Cancer in my Code?

Heuristic discovery of SOLID principle violations

Name Withheld per double blind review process
Affiliation Withheld
Location Withheld
Email@withheld.com

***Abstract*—The SOLID principles are based on coupling and cohesion—all metrics that seem to indicate good object-oriented design. Nevertheless, there are no standard calculations for these metrics and casual violation of the principles is defended daily by practitioners. This paper suggests a new metric for SOLID principle violations based on a cancer-growth model. These metrics are validated by manual code review and sample "tumors" are given for analysis and discussion.**

Keywords-coupling, cohesion, SOLID, metric, tool

I. Introduction

Coupling and cohesion have been standard metrics for object-oriented software almost since they were first introduced in 1974 [24]. Conceptually, they are fairly straight forward: things internal to a class should work closely together, and only loosely work with things external to the class boundary. Based on coupling and cohesion, the SOLID principles track a variety of specific concern related to class boundaries and inheritance relationships. More generally, OO design is littered with concepts which similarly relate to the sanctity of the class boundary: encapsulation, polymorphism, abstraction, etc. The importance of these concepts is recognized by practitioners and researchers alike.

While experienced programmers take these principles for granted, the novice is perpetually tempted to violate them. After all, does it really matter if a class has two responsibilities? Or a shared responsibility? Can't I just compose types rather than interfaces? I just want to copy that method from class A to class B. Is that so bad? While the experienced programmer may cringe inwardly even to hear such statements, there are still arguments about the validity of SRP [15], whether duplication [17] is really a problem, and contempt is shown for these principles whenever it suits the needs of the moment. Apparently, they are more what you'd call "guidelines" than actual rules.

A. *Metrics and Methods*

The differences between developers styles, the devout adherents of SOLID vs the YAGNI crowd, could be partially resolved if there was a recognized metric for cohesion and coupling. CodeGrep [21] finds a variety of reasons to flag a class for high coupling including too many imports, referencing a specific collection class, and referencing a type from a different package. While these rules of thumb may be useful, it is patently impossible to build a multi-package application without violating one of them. Sonarqube [13] uses a slightly more mathematical approach by counting the number of classes which reference or are referenced by a class. One seminal work in metrics [6] calculates coupling as the number of "methods of one class [which] use methods or instance variables of another class". By each of these methods the same class might be said to be tightly coupled, but more importantly the result may be completely different for each metric.

This approach to design metrics may be called the "beautiful numbers" method. If a design meets, exceeds, or is below some specific set of numerical thresholds it is good; anything else it bad. As noted previously, it is fraught with ambiguity and variability—two traits which hardly inspire confidence in developers. Moreover, this approach lacks a more complex view of the source code, which takes into account the relationships between classes, packages, class members, and ultimately every line of the code. Researchers [3] have shown that complex relationships may be noticed only when higher-order metrics are considered.

This complexity suggests the use of graph theory, an approach which has proved beneficial to various researchers. A review of coupling and cohesion over the evolution of FindBugs [23] was enabled by graphic tools and analysis metrics. In 2004 [4] and 2006 [5] link analysis in a graph was used to identify the god class antipattern. Examination of common graph metrics compared with program element relationships [14] reveals some correlation between these properties and data relationships. Clustering algorithms have also been used to discover components with high-affinity within a graph. Clusters have been used to identify classes for Extract Class refactoring [10] and to discover and document architectural elements [22]. Graphical complexity methods provide a much more appropriate view of higher-order software relationships.

B. *Cancerous Growth in Code*

Manual classification of code defects [7] [8] gives some evidence that defects within software grow, especially once a violation of the SOLID principles has already occurred. A conceptual model inspired by biological cancers may be used as a tool to aid in the classification of these patterns. The cancer model is consistent with [20], [19], [16] and [3] as well as numerous factors reported to contribute to a loss of software quality over time by [9]. It is also consistent with [12] which found that subsystems exhibited growth patterns not representative of the full application. If code contains cancer of this type and if cancerous elements grow, can they be detected? Can it be tracked? Can it be measured?

