2017 IEEE 25th International Conference on Program Comprehension (ICPC)

On the Properties of Design-relevant Classes for Design Anomaly Assessment

Liliane N. Vale*,† and Marcelo A. Maia†
*Computer Science Department - Federal University of Goiás, Catalão – GO – Brazil
†Faculty of Computing - Federal University of Uberlândia, Uberlândia – MG – Brazil
Email: *liliane.ufg@gmail.com, †marcelo.maia@ufu.br

***Abstract*—Several object-oriented systems have their respective designs documented by using only a few design-relevant classes, which we will refer to as key classes. In this paper, we automatically detect key classes, and investigate some of their properties, and evaluate their role for assessing design. We propose focusing on such classes to make design decisions during maintenance tasks as those classes of this type are, by definition, more relevant than non-key classes. First, we show that key classes are more prone to bad smells than non-key classes. Although, structural metrics of key classes tend to be, in general, higher than non-key classes, there are still a significant set of non-key classes with poor structural metrics, suggesting that prioritizing design anomaly assessment using key classes would likely to be more effective.**

***Keywords*-key classes; program comprehension; dynamic analysis; structural complexity.**

I. Introduction

The cost and effort needed to understand and adapt internal elements of software systems is related to the effort to investigate artifacts such as source documentation and code [3]. In well-designed systems, software artifacts are organized in such a way that, they can be understood and changed independently from each other. So, assessing design is an important task for improving software maintenance and evolution. However, in many cases, documentation of design decisions is missing, or when it exists, it is neither updated nor complete. In that case, developers are induced to analyze the source code, which is the only reliable artifact to assess the software design.

Instead of recovering architectural components, as traditional architecture reconstruction approaches, which are still difficult to apply and have low accuracy [4], in this work we propose automatically finding *design-relevant (key)* classes in object-oriented systems as an alternative to alleviate barriers related to documentation problems and code/architecture comprehension hurdles. Our motivation comes from the observation that several real-world systems such as Lucene[1], Tomcat[2] and Javac[3], use some few classes to document its architectural design. Key classes are those classes that implement concepts that the developer understands as the most important ones to explain the system design. The automatic finding of key classes was initially proposed by Zaidman and Demeyer [12], but there is still no concrete evidence that the awareness of them is a useful information for developers. So, we claim on the importance in investigating the role of key classes as a starting point for understanding and assessing software design.

In this paper, we propose to evaluate the properties of automatically identified key classes. Prioritizing the design assessment, using a small set of design-relevant classes, would possibly alleviate problems related to where to investigate and mitigate design anomalies. Then, structural properties of key classes are investigated to provide some evidence that these classes are better candidates to be prioritized in design anomaly assessment. The first studied property is the likelihood of key classes association to bad smells. First, we analyze the proneness of occurrence of bad smells in key classes compared to the rest of the classes. The second studied property is association of the occurrence of specific bad smells with different levels in cohesion and coupling metrics.

As a contribution of this paper, our results indicate that key classes manifest more often the presence of *Complex Class* code smell compared to non-key classes. Values for coupling and cohesion metrics are also significantly worse for key classes than for non-key classes, although that there are still non-key classes with those symptoms, suggesting that among those classes with design anomaly symptoms, the design-relevant classes would be more likely to impact design anomaly as a whole.

This paper is organized as follows. Section II shows an update of the automatic approach to identify key classes from [2]. Section III presents a study on the properties of key classes that would impact of the structural assessment of design. Related work is presented in Section IV, and finally, the last section presents the conclusion and future work.

II. Mining Design-relevant Classes

In this section, we depict the approach aimed at identifying a reduced set of key design-relevant classes that should be considered as relevant to explain software design.

A. Automated approach to mine design-relevant classes

The approach was adapted from [2] for ranking key classes which had already assessed precision and recall, presenting adequate results.

