Towards an Approach to Prevent Long Methods Based on Architecture-Sensitive Recommendations

Marcos Dósea[1, 2], Cláudio Sant'Anna[1], Cleverton Santos[2]

[1] Department of Computer Science
Federal University of Bahia – Salvador, BA – Brasil

[2] Department of Information Systems
Federal University of Sergipe – Itabaiana, SE – Brasil

dosea@ufs.br, santanna@dcc.ufba.br, clevertonmaggot@gmail.com

Abstract. *Long methods can be a software design erosion symptom. Existing approaches to identify this code smell are mainly based on the use of the number of lines of code metric together with a generic threshold value. However, using generic threshold values generates many false positives and false negatives because the class architectural concern is disregarded. This work presents a new approach that considers architectural concerns to identify threshold values and recommend long methods to developers. We evaluate the proposal with nine versions of MobileMedia system and the initial results show higher accuracy when compared with existing approaches.*

1. Introduction

The gradual increase of long methods in source code accelerates software design erosion. Design erosion is inevitable because of the way software is developed. However, good development methods help to increment system longevity [van Gurp & Bosch 2002]. According to Fontana et al. (2013), long methods are among the most common code fault in different application domains.

Code review is a common practice to maintain the quality of source code. Recently, automatic metrics-based approaches have been proposed to help this process [Marinescu 2004; Arcoverde et al. 2012; Palomba et al. 2013]. They define generic threshold values before code analysis for each used metric. A code smell is identified when established threshold values are exceeded. For instance, a long method is identified when the number of lines of code per method (SLOC/Method) exceeds a predetermined threshold value.

However, generic threshold values disregard system design decisions and the architecture concern of each class. The class architectural concern defines the main responsibility of the class within the evaluated code design. For example, in a system that follows the layer architectural style, are there differences in the average of SLOC/Method of classes in the persistence layer and classes in the business layer? If this occurs, generic thresholds may hide design problems (false negatives) or detect unimportant problems (false positives). Zhang et al. (2013) show evidences that information about application context, such as application domain, programming language and age should be considered in the metrics utilization. When this context is not considered, a large number of false positive and false negative can be sent to software developers. The excessive amount of warnings may lead to mistrust and lack of motivation to use automatic review approaches.

Furthermore, common wisdom suggests that software aging [Parnas 1994] and novice developers are the main cause of design erosion. However, recent studies show that code smells affect code artifacts since their creation [Tufano et al. 2015]. Expert developers are also responsible for introducing code anomalies in source code produced under the pressure of organizational factors [Lavallee & Robillard 2015]. However, the major approaches to check code anomalies are executed after the source code development. Postponing the repair of anomalies to final of development phase can lead to lack of motivation and time to fix software problems.

In this context, this paper presents a new approach for detecting and recommending long methods for software developers. Our approach allows to identify and to define specific thresholds for each architectural concern in a system. The threshold values of these architectural concerns are extracted from a design reference system that follows the same reference architecture [Angelov et al. 2009]. The reference system may be an earlier version of the same system or another system which follows the same reference architecture.

This approach differs from the existing proposals which suggest threshold values extracted from a set of systems [Arcelli Fontana et al. 2015; Vale & Figueiredo 2015]. However, the use of a set of different systems disregards design decisions of the system under evaluation, often requiring manual calibration of the values found. Furthermore, the quality of the systems could not be properly evaluated. Our approach uses only one system as a sample which can easily be reviewed by a quality team.

The remainder of this paper is organized as follows. Section 2 presents related work. Section 3 describes the proposed method to identify long method using class architectural context. Section 4 presents the experiment setup. Section 5 discusses threats to validity of the evaluation. Section 6 shows the results and discusses the research questions. Finally, Section 7 presents conclusions and insights for future work.

2. Related Works

A variety of approaches have been proposed to avoid design erosion through metric-based code smells detection strategies. Some studies discuss the importance of considering context in the assessment of source code. Marinescu (2006) shows how detection accuracy of Data Class and Feature Envy smells can be improved by taking into account particularities of enterprise applications. The study revealed that considering architecture roles has a big impact on eliminating false positives from classical detection rules.

Guo et al. (2010) claim that code smells detection rules should also take domain-specific characteristics into consideration. Setting thresholds is critical to an effective approach for detecting domain-specific code smells. Our work distinguishes from theirs in the way that we aim to propose a generic method to identify architectural concerns. Our approach does not require from the developer prior knowledge of the design decisions. The design decisions are automatically extracted from a reference system.