This paper describes a novel approach employing graphical analysis methods to detect potentially cancerous growths in various stages. Section 2 describes the metrics which were created and give examples of typical findings. Section 3 describes the experimental protocol used to evaluate the new metrics. Section 4 discusses the results of that evaluation including some examples of cancer detected in sample code. Section 5 draws conclusions and discusses the limitations of the approach and potential for future research.

II. Graph-Related Metrics

The first step in creating graph-related metrics is to establish a graph which represents the key elements of the program and their relationships. This was done using a proprietary tool based on a Java 7 ANTLR grammar that created a code graph showing structural elements (e.g., class A owns method B), inheritance (e.g. method C overrides the original implementation in method D), call dependencies (e.g., method E calls method F), and data dependencies (e.g. field G is written to by method H). This graph could then be used as the basis of all graph-related metrics.

Another early decision was made to ignore class and package boundaries. This seems counter-intuitive as the sanctity of class boundaries is at the heart of the SOLID principles. Classes and packages, however, represent the organization which the developer ***believes*** exists. In one sense, they are walls drawn in chalk on the floor and actual relationships between program elements may blithely ignore them. While the metrics will eventually depend greatly on developer-defined class boundaries, these boundaries are initially ignored.

A. Discovering Class Boundaries

At this point the metric framework consisted of a graph with no boundaries. A variety of methods may have been used to detect clusters of related program elements. For this research the Q algorithm produced by [2] was selected. This algorithm depends on the concept of a "module" containing closely related vertices within a graph. The algorithm used fine movements (i.e., one vertex at a time) to change the map of inter- and intra-module edges. By minimizing the weighted value of all inter-module edges (i.e., the Q value) the algorithm was able to detect modules of closely-related vertices without the aid of a human.

This algorithm had to be adapted in a number of ways. First, not all edges are equally weighted. Not only should weight counts indicate the strength of relationships (e.g., method A calls method B three times, but only calls method C once), but some relationships may be considered positively beneficial and thus should be excluded from the calculation. Consider, for example, the Interface Segregation Principle (ISP) which says that role interfaces should be used to completely separate types during composition. As shown Figure 1, members of each class have direct contact with the interface, but no direct contact with each other. This pattern of behavior should be recognized and rewarded by the Q algorithm. For similar reasons, accessor methods and the reuse of external libraries (e.g., java.util) are considered a positive pattern.

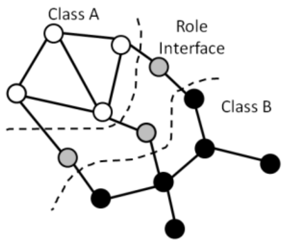

Figure 1. Interface segregation between classes

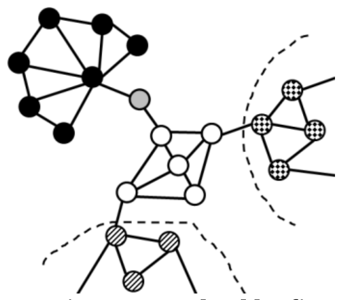

Figure 2. Formation not resolved by fine movements

In order to remove these edges from consideration in the Q algorithm, the weight of all interface, external, and accessor calls was reduced to zero. As a result, classes which were loosely coupled by an appropriate interface would be entirely separated in the graph. Natural cohesion between members on either side would cause the class members to coalesce into a module. This behavior was observed in early testing and the graph-related algorithm appeared to benefit from this approach.

In addition, the Q algorithm had to be adapted to allow gross movements (i.e., combination of whole modules). Fine movements within the graph sometimes created unstable conditions like that shown in Figure 2. Clearly the white module in the center is closely related and there are inter-module connections with the checked and striped modules at right and bottom. Consider, however, that the black module is only connected to the white module and by a single node without a positive preference for either side. Such patterns occurred frequently within single classes and created the false impression of two unrelated responsibilities. Gross movements (e.g., combining the black and white modules) allowed the inter-related modules to combine to the net benefit of the code graph.

Using the newly adapted Q algorithm it was possible to identify two new patterns created as artifacts of the process. Some modules were entirely separated from the rest of the code base because of the ISP or because of a lack of references in the rest of the graph. In a few cases, single nodes showed the same lack of connectivity. Both cases are shown in Figure 3. These cases were identified and separated from the rest of the modules. In cases where all the members of a single class were lone nodes or part of a well-separated module, the entire class would be classified as an API class. The most charitable interpretation is that an API class may serve a useful purpose by defining the interface for a web-service or client application. It may also simply be unused cruft.

