Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 109C (2017) 521–528

Procedia Computer Science

www.elsevier.com/locate/procedia

The 8th International Conference on Ambient Systems, Networks and Technologies (ANT 2017)

Applying Design Patterns to Remove Software Performance Antipatterns: A Preliminary Approach

Davide Arcelli[1], Daniele Di Pompeo[1]

[a]*DISIM - University of L'Aquila - Via Vetoio, L'Aquila 67100, Italy*

Abstract

Patterns and antipatterns represent powerful instruments in the hands of software designers, for improving the quality of software systems. A large variety of design patterns arose since decades, as well as several performance antipatterns have been defined.

In this paper we propose a preliminary approach for antipattern-based refactoring of software systems, driven by design patterns application. The approach is focused on refactoring software artifacts (i.e., models, code) by applying design patterns, with the aim of removing possible performance antipatterns occurring on such artifacts. Based on our approach, design patterns are ranked in order to drive the refactoring choice. We also provide an illustrative example as a preliminary validation of our approach, showing how the ranking method works over three design patterns for removing the Empty Semi-Trucks performance antipattern, and we finally identify future research directions of our work.

Peer-review under responsibility of the Conference Program Chairs.

Keywords: Performance Antipatterns; Design Patterns; Software Refactoring; Software Performance Engineering

1. Introduction

In the software development process, the concept of design pattern has been introduced several decades ago for defining good practices to design software[6]. Conversely, few decades ago the concept of antipattern has been introduced for characterizing bad design practices. In this context, Smith and Williams[13] introduced particular kinds of antipatterns, namely *performance antipatterns*, which are bad design practices that may lead performance to degrade.

A large variety of design patterns and antipatterns has been defined in literature, and they have been (and still are) extensively used in industry, because they revealed to be powerful instruments in the hands of software designers for improving the quality of the software product.

* Corresponding author
* Tel.: +39-0862-433182.
E-mail address: davide.arcelli@univaq.it

Peer-review under responsibility of the Conference Program Chairs.
10.1016/j.procs.2017.05.330

Performance antipatterns have been conceived to address performance issues since the early phases of the software life-cycle [13], where it is necessary to introduce a particular actor into the life-cycle – namely *performance expert* – that works towards the fulfillment of performance requirements. Introducing such an actor is costly and, for this reason, it is not widespread in software development, thus delaying performance issues resolution to the testing phase.

Goal of this paper is to reduce the gap between design patterns and performance antipatterns, by providing a preliminary approach aimed at applying the former ones in order to remove the latter occurring into software artifacts. Our approach exploits a synergy between design patterns and performance antipatterns, towards the fulfillment of performance requirements and the improvement of the software quality. Moreover, we work in a fuzzy context, where threshold values related to performance antipattern metrics (e.g., the number of connections that a component has with other components in the system is too high) cannot be determined, but only their lower and upper bounds do. In this context, a ranking criteria for design patterns is proposed in order to drive the choice of the one that is most suitable to apply for removing a certain performance antipattern. Such a ranking is based on a score assigned to each design pattern, quantifying the probability that the pattern removes the corresponding antipattern.

We apply our approach to a performance antipattern (i.e. Empty Semi-Trucks) and three design patterns (i.e., Session Façade, Batching, and Aggregate Entity), showing how the latter may be ranked towards the resolution of the former.

The benefit of our approach is two-fold: On the one hand, removing performance antipatterns likely enhances the performance of the software system; On the other hand, introducing design patterns likely improves the quality of the software design.

This paper is organized as follows. Section 2 describes our refactoring process. Section 3 describes our ranking criteria to drive the choice of design pattern towards the removal of performance antipatterns. Section 4 provides a preliminary validation on an illustrative example; Section 5 discusses related work. Finally, Section 6 concludes the paper and presents future work.

2. Design Patterns Vs Performance Antipatterns Approach

In this section we describe the envisioned refactoring process and the ranking criteria to drive the choice of design patterns towards the removal of performance antipatterns.

The approach is illustrated in Figure 1. It starts with a software artifact (i.e., a design model or the current version of the application code) that does not fulfill some performance requirements (e.g., the response time of a service has to be less than 2 seconds). In order to verify if the latter ones are met, it is necessary to conduct a performance analysis, by means of a specific performance model as a Queuing Network (in case of a design model) or by monitoring the running application (in case of the current version of the application code). We assume this analysis to be implicit in our approach, because it is not the focus of this paper.

