Applying the Single Responsibility Principle in Industry: Modularity Benefits and Trade-offs

Apostolos Ampatzoglou, Angeliki-Agathi Tsintzira, Elvira-Maria Arvanitou, Alexander Chatzigeorgiou, Ioannis Stamelos, Alexandru Moga, Robert Heb, Oliviu Matei, Nikolaos Tsiridis, Dionisis Kehagias

Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece
Department of Informatics, Aristotle University of Thessaloniki, Thessaloniki, Greece
Holisun, Baia Mare, Romania
Open Technology Services, Thessaloniki, Greece
Information and Technology Institute, Center for Research and Technology, Thessaloniki, Greece

apostolos.ampatzoglou@gmail.com, angeliki.agathi.tsintzira@gmail.com, earvanitoy@gmail.com, achat@uom.gr, stamelos@csd.auth.gr, alexandru.moga@holisun.com, robert.heb@holisun.com, oliviu.matei@holisun.com, ntsiridis@gmail.com, diok@iti.gr

ABSTRACT

Refactoring is a prevalent technique that can be applied for improving software structural quality. Refactorings can be applied at different levels of granularity to resolve ‘bad smells’ that can be identified in various artifacts (e.g., methods, classes, packages). A fundamental software engineering principle that can be applied at various levels of granularity is the Single Responsibility Principle (SRP), whose violation leads to the creation of lengthy, complex and non-cohesive artifacts; incurring smells like Long Method, God Class, and Large Package. Such artifacts, apart from being large in size tend to implement more than one functionalities, leading to decreased cohesion, and increased coupling. In this paper, we study the effect of applying refactorings that lead to conformance to the SRP, at all three levels of granularity to identify possible differences between them. To study these differences, we performed an industrial case study on two large-scale software systems (more than 1,500 classes). Since SRP is by definition related to modularity, as a success measure for the refactoring we use coupling and cohesion metrics. The results of the study can prove beneficial for both researchers and practitioners, since various implications can be drawn.

CCS CONCEPTS

Software and its engineering → Software creation and management → {Software development techniques → Object-oriented development}

EASE '19, April 15-17, 2019, Copenhagen, Denmark
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3319008.3320125

KEYWORDS

Refactorings, industrial case study, modularity, software metrics

ACM Reference Format

A Ampatzoglou, A. A. Tsintzira, E. M. Arvanitou, A. Chatzigeorgiou, I. Stamelos, A. Moga, R. Heb, O. Matei, N. Tsiridis, and D. Kehagias, “Benefits of Applying the Single Responsibility Principle to Repay Technical Debt: An Industrial Case Study”, In *Proceedings of 23rd Conference on the Evaluation and Assessment in Software Engineering (EASE’ 19), Copenhagen, Denmark, 15-17April 2019*.

1 Introduction

According to the seminal book of Hans van Vliet, software design should consider four aspects: abstraction, modularity, information hiding, and complexity [18]. Among those, in this paper we focus on software modularity, which is defined as the “*degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components* [1]”. According to Martin [14] the levels of modularity can be assured by applying the Single Responsibility Principle (SRP). SRP states that every module should have exactly one responsibility, i.e., be related to only one functional requirement, and therefore have only one reason to change. The term single responsibility has been inspired by the functional module decomposition, as introduced by Tom De Marco [7]. To assess if a class conforms to the SRP, one needs to assess its cohesion [14], which is related to the number of diverse functionalities that a class is responsible for [7]. However, by considering the inherent reverse relation between coupling and cohesion, proper application the SRP, shall not only consider the improvement of artifacts’ cohesion, but also possible trade-offs between coupling and cohesion (i.e., enhancing one can diminish the other) [18].

