Performance Analysis Patterns for Requirements Analysis

Azadeh Alebrahim

Paluno - The Ruhr Institute for Software Technology, University of Duisburg-Essen, Germany
azadeh.alebrahim@paluno.uni-due.de

Abstract. Many problems might arise when performance requirements are not taken into account from the beginning of software development. Architectural solutions such as performance patterns represent design decisions on the architecture level that might constrain quality requirements significantly. Knowledge which is gained in the solution space, for example from performance patterns, should be reflected in the requirements engineering to obtain sound architectures and correct requirements. We propose to reuse performance architectural patterns in the requirements engineering in a systematic manner to equip requirement models with performance solution approaches early in the software development process. To this end, we propose *performance analysis patterns*. The performance requirement models elaborated with performance analysis patterns can easily be transformed into a particular solution at the design level.

1 Introduction

Many software systems fail to achieve their quality objectives due to neglecting quality requirements at the beginning of the software development life cycle [1]. Problems such as loss of productivity, loss of revenues, loss of customers, cost overruns, etc. arise when software systems are constructed without having performance in mind [2]. Fixing such problems afterwards (by modifying the code) is costly or even hardly possible. The software after fixing such problems might be erroneous or might not perform as well as software which has been constructed under performance considerations [3]. Therefore, performance as one of the critical quality requirements to the success of a software system must be integrated from the beginning into software development to prevent performance problems.

Patterns describe solutions for commonly recurring problems in software development. They are defined for different software development phases, such as problem frames [4], architectural styles [5] for the architectural level and design patterns [6] for the design level. Most patterns provide information about the proposed solution including the intent, structure, participants, and collaborations, whereas the problem to be solved is illustrated only by scenarios and situations in which the specific pattern can be applied. The essence of the problem and its structure, which is important for being able to apply a pattern, is not provided.

In this paper, we propose patterns that can be applied during the requirements engineering phase to refine the requirement models. These patterns reuse the knowledge gained in the design phase (solution space) to equip the requirement models (problem

space) with solution approaches early in the software development process. We leverage existing performance patterns for the architecture level as solutions for performance problems in order to adapt them for the requirements analysis phase. We call the adapted patterns *performance analysis patterns*. That is, we do not only elicit and model performance requirements, but also provide solution approaches for these requirements.

The benefit of the proposed performance analysis patterns is manifold. First, they provide guidance for refining performance problems located in the problem space Thus, they nudge software engineers to think about performance problems and early solution approaches as early as possible in the software development. Second, the elaborated performance requirement models can easily be transformed into a particular solution at the design level. Thus, it bridges the gap between performance problems and performance solutions. Third, it supports less experienced software engineers in applying solution approaches early in the requirements engineering phase in a systematic manner.

As a basis for requirements analysis, we use the problem frames approach [4]. We use the problem frames approach, because 1) it allows decomposing the overall software problem into subproblems, thus reducing the complexity of the problem, 2) it makes it possible to annotate problem diagrams with quality requirements, such as performance requirements, 3) it enables various model checking techniques, such as requirements interaction analysis and reconciliation [7] due to its semi-formal structure, and 4) it supports a seamless transition from requirements analysis to architectural design (e.g. [8]).

The remainder of this paper is organized as follows. Section 2 introduces the example CoCoME. We give a brief description of problem frames in Section 3. Section 4 describes the performance analysis patterns for the requirements analysis phase. We describe how to use the proposed patterns in Section 5. Section 6 presents related work and Section 7 concludes the paper and points out suggestions for future work.

2 Application Example CoCoME

To illustrate the applicability of our approach, we use the Common Component Modeling Example (CoCoME) [9], which is a trading system to be deployed in supermarkets for handling sales. Different tasks can be performed using this trading system, such as scanning the products using a *bar code scanner*, paying by cash or credit card taking place at a single *cash desk*, logging sale information and updating the stock inventory as well as administrative tasks such as generating reports or ordering products. A *store* contains several cash desks, called a *cash desk line*. Each cash desk is connected to a *store server* which itself is connected to a *store client*. Detailed description of the example CoCoME is described in [9].