While performance requirements are not met (otherwise the process stops), the process starts. Its core consists of a performance antipatterns detection phase followed by a design patterns ranking procedure that drives the choice of the pattern to apply for removing a certain antipattern.

The *Performance Antipatterns* knowledge is represented by the formal representation that we have provided in our previous work, where we associated a first-order logical formula in conjunctive normal form to each antipattern, expressing a set of conditions under which it occurs [5]. Each logical predicate contains literals, each composed by a metric (namely F) and a threshold (namely Th). Metrics have to be extracted from the software artifact (*Metrics Calculation*), and we distinguish design metrics (e.g., the number of messages generated by a component) and performance metrics (e.g., hardware nodes utilization). Such metrics are compared to thresholds, whose values have to be set (*Thresholds Binding*), e.g. by means of some heuristics [5]. We also considered range of values around such heuristics, but the main issue was to set the suitable width to capture the actual bad practices [1,2].

We have then moved a step ahead by defining thresholds' lower and upper bounds (namely Th_{LB} and Th_{UB}, respectively), in order to work into a fuzzy context (*Fuzzy Thresholds*). Moreover, we have defined an approach for assigning an occurrence probability to each performance antipatterns, and we can rely on that to drive the choice of the most suitable antipattern to remove for requirements fulfillment [3]. Such approach represents the basis which this paper is grounded on.

