Conciliating Model-Driven Engineering with Technical Debt Using a Quality Framework

Fáber D. Giraldo[1,2](✉), Sergio España[2], Manuel A. Pineda[1], William J. Giraldo[1], and Oscar Pastor[2]

[1] System and Computer Engineering, University of Quindío, Quindio, Colombia
{fdgiraldo,mapineda,wjgiraldo}@uniquindio.edu.co

[2] PROS Research Centre, Universitat Politècnica de València, Valencia, Spain
{fdgiraldo,sergio.espana,opastor}@pros.upv.es

Abstract. The main goal of this work is to evaluate the feasibility to calculate the technical debt (a traditional software quality approach) in a model-driven context through the same tools used by software developers at work. The *SonarQube* tool was used, so that the quality check was performed directly on projects created with Eclipse Modeling Framework (EMF) instead of traditionals source code projects. In this work, XML was used as the model specification language to verify in SonarQube due to the creation of EMF metamodels in XMI (XML Metadata Interchange) and that SonarQube offers a plugin to assess the XML language. After this, our work focused on the definition of model rules as an XSD schema (XML Schema Definition) and the integration between EMF-SonarQube in order that these metrics were directly validated by SonarQube; and subsequently, this tool determined the technical debt that the analyzed EMF models could contain.

Keywords: Model-driven engineering · Technical debt · EMF · SonarQube

1 Introduction

Two representative trends for the software development industry that appeared in the nineties were the *model-driven* initiative and the *technical debt* metaphor. Both trends promote software quality each in its own way: high abstract levels (models) and software process management (technical debt). However, despite the wide exposition of these trends in the literature, there are not more indications about the combination of them into software development scenarios; each initiative is implemented in a separated way.

More than 20 years ago, the *technical debt* term was introduced as a way to describe the long-term costs associated with problems of software design and implementation. Some typical examples of technical debt exposed in [17] include: *glue* code, code done and fixing it after release, hundreds of customer-specific branches on same code base, *friendly* additions to interfaces, multiple codes for

© Springer International Publishing Switzerland 2015
S. Nurcan and E. Pimenidis (Eds.): CAiSE Forum 2014, LNBIP 204, pp. 199–214, 2015.
DOI: 10.1007/978-3-319-19270-3_13

the same problem, and so on. The technical debt approach has been used as a control mechanism for projects to lower maintenance costs and reduce defects.

In traditional software development projects (those involving manual programming), technical debt is mainly focused in quality assurance processes over source code and related services (e.g., common quality metrics are defined over source code). However, model-driven engineering (MDE) promotes for modelling instead of programming [3]. A review of the literature reveals that there is currently no application of the technical debt concept to environments outside the traditional software development. There exist approaches to the measurement of model quality [11,13,15,18], but these do not include technical debt calculus. Therefore, we claim that dealing with technical debt in MDE projects is an open problem.

Two issues pose challenges to the inclusion of technical debt into MDE. *(i)* Different authors provide conflicting conceptions of quality in model management within MDE environments [7]. *(ii)* The MDE literature often neglects techniques for source code analysis and quality control[1]. Therefore, in model-driven developments it is difficult to perform an analysis of the state of the project that is important for technical debt management: establishing what has been done, what remains to be done, how much work has been left undone. Also, other specific issues that belong to model theory such as: number of elements in the metamodel, coverage for the views, complexity of the models, the relationship between the abstract syntax and the concrete syntax of a language, quantity of OCL verification code, among others, contribute to increase the technical debt in model-driven projects.

Similar to software projects, model-driven projects could be affected by events that impact the quality of the conceptual models and its derived artifacts. The technical debt incidents for model-driven contexts come mainly from the software development inherent practices and model specific issues. Also, the lack of a standardized definition about quality in models increase the complexity of modelling tasks, so that, the bad modelling practices become specific according to the kind of modelling project that is performed.

The main contributions of this paper are the following:

1. A discussion about the importance of considering the technical debt calculus in model-driven projects, as part of a model quality initiative.
2. A demonstration of a integration between model-driven and technical debt tools for supporting a technical debt calculus process performed over conceptual models.
3. The operationalization of a recognized framework for evaluating models.

In this work, we used the principles of research in quality over models to generate quality metrics that can be useful to validate these models with a technical debt focus. This work is organized as follows: Sect. 2 introduces the motivation of

[1] Neglecting the code would seem sensible, since MDE advocates that the model_is_ the code [4]. However, few MDE tools provide full code generation and manual additions of code and tweakings are often necessary.

our idea, Sect. 3 presents the technical solution implemented, Sect. 4 exposes a preliminary validation process around of our proposal, Sect. 5 presents the state of the art; and finally, the conclusions and further works derived of our proposal are presented.

