2011 18th Working Conference on Reverse Engineering

An Empirical Validation of the Benefits of Adhering to the Law of Demeter

Yi Guo, Michael Würsch, Emanuel Giger, and Harald C. Gall
s.e.a.l. – software evolution & architecture lab
University of Zürich, Department of Informatics
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
{guoyi,wuersch,giger,gall}@ifi.uzh.ch

***Abstract*—The Law of Demeter formulates the rule-of-thumb that modules in object-oriented program code should "only talk to their immediate friends". While it is said to foster information hiding for object-oriented software, solid empirical evidence confirming the positive effects of following the Law of Demeter is still lacking. In this paper, we conduct an empirical study to confirm that violating the Law of Demeter has a negative impact on software quality, in particular that it leads to more bugs. We implement an Eclipse plugin to calculate the amount of violations of both the strong and the weak form of the law in five Eclipse sub-projects. Then we discover the correlation between violations of the law and the bug-proneness and perform a logistic regression analysis of three sub-projects. We also combine the violations with other OO metrics to build up a model for predicting the bug-proneness for a given class. Empirical results show that violations of the Law of Demeter indeed highly correlate with the number of bugs and are early predictor of the software quality. Based on this evidence, we conclude that obeying the Law of Demeter is a straight-forward approach for developers to reduce the number of bugs in their software.**

I. Introduction

One of the crucial questions in software engineering is how to lower software maintenance costs, which can make up as much as 67% of the total development costs of a software system [1]. Maintainability of software strongly depends on the quality of its source code, *i.e.,* whether or not it is well-structured and easily understandable by its developers and maintainers. Several design principles and heuristics, have been proposed to guide software developers towards better code.

The most fundamental principles are to encapsulate data properly and to aim for low coupling and high cohesion for any given module. While the importance of these principles is beyond controversy, many different heuristics have been formulated that break down the difficult task of writing "good" code to a simple set of rules which developers can follow. One of those rules-of-thumb is the *Law of Demeter (LoD)*, formulated by Lieberherr and Holland in 1989 [2], [3], [4]. Its intent is to achieve loose coupling by limiting the knowledge that one unit of source code has about other units in the system. In particular units should, if at all, *"only talk to their immediate friends."*

There are few studies which focus on the relationship between the number of LoD violations and the number of bugs. Moreover, empirical evidence of the LoD benefits is still missing. To address this issue, we design a set of experiments to validate the consequence of violating LoD with regard to their significance to predict the occurrence of bugs on the class level. We then report on the correlations found and we show how effective LoD violations are for bug prediction compared to other predictors such as lines of code(LOC).

For the experiment, we formulate two hypotheses and design our empirical study to test them:

> H_1 – *The number of violations of the Law of Demeter correlate positively with the number of bugs*
> H_2 – *The number of violations of the Law of Demeter improves prediction models*

Five sub-projects are analyzed in our experiments. For each of them, we obtain the source code of each class for the most recent release and link them to bug reports. We then conduct a correlation analysis to empirically validate the relationship between the strong and the weak form of the LoD violations and the bug-proneness. We also investigate the impact of the LoD violations on bug proneness and compare it to the Chidamber & Kemerer metrics suite(C&K metrics) and LOC in three of sub-projects, which have more than 500 classes. Using both forms of the LoD violations with these metrics as independent variables(IVs), we then build prediction models of bug-proneness for the software systems under investigation.