At last the Q algorithm was able to generate meaningful modules. A short survey of the algorithm’s effect on various codebases showed that it was turning out results consistent with, if not identical to, the class boundaries defined by the developers. Most importantly, the entire process was automated and would not require a human in the loop. The next question was how well these modules correlated with classes as defined by the developer.

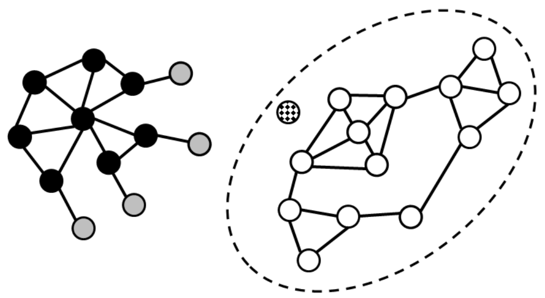
Figure 3. Well-separated sub-graph and lone node

Table 1. Perfect modules in sample codebases

Perfect	All Classes	Percent
45	111	41%
29	228	13%
150	374	40%
130	449	29%
180	552	33%
288	1014	28%
228	1213	19%
927	3480	27%
1141	4380	26%

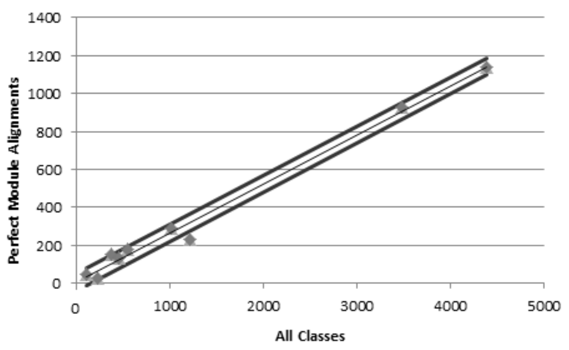

Figure 4. Linear regression of perfect modules

B. Integrity Metrics

Based on the results of the Q algorithm the graph now consisted of nodes with an assigned module and a known class. These boundaries were not always perfectly aligned, but in a large number of cases they were. As shown in Table 1, nine code bases were tested against the Q algorithm with varying results. Figure 4 shows the 95% confidence interval from linear regression analysis, predicting that 23.5-28.3% of all classes will perfectly align with modules. Therefore, while the majority of classes were misaligned to some degree or another, a significant portion of the Q algorithm results were identical with the human-defined class boundaries. This finding indicates that the algorithm correlates with human intuition at least some of the time.

In the remaining cases, the number of inconsistencies had to be considered in a numeric way. To this end, the integrity of each module and class was calculated based on the majority share of the component members. Consider an example where a class has members which belong to multiple modules. The size of each module within the class may be calculated and the module or modules with the largest share would define the most integrated portion of the module. Thus if the largest component contains 10 out of 25 class members the class is 40% integrated. Equations 1 and 2 give the calculations for class integrity. Module integrity would be calculated along similar lines.

$$\forall m \exists M \mid Module_m \leq Module_M \tag{1}$$

$$I_{Class} = \frac{|Module_M|}{\sum |Module_m|} \tag{2}$$

C. Diagnosing Cancer

Using the values for class and module integrity, new analysis can be performed to classify the minor fragments that are associated with a module or class. Of key interest is the degree to which one module has intruded on the class or module which it borders. In Figure 5, the black module is invading the class boundary of the highly integrated white class. This sort of phenomenon is a violation of the SOLID principles and might be recognized by an experienced developer as feature envy, inappropriate coupling, or a lack of cohesion. In keeping with the conceptual model of cancer, however, it has been termed stage I dysplasia–a minor intrusion of one tissue upon another.

