FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

FEUP

Improving Variability of Applications using Adaptive Object-Models

João Gradim Pereira

Master in Informatics and Computing Engineering

Supervisor: Ademar Aguiar (Ph.D)

Supervisor: Hugo Ferreira (Ph.D AbD)

17th January, 2011

Improving Variability of Applications using Adaptive Object-Models

João Gradim Pereira

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: João Pascoal Faria (Ph.D)

External Examiner: António Nestor Ribeiro (Ph.D)

Supervisor: Ademar Aguiar (Ph.D)

Supervisor: Hugo Ferreira (Ph.D AbD)

11th February, 2011

Abstract

Most software systems exist in an ever-changing environment. The difficulty to capture the requirements and craft the system to mirror these requirements often requires higher costs and development time. Moreover, these systems are not static in nature and often need to be modified to reflect changes in the environment and stakeholders' expectations.

An answer to these problems is to create information systems that are flexible enough to introduce model-level changes as quickly and cheaply as possible. One particular point of interest is to allow the end-users to modify the system definition — which implies that modifications have to be performed without any programming knowledge or system redeploying. This can be achieved by using a meta-architecture pattern known as Adaptive Object-Modeling (or AOM).

An Adaptive Object-Model architecture is a system architecture that represents classes, attributes, relationships, and behavior as *metadata*, where the system definition is based on instances of model abstractions rather than classes: this allows for the easy modification of the domain model in runtime, discarding the need for redeploying. Also, a set of techniques have been developed to build the user interface automatically from the domain model, in order to create truly adaptive software.

The platform *escolinhas.pt* is an expanding Ruby on Rails application that works as a private social network for schools, students aged 6 to 10, along with their teachers and parents. It is based on the structure of real Portuguese primary education schools and promotes the use of digital tools to create collaborative work.

This report describes the work and research involved in adapting the ideologies and techniques used in building AOM architectures to already existent applications based on static database schemas and MVC architectures, such as *escolinhas.pt*.

This research focuses on studying the current design for the *escolinhas.pt* platform and pinpointing which modules of the application have special needs regarding their variability. Three modules or areas of the application were chosen to conduct this study: user roles (and authorization logic), the social network, and the document editor. Each one of these areas is studied in order to determine possible problems and gather the necessary variablity requirements. For each one of the areas, small, proof-of-concept prototypes are built, applying the most appropriate design patterns from a set of candidates, detailing implementation details for each one, as well as studying the impact that each prototype had.

Results show that the application of these design patterns to a MVC based framework such as Ruby on Rails are not only possible, but can have a positive impact on the overall design and performance of the application.

Resumo

A grande maioria dos sistemas de informação existem em ambientes altamente variáveis. A dificuldade em capturar os requisitos de modo a modelar o sistema de acordo com estes causa custos de produção e tempos de desenvolvimento mais altos. Muitos destes sistemas não são estáticos e sofrem modificações regulares de modo a reflectir as mudanças sofridas pelo meio ambiente e expectativas dos clientes.

Uma resposta possível para este problema é a criação de sistemas de informação que sejam flexíveis o suficiente para que a introdução de modificações ao nível do modelo de domínio sejam o mais rápido e com o menor custo possível. Um ponto de interesse em particular é o de permitir que os utilizadores finais do sistema possam modificar o seu modelo de domínio — o que implica que estas modificações sejam efectuadas sem qualquer conhecimento de programação ou da instalação do sistema em si. Este objectivo pode ser conseguido através da utilização de um padrão de desenho de meta-arquitecturas conhecido como *Adaptive Object-Models* (ou AOM).

Uma arquitectura baseada em *Adaptive Object-Models* é uma arquitectura de sistemas que representa classes, atributos, associações e comportamentos como meta-dados, em que a definição do sistema se baseia em instâncias de abstrações do modelo em vez de classes: isto permite uma fácil modificação do modelo do sistema, removendo a necessidade de re-instalação. A par do desenvolvimento deste tipo de sistemas, uma série de técnicas e padrões foram criados para que a interface do utilizador seja passível de ser criada automaticamente.

A plataforma *escolinhas.pt* é uma applicação criada em Ruby on Rails e que funciona como uma rede social para alunos de 6 a 10 anos de idade, conjuntamente com os seus encarregados de educação e professores. A sua estrutura é baseada nas escolas primárias Portuguesas, e promove o uso de ferramentas digitais para a criação de trabalhos colaborativos.

Este relatório descreve o trabalho e pesquisa envolvido na adaptação de ideologias e técnicas usadas na construção de arquitecturas AOM a aplicações já existentes, assentes em esquemas de bases de dados estáticas e arquitecturas MVC, como é o caso da plataforma *escolinhas.pt*.

Este trabalho foca-se no estudo do desenho actual da plataforma *escolinhas.pt* e em capturar quais os módulos da aplicação que têm maiores necessidades de variabilidade. Três módulos da aplicação foram escolhidos para este estudo: os papéis (*roles*) dos utilizadores, a rede social e o editor de documentos da plataforma. Cada uma destas áreas é estudada de modo a determinar possíveis problemas e recolher os requisitos de variabilidade de cada uma. Para cada um destes módulos pequenas provas de conceito foram construídas como protótipos, aplicando os padrões de desenho mais apropriados de

um conjunto de padrões candidadatos, referindo os detalhes de implementação bem como estudando o impacto que cada uma destas soluções teve na aplicação.

Resultados mostram que a aplicação de padrões de desenho associados a arquitecturas AOM a uma framework baseada em MVC são não só possíveis, como podem ter um impacto positivo tanto no desenho como no desempenho da aplicação.

Acknowledgements

To everyone at *escolinhas.pt*, for making me feel at home throughout all these months.

To Ademar Aguiar and Hugo Ferreira, for guiding me and for their patience when I had nothing more than questions.

To Gonçalo and Paulo, for continuously motivating me, supporting my work and helping me whenever I needed.

To everyone who shared the last 5 and half years with me, night, and day, and then some more.

To my family, for their undying patience and support, for making me believe in myself and allowing me to be whomever I want to be.

João Gradim

“Computers are good at reading instructions, not at reading your mind”

Donald E. Knuth

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Motivation and Objectives . . . 2
- 1.3 Report Overview & Structure . . . 3

2 State of the Art **5**
- 2.1 Approaches to Create Adaptable Software . . . 5
- 2.2 Software Product Lines . . . 5
- 2.3 Domain-Driven Design . . . 6
- 2.4 Model-Driven Engineering . . . 6
- 2.5 Frameworks . . . 7
- 2.6 Metaprogramming . . . 7
- 2.7 Aspect-Oriented Programming . . . 7
- 2.8 Design Patterns . . . 8
- 2.9 Adaptive Object-Models . . . 9
  - 2.9.1 AOM Architecture . . . 9
  - 2.9.2 Pattern Composition . . . 14
  - 2.9.3 AOM GUI Generation . . . 15
  - 2.9.4 Oghma . . . 15
- 2.10 Ruby . . . 17
  - 2.10.1 Ruby Metaprogramming . . . 17
  - 2.10.2 Ruby On Rails . . . 18
- 2.11 Conclusions . . . 18

3 Problem Statement **21**
- 3.1 Problem Description . . . 21
- 3.2 Architectural & Design Patterns . . . 22
- 3.3 Case-study Areas . . . 22
  - 3.3.1 Roles . . . 22
  - 3.3.2 Social network and contacts . . . 23
  - 3.3.3 Document Editor . . . 23
- 3.4 Conclusions . . . 23

4 Approach & Results **25**
- 4.1 Research Design . . . 25
  - 4.1.1 Current Design Analysis . . . 25
  - 4.1.2 Variability Analysis . . . 26

4.1.3 Implementation & Impact Analysis 26
4.2 User Roles 27
4.2.1 Variability Requirements 27
4.2.2 Candidate Patterns 27
4.2.3 Chosen Patterns & Rationale 28
4.2.4 Implementation 29
4.2.5 Impact Analysis 29
4.3 Social Network 30
4.3.1 Variability Requirements 31
4.3.2 Candidate Patterns 32
4.3.3 Chosen Patterns & Rationale 32
4.3.4 Implementation 33
4.3.5 Impact Analysis 35
4.4 Documents 36
4.4.1 Variability Requirements 36
4.4.2 Candidate Patterns 37
4.4.3 Chosen Patterns & Rationale 38
4.4.4 Implementation 39
4.4.5 Impact Analysis 41
4.5 Conclusions 44

5 Conclusion **45**
5.1 Conclusions 45
5.2 Further Work 46

References **47**

List of Figures

2.1 MOF levels. 9
2.2 TYPE-OBJECT pattern 10
2.3 PROPERTY pattern 11
2.4 TYPE-SQUARE 11
2.5 STRATEGY pattern, as applied to an AOM architecture 12
2.6 INTERPRETER pattern, as applied to an AOM architecture 12
2.7 SYSTEM MEMENTO pattern 13
2.8 ACCOUNTABILITY pattern 13
2.9 Revised PROPERTY pattern 14
2.10 AOM core architecture and design 14
2.11 Implementation model of the structural meta-model for the Oghma framework 16

4.1 Current User Roles Model. 28
4.2 Conceptual User Roles Model. 29
4.3 Current User Network Model. 31
4.4 Simplified Network Models. 32
4.5 Accountability Implementation for User Network. 33
4.6 Average number of queries performed per number of contacts. 35
4.7 Current Documents Model. 37
4.8 Conceptual Documents Model. 38
4.9 Average number of queries performed per number of *Blocks* in a single Document. 42

LIST OF FIGURES

List of Tables

4.1 AccountabilityTypes created to describe every type of existent relation. . . 34

LIST OF TABLES

Abbreviations

ACL	Access Control List
AOP	Aspect-Oriented Programming
API	Application Programming Interface
AR	ActiveRecord (Design Pattern)
CRUD	Create, Read, Update, Delete
DBMS	Database Management System
DDD	Domain-Driven Design
DSL	Domain Specific Language
DSML	Domain Specific Modeling Language
GUI	Graphical User Interface
MVC	Model, View, Controller (design pattern)
OOP	Object-Oriented Programming
ORM	Object-Relational Mapping
PIM	Platform-Independent Model
RoR	Ruby on Rails
UI	User Interface
UML	Unified Modeling Language

Chapter 1

Introduction

An Adaptive Object-Model (AOM) is an architectural style based on metamodeling and object-oriented design that exposes its domain model to the end-user, and aims to create better mechanisms for the evolution and adaption of software systems to their environments.

This document presents a study that shows that the patterns and ideologies described as being part of AOM architectures can be successfully adapted to other architectures, with objectives similar to the original ones.

In this introductory chapter, the background and context of the problem are presented, along with the motivation and objectives. The structure of the document is presented, as well as some typographical conventions used to enhance readability.

1.1 Context

Software systems are usually designed with a specific purpose in mind. They rely on a series of requirements that are often very difficult to capture and maintain, as they have a tendency to evolve faster than the implementation. This is caused mainly by the poor understanding by the stakeholders about their needs and expectations of what a software system should be able to do [PT07]. These situations lead to higher costs in software development, as creating and maintaining software systems is a knowledge intensive task [AdOdSBD07]. Moreover, most of these systems are not static, and have a constant need to evolve in order to adapt themselves to their environment and new business rules, shifting the stakeholders' needs and expectations about these software systems.

In face of these situations, new development methods started to focus more on iterative and incremental approaches, accepting *incompleteness* as part of every software system's development cycle [WC03]. At the same time, many new systems are being developed with an emphasis on flexibility and run-time configuration [YJ02]. These approaches present a clear contrast with an up-front, full specification for a software system, which, albeit beneficial for some cases, is impractical in constant evolution scenarios. This may lead to

sudden changes in software requirements and, inevitably, refactoring. In many cases, there are deadlines to be met and budgets to adhere to: if the resources are insufficient, these situations often lead to a BIG BALL OF MUD — an "haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle" [JY97] — ultimately leading to unmaintainable systems, very costly to modify and adapt to the stakeholders needs [FY97].

1.2 Motivation and Objectives

Developing quality software is a hard process. Maintaining, as well as adding new features to these software systems is costly and time-consuming. If the process of shaping an information system is made to be as streamlined as possible, the modification of the system's architecture (model and relation-wise) becomes a simple task of adapting the platform according to the natural evolution of the business rules, environment changes and the users and stakeholders needs.

The type of systems mentioned in 1.1 has very specific architectures, which allow for the modification of the system model in runtime. While this may not be desirable in every situation — either because of performance reasons or because the usage of such an architecture could bring an undesirable level of complexity — part of the logic, ideas and patterns of these meta-architectures could be applied to different types of software architectures in order to achieve similar objectives.

Ruby on Rails is one of the most widely used web development frameworks. It provides a series of tools and conventions that allow a developer to focus on the application design rather than implementation details, as most of that work is automatically handled by the framework itself [ORP]. However, when the kind of aforementioned needs arise in these type of applications, the framework could present itself as a barrier to easily building adaptive software as it is tied to a single architectural pattern (MVC) and a series of conventions that favour using relational databases coupled with *Object-Relational Mapping* (ORM) for data manipulation — meaning that, in most applications, the database schema is an integral part of the platform design[1]. As such the association of these types of design patterns to a full-stack framework such as Rails presents an interesting challenge: how to effectively and harmoniously combine these apparently different ideologies — Rails is based on a static, descriptive model and AOMs use data to describe the system — to take advantage of the best of both worlds.