II. The Law of Demeter

The Law of Demeter (LoD) [2] was introduced as a heuristic for object-oriented programmers to improve *information hiding*, *i.e.,* to encapsulate minimal information in modules and make as little use of public knowledge as possible [5]. The LoD states, "Only talk to your immediate friends," which implies that each unit of source code should only have limited knowledge about other units, and only about their so called *friends*. The LoD is an application of the *low coupling principle* by making the notion of bad coupling explicit and checkable by tools [4]. The LoD comes in two forms: The object form and the class form. Only the latter is designed to be able to statically enforced and checked by tools. Therefore,

1095-1350/11 $26.00 © 2011 IEEE
DOI 10.1109/WCRE.2011.36

we focus on the class form of LoD in our research. The class form of LoD is defined as follows:

- In the class form of the LoD, a method *M* of a class *C* can only send messages to the objects of the following classes:
  - instance variables of *C*,
  - the argument class(es) of *M*,
  - any classes of the instances created within *M*,
  - any direct properties/fields of *C*

The violation of the class form of LoD can be further categorized into the Weak LoD and the Strong LoD. We abbreviate violations of the Weak LoD as *WVLoD* and the ones of the Strong LoD as *SVLoD*. The Strong LoD excludes the inherited instance variables from the set of friends, whereas the weak LoD includes the inherited instance variables. Formally, in the weak form of LoD, a friend set C_f^w is a set of friend classes for the Class C

$$C_f^w = C_{this} \cup C_{fields} \cup C_{parameters} \cup C_{ov} \cup C_{iv} \quad (1)$$

whereas in the strong form of LoD, a friend set C_f^s is a set of friend classes for class C

$$C_f^s = C_{this} \cup C_{fields} \cup C_{parameters} \cup C_{ov} \quad (2)$$

where C_{ov} represents the own variables of C and C_{iv} represents the inherited variables of C.

Each category has some implications. On the one hand, adhering to the Strong LoD guarantees that any change to the underlying data structure will only affect methods declared by the classes that were changed, and methods of unchanged classes will not require modifications. On the other hand, if software development adheres to the Weak LoD, any change to the underlying data structure will affect the methods of the changed class, as well as any class which that is derived from them. Our hypotheses H_1 and H_2 are proposed based on

```
public void persistLineSeparatorPositions(){
   if (this.scanner.recordLineSeparator){
      this.compilationUnit
         .compilationResult
         .lineSeparatorPositions
         = this.scanner.getLineEnds(); //...(1)
   }
}
```

Listing 1: An example of LoD violation in JDT core

observations we made in production code, namely that there are method invocations in certain classes, which apparently violate the LoD and then later lead to bugs issued in the bug repositories. For example, Listing 1 shows a method declaration fragment in the JDT core source code. One of its statements (1) clearly violates the LoD. When developers change this code—for example when implementing a new feature or fixing a bug—they need to investigate three distinct objects to set the fields correctly: *compilationResult*, *compilationUnit*, and *lineSeparatorPositions*. This puts additional cognitive load on them because they cannot only focus on local changes but also have to investigate their potential impact in other source code locations. In fact, looking at the version history of the code given in this example, we found that there are changes that developers specifically made for fixing bugs Table I, most likely resulting from improper encapsulation. Based on this observation, we are curious to see whether we can find empirical evidence that too many statements violating the Law of Demeter in a class go hand in hand with an increased number of bugs occurring.

Bug 49908	Renaming of DefaultCodeFormatterConstants.java
Bug 49968	[formatter] Alignment API
Bug 48489	[DCR] AST support for a single expression (vs. CU)
Bug 49327	formatter can return null TextEdit when parsing valid java

TABLE I: Reported bugs related to the statements in Listing 1

	bug-prone	**Not bug-prone**
jdt.core	1067	114
pde	375	58
jface	331	41

TABLE III: Results of the classification of bug-proneness

III. Empirical Study

A. Data Collection

A summary about the sub-systems under analysis is given in Table II. The sub-projects we have chosen cover plug-ins of various size, ranging from 12k LOC to 290k LOC.