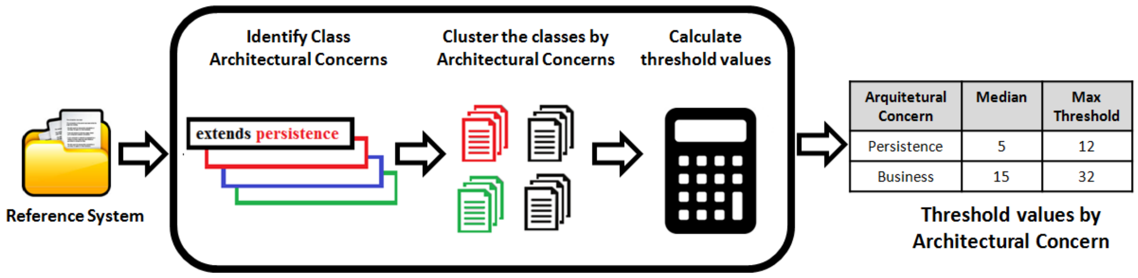
Macia et al. (2013) have explored the binding of the architecture structure with code smells. They propose a suite of detection strategies that use architecture-sensitive metrics. Their evaluation indicated on average 50% more architecturally-relevant code anomalies than those gathered with the use of conventional strategies. However, their approach uses threshold values extracted from distinct systems. Our approach differs because we intended to use threshold values extracted from a system that follows the same reference architecture of the evaluated system.

3. The Proposed Approach

The proposed approach aims to identify long methods according to the architectural concern of each evaluated class. The class architectural concern identifies the main architectural responsibility of each class. On systems that adopt reference architectures, class responsibility is commonly assigned by class inheritance, class annotations or interfaces implementation of the reference architecture.

Our approach uses this knowledge to find out threshold values for each architectural concern. These values are extracted from a reference system that must follow the same reference architecture and design rules. The goal is to extract knowledge from a system that has the same design decisions and can be easily reviewed by a quality team to become a benchmark of code quality. We strongly recommend that this review is always carried out in the version used as code reference design. If such review is not performed, the approach can recommend code anomalies based on a code with inadequate design decisions.

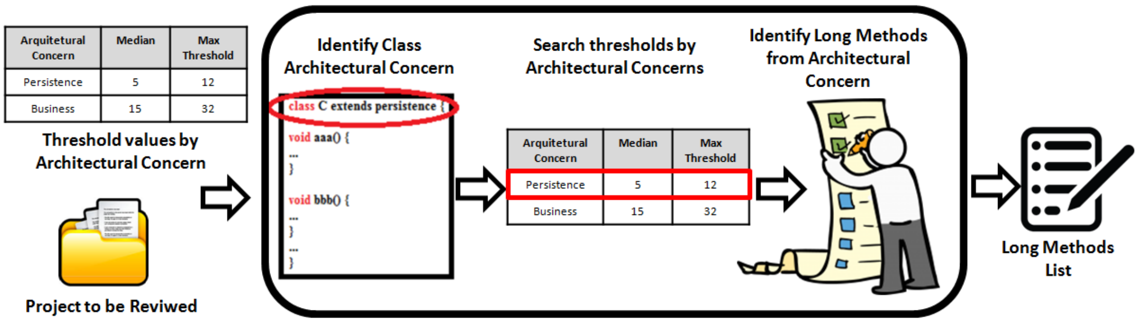
The approach is divided into two phases. The first phase calculates thresholds for identified architectural concerns in the reference system (Figure 1). In the second phase (Figure 2), each class to be evaluated is classified into an architectural concern. This classification will select the threshold values, identified in the first phase, which will be used to evaluate the methods of this class.


Reference System
Identify Class Architectural Concerns
extends persistence
Cluster the classes by Architectural Concerns
Calculate threshold values
Threshold values by Architectural Concern

Arquitetural Concern	Median	Max Threshold
Persistence	5	12
Business	15	32

Figure 1. Defining threshold values from a reference system.

Figure 1 illustrates the first process phase. The approach is divided into three steps: (i) identifying the main architectural concern of each class in the sample system, (ii) grouping these classes according to the architectural concern and (iii) calculating threshold values for each identified architectural concern. The output of this process is a table holding all the identified architectural concerns and their respective threshold values.