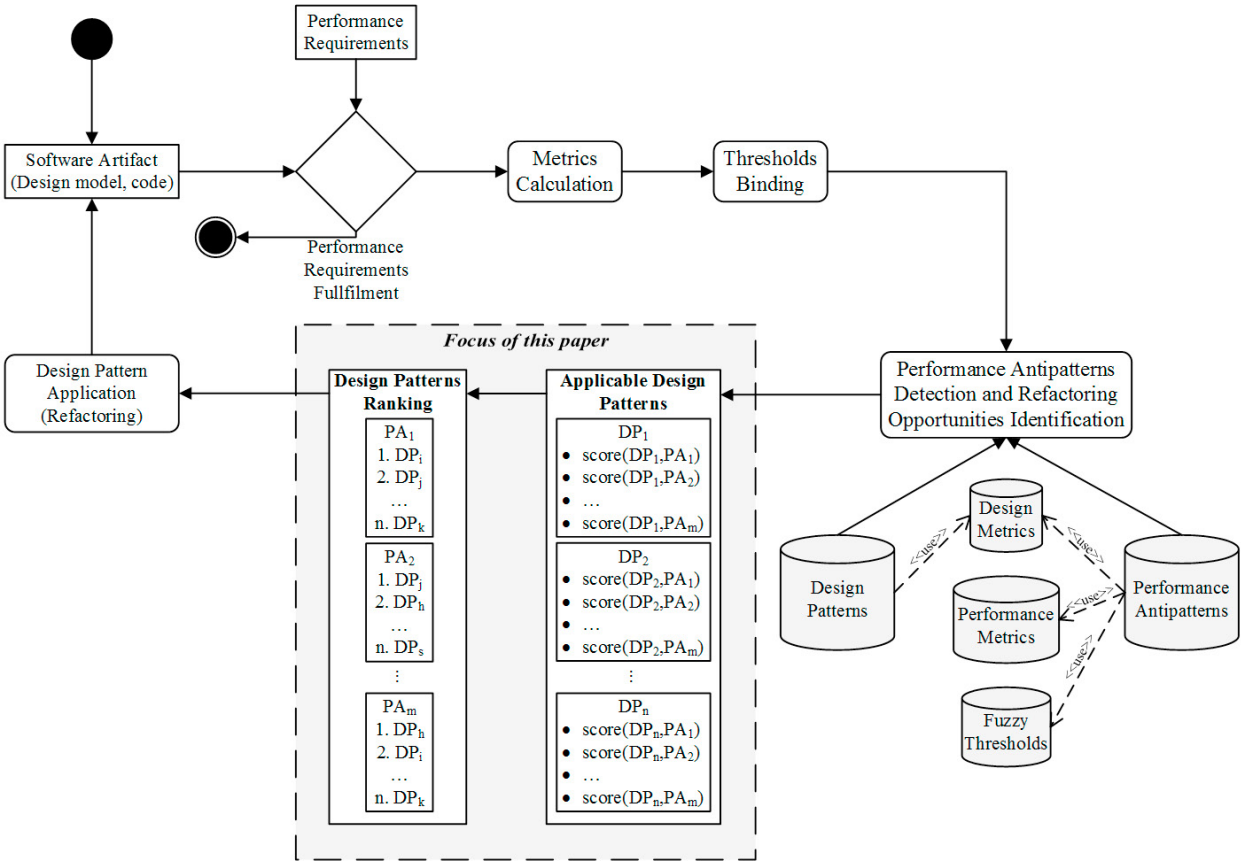

Fig. 1: Proposed approach for antipattern-based software refactoring driven by design patterns application.

Together with performance antipatterns detection, also applicable design patterns that remove antipatterns are identified (i.e., Refactoring Opportunities). In general terms, each design pattern influences a (possible empty) set of performance antipattern metrics, with the aim of overcoming the corresponding thresholds. This knowledge is held by software designers, which are in charge of defining it by "binding" antipattern metrics to design patterns.

A design pattern may be applied to zero or more performance antipatterns (*Applicable Design Patterns*), which are identified by matching the set of design metrics that the former influences with the ones involved in the antipattern specifications. The design pattern application is aimed at falsifying (e.g., increasing a metric such that it goes over the corresponding upper bound threshold) one or more predicates of the performance antipattern logical formula. In order to drive the choice of the design pattern to apply, a set of *score*s is assigned to each pattern (a score for each performance antipattern), which represents the probability that its application to the software artifact would lead to the removal of performance antipatterns. Once each design pattern has its own set of scores, they are ordered by decreasing scores, thus producing a list of the most suitable applicable design patterns for each performance antipattern. Scores calculation and the subsequent ranking are the focus of this paper, and they are detailed in the following sections.

The envisioned approach ends with the application of the chosen design pattern to the software artifact (*Refactoring*). A further performance analysis is needed on the obtained software artifact for verifying if performance requirements are met, and a new process iteration may eventually start.

Note that the application of a design pattern might introduce performance flaws, as for example the Blob performance antipattern [10]. In this paper we do not face this challenge, because it needs a deep analysis. However, we leave this as a future work.

3. Design Patterns Ranking

As stated before, system metrics (namely F) are compared to thresholds (namely Th), thus composing literals of performance antipatterns' formulae, in the form "$F\ op\ Th$", where $op \in \{>, \geq, =, \leq, <\}$. In general, the application of a design pattern affects one or more system metrics, thus changing their original values. In the remaining of this paper, any changed system metric value is referred as F'.

Our design pattern ranking procedure associates a score to each literal of each clause of detected performance antipattern's formulae. A literal score represents the probability for the considered literal to be falsified by applying a certain design pattern. Literals scores are then combined accordingly to the logical formula of detected antipatterns to produce design pattern scores.

By considering: DP as the design patterns knowledge, PA as the performance antipatterns knowledge, $C(pa)$ with $pa \in PA$ as the set of all the clauses in pa's logical formula, and $L(c)$ with $c \in C(pa)$ as the set of all the literals in the clauses of pa's logical formula, we define in the following the *score* of a design pattern $dp \in DP$, depending on the operator within a performance antipattern literal l:

(i) $F \geq Th$

$$score(dp, l) = \begin{cases} 1 & F' < Th_{LB} \\ \frac{Th_{UB} - F'}{Th_{UB} - Th_{LB}} & Th_{LB} \leq F' < Th_{UB} \\ 0 & F' \geq Th_{UB} \end{cases} \quad (1)$$

(ii) $F > Th$

$$score(dp, l) = \begin{cases} 1 & F' \leq Th_{LB} \\ \frac{Th_{UB} - F'}{Th_{UB} - Th_{LB}} & Th_{LB} < F' \leq Th_{UB} \\ 0 & F' > Th_{UB} \end{cases} \quad (2)$$