We use EVOLIZER [6] to obtain a complete, query-able history of the systems. In particular, we import the version history by retrieving the CVS logs from Eclipse CVS and by parsing all the CVS commit messages. Next we run EVOLIZER's bug-to-revision linking strategy and labels class changes with bug numbers: Commit messages including terms that indicate that the commit is a bug fix for the previous revision are labeled by an explicit bug id. Next, we calculated the bug-proneness for each class. Bug-proneness is the probability that a compilation unit in our study will be reported to link to an issued bug. For each compilation unit i, we conduct the linear transformation [0,1] to calculate the bug-proneness $P(x_i)$ as:

$$P(x_i) = \frac{x_i - min(X)}{max(X) - min(X)} \quad (3)$$

where x_i is the number of bugs in class i, X represents the set of the number of bugs in each project, $min(X)$ and $max(X)$ represent the minimum and the maximum of the number of bugs in each project respectively. Following the approach in [7], we define a statistical lower-confidence-bound on all bugs for each project and set it to 0.001. Classes with fewer bugs than the lower-confidence-bound are classified as not bug-prone, whereas other classes as bug-prone.

With regard to the computation of the violation metrics, we develop an Eclipse plug-in called *Violation Detector* to

Plugins	Last Release	#Files	LOC	#Bugs	#Violations	Max # of Violations
eclipse.jdt.core	May 11, 2010	1181	290,656	18834	35920	1882
eclipse.pde	May 07, 2010	519	50,690	2226	6499	363
eclipse.jface	May 18, 2010	381	43,202	3127	2519	144
eclipse.compare	June 15, 2009	154	20,574	1402	1712	495
eclipse.debug.core	June 20, 2010	188	12,511	1703	1476	255

TABLE II: Plugins collected from the Eclipse project

calculate the SLoD and WLoD. *Violation Detector* traverses the Abstract Syntax Tree for each compilation unit using the visitor pattern [8]. To detect violations, it investigates expressions including method invocations and field access, and validates whether the message will be sent to the objects which are instances of either C_f^w or C_f^s. If the class of a message target is not included in either C_f^w or C_f^s, then we count the expression carrying the message as an occurrence of violation of LoD. We aggregate the occurrence of violation expressions of the Weak LoD in WVLoD and the one of the Strong LoD in SVLoD. The results are then stored in EVOLIZER's RHDB for later retrieval in the empirical analysis step.

B. Empirical Analysis

We calculate Spearman's rank correlations between the number of bugs of each compilation unit and all of these measures (including the violations of LoD, Chidamber & Kemerer (C&K) OO metrics, and LOC), which are a set of theoretically-grounded metrics of OO softwares. These metrics are empirical validated and are well recognized in the OO research field. Table IV shows that both the SVLoD and WVLoD positively correlate with the number of bugs. In *jdt.core*, the ρ value of SVLoD (0.67) and the one of WVLoD (0.59) rank third and fifth among all measures respectively; whereas in *compare*, the ρ value of SVLoD (0.70) and the one of WVLoD (0.62) rank second only to LOC and fourth. It is interesting to observe that the ρ value of the SVLoD and the WVLoD do not differentiate themselves with some metrics in C&K metrics suite, such as CBO, WMC and RFC in all projects. The *p* value for corresponding ρ is 0.000098, which shows a significant positive correlation between violations and number of bugs. Therefore, we can accept our hypothesis H_1.

To explore the LoD violations' ability to predict the bug-proneness for each class, we perform the univariate logistic regression analysis for each metric (known as *Independent Variables(IVs)*) against the bug-proneness. Since logistic regression overestimates odds ratios and β coefficients in studies with small samples size (less than 500), we only consider three plug-ins whose sample size is more than, or close to 500. Table V shows the univariate logistic regression models for each metric.

The univariate logistic regression result can be considered first indicators to build prediction models for the bug-proneness. In order to build an optimal prediction model, we conduct a multivariate logistic regression model using significant metrics as *IV*s and then solve the multicollinearity problem using Principal Component Analysis (PCA) because it does not rely on any assumptions of distribution types. To further ease interpretation of the principle components, we show the PCA result as the table of rotated components that show a clearer pattern of loadings, where the variables either have a very low or high loading, thus showing either a negligible or a significant impact on the principle components [9]. Table VI shows the rotated components obtained from our selected metrics applied to the corresponding plug-in projects. We select orthogonal dimensions which capture over 98% of the variance in *jdt*, *pde*, and *jface*. Note that SVLoD exhibits