The functionality of the trading system CoCoME is described as use cases, each of which is concerned with several activities that can be divided into several functional requirements. By means of the requirement *R5*, we illustrate the use of performance analysis patterns. The requirement *R5* requires *logging of sale information and updating the stock of inventory*.

3 Background

This section outlines the basic concepts of problem frames as a requirements engineering approach. Problem frames are a means to understand, describe, and analyze software development problems proposed by Michael Jackson [4]. A problem frame is described by a *frame diagram*, which basically consists of *domains*, *interfaces* between them, and a *requirement*. The task is to construct a *machine* (i.e., software) that improves the behavior of the environment (in which it is integrated) in accordance with the requirements. A problem diagram represents an instance of a problem frame.

We use a UML-based enhancement of problem frames, which is extended by a specific UML profile for problem frames (UML4PF) proposed by Hatebur and Heisel [10]. The software to be developed (*machine* in problem frames terminology) is represented by the stereotype ≪`machine`≫. Jackson distinguishes the domain types biddable domains (represented by the stereotype ≪`BiddableDomain`≫) that are usually people, causal domains (≪`CausalDomain`≫) that comply with some physical laws, and lexical domains (≪`LexicalDomain`≫) that are data representations. Figure 1 shows the problem diagram for the client side of the requirement *R5* which is concerned with logging sale information and updating the stock of inventory. It describes that the machine domain *LogUpdateStockClient* obtains sale information from the lexical domain *SaleInformation* and sends it through the domain *Network* to the server.

In problem diagrams, *interfaces* connect domains and they contain *shared phenomena*. Shared phenomena may, e.g., be events, operation calls or messages. They are observable by at least two domains, but controlled by only one domain, as indicated by "!". The notation $SI!\{contentOfSI\}$ (between the domains *LogUpdateStockClient* and *SaleInformation*) in Figure 1 means that the phenomenon *contentOfSI* is controlled by the domain *LogUpdateStockClient*. When we state a requirement we want to change something in the world with the machine to be developed. Therefore, each requirement expressed by the stereotype ≪`requirement`≫ constrains at least one domain. This is expressed by a dependency from the requirement to a domain with the stereotype ≪`constrains`≫. A requirement may refer to several domains in the environment of the machine. This is expressed by a dependency from the requirement to a domain with the stereotype ≪`refersTo`≫. The requirement *R5-1* in Figure 1 constrains the domain *Network*. It refers to the domain *SaleInformation*.

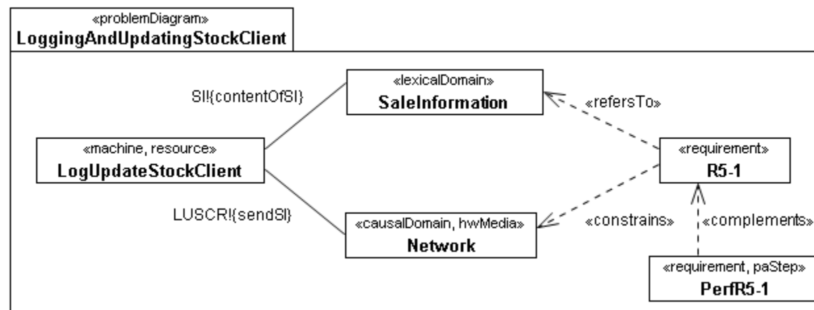

Fig. 1. Problem diagram for logging sale information and updating the stock of inventory

To be able to annotate problem diagrams with quality requirements, we extended the UML profile for problem frames [8]. It enables us to complement functional requirements with quality requirements. A dependency from a quality requirement to a functional one is expressed with the stereotype ≪complements≫. To specify and model quantitative performance annotations, we use the UML profile MARTE [11]. In Figure 1, the performance requirement *PerfR5-1* complements the functional requirement *R5-1*.

4 Performance Analysis Patterns

In this section, we propose an adaptation for existing performance patterns from the literature [12,13]. Our adaptation allows the use of such performance patterns in the requirements analysis phase for the analysis of performance problems.

