UNIVERSITY OF UTRECHT

DEPARTMENT OF INFORMATION AND COMPUTING SCIENCES

A Decision Support Model for using an Object-Relational Mapping Tool in the Data Management Component of a Software Platform

Rares George Sfirlogea

Supervisors:

dr. R.L. Jansen
dr. ir. J.M.E.M. van der Werf

Friday 6th February, 2015
Academic year 2014/2015

Abstract

The usage of an ecosystem-based application framework gives software companies a competitive advantage in delivering stable, feature rich products while keeping the completion time to a minimum. It is seldom the case that a platform is selected by looking at its software architecture although it can reveal a lot of details about its limitations and functionality. The Object-Relational Mapping (ORM) tool in the data management component imposes extendability restrictions on the software platform. The software architect or developer that is responsible of making this decision is often unaware of the platform traits leading to breaking the general conventions or even considering a costly rewrite of the entire application in the future. The aim of this research thesis is to create a decision support model regarding the inclusion of an ORM tool in the platform architecture and the consequences it imposes on the software platform's quality attributes. With this artefact, any individual in charge with the product architecture can make a more knowledgeable decision, by aligning the platform capabilities with his data requirements.

Acknowledgements

I would like to express my sincere appreciation for all the people who helped this research reach its final state. With a special mention going to my thesis coordinators, the experts who agreed to be interviewed and of course my girlfriend, family and friends who put up with me during this long period of time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The pace at which new products and services are released in the software industry has seen an impressive ascension in the last decade (Fichman, 2004). This has been possible due to the availability of advanced software platforms, also known as application frameworks, that provide a basic starting point for an application on top of which domain specific features are to be built (Evans, Hagiu, & Schmalensee, 2006). An application framework can be defined as a instantiation of software reuse techniques in the form of a software package. The package usually contains an abstract design for a type of software application, providing a set of core components which can be easily extended (?, ?).

Considering the level of abstraction of the application base that they offer, some platforms can provide only a foundation for a certain technology, for instance a web application framework, while others may be more functional oriented. Examples of platforms come from both the open source domain (such as Ruby on Rails, Joomla, Drupal, Node.js, Android) or the proprietary software domain (such as iOS, Microsoft CRM, SAP NetWeaver, Oracle Application Development Framework).

Also outlined by Fayad and Schmidt (1997), the main benefits from the usage of an application framework are a result of the following key features:

- **Modularity** - Having the framework carefully structured in easily distinguishable components reduces the effort of understanding how it works and what are its core capabilities and limitations.

- **Reusability** - Provides default built-in functionality in a stable form eliminating the need to implement and test basic, common features.
- **Extensibility** - The platform's components are built with extension in mind, allowing the creation of new functionality.

One of the decisive factors of choosing a specific platform is the amount of functionality that is included in the framework which covers the requirements of the project at hand. The advantages of software reuse are best applied when using a software platform that is part of a software ecosystem (Frakes & Kang, 2005; Griss, 1997). As defined by Jansen (2013), an ecosystem based software platform consists of a group of software artifacts that can be perceived as a coherent whole and with which third parties can create applications for their individual purpose. The architecture of application frameworks is adaptable, taking future changes into account, through the implementation of extension points. This allows platform extenders to add new functionality and avoids quick degeneration of the platform (Eick, Graves, Karr, Marron, & Mockus, 2001).

The software architecture of a platform can be defined as "a set of structures needed to reason about the system, which comprise software elements, relations among them and properties of both" (Bass, Clements, & Kazman, 2012). To put this in a simpler form it can be interpreted as the common understanding of several experts of how the major components work in a complex software system (Fowler, 2003). Although presented as a high level set of artefacts that describe the system and its behaviour(Shaw & Garlan, 1996), a close analysis can reveal qualities that influence how well the end product will perform. In the research of Kabbedijk, Salfischberger, and Jansen (2013) two architectural patterns have been identified that show how online applications currently adapt to dynamic functionality adjustments. In order to reveal the impact that each discovered pattern has on the overall platform the affected quality attributes are further analysed and discussed.

One critical part of a modern platform architecture is the way data is being accessed, managed and applied. This element of the application takes care of the retrieval and storage of information useful to its users. Failure to facilitate the perfect operation of the data management component can lead to a decrease in data quality and application performance. In turn this may cause a functionality fault and an overall bad user experience for the developed product. Although the term of *data management* can be used to describe a vast array of topics, in this research project the concept will refer to the platform's component or unit that handles the management of data from the low level queries to the actual usage in the application business logic implementation.

Information found in software systems can be stored using a vast array of methods and mechanisms depending on the data type and its purposed longevity. While information necessary only for the lifetime of a user request (also know as in-memory data) is automatically handled by the used programming language or software platform, persistent data may be kept in independent retention systems. Regardless of the location where the storage system exists in relation to the software platform, internally or externally, it is also reffered to as a data source. The software platform communicates with each data source through particular adapters forming a data gateway between the application and the storage system.

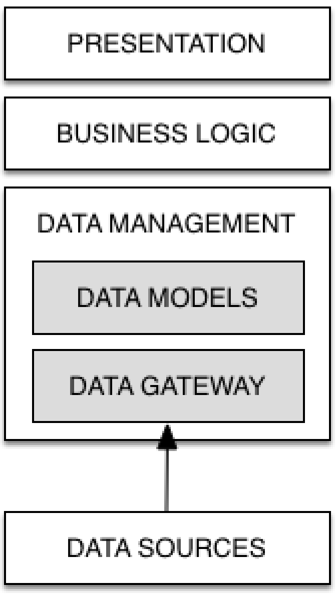


Figure 1.1: Bottom-up application architecture from data perspective

In addition to having a data source and a data gateway for that specific source type, complex applications make use of data modeling in order to provide a simple interface and adding more structure to information. Figure 1.1 shows an example of architecture layers from a data perspective. The data sources feed the application with information from storage solutions or other services. The application uses the data gateway to handle the connection and communication with any data source. Additionally the data management component can model the received information into abstract objects called data models which would then be used in constructing the business logic and presenting it to the user.

It should be noted that the data models we are referring to are not necessarily the logical models also present as database physical concepts (such as tables). To create a better understanding between the business requirements and the business logic which is going to be implemented in the platform a conceptual data model can be created. This abstraction is technology independent and is a distinguishable entity present in the dictionary of the business requirements.

Fowler (2002) defines a set of patterns that are applied for modelling data entities, their relationships and behavior. In handling data sources he identifies four types of data gateways:

- **Table Data Gateway** - An object that acts as a gateway to a database table. One instance handles all the rows in the table.
- **Row Data Gateway** - Can be defined as an object that acts as a gateway to a single record in a data source. There is one instance of such an object per row.
- **Active Record** - This gateway wraps a row in a database table or view, encapsulates the database access, and adds business logic on that data. The difference between this pattern and the Row Data Gateway is that it contains additional business logic for each row object.
- **Data Mapper** - A layer of independent objects called Mappers that transfer data between them and a database. This type of gateway completely separates the in-memory objects from the database instances making it possible to add complex business logic relationships like inheritance.

These patterns help in making the platform database agnostic by implementing universal methods of accessing and querying the data storage solution. In order to allow higher customization they often accept statements in the native procedural languages provided with every major database, such as Structured Query Language (SQL).

Although four basic data gateways are outlined above, most of the application frameworks at the time of this research use two orthogonal object-oriented design patterns: the Data Access Object (DAO) pattern and Object Relation Mapping (ORM). Both standards come as an abstraction layer above the data sources and handle the connection and communication between the application and the information provider. The similarities and differences are presented in Table 1.1.

Similarities	**Differences**
- acts as an object oriented intermediary between the application and a type of database or other data persistence storage solution	- *DAO* refers to singular objects that create an abstract interface with the database tables
- does not expose the details of the underlying database	- *ORM* refers to creating a virtual object database that can model multiple entities (objects) and the relationship between them
- is bidirectional, moving data back and forth between objects and the database	
- handles database connections and contains an abstract interface for running CRUD (create, replace, update, delete) queries	

Table 1.1: Differences and similarities between DAO and ORM

While DAO tries to simplify the database access by creating an abstract persistent interface to the data source, the Object Relation Mapping pattern can relate to as a type of Active Record or Data Mapper, incorporating complex business logic and making it easy to map any relationship between models.

1.2 Problem statement

Although one of the main reasons for selecting a specific software platform is the fact that it satisfies a large part of the required functionality, it is often encountered that this does not fully incorporate the needed features for the situation at hand. In this case the software businesses that are using the platform must come up with other solutions.

For example, a closed-source software platform provides only a set of limited public extendability points, therefore substantial extensions of the framework often requires bypassing the limitations that the platform's architecture imposes (Bourquin & Keller, 2007). This problem can be avoided by doing a more comprehensive analysis during the platform selection process.

The decision to use an application framework is rarely influenced by its software architecture (Jansen, 2013), therefore leading to initial unawareness of its limitations. This makes it difficult to avoid future situations that involve breaking the general conventions of the software platform in order to fulfil the business requirements of the product. The problem of how data is being managed is critical for successfully implementing a software project, in consequence the selection of an appropriate pattern for the data management architecture upon which the product will be built must entail understanding of the inherent constraints.

Because it provides a simplified, object-oriented way to work with databases, an ORM component is often considered to be used in the data management part of the product architecture. Even though it provides immediate benefits when employed, it also brings negative effects to various application quality attributes.

The current scientific literature in the software architecture domain does not specifically address the issue of architectural data management patterns and the effect that they have upon the overall platform. This research project looks at one of the popular patterns employed in the data management component, the Object Relational Mapping tool. By analysing the outcomes of using such a component we provide valuable insight to any of the interested parties, i.e. platform creators and extenders.

> *The usage of an ORM component as part of a software platform data management architecture brings many immediate advantages to the platform extender. However, because of the complexity of the problem, the decision to include such a component is made without understanding the imposed constraints and leads to future unpredictable shortcomings.*

1.3 Thesis outline

Chapter 1 introduced the reader to the problem that is being addressed in this research project by outlining the context and key concepts related to the subject.

Chapter 2 will present the research objective and supporting research questions along with the methods that are going to be employed to achieve the end goal. Along with the detailed description of the research approach a Process-Deliverable Diagram is included to elucidate the steps that were undertaken and their respective deliverables. At the end of this section a validity analysis is made.

Chapter 3 gives a detailed look on the key concepts of the research topic resulting from the structured literature review, case study and documentation analysis. The first section presents the candidate architecture of a data management component in a modern software framework. The next sections continue on developing the subject of what is an ORM and what challenges does implementing one may have.

Then, in Chapter 4 a presentation of the list of affected quality attributes when an ORM component is used is given. Each quality attribute is then further discussed. A set of measurement criteria are established and an experiment is performed for each one to validate our assumptions.

Chapter 5 starts by describing the general decision-making process and the available methods that can be employed. Each method relevancy is discussed and a selection is made for the construction of the research project final deliverable. The last section of this chapter presents the created model.

Chapter 6 entails how the evaluation of the model was carried out and what were the results of this evaluation.

Chapter 7 further discusses the decision support model, how can it be used and what are the implications in that regard.

Chapter 8 closes the research project presenting the conclusions and discussing how the research could be extended with future investigation.

Chapter 2

Research approach

This chapter describes the end objective of the research project and what are the steps which will be applied in order to attain it, including a set of supporting research questions, detailed explanation of the used research methods and a Process-Deliverable Diagram depicting the research approach.

2.1 Research objective

Considering the issues explained above, we can bring a clear definition of our main research objective:

> To construct a decision support model for determining whether an Object Relational Mapping tool is suitable for the data management component of the used software platform.

The main goal of this model is to present the capabilities and constraints of the ORM pattern in the light of the data requirements of the end product. This will be shown in relation with the effect it has upon the platform's quality attributes. A practical outcome of this document would be to support the selection decision of a specific software platform.

2.1.1 Stakeholders

The key stakeholders would need to have a role that involves direct and significant influence upon the software product architecture. Depending on the size of the project this can be a Software Architect, Software Developer or a Product Engineer.

Secondary stakeholders that intermediate the communication and extract requirements can also be accounted for, i.e. Project Managers or Business Consultants.

2.2 Research questions

The main research question can be formulated in the following manner:

> ***Q: What are the criteria based on which a software architect can make a decision for using an ORM tool in a software platform?***

In order to distinguish the criteria needed for making a knowledgeable decision we need to form a greater understanding on the ORM pattern and the challenges that its implementation faces. In result we can contrive the first supporting sub-question as:

> *Q1: What are the challenges that come from applying the Object Relational Mapping pattern?*

By evaluating the obstacles that an ORM tool implementation might encounter we identify any positive or negative effects that the usage of such a tool will trigger, leading to our second sub-question:

> *Q2: What are the consequences for using an ORM tool in the data management component of a software platform?*

In sequence the previously found consequences would need to be translated into something comparable and measurable in connection with the overall software platform and end product. This can be done by identifying the affected quality attributes. Hence, the sub-question:

> *Q3: Which quality attributes are influenced by the usage of an ORM tool?*

Each significantly affected quality attribute is individually evaluated creating an overview on how the data management component of the architecture influences the whole framework revealing the objective of this research.

2.3 Research process

This section explains the method applied for achieving the research project's end objective. On account of having a decision support model as the main deliverable, we can classify our research approach as being a design science research. Also outlined by Hevner, March, Park, and Ram (2004), design science "creates and evaluates IT artefacts intended to solve identified organizational problems". The artefact in this project involves detailed descriptions of the data management ORM architectural pattern merged with the effect that it has upon the platform quality attributes.

As a base for conducting this research a grounded theory (Glaser & Strauss, 1967) approach is considered. The topic addressed in this project is not extensively addressed in the literature, therefore a relevant initial theoretical framework could not be applied. By collecting data from various sources and analysing them at a conceptual level we can draw conclusions with objective theoretical insights.

Research question	**Research approach**
Q1	Literature study Documentation analysis
Q2	Literature study Case study
Q3	Literature study Case study Design of experiments
Q	Design science

Table 2.1: Research approach used for each research question

In order to better outline the boundaries of the topic and establish a clearer context, a set of qualitative research methods are going to be used: literature review, documentation analysis and a case study. The gathered information would then be employed for identifying the main characteristics of a data management component. This would contribute to defining the main challenges and consequences of the data management ORM pattern.

The research process will consist of several non-sequential steps which are going to be detailed in the subsections below.

2.3.1 Structured literature review

A structured literature review will be performed in order to identify the challenges of the Object Relation Mapping pattern as part of a data management component. A part of the ideas found in the scientific literature have already been discussed above. This step will improve the information on already discovered concepts and extend the list with new ones.

Aside from discovering architecture patterns the literature will be examined on the topic of software applications quality attributes. In order to infer the relationship between the ORM tool and the platform's quality attributes a greater understanding about what they describe and how can their performance be measured is necessary.

A list of preliminary keywords will be created based on the initial research which will be used for querying academic research search engines, such as Google Scholar, in order to discover related publications. The returned results and their references are analysed and filtered. This process will be repeated several times after the list of keywords is refined contingent on our findings. To ensure higher relevance the publications from the last 10 years will be favoured against the older ones.

2.3.2 Documentation analysis

A set of ecosystem based software platforms will be chosen for closer analysis. To ensure the availability and completeness of their documentation, the most popular at that time will be selected. Each framework is inspected so that the main architecture characteristics and functionality of the data management component is recognized.

The framework selection process will consist of the following steps:

1. Find out which are the most used (on the basis of market share) and the most popular (that have a significant positive usage trend) programming languages to date. Use quantitative ranking systems such as the TIOBE Programming Community Index.

2. Create a short list of programming languages which cover more than 80% of the market share.
3. In consensus with its market share select an appropriate number of frameworks to be analysed for each programming language.

Each selected software framework will then be analysed from the perspective of the research topic. As such the main data management component will be identified and its main features explained. The interdependence between the component and the entire software platform is also an important factor to be noted when analysing the architecture.

2.3.3 Case study

As a way to comprehend the context of the research topic and get a grasp of its application in a real life situation, an explanatory holistic multi-case study (Yin, 2009) with data collected from multiple sources will be performed. The data gathering techniques include:

- Conducting interviews at several software businesses that make use of ecosystem based platforms and identify how do they use the data management component of the framework. To support the validity of their knowledge, the people that are going to be interviewed should be directly involved in the development process of the product. This includes both computer programming or designing the software architecture. The interview will be focused on two parts: the method in which they employ the data management component and what has been the effect of that component on the quality attributes of the application. A preliminary interview guideline document has been created and is present in Appendix B.
- Gathering data about existing data management platform components and their usage issues from querying development Q&A websites (such as StackOverflow, CodingStack, CodeRanch). As the afore mentioned websites generally include issues with a high degree of specificity, this step will be executed in a latter phase in order to acquire the necessary knowledge to be able to filter out and classify the information.

The case study does not have the purpose of answering our research questions or providing clear explanations for what are the challenges when using an ORM, but rather give an indication of what the experts in the domain know about the issue and how do they usually choose to handle it. Further elaboration on the subject will be done with some of the case study subjects as they are going to be used to evaluate our final deliverable.

2.3.4 Quality attributes impact measurement

Once the research context is clarified and the challenges for using an ORM tool are identified we pursue on discerning the affected quality attributes of the application framework. This is done to show that the usage of an ORM component has a significant effect on the platform. Along with identifying the influenced software qualities, this also helps in quantifying the consequences as they are more extensively covered in the literature.

The list of the influenced quality attributes is constructed through logical inferences from the previous gathered data and through the design of several computer experiments which measure the impact upon the software platform. The process of establishing the measurement criteria and performing the measurements is further explained in Chapter 4.

2.4 Process-Deliverable Diagram

With the purpose of creating a simple but insightful view upon the employed research approach, a Process-Deliverable Diagram (PDD) has been built according to the method described by van de Weerd and Brinkkemper (2008). The resulting diagram, depicted in Figure 2.1, represents the main activities employed in the research method along with every deliverable that emerge from them. Every activity is put in chronological order, outlining concurrency when applicable, and having the outcome deliverable on the right side.

As support for the PDD, every activity and concept (deliverable title) are further explained in Table A.1 and Table A.2 present in Annex A.

2.5 Validity

The assessment of the quality of this research project is done by applying the validity tests as described by Yin (2009), therefore this section covers the topics of construct validity, internal validity, external validity and reliability.