Metrics	**Coefficient(β)**	**Deviance**	σ_β
SVLoD	0.19	639.23	0.0369
WVLoD	0.42	671.24	0.1087
CBO	0.34	605.34	0.0481
NOC	0.02	749.51	0.0443
WMC	0.22	632.18	0.0291
RFC	0.03	663.67	0.0046
LOC	0.03	598.57	0.0040
DIT	0.31	726.62	0.0703
LCOM	0.02	697.20	0.0029
			dfe=1171

(a) jdt.core

Metrics	**Coefficient(β)**	**Deviance**	σ_β
SVLoD	0.08	315.41	0.0282
WVLoD	0.13	315.47	0.0439
CBO	0.11	327.73	0.0380
NOC	-0.02	340.93	0.0688
WMC	0.08	324.96	0.0266
RFC	0.01	340.35	0.0064
LOC	0.01	313.80	0.0035
DIT	-0.11	339.39	0.0817
LCOM	-0.00	341.04	0.0035
			dfe=423

(b) pde.core

Metrics	**Coefficient(β)**	**Deviance**	σ_β
SVLoD	1.22	208.18	0.4555
WVLoD	1.07	224.73	0.4362
CBO	0.63	227.55	0.1673
NOC	-0.09	255.75	0.0599
WMC	0.33	202.71	0.0733
RFC	0.12	211.77	0.0317
LOC	0.07	189.17	0.0194
DIT	0.62	242.52	0.1786
LCOM	0.03	224.65	0.0061
			dfe=362

(c) jface

TABLE V: Summary of univariate logistic regression

Plugins	SVLOD	WVLOD	CBO	NOC	WMC	RFC	LOC	DIT	LCOM
jdt.core	**0.67**	0.59	**0.71**	0.05	**0.66**	0.44	**0.76**	0.14	0.51
pde.core	**0.61**	0.61	0.56	-0.17	0.50	0.21	**0.64**	-0.08	0.15
jface	**0.73**	0.67	0.62	-0.04	**0.75**	0.65	**0.82**	0.40	0.65
debug.core	0.43	0.42	0.57	-0.20	0.58	0.54	**0.62**	0.30	0.55
compare	**0.70**	0.62	0.60	-0.21	0.65	0.62	**0.72**	0.39	0.59

TABLE IV: Results of Spearman's rank correlation (ρ)

	PC1	PC2
EigenValue:	4.4872	0.0470
CumPercent:	0.9760	0.9862
SVLoD:	0.1302	-0.3354
WVLoD:	0.0542	-0.2247
CBO:	0.0193	0.0342
WMC:	0.0360	0.1319
RFC:	0.0425	0.9009
LOC:	0.9881	0.0110
DIT:	-0.0002	0.0118
LCOM:	0.0155	0.0812
β	0.0148	0.0051
p	0.0000	0.0021

(a) jdt

	PC1	PC2	PC3
EigenValue:	2.3706	0.1539	0.0375
CumPercent:	0.9029	0.9615	0.9814
SVLoD:	0.2126	0.1407	0.5994
WVLoD:	0.1627	0.1211	0.5396
CBO:	0.0362	0.0081	-0.0248
WMC:	0.0575	-0.0616	-0.1419
RFC:	0.0577	-0.1956	-0.4708
LOC:	0.9556	0.0483	-0.2095
DIT:	-0.0001	-0.0132	-0.0222
LCOM:	0.0846	-0.9596	0.2506
β	0.0141	0.0084	0.0053
p	0.0003	0.0350	0.6454

(b) pde.core

	PC1	PC2	PC3
EigenValue:	3.3988	0.1648	0.1062
CumPercent:	0.9214	0.9661	0.9949
SVLoD:	0.2675	-0.0128	-0.0182
WVLoD:	0.0466	-0.0082	-0.0283
CBO:	0.0178	0.0065	0.0055
WMC:	0.0677	0.0514	0.0223
RFC:	0.1627	0.9538	-0.2452
LOC:	0.9740	-0.1885	-0.0717
DIT:	0.0026	0.0217	-0.0014
LCOM:	0.1146	0.2265	0.9660
β	0.0441	0.0214	-0.0077
p	0.0000	0.1275	0.1827

