Preemptive Management of Model Driven Technical Debt for Improving Software Quality

Clemente Izurieta
Department of Computer Science
Software Engineering Laboratory
Montana State University, USA
1-406-994-3720
clemente.izurieta@cs.montana.edu

Gonzalo Rojas
Department of Computer Science
Faculty of Engineering
University of Concepción, Chile
56-41-220-4305
gonzalorojas@inf.udec.cl

Isaac Griffith
Department of Computer Science
Software Engineering Laboratory
Montana State University, USA
1-406-994-4780
isaac.griffith@msu.montana.edu

ABSTRACT
Technical debt has been the subject of numerous studies over the last few years. To date, most of the research has concentrated on management (detection, quantification, and decision making) approaches –most performed at code and implementation levels through various static analysis tools. However, if practitioners are to adopt model driven techniques, then the management of technical debt also requires that we address this problem during the specification and architectural phases. This position paper discusses several questions that need to be addressed in order to improve the quality of software architecture by exploring the management of technical debt during modeling, and suggests various lines of research that are worthwhile subjects for further investigation.

Categories and Subject Descriptors
D.2.10 [Software Engineering]: Design – Design Concepts, Object-oriented design methods; D.2.11 [Software Engineering]: Software Architectures – patterns.

General Terms
Measurement; Design; Experimentation.

Keywords
Technical debt; model driven development; software quality; software maintenance; model and architectural quality.

1. INTRODUCTION
Traditional engineering has always used models to capture simplified abstractions of behavior and structure in the setting of their domains before building their products. Models are abstract, yet they can be precise, predictive, comprehensive and cheap [28]. Software development projects are different. Attempts by the software modeling community have fallen short of their goals and expectations to gain meaningful adoption by practicing engineers.

QoSA'15, May 04–08, 2015, Montreal, QC, Canada.

http://dx.doi.org/10.1145/2737182.2737193

Practitioners in industry have not embraced Model Driven Development (MDD) [6], and when adopted, it is only as a distinct activity during design, and certainly not treated as a first class citizen during the Model Driven Engineering (MDE) process [8]. Software models and their respective implementations have exhibited large semantic gaps, and although MDD tools are increasingly improving their usability and features, most of them lack validation, are cumbersome to use, and changes that occur in either the model or the code are not reflected in their counterparts –*traceability* is too expensive for practitioners. These, among other problems, have contributed to relegating modeling as a documentation exercise at best.

Our **position** is that we must enhance the value added by MDE. One method to accomplish this goal is by recognizing disharmonies early in the architecture of a design and generating clean implementations early. By characterizing and removing deficiencies before executable components are generated we add value. By using Technical Debt (TD) [4] as an overarching metaphor that captures quality deficiency concepts we can abstract debts to models. To date however, refactoring efforts defined by Fowler [7], and TD studies have focused on detection, measurement, and analysis techniques on source code. Preliminary definitions and taxonomies of TD are based on decisions made at implementation phases. However, an important part of the implemented code is obtained from implementation-independent models through a mapping process, and to our knowledge there is no body of work that addresses the detection, quantification, and decision making of TD during the modeling and mapping activities of the specification phase. By proactively reducing TD at the architectural modeling level through improved MDE tools, we can significantly enhance adoption of MDD because we are addressing potential disharmonies early. We posit that practitioners do not perceive the added value that conceptual models provide, which contributes to a lack of adoption of conceptual modeling as a viable method for improving software quality. Developers thus sacrifice software quality (whether intentionally or unintentionally) due to not adopting conceptual modeling techniques. Our hypothesis is that TD-aware modeling is a viable tool that can help preserve software quality allowing practitioners to manage debt in the early stages of the software lifecycle, independent of programing languages. It is thus critical for us to seek answers through quantifiable means to three foundational activities in the software specification phase.