Lack of modularity can lead to the existence of various smells, based on the artifact that it is applied to: *Long Methods* (that are resolved through the *Extract Method* refactoring [4]), *God Classes* (that are resolved through the *Extract Class* refactoring [9]), and

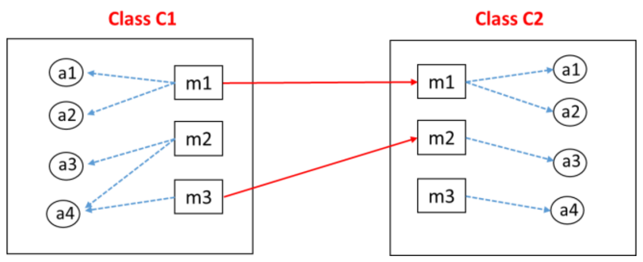
Large Packages (that are resolved through the *Move Class* refactoring [16]). All the aforementioned smells, follow the same pattern: there is a large in size artifact that among others, is related to more than one responsibilities. The solution would be to split this artifact into smaller ones, maintaining the external behavior of the system. For the majority of the cases, the presence of the smell is resolved by examining the cohesion of the long artifact, and create new ones with better levels of cohesion. Although the benefit of applying these refactorings in terms of cohesion is safeguarded by the nature of the proposed approaches (i.e., cohesion-based optimizations), the effect on modularity remains vague, since if coupling substantially deteriorates, then modularity might be harmed.

Driven by the above setting, in this paper we investigate the effect of applying the Single Responsibility Principle on software modularity at three levels of granularity: (a) method-, (b) class-, and (c) package-level. Additionally, by considering that quality trade-offs rarely occur in small-scale applications, we preferred to perform a case study on real-world artifacts, retrieved from an industrial setting. In particular, we have studied two systems with long evolution history and large size, and manually performed refactorings at all three levels. Then we compare: (a) the effect of the refactorings on modularity, regardless of the level of granularity, (b) the effect of the refactorings on modularity, given the level of granularity, and (c) the trade-offs between coupling and cohesion when applying the refactoring, in all levels of granularity.

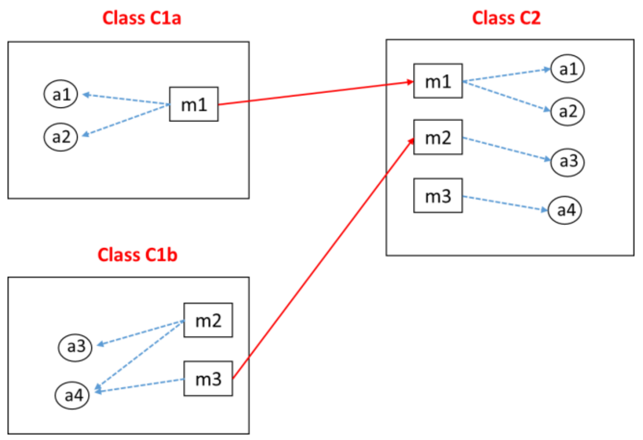
The rest of the paper is organized as follows: Section 2 provides brief background information on coupling, cohesion, the metrics that have been used for measuring them, and the tools that we have opted for getting refactoring suggestions. Section 3 presents the case study design. In Section 4 we present the results of the industrial case study, which we discuss in Section 5, in which we also conclude the paper.

2 Background Information and Used Tools

In a typical Object-Oriented (OO) system, methods and attributes are grouped together in classes, based on their functional similarity. In order for a class to be modular, methods that belong to the same class are expected to highly interact with the attributes of the class (high-cohesion), whereas dependencies to methods belonging to different classes should be limited (low-coupling). For example, in Figure 1(a), we can observe (through an artificial example) that in class `C1` there are two groups of pairs of methods and attributes: (a) method `m1` uses attributes `a1` and `a2`, and (b) method `m2` uses `a3` and `a4`, whereas method `m3` uses `a4`. Thus, based on SRP, class C1 needs to be slit. It should be noted that class C1 is coupled to class C2 because of method invocations (one coupling relationship). After the application of SRP (see Figure 1(b)), we split class `C1` into two new classes: (a) `C1a—m1` method with `a1` and `a2` attributes, and (b) `C1b—m2` method with `a3` and `a4` attributes and `m3` method. By assessing the modularity of the system, we can observe that the lack of cohesion after the application of SRP becomes zero, whereas the coupling increases (two coupling relationships). Therefore, the assessment of modularity cannot be conclusive, since there is a trade-off between the two quality properties that comprise it. We note, that since the goal of this example is to only demonstrate SRP, we do not continue the narration on how C2 could be split.