(c) jface

TABLE VI: PCA summary: Rotated components for violations and OO metrics

the second highest metric in PC1, which p value is lower than the 5% significance level. It provides us with the intuitive evidence that SVLoD could be a complementary indicator when building up bug-proneness prediction models. Therefore, H_2 could be accepted.

C. Prediction Quality

To measure the performance of prediction models, we calculate precision, recall and the area under the receiver operating characteristics curve(AUC) to provide a prior-independent approach for comparing the quality of prediction models. Table VII compares the PCA prediction results with the univariate prediction using solely LOC. We set the threshold equal to the prior probability of each project, because the bug-proneness class distributions of three projects are highly skewed. For example, in *jdt*, we have 1067 bug-prone classes and 114 ones that are not bug-prone. According to the confusion matrix shown in Table III, the prior probability is skewed as 90.34%. We get the precision and recall values of both prediction models on the threshold of the prior probability. The dominance of the ROC curve of PCA is reflected by a larger AUC. The AUC of *jdt.core* increases from 0.8311 to 0.8546; the AUC of *pde* increases from 0.6638 to 0.7152; and the AUC of *jface* increases from 0.8546 to 0.8590. As a result, H_2 can be accepted.

In a nutshell, according to the Table VII, the prediction quality improves when we build up the multivariate logistic regression predictors and conduct PCA. The key principle components, which cover more than 98% of variance, are composed of LOC, SVLoD, WVLoD, and some C&K OO metrics. Moreover, SVLoD weighs second only next to LOC in the first principle component. Therefore, it can be used to compose the first principle component as a predictor for the bug-proneness at the compilation unit level.

IV. Related Work

In 1996, Pal and Minsky [10] discussed how to impose the LoD on a system developed under Darwin-E. Lieberherr defined a stronger form of the LoD: the Law of Demeter for Concerns (LoDC). For that, he restricted the term "friends" further. Each communication should be restricted to those preferred supplier objects only that contribute to the current concerns among all concerns in play. They concluded that both Aspect-Oriented and Adaptive Programming would benefit from observing the LoDC to find a proper decomposition. In 2003, Lieberherr *et al.* defined a generic join point model for checking the LoD and illustrate how the joint point form is mapped to the object and class form of LoD [11].

A large number of research papers empirically investigated the relationship between design properties and external software quality. Basili *et al.* [12] demonstrated that several of Chidamber and Kemerer's Object-Oriented metrics appeared to be useful to predict class fault-proneness during the early phases of the life-cycle based on their empirical and quantitative analysis. In their dataset, Object-Oriented metrics are better predictors than "traditional" code metrics, which can

	Univariate Logistic LOC			PCA			Prior Probability (threshold)
	precision	recall	AUC	precision	recall	AUC	
jdt.core	0.9764	0.6317	0.8311	0.9825	0.6982	0.8546	0.9034
pde	0.9439	0.4933	0.6638	0.9468	0.6747	0.7152	0.8661
jface	0.9822	0.6677	0.8546	0.9912	0.6767	0.8590	0.8898

TABLE VII: Results of the univariate model and the PCA models for bug-proneness prediction

only be collected at a later phase of the software development processes. Gyimóthy *et al.* [13] analyzed *Mozilla* source code employing logical and linear regression and decision tree and neutral network methods to assess the applicability of the Chidamber and Kemerer's Object-Oriented metrics. Later, Zimmermann *et al.* [14] investigated the Eclipse bug data set and showed that the combination of complexity metrics can predict defects and suggested that more complex code it, the more defects it has. Lessmann *et al.* [15] compared different classification models for software defect prediction using AUC as benchmark, while Giger *et al.* [16] used similar analysis and performance evaluation criteria but, instead of focussing on failure-proneness, aim at providing models to predict the fix time of bugs. Recently Bird *et al.* [17] found evidence that there is a systematic bias in bug datasets. This may effect prediction models relying on such biased datasets. Therefore, they pursued two approaches to reduce the bias effect: Manually create links for unlinked bugs, or switch to commercial datasets that have nearly 100% linking to conduct Monte-Carlo simulations [17].