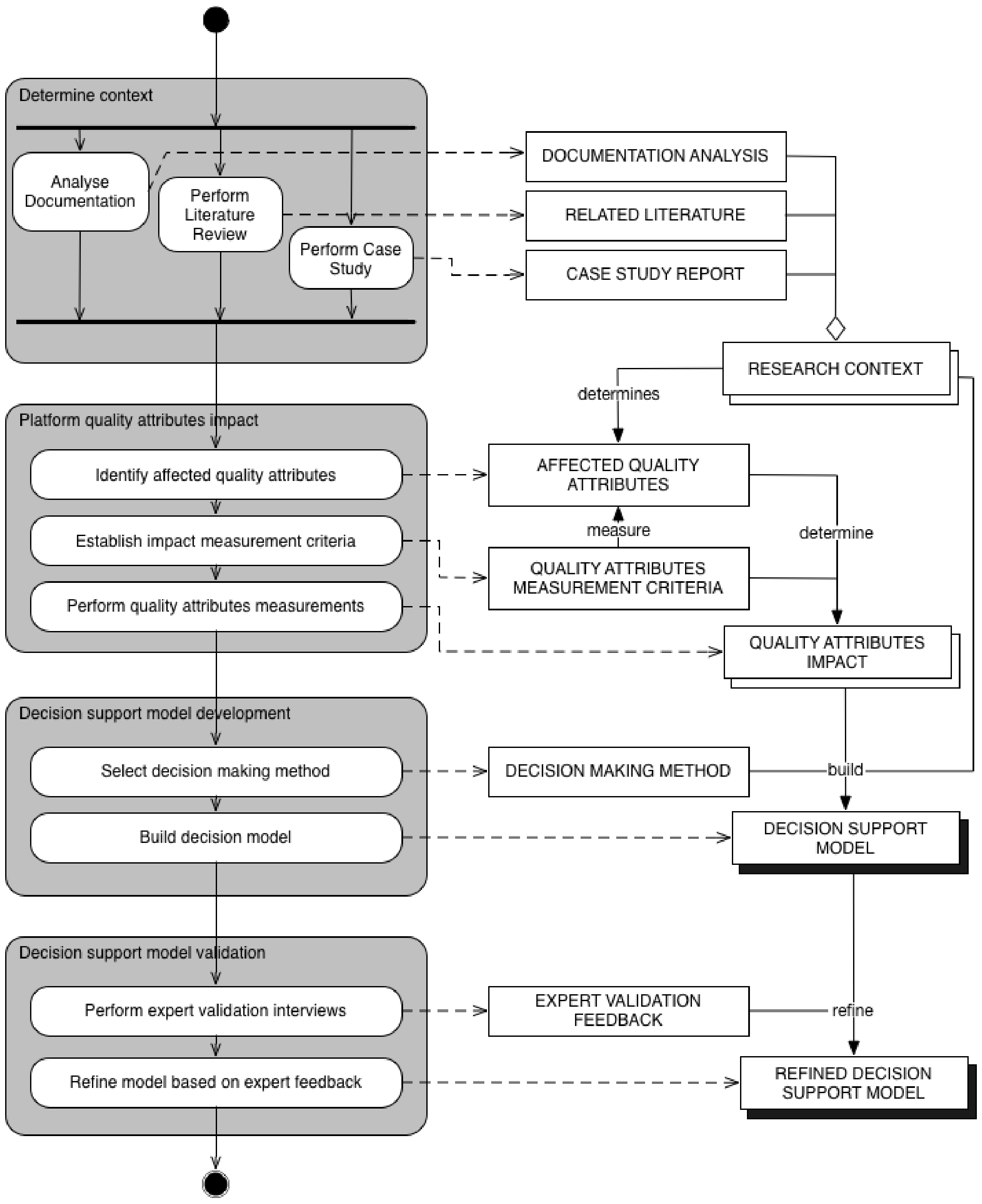

Determine context
Analyse Documentation
Perform Literature Review
Perform Case Study
DOCUMENTATION ANALYSIS
RELATED LITERATURE
CASE STUDY REPORT
RESEARCH CONTEXT
determines
Platform quality attributes impact
Identify affected quality attributes
Establish impact measurement criteria
Perform quality attributes measurements
AFFECTED QUALITY ATTRIBUTES
measure
determine
QUALITY ATTRIBUTES MEASUREMENT CRITERIA
QUALITY ATTRIBUTES IMPACT
Decision support model development
Select decision making method
Build decision model
DECISION MAKING METHOD
build
DECISION SUPPORT MODEL
Decision support model validation
Perform expert validation interviews
Refine model based on expert feedback
EXPERT VALIDATION FEEDBACK
refine
REFINED DECISION SUPPORT MODEL

Figure 2.1: Process-Deliverable Diagram of the Research approach

2.5.1 Construct validity

The issue of construct validity refers to the decision of correctly adopting operational measures when studying the concepts involved in the research. For the most part being an issue for projects in the data collection phase especially when a case study research method is being used (Tellis, 1997), the solutions proposed by Yin (2009) involve ensuring the objectivity of process through the use of multiple sources of evidence and the identification of the chain of evidence.

To conform with these requirements a multi-case study is performed at companies, which don't have a direct business connection and use different technologies for developing their products. This is done not only to provide a broad view upon the current state of the software industry on the research topic but also ensures that the created decision model will receive feedback from a contrasting collective.

2.5.2 Internal validity

Internal validity relates to the correctness of inferring causal relationships. Being only an issue for explanatory studies in the data analysis phase, it raises concerns when not all involved factors are considered.

Due to the fact that this research focuses on a problem which only has two states (the presence or absence of an ORM tool in the software architecture) it significantly reduces the number of failures to catch certain conditions or factors when interpreting the gathered data. All the causal relationships being made are validated through experiments and expert validation.

2.5.3 External validity

The problem of external validity is focused on establishing the boundaries to which the study's presumptions can be generalized beyond the context of the case study or other collected data. This ensures an objective view on the studied topic and eliminates the failure to extract outcomes specific to a broader population.

The decision support model we are creating includes data from multiple sources that have real implementations in the software industry and a replication logic has been applied when collecting the data. As a support method, a set of experiments have been carried out to validate the quality attributes impact presumptions.

2.5.4 Reliability

The reliability of the project stands for the ability to replicate the study results when the data collection and analysis procedures are applied in a similar manner. This implies a thorough documentation of all the steps taken during the research as well as rigorous explanation on how to employ them.

Along with a Process-Deliverable Diagram (see Section 2.4) which describes the whole process activities along with their consequent deliverables, the case study interviews have been conducted following the guidelines present in Annex B and all the computer experiments followed are detailed in Annex D. The structure of this document follows the logical sequence of the steps followed.

Chapter 3

Research Study Findings

This chapter is based on the detailed examination of the project context by applying the following three research methods: performing a structured literature review on the topic, analysing the documentation of several software application platforms and conducting a case study.

As a result, the first section goes into more details on how the data management component works and presents its architecture fragments. We continue with a section on where in this architecture does the Object-Relation Mapping module fits and what are its main functions. The last section discusses the challenges that arise when trying to build an ORM tool.

3.1 Data management component architecture

As a preliminary step for establishing a clear image of what the data management component of an application framework might look like, the scientific literature in the domain has been analysed.

Having a vital role in the software creation process, the software architecture consists of several architectural elements that have a particular form (Perry & Wolf, 1992). This architectural form on its own is composed of properties and relationships that have the function of defining constraints either in choosing that specific element or in its placement (organization and interaction with other architectural elements).

Because the relevant literature did not detail the topic to the extent that is needed in this study we have pursued in analysing the documentation of several application frameworks currently used in the software industry.

As a starting point we selected nine programming languages used in enterprise applications by looking at the TIOBE Programming Community index. This index is an indicator of how favoured are certain programming platforms based on the number of engineers specialised in that language, courses, third-party vendors and relevant hits on more than 20 search engines.

Considering the gathered short list of programming languages and the availability of ecosystem based application frameworks for them, we selected a number of eleven frameworks for which the documentation will be closely analysed so that the data management component architecture can be distinguished. A detailed report of the documentation analysis can be found in Appendix C.

One observation that could be made immediately after analysing the report is that almost all the popular modern application frameworks have a strictly modular architecture following the single responsibility principle (Martin & Martin, 2006). Although created more than 25 years ago by Krasner, Pope, et al. (1988), the Model-View-Controller (MVC) paradigm is still used in most of the platforms, providing a straightforward separation between data entities, their behaviour and the end-user presentation (views). This means that the data management component will always reside in a completely separate module which for some instances can also be completely detached from the application.

With the rise of agile development several software design paradigms have gained popularity that adapted to the change-responsive release cycle. During the documentation analysis it was noticed that there are frameworks which give a high importance to implementing these paradigms such as *don't repeat yourself* (DRY), first presented by Hunt and Thomas (2000), or Convention over configuration (CoC).

A layered system architectural style (Garlan & Shaw, 1994) was applied to model the data management component for modern frameworks (see Figure 3.1), providing a hierarchy of the inner components. Each layer provides services to the layer above it.

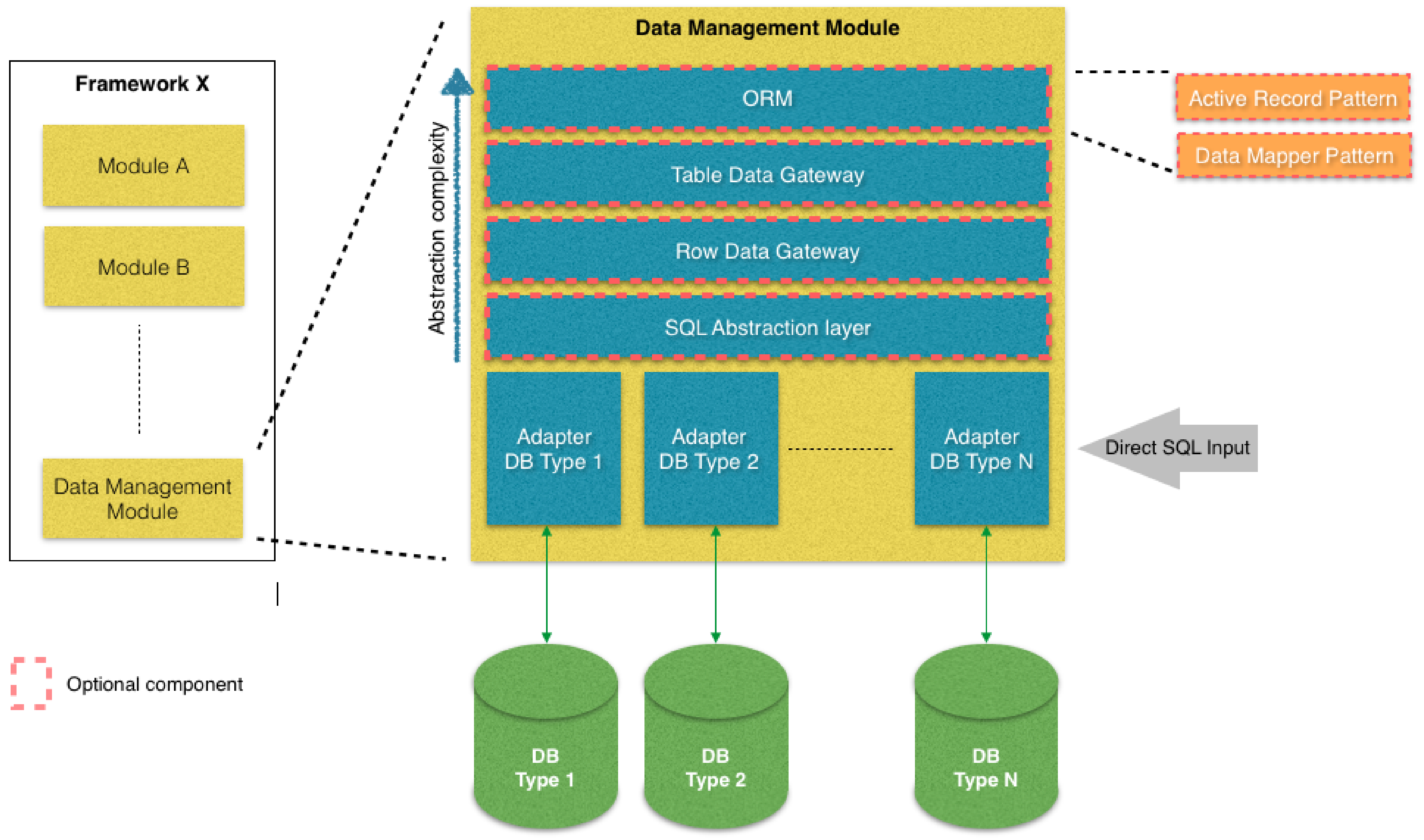


Figure 3.1: Data management component architecture

3.2 Object Relational Mapping

In general an ORM refers to a software component that maps data between incompatible type systems in object oriented programming languages by creating a virtual database that can be used from within the language. By doing this it creates a complex persistence layer for the application that falls beyond the scope of the developed environment into a storage solution.

A common example would be mapping the data from an SQL database in an object oriented environment, including not only the translation between the object state and a database column/row but also a definition on how to generate manipulation queries. This means that the ORM component would be handling the database connection, create an abstract interface that would allow basic operations to be performed in an object oriented fashion and maintain a bidirectional synchronization between the created objects and their persistent storage solution counterparts.

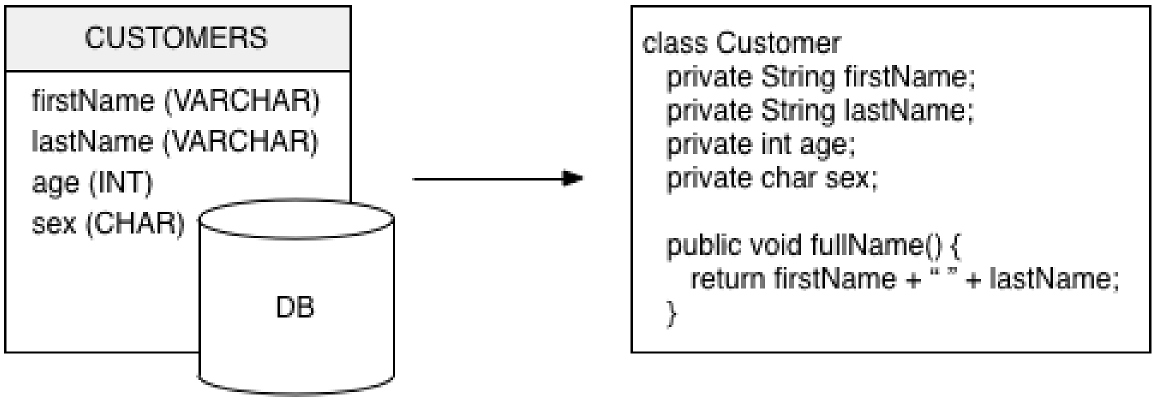
The object paradigm borrows concepts from the software engineering field, whereas the storage solution focuses on the mathematical set theory (Ambler, 2000). Applying an object-oriented approach for controlling the underlying application data streamlines the development process and brings several benefits (Rumbaugh et al., 1991).

An ORM exists in the form of commercial or open source packages, independent or built into a software framework, but often the developer can also opt to build one from scratch in order to better adapt to the situation at hand. Even though a customised solution would seem enticing at first the challenges faced by building an ORM tool are not trivial and raise multiple issues on their own as explained in section 3.3.

Often following the Active Record pattern described by Fowler (2002), an ORM tool allows creating prototype classes for business entities and links them back to the storage solution, not only connecting the object oriented environment with the actual data but also modelling behaviour (ex. calculating any customer benefits as soon as his age changes).

Also mentioned by Juneau (2013), by applying a higher level of abstraction on the data present in the storage system an ORM component also makes use of the following features:

- Defining abstract entities which have a direct correspondent in the business side of the software application. By making sure the terminologies used are the same, the development process better aligns with the product business strategy eliminating any eventual language non-conformity. An example of how a database table could be linked to a prototype entity class can be seen in Figure 3.2.



```
class Customer
  private String firstName;
  private String lastName;
  private int age;
  private char sex;

  public void fullName() {
    return firstName + " " + lastName;
  }
```

Figure 3.2: Example of mapping database table to prototype class

- Automatically synchronize entity specification with database schema regardless of the database type through the use of adapters (database agnostic). More advanced ORM tools can manage all the database migrations that were applied over time and keep track of the state in which a certain database resides.
- Modelling relationships (one-to-one, one-to-many, many-to-one, many-to-many) between the defined entities. By creating tangible connections between concepts it creates an opportunity for assuring tight coupled data that can be accessed and validated in a straightforward manner.
- Easy to understand and write queries independent of the underlying query language of the storage solution. It is often the case that an ORM will include an implementation of a high level query interface that is very similar to the DAO object pattern. The readability benefit that comes from this can also be depicted from the example found in Table 3.1.

Ruby on Rails ORM	**Generated SQL**
`company = Company.find(2)`	`SELECT ‘companies‘.* FROM ‘companies‘ WHERE ‘companies‘.‘id‘ = 2 LIMIT 1`

`company.sites.where ("created_at > ?", 1.day.ago)`	`SELECT ‘sites‘.* FROM ‘sites‘ WHERE ‘sites‘.‘company_id‘ = 2 AND (created_at > ’2014-07-16 10:32:26’)`
`company.update_attributes (name: "Test")`	`UPDATE ‘companies‘ SET ‘name‘ = ’Test’ WHERE ‘companies‘.‘id‘ = 2`
`company.destroy`	`DELETE FROM ‘companies‘ WHERE ‘companies‘.‘id‘ = 2`

Table 3.1: Example of Ruby on Rails ORM query transition to SQL

- Add validation of object attributes to improve data integrity and consistency. Because of the presence of concrete entity relationships the validation process can also include complex interdependencies that enforce cross-entity limitations.
- Implement behaviour triggers that mimic the actual business process. These callbacks are launched as soon as the object state is being altered (on/before create, update, destroy).

The complexity of the ORM tool varies between available implementations. Simple solutions such as *ORMlite* provide only basic database access and query abstraction along with entity definition, while more intricate ones such as *Hibernate* include the whole array of features mentioned above plus different adaptable caching techniques (for example lazy loading or eager loading), scalability support or even complete object-oriented idioms support for the concept classes.

The creation of an effective ORM tool is considered a notable difficult problem because of the vast array of issues that it needs to address (Neward, 2006). The next section presents an overview of these concerns.

