A modular model-driven engineering approach to reduce efforts in software development teams

Jhon Alexander Cruz Castelblanco
A modular model-driven engineering approach to reduce efforts in software development teams

Jhon Alexander Cruz Castelblanco

A thesis submitted in partial fulfillment of the requirements for the degree of:
Master in Systems Engineering and Computer Science

Advised by:
MSc. Henry Roberto Umaña Acosta

Research Line:
Software Engineering

Research Group:
Colectivo de Investigación en Ingeniería de Software (ColSWe)

Universidad Nacional de Colombia
Engineering Faculty, Department of Systems and Industrial Engineering
Bogotá, Colombia
2014
Dedicated to my parents and my girlfriend
Acknowledgements

I would like to thank to all the people who was involved and brought me support during the development of this work. To the ColSWe research group, especially to Henry Umaña, associate professor of the National University of Colombia, for his important advice and supervision of this work. I also thank to all my friends and partners of Seven4N LTDA and Skaphe S.A.S, because without them the technical support and the application of related work in the colombian industry would not be possible. I own them most of my knowledge and expertise in this field.

Finally, but not least important I really appreciate the support and patience of my family, my friends and specially my girlfriend Alejandra. Without them, finishing this master’s thesis would be hardly possible.
Abstract

Model-driven development (MDD) raises as one of the promising paradigms to solve classical software development problems like bad estimation, low productivity and bad quality. The main goal of this work is to create a Model-driven software development tool called MoDev to reduce efforts in the process of developing applications. This tool focuses mainly on significant improvements in terms of productivity and quality, allowing developers to model applications and generate a meaningful portion of code. MoDev requires predefined software architectures as input and it does not only serves to improve some issues related to software development, but it was also implemented to be easily extended and reusable. Aimed to achieve that, the tool is created following a proper methodology for this approach. By means of a case study, the promises of the MDD were confirmed elucidating important improvements in terms of productivity, quality, maintainability and flexibility. However, some minor issues related to cultural challenges, the need of people with expertise in language engineering and training times were detected as well.

Keywords: model-driven software development, domain specific languages, software engineering, code generation, software architecture.

Resumen

El desarrollo de software basado en modelos (MDSD por sus siglas en inglés) surge como uno de los paradigmas promisorios para resolver problemas clásicos del desarrollo de software como mala estimación, baja productividad y mala calidad. El objetivo principal de este trabajo es crear una herramienta de MDSD llamada MoDev para reducir esfuerzos en el proceso de desarrollo de aplicaciones. Esta herramienta se enfoca principalmente en mejoras significativas en términos de productividad y calidad, permitiendo a los desarrolladores modelar aplicaciones y generar automáticamente una parte importante del código. MoDev requiere como entrada arquitecturas de software predefinidas y no solo sirve para ayudar resolver los problemas mencionados sino que puede ser reutilizada y extendida de forma sencilla. Para lograr esto, la herramienta es creada siguiendo una metodología apropiada. Finalmente y por medio de un caso de estudio, las promesas de MDSD son confirmadas mostrando importantes mejoras en términos de productividad, calidad, mantenibilidad y flexibilidad. Sin embargo, algunas desventajas menores fueron detectadas como los nuevos retos culturales a ser adoptados, la necesidad de personal experto en ingeniería de lenguajes y tiempos adicionales para aprendizaje.

Palabras clave: desarrollo de software basado en modelos, lenguajes de dominio específico, ingeniería de software, generación de código, arquitectura de software.
# Contents