(iii) $F \leq Th$

$$score(dp, l) = \begin{cases} 1 & F' > Th_{UB} \\ \frac{F' - Th_{LB}}{Th_{UB} - Th_{LB}} & Th_{LB} < F' \leq Th_{UB} \\ 0 & F' \leq Th_{LB} \end{cases} \quad (3)$$

(iv) $F < Th$

$$score(dp, l) = \begin{cases} 1 & F' \geq Th_{UB} \\ \frac{F' - Th_{LB}}{Th_{UB} - Th_{LB}} & Th_{LB} \leq F' < Th_{UB} \\ 0 & F' < Th_{LB} \end{cases} \quad (4)$$

In case (v) $F = Th$ (i.e., $Th_{LB} \leq F \leq Th_{UB}$ due to Th fuzziness), $score(dp, l)$ is calculated using Equation (1) if $F' < F$, Equation (3) if $F' > F$.

By considering literals falsifications as stochastically independent events, the score of a design pattern dp with respect to a performance antipattern pa is defined as in Equation (5), where $C(pa)$ is the set of all the clauses in pa's logical formula and $L(c)$ is the set of the literals in $c \in C(pa)$ ([1]).

$$score(dp, pa) = \frac{\sum_{c \in C(pa)} \prod_{l \in L(c)} score(dp, l)}{|C(pa)|} \quad (5)$$

This score allows to introduce an ordering aimed at understanding which design antipattern is better to apply, among the applicable ones, for removing a certain performance antipattern (e.g., chosen by means of our previous work[3]).

[1] Considering literals falsifications as independent events is a simplification that we assume since this is a preliminary approach. In fact, a design pattern might improve a metric value while worsening other metric(s) in the logical formula.

4. Preliminary Validation

In this section we show a preliminary validation of our software refactoring approach, giving evidence of how the proposed ranking scheme works over three design patterns, with the aim of removing the Empty Semi-Trucks performance antipattern (EST)[10]. Given the preliminary nature of this validation, we left as future work the validation of our approach with respect to more performance antipatterns occurring at the same time. However, the choice of the performance antipattern that has to be targeted first among the detected ones might be supported by our previous work performance antipatterns ranking in fuzzy contexts[3]. This shall maintain our design patterns ranking scheme valid, although a refinement of the latter shall be needed in order to take into account the fact that different performance antipatterns may share some metrics and/or thresholds, hence applying a design pattern to remove a certain antipattern may affect other antipattern occurrences.

Performance Antipatterns Knowledge

The formalization of the EST[5] is reported in Equation (6), where $sw\mathbb{E}$ and $\mathbb{S}$ represent the set of all software entities and services, respectively. In detail, the EST formula contains two clauses, composed by one and two literals, respectively. The unique literal of the first clause verifies if, in certain service S, a software entity swE_x sends a number of remote messages (i.e., passing over a network), namely $F_{remMsgs}(swE_x, S)$, which is greater than a threshold $Th_{remMsgs}$. The second clause has two literals: the first one verifies if the usage of the network, i.e. $F_{netUtil}(P_{swE_x}, swE_x)$ where P_{swE_x} is the node which swE_x is deployed to, is below a threshold $Th_{netUtil}$; the second literal, which refers to $F_{remInst}(swE_x, S)$, verifies if the number of remote software entities involved in the service S is greater than a threshold $Th_{remInst}$.

$$\begin{aligned} EST = \exists swE_x \in \mathrm{sw}\mathbb{E}, S \in \mathbb{S} \mid \\ F_{remMsgs}(swE_x, S) \geq Th_{remMsgs} \wedge \\ \wedge (F_{netUtil}(P_{swE_x}, swE_x) < Th_{netUtil} \vee \\ F_{remInst}(swE_x, S) \geq Th_{remInst}) \end{aligned} \tag{6}$$

Design Patterns Knowledge

In order to remove the EST, we consider three suitable design patterns, namely Session Façade (SF) ([2]), Batching (Batch)[4], and Aggregate Entity (AE)[8].

Session Façade (SF)

The application of SF restructures the communication between swE_x and the involved remote software entities, with the main goal of reducing the number of messages sent over the network. In particular, two new software entities (namely façades) are introduced: one is local to swE_x and the other one is local to the involved remote entities. The former accumulates messages from swE_x and forwards a single message to the latter, which, in turn, delivers the original messages to the corresponding recipients.
As consequences of the application of SF:

1. The number of remote messages is reduced to 1.
2. The network utilization should augment, due to the overhead o needed to deliver each data to the corresponding remote software entity after message assembling.
3. The number of remote software entities exchanging messages over the network is reduced to 1, i.e. the façade local to the involved remote software entities.