V. Conclusion and Future Work

In our study, we collected the source code and repository history of five well-known Eclipse projects to explore the relationship between the violation of the Law of Demeter (LoD) and the bug-proneness of classes and designed an empirical studying to explore the relationship between the violations and the bug-proneness in the class level. The empirical validation results shows that violations of LoD can be used as early indicators for software's bug-proneness. Moreover, our results show that they are substitute predictors for other coupling-concerned OO metrics in prediction models. We also noticed that the predictive quality improves when we combine violations of LoD with other metrics to generate principle components.

Our future research will continue the search for effective prediction models using violations of LoD as indicators. We plan to label LoD violations based on their severities, and then dig into the consequences of each label on the software quality perspective. Moreover, we will improve the violation detection algorithms to eliminate the bias generated by some harmless violations, which do not have much impact on the software quality.

References

[1] M. V. Zelkowitz, A. C. Shaw, and J. D. Gannon, *Principles of Software Engineering and Design*. Prentice Hall Professional Technical Reference, 1979.

[2] K. J. Lieberherr and I. M. Holland, "Assuring good style for object-oriented programs," *Computer*, vol. 22, no. 9, pp. 38–44, Sep. 1989.

[3] K. J. Lienberherr, "Formulations and benefits of the law of demeter," *SIGPLAN Not.*, vol. 24, pp. 67–78, March 1989. [Online]. Available: http://doi.acm.org/10.1145/66083.66089

[4] K. J. Lieberherr, "Controlling the complexity of software design," in *ICSE*. IEEE Computer Society, 2004, pp. 2–11. [Online]. Available: http://csdl.computer.org/comp/proceedings/icse/2004/2163/00/21630002abs.htm

[5] D. L. Parnas, "A technique for software module specification with examples," *CACM*, vol. 15, no. 5, pp. 330–336, May 1972.

[6] H. Gall, B. Fluri, and M. Pinzger, "Change analysis with evolizer and changedistiller," *IEEE Software*, vol. 26, no. 1, pp. 26–33, 2009. [Online]. Available: http://dx.doi.org/10.1109/MS.2009.6

[7] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 91–100. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595713

[8] R. J. Gamma Erich, Richard Helm and J. Vissides, *Design Patterns: Elments of Resuable Object-Oriented Software*. Addison-Wesley, 1995.

[9] L. C. Briand and J. Wüst, "Empirical studies of quality models in object-oriented systems," *Advances in Computers*, vol. 56, pp. 98–167, 2002.

[10] P. pratim Pal and N. H. Minsky, "Imposing the law of demeter and its variations," 1996.

[11] K. J. Lieberherr, D. H. Lorenz, and P. Wu, "A case for statically executable advice: checking the law of demeter with aspectJ," in *AOSD*, 2003, pp. 40–49. [Online]. Available: http://doi.acm.org/10.1145/643603.643608

[12] V. Basili, L. Briand, and W. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751–761, October 1996.

[13] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Trans. Software Eng*, vol. 31, no. 10, pp. 897–910, 2005. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TSE.2005.112

[14] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, Minneapolis, MN, May 2007, to appear.

[15] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans. Software Eng*, vol. 34, no. 4, pp. 485–496, 2008. [Online]. Available: http://dx.doi.org/10.1109/TSE.2008.35

[16] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, ser. RSSE '10. New York, NY, USA: ACM, 2010, pp. 52–56. [Online]. Available: http://doi.acm.org/10.1145/1808920.1808933

[17] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: bias in bug-fix datasets," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 121–130. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595716