Ranking the key Classes based on Level-Analysis: This step has been adapted from [2], which has a cumulative

[1]https://lucene.apache.org/core/4_4_0/core/overview-summary.html
[2]https://tomcat.apache.org/tomcat-5.5-doc/architecture/overview.html
[3]http://openjdk.java.net/groups/compiler/doc/compilation-overview/

978-1-5386-0535-6/17 $31.00 © 2017 IEEE
DOI 10.1109/ICPC.2017.17

characteristic, i.e., for each analyzed level of subtrees, it increases the number of recovered roots (key classes). Ideally, it is expected that classes of interest should be located at levels near to the root, otherwise the number of irrelevant classes increases considerably whenever the analysis descends. To alleviate this problem, we propose a strategy that ranks recovered candidate keys classes. Suppose for example, that there are three candidate subtrees, and the developer wants to retrieve only one key class ($kc = 1$). To determine which root of the subtree should be selected as the key class, we used a ranking algorithm to determine an order of relevance of the discriminant attributes in trees. In this context, the Ranker method available in Weka[4] software classified attributes assigning weights. So, the weights were: size of subtrees=1; number of distinct methods=0.24; number of distinct classes=0.22; number of distinct packages=0.15 and distance of the subtree to the main root=0.02.

Algorithm 1 is the pseudocode the process that combine the above steps. The function *extractkeyClasses* evaluates the trace subtrees classified by *naiveBayesClassifier* to find roots or subroots candidates for key classes. For each subtree, the value for each attribute is calculated. The subtree with the highest value has its root extracted to define a key class. Supposedly those key classes should be at the highest levels of the tree. If these nodes are at the highest levels of the tree, those nodes have strong control over the application, because all other method calls will be controlled by this root having a higher chance of being a key class. From this subtree new subtrees will be extracted through the expansion method *NISubtreeExtractor* described previously [2].

Algorithm 1: Process to select the key classes from the key subtrees.

```
   Input:
   A set of trace subtrees subtreeList;
   The target number of key classes k;
1  init
2      SB ← naiveBayesClassifier(subtreeList);
3      extractkeyClasses(SB, k);
4      return SB;
5  function extractkeyClasses(SB, k)
6      depthSubtree ← 1;
7      higherWeight ← 0;
8      auxHigherWeight ← 0;
9      tempSubtree;
10     while listRoots.size() < k do
11         foreach SB_i in SB do
12             auxHigherWeight ← 1*sizeOfSubtree + 0,24*numberOfDistinctMethods +
               0,22*numberOfDistinctClasses + 0,15*numberOfDistinctPackages + 0,02*DistanceOfRoot;
13             if (auxHigherWeight > higherWeight) then
14                 higherWeight ← auxHigherWeigh;
15                 tempSubtree ← SB_i;
16         listRoots.add(extractRoot(tempSubtree, depthSubtree));
17         SB.add(NISubtreeExtractor(SB, tempSubtree, depthSubtree, 0));
18         depthSubtree← depthSubtree+1;
19         higherWeight ← 0;
20         if (listRoots.size() == k ) then
21             break;
```

B. Design-relevant classes mined from the subject systems

This study investigates key classes extracted from four software systems, namely Apache PDFBox[5] an open source Java application, a school software, financial software and a service order, we omitted their real names because they are proprietaries applications of a software development company (Table I). The choice of the systems for analyzing was driven by distinct size, design and application domain, developers' interest in collaborating, and absence of documentation.

Table I: Characteristics of the systems under analysis.

System	# Scenarios	LOC	Packages	Classes
PDFBox	8	116464	110	1166
Financial	9	36702	21	130
School	11	59427	40	424
Service order	14	558534	183	3361

The automated approach extracted key classes from last version of the target systems as shown on the Table II. To *financial software, school software* and *service order software* we asked to developers the number of key classes *kc* to recover and they indicated ten key classes, due to the size of the systems and the standard amount of key classes recovered in traditional systems such as Javac, Lucene and Tomcat. For Apache PDFBox we have contacted the developers only to evaluate the key classes' quality, so we were guided by number of features available on the original documentation and then, we defined to be 13 key classes.

Table II: List of Mined Classes

School	Financial	Service Order	PDFBox
EnrollmentRec	AccountMgt	ServiceOrderMgt	PDFParser
AttendanceFinalRec	PaidBillReport	MoveViewIdMov	FontFileFinder
EarlyLeaving	CheckMgt	ProductGroupIFMgt	PDDocument
CourseMatrixMgt	BillMgt	WinOS	PDAnnotationTextMarkup
AttendanceListMgt	FinanceToolbar	WSMovMoveItems	PDPageContentStream
LateArriving	FinancialMain	PGCFactory	PDFontDescriptor
SchoolMain	CashFlowMgt	WSCompany	COSWriter
Timetabling	ReceiptMgt	IdentifierAccountMgt	COSDocument
AttendanceView	MontlyReceiptReport	PartnerMgt	TrueTypeFont
DiaryAttendance	ReceiptViewTableRenderer	ParameterSys	PDGraphicsState
			PDPage
			PDFTextStripper
			FontFormat