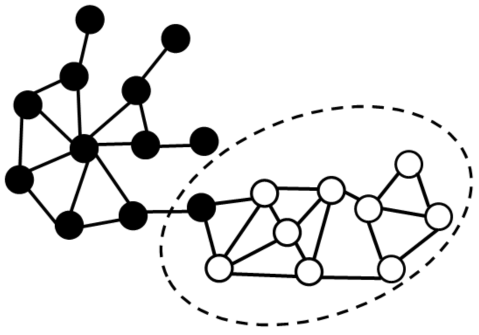
Figure 5. Stage I dysplasia

Once a beachhead has been established within another class or module, the intrusion will be increased. One of the examples cited in [7] and again in [8] showed that long association between two classes can lead to a comingling of functionality which prevents the classes from being easily separated. This condition is not as easily recognized. There are legitimate dependencies on the same program elements from multiple classes. Determining which class should have final ownership and how the classes will interact is a non-trivial refactoring problem.

Figure 6 shows the dilemma more clearly. If one observed the initial intrusion in Figure 5, it is apparent that the black module has continued its intrusion into the white class. There are, however, three vertices that are clearly tied to both sub-graphs and it is debatable whether or not some of the white vertices belong to one sub-graph or another. Therefore, a developer who happens to come upon this relationship, without the benefit of colored-module overlays, may not easily discern what has occurred and what the solution may be.

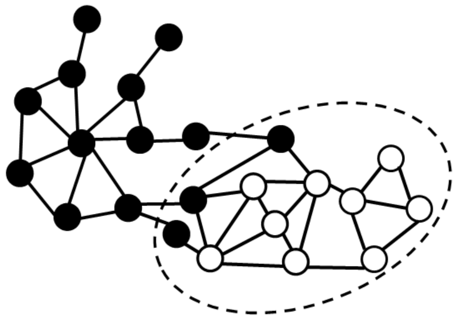
Figure 6. Stage II dysplasia

Finally, the cumulative information can be used to identify areas of very real concern. Consider that the Q algorithm has discovered modules of program elements which appear to be closely related. By identifying dysplasia as an intrusion of that module upon one or more classes, it is possible to form a boundary around the classes which form the core of the module and those members which have intruded into other classes. If the portion of the module identified as cancerous intrusions is above a given threshold (see Equation 3) it is a compelling argument that the module is in fact a tumor.

$$P \ll \frac{N_{Cancer}}{N_{Competent}} \qquad (3)$$

In Figure 7 a typical tumor is shown. In this diagram and with these pattern overlays it is apparent that the tumor is impinging upon the class borders on all sides. Note that the major portion of the tumor is self-contained. It may have begun with only a few members with some shared responsibility. On the borders, however, it has intruded into five other classes to one degree or another. This sort of behavior is consistent with [16] and [8] both of which refer to growth at the edges of existing patterns.

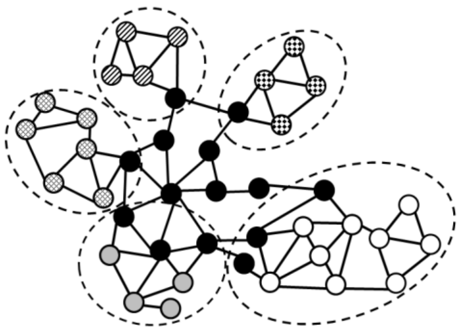
Figure 7. Tumor intruding upon various classes

III. Evaluation Methodology

Sample applications were taken from open source libraries and contributed by research partners at various commercial and public organizations. Six applications were used representing different developers, styles, processes, and application domains. All six applications were written in Java 7, a limitation enforced by the proprietary tool which created the initial graph. Each of applications was scanned for code cancer using the automated process. Reports of tumors and unaffiliated dysplasia (i.e., not affiliated with any of the identified tumors) were reported.

Experienced developers reviewed a statistical sample of the reports, grading the report on two criteria. First, the accuracy of the report was graded from 0 (no confidence) to 4 (complete confidence) with intermediate values indicating partial confidence in the report. Second, the utility of the report was graded from 0 (not actionable) to 4 (immediate action possible). Again, the developer could use intermediate values to indicate shades of utility.

The data from the expert evaluation is shown in tabular and graphic form (see Table 2 and Figure 8) as an ordered pair (accuracy, utility). The sample size varied between applications depending on the size of the report. In the tabular form, data from each application is shown separately, while the graphic form summarizes data from all applications. The size of the white bubbles indicates the relative frequency of each grade. A centroid is calculated for the cumulative result and shown as a black bubble on the graphs. Any apparent changes in the size of the centroid are not significant.

