Diagnosing Degenerate Forms in Software

Brian S. Dillon

Naval Surface Warfare Center, Dahlgren Division (NSWCDD)
Virginia Polytechnic Institute and State University
briand81@vt.edu

Abstract

The degeneration of source code due to maintenance is a long known but little understood phenomenon. Currently, researchers face significant logistical challenges when conducting empirical studies and experiments, studying large-scale projects, and characterizing the development and growth of degenerative forms. These logistical challenges can be partially alleviated by developing automated metrics designed to identify degenerate forms. Furthermore, such metrics are essential for targeted refactoring and repairing degenerative forms. This dissertation research investigates a set of metrics targeted at specific degenerate forms common in software. The successful implementation and characterization of such metrics will enable further research in many forms of software maintenance.

Categories and Subject Descriptors D.2.9 [**Software Engineering**]: Management – software quality assurance.

General Terms Measurement, Management, Design, Experimentation.

Keywords software evolution; degeneration; software metrics; diagnostics; refactoring

1. Introduction

Software evolution is a problem long known to computer science and has been addressed by various names, e.g., aging, rot, entropy, erosion. Essentially, software that undergoes maintenance must change over time, and the cumulative effects of those changes create adverse conditions for future maintenance. These adverse conditions eventually make maintenance infeasible. This problem is found in most large-scale, long-lived software projects but is still little understood.

In order to enable further research in this area, improved metrics are required. This dissertation research investigates several metrics that have been designed to identify specific degenerate software forms commonly found in software demonstrating the effects of software evolution. The goal of this research is to characterize the accuracy and effectiveness of these metrics in locating and diagnosing degenerate forms as an aide to future work.

OOPSLA'14, October 20–24, 2014, Portland, Oregon, USA.
ACM 978-1-4503-2585-1/14/10.
http://dx.doi.org/10.1145/2660252.2660255

2. Problem Statement

The costs associated with software degeneration are high. Even with the many man-hours invested to develop a current software baseline, at some point the cost of maintaining degraded software is more than the cost of new development. Three examples from industry were described in [9] where software was abandoned for this reason. These cases indicated the need to abandon new software shortly after it was released and even while it was still under development. Each case was described as requiring a "massive effort," and they concluded "…[r]edeveloping software … is a very expensive and lengthy procedure…." and in the end "was only partly successful." The cost of redevelopment on multiple software products, over many years and by many different developers, is a monumental expenditure that can and must be eliminated.

Up to this point, research into this phenomenon has been dominated by Lehman's eight laws of software evolution [7]. Lehman continued to revise his own laws over the years as his understanding of the phenomenon changed. Researchers in this field tend to agree that software evolution is a naturally occurring, degenerative phenomenon, which is at least partially the product of development practices, resource constraints, and time. Nevertheless, researchers have not done a great deal to characterize the development of these degenerate software forms, principally because of the logistical concerns in carrying out even a limited study of software "in the wild." While there are a large number of strategies for improving software development, there is little empirical evidence to determine how these strategies affect the rate of degeneration.

The lack of empirical studies, according to multiple researchers, is the result of logistical difficulties in producing statistically significant quantities of sample data. Without such samples, it is impossible to draw meaningful conclusions as to a positive or negative effect derived from specific development strategies. Kemerer and Slaughter [6] noted that research of this type requires at least two different data points from two different times. "This," they note, "creates practical difficulties in terms of sustaining support for the [research] project over this period or finding an organization that collects and retains … data or the software artifacts themselves." Researchers are further hampered by their inability to accurately measure the quality of the software without resorting to expert classification. Metrics do exist that point to various characteristics of code, but none give an objective and quantifiable measure of the degeneration that has taken place.

3. Motivation for Dissertation Research

The author's personal motivation to investigate this problem came from his work as a professional developer. That work involved the refactoring of multiple degenerated legacy software products in preparation for new development. This led to first-hand experience with degenerate software forms and knowledge of the current limits of refactoring. Refactoring has had mixed results across the industry as a whole for reasons identified by [2]. They indicate that i) identifying code smells requires a priori knowledge of the code, ii) code smells and refactoring focus on a small subset of code without any planned effect on the whole, and iii) the relative value of code smells is not easily quantified. In short, the value of refactoring is strongly associated with the experience and knowledge of the developer and, even with an experienced developer, manual detection and refactoring of degenerate forms is limited by these same factors.

