Engineering the Performance of a Meta-modeling Architecture

Sara Fioravanti
University of Florence
Florence, Italy
sara.fioravanti@unifi.it

Fulvio Patara
University of Florence
Florence, Italy
fulvio.patara@unifi.it

Enrico Vicario
University of Florence
Florence, Italy
enrico.vicario@unifi.it

ABSTRACT

The Reflection architectural pattern is an elegant reusable solution to design software applications based on a meta-model that provides a self-representation of the types used in the domain model. This provides significant benefits in terms of adaptability, maintainability, self-awareness, and direct involvement of domain experts in the configuration stage. However, while virtuous in the perspective of object-oriented development, the meta-model adds a level of indirection that may result in poor performance. The complexity is further exacerbated when the object-oriented domain model is mapped to a relational database. We identify four performance anti-patterns that may naturally occur in the design of a meta-modeling architecture, and for each of them we propose a refactoring intervention on the object model and on the database mapping strategy. Experimental results are reported to characterize the gain obtained applying the proposed refactoring techniques to a real case of data management system, in order to provide a roadmap for engineering the performance of meta-modeling architectures.

Keywords

Performance, Anti-patterns, Reflection pattern, Meta-modeling architecture, Dynamic architectures, Relational database, Mapping, Electronic Health Record (EHR) systems

1 INTRODUCTION

In the common practice of software development, the domain model of a system represents a conceptualization of the entities involved in a particular application domain. Consolidated object-oriented systems generally represent business entities as separate classes hard-coded directly into software and database models [1, 2]. This approach, which could be said *static*, fits well the development of systems demanding limited complexity of the domain ontology, rapid development, with expected low rate of change and limited evolutionary maintenance. However, it inevitably exposes its limitations when system requirements and domain concepts change often, leading to a continuous cycle of system changing, re-building, re-testing, and re-deploying.

In domains characterized by higher volatility and where both flexibility and run-time configurability [3] are required, a more convenient solution can be provided by more abstract architectures, open to modification, extension and evolution over time. In the literature, this kind of *adaptable* solutions, typically characterized by two different levels of abstractions, are called *meta-modeling* (a.k.a. *reflective*) architectures [4][5].

Yet, all that glitters is not gold. Designing systems able to change structure and behavior dynamically inevitably results in a more complex software architecture, making the reference model more abstract, less intuitive [6], and hard to develop [5]. Moreover, a meta-modeling architecture is often exposed to performance inefficiencies, determined in the design activity but made evident only after the deployment phase, and usually solved with expensive and partially resolutive interventions [7]. The high degree of abstraction of the underlying meta-model requires to process and instantiate, at run-time, an increased number of objects and relationships to reproduce the whole domain. This drawback is further exacerbated when the meta-model is made persistent through an Object-Relational Mapping (ORM) layer, which increases the degree of indirection. For these reasons, performance engineering comprises an essential question to be properly integrated along the whole development lifecycle.

In this paper, we address performance engineering of a meta-modeling architecture through a suite of refactoring actions, aimed at improving performance while supporting and preserving reusability and maintainability. To this end, we identify and characterize four performance anti-patterns and their applicable solutions. This contributes to the literature and practice of performance anti-patterns by providing a roadmap that should be taken into account during the design phase of meta-modeling architectures. For the sake of concreteness, but without loss of generality, we refer to the case of an Electronic Health Record (EHR) system [8] that leverages the meta-modeling architectural approach in order to achieve flexibility and run-time configurability required for recording, retrieving, and manipulating clinical information in a medical context.

2 BACKGROUND

An adaptable system must be able to change its structure and behavior dynamically so as to adapt itself to a variety