2 Motivations

The technical debt definition was originally focused on source code; but as shown in [12], this concept could be extended to other activities and artifacts belonging to the software construction process. Technical debt focuses on the management of the consequences of anything that was not done intentionally or unintentionally, and subsequently, it is materialized as bugs or anormal situations that affect a software project or product. Currently, it is possible to evidence how software companies have assimilated the importance of technical debt control in its software projects, highlighting the use of tools like SonarQube[2], responsible for assessing the presence of technical debt through evidence of malpractices embodied on software artifacts like source code. Also from a technical viewpoint, these kind of tools support project management very close to code and low-level artifacts.

The technical debt practice has become an important strategy in current quality assurance software processes. Its application can help to identify problems over the artifacts quantifying the consequences of all the work that was not done in order to contrast it regarding the budgetary constraints of the project.

Despite the several particular approaches involved in software quality assurance, it has certain maturity levels due to the effort of software quality practitioners for encompassing these approaches around the fulfillment of expectations, requirements, customer needs, and value provisions [9]. It is supported by descriptive models and standards that define the main issues of software quality. In this way, activities such as defect detection and correction, metric definition and application, artifact evolution management, audits, testing, and others, are framed into these software quality definitions. Software quality involves a strategy towards the production of software that ensures user satisfaction, absence of defects, compliance with budget and time constraints, and the application of standards and best practices for the software development.

Instead, it is possible to identify a proliferation of model quality definitions in the model-driven context with multiple divergences, different motivations and additional considerations due to the nature of the model artifacts. Quality in the MDE context is particularly defined according to the specific proposals or research areas developed by the MDE practitioners. In [7], authors note that the *quality in models* term does not have a consistent definition and it is defined, conceptualized, and operationalized in different ways depending on the discourse of the previous research proposals. The lack of consensus for the model quality definition produce empirical efforts for verifying quality over specific features of models.

[2] http://www.sonarqube.org/.

Within the MDE literature is possible to find proposals which extrapolate particular approaches for evaluating software quality at model levels (supported by the fact that the MDE is a particular focus to the software engineering), such as the use of metrics, defect detection over models, application of software quality taxonomies (in terms of characteristic, sub-characteristic and quality attributes), best practices for implementing high quality models and model transformations; and even, it is possible to see a research area oriented to the usability evaluation of modelling languages [23], where the usability is prioritized as the main quality attribute.

Most of the model quality frameworks proposed act over specific model artifacts, generally evaluation of notations or diagrams. These frameworks do not consider the implications around the performed activities over models in terms of the consequences of the good practices that were not made. This is a critical issue because the model-driven projects have the same project constraints with respect to software projects. The only difference is the high abstract level of the project artifacts and the new roles with respect to domain experts and languages users.

Notations and diagrams are the main way of interaction for the final users of a language, and in this sense, most of the model quality proposals are around specific attributes of interaction, cognition, readability, usage and comprehensibility. The evaluation of the global quality of a conceptual model is a very complex task. A first important attempt is the quality evaluation based on notations used by the model, avoiding the incorrect combinations of conceptual constructs and ambiguous situations that could violate the principles and rules of the language and its associated constructs. Notations in a key aspect for the evaluation of model quality. However, we claim that model-driven activities can contribute to establishing a technical debt for modelling projects beyond a notation perspective, because it considers both modelling issues and software practices involved. The technical debt for model-driven projects could be more complex than software technical debt. Also, the use of technical debt at the model-driven context could help to manage and evaluate the employed process over a model-driven specific context.

The main concern of the technical debt is the consequence of poor software development [27]. This is a critical issue not covered in model-driven processes whose focus is specific operation over models such as model management or model transformations. A landscape for technical debt in software is proposed in [12] in terms of evolvability challenges and external/internal quality issues; we think that model-driven iniatives cover all the elements of these landscapes taking into account that authors like [20] suggest models as elements of internal quality software due to its intermediate nature in a software development process. Integration between model-driven and technical debt have not been considered by practicioners of each area despite the enormous potential and benefits for software development processes.

3 Our Proposal

3.1 Proposal in a Nutshell

In order to demonstrate the feasibility to calculate technical debt over models in a model-driven working environment, we performed the following steps:

1. We operationalized a quality framework for models to derive technical debt evidences w.r.t. a previous quality reference (Sect. 3.2).
2. An integration of a MDE working environment with an instance of a SonarQube server (the selected technical debt tool) was implemented. This was made through a plugin that automatically invokes the SonarQube tool (Sect. 3.3).
3. A technical debt verification process is performed over a model sample. Since the models workspace the Sonarqube instance is invoked. This instance uses the operationalization of the quality framework to find technical debt over the model sample under evaluation (Sect. 3.4).

3.2 Definition of an XSD for SonarQube

One of the most critical issues in a technical debt program is the definition of metrics or procedures for deducting technical debt calculations; in works like [6,10] it is highlighted the absence of technical debt values (established and accepted), and features such as the kinds of technical debt. Most of the technical debt calculation works are focused on software projects without an applied model-driven approach; some similar works report the use of high level artifacts as software architectures [22], but they are not model-driven oriented. Emerging frameworks for defining and managing technical debt [24] are appearing, but they focus on specific tasks of the software development (not all the process itself).

