Towards Assessing the Technical Debt of Undesired Software Behaviors in Design Patterns

Derek Reimanis, Clemente Izurieta
Computer Science Department
Montana State University
Bozeman, Montana, USA, 59715
derek.reimanis@msu.monana.edu, clemente.izurieta@montana.edu

Abstract—**Providing software developers and researchers with useful technical debt analysis tools is an instrumental outcome of software engineering and technical debt research. Such tools aggregate industry best practices to provide users with organized and quantifiable metrics that can help minimize the time it takes to synthesize and make an intelligent future decision regarding a system. Today, most tools rely primarily on structural measurements from static analysis to generate results. However, it is also necessary to consider measurements that capture the behavior of software, as these represent additional complexities within a system that structural measurements are incapable of detecting. Herein, we present our position; that more effort needs to be placed towards understanding software behavior so that technical debt analysis tools can begin supporting them, in order to provide tool users with a more accurate and complete view of their system. In this paper, we describe this problem in the context of design patterns and outline an effective method to talk about behaviors in the future. We create and classify two example behaviors using our method, both of which increase the technical debt in their respective design pattern applications.**

Keywords—technical debt, software behavior, design patterns, software architecture

I. Introduction

Design patterns are common solutions to recurring problems that appear in the design phase of software development. The use of design patterns is known to be a good practice because patterns offer maintainable, understandable, and reusable solutions to problems [1]. However, empirical results have shown that design patterns tend to stray from the initial intent as they evolve, which can cause the pattern to lose its beneficial qualities [2] [3] [4] [5] [6] [7]. Recent research has attempted to quantify this loss by studying the effects that either the addition of non-pattern elements to a pattern or the removal of vital pattern elements have on a pattern instance and the software project as a whole [2] [3] [4] [5] [6] [7]. It has been found that either or both of these cases cause the quality of the pattern instance and the software project to decrease, especially when considering maintainability and testability factors.

The structural diagram of a design pattern describes valid classes and relationships that are contained within a pattern, and characterizes instances of a pattern. Structural characteristics are generally represented by class diagrams in the Unified Modeling Language (UML) [8]. Alternatively, the behavioral characteristics of a pattern refers to the runtime events that occur as the pattern is executed by a controlling element in the software. In nearly all cases, excluding patterns with a linear control flow, a pattern has a potential to execute numerous behavioral alternatives. This is because the flow through a design pattern has the potential to change based on the events that transpire before and during the pattern's runtime execution. All previous work has studied patterns from the structural rather than behavioral perspective. It is the latter that serves as motivation for this line of research. The long-term goal of this research is to understand the effects that various unintentional elements play in the role of a design pattern, especially as the design pattern is executing. We hypothesize that many design patterns have unintentional elements that have an adverse effect on the runtime behavior of the pattern. Furthermore, it is likely that many of these elements only appear under intricate circumstances, which require thorough evaluation to detect. We refer to these unintentional elements as unintentional runtime artifacts. Our work focuses on the identification and the categorization of these unintentional runtime artifacts, and we present these in the context of two design patterns [9].

Our **position** is that more effort needs to be directed at understanding software behaviors so that analysis tools may begin incorporating them to provide practitioners and researchers alike with a more accurate and holistic view of a software system. Today, most tools use structural static analysis strategies to provide an assessment of quality or a calculation of technical debt, and behavioral analytic methods are technologically lagged compared to their respective structural counterparts. This is indicative of a research gap. Ideally, both structural and behavioral analysis strategies should be incorporated in the assessment of technical debt to offer a more rich and holistic view of a software system. Therefore, increased focus should be brought towards understanding software behaviors. To this end, we focus on design patterns to identify, characterize, and evaluate behaviors so that deviations from the norm can also be accounted for in technical debt assessments.

We look at identifying and characterizing two specific behaviors. We describe these behaviors in terms of the Role-Based Metamodeling Language (RBML), which is an established language used to describe solution domains at the metamodel level [10]. To validate our approach, we utilize *in-*

vitro [11] synthetic examples that are meant to be representative of real word behaviors. The examples are restricted to instances of design patterns, as they have formal well-defined specifications that describe required structure and behavior [12].