This project will be applied to the *Escolinhas* [TC] project, a growing, Portuguese, Ruby On Rails based project, in order to enhance the design and provide the means to easily evolve the platform. Escolinhas aims at supporting social and collaborative work for

[1] The somewhat recent appearance of modern non-relational databases such as MongoDB and CouchDB and others shifted this definition towards the code instead of the database schema. However, Rails initially only supported relational databases such as MySQL, and the usage of non-relational databases is outside the scope of this dissertation.

children in elementary schools involving students, teachers and parents as its users. With the users demanding better and more adaptable teaching tools, it becomes an excellent case-study application to research, test and apply all the work and discoveries made throughout the course of this dissertation.

1.3 Report Overview & Structure

This report is structured as follows:

Chapter 2: "State of the Art" reviews the most important methodologies and patterns used to make software as adaptable and maintainable as possible.

Chapter 3: "Problem Statement" exposes the problem to be addressed and thoroughly explains it, while explaining its usefulness.

Chapter 4: "Approach & Results" reviews the current design of the platform, and performs a variability analysis of the system, choosing three focus areas. Then, for each one, demonstrates its variability requirements, candidate patterns, the chosen patterns and rationale behind such decisions, implementation details and impact analysis.

Chapter 5: "Conclusion" reviews the project, drawing conclusions about the issues addressed in Chapter 4. It also provides a summary of contributions and some insights on which future developments have been considered.

Some typographical conventions are used throughout the document to improve its readability. Pattern names are displayed with SMALL CAPS. Whenever a class (entity) currently part of the escolinhas.pt is referred, it will be typefaced in *italics*. `Typewriter text` is used when referring to entity attributes or a certain programming language term. References and citations appear inside [square brackets] and in highlight color. Highlighted text will act as an hyperlink when visualizing this document in a computer.

Chapter 2

State of the Art

This chapter presents a bibliographic review on the areas of interest and contribution of this study and it is mainly divided in three parts. The first one presents the most commonly used techniques, approaches and ideologies to build adaptable software. The second part is concerned with the current research on AOMs, describing the main design patterns and how they work together to create an AOM architecture. A current implementation of this architecture is presented. The last part briefly presents the Ruby programming language and describes how some of its concepts can be seen as analogous to the techniques and patterns described before.

2.1 Approaches to Create Adaptable Software

There are two main approaches to make systems adaptable: generative programming approaches and meta-architectures.

Generative programming methods approach the adaptability of a system by using an ontological model representative of this system to automatically create executable artifacts or code skeleton than can be further refined according to different needs.

Rails scaffolding and model generation is an excellent example of this approach, which will be further explained in Section 2.10.2.

On the other hand, systems created with a special architecture designed to adapt at runtime to new user requirements and rules by using descriptive information about the system model that can be interpreted at runtime are sometimes said to possess a "reflective architecture" or a "meta-architecture" [YBJ01].

2.2 Software Product Lines

The Software Product Lines (SPL) paradigm is promoted as a means of reducing time to market, increasing productivity, improving quality and gaining cost effectiveness and

efficiency through large-scale reuse [TC06]. Software product line methods (SPLMs) are practices-based, or plan-driven, software development approaches in which a set of software-intensive systems that share a common, managed set of features are produced from a set of re-usable core assets in a predictive, rather than opportunistic way — meaning artifacts are only created when the need for their use arises, instead of providing generic, reusable components.

2.3 Domain-Driven Design

Domain-Driven Design (henceforth referred to as DDD) is a philosophy and way of thinking, first described by Eric Evans in [Eva03], aimed at accelerating software projects that have to deal with complicated domains, with a two-fold premise [EGHN]:

- For most software projects, the primary focus should be on the domain and domain logic; and
- Complex domain designs should be based on a model.

A third, informal premise is usually associated with DDD, wherein a collaboration between technical domain experts must exist, in order to quickly, over various iterations, reach the core of the problem's concept.

2.4 Model-Driven Engineering

Model-Driven Engineering first appeared in 2001 [Mil03] as an answer to the growing complexity of system architectures. This growing complexity and the lack of an integrated view "forced many developers to implement suboptimal solutions that unnecessarily duplicate code, violate key architectural principles, and complicate system evolution and quality assurance" [Sch06].

To address these issues, Model-Driven engineering combines *domain-specific modeling languages* (DSML) with *transformation engines* and *generators* in order to generate various types of artifacts, such as source code or alternative model definitions.

The usage of DSML ensures that the domain model is perfectly captured in terms of syntax and semantics. This guarantees a flatter learning curve as the concepts present in the language are already known by the domain experts. This also helps a broader range of experts, such as system engineers and experienced software architects, to ensure that software systems meet user needs.

The ability to synthesize artifacts from models helps to ensure the consistency between application implementations and analysis information associated with functional and quality of service requirements captured by models.

2.5 Frameworks

Frameworks provide a series of loosely-coupled components created for a specific purpose that provide generic functionality for the creation of software systems. These components can be overridden or specialized in order to create specific functionality. Frameworks can improve developer productivity and improve the quality, reliability and robustness of new software. Developer productivity is improved by allowing developers to focus on the unique requirements of their application instead of spending time on application infrastructure. XNA [Mic] and Ruby on Rails (RoR) [Han] are good examples of popular frameworks that aim to cut development time and costs in very different scenarios.

Frameworks can also be used together with code-generation techniques [DH04, Che08] to improve the overall production speed and ease of use.

2.6 Metaprogramming

As defined by Robert D. Cameron and M. Robert Ito in [CI84],

> "A metaprogramming system is a programming facility (subprogramming system or language) whose basic data objects include the programs and program fragments of some particular programming language, known as the target language of the system. Such systems are designed to facilitate the writing of metaprograms, that is, programs about programs. Metaprograms take as input programs and fragments in the target language, perform various operations on them, and possibly, generate modified target-language programs as output."

To put it simply, metaprogramming is code that manipulates code. The advantages of this paradigm have been described many times in software reuse literature. The translation of high-level descriptions into low-level implementations (by means of application generators) allows a developer to focus on specification based on tested standards rather than implementation, making tasks like system maintenance and evolution much easier and affordable [Bas87, Cle88].

2.7 Aspect-Oriented Programming

Aspect-Oriented Programming is a programming paradigm that isolates secondary or auxiliary behaviours from the implementation of the main business logic. It has been proposed as a viable implementation of modular crosscutting concerns. Since crosscutting concerns cannot be properly modularized within object-oriented programming, they are expressed as aspects and are composed, or woven, with traditionally encapsulated functionality referred to as components [KM05, Ste06]. This paradigm has been gaining some momentum

particularly because even conceptually simple crosscutting concerns such as tracing during debugging and synchronization, lead to tangling, in which code statements addressing the crosscutting concern become interlaced with those addressing other concerns within the application [LC03].

Many AOP implementations work by automatically injecting new code into existing classes (a process which is commonly called *decorating* a class), effectively adding new functionalities that promote extensive code reuse [Ste06, LC03, MS03].

2.8 Design Patterns

As stated by the architect Christopher Alexander in [AIS+77]:

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

Despite applying the term *design pattern* to architectural problems, it is possible to transpose these ideas to object-oriented design. As such, a design pattern is, to put it simply, a common solution to a recurring problem. [BMR+96a, AIS+77, GHJV94] then defines the four essential components of a design pattern:

- The **Name**, which captures the essence of a design pattern and describes it in as few words as possible (one or two, usually);

- The **Problem**, which describes the conditions necessary to apply the pattern, by explaining the problem and its context;

- The **Solution**, which describes the elements that make up the design, their relationships, responsibilities, and collaborations, while keeping it as general as possible, so that its application is possible within multiple contexts:

- The **Consequences**, which denotes the results and trade-offs pertaining to the application of the pattern, allowing the evaluation of alternative designs and understanding the costs and benefits of applying the pattern.

2.9 Adaptive Object-Models

Despite all of the advances in the aforementioned areas, most (if not all) the currently used techniques and methodologies still require a programmer or a domain expert in order to modify a system definition and model. A specific type of architecture is required for the end-user to be able to modify part or the entirety of a system to fit the underlying model to their own needs.

As such, an end-user is not expected to have any kind of knowledge about programming, design patterns and system architectures, or about a system of this type comes to be. However, the end-user is sometimes the most knowledgeable part of the development chain, in regards to how the system should behave.

2.9.1 AOM Architecture

An Adaptive Object-Model pattern is a system architecture that represents classes, attributes, relationships, and behavior as *metadata*. The system definition is based on instances of model abstractions rather than classes. This allows the modification of these model abstractions in runtime to reflect changes in the domain, effectively modifying the system behavior. As a direct consequence, the system instantly reacts to these changes, making it adaptable to change, without the need for recompiling or redeploying [YBJ01]. This architectural pattern is akin to the Meta Object Facility (MOF), which main purpose is to provide a metadata management and implementation framework. The basic architecture is divided into four tiers, each one compliant with the higher level [OMG06], as depicted in Fig. 2.1.

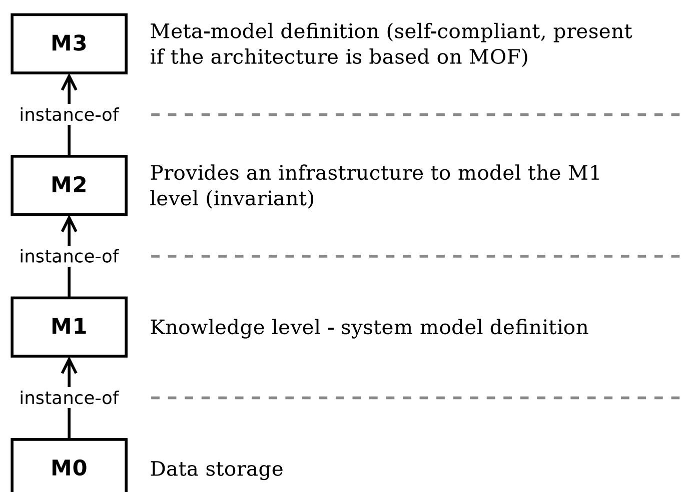


Figure 2.1: MOF levels.

This kind of architecture relies on a series of design patterns: the TYPE-OBJECT and PROPERTY patterns form the basic building blocks whereupon the AOM architecture settles. Despite being extremely simple, they create the fundamental infrastructure able to decouple the model definition from code-level implementation. In addition to these two patterns mentioned, other design patterns that are able to form an AOM architecture will be described, as well as the interactions between them.

2.9.1.1 TYPE-OBJECT Pattern

Object-oriented languages usually structure a program as a set of classes that define the structure and behavior of objects, usually organizing them as a separate class for each object type, which means that any structural change to the model requires code-level modifications. However, variable systems are usually faced with the problem of having a class that will be subclassed by an arbitrary number of specializations. The key to solve this problem is to detach the object definition from the code level and instead define it using meta-data — generalizing objects and describing their variation as parameters. TYPE-OBJECT works by splitting a class in two [YBJ01]: the meta-class for the object to be created — EntityType, and an instance of that class — Type. Fig. 2.2 shows the UML class diagram for this design pattern.

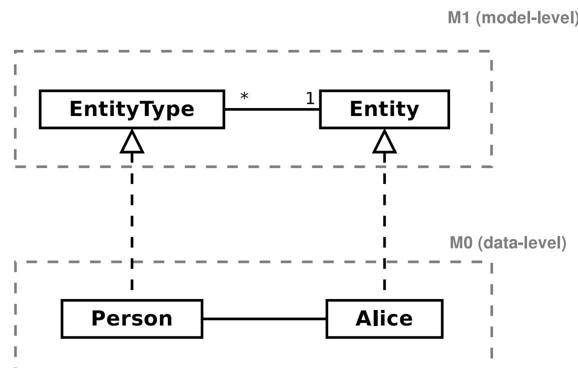


Figure 2.2: TYPE-OBJECT pattern, adapted from [YFRT98].

2.9.1.2 PROPERTY Pattern

Similar to the problem solved by the TYPE-OBJECT, the PROPERTY pattern addresses the analogous issues of having the need to change the attributes (sometimes called *members*) of a class. The anticipation of these structural changes leads to the PROPERTY pattern, where an attribute is split in two classes: the meta-class for the object to be created — PropertyType, and an instance of that attribute — Property. Fig. 2.3 represents the UML model for the PROPERTY design pattern.

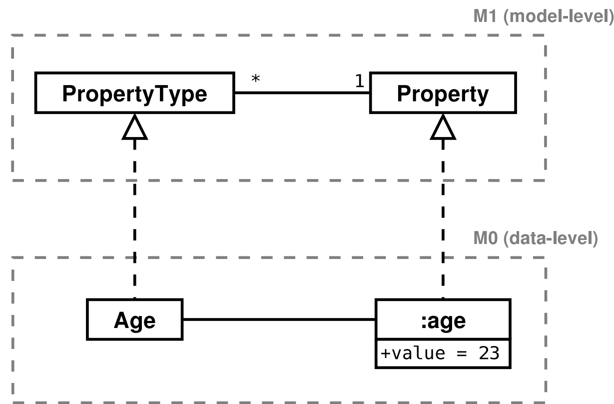


Figure 2.3: PROPERTY pattern, adapted from [YFRT98].