Table II shows the list of mined classes. To get recall and precision, developers classified each key class using a Likert scale (from -2: *Strongly disagree* to 2: *Strongly agree*) specifying their level of agreement on a class to be key or non-key. To calculate recall and precision, a class is considered key class if it is classified as *Strongly agree* or *agree* and has Weighted average ≥ 1. The PDFBox developer indicated more five potential key classes such as *PDFStreamEngine, PDFont, PDFontLike, COSBase* and *PDStream*. For the proprietary systems, developers did not mention missing key classes, but because we agreed to find 10 key classes, we calculate recall and precision considering this number. Thus, the recall values were: school=90%, financial=80%, service order=70% and PDFBOX=64%. The precision values were: school=90%, financial=80%, service order=70% and PDFBOX=69%.

III. Structural assessment

As key classes should be intrinsically related to design, we investigate if there is any association to the occurrence of bad smells [1]. Moreover, to understand how key classes may impact design quality, we investigate if classical indicators for assessing modularity (coupling and cohesion) have different levels in key classes compared to non-key classes. We also

[4]http://www.cs.waikato.ac.nz/ml/weka/documentation.html
[5]https://pdfbox.apache.org/

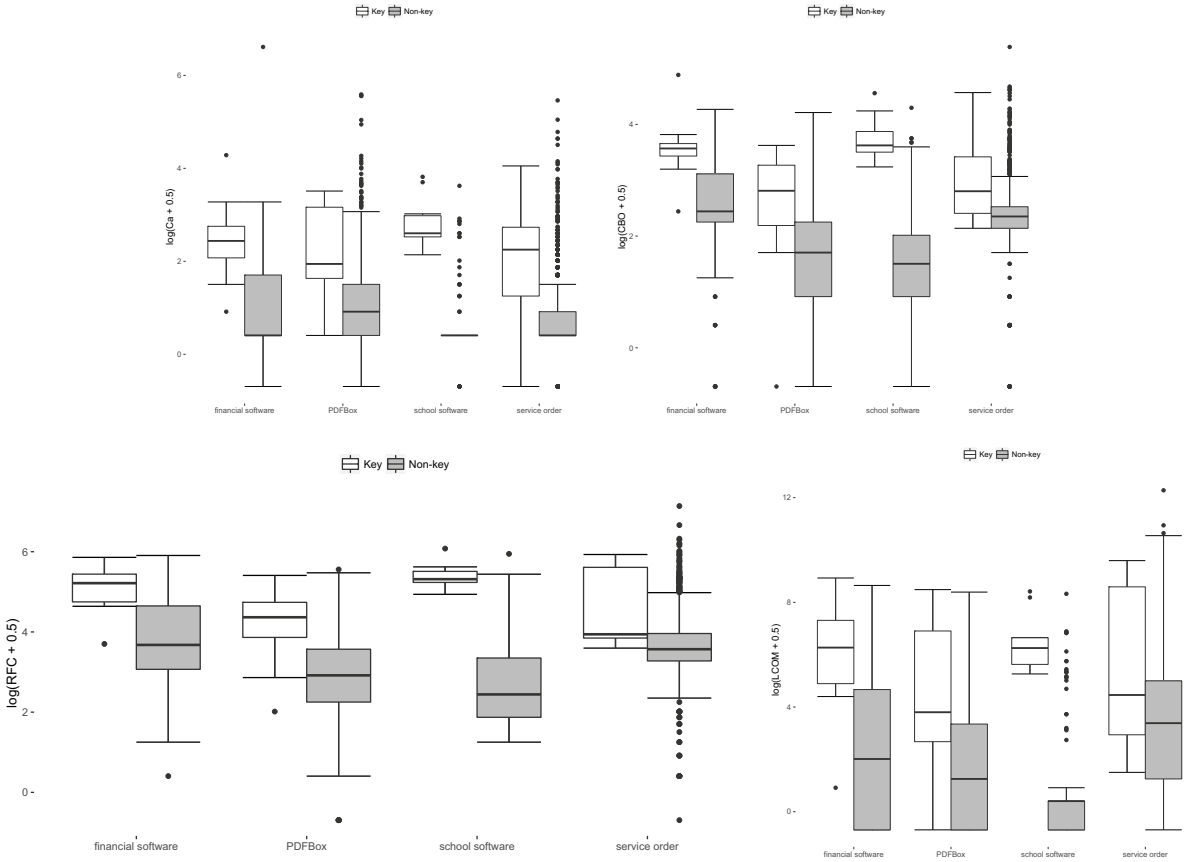

Figure 1: Metrics CA, CBO, RFC and LCOM in key classes and non-key classes

investigate the interplay between these indicators and the occurrence of smells. For that, we pose the following questions.

RQ$_1$: *Are key classes more prone to the occurrence of specific bad smells compared to non-key classes?* This question aims at investigating whether key classes are more prone to bad smells and which kind of bad smells are more common in key classes. We use DECOR [7] because as it is considered a state-of-the-art tool for detecting smells [11]. The answer to that question is based on the analysis of the the relative frequency of the several kinds of smell in key classes *(kc)* compared to non-key classes *(nkc)* of the system, shown in Table III. The *ComplexClass, LongMethod* and *LongParameterList* smell kinds are the most frequent. *Long Method* are more prevalent in key classes across all systems. The *Long Parameter List* smell is not very prevalent considering the universe of methods. However, except for the *Service Order* system, key classes present a slight higher prevalence of this kind of smell, possibly indicating that although key classes seems to be more complex their methods do not suffer much from being long.