(a) Design-before the application SRP



(b) Design-after the application SRP

Figure 1. Modularity Example

The measurement of coupling and cohesion is differentiated, based on the artifact that is being examined: At the *package / architecture level*, we employ the metrics presented by Skiada et al. [17], namely ***Average Coupling Afferent (ACa)***, which represents the average afferent coupling of packages. Afferent coupling is the number of outgoing dependencies of a package to other packages; and ***Cohesion among Package Classes (CaPC)***, which assesses how closely two classes that belong to the same package collaborate with each other. The metric is inspired by reversing the calculation of Lack of Cohesion of Methods [6]: we compute the total number of pairs of classes that belong to one package, and then we investigate the percentage of these pairs that are coherent (i.e., they are coupled to each other). *At the class level* we use: ***Message Passing Coupling (MPC)***, which measures the number of method calls defined in methods of a class to methods in other classes, and therefore the dependency of local methods to methods implemented by other classes [12]; and ***Lack of Cohesion-5 (LCOM5)***, which measures the degree to which methods and fields within a class are related to one another, providing one or more components [11]. *At the method level*, we use the transformation presented by Charalampidou et al. [5] to develop ***method-level metrics from LCOM5 and MPC***. Finally, ***modularity*** is obtained by dividing coupling by cohesion: when lack of cohesion is measured, we reversed them (`1-metric score`) to obtain cohesion (all metrics are bounded to 1).

Regarding the identification of refactoring opportunities, we have used three different tools, described as follows:

- ***Method-level refactoring***: We used the SEMI tool developed by the University of Groningen [4].
- ***Class-level refactoring:*** We used jDeodorant, developed by the University of Macedonia and Concordia University [9].
- ***Architecture-level refactoring***: We used the MCR tool, developed by the University of Western Macedonia.

All tools have been used with their initial configuration for refactorings opportunities identification. The tools calculate the selected metrics before and after the application of the change. The only exception is jDeodorant that does not calculate LCOM5 at class level; thus, we used a NetBeans plug-in for this calculation.

3 Case Study Design

To investigate the effect of applying the Single Responsibility Principle on modularity, we performed an industrial case study in two small-medium enterprises (SMEs), one in Greece and one in Romania. The Greek SME is active in enterprise applications, whereas the Romanian one in Augmented Reality systems for Smart Manufacturing. The case study is designed according to the guidelines by Runeson et al. [15].

Objective and Research Question. This study aims to compare: (*goal-a*) the effect of the refactorings on modularity, and (*goal-b*) the trade-offs between coupling and cohesion. (*Goal-a)* is examined by first not considering the level of granularity of the artifact, in which the refactoring takes place, and (*Goal-b)* by taking this parameter into account. To this end, we derived three questions:

RQ$_1$: What is the effect of applying the SRP on modularity?

RQ$_2$: Is the effect of applying the SRP on modularity, different based on the granularity of the artifact?

RQ$_3$: Are the trade-offs between coupling and cohesion different based on the granularity of the artifact?

RQ$_1$ and RQ$_2$ are related to *goal-a*, whereas RQ$_3$ is related to *goal-b*. We preferred not to split *goal-b* to two research questions, due to space limitations. Achieving *goal-a* is expected to shed light on the effect of the refactoring on modularity as a whole, whereas in *goal-b*, we aim digging further into the two quality properties that comprise modularity.