The adaptation encompasses a generic template for describing performance patterns textually. It has to be instantiated for each performance pattern explicitly. In addition to the template, we provide a problem frame that describes the generic structure of the problem. Then, we describe the solution approach by introducing a new problem frame that describes the generic solution structure. We call such performance patterns adapted in order to be used for requirements analysis *performance analysis patterns*.

4.1 Template for Performance Analysis Patterns

According to Gamma et al. [6], a pattern describes a recurring problem and the core of the solution for that problem. Architectural patterns can be applied to meet quality requirements. They represent design decisions on the architecture level that affect the achievement of quality requirements significantly. A performance pattern conveys essential performance-specific information and principles for facilitating the reuse of performance knowledge.

We make use of a collection of existing performance patterns taken from current literature [12,13] in order to adapt them for the requirements analysis phase. To this end, we propose a template that represents the concepts contained in such patterns in a way that they can be used in a higher abstraction level than architecture level, namely in the requirements analysis phase. Our proposed template illustrated in Table 1 is inspired by the template for design patterns from Gamma et al. [6]. We modified the template for design patterns to describe and represent performance-specific information. Our template contains additional information for modeling a performance analysis pattern using the UML profiles UML4PF and MARTE. The proposed template allows to define new performance analysis patterns according to this structure as well.

The fields *problem* and *applicability* describe when the pattern can be applied. They represent the pre-conditions for the pattern at hand. The fields *solution*, *collaboration*, *benefits*, and *consequences* describe the solution including its elements, their relationships, and their behavior. They represent the post-conditions for the pattern at hand.

Table 1. Template for performance analysis patterns

1) Name	Name of the pattern
2) Description	Brief description of the pattern
3) Also known as	Other well-known names for the pattern, if any
4) Problem	Situation and structure of the problem
5) Applicability	Conditions under which the pattern can be applied
6) Solution	Structure of the solution using stereotypes from UML4PF and MARTE
7) Collaboration	Behavior Description of solution elements
8) Benefits	Benefits of applying the pattern
9) Consequences	Consequences and hints to be considered when applying the pattern
10) Related patterns	Another pattern related to the pattern

4.2 Performance Analysis Patterns

The original performance patterns only describe the principle of the solution. They do not provide any structure of the problem. We provide the structure of the problem as a specific problem frame in addition to the textual description in the template. Note that the structure of the problem for all performance patterns presented in this paper is similar (see the field *problem* in the template). The reason is that the lack of resources is the essence of most performance problems. This is the case when more requests have to be processed at the same time than the resources can process. Hence, there is only one problem frame describing the generic problem structure for all patterns. Nevertheless, the conditions under which the patterns can be applied are different (see the field *applicability* in the template).

We adapt three existing performance patterns [12,13], namely *First Things First*, *Flex Time*, and *Load Balancer* for requirements analysis. These patterns provide solutions for the problem situation where an overload of the system is expected. Figure 2 shows the problem frame describing the *generic problem structure*. Domains contained in this problem frame are:

- One *Machine* domain. It represents a resource expressed by the stereotype ≪resource≫ responsible for responding to the requests. The resource is expected to be the bottleneck which cannot complete all inbound requests (see the stereotype ≪bottleneck≫ in the generic problem frame in Figure 2). It is also possible that the bottleneck is a hardware resource which is consumed by the *Machine* domain. In such a case, the hardware resource is modeled explicitly.