II. Background and Related Work

A. *Software Behavior*

Software behavior can be categorized into two classes; *internal* and *external* [13]. Internal behavior refers to the interior mechanisms and API calls that occur during a system at runtime. Internal behavior draws parallels from white-box testing in the software testing domain [14]. External behavior refers to the external, observable results that the system produces. Conversely from internal behavior, external behaviors draw parallels from black-box testing in the software testing domain [14]. In this sense internal behaviors are temporary artifacts that exist only for the duration of their execution, while external behaviors are representative of system states or goals. External behaviors are caused by internal behaviors.

B. *Software Decay*

The notion of decay in the field of software engineering entails that software evolves or change over time. As a consequence, code decay refers to any code that is "harder to change than it should be" [15]. Design pattern decay is a specific type of code decay and is defined as the addition or loss of undesired elements, or artifacts, in a design pattern instance, over the lifetime of the design pattern [2] [3] [4]. Design pattern decay targets designs and can be considered sub-domains of the "model smell" concept –a high level counterpart of code issues, and model refactoring (originally explored by Sunyé el al. [16]).

Previous explorations of design pattern decay have focused on identifying and measuring undesirable elements only from a structural perspective [2] [3] [4] [5] [6] [7] [17]. Prior art [4] suggests the classification of these elements as *design pattern grime* and *design pattern rot*. Grime refers to artifacts that obscure the integrity of the original pattern, and rot refers to artifacts that obsolete the integrity of the original pattern. Design pattern grime can be either *structural* or *behavioral* [9]. Structural grime refers to the structural elements contained in a pattern, considering the packages, classes, operations, and relationships of the pattern elements to other elements in the pattern or software project as a whole [2]. A pattern instance is said to have structural grime when the elements within the pattern instance do not exist in the pattern's specification. Specifically, this occurs when the pattern instance does not exactly match the Structural Pattern Specification (SPS) definition for that pattern [12]. The SPS consists of two facets; a class diagram described in the Role-Based Meta-Modeling (RBML) [18] and constraints described in the Object-Constraint Language (OCL) [19].

Behavioral grime refers to the behavioral elements contained in a pattern, specifically pertaining to the constructor(s) of classes within the pattern as well as operations within that pattern [9]. Akin to structural grime, a

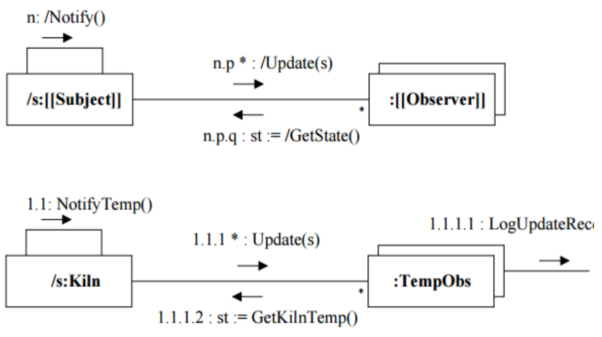

Figure 1 – Diagram of the RBML sequence diagram for the Observer pattern [10]. The Subject and Observer roles and their sequence of behaviors are shown on the top, and a conforming pattern instance is shown on bottom. The pattern instance conforms to the RBML sequence diagram.

pattern instance is said to have behavioral grime when the behavioral elements within the pattern instance do not exist in the pattern's specification. The specification for behavioral grime is referred to as an Interaction Pattern Specification (IPS) [12]. Similarly to an SPS, a pattern's IPS consists of two facets; a sequence diagram described in RBML and constraints described in OCL [19]. Figure 1 [10] shows the RBML sequence diagram of the IPS for the Observer pattern and a conforming pattern instance that tracks the temperature and pressure of a kiln. The RBML sequence consists of the two roles in the Observer pattern; the Subject and the Observer. The two roles and their expected behaviors are shown on the top of the diagram. For behavioral conformance, it is expected that the Subject role calls the Update() operation on all the Observer roles when the Notify() operation is called. Then, the Observer calls the Subject using a GetState() operations. The bottom of the figure shows a pattern instance. Notice that the implementing classes in the sequence diagram conform to their role expectations.