[2] Sun Microsystems. Session Facade. Address: http://developer.java.sun.com/developer/restricted/patterns/SessionFacade.html

Batching design pattern (Batch)

Similarly to SF, Batch restructures the communication between swE_x and the involved remote entities. In order to reduce the number of messages sent over the network, all the messages between swE_x and the remote entities are encapsulated in a unified batch program. The latter is remotely executed and the result is returned to swE_x.
As consequences of the application of Batch:

1. The number of remote messages is reduced approximately to $F_{remInst}(swE_x, S)$, due to the unique batch program that is introduced for each remote instance.
2. The network utilization should augment, i.e. from n messages having a mean size of x bytes to 1 message having size $x+o$, where o is the overhead needed to transfer the batch program and to deliver each data to the corresponding remote software entity.
3. The number of remote instances remains unchanged.

Aggregate Entity Pattern (AE)

AE introduces a kind of façade between swE_x and the involved remote entities, locally to these latter ones. By exploiting the Bridge design pattern[6], all the messages exchanged among swE_x and the remote entities pass through the façade, resulting in a reduction of involved remote entities.
As consequences of the application of AE:

1. The number of remote messages remains unchanged.
2. The network utilization remains (almost) unchanged, because just the recipient changes, i.e., façade.
3. The number of remote instances is reduced to 1.

Design Patterns Ranking Example

For sake of our preliminary validation, we assume the following sample EST metric values and thresholds' bounds (LB and UB stand for Lower and Upper Bound, respectively).

- $F_{remMsgs} = 15$, $Th^{LB}_{remMsgs} = 8$, $Th^{UB}_{remMsgs} = 13$.
- $F_{netUtil} = 0.36$, $Th^{LB}_{netUtil} = 0.6$, $Th^{UB}_{netUtil} = 0.2$.
- $F_{remInst} = 11$, $Th^{LB}_{remInst} = 9$, $Th^{UB}_{remInst} = 3$.

Table 1 reports the scores of the three considered design patterns with respect to each EST literal: the first column reports the considered design patterns; the second, third, and fourth columns report new EST metric values and their scores (inside parenthesis) for the corresponding EST literals; the last column reports design patterns scores for the whole formula as defined by the equations in Section 3.

Table 1: Design patterns ranking example.

dp	New EST metrics values and scores			score(dp,EST)
	$F'_{remMsgs}$	$F'_{netUtil}$	$F'_{remInst}$	
SF	1 (1)	0.65 (1)	1 (1)	1
Batch	11 (0.4)	0.58 (0.95)	11 (0)	0.2
AE	15 (0)	0.16 (0)	1 (1)	0

As can be seen from Table 1, with respect to sample values and bounds, SF has the maximum possible score (i.e., 1), thus resulting in being the best design pattern to apply among the other two ones. This holds, in general, for each assignment of the EST literals, because such design pattern is aimed at falsifying all the EST literals.

Batch has a score greater than zero because it may be useful to apply it even in the considered case where all the EST literals are true. However, with the given assignment, its score is close to zero because both $F'_{remMsgs}$ and $F'_{netUtil}$ are between their thresholds bounds.

Note that even in case $F'_{netUtil}$ had been falsified, the whole second clause of the EST would not, because $F'_{remInst} = F_{remInst}$, thus the corresponding EST literal remains true. However, the Batching design pattern influences $F_{remMsgs}$,

hence there is a non-zero probability to falsify the corresponding literal. If the literal $F_{remInst} \geq Th_{remInst}$ had been false in the EST assignment, then the Batching design pattern would have been greater than 0.2.

Finally, AE score is zero. This is because, with the considered EST assignment, falsifying the literal $F_{remInst} \geq$-$Th_{remInst}$ is not sufficient to falsify the second EST clause. However, differently from Batch, AE does not influence $F_{remMsgs}$, thus resulting unuseful at all for such EST assignment. If the literal $F_{netUtil} < Th_{netUtil}$ had been false in the EST assignment, then the Aggregate Entity would have been a score greater than zero.