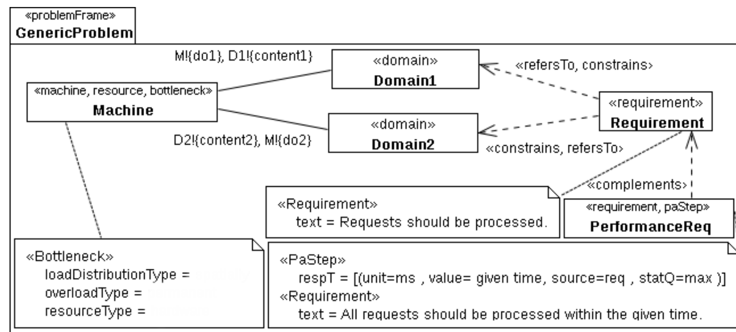

Fig. 2. Problem frame describing generic problem structure

Table 2. First Things First Pattern

1) Name	First Things First (FTF)
2) Description	FTF ensures that the most important tasks will be processed if not every task can be processed.
3) Also known as	-
4) Problem	A temporary overload of inbound requests is expected. This situation may overwhelm the processing capacity of a specific resource (see the generic problem frame in Figure 2).
5) Applicability	FTF pattern is only applicable when there is a temporary overload. That is, the attribute *overloadType* of the stereotype ≪`bottleneck`≫ in Figure 2 should have the value *temporary*.
6) Solution	The solution uses the strategy of prioritizing tasks and performing the important tasks with high-priority first. A new machine is introduced that takes the responsibility for prioritizing the tasks and assigning them to corresponding domains.
7) Collaboration	When requests are issued, they arrive through *Domain1* at the newly introduced machine domain *FTF*, which takes the responsibility to prioritize the requests and forward them to the corresponding machine that performs the requests using the domain *Domain2*. Note that there exists only one machine domain *Machine*.
8) Benefits	FTF reduces the contention delay for high-priority tasks.
9) Consequences	In the case of a permanent overload, applying this pattern would cause the starving of low-priority tasks.
10) Related patterns	LB pattern can be used to improve the processing capacity if the overload is not temporary.

Table 3. Flex Time Pattern

1) Name	Flex Time (FT)
2) Description	FT moves the load to a different period of time where the inbound requests do not exceed the processing capacity of the resource.
3) Also known as	-
4) Problem	An overload of the system is expected. The inbound requests exceed the processing capacity of a specific resource (see the generic problem frame in Figure 2).
5) Applicability	FT is only applicable when some tasks can be performed at a different period of time. That is, the attributes *loadDistributionType* and *overloadType* of the stereotype ≪`bottleneck`≫ in Figure 2 have the values *temporally* and *permanent*.
6) Solution	The solution uses the strategy of spreading the load at a different period of time. A new machine is introduced that takes the responsibility for modifying the processing time of the tasks and assigning them to corresponding domains for processing in the specified time.
7) Collaboration	When requests are issued, they arrive through *Domain1* at the newly introduced machine *FTF/FT/LB*, which takes the responsibility to spread the requests at a different period of time to be processed by the corresponding machine using the domain *Domain2*. Note that there exists only one machine domain *Machine*.
8) Benefits	FT pattern reduces the load of the system by spreading it temporally.
9) Consequences	The order of satisfying requirements will be changed. It has to be checked that this modification does not cause new bottlenecks.
10) Related patterns	LB pattern can be used to reduce the load if the tasks cannot be performed at a different period of time.

– One *Domain1* domain, which transmits the requests to the machine domain *Machine*.
– One *Domain2* domain, which represents the domain that might be required for processing the requests by the *Machine*.
– One *Requirement*, which describes the functional requirement to be satisfied. It requires the processing of the requests.
– One *PerformanceReq*, which describes the performance requirement to be satisfied. It requires the satisfaction of the functional requirement *Requirement* within a specific time.

We describe each pattern as one instance of the template given in Table 1. The instantiation of the template for the First Things First pattern is shown in Table 2. Tables 3 and 4 illustrate the instantiation of the template for the Flex Time pattern as well as for the Load Balancer pattern.