Case Selection and Units of Analysis. To collect data for our case study, we executed the three tools mentioned in Section 2 on the source code of two projects (written in Java) of the collaborating SMEs, as described below:

- ***YDATA*** (developed by OTS) deals with customer management and billing of the national water supplier. It consists of 651 classes (45K lines of code) that have been developed and maintained for 384 commits between 2015 and 2017. YDATA can be decomposed into 6 main sub-systems, each one managing the following entities: (a) Hydrometers, (b) Bills, (c) Users, (d) Consumption Statements, (e) Payments, and (f) Alerts to Users.
- ***MaQuali*** (developed by Holisun—HS) is a software application for the handling of quality management systems (ISO 9001) along with business processes. It consists of 990 classes (152K lines of code) that have been developed between 2009 and 2018. The system consists of 6 main modules, managing the following entities: (a) fiches of progress, (b) actions to be taken, (c) documents involved in ISO quality control, (d) planning, (e) useful information, and (f) milestones.

Regarding the identification of refactorings we used the complete code base, and considered the most urgent ones based on the suggestions of the tools (usually in terms of severity). Therefore, the units of our analysis are 131 artifacts (packages, classes, and methods) that are selected based on the aforementioned strategy. As observed in Table I, the dataset can be split into 6 distinct datasets, based on the company from which data have been retrieved and the level of granularity at which refactoring is applied.

TABLE I. UNITS OF ANALYSIS

Dataset	Company	Level
DS1	HS	Packages
DS2	OTS	Packages
DS3	HS	Classes
DS4	OTS	Classes
DS5	HS	Methods
DS6	OTS	Methods

Data Collection. To answer the stated research questions, the next steps are followed:

- identify refactoring opportunities (see Section 2 for tools)
- identify the artifacts that need refactoring
- for these artifacts calculate coupling and cohesion (`coupling`$_{\texttt{before}}$ and `coupling`$_{\texttt{before}}$)
- apply the refactorings
- for the resulting artifacts calculate coupling and cohesion (`coupling`$_{\text{after}}$ and `coupling`$_{\texttt{after}}$)
- Finally, we calculate `modularity`$_{\texttt{before}}$ and `modularity`$_{\texttt{after}}$ the application of the change by dividing cohesion to coupling.

Our dataset consists of 131 rows and 9 columns as follows:

[V1] *company*: OTS / HS

[V2] *level*: architecture / design / implementation

[V3–V5] *cohesion metrics*: coh$_{\text{before}}$, coh$_{\text{after}}$, coh$_{\text{diff}}$

[V6–V8] *coupling metrics*: coup$_{\text{before}}$, coup$_{\text{after}}$, coup$_{\text{diff}}$

[V9–V11] *modularity metrics*: mod$_{\text{before}}$, mod$_{\text{after}}$, mod$_{\text{diff}}$

Data Analysis. As part of data analysis, we first present some demographics on the `before` and `after` variables of the quality properties of interest (coupling, cohesion, and modularity). Then, we perform independent sample t-tests for investigating possible differences between the two industrial codebases, and if their results can be treated as one dataset. For answering the aforementioned research questions we are using the analysis strategy presented in Table II, which includes visualization techniques and hypothesis testing. We note that V1 is used only for demographic reasons, and V2 is used for splitting purposes in RQ_2 and RQ_3.

TABLE II. ANALYSIS STRATEGY

RQ	Dataset	Variables	Analysis
RQ_1	Complete	V11	Pie chart
		V9, V10	Paired-Sample t-test
RQ_2	DS1 + DS2 DS3 + DS4 DS5 + DS6	V11	Pie chart
		V9, V10	Paired-Sample t-test
RQ_3	DS1 + DS2 DS3 + DS4 DS5 + DS6	V5, V8	Pie chart
		V3-V4 and V6-V7	Paired-Sample t-test

4 Results

In this section we present the results of our industrial study. In Table III we present the descriptive statistics of our sample. Additionally, a hypothesis testing has been performed, so as to investigate if the mean values presented in Table III are statistically different between the two companies. The results of the analysis suggested that the mean values *do not differ significantly*, and that therefore the sample can be used as a whole, without a need for reliability and generalization assessment [2]. We note that in this section we do not provide any interpretation of results, since they are thoroughly discussed in Section 5.