RQ$_2$: *Are key classes different in terms of cohesion and coupling metrics compared to non-key classes?* This question aims at investigating usual indicators concerning the quality of software projects, namely cohesion and coupling. We evaluated four cohesion and coupling metrics comparing those metrics within two different groups: key and non-key classes. The COPE (*Component Adaptation Environment*)[5] tool was used

Table III: Occurrence of smell in key classes (kc) and non-key classes (nkc)

	# occurrence		% occurrence	
	nkc	kc	%nkc	%kc
School				
LongMethod	6/4397	4/616	0.001	0.006
LongParameterList	5/4397	1/616	0.001	0.001
LazyClass	2/414	1/10	0.004	0.1
SpaghettiCode	6/414	1/10	0.014	0.1
AntiSingleton	0	1/10	0	0.1
ClassDataShouldBePrivate	2/414	1/10	0.004	0.1
ComplexClass	4/414	2/10	0.009	0.2
Financial				
LongMethod	24/2008	4/10	0.012	0.4
LongParameterList	12/2008	3/10	0.005	0.3
LazyClass	13/120	1/10	0.108	0.1
ClassDataShouldBePrivate	2/120	1/10	0.017	0.1
ComplexClass	19/120	3/10	0.16	0.3
Service Order				
LongMethod	435/49186	3/10	0.009	0.3
LongParameterList	220/49186	4/10	0.004	0.4
AntiSingleton	64/3351	1/10	0.019	0.1
ClassDataShouldBePrivate	64/3351	1/10	0.019	0.1
ComplexClass	242/3351	3/10	0.072	0.3
PDFBox				
LongMethod	248/8451	3/13	0.029	0.230
LongParameterList	80/8451	2/13	0.009	0.153
ComplexClass	203/1166	10/13	0.174	0.769

to extract the metrics *Ca* (Afferent couplings), *LCOM* (Lack of cohesion in methods), *RFC* (Response for a Class) and *CBO* (Coupling between object classes).

Figure 1 shows the distribution of the values of the cohesion and coupling metrics. We conducted the Mann–Whitney–Wilcoxon test on all metrics and the results confirm significant differences between key classes and non-key classes ($p < 0.05$). However, there is an interesting point to observe which is although the medians are significantly different, we can observe that those metrics are not able to precisely define which classes should be considered key classes because there is a significant number of non-key classes (generally the 25% upper values) that are mostly coincident with the values for key classes. In other words, we observe that key classes are in general more prone to worse metrics, however the inverse is not in general true, i.e., a reasonable part of non-key classes are also prone to worse metrics.

RQ$_3$: *What kind of relationship can be found between cohesion/coupling metrics and bad smells?* We want to understand what kind of bad smells are mainly related to cohesion and coupling. An hypothesis is that a bad smell *large and complex* key class is associated with worse cohesion and coupling. Thus, the complexity associated with these key classes can be related to the fact that classes are more involved in specific smells.