1. **Acknowledgments** ................................................................. vii
2. **Abstract** ........................................................................... ix
3. **Introduction** ....................................................................... 2
   1.1. Goals .................................................................................. 3
   1.2. Outline of this work ........................................................... 4
4. **Context and related work** ..................................................... 5
   2.1. Motivation .......................................................................... 5
   2.2. Model-Driven Software Development .................................... 6
       2.2.1. MDSD methodology ..................................................... 8
   2.3. Domain-Specific Languages ................................................. 11
   2.4. MDD benefits and issues .................................................... 14
   2.5. Related Work ..................................................................... 16
       2.5.1. Yeoman ........................................................................ 18
       2.5.2. GenMyModel ............................................................... 19
       2.5.3. Sculptor ....................................................................... 20
   2.6. Alternatives ........................................................................ 22
5. **MoDev** ............................................................................... 23
   3.1. Requirements ...................................................................... 23
   3.2. The Software Architecture .................................................. 25
       3.2.1. Layers ......................................................................... 26
       3.2.2. Technologies ............................................................... 26
       3.2.3. Rules ........................................................................... 30
   3.3. Implementing MoDev ........................................................... 31
       3.3.1. Technologies ............................................................... 31
       3.3.2. Main concepts ............................................................. 33
       3.3.3. Reference Implementation and Analysis ......................... 35
       3.3.4. Meta-models ............................................................... 36
       3.3.5. DSLs ........................................................................... 40
       3.3.6. Generators ................................................................... 42
3.3.7. Editors .................................................. 43
3.3.8. Supporting re-generations with protected regions ................. 45
3.4. A development process with MoDev .................................. 45
3.5. Facing new software architectures .................................... 46

4. Case study and evaluation .............................................. 48
   4.1. Case study: The authentication and authorization system .......... 48
       4.1.1. Development of the case study ................................ 50
       4.1.2. Evaluation .................................................. 52

5. Conclusions .......................................................... 59
   5.0.3. Future work .................................................. 61

A. Appendix: Meta-models developed for the software architecture .... 62

B. Appendix: DSL instances created for the case study ................. 66

. Bibliography ............................................................ 69
List of Figures

2-1. Relationship between MDE, MDD and MDA 6
2-2. Domain Analysis and architecture development as a basis to formal modeling 7
2-3. The followed methodology as an iterative process for developing MDSD tools 9
2-4. Scaffolding web applications with the Yeoman command-line interface. 19
2-5. A screenshot of the GenMyModel application showing some UML diagrams. 20
2-6. A sample blog application modeled with Sculptor using textual representations. 21
3-1. Layers for the software architecture. 27
3-2. Example of artefact organization for applications using the architecture. 32
3-3. General concepts about MoDev and its building 34
3-4. Overview of the models developed for this architecture 39
3-5. An example of reusability in MoDev 40
3-6. The document DSL. An example of the DSL that conforms to the `document` meta-model. 41
3-7. The route template implemented in Xtend 2 42
3-8. An extract of the document template implemented in Xtend 2 43
3-9. An example of some editor’s features like syntax colouring, validation and content-assist. 44
3-10. Suggested development process with MoDev 47
4-1. Developing with the traditional approach 52
4-2. Example of generated source code documentation 55
4-3. Example of generated documentation for REST endpoints 56
A-1. The document meta-model 62
A-2. The DB-Config meta-model 62
A-3. The Mongo-Config meta-model 63
A-4. The entity meta-model 63
A-5. The repository abstract meta-model 63
A-6. The document-repository meta-model. An extension of the repository meta-model. 64
A-7. The relational-repository meta-model. An extension of the repository meta-model. 64
A-8. The business meta-model. .................................................. 64
A-9. The route meta-model. .................................................... 65
A-10. The dto abstract meta-model. .......................................... 65
B-1. The document DSL instance for the case study .................. 66
B-2. The repository DSL instance for the case study ................ 67
B-3. The business DSL instance for the case study .................. 67
B-4. The layeredApp DSL instance for the case study .............. 68
List of Tables