From one technical perspective, the SonarQube tool demands an XSD (XML Scheme Document) configuration file that contains the specific rules for validating the code; or in this case, a model. Without this file, the model could be evaluated like a source code by default. In order to define these rules, we chose one of the most popular proposals for validating models (*Physics of notations* - *PoN* - of Moody [21]) due to its relative easiness to implement some of its postulates in terms of XSD sentences.

In the case of this work, visual notation was taken as the textual information managed by XMI entities from EMF models (text are perceptual elements too), focusing that each item meets syntactic rules to display each information field regardless about what is recorded as a result of the EMF model validation. The analysis does not consider the semantic meaning of the model elements to be analyzed.

The operationalization of Moody principles over the XSD file posteriorly loaded in SonarQube was defined as follows:

- *Visual syntax - perceptual configuration*: in the XSD file, it is ensured that all elements and/or attributes of the modelled elements are defined according to the appropriate type (the consistence between the values of attributes and its associated type is validated).
- *Visual syntax - attention management*: a validation order of the elements is specified by the usage of order indicadors belonging to XML schemes.
- *Semiotic clarity - redundant symbology*: a node in the model can only be checked by an XSD element.
- *Semiotic clarity - overload symbology*: an XSD element type only validates a single model node type.
- *Semiotic clarity - excess symbolism*: a metric to validate that there are no blank items was implemented (for example, we could create several elements of *Seller* type, but its data does not appear).
- *Semiotic clarity - symbology deficit*: a validation that indicates the presence of incomplete information was made (e.g., we could have the data of a *Customer* but we don't have his/her name or identification number). For this rule, we made constraints with occurrence indicators to each attribute.
- *Perceptual discriminability*: in the XML model, nodes must be organized in a way that they can be differentiated, e.g., one *Seller* element does not appear like a *Location* element. This is ensured by reviewing in the XSD that it does not contain elements exactly alike, and in the same order.
- *Semantic transparency*: this was done by putting restrictions on the names of the tags, so that the tags correspond to what they must have, e.g., a *data* label must be of *data* type.
- *Complexity management*: this was done by the *minOccurs* and *maxOccurs* occurrence indicators. With these indicators it is possible to define how many children one node can have.
- *Cognitive integration*: this was done using namespaces in the XSD file, so that it is possible to ensure the structure for the nodes independent from changes in the model design.
- *Dual codification*: this was done by measuring the quantity of commented code lines with respect to the XML lines that define the elements of the model.
- *Graphic economy*: we established a limit for different items that can be handled in the XSD, and reporting when different elements are found marking the mistake when these data types are not found in the schema.
- *Cognitive fit*: this was done by creating several XSD files where each one is responsible for reviewing a specific view model.

Figure 1 exposes a portion of the XSD code implemented for some Moody principles.

3.3 Implementation of a Technical Debt Plugin for EMF

We implemented an Eclipse plugin for integrating the EMF environment with SonarQube; so that, results of the technical debt can be shown directly on the Eclipse work area instead of changing the context and opening a browser with

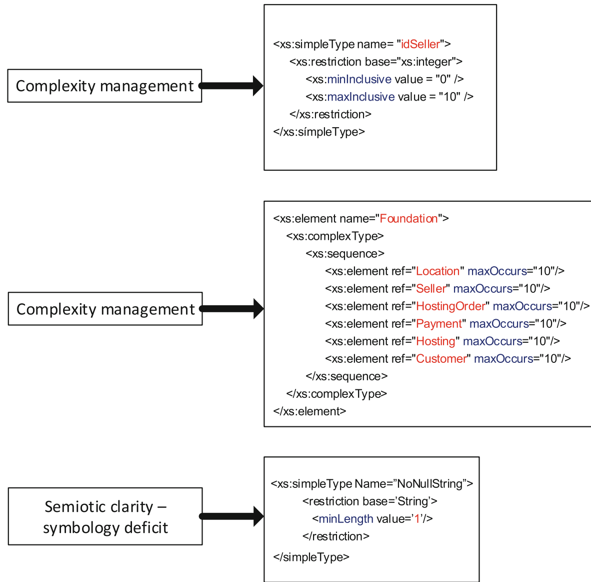

Fig. 1. Mapping between some Moody principles to XSD code.

the SonarQube report. Figure 2 exposes main issues of the developed plugin. We used configuration options belonging to EMF *XMIResource* objects to export the XMI file as an XML without the specific XMI information tags (Fig. 2, part *C*). Also, the integration with the Eclipse IDE was done by a button and a menu as it can be seen in part *A* and *B* of the same figure.