In order to enable further research in this area and improve the quality of refactoring in general, there is a need for new automated metrics. Researchers are unable to greatly further our understanding of the degeneration phenomenon in general because they are unable to conduct meaningful studies and experiments on large-scale source code. Developers are unable to efficiently locate, identify, and resolve these degenerate forms because they are limited by human experience and speed. Current software metrics, biased toward easily quantifiable characteristics such as method size and nesting loops, are incapable of improving these conditions. While such metrics may or may not indicate areas of concern, they are hardly diagnostic in the strict sense. New metrics must be developed that are objective and yield quantitative results capable of diagnosing and locating degenerate forms more efficiently.

4. Approach

The proposed research will develop just such a set of metrics that can be used to identify and locate specific degenerate forms. Preliminary surveys of degenerate forms were conducted using four source codes developed by diverse groups and in two different languages, C# and Java. During the four refactoring case studies, a list of 24 common degenerate forms was compiled. These degenerate forms include violations of commonly accepted principles of good software engineering, such as encapsulation and interface segregation. Others are related to common areas of concern such as unreachable code, unused variables, incorrectly modified class members, and poorly named variables. All of these were identified in production code and were unidentified by the compiler or other tools.

The author designed new metrics to detect these degenerate forms without a priori knowledge and in a quantifiable and objective way that approximates the classification of an expert human developer. Typically, two or three metrics work cooperatively to identify the degenerate forms. A subset of metrics that cooperatively exposed a large group of degenerate forms was selected for further development and experimentation. The results from the selected metrics are mainly quantifiable rather than merely indicative. With these metrics, it should be possible to track and target degenerate forms where they occur in wild code during refactoring and degeneration research.

The first metric is a novel approach to state. Several degenerate forms are associated with inconsistent state, overly complex state, and co-dependent state variables, but identifying state variables generally requires a human intelligence. The author has created a method that uses the statistical properties of code to determine which variables are likely state variables and which have no effect on the control structure of the program. The resulting conservative classification of state variables can be used to determine if one of these degenerate forms exists.

```
public class ClassB
{
   public void Use(int hr, int hnr, int pw)
   {
     int a = hr+pw;<--- found by pmd
     a++;
   }
}


public class ClassA
{
   private int neverUsed;//<--- found by PMD
   private int neverRead;//<--- found by PMD
   private int neverWritten;//<--- UNDETECTED
   private int hiddenRead;//<--- UNDETECTED
   private int hiddenNoRead;//<--- UNDETECTED
   private int postWrite;//<--- UNDETECTED
   private int notGood;<--- found by PMD

   public static void main(String [] args)
   {
     ClassA a = new ClassA();//<--- Properly Used
     a.neverRead = 2;
     System.out.println("NW is "+neverWritten);
     ClassB b = new ClassB();//<--- Properly Used
     b.Use(hiddenRead, hiddenNoRead, postWrite);
     postWrite = 4;
     notGood++;
     ClassA a2 = new ClassA();//<--- found by PMD
   }
}
```

Figure 1: Program demonstrating expanded definition of use

The second metric expands upon the definition of "use" in order to capture more completely the usage that can be attributed to the members of a class. Wagner et al [10] indicated that current tools such as PMD are highly accurate in identifying unused variables. Yet, in a simple experiment—shown in Figure 1—this expanded definition of "use" created by the author was used to detect 80 percent more unused variables than PMD. This expanded definition will identify variables and methods that have no semantic value in the program and "can be ignored while still producing optimal behavior." [5]

The third metric relies on a modified form of the module detection algorithm described by Blondel [1]. In the modified form, the algorithm is capable of identifying optimal module membership based on member-to-member access in the code. This allows the metric to identify probable package membership without any a priori knowledge or human interaction. The contrast between the suggested and current package structure may be used to identify inconsistencies in the design. The fact that this algorithm can be applied to class or package membership means it can also identify candidate encapsulation concerns.

The remaining metrics apply principles of graph theory to examine the source code and determine if other inconsistencies exist. Is some public variable, for example, accessed by other classes or should it be reclassified as private? Are two classes demonstrating feature envy by their strong affiliation? Are there any portions of the source code that are reachable but have no side effects? By using the three metrics above to inform this graphical analysis, it should be possible to identify several degenerate forms and bring them to the developer's attention for further analysis.

4.1 Expected Results

The selected metrics focus on principles of object-oriented software and were developed exclusively for detecting degenerate forms in OO. As such, they are of limited value as metrics, but they will make it possible to meet some basic proof of concept objectives that include:

- Evaluation in terms of false negative and positive rate,
- Validation in terms of diagnostic utility to developers,
- Development of simple automated repair functions,
- Demonstration of layers of degenerate forms, and
- Examination of abandoned software, long revision histories, and software currently under development.