2-1. ISO 9162 Software Product Quality Characteristics 15
2-2. Problems and alternatives detected in the related work 22
3-1. Quality attributes chosen for the software architecture 25
3-2. Shared technologies between the two groups of platforms 28
3-3. Libraries and frameworks for the Node.js platform 29
3-4. Libraries and frameworks for the Node.js platform 30
3-5. Rules for organizing the artefacts on applications using this architecture 31
3-6. Some generic components detected from the reference implementation 35
3-7. Some repetitive components detected from the reference implementation 36
3-8. Some individual components detected from the reference implementation 36
3-9. Models developed for the architecture 37
4-1. Functional requirements for the first study case: The authentication and authorization system 49
4-2. Some illustrative non-Functional requirements for the first study case: The authentication and authorization system 50
4-3. SonarQube metrics for the first study case 54
5-1. The good and the bad of MoDev 60
1. Introduction

Classical software development problems like bad estimation of times and budgets, low productivity, bad quality, poor communication among stakeholders, bad design and wrong metrics have been tried to be solved by multiple methodologies and techniques. Additionally, the complexity of current technologies and quality expectations have raised significantly in the last years, forcing enterprise to dedicate more resources to implement a software system\[48\][29]. Besides, software architectures are also defined to guide the application development. But sometimes developers create applications without conforming to the architectural rules and decisions because of problems in communication and bad development discipline and practices. Model-driven software development arises as one those promising paradigms, which aims to reduce complexity in the software development process by raising a certain level of abstraction at which developers design software systems and thereby increases productivity and the quality of an application.

Model-Driven Development\(^1\) (MDD) is presented like one of the multiple solution to some of the inherent software development difficulties. MDD focuses on the use of abstract conceptual representations in order to improve productivity, making easier the design and specification process of a software system\[13\]. Time saving is one of the most representative advantages but not the only one. Software quality, cost reduction, reusability, integration, modifiability and better time to market are also the promises of MDD applied in software projects\[19\][32].

Stahl et al. \[52\] proposes a methodology to create model-driven-development tools. This work focuses on it and customizes it as well. At first, a prototype phase comes with the development of proofs of concept using a specific platform within the domain where the MDD tool is intended to be used. If a software architecture does not exist, this is the right point to define it. Then, a reference implementation is created, that is, a sample and minimal application written manually where all elements of the software architecture are taken into account. After that, an analysis is performed over the reference implementation. Generic, repetitive and individual components are identified in order to know what is able to be abstracted and what does not. Later, the meta-models and the domain-specific languages are designed, defining the abstract syntax, the concrete syntax and the corresponding seman-

\(^1\)The often used term model-driven development (MDD) is an specialization of Model-Driven Engineering (MDE). MDD is also known as model-driven software development (MDSD)
tics. Finally, an editor for the languages is implemented which supports features like syntax colouring and highlighting, refactoring, code folding, among others.

Having a defined methodology, a MDD tool called MoDev is created with the main purpose of reducing efforts and improving the quality in software development teams. One of the requirements to create and extend this tool is to have a defined software architecture. That is why, an illustrative software architecture is described and taken as a reference for this entire work. Every step of the methodology applied to MoDev is explained, including the requirements of the tool, the technologies to be used, the reference implementation, metamodels, languages and editors. A very important point to highlight, is that MoDev is not only guided to help developers improving productivity by means of the illustrative software architecture defined in this thesis, but also to be easily extended and reused in any software architecture with components able to be abstracted. This can lead to better improvements and evaluations of MoDev. It also lays the foundations to further research in other interesting fields like cloud computing together with model-driven development.

With this approach, expectations of using MDE in the software industry were confirmed by means of a case study. Results of an evaluation elucidate that there was important improvements in terms of productivity, quality, maintainability, flexibility, testability and portability. Some other minor drawbacks of the tool like cultural challenges and training times were also illustrated.

1.1 Goals

The general purpose of this work is to improve the development process in software teams, turning it into a more productive and flexible process with high-quality and better maintainability. This is accomplished throughout a model-driven development tool called MoDev, which is capable of generating software artefacts based on pre-defined software architectures. The following are the specific goals to achieve in this work:

1. To select an illustrative software architecture which is used to prove the benefits of the tool.

2. To design a collection of suitable meta-models and domain-specific languages for the proposed software architecture.

3. To implement a model-driven development tool in a modular and reusable way, that supports, in principle, an illustrative software architecture. This tool should reduce software development efforts in activities like testing, configuration and coding.
4. To illustrate the advantages and disadvantages of the tool by means of a proof of concept.