3.3 Object-relational impedance mismatch

The challenges that arise in the process of mapping data between the object-oriented environment of the development language and the storage system are incorporated under the umbrella term called *Object-relational Impedance Mismatch*. As outlined by Sam-Bodden and Judd (2004) the problems encapsulated in this term do not refer only to technical issues but also to a cultural roadblock formed by two groups of people with different ideologies: on one hand trying to represent everything as objects with the scope of solving business requirements, on the other hand trying to flatten and normalize all the collected data in order to provide efficient and performance concerned storage and access.

It is therefore critical to understand that the object and relational approaches are constructed on different foundations, that in turn come with distinct abstractions or organization rules. Ireland, Bowers, Newton, and Waugh (2009b) classify the problems of object-relational impedance mismatch under six categories:

- **Structure** - is relevant when discussing about the design of the representation and the way it is organized in the system. It should be noted that a class could provide additional arbitrary structure through the use of methods which makes the object structure volatile in comparison to a table row which has a fixed anatomy. In object-oriented programming a repetition mechanism is available through the use of inheritance and polymorphism creating supplementary hierarchy (such as a class extending another class) not present in relational databases. In short this means that the representation of an object-oriented class has no direct correspondent in SQL making the bidirectional synchronization more difficult. Differences are depicted in Figure **??**
- **Instance** - calls attention to the instantiation of concept representations, their state and how they behave. In accordance with its foundation ground (set/relational theory) a row depicts a definite statement about a domain of discourse, in opposition to an object which can contain arbitrary structures or behaviour. This entails that the state of an object for example cannot not be fully preserved in a relational storage system and certain limitations need to be fixed. Moreover while the primitive data types in object oriented programming languages are similar, those from relational storage systems vary to a higher degree usually adapting to the type of stored data (see Figure 3.4 for example). This means that certain storage data types may need additional

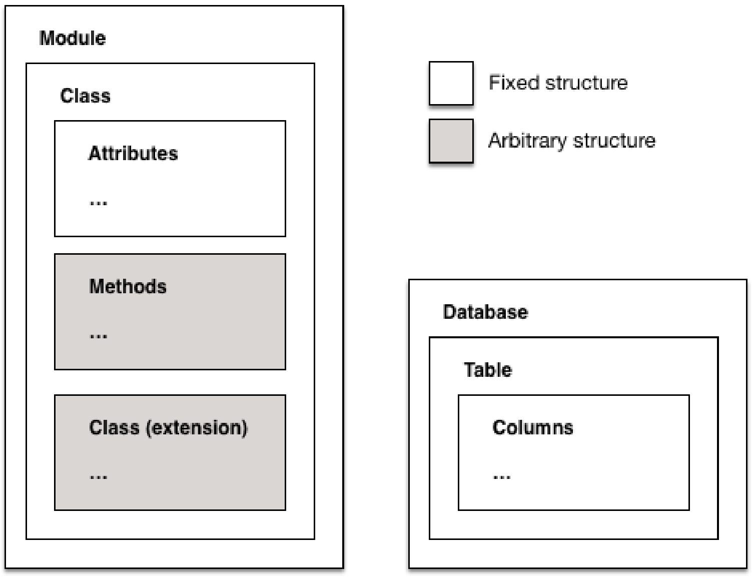


Figure 3.3: Structure hierarchy differences between object-oriented and relational systems

translation logic when mapped.

- **Encapsulation** - refers to the distinction present in how the structural and functional unit (i.e. an object or a table row) is organized and accessed. In contrast to having a hidden representation with restricted access to its properties and contents, a relational table row does not provide this functionality leading to security concerns on its condition consistency. Encapsulation is one of the four fundamental paradigms that are present in object-oriented programming and it can exist even in high-level structural units such as classes or modules.
- **Identity** - The concept of identity is different between instances of classes and a relational system. While in a database a row is always persistent conserving it's state and identity, objects exist in a temporary in-memory environment where the identity differs from its state. This leads to the existence of two objects with equivalent states but different identities raising further issues on handling such occurrences.
- **Processing Model** - Due to the above mentioned structural differences, managing changes on a certain instance requires further attention maintaining the state correctly synchronized. This has to be done keeping in mind that there are transactional differences (the unit of

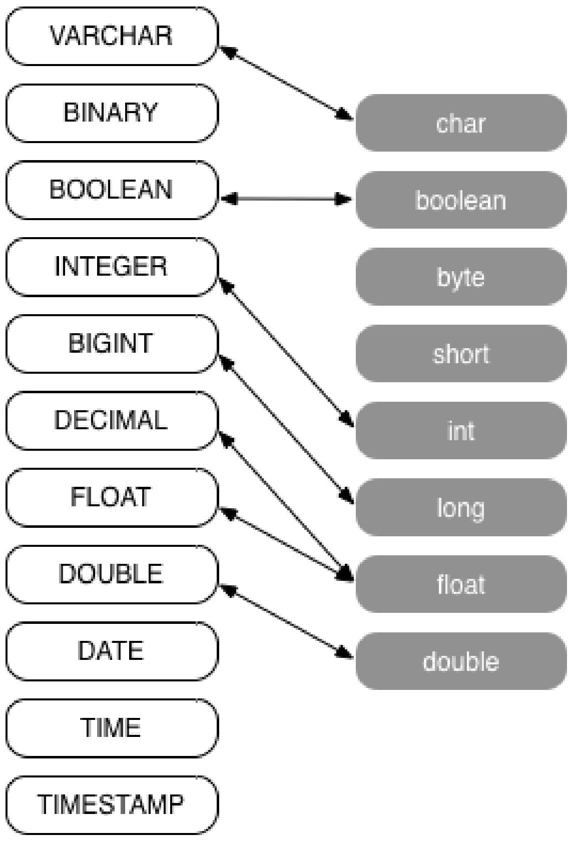


Figure 3.4: Example of primitive data type mapping between SQL and Java

work for OOP instructions is smaller than database transactions), manipulative differences (queries include only primitive declarative operations whereas objects can perform more complex imperative operations – see Table 3.2) and environment differences (objects are stored in limited memory space).

Operation type	**Example**

Declarative operation
(what to do)
	`UPDATE ‘companies‘`
	`SET ‘status‘=’active’`
	`WHERE ‘companies‘.‘id‘ = 2 LIMIT 1`

Imperative operation
(how to do)
	`def activate!(company_id)`
	`  Company.find(company_id)`
	`  .update_attributes(status: ’active’)`
	`end`

Table 3.2: Example of declarative and imperative operations

- **Ownership** - involves the responsibility prioritization for maintaining the functionality and integrity of the system. A relevant example might be that the class model and database schema are owned by different teams creating a necessity for change processing rules to create a valid correlation.

Another highly debated matter regarding ORM tools is the issue of extreme convenience to the point of ignorance (Fowler, 2012; Keith & Schnicariol, 2010). Due to the highly abstracted interface for persistent storage systems, software developers using an ORM component often tend to ignore the underlying database technology and write code without any significant knowledge on the topic. While the solution of mapping persistent storage units to object-oriented classes creates a streamlined development environment the underlying problems still exist. Not being aware of the challenges of the ORM makes it problematic not only to fix the potential issues but also to predict them.

The diversity of IT systems and technologies makes it impossible to successfully create a completely database agnostic solution that accommodates any use case. The mapping problem should not be hidden from the platform extender, this is why the scope of an ORM component should be limited to providing a way to avoid rewriting repetitive storage related tasks (reusability of existing data management practices).

In an effort to provide a clear process on how to tackle the problem of implementing an Object-Relational Mapping tool, Ireland, Bowers, Newton, and Waugh (2009a) provide a framework for identifying the common levels of abstraction and the root cause for the mapping mismatch. The created process not only helps understanding the source of any issues that may be encountered, but also gives instructions on how to address them at the most convenient level of abstraction.

Chapter 4

Affected quality attributes

Every software project begins with creating a candidate application architecture that defines the relevant components and the relationship between them. Problems identified early in the development process can be solved in a quicker manner, as such evaluating the software architecture of a platform even in its early stages avoids bigger costs for resolving issues in the product testing phase (Clements, Kazman, & Klein, 2003). The evaluation forces the stipulation of certain quality goals which better scope the architecture and settle any potential conflicts. This is done through the analysis of quality attributes.

Quality attributes, also referred to as non-functional requirements, are system properties that help define and measure how the product is performing. The use of concrete metrics to estimate software quality identifies potential problems and leaves space for further improvement of the system. To assure better customer satisfaction, it is a good practice to consider targeting certain quality attribute goals from the planning and design phase (Kan, 2002).

The failure to meet user necessities often is a result of software developers focusing on resolving all business requirements through the application functionality without correlating them with the impact on the quality parameters of the system (Barbacci, Klein, Longstaff, & Weinstock, 1995). It should be noted however that the quality attributes of the system are frequently incongruous, making it difficult to create an all-around linear system metric. This constrains the designer to achieve a balance between several attributes where the trade-off is also acceptable for the customer.

As a result of careful analysis of the data management component of multiple software platforms, along with the information gathered from industry experts via the performed case study and the structured literature review, we have come to the deduction that an ORM software component in general will affect the following quality attributes: **performance, maintainability, scalability**.

Although other quality attributes might be affected by this component, we believe that it is to a smaller degree or not common to all ORM tools regardless of how it is implemented. As such the following list of common quality attributes was discarded:

- **Reusability**. While an ORM component can contain a vast array of features which perform common practices when working with databases, this is not the case for all implementations. The degree of reusability is relative to the complexity of each implementation, most of them being limited to the basic CRUD (Create / Retrieve / Update / Delete) operations relative to abstract concepts.
- **Availability**. Although one can argue that the proportion of time when the system is functional can be smaller due to a higher number of components included in the architecture that can fail, the available ORM components are generally stable and influence this quality attribute in a smaller degree.
- **Manageability**. The management of a platform which contains an ORM component brings no advantages or disadvantages through its functionality to the person responsible for this task.
- **Reliability**. Although the reliability of a system can be affected by the negative effect on performance or scalability, we feel that the impact on reliability is dependent on the influence pertained on the other two quality attributes. Because of its secondary status we have discarded it from our analysis.
- **Security**. An ORM component can contain various security practices implemented in the storage process (such as SQL injection). Due to the limited availability of these features and its direct link with the reusability quality attribute we have chosen to not analyse it further.
- **Usability**. Although the abstraction of the storage solution brings several usability benefits such as the ability to write common operations more efficient, this quality attribute was dismissed due to the component's scope. As its purpose is not to completely replace working with the underlying technology, an ORM would require an additional effort for learning how to use it, while continuing to partially check and

optimize how its operations translate to database queries.

- **Supportability, Testability**. Not applicable due to the component scope and functionality.

The next sections further detail each analysed quality attribute, the cause and effect of an ORM on that parameter and presents the validation experiments performed during the research study.

4.1 Performance

Performance is one of the indicators for the quality of service of a software product along with availability and reliability (Hasselbring & Reussner, 2006). This quality attribute is concerned with how long does it take for the application to respond to a certain event, either triggered by the user or by other internal/external stimuli, and its throughput for a fixed time interval. The difficulty of establishing a system performance comes from the variety of request sources and arrival patterns (Bass et al., 2012).

Failure to achieve a satisfactory performance affects the application responsiveness and the productivity of its users causing loss of revenue and supplementary costs for the necessary adjustments on the system (Williams & Smith, 1998). Moreover the changes applied on the product in a latter stage has increased costs, which is why a performance evaluation should be started when designing the software architecture. Kazman et al. (1998) proposes a method for evaluating the architecture quality called Architecture Trade-off Analysis Method (ATAM). This method should be considered in order to validate the architectural decisions made that have significant impact on the application quality attributes.

Due to the fact that an ORM tool constitutes an additional abstraction layer above the database connection adapter addressing a non-trivial mapping problem between two incompatible systems (as detailed in Section 3.3), we infer that the inclusion of such a component may have a negative impact on the performance of the platform. This has been also confirmed by several of our case study interviewees who have encountered performance issues with using ORM implementations in their software projects.

4.1.1 Measurement criteria

The **Response time** is a metric that represents the amount of time it takes the software application to return a response on a given input. Because of the nature of our studied topic we will consider that the request for service will involve only database queries eliminating other solicitations (such as network activity, disk IO) from our discussion. The response time is composed of the wait time and the service time, which represent the time spent in queue waiting for the system to take over the task and the actual time spent to resolve the task.

Throughput refers to the number of requests that can be completed by the system in a fixed interval of time. This can be calculated by dividing the time period by the average response time. Due to the fact that various factors can intervene with the throughput it is usually the case that separate tests are ran so that real environment conditions are considered.

4.1.2 Experiment

This experiment has been conducted in aid of our statement that an ORM component has a negative impact on the platform performance quality attribute. Due to the high abstraction being done between two incompatible systems, the supplementary logic has a significant influence on the response time of the software framework. This is a result of several mapping difficulties such as: managing primary data types which do not exist in one of the system or synchronization of information state due to differences in the size of the unit of work.

It should be noted that because of the differences between programming languages, software frameworks and ORM implementations this test should not be considered as an exact benchmark for performance, but more as a validation for the negative trend that is common regardless of technology used. The

In order to test the performance of the data management component of the system we have created a concept schema composed of three models: a **Company** which can have multiple **Employee**s which can hold multiple **Document**s. Each entity has three description attributes along with a primary key for identification, the appropriate foreign keys for relationship modelling and two logging attributes that register the date at which the

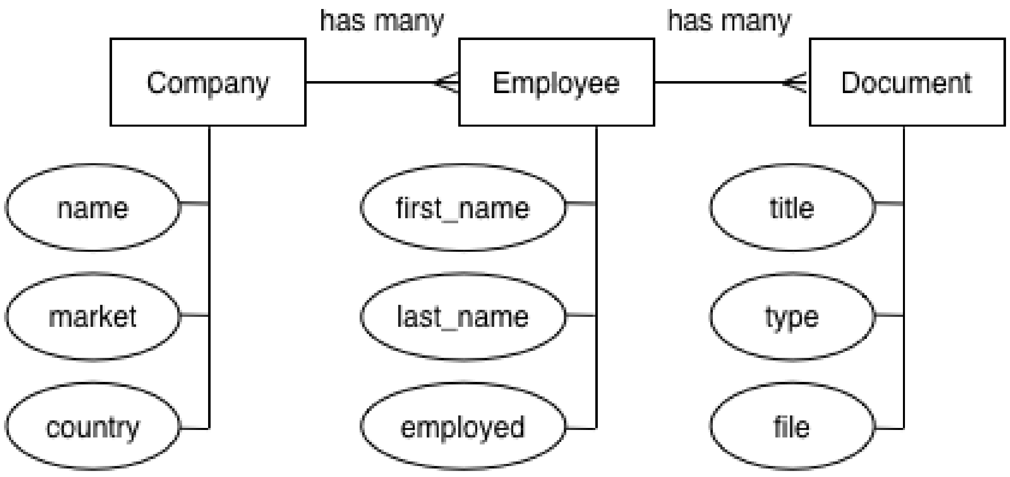


Figure 4.1: Concept schema ERD

record was created and updated. A graphical representation in the form of an Entity-Relationship Diagram (ERD) is depicted in Figure 4.1.

A persistent storage solution has four basic operations that are frequently ran by a data management component: create, read, update and delete. Also known under the acronym CRUD, these actions also refer to the SQL commands CREATE/INSERT, SELECT, UPDATE and DELETE. The following pseudo-code (Figure 4.2) describes the experiment process steps that we undertook to measure the average response time of the data management component.

The test was divided into three sections which were benchmarked:

- Create entity instances and the relationships between them.
- Browse through records and their relationships and update them.
- Retrieve records and their relationships and delete them.

Each section provides a response time value upon running the test which indicates the time from start to finish to perform the operations as measured by a regular clock. In order to avoid inaccuracies all tests were performed several times and an average response time value was calculated.

To prove the results' independence from a specific software platform the experiment was done for several frameworks. The selection of the test platforms was carried out taking into consideration the previously analysed documen-

```
Benchmark time to run:
  Create Company
  For i from 1 to 10 run:
    Create Employee i for Company
    Create 10 Documents for Employee i

Benchmark time to run:
  Read Company
  Update Company
  For each Employee i from Company
    Read Employee i
    Update Employee i
    For each Document j from Employee i
      Read Document j
      Update Document j

Benchmark time to run:
  Read Company
  For each Employee i from Company
    Read Employee i
    For each Document j from Employee i
      Read Document j
      Delete Document j
    Delete Employee i
  Delete Company
```

Figure 4.2: Performance experiment pseudo-code for measuring response time

tation and the cost of setting up a valid test environment, therefore four open-source frameworks were chosen written in different programming languages: Grails (Java), Yii Framework (PHP), Django (Python), Ruby on Rails (Ruby). All chosen platforms have the ORM component loosely coupled, for that reason they can be either removed or disabled for the purpose of this test.

The experiment results are aligned with our previous statement showing a significant negative trend for running database queries through an ORM component as opposed to running them directly through the database adapter by using SQL. Therefore the response times recorded when using an ORM

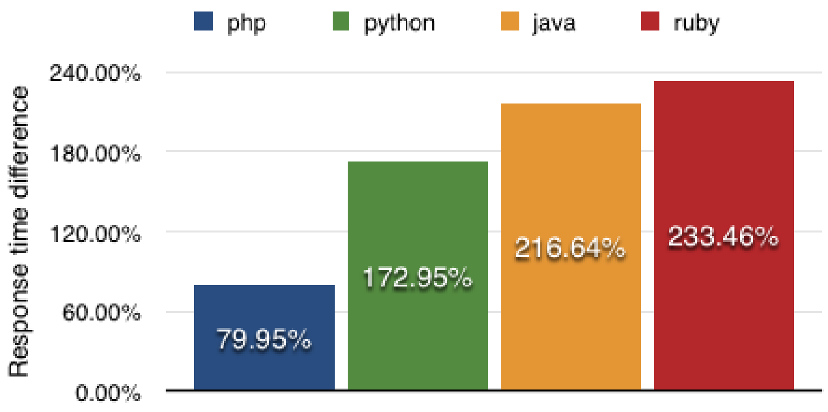


Figure 4.3: Performance chart with experiment test results

This performance chart shows the difference between response times when the ORM component is used or when direct SQL input queries are applied. For example the PHP test results imply that the average response time when an ORM was used (0.19262s) is almost 80% slower then the response time where direct SQL queries were used on the database adapter (0.107040s).

component (Figure 4.3) are 80% to 230% slower than providing direct SQL input queries. The complete set of results and accompanying comparison charts can be found in Annex D.1.