2.9.1.3 TYPE-SQUARE Pattern

Usually a class is modeled with a number of different attributes, representing their real world counterparts. So, in order to make a runtime modifiable class, an user must be able to modify both its *definition* and *attributes*. The answer to this issue is to use both TYPE-OBJECT and PROPERTY patterns at the same time, creating what is known as the TYPE-SQUARE pattern — which forms the basis of any AOM architecture [YJ02]. Figure 2.4 shows the UML model for this pattern.

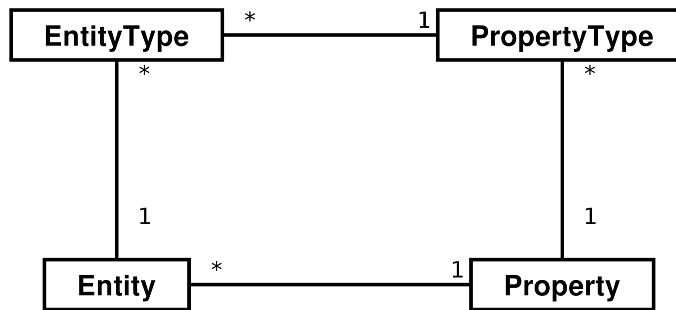


Figure 2.4: TYPE-SQUARE pattern, adapted from [YBJ01].

2.9.1.4 STRATEGY and RULE OBJECT Pattern

The original goal of the STRATEGY pattern is to, given a set of algorithms, encapsulate each one in order to use them interchangeably, which allows the easy usage of different strategies to solve different problems [GHJV94].

As AOM architectures are concerned, this pattern is used to perform user-defined validations for user input, whichever they may be. An example can be found on Fig. 2.5.

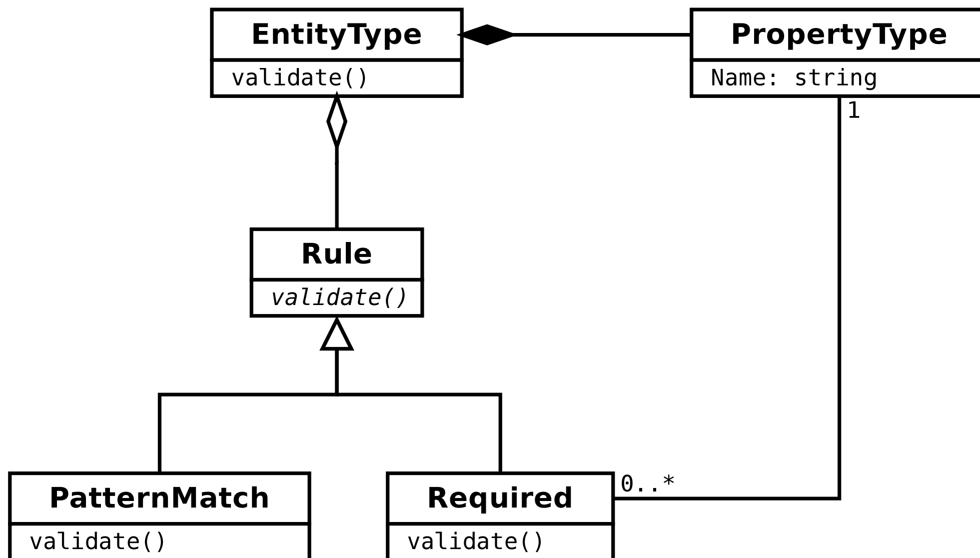


Figure 2.5: STRATEGY pattern, as applied to an AOM architecture, adapted from [GHJV94].

2.9.1.5 INTERPRETER Pattern

The INTERPRETER pattern commonly is used when a recurring problem within a platform arises. When this is the case, it might be fruitful to define a simple language to solve these types of problems [GHJV94]. Regarding AOMs, this pattern, coupled with STRATEGY (described in 2.9.1.4), is used to allow users to express complex restrictions on the values of properties present in the system. Fig. 2.6 shows how this pattern is usually applied to AOM architectures, as described in [Fer10].

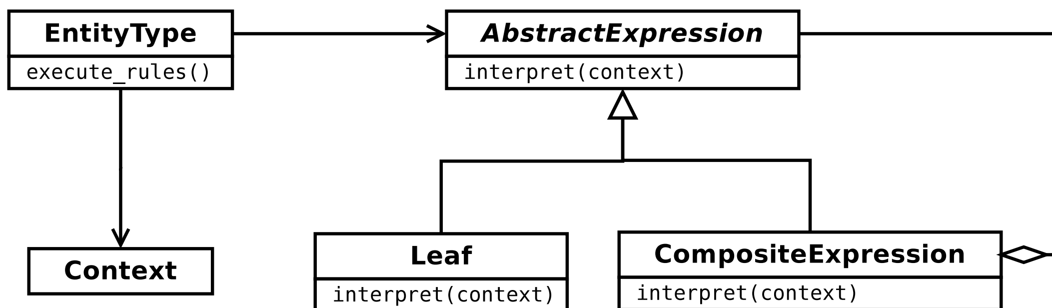


Figure 2.6: INTERPRETER pattern, as applied to an AOM architecture, adapted from [GHJV94].

2.9.1.6 SYSTEM MEMENTO Pattern

The SYSTEM MEMENTO Pattern is used when one wishes to preserve the different states a system has achieved upon its evolution. The usage of this pattern allows the decoupling of the state from the objects themselves and promotes reusability as the same versioning mechanism can be applied to any entity present in the system. This pattern usually states

that an entity in a system (henceforth referred to as a *Thing*) is a composition of *States* (as shown on Fig. 2.7), which, when timestamped, work as evolutionary line for each of the *Things* it is related with. Finally, a *Version* is a collection of all of the *States* in a system, effectively creating a snapshot of the whole system at a given point in time.

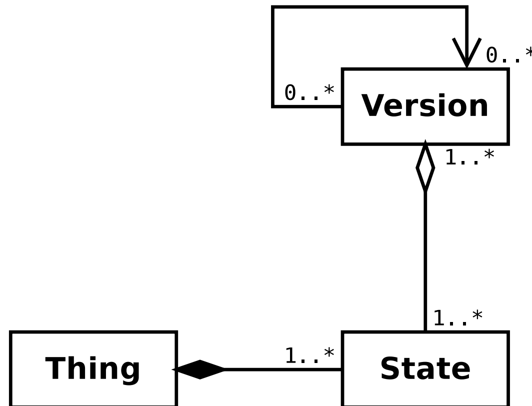


Figure 2.7: SYSTEM MEMENTO pattern, adapted from [FCW08].

2.9.1.7 Relationships Between Entities

Yoder *et al.* [YJ02] describes the relationships between AOM classes by using the ACCOUNTABILITY pattern [Fow97, Hay98]. The ACCOUNTABILITY pattern is used when there is a need to express multiple types of relationships between *parties* (regardless of what their types may be), all of which carry a different meaning [Fowa]. This pattern is shown on Figure 2.8.

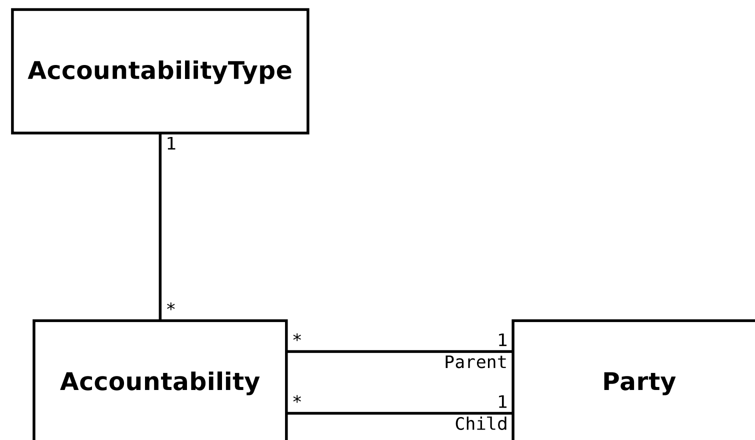


Figure 2.8: ACCOUNTABILITY pattern, adapted from [Fowa].

Most OOP languages, however, describe object attributes as either primitive values or references to other objects. Some languages, such as Ruby and Smalltalk, treat everything as an object and do not make any difference between references and primitive values. These

concepts can be used to extend the PROPERTY pattern, and make it aware of relationships between entities, using attributes such as cardinality, navigability or role [FCAF10].

The revised PROPERTY pattern is depicted in Figure 2.9:

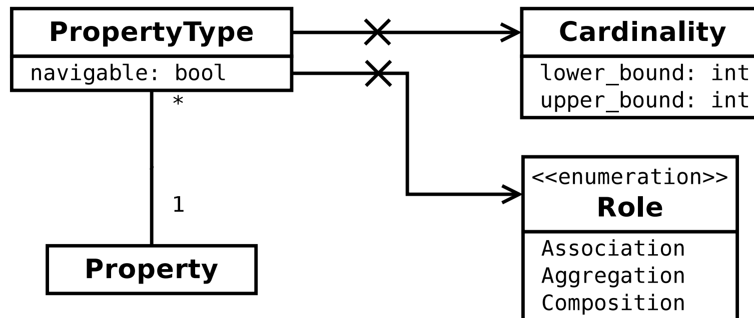


Figure 2.9: Revised PROPERTY pattern, adapted from [FCAF10].

2.9.2 Pattern Composition

By themselves, the design patterns described before do not make up an AOM architecture. Mixing up patterns does not provide any concrete implementation of this kind of architectures, nor it points to what a framework for AOM would look like — which means that the usual architecture of an AOM is usually the result of composing one or more of the aforementioned patterns in conjunction with other object-oriented patterns. The result of this conjunction can be seen in Fig. 2.10

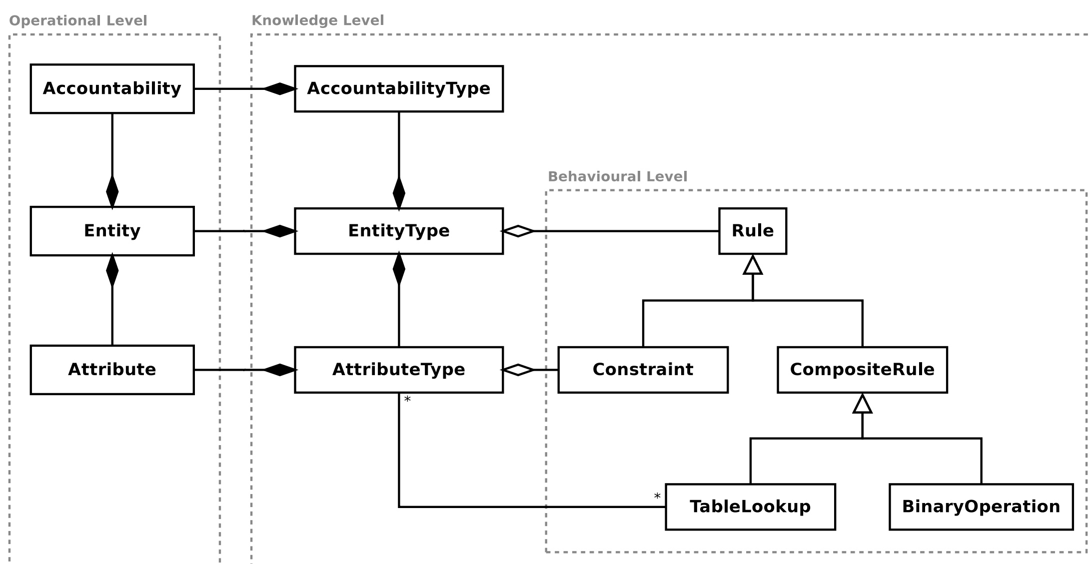


Figure 2.10: AOM core architecture and design, adapted from [YBJ01].

2.9.3 AOM GUI Generation

A system is only as powerful as the interaction with that system allows you to manipulate it — meaning that a poor interface (be it a GUI or an API or any other kind of interface) does not afford a full-fledged use of the system in question. As an example, this may happen when a software library has a number of functionalities implemented but the API exposed to manipulate does not use said implementations, rendering them useless and limiting its usage.

Regarding AOM systems, as the system is interpreted and modified in runtime, the GUI must also be automatically built in runtime from the interpretation of the M1 level model. This implies a standardized approach to rendering entities and entity properties, in order to minimize code redundancy and provide a consistent look & feel. The main problem identified by Welicki et al.[WYWB07] was the redundancy that arose from the need to render each type of property, leading to a higher degree of maintenance and potencial UI inconsistency. The solution devised was to create rendering objects responsible for rendering the user interface for each type of property within a given context, creating what is know as the PROPERTY RENDERER pattern, which enforces a strong separation between domain entities and respective visualization.