Summarizing, from our preliminary validation, we can observe that the scoring procedure strictly depends on the effect that a design pattern application induces on each antipattern literal, with respect to the original literals assignment. This means, for example, that:

- Since SF is the unique design pattern that always falsifies the literal $F_{remMsgs} \geq Th_{remMsgs}$ (i.e., the first EST clause), it is the most suitable one to apply for the EST, due to the fact that such literal is in AND with the rest of the antipattern formula.
- Conversely, the application of AE never brings to falsifying the literal $F_{remMsgs} \geq Th_{remMsgs}$. Hence, AE is not the most suitable design pattern to apply for the EST, due to the fact that such literal is in AND with the rest of the antipattern formula. AE score will be thus determined by its effect on the literals $F_{netUtil} < Th_{netUtil}$ and $F_{remInst} \geq Th_{remInst}$, which are in the second EST clause. The scores for such literals, i.e., 0 and 1, are multiplied each other, thus returning zero for the whole second clause. This is intuitive, since falsifying one literal among a set of literals in OR within the same clause is weaker than falsifying one literal among a set of literals in AND within the same clause (e.g., as it happened for SF with respect to the literal $F_{remMsgs} \geq Th_{remMsgs}$).
- Similarly to AE, Batch is not the most suitable design pattern to apply for the EST, due to the fact that it does not certainly falsify the literal $F_{remMsgs} \geq Th_{remMsgs}$. However, differently from AE, such clause may be falsified with a certain probability (i.e., 0.4). Since the probabilities to falsify the second EST clause is close to 1 for both Batch (i.e., 0.95) and AE (i.e., 1), the fact that the literal $F_{remMsgs} \geq Th_{remMsgs}$ may be falsified with a certain probability greater than zero (i.e., 0.4) is sufficient to obtain a score for Batch which is greater than the AE score.

5. Related Work

Since our approach can be used both in model-based and code-based software artifacts, we consider as related work approaches working both on models and code. At the best of our knowledge the following ones are the most suitable related works with respect to our refactoring approach.

Mani et al.[9] considered performance effects caused by design pattern application in a specific architectural pattern (i.e., Service Oriented Architecture). Our approach differs from Mani's one because we do not target a specific architectural style and we give to designers a list of design patterns that are ordered with respect to their score.

Stephan et al.[14] presented an approach for detecting design patterns and antipatterns into the Model-Driven Engineering workflow. They have used the *Model Clone Detection* for discovering (anti)pattern occurrences. Differently from them, our approach grounds on performance antipattern specifications expressed as first-order logical formulas[5], which are at a higher level of abstraction with respect to models. Hence, our approach supports both MDE and code developing (e.g., agile) contexts.

Jafaar et al.[7] evaluated the impact of design pattern with respect to changes and faults. They focus on analyzing intra-class relations, which can be broken by design pattern application. Differently from this work, our approach focuses on non-functional attributes evolution (i.e., performance regression). Thanks to our approach, the designer can consciously choose which design pattern is better for removing her own performance issues.

Moha et al.[15] presented the *SODA* framework to define and discover antipatterns in Service-Based Systems. SODA introduces a DSL for describing (anti)patterns and, subsequently, an automatic (anti)pattern detection process can be activated. Differently to Moha's approach, we did not create a DSL for describing antipatterns, but we use first-order logical formulas to this aim. Another difference is that we target performance antipatterns rather then design ones, and we relate them to design patterns by means of our ranking procedure.

The approach by Stoianov and Sora[11] is the most similar to ours. Both are grounded on a logic-based antipatterns representation. However, in the work by Stoianov and Sora, Prolog predicates are defined and they represent

(anti)pattern verification rules, whereas in this paper we address performance antipattern rather than design ones, and we go beyond the simple detection process by providing a design pattern ranking procedure that drives the refactoring choice.

6. Conclusion and Future Work

In this paper we have introduced an approach for antipattern-based refactoring of software systems, driven by design patterns application. The approach is focused on refactoring software artifacts (i.e., models, code) by applying design patterns, with the aim of removing possible occurring performance antipatterns. The choice of a design pattern to apply is supported by a ranking procedure that we have described in this paper. We have also provided a preliminary validation of our approach, showing how the ranking procedure works over three design patterns for removing the Empty Semi-Trucks performance antipattern.