C. *Technical Debt*

Technical debt (TD) is a metaphor established by Ward Cunningham to describe the gap between the current state of a software system and the ideal state [20]. In essence, TD captures the effects of decisions that sacrifice good design principles for on-time delivery of software. Many times these decisions take the form of shortcuts or workarounds in code that complete the task at hand, but at the expense of decreased quality. TD is analogous to financial debt in that some debt is beneficial, because it facilitates growth, but too much debt becomes a burden because of the need to repay it at the expense of valuable resources. Drawing parallels from financial debt, principal and interest are two attributes of TD. Given a task to implement, principal refers to the cost in effort to complete the task. Interest refers to the gap between maintenance costs under ideal conditions versus conditions where maintenance is higher due to accrued debt from tasks where TD is not repaid. Effectively managing TD is multi-faceted problem, because the need the need to implement new features must be leveraged with the need to refactor to cleanse the code-base.

III. Preliminary Work

A. Approach

While the end-goal of this research is to provide TD analysis tools that can capture errant behavior, it is necessary to first establish a common vernacular to describe behavior. We focus on the identification and classification of undesired, or errant, behaviors. Specifically, we were able to identify two errant behaviors based on common mistakes that can occur while implementing design patterns. We know these behaviors are errant because they detract from the intent of the design pattern and thus, have an undesired effect on TD. We classify these two errant behaviors as *Excessive Action(s)* and *Improper Order of Sequences*, and illustrate how they violate the IPS of the observer design pattern.

B. Errant Behaviors

1) Excessive Action(s)

The first errant behavior we consider involves one or more 'excessive' action(s) that occur during the standard runtime operation of the pattern. The excessive action(s) perform operations that are un-essential to the functional runtime behavior of the pattern. That is, if the excessive action(s) were to be removed, the pattern would behave in entirely the same manner (as expected). We characterize this type of behavior as behavioral grime because the excessive action(s) cause the pattern instance to not conform to the pattern's IPS. The excessive action(s) are not necessarily structural grime, but may be the result of implementing new functional requirements. Regardless, the intent of the pattern, according to its IPS is violated. The addition of excessive action(s) affects software maintainability and TD, because while the pattern will achieve the same external behavior, modifying the pattern in the future will require domain knowledge of the action(s) and their intent.

Our illustrative implementation of this errant behavior uses a loop construct that counts to 100 and sums values along the way. The loop's internal and external behaviors are not referenced by any other components in the remainder of the software application. Practically speaking, this may happen when a developer forgets to delete debugging code, or decides upon a different strategy for the implementation of an algorithm half-way through development and forgets to delete the original code.

Although all design patterns allow for the introduction of new tasks (i.e., excessive actions) as interspersed elements of existing behaviors, the introduction itself needs to be explicitly described by elements of the RBML IPS diagram. If not, then the introduction of the behavior is unintended, even if it provides needed functionality for the software. The IPS of a design pattern must then be responsible for capturing the strictness of adherence with which instances are created. Only then, new functionality can be planned for without affecting TD. To illustrate how this behavior violates the RBML IPS diagram in our implementation, refer to figure 2 [10]. The Observer RBML IPS is shown on the top. Our implementation of the Observer pattern is shown on the bottom. This figure is similar to figure 1, with the exception that the excessive behavior has been injected. The excessive

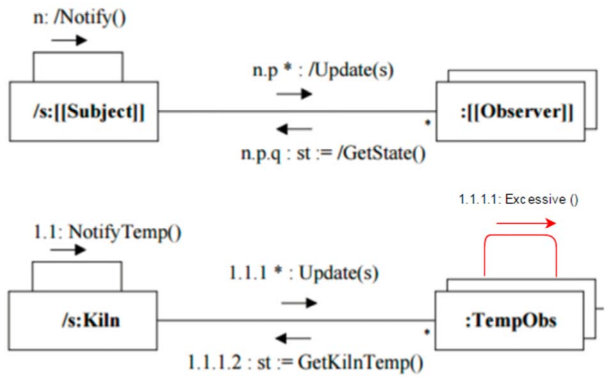

Figure 2 – Excessive behavior grime in an Observer pattern instance (ObserverExcess) [10]. The top sequence of this image illustrates the RBML IPS of the Observer pattern. The bottom sequence illustrates our application of the Observer, with the injected behavior 'Excessive()'. The injected behavior constitutes behavioral grime because it does not conform to the pattern's IPS.

behavior is shown in red, and labeled with the operation 'Excessive()'. Notice that in the RBML IPS, the Observer role only performs one operation, which is to call GetState() from the Subject role. Conversely, the TempObs class performs two operations, Excessive() and then GetState(). Because of this, TempObs has behavioral grime.

