UNIVERSITÀ
DEGLI STUDI
FIRENZE

Università degli Studi di Firenze
Dipartimento di Ingegneria dell'Informazione (DINFO)
Corso di Dottorato in Ingegneria dell'Informazione

Curriculum: Ingegneria Informatica
SSD ING-INF/05

Performance engineering of multi-level meta-modeling architectures based on J2EE stack

Candidate
Sara Fioravanti

Supervisor
Prof. Enrico Vicario

PhD Coordinator
Prof. Luigi Chisci

---

ciclo XXX, 2014-2017

Università degli Studi di Firenze, Dipartimento di Ingegneria dell'Informazione (DINFO).

To Emanuele and my parents

Acknowledgments

I would like to show my warm thank to my advisor Prof. Enrico Vicario, for his guidance and advices during these three years. Thanks to Dr. Fabio Mori, the head of the Cardiovascular Imaging Unit at the Careggi Hospital in Florence, Dr. Emanuele Crocetti, the director of the Tuscany cancer registry, and President of the Italian network of cancer registries (AIRTUM) since 2014, and Dr. Andrea Martini, statistician employed at the Institute of Oncological Study and Prevention (ISPO) for their precious collaboration and support.

I would also like to thank present and past colleagues in the Software Science and Technology Laboratory (STLab) at the University of Florence (in alphabetical order): Nicola Bertocci, Marco Biagi, Irene Bicchierai, Andrea Bonacchi, Laura Carnevali, Matteo Lapini, Tommaso Magherini, Sandro Mehic, Carlo Nocentini, Manuel Pagliai, Marco Paolieri, Jacopo Parri, Samuele Sampietro, Valeriano Sandrucci, Francesco Santoni, Francesco Scutifero, Kumiko Tadano, Fabio Tarani, Jacopo Torrini.

I'd like to give special thanks to Fulvio Patara and Simone Mattolini: it has been a pleasure and a honor to work with you.

Last but not least, thanks to my husband, my mom and dad, family members and friends, without whom I was nothing: they not only assisted me financially but also extended their support morally and emotionally.

Abstract

Non-functional requirements of changeability and adaptability have primary relevance for a large class of software intensive systems that are intended for managing great volumes of data with a high degree of variety in the structure of contents. The attainment of these qualities can be largely facilitated by the assumption of a tailored software architecture.
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, design by patterns does not account for performance as first-class requirement, and naturally incurs in well-known performance anti-patterns, which may become crucial when volume and variety must meet also velocity. The complexity is further exacerbated when the object oriented domain model is mapped to a relational database.
The aim of this dissertation is to address performance engineering of a meta-modeling architecture based on J2EE technological stack, documenting common performance issues connected to the persistence layer and proposing some solutions.
In this work are presented four selected performance anti-patterns about object-relational mapping strategies and proposed refactoring solutions. This research project also reports comparative experimental performance results attained by combining the pattern-based domain logic with a persistence layer based on NoSQL paradigm, and proposes techniques to identify and improve performance issues in a J2EE architecture.
Experimental results are obtained by applying proposed solutions in the concrete case of a real application of data management in Healthcare context based on Reflection architectural pattern. Those results indicate the gain

obtained in several use cases by using refactoring actions in the relational database scenario, and by replacing the persistence layer with NoSql technology (in particular MongoDB) in the secondo scenario.

Contents

Chapter 1

Introduction

My research activity in these years has been focused on the advanced *software architectures* topic, with particular focus on non-functional requirements like high performance, maintainability and adaptability [3,45], in the experimental and application context of Healthcare information systems.
A *software architecture* can be described following some recurrent scheme and architectural style called *architectural patterns* [17], that, together with the *design patterns* that operate in the modules and relationships domain [36], allow to define systems and frameworks. The use of such solutions leads to some important advantages such as maintainability, scalability and modularity [3, 45].

During these three years of research a key role has been played by the study and application of the *Reflection* architectural pattern and *Observations and Measurements* analysis pattern, that suggest methods to model a system capable of changing its structure and behaviour in a dynamical way. Thanks to this pattern it has been possible to design *Adaptable Systems* [20], that are systems capable of evolving in time, adapting to modifications of the functional requirements, of the data types to manage, and of the technologies used.

1.1 The objective

The goals are focused around various arguments in the Software Methods and Technology area, with reference to themes of *Information Management*

Systems (IMSs)[1], in particular:

- *Adaptable Systems* applied to *IMSs*
- *Performance* and *engineering* of *IMSs* based on *J2EE technological stack*
- *Performance* evaluation of *IMSs* using different data persistence technologies.

The medical domain fits perfectly the experimentation of software architectures implementing the Reflection pattern, given its complex and evolving nature. Thanks to the ongoing collaboration between the Software Technologies Lab (STLab) and the main hospital of Florence Careggi (AOUC) and the Institute for Oncological Study and Prevention (ISPO), it has been possible to apply reflection's modeling principles inside the process of gathering and organization of clinical data, thanks to the help of instruments like *Electronic Health Record Systems (EHRSs)* [46] and *Clinical Decision Support Systems (CDSSs)* [14].

In particular, *EHR* systems must be able to adapt to a variety of medical concepts from different specializations [82]. The reference standard for the clinical records is Open-EHR [11] [10] which is defined through archetypes.

Generally an *EHRS* has some conflicting requirements, such as the need to represent data in a structured way, to integrate and verify information coming from different sources, and to adapt to users with different levels of specialization. Moreover, an *EHRS* should be adaptable to diversified contexts that require deep customization of information's structure.

The Regional Registers is another application context, in which information about the individuals that live in a specific territory are gathered. This information include the kind of the diagnosed disease, biographical data of the patient, its clinical condition, the treatments he has been given and that he is still following, and the disease evolution. This kind of application has

[1]The term *Information Management Systems* can be reductive for this kind of architecture. In literature systems are mainly differentiated in *Data*, *Information* and *Knowledge* systems [4,13,86]. Due to the abstraction introduced by multi-levels, this architecture can be able to collect low level data (e.g. raw data from sensors), offer data aggregation (i.e. information, e.g. dossiers or documents), and also support data mining techniques for knowledge discovery. For this reason these terms are used interchangeably in this thesis.

some common requirements with EHRs (i.e. represent data in a structured way and high-configurability to deal with data structures that change over time) but also needs to address the scalability issue because of the increasing volume of data. Scalability refers to the characteristic of a system to increase performance by adding additional resources or growing in complexity.

1.2 State of the art

Modeling adaptable systems and relative challenges is a topic largely addressed in literature [17, 20, 31, 83]. In particular, several patterns have been described to deal with data variability and requirements evolution, in order to realize *meta-modeling* architectures [83] (also referred as *reflective* architectures [17]): *Adaptive Object-Model* (AOM) [47, 82], *Reflection* architectural pattern [17] and *Observations and Measurements* analysis pattern [34].
Healthcare systems are analyzed in [45, 72] where emerges that adaptability and changeability are primary requirements for this context. The usage of *Adaptable Systems* in healthcare context is reported in [11] as a reference standard, and in other works [34, 61].

Performance is one of systems and software requirements [3, 45] that defines the quality of the product. One way to evaluate performance is through the identification of software performance anti-patterns, first described by Smith and Williams in [68, 69, 71]. Several approaches have been proposed for their detection or to evaluate the overall performance of software systems [5, 22, 51, 74, 75].
Performance anti-patterns are also contextualized to performance evaluation of J2EE architectures in [18, 28].
Finally, performance evaluation of persistence layers for J2EE architectures are presented in [65, 80, 85], and specifically to NoSql technologies in [42, 57, 81].

1.3 Contributions

The developing of systems capable of dynamically changing their structure and their behaviour inevitably translates to a bigger architectural complexity, making the *reference model* more abstract, less intuitive [8] and harder to develop [83]. In particular, a meta-modeling architecture often suffers of *per-*

formance problems. In fact the high abstraction level of the model requires the instantiation of a growing number of objects and relations at run-time, that can have an impact on the use of the in-memory resources. Such critical issue can be more evident whenever it's needed to have a *database* persistence layer, like in *web applications*, leading to a growth in number and complexity of the query needed to retrieve the information.

The issue of performance has been faced in my research with two parallel and complimentary studies:

- the identification of performance anti-patterns [68,69,71] in the context of adaptable architectures
- the experimentation of different layers for data persistence (i.e. relational databases and NoSql technologies [38])

One of the goals of the research was to extend literature in the topic of performance anti-pattern defining four new recurrent problems and their solution, that can be kept in mind during the design and development phases of a meta-modeling architecture. The proposed solutions have been experimented, without losing their general purpose, even in a real application of medical record in use at the Florence AOUC. The results show how some design choices, that can appear counterintuitive from the modeling point of view, translates to a relevant performance improvement. Specifically, two of the most important applicative use cases have been analyzed along with how their mapping on a MySql relational database influences the system response times, comparing the results of the original implementation with the ones measured after the proposed refactoring.
The article [33] has been presented at the conference *International Conference on Performance Engineering 2017* (ICPE 2017).

The second goal of the research was instead focused on the usage of different data management tools from the usual ones, like non-relational dbms, that can guarantee the persistence of a high volume of information, always available online, and above all can easily acquire data without a fixed schema. The research has been focused not on the evaluation of the available technologies in the non-relational databases area, but more on the performance engineering of existing applications through the substitution of the persistence scheme, keeping the domain logic structure unchanged. The choice of technologies *Not Only Sql* (NoSQL) under test is made because of

their data structure and the promising performance improvement [42] [81].

Through an experimental analysis has been possible to measure the performance gain using technologies like *Neo4j* and *MongoDB* showing how the proposed data representation using these databases can be considered equivalent and complete, in comparison to the original relational representation. The evaluation has been done referring the main interaction scenario of one of the medical applications: a medicine doctor opens the patient file to look at his past clinic exams and read the gathered data, through a *read* operation on the database.
The article [32] has been presented at the conference *International Conference on Performance Engineering 2016* (ICPE 2016).

1.4 Roadmap

The rest of the thesis is organized as follows.

In Ch. 2, I expose characteristics of multi-level meta-modeling architectures and their usefulness in contexts where flexibility and high-configurability are essential requisites even at run-time. I also present some popular pattern-based solution in literature and few benefits and limits of these approaches.

In Ch. 3 I report my laboratory's experience with a Data Collection System (DCS) in the Healthcare domain, based on a meta-modeling architecture, that has been the subject of my performance research and improvement. For completeness, I refer also to others DCS in Healthcare and other applicative contexts.

In Ch. 4 I briefly describe Java EE specifications used to realized the meta-modeling architectures previously presented and I introduce the related performance topic, mainly related to the data persistence tiers of the J2EE stack.

Ch. 5 and Ch. 6 are the novel contribution of this thesis. Each is structured with a presentation of the context, some detected performance issues, a proposed solution, an experimental evaluation and a final discussion of results. In particular, Ch. 5 addresses performance engineering of a meta-modeling architecture through a suite of anti-patterns and refactoring ac-

tions, detailing results of this refactor applied to the EHR system mentioned before. Ch. 6 reports the result of experimentations on the EHR aimed at measuring the performance gain obtained moving from relational database (*MySql*) to two different NoSQL technologies (*Neo4j* and *MongoDB*).

Discussion and some ideas for future work are provided in Ch. 7.

Chapter 2

Meta-modeling architectures for adaptable systems

This Chapter shows characteristics, advantages and issues of multi-level meta-modeling architectures and their usefulness in contexts where flexibility and high-configurability are essential requisites even at run-time.

2.1 Motivations

Non-functional requirements of *changeability* and *adaptability* [45] have primary relevance for a large class of software intensive systems that are intended for managing great volumes of data with a high degree of variety in the structure of contents. In fact, domains characterized by high volatility and flexibility, where software requirements are not stable but evolve over time, require architectures that are capable of support variation.

Consolidated object-oriented systems generally represent business entities as separate classes hard-coded directly into software and database models [24, 41]. 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. Typically, object-oriented systems use distinct classes to represent domain concept, hard-coding into them information about their properties. There are particular application classes in which the domain concepts can change in time, either after a natural evolution or because

the context complexity can lead to a difficulty in knowledge transfer between the domain expert and the developer, often due to the difficulty in acquiring and inferring the requirements, calling for subsequent fixing interventions [40]. Every time an update is necessary, this results in a cycle of re-coding, re-building, re-testing, and re-deploying of the software application. For example let's theorize to have to represent a library and the books inside: with a classical approach, every book should be represented with a class with its attributes (e.g., title, author, classificationIdentifier, ...). Whenever, after a reorganization, the classification system is modified, it could be necessary to transform the attribute *classificationIdentifier* in two different attributes: *category* and *identifier*. This change will affect methods, tests, persistence layer, user interface, etc. and it will need a system shutdown, a new deploy and a reboot. Therefore, it is clear that the complexity deriving from particular variations should be managed in a more *dynamic* way, through a mechanism that allows run-time configurability [45].

2.2 Pattern-based solutions

To address the problem of variability many solutions can be found in literature. Model-driven engineering systems have focused on generative approaches, where code is generated automatically from model, enabling subsequent changes or refined of components. However, these changes must be introduced by developers (change agents) and produce a re-compile and redeploy cycle.
Architectures that needs to (a) allow the configuration of the systems' variability at run-time (b) expand the notion of change agents outside the scope of the development team (c) represent a huge domain evolving over time, can be instead built using pattern-based solutions [29].
In the literature [31], adaptable architectures are mainly addressed as *metamodeling* architectures [83] or *reflective* architectures [17].
The real-world scenarios that can benefit from these solutions are all those that have a very complex domain, that is subject to frequent changes in terms of information representation or organizational process: for example, healthcare, public administration, business, and industry.

2.2.1 The Adaptive Object-Model

An Adaptive Object-Model (AOM) is a system that represents classes, attributes, relationships, and behavior as metadata. The system is a model based on instances rather than classes. Users change the metadata (object model) to reflect changes in the domain. These changes modify the system's behavior.

Adaptive Object-Model architectures are usually made up of several smaller patterns:

- `TypeObject`: most object-oriented languages structure a program as a set of classes. A class define the structure and behavior of objects. Object-oriented systems generally use a separate class for each kind of object, so introducing a new kind of object requires making a new class, which requires programming. However, developers of large systems usually face the problem of having a class from which they should create an unknown number of subclasses [47].

  Each subclass is an abstraction of an element of the changing domain. TypeObject makes the unknown subclasses simple instances of a generic class; new classes can be created at run-time by instantiating the generic class. Objects created from the traditional hierarchy can still be created but making explicit the relationship between them and their type. Fig. 2.1 shows an example of how a Car class with a set of subclasses such as Caravan. Camry. and Explorer is transformed into a pair of classes, Car and CarType. These transformed classes represent the class model of the interpreter and are used at runtime to represent the Entities and EntityTypes for the system. Replacing a hierarchy like this is possible when the behavior between the subclasses is very similar or can be broken out into separate objects. In these cases, the primary differences between the subclasses are the values of their attributes.

- `Property`: the attributes of an object are usually implemented by its instance variables. These variables are usually defined in each subclass. If objects of different types are all the same class, how can their attributes vary? The solution is to implement attributes differently. Instead of each attribute being a different instance variable, make an instance variable that holds a collection of attributes (Fig. 2.2). This can be done using a dictionary, vector, or look up table. In our exam-

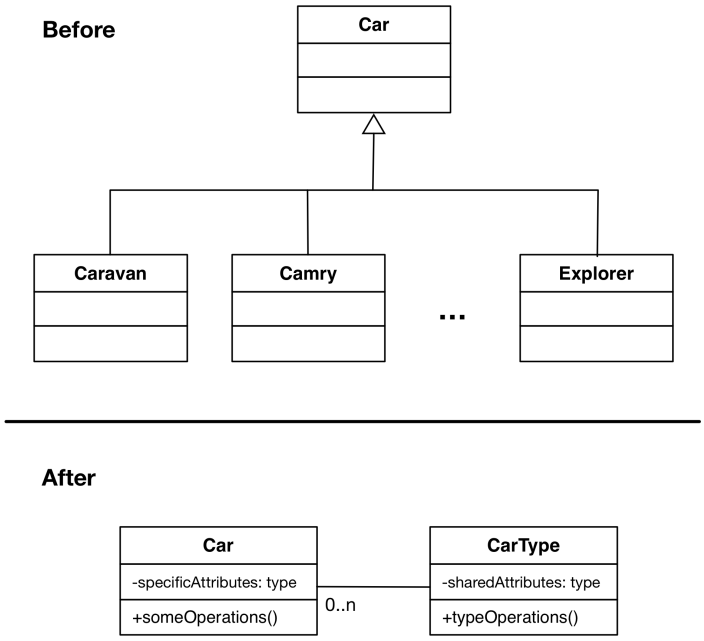


Figure 2.1: Type-Object Pattern

ple, the Property holds on to the name of the attribute, its type, and its current value.