IV. Results

Naturally, false positives and false negatives must be considered. Because the cancer threshold is a variable and may be changed, the balance of type I and type II errors may also change. For this evaluation a P of .5 was used. False positives in the data set would be indicated by the ordered pair <0,x>. Because of the difficulty in identifying false negatives, they are not recorded, nor do they feature in this analysis. The observed false positive rate was 15% for Tumors and 16% for Dysplasia. These values are well within the 30% recommended by [1] and [18] as acceptable and even within the 20% recommended as stable.

Reports which have no actionable value (i.e. <x,0>) are as damaging to the utility of a tool as a false negative. These are equally useless to the developer and will likely result in the non-use of any tool. From the set of true positives, 96% of Tumors and 95% of Dysplasia were actionable. Overall 82% of reported Tumors and 80% of reported Dysplasia were accurate and actionable. Again, these values demonstrate that the tools are acceptable and stable in their assessment of object oriented design.

Table 2. Tabular view of expert evaluation data

Application	Tumor		Dysplasia	
#1	4	4	4	4
	0	0	0	0
	4	4	4	4
	4	4	4	4
	3	4	4	4
	4	4	4	4
	4	4	4	4
#2	1	4	0	0
	4	4	4	4
	4	4	4	4
	4	4	4	4
	4	4	4	4
	4	4		
#3	0	0	3	4
	0	4		
	3	4		
	3	4		
#4	4	4	1	0
	4	4	4	4
	4	4	1	4
	3	0	4	4
	4	4	0	0
	4	4	4	4
	1	4		
#5	4	4	0	0
	4	4	4	4
	4	4	4	4
#6	4	4	3	4
	4	4	4	4
	0	0	2	4
	0	0		
	4	4		
	4	4		
Centroid	3.090909	3.393939	2.96	3.2

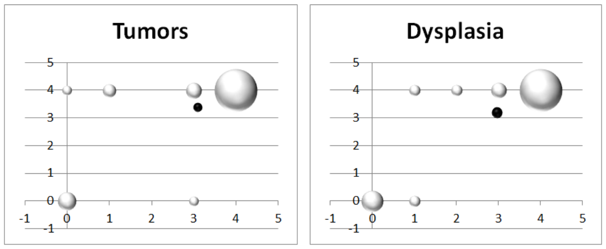

Figure 8. Graphic view of expert evaluation data

There are a few results which are notable for their eccentricity in the data set. First one tumor report was graded <0,4> which would seem to indicate that the finding was inaccurate but actionable. The developer's notes indicate that while the report was not considered to be a new tumor, the finding was definitely related to another report which was also accurate. Thus, both reports were actionable. Similarly, three tumor and dysplasia reports were graded <1,4>, again indicating a low level of confidence in the report, but actionable information. Here the notes reveal that while the developer did not consider the problem to be a serious concern, various forms of action from further investigation to minor fixes would be instituted.

These results are encouraging for two reasons. First, the low false positive rate and the high utility rate indicate that these metrics may be usefully employed to identify growing areas of SOLID principle violations. Second, the developer notes compare favorably with the patterns identified by [8], which were the original inspiration for this work. An analysis of the developer comments revealed a number of patterns which occurred repeatedly. Table 3 enumerates the list.

Table 3. Cancer patterns discovered

Pattern	Occurrences	
Introduce design pattern	3	6%
Remove poltergeist	4	9%
Introduce cross-cutting concern	6	13%
Repair/restructure hierarchy	8	17%
Extract and consolidate responsibility	26	55%

A. *Extract and Consolidate Responsibility*

Over half of the reports belonged to this category, where a responsibility had been distributed among multiple classes. The first step in correcting this pattern was always to extract the like-components using various refactoring actions (e.g., Move method, Extract class, etc.) to separate those program elements from unrelated functionality. The second step would consolidate all the related functionality into a single class (e.g., Inline class). The complexity of this process cannot be understated as it requires considerable revision of the existing design to normalize the "longstanding reciprocal relationships between classes [which] were brittle, strongly coupled, and deviously difficult to tease apart" [8].