1.2. Outline of this work

This work is divided into 5 chapters. Chapter 1 introduces the research work by mentioning what problem is identified the goals of this research and how all goals will be achieved. Chapter 2 describes the state of the art of Model-Driven Software Development (MDSD), explains some basic terminology and describes the methodology used to create a model-driven engineering tool as well. Then, all related research work concerning MDSD, MDSD applied in the industry and some commercial and open-source tools are examined. Chapter 3 explains everything about MoDev, the reusable and modular MDSD tool for reducing efforts in software development teams. Some of the topics explained in this chapter are: main requirements of the tool, the software architecture selected to prove that MoDev really works and how the tool was built. Chapter 4 explores two study cases that fit perfectly to the selected software architecture and where MoDev is used to create fully functional applications. Both cases are studied showing the advantages and disadvantages of involving such a tool in the development process. Chapter 5 finally draws the conclusions of this work and presents some future work and research.
2. Context and related work

2.1. Motivation

A software architecture (SA) is presented as a blueprint for a system’s construction and evolution and provides the highest level of abstraction of a system. This abstraction includes the set of principal design decisions. Those design decisions encompass every facet of the system under development including structure, behaviour, interaction and non-functional requirements. One of the problems of SA is that implementations are commonly decoupled from the architecture rules. Communication problems, bad engineering requirements and the lack of documentation causes developers of a project to give up the SA leading to an implementation that does not conform to the rules and decisions in an architectural model.

There is currently a gap between theory and practice in software architecture. The theory resolves around advanced architecture description and modeling languages (also known as Architecture Description Languages), while in practice, the software architect, in case of he exists, does not have the appropriate tools to create a trustworthy architecture that is useful during the software life cycle. Besides, there are cases where software architecture documentation does not even exists because it requires naturally a lot of time and dedication. When software architectures are defined, they are sometimes presented in power point slides with boxes and arrows, pretending to communicate components, layers and other constructs with no practical value [14].

Obviously SA requires implementation, and this adds some additional problems like the classical software development ones such as bad estimation of times and budgets, low productivity, bad quality, poor communication among stakeholders, bad design and wrong metrics. Additionally, If a developer writes high-quality code, it does not necessarily means that it is translated into a high-quality software architecture with decoupled components and rules governing the structure and communication [39].

The lack of an direct connection between a software architecture and the source code is difficult to overcome and makes it hard to understand how a software system works and where the changes belong. This happens specially for code written by somebody else which is a common scenario in large applications with multiple developers. All the mentioned problems have been tried to be solved by multiple methodologies and techniques. Model-Driven
Software development arises as one of those promising paradigms. This work is guided to help developers to reach better productivity and maintainability levels without sacrificing quality. This quality does not refer only to code quality, but architecture quality respecting all decisions and rules of a pre-defined software architecture.

This chapter explores what the MDD paradigm is about, a methodology based on it and what are the benefits and issues when it is applied in software development and SA. Finally, a list of related work is presented, split up into research work and commercial products.

2.2. Model-Driven Software Development

Model-Driven Software Development (MDSD) or just Model-Driven Development (MDD) focuses on the use of abstract conceptual representations or models aiming to provide improvements in productivity, making easier the design process and simplifying changes to the specifications of a system’s design. It is also referred as Model-Driven Engineering (MDE) and should not be confused with Model-Driven Architecture (MDA) which proposes an implementation of MDD using UML profiles (see Figure 2-1).