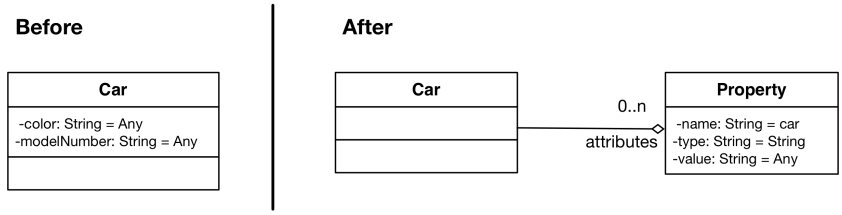


Figure 2.2: Property Pattern

- `TypeSquare`: in most Adaptive Object Models, TYPE OBJECT is used twice: once before using the PROPERTY pattern, and once after it. TYPE OBJECT divides the system into Entities and EntityTypes. Entities have attributes that can be defined using PROPERTIES. Each

Property has a type, called PropertyType, and each EntityType can then specify the types of the properties for its entities.

Fig. 2.3 represents the resulting architecture after applying these two patterns, which we call TypeSquare [82]. It often keeps track of the name of the property, and also whether the value of the property is a number, a date, a string, etc. The result is an object model similar to the following: Sometimes objects differ only in having different properties. For example, a system that just reads and writes a database can use a Record with a set of Properties to represent a single record, and can use RecordType and PropertyType to represent a table.

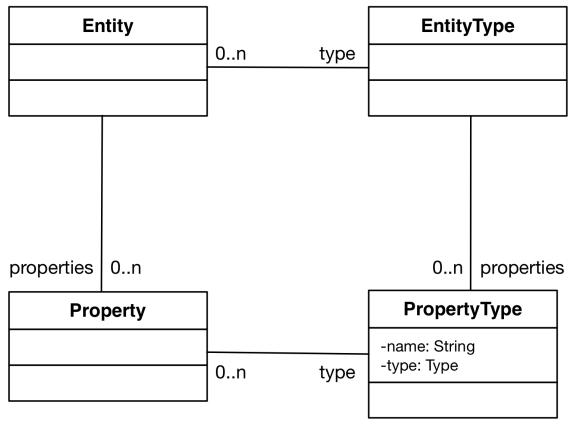


Figure 2.3: Type Square Pattern

- `Composite`: Composite [37] is used for building dynamic tree structure types or rules. For example, if the entities need to be composed in a dynamic tree like structure, the Composite pattern is applied.

2.2.2 The Reflection architectural pattern

The *Reflection* architectural pattern [17] provides a mechanism that allows for dynamically changing data structure and system behaviour at run-time. To this end, the domain logic is modeled using two different levels of abstraction (Fig. 2.4):

- a *meta* level provides a self-representation of the system encoding knowledge about data type structures, algorithms, and relationships;
- a *base* level application logic carries concrete data whose interpretation is determined by the values of so-called *metaobjects*.

Metaobjects in the meta level encapsulate and represent information about the software that may change, while providing an interface to facilitate modifications to the meta-level. The base level defines the application logic and uses information provided by the meta level. Changing metaobjects changes the way in which base-level components communicate, but without modifying the base-level code. Furthermore, it's important to note that, in contrast to a layered architecture, there are mutual dependencies between both layers. The base level builds on the meta level, and vice-versa. Finally, through the *metaobject protocol* (MOP), it's possible to manipulate the metaobjects and to check the correctness of the change specification.

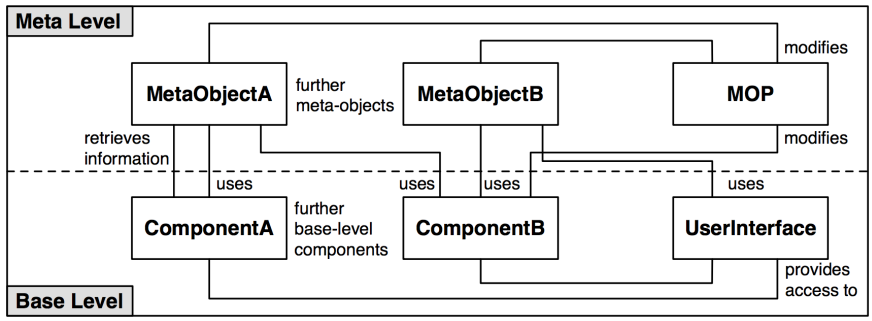


Figure 2.4: The Reflection Architectural Pattern

2.3 Benefits and limits

A meta-modeling architecture represents classes, attributes, relationships and behaviors as metadata, providing great flexibility for applications. It allows relationships, attributes and behaviors to be changed at runtime by programmers, and sometimes by end users, carrying out an *inversion of responsibility* since domain expert are directly involved in the configuration stage, overcoming misunderstandings between domain specialists and software developers. In so doing, the whole development cycle is speeded up, reducing significantly the number of required maintenance interventions.
There are several reasons to implement this kind architecture:

- The number of subclasses is unknown upfront
- The number of subclasses is huge
- Business rules rapidly change
- Changes to the system have to be made without the system going down

Furthermore, design by patterns brings a number of further benefits, mostly linked to the quality of the code, and notably to maintainability, reusability, and consolidated understanding of implementation choices and consequences.

However, modeling systems that emphasize changeability and adaptability as primary requirements result in a more complex software architecture, with various drawbacks. A pattern-oriented architectural design can partially mitigate hurdles resulting from the increased complexity induced by the application of meta-modeling principles, but the system remains hard to understand, code, test, and maintain [83].

Secondly, developing adaptable systems implies some relevant implementation challenges:

- mapping the high-level reference model into a low-level data layer, in order to make the model persistent;
- adapting Graphical User Interfaces (GUIs) to volatile domain concepts at run-time;
- supporting system maintenance for both software developers and domain experts, through the use of specific tools and GUIs.

- a meta-modeling architecture is often exposed to performance inefficiencies.

Chapter 3

Adaptable systems in healthcare

This Chapter describes how meta-modeling architectures principles are particularly suited in the Electronic Health Record (EHR) systems context and how they can be applied to build a Data Collection System in the Healthcare domain.
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. However, the usage of Adaptable Systems in healthcare context results particularly appropriate.
The main contribution to the Empedocle's architecture developed in the Software Technologies Lab (STLab) and reported in 3.2 is provided by Valeriano Sandrucci, Fulvio Patara [61], Simone Mattolini and others. I report here some details of this architecture to clarify the performance research and improvement in the next chapters.

3.1 Electronic Health Record systems

The literature reports a number of real usages and examples application of adaptable systems that emphasize flexibility and run-time adaptability via a meta-modeling architectural style, covering a variety of different domains: from generic frameworks to represent and manipulate attribute com-

posite objects [30], to health information systems to collect clinical observations [34].

The process of gathering and organization of clinical data is typically achieved thanks to the help of instruments like *Electronic Health Record Systems (EHRSs)* [46] and *Clinical Decision Support Systems (CDSSs)* [14].
`EHR system`: is a software system to record, retrieve, and manipulate repositories of retrospective, concurrent, and prospective information regarding the health status of a subject-of-care.

Modeling and collecting heterogeneous clinical data via an EHR system have many challenges:

- Structured vs. Free-text data: Clinical information can be designed in a mixed solution combining well-defined structures and legible text that suits human natural language;
- Adaptability to different contexts-of-use: The main differences are in terms of which kind of information is managed, which level of granularity is used to represent clinical data, and which subjects-of-care are involved;
- Domain expert empowerment: Medical experts should change system structure and behavior, improving system maintainability and reducing efforts and delays induced by the intermediation of ICT experts;
- Interoperability at different levels of abstraction: Sharing information between different users or departments/hospitals is essential to achieve a high level of quality about health services provided to a subject-of-care;
- Requirements evolution over time: Medical domain is characterized by high volatility in terms of medical concepts needed to be taken into account.

For these reasons, the usage of *Adaptable Systems* in healthcare context is particularly appropriate.

The reference standard for the clinical records is `Open-EHR` [11] [10] which is defined through archetypes.
In addition to that, `Health Level-7` (HL7) refers to a set of international

standards, guidelines, and methodologies for the exchange, integration, sharing, and retrieval of electronic health information that supports clinical practice and the management, delivery and evaluation of health services [26].

3.1.1 The Observations & Measurements analysis pattern

The *Observations & Measurements* analysis pattern [34] implements the reflection principle specializing the abstraction for the case of clinical process. This pattern mimics the separation in two layers of the *Reflection* pattern, that in this context are defined as:

- `operational` level: contains the *observations* relative to medical concepts
- `knowledge` level: contains the *observation types* representing the fixed medical knowledge and defines the configuration of the operational level.

A classical modeling approach would record the information of an object as its attributes. If we examine the case of a single person data, its height could be stored as a simple integer linked to such entity. Specifically, height is a *quantity*, that combines a number with a unit of measure: modeling both as objects we can define, for example, methods to convert quantities. The quantity object (Fig. 3.1) can be modeled with two attributes:

- value: Number
- unit: Unit

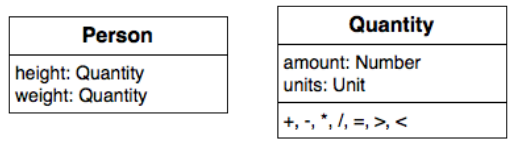


Figure 3.1: Person, Quantity and Unit of Measurement

In cases where is necessary to record a lot of information about an object, representing these as properties can lead to a bad programming anti-pattern

like the *God Object*, because its attributes and the operations inside the class would grow a lot.
A possible solution is to treat as object every possible measurement and to introduce types for those objects, called *phenomenon types*. In this way the person will only have a single attribute *measurement* that will contain a multitude of associations between measurements (quantity) and phenomenon types (Fig. 3.2).

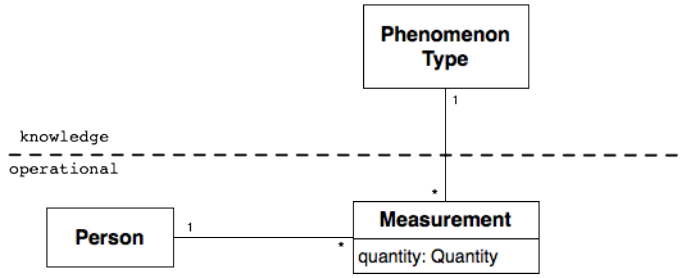


Figure 3.2: Separation in two levels to represent measurements and phenomenon types

The measurements can be added daily at the *operational level*, while phenomenon types will be created less frequently at the *knowledge level*, because they represent the knowledge of what is measured.

The concept of *observation* extends the pattern even more, allowing to manage not only quantitative information, but also *qualitative* ones, like gender, blood type etc. They will have a *category* instead of a quantity. We can then define a new type, the *observation*, that acts like a super-type for a measurement and a qualitative observation (Fig. 3.3).

The gender of a subject is then an instance of the phenomenon type, while male/female are category instances.
To enforce the usage of some categories only on certain phenomenon types a relationship between category and phenomenon type must be added. This can be achieved moving the category at the knowledge level and naming it *phenomenon* (Fig. 3.4).

For example the fact that a subject has “blood type A” is stored as an *Observation Category* of the subject, which *Phenomenon* is “group A”. Said phenomenon is linked to the *phenomenon type* “blood type”.

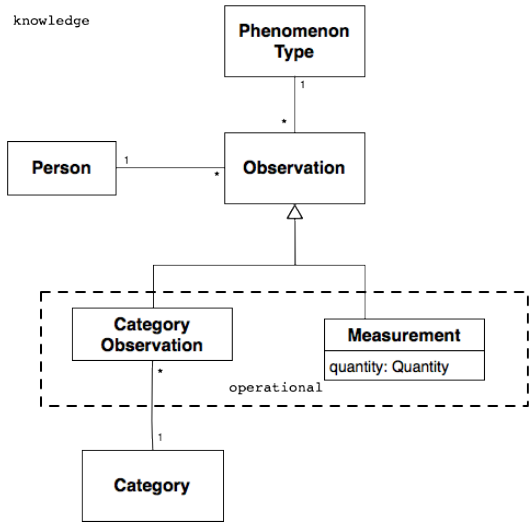


Figure 3.3: Qualitative observation and quantitative measurement

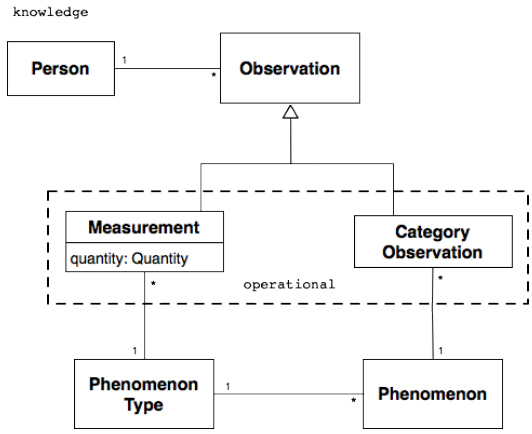


Figure 3.4: Phenomenon

Note that the Observations & Measurements pattern exploits the same *typesquare*'s principles to separate clinical Observations from their PhenomenonTypes.

3.2 The Empedocle EHR system

As seen in sect.3.1, an *EHRs* has some conflicting requirements, such as the need to represent data in a structured way, to integrate and verify information coming from different sources, and to adapt to users with different levels of specialization. Moreover, an *EHRS* should be adaptable to diversified contexts that require deep customization of information's structure.
We propose an architecture based on meta-modeling principles that implements an EHR system characterized by adaptability and changeability as primary requirements.

The implemented EHR system, named *Empedocle*, combines the basic functionalities that comprise the expected commodity level of any EHR system, with specific requirements posed by an outpatient scenario, where a variety of medical specialities take part (Fig. 3.5).

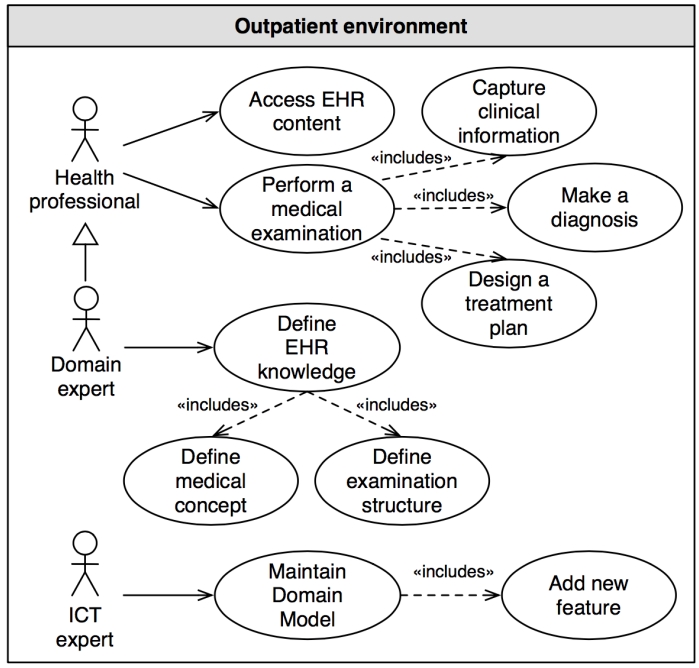


Figure 3.5: A typical outpatient scenario, specifying the major actors involved in the care process and their interaction with an EHR system.