Figure 9 shows two tumors which demonstrate the need for extraction. In all future figures the vertex for a program will be labeled with an ID indicating the class to which it belongs. Although two tumors were reported separately, the developer found that they were related. Class 14 repeated a simple pattern: a private field in layer A, an accessor method in layer C, and a hard-coded default value in layer B. In some cases, a class would read and store the setting value in a local private variable in layer D, somewhat defeating the purpose of centralized state. For Boolean settings two utility methods (gray vertices on right) were used to handle file I/O.

The six Boolean settings in the center of the image followed this pattern exactly. The four on the right did not. It is unclear if these differences are an artifact of design changes in the application's history or if the implementations were imperfect or incomplete. The setting on the far left, however, was reported as a separate tumor. These program elements were in the same class, but they shared no cohesive properties in common with the rest of the class. There were also functional differences (e.g., reading values from setting files, vs. lazy evaluation of the operating environment) and a distinct standards for identifiers in each group. Finally, while the user classes on the right accepted setting values without change, the user classes on the left universally aggregated additional information to adapt the "setting" for their own use. In no way are the two groups similar and the functionality on the left only exhibits coincidental cohesion [24] with the rest of the class.

Figure 10 shows a tumor which suggests the importance of consolidating like functionality into a single class. The program was a poorly implemented Model-View-Controller Pattern [11]. Notionally, data was held in the model with private internal (A) and external (B) components accessed through public methods. In keeping with a tabular data model, separate methods were used for row (C) and column (D) manipulation. Class 5 was predominantly the data model, with class 1 as the GUI component (i.e., View), and class 4 controlling the model. Contrary to the intended design, however, there were control elements in both class 1 and 5. In addition, the strong coupling between the three classes reduced the flexibility and effectiveness of the design pattern. Extensive refactoring would be required to consolidate and appropriately decouple the classes.

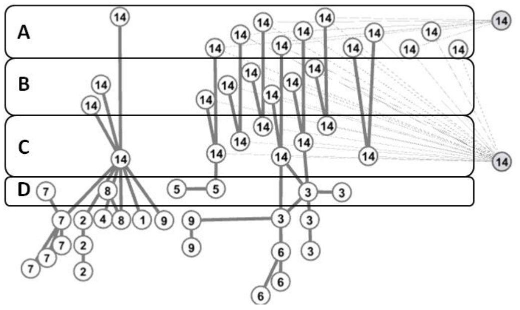

Figure 9. Sample tumors bisecting a class

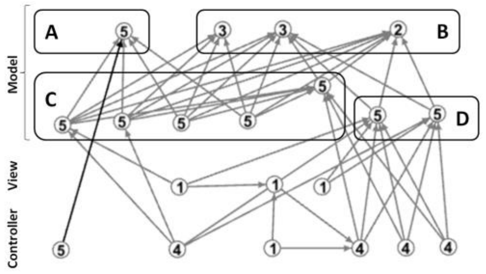

Figure 10. Sample tumor with mixed responsibilities

B. Repair/Restructure Hierarchy

A number of tumors demonstrated that the developers had created a hierarchy which was deficient in some way. This varied from multiple, unrelated classes which implemented identical functionality to abstract classes which were only implemented by a single child class. Each case demonstrated a lack of proper encapsulation and unnecessary duplication. The sample tumor in figure 11 shows four classes in a hierarchy with each horizontal layer representing one class. Class 4 was DefaultListModel from the Java library. Class 1 was an inexpert implementation which moved away from generics in favor of Objects. Classes 2 and 3 were type-specific models which implemented nearly identical functionality. Event listener classes 5 and 6 were private classes within each of the type-specific models.

As shown in Figure 11, four areas of key functionality were severely duplicated: removing an element (A), adding an element (B), construction (C) and searching (D). Note that most of the members implemented in class 1 were not used and when they were used, there was a mismatch with the parent class. For example, DefaultListModel implements three kinds of add functionality: add at next index, insert at index, add/overwrite at index. Class 1 implemented exactly the same methods, only one of which was used by the child classes, but all class 1 methods called completely different members in the super class. This mismatch created the false impression in classes 2 and 3 that they were adding at next index when in fact they were adding or overwriting any matching element first. This hierarchy shows a great deal of duplication, misdirection and the loss of generics—all good reasons for repairing the structure.