2) Improper Order of Sequences

The second errant behavior we consider involves cases where the order of operations that a pattern should be following, according to the SPS and IPS, is improperly sequenced. In this case, the pattern instance's external behavior will satisfy the desired functionality, but the sequences of method or class calls that occur during the internal behavior of the pattern instance are not correctly ordered. This type of errant behavior constitutes behavioral grime because it causes the pattern instance to not conform to the IPS of the pattern. Additionally, this type of errant behavior affects the maintainability and TD of the pattern for much the same reasons as the excessive action(s) does; any modification of the pattern instance in the future will be hindered by the need to first understand the order of sequences in the application.

In our implementation of this errant behavior, we injected a class that represents a valid extension of each pattern. However, the injected class was only instantiated from the incorrect class role in each pattern. In other words, in our Observer pattern instance we injected an observer that only received updates from other observers, and in the Visitor pattern instance we injected a visitor that only received contacts from other visitors. This is incorrect behavior for both patterns because in the Observer pattern instance, the subject should be responsible for updating the observer, and in the Visitor pattern instance the client should be responsible for contacting the visitors. This injection causes the external behavior of the pattern to be preserved, but also causes the internal behaviors to happen out of order. Practically speaking, this would happen if a developer who was unfamiliar with the Observer or Visitor pattern, either a novice developer or a new hire, made changes to the existing pattern.

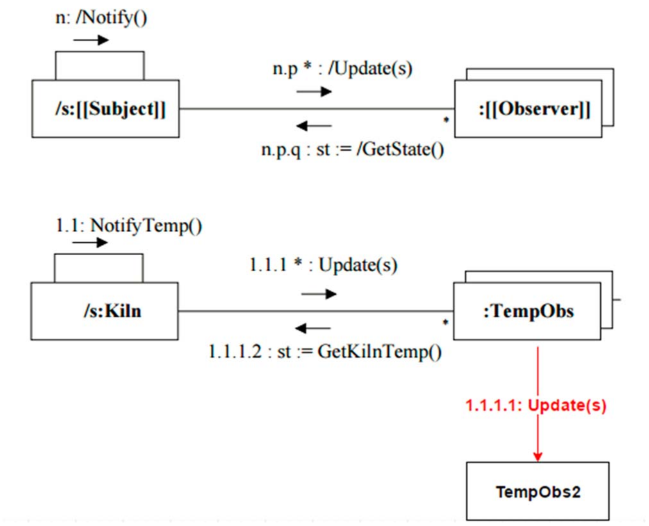

Figure 3 – Improper order of sequences behavior grime in an Observer pattern instance (ObserverImprop) [10]. The top sequence of this image illustrates the RBML IPS of the Observer pattern. The bottom sequence illustrates our application of the Observer, with the injected class TempObs2 and injected operation Update(s) (shown in red). The injected behavior constitutes behavioral grime because it does not conform to the pattern's IPS.

Figure 3 [10] illustrates the improper order of sequences errant behavior in our application. The top sequence of the figure features the RBML IPS of the Observer pattern. The bottom sequence illustrates the behavior of the Observer pattern in our application. The TempObs2 class was added to the pattern, which has the potential to be a proper extension to the pattern instance. However, its state is being updated from the TempObs class, which belongs to the Observer role. This is a violation of the RBML of the Observer pattern because the Subject is responsible for updating the Observers. TempObs2 is being updated immediately after the update to TempObs, even before TempObs has called the GetState() operation. This type of behavior constitutes behavioral grime.