In addition, the development of these metrics will allow for more extensive future experiments and tool development based on the principles learned. Refactoring may be performed in a more cost-effective manner. Development paradigms, tools, and programming practices can be examined for their efficiency in preventing the development of degenerate forms. New metrics may also be developed to identify degenerate forms that affect other programming paradigms including procedural and multi-core. As new metrics and degenerate forms are identified, the relationship between them can also be studied.

5. Evaluation Methodology

The primary outcome from this research will be the set of automated metrics and repair functions. These will be developed by testing on a large sample large-SLOC count, open source military, commercial and academic programs written in C# and Java. The sample will provide sufficient source code for baseline development as well as evaluation of the metrics. Once the metrics have been perfected, they will be individually assessed for correctness by contrasting the automated results with an expert classification by a jury. The jury will be given a random selection of classes from the source code sample and asked to determine specific characteristics related to the metrics and specific degenerate forms. The results from the jury will be used to quantify type I and type II errors for each metric.

In addition, the metrics will be tested on "live" code that is currently under development. The automated findings will be assessed by the software development team based on the degree of accuracy and the perceived value to the developers. The identification of hitherto unknown but correctly identified degenerate forms would rate high. Incorrectly identified or low-value items would rate low. These automated findings may also identify areas of degraded quality of which the developers are already aware. The results from this experiment will mirror [10] and [8] in demonstrating the value of the automated tools and the willingness of the developers to rely on those findings.

Secondarily, this research may afford future developers the ability to characterize degeneration as it occurs. Based on the work of [4], [3], and others, there are a number of causative and contributing factors that appear to lead to software degeneration. While a full study is infeasible, the validation of any of these factors would be a step in the right direction. Studies of these characteristics will assume the metrics, individually evaluated as described above, are correct and will rely on their findings to characterize the degeneration found in a sample source code. Thereafter, the selection of the source code for these contributing factors will help to prove correlation.

6. Conclusions

The proposed dissertation research will develop metrics capable of detecting degenerate forms in software. These metrics will be a first sample set to prove the value of such targeted metrics in diagnosing degenerate forms. With the addition of more metrics, it will be possible to more fully detect the extent and limits of degenerate forms. Improved detection may be used as an enabling technology to add visible, quantifiable, objective software quality metrics to development. As a result, it will be possible to:

- Overcome the sample size and other logistical concerns to enable more research on this topic,
- Conduct consistent quality assessment studies of any large-scale software project with minimal human effort,
- Identify causative or contributory factors that lead to greater risk of software degeneration, and
- Perform targeted refactoring of degenerate forms with limited knowledge of the software.

Acknowledgments

The author gratefully acknowledges the U.S. Navy for its support, the NSWCDD Software Developers' Lecture Series and Community of Practice, and his dissertation committee for its assistance. The author also lovingly acknowledges the patience of his wife and children.

References

[1] Blondel, V. D., Guillaume, J. L., Lambiotte, R., & Lefebvre, E. (2008). Fast unfolding of communities in large networks. *J. of Statistical Mechanics: Theory and Experiment*, *2008*(10), P10008.

[2] Bourquin, Fabrice, and Rudolf K. Keller. "High impact refactoring based on architecture violations." *Software Maintenance and Reengineering, 2007 CSMR'07. 11th European Conference* on. IEEE, 2007.

[3] Dvorak, J. (1994). Conceptual entropy and its effect on class hierarchies. *Computer*, *27*(6), 59–63.

[4] Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., & Mockus, A. (2001). Does code decay? assessing the evidence from change management data. *Software Engineering, IEEE Transactions on*, *27*(1), 1-12.

[5] Jong, N. K., & Stone, P. (2004). Towards learning to ignore irrelevant state variables. In *The AAAI-2004 Workshop on Learning and Planning in Markov Processes–Advances and Challenges*.

[6] Kemerer, C. F., & Slaughter, S. (1999). An empirical approach to studying software evolution. *Software Engineering, IEEE Transactions on*, *25*(4), 493–509.

[7] Lehman, M. M., Perry, D. E., & Ramil, J. F. (1998, November). Implications of evolution metrics on software maintenance. In *Software Maintenance, 1998. Proceedings, Inter. Conf. on* (pp. 208–217). IEEE.

[8] Murphy-Hill, E., Parnin, C., & Black, A. P. (2012). How we refactor, and how we know it. *Software Engineering, IEEE Transactions on*, *38*(1), 5–18.

[9] Van Gurp, J., & Bosch, J. (2002). Design erosion: problems and causes. *J. of Systems and Software*, *61*(2), 105–119.

[10] Wagner, S., Jürjens, J., Koller, C., & Trischberger, P. (2005). Comparing bug finding tools with reviews and tests. In *Testing of Communicating Systems* (pp. 40–55). Springer Berlin Heidelberg.