3.4 Verification of Technical Debt from EMF Models

In order to demonstrate the integration of both tools (EMF-SonarQube), a sample metamodel (Fig. 3) was made in EMF. This model is extracted from the case study formulated in [8], and it is complemented with data patterns exposed in [2] such as *Location*, *Client*, *Payment* and *Master/Detaill*. Regarding to the rules specified in the Sect. 3.2 we introduce some errors like *no valid options*, *date format* and *specific quantity of elements*, to evidence abnormalities not covered with model conceptual validation approaches like OCL.

Once the validation option had been chosen (by the SonarQube button or menu), we obtain a report similar to Fig. 4. Part *A* indicates the number of lines of code that have been tested, comment lines, and duplicate lines, blocks or files. Also, part *B* of this figure reports the total of errors that contain the project (in this case the EMF model), as well as the technical debt graph (part *C*), which shows the percentage of technical debt, the cost of repair, and the number of men needed to fix errors per day (this information was not configured for this case).

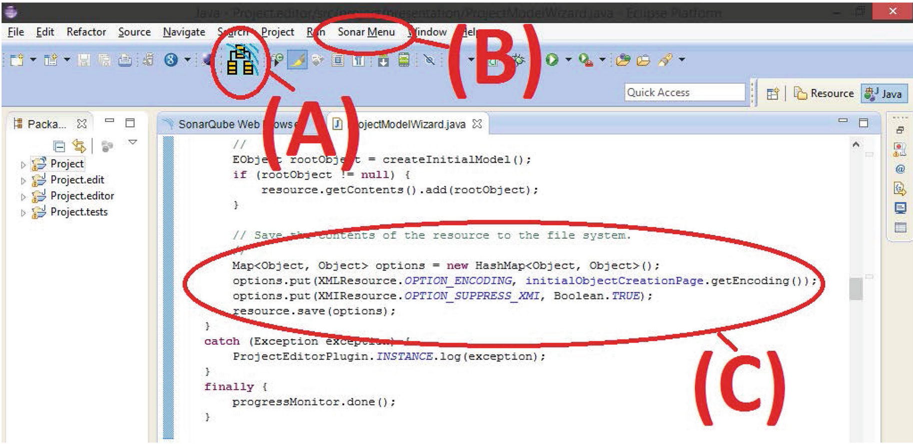

Fig. 2. Supressing XMI tags to analyze the EMF model as a XML document.

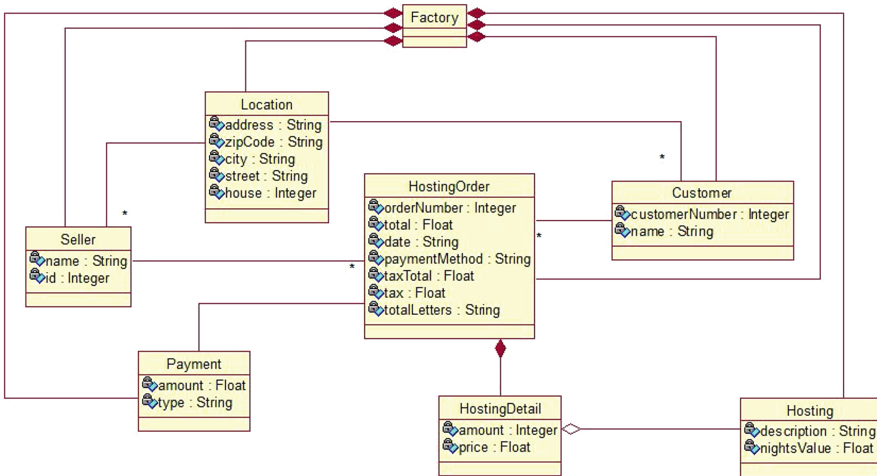

Fig. 3. Sample metamodel implemented over EMF.

SonarQube offers an *issues* report where it indicates the number of errors found; and consequently, the error list distributed in order of importance from highest to lowest:

– *Blocker*: they are the most serious errors; they should have the highest priority to review.
– *Critical*: they are design errors which affect quality or performance of the project (model errors can be classified in this category).
– *Major*: although these errors do not affect performance, they require to be fixed for quality concerns.