IV. Conclusion

Providing both software developers and researchers alike with useful technical debt analysis tools is an important outcome of software engineering and technical debt research. To date, tools that quantify technical debt focus on the structural aspects of the systems they are analyzing. We believe that while structural aspects are important to analyze, it is equally important to consider behavioral aspects. To address this position, we identified and characterized two errant behaviors that can negatively affect technical debt. This is a vital step towards implementing behavioral measurements in tools that quantify technical debt because it provides a starting point from which metrics can be built. Additionally, we have provided a common language and process that can be used to discover and classify more errant behaviors.

In future work, we would like to identify classes or categories of more errant behaviors. We only considered *excessive action(s)* and *improper order of sequences* in this study. Additionally, we would like to identify or propose metrics that capture these errant behaviors, and especially their effects on the technical debt of a system.

References

[1] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, '94.

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

[3] Izurieta, C. 2009. Decay and Grime Buildup in Evolving Object Oriented Design Patterns. Ph.D. Dissertation. Colorado State University, Fort Collins, CO, USA. Advisor(s) James Bieman. AAI3385139.

[4] Izurieta, C., Bieman, J. 2013. A multiple case study of design pattern decay, grime, and rot in evolving software systems. J. Software Quality. 21, 2 (Jun. 2013), 289-323.

[5] Griffith, I., and Izurieta, C. 2013. Design Pattern Decay: An Extended Taxonomy and Empirical Study of Grime and its Impact on Design Pattern Evolution. In Proceedings of the 11th ACM/IEEE International Doctoral Symposium on Empirical Software Engineering and Measurements, USA

[6] Griffith, I., and Izurieta, C. 2014. Design pattern decay: the case for class grime. In Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '14). ACM, New York, NY, USA, Article 39, 4 pages.

[7] Dale, M.R., and Izurieta, C. 2014. Impacts of design pattern decay on system quality. In Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '14). ACM, New York, NY, USA, Article 37, 4 pages.

[8] Fowler, M. (2004). UML distilled: a brief guide to the standard object modeling language. Addison-Wesley Professional.

[9] Reimanis D., Izurieta C., "A Research Plan to Characterize, Evaluate, and Predict the Impacts of Behavioral Decay in Design Patterns," IEEE ACM IDoESE, 13th International Doctoral Symposium on Empirical Software Engineering, Beijing, China, October 19 2015.

[10] E. S. Robert France, Dae-Kyoo Kim and S. Ghosh, Metarole-Based Modeling Lanugage (RBML) Specification V1.0, 2002.

[11] Juristo, Natalia, and Moreno, Ana M.. Basics of software engineering experimentation. Springer Science & Business Media, 2013.

[12] Kim, D. The Role-Based Metamodeling Language for Specifying Design Patterns. In Toufik Taibi, editor, Design Pattern Formalization Techniques. Idea Group Inc., 2006.

[13] Plasil, F., & Visnovsky, S. (2002). Behavior protocols for software components. IEEE transactions on Software Engineering, 28(11), 1056-1076.

[14] Ammann, P., & Offutt, J. (2008). Introduction to software testing. Cambridge University Press.

[15] Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S., and Mockus, A. Does code decay? Assessing the evidence from change management data, Software Engineering, IEEE Transactions on, 27(11), pp.1-12, Jan 2001.

[16] Sunyé, G., Pollet, D., Le Traon, Y., & Jézéquel, J.M. "Refactoring UML models," In: Proc. Int. Conference Unified Modeling Language (pp. 134-138), Lecture Notes in Computer Science 2185, Springer, Heidelberg. 2001.

[17] Schanz, T., and Izurieta, C. 2010. Object oriented design pattern decay: a taxonomy. In Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10). ACM, New York, NY, USA, Article 7, 8 pages.

[18] Kim, D. 2004. A Meta-Modeling Approach to Specifying Patterns, Ph.D. Dissertation. Colorado State University, Fort Collins, CO, USA. Advisor(s) Robert France.

[19] Warmer, J. B., & Kleppe, A. G. (1998). The Object Constraint Language: Precise Modeling With Uml (Addison-Wesley Object Technology Series).

[20] Cunningham, W. (1993). The WyCash portfolio management system. ACM SIGPLAN OOPS Messenger, 4(2), 29-30.