda, R., Zanoni, M., Raibulet, C., Capilla, R.: An experience report on detecting and repairing software architecture erosion. In: Proc. 13th WICSA 2016 Conference. IEEE, Italy (Apr 2016)

[5] Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Software Eng. 20(6), 476–493 (1994)

[6] Coad, P., Yourdon, E.: Object-oriented Design. Yourdon Press (1991)

[7] Dietrich, J., McCartin, C., Tempero, E., Shah, S.M.A.: On the existence of high-impact refactoring opportunities in programs. In: Proc. 35th Australasian Comp. Sci. Conf. (ACSC'12). ACS, Australia (Feb 2012)

[8] Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA (1999)

[9] Garcia, J., Popescu, D., Edwards, G., Medvidovic, N.: Identifying architectural bad smells. In: Proc. CSMR 2009. IEEE, Germany (2009)

[10] hello2morrow GmbH: Metrics and Queries Doc. v.7.2 (May 2011)

[11] Lanza, M., Marinescu, R.: Object-Oriented Metrics in Practice. Springer (2006)

[12] Letouzey, J.L.: The SQALE method for evaluating technical debt. In: Proc. 3rd Int. Work. on Managing Technical Debt (June 2012)

[13] Lippert, M., Roock, S.: Refactoring in Large Software Projects: Performing Complex Restructurings Successfully. Wiley (Apr 2006)

[14] Marinescu, R.: Assessing technical debt by identifying design flaws in software systems. IBM Jour. of Research and Development 56(5) (2012)

[15] Martin, R.C.: Object oriented design quality metrics: An analysis of dependencies. ROAD 2(3) (Sept–Oct 1995)

[16] Shah, S.M.A., Dietrich, J., McCartin, C.: Making smart moves to untangle programs. In: Proc. CSMR 2012. IEEE, Hungary (Mar 2012)