However, the separate rendering of single properties is not enough to capture the complexity of an entity. There needs to be a coordination of the various PROPERTY RENDERERS in order to produce a more complex output (be it a UI fragment or a complete interface). The solution is to build view components which coordinate the presentation of several property renderers of an entity to produce different complex UI fragments. Each property renderer is specialized to generate UI code for instances of a property type in a certain context (viewing, editing, different visualization formats, etc). A view component will coordinate several fine-grained renderers and produce more complex UI code for an entity. As such, the same basic concept from PROPERTY RENDERER can be applied to entities rendering, creating the ENTITY VIEW pattern [WYWB07].

2.9.4 Oghma

Oghma is a reference framework for the development of AOM systems, developed in C#. It allows the rapid creation of highly variable, dynamic systems. It is currently being developed in the context of the doctoral dissertation of Hugo Ferreira [Fer10]. It is a very complete framework able to create, manage and persist AOM systems, from backend to GUI generation, by making extensive use of metaprogramming and metamodeling techniques, by following the general guidelines of the NAKED OBJECTS architectural pattern [PW03].

Oghma is thus a concrete implementation of a framework based on the reference architecture to develop AOM-based systems established in [Fer10], that balances several design

and engineering forces. It supports the creation of models resembling MOF [OMG06] and UML [OMG10], and aims at covering the entire cycle of system creation and evolution. As an AOM, it allows the introduction of changes to the system during runtime, thus providing a particular kind of confined end-user development. Fig. 2.11 show the core architecture for this framework.

It is structured as a collection of LAYERED COMPONENT LIBRARY [BMR+96b], allowing their high-level composition to achieve different functional architectures, e.g. client-server v.s. single-process.

Furthermore, the framework leverages the infrastructure used to support system evolution to provide additional features, such as auditing over the system's usage, and time-traveling to an arbitrary point along its evolution (i.e. to set the system in a past state).

Oghma includes a set of interchangeable components designed to have an high degree of flexibility, as it was designed to support several types of persistency engines — be it in memory or a DBMS.

Finally it adds the ability to specify a wide set of behavioural and validation rules by using the INTERPRETER pattern[GHJV94], presenting to the end-user as a DSL, either through a textual syntax or a graphical interface representative of the rules to be enforced.

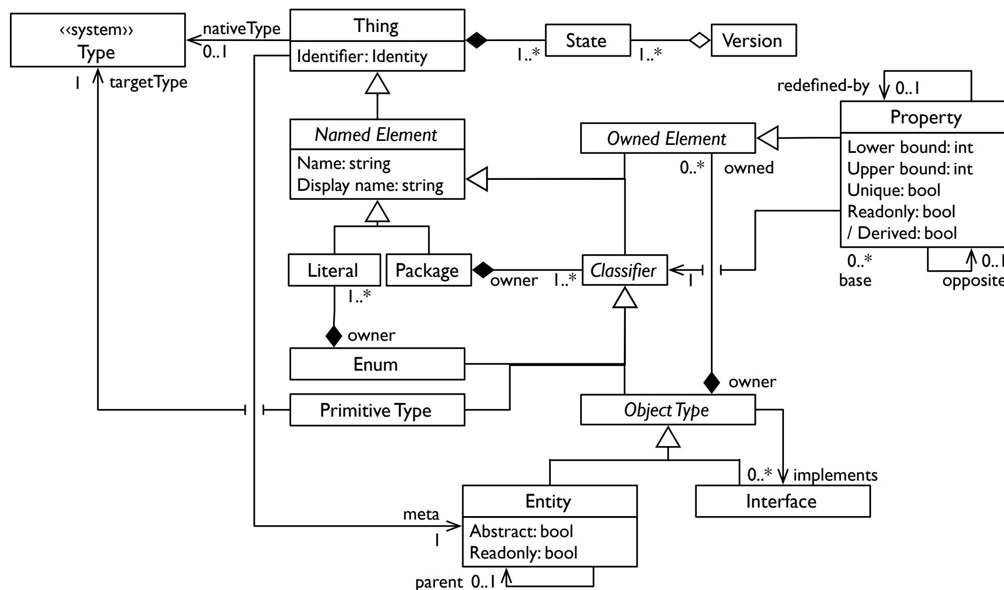


Figure 2.11: Implementation model of the structural meta-model for the Oghma framework, adapted from [Fer10].

2.10 Ruby

Ruby is a dynamic, purely object-oriented programming language. It was created by Japanese programmer Yukihiro Matsumoto and it was first released to public in 1995. In its author words, Ruby was created to be a “a dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write” [Mat]. It was inspired by many different languages such as Lisp, Smalltalk, Perl and Ada, and possesses a series of characteristics that make it extremely attractive [Mat].

The Ruby language was designed with a meta-architecture in mind: it allows for changes to class definitions in runtime, constantly adapting to change. It does so by providing open, active class definitions. A common part of the development process when writing a Ruby program is to extend the language by extending the core language classes (such as `String` and `Float`) with custom methods [Tho07].

2.10.1 Ruby Metaprogramming

Ruby uses metaprogramming techniques extensively — in fact, metaprogramming is an integral part of some of the language constructs. Take the example from Listing 2.1

```
1 class Person
2   attr_accessor :name, :age
3 end
```

Listing 2.1: Metaprogramming in Ruby classes.

The use of the `attr_accessor` declaration is actually a shortcut method for a *getter* and *setter* for the `name` and `age` attributes of the `Person` class. It does so by automatically creating the necessary methods inside a `class_eval` context — effectively modifying the class definition by evaluating code in runtime. Listing 2.2 represents the code generated internally by the Ruby interpreter when interpreting Listing 2.1:

The usage of the metaprogramming facilities present in Ruby is important in the context of AOM architectures: as a dynamic language, it is able to manipulate and generate code in runtime. This special property of Ruby (and other dynamic languages, such as Python) allows the simplification of many of the patterns described before, especially PROPERTY — by using the capabilities of the language, these patterns are absorbed by the language itself, becoming a normal part of development.

```ruby
class Person
  def name=(val)
    @name = val
  end
  def name
    @name
  end

  def age=(val)
    @age = val
  end
  def age
    @age
  end
end
```

Listing 2.2: Code generated by Ruby metaprogramming constructs.

2.10.2 Ruby On Rails

Ruby on Rails is a full-stack Web framework, initially developed by Hansson in 2003, based on the MVC design pattern. As stated by [Han]:

> "Ruby on Rails is an open-source that's optimized for programmers happiness and sustainable productivity. It lets you write beautiful code by favoring convention over configuration."

In regards to code generation, the Ruby on Rails framework includes a series of mechanisms for system artifacts generation, be it Models, Controllers or even Views. It does so by analyzing the underlying relational database model and deriving the model specifications from the column's type and name. However, RoR does not generate a static model definition, as it deduces the necessary information whenever the system is loaded. Instead, it uses these informations to create an adequate code skeleton for basic CRUD operations in views, greatly accelerating the development process by providing the developers with a basic blueprint of a fully functional system that can be refined and tailored to specific needs [Che08].

2.11 Conclusions

The Adaptive Object-Model architectural style can be defined as a metamodeling, object-oriented approach to build software architectures that, by relying on a series of techniques, have the property of being reflective and specially tailored to allow end-users to make modifications to the domain. A series of patterns that make this possible were described, as

well as their connection to the Ruby language, on top of which the *escolinhas.pt* platform is built.

Chapter 3

Problem Statement

In this chapter, the problem will be described and justified, with reference to the bibliographic review presented in Chapter 2. Section 3.1 includes a more detailed description of the study and its objectives. Section 3.2 the approach to be taken in order to try and solve the described problem. Finally, Section 3.3 presents the case-study areas that will be the focus of the study described in this document and why they were chosen.

3.1 Problem Description

As stated by Hanson, Ruby on Rails is built to maximize the developers productivity [Han]. It does so by providing a comprehensive set of tools that perform most of the work. The RoR framework is also designed for easy system evolution through *migrations* — which allow developers to easily add, remove and modify tables and table rows, while minimizing the effort of maintaining application consistency. While this is a solid, proven approach to improve an application's variability over time, it is often less flexible than one might wish, often leading to complicated data migration tasks that may involve modifying production data while ensuring consistency — this can pose as a problem when the amount of data is extensive and highly variable in nature. As such, the use of a static database schema coupled with migrations may not be entirely desirable for applications highly variable in nature.

As presented on Chapter 2, the usage and application of adequate design patterns is capable of mitigating some of these issues, improving both an application's variability and maintainability.

The main concerns of this dissertation are how these architectural and design patterns can be effectively applied to a somewhat (architecturally speaking) restrictive framework, and how these techniques can be combined with a schema-oriented MVC architecture used by the Ruby on Rails Framework with a relational database (MySQL) as the persistence

layer. One other point this work is concerned with is how can the variability of such systems increase and how effective they are in terms of development and performance.

The work will be developed using the *escolinhas.pt* platform as a real and industrial case study.

3.2 Architectural & Design Patterns

The evolution of architectural patterns is outside the scope of this project, mainly because the project that will serve as a base to the case studies herein described is tied to the Ruby on Rails framework and its implementation of the MVC architecture for software systems. Trying to tackle the problem by changing the underlying architecture would mean that the application would mostly have to be rebuilt from scratch, with the team having to stop development to rewrite the application and learning a whole new set of skills, which is simply not feasible. This process is usually done incrementally and opportunistically, as needs and opportunities appear. Moreover, financial issues would arise from the fact that the development team would have to be trained in whichever new platform would be used.

As the modification of the underlying architecture of *escolinhas.pt* has been discarded, the next step is to try to modify only *parts* of the application design, focusing on a number of different areas — also known as hotspots — chosen for their unique requirements in terms of variability. Having chosen these areas, it is important to define which AOM-related design patterns should be applied to solve their problems, and why.

3.3 Case-study Areas

In order to identify the most adequate areas to focus in, a study of the design of the whole system was necessary — this allowed the identification of the areas that had naturally occurring highly-variable requirements. For this study, three hotspots within the application were chosen.

3.3.1 Roles

In *escolinhas.pt*, a user is associated with a number of different roles. These roles allow to users to perform many different tasks and to have access to a number of distinct sections of the platform — meaning they are used as the credentials to access and interact with the application. These roles are implemented using a variant of the ORGANIZATION HIERARCHY as described by Martin Fowler in [Fowa]. This implementation leads to a restrictive design, making the task of creating new roles or access rules unnecessarily complicated.

3.3.2 Social network and contacts

Escolinhas.pt is a social platform for children aged 6 to 10 years old. As a social platform, the network generated by the users is a big part of the application. At this time, this network is built dynamically upon request, by analyzing the relations between users. These relations are derived from the user's school, groups, and friendships. However dynamic, this method proves too restrictive, as there is no convenient and correct way to introduce exceptions in the network — the only way would be to pollute the code with hard-coded rules. As such, the need for a more flexible solution arises.

3.3.3 Document Editor

The document editor provided in escolinhas.pt is one of the most used components of the platform. It produces documents that, in their simplest form, have a title and a series of orderable blocks that serve as a placeholder for many different types of content (text, images, drawing, maps, etc). These documents are able to maintain a history of the modifications, in order to audit changes to its content, and to be able to publish a specific version of the document to the platform. While the editor was built with expansibility in mind, the process of creating a new type of block is not as streamlined as one may wish. On top of that, the versioning system implemented is very tightly coupled with the ACTIVERECORD (AR) implementation provided by the Ruby on Rails framework, which makes the logic for versioning unnecessarily complicated. This makes the document editor a very interesting area for this kind of study.

3.4 Conclusions

In this section, the problem was described and the main objectives laid out, as well as the approach to be taken. The next chapter will focus on choosing the appropriate design patterns that solve each one of the problems, as well as the impact each solution had on the platform.

Chapter 4

Approach & Results

This chapter presents the main body of work resultant of the study performed for this dissertation, and it is divided in two main parts. The first one is related to research design and it starts by describing how the current design of the *escolinhas.pt* platform was performed and which tools were used to perform the analysis, and how the variability needs for each one of the areas described in 3.3 were identified. The second part describes the development process of each one of the case-study areas: Roles, Social Network and Document Editor. For each one of these areas, it will present the current design and variability requirements. Based on these informations, candidate patterns will be presented, as well as the chosen patterns used to solve the problem and the rationale behind it, outlining why certain patterns were discarded. Then, implementation details will follow, concluding each section with an analysis of the impact each solution had, as well as some results (where applicable).

4.1 Research Design

The first step taken in order to identify potential problems was to study its current design. This step led to the identification of several points that negatively impacted the variability of the application. After the identification of these points, variability requirements were gathered, so that small, functional prototypes could be built and their impact analyzed within the platform.

4.1.1 Current Design Analysis

A thorough analysis of the current design of the application was the first step taken, in order to identify potential problems within the platform. This study was conducted using a series of tools[1] that, unfortunately (at the time of writing), proved unsuccessful in extracting an *Entity Relationship Diagram* from the code of the application — either they did not work

[1] Rubymine 2.0 [Jet] and Rails ERD [Tim]

with the current versions of Ruby/Rails used by the platform (Rails ERD) or generated an inaccurate graph (Rubymine 2.0). As such, another approach was taken: the product owner of the platform was queried on what he felt were the variability "hotspots" of the application: this narrowed the scope of the current design analysis, allowing to focus on three different areas, as stated in 3.3. These areas were then manually studied — by analyzing the current database schema, as well as the Model (MVC) source files — in order to extract the *Entity Relationship Diagrams* relative to each one — shown ahead on 4.2.1, 4.3 and 4.4 — which allowed analyzing each one as independently as possible, so that the applied design patterns (if any) would emerge — making the task of pinpointing exactly what was wrong with each approach much easier.