The performance consequences of using an ORM are very relevant for applications that have a higher number of users and demanding work cycles. This can also be the case for products where future scaling is considered. On that account it is often the case that the current project at hand does not include a completely performance centric process, but rather encompasses a handful of parts that have higher execution demands making the inclusion of an ORM more likely. For that reason it is important to rigorously identify potential hotspots in the application workflow that present performance threats.

4.2 Maintainability

Maintainability is the quality attribute of a system that pinpoints the difficulty with which changes can be applied to the application to fix errors, improve on existing components or build new functionality (Somasegar, Guthrie, & Hill, 2009). The importance of this characteristic is a result of the high costs of maintaining a software product which is usually around 40% of the cost of development (Coleman, Ash, Lowther, & Oman, 1994). A system with high maintainability will reduce the tendency of code entropy and reduce the costs of any alterations which undergo in its life cycle.

Oman and Hagemeister (1992) propose a hierarchical structure of maintainability attributes which can be combined to form a singular metric that can be used to measure a software system general maintainability. The process of automation for analysing a system's maintainability quality has proven to be a helpful asset in the decision making process. Thus it can be used for decisions regarding quality of subcomponents, testing resources allocation or even development-purchase trade-off analysis.

The issue of maintainability sits at the core of building an ORM tool, its scope being to provide a familiar setting with the rest of the development environment. Having an object-oriented interface for working with the persistent storage system does not force the developer to switch to another design philosophy when writing the application code and improves its readability. Moreover the implementation of common relational data concepts and operations speed up the development process and its ability to adapt to change.

4.2.1 Measurement criteria

Because measuring software maintenance and its relation to the actual code base of the product is problematic, several index computation techniques have emerged. These techniques are often used interchangeably or simultaneously to form a unique value to measure software maintainability. Among the used methods we can enumerate:

- **source lines of code (SLOC)** - one of the popular methods of measuring the effort put into creating a software product by counting the number of source code text lines. In order to make the metric more relevant for several types of code writing, a distinction was made between counting the physical and the logical (independent statements) lines of code.
- **depth of nesting or inheritance** computation refers to the degree of encapsulation of functional units (functions, classes, modules etc.) within the application code. A highly nested functional unit is more complex and intricate to maintain.
- **module coupling and cohesion** measures the degree of interdepen-

dency between application modules and the degree with which the elements inside each module belong together respectively (Stevens, Myers, & Constantine, 1974). To have high maintainability it is good practice to implement loosely couple modules with high cohesion.

- **cyclomatic complexity** measure - a software metric first introduced by McCabe (1976) to compute the complexity of a computer program's code by analysing its control flow graph. It can be applied on different abstraction levels such as command, function, class, module, and counts the number of linearly independent paths that can be taken through the control flow graph.
- **Halstead complexity** measures, as the name suggests first introduced by Halstead (1977), are a set of software metrics that calculate several characteristics of the implementation code, such as the difficulty of comprehension or creation effort.

External metrics can be also used for measuring the maintainability of a software system but are beyond of the scope of our research since they can only be applied on existing product implementations.

4.2.2 Experiment

To prove the positive trend for maintainability when using an ORM an experiment has been performed. Because our research topic is limited to evaluating the effects of an ORM inside the data management component of a software platform, there are several previously presented metrics that do not fall inside the scope of the experiment. This being said the intention is not to analyse the ORM implementation libraries but rather see the differences between the code used for building basic specific functionality with or without this component. Therefore the cyclomatic complexity, coupling/cohesion and depth of nesting/inheritance are not relevant for our experiment.

This experiment has been conducted in aid of our statement that an ORM component has a positive impact on the platform maintainability quality attribute. Due to the abstraction of the persistent storage system in an object oriented fashion, the coding style is consistent and more compact having an impact on the overall effort of development and code complexity.

With the reasoning of covering the whole array of basic database operations, the maintainability metrics were measured upon the source code benchmarked in our performance experiment. The pseudo-code for the analysed implementation is present in Figure 4.2.

For each of the four platforms that was previously tested, we extracted the benchmarked code snippets and put them side by side for further analysis. To measure the maintainability aspect of the code we used two metrics: the source lines of code measurements and the Halstead complexity computation.

In order to correctly assess the complexity of the source code we have considered that all the SQL queries present in the code are separate logical statements that need independent analysis from the rest of the code. This is due to having a completely different syntax and requiring the developer to think outside the platform environment in which the code is being built. As such, although both the physical and logical source lines of code metrics were measured, we feel that the physical statement measures are not relevant considering that we are analysing only a small part of the software system.

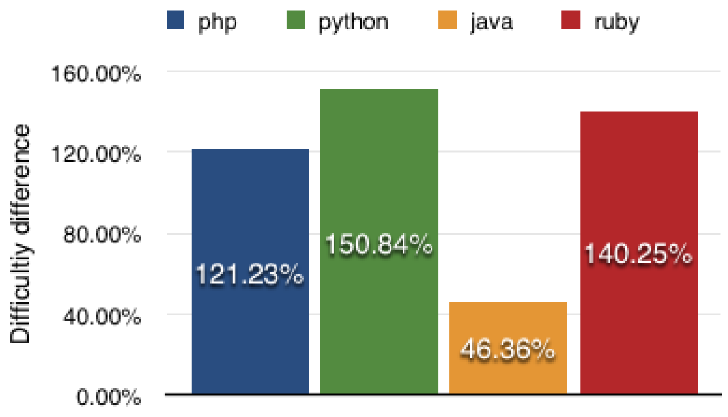


Figure 4.4: Difficulty of code comparison chart

Part of the Halstead complexity, the difficulty of code tries to quantify how easy it is to write the analysed code. This chart shows that according to this metric it is about 50% to 150% less difficult to comprehend the written code when an ORM component is used due to streamlining the development process.

The results confirm that there is a positive impact on the application code maintainability once an ORM is used. This is the case for both metrics used. Therefore, while there is little difference in the physical lines of code between the two implementations, writing data queries without an ORM adds an extra 8% to 65% more logical statements.

All of the Halstead complexity metrics show a significant trend for the code when not using an ORM. It is both more difficult to comprehend and build while attracting almost double the amount of delivered bugs. It is also noticeable that the Java-based ORM implementation (Hibernate) stands out from the others because of its high complexity, reducing the maintainability differences. The complete set of results and accompanying comparison charts can be found in Annex D.2.

4.3 Scalability

Scalability refers to the system ability to react to significant increases in work load without the degradation of performance (Weyuker & Avritzer, 2002). The term is also used for measuring the ability of the product to be easily modified as so it adapts to higher demands. The scalability characteristic of a system is in direct connection with its long-term success (Bondi, 2000). An unscalable system does not necessarily mean that it has poor performance under higher work loads but also that the costs necessary for keeping the product running exceed expectations.

Although it is frequently present as a requirement that a system is scalable, the factors that contribute to this software quality are not as clear as for other attributes. This makes it difficult to identify the necessary steps needed for improving this trait. As such, a set of important performance measurements are established for the project at hand which contribute for estimating the software's ability to adapt to more demanding conditions.

Homer, Sharp, Brader, Narumoto, and Swanson (2014) have created a comprehensive guide on patterns and practices that are usually applied for increasing a system's scalability. Although they have a focus on cloud-hosted applications, most of these practices can be applied for any type of software product where scalability is a concern. Cloud computing is an umbrella term referring to software applications offered as a service on the Internet and the software/hardware setup present in the datacenters which offer those services (Armbrust et al., 2010).

The reason for choosing to focus on cloud-hosted applications is the significant benefits of cloud computing when it comes to scalability. A cloud solution offers total flexibility for assigning the necessary resources for running your application while keeping a transparent overview on the involved costs. What this mean is that it provides purportedly infinite scalability in a seamless fashion as soon as the workload increases with an easily predictable cost.

4.3.1 Measurement criteria

As the term's definition suggests, scalability can measure both the performance degradation under different workloads or the system's ability to be changed to adapt to such situations. This can mean monitoring either the performance metrics under certain environment conditions or calculating other quality metrics (such as maintainability, running cost) under different circumstances.

Among the common techniques to measure this quality attribute is selecting a set of performance metrics that are appropriate for the product and keeping track of how they evolve. As a result the **response time** and **requests throughput** are valid candidates for the observation. After the selection, the scalability measurements imply monitoring the performance metric reaction to changing the size and volume of the workload. An example for this would be simulating increasingly **high number of concurrent requests** or the degradation of performance because of a **big volume of data stored in the persistent storage system**.

It should be noted that the monitoring process for the performance metrics when trying to evaluate a system's scalability often results in bad measurement results along with system failures. The assessments should therefore be built with this in mind, looking also for **request timeouts**, **components freezing** or **resource bottlenecks**.

Because the factors that contribute to the scalability of the system are vague, establishing a measurement criteria for this quality inherits the same shortcoming. Therefore a framework such as the one created by Kaner and Bond (2004) or Schneidewind (1992) is recommended to be applied for the project at hand in order to evaluate if the scalability metric chosen is relevant.

4.3.2 Experiment

As it is directly linked with the performance quality attribute, we infer that the scalability of a platform should be negatively affected by the inclusion of an ORM in the data management component. Therefore an experiment was conducted to sustain this statement.

Because the performance degradation of response times was demonstrated in a previous experiment, we decided to expand it in order to show its effect on scalability. All the test platforms are web frameworks, this why the tool **httperf** (Mosberger & Jin, 1998) was used for gathering various metrics:

number of concurrent requests, number of replies, total test time and number of timeouts.

For each platform the test ran involved sending a set of 1 to 20 concurrent requests and measuring the total time it took the framework to send out the responses. As a potential system requirement we opted for a timeout of 5 seconds so it limits the testing time.

It should be taken into account that the maximum number of concurrent requests used in this experiment was not randomly chosen, but a result of a series of preliminary tests. For that reason it was checked prior to starting the experiment what are the limits for each platform in the number of concurrent requests they can handle without the appearance of any timeouts. While this limit varied from platform to platform we have chosen to select the maximum so that the results would be valuable regardless of the differences in performance.

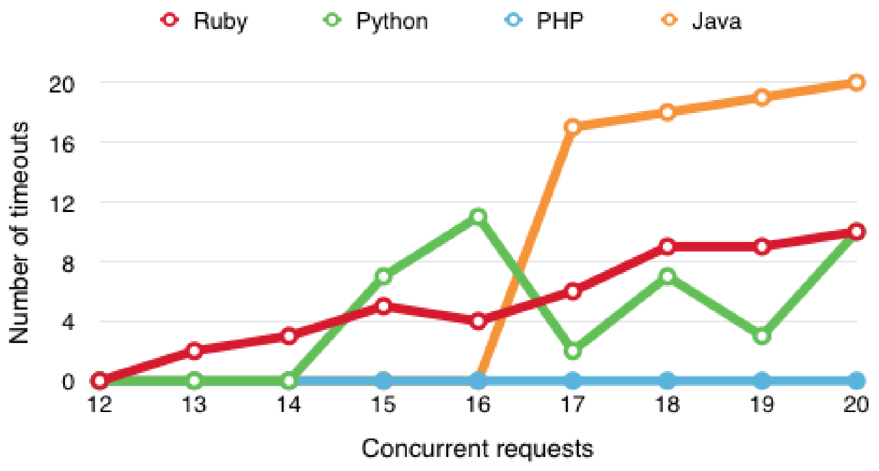


Figure 4.5: Scalability timeouts per number of concurrent requests by platform

This chart shows the number of timeouts that occurred on a platform with an ORM component when a certain number of concurrent requests were made. For example, while the PHP framework had greater performance than the other frameworks with no timeouts occurring during the tests, the Java-based one stopped giving back responses after 17 concurrent requests per second were sent. The test results for all platforms returned no timeouts when the ORM component was not used.

The results are on par with our assumption and show significant response time differences once the workload increases. As such, the platforms correctly responded to all the concurrent requests when no ORM component was used with a slight variation on the total time as a result of the difference in programming language performance. However when an ORM component was utilised the performance started degrading slowly to the point that requests were timed out (Figure 4.5).

An exception to the rule was the Yii Framework (PHP) which showed better performance (no timed out requests) than the other platforms as a result of the lighter ORM implementation but experienced a more abrupt response time degradation towards the end of the test (Figure D.10).

This shows that the additional logic necessary for the bidirectional synchronization of the ORM component imposes scalability repercussions regardless of the size of the library. Reducing the number of features in the ORM does increase its performance but still presents scalability issues on the long term.

The complete set of results can be found in Annex D.3.

Chapter 5

Decision support model

The individuals in charge with making decisions regarding the software architecture of a product are often put in situations where not enough information is available to support their selection. Therefore a decision is adopted based on previous experience and scarce facts available in the documentation of reused components. This alters the objectivity of the decision-making process and may lead to future unintended consequences.

In this chapter we detail the decision making process along with the methods that can be employed to reach a valid decision. Afterwards a suitable method is selected for the research scope and a decision model is created based on the fixed method.

5.1 Decision-making process

The activity of choosing a candidate option out of several requires the decision-maker to have sufficient knowledge in multiple aspects of the problem. In turn the decision objectives need to be identified and a trade-off analysis needs to be done, all while balancing the involved risks (Keeney & Raiffa, 1993).

As a preliminary action to starting the decision process the problem needs to be clearly defined and detailed. In the case of our research the trigger is the little consideration for the data management component architecture when selecting a specific software framework (see Section 1.2). In other words an individual in charge with a product architecture takes decisions regarding the data management component without thinking carefully about its implications, although it directly affects the product. Often misguided by its immediate benefits, a decision-maker may come to a conclusion that the inclusion of an ORM component does not have any repercussions on the quality attributes of the product. As discussed in Section 3.3 and Section 4, the challenges and consequences are not immediately visible and can lead to potential issues.

The problem of using an ORM tool needs to keep in mind a number of requirements such as:

- including support for the persistent storage system that will be used
- facilitating the queries of the storage system necessary to accommodate the application information workflow
- maintaining a sustainable performance level within the predictable levels of usage growth

For that reason the mandatory goal of this process is making a knowledgeable decision that maximizes the maintainability of the product code while making sure that the performance is in acceptable limits and is not affected by any sudden increases in work load.

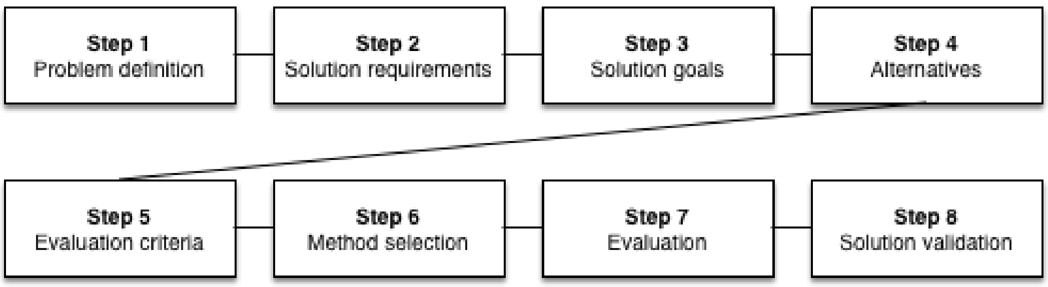


Figure 5.1: General decision making process as described by Baker et al. (2002)

As a result of making a decision, the individual responsible may come to the alternative to exclude the ORM component or include it for partial/complete usage. Although the number of alternatives is limited, the complexity of the underlying problem makes it difficult to make a well informed choice.

Once the problem has been carefully defined along with its objectives, goals and available alternatives, we continue on selecting a decision making method. This method will be used in the creation of a support model that aims to assist the decision maker.

5.2 Decision-making methods

Decision making methods are generally accepted techniques in implementing objective reasoning for the analysis of several available choices (Baker et al., 2002). These procedures are mainly used when the difference between the present alternatives is vague and can spawn difficulties in the selection process.

The process of selecting a technique to be used in our main deliverable is also a result of the analysis of currently available methods in the scientific literature. In this section we present some of the more popular decision making methods and their applicability to our research scope.

5.2.1 Pros and Cons Analysis

The pros and cons analysis is a comparison method often used for simple decisions with a limited number of alternatives. It implies listing the positive and negative aspects in parallel and choosing the alternative which offers the maximum number of advantages and minimum number of drawbacks. Each point on the list needs to be further explained to provide insight and the rationale behind the statement.

Although it offers a simple approach for making a decision without requiring mathematical aptitudes there is a risk of oversimplifying the problem. This is true for complex issues which often do not have only two contradicting sides. An example of this would be considering scaling up a system only via hardware or exclusively with software optimizations. While this is possible, it does not necessarily provide the optimal solution.

Due to the fact that not all points mentioned in a pros and cons analysis have equal importance for the decision maker, it can be misleading to compute a solution just by counting the number and cons and pros. For that reason it is frequent that a weight is assigned to each entry marking it relevant for the situation at hand.

A score is to be computed for each alternative using the following formula:

$$Solution = max(Score_i) \text{ where } Score_i = Total_{pros_i} - Total_{cons_i}$$

This can be translated in written form as: the solution is equal to the alternative with the maximum score where the score is the difference between the total number of pros and the total number of cons. If the analysis contains points with unequal importance the formula stays unchanged with the mention that the total number is replaced with the sum of weights per pros and cons respectively.

5.2.2 Cost-Benefit Analysis (CBA)

The Cost-Benefit Analysis (CBA) is a quantitative method for comparing decision alternative's costs in relation to their long term advantages (Baker et al., 2002). In order for this method to be successfully applied it is mandatory that both the costs and benefits can be clearly quantified using the same unitary system, such as time spent, man hours or monetary currency. The chosen quantification unit needs to have economic relevance to the decision maker. This also constitutes the main difficulty for using this method as it is problematic to assign a pertinent value for the implementation expenses and the rewards that are generated by it (Cellini & Kee, 2010).

The generic process of analysing the costs and benefits as described by Boardman (2010) contains the following steps:

1. Identify decision alternatives
2. Identify stakeholders who are involved or affected by the decision
3. Determine the measurement criteria and unit for all the costs and benefits
4. Estimate the measurements of costs and benefits for the applicable time blocks
5. Convert the concluded measurements into a common unit with economic relevance for the decision-maker
6. Compare the adapted values taking into account the implementation risks and value depreciation over time
7. Apply linear relationships such as maximum profit or lowest cost
8. Select an optimal solution

While for the pros and cons analysis the costs are referred to only if they sit in an extreme: either being too high or very low, the CBA puts them in the centrefold of the method eliminating involuntary subjectivity. Although it brings additional advantages, this decision-making method's validity is entirely dependent on the correctness of the costs and benefits estimation. Therefore any ambiguity present in the measurement phase can produce flawed results.