Q1. *Do MDD modeling and mapping (model-to-code transformation) processes introduce additional debts to the code?*

***Q2**. Does the evolution of conceptual models at the architectural level introduce additional debts? And*

***Q3**. Which debts need to be removed?*

If the answer to either Q1 or Q2 is positive, then our modeling efforts could be contributing to inadvertent TD [32] – *"… whilst this form of debt is not entered into consciously, it still accrues interest and needs to be repaid"*. By using the TD metaphor, we can develop taxonomies and measurable quantitative methods that can reduce the principal and interest (cf. section 2) in models before mapping such models to executable code. TD principal and interest metrics as well as treating intentional debts as first class citizens will help provide recommendations that address the concerns asked by Q3 and allow for prioritization of debt removal at the modeling level. Not all debt is strategic debt [17], and some debt may not necessarily need to be proactively removed because it may either not have long-term consequences or because the mapping process from the MDD to the implementation phase may not reflect such disharmonies. This position paper is organized as follows: in section 2 we provide an abridged vernacular for MDD and TD, in section 3 we describe various approaches to important topics that must be addressed in order to manage TD in models towards helping researchers address the questions posed in the introduction. In section 4 we provide a discussion, in section 5 we describe related work, and conclude with future directions in section 6.

2. DEFINITIONS

i. *Technical Debt Principal* – Refers to the cost or effort (measured monetarily or in time units) necessary to restore a software artifact back to health.
ii. *Technical Debt Interest* – A measure of the amount of extra work (above and beyond the normal maintenance effort) that is required to restore a software artifact back to health.
iii. *Technical Debt Interest Probability* – Refers to a time sensitive measure. The probability that if the interest on a software artifact is left unpaid, then it will make it (and possibly other artifacts) more expensive to fix in the future.
iv. *Model Driven Development (MDD)* – Refers to a development paradigm that uses models as the main artifact of design. It is important to distinguish MDD from Model Driven Architecture (MDA) which is a specific implementation of MDD by the Object Management Group (OMG [23] [24]).
v. *Model Driven Engineering (MDE)* – Refers to an overarching process that subsumes MDD. MDE also refers to all other aspects of software engineering beyond modeling that use models.
vi. *Model Driven Technical Debt (MDTD)* – Extends the preliminary definitions of TD by including TD introduced during model evolution and during the model-to-code mapping processes (cf. section 3.1).
vii. *Model Rot* – Refers to the deterioration of the structural integrity of a model when compared to its meta-model. This is a corollary to design pattern rot [14].
viii. *Model Grime* – Refers to model disharmonies that do not break the structural integrity of the model, but contribute to the obfuscation of said model. This is a corollary to design pattern grime [14].

3. APPROACHES

We pose that in order to address the questions introduced in section 1, the following methods could be followed. To find answers to Q1 researchers can use existing code based approaches that measure TD immediately following the MDD modeling and mapping phases, and to answer Q2 researchers can track the level of conformance that models have with respect to their meta-models as a longitudinal function. Deviations can be addressed and model-refactored [18] before mapping to an executable model. Deviations from a meta-model come in two forms, *model rot* and *model grime* (defined in section 2). The answer to Q3 will come in the form of a prioritized list with associated principal and interest scores. We suggest four approaches borrowed from a proposal by Seaman [27].

Addressing Q1 will reveal new types of debts, or debts in an abstract form that will facilitate the development of a new taxonomy of TD observed at the architectural and modeling level, and an underlying theory for improving our understanding of Model Driven Technical Debt (MDTD), while addressing Q2, which relies on responses to Q1, aims to provide concrete empirical evidence that additional debts can be introduced through the evolution of a model and before the implementation phase thus increasing the TD principal and interest of the overall system. Addressing Q3 is necessary if we are to increase the adoption of MDE by practitioners.

3.1 Model Driven Technical Debt (MDTD)

While TD has been recognized as an intrinsic characteristic of software products, its original description (*"not quite right code which we postpone making it right"* [4]) has focused its detection and analysis on code. MDTD is a term borrowed from TD, but applied at a higher level of abstraction. We could be justified in saying *"not quite right model which we postpone making it right."* As previously stated, current definitions and taxonomies of TD are based on decisions made at implementation phases without regard to code obtained from implementation-independent models that occur at the architectural level during the specification phase. By extending existing taxonomy to the architectural specification level, we can measure and remove technical debt before the implementation phase and thus improving the adoption of MDD.