In this section I describe how the *Reflection* architectural principles [17] can be implemented through a powerful combination of the *Observations & Measurements* analysis pattern [34] and the *Composite* pattern [36] to implement an EHR system able to deal with medical concepts and clinical data characterized by complexity and volatility.

3.2.1 A two-level meta-modeling EHR architecture

Fig. 3.6 represent the UML class-object diagram that provides a high-level specification of the domain model implemented in the core of the *Empedocle* EHR system. This pattern-based architecture is in use since more than 4 years in various units of the major hospital of Tuscany Region (Careggi hospital, in Florence). Its use in more than one real-world scenario proved how the meta-modeling approach has been useful under different aspects. Especially, the flexibility, re-configurability and extensibility attributes allowed on one hand to create instances of the software in different contexts in a few hours, on the other to directly involve doctors and domain experts in the configuration process, without prior definition of all the context concepts.

As can be seen, the domain logic of *Empedocle* is split in two layers so as to support dynamic adaptation of the system in response to changing requirements. On the one hand, medical concepts are represented in a so-called *knowledge level*. On the other hand, clinical data are represented in a so-called *operational level*. These principles are obtained through a pattern-oriented design, addressed in the architectural perspective by the Reflection pattern [17], and in the conceptual perspective by the Observations & Measurements pattern [34].

Specifically, during each `Examination` in *operational level*, a series of clinical information items like signs (i.e., objective evidences noticed), symptoms (i.e., subjective evidences reported by patient), and other clinical observations are captured by health professionals as instances of the `Fact` class.
Conversely, all medical concepts can be defined directly by domain experts as instances of the `FactType` class in *knowledge level*.
Hierarchical structured data resulting from repeated aggregation of basic observations and measurements can be cast in the representation through a mix-in of the *Composite* pattern [36], by allowing an observation or measurement be implemented as a collection of references to other observations or measurements.

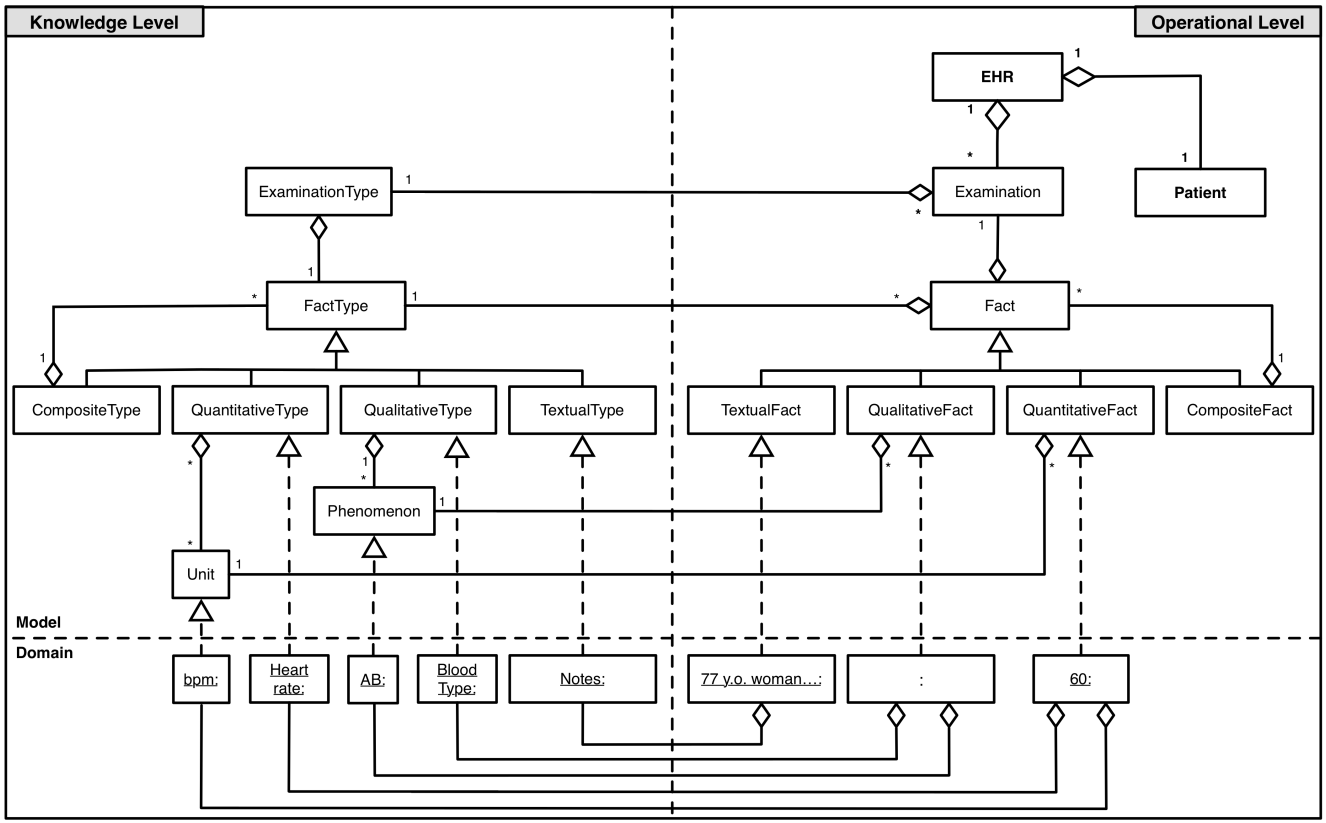


Figure 3.6: The two-level domain model of the *Empedocle* EHR system

Accordingly, four different categories of knowledge can be identified:

- `TextualType`, for free-text information (e.g., patient's *anamnesis*);
- `QualitativeType`, for values in a finite range of acceptable *Phenomena* (e.g., *blood type* with groups *A*, *B*, *AB*, and *0*);
- `QuantitativeType`, for quantities with a specified set of acceptable *Units* (e.g., *heart rate*, measured in *beats-per-minute*);
- `CompositeType`, for composing `FactTypes` in a hierarchical structure (e.g., *vital sign* including *temperature*, *blood pressure*, *heart* and *respiratory rate*).

The same categories can be identified at the operational level:

- `TextualFact`
- `QualitativeFact`
- `QuantitativeFact`
- `CompositeFact`.

`ExaminationType` class represents the structure of an `Examination` in terms of which `FactTypes` (and related `Facts`) have to be considered during a medical examination.

To give another example of the utilization of the meta-modeling architecture, I reported in Fig. 3.7) the domain model of the *Regional Cancer Registry*. In this Registry information about the individuals that live in a specific territory are gathered, in particular diagnosed neoplasms and clinical sources that testify the presence of the disease.
Differently from Empedocle EHR system, the major point of variation for this scenario arises not in the necessity of addition of new data to collect (i.e., add instances of `FactType`) but in the requirement to deals with new kind of clinical sources (i.e., add instances of `SourceType`).
Furthermore, this scenario needs to address the scalability issue because of the increasing volume of data, more than in Empedocle's case.

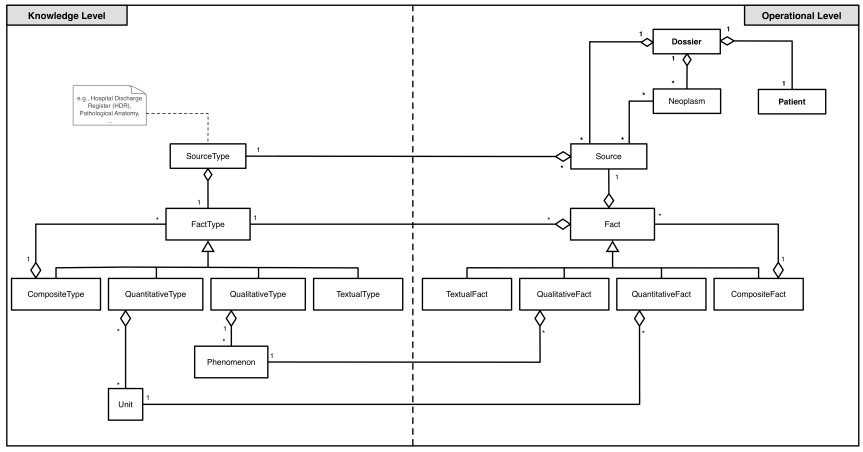


Figure 3.7: The two-level domain model of the *Regional Cancer Registry* system

In the same way, we applied the same paradigm to develop systems that deviate from the healthcare context, for example a system for monitoring the state of various types of buildings after a natural disaster, and a system for representing different University courses of study.

Chapter 4

Performance evaluation of a J2EE architecture

This Chapter introduces the Java EE specifications, with references to consolidated architectural patterns. Late it will be illustrated how a meta-modeling architecture can be implemented as a web application, using j2ee technologies, especially referring to Empedocle EHR System. Finally, some performance problems are shown, mainly related to the lower tiers of the j2ee stack, that are the ones responsible of data persistence.

4.1 J2EE

Java EE provides a standard way to handle transactions with Java Transaction API (JTA), messaging with Java Message Service (JMS), or persistence with Java Persistence API (JPA). Java EE is a set of specifications intended for enterprise applications. It can be seen as an extension of Java SE to facilitate the development of distributed, robust, powerful, and highly available applications. The J2EE platform is a multi-tiered system (Fig. 4.1), with various advantages :

- force separation of user interface logic and business logic, following Model-View-Controller approach;
- business logic sits on small number of centralized machines (may be just one);

- it's easy to maintain, to manage, to scale thanks to the loosely coupling of tiers.

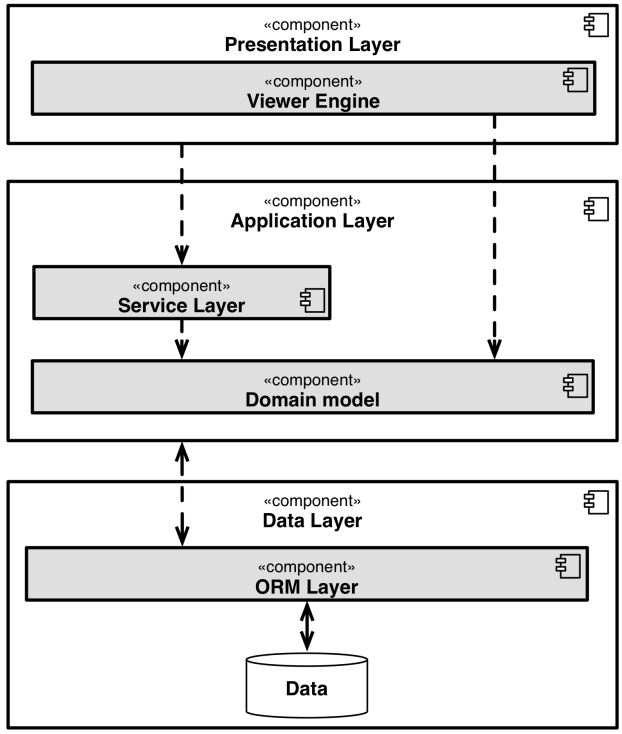


Figure 4.1: Typical architecture of a J2EE application.

In common practice, the tier separation of the J2EE architecture is realized through several architectural patterns [23]. In Fig. 4.2 some of the main ones are shown along as some of the technologies used to realize them, in the context of web applications:

- `Presentation Layer`: displays information to the user and handle user request (mouse clicks, keyboard hits); provides services, HTTP requests, command-line invocations, batch API; can be a command-line or text-based menu system, a rich-client graphics UI (e.g., based on Swing), or an HTML-based browser UI (xhtml, jsf...).

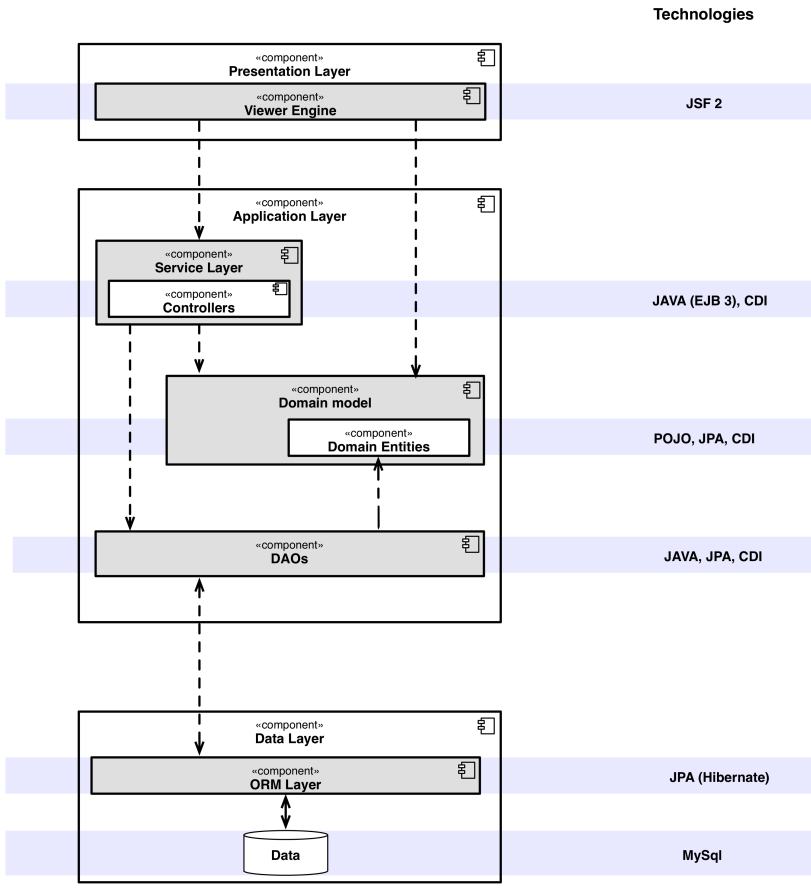


Figure 4.2: Architectural patterns and technologies in J2EE architectures.

- `Controllers (or Business Logic)`: play the work that the application needs to do for the domain you're working with; carries out calculations based on inputs and stored data, validation of data from the presentation, selection of data source logic to dispatch, reacting to commands from the presentation. Business Logic directly depends on use cases, decoupling Domain Entities from use cases. These objects are made of java + CDI annotations

- `Domain Entities (or Business Objects)`: represent concepts of do-

main and contain indications of how to store these objects in the underlying layers.
From the technological point of view these objects are made of Java, JPA annotations, CDI annotations.

- `Data Access Objects (DAOs)`: encapsulate access to data source, decoupling business logic from persistence operations. The DAO object manages the connection with the data source to retrieve and/or persist data. Since the interface exposed by the DAO doesn't change when the underlying data source changes implementation, this pattern allows the Data Access Object to adapt to different implementation schemes without affecting in any way the client or the business layer components. Substantially the DAO works as an adapter between the business tier components and the data source.
  From the technological point of view these objects are made of Java, JPA operations, CDI annotations.

- `Data Layer`: consists of Object-Relational Mapping (ORM), a data-access layer that resolves the *object-relational impedance mismatch* [44], and Data Storage (e.g., relational databases, noSql databases, in-memory databases,..).

4.2 Performance testing

Performance testing can be defined as the testing practice performed to determine how a system performs in terms of responsiveness and stability under a particular workload. Performance of web applications can be determined in terms of Availability, Response Time, Throughput, Utilization and Latency [55].

Load testing and stress testing are the most common kind of performance tests.

Load testing is the simplest form of performance testing. A load test is usually conducted to understand the behaviour of the system under a specific expected load. This load can be the expected concurrent number of users on the application performing a specific number of transactions within the set duration. This test will give out the response times of all the important business critical transactions. The database, application server, etc.

are also monitored during the test, this will assist in identifying bottlenecks in the application software and the hardware that the software is installed on.