5.2.3 Analytic Hierarchy Process (AHP)

First introduced by Saaty (1988), the Analytic Hierarchy Process (AHP) is a systematic process of making a decision by organizing the problem's alternatives and decision criteria into a hierarchy and performing pair comparisons. This is done using a nine-point scale (1 to 9) to rank the importance difference between each two criteria and performing multiple mathematical calculations for establishing a numerical score for each alternative. The solution is selected based on the maximum total score.

The series of actions needed to apply the AHP method according to Saaty (1990) are the following:

1. Structure the decision problem by identifying its characteristics, issues, stakeholders and descend from the goals to the criteria, subcriteria and alternatives

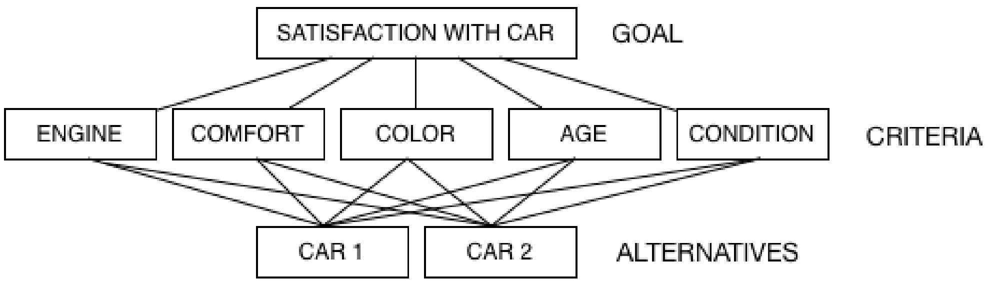


Figure 5.2: Example of structuring the decision problem into a hierarchy

2. Carry out pair comparisons between each of the identified criteria using a level of importance relative scale and compute priority vector
3. Carry out pair comparisons between each of the identified criteria and the alternatives using the same scale and compute priority vector
4. Calculate global priorities for each alternative
5. Select the solution with the maximum global priority score

In the structuring phase of the method it is presumed that the set goal for this decision does not produce alternatives that are significantly different. As an example, in the selection process for a car the options available would not include both small city cars and industrial trucks. Pertaining a similar level of magnitude gives the used comparison relative scale higher accuracy.

The comparison phase consists of comparing pairs of criteria and pairs of alternatives in relation to each criteria.The fundamental relative scale measures the impact one criteria has over the other in reaching the set goal. As such a 1 is assigned if the first criteria is of the same of importance as the other and 9 is assigned if the first criteria is of extreme importance when compared to the other one. After everything has been rated a comparison matrix is built using the formula:

$$A = (a_{ij}) \text{ where } a_{ij} = w_i/w_j$$

So for example if comparing criterion i with criterion j and we would set a value of 9 (criterion i is very important to the end goal when compared to criterion j):

$$a_{ij} = 9 \text{ resulting that } w_i = 9, w_j = 1 \text{ , so } a_{ji} = \frac{1}{9}$$

The priority vector represents the principal eigenvector of the comparison matrix. This needs to be calculated for all the created comparison matrices along with a consistency check. As a final step a global priority vector is computed that lists the concluding scores for each alternative. The maximum between the resulting scores is the optimal solution that would solve the decision-maker's problem.

Although proven to be very useful for many decision makers, the Analytical Hierarchical Process is disputed from a theoretical point of view for the occurrence of rank reversals when subsequently a non-optimal alternative is introduced to the same problem (Ishizaka & Labib, 2009).

5.2.4 Kepner-Tregoe Decision Analysis (K-T)

A decision making method introduced in the 1960s by Kepner and Tregoe (1976), the Kepner-Tregoe Decision Analysis puts an emphasis on weighting the requirements and dividing them into two groups by their importance: musts and wants. The first category acts as an elimination step, while the latter further evaluates the options for better accuracy. As with the other presented methods the solution is selected after a final score is calculated for every alternative.

A potential user of this method needs to go through the following stages:

1. Define the decision statement. It should contain the reasoning behind it, the desired result and the implementation process to reach the end goal.

2. Identify the objectives and divide them into two groups: **musts** and **wants**. The musts consist of critical objectives in the absence of which an alternative would be not be considered anymore. The wants provide a lower level list of requirements that can make a differentiation between the choices.
3. Identify the alternatives. The choices need not to be aligned to the objectives discovered in the previous step as this will be done in the next step. The only requirement is that they fall as a potential solution for the initial decision statement.
4. Filter out alternatives that do not succeed in achieving the musts. Due to the critical importance of the musts to the business strategy these choices are completely eliminated from any latter phase.
5. Assign a weight for the items in the list of wants. This is done on a relative ten point scale (1 to 10) where 1 ranks the entry as being least important in reaching the end goal and 10 ranks it as very important. Note that if one of the entry is critical it should be included on the list of musts and not wants.
6. Score alternatives in relation to each want. The score is calculated by multiplying the weight of the want with a rating assigned to that respective alternative. The rating is on the same ten point scale and measures the degree to which the selected alternative satisfies the objective.
7. Select the first two alternatives by total score and perform a problem analysis. By analysing each alternative against the negative effects that may occur when implementing it, a risk assessment is made. This is done by listing each negative effect along with scoring its probability of occurrence and its significance resulting in a total weighted score.
8. Select solution by comparing their total weighted score and risk assessment scores.

The technique is widely used in business management circles and brings comprehensible structure to the decision process. By filtering out alternatives through critical requirements it reduces the time needed to make the decision, preventing complications caused by an overwhelming number of options. Its validity however is still strongly related to the objectivity of the individual in charge of assigning the objective weights and their scores.

5.2.5 Multi-Attribute Utility Theory Analysis (MAUT)

The Multi-Attribute Utility Theory Analysis (MAUT) refers to a quantitative comparison method which applies utility functions to transform different concept measures in a common scale supporting the user to make a more relevant decision (Baker et al., 2002). The application of MAUT is generally focused on problems that need to examine the tradeoff between multiple objectives. In its early applications it was used for solving decision making problems in the public sector.

Similar to the Kepner-Tregoe Decision Analysis, it employs a systematic approach where the criteria is identified in a hierarchical manner, a weight is assigned for each criterion to represent the importance it has for the case at hand and the alternatives are scored per criterion. The difference appears when calculating the global score for each alternative. This computation is done by an utility function that needs to be accurately be identified by the method user. Although it should rescale any input to values in the 0-1 limits where 0 is the least preferred solution and 1 is the best, this does not mean that the function should be linear such as the one described by K-T. As such, the correctness of the method proposed solution is dependent on the accuracy of the selected utility function.

5.2.6 Custom Tailored Tools

In order to better understand certain constraints or the complex behaviour of a problem, custom-tailored tools may be used or implemented to support the decision process. Although the development of such an artefact requires noteworthy resources, it can produce results with a higher level of relevance. This should only be taken into consideration if the previously described methods are not applicable or provide unsatisfactory results.

5.3 ORM usage support model

Having explained the decision process and the afferent methods that can be employed to support it, we proceed in this chapter to describe the decision model for using an ORM in the data management component of a software platform. For this purpose we have chosen to divide the model into several parts.

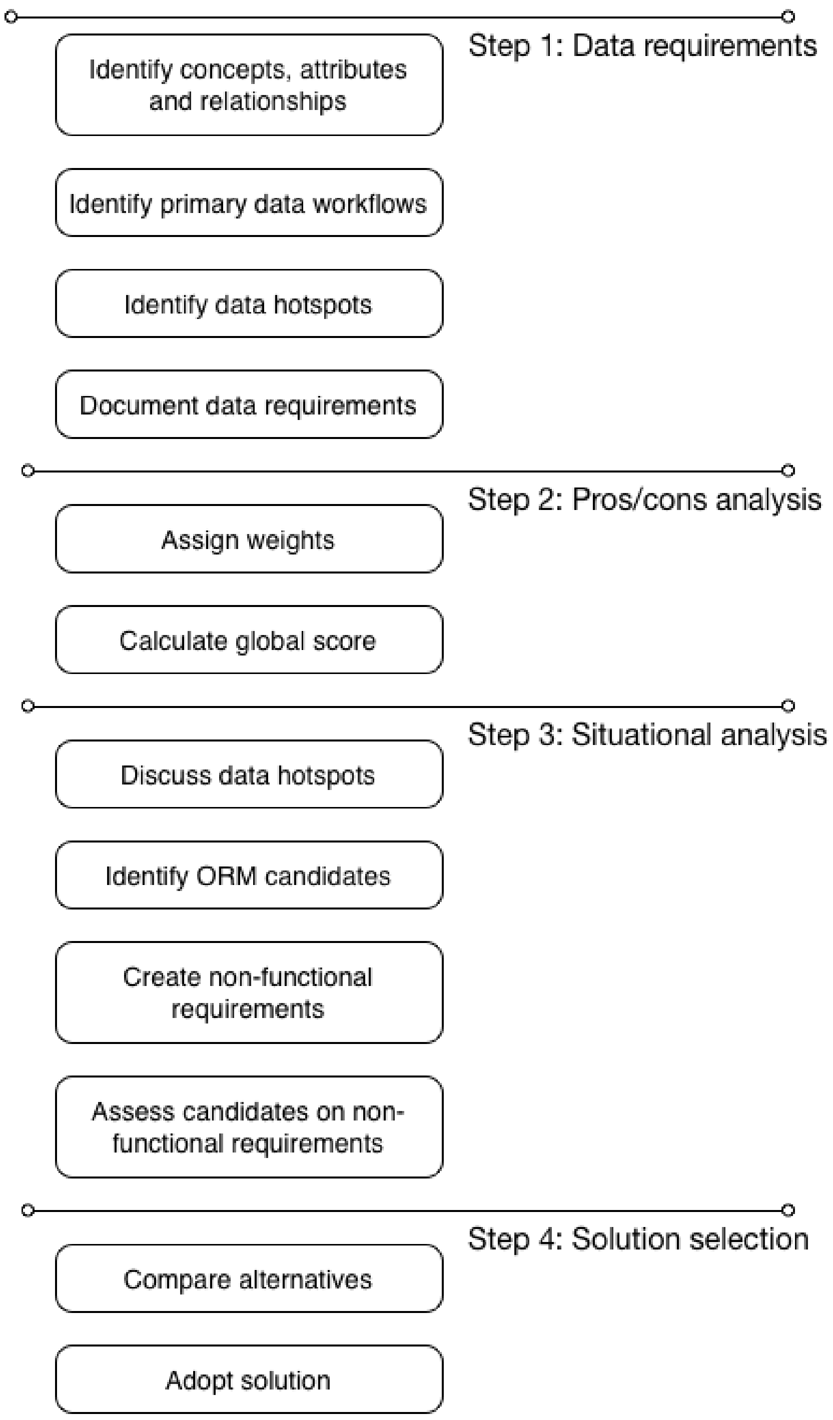

Step 1: Data requirements
Identify concepts, attributes and relationships
Identify primary data workflows
Identify data hotspots
Document data requirements
Step 2: Pros/cons analysis
Assign weights
Calculate global score
Step 3: Situational analysis
Discuss data hotspots
Identify ORM candidates
Create non-functional requirements
Assess candidates on non-functional requirements
Step 4: Solution selection
Compare alternatives
Adopt solution

Figure 5.3: ORM decision support model overview diagram

As a prerequisite the user will need to clearly identify its data requirements for the project at hand. This step is a preparation for the following model stages. It provides the necessary information for weighing or scoring certain statements that are dependent on the project data requirements. It is also important to identify the primary role of the data and its workflow. This can lead to storage division by role which in turn means that the following model steps would need to be repeated for each partition.

The next step would involve providing better understanding on what constitutes an ORM component and list the advantages and disadvantages that it imposes. The user will then assign a weight for each item in the pros and cons list and calculate a global score. This score acts as a filter for extreme cases where the solution is distinguishable and does not require further analysis. It also forces the decision maker to separately think about each item and establish the importance for the project at hand.

If the previously calculated score does not sit in the extremes, it is mandatory that further examination is pursued. This will be done using a straightforward procedure which analyses available ORM implementation candidates in relation to data hotspots that can cause potential issues. By fixating non-functional requirements for performance and scalability the data management component alternatives can be successfully assessed and a relevant solution selection can be done.

In the next subsections we continue on detailing each method step, its outcome and continuation.

5.3.1 Identify data requirements

The identification of relevant data requirements is a very important step for designing an efficient software system. It provides a list of non-functional requirements extracted from the gathered functional requirements by analysing the potential data workflow in the system. Because in itself requirements engineering constitutes a complex process that is outside of the scope of this research we will consider that the complete set of functional requirements has been gathered, discussed upon and agreed with. With this in mind, we proceed on how the process of data requirements extraction is done:

1. **Identify independent concepts, their attributes and their relationship** from analysing each functional requirement. This step can also be translated into the creation of an entity-relationship (ER) model. Although not necessary in some cases (such as working only

with non-relational data), generating an ER model not only brings structure and clarity, it also reveals the system's complexity and can be easily instantiated into a database.

2. **Identify primary data workflows**. The primary data workflows consist of the system simple or complex operations that need to be performed upon one request or trigger. For the purpose of reducing the process complexity and redundancy multiple flows which execute the same logical task may be grouped together. This can be later split for further analysis if necessary. Each workflow may be further detailed by including the entangled concepts along with the CRUD operation performed on it.

3. **Identify data hotspots**. A data hotspot is a workflow or data entity that can cause potential issues on one of the quality attributes of the system (such as performance or scalability) when certain environment conditions change. Rigorously determining these hotspots prevents the user in making decisions that may eventually break the platform due to their high change sensitivity. In order to identify them, one must analyse the previously found concepts and workflows and look for one of the following:

    - Unusual concept attribute type (which can cause difficulties to store/manipulate it)
    - High number of potential records for a specific concept
    - Intricate relationship between two concepts
    - Extreme complexity of a specific workflow (requires significantly more resources to run or involves a lot of concepts)
    - High number of triggers for a specific workflow (caused either by high user traffic or an automatic event)

    This step requires the model user to be able to analyse the previously discovered items with predictable production environment conditions. The validity of this step is dependent on this prediction which is often based on previous experience.

4. **Document data requirements**. Once the previous steps have been successfully completed, a list of data requirements will need to be created. The enumeration will need to include information about the used storage systems (relational or non-relational) and their role, the concepts which are going to be stored and data hotspots concerns together

with potential solutions.

The construction of the list of data requirements is a practical initiative not only for the purpose of this model but also for any architecture decision related to the data management component. Between the great benefits that it provides we can mention: the clear reasoning behind choosing specific storage solutions and the identification of potential issues that the data for the project at hand may impose on the software platform.

It is worth mentioning that the selection of a certain storage solution should be done keeping in mind that it can directly affect the final decision of this model. The main reason for this can be the limited support of the existing ORM candidates for this specific solution or the discrepancy between the purpose of the component and the data that needs to be stored.

In the next subsection we continue on informing the decision maker on what does an ORM component consists of and how can it fit in the candidate architecture of the software platform.

5.3.2 Pros and cons analysis

An **Object Relational Mapping (ORM)** tool is a subcomponent of the data management module of a software platform that maps data between incompatible type systems in object oriented programming languages by creating a virtual database that can be used from within the language. It is therefore creating a bridge between the implemented storage solutions and the object oriented language by creating a persistence layer. This abstraction layer facilitates the work with the storage solution by creating a streamlined development environment.

It should be noted that although it provides significant abstractions, its purpose is not to completely hide the underlying database technology, but to increase productivity and efficiency by simplifying the implementation of best practices on common data operations and workflows. As such the software platform extenders will need to possess sufficient knowledge about the technology behind the used storage systems. To successfully use such a component, it is mandatory that the developers understand how every part of the ORM works and how the used abstractions translate to other query languages.

In order to better assess the applicability of such a component in the project's architecture a list of the main advantages and disadvantages has been constructed below. Because some cases can clearly push the balance in favour of a certain alternative, we ask the decision maker to assign a weight on each item on the list on a five point relative scale, where 5 represents an extreme level of importance and 1 represents a low level of importance or the project at hand.

Pros:

- Database agnostic architecture
- High-level concept and relationship modelling
- Reduces development effort
- Reduces maintenance costs

Cons:

- Require additional learning time to become proficient in
- Decrease in performance for data operations
- Additional optimization may be required for scaling
- Cannot accommodate architectures with high scalability

The weight assignation needs to be done keeping in mind all the previously discovered data requirements. For example, it is obvious that an application with a lot of data hotspots will show high interest in keeping a good performance if it has the potential to scale in the future, while for smaller projects the development effort or costs may have a more critical stance.

A global score can then be calculated from the difference between the total assigned weights for the pros and cons. This serves as a guideline for the decision maker as to how the addition of an ORM component aligns with its data requirements. Although not definitive (a situational analysis still being needed), the score result can be interpreted as:

Score (s)	**Interpretation**
$s >= 8$	Highly probable that an ORM will bring more benefits than disadvantages.
$s <= -8$	Highly probable that an ORM will bring more disadvantages than benefits.
$-8 < s < 8$	Requires situational analysis for clarification.

Table 5.1: Pros and cons analysis score interpretation

On the completion of this step the model user should have a clear understanding on what is an ORM component, what implementation benefits and drawbacks it imposes and how do these relate to its earlier determined data requirements. In the next subsection a more advanced analysis is made on how does an ORM component affect the software platform quality attributes.

5.3.3 Situational analysis

In order to better assess the applicability of this data management component to the current architecture a customized version of the Architecture Tradeoff Analysis Method (ATAM) (Kazman et al., 1998) will be used. However, because of the technical aspect of the problem, in order to analyse the tradeoffs of the architecture only the architecture stakeholders will be involved.

The addition of an ORM component at the architecture level of a software platform comes with significant effects on its quality attributes. As such, additionally to having development value through the streamlined language, it has a positive impact on the overall system maintainability. This comes with the cost of added implementation logic to accommodate the difference between the two incompatible systems (the object oriented software platform and the storage solution), in result having a negative impact on performance and scalability. If the decision maker is not completely aware of these circumstances the following risks may appear:

- Inefficient data queries due to extreme convenience.
- Inefficient data queries due to language standardization logic.
- Incapacity of handling current or future workload due to data performance issues.
- In case of a decision to remove the ORM, necessity to do a costly application rewrite because it sits on the base of all the information the application uses.