![Figure 2-1.: Relationship between MDE, MDD and MDA](image)

MDA is just an specialization of MDD standardized by the Object Management Group (OMG) and is based on widely-used industry standards for visualizing, storing, and exchanging software designs and models. The best-known standard for this is the Unified Modeling Language (UML). MDA heavily emphasizes on creating designs or machine-readable models stored in standardized repositories. MDA models are understood by automatic tools from multiple vendors that generate schemas, code skeletons, test harnesses, integration code and deployment scripts for multiple platforms used in a typical project [35].
Although this work aims to have the same features and benefits, MDA is a very scoped specialization of MDD, whose standard forces the use of the UML standard and its profiles. Avoiding such limitation, this work explores a custom MDD approach which allows creating models in a more convenient way, for example, using textual representations.

The idea under MDD is to use models in different abstraction levels until the most specific one which corresponds to the platform specification (for example Java or .NET). This ensures a high level of automation in the software development field using model transformations based on certain rules organized by transformation languages like QVT (Query/View/Transformation) or Xtend for model to text (M2T) transformations.

To start a development process with MDSD, a separation between domain analysis and domain architecture should be established in order to have a basis for the formal modelling. Formal modelling connects the concepts of the concrete application which comes with the domain architecture. Formal modelling results are then transformed with the help of a generator and finally mapped to a target platform.

![Diagram](image)

Figure 2-2.: Domain Analysis and architecture development as a basis to formal modeling

Some terminology should be highlighted and entitled as this work goes along. The term *domain* or *domain problem* denotes the main point of interest or knowledge where the MDSD process starts. For this thesis the domain problem is software architecture. The term *meta-model* defines formally the structure of a domain. It is mandatory to define a meta-model. A meta-model connects a set of concepts related to each other and within a certain domain. It is the abstraction of a model which highlights the properties of the model itself. For this work, a set of meta-models are defined according to different elements of a software architecture. A *meta-meta model* on its behalf, is another level of abstraction for the meta-model
in order to specify it formally. It is important to note that all levels of abstraction are based on previously formalized concepts. For this work, the *meta-meta model* is defined by the Eclipse Modeling Framework (EMF) with specifications like Ecore and Genmodel (see 3.3.1). Once meta-models are defined, they serve as basis for *domain-specific languages* or DSLs. A DSL is a programming language that is targeted to a specific problem. Details about DSLs are shown in section 2.3. Then, model should be transformed into a *platform*, which serves as support of the realization of the domain, that is, where model transformations come into or where the generated code will run. Such *transformations* are based on the meta-model, implement semantics of the DSL and their result are instances of the same one. Two types of transformations are distinguished; Model to Model (M2M) when a model is created by other model and Model To Platform (M2P) or Model to Text (M2T) transformations when models are finally converted to generated artefacts of the platform.

### 2.2.1. MDSD methodology

As a relative new paradigm in the software development field, MDSD suggests some basics steps in order to build tools that support it. In this work, a methodology proposed by [52] is followed and customized. It is a top-down approach were at first, common elements of the target platform are detected and formalized in a meta-model and then it is used to generate software artefacts like source code using some transformation rules. The process is executed iteratively as additional components are added to the domain. The following sections provide a description of each step indicating the necessary activities and artefacts (see Figure 2-3).

**Prototyping phase**

If one wants to generate artefacts for a specific platform within a determined domain, they both in most cases already exist. Be they Java using a Java Enterprise Edition (JEE) architecture or .NET using a recommended standard by Microsoft, it is always important to acquire some experience in developing with such platform and then proceed to analyse it. Some elements created in this phase are a set of *proofs of concept* related to the platform.

**Reference Implementation phase**

As a result of the previous phase, a reference implementation should be created. This implementation serves as an example of how the transition from model to implementation (M2T) is managed on the respective platform(s). This implementation is generally created manually by platform and architecture experts and concrete functional content is irrelevant in this phase. Then, a generative implementation of the reference model plus the manually programmed domain logic will result in a complete, executable reference implementation.
Figure 2-3.: The followed methodology as an iterative process for developing MDSD tools
Analysis phase

This phase consists of analysing and identifying three components of the reference implementation, being a component an element of the platform or the software architecture like source code or documentation.