Stress testing is normally used to understand the upper limits of capacity within the system. This kind of test is done to determine the system's robustness in terms of extreme load and helps application administrators to determine if the system will perform sufficiently if the current load goes well above the expected maximum.

There are a lot of tools to create and simulate end user work flows (e.g., HP LoadRunner, NeoLoad, Apache JMeter, Selenium WebDriver, Rational Performance Tester, Silk Performer and Gatling [66]). Most of the tools perform "Record & Replay" tests, where the testing tool captures all the network transactions which happen between the client and server.

According to the Microsoft Developer Network [25] the Performance Testing Methodology consists of the following activities:

- Identify the *Test Environment*
- Identify *Performance Acceptance Criteria* (e.g., response time, throughput, and resource-use goals and constraints)
- *Plan* and *Design* Tests (scenarios, variability, metrics to be collected)
- *Configure* the Test Environment
- *Implement* the Test Design
- *Execute* the Test
- Analyze *Results*, make a *Tuning Change* and *Retest*.

4.2.1 Performance evaluation of a meta-modeling architecture

Our contribution is focused on *response time*, one of the most important parameter to reflect the quality of a Web Service. Response time is the total time it takes after the client sends a request till it gets a response. This includes the time the message remains in transit on the network, which can't be measured exclusively by any load-testing tool. So we're restricted to testing Web Services deployed on a local machine.

As seen in Sec. 4.1, the loose coupling between the J2EE architecture layers encourages the experimentation of different technologies and development solutions to measure the various effects on the performance of the web application. Specifically, we addressed performance issues occurring in Data Layer and in components responsible for communications with this level.

Referring to Performance Testing Methodology in previous section, testing activities were limited to last steps, since the experimentation has relied on the analysis of an application under production (Empedocle EHR System). The acceptance criteria were already well defined by what end users expect from the software system and our goal was to carry out a performance-driven software refactoring without altering system's functionalities [35]. In addition, having full access to the source code, our approach to the profiling and the identification of the major performance bottlenecks was able to follow a white-block strategy, without using external tool to evaluate the overall system.

While referring to our Empedocle EHR System for the sake of experimentation concreteness, most of the subsequent discussions about the development of this kind of model as well as about its impact on system performance are more generally applicable to most schemes that can be designed in the style of the *Reflection* architectural pattern.

In fact, design by patterns does not account for performance as first-class requirement, and naturally incurs in well-known performance anti-patterns [7, 69], which may become crucial when *volume* and *variety* must meet also *velocity* [27]. These drawbacks are largely exacerbated when the domain logic is persisted over a relational storage layer, due to the nature of the domain model and its mismatch with the relational tier [6].

In general, the persistence of a domain model with complex structure into a relational database comes with a number of performance penalties, that translate in longer time required for key persistence operations. These issues can be partially mitigated with ad-hoc optimizations in the design of the relational database [65], pertaining to the choice of a particular representation for class inheritance, the use of auxiliary tables to store additional information, and the smart use of data fetching.

The interposition of an object-relational mapping (ORM) layer between

the domain logic and the storage layer can mitigate this problem. In the practice of development of Java enterprise applications, Java Persistence API (JPA) specification represents a mature and state-of-the-art ORM solution which grants many benefits [15]. First of all, it allows to persist domain classes with a minimal boilerplate code, thanks to simplified annotation facilities. Also, it provides full integration with the Java application stack, composed by other technologies such as EJB (for encapsulating the business logic) and CDI (for implementing the *Inversion of Control* pattern [53]). However, JPA further increases the degree of indirection and this can have negative effects on the system performance, also due to the loss of design control on the impact that domain logic operations have on the storage process.

In the following sections, I report the evaluation of different approaches aimed at performance improving in the case of a meta-modeling architecture for data management.

Chapter 5

Performance anti-patterns roadmap for a meta-modeling architecture

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 [8], and hard to develop [83]. 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 [70]. 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 section, I address performance engineering of a meta-modeling architecture through a suite of anti-patterns and refactoring actions, aimed at improving performance while supporting and preserving reusability and maintainability.

5.1 Performance anti-patterns and ORM's performance

Performance issues have been widely addressed in the literature of performance anti-patterns, which describes recurring problems with significant impact on performance. Note that an anti-pattern is not only a simple bad habit, bad practice, or bad idea: it is a commonly used process, structure, or pattern of action that despite initially appearing to be an appropriate and effective response to a problem, has more bad consequences than good ones. In [69] 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 [75] and [60], followed by [7], 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 [78] and [67]. Hibernate performance is also discussed in [77] and [48], comparing with outdated solutions of object-oriented databases through benchmarks.

5.2 Reflective Architecture: growing model complexity

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 the domain model grows in complexity. For the sake of concreteness, but without loss of generality, we refer to the case of an Electronic Health Record (EHR) system [46], named *Empedocle* [62], in which the complexity is related to the specialty under consideration. For instance, performance was not an issue for the context of Ophthalmology, whose basic examination results in a lightweight tree-like structure, where only few tens of observations were collected for each patient.

Table 5.1 highlights that the complexity of object models is characterized by the number of nodes, leaves and depth of the tree-like structure used to represent a medical examination, as reported in Fig.5.1. In particular, the

Table 5.1: Comparison of Ophthalmology and Cardiology examination structures.

Specialty configuration	**Nodes (#)**	**Leaves (#)**	**Max depth**	**Average depth**
Ophthalmology	64	45	5	3.7
Cardiology	639	495	7	3.6

number of nodes and leaves in a Cardiology examination is ten times bigger than values for an Ophthalmology examination.

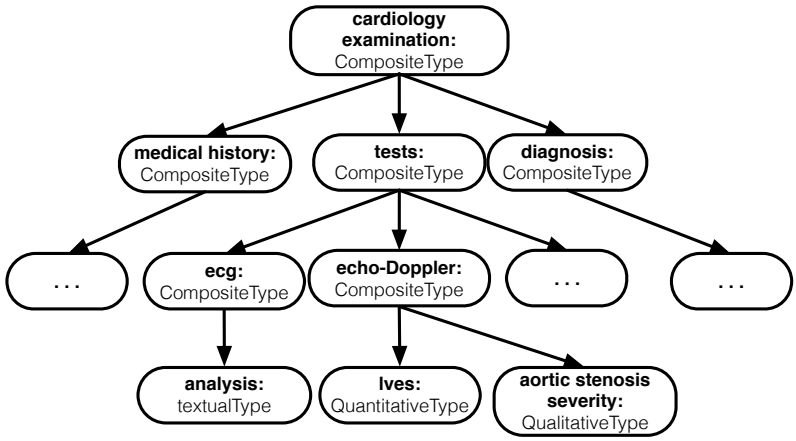


Figure 5.1: A simplified cardiology examination represented in Empedocle as a tree-like structure. Rounded boxes represent instances of `FactType` and define medical concepts that are to be taken into account during a typical cardiology examination. In the Operation Level, a similar structure of `Facts` will be instantiated to represent collected clinical observations.

Performance limitations were observed after the transition from the Ophthalmology's context to Cardiology, in two main scenarios of interaction,

namely "*performing a medical examination*" and "*accessing the patient's EHR content*" (see Fig. 3.5).

In the first scenario, in which the user *performs a medical examination*, the examination structure is first loaded; then clinical information corresponding to observed facts are filled by the user; finally, the examination is marked as concluded and the system stores clinical data collected during the examination into the EHR of the current patient.

In the second scenario, the health professional *accesses the patient's EHR content* to load a performed medical examination and display collected clinical information.

Loading the examination structure and loading or saving the clinical information can be a very expensive task, especially if the number of fields that form the examination is considerable.

The scenarios under exam have been selected as performance bottlenecks via measurements of the main use cases after the user/client sends a request till he gets a response. 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, so as to evaluate time and number of queries executed for completing each tasks.

5.3 Model refactoring for performance improvement

The analysis presented in Sect. 5.2 suggests that performance issues can be related to the number and complexity of queries that underlie the execution of each task, due to the strongly characterization of abstraction on the architecture under consideration.

We identified four performance *anti-patterns* about mapping strategies. Specifically, we detected that some choices in mapping strategies can provide tangible benefits at the application level, by facilitating navigability between objects and by simplifying the code, but conversely they can also affect data access performance, and increase the number of queries and their complexity.

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 as, for instance, in [70]:

- Problem: the recurrent situation that causes negative consequences
- Context: explanation of the context where we can find the anti-pattern in a meta-modeling architecture
- Solution: how can we avoid, minimize or refactor the anti-pattern
- Sample Code: an example of the anti-pattern related to the problem and an example of the proposed solution (in Java language with JPA annotations)

5.3.1 Mapping inheritance structures: joins explosion

Problem

Hierarchical structures can create problems when they are mapped to a relational database which does not natively support inheritance. To overcome this lack, different techniques were introduced, as reported in [6]:

- map the entire class hierarchy to a single table;
- map each concrete class to its own table;
- map each class to its own table;
- map the classes into a generic structure.

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 approach requires to create separate tables for each entity and related direct descendants in the object oriented hierarchy with one-to-one relationships: the table corresponding to a generic `Fact` (or `FactType`) contains one column for each attribute common to its children, while each table corresponding to a subtyped entity contains columns specific to its own attributes and one extra column as foreign key for uniquely identifying a row in the hierarchy.

This strategy offers various well-known advantages in the perspective of software architecture [6], 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 every sub-class 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).

Context

In the case of a reflective architecture, as depicted in Fig. 3.6, `Facts` (or `FactTypes`) are specialized in different kind of entities to represent various concepts and information. 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.

Solution

In order to overcome performance issues caused by the joined approach, a mapping strategy based on a *single table* approach should be preferred (see Fig. 5.2). 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 (i.e., `@DiscriminatorValue` in Listing 5.2). Single table strategy collects all data are 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 a performance perspective, the reduction of queries complexity and limited use of joins translate into a significant decrease of time to access the data. 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. This is, in fact, the case of Empedocle.
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.

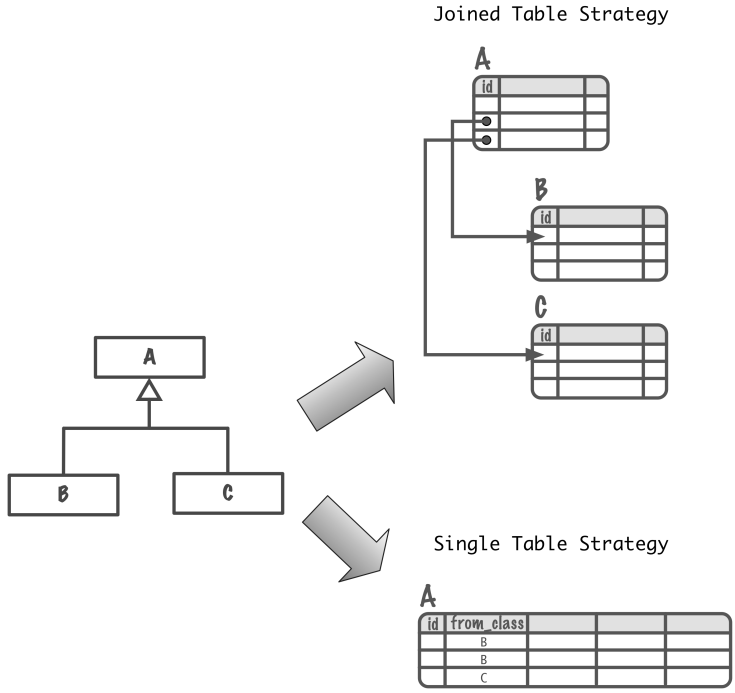


Figure 5.2: Mapping inheritance in Joined or Single Table strategy.

Sample Code

Mapping inheritance structures

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@Table(name = "facts")
public abstract class Fact {

  private Long id;
  ...

}

@Entity
@Table(name = "textual_facts")
@PrimaryKeyJoinColumn(name="FACT_ID")
public class TextualFact extends Fact {
```

```
15
16    private String text;
17    ...
18  }
19
20  @Entity
21  @Table(name = "qualitative_facts")
22  @PrimaryKeyJoinColumn(name="FACT_ID")
23  public class QualitativeFact extends Fact {
24
25    private Phenomenon phenomenon;
26
27    @ManyToOne(cascade = CascadeType.REFRESH)
28    @JoinColumn(name = "phen_id")
29    public Phenomenon getPhenomenon() {
30      return phenomenon;
31    }
32    ...
33  }
34  ...
```

Listing 5.1: anti-pattern: *joined table strategy* to map inherithance of TextualFact, QualitativeFact, QuantitativeFact, and CompositeFact from Fact class

```
1
2   @Entity
3   @Table( name = "facts" )
4   @Inheritance( strategy=InheritanceType.SINGLE_TABLE )
5   @DiscriminatorColumn(
6     name= "from_class",
7     discriminatorType=DiscriminatorType.STRING )
8   public abstract class Fact {
9
10    private Long id;
11    ...
12  }
13
14  @Entity
15  @DiscriminatorValue( "TX" )
16  public class TextualFact extends Fact {
17
18    private String text;
19    ...
20  }
21
22  @Entity
```

```
23 @DiscriminatorValue( "QL" )
24 public class QualitativeFact extends Fact {
25
26   private Phenomenon phenomenon;
27
28   @ManyToOne
29   @JoinColumn( name = "phen_id" )
30   public Phenomenon getPhenomenon() {
31     return phenomenon;
32   }
33   ...
34 }
35 ...
```

Listing 5.2: solution: *single table strategy* to map inherithance of TextualFact, QualitativeFact, QuantitativeFact, and CompositeFact from Fact class

5.3.2 Mapping hierarchical structures: queries explosion

Problem

In Sect. 5.3.1 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 sub-tree?

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 loading in memory the whole structure requires one query per nodes, growing 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.

Solution

To optimize the amount of queries, ancestor-descendant relations were added in the hierarchical structure, where each node of the hierarchy maintains a list of its ancestors (note that each node is also ancestor of itself). In so doing, the hierarchy can be represented as an ordered directed tree (see Fig. 5.3), where only one query is required for retrieving all the information contained in the whole structure. Moving from a per node query to a single query results in a considerable gain of performance, because of parsing, compilation and optimization times spent for each of them.

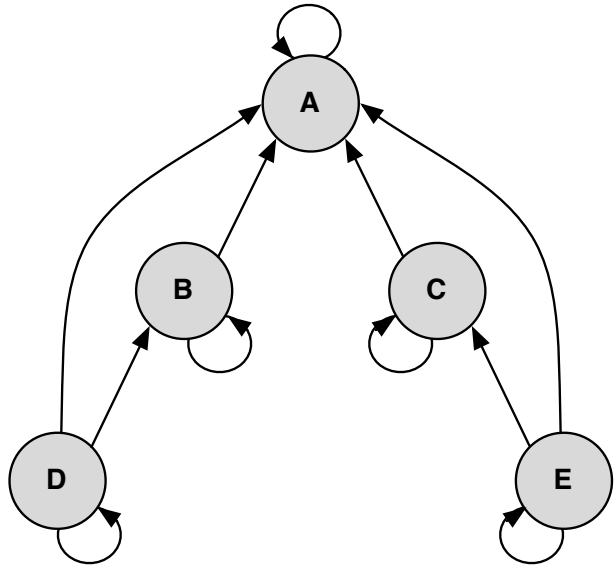


Figure 5.3: Ancestor-descendant relations in a hierarchical structure.

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: $A = 1 + \sum_{d=1}^{D} d * n_d$.

Sample Code

Mapping hierarchical structures

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@Table(name = "facts")
public abstract class Fact {
  private Long id;
  private FactType type;
  private List<Fact> facts = new ArrayList<Fact>();

  @OneToMany(cascade = CascadeType.ALL)
  @JoinColumn(name = "cmp_fact_id", nullable = false)
  public List<CompositeFactItem> getItems() {
    return items;
  }
}
```

Listing 5.3: anti-pattern: retrieving nodes in a sub-tree through *tree traversal*

```

@Entity
@Table( name = "facts" )
@Inheritance( strategy=InheritanceType.SINGLE_TABLE )
@DiscriminatorColumn(
  name= "from_class",
  discriminatorType=DiscriminatorType.STRING )
public abstract class Fact {

  private Long id;
  private FactType type;

  private Set<Fact> ancestors;
  private Set<Fact> descendents;

  @ManyToMany( fetch=FetchType.LAZY )
  @JoinTable(
    name = "fact_ancestors",
      joinColumns = { @JoinColumn( name = "fact_id",
    referencedColumnName="id" ) },
      inverseJoinColumns = { @JoinColumn( name = "
    ancestor_fact_id", referencedColumnName = "id") } )
  protected Set<Fact> getAncestors() {
    return ancestors;
  }

```

```
25   @ManyToMany( mappedBy = "ancestors", fetch=FetchType.LAZY )
26   protected Set<Fact> getDescendents() {
27     return descendents;
28   }
29
30 }
```

Listing 5.4: solution: retrieving nodes in a sub-tree through *ancestors strategy*

5.3.3 Mapping entity associations: fetching overloading

Problem

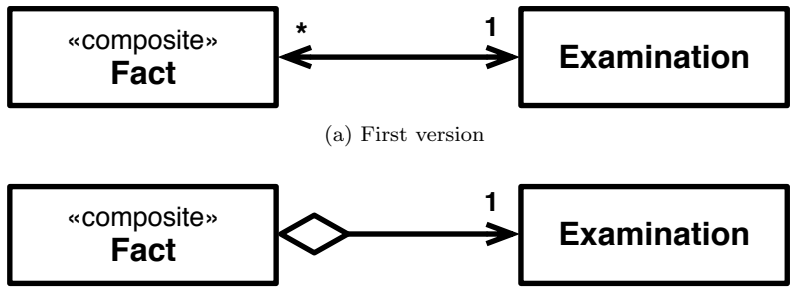
In a domain model, associations represent the relationships between classes. While object-oriented languages represent associations using object references that are navigable, in the relational world, an association is represented as a foreign key column that it is not a directional relationship by nature. In order to smooth the object/relational paradigm mismatch, association mapping plays a lead role.