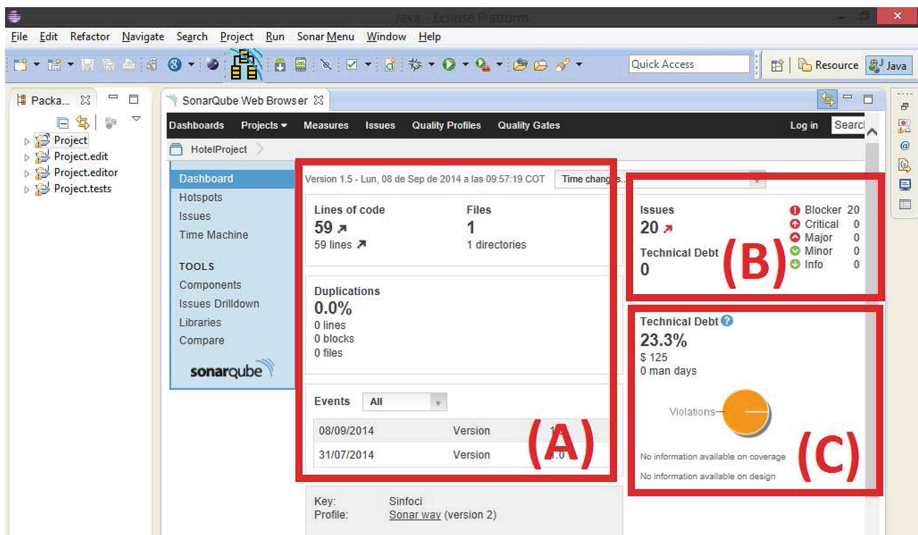

Fig. 4. SonarQube screen report loaded into EMF work area

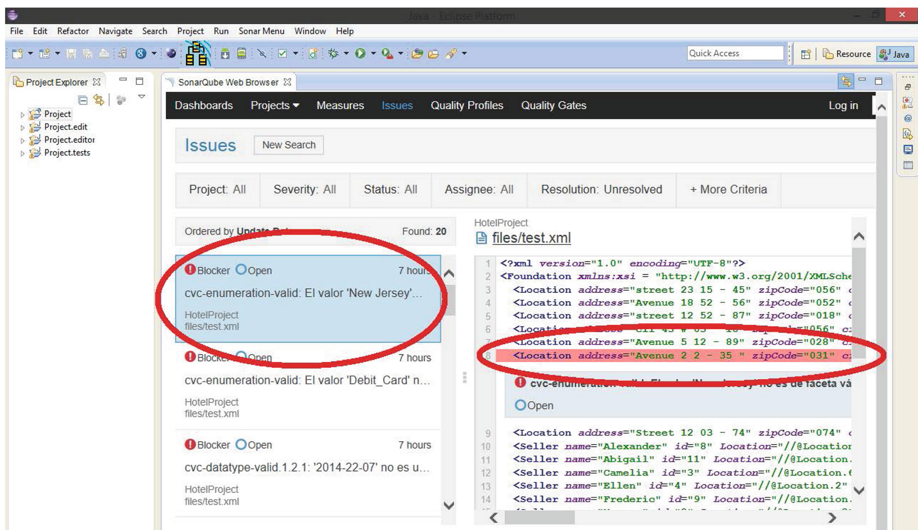

Fig. 5. Example of error (issue) detected by SonarQube over the EMF model.

- *Minor*: they are minor errors that do not affect the operation of the project.
- *Info:* they are reporting errors, not dangerous.

Figure 5 present the reports about technical debt errors detected over the sample model. In the first place, an error category was chosen. For the respective category,

the error list associated is show in detail posteriorly. Intentionally, we introduced errors over the XML information of the model to test the respective detection by SonarQube according with the rules defined in the XSD file from the Moody proposal.

4 Validation

A first validation of our proposal was performed using some basic usability testing procedures. The main goal of these validation was to identify interaction issues associated to the Eclipse EMF-SonarQube integration. Also, the utility of this integration (from a model-driven practioners perspective) was checked. The participant population were students from software engineering courses that work with conceptual models and structural data models in Eclipse EMF, and software engineering researchers (experts) who work with EMF in real software projects using the model-driven paradigm. The chosen public was invited by their previous knowledges in EMF and data productions with EMF. The total population was 17 participants.

4.1 Test Design and Procedure

In first instance we defined a data production for the model exposed in the Fig. 3. Intentionally we have introduced ten defects over the data of the generated EMF model production. These defects are related to rules configured in a XML schema previously defined and associated to SonarQube instance server. These rules are derived from some Moody PoN principles (Sect. 3.2). The employed rules over the model were the following:

- Accepted payment types are *Cash*, *Credit_Card* and *Bank_Check*.
- Cities that are permitted for the data model are *London, Medellin, SaoPaulo, NewYork, Washington, Valencia, Madrid, Paris, Bogota*.
- Seller ID must be a number between 0 to 10.
- Valid date format is YYYY-MM-DD.
- In the production, the maximum number of elements type *Location, Seller, HostingOrder, Payment, Hosting* and *Customer* are ten elements for each one.

These rules were accesible for all participants during the test execution.

For the usability test a Tobii X2-30 Eye Tracker[3] device was employed in order to precisely capture and determine where the participants are looking when we shown them the Eclipse enviroment with the EMF-SonarQube integration mechanisms. Figure 6 presents the test scenario with the hardware and software used with each participant. Figure 7 presents the software testing scenario (supported by the Tobii Studio Eye Tracking Software).