Table 4. Load Balancer Pattern

1) Name	Load Balancer (LB)
2) Description	LB pattern is used to distribute computational load evenly over two or more hardware resources.
3) Also known as	-
4) Problem	An overload of the system is expected. The inbound requests exceed the processing capacity of a specific hardware resource (see the generic problem frame in Figure 2).
5) Applicability	LB pattern is only applicable when the resource which is the bottleneck is a hardware resource, the overload is permanent, and the load can be spread spatially. That is, the attributes *loadDistributionType* and *overloadType*, and *resourceType* of the stereotype ≪`bottleneck`≫ in Figure 2 have the values *spatially*, *permanent*, and *hardware*.
6) Solution	The solution uses the strategy of spreading the load over several hardware resources.
7) Collaboration	When requests are issued, they arrive through *Domain1* at the newly introduced machine *FTF/FT/LB*, which takes the responsibility to forward the request to one corresponding machine which is free. The selected machine processes the request, creates a response using the domain *Domain2*, and sends the response. Note that there exists at least two machine domains of the same type.
8) Benefits	LB pattern reduces the load of the system by spreading it spatially.
9) Consequences	Efficient algorithm for allocating the requests to responders is required to ensure that the newly introduced *LBMachine* does not become the new bottleneck.
10) Related patterns	FT pattern can be used if the tasks can be performed at a different period of time. FTF pattern can be used when there is a temporary overload.

Figure 3 shows the problem frame describing the *generic solution structure*. It is a *composition frame* that composes several subproblems using a new introduced machine domain. We introduce the new machine domain *FTF/FT/LB* to compose several machine domains that are bottlenecks (see machine domain *Machine* in Figure 2) in order to prevent the overload for each single machine domain. There exist only one machine domain *Machine* for the performance analysis patterns *FTF* and *FT* and at least two machine domains *Machine* of the same type for the performance analysis pattern *LB*.

Domains contained in this problem frame are:

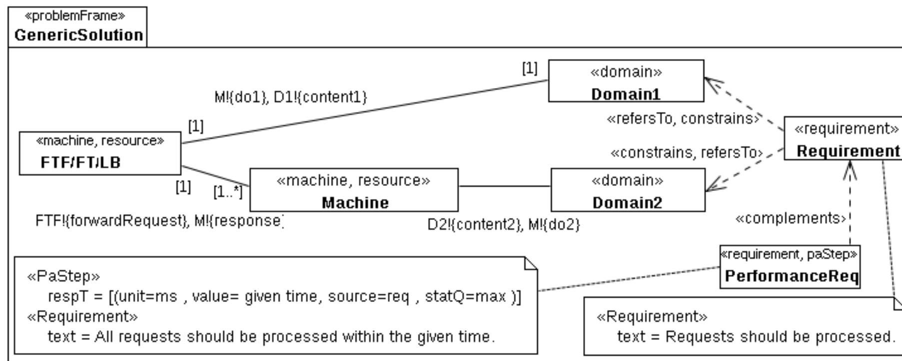

Fig. 3. Problem frame describing generic solution structure

- One domain *FTF/FT/LB* as a machine domain and as a resource with the stereotypes ≪machine≫ and ≪resource≫.
- At least one domain as machine domain and as resource (stereotypes ≪machine≫ and ≪resource≫) responsible for responding to the requests (see *Machine* in Figure 3).
- One domain *Domain1* with the stereotype ≪domain≫, which transmits the requests to the machine domain *Machine*.
- One domain *Domain2* with the stereotype ≪domain≫ required for processing the requests by the machine domain *Machine*.
- One functional requirement *Requirement* with the stereotype ≪requirement≫ to be satisfied by the machine domains *Machine* (at least one machine domain).
- One *PerformanceReq* with the stereotypes ≪requirement≫ and ≪paStep≫ to be satisfied by the machine domains *Machine* (at least one machine domain).

5 Selection and Application of Performance Analysis Patterns

In this section, we describe how the proposed patterns can be applied within our problem-oriented software development method [8]. To this end, we make use of the CoCoME example introduced in Section 2. By means of the requirement *R5*, we illustrate the use of performance analysis patterns. The requirement *R5* requires *logging of sale information and updating the stock of inventory*.

The starting point is a set of problem diagrams annotated with performance requirements. The problem diagram for the functional requirement *R5* has to be decomposed into two subproblems one for the client side and the other for the server side. Figure 1 illustrates the problem diagram for the client side. Also the corresponding performance requirement has to be decomposed into two performance requirements *PerfR5-1* and *PerfR5-2*. The second subproblem for logging and updating the stock server (*R5*) concerning the server side is shown in Figure 4.