ICPE '17 Companion, April 22-26, 2017, L'Aquila, Italy

DOI: http://dx.doi.org/10.1145/3053600.3053647

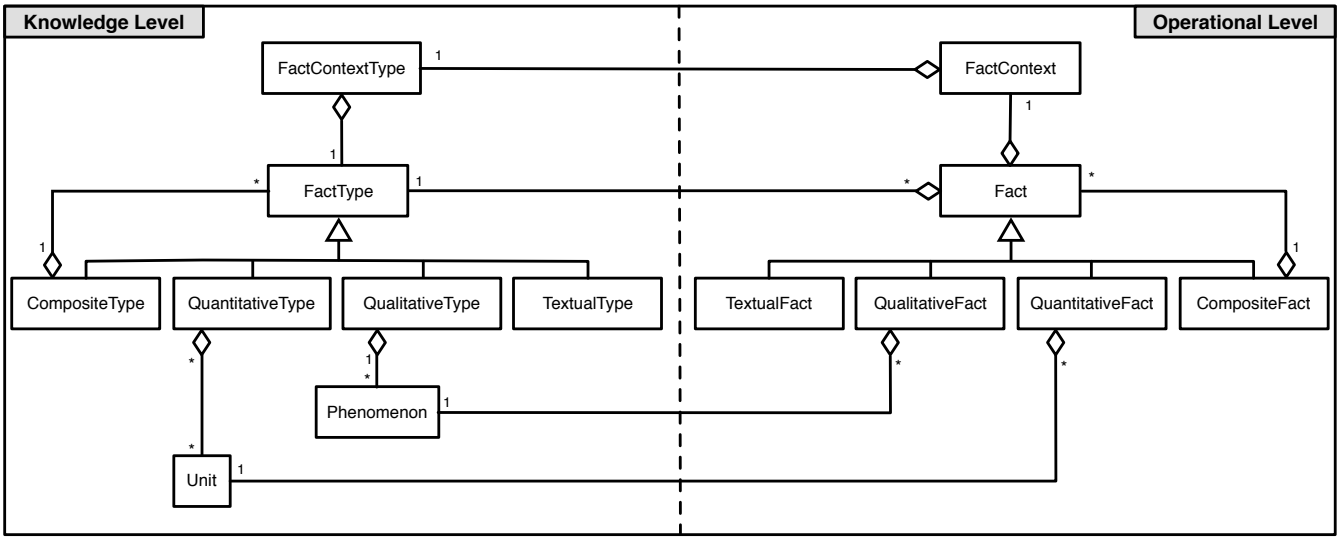

Figure 1: The UML class-object diagram of a general data collector domain based on the underlying meta-modeling paradigm, addressed in the architectural perspective by the *Reflection* pattern [4] and in the conceptual perspective by the *Observations & Measurements* pattern [9].

of concepts produced by different domains. Meta-modeling architecture may be suitable for any example of data collector system that pursues two goals: *i)* achieve a high level of adaptability and changeability [3] in order to fit into different domains; *ii)* enable domain experts to deal with the definition and maintenance of domain concepts.

From an architectural perspective, these principles are addressed through the *Reflection* pattern [4], in which the architecture is split into two parts (i.e., *meta* and *base* levels) so as to support dynamic adaptation of the system in response to changing requirements.

Various applications have been reported in the specific context of Electronic Health Record (EHR) systems [8], which structurally demand for flexibility and run-time configurability so as to dynamically adapt data structures and behavior to a variety of context-of-use. To this end, in [9], Fowler proposes the *Observations & Measurements* analysis pattern, a conceptualization of the *Reflection* pattern which allows the separation of medical concepts, represented in a so-called *knowledge level*, from clinical information, represented in a so-called *operational level*.

Fig. 1 provides a high-level specification of the domain model of a reflective architecture that we have implemented, named *Empedocle EHR system* [10], starting from Fowler's concepts: *i)* on the left, the knowledge level collects classes (i.e., `FactType` and its subclasses) and instances of medical concepts that have to be taken into account and kept up-to-date; *ii)* on the right, the operational level contains classes (i.e., `Fact` and its subclasses) and instances of clinical observations. Through a many-to-one relationship with `FactType`, the `FactContextType` class composes into a tree data structure all concepts collected in a given `FactContext` (e.g., a clinical examination). Note that the medical context is only one of those possible application scenarios based on a meta-modeling architecture: for instance, we applied the same paradigm to develop a system for monitoring the state of various types of buildings after a natural disaster, or for representing different University courses of study.

3 PERFORMANCE ANTI-PATTERNS OF META-MODELING ARCHITECTURES

A meta-modeling architecture natively requires a large amount of objects to be created at run-time to encode knowledge concepts and operational values. This verbose objects mechanism hinders performance especially when both the domain model grows in complexity and is made persistent into a relational database.