4.1.2 Variability Analysis

Each one of the areas — Roles, Social Network and the Document Editor — referred in 3.3 was then studied in order to determine their variability requirements, and to which degree this variability should exist: developer, system administrator, or even end-user. These requirements were defined by the carefully considering the opinions from the users of the platform, the product owner and the main developers. These requirements were then used to determine which design patterns would be more appropriate to achieve the problems posed by each of the areas, and why.

4.1.3 Implementation & Impact Analysis

For each one the three focus areas, functional prototypes were developed, either directly on top of the platform (Social Network, 4.3, and User Roles, 4.2) or as an independent proof-of-concept application (Document Editor, 4.4), that, as soon as possible, will be integrated into the main platform. These prototypes, however simplistic (and not yet in production), will be able to validate the research and decisions made throughout the next sections. All of the prototypes are implemented using Ruby 1.9.1 and Rails 2.3.9, as used in the production and development environments of *escolinhas.pt*.

The impact analysis for each of these areas will be made in two different fronts: variability and performance. Variability impact analysis will be concerned with the increase in flexibility/configurability of a certain component, and how it measures to the objectives established beforehand. Performance analysis will (if applicable) examine the how much more (or less) performant each one of the implementations is, compared to the previous solutions.

After studying the current design of the application and collecting the necessary requirements, the development phase took place: the major variability requirements were outlined, and a set of design patterns carefully considered, with their benefits and disadvantages weighed in order to correctly choose the most appropriate solution. Finally,

proof-of-concept prototypes were built so that the choice of patterns could be validated and the impact resultant from the implementations measured.

4.2 User Roles

Roles play a very important part in any application: by attaching them to users, they allow the application to authorize users based on their roles. If an application has a strong tendency to evolve, so do its roles and authorization sets. This section describes the work involved in making the current system of roles used in *escolinhas.pt* as adaptable as possible.

4.2.1 Variability Requirements

Authorization is one of the most sensitive areas of any closed software system: it ensures everyone does only what it should in order to guarantee everything works as expected. In a constantly evolving system, the accesses granted by user roles have a tendency to shift and evolve alongside the application — either because of new features or a new type of user is required in the system to perform specific tasks. In the context of *escolinhas.pt* this problem ties itself with the ACL used: because of the diversity of roles (Fig. 4.1) and the three different usage plans, a lot of different rules are applied to determine if a user can or can not perform certain actions; allied to the growing number of features of the platform, this means that the authorization scheme has to be as flexible as possible to ensure minimal overhead when determining new types of permission sets.

4.2.2 Candidate Patterns

The current logic on roles and users states that a user may be a professor, a student, a parent, a coordinator (in which case it also has a professor role associated), or an administrator. Of these five different types of roles, only two of them are allowed editing privileges, which means that only students and professors have to ability to create and edit documents, which leads to an unnecessary level of complexity. If, for example, it was necessary to have a parent with editing privileges, a new Role, descendant of Role::Editor, would have to be created just for that user.

An obvious solution to this problem would be to tie a "traditional" ACCESS CONTROL LIST (as described in [MMN02]) to the roles actually in use: this would allow to fine-tune each one of the users permissions and authorization sets while maintaining the codebase clean — however, the logic surrounding authorization schemas and user roles is built around the *CanCan* Ruby gem [Bat], which authorizes an user based on his or her roles, while keeping the necessary logic to a minimum.

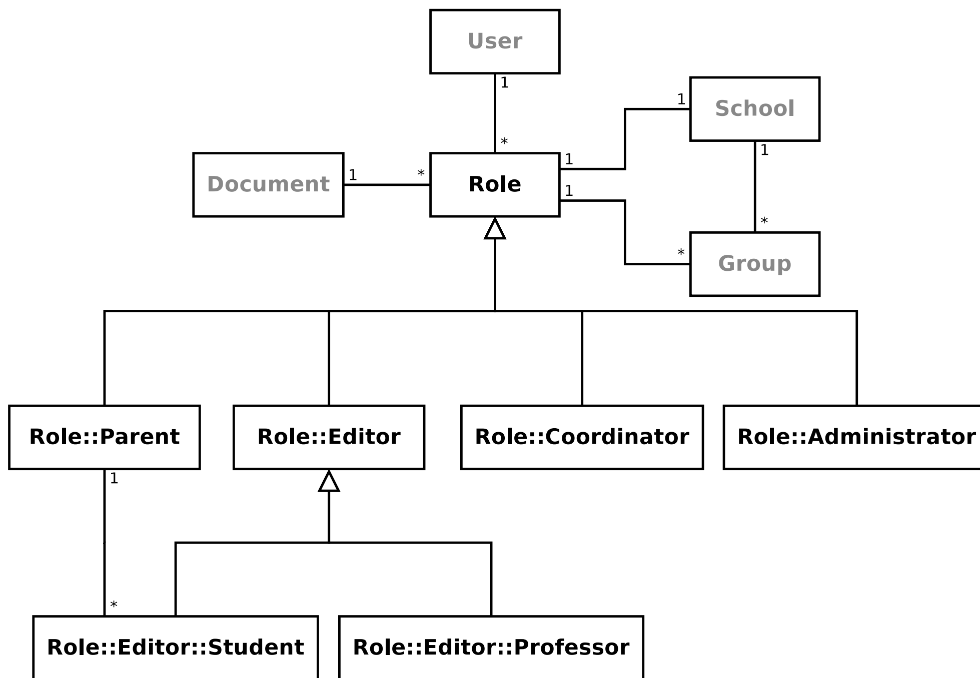


Figure 4.1: Current User Roles Model.

As such, the usage of a new and fully customized ACL is unnecessary. As *CanCan* rules are written in Ruby and are based on the ACTIVERECORD (AR) engine used by Rails, *CanCan* is capable of handling authorizations based either on Models (MVC) or *instances* of these Models. The application of an ACL to define user permissions would allow a fine-grained control over the actions of every individual — instances of *User* Model — on the system. However, the cases when this kind of control is necessary are rare, which would mean that the increase in complexity brought with the usage of an ACL would not be surpassed by its usefulness: it would be necessary to rewrite every rule already defined within *CanCan* and then associate each user on the system with a specific set of rules, instead of maintaining the current setup and writing a few (rare) exceptions for the users the system administrator saw fit.

4.2.3 Chosen Patterns & Rationale

As the implementation of a new ACL was discarded in 4.2.2, the best solution is to enhance the already present roles system: a flat Role hierarchy, as described in [BRSW97] would allow for a more flexible authorization scheme, where a User could have one or more roles associated, depending on what actions he would be allowed to do, as shown on Fig. 4.2. This also makes the task of creating new roles with different authorization schemes much easier, as there is not a need to conform to any special hierarchy scheme: a new role simply means a new type of user. This clearly contrasts with the previous role's logic, where a multi-level hierarchy was in place.

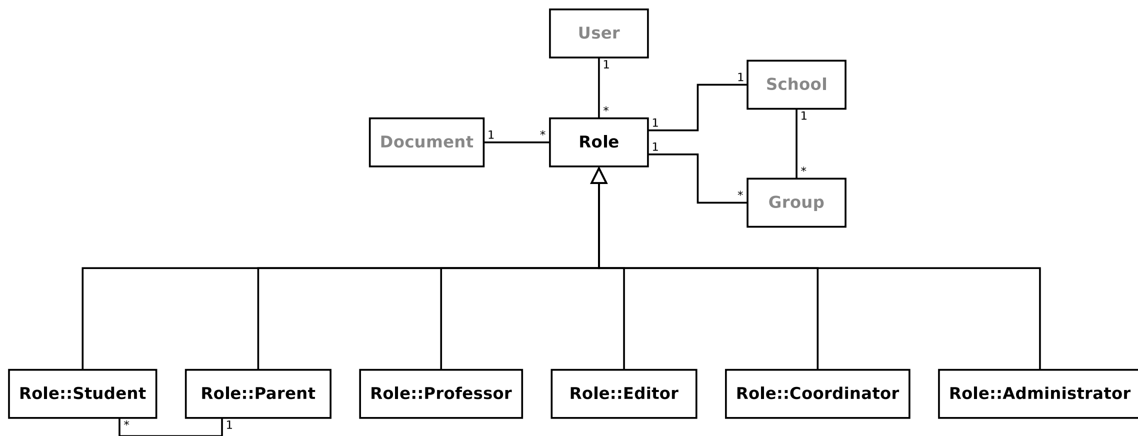


Figure 4.2: Conceptual User Roles Model.

4.2.4 Implementation

The refactoring of the roles infrastructure was of very low impact, as codebase and database schema are regarded. The hierarchy of Role classes was flattened, keeping it at only two levels: a generic, non-instantiable class *Role*, and all of its currently existent subclasses: *Editor, Parent, Student, Professor, Coordinator* and *Administrator*. Then, the existent rules were adapted to fit this new hierarchy. The last steps pertains to the modification of the current data to fit the new role organization, which entails analyzing each user current roles and performing the adequate substitutions from the previous roles schema.

4.2.5 Impact Analysis

The main issues related to the current Roles schema and consequent authorization strategies are caused by the difficulty to capture the constantly evolving necessities of different types of users. As the platform evolves, so do its users and their associated roles. If a new feature is added to the application, it is necessary to define the privileges each user type has over it. The flattening of the Roles hierarchy allows the representation of these Roles as separate, independent objects, allowing the different contexts they refer to be kept separate and also simplify the system configuration.

The usage of a *matricial* ACL implementation, as discussed in 4.2.2 would simplify the configuration regarding the privileges of specific users — allowing a system administrator to have full control over them. Ultimately, this means that the roles would play a very small part in authorization granting, serving only as pre-defined rule sets to be applied to new users.

The usage of a *declarative* ACL (*CanCan*) in conjunction with the ROLE OBJECT pattern, leads to some important consequences: despite losing the ability to *easily* control the specific set of rules of each user[2] and increasing the difficulty of maintaining constraints

between roles, it allows the independent evolution of each *Role*, while making the task of defining their key abstractions regarding each role's position within the platform much more simple and concise.

Comparing the solution described in 4.2.3 with the previous system, some conclusions can be drawn: by eliminating complex hierarchies and keeping all roles on the same level, *Roles* are easily evolved, as each *Role* is independent from every other one. However, this solution must be be used with some caution, as maintaining constraints between roles becomes difficult: since a *User* rule set consists of several *Roles*, maintaining constraints and preserving the overall subject consistency might become difficult due to conflicting rules. Nevertheless, the ability to easily add new *Roles* and write the appropriate rules (as the example from Listing 4.1 shows) is enough to warrant the use of this solution. Finally, the rules currently in place for the *escolinhas.pt* platform are a mix of rules based on *Roles* and actions that can performed on objects: this makes defining new *Roles* and rule sets somewhat more complicated than defining them based solely based on the application *Roles*. The usage of the ROLE OBJECT pattern, combined with a rewrite of the current rules will make the act of creating new *Roles* much more affordable.

```
1   class Ability
2     include CanCan::Ability
3
4     if user.is_admin?     # set of rules for users with 'Admin' role associated
5       can :manage, :all
6     end
7
8     if user.is_guest?     # set of rules for users with no roles associated
9       can :read, [Post, Comment]
10    end
11  end
```

Listing 4.1: CanCan rule definition example.

4.3 Social Network

As a social platform, *escolinhas.pt* makes heavy use of the social network created by its users. Due to the nature of the application, which closely mimics the Portuguese primary school system, there is a need to control how the users (especially the students and parents) interact with each other; in other words, the application needs to have mechanisms that allow the manipulation of the social network. This section describes the study and work

[2]This ability is not completely lost. If necessary, *CanCan* can be used to define authorizations based on a User instance (a specific user)

involved in creating such mechanisms, as well as implementation details and its impact details, both in terms of variability and performance.

4.3.1 Variability Requirements

The current design for the escolinhas.pt social network is based on the relationships formed through the connections between users and their schools, groups, and other users, as depicted in Fig. 4.3. The User's Roles play an important part in the definition of this network, as an user is only linked directly to other users; any other connection is mediated through their Roles, creating their association with Schools and Groups. This allows the construction of a network where all the connections can be inferred dynamically and where an user can be identified to another through these same connections: friends, classmates, parent–child, teacher–student, and so on. This network is used mainly in the internal communication systems of *escolinhas.pt*, which allows users to communicate with each other within the platform.