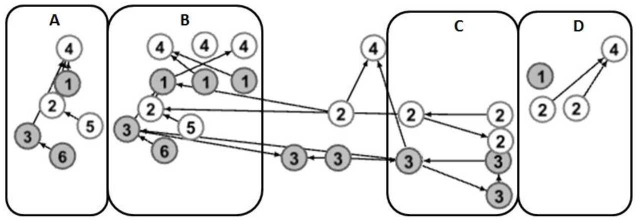

Figure 11. Sample tumor with confused hierarchy

C. Introduce Cross-cutting Concern

Cross-cutting concerns are single responsibilities which are utilized by a variety of classes. Sometimes separating the concern from the utility classes is a problem. Other cases involve normalizing the relationship across all user classes. Figure 12 shows a tumor which identified a cross cutting concern. An investigation of the report showed that 33 classes were dependent upon the cross-cutting concern for one reason or another. A spring framework was used to inject a single instance of class 2 as a private reference variable at layer B. The tumor report, however, showed that some classes were directly accessing functionality from class 2 from methods at layer A which were not dependent on the instance. Developers needed to review the entire paradigm and ensure that each class was correctly accessing the functionality of the cross-cutting concern.

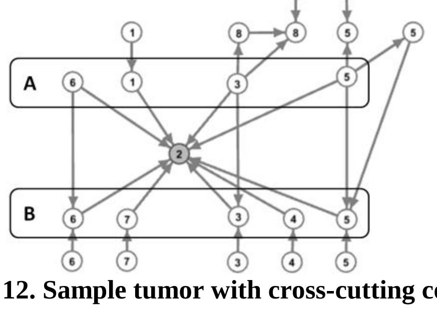

Figure 12. Sample tumor with cross-cutting concern

D. Remove Poltergeist Methods

A poltergeist class occurs when a reference in one class "may be pressed into service as a go-between" for two classes which lack a direct interface [8]. Figure 13 shows a sample where this problem was particularly evident. In this image, members of class 40 (A) are used by multiple classes to access members of class 15. Moreover, members of class 12 (B) are being pressed into a similar service by class 7. This is evidence of the "extreme version" of this pattern [8] where longer chains of poltergeists may develop. Although the initial tumor report only contained the four members in gray at the top of the image, the full scope of the poltergeist relationship was apparent to the developer upon further investigation.

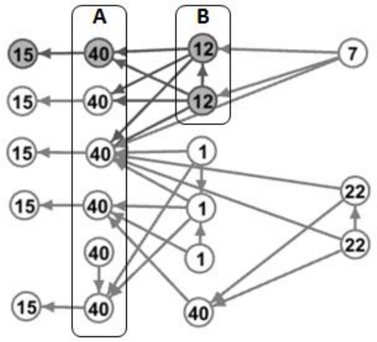

Figure 13. Sample tumor with poltergeist methods

V. Conclusions

This paper discussed the importance of coupling, cohesion, and SOLID principles in maintaining good object-oriented design as well as the challenges developers face in implementing these principles. Moreover, this paper used a cancer-growth model to predict how violations of these principles would occur. Using graph theory to examine relationships between program elements, metrics were developed which would identify violations predicted by the cancer-growth theory. Experienced developers then evaluated these metrics by validating reported tumors from six different software projects. Examples of the evaluated tumors were given and discussed.

The expert evaluation data suggests that these metrics are highly accurate (i.e., low false positive rate) and that the areas of concern predicted by the metrics are immediately actionable. Developer notes suggest that these metrics are useful in discovering relationships between class members which may have gone unnoticed by busy developers.

Based on the success of these metrics there is more support for the cancer-based growth model, but it is by no means proven. The experiment was limited to an evaluation of the metrics, not the underlying model. Furthermore, the metrics were based on a proprietary code graph generation tool which was limited to a Java 7 grammar. The generality of these results for all objected-oriented languages seems likely, but is not proven.

Future work in this area may be used to confirm the cancer-growth model. If the model is accurate then a software repository tracking the development of a longstanding software program may reveal the growth of such tumors over successive versions. These metrics, developed to detect areas of concern, may be used to catalog and track that growth. Similarly, any software project which was abandoned in favor of a new design might show the growth and spread of such tumors over the life, and subsequent death, of the program.