We report on four refactoring solutions that have been adopted so as to overcome the negative consequences of these performance issues, while supporting maintainability and preserving reusability. Each anti-pattern is described using a systematic template: *i) Problem*: the recurrent situation that causes negative consequences; *ii) Context*: explanation of the context where we can find the anti-pattern in a meta-modeling architecture; *iii) Solution*: how can we avoid, minimize or refactor the anti-pattern.

3.1 Mapping inheritance structures: joins explosion

3.1.1 Problem

Inheritance structures can create problems when they are mapped to a relational database which does not natively support inheritance. In cases where the domain model can change very often in terms of attributes or sub-classes, the most natural solution consists in mapping each class to its own table (i.e., *joined-tables strategy*). This strategy offers various well-known advantages in the perspective of software architecture [11], related to understandability, support for polymorphism, and maintainability of class inheritance hierarchies. However, it exposes some limits, due to the number of tables generated, one for each sub-typed entity in the hierarchy. Data reading and data writing result in heavier operations because they require the joining of multiple tables for polymorphic queries (e.g., the total set of

attributes for a particular instance is represented as a join along all tables in its inheritance path).

3.1.2 Context

In the case of a reflective architecture, as depicted in Fig. 1, `Facts` (or `FactTypes`) are specialized in different kind of entities to represent various concepts and informations. Using the *joined-tables strategy*, a table corresponding to a generic `Fact` (or `FactType`) is generated, which contains one column for each attribute in common with its children, while each sub-typed entity is mapped in a different table that contains only columns specific to its own attributes and one extra column as foreign key for uniquely identifying a row in the hierarchy. Data size grows in direct proportion to growth of the number of objects with strong impact on performances.

3.1.3 Solution

In order to overcome performance issues caused by the joined approach, a mapping strategy based on a *single table* approach should be preferred. In so doing, all attributes of super- and sub-classes are mapped into the same table, and the type of each instance is distinguished by a special discriminator column. Single table strategy collects all data in one table, and queries result less complicated due to the reduced number of join required (from a join along all tables to a single join). In general, this migration is largely eased in the case of models characterized by simple and static hierarchies, and with minimal overlapping (in terms of attributes in common) between classes in hierarchies. Nevertheless, it should be noted that single table strategy limits the power of the normalization in relational database and requires more attention to be paid at the application level to avoid inconsistencies in the data.

3.2 Mapping hierarchical structures: queries explosion

3.2.1 Problem

In Sect. 3.1 we have introduced the problem of mapping inheritance structures to relational databases, and we have suggested a refactoring solution able to reduce the complexity of executed queries. However, another performance question still remains open: how to reduce the number of queries required to retrieve all nodes in a hierarchy?

3.2.2 Context

In the context of a reflective architecture, domain structures are characterized by two different hierarchical levels: one resulting from `Fact` and `FactType` inheritance (discussed in the previous section), and another one from composition of those entities in part-whole hierarchies through the `CompositeType` class. Since tree traversal for retrieving the whole structure requires one query per node, increasing the dimension of the tree increases the number of queries required. Specifically, to retrieve the whole tree, the traversal function starts from the root node, stores all children of that node, and then repeats the traversal for each child until every leaf is visited, requiring almost one query per node.

3.2.3 Solution

To optimize the amount of queries, hierarchies can be enriched with ancestor-descendant relations, so that each node maintains a list of its ancestors. In so doing, the hierarchy can be represented as an ordered directed tree, where only one query is required for retrieving all the information contained in the whole structure.

The proposed solution drastically reduces the number of queries but requires to maintain a list of ancestors for each node: in the worst case, when all nodes belong to the same path from the root to the leaf, given a tree with depth D, being d the current depth, and n_d the number of nodes at d level, the total number of ancestors A results: $A = 1 + \sum_{d=1}^{D} d * n_d$.