Performance is concerned with the *workload* of the system and the available *resources* to process the workload [14]. The system has to process the requests (or sa-

Fig. 4. Problem diagram *LoggingAndUpdatingStockServer* annotated with performance-specific domain knowledge

tisfy the requirements) caused by the workload using available resources. We elicit and model the workload and resources of the system as *domain knowledge* [15] as illustrated in Figure 4. The workload is represented by the stereotype ≪`gaWorkloadEvent`≫. Resources are modeled using the stereotypes ≪`hwMedia`≫, ≪`hwProcessor`≫, ≪`storageResoure`≫, etc.

Now, subproblems containing potential performance problems (bottlenecks) have to be identified. This has to be supported by a performance analyst to analyze whether the processing capacity of existing resources (modeled as domain knowledge) suffices to satisfy performance requirements for each subproblem with regard to the existing workload and frequency of occurring problem diagrams. We identify the problem diagram *LoggingAndUpdatingStockServer* as a critical one. Therefore, we mark it as a potential bottleneck. After identifying performance problems, we select appropriate performance analysis patterns for those subproblems containing bottlenecks. Only when subproblems are valid instances of the *generic problem structure* (Figure 2), we can apply patterns to the subproblems. The subproblem *LoggingAndUpdatingStockServer* represents a valid instance of the problem frame describing the *generic problem structure*. We describe the instantiation of the subproblem *LoggingAndUpdatingStockServer*. It contains the following elements:
One machine domain *LogUpdateStockServer*, which is responsible for responding to the requests. The bottleneck is a hardware resource that is modeled explicitly (*CPU*).
One domain *Network*, which transmits the requests to the machine domain *LogUpdateStockServer*.
One domain *Log_Stock*, which is used by the machine domain *LogUpdateStockServer* for processing the request, namely for logging sale information.
One requirement *R5-2*, which describes the functional requirement.
One performance requirement *PerfR5-2*, which describes the performance requirement corresponding to the functional requirement *R5-2*.

A pattern can be applied to a subproblem being a valid instance of the problem frame describing the *generic problem structure* only if the subproblem fulfills all preconditions of the specific pattern given in the field *applicability* in the description of the pattern template. The type of the overload for the subproblem *LoggingAndUpdatingStockServer* is permanent (*overloadType=permanent*) and the subproblem can be moved to a different period of time (*loadDistributionType=temporally*) as shown in Figure 4. These attributes of the stereotype ≪`bottleneck`≫ correspond to the preconditions for the application of performance analysis patterns given in the pattern template in the field *applicability* of the *Flex Time* pattern. Table 5 provides an overview of performance analysis patterns and their selection criteria.

Hence, we select *Flex Time* pattern for applying to the subproblem *LoggingAndUpdatingStockServer* (see Figure 5).

Table 5. Performance analysis patterns and their selection criteria

	selection criteria		
	type of load distribution	type of overload	type of resource
First Things First (FTF)	spatially / temporally	temporary	software / hardware
Flex Time (FT)	temporally	permanent	software / hardware
Load Balancer (LB)	spatially	permanent	hardware