A conceptual model is an abstract representation of the concepts that compose one of the views of a software system. It facilitates the communication among developers and with stakeholders about the product being developed, abstracting the complexity of implementation details. From prescriptive plan-driven to agile proposals, different development methods make use of conceptual models, supporting the decision making about several aspects of the product and its development process. These modeling decisions are later reflected in code. For this reason, the possibility of adding TD at conceptual level is worthy of attention.

Figure 1 describes a scenario of TD in conceptual models. In this scenario, a conceptual model evolves in time through different versions (exemplified in Figure 1, as Mv1 and Mv2). Each version may incur TD (TD1 and TD2, respectively). From one version to another, each evolution step is defined by a set of changes, some of them (Δ2) adding TD (TDx). As a result, the TD of Mv2 corresponds to the accrual of debt from its preceding version (i.e. Mv1) and the newly introduced debt (TDx) as a consequence of Δ2. It is important to note that this example depicts a degenerate case where we assume that a linear function can capture the changes in technical debt in an evolutionary step. Our position is that this is significantly more complex, and that the understanding of a function is dependent on empirical validation and multiple factors, some of which will be more dominant than others.

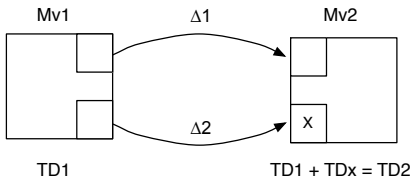

Fig. 1. Technical Debt introduced in the modeling process

Thus, in a generalized model, *TD2 = f (TD1, Δ2)*. In order to fully specify this scenario, we need to:

a) Describe and measure TD of a model version,
b) Describe and measure TD introduced by modeling tasks, and
c) Identify and describe these modeling tasks (changes) that introduce TD.

Another aspect of MDTD concerns the effects of model mapping and its ripple effects on the implementation. Whether performed manually, automatically or semi-automatically, we must be aware that **the model-to-code mapping process can add technical debt** as well. Our concept of MDTD also includes the debt incurred by developers when adopting decisions with respect to which conceptual elements must be mapped onto code, which mapping rules should be applied, and the very implementation and execution of these rules. These decisions can greatly affect the future maintenance of the product, so the characterization of this **model-to-code TD** is necessary. Model-based code-generation tools can be especially useful for this purpose.

Figure 2 illustrates the degenerate scenario (where we assume linearity) of model-to-code mapping process, in which the two versions of a conceptual model from Figure 1 generate (in a systematic or automatic way) corresponding pieces of code (Cv1 and Cv2) at the implementation level. As a result of the mapping process, TD from source models (e.g., TD1) is likely to be transferred to code, but characterized at a lower abstraction level (TD1'). Furthermore, the mapping process itself may introduce an additional TD (TDm2c in Figure 2) to the resulting code.