3.3 Mapping entity associations: fetching overloading

3.3.1 Problem

In a domain model, associations represent relationships between classes. While object-oriented languages represent associations using object references that are navigable, in the relational world, an association is represented as a foreign key column that it is not a directional relationship by nature. In order to smooth the object/relational paradigm mismatch, association mapping plays a lead role. Object-Relational Mapping (ORM) layers often include the ability to make one-to-many relationships either unidirectional or bidirectional. Since unidirectional associations are more difficult to query, one of the best practice for large scale applications suggests to turn almost all associations navigable in both directions [12]. However, in some contexts, this approach can result in retrieving data not really necessary in every use-cases, leading to memory overload.

3.3.2 Context

Referring to Fig. 1, all information related to a particular context may be easily fetched without recurring to an explicit query using bidirectional associations between `Fact` and `FactContext`, navigating through and iterating over persistent objects.

However, while this solution brings evident advantages at the object level, it may not be the most convenient choice in terms of performance, relying on the specific context-of-use of the system: in the practice of EHR systems, for example, a medical examination is not required to be aware about clinical information collected during its execution; viceversa, it is mandatory for a clinical information to know its own context.

3.3.3 Solution

Moving from bidirectional to unidirectional association and removing the redundant one-to-many association mapping preserves the capability to retrieve the `FactContext` related to a specific `Fact` simply exploiting the entity association. On the other hand, retrieving all `Facts` belonging to a specific `FactContext` can be done through a single query.

This results in some interesting benefits. First, dependencies between classes and packages are lower and clearer, thanks to the increased number of unidirectional relationships. Second, the whole code appears well structured in self-contained areas, and this supports testing and future refactoring interventions. Third, from a performance perspective, queries become more efficient because only small objects are loaded, and additional information can be retrieved through dedicated queries. Finally, note that this

kind of mapping reduction is applicable only to classes that are in a weak form of association. Conversely, the bidirectional mapping must be preserved when classes are in a strong form of relationship (e.g., composition), in order to emphasize the dependency of the contained class to the life cycle of the container class.

3.4 Inheritance vs. aggregation: fetching overloading

3.4.1 Problem

One of the most common technique for reusing functionality in object-oriented systems is class inheritance, where a class may inherit fields and methods of its superclass and may override some of those fields and methods to alter the default behavior. However, this approach leads up to some inherent hurdles. On the one hand, the whole model design is affected by this choice, resulting in a less intuitive representation of information, due to the workarounds often required to overcome some implementation barriers (e.g., multiple inheritance). On the other hand, increasing the number of specialized classes, inheritance-based model ends up including several classes derived from the base one, very different from each other, driving to more complex (and slow) queries, as described in Sect. 3.1 and in Sect. 3.2.

3.4.2 Context

In the meta-modeling architecture of Fig. 1, a special category of knowledge is represented by `QualitativeType` instances, i.e. domain concepts where related information can assume only specific values (called `Phenomenon`) in a finite range. In general, a `Phenomenon` is sufficiently characterized by its label, but sometimes a plain string is not enough and it is necessary to encode extra and more structured information. In the context of healthcare, a typical example is represented by the International Classification of Diseases (ICD) [13], a standard designed to map diseases and other health problems to codes.

3.4.3 Solution

As first solution, ICD codes may be treated as specializations of `Phenomenon` types, in order to support their exploitation inside qualitative information. However, aggregation represents a more efficient solution in terms of performance. In this way, classes corresponding to special phenomenon categories can be placed in weak association with the `Phenomenon` class, and consequently, they are responsible for generating associated phenomena, starting from extra collected information. The resulting model is easier to maintain, test, and extend, where queries concerning phenomena become simpler, faster and more efficient.

4 EXPERIMENTAL EVALUATION

We conducted some experiments and collected performance measurements on a dataset of clinical examinations acquired by a real case of context-specific EHR system, named *Empedocle* [10], presently in use in various clinics at AOUC, the main hospital in Florence.

4.1 Experimental methodology

In the context of EHR systems, the complexity is often related to the medical specialty under consideration. In our experience, performance is not an issue for the context of Ophthalmology, whose basic examination results in a lightweight data structure, where only few tens of observations are collected for each patient. Otherwise, when the system was configured for operating in the Cardiology department, performance issues have made evident.

Table 1: Comparison of Ophthalmology and Cardiology examination structures in *Empedocle*.

Specialty configuration	Nodes	Leaves	Max depth	Avg depth
Ophthalmology	64	45	5	3.7
Cardiology	639	495	7	3.6

Table 1 highlights the complexity of object models in terms of number of nodes (i.e., `FactTypes`), leaves and depth (i.e., the maximum distance from the root node) of the data structure used to represent a medical examination (i.e., the tree-like structure in the knowledge level). In particular, the number of nodes and leaves in a Cardiology examination is ten times bigger than values for an Ophthalmology examination.

Performance limitations were observed in two main scenarios of interaction, namely "*performing a medical examination (UC1)*" and "*accessing the patient's EHR content (UC2)*". In *UC1*, the examination structure is first loaded; then clinical information corresponding to observed facts are filled by the user; finally, the system stores clinical data collected during the examination. *UC1* combines *fetching* operations to retrieve the data structure from the knowledge level (i.e., `FactContextType`), and *writing* operations to persist collected data at the operational level (i.e., `FactContext`). In *UC2*, the health professional accesses a performed medical examination and consults collected clinical information. Since this scenario requires to just retrieve data structure and content, *UC2* is characterized by *read-only* operations.

In order to isolate performance issues closely related to the reflective architecture from those more related to technology solutions adopted, specific performance tests focused on the meta-level domain model were implemented and run. We investigate the time to execute each task, and, in particular, the number of queries and joins generated during each scenario under consideration, which significantly impact on performance as documented by [14] in terms of "*N+1 queries*" and "*Circuitous Treasure Hunt*" anti-patterns.

The time to perform each task was evaluated for all 22 000 examinations in the Ophthalmology dataset and for all 13 000 examinations in the Cardiology dataset. Each experiment has been repeated 100 times to estimate the mean time value in order to limit impact of the start-up time required by ORM and any outer factor that can influence performance. All reported experiments were performed on a MacBook Pro with 2.8 GHz Intel Core i7 and 16GB of 1066 MHz SDRAM DDR3L installed in pairs (two 8GB modules).

4.2 Experimental Results

The performance of the refactored model obtained after removing the four anti-patterns described in Sect. 3 was tested through the measurements and comparison of time to perform the two scenarios discussed in Sect. 3.4, so as to eval-

uate the performance gain obtained applying the proposed refactoring solutions.

Table 2 and Table 3 illustrate the impact on performance produced by the complexity of the object model of *Empedocle* in the different configurations of Ophthalmology and Cardiology. Specifically, the upper part of Table 2 compares the average time and the coefficient of variation for completing the *UC1* scenario measured before and after the refactoring interventions, while the upper part of Table 3 compares the number of queries and joins related to the same scenario, as determined by the structure of the examinations involved in the experimentation. In the same way, the lower part of Table 2 and Table 3 are related to the *UC2* scenario.

Table 2: Time mean value (μ) and coefficient of variation (c_v) for *UC1* (fetch and write) and *UC2* (read-only) before and after refactoring (100 repetitions).

	Specialty configuration	Before		After	
		μ (ms)	c_v	μ (ms)	c_v
UC1	Ophthalmology	203.65	0.22	163.67	0.29
	Cardiology	2004.89	0.08	1755.57	0.85
UC2	Ophthalmology	252.3	0.12	44.18	0.7
	Cardiology	585.43	0.13	196.02	0.41

Table 3: Number of queries and joins for *UC1* (fetch and write) and *UC2* (read-only) before and after refactoring.

	Specialty configuration	Before		After	
		Queries	Joins	Queries	Joins
UC1	Ophthalmology	557	261	630	25
	Cardiology	5755	561	6167	71
UC2	Ophthalmology	141	6274	10	29
	Cardiology	322	13701	39	65

The comparison of results in Table 2 reveals a gain of performance by a factor of 1.24 for Ophthalmology and 1.14 for Cardiology in the *UC1* scenario. Indeed, most of the reported optimizations in the refactored model give their major contribution to the *UC2* scenario. For this reason, it is relevant to pay attention on data reported in the lower part of Table 2, where a huge improvement in read-only operations emerges. Performance for the second scenario presents a gain of almost 5.7 and 3.0 times for Ophthalmology and Cardiology, respectively.