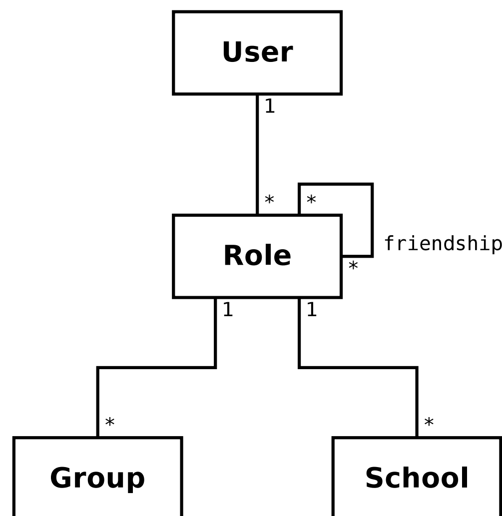
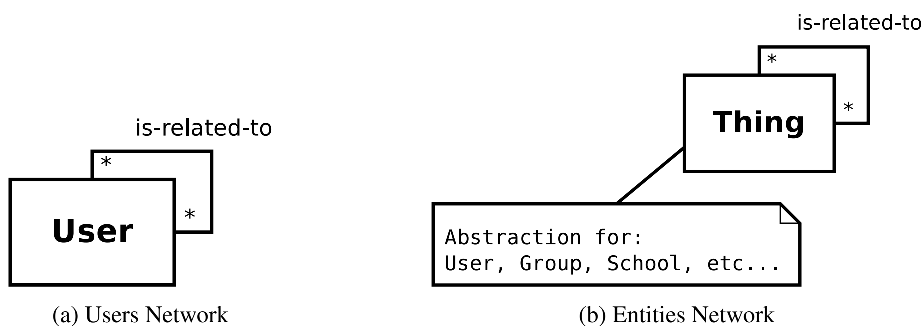


Figure 4.3: Current User Network Model.

This model, however useful, offers a very small degree of variability. Due to the closed nature of the platform, there is a need to provide mechanisms able to fine-tune these connections in order to accommodate to each school specific needs — some Schools may not want to appear in search results; there may be some Students or even Parents who need to have special communication privileges. These mechanisms need to be available at the system administrator level, in order to easily manipulate these links without the need to pollute the application's codebase with hard-coded rules and without the need for redeployement.

4.3.2 Candidate Patterns

Ideally, the user network would be described with a simple, self-referencing model, as shown in figure 4.4a. This would allow the creation of static relationships between any two users that could be edited as needed. This would work great if all that was needed was to create relationships between users. However, it is often necessary to create connections between users and other entities in the system, such as groups and schools. Thus, this simple model needs to be abstracted in order to connect any two entities present in the system, whichever they may be, as shown in Fig. 4.4b.



(a) Users Network

(b) Entities Network

Figure 4.4: Simplified Network Models.

4.3.3 Chosen Patterns & Rationale

The presented solutions are inspired by Martin Fowler's work regarding Organization Structures, and based on the ORGANIZATION HIERARCHY design pattern. Despite solving the majority of the problem, the solutions described in 4.3.2 are less than ideal, as they do not allow the identification of an user towards another, because only a direct connection between two different entities is contemplated.

This could be solved by introducing an associative class containing these informations — however, the resulting structure would still not convey enough information to properly represent the existing hierarchies, as a simple relationship between two entities (as shown in Fig. 4.4) cannot identify who is the superior and the subordinate of the relationship.

As such, for this particular problem, it is necessary to be able to connect any two entities in the system, identify their place in the hierarchy (parent or child), with an optional third entity to serve as hint as to how the original entities are connected. This problem can be solved by using the ACCOUNTABILITY pattern (see 2.9.1.7) by Martin Fowler [Fowa]: it allows a bi-directional, hierarchical relationship between two entities (also known as *parties*) while maintaining an AccountabilityType which can be used to store additional data about the connection. As such, this AccountabilityType can be used to store an optional third party, responsible for identifying how the two other parties are connected —

effectively granting means to identify an user before an other, which is part of the original problem formulation (4.3).

4.3.4 Implementation

A variant of the ACCOUNTABILITY design pattern (as described in section 2.9.1.7) was chosen (shown in Fig. 4.5). This implementation follows the original description of the pattern by using all the usual entities present in the original ACCOUNTABILITY pattern [Fowa] — however, it denormalizes the AccountabilityType entity *into* the Accountabilities themselves, by placing the AccountabilityType attributes (`type`, `through`, `school_year`, `active`) in the Accountability. Despite creating some data redundancy, this option provides a more performant implementation: as the Accountabilities table is to be constantly accessed, the decision to have the AccountabilityTypes in a separate table would lead to expensive `JOIN` operations. This, in turn, would lead to a less than desirable performance and complexity.

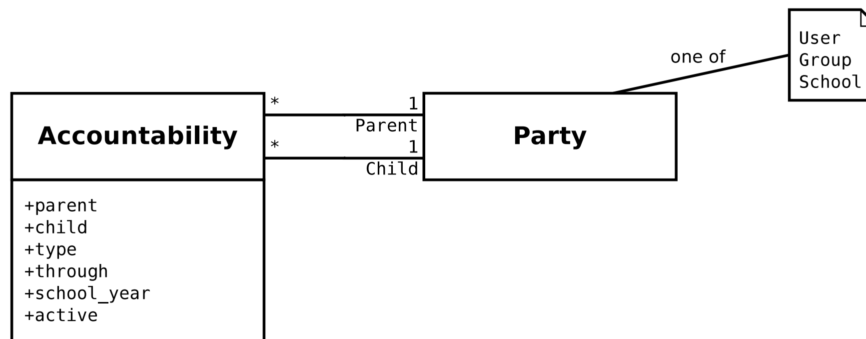


Figure 4.5: Accountability Implementation for User Network.

For performance reasons (explained in 4.3.5), a series of different AccountabilityTypes were created, in order to cater to a multitude of relationship types, as described in Table 4.1

Type	Parent	Child	Through	Description
group_professor	Group	Professor	—	establishes a connection between a *Group* and a *Professor*, meaning that the user is one of the teachers of *Group*
professor	Professor	Student	Group	establishes a connection between two *Users* — a *Professor* and a *Student* — creating a teacher-student relationship between them through whichever *Group* they are related to

group_student	Group	Student	—	establishes a connection between a *Group* and a *Student*, meaning that the user is part of the *Group* and taught by the **group_professors** associated with the aforementioned *Group*
school_professor	School	Professor	—	establishes a connection between a *School* and a *Professor*
school_student	School	Student	—	establishes a connection between a *School* and a *Student*
parent	Parent	Student	—	establishes a parenthood relationship between two users
school_cordinator	School	Coordinator	—	dictates an *User* is a coordinator (also known as an administrator) of a certain *School*
colleague	Professor	Coordinator	School	establishes a connection between two *Users* — either a *Coordinator* or a *Professor* — through a *School* they both work in
student	Coordinator	Student	School	establishes a relationship between a *Coordinator* and a *Student* through a *School*
school_parent	Coordinator	Parent	School	establishes a relationship between a *Coordinator* and a *Parent* through a *School*
friend	User	User	—	establishes a connection between any two *Users* of the system — whichever their roles may be — to indicate a friendship relation exists between them

Table 4.1: AccountabilityTypes created to describe every type of existent relation.

Some of the aforementioned AccountabilityTypes are representative of every type of interpersonal relationship existent in the escolinhas.pt platform. At first sight, some of the AccountabilityTypes created may seem redundant, such as student and school_parent: they exist because the school coordinator needs to be able to contact everyone who is part of the school. One could argue these connections could easily be inferred through the relations between the coordinator and his or her school, and the relations existent between the school and its students, and finally use the existent parenthood relationships. However, as described in 4.3.1, one of the major design flaws (regarding variability), was the completely dynamic nature of the contacts network — as the network was always

built upon request, there was no viable way of modifying it without using hardcoded rules or a configuration setup external to the code. Thus, the choice to implement apparently redundant AccountabilityTypes tied itself with the necessity to have full control over the existent social relationships. The remainder of the AccountabilityTypes are used to store and facilitate access to membership-like relationships, by stating a certain user is part of a school or group at a given school year. This also allows the platform to keep a history of past (inactive) relationships between entities in the system.

4.3.5 Impact Analysis

The usage of this design pattern not only solved some of the existing variability and performance problems, but introduced a new possibility: the ability to create relationships between any two entities in the system. This leads to a very flexible network, capable of being modified at the M0 (data) level (see 2.9.1), which is a pre-requisite for end-user level variability.

A second objective pertaining to the application of this pattern was to improve the performance related to contact list creation and the identification of these before the user. This task is currently extremely expensive, with an edge case of 5724 queries needed to fetch and identify 715 contacts. A user with only 18 contacts generates 154 queries. This means that an average of 8 queries are performed for each one of the contacts, meaning the cost of this operation is approximately linear in nature, as depicted in Fig. 4.6.

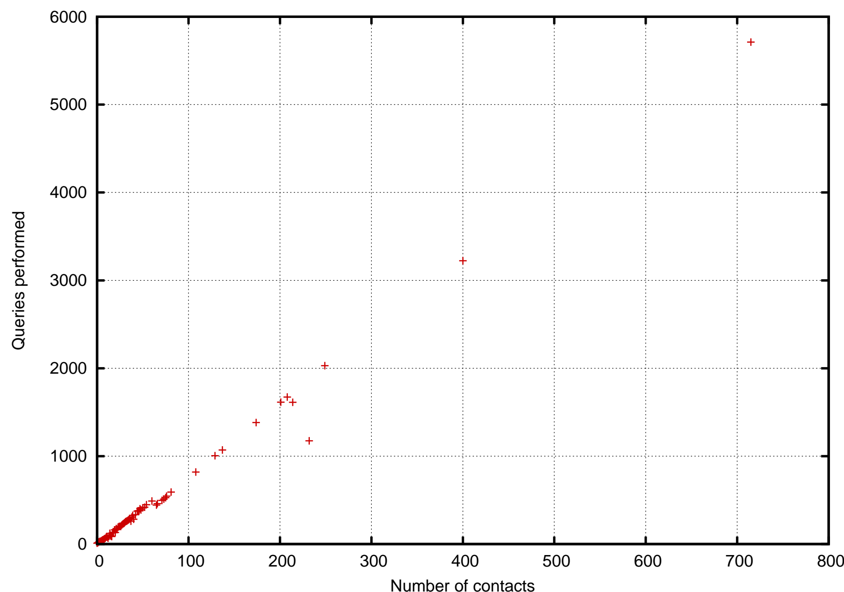


Figure 4.6: Average number of queries performed per number of contacts.

The graph in Fig. 4.6 represents the average number of queries performed per number of contacts a user has, and it was sampled from a random population of 10000 real users of the platform. As stated before cost growth of the function is only approximately linear: due to the dynamic nature of the network, some users may have a sparser network — e.g. less groups associated with, but more users associated with each group the user is part of — which can explain the unexpected decrease in the number of queries around the 200-mark and the irregularities in users with less than 100 contacts. However, in practice, this cost can be extrapolated to a linear cost, to a point where one can infer that the number of queries performed is approximately 8 times the number of contacts, which represents a very serious performance issue for one of the most used features of the platform.

The implementation of the ACCOUNTABILITY pattern to maintain the relationships between users was able to reduce the cost of the abovementioned task to from $O(n)$ (Fig. 4.6) $O(1)$: by using the RoR ACTIVERECORD API to access and manipulate these connections, only 11 queries are performed to fetch and identify an user contacts, regardless of the size of said contact list. This means that the platform is able to sustain a considerable growth without suffering serious impacts on the performance of seemingly trivial operations.

4.4 Documents

The last studied area — the Document Editor — is also the most used and most complex component of the system. It is the main means of collaboration between students and teachers, and used to create essays, using a series of different digital contents, such as images, videos, or maps. As one of the most used features of the system, it is also the one most subject to modification. This section details the work performed in order to make the process of modifying the Document Editor as agile as possible.

4.4.1 Variability Requirements

The document editor present in *escolinhas.pt* is one of the core components of the system and one of the most used features of the platform. This being the case, and due to the constantly evolving nature of the product and the product itself, it is also one of the most modified parts of the system. As it can be seen in Fig. 4.7, this structure has to grow both in size and complexity every time a new type of *Block* content is introduced — represented by the gray entities in Fig. 4.7. This means that whenever a new type of content is introduced in the system, which happens somewhat frequently — from three types of *Blocks* (*Paragraphs*, *Drawings* and *ImageDocuments/Photos*) in September 2009 to seven in April/May 2010 — it is necessary to setup a new ACTIVERECORD class (along with all the logic for versioning) and a new CONTROLLER (MVC) to accept the requests necessary to create, edit or delete any of these entities. Despite working as intended, this

workflow is not adequate to the constant evolution and prototyping the document editor is subjected to.