Acknowledgment

Full acknowledgement withheld for the purposes of blind review process.

References

[1] A. Bessey, K. Block, B. Chelf, A. Chour, B. Fulton, S. Hallem, A. Kamsky, S. McPeak and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, p. 66–75, February 2010.

[2] V. D. Blondel, J. L. Guillaume, R. Lambiotte and É. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, 9 October 2008.

[3] F. Bourquin and R. K. Keller, "High-impact Refactoring Based on Architecture Violations," in *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, Amsterdam, the Netherlands, 2007.

[4] A. Chatzigeorgiou, S. Xanthos and G. Stephanides, "Evaluating object-oriented designs with link analysis," *Proceedings of the 26th International Conference on Software Engineering*, IEEE Computer Society, 2004.

[5] A. Chatzigeorgiou, N. Tsantalis and G. Stephanides, "Application of graph theory to OO software engineering," *Proceedings of the 2006 international workshop on Workshop on interdisciplinary software engineering research*, ACM, 2006.

[6] S. Chidamber and C. Kemerer "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, June 1994.

[7] Anonymous Authors. Omitted per double blind reviewing.

[8] Anonymous Authors. Omitted per double blind reviewing.

[9] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron and A. Mockus, "Does Code Decay? Assessing the Evidence from Change Management Data," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, January 2001.

[10] M. Fokaefs, N. Tsantalis, E. Stroulia and A. Chatzigeorgiou, "Identification and application of extract class refactorings in object-oriented systems," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2241-2260, 2012.

[11] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design patterns: elements of reusable object-oriented software, Boston, MA, MA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[12] M. Godfrey and Q. Tu "Evolution in Open Source Software," *Intenational Conference on Software Maintenance*, 2000.

[13] "How sonar calculates afferent/efferent coupling," [Online]. Available: http://sonarqube-archive.15.x6.nabble.com/How-sonar-calculates-afferent-efferent-coupling-td5005556.html. [Accessed 29 March 2017].

[14] M. Ichii, M. Matsushita and K. Inoue, "An exploration of power-law in use-relation of java software systems," *19th Australian Conference on Software Engineering (ASWEC)*, IEEE, 2008.

[15] "I don't love the single responsibility principle," [Online]. Available: https://sklivvz.com/posts/i-dont-love-the-single-responsibility-principle. [Accessed 29 March 2017]

[16] C. Izurieta and J. M. Bieman, "A multiple case study of design pattern decay, grime, and rot in evolving software systems," *Software Quality Journal*, vol. 21, no. 2, pp. 289-323, 2013.

[17] C. Kapser and M. W. Godfrey, "'Cloning considered harmful' considered harmful," in *13th Working Conf. on Reverse Engineering*, Benevento, Italy, 2006, pp. 19-28.

[18] T. Kremenek, A. Y. NG and D. Engerl, "A factor graph model for software bug finding," in *Proceedings of the 20th international joint conference on artifical intelligence*, Hyderabad, India, 2007.

[19] M. Lehman, "Software Evolution Threat and Challenge," *Professoral and Jubilee Lecture*, Middlesex University, Oct 2003.

[20] D. L. Parnas, "Software aging," in *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, pp. 279-287, 1994.

[21] "Rules which find instances of high or inappropriate coupling between objects and packages," [Online]. Available: grepcode.com/file/repo1.maven.org/maven2/net.sourceforge.pmd/pmd/5.0.1/rulesets/java/coupling.xml. [Accessed 29 March 2017].

[22] G. Scanniello, A. D'Amico, C. D'Amico and T. D'Amico, "Using the kleinberg algorithm and vector space model for software system clustering," *18th International Conference on Program Comprehension (ICPC)*, IEEE, 2010.

[23] R. Silva and H. Costa, "Graphical and statistical analysis of the software evolution using coupling and cohesion metrics—An exploratory study," *Latin American Computing Conference (CLEI)*, IEEE, 2015.

[24] W. P. Stevens, G. J. Myers and L. L. Constantine, "Structured Design*," IBM Systems Journal*, vol. 13, no. 2, pp. 115-139, 1974.