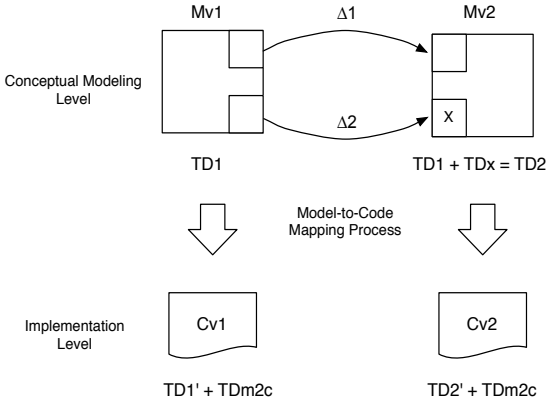

Fig. 2. Technical Debt introduced by the model-to-code mapping process

3.2 Characterization of MDTD

Characterization of TD in models, just as in code, should be based in a definition of a **model-based taxonomy of technical debt**. Toward this goal, previous code-based taxonomies and detection techniques may be reused, by analyzing how said taxonomies can be raised to a conceptual modeling level. For example, the theoretical framework of TD proposed by Tom et al. [32] can be extended to include MDTD. Additionally, an initial TD landscape [15] could be extended to include MDD with a goal to further understand the gaps and overlaps that may exist at higher levels of abstraction. From another perspective, previous research on *conceptual model quality* [19], and concepts from model refactoring, such as *model smells* (redundancies, ambiguities, inconsistencies, incompleteness, non-adherence to design patterns, abuse of the modeling notation, etc.) [18] could serve as the basis for introducing a model-based taxonomy counterpart to existing code-based approaches to TD, and to establish a reference that can be used as a blueprint to which researchers can compare model versions and calculate debt. The work performed by Giraldo et al. [11] used Moody's rule definitions [20] to characterize TD at the model level. Work by Izurieta and Bieman [14] compare instances of design patterns to model characterizations of patterns (meta-models) written in RBML [16] used to represent the abstractions of design patterns. Basic concepts of TD, such as *principal*, *interest* and *interest probability* must be defined at the model-level as well.

3.3 Quantification of MDTD

Quantification of TD in models, on the other hand, needs a previous definition of corresponding **model-based metrics of technical debt**. Our position is that researchers should analyze and re-visit code-level metrics, but make them relevant at a conceptual modeling level. Nugroho et al. [21] identified TD through static analysis of code. They used Lines of Code (LOC), McCabe's complexity number and code duplication to estimate the amount of expected maintenance for a software artifact. CAST [3] also used static analysis of code to identify and quantify TD, and the Sonar tool [29] has surfaced as a widely used tool in the community that also uses static analysis techniques to produce a TD number in terms of a dollar amount that represents the TD principal necessary to repay the debt.

From a modeling perspective, quality metrics of conceptual models [10] may be studied in order to select which evaluated quality criteria fit with the characterization of TD that is proposed. It is important to note that quantifying TD at the architectural modeling level is not necessarily tied to the Unified Modeling Language (UML) [33]; however UML does provide a de-facto standard that can be used to measure deviations from structure as well as behavior. For example, declarative meta-models of structure and behavior have been developed by France et al. [9] using the RBML language [16]. The later can be augmented with constraints of semantic equivalence to formal methods.

3.4 Model Evolution and MDTD

Concerning the evolution of models through different versions, researchers will need to describe how the delta from one version to the next affects the calculation of MDTD in a model. Characterizing modeling tasks that introduce MDTD will help understand the nature of these modeling decisions, their likeliness and intentionality. Furthermore, it will help predict the type and amount of MDTD to be added and its interest probability, from changes that modelers introduce into different versions. The research community should be able to express every change from one version to the next in terms of cost and value over time. Possessing the knowledge that longitudinal changes are not independent, we argue that the complexity of the probable correlation between technical debts added by different changes can be more manageable at a high abstraction level. Research and case studies on model version management and model refactoring

techniques will help support and build empirical evidence toward understanding complex dependencies which can be addressed before coding phases.

To exemplify model evolution we draw from previous work on design pattern evolution [14] [13] [12]. *Design pattern grime* is a particular type of MDTD that can be analyzed from a high abstraction level. By focusing on design patterns we can identify code constructs that conflict with well-formed pattern structures or models of said patterns. *Grime buildup* in design patterns is a form of MDTD that does not break the structural integrity of a pattern but can reduce system testability and adaptability because the structure of the pattern becomes obfuscated as the pattern realization and/or its surrounding environment evolves. The agreed upon structure of design patterns provide a unique opportunity where researchers can compare pattern realizations against their intended structure. Schanz et al. [26] developed a taxonomy of grime and characterized its nature in design patterns to allow for objective quantification. By focusing on design patterns (micro-architectures of models), researchers can examine well-formed structures against design quality violations much more accurately and earlier in the lifecycle of the software.