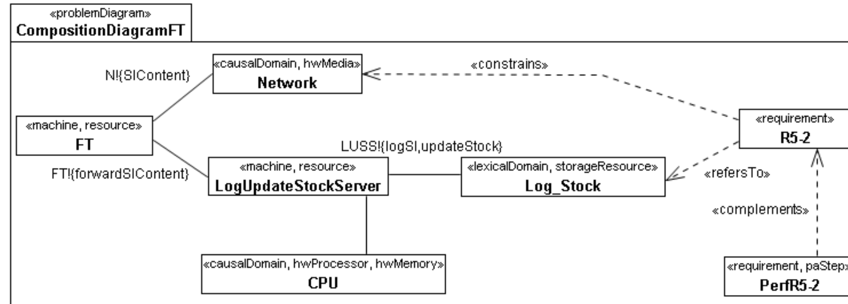

Fig. 5. Flex Time application to the subproblem *LoggingAndUpdatingStockServer*

It is a valid instance of the composition frame describing the *generic solution structure* illustrated in Figure 3. It contains the following elements:
One domain *FT* as a machine domain and as a resource with the stereotypes ≪machine≫ and ≪resource≫.
One domain *LogUpdateStockServer* as machine domain and as resource.
Onel domain *Log_Stock* used by the machine domain for processing the requests.
One domain *Network*, which is responsible for transmitting the requests.
One functional requirement *R5-2* to be satisfied by the machine *LogUpdateStockServer*.
One performance requirement *PerfR5-2*, which represents the performance requirement to be satisfied by the machine *LogUpdateStockServer*.

In this way, we successfully instantiated and applied the performance analysis flex time pattern in order to resolve the identified bottleneck. We only showed the instantiation of the graphical part of the pattern. The corresponding template can be easily filled out.

Benefits. Performance analysis patterns allow software engineers not only to think about potential performance problems as early as possible in the software development life cycle, but also to think about solution approaches resolving such problems. By exploring the solution space, we find appropriate solution mechanisms, which can be used for refining performance requirement models in the requirement engineering phase. However, performance analysis patterns do not replace the application of "classical" performance patterns. They do not really apply a solution. They only enforce the requirements analyst to think about the problem, its solution and the consequences of applying a specific solution as early as possible. This results in preparing the requirement models for applying the "classical" performance patterns later on in the design phase. Performance analysis patterns are located in the problem space aiming at structuring and elaborating performance problems while "classical" performance patterns are accommodated in the solution space aiming at solving performance problems. Hence, performance analysis patterns in the requirements engineering phase represent the counterpart to the "classical" performance patterns in the design phase.

6 Related Work

In a software development process, typically performance solutions are considered as architectural decisions to be made in the architecture and design phases. A number of approaches that contributed to software performance development, have focused on architectural solutions. Nevertheless, information and knowledge needed for dealing with performance issues, have to be collected and analyzed early in the software development process. Similar to our approach, Williams and Smith [16] explore the information needed to construct and evaluate performance models. This information has to be captured during the analysis and design process. They define a similar set of information required for early life cycle software performance engineering. They use the terms “execution environment” for resource capacity and “resource requirement” for resource utilization and resource type.

Bass et al. [17] analyze how architectural mechanisms such as fixed priority scheduling and caching help achieve performance as one specific quality requirement. They introduce three performance strategies resource allocation, resource arbitration, and resource use are introduced by Bass et al. [17]. Each strategy provides a set of performance tactics to achieve performance requirements. The tatic fixed priority scheduling that prioritizes processes to a fixed priority uses the strategy resource arbitration. The authors only describe two tactics fixed priority scheduling and caching. They do provide neither details regarding the structure and behavior of the tactics nor about the applicability of them.

Composition frames used in this paper are used in some other approaches as well. Laney et al. [18] propose a systematic approach to resolve inconsistencies in the problem frames. The authors introduce composition frames in order to deal with composing conflicting requirements. Composition frame are also used in an aspect-oriented method based on problem frames to restructure requirement models using security patterns [19]. Composition frames in this approach serve as a means to weave security aspects into the functional structures. We use composition frames in a different way. A composition frame in our paper represents the generic solution structure for performance analysis patterns. We use composition frames to apply performance analysis patterns to problem diagrams containing potential performance problems.

7 Conclusion

In this paper, we adapted three existing performance architectural patterns for the requirements analysis phase. The adaptation allows us to use such performance patterns for the analysis of performance problems. We have provided a systematic structure for performance analysis patterns consisting of a template containing performance-specific information for modeling performance analysis patterns using the UML profiles UML4PF and MARTE and a two-part graphical pattern describing the generic problem structure and the generic solution structure.

Using the application example CoCoME, we showed how the proposed performance analysis patterns can be used in a the requirements analysis phase based on problem frames to refine the requirement models and prevent potential performance problems.