Object Relational Mapping 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 application, suggests to turn almost all associations navigable in both directions in queries [63]. However, in some contexts, this approach can result in retrieving data not really necessary in every use-cases, leading to memory overload.

Context

Referring to Fig. 3.6, 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.



(a) First version

(b) Refactored version

Figure 5.4: Bidirectional vs unidirectional relationships.

Solution

Moving from bidirectional (see Fig. 5.4a) to unidirectional association (see Fig. 5.4b) 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 advantages. 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 facilitates testing and future refactoring interventions. In a performance perspective, queries are more efficient because only objects really necessary are loaded and additional information can be retrieved through dedicated queries. 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 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 association (e.g., composition), in order to emphasize the dependency of the contained class to the life cycle of the container class.

Sample Code

Mapping entity associations

```
@Entity
@Table( name = "examinations" )
public class Examination {

  private Long id;
  private ExaminationType examinationType;
  private Patient patient;
  private List<Fact> facts = new ArrayList<Fact>();

  @OneToMany(cascade = { CascadeType.PERSIST, CascadeType.MERGE,
      CascadeType.REFRESH })
  @JoinColumn(name = "exam_id")
  public List<Fact> getFacts() {
    return facts;
  }
  ...
}


@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@Table(name = "facts")
public abstract class Fact {

  private Long id;
  private FactType type;
  private Examination examination;

  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "exam_id")
  public Examination getExamination() {
    return examination;
  }
  ...
}
```

Listing 5.5: anti-pattern: *bidirectional relationships* between Examination and Facts that belong to it

```
@Entity
@DiscriminatorValue( "EX" )
public class Examination {

  private ExaminationType type;
```

```
6
7   @ManyToOne( fetch = FetchType.EAGER )
8   @JoinColumn( name = "exam_type_id" )
9   public ExaminationType getType() {
10    return type;
11  }
12  ...
13 }
14
15 @Entity
16 @Table( name = "examinations" )
17 @Inheritance( strategy=InheritanceType.SINGLE_TABLE )
18 @DiscriminatorColumn(
19   name= "from_class",
20   discriminatorType=DiscriminatorType.STRING )
21 @DiscriminatorValue( "BS" )
22 public class Examination {
23   ...
24 }
25
26 @Entity
27 @Table( name = "facts" )
28 @Inheritance( strategy=InheritanceType.SINGLE_TABLE )
29 @DiscriminatorColumn(
30   name= "from_class",
31   discriminatorType=DiscriminatorType.STRING )
32 public abstract class Fact {
33   private Type type;
34   private Examination examination;
35
36   @ManyToOne( fetch=FetchType.LAZY )
37   @JoinColumn( name = "examination_id" )
38   public Examination getExamination() {
39     return examination;
40   }
41   ...
42 }
```

Listing 5.6: solution: *unidirectional relationships* between Examination and Facts that belong to it

5.3.4 Inheritance vs. aggregation: fetching overloading

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. 5.3.1 and in Sect. 5.3.2.

Context

In the meta-modeling architecture of Fig. 3.6, 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 (e.g., for blood type, allowable phenomena are *0*, *A*, *B*, and *AB*). 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. However, sometimes, a plain string is not enough to fully describe a `Phenomenon`, and it is necessary to encode extra and more structured information, in order to deal with this special category of phenomena.

A typical example is the International Classification of Diseases (ICD) [58], which represents the international standard diagnostic tool for epidemiology, health management and clinical purposes, maintained by the World Health Organization. This standard is designed to map diseases and other health problems to generic categories together with specific variations, assigning an ICD code, depending on the standard version (e.g., in the ICD-9-CM version, 427.31 corresponds to atrial fibrillation in the Cardiac dysrhythmias category).

At the beginning, ICD-9-CM codes were treated as specializations of `Phenomenon` types (e.g., `ICD9CM`), so as to support their exploitation inside qualitative clinical information (see Fig. 5.5a). Increasing the number of

special phenomenon types to deal with (e.g., different ICD versions to manage), inheritance-based model ends up to include several classes derived from `Phenomenon`, very different from each other, driving to more complex (and slow) queries.

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 (Fig.5.5b). In this way, classes corresponding to special phenomenon categories (e.g. `ICD9CM`) 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. In terms of performance, queries concerning phenomena become simpler, faster and more efficient, due to the smaller information retrieved.

Sample Code

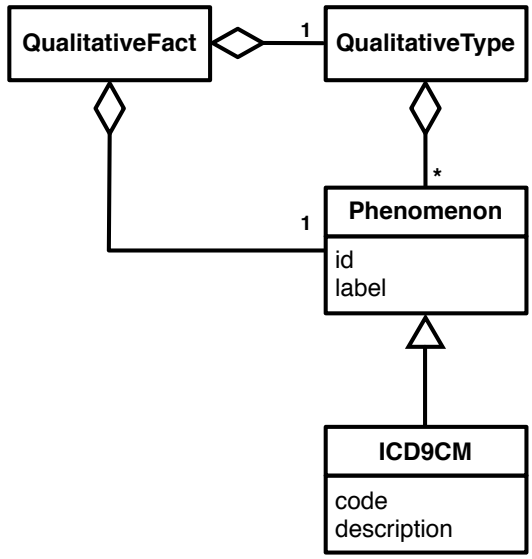
Inheritance vs. aggregation

```
@Entity
@Table(name = "phenomena")
public abstract class Phenomenon {

  private Long id;
  private String label;
  ...
}

@Entity
@Table(name = "icd9cm")
public class Icd9cm extends Phenomenon{

  private Long id;
  private String code;
  private String description;
  ...
}
```

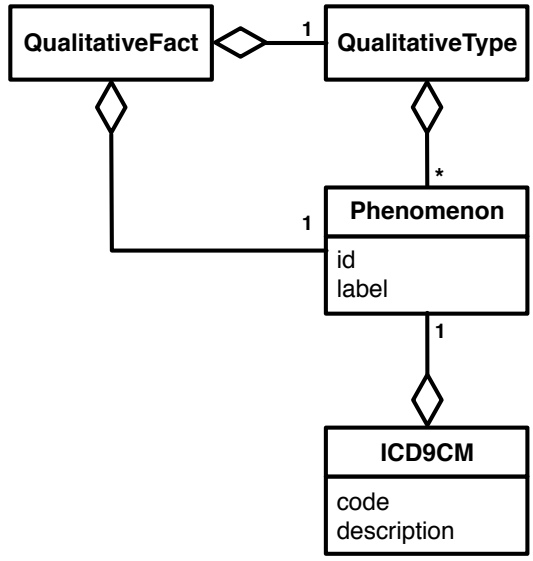
Listing 5.7: anti-pattern: *inheritance strategy* to map specific phenomenon type

```
@Entity
@Table( name = "phenomena" )
public class Phenomenon {

  private Long id;
  private String label;
  ...
}

@Entity
@Table(name = "icd9cm")
public class Icd9cm {

  private Long id;
  private String code;
  private String description;
  private Phenomenon phenomenon;

  @ManyToOne( cascade = { CascadeType.ALL } )
  @JoinColumn( name = "phen_id" )
  public Phenomenon getPhenomenon() {
    return phenomenon;
  }
  ...
}
```

Listing 5.8: solution: *aggregation strategy* to map specific phenomenon type



(a) First version



(b) Refactored version

Figure 5.5: From inheritance to aggregation.

5.4 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* [62], presently in use in various clinics at AOUC, the main hospital in Florence.

5.4.1 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 (see Table 5.1). In particular, the number of nodes and leaves in a Cardiology examination is ten times bigger than values for an Ophthalmology examination.

As highlighted in sect. 5.2 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.

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 [69] 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).

5.4.2 Results

The performance of the refactored model obtained after removing the four anti-patterns described in Sect. 5.3 was tested through the measurements and comparison of time to perform the two scenarios discussed in Sect. 5.4.1, so as to evaluate the performance gain obtained applying the proposed refactoring solutions.

Table 5.2 and Table 5.3 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 5.2 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 5.3 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 5.2 and Table 5.3 are related to the *UC2* scenario.

Table 5.2: 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

The comparison of results in Table 5.2 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

Table 5.3: 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

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 5.2, 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 5.3), 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 5.3 by factor of 14.1 for Ophthalmology and 8.3 for Cardiology.

Turning to details, the solution provided in Sect. 5.3.1 decreases the number of join operations during the *UC2*; furthermore, the refactoring proposed in Sect. 5.3.2 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. 5.3.3 is intended to avoid unnecessary references that carry an overload of objects in memory. Finally, moving from inheritance to aggregation as reported in Sect. 5.3.4 limits the amount of retrieved data when a lightweight `Phenomenon` associated to a `QualitativeFact` is requested.

5.4.3 Reproducibility of results

The experiments have been run using a medical examinations dataset, from a real-world scenario of the major hospital fo Florence.
The Table 5.1 contains the exact structure configurations of the two medical

examination types used to measure the performance of the use cases, as shown in Table 5.2 and Table 5.3. Whoever is interested in developing a model based on the Fig. 3.6, can easily reproduce the experiments configuring an ExaminationType following the structure in Table 5.1. Clinical data are protected by strict privacy rules, but it's possible to generate a test dataset of synthetic observations using random strings and quantities.

5.5 Discussion

Designing and implementing a software intensive system based on a 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 systems 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 research, 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.

The significant and positive impact of the proposed refactoring strategies was assessed through experimentation on an EHR in use at a major hospital of Tuscany, referring to real scenarios of the clinical practice consisting in the execution of a clinical examination and in the access to a patients medical history. 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.

Finally, since model refactoring to improve performance may affect other functional and nonfunctional system requirements (e.g. reliability, maintainability, etc.), consistency and tradeoffs among these properties must be taken into account by sw developers during this kind of intervention.

Chapter 6

Performance evaluation of different data models over NoSQL persistence layers

The previous chapter investigated about performance improving provided by refactoring actions applied to the reflective domain model and its mapping strategies. The recent rise of the NoSQL movement motivates investigation on the performance impact that new persistence approaches can bring in the model-driven re-engineering of a consolidated object-oriented software architecture. In this chapter, I report on the performance engineering of a three-tier web-application focused on the replacement of a relational + ORM persistence stack by two different NoSQL technologies. I describe how a reflection-based architecture can be modeled over the graph-oriented Neo4j [2] and the document-oriented MongoDB [1] databases, and discuss challenges in conversion of the original model in the other ones. Finally, I compare experimental performance results achieved by the different solutions with performance of the original relational implementation.

6.1 Not Only Sql (NoSQL)

As seen in previous chapters, the persistence of a domain model with complex structure into a relational database comes with a number of performance penalties, that translate in longer time required for key persistence operations. These issues can be partially mitigated with ad-hoc optimizations in the design of the relational database [65], pertaining to the choice of a particular representation for class inheritance, the use of auxiliary tables to store additional information, and the smart use of data fetching.
The interposition of an object-relational mapping (ORM) layer between the domain logic and the storage layer can mitigate this problem. In the practice of development of Java enterprise applications, Java Persistence API (JPA) specification represents a mature and state-of-the-art ORM solution which grants many benefits [15]. First of all, it allows to persist domain classes with a minimal boilerplate code, thanks to simplified annotation facilities. Also, it provides full integration with the Java application stack, composed by other technologies such as EJB (for encapsulating the business logic) and CDI (for implementing the *Inversion of Control* pattern [53]). However, JPA further increases the degree of indirection and this can have negative effects on the system performance, also due to the loss of design control on the impact that domain logic operations have on the storage process.

With the rise of the *Not Only Sql (NoSQL)* movement [73], other options in the design of the storage layer are now available, and provide various advantages, including reduced access time through the clustering of similar data [64], and increased adaptability to the variety and variability of data over time through the use of a *schemaless* structure.

This motivates the investigation on engineering the performance of existing applications by changing the storage schema from a relational + ORM persistence stack to a NoSQL solution, while preserving the domain logic structure. In particular, this subtends a problem of re-modeling content representation in the schema of some NoSQL technology and quantitatively evaluating the performance gain that can be attained. In so doing, different NoSQL paradigms are more or less close to the domain model and suited for its main operations [52, 76], and a pattern-based organization of the domain logic can drive the refactoring of the data model towards more efficient performance results.

6.2 Reflective Architecture: model details

To fully understand the following discussion it's necessary to make some statement about the architecture presented in Sect. 3.2. In particular, it's necessary to introduce some details that were omitted as a simplification.

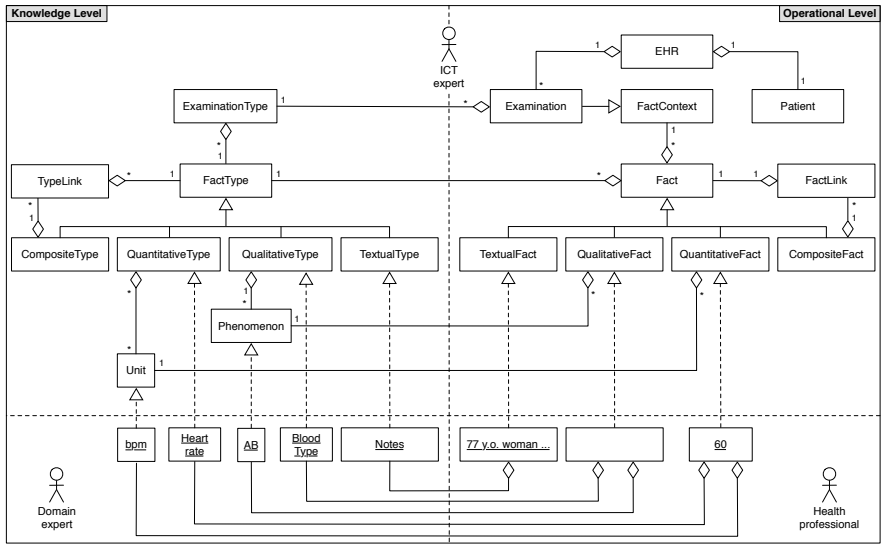


Figure 6.1: The domain model of the *Empedocle* EHR system with Type and Fact Links explicitated

The UML class-object diagram of Fig. 6.1 provides a high-level specification of the domain model implemented in the core of the *Empedocle* EHR system. Accordingly, four different categories of knowledge can be identified: `TextualType`, for free-text information (e.g. patient's *anamnesis*); `QualitativeType`, for values in a finite range of acceptable `Phenomena` (e.g. *blood type* with groups *A*, *B*, *AB*, and *0*); `QuantitativeType`, for quantities with a specified set of acceptable `Units` (e.g. *heart rate*, measured in *beats-per-minute*); and `CompositeType`, for composing `FactTypes` in a hierarchical structure through a *Composite* pattern implementation (e.g. *vital sign* including *temperature*, *blood pressure*, *heart* and *respiratory rate*). The same categories can be identified at the operational level: `TextualFact`, `QualitativeFact`, `QuantitativeFact`, and `CompositeFact`.