Design pattern grime is a form of decay that does not break the structural integrity of a pattern; instead, it is the buildup of unrelated artifacts in classes that play roles in a design pattern realization. Unrelated artifacts do not contribute to the intended role of a design pattern and increase the MDTD of a system. Although many types of grime have been identified, a significant contributor to MDTD is *modular grime*, which is indicated by increases in the coupling of the pattern as a whole by tracking the number of relationships (generalizations, associations, dependencies) pattern classes have with external classes. Figure 3 shows a visual intuition of grime buildup. In 3(a) we display an example of three interacting classes through agreed upon relationships. Over time, these classes develop unintended relationships (shown in 3(b)) that contribute to the obfuscation of the design, thus reducing its quality.

Different types of grime relationships that accumulate over time have consequences on the adaptability and the testability of the software [13]. The effort required to maintain these relationships above and beyond maintaining the same design pattern without these relationships represents MDTD interest. This is the additional *pain* incurred on developers or maintainers of the design pattern, i.e. the model.

Strasser et al. developed a toolkit [30] that measures the distance that a given pattern realization model (reversed engineered from the code) is from an ideal implementation of that pattern (a meta-model described in RBML). The distance represents the TD principal. The approach used by this tool calculates technical debt holistically, i.e. it yields an overall assessment of the total TD in a pattern, but does not point to specific problematic parts of the design pattern, or specific remedies applicable to those parts.

3.5 MDTD Removal

Debt removal is an area that has not been explored in detail – whether in the specification phase of the software process. There are two aspects that require consideration when addressing debt removal: deciding which software artifacts require attention, and once a decision is made, refactoring. In a position paper by Seaman et al. [27], the authors explain that long term financial tradeoffs that affect the quality and maintenance of software products are usually not incorporated into the decision making of which enhancements or new functionalities need to be tackled next. Developers typically tackle issues that have short-term immediate impact. Seaman et al. propose four approaches that can also be used during the specification phases and modeling of software: Simple Cost-Benefit Analysis, Analytic Hierarchy Process (AHP), the Portfolio Approach (PA), and Options. Any one of these approaches can be equally applied at higher levels of abstraction provided researchers have an agreed upon MDTD taxonomy (as previously proposed in section 3.1) and a set of metrics that help normalize these techniques. A significant body of literature on refactoring approaches is available. From single view techniques to the application of chains of refactorings to improve a model exist; however some of the first research done on refactoring models (at high abstraction levels) was performed by Sunyé et al. [31] using the Object Constraint Language (OCL).

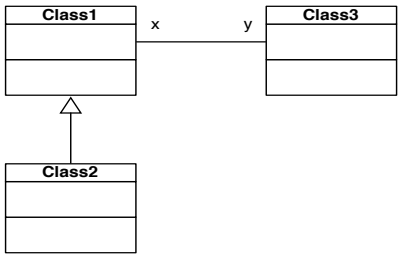

Fig. 3a. A UML structural diagram of a clean design

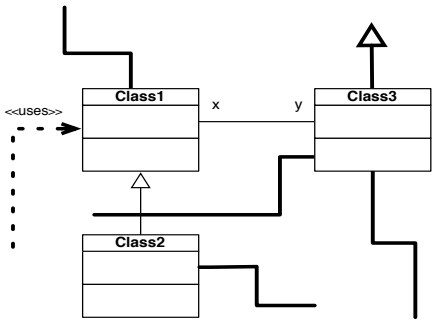

Fig. 3b. A visual intuition of modular grime buildup. Some relationships (shown as thick lines) represent the grime that has accumulated as the pattern evolves

With regards to our example (cf. 3.4) the decision to remove grime from a design pattern is dependent on the type of grime observed. Studies by Bieman and Wang [1] suggest that the removal of coupling dependencies depends on the type of said dependency. Relationships between classes are classified in an ordinal scale. A low amount of effort (principal) is necessary to remove dependency relationships. Associations amount to higher principals and generalizations are even more costly.