We summarize the benefits of proposed performance analysis patterns as follows:

– Performance analysis patterns provide support for refining performance problems located in the problem space using the knowledge located in the solution space such as performance patterns . Thus, they nudge software engineers to think about performance problems and possible solution approaches as early as possible in the software development.
– less experienced software engineers are supported in applying solution approaches in terms of performance analysis patterns early in the requirements engineering phase in a structured manner.

In this paper, we only considered one performance metric, namely response time. In the future, we strive for dealing with other performance metrics that might be also relevant for an overload scenario such as throughput. Furthermore, we will extend the basis of our existing performance analysis patterns with more patterns in order to provide a catalog of performance analysis patterns for requirements analysis addressing more kinds of performance issues.

References

1. Chung, L., Nixon, B.A., Yu, E., Mylopoulos, J.: Non-functional requirements in software engineering. Klewer Academic (2000)
2. Smith, C.U., Williams, L.G.: Five steps to establish software performance engineering. In: Int. CMG Conference, 507–516 (2006)
3. Smith, C.U., Williams, L.G.: Software performance engineering: A case study including performance comparison with design alternatives. IEEE Trans. Software Eng. 19(7), 720–741 (1993)
4. Jackson, M.: Problem Frames. Analyzing and structuring software development problems. Addison-Wesley (2001)
5. Shaw, M., Garlan, G.: Software Aechitecture: Perspectives on an emerging discipline. Prentice Hall (1996)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley (1995)
7. Alebrahim, A., Choppy, C., Fabender, S., Heisel, M.: Optimizing functional and quality requirements according to stakeholders goals. In: System Quality and Software Architecture (SQSA). Elsevier, 75–120 (2014)
8. Alebrahim, A., Hatebur, D., Heisel, M.: A method to derive software architectures from quality requirements. In Thu, T.D., Leung, K., eds.: Proc. of the 18th Asia-Pacific Software Engineering Conf. (APSEC), IEEE Computer Society, 322–330 (2011)
9. Rausch, A., Reussner, R., Mirandola, R., Plasil, F.: The Common Component Modeling Example: Comparing Software Component Models. 1st edn. LNCS 5153, Springer (2008)
10. Hatebur, D., Heisel, M.: Making pattern- and model-based software development more rigorous. In: Proc. of 12th Int. Conf. on Formal Engineering Methods (ICFEM). LNCS 6447, Springer, 253-269 (2010)
11. UML Revision Task Force: UML Profile for MARTE: Modeling and Analysis of Real- Time Embedded Systems. http://www.omg.org/spec/MARTE/1.0/PDF
12. Ford, C., Gileadi, I., Purba, S., Moerman, M.: Patterns for Performance and Operability. Auerbach Publications (2008)

13. Smith, C.U.,Williams, L.G.: Performance solutions, a practical guide to creating responsive, scalable software. ADDISON WESLEY (2001)
14. Bass, L., Clemens, P., Kazman, R.: Software architecture in practice. Second edn. Addison-Wesley (2003)
15. Alebrahim, A., Heisel, M., Meis, R.: A structured approach for eliciting, modeling, and using quality-related domain knowledge. In: Proc. of the 14th Int. Conf. on Computational Science and Its Applications (ICCSA). LNCS 8583, Springer, 370–386 (2014)
16. Williams, L.G., Smith, C.U.: Information requirements for software performance engineering. In: Proc. of the Int. Conf. on Modeling Techniques and Tools for Computer Performance Evaluation, Springer, 86–101 (1995)
17. Bass, L., Klein, M., Bachmann, F.: Quality attributes design primitives. Technical report, Software Engineering Institute (2000)
18. Laney, R., Barroca, L., Jackson, M., Nuseibeh, B.: Composing requirements using problem frames. In: Proc. of the 4th int. conf. on Requirements Engineering, Press, 122–131 (2014)
19. Alebrahim, A., Tun, T.T., Yu, Y., Heisel, M., , Nuseibeh, B.: An aspect-oriented approach to relating security requirements and access control. In: Proc. of the CAiSE Forum. CEUR Workshop Proceedings, 15–22, CEUR-WS.org (2012)