This test was split into three parts or *momentums* as follows:

[3] http://www.tobii.com/en/eye-tracking-research/global/products/hardware/tobii-x2-30-eye-tracker/.

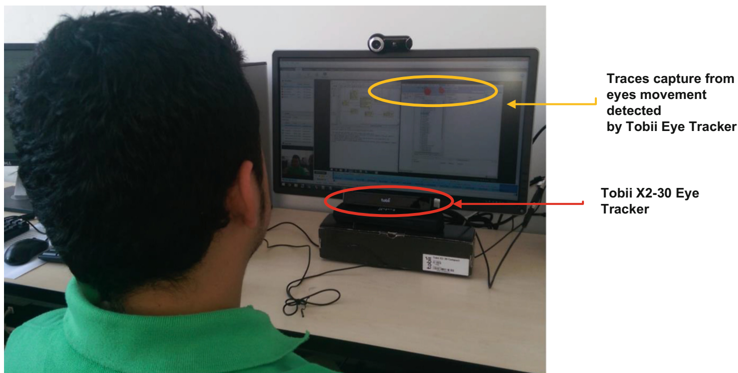

Fig. 6. Test enviroment used for the validation.

– *Momentum 01:* for each participant he/she was asked about evidenciable defects over the EMF production directly. The main goal of this part is *to check if the participant can detect the defects intentionally introduced in the production using the information given by EMF.*
– *Momentum 02:* in this section of the test researchers request to each participant to validate the model in SonarQube using the provided mechanism (button or menu). The main goal of this was detect if *the participants could recognize the graphical elements (plugin) that link the EMF with SonarQube.*
– *Momentum 03:* each participant was asked about how *to access to the defects reported by SonarQube using the user interface provided for this tool and loaded into Eclipse.*

Finally, a *Retrospective Think Aloud* (RTA) procedure was performed with each participant. Using the recordings of the previous usability test researchers asked to the participants about their actions during the usability test. The RTA activity considered these issues:

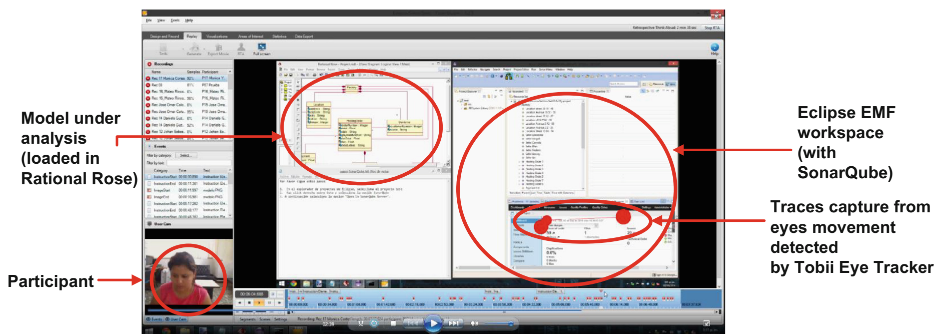

Fig. 7. Test software environment supported by the usability testing tool.

- The identification of defects over the EMF production directly.
- The easiness for identifying defects directly from the EMF production.
- The easiness to invoce the SonarQube validation over the EMF model.
- The sufficiency of the information provided by SonarQube in order to find defects in the EMF production.
- The usefulness of the EMF-SonarQube integration.

4.2 Results

Results of the usability test are exposed in Table 1. It is splited in the three *momentums* described above. For the *momentum 01 - the participant detected defects over the production* - the percentage of participants that report defects directly over the EMF production is high but the defects reported (in average) is too low with respect to the total of defects intentionally introduced (15 % in average). This reflects that finding defects directly over the EMF production is a hard task, and the probability of accidently discard defects are representative.

Table 1. Results of the usability test over the EMF-SonarQube integration.

	Momentum 01			Momentum 02		Momentum 03		
	No	Yes	Def. Av.	Button	Menu	No	Yes	Time Av.
Experts	16,67 %	83,33 %	1,80	66,67 %	33,33 %	16,67 %	83,33 %	83,00
Students	18,18 %	81,82 %	1,33	90,91 %	9,09 %	45,45 %	54,55 %	89,83
Participants	**17,65 %**	**82,35 %**	**1,50**	**82,35 %**	**17,65 %**	**35,29 %**	**64,71 %**	**86,73**

For the *momentum 02 - Can the user validate the model in SonarQube from Eclipse EMF?*- all the participant reports that they did this, mainly through the button exposed in the Fig. 2-A. EMF experts users have found the new graphical elements associated with SonarQube so that they access directly to these elements to make the validation. However, in the case of the students, 45,45 % of them request an additional explanation to researchers in order to identify the elements and make the validation. It's due to the low contact of the students with quality platforms in their software engineering courses.

Finally, in the *momentum 03 - Do the participant access to the reported defects in SonarQube?* - we found a representative percentage of participants who reported no access to the defects of the model reported by SonarQube. This is consequence of the native navigation model of SonarQube (no considered in the scope of our validation process as such).