Threshold values by Architectural Concern
Project to be Reviwed
Identify Class Architectural Concern
Search thresholds by Architectural Concerns
Identify Long Methods from Architectural Concern
Long Methods List

Arquitetural Concern	Median	Max Threshold
Persistence	5	12
Business	15	32

```
class C extends persistence {
void aaa() {
...
}
void bbb() {
...
}
...
}
```

Figure 2. Identifying long methods based on architectural concerns.

Figure 2 shows the identification process of long methods considering the architectural concern. The input of the process is the table which contains threshold values for each architectural concern calculated in the first phase. The process is divided into three steps: (i) identifying the evaluated class architectural concern, (ii) obtaining threshold values in the table corresponding to architectural concern and (iii) identifying long methods according to the values found in the previous phase. The output of this process is a list of all the long methods in the evaluated system.

A key step in the two phases is the process of grouping classes in architectural concerns in both sample system and evaluated system. The method uses the hierarchy of classes to group them and then calculating the threshold values of each group. The first criterion, groups together the classes that extend the same class. After we group the classes that implement a system-defined interface itself. Finally, we group the classes that implement a non-system-defined interface. When a class is not categorized according to the specified criteria, a new group called *util* is created to link these classes. Additionally, the groups with only one element are merged with *util* group. The algorithm below describes how architectural concerns are identified:

Algorithm: Let $C = \{c1, c2, ..., cn\}$ be a set of classes of a system, $1 < i \leq n$. The architectural concern of a class ci is represented by the function AC(ci) = (Di, Ii, Ei), where:

i) Di is the set of classes extended by the class ci.
ii) Ii is the set of interfaces defined by the system (internal interfaces) and implemented by the class ci.
iii) Ei is the set of interfaces not defined by the system (external interfaces) and implemented by the class ci.

Let AC = $\{AC_1, AC_2, ..., AC_m)$ a set of architectural concerns of the sample system,$1 < j \leq m$. Each architectural concern in AC is represented by the tuple ACj = (Dj, Ij. Ej). A class ci is associated to an architectural concern ACj according to the following criteria:

1. c_i is associated to the AC_j when $D_i \neq \{\}$ and $D_i \subset$ Dj. If $D_i \neq \{\}$ and it is not associated to any element of AC, a new architectural concern is added to AC represented as follows AC_{m+1} = (Di, {}, {})
2. c_i is associated to the AC_j when $I_i \neq \{\}$ and $I_i \subset I_j$. If $I_i \neq \{\}$ and it is not associated to any element of AC, a new architectural concern is added to AC represented as follows $AC_{m+1} = (\{\}, I_i, \{\})$
3. c_i is associated to the AC_j when $E_i \neq \{\}$ and $E_i \subset E_j$. If $E_i \neq \{\}$ and it is not associated to any element of AC, a new architectural concern is added to AC represented as follows $AC_{m+1} = (\{\}, \{\}, E_i)$
4. c_i is associated to the AC_{util} when $D_i = \{\}$, $I_i = \{\}$ and $E_i = \{\}$. AC_{util} is represented as follows $AC_{util} = (\{\}, \{\}, \{\})$.

At the end of processing of all classes, the algorithm observes if there are some element in the AC set that have only one associated class. These elements are excluded from the set and joined with AC_{util} elements. Finally AC_{util} is added to the set AC.

We defined the standard threshold values for long methods identification in each AC_J as the median and the 75th percentile (third quartile) of the ordered set of values of the NLOC/Method metric. The percentile value can be adjusted to another value.

4. Experimental Setup

The main objective of this study is to *analyze* our approach to detect long methods *for the purpose of* evaluating *with the respect to their* accuracy *from the point of view of* software developers *in the context of* available approaches to detect long methods.

Target Systems: To achieve this goal we used nine versions of the MobileMedia system, a software product line (SPL) for applications that manipulate photo, music, and video on mobile devices [Figueiredo et al. 2008]. We selected this system because it have already been used in a study conducted by [Paiva et al. 2015] that analyzed the accuracy of InFusion, JDeodorant and PMD tools using the default settings in relation to a reference list of three code smells*: feature envy, god class and god method*. According to Paiva et al. (2015), experts identified the code smells occurrences that form this reference list. There is a reference list for each of the nine versions. We only used the data about the god method smell, considering that a god method is also a long method. The difference is that the existing approaches use other metrics, besides NLOC/Method, to identify god methods.