The benefits of removing grime can be quantified by observing the number of tests cases necessary to test said relationships. By effectively removing MDTD represented as relationships we are proactively reducing the number of test cases that would be necessary to test the software after the mapping phase to an implementation. As systems evolve, new relationships develop between classes. Added relationships represent added test requirements. These relationships may or may not have been intended in the original design. More often than not, such relationships are the consequence of modular grime buildup. Without the necessary updates to the testing suite of such systems,

the possibility of faults grows. Reducing the number relationships identified as MDTD is then significant when improving the quality of resultant systems. The formulas to count the number of tests were originally proposed by Binder [2] and then expanded by Izurieta et al. [13] in the context of implemented systems. The number of test cases that are not necessary as a result of paying off the principal associated with a particular design pattern constitutes a reduction of maintenance and thus interest.

A higher number of dependencies decrease the comprehensibility of the model. Even though the design pattern realization remains as the system evolves, it becomes obscured thus reducing the adaptability of the pattern. The effort (principal) required to extract the realization from a design, or to make changes to the pattern increases because the developer needs to understand and account for the additional couplings that distort the realization of the pattern.

4. DISCUSSION

Although model-based quality analysis *per se* is a worthy research line on its own, the effects of TD at modeling levels (MDTD) are perceptible in final products. For this reason, it is relevant to study how the TD that has been identified, classified and measured at conceptual levels is transmitted to the implementation level. Studies will help researchers conclude if there exists a correlation between model-based and code-based TD, or if both are absolutely independent of each other, or if there are dependencies between them, not necessarily one-to-one, that are worthy of analysis, or if modeling tasks introduce a type of debt that is not transferred to code. We expect that this line of inquiries will help us determine whether TD can be actually added at conceptual levels or, if talking about TD in models makes sense.

Another aspect that reinforces the idea of studying TD from a model-driven perspective is the fact that many quality issues that are used to describe this concept correspond to those that the model-driven community has targeted to reduce. For instance, from the TD framework proposed by Tom et al. [32], model-driven development directly influences the following types of debt: design and architectural debt (facilitating its analysis independently of implementation languages); environmental debt (automatic code generation can relieve manually introduced TD), documentation debt (model-driven documentation, from models, can help preserve traceability), monetary cost (cost associated to conceptual models), and amnesty (improving reuse of models in different platforms, or core concepts in different domains).

Finding answers to these research questions will open many more areas worthy of investigation. For example, *What percentage of TD is introduced during the modeling phase?* or *Does MDD help address TD earlier?* and *Does the value added by MDD with respect to addressing TD outweigh the current limitations of MDD?* If the answer is 'yes', then software engineers will begin adopting MDD.

5. RELATED WORK

Although much literature is available on model driven techniques, and especially (as of the last five to ten years) technical debt, it is impossible to succinctly summarize these contributions in a short missive; however, as it relates to this paper, it is important to mention the work by Giraldo et al. [11] where the authors confirm that there exists no corpus of work on technical debt calculus outside implementation phases of development. They discuss the importance of calculating technical debt in a model-driven context. In their work they used SonarQube [29] on projects created on the Eclipse Modeling Framework (EMF) and where XML was used as the model specification language. Rather than focus on calculating technical debt at the code level, they developed rules (codified in XSD –XML Schema Description) based on Moody [20] to evaluate debt at the model level. Further, as stated in [5] "*projects are increasingly adopting model-based engineering tools such as SCADE or Simulink to specify the functional architecture,"* and to generate code automatically from models. Although the later is being researched in the context of reducing system complexity, clearly metrics developed in this space [25] [22] are relevant to quantifying and thus reducing MDTD.