TABLE III. DESCRIPTIVE STATISTICS

Metric	Min	Max	Mean	SDev.
cohesion_before	0.000	0,980	0,173	0,337
cohesion_after	0.000	0,954	0,142	0,275
coupling_before	0.045	36,500	15,112	14,654
coupling_after	0,042	36,000	14,773	14,379
modularity_before	0,000	268,000	5,241	20,934
modularity_after	0,000	200,000	5,4772	16,840

Effect of Refactorings on Modularity. As a first step of investigating the effect of applying SRP-driven refactorings on artifacts' modularity (regardless of granularity), we treat the complete dataset as a whole (RQ_1). The overview presented in Figure 2, suggests that in 80% of the cases the refactoring has a positive effect on artifacts' modularity. However, the performed hypothesis testing (paired-sample t-test) suggested that this result is not statistically significant, i.e., the differences in the mean values of modularity `before` and `after` the application of the refactoring are not statistically significant. A possible interpretation of this observation is the fact that in 28% of the cases the improvement was marginal (e.g., 0.001), especially in architecture level (packages).

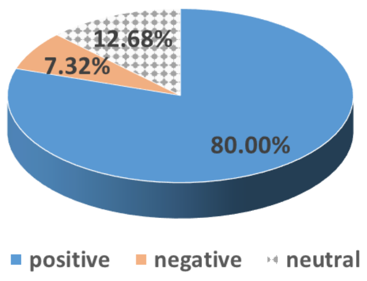
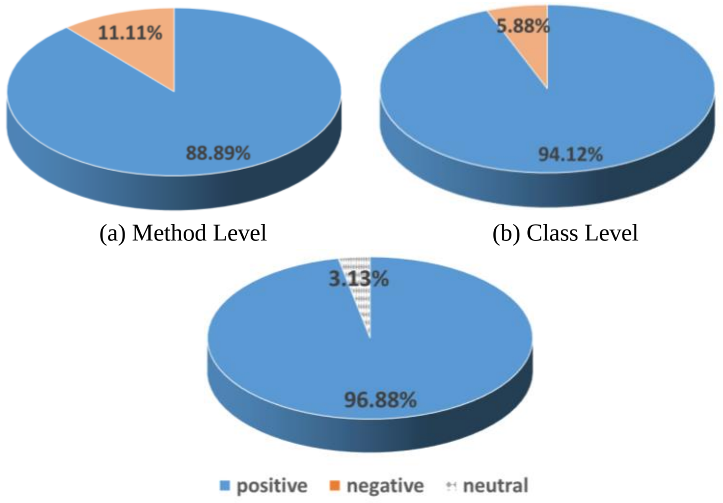


Figure 2. Effect on Modularity (no distinction of granularity)

As a next step, we treat each level of granularity separately and repeat the analysis. The obtained results are presented in Figure 3 and Table IV. The results suggest the SRP-driven refactoring is having a positive influence (that is statistically significant) at all levels of granularity. However, the expected benefit at the architecture level in absolute numbers is lower. Nevertheless, based on Figure 3 we can observe that at the architecture level, we are only having positive and limited neutral effects on modularity.



(a) Method Level (b) Class Level

(c) Architecture Level

Figure 3. Effect on Modularity in different levels of granularity

TABLE IV. HYPOTHESIS TESTING FOR MODULARITY

Level	Before	After	Improvement	t-value	sig.
Method	0.45	1.03	128.89%	-3.331	0.00
Class	0.66	1.51	128.79%	-2.297	0.03
Architecture	1.64	2.34	42.68%	-2.546	0.01

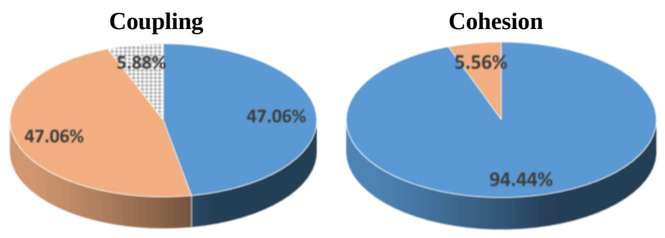
SRP-driven refactoring approaches are in most of the cases improving the modularity of the software. The improvement is more evident in terms of actual impact at the method and class level. However, at the architecture level the frequency of cases when the refactoring is beneficial is higher compared to the other levels, and there are no cases that the refactoring is harmful.