Having known the purported risks, the decision maker along with any architecture stakeholders that are relevant to the task will need to follow the subsequent analysis procedure:

1. **Discuss data hotspots solutions.** Due to the high risks that they impose on the final product each item found on the data hotspots list needs to be further analysed and discussed. Potential solutions to reduce the quality attribute degradation will need to be discovered before

any decision regarding the data management component is made.

2. **Identify viable ORM implementation candidate.** When creating a software product the underlying technology used (such as programming language or low level framework) is rarely influenced by preliminary study but rather by the availability of developer resources or subjective preferences. This means that the available ORM implementations may vary both in number or complexity depending on which technology is being used. This step involves identifying viable candidates and creating a short list to be used later for the assessment. One additional important trait that also needs to be considered is the ability to reduce development effort.

3. **Create non-functional requirements that establish performance and scalability caps.** In order to accurately evaluate how the ORM component will accommodate the current architecture a set of non-functional requirements need to be set. Real-life environment conditions will need to be considered when fixating the specifications along with a pragmatic view on how the application will scale in the near to distant future.

4. **Assess data management component against non-functional requirements.** Having established a set of requirements for the performance and scalability of the application, we can then pursue in testing the ORM candidates. Experiments should mimic the workflow of the data hotspots and provide a measurement on how well each ORM candidate performs.

The experiments performed for measuring the candidate ORMs against the set non-functional requirements would mitigate any risks appearing from workload changes, with a focus on the data hotspots that are more likely to cause issues. This should be done by identifying workflows with peak data operations present in the application. By setting up a test environment for measuring requests response time, timeouts or throughput, the dissimilarity between the implementations can be discovered. The differences could be the result of additional logic or supplementary optimization.

On the completion of this step the model user should have a clear understanding on where an ORM could have a potential negative effect on the quality attributes of the platform and what solutions does he have in order to fix them. The next subsection presents the procedure to select a viable alternative.

5.3.4 Solution selection

The final step of the method combines the information gathered in all the previous phases. As such, the user should be aware of its exact data requirements, what is an ORM component along with its advantages/disadvantages and the potential issues that may arise from his main data workflows. Also having tested the applicability of each candidate implementation in relation to the fixed non-functional requirements, the method user can decide if selecting an existing solution is relevant to its situation.

The solution should be selected keeping in mind the following:

- The data requirements should be completely fulfilled by the solution or adapted so that they still encompass the product objectives and satisfy the requirements.
- The potential risks caused by future product workload growth should be encompassed in the non-functional requirements.
- The performance and scalability traits of the solution are in the limits fixed by the non-functional requirements.
- For alternatives which meet the above conditions the decrease in development effort should be the decisive factor.

Having clarified all of the above, the method user can make a knowledgeable decision on whether an ORM data management component is a relevant addition to the software framework architecture and which alternative brings the most advantages to the project at hand.

5.3.5 Example

For the purpose of providing an example of how to use the previously described model we will consider the following case:

> An online newspaper application needs to be built using a modern software platform. It currently has a traffic potential of an average of 10000 unique visitors per day. It will have an average of 300 articles published at any time. Given the functional requirements presented below, is an ORM component a valid addition to the platform software architecture?

#	Functional requirement
1	The newspaper is accessed through the Internet via a public website where all the following functionality is available.
2	The application has a private dashboard where **administrators**, **editors** and **moderators** can login with an email and password.
3	**Administrators** can manipulate (CRUD) all system **users**.
4	**Editors** can manipulate (CRUD) and list/unlist **article categories**.
5	**Article categories** order is determined by a weight established through a priority attribute.
6	**Editors** can manipulate (CRUD) and publish/unpublish **articles** (rich-text format).
7	**Articles** order is determined by a weight established through a priority attribute.
8	Website visitors can access published **articles** while browsing through **article categories** or through the front page.
9	Website visitors can create a **user** account and **comment** on each **article**.
10	**A moderator** will validate each **comment** before it is made public.

Table 5.2: Example functional requirements

Step 1: Identify data requirements

Concept	Attributes	Relationships
Article	title (string) content (text) published (boolean) priority (integer)	has many Comments belongs to User belongs to Article Category
Category	name (string) listed (boolean) priority (integer)	has many Articles
Comment	content (text) moderated (boolean)	belongs to User belongs to Article
Role	name (string)	has many Users
User	email (string) password (string)	has many Articles has many Comments belongs to Role

Table 5.3: Example concepts, attributes and relationships identification

Data workflow	**Estimated peak requests/s**
User login	10
User (administrator) manipulates other users	1
User (editor) manipulates a Category	2
User (editor) manipulates an Article	5
Visitor retrieves an Article	27
User comments on article	9
Moderator validates Comment	2

Table 5.4: Example primary data workflows identification

By looking on the functional requirements and the previously identified concepts and data workflows we infer the main data hotspots for this application are: the article data entity and the article retrieval workflow. Having identified the main data hotspots we then pursue in documenting the main highlights of our findings into a data requirements list:

- The application requires a relational storage system to persist all the necessary data. Due to its high popularity, an SQL based solution would be preferred.
- The Article data entity contains rich-text content that may contain media (such as audio, images or video), storing this concept may require additional analysis.
- The high traffic for article retrieval may need optimization work for this particular workflow in order to provide acceptable performance.

Step 2: Pro and cons analysis

Pros	**W**	**Cons**	**W**
Database agnostic architecture	1	Learning time proficiency	2
Concept/relationship modelling	4	Performance decrease	4
Reduces development effort	4	Additional optimization for scaling	2
Reduces maintenance costs	5	Cannot accommodate high scalability architectures	1
Total pros	**14**	**Total cons**	**9**

Table 5.5: Example pros and cons analysis

Step 3: Situational analysis

Having a pros and cons analysis global score of 5, means that the addition of an ORM component has a positive trend towards bringing more advantages than drawbacks to the platform architecture. By looking at the previous steps we infer that the main concern is the performance and scalability drawbacks caused by the data hotspots. In order to fix that, the following solutions were proposed:

- To store the content of the articles, they are going to be modelled into other data entities or linked through to a local file storage system or a Content Delivery Network (CDN).
- Article retrieval can have its performance drastically increased by caching all records so that a database query is not performed.

ORM implementation candidates:

- ORM1 (included in used software framework) provides tight integration with the other framework components with database migration management, event callbacks, concept and relationship modelling, concept validation and inheritance
- ORM2 provides lightweight implementation of concept and relationship modelling
- ORM3 provides concept and relationship modelling, concept validation and inheritance, limited SQL database support

Non-functional requirements:

1. System should be able to handle at least 40 concurrent requests without response timeouts.
2. The response time for an article retrieval should be less than 2 seconds, regardless of current workload.

Candidate	**Avg. resp. time (s)**	**Max concurrent req.**
ORM1	1.278	40
ORM2	0.918	67
ORM3	1.422	35

Table 5.6: Example assessment against non-functional requirements

Step 4: Solution selection

By looking at the pros and cons analysis score and the situational analysis done in the previous section we conclude that the newspaper application can benefit from the inclusion of an ORM tool in the data management component of its architecture. Because it provides better integration with the existing framework components and additional features that would benefit the product maintainability, the ORM1 alternative was selected.

Chapter 6

Evaluation

In order to assess the created decision model we have conducted interviews with experts in the domain. All of their suggestions and feedback was kept into account when creating the conclusive version of the support model. The interviews were conducted with a total of 3 people with experience in the field ranging from 3 to 8 years. The highlights of these discussions are incorporated in this chapter.

6.1 Method

The evaluation was done by presenting the support model to each interviewee, followed by a conversation where the feedback was recorded. The interview was done in a semi structured fashion acting in according with the following fixed topics and questions that were applied for the whole model as well as for each step:

- **Comprehension**

  What was your understanding that this model would do?

  Were you at any time unsure of the action that needs to be performed for a certain step?

- **Applicability**

  Would you use this decision model when having to decide to incorporate an ORM in your architecture?

  Do you think it is relevant to apply this step?

- **Suggestions**

  Is there any other additional action that you would do when making this decision?

  Do you find a step in the model to be unnecessary?

After the feedback was received the decision support model was reviewed and updated accordingly.

6.2 Feedback

The model has predominantly received positive feedback. All of the interviewed individuals agreed with the fact that the process follows the logical series of actions needed in order to achieve the end objective: make a knowledgeable decision for the inclusion of an ORM in the platform architecture. The language used was clear and there was no confusion in what is needed to be done when performing a certain step.

One of the experts argued that choosing the underlying storage technology was introduced in the decision model and its example but that it was not easy to perceive the fact that this decision in itself can have a big effect on the final solution. It was agreed upon the aspect that the two are not mutually exclusive and need to be done with great consideration for each other. A better explanation on this topic was added to the first section.

For the problem of scalability it was also suggested that it would be a problem if it wasn't already considered for the part of the architecture outside of the scope of the data management component. This was further described in the appropriate step of the situational analysis.

Although should be left as an open subject, the way the ORM candidates are assessed against the non-functional requirements was highlighted as being a bit confusing for somebody with less experience. As such, a suggestion of what should these experiments should measure and the reasoning behind it were added.

Another issue raised by one of the interviewees was that the development effort and maintainability aspect of the problem was not given sufficient attention. The model was constructed with the objective of mitigating any potential risks that would cause the project to fail. We have found during the case study interviews that the benefits that cause the positive effect on maintainability are very often influenced by the subjectivity of the decision maker. The opinions on this are either that the component is significantly limiting the ability to communicate with the storage solution or that it makes the development process extremely efficient. Because of the bipolar views, the development effort was chosen as a criteria of lower importance.

The applicability of the model was a highly debated subject. It was confirmed during the interviews that the support model is capable of aiding an individual into making a more knowledgeable decision. However the level of experience in the domain may push you to think that you can reconstruct the steps from previous occurrences. Due to its clear structure some of the experts stated that they would use it if a more compressed version was available without the extended explanations.

6.3 Support model checklist

Because it was suggested that the decision support model should have a minified version that can be easily applied, we created a checklist that goes through all the steps present in the model. It is recommended that you use this list only if you have applied the full decision model on previous occasions. The full support model can always be consulted when having unclarities.

☑ Identify concepts, attributes and relationships

☑ Identify data workflows

☑ Identify data entity/workflow that can cause potential issues (hotspot)

☑ Establish potential used storage technologies and their role

☑ ORM component purpose is relevant to project data requirements

☑ Find data hotspot solutions

☑ Identify ORM implementation candidates

☑ Predict future application workload (how it will scale)

☑ Establish performance and scalability caps (non-functional requirements)

☑ Create experiments to measure how each ORM candidate performs

☑ Compare experiment results

☑ Select solution

Chapter 7

Discussion

There were no major barriers encountered in carrying out the research project, despite the fact that some changes occurred in the process in order to accommodate different issues. The main problem was caused by the scarce availability of information directly linked to the studied topic. Although from a technical standpoint sufficient knowledge was found in the literature, there was little business-oriented work that pinpointed the implications for architectural decisions.

Moreover the case study confirmed once again that the selection of underlying technologies such as the software framework is rarely influenced by its software architecture, as also described by Jansen (2013). As an example, the used technology for the products most of our interviewees were building was a result of the decision made by looking at the availability of developer skills and resources or a strict personal preference. Because of this we can infer that subjectivity is still a strong characteristic when making architectural decisions.

Choosing the data management component will inevitably be very dependent on the underlying technology due to the highly volatile characteristics that separates them. For this reason the popularity, complexity or usability of the used technology can counter balance a decision in a certain direction. For example, a product that is created with a lower-level framework will naturally not have a wide range of available ORMs and furthermore the creation of such a tool from scratch can prove to be a high complexity project on its own, which in the end does not bring sufficient benefits.

An Object-Relational Mapping tool is an abstraction that adds significant logic to the application framework which can affect the performance or scalability of the product. The performance of the component is in direct coordination with the performance of the underlying programming language, therefore it is evident that the final solution provided by using the support model may differ when trying to build the same project with distinct technologies.

It should be taken into account that the created support model retains a general note in order to cover all the cases mentioned above while still providing aid for the decision maker with its clear structure and applicability. Moreover the deliverable generated by the first step of the model can be used for other purposes and can also be of added value to the software architecture documentation.

The situational analysis of the support model addresses the risks ensued by the data management component in general and a potential ORM component. By discussing the issues and evaluating them against actual implementations, we eliminate any future unpredictable shortcomings and raise the awareness of the component constraints.

As a result of its technical nature, the applicability of the support model requires the decision maker to have extensive knowledge in storage solutions and software architectures in general. The method steps involve the close collaboration with the development team and the individuals in charge with gathering the end product requirements. This ensures that the formulated method requirements are based on realistic environment conditions and the architecture will have a valid implementation instance.

7.1 Limitations

To make it more relevant for enterprise applications we have excluded the comprehensive analysis of NoSQL databases support for the data management component. Therefore all the tests included in this research are mainly done with SQL-based storage solutions.

Due to the time constraints of this project, we were forced to consider only a limited number of software platforms to analyse. This may affect the objectivity with which our candidate data management architecture is built.

Although some literature addresses the topic of data management in a software framework, we felt that there is a lack of prior research studies on this topic with more recent data. As such some of the patterns discovered might be considered outdated or not relevant to modern day software platforms.

The problem of mapping two incompatible systems is very complex, for that reason the measurements done for estimating the effect on the platform is dependent on a high number of factors. We acknowledge that some of these factors may be critical in lowering the impact on the platform when using an ORM. The decision to not try and optimize the testing environment in any way was made in order to show that additional steps are needed when including the ORM component. The quantification of the magnitude of these steps are outside of the scope of this research project.

Chapter 8

Conclusion

To conclude our research we will review the trigger, objectives and outcomes of our work in this chapter. The selection of a software platform is closely linked to the degree with which it satisfies the required functionality, however it is often the case that it does incorporate the needed features. Due to its importance and low level locality in the product architecture, the data management component can cause unwanted issues in advanced stages of development. By making a knowledgeable decision in an early phase of the product development, the software extender can avoid future problems caused by this component.

The objective of this study is to create a decision model that comes in support for determining whether an Object Relational Mapping tool is suitable for the data management component of the used software platform.

In aid of our end objective the following research questions were formulated and answered:

> *RQ1: What are the challenges that come from applying the Object-Relational Mapping pattern?*
>
> The Object-Relational Mapping pattern's main objective can be considered a challenge on its own by trying to map data between incompatible type systems. The difficulties that arise from this are known under the name of object-relational impedance mismatch and come from the fact that the object and relational approaches are constructed on different foundations (i.e. software engineering vs mathematical set theory). Therefore it raises concerns on how to map the dissimilarities in structure, identity, processing and own-

ership.

RQ2: What are the consequences for using an ORM tool in the data management component of a software platform?

Although the immediate benefits that it provides are easily recognizable, the usage of an ORM in the data management component also has consequences. Having a database agnostic architecture that allows easy to use high-level concept modelling comes at the cost of adding a layer of complex logic. This causes a decrease in performance for data operations and the necessity for additional optimization for scaling. As a further matter, because of the strong abstraction being done, supplementary learning time is needed for proficiency and may cause convenience to the point of ignorance of the underlying storage technology.

RQ3: Which quality attributes are influenced by the usage of an ORM tool?

On account of having an extra layer of logic in the architecture the ORM will inevitably bring an adverse effect on the product's performance and scalability. The experiments performed in our research show that actions dependent on heavy storage queries will show a 2 to 3 times decrease in response time. The benefits of the component are seen when looking at how it influences maintainability. Depending on the complexity of the ORM implementation it can significantly lower the number of lines of code needed to develop storage related tasks, with it also the effort for comprehension, time to program and number of delivered bugs.

RQ: What are the criteria based on which you can make a decision for using an ORM tool in a software platform?

The points of reference to which a decision to include an ORM in the architecture of a software framework is made are:

- Clear identification of the data requirements of the project at hand along with potential data hotspots
- The relevance of the component objective for the data requirements
- Availability of implementation alternatives
- Alternative assessment against non-functional requirements set on real-world environment conditions

Main deliverable: Decision support model for including an Object-Relational Mapping component in the architecture of the used software framework.

We find that the created support model can come of great use to individuals in charge of a software product architecture, as it provides a structured process with which a knowledgeable decision is made preventing future undesirable complications. The deliverable is a representation built with the knowledge gathered from the literature and industry experts, having real applicability for modern-day software frameworks.

8.1 Further research

This systematic investigation was revolved around a software architecture problem of current application frameworks. Because of the broad nature of the topic, this research can be further extended or studied upon spawning work in:

- *The development of a generic data management component decision model.*

  A direct extension of this research study would involve the generalisation of the decision model to incorporate any type of data management components. The benefits of this would include among others: increased awareness of existing solutions, quicker decision process and a broad scope that can accommodate more circumstances.

- *A knowledgeable selection of a software framework.*

  As mentioned in the research introduction, software extenders do not follow a clear process of selecting a software platform with which to build their products. Their criteria is often mislead by their own subjectivity or is strictly dependent on the personal preference of the decision maker. A coherent method that would aid the selection process is required to consider the architecture behind each alternative.

- *The main reasons for the appearance of architectural breaches.*

  Software application frameworks often give a long list of features and functionality out of the box, giving software extenders the ability to deliver stable and complex applications in a convenient time span. However, developers sometime choose to ignore the conventions imposed by the platform in order to get access to functionality which is otherwise inaccessible. One of the reasons for this is presented in this research (not well informed architectural decision on the data management component). Further effort could be done in discovering any other justification that these architectural breaches appear.

References

Ambler, S. W. (2000). Mapping objects to relational databases: What you need to know and why. *developerWorks, downloaded from: www-4. ibm. com/software/developer/library/mapping-to-rdb. index. html*, 1–9.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., ... others (2010). A view of cloud computing. *Communications of the ACM*, *53*(4), 50–58.

Baker, D., Bridges, D., Hunter, R., Johnson, G., Krupa, J., Murphy, J., & Sorenson, K. (2002). Guidebook to decision-making methods. *Retrieved from Department of Energy, USA: http://emiweb. inel. gov-/Nissmg/Guidebook_2002. pdf*.

Barbacci, M., Klein, M. H., Longstaff, T. A., & Weinstock, C. B. (1995). *Quality attributes.* (Tech. Rep.). DTIC Document.

Bass, L., Clements, P., & Kazman, R. (2012). *Software architecture in practice.* Pearson Education. Retrieved from `http://books.google.nl/books?id=-II73rBDXCYC`

Boardman, A. (2010). *Cost-benefit analysis: Concepts and practice.* Prentice Hall. Retrieved from `http://books.google.nl/books?id=ZgNiuQAACAAJ`

Bondi, A. B. (2000). Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on software and performance* (pp. 195–203).