Research Questions: We made two comparisons with existing approaches. First, we compared a single generic threshold extracted from a sample system. Second, we compared reference values for each identified architectural concern also extracted from a sample system. As existing approaches we used results obtained by Paiva et al. (2015). We formulated the following research questions, one for each situation:

RQ1. *Does using a threshold value extracted from a sample system that follows the same reference architecture improve the accuracy of long method detection?*

RQ2. *Does using threshold values for each architectural concern, extracted from a sample system that follows the same reference architecture, improve the accuracy of long method detection?*

Data Collection: We chose the first version as the sample system to calculate the thresholds and evaluated the other versions in order to detect long methods. All the nine versions follow the same architecture reference. We also collect data using two percentiles (75 and 90) for setting the maximum threshold values. These percentiles were selected by convenience for an initial assessment of their influence on the results. To apply of the proposed approach and allow the reproducibility of study, we developed an open source plug-in for Eclipse, called ContextLongMethod[1]. Also, the collected data were compiled and made available on a website[2].

Data Analysis: We calculate recall, precision and F-score to compare the obtained results with the proposed approach and available tools. The F-Score or F-measure is a measure of accuracy. It considers both precision and recall measures to compute the score. The best F-score has value 1 and the worst score has value 0.

5. Threats to Validity

Below we present potential threats to the validity of the experiment and the actions taken to minimize them.

Internal validity: The application of the approach is automatic. The only threats to internal validity are related to the selection of the sample system and the percentile to

[1] https://github.com/marcosdosea/ContextSmellDetector/
[2] https://sites.google.com/site/cbsoft2016/

be considered to calculate the thresholds. As our study is exploratory, we selected two different percentiles in order to verify the variation on the results. Choosing initial versions to evaluate later ones seems to make sense when the versions follow the same architecture. But this is a decision we intend to further evaluate in future studies.

External validity: The results obtained are valid only for MobileMedia system and the long method smell. We do not suggest generalizing the results to other systems and other code smells.

Construct validity. The use of reference lists produced by experts to calculate metrics such as precision and recall usually represent threats to validity. In our study, we used a reference list produced by other researchers, so there is no bias in favor of our approach. In addition, the results of other approaches we compared our results with were calculated based on the same reference list.

6. Results and Discussion

Table 1 summarizes the findings obtained by Paiva et al. (2015) compared with the our findings. The results of precision, recall and F-Score of each approach are shown. Each value represents the average of the values obtained for all nine versions (versions 1 to 9). Detailed results, including the number of false positives and false negatives used to calculate precision and recall of each method, are available on our website.

The first three lines show the results obtained with the use of InFusion, Jdeodorant and PMD. The following lines present the four configurations executed in our study. The third and forth lines show the results of the use of a generic threshold value extracted from a sample system (the first version of MobileMedia). The last two lines are about the use of a different threshold for each architectural concern.

Table 1. Results of analyzes performed in MobileMedia

		% Precision	% Recall	% F-Score
Paiva et al. (2015)	inFusion	100	26	41,27
	Jdeodorant	35	50	41,18
	PMD	100	26	41,27
Proposed Approach	Percentile 75	27	100	42,52
	Percentile 90	56	95	70,46
	Percentile 75 + architectural concern	32	100	48,48
	Percentile 90 + architectural concern	60	89	71,68

Given these results, we discuss the two research questions as follows:

***RQ1**. Using a threshold value extracted from a sample system that follows the same reference architecture improves the accuracy of long method detection?*

The results obtained by using a generic threshold extracted from the first version of MobileMedia showed improvements in F-score. Despite the precision of 100% obtained with inFusion and PMD, their recall values show that these tools have not found a high number methods from the reference list. JDedorant found 50% of the reference list of methods, but its low precision (35%) generates the lowest F-Score (41,18%). Our approach using the 75th percentile, despite the low precision (27%), found all the methods of the reference list (100%). With the 90th percentile we found the best F-score (70.46%) when compared with the tools. A precision of 56% and a recall of 95% generate this high value of F-Score. Thus, we found that only using a previous version for extracting the threshold value was already enough to improve the accuracy of long method detection.

We notice that the F-Score with the 90th percentile was higher than with the 75th percentile. However, it is noteworthy that by using the 90th percentile we did not reach 100% of recall, as with the 75th percentile. We believe that achieving greater recall in this case is more important than a high precision. It is easier for developers to analyze false positives (low precision) than analyzing false negatives (low recall). In the evaluation performed by Paiva et al. (2015), no approach was able to detect all the long methods identified by the experts, i.e. none approach obtained 100% recall. This can lead to a false sense that the system has few design flaws.