Trade-offs between Coupling and Cohesion. To investigate the trade-offs between coupling and cohesion when refactoring, we have followed the same process as before. The results are presented in Table V and Figure 4. We note that all tools that have been used for identifying refactoring opportunities are optimizing one of the two quality properties (*directly affected*): method and class level refactorings are extracted based on cohesion, whereas at the architecture level the optimization is performed based on coupling. Therefore, while studying trade-offs, by construction, the used tools guarantee the improvement of one quality property, and the levels of the other one (*indirectly affected*) is being investigated. We report the findings for each level of granularity separately:

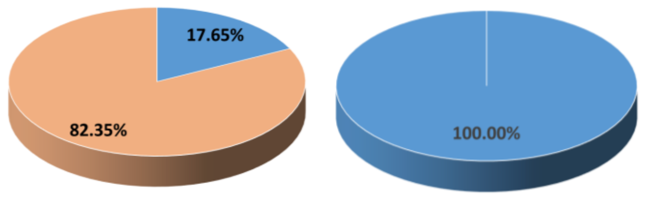
- *Method level*: We can observe that there is a marginal trade-off between the quality properties in terms of mean values; however, the results on the deterioration of coupling are marginal and not statistically significant. Regarding the frequency of improvement and deterioration, in Figure 4a, we can observe that the sample is balanced.
- *Class level*: This is the only level at which substantial trade-offs are evident (i.e., benefit in cohesion and deterioration of coupling in Table V when extracting a class from a God one). Figure 4b, suggests that the count of cases in which coupling deteriorates is higher compared to the times it improves.
- Architecture-level: Finally, with respect to architecture no trade-offs are evident. More specifically, coupling is always improving (see Figure 4c) and the difference between coupling scores before and after is statistically significant. However, with respect to indirectly affected quality property (i.e., cohesion) the difference is not statistically significant, although the effect is positive in average. This observation is due to the fact that in 75% of cases that coupling is improving, there is no effect on cohesion (grey area in Figure 4c).

TABLE V. HYPOTHESIS TESTING FOR TRADE-OFFS

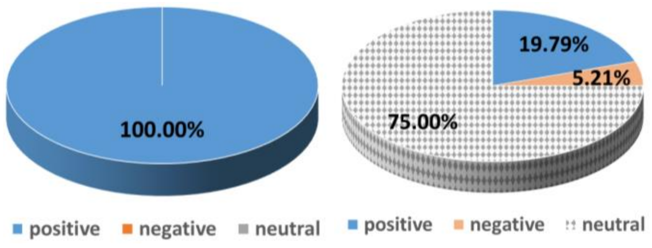
Level	Metric	Before	After	Improvement	t-value	sig.
Method	Cou	0.191	0.192	-0.52%	0.074	0.94
	LCoh	0.933	0.888	4.82%	4.041	0.00
Class	Cou	0.230	0.486	-111.30%	-4.641	0.00
	LCoh	0.876	0.515	41.21%	11.653	0.00
Architecture	Cou	26.000	25.239	2.93%	3.244	0.00
	Coh	0.103	0.115	11.65%	-1.783	0.07



(a) Method Level



(b) Class Level



(c) Architecture Level

Figure 4. Trade-offs between Coupling and Cohesion

Applying the SRP improves the quality property (coupling or cohesion) that drives the refactoring, at a statistically significant level. Regarding trade-offs, at the architecture level we observed that both quality properties are improved; whereas at the method and class level trade-offs take place. Nevertheless, trade-offs at class level are more impactful.