Bourquin, F., & Keller, R. K. (2007). High-impact refactoring based on architecture violations. In *Software maintenance and reengineering, 2007. csmr'07. 11th european conference on* (pp. 149–158).

Cellini, S. R., & Kee, J. E. (2010). Cost-effectiveness and cost-benefit analysis. *Handbook of practical program evaluation*, 493.

Clements, P., Kazman, R., & Klein, M. (2003). *Evaluating software architectures.* .

Coleman, D., Ash, D., Lowther, B., & Oman, P. (1994). Using metrics to evaluate software system maintainability. *Computer*, *27*(8), 44–49.

Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., & Mockus, A. (2001). Does code decay? assessing the evidence from change management data. *Software Engineering, IEEE Transactions on*, *27*(1), 1–12.

Evans, D. S., Hagiu, A., & Schmalensee, R. (2006, September). Invisible Engines: How Software Platforms Drive Innovation and Transform Industries. *The MIT Press*.

Fayad, M., & Schmidt, D. C. (1997). Object-oriented application frameworks. *Communications of the ACM*, *40*(10), 32–38.

Fichman, R. G. (2004). Real options and it platform adoption: Implications for theory and practice. *Information Systems Research*, *15*(2), 132–154.

Fowler, M. (2002). *Patterns of enterprise application architecture.* Addison-Wesley Longman Publishing Co., Inc.

Fowler, M. (2003). Who needs an architect? *IEEE Software*, *20*(5), 11–13.

Fowler, M. (2012). *OrmHate.* `http://martinfowler.com/bliki/OrmHate.html`. (Online; accessed 15-August-2014)

Frakes, W. B., & Kang, K. (2005). Software reuse research: Status and future. *Software Engineering, IEEE Transactions on*, *31*(7), 529–536.

Garlan, D., & Shaw, M. (1994). An introduction to software architecture.

Glaser, B. G., & Strauss, A. L. (1967). *The discovery of grounded theory.* Chicago: Aldine Publishing.

Griss, M. L. (1997). Software reuse architecture, process, and organization for business success. In *Computer systems and software engineering, 1997., proceedings of the eighth israeli conference on* (pp. 86–89).

Halstead, M. H. (1977). Elements of software science.

Hasselbring, W., & Reussner, R. (2006). Toward trustworthy software systems. *Computer*, *39*(4), 91–92.

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS quarterly*, *28*(1), 75–105.

Homer, A., Sharp, J., Brader, L., Narumoto, M., & Swanson, T. (2014). Cloud design patterns: Prescriptive architecture guidance for cloud applications.

Hunt, A., & Thomas, D. (2000). *The pragmatic programmer: from journeyman to master.* Addison-Wesley Professional.

Ireland, C., Bowers, D., Newton, M., & Waugh, K. (2009a). A classification of object-relational impedance mismatch. In *Advances in databases, knowledge, and data applications, 2009. dbkda'09. first international conference on* (pp. 36–43).

Ireland, C., Bowers, D., Newton, M., & Waugh, K. (2009b). Understanding object-relational mapping: A framework based approach. *International Journal on Advances in software*, *2*(2 and 3), 202–216.

Ishizaka, A., & Labib, A. (2009). Analytic hierarchy process and expert choice: Benefits and limitations. *OR Insight*, *22*(4), 201–220.

Jansen, S. (2013). How quality attributes of software platform architectures influence software ecosystems. In *Proceedings of the 2013 international workshop on ecosystem architectures* (pp. 6–10).

Juneau, J. (2013). Object-relational mapping. In *Java ee 7 recipes* (pp. 369–408). Springer.

Kabbedijk, J., Salfischberger, T., & Jansen, S. (2013). Comparing two architectural patterns for dynamically adapting functionality in online software products. In *Patterns 2013, the fifth international conferences on pervasive patterns and applications* (pp. 20–25).

Kan, S. H. (2002). *Metrics and models in software quality engineering.* Addison-Wesley Longman Publishing Co., Inc.

Kaner, C., & Bond, W. P. (2004). Software engineering metrics: What do they measure and how do we know? *methodology*, *8*, 6.

Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., & Carriere, J. (1998). The architecture tradeoff analysis method. In *Engineering of complex computer systems, 1998. iceccs'98. proceedings. fourth ieee international conference on* (pp. 68–78).

Keeney, R. L., & Raiffa, H. (1993). *Decisions with multiple objectives: preferences and value trade-offs.* Cambridge university press.

Keith, M., & Schnicariol, M. (2010). Object-relational mapping. In *Pro jpa 2* (pp. 69–106). Springer.

Kepner, C. H., & Tregoe, B. B. (1976). The rational manager; a systematic approach to problem solving and decision making.

Krasner, G. E., Pope, S. T., et al. (1988). A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, *1*(3), 26–49.

Martin, M., & Martin, R. C. (2006). *Agile principles, patterns, and practices in c#.* Pearson Education.

McCabe, T. J. (1976). A complexity measure. *Software Engineering, IEEE Transactions on*(4), 308–320.

Mosberger, D., & Jin, T. (1998). httperfa tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, *26*(3), 31–37.

Neward, T. (2006). The vietnam of computer science. *The Blog Ride, Ted Newards Technical Blog.*

Oman, P., & Hagemeister, J. (1992). Metrics for assessing a software system's maintainability. In *Software maintenance, 1992. proceerdings., conference on* (pp. 337–344).

Perry, D. E., & Wolf, A. L. (1992). Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, *17*(4), 40–52.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. E., et al. (1991). *Object-oriented modeling and design* (Vol. 199) (No. 1). Prentice-hall Englewood Cliffs, NJ.

Saaty, T. L. (1988). *What is the analytic hierarchy process?* Springer.

Saaty, T. L. (1990). How to make a decision: the analytic hierarchy process. *European journal of operational research*, *48*(1), 9–26.

Sam-Bodden, B., & Judd, C. (2004). Object-relational mapping. In *Enterprise java development on a budget: Leveraging java open source technologies* (pp. 333–405). Springer.

Schneidewind, N. F. (1992). Methodology for validating software metrics. *Software Engineering, IEEE Transactions on*, *18*(5), 410–422.

Shaw, M., & Garlan, D. (1996). *Software architecture: perspectives on an emerging discipline* (Vol. 1). Prentice Hall Englewood Cliffs.

Somasegar, S., Guthrie, S., & Hill, D. (2009). Microsoft application architecture guide. *Microsoft,*.

Stevens, W. P., Myers, G. J., & Constantine, L. L. (1974). Structured design. *IBM Systems Journal*, *13*(2), 115–139.

Tellis, W. (1997). Application of a case study methodology. *The qualitative report*, *3*(3), 1–17.

van de Weerd, I., & Brinkkemper, S. (2008). Meta-modeling for situational analysis and design methods. *Handbook of research on modern systems analysis and design technologies and applications*, *35*.

Weyuker, E. J., & Avritzer, A. (2002). A metric to predict software scalability. In *Software metrics, 2002. proceedings. eighth ieee symposium on* (pp. 152–158).

Williams, L. G., & Smith, C. U. (1998). Performance evaluation of software architectures. In *Proceedings of the 1st international workshop on software and performance* (pp. 164–177).

Yin, R. K. (2009). *Case study research: Design and methods* (Vol. 5). Sage.

Appendix A

PDD Activity and Deliverable tables

Activity	**Sub-activity**	**Description**
Determine context	Analyse documentation	Perform an analysis of the documentation of several ecosystem-based software platforms, pinpointing the main functionality and design characteristics of the data management component and the ORM tool
	Perform literature review	Extend on the concepts already discovered by performing a literature study and select context relevant resources
	Perform case study	Conduct interviews at several software businesses and support the findings with data gathered from development Q&A websites
Platform quality attributes impact	Identify affected quality attributes	Determine which platform quality attributes are affected by the ORM tool component
	Establish impact measurement criteria	Identify how each affected quality attribute can be measured so that the impact can be compared or studied
	Perform quality attributes measurements	Apply the previously established measurements on each affected quality attribute

Decision support model development	Select decision making method	Determine which decision making method is suitable for creating a valuable decision support model
	Build decision model	Create the actual research artefact based upon the selected decision making method and the previously gathered data
Decision support model validation	Perform expert validation interviews	Get feedback on the created decision model from the experts by conducting interviews
	Refine model based on expert feedback	Improve and clarify the decision support model based on the received feedback

Table A.1: PDD activity table

Concept	**Description**
DOCUMENTATION ANALYSIS	A deliverable comprised of the main designs and functionality of various software platform data management architectures
RELATED LITERATURE	Extended literature review on data management architecture, the ORM pattern, its challenges and consequences
CASE STUDY REPORT	Deliverable incorporating findings resulted from the company interviews and Q&A website queries
RESEARCH CONTEXT	Combined and processed gathered information from the documentation analysis, literature review and case study
PLATFORM QUALITY ATTRIBUTES	A list of all the quality attributes of ecosystem-based software platforms
AFFECTED PLATFORM QUALITY ATTRIBUTES	Short list of the quality attributes of ecosystem-based software platforms based on whether they are influenced by the ORM data management component
QUALITY ATTRIBUTES MEASUREMENT CRITERIA	The guidelines which apply in order to measure the affected quality attributes

QUALITY ATTRIBUTES IMPACT	A set of measurements that quantify the impact the ORM tool has upon the quality attributes of the platform or product
SELECTED DECISION MAKING METHOD	The specific decision making method used in building of this research artefact
DECISION SUPPORT MODEL	A decision support model document that offers a comprehensive description of the ORM data management pattern, its challenges, consequences of usage, effect on the platform quality attributes in relation with the data requirements of the project at hand
EXPERT VALIDATION FEEDBACK	Domain expert feedback about the created DECISION SUPPORT MODEL resulting from a set of interviews
REFINED DECISION SUPPORT MODEL	An improved variant of the DECISION SUPPORT MODEL that takes into account the EXPERT VALIDATION FEEDBACK

Table A.2: PDD concept table

Appendix B

Semi-structured interview guidelines

Start the interview by introducing yourself and the research you are conducting. Reassure the interviewee that his opinion/knowledge is anonymyzed and ask for the possibility to record the conversation.

The interview should be focused on five main areas of interest:

Opinion validity Try to find out more about the role and responsibility of the interviewee inside the company in order to validate his knowledge relevance to the research. Example questions:

- What is your technical background?
- What is your role inside the company?
- What responsibilities does your role impose?
- Are you directly involved in the development process?
- Are you directly involved in the design of the software product architecture?

Data management component usage Find out details about how do they currently use the software platform's data management component. Example questions:

- What software platform are you using?
- What were the reasons behind selecting this software platform?

- Did you consider looking at any benchmarks when choosing this platform?
- Do you use the default data management component of the software platform?
- Do you use an ORM (Object Relation Mapping) tool/module ?
- If not, how do you handle complex cross-entity queries? Did you built a custom ORM ?
- If yes, how did you end up using that ORM module?
- What databases do you use for storage?
- What is a typical data flow in your product?
- Do you have any views/background jobs that make extensive use of the data management component?

Quality attributes performance during product usage/development Figure out if they encountered any issues while the product was in development or production. If not, establish which could be the limits that they may surpass in the future. Example questions:

- How many potential users will use your product?
- Have you ever tested the product performance?
- Did you notice anything missing from the data management component?
- Did you have to do any optimization on the data queries to increase performance?
- Did you encounter any issues while developing the product?
- Do you consider changing the database type in the future?
- What do you plan to change in the next iteration of the product?

ORM usage in a new project Figure out what are the criteria on which they would select an ORM tool for a new fictive project. Example questions:

- Considering you would have to build a software product from scratch, would you consider using an ORM tool? Why?
- Do you think using an ORM is dependant on which software platform you are choosing for development?

- What would be the data requirements for a product that would push you to not use an ORM?
- Do you think it may be a good practice to create your own ORM from scratch? Why?
- Do you think it makes sense to use an ORM tool just for certain parts of an application?

Decision support model Discover characteristics that could be included in the decision model that could better help the decision maker. Example questions:

- Would you use a model that puts the quality attributes in contrast with the project data requirements to make the decision of using an ORM tool?
- Besides providing you a clear list of benefits/disadvantages, what else can help you in making a decision to (not) use an ORM tool?

Appendix C

Software platforms documentation analysis

This section provides a summary of each software framework documentation analysed when conducting this study. For each framework a short description is included along with a more comprehensive discussion about what does the data management component have to offer and how does it work at the architecture level.

A side note on the C programming language: Although one of the most popular programming languages even to this day, being an imperative (procedural) and low-level programming language, C did not spawn any ecosystem-based software platforms but rather created a base on which new programming languages have been built on directly or indirectly (such as C#, Go, Java, JavaScript, Python, PHP and many others). Its main method of extension are libraries.

C.1 .NET (C# / VisualBasic.net / J#)

The most popular software framework developed by Microsoft for their own ecosystem (available for other platforms as well as third party implementations) released in February 2002. It consists of two major components: a large framework library (Framework Class Library - FCL) and an application virtual machine runtime environment (Common Language Runtime - CLR).

The basic component from the .NET framework which handles access to data and data services is called ADO.NET . It includes support for Microsoft SQL Server and XML but most of the popular database solutions can also be plugged in through the OLE DB and ODBC standards. With the scope of providing a more abstract solution for data management two other components have been built on top of ADO.NET providing mainly ORM-like functionality: LINQ and the more functionality advanced Entity Framework.

C.2 Grails (Java)

Java-based full stack web development framework aimed at simplicity by the implementation of the Don't Repeat Yourself (DRY) principle. It was released in July 2005 and was built by looking at the popular modern frameworks such as Ruby on Rails, Django, TurboGears. It improved on the existing Java technologies such as Hibernate or Spring and wraps all these technologies together with the Groovy language, having a wide array of Domain Specific Languages (DSLs).

It ships by default with a ORM implementation called GORM (Grails' object relational mapping) which is actually Hibernate under the hood that allows entity definitions in a simplified DSL.

C.3 Struts 2 (Java)

Open source web application framework built upon the Java Servlets and following the MVC (Model-View-Controller) pattern. Released in October 2006 as a readaptation of the initial Apache Struts containing the WebWork framework which was forked from it.

It does not ship with its own data management solution but recommends either building your own DAO implementation or use one of the existing popular solutions, such as Apache Cayenne, Enterprise Java Beans, Hibernate, myBATIS.

C.4 Spring (Java)

Released in October 2002, the Spring Framework is a Java-based application framework and a inversion of control container for the platform in general. While most of its core functionality is often used in any Java application, it has multiple components that provide all the means necessary for building complex applications on top of the Java EE platform.

The data access component included in the framework acts as a wrapper for commonly used Java data management modules (such as JDBC, Hibernate, Apache Cayenne) providing resource management, exception handling and other features through the usage of template classes.

C.5 Android SDK (Java)

The Android SDK is the complete set of tools needed for application development on the Google Android mobile platform. With an official release in September 2008, the cross-platform development kit consists of a debugger, libraries, a handset emulator and afferent documentation.

As a data storage solution the platform allows: storing private basic data in key-value sets (preferences), storing raw data on the internal or external storage of the device, storing structured data in a SQLite database and through network services. Because of the client-centric applications built with the platform that do not require a vast amount of data stored, the data management component only provides data access adapters and a low level query abstraction library.

C.6 Node.js (JavaScript)

Only notable framework outside the client-side scope for the JavaScript programming language is *Node.js*, released in May 2009 . The platform is built upon Google's Chrome JavaScript runtime engine (V8) with the purpose of creating fast, scalar network applications. To achieve this goal it uses an event-driven, non-blocking I/O model which basically translates into: every I/O operation must use a callback.

Although the framework does not come with a complex built-in data management component, it does provide low-level modules for accessing file systems and network data. Due to the simple module loading system (1 to 1 correspondence between files and modules) and the included package manager (npm - node package manager), the open-source community has created a vast array of plugins, such Sequelize, Persistence.js, node-db. They provide both ORM solutions or basic SQL abstraction layers.

C.7 iOS SDK (Objective C / Swift)

Released in March 2008, the iOS SDK is Apple's application development kit for their mobile devices. Due to the strict terms and conditions of their proprietary platform, the applications may not be built with any other technology.

As part of the framework, by default data can be managed in three ways: using the included Core Data component, using an SQLite database and by using HTML5 localStorage. While the SQLite and HTML5 localStorage solution provide only a low level procedural implementation, the Core Data component provides a general purpose complete data management solution. It uses the included SQLite database system and provides a persistent object oriented storage solution and other complex ORM functionality.

C.8 CodeIgniter (PHP)

One of the popular web application frameworks in the PHP ecosystem released in February 2006, CodeIgniter, is a lightweight platform for building dynamic websites with speed in mind. It follows the Model-View-Controller (MVC) pattern and as its underlying programming language it is cross-platform.

The data management component is loosely coupled and provides access to a wide variety of databases through PHP's database abstraction layers (PDO, ODBC etc.). By default, a simple SQL abstract interface is available along with a simple Active Record class that follows the pattern with the same name.

C.9 Zend (PHP)

The Zend framework was initially released in March 2006 and is identified as an open source web application framework. As the other popular frameworks, it follows a use-at-will architecture with loosely coupled components and follows the MVC pattern.

The Zend\Db module contains all the functions regarding data management. It provides PHP database drivers adaptation through the Zend\Db\Adapter sub-module together with an SQL abstraction layer (Zend\Db\Sql), a Table Gateway and a Row Gateway implementation.

C.10 Django (Python)

Open source full-stack web application framework written in Python programming language released in July 2005. It follows the Model-View-Controller (MVC) pattern.

The data management component is present in the *django.Db* module and exists in the form of an ORM (Object-relation mapping) tool. The default implementation follows Fowler's Active Record pattern. While the implementation technique adopts a loosely coupled modules philosophy this does not imply that decoupling the data management component will not have consequences, in result replacing it will need an extra effort for making the new component fit the system.

C.11 Ruby on Rails (Ruby)

Open source full-stack web application framework written in Ruby programming language released in December 2005. It follows the Model-View-Controller (MVC) pattern.

The data management component is called ActiveRecord and is actually an ORM tool. Although seen as an important part of the framework, it is modular and can be replaced with other custom data management components such as DataMapper or MongoMapper.

If follows the Active Record pattern described by Martin Fowler in his book Patterns of Enterprise Application Architecture. It creates objects that carry both persistent data and behaviour which operates on that data. Active Record takes the opinion that ensuring data access logic is part of the object will educate users of that object on how to write to and read from the database.