RQ2*. Using threshold values for each architectural concern, extracted from a sample system that follows the same reference architecture, improves the accuracy of long method detection?*

Comparing the two configurations of our approach, we noticed small differences on the F-Score when architectural concern was considered. With the 75th percentile, we observed a small improvement in the precision (from 27% to 32%) when we considered architectural concerns. With the 90th percentile, there was a decrease in recall (from 95% to 89%), but there was also improvement in precision (from 56% to 60%) when we considered architectural concerns. This last configuration (90th percentile + architectural concerns) produced the best F-score (71.68%) among all approaches.

In summary, because the differences were too small, we are not able to claim that the use of architectural concerns in our approaches improves the accuracy of detecting long methods. The only claim that we can make is that both configurations of our approaches (with and without architectural concerns) improved the accuracy in comparison with existing tools. We intend to carry out further studies with larger systems to verify whether there are advantages on the use of architectural concerns.

7. Conclusion and Future Works

In this work we present a new approach to indicate long methods for developers. We also show an initial assessment of this approach. The approach proposes to extract threshold values of a sample system considering the class architectural concern. The initial evaluation of this approach showed better results compared to existing tools. One major benefit of the method is the increase of recall rate. This means reduction in the number of false negatives which are usually more difficult to be detected by developers.

As future work, we are going to extend the architectural concern identification method to consider more levels of class hierarchy and improve the precision of the approach. We are also extending the approach to consider other code smells. We also intend to perform a detailed analysis of classes classified in each group to understand how the proposed algorithm works in different system architectures. Finally, we plan to evaluate our approach in the context of real systems for more reliable results.

Acknowledgements. This work was supported by CNPq: grant 573964/2008-4 (National Institute of Science and Technology for Software Engineering) and grant 486662/2013-6.

References

Angelov, S., Grefen, P. & Greefhorst, D., 2009. A classification of software reference architectures: Analyzing their success and effectiveness. *2009 Joint Working*

IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture, pp.141–150.

Arcelli Fontana, F. et al., 2015. Automatic Metric Thresholds Derivation for Code Smell Detection. In *2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics*. IEEE, pp. 44–53.

Arcoverde, R. et al., 2012. Automatically detecting architecturally-relevant code anomalies. In *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*. IEEE, pp. 90–91.

Figueiredo, E. et al., 2008. Evolving software product lines with aspects. In *Proceedings of the 13th international conference on Software engineering - ICSE '08*. New York, New York, USA, New York, USA: ACM Press, p. 261.

Fontana, F.A. et al., 2013. Investigating the Impact of Code Smells on System's Quality: An Empirical Study on Systems of Different Application Domains. In *2013 IEEE International Conference on Software Maintenance*.

Guo, Y. et al., 2010. Domain-specific tailoring of code smells: an empirical study. *2010 ACM/IEEE 32nd International Conference on Software Engineering*.

van Gurp, J. & Bosch, J., 2002. Design erosion: problems and causes. *Journal of Systems and Software*, 61(2), pp.105–119.

Lavallee, M. & Robillard, P.N., 2015. Why Good Developers Write Bad Code: An Observational Case Study of the Impacts of Organizational Factors on Software Quality. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, pp. 677–687.

Macia, I. et al., 2013. Enhancing the detection of code anomalies with architecture-sensitive strategies. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pp.177–186.

Marinescu, C., 2006. Identification of Design Roles for the Assessment of Design Quality in Enterprise Applications. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE, pp. 169–180.

Marinescu, R., 2004. Detection strategies: metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance*.

Paiva, T. et al., 2015. Experimental Evaluation of Code Smell Detection Tools. *3th Workshop on Software Visualization, Evolution, and Maintenance (VEM 2015)*.

Palomba, F. et al., 2013. Detecting bad smells in source code using change history information. *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, pp.268–278.

Parnas, D.L., 1994. Software aging. In *Proceedings of 16th International Conference on Software Engineering*. IEEE Comput. Soc. Press, pp. 279–287.

Tufano, M. et al., 2015. When and Why Your Code Starts to Smell Bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*.

Vale, G.A. Do & Figueiredo, E.M.L., 2015. A Method to Derive Metric Thresholds for Software Product Lines. *2015 29th Brazilian Symposium on Software Engineering*.

Zhang, F. et al., 2013. How Does Context Affect the Distribution of Software Maintainability Metrics? In *2013 IEEE International Conference on Software Maintenance*. IEEE, pp. 350–359.