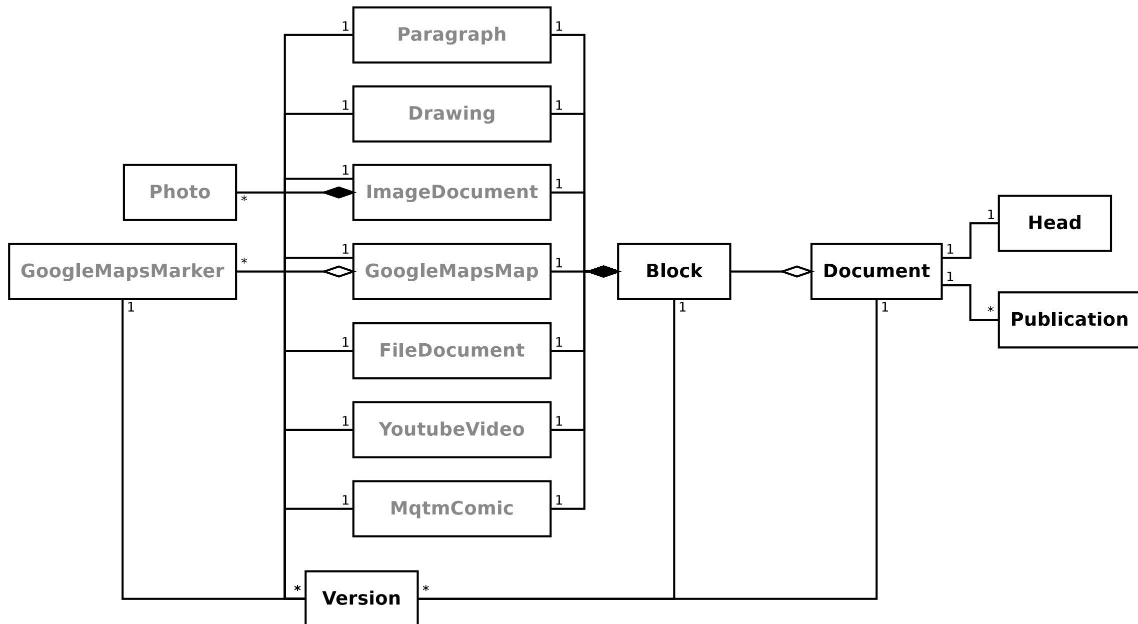


Figure 4.7: Current Documents Model.

4.4.2 Candidate Patterns

One of the major causes of slow and irksome development of new types of *Blocks*, is, as stated in the previous section, the sheer amount of code duplication involved. This involves painstakingly copying the Model, the Controller and all the Views from a previous existing *Block* type, and then modifying the required parts. This approach is brittle and extremely prone to errors. However, if a more general and abstract analysis is performed, a *Block* type is nothing more than a very complex variable. As such, the problem can now be approached to as finding the best way to create new types of variables, on demand, and with as little work involved as possible — and the current research on AOM architectures has already solved this problem, by means of the TYPE-OBJECT and PROPERTY (and, consequently, the TYPE-SQUARE, described in 2.9.1.1, 2.9.1.2 and 2.9.1.3, respectively) patterns.

The research on versioning and maintaining a temporal history for objects is extensive. In [FCW08], Hugo Ferreira *et al.* describe the most appropriate patterns pertaining to data and metadatada evolution within the context of AOM architectures. The most important one is SYSTEM MEMENTO, as described in 2.9.1.6. Martin Fowler has also compiled a collection of patterns that aim to deal with temporal changes to an object or system state, in [Fowb]. Two of the most promising patterns described are the TEMPORAL OBJECT [Fowd] and SNAPSHOT [Fowc]. However, neither one of the aforementioned

patterns is completely adequate to solve the problem at hand. On one hand, TEMPORAL OBJECT assumes the creation of a dedicated entity just to be able to access a certain object at a given point in time — as one of the objectives of the refactoring of this hotspot is to reduce the associated codebase and work necessary to create new types of *Blocks*, the usage of this pattern must be discarded. On the other hand, the usage of SNAPSHOT assumes that the Snapshot itself is to be used as an adapter on the underlying object, which once again assumes the usage of a separate object to handle recording the state of any given entity in a particular point in time.

One important part of the document editor is the validation of user input. As it is the only focus area which deals directly with free-form user input, the validation of the values need to be taken into account. The STRATEGY and INTERPRETER pattern (described in 2.9.1.4 and 2.9.1.5, respectively), are two patterns that would be adequate to write and perform all of the validations required by the *Block* `item` logic. However, these two patterns are mainly directed to when a system needs to have end-user level variability. In this particular case, the definition and creation of new types of *Blocks* is a task for the development team. As such, the usage of these patterns is unnecessary as far as validations are regarded. In order to simplify development and validations themselves, Ruby will act as both the *language* and *execution* engine for the validations. This is done by using a custom, extensible DSL built atop of the language itself; thus, the STRATEGY and INTERPRETER pattern are effectively "absorbed" by the host language (Ruby) atop of which the platform is built.

4.4.3 Chosen Patterns & Rationale

The solution devised for the *Documents* infrastructure is, as required by all the necessities stated in 4.4.1 and 4.4.2, the simplest structure possible — as seen in Fig. 4.8 — that makes everything work as before and makes the data transition tasks as simple and seamless as possible. This is due to the metaprogramming facilities of the Ruby language and the excellent support present in the serialization and ACTIVERECORD tools provided by RoR.

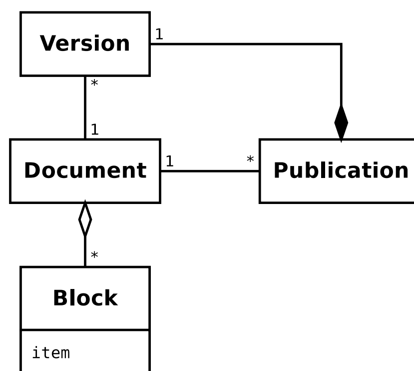


Figure 4.8: Conceptual Documents Model.

The pattern used is a *composite* design pattern as described in [Rie97], where various smaller design patterns work in tandem to create a more complex pattern:

- SYSTEM MEMENTO - used for versioning and handling *Publications* as special states
- PROPERTY (simplified variant) - used for decoupling a *Block* `item` from the database schema

The variant of the PROPERTY pattern implemented is simplified due to the highly dynamic nature of the Ruby language — which means that, for this particular problem, it is able to build new types of objects or even create new class definitions in runtime, which allows discarding the PROPERTY–PROPERTYTYPE pair (see 2.9.1.2) in favour of a single entity, `item`, as shown on Fig. 4.8.

4.4.4 Implementation

The implementation of this composite pattern takes advantage of the highly dynamic nature of the Ruby language and the API provided by the ACTIVERECORD implementation of Rails.

4.4.4.1 Basic Structure

Document, *Block*, *Version* and *Publication* are all AR objects, which means that, according to the AR pattern and the Rails framework conventions, each one of them is stored inside an SQL table, with a column for each one of the attributes. This structure provides the basic blueprint (as stated in 3.3.3) for the documents to be produced by the editor — it allows a title, an arbitrary number of orderable *Blocks*, and a snapshot (version) of each modification. It also allows for publications, which essentially point to a specific version of a document.

There is nothing really remarkable about *Blocks*, *Documents* of even *Publications* — they are ordinary ACTIVERECORD objects, with associations to each other (as pictured in Fig. 4.8), and explaining how they work is outside of the scope of this study.

4.4.4.2 Versioning & Block Items

However, a *Version* is a bit more complex than a simple AR object, in the sense that it contains a full representation of another AR object at a given point in time — in this case, a *Document*. This is achieved by serializing a *Document* and all its associations (*Blocks*) in the JSON format, which preserves all the necessary information needed to rebuild a specific *Document* at whichever time that *Version* refers to — which means that a *Version* effectively implements the MEMENTO design pattern to keep a history of each *Document*.

Finally, a *Block* `item` possesses special properties that, together with AR, create a dynamic and variable foundation for the development of different types of content. As a *Block* is simply a generic container for an arbitrary type of item, a *Block* `item` can't be constrained to a single class or object type. The solution is to serialize the content inside the *content* attribute of a *Block*. This way, a *Block* `item` is simply a string that represents a serialized object — which can be de-serialized, accessed and modified at runtime. This means that, whichever a *Block* `item` may be, the `item` itself is responsible for its representation and life cycle.

4.4.4.3 Validation and Defaulting Mechanisms for Block Items

In order to further simplify and streamline the development, a *DocumentItem* (super)class was created. This class serves as a basis for further specialization through inheritance, and handles cross-cutting concerns such as object initialization, default values and validations for each of these attribute's values. The need for a specific controller for each one of the different *Block* `items` has also been discarded in favour of a single Controller (MVC), responsible for handling the user input regarding the modification of *Blocks* and their `item` contents.

The Document Editor prototype is the one that takes most advantage of the Ruby language, by extensively using metaprogramming techniques, along with JSON [Cro] serialization and parsing.

The defaulting and validation mechanisms implemented for the *DocumentItem* class make use of the metaprogramming capabilities of the Ruby language to easily create *class-level instance variables* [Nun06] that can be shared across all the instances of each subclass of *DocumentItem*.

4.4.4.4 JSON Serialization

JSON is exclusively used for serialization, and its usage objectives are two-fold: allowing serialization and de-serialization of any Ruby object, and the ability to use a serialized object directly in Javascript, without any need for a server-side transformation. JSON usage also allows for a more robust de-serialization of objects: as a JSON object is just a hash of key-value pairs, objects are built from a class method, *DocumentItem.from_json*, which takes a JSON-serialized object represented as a native Ruby hash, and creates an object of the appropriate class. The difference of this approach from other serialization methods[3] is that the constructor of the class to be built is called manually[4], which is vital to deal with system evolution. If at any point during the development the definition of a *DocumentItem* specialization is modified (i.e. a new attribute is added to the class

[3] Marshalling, YAML, etc.

[4] Marshalling and YAML mechanisms do not call the serialized object's constructor, only rebuild it

definition), the previous objects stored will not have these new attributes. The defaulting mechanism implemented solves this problem by merging the existing default values with the ones passed to the class constructor, thereby creating a valid object, regardless of the information stored in the serialized object.

4.4.5 Impact Analysis

The refactoring of the Document Editor infrastructure had two major points of impact: performance, and variability.

With the current foundation for the editor, the number of queries grows linearly with the number of *Blocks* that constitute a document, as it is necessary to perform a query for each one the items related to each one of the *Blocks*. The usage of eager loading is limited due to the polymorphic nature of the *Blocks* and each respective content, which is unknown *a priori*.

From an universe of 4000 documents used in the study, the graph present in Fig. 4.9 shows a linear growth in the number of queries necessary to render a Document: the number of queries necessary are directly proportional to the number of *Blocks* in a document with a 1:1 ratio. Just like in 4.3.5, this represents a serious performance issue: as the most used feature in the *escolinhas.pt* platform, a sustainable growth is very difficult to achieve if the database load increases linearly with the number of existent *Blocks* in Documents.

The introduction of the model described in 4.4.3 makes the number of queries necessary to display a *Document* to be constant (only 2 queries are performed), as only the *Blocks* and the Document itself are AR objects — as the item that constitutes a *Block* is an integral part of a *Block*, no queries are necessary to fetch it.

Using the aforementioned infrastructure for the document editor, the work necessary to create new types of *Blocks* has been greatly reduced. This allows for much shorter prototyping and testing iteration times — no database schemas or migrations to worry about, allowing the developers to focus on the details of the model rather than implementation details — which ultimately leads to a higher degree of variability.

Listings 4.2 and 4.3 show how the work necessary to create a *Block* `item` has been greatly reduced, while keeping the necessary logic to a minimum. The logic for versioning is now handled by the `item` *Block* instead of the `item` itself, which allows its logic and structure to be succinctly described.

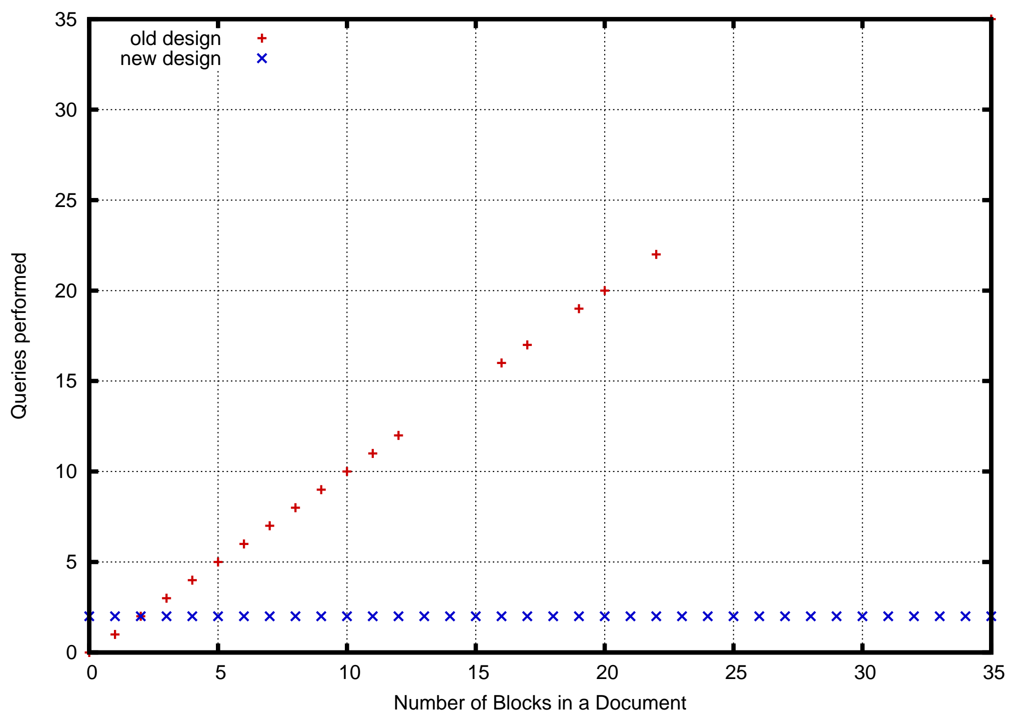


Figure 4.9: Average number of queries performed per number of *Blocks* in a single Document.

```
1  class GoogleMapsMap < ActiveRecord::Base
2
3    acts_as_paranoid_manually_versioned
4    has_one :block, :as : :item
5
6    has_many :google_maps_markers, :dependent : :destroy
7
8    # shortcut methods
9    def markers
10     self.google_maps_markers
11   end
12
13   # save markers associations
14   def before_save
15     self.markers_for_version = self.markers.map(&:id).to_yaml
16   end
17
18   def before_update
19     self.block.document.head.version += 1
20     self.block.document.head.save
21
22     self.set_new_version(self.block.document.head.version)
23     self.save_version(self.block.document.head.version)
24   end
25 end
```

Listing 4.2: Original “Map” Block item.

```
1  class DocumentItem::Map < DocumentItem
2
3    attr_accessor :lat, :lng, :map_type, :markers
4
5    defaults :lat,      0.0
6    defaults :lng,      0.0
7    defaults :map_type, 'roadmap'
8    defaults :markers,  []
9
10 end
```

Listing 4.3: Refactored “Map” Block item.