6. CONCLUSION AND FUTURE DIRECTIONS

Adopting a new taxonomy or complementing an existing one with concepts that define technical debt at the modeling level during software specification is necessary if our goals are to increase the adoption of modeling with regular practitioners and to decrease the overall amounts of technical debt in software. A taxonomy will allow for the identification of new metrics that will allow researchers to quantify possible debt before the implementation phase. It is our position that reducing MDTD will in turn reduce TD in implemented systems thus improving the overall quality of a system and reducing maintenance costs.

Taxonomy of TD at code levels and techniques for its detection, quantification, and removal, will form the basis for defining corresponding taxonomy and techniques at the conceptual level, which can be extended according to the nature of decisions adopted in the modeling process. Empirical studies that help validate this theoretical framework can be supported through MDD tools. For instance, conceptual models can be automatically analyzed in order to detect the realization of design constraints or patterns, while MDTD metrics measure the amount of associated TD principal that a model incurs, described as the distance between the actual and ideal realization of the rule.

Empirical analysis of such implementation could help identify and classify those model modifications that add MDTD. A complementary functionality could warn developers about the TD interest probability of the modifications to be introduced into the model. Analogously, some model modifications can help reduce MDTD. By means of model-to-model transformation rules, refactoring can be implemented to reduce or remove detected MDTD. In order to study MDTD introduced in the mapping process, we can take advantage of existing code generation tools in order to estimate how much TD is added in the model-to-code mapping process. A comparative analysis of MDTD in equivalent pieces of code, one automatically generated from a model and the other written from scratch, can help fully describe the concept of MDTD and understand the nature of the TD incurred according to the strategy adopted to obtain code.

7. REFERENCES

[1] J. M. Bieman and H. Wang, "Design pattern coupling, change proneness, and change coupling: a pilot study," Technical Report. Colorado State University, 2006.

[2] R. Binder, Testing Object Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Publishers, 2000.

[3] CAST, "Cast worldwide application software quality study: Summary of key findings," Technical Report, Available:

http://www.castsoftware.com/resources/resource/cast-research-labs/cast-worldwide-application-software-quality-study-2010/, 2010.

[4] W. Cunningham, “The WyCash portfolio management system,” in Proceedings on Object-oriented programming systems, languages, and applications (Addendum) (OOPSLA '92), ACM, New York, NY, USA, pp. 29-30, 1992.

[5] J. Delange. Blog: Managing Model Complexity. http://blog.sei.cmu.edu/post.cfm/managing-model-complexity. Software Engineering Institute (SEI), Carnegie Mellon University. Accessed 12/29/2014.

[6] B. Dobing and J. Parsons, “How UML is used,” *Commun. ACM*, 49, 5, pp.109-113, May 2006

[7] M. Fowler, Refactoring: Improving the Design of Existing Code. Addison-Wesley. 1999.

[8] A. Forward and T. C. Lethbridge, “Problems and opportunities for model-centric versus code-centric development: A Survey of Software Professionals,” in Proceedings of the International Workshop on Models in Software Engineering (MiSE’08)@ICSE’08, pages 27-32, New York, 2008.

[9] R. B. France, D. K. Kim, E. Song, and S. Ghosh. 2002. Patterns as Precise Characterizations of Designs. Technical Report. Colorado State University.

[10] M. Genero, M. Piattini, and C. Calero. Metrics for software conceptual models. London: Imperial College Press, 2005.

[11] F. D. Giraldo, S. España, M. A. Pineda, W. J. Giraldo, O. Pastor., “Integrating Technical Debt into MDE,” 26th International Conference on Advanced Information Systems Engineering, Pre-proceedings CAISE ’14 Forum, 16-20 June, Greece.

[12] C. Izurieta and J. Bieman, “How software designs decay: A pilot study of pattern evolution,” in Proceedings of the First Symposium on Empirical Software Engineering and Measurement (ESEM 2007), Madrid, Spain, pp. 449-451, September 2007.

[13] C. Izurieta and J. Bieman, “Testing consequences of grime buildup in object oriented design patterns,” in Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST 2008), Lillehammer, Norway, pp. 171-179, April, 2008.