Moreover, the overall number of queries and joins is being limited. Specifically, while some additional queries are required by the refactored model to perform the first scenario (as reported in the upper part of Table 3), a substantial reduction in the number of accesses to the database is highlighted for the *UC*2 scenario, as depicted in the lower part of Table 3 by factor of 14.1 for Ophthalmology and 8.3 for Cardiology.

Turning to details, the solution provided in Sect. 3.1 decreases the number of join operations during the *UC2*; furthermore, the refactoring proposed in Sect. 3.2 limits the number of queries needed for this scenario, by reducing them to only one. The different way to map entity associations shown in Sect. 3.3 is intended to avoid unnecessary references that carry an overload of objects in memory. Finally, moving from inheritance to aggregation as reported in Sect. 3.4 limits the amount of retrieved data when a lightweight `Phenomenon` associated to a `QualitativeFact` is requested.

5 RELATED WORKS

From an architectural perspective, meta-modeling architectures have been studied and discussed, as we highlighted in Sect. 2, by [4], [15], [16] and Fowler in [9]. Also Beale in [17] introduces two different levels of abstraction: while an *information level* of classes describes the reference model, a *knowledge level* defines domain concepts which have to be processed as instances of reference model classes, driving the system at run-time through archetypes and templates concepts. Similar principles are achieved in the conceptual perspective through the *Item-Description* pattern [15], also known as *Type Object* pattern [16], which allows to dynamically define new business entities at run-time, decoupling an object from its description (i.e. type information).

In the context of healthcare, the impact of meta-modeling architectures for EHR applications and interoperability is mentioned in openEHR [18] and Health Level 7 (HL7) [19] standards.

Performance issues have been widely addressed in the literature of performance anti-patterns, which describes recurring problems with significant impact on performance. In [14] and their following works, 14 generic problems are identified and corresponding solutions are suggested. Several techniques aimed at automated detection of performance anti-patterns in software architectural models are proposed by [20] and [21], followed by [22], which defines refactoring actions after problems detection.

Performance of object-relational mapping is analyzed in few studies. The influence of optimizations and configurations on the performance of the object-relational mapping tool Hibernate is evaluated in [23] and [24]. Hibernate performance is also discussed in [25] and [26], comparing with outdated solutions of object-oriented databases through benchmarks.

6 CONCLUSIONS

Designing and implementing a software intensive system based on a meta-modeling architecture offers several proven benefits in the software engineering perspective: improved maintainability; high degree of adaptability to fit the needs of complex and volatile domains; inversion of responsibility to delegate changes on the system structure and behavior to domain experts, without intermediation of software engineers. However, a meta-modeling architecture also carries performance consequences, that often remain hidden until testing and deployment.

In this paper, we identified and illustrated four performance anti-patterns which may occur when a meta-modeling architecture is mapped into a relational database, and we discussed design and mapping choices that may have a sound rational in the perspective of object-oriented design but can have significant and negative impact in the performance perspective. For each of these anti-patterns, we proposed design solutions and implementation choices that comprise a performance-oriented guideline for software developers.

Benefits of the proposed refactoring strategies were assessed through experimentation on a real case of EHR system, referring to actual scenarios in the clinical practice. Specifically, while our proposed solutions preserve performances in write-dominant scenarios, a huge improvement emerges when read-only operations are performed, since refactoring interventions are mainly focused on three specific targets: *i)* decrease the number of queries, in order to minimize access to database; *ii)* reduce the queries complexity, so as to downsize the number of join operations between tables; *iii)* limit the retrieved data to the minimum necessary amount.

More works is needed to evaluate as proposed and new anti-patterns may affect other functional and non-functional system requirements (e.g., reliability, maintainability), in order to guarantee both the consistency and the right trade-offs among these properties. At the same time we have an on-going experimentation to compare performances using no relational databases underlying the objects model [27].

7 References

[1] B. Grady, *Object-oriented analysis and design with applications*. Addison Wesley Longman, 1994.

[2] C. Date, *An Introduction to Database Systems*, 8th ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[3] "Iso/iec 25010:2011: Systems and software engineering – systems and software quality requirements and evaluation (square)," the International Organization for Standardization, Tech. Rep., 2011.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture, Vol1:A System of Patterns,Wiley, 1996*.