`ExaminationType` class represents the structure of an `Examination` in

terms of which `FactTypes` (and related `Facts`) have to be considered during a medical examination.

The main difference between the model previously referred is the introduction of `TypeLink` and `FactLink` associations, that specify the multiplicity of occurrence of each `Fact` in order to dynamically adapt the structure to multiple contexts-of-use that require a different number of instances to be recorded. In so doing, the reuse of already defined *named* `FactTypes` is supported, so as to avoid their proliferation, just referencing them in multiple parts of the structure. Alternatively, *anonymous* `FactType` instances (i.e., `FactTypes` that do not need to be referenced by others) can be used, and the definition of their structures is directly included inside the parent structure. As relevant consequence, as depicted in Fig. 6.2, the `FactType` structure will

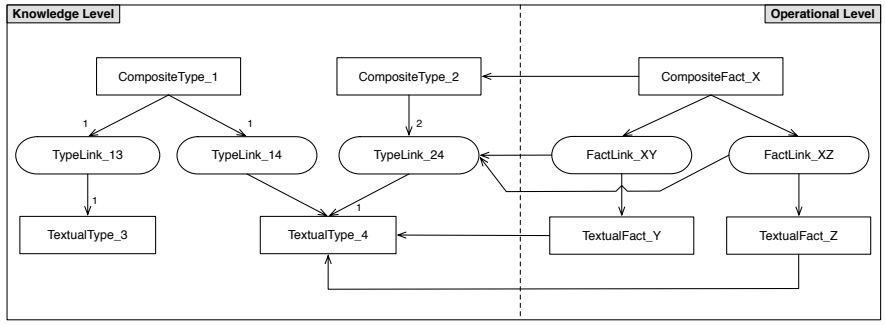


Figure 6.2: An example of `Examination` structure as represented using the domain model shown in Fig. 6.1: on the left, a direct acyclic graph obtained composing `FactTypes` and `TypeLinks`; on the right, a tree-like structure as resulting from the composition of `Facts` and `FactLinks`. Note that rectangles represent instances of `FactType` and `Fact` classes and define, respectively, medical concepts and clinical observations that are to be taken into account during a clinical examination. Rounded boxes represent instances of `TypeLink` and `FactLink` classes and are used to increase the expressiveness of each `Type`-to-`Type` and `Fact`-to-`Fact` association (for example, through the definition of its cardinality).

result in a direct acyclic graph, while the derived `Fact` structure will result in a tree, usually with an increased number of nodes due to the multiplicity attribute.

6.3 Modeling Reflection over a NoSQL persistence layer

In the common practice of software development, the persistence layer deals with retrieving data from and storing data to a relational data store, usually through the interposition of an ORM layer. In this kind of approach, the persistence model is largely determined by the object-oriented design of the domain logic.

By contrast, when persistence relies on a NoSQL solution, design gives space to alternative choices in the definition of the storage data model, which is, to a large extent, independent from the structure of object types. In fact, the absence of a fixed schema provides multiple options concerning the definition of the database structure, facilitating the representation of heterogeneous data characterized by high variability over time. The overall design results more flexible, but inevitably more complex and harder to understand for software developers used to deal with traditional relational databases [16]; it also requires to take into account some specific aspects so as to realize data migration in the most opportune way [64].

In the rest of this Section, I describe two new data models as implemented using different NoSQL technologies, *Neo4j* [2], and *MongoDB* [1]. The choice of these two technologies was made so as to experiment with their data structure and promising performance improvement [42, 81], and to compare graph- and document-oriented NoSQL solutions applied to the case of a reflection software architecture that combines the *Observations & Measurements* and *Composite* patterns, as described in Chap. 3. Finally, the validity of the proposed models is proved, in terms of integrity of persisted data and equivalence of data representations.

6.3.1 A model for Neo4j

Neo4j [2] relies on a *graph-oriented* structure, which can natively represent the domain logic of a reflection architecture, whose data structures are direct acyclic graphs and trees [79]. As a schemaless database, the data model in Neo4j is inherently defined by the *nodes* and *relationships* persisted in the database. Every node and relationship can also be characterized by an arbitrary number of *properties*.

From version 2.0, Neo4j developers tweaked its schemaless nature by introducing *labels* and *indexes*, two concepts that help modelling data in a more organized way, without losing the database original adaptability. Specifically, labels can be used to group together nodes, and each node can optionally be labeled with one or more text descriptions, and indexed to improve query expressiveness and flexibility. Moreover, indexes can be defined on properties of labelled nodes, to improve performance during query operations, similarly to the relational case. Both labels and indexes are optional.

In our concrete case, modeling the domain logic in Neo4j comes down to: *i)* identifying the node structure that forms the model; *ii)* defining the relationships between nodes; *iii)* defining the properties that characterize nodes and relationships, and *iv)* labeling with the appropriate qualifiers.

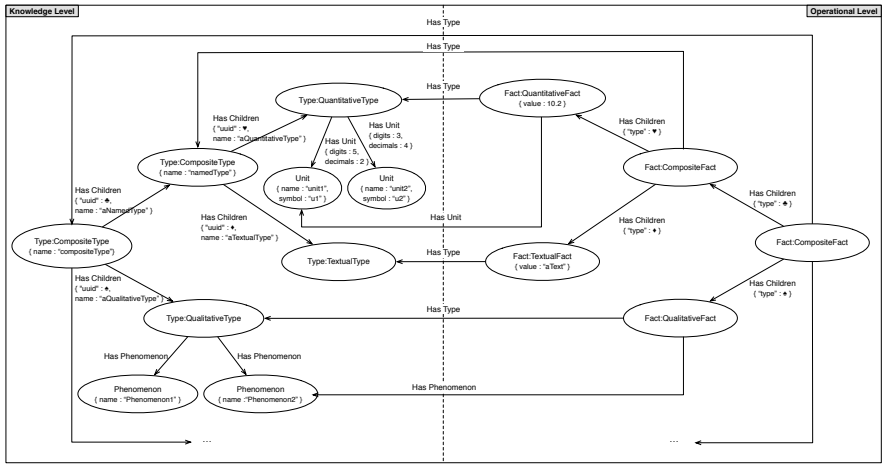


Figure 6.3: The representation of an instance of the domain model described in Sect. 6.2 on the graph model of Neo4j database. Oval shapes represent nodes, and arcs between nodes represent relationships, with labels written in bold, and properties reported between braces. For example, a *Type:CompositeType* node is characterized by multiple labels: the first one specifies that it is an instance of `FactType` class, the second one identifies its role in the hierarchy. The *Has Children* relationship identifies children nodes. For reasons of readability, *uuid* property values have been replaced with symbols.

Specifically, as depicted in the schema of Fig. 6.3, each class that is an entity in the original model has been represented as a node in the target model (i.e., `FactType` and `Fact` hierarchy classes, and `Phenomenon` and `Unit` classes). The resulting nodes have been labeled with a correspondent qualifier and, in addition to that, nodes that are part of the `FactType` and `Fact` hierarchies contain an extra label to identify their role in the class hierarchy (e.g. *Fact:QualitativeFact* qualifies a `QualitativeFact` inside a `Fact` hierarchy).

As it can be observed in the schema, the *name* property is used for identifying, at the knowledge level, a *named* `FactType`. The *value* property is used to record, at the operational level, the value assumed by a `TextualFact` or a `QuantitativeFact` node: a string of text in the first case, and a double precision number in the latter case. In this model, there is basically no difference between *named* and *anonymous* `FactTypes`: both are modeled using a node, and the only distinction between them is the presence of the *name* property.

Another characteristic of the graph model in Fig. 6.3 is the capability of modeling `TypeLink` and `FactLink` classes using relationships. These two classes were introduced in the original model to represent the parent-child relationship between `FactType` or `Fact` classes. For this reason, they can be naturally represented as a relationship in a graph-oriented model. In addition, since Neo4j represents relationships as directed arcs that can be traversed in both directions, this allows to simplify the model introducing a single relationship, called *Has Children*, for modeling `TypeLink` and `FactLink` classes, without any impact on query capabilities. Note that Neo4j allows to put a relationship only between two nodes, and this precludes the possibility to use a relationship to represent the reference between `TypeLink` and `FactLink`, as in the original model. Properties have been used to solve this problem, as follows: *i)* the *uuid* property of each `TypeLink` is used for storing an identifier value; *ii)* the same value is copied into the *type* property of the related `FactLink`. Properties have been used to solve this problem without transforming these two classes from relationships to nodes. Finally, the *Has Type* relationship is used to link together `Fact` and `FactType` nodes.

The self-explanatory *Has Unit* and *Has Phenomenon* relationships are ambivalent across the knowledge and the operational level, and are used to connect a `QuantitativeType` or `QualitativeType` node with a set of possible `Unit` or `Phenomenon` nodes, and the corresponding `QuantitativeFact`

or `QualitativeFact` node with the selected `Unit` or `Phenomenon` node.

6.3.2 A model for MongoDB

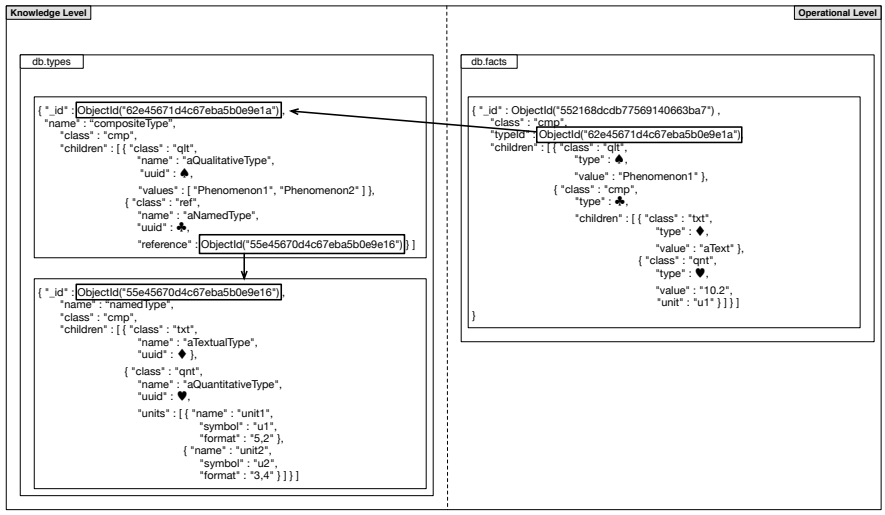


Figure 6.4: The representation of an instance of the domain model described in Sect. 6.2 on the document model of MongoDB database. The two sides of the figure show the collections used to persist `FactType` and `Facts` instances, named *db.types* and *db.facts*, respectively. At knowledge level, two *named* types have been persisted, with names *compositeType* and *namedType*. The first type includes the second one, as noted by the use of the *ObjectId* reference, and both of them include *anonymous* types as sub-documents. For reasons of readability, *uuid* property values have been replaced with symbols.

MongoDB [1] data model is based upon a *document-oriented* structure. A document is a collection of attribute-value pairs, with values that can be basic types, array of values or nested sub-documents. Documents with similar characteristics are grouped together and stored in *collections*. Relation between documents can be represented using *references*, that produce a normalized data model, or by *embedding* related data in documents, producing denormalized models. In particular, the use of denormalization techniques [49] is promoted by document-oriented NoSQL solutions for discouraging the usage of JOIN queries, and solving typical performance issues

that affect relational databases, preserving data consistency and completeness [84].

The schema of Fig. 6.4 illustrates the document-oriented model used in our concrete case, representing data in accordance with the domain model of Fig. 6.1. Usually, modeling an object-oriented domain logic using a document-based data model can be achieved in a direct way, but, in the case of study of a reflection architecture, this simplicity is weakened from the indirect structure of the model. The proposed solution attains a good balance, *mixing* together documents embedding approaches with references techniques [50] for obtaining a flexible data representation without performance degradation. In particular, the `FactType` hierarchy comprises a neat example of *mixed modelation*. In fact, while *named* `FactType` instances are persisted as documents, and are referenced by other documents using their *ObjectId*, *anonymous* `FactType` instances are persisted as embedded documents inside the named `FactType` document in which they are defined.

To efficiently recognize the subtyping-class of an instance in the `FactType` or `Fact` hierarchy, every persisted document has a property called *class* that can assume the following values: *i) txt*, for referring to a `TextualType` or `TextualFact` instance; *ii) qlt*, for referring to a `QualitativeType` or `QualitativeFact` instance; *iii) qnt*, for referring to a `QuantitativeType` or `QuantitativeFact` instance; and, *iv) cmp*, for referring to a `CompositeType` or `CompositeFact` instance. In so doing, it is sufficient to check the *class* property value of a document to recognize its nature, avoiding to pre-emptively explore its properties. In the case of *named* `FactTypes`, the *class* property is valued with the string value *ref*, and an additional property called *reference* contains the *ObjectId* of the *named* `FactType`.

This different behaviour in `FactType` persistence drops the need to persist the `TypeLink` class as a separate entity. For this reason, `TypeLink` and `FactType` classes are modeled in MongoDB as a single entity, and the *name* property of embedded documents inside `CompositeType` instances corresponds to the `TypeLink` *name* property of the original model. Note that since the embedded documents are always *anonymous*, the `FactType` *name* property is specified only for the root document of a `FactType` instance.

The `Fact` hierarchy does not have the same need for reusability and referencing that characterize `FactTypes`. For this reason, `Fact` instances can always be represented as a single document, in which `Fact` children are

embedded as sub-documents. In so doing, the number of queries for data retrieval is considerably limited.

`Fact` and related `FactType` instances are linked together with different strategies, based on the nature of the `Fact`. In the case of a `Fact` root document, the *typeId* property is used to store the *ObjectId* of the referenced `FactType` instance. Otherwise, when dealing with sub-documents of the `Fact` root, the *type* property is used to refer to the *uuid* value of the corresponding `FactType`. Consequently, for completely retrieving a `Fact` and its `FactType`, it is necessary to: *i)* query for the `Fact`; then, *ii)* query for the corresponding `FactType` using the *ObjectId* referenced by the `Fact` root; *iii)* link together the retrieved `Fact` and `FactType` instances using the *type* property.

For the sake of completeness, `Phenomenon` entities are modeled as embedded documents inside `QualitativeFact` and `QualitativeType` documents with the intent of minimizing the number of retrieval query in reading operations. In the same manner, `Unit` entities are modeled inside `QuantitativeFact` and `QuantitativeType` documents.

6.4 Information equivalence across data models

A comparison of the performance among different data storage implementations (i.e., from relational to a graph- or document-oriented model) requires that they are in some sense equivalent. Since data can be modeled in various ways through the use of different data structures offering the same *information capacity*, a notion of model equivalence, or hierarchy of equivalences [43], is required to be defined. In a general sense, two data structures can be considered equivalent in terms of information-capacities if they can be associated to the same number of states, such that each state of a data structure can be mapped to a *database state* of the other structure, preserving any relationship attribute value.