4.5 Conclusions

This chapter detailed how the study of the current design of the platform was conducted, and the tools used to do so. It described how the data gathered from that study was used to identify what the main problems within the platform were, and how they could be solved. It also described a vast set of applicable design patterns and how this set was reduced to choose the patterns that best solved the problem at hand. Finally, this chapter details the impact regarding the usage of each chosen pattern in terms of variability and performance.

It was shown that the usage of the appropriate design patterns solved the problems that were identified in section 3.3: for each one the the areas described in 4.2, 4.3 and 4.4, the problem was analyzed, and from a set of possible solutions, the most appropriate one was chosen. Small, focused, proof-of-concept prototypes were built to verify the validity of the chosen solutions — which lead to an overall increase in variability for each one of the aforementioned areas. Along with an increase in variability, performance was also increased in two areas — the Social Network and Documents — although this increase was not always a direct objective: regarding the Social Network, performance was a serious factor for choosing the appropriate solution, while for the Documents it was simply a side-effect that arose from good design.

Chapter 5

Conclusion

The problem described in chapter 3 has been solved according to the solution presented in chapter 4 and the results achieved from this approach have been presented in the same chapter. In this last chapter, Section 5.1, some last remarks will be made about this study, as well as a summary of the main achievements and results. In Section 5.2 the further improvements to the project will be presented.

5.1 Conclusions

Adaptive Object-Model architectures provide the best framework for building adaptable systems that are liable of modification by the end-users (which assumes no compilation or deployment processes). A lot of thought and research has gone into the best practices for the complete implementation of these types of systems, from model creation, maintenance and persistence to GUI generation.

Despite being built with a MVC architecture, the Ruby On Rails framework is, using the right approach, capable of working with some architectural and design patterns — present in AOM architectures — not obviously connected with MVC and AR. The application of these patterns is capable of increasing the level of variability present in a common Rails application, and a harmonious integration with the Rails 2.3.x infrastructure — especially the ACTIVERECORD engine — is possible and works elegantly.

Although the majority of the work present in this dissertation is concerned with increasing the variability of software systems built on top of the Ruby on Rails framework, it is possible to conclude that the application of the appropriate design patterns is able to increase not only the variability and configurability needs of a specific part of an application, but also its performance. However, this should not be taken as an universal truth, as it depends on a series of factors that may not be present in all implementations, such as a previous innapropriate design.

5.2 Further Work

The main improvements and additions that can be made to the final work presented are divided into 2 groups: implementation and optimizations.

Regarding implementation, each one of the prototypes herein described are in *pre-alpha* state: they provide working prototypes that, despite functional, are not yet ready for production. Further work is needed to integrate them within the platform so that these prototypes can be successfully put to good use.

In regards to optimizations, perhaps one of the most important optimizations to be performed deals with the Social Network and the Accountabilities created: while it is a relatively inexpensive process, it must be taken into account that the task of fetching an user's contacts and identifying each one still requires 11 queries to the database. For a platform with a substantial amount of active users, this could pose as a problem if this number suddenly begins to rise. As such, there are two possible optimizations for this area: trying to reduce the number of queries performed by using raw SQL instead of the RoR ActiveRecord API; another possible solution would be to keep all the accountabilities *in memory*, in order to reduce the heavy usage of the database server. This could be achieved by using a *memcached*[1] server to create an *in-memory* cache of the desired database contents.

[1]http://www.memcached.org/

References

[AdOdSBD07] Nicolas Anquetil, Káthia M. de Oliveira, Kleiber D. de Sousa, and Márcio G. Batista Dias. Software maintenance seen as a knowledge management issue. *Inf. Softw. Technol.*, 49(5):515–529, 2007.

[AIS+77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, New York, 1977.

[Bas87] P. G. Bassett. Frame-based software engineering. *IEEE Softw.*, 4(4):9–16, 1987.

[Bat] Ryan Bates. Cancan. https://github.com/ryanb/cancan.

[BMR+96a] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, Chichester, UK, 1996.

[BMR+96b] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture volume 1: A system of patterns*. Wiley, August 1996.

[BRSW97] D. Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. The role object pattern. In *Proceedings of PLoP*, volume 97, pages 97–34, 1997.

[Che08] Frederick Cheung. Migrations. http://guides.rubyonrails.org/migrations.html#creating-a-migration, September 2008. [Online] Accessed 16th July 2010.

[CI84] Robert D. Cameron and M. Robert Ito. Grammar-Based Definition of Metaprogramming Systems. *ACM Transactions on Programming Languages and Systems*, 6(1):20–54, January 1984.

[Cle88] J. Craig Cleaveland. Building application generators. *IEEE Softw.*, 5(4):25–33, 1988.

[Cro] Douglas Crockford. The application/json media type for javascript object notation (json).

[DH04] Ivo Damyanov and Nick Holmes. Metadata driven code generation using .net framework. In *CompSysTech '04: Proceedings of the 5th international conference on Computer systems and technologies*, pages 1–6, New York, NY, USA, 2004. ACM.

[EGHN] Eric Evans, Vladimir Gitlevich, Ying Hu, and Jimmy Nilsson. What is domain-driven design? | domain-driven design community. http://www.domaindrivendesign.org/resources/what_is_ddd. [Online] Accessed 10 January 2011.

[Eva03] Eric Evans. *Domain-driven design: Tackling complexity in the heart of software*. Addison-Wesley Professional, August 2003.

[FCAF10] Hugo Sereno Ferreira, Filipe Figueiredo Correia, Ademar Aguiar, and João Pascoal Faria. Adaptive Object-Models: a Research Roadmap. 2010.

[FCW08] Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Leon Welicki. Patterns for data and metadata evolution in adaptive object-models. In *Proceedings of the 15th Conference on Pattern Languages of Programs*, PLoP '08, pages 5:1–5:9, New York, NY, USA, 2008. ACM.

[Fer10] Hugo Sereno Ferreira. *Adaptive Object-Modelling: Patterns, Tools and Applications*. PhD thesis, Faculdade de Engenharia da Universidade do Porto, December 2010. To be published.

[Fowa] Martin Fowler. *Organization Structures*.

[Fowb] Martin Fowler. Patterns for things that change with time. http://martinfowler.com/ap2/timeNarrative.html. [Online] Accessed January 12 2011.

[Fowc] Martin Fowler. Snapshot. http://martinfowler.com/ap2/snapshot.html. [Online] Accessed January 12 2011.

[Fowd] Martin Fowler. Temporal object. http://martinfowler.com/ap2/temporalObject.html. [Online] Accessed January 12 2011.

[Fow97] Martin Fowler. *Analysis Patterns, Reusable Object Models*. Addison-Wesley, 1997.

[FY97] B. Foote and J. Yoder. *Pattern Languages of Program Designs*. Addison-Wesley, 1997.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design patterns:Elements of reusable object-oriented software*. Addison-Wesley Professional, November 1994.

[Han] David Heinemeier Hansson. Ruby on rails. http://rubyonrails.org/. [Online] Accessed 16th July 2010.

[Hay98] David Hay. *Data Model Patterns, Conventions of Thought*. Dorset House Publishing, 1998.

[Jet] Jetbrains. Rubymine. http://www.jetbrains.com/ruby/.

[JY97] Brian Foote Joseph and Joseph Yoder. Big ball of mud. In *Pattern Languages of Program Design*, pages 653–692. Addison-Wesley, 1997.

[KM05] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, New York, NY, USA, 2005. ACM.

[LC03] Donal Lafferty and Vinny Cahill. Language-independent aspect-oriented programming. *Proceedings of the 18th annual ACM SIGPLAN*, 38(11):1, November 2003.

[Mat] Yukihiro Matsumoto. Ruby programming language. http://ruby-lang.org. Accessed 16th July 2010.

[Mic] Microsoft. Xna. http://www.xna.com/. Accessed July 16, 2010.

[Mil03] J. Miller. MDA Guide Version 1.0. 1. *Object Management Group*, (June), 2003.

[MMN02] C.J. McCollum, J.R. Messing, and L. Notargiacomo. Beyond the pale of MAC and DAC-defining new forms of access control. In *Research in Security and Privacy, 1990. Proceedings., 1990 IEEE Computer Society Symposium on*, pages 190–200. IEEE, 2002.

[MS03] David Mertz and Michele Simionato. Metaclass programming in python. http://www.ibm.com/developerworks/linux/library/l-pymeta.html, 2003. [Online] Accessed July 16 2010.

[Nun06] John Nunemaker. Class and instance variables in ruby // railstips by john nunemaker. http://railstips.org/blog/archives/2006/11/18/class-and-instance-variables-in-ruby/, November 2006. [Online] Accessed January 12 2011.

[OMG06] OMG. Meta Object Facility (MOF) Core Specification. *Citeseer*, (January), 2006.

[OMG10] OMG. Omg unified modeling language (omg uml), infrastructure. March 2010.

[ORP] Tim O'Reilly, Sam Ruby, and Bruce Perens. Ruby on rails: Quotes. http://rubyonrails.org/quotes. [Online] Accessed 14th January 2011.

[PT07] Carla Pacheco and Edmundo Tovar. Stakeholder identification as an issue in the improvement of software requirements quality. In *CAiSE'07: Proceedings of the 19th international conference on Advanced information systems engineering*, pages 370–380, Berlin, Heidelberg, 2007. Springer-Verlag.

[PW03] Richard Pawson and Vincent Wade. Agile development using naked objects. In Michele Marchesi and Giancarlo Succi, editors, *Extreme Programming and Agile Processes in Software Engineering*, volume 2675 of *Lecture Notes in Computer Science*, pages 1010–1010. Springer Berlin / Heidelberg, 2003.

[Rie97] Dirk Riehle. Composite design patterns. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '97, pages 218–228, New York, NY, USA, 1997. ACM.

[Sch06] D.C. Schmidt. Model-driven engineering. *IEEE computer*, 39(2):25–31, 2006.

[Ste06] Friedrich Steimann. The paradoxical success of aspect-oriented programming. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 481–497, New York, NY, USA, 2006. ACM.

[TC] Lda. Tecla Colorida. Escolinhas :: Plataforma colaborativa e social para o eb1/2. http://www.escolinhas.pt/. Accessed July 8th 2010.

[TC06] Kun Tian and Kendra Cooper. Agile and software product line methods: are they so different. *1st International Workshop on Agile Product Line*, 2006.

[Tho07] Dave Thomas. Metaprogamming - extending ruby for fun and profit, December 2007. [Presentation, Online]. Available: http://www.infoq.com/presentations/metaprogramming-ruby.

[Tim] Rolf Timmermans. Rails erd. http://rails-erd.rubyforge.org/.

[WC03] Laurie Williams and Alistair Cockburn. Guest editors' introduction: Agile software development: It's about feedback and change. *Computer*, 36(6):39–43, 2003.

[WYWB07] León Welicki, Joseph W. Yoder, and Rebecca Wirfs-Brock. Rendering patterns for adaptive object-models. In *PLOP '07: Proceedings of the 14th Conference on Pattern Languages of Programs*, pages 1–12, New York, NY, USA, 2007. ACM.

[YBJ01] Joseph W. Yoder, Federico Balaguer, and Ralph Johnson. Architecture and design of adaptive object-models. *SIGPLAN Not.*, 36(12):50–60, 2001.

[YFRT98] Joseph W. Yoder, Brian Foote, Dirk Riehle, and Michel Tilman. Metadata and active object-models. In *Addendum to the 1998 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages –A22, New York, NY, USA, 1998. ACM.

[YJ02] Joseph W. Yoder and Ralph E. Johnson. The adaptive object-model architectural style. In *WICSA 3: Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture*, pages 3–27, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.