Long methods and complex class are related to lower cohesion in classes. Moreover, long parameter lists and complex classes are also related to high coupling. So, addressing those bad smells seems to be a natural way to improve these modularity indicators. Moreover, key classes seems to have higher priority due to its higher impact on the overall design.

Main Findings: Key classes have proportionally more *Complex class, Long Methods and Long Parameter List* smells compared to non-key classes. Also, median values for coupling and cohesion metrics for key classes are significantly worse than for non-key classes. However, there is a significant number of non-key classes with bad smells and poor metrics, so we suggest that prioritizing design assessment based on key classes analysis instead of based on ranked lists of poor-metric classes provides a more focused way to find more relevant design anomalies supported by the design nature of key classes.

IV. Related Work

Key classes presented important properties in a previous work, most closely related to ours [12] which showed that such classes have a strong control over the application. Moreover, these key classes are prone to bad smells too. In fact, knowing which bad smells point to important design problems would help to focus developers' efforts [6], [10]. Another approach [8] showed that architectural problems are much more often reflected as anomaly agglomerations rather than individual anomalies in the source code [9].

V. Conclusion

In this paper, four systems were mined to retrieve a reduced number of key classes to study if the structural properties of those classes are an indicative that these classes are more adequate to be prioritized during design anomaly assessment. The presence of specific bad smells in key classes and the relationship with the metrics of cohesion and coupling were investigated. Our results suggest that that developers should prioritize key classes when assessing design because: first, key classes have more *Complex class* and *LongMethod* smells compared to non-key classes, and second, using conventional structural metrics to prioritize assessment would indicate several non-key classes with poor metrics. So, by definition, prioritizing design-relevant classes with bad symptoms would be more effective to find more relevant design anomalies.

As future work, we will verify if non-key classes are actual non-key classes to check for true negatives with developers. Another point is related to the size of the systems we will investigate if there is any correlation between the performance of their approach and the size of the system analyzed in terms of recall and precision. Finally, key classes could be used to prioritize other activities in software maintenance and evolution such as bug location.

Acknowledgment

We acknowledge the Brazilian agencies FAPEG, FAPEMIG, CAPES and CNPq for partially funding this research.

References

[1] W. J. Brown, R. C. Malveau, H. W. McCormick, III, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[2] L. do Nascimento Vale and M. A. Maia. Keecle: Mining key architecturally relevant classes using dynamic analysis. In *Proc. ICSME*, pages 566–570. IEEE, 2015.

[3] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Trans. Software Eng.*, 35(4):573–591, 2009.

[4] J. Garcia, I. Ivkovic, and N. Medvidovic. A comparative analysis of software architecture recovery techniques. In *ASE*, pages 486–496. IEEE, 2013.

[5] G. Kakarontzas, E. Constantinou, A. Ampatzoglou, and I. Stamelos. Layer assessment of object-oriented software. *J. Syst. Softw.*, 86(2):349–366, Feb. 2013.

[6] A. Lozano, M. Wermelinger, and B. Nuseibeh. Assessing the impact of bad smells using historical information. In *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*, IWPSE '07, pages 31–34, 2007.

[7] N. Moha, Y. gaël Guéhéneuc, L. Duchien, and A. françoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1), 2010.

[8] W. N. Oizumi, A. F. Garcia, T. E. Colanzi, M. Ferreira, and A. v. Staa. When code-anomaly agglomerations represent architectural problems? an exploratory study. In *2014 Brazilian Symposium on Software Engineering*, pages 91–100, Sept 2014.

[9] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, and D. Poshyvanyk. Detecting bad smells in source code using change history information. In *Proc. of the 28th IEEE/ACM Int. Conf. on Automated Software Engineering, ASE'2013, 2013*, pages 268–278, 2013.

[10] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. D. Lucia. Mining version histories for detecting code smells. *IEEE Trans. Software Eng.*, 41(5):462–489, 2015.

[11] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. When and why your code starts to smell bad. In *Proc. of the 37th Intl. Conf. on Software Engineering (ICSE'2015)*, pages 403–414, 2015.

[12] A. Zaidman and S. Demeyer. Automatic identification of key classes in a software system using webmining techniques. *J. Softw. Maint. Evol.*, 20(6):387–417, Nov. 2008.