For the purpose of our experimentation, it is not necessary to prove the *complete* equivalence between two representations, but it is sufficient to prove the *query* equivalence of two models [9], i.e., the possibility to extract the same information from both models through query operations. Specifically, the equivalence problem consists in casting information data into structures (i.e., graphs or tree) of the same type. Comparing and matching graphs (i.e.,

graph homomorphism) is a well-known NP-complete problem [39], and different approaches have been proposed to determine the distance between two graphs using specific heuristic [19, 21]. In our case, proving the equivalence of *Neo4j* and *MongoDB* data models with respect to the actual relational model means showing that they have the same representativeness of information. This means that the equivalence problem will be focused on showing that two data structures are exactly identical in the information they carry, rather than identifying similarities and differences between data models. Furthermore, it is not necessary to verify the *query dominance* for the new data model, but simply proving that it is possible to query the same `Examination` and `ExaminationType` structure across different representations.

In a practical manner, we consider equivalent two data representations of the same domain logic using different persistence models when the carried information can be serialized into an equivalent string of information. In so doing, given two different persistence models, named A and B, A and B are equivalent if it is possible to generate the same string serialization for each given `Examination` and `ExaminationType` instance represented in A and B. Consequently, if A is a valid model, and A and B are equivalent, than B is also valid. Note that we assume that the actual relational model is a valid reference model, from which we want to prove the validity of the converted NoSQL models.

We have started by choosing a dataset with an arbitrary number of clinical information data persisted in the relational model. Then, we have retrieved all the `Examinations` and `ExaminationTypes` instances contained in the dataset, and we have serialized the information data in a string representation. Finally, for testing the equivalence, we have converted information data from the relational model to the target NoSQL model, serializing again the information data, and comparing the resulting string with the string obtained from the relational model at the previous step. The validation process is considered successful, if we are able to obtain an equivalence between the reference relational model and the target NoSQL model for every string of information.

Fig. 6.5 illustrates an example of the string produced during the serialization process applied to the information data as so represented using the models depicted in Figs. 6.3 and 6.4. The structure of the serialization is deliberately similar to a JSON document, due to its simple and readable

syntax. This string serialization can be also used to verify which are the essential properties that a model must implement to be valid.

Note that the completeness of the new representation is also granted by the structures of target models. In fact, the conversion from *MySQL* to *Neo4j* model is the most natural way since it allows to maintain nodes and relationships according to the structure of the original tree or graph. Moreover, the MongoDB document representation is modeled in a way that can be considered an inverse operation of vertical decomposition during normalization process, as discussed in [9] and [12].

```
    "compositeType" : {
      "aNamedType" : {
        "aQuantitativeType" : "10.2 u1"
        "aTextualType" : "aText"
      }
      "aQualitativeType" : "Phenomenon2"
    }

```

Figure 6.5: An example of serialization of a clinical `Examination`. The pattern used to serialize the information is as follows: *type.name : fact.value*. `CompositeFact` values are described by the list of values assumed by children `Facts` defined between braces.

6.5 Experimental Evaluation

An experimentation was carried out to evaluate the performance of the three different implementations based on MySQL+Hibernate, Neo4j, and MongoDB, and their sensitivity to the characteristics of the dataset.

The evaluation was focused on the *Access EHR content* use case (see Fig. 3.5), in which a health professional actor access past medical examinations related to a specific patient in order to review collected clinical information. This use case has turned out to have the most relevant impact on the perceived performance in the context-of-use and, at the same time, constitutes a major scenario of interaction in EHR systems.

6.5.1 Methodology

We can expect that the response time of different storage schemes be dependent on the complexity of the collection of domain logic objects that are read-from or written-to the persistence layer. Due to the pattern-based architecture of classes in the domain logic, objects are organized in an almost tree-like structure, and their complexity can thus be characterized in terms of number of nodes and depth of the tree in the examination structure.

For this reason, we experimented with two different kind of datasets: *i)* a *real* dataset of clinical examinations acquired in the *Empedocle* EHR system for which we provide a description of the statistics about the number of nodes and the depth of the tree structure; *ii)* a *synthetic* dataset for which we can control the statistics so as to stress the indexes of complexity.

The *real* dataset consists of about 13 000 examinations [1] that belong to the same medical speciality and thus share the same structure. Table 6.1 summarizes the complexity of the examination structure, i.e., the number of `FactTypes` included in each examination, which is the number of meta-objects in the knowledge level. The structure of the examination includes 243+110+99 fields, which are organized in a graph whose depth (intended as the maximum distance from the root node) is equal to 8, and which includes 144 `FactTypes` that act as composition nodes.

Note that, at the operational level, the complexity of the tree structure depends on the course of each specific examination, and its statistic is re-

[1] The real dataset was conveniently anonymized by omitting patients' personal information, and by obfuscating textual observations recorded during each clinical session.

Depth	8	
Number of nodes	596	
	CompositeType	144
	QualitativeType	243
	QuantitativeType	110
	TextuaType	99

Table 6.1: Characteristics of the considered examination structure in the dataset, with additional details about the distribution of type nodes contained in the structure. Of the 596 nodes that form the examination type, 452 nodes are leaf nodes, which actually contain a value.

sumed in Figs. 6.6 and 6.7. Fig. 6.6 reports the distribution of examinations per number of nodes. Fig. 6.7 characterizes the distribution of examinations per depth of the tree structure. From these statistics, it is possible to note that the size of the tree-like structure (composed by `Facts`) is always much lower than the size of the corresponding graph structure (composed by `FactTypes`), which depends on the fact that, during a standard clinical session, not all the observations allowed by the examination structure (≈ 600) are actually recorded.

The *synthetic* dataset contains generated examinations with a full binary tree structure, with depth ranging from 2 to 8. For each depth, a fixed number of 100 examinations has been generated. Being a full binary tree, the number of nodes n in each of the trees of depth d is given by:

$$n = 2^{d+1} - 1,$$

ranging from 3 to 511 nodes. The synthetic dataset does not correspond to a real situation in the present context-of-use of our EHR system, but it can become a possible scenario in the evolution of the use of the *Empedocle* EHR system, and, for this reason, represents a relevant part of the motivation for this performance engineering investigation. In the more general perspective of a reflection architecture, this corresponds to the case where different courses can be described on a structure with different degrees of completeness.

The evaluation has been carried out with reference to a major scenario of interaction: a health professional accesses a patient's EHR content in order

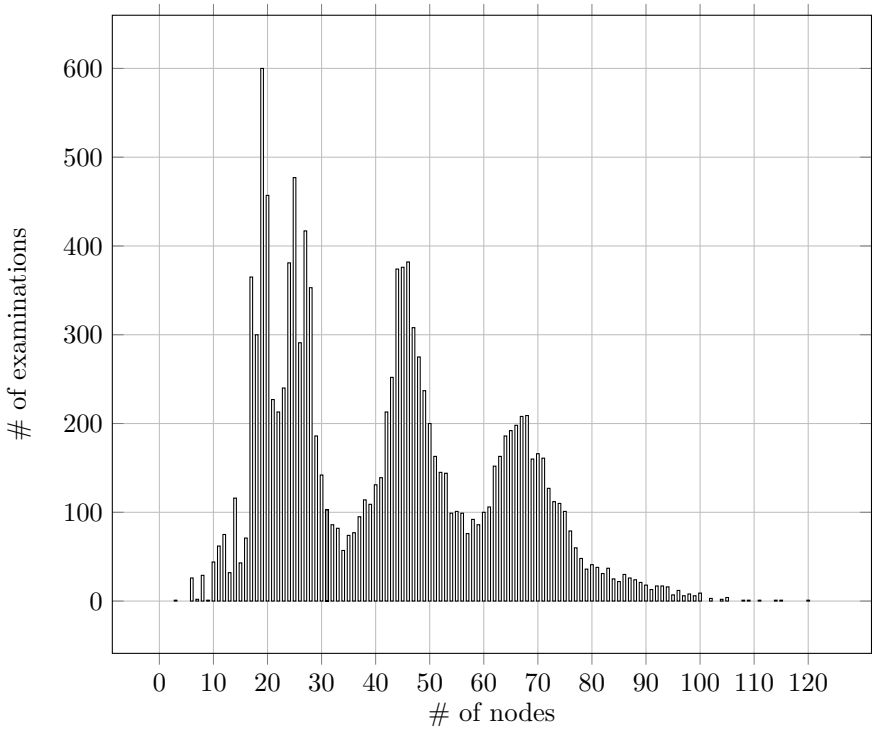


Figure 6.6: The histogram describes the distribution of examinations as the number of nodes varies. Note that about 35% of the examinations in the dataset are in the neighbourhood of 23±6, with peaks in 19, 20, 25 and 27. This shows clearly how, usually, only a small part of the examination structure, comprising 596 nodes, is actually filled out by health professionals. Only about 9% of the examinations in the dataset have more than 70 nodes filled out.

to review past medical examinations and read collected clinical information. To do that, each examination in the dataset has first been retrieved and, then, a read-only operation has been performed in order to simulate the real interaction of users with the EHR system through the interfaces exposed by the *Presentation Layer*.

As a metric of performance, we evaluated the total time required to complete the selected scenario, from data retrieving to data serialization,

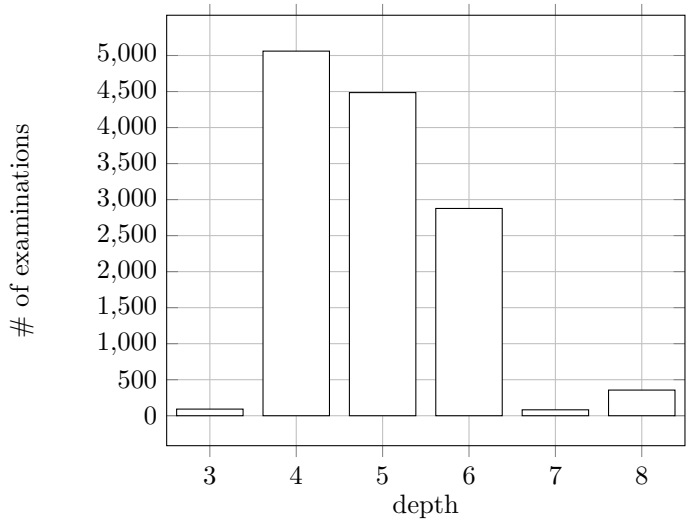


Figure 6.7: The histogram describes the distribution of examinations as the depth increases. Note that 96% of the examinations in the dataset have depth comprises between 4 and 6.

for all compared models. Note that the examination retrieval also implies the retrieval of the associated type structure. In the process of measuring the total time, we do not distinguish time passed by the various phases of data retrieval and process: specifically, we measure the time to complete the whole use case, that comprises database retrieval operations, interactions with Java database APIs or with the ORM persistence layer present only in the relational case.

Experimental results not reported here indicate that the ORM layer, implemented by *Hibernate* in the current application stack, does not significantly impact the overall performance, since it is optimized for the underlying database technology [56, 80]. No comparison has been carried out regarding storage space requirements for the considered technologies, not representing a critical aspect for the EHR system in the case under consideration.

The experiments were conducted on a computer with the following characteristics: Debian 3.2.60 operating system, with 2 x Intel Xeon E5640 @ 2.66 GHz 64-Bit CPU, and 32 GB of RAM Memory.

	μ (ms)	CV	min (ms)	max (ms)
MySQL + Hibernate	76.06	0.031	70.79	81.94
Neo4j	51.29	0.0024	50.94	51.57
MongoDB	2.27	0.064	1.82	2.57

Table 6.2: Comparison between MySQL+Hibernate, Neo4j, and MongoDB, evaluated using the *real* dataset comprising 12953 examinations. Table reports the mean value (μ) and the coefficient of variation (CV) for the execution of a single examination, as well as the minimum (min) and maximum (max) execution time registered during the 100 iterations for a specific technology.

6.5.2 Results

Table 6.2 reports the results of the experimentation on the real dataset, showing the mean value (μ), measured in ms, and the coefficient of variation (CV) of the time spent to complete the read-only operation for a single examination in the three implementations under test. These statistical indexes were evaluated by repeating the task for 100 times on all 12 953 examinations in the dataset.

In comparison with the MySQL + Hibernate implementation, Neo4j reduces the retrieval time by approximately 1.5 times, and MongoDB reduces it by more than 33 times. Performance with relational database such as MySQL are deeply linked to the number of JOIN in the executed queries. For this reason, in the considered model, the retrieval of specific classes of `Facts` has a different impact on the complexity of the query. In particular, since `CompositeFacts` represent hierarchical structures in the `Facts` tree, querying operations result in a higher number of JOINs, which produces a significant impact on performance, documented by [69] as "*N+1 queries*" data access anti-pattern or "*Circuitous Treasure Hunt*" problem. In a similar way, `QuantitativeFacts` and `QualitativeFacts` also produce more complex queries, since an additional JOIN operation is required to retrieve related `Phenomena` and `Units`.

Table 6.3 shows the results of experimentation on the *synthetic* dataset. We report the mean value (μ), measured in ms, and the coefficient of variation (CV) of the time spent to complete the read-only operation for a single

examination in the three implementations under test, evaluated by repeating the task for 100 times on all 100 examinations in the dataset. Results indicate that MongoDB attains by far a better performance and slower sensitivity to the examination depth. It should also be noted that the MySQL + Hibernate implementation performs better than Neo4j for examination with depth lower than 7.

Depth	**MySQL + Hibernate**		**Neo4j**		**MongoDB**	
	μ (ms)	CV	μ (ms)	CV	μ (ms)	CV
2	6.93	0.12	18.76	0.12	1.07	0.05
3	9.54	0.11	19.41	0.09	1.2	0.06
4	12.6	0.1	21.57	0.09	1.38	0.07
5	18.87	0.09	26.04	0.08	1.64	0.07
6	28.17	0.09	33.94	0.05	2.2	0.07
7	48.18	0.08	44.03	0.05	3.05	0.08
8	121.29	0.04	72.93	0.05	4.88	0.07

Table 6.3: Comparison between MySQL+Hibernate, Neo4j, and MongoDB, evaluated using the *synthetic* dataset comprising 100 examinations with increasing depth. Table reports the mean value (μ) and the coefficient of variation (CV) for the execution of a single examination. Results in the table show that the MySQL + Hibernate implementation performs better than Neo4j for examination with depth lower than 7, while MongoDB attains by far a better performance and slower sensitivity to the examination depth.

6.5.3 Reproducibility of results

Both MongoDB and Neo4j synthetic databases containing the data used for test the implementation of the considered domain model are shared with community through *Zenodo*, a research data repository created by OpenAIRE and CERN.
The repository contains also a Java benchmark scripts used to test the performances of these implementations [54].

Results of this research test also a MySQL+JPA implementation that is used as a reference in the comparison of the experimentation results with

other technologies. Database and code for this implementation is not public, since it would mean exposing some legacy Java library actually used at the major hospital of Florence (AOUC) and Institute for Oncological Study and Prevention (ISPO).
For the same reason, real dataset of clinical examinations is protected by severe privacy laws. However, it's possible to reproduce results of performance comparison generating synthetic observations following characteristics reported in Table 6.1 or in Table 6.3.

6.6 Discussion

In this dissertation I described a consolidated software architecture, pattern-based, which persistence data layer was originally based on the MySQL relational database and JPA. Using a model-driven approach I described new data persistence models based on promising NoSQL technologies, such as *Neo4j* [2] and *MongoDB* [1]. These models have been engineered to balance in the best way elements such as: ease of conversion, embedding and references, granting data integrity and equivalence in the information representation with the relational model.

I presented experimental results on performance gain achieved through the use of such databases. The comparison is based on the study of the real world scenario of our EHR system, called *Empedocle*, based on the *Observations & Measurements* [34] and *Composite* [36] patterns, where the main requirement is the structure flexibility. Since the considered NoSQL databases do not have a fixed schema, non-functional requirements of changeability and adaptability can be easily achieved. In addition, they constitute a good solution for big clusters of data which structure is subject to change over time.