Appendix D

Experiments performed

This annex includes the complete validation experiments results also mentioned in Chapter 4. The experiments were based on multiple implementations of the pseudo-code found in Figure 4.2 which was done for a number of four frameworks: Grails (Java), Yii Framework (PHP), Django (Python), Ruby on Rails (Ruby). The code was then ran under the same limited resources conditions in a virtual machine.

D.1 Performance experiment

For the performance experiment the tests are measuring the response time using a time difference technique (register time on operation start/end) for the following sections:

- Create entity instances and the relationships between them (R1)
- Browse through records and their relationships and update them (R2)
- Retrieve records and their relationships and delete them (R3)

Each section was ran 10 times and an average was made with the following formula:

$$AVG(Rx) = \frac{\sum_{i=1}^{10} Rx_i}{10}$$

This was also done for the whole code by summing up all the individual section response times. After this, the average response times were compared for each platform and a percentile was calculated as: $D = \frac{AVG(R)_{ORM}}{AVG(R)_{SQL}}$

Platform	**AVG(R1)**	**AVG(R2)**	**AVG(R3)**	**AVG(R1+R2+R3)**
java (ORM)	0.282132	0.250973	0.169387	0.702492
java	0.074864	0.073737	0.073259	0.221860
php (ORM)	0.06425	0.07227	0.05610	0.19262
php	0.032103	0.038910	0.036026	0.107040
python (ORM)	0.086445	0.131038	0.094342	0.311825
python	0.034631	0.042338	0.037272	0.114241
ruby (ORM)	0.238332	0.186045	0.131501	0.555878
ruby	0.052164	0.061461	0.053072	0.166698

Table D.1: Performance response time results

Platform	**ORM**	**no ORM**	**Times slower (D)**
java	0.702492	0.221860	3.17
php	0.19262	0.107040	1.80
python	0.311825	0.114241	2.73
ruby	0.555878	0.166698	3.33

Table D.2: Performance response time results comparison

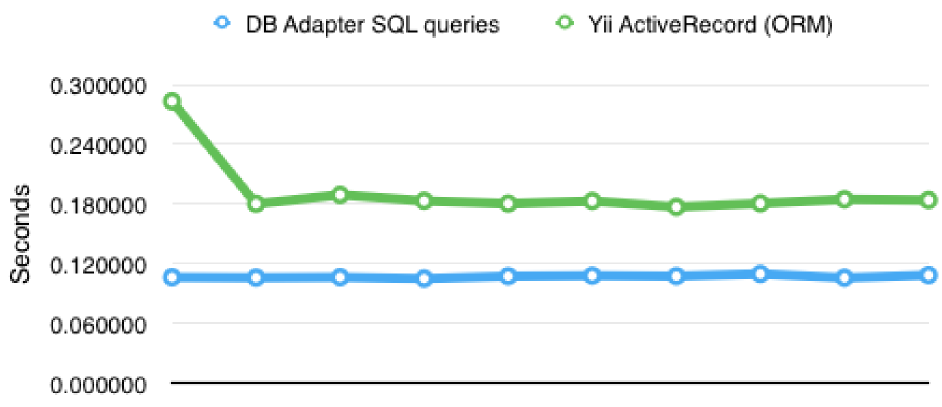


Figure D.1: Response time performance chart for Yii Framework (PHP)

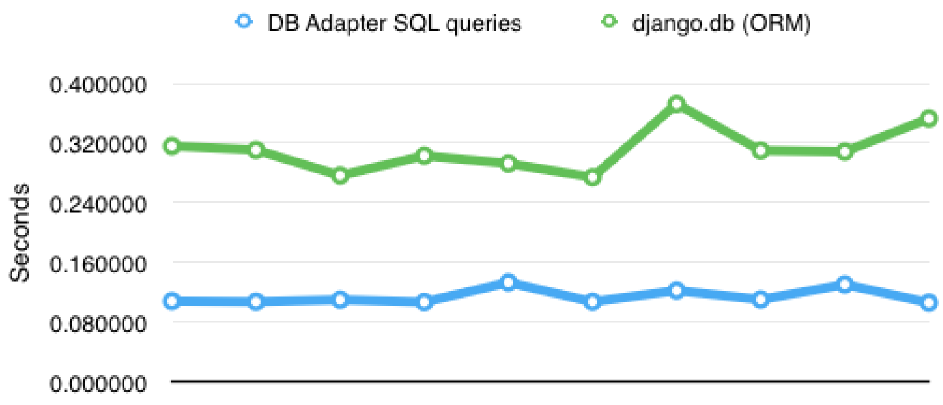


Figure D.2: Response time performance chart for Django Framework (Python)

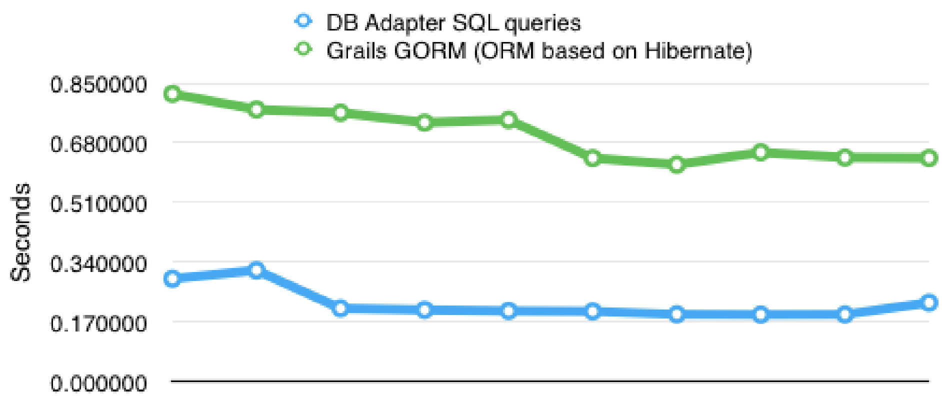


Figure D.3: Response time performance chart for Grails Framework (Java)

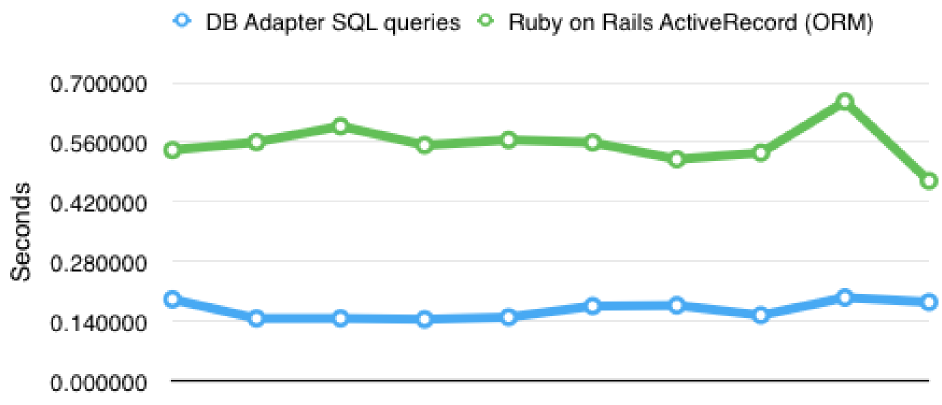


Figure D.4: Response time performance chart for Ruby on Rails (Ruby)

D.2 Maintainability experiment

The maintainability experiments involved extracting the code that represents the pseudo-code from Figure 4.2 and evaluating it by using the number of lines of code (physical/logical) and the Halstead complexity. Please refer to the legend in Table D.3 when analysing the results. For the Halstead complexity the following formulae were used:

$Vocabulary(O) = O1 + O2$

$Length(N) = N1 + N2$

$Volume(V) = N * \log_2 O$

$Difficulty(D) = \frac{O1}{2} * \frac{N2}{O2}$

$Effort(E) = D * V$

Term	**Definition**
LOC	Physical source lines of code
LLOC	Logical lines of code
O1	Number of distinct operators
O2	Number of distinct operands
N1	Total number of operators
N2	Total number of operands

Table D.3: Maintainability experiment results legend

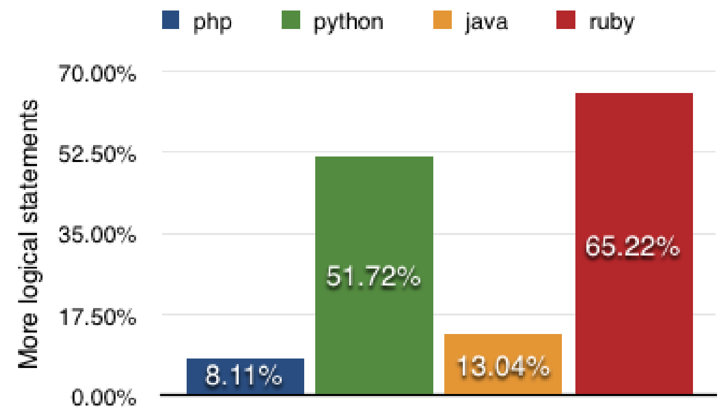


Figure D.5: Logical statements comparison chart

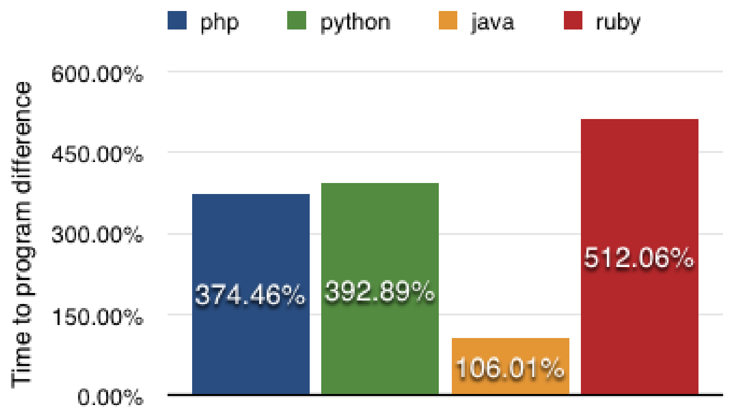


Figure D.6: Time to program comparison chart

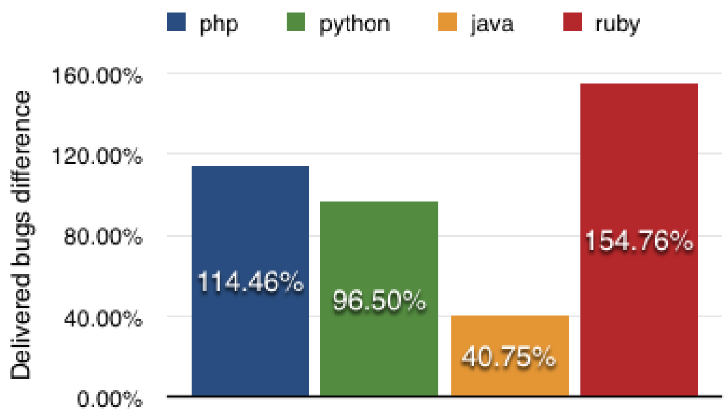


Figure D.7: Delivered bugs comparison chart

Platform	**ruby**	**ruby** (no ORM)	**php**	**php** (no ORM)	**java**	**java** (no ORM)	**python**	**python** (no ORM)
LOC	25	25	56	48	45	27	29	29
LLOC	23	38	46	52	37	40	29	44
-	-	-	-	-	-	-	-	-
O1	14	19	16	20	17	19	13	20
O2	33	39	37	45	40	38	39	39
N1	90	242	209	413	165	253	127	267
N2	76	159	118	254	127	158	92	150
Vocabulary	47	58	53	65	57	57	52	59
Length	166	401	327	667	292	411	219	417
Volume	922.06	2349.05	1873.03	4016.92	1703.20	2397.32	1248.40	2453.06
Difficulty	16.12	38.73	25.51	56.44	26.99	39.50	15.33	38.46
Effort	14864.75	90980.53	47787.58	226732.78	45965.21	94694.05	19142.08	94348.54
Time to program	825.82	5054.47	2654.87	12596.27	2553.62	5260.78	1063.45	5241.59
Delivered bugs	0.31	0.78	0.62	1.34	0.57	0.80	0.42	0.82

Table D.4: Maintainability experiment results

D.3 Scalability experiment

The scalability experiment involved sending concurrent requests to each framework for executing the code and analysing the time it takes the system to respond to all requests along with the number of timeouts generated. In order to limit the tests running time and also add a realistic non-functional requirement to the experiment, a timeout of 5 seconds was enforced on each request.

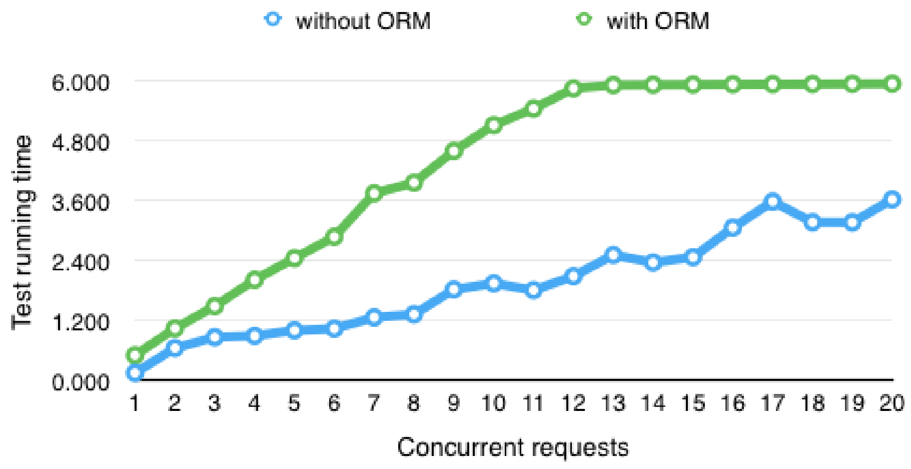


Figure D.8: Scalability experiments results (Ruby on Rails - Ruby)

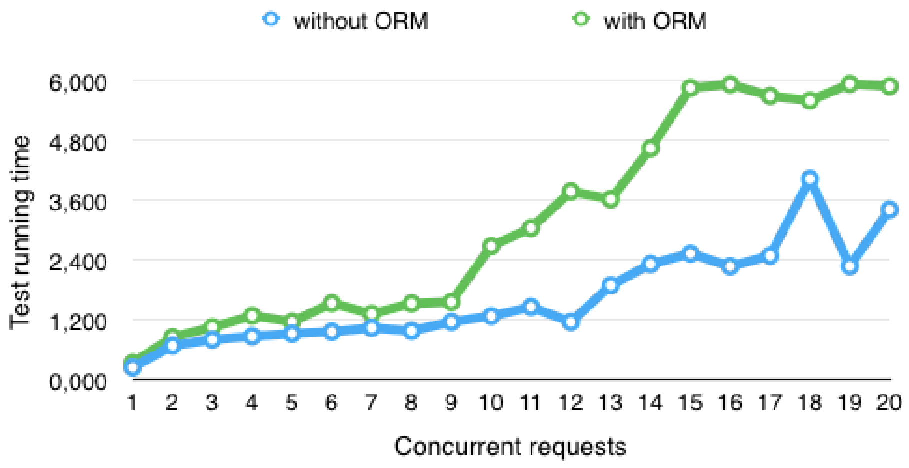


Figure D.9: Scalability experiments results (Django - Python)

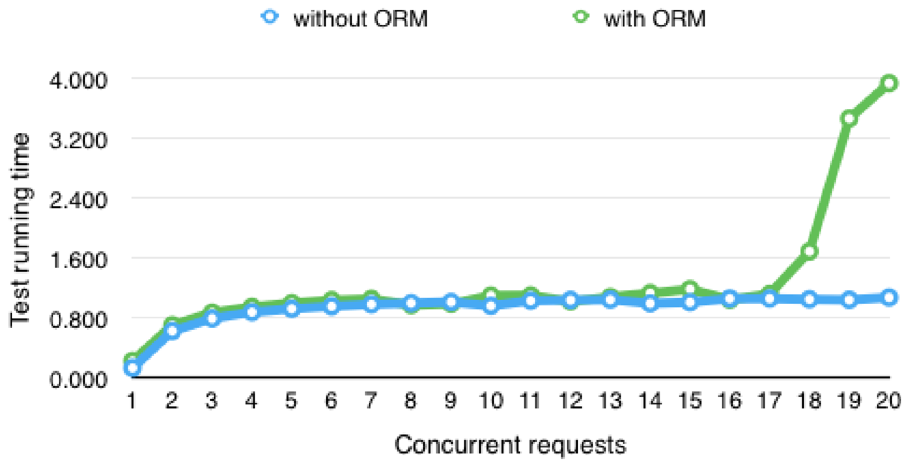


Figure D.10: Scalability experiments results (Yii Framework - PHP)

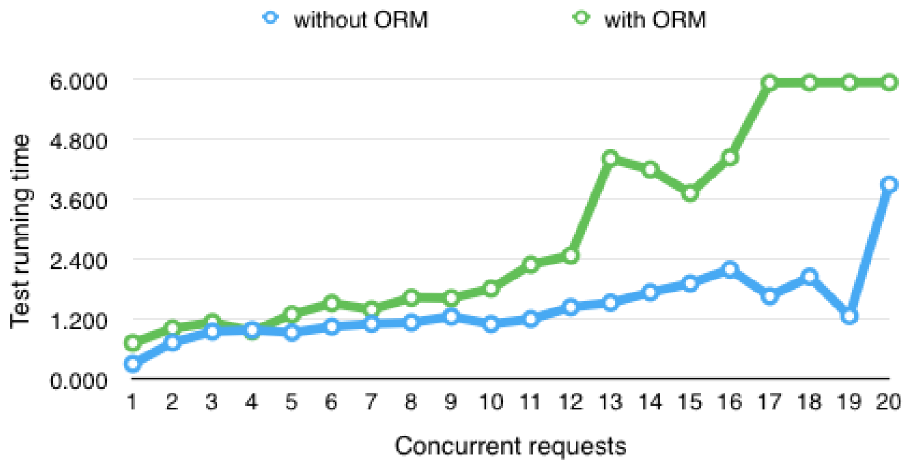


Figure D.11: Scalability experiments results (Grails - Java)

Appendix E

Statement of authenticity

I hereby confirm that this thesis document represents my own work. It was not composed by anyone else for my benefit and I did not copy its content from another person. All sources that I have used have been properly and clearly documented.

I further attest that if I have used the ideas, words, or passages from an external source, I have quoted those words or paraphrased them and have provided a clear and appropriate documentation of the source of that material.

Friday 6th February, 2015