In the future, we intend to enlarge the specter of considered performance antipatterns and design patterns. This would allow to refine our approach by taking into account the fact that: (i) Different performance antipatterns may share some metrics and/or thresholds, hence applying a design pattern to remove a certain antipattern may affect other antipattern occurrences; (ii) Besides the removal of a certain performance antipattern a design pattern application introduces new antipatterns.

Furthermore, introducing performance patterns could provide antipattern removal solutions that would be more "performance-oriented", due to the fact that their definitions not only involve design metrics, but also performance ones. This would fasten performance requirements fulfillment, which is the main goal in our context.

Finally, developing tool support and conducting experimentation on several software artifacts from real case studies would allow to provide an extensive validation of our approach.

References

1. Arcelli D, Cortellessa V, Trubiani C. Experimenting the Influence of Numerical Thresholds on Model-based Detection and Refactoring of Performance Antipatterns. In: *ECEASST* 2013. **59**.
2. Arcelli D, Cortellessa V, Trubiani C. Influence of numerical thresholds on model-based detection and refactoring of performance antipatterns. In: *First Workshop on Patterns Promotion and Anti-patterns Prevention*. 2013.
3. Arcelli D, Cortellessa V, Trubiani C. Performance-Based Software Model Refactoring in Fuzzy Contexts. In: Egyed A, Schaefer I, editors. *Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Springer; 2015. p. 149-164.
4. Ballesteros FJ, Kon F, Patino M, Jiménez R, Arévalo S, Campbell RH. Batching: A design pattern for efficient and flexible client/server interaction. In: *Transactions on Pattern Languages of Programming* 2009. **1**:1.
5. Cortellessa V, Di Marco A, Trubiani C. An approach for modeling and detecting Software Performance Antipatterns based on first-order logics. In: *Journal of Software and Systems Modeling* 2012. **13**:1.
6. Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-oriented Software. Boston: Addison-Wesley Longman Publishing Co., Inc.; 1995.
7. Jaafar F, Guéhéneuc YG, Hamel S, Khomh F, Zulkernine M. Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults. In: *Empirical Software Engineering* 2015. **21**:3.
8. Larman C. Enterprise JavaBeans 201: The Aggregate Entity Pattern. In: *Software Development Magazine* 2000. **8**:4
9. Mani N, Petriu DC, Woodside M. Studying the Impact of Design Patterns on the Performance Analysis of Service Oriented Architecture. In: *37th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2011, Oulu, Finland, August 30 - September 2, 2011*. IEEE Computer Society; 2011. p. 12-19.
10. Smith CU, Williams LG. More New Software Antipatterns: Even More Ways to Shoot Yourself in the Foot. In: *29th International Computer Measurement Group Conference, December 7-12, 2003, Dallas, Texas, USA, Proceedings*. Computer Measurement Group; 2003. p. 717-725.
11. Stoianov A, Sora I. Detecting patterns and antipatterns in software using Prolog rules. In: *2010 International Joint Conference on Computational Cybernetics and Technical Informatics*. IEEE Computer Society; 2010. p. 253-258
12. Sun Microsystems. Session Facade. Address: http://developer.java.sun.com/developer/restricted/patterns/SessionFacade.html. 2001
13. Smith CU, Williams LG. New Book - Performance solutions: a practical guide to creating responsive, scalable software. In: *27th International Computer Measurement Group Conference, Anaheim, CA, USA, December 2-7, 2001*. Computer Measurement Group; 2001. p. 355-358.
14. Stephan M, Cordy JR. Identifying Instances of Model Design Patterns and Antipatterns Using Model Clone Detection. In: *Proceedings of the Seventh International Workshop on Modeling in Software Engineering*. Piscataway: IEEE Press; 2015. p. 48-53.
15. Moha N, Palma F, Nayrolles M, Conseil BJ, Guéhéneuc YG, Baudry B, Jézéquel JM. Specification and Detection of SOA Antipatterns. In: Liu C, Ludwig H, Toumani F, Yu Q. *Service-Oriented Computing - 10th International Conference, ICSOC 2012, Shanghai, China, November 12-15, 2012. Proceedings*. Springer; 2012. p. 1-16.