Performance results obtained during experimentations in *real* and *synthetic* datasets indicate a clear gain in performance through the use of MongoDB database, and more generally, a better scalability of NoSQL solutions when the depth of the examination structures grows, due to the increased number of JOINs and reference operations affecting the MySQL solution. Moreover, both tested NoSQL technologies offer advantages in terms of flexibility in the data model, scalability and reliability.

Results also indicate a counter-intuitive conclusion: the graph-oriented

data model of Neo4j allows a more natural and direct data conversion, which also permits a simpler implementation; however, the document-oriented data model of MongoDB produces by far better performance results. Specifically Neo4j, which modeling is more natural in our software architecture context, presents a performance increase of 1.5 times compared to MySQL + Hibernate. On the other hand, MongoDB, which required a bigger engineering investment to convert our data model balancing between redundancy, adaptability and performance, presents a gain of almost 33 times compared to MySQL + Hibernate.

The present investigation is completely open to explore the performances of NoSQL databases in other use cases, not only limited to read-only operations, but also extended to write and update scenarios, whose impact on the application is less relevant but nonetheless interesting to have a full comparison between the various models [59].

Chapter 7

Conclusion

This chapter summarizes the contribution of the thesis and discusses avenues for future research.

7.1 Summary of contribution

In this research project we described a consolidated software meta-modeling architecture, pattern-designed, based on j2EE technological stack. The illustrated Reflection architectural pattern provides significant benefits in terms of adaptability, maintainability, self-awareness, and direct involvement of domain experts in the configuration stage.
Healthcare comprises a notable domain of interest, where the availability of a large amount of information can be exploited to take relevant and tangible benefits in terms of efficiency of the care process, improved outcomes and reduced health system costs.
However, design by patterns does not account for performance as first-class requirement, and naturally incurs in well-known performance anti-patterns, which may become crucial when volume and variety must meet also velocity. The complexity is further exacerbated when the object oriented domain model is mapped to a relational database.

With this experimentation we addressed performance engineering of a meta-modeling architecture and we presented experimental results indicating the gain obtained by applying refactoring in the concrete case of a real application of data management in Healthcare context [33].

This research project reported also comparative experimental performance results attained by combining the pattern-based domain logic with a persistence layer based on NoSQL paradigm, using a model-driven approach to balance in the best way elements such as: ease of conversion, embedding and references, granting data integrity and equivalence in the information representation with the relational model [32].

7.2 Directions for future work

The research activity still ongoing is aimed at extend the work about performance:

- widening investigation in growing complexity model (especially for the experimentation about anti-pattern, through the utilization of a synthetic dataset where statistics can be controlled);
- analyzing more *user-goal* level use cases and operations (e.g. write-dominant scenario for NoSql experimentation);
- testing different databases technologies;
- discussing an hybrid solution between MySql and NoSql databases (Polyglot Persistence).

We want also to consider other software quality parameters in these different mapping and technological choices. Specifically, we want to take into account the ability to provide efficient *indexes* for researching data stored into the database, and the evaluation of *disk space* employed by data representation in the technology.

At the same time, the *performance testing* topic has been refined in the context of J2EE architectures. It is hard to adhere to a single standard as software performance testing has to be tailored to their contextual environments. In particular, in J2EE architectures, a performance bottleneck can be difficult to be identified and can be hard to figure out the layer that is responsible of this problem. Referring to Fig. 4.2, we focused on performance analysis of the Data Layer and related component, but we would like to offer a formalized way to identify the J2EE layer that represents a performance slowdown and to provide an overview of possible methods to change or improve the portion of code that lies in this layer. Additionally, as a future

work, we would like to investigate empirically the diffusion of the proposes anti-patterns in systems based on a meta-modeling architecture.

At last, the issue of *functional testing* of a model that follows the schema of the *Reflection* pattern is especially interesting. In fact, the loosening of *type-checking* requires, under standard and formal methods, the realization of a test-suite focused on verify the model resilience, making its design a delicate task.

Appendix A

Publications

This research activity has led to some publications in international conferences.

International Conferences and Workshops

1. **S. Fioravanti, S. Mattolini, F. Patara, E. Vicario**, "Experimental performance evaluation of different data models for a reflection software architecture over NoSQL persistence layers", in *7th ACM/SPEC International Conference on Performance Engineering* (ICPE 2016), March 12-16 2016.
2. **S. Fioravanti, F. Patara, E. Vicario**, "Engineering the Performance of a Meta-modeling Architecture", in *8th ACM/SPEC International Conference on Performance Engineering* (ICPE 2017), April 22-26 2017.

Bibliography

[1] "MongoDB for GIANT idea," https://www.mongodb.org/.

[2] "Neo4j the world's leading graph database," http://neo4j.com/.

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

[4] R. L. Ackoff, "From data to wisdom," *Journal of applied systems analysis*, vol. 16, no. 1, pp. 3–9, 1989.

[5] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya, "Software architecture optimization methods: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 658–683, 2013.

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

[7] 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.

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

[9] P. Atzeni, G. Ausiello, C. Batini, and M. Moscarini, "Inclusion and equivalence between relational database schemata," *Theoretical Computer Science*, vol. 19, no. 3, pp. 267–285, 1982.

[10] T. Beale, "Archetypes: Constraint-based domain models for future-proof information systems," in *OOPSLA 2002 workshop on behavioural semantics*, vol. 105, 2002.

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

[12] C. Beeri, P. A. Bernstein, and N. Goodman, "A sophisticate's introduction to database normalization theory," in *Proceedings of the fourth international conference on Very Large Data Bases-Volume 4*. VLDB Endowment, 1978, pp. 113–124.

[13] G. Bellinger, D. Castro, and A. Mills, "Data, information, knowledge, and wisdom," 2004.

[14] E. S. Berner, *Clinical Decision Support Systems*, ser. Theory and Practice. Springer, 1999.

[15] S. N. Bhatti, Z. H. Abro, and F. R. Abro, "Performance evaluation of java based object relational mapping tool," *Mehran University Research Journal of Engineering and Technology*, vol. 32, no. 2, pp. 159–166, 2013.

[16] F. Bugiotti, L. Cabibbo, P. Atzeni, and R. Torlone, "How I learned to stop worrying and love NoSQL databases," in *SEBD Italian Symposium on Advanced Database Systems*, 2015.

[17] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, 1996.

[18] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "Performance and scalability of ejb applications," in *ACM Sigplan Notices*, vol. 37, no. 11. ACM, 2002, pp. 246–261.

[19] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," *SIGMOD Rec.*, vol. 25, no. 2, pp. 493–504, Jun. 1996. [Online]. Available: http://doi.acm.org/10.1145/235968.233366

[20] L. Chung and N. Subramanian, "Adaptable system/software architectures," *Journal of Systems Architecture: the EUROMICRO Journal*, vol. 50, no. 7, pp. 365–366, 2004.

[21] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "Performance evaluation of the VF graph matching algorithm," in *Image Analysis and Processing, 1999. Proceedings. International Conference on*. IEEE, 1999, pp. 1172–1177.

[22] V. Cortellessa, A. Di Marco, and P. Inverardi, *Model-based software performance analysis*. Springer Science & Business Media, 2011.

[23] W. Crawford and J. Kaplan, *J2EE Design Patterns: Patterns in the Real World*. " O'Reilly Media, Inc.", 2003.

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

[25] P. Dhawan, "Performance comparison: Security design choices," *Microsoft Developer Network*, 2002.

[26] R. H. Dolin, L. Alschuler, S. Boyer, C. Beebe, F. M. Behlen, P. V. Biron, and A. Shabo, "Hl7 clinical document architecture, release 2," *Journal of the American Medical Informatics Association*, vol. 13, no. 1, pp. 30–39, 2006.

[27] L. Douglas, "3d data management: Controlling data volume, velocity and variety," *Gartner. Retrieved*, vol. 6, 2001.

[28] B. Dudney, S. Asbury, J. K. Krozak, and K. Wittkopf, *J2EE antipatterns*. John Wiley & Sons, 2003.

[29] H. S. Ferreira, A. Aguiar, and J. P. Faria, "Adaptive object-modelling: Patterns, tools and applications," in *Software Engineering Advances, 2009. ICSEA'09. Fourth International Conference on*. IEEE, 2009, pp. 530–535.

[30] H. S. Ferreira, F. F. Correia, and A. Aguiar, "Design for an adaptive object-model framework," in *Proceedings of the 4th Workshop on Models@ run. time, held at the ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, 2009.

[31] H. S. Ferreira, F. F. Correia, A. Aguiar, and J. P. Faria, "Adaptive object-models: A research roadmap," *International Journal On Advances in Software*, vol. 3, 2010.

[32] 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.

[33] S. Fioravanti, F. Patara, and E. Vicario, "Engineering the performance of a meta-modeling architecture," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ACM, 2017, pp. 203–208.

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

[35] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[36] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[37] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[38] A. Gandini, M. Gribaudo, W. J. Knottenbelt, R. Osman, and P. Piazzolla, "Performance evaluation of NoSQL databases," in *Computer Performance Engineering*. Springer, 2014, pp. 16–29.

[39] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York, NY, USA: W. H. Freeman & Co., 1979.

[40] R. Garud, S. Jain, and P. Tuertscher, "Incomplete by design and designing for incompleteness," in *Design Requirements Engineering: A Ten-Year Perspective.* Springer, 2009, pp. 137–156.

[41] B. Grady, "Object-oriented analysis and design with applications," 1994.

[42] F. Holzschuher and R. Peinl, "Performance of graph query languages: comparison of Cypher, Gremlin and Native Access in Neo4J," in *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, ser. EDBT '13. New York, NY, USA: ACM, 2013, pp. 195–204. [Online]. Available: http://doi.acm.org/10.1145/2457317.2457351

[43] R. Hull, "Relative information capacity of simple relational database schemata," *SIAM Journal on Computing*, vol. 15, no. 3, pp. 856–886, 1986.

[44] C. Ireland, D. Bowers, M. Newton, and K. Waugh, "A classification of object-relational impedance mismatch," in *Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA'09. First International Conference on.* IEEE, 2009, pp. 36–43.

[45] ISO/IEC, *ISO/IEC 9126. Software engineering – Product quality.* ISO/IEC, 2001.

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

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

[48] 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, pp. 72–83.

[49] A. Kanade, A. Gopal, and S. Kanade, "A study of normalization and embedding in MongoDB," in *Advance Computing Conference (IACC), 2014 IEEE International.* IEEE, 2014, pp. 416–421.

[50] I. Katsov, "NoSQL data modeling techniques," *Highly Scalable Blog*, 2012.

[51] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Performance Evaluation*, vol. 67, no. 8, pp. 634–658, 2010.

[52] J. R. Lourenço, B. Cabral, P. Carreiro, M. Vieira, and J. Bernardino, "Choosing the right NoSQL database for the job: a quality attribute evaluation," *Journal of Big Data*, vol. 2, no. 1, pp. 1–26, 2015.

[53] R. C. Martin, "The dependency inversion principle," *C++ Report*, vol. 8, no. 6, pp. 61–66, 1996.

[54] S. Mattolini, S. Fioravanti, F. Patara, and E. Vicario, "Experimental performance evaluation of different data models for a reflection software architecture over NoSQL persistence layers (Data)," Feb. 2016. [Online]. Available: https://doi.org/10.5281/zenodo.46065

[55] I. Molyneaux, *The art of application performance testing: Help for programmers and quality assurance.* " O'Reilly Media, Inc.", 2009.

[56] E. J. O'Neil, "Object/relational mapping 2008: Hibernate and the Entity Data Model (EDM)," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data.* ACM, 2008, pp. 1351–1356.

[57] K. Orend, "Analysis and classification of nosql databases and evaluation of their ability to replace an object-relational persistence layer," *Architecture*, vol. 1, 2010.

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

[59] Z. Parker, S. Poe, and S. V. Vrbsky, "Comparing NoSQL MongoDB to an SQL db," in *Proceedings of the 51st ACM Southeast Conference.* ACM, 2013, p. 5.

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

[61] F. Patara, "Multi-level meta-modeling architectures applied to ehealth," Ph.D. dissertation, University of Florence, 2016.

[62] F. Patara and E. Vicario, "An adaptable patient-centric electronic health record system for personalized home care," in *2014 8th International Symposium on Medical Information and Communication Technology (ISMICT).* IEEE, 2014, pp. 1–5.

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

[64] A. Schram and K. M. Anderson, "MySQL to NoSQL: data modeling challenges in supporting scalability," in *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity.* ACM, 2012, pp. 191–202.

[65] B. Schwartz, P. Zaitsev, and V. Tkachenko, *High performance MySQL: optimization, backups, and replication.* " O'Reilly Media, Inc.", 2012.

[66] M. Sharma, V. S. Iyer, S. Subramanian, and A. Shetty, "A comparative study on load testing tools," 2016.

[67] 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.

[68] C. Smith and L. G. Williams, "New software performance antipatterns: More ways to shoot yourself in the foot," in *Int. CMG Conference*, 2002, pp. 667–674.

[69] C. U. Smith and L. G. Williams, "Software performance antipatterns," in *Proceedings of the 2Nd International Workshop on Software and Performance*, ser. WOSP '00. New York, NY, USA: ACM, 2000, pp. 127–136. [Online]. Available: http://doi.acm.org/10.1145/350391.350420

[70] ——, *Performance solutions: a practical guide to creating responsive, scalable software*. Addison-Wesley Professional, 2001.

[71] ——, "More new software performance antipatterns: Even more ways to shoot yourself in the foot," in *Computer Measurement Group Conference*, 2003, pp. 717–725.

[72] X. Song, B. Hwong, G. Matos, A. Rudorfer, C. Nelson, M. Han, and A. Girenkov, "Understanding requirements for computer-aided healthcare workflows: experiences and challenges," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 930–934.

[73] M. Stonebraker, "SQL databases v. NoSQL databases," *Communications of the ACM*, vol. 53, no. 4, pp. 10–11, 2010.

[74] C. Trubiani, A. Bran, A. van Hoorn, A. Avritzer, and H. Knoche, "Exploiting load testing and profiling for performance antipattern detection," *Information and Software Technology*, 2017.

[75] 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, pp. 19–30.

[76] B. G. Tudorica and C. Bucur, "A comparison between several NoSQL databases with comments and notes," in *Roedunet International Conference (RoEduNet), 2011 10th*. IEEE, 2011, pp. 1–5.

[77] P. Van Zyl, D. G. Kourie, and A. Boake, "Comparing the performance of object databases and orm tools," in *Proceedings of the 2006 annual research conference 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.

[78] 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 *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists.* ACM, 2009, pp. 150–159.

[79] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A comparison of a graph database and a relational database: a data provenance perspective," in *Proceedings of the 48th annual Southeast regional conference.* ACM, 2010, p. 42.

[80] Q. Wu, Y. Hu, and Y. Wang, "Research on data persistence layer based on hibernate framework," in *Intelligent Systems and Applications (ISA), 2010 2nd International Workshop on.* IEEE, 2010, pp. 1–4.

[81] W. Xu, Z. Zhou, H. Zhou, W. Zhang, and J. Xie, "MongoDB improves big data analysis performance on Electric Health Record system," in *Life System Modeling and Simulation.* Springer, 2014, pp. 350–357.

[82] J. W. Yoder, F. Balaguer, and R. Johnson, "Architecture and design of adaptive object-models," *ACM Sigplan Notices*, vol. 36, no. 12, pp. 50–60, 2001.

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

[84] G. Zhao, Q. Lin, L. Li, and Z. Li, "Schema conversion model of SQL database to NoSQL," in *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2014 Ninth International Conference on.* IEEE, 2014, pp. 355–362.

[85] Z. Zhou and Z. Chen, "Performance evaluation of transparent persistence layer in java applications," in *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2010 International Conference on.* IEEE, 2010, pp. 21–26.

[86] C. Zins, "Conceptual approaches for defining data, information, and knowledge," *Journal of the Association for Information Science and Technology*, vol. 58, no. 4, pp. 479–493, 2007.