Main findings from the RTA procedure were: *(i)* The relative big size of the proposed button proportional to the Eclipse tools area. This in particularly important due that this feature let to users (mainly experts) to identify the new proposed tool. *(ii)* The image icon used in the SonarQube button does not associate it to the model validation process itself. Most of the users request a new

icon that express the model validation more natively from Eclipse EMF. *(iii)* A new requirement from the participants that exist a doble via navigation between the defects of the model reported in SonarQube and the EMF model/production in order that the context of the validation does not disappear when the SonarQube browser is invoked. All these findings promote a second version of our proposal.

5 State of the Art

There are not major reports about the integration of technical debt with model-driven works; it is evident the works where technical debt is applied jointly with specific methods of software quality [14]. A closer work is reported in [16] where a technical debt evaluation framework was proposed, and it was applied over the EMF project for determining the technical debt of this Eclipse project based on all the versions of it. EMF was chosen because it contains some features expected by the framework (popularity, maturity, proficiency and open source), but the quality assessment was made with a tool different to SonarQube.

The main challenge of this kind of work is the derivation of quality metrics or rules from model quality frameworks. High abstraction and specific model issues influence the operationalization of model quality frameworks, so that quality rules or procedures could no be full implemented by operational mechanisms such as XSD schemas. Authors in [26] expose an attempt to make operational the *Physics of notations* evaluation framework, but this operationalization (and any similar proposal) could be ambiguous as consequence of the lack of precision and detail of the framework itself. Also, they suggest the need of a guideline for the evaluation framework prior to the production of its associated metrics.

Regarding the usage of the SonarQube platform to evaluate models a similar work is exposed in [25] where a SonarQube plugin was implement in order to support the evaluation of business process models described in the event-driven process chains language. This plugin uses the software quality model ISO 9126 (in terms of characteristics and subcharacteristics) and other measures previously formulated.

An example of model quality assurance tools as reported in [1] where it is presented an operational process for assessing the quality through static model analysis to check model features like consistency (with respect of language syntax), conceptual integrity, and the conformity with modelling conventions. Instead of having an operational model quality framework, we can see how a quality framework like *6C* [19] has been used as a conceptual basis for derivating a quality assurance tool.

6 Conclusions

In this work we show the technical feasibility to integrate a technical debt tool like SonarQube with a model-driven development enviroment such as the Eclipse modelling framework. We present an example of technical debt validation applied

over a sample metamodel implemented for testing purposes. Thereby, we demonstrate the technical feasibility for measuring any artefact used in an model-driven engineering process [5]. However, the main challenge is the definition of the model quality metrics and the operationalization of the model quality frameworks reported in terms of expressions that can generate metrics, and its association with a model-driven development process.

A plethora of model quality frameworks are proposed, but their operationalization is very incipient and these are used as reference frameworks. A metric/rules derivation process from quality frameworks is needed taking into account its operationalization in order to support a model quality assurance process by tools. An important further work is the applicability of technical debt to the visual quality of diagrams because these are the most representative quality proposals for models; it means, evaluting the quality of diagrams in a similar way as SonarQube evaluates quality at the source code. Also, the implementation of automatic checks over the OCL code could be an important strategy to verify quality issues over models.

From a technical perspective, as another further work, we propose to use SonarQube plugins that offer technical debt evaluation through specific approaches like the SQALE Methodology (software quality assessment based on lifecycle expectations)[4] [14]. The main challenge of this proposed work is the extrapolation of the particular technical debt method to the model-driven context; this could be supported by the quality taxonomy of characteristics/sub-characteristics/metrics or quality attributes common employed in model quality proposals.

Acknowledgments. F.G, thanks to Colciencias (Colombia) for funding this work through the Colciencias Grant call 512-2010. F.G. and M.P. thanks to David Racodon (david.racodon@sonarsource.com) and Nicla Donno (nicla.donno@sonarsource.com) for their suppport with the SQALE plugin for SonarQube. This work has been supported by the Spanish MICINN PROS-Req (TIN2010-19130-C02-02), the Generalitat Valenciana Project ORCA (PROMETEO/2009/015), the European Commission FP7 Project CaaS (611351), and ERDF structural funds.

References

1. Arendt, T., Taentzer, G.: A tool environment for quality assurance based on the eclipse modeling framework. Autom. Softw. Engg. **20**(2), 141–184 (2013)
2. Blaha, M.: Patterns of Data Modeling. CRC Press, Boca Raton (2010). ISBN 1439819890
3. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, San Rafael (2012)

[4] http://www.sonarsource.com/products/plugins/governance/sqale/installation-and-usage/.