[14] C. Izurieta C., and J. Bieman, “A multiple case study of design pattern decay, grime, and rot in evolving software systems,” Software Quality Journal, vol.21, pp. 1-35, June 2013.

[15] C. Izurieta, A. Vetro, N. Zazworka, Y. Cai, C. Seaman, and F. Shull, “Organizing the technical debt landscape,” in Proceedings of the Third International Workshop on Managing Technical Debt (MTD’12), Zurich, Switzerland, pp. 23-26, June 2012.

[16] D. K. Kim, “A meta-modeling approach to specifying patterns.” Colorado State University PhD Dissertation, June 21, 2004.

[17] S. McConnell, “Technical debt,” in: 10x Software Development, Available from: http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx (Online), 2007.

[18] T. Mens, Taentzer, G., and Müller, D., "Model-Driven Software Refactoring," in Model-Driven Software Development: Integrating Quality Assurance, IGI Global, pp.170-203, 2009.

[19] D.L. Moody, G. Sindre, T. Brasethvik, and A. Sølvberg, "Evaluating the quality of information models: empirical testing of a conceptual model quality framework," in Proceedings of the 25th International Conference on Software Engineering. IEEE Computer Society, Portland, OR, USA, pp. 295-305, May 2003.

[20] D. L. Moody. “The physics of notations: Toward a scientifc basis for constructing visual notations in software engineering,” IEEE Transactions on Software Engineering, 35(6):756-779, 2009.

[21] A. Nugroho, J. Visser, T. Kuipers, “An empirical model of technical debt and interest,” in Proceedings of the Second International Workshop on Managing Technical Debt (MTD’11), New York, NY, USA, pp. 1-8, May 2011.

[22] M. Olszewska. “Simulink-Specific Design Quality Metrics,” TUCS Technical Reports 1002, Turku Centre for Computer Science, 2011.

[23] OMG, Object Management Group: Meta Object Facility (MOF) 2.0 Core Specification. Final Adopted Specification pct/03-10-04 edn. from http://www.omg.org/docs/ptc/03-10-04.pdf, 2003.

[24] OMG, Object Managament Group: MDA Guide Version 1.0.1. Document omg/03-06-01 edn from http://www.omg.org/docs/omg/03-06-01.pdf, 2003.

[25] J. Prabhu. “Complexity Analysis of Simulink Models to improve the Quality of Outsourcing in an Automotive Company,” Manuscript, August 2010. http://alexandria.tue.nl/extra1/afstversl/wsk-i/prabhu2010.pdf

[26] T. Schanz and C. Izurieta, “Object oriented design pattern decay: a taxonomy,” in Proceedings of the 4th ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2010), Bolzano-Bozen, Italy, pp. 1-8, September, 2010.

[27] C. Seaman, Y. Guo, C. Izurieta, Y. Cai, N. Zazworka, F. Shull, and A. Vetrò, “Using technical debt data in decision making: potential decision approaches,” in Proceedings of the Third International Workshop on Managing Technical Debt (MTD’12), Zurich, Switzerland, pp.45-48, June 2012.

[28] B. Selic, “The pragmatics of model-driven development,” IEEE Software, vol.20, pp.19-25, September-October 2003.

[29] Sonar. Available online: http://www.sonarsource.org/

[30] S. Strasser, C. Frederickson, K. Fenger, C. Izurieta, “An automated software tool for validating design patterns,” ISCA 24th International Conference on Computer Applications in Industry and Engineering. CAINE ’11, Honolulu, HI, USA, November 2011.

[31] G. Sunyé, D. Pollet, Y. L. Traon, J. M. Jézéquel, “Refactoring UML Models. UML ’01, LNCS, vol. 2185, pp. 134-148. Springer (2001).

[32] E. Tom, A. Aybuke, and R. Vidgen, “An exploration of technical debt,” The Journal of Systems and Software 86, pp. 1498-1516. 2013.

[33] Unified Modeling Language (UML). Available online: http://www.uml.org