5 Discussion / Conclusions

In this paper we examined the effect of applying the SRP (at various level of granularity) on software artifacts' modularity. To achieve this goal, we performed an industrial case study on two software development companies, exploring 131 software artifacts.

Interpretation of the results. Based on the empirical evidence that we have been able to deliver, we suggest that the application of the Single Responsibility Principle is beneficial concerning the modularity of the two industrial systems. Detailed findings are presented and interpreted below:

- *Indifferent impact of SRP regardless of the level of granularity*. The results of the study suggest that the effect of SRP-driven refactorings on modularity is not statistically significant, when not discriminating among the different levels of granularity. This finding is expected in the sense that the level of magnitude for each refactoring is different, and the effect on quality varies across difference scales. Such findings are common in the software engineering literature: e.g., Feitosa et al. [8], investigated the impact of patterns on quality, and the results appeared to be controversial without discriminating per pattern type.
- *Effect of Refactorings on Modularity per level*. The findings of this study can be interpreted based on two data-sources: (a) the frequency of cases in which the refactoring is beneficial, and (b) the effect size—the absolute value of the change in the modularity metric. Regarding the frequency of beneficial refactorings, we can observe that as the level of granularity of the refactorings increases (i.e., from method to architecture) the more probable it is to obtain a benefit. However, the effect size is decreasing. This observation can be explained by the fact that re-

factorings at the architecture level are expected to be more impactful [13]; however, according to Arvanitou et al. [3] the metrics at the architecture level are more stable (i.e., their values are not easily fluctuating in successive releases). In other words, applying the SRP at architecture artifacts has a more definite impact, but it is more unlikely to sense the change by metrics.

- *Refactoring Trade-offs*. The findings regarding trade-offs suggested that coupling and cohesion are inversely related properties, that are very sparse to optimize simultaneously. Therefore, we have delivered evidence on the existence of trade-offs at class level, and identified marginal trade-offs at method level. The case study has not revealed any trade-offs at the architecture level. This finding occurs due to the fact that while optimizing coupling at the package level, the cohesion of the system remains unaffected (neither positive nor negative impact). A possible interpretation of this observation is that the calculation of the metric (pct. of intra-package dependencies) remains unaffected by moving one class from one package to another, because usually the number of classes within a package is large.
- *Differences and Similarities among Artifacts*. The findings of the study suggest that the results at the method and the class level are similar to each other, and substantially differ from those at the architecture level. This is considered an expected outcome in the sense that methods and classes are very close in terms of granularity, compared to architecture artifacts which are substantially larger and their investigation goes to a completely different scale. The results imply that allocation of instructions to methods and of methods to classes pertain to design, while the allocation of classes to packages pertains to architecture, and the two processes differ substantially.

Implications to Researchers and Practitioners. The outcomes of this work provides useful insights on the application of SRP. Regarding researchers interesting future work opportunities are:

- *Need for more studies on the class and method level*. The fact that class- and method-level refactorings produce trade-offs between coupling and cohesion, suggest that there is a need for further improvement in these areas, which would lead to the development of methodologies that treat the problem as a multi-criteria one, since the optimization only in terms of cohesion might deteriorate coupling. Our results suggest that the need is more intense at the level of classes. One promising line of research is that of Search-Based Software Engineering (SBSE) [10] which treats the allocation of code, methods, and classes as a search-space optimization problem.
- *Replication*. The study needs to be replicated with other programming languages, tools for refactoring identification and a larger dataset. This would strengthen the generalizability of the suggested results, which at this stage is limited to Java, three tools, and two industrial projects. It would be equally interesting to investigate trade-offs between the qualities affected by the application of SRP using a wider set of metrics.