4. Embley, D.W., Liddle, S.W., Pastor, O.: Conceptual-model programming a manifesto. In: Embley, D.W., Thalheim, B. (eds.) Handbook of Conceptual Modeling, pp. 3–16. Springer, Heidelberg (2011). ISBN 978-3-642-15864-3
5. Bertoa, M.F., Antonio, V.: Quality attributes for software metamodels. In: Proceedings of 13th TOOLS Workshop on Quantitatives Approaches in Object-oriented Software Engineering, QAAOSE 2010, 2 July, Málaga, Spain, February 2010
6. Falessi, D., Shaw, M.A., Shull, F., Mullen, K., Keymind, M.S.: Practical considerations, challenges, and requirements of tool-support for managing technical debt. In: 2013 4th International Workshop on Managing Technical Debt (MTD), pp. 16–19 (2013)
7. Fettke, P., Houy, C., Vella, A.-L., Loos, P.: Towards the reconstruction and evaluation of conceptual model quality discourses – methodical framework and application in the context of model understandability. In: Bider, I., Halpin, T., Krogstie, J., Nurcan, S., Proper, E., Schmidt, R., Soffer, P., Wrycza, S. (eds.) EMMSAD 2012 and BPMDS 2012. LNBIP, vol. 113, pp. 406–421. Springer, Heidelberg (2012)
8. Giraldo, W.J.: Framework for the development of interactive groupware systems based on the integration of process and notations. Ph.D. thesis (2010)
9. ISO/IEC. ISO/IEC 9126. Software engineering - Product quality. ISO/IEC (2001)
10. Izurieta, C., Griffith, I., Reimanis, D., Luhr, R.: On the uncertainty of technical debt measurements. In: 2013 International Conference on Information Science and Applications (ICISA), pp. 1–4 (2013)
11. Krogstie, J.: Quality of models. In: Krogstie, J. (ed.) Model-Based Development and Evolution of Information Systems, pp. 205–247. Springer, London (2012). ISBN 978-1-4471-2935-6
12. Kruchten, P., Nord, R.L., Ozkaya, I.: Technical debt: from metaphor to theory and practice. IEEE Softw. **290**(6), 18–21 (2012)
13. Lange, C.F.J., Chaudron, M.R.V.: Managing model quality in UML-based software development. In: 2005 13th IEEE International Workshop on Software Technology and Engineering Practice, pp. 7–16 (2005). LCCN 0029
14. Letouzey, J., Ilkiewicz, M.: Managing technical debt with the sqale method. IEEE Softw. **29**(6), 44–51 (2012)
15. Marín, B., Giachetti, G., Pastor, O., Abran, A.: A quality model for conceptual models of mdd environments. Adv. Soft. Eng. **2010**, 1:1–1:17 (2010)
16. Marinescu, R.: Assessing technical debt by identifying design flaws in software systems. IBM J. Res. Dev. **56**(5), 9:1–9:13 (2012)
17. McConnell, S.: Managing technical debt. In: Fourth International Workshop on Managing Technical Debt in conjunction with ICSE 2013 (2013)
18. Mohagheghi, P., Dehlen, V.: Developing a quality framework for model-driven engineering. In: Giese, H. (ed.) MODELS 2008. LNCS, vol. 5002, pp. 275–286. Springer, Heidelberg (2008)
19. Mohagheghi, P., Dehlen, V., Neple, T.: Definitions and approaches to model quality in model-based software development - a review of literature. Inf. Softw. Technol. **51**(12), 1646–1669 (2009)
20. Moody, D.L.: Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions. Data Knowl. Eng. **55**(3), 243–276 (2005)
21. Moody, D.L.: The 'physics' of notations: Toward a scientific basis for constructing visual notations in software engineering. IEEE Trans. Softw. Eng. **35**(6), 756–779 (2009)

22. Nord, R.L., Ozkaya, I., Kruchten, P., Gonzalez-Rojas, M.: In search of a metric for managing architectural technical debt. In: 2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), pp. 91–100 (2012)
23. Schalles, C.: Usability Evaluation of Modeling Languages: An Empirical Research Study, vol. 1, p. 197. Springer Gabler, Heidelberg (2013). ISBN 978-3-658-00051-6
24. Seaman, C., Guo, Y.: Chapter 2 - Measuring and Monitoring Technical Debt. Advances in Computers, vol. 82. Elsevier, London (2011)
25. Storch, A., Laue, R., Gruhn, V.: Measuring and visualising the quality of models. In: 2013 IEEE 1st International Workshop on Communicating Business Process and Software Models Quality, Understandability, and Maintainability (CPSM), pp. 1–8, September 2013
26. Störrle, H., Fish, A.: Towards an operationalization of the "physics of notations" for the analysis of visual languages. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) MODELS 2013. LNCS, vol. 8107, pp. 104–120. Springer, Heidelberg (2013)
27. Tom, E., Aurum, A., Vidgen, R.: An exploration of technical debt. J. Syst. Softw. **86**(6), 1498–1516 (2013)