[5] J. W. Yoder and R. Johnson, "The adaptive object-model architectural style," in *Software Architecture*. Springer, 2002, pp. 3–27.

[6] C. Atkinson and T. Kühne, "Reducing accidental complexity in domain models," *Software & Systems Modeling*, vol. 7, no. 3, pp. 345–359, 2008.

[7] C. U. Smith and L. G. Williams, *Performance solutions: a practical guide to creating responsive, scalable software*. Addison-Wesley Professional, 2001.

[8] ISO/TR, *ISO/TR 20514:2005. Health informatics — Electronic health record — Definition, scope and context*, 2005.

[9] M. Fowler, *Analysis patterns: reusable objects models*. Addison-Wesley Longman Publishing Co., Inc, 1996.

[10] F. Patara and E. Vicario, "An adaptable patient-centric electronic health record system for personalized home care," in *8th International Symposium on Medical Information and Communication Technology (ISMICT)*. IEEE, 2014.

[11] S. Ambler, *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. New York, NY, USA: John Wiley & Sons, Inc., 2003.

[12] L. Red Hat Middleware. (2004) Hibernate best practices. [Online]. Available: http://docs.jboss.org/hibernate/core/3.3/reference/en/html/best-practices.html

[13] W. H. Organization, *International statistical classification of diseases and related health problems*. World Health Organization, 2004, vol. 1.

[14] C. U. Smith and L. G. Williams, "Software performance antipatterns," in *Proceedings of the 2Nd International Workshop on Software and Performance*, ser. WOSP '00. NY, USA: ACM, 2000.

[15] P. Coad, "Object-oriented patterns," *Communications of the ACM*, vol. 35, no. 9, pp. 152–159, 1992.

[16] M. R. Johnson, "Type object," in *Pattern Languages of Program Design 3*. AddisonWesley, 1997, pp. 47–65.

[17] T. Beale, "Archetypes: Constraint-based domain models for future-proof information systems," vol. 105, 2002.

[18] T. Beale, S. Heard, D. Kalra, and D. Lloyd, "OpenEHR architecture overview," *The OpenEHR Foundation*, 2006.

[19] R. H. Dolin, L. Alschuler, C. Beebe, P. V. Biron, S. L. Boyer, D. Essin, E. Kimber, T. Lincoln, and J. E. Mattison, "The hl7 clinical document architecture," *Journal of the American Medical Informatics Association*, vol. 8, no. 6, pp. 552–569, 2001.

[20] C. Trubiani and A. Koziolek, "Detection and solution of software performance antipatterns in palladio architectural models," in *ACM SIGSOFT Software Engineering Notes*, vol. 36, no. 5. ACM, 2011.

[21] T. Parsons and J. Murphy, "A framework for automatically detecting and assessing performance antipatterns in component based systems using run-time analysis," in *9th International Workshop on Component Oriented Programming*, vol. 4, 2004.

[22] D. Arcelli, V. Cortellessa, and C. Trubiani, "Antipattern-based model refactoring for software performance improvement," in *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures*. ACM, 2012, pp. 33–42.

[23] P. Van Zyl, D. G. Kourie, L. Coetzee, and A. Boake, "The influence of optimisations on the performance of an object relational mapping tool," in *Conf. of the South African Institute of Computer Scientists and Information Technologists*. ACM, 2009, pp. 150–159.

[24] R. Singh, C.-P. Bezemer, W. Shang, and A. E. Hassan, "Optimizing the performance-related configurations of object-relational mapping frameworks using a multi-objective genetic algorithm," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ACM, 2016, pp. 309–320.

[25] P. Van Zyl, D. G. Kourie, and A. Boake, "Comparing the performance of object databases and orm tools," in *Conf. of the South African institute of computer scientists and information technologists on IT research in developing countries*. South African Institute for Computer Scientists and Information Technologists, 2006, pp. 1–11.

[26] R. Kalantari and C. H. Bryant, "Comparing the performance of object and object relational database systems on objects of varying complexity," in *British National Conference on Databases*. Springer, 2010.

[27] S. Fioravanti, S. Mattolini, F. Patara, and E. Vicario, "Experimental performance evaluation of different data models for a reflection software architecture over nosql persistence layers," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ACM, 2016, pp. 297–308.