Concerning practitioners, the findings of this study guide software engineers in the possible problems that might occur unintentionally, when refactoring a source code, based on tool suggestions, without having an in-depth knowledge of the consequences of the refactoring process. Therefore, all decisions shall be thoroughly considered, by treating suggestions with caution and by paying special attention to possible trade-offs between refactoring opportunities. In the context of continuous integration which gradually becomes the norm, the observed trade-offs call for the use of appropriate monitoring tools that will be able to pinpoint artifacts which are adversely affected by an attempted refactoring. However, the results suggest that refactorings at the architecture level appear to be safer than those at the source code level, since they are having a larger probability to increase one quality property, without affecting the other.

ACKNOWLEDGMENTS

Work reported in this paper was financially supported by the action "Strengthening Human Resources Research Potential via Doctorate Research" of the Operational Program "Human Resources Development Program, Education and Lifelong Learning, 2014-2020", implemented from State Scholarship Foundation (IKY) and co-financed by the European Social Fund and the Greek public (National Strategic Reference Framework (NSRF) 2014–2020).

REFERENCES

[1] 25010-2011 ISO/IEC Systems and software engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — System and Software Quality Models, 2011.

[2] 1061-1998: IEEE Standard for a Software Quality Metrics Methodology, IEEE Standards, IEEE Computer Society, reaffirmed December 2009.

[3] E. M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, "Software Metrics Fluctuation: A property for assisting the metrics selection process", Information and Software Technology, Elsevier, 72 (4), 2016.

[4] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, A.Gkortzis, and P. Avgeriou, "Identifying Extract Method Refactoring Opportunities Based on Functional Relevance", Transactions on Software Engineering, IEEE, 43, 2017.

[5] S. Charalampidou, E. M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou, and I. Stamelos, "Structural Quality Metrics as Indicators of the Long Method Bad Smell: An Empirical Study", 44th Conference on Software Engineering and Advanced Applications (SEAA' 18), IEEE, August 2018.

[6] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design", Transactions on Software Engineering, IEEE, 20 (6), 1994.

[7] T. De Marco, Structured Analysis and System Specification. Yourdon, 1979.

[8] D. Feitosa, A. Ampatzoglou, P. Avgeriou, and E. Y. Nakagawa, "What can violations of Good Practices tell about the Relationship between GoF patterns and Run-Time Quality Attributes", *Information and Software Technology*, Elsevier, 2019.

[9] 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, Elsevier, 85 (10), pp. 2241-2260, 2012.

[10] M. Harman, B. F. Jones, "Search-based software engineering", Information and Software Technology, Elsevier 43(14), pp. 833-839, 2001.

[11] B. Henderson-Sellers, "Object-Oriented Metrics Measures of Complexity", Prentice-Hall, 1996.

[12] W. Li, and S. Henry, "Object-oriented metrics that predict maintainability", Journal of Systems and Software, Elsevier, 23 (2), pp. 111-122, 1993.

[13] Z Li, P Liang, P Avgeriou, N Guelfi, and A. Ampatzoglou, "An Empirical Investigation of Modularity Metrics for Indicating Architectural Technical Debt", 10th International Conference on the Quality of Software Architectures (QoSA'14), ACM, 2014.

[14] R. C. Martin "Agile software development: principles, patterns and practices", Prentice Hall, 2003.

[15] P. Runeson, M. Höst, A. Rainer, and B. Regnell, "Case Study Research in Software Engineering: Guidelines and Examples", John Wiley and Sons, 2012.

[16] S. M. A. Shah, J. Dietrich, C. McCartin, "Making Smart Moves to Untangle Programs", 16th European Conference on Software Maintenance and Reengineering (CSMR 2012), Szeged, Hungary, IEEE, 27–30 March 2012.

[17] P. Skiada, A. Ampatzoglou, E. M. Arvanitou, A. Chatzigeorgiou, and I. Stamelos "Exploring the Relationship between Software Modularity and Technical Debt", 44th Conference on Software Engineering and Advanced Applications (SEAA' 18), IEEE, August 2018.

[18] H. van Vliet, "Software Engineering: Principles and Practice", John Wiley and Sons, 2008.