- **Generic components**: The components that are very specific to the platform. They are merely elements that cannot be changed and that have to be included to get a fully functional application. For example, in a JEE architecture with a Java platform, components like Enterprise Java Beans (EJBs) classes, Java Server Faces (JSF) pages, and Java Persistent API classes have to be included.

- **Repetitive components**: The code that has always the same structure according to the domain or is even exactly the same in every application. Packages structure, some type of class or interface declaration, certain software design patterns like Factories are some examples.

- **Individual components**: Every application has its particularities. Obviously these particularities cannot be abstracted and generated automatically, but developers must include at the end along with the generated components.

Meta-modelling and DSL design phase

This phase consists of analysing and designing the meta-models and a set of suitable domain-specific languages. The use of UML is typical but not mandatory and design rules for DSLs should be taken into account (see 2.3). With the design of the DSL, designers establish a line between generated code and domain logic, and this the freedom degree of the developer to create applications. In this phase validations to meta-models and DSL are created as well. Thus, the consistency of the model transformations can be guaranteed.

Model transformations building phase

This step formalizes the mapping of a DSL to a platform and programming model without forgetting the domain, to the point that an automatic transformation can convert a given application model into a fully implementation or a big part of it. Here, code templates are the most common way to provide model-to-text transformations through frameworks like Xtend or Acceleo.

DSL editor building phase

Once the DSL is ready to use, it is important to give the developers an editor where DSL instances can be created easily. This editor must be capable of validate, suggest and auto-complete DSL code fragments. It is also helpful for developers when no documentation about
the language exists or is not provided. In this case the editor can guide in the process of creating models. The following are the most important characteristics of a DSL editor [40]:

- **Code completion**: It is maybe the most essential service provided by a language editor. When activated, the editor shows all valid targets for a reference in a particular location. Selecting one of them will complete the reference.

- **Syntax Coloring**: This can be achieved in two flavours. Syntactic highlighting and semantic highlighting. The first one is used to colour keywords, the second colours code fragments based on the abstract syntax.

- **Pretty printing**: This manages how the code should be structured to be pretty and easy to read. For example, it manages the white-spaces and tabulators within the code.

- **Quick Fixes**: It is a semi-automatic fix for a constraint violation. It is normally presented as a contextual menu with alternatives to rectify the problem that causes the constraint violation.

- **Refactoring**: This allows changing the program structure without changing its behaviour. For example, to rename a variable and its references.

- **Outline**: It provides an overview of the model contents. It is normally presented as a tree with their branch providing quick access to model elements.

- **Code folding**: Refers to the small minuses and pluses in the gutter of an editor that let to collapse/restore code regions.

- **Tooltips / Hover**: It is a small, typically yellow window that is shown if the user hovers the mouse over a model element. It may show the documentation or more information about that particular element.

**Testing phase**

Finally, a testing phase provides the certainty that development teams need to trust a MDSD tool. The finished tool is used in real examples to assure that it could really help developers to build applications with less effort and better quality.

**2.3. Domain-Specific Languages**

Domain-specific languages deserve a special section because they play a very important role within the MDSD methodology. This section presents what is a DSL, types of DSL, their
advantages and disadvantages and some rules to build them.

A programming language is defined by [24] like a standardized communication technique for expressing instructions to a computer. It is a set of syntactic and semantic rules used to define computer programs. A language enables a programmer to precisely specify what data a computer will act upon, how these data will be stored or transmitted, and precisely what actions to take under various circumstances. Although it is a good definition, an agreement about how to define programming languages between authors and experts is hard to find. Programming languages can be classified into different criteria according to their purpose (e.g. concurrent programming or web programming), their generation (from 1GL to 5GL), their paradigm (object oriented like Java or functional like Clojure) or whether it is declarative or imperative and domain-specific or general-purpose.

General-purpose languages (GPL) are designed to be used in multiple domains from the enterprise context to scientific areas, being Java and C++ some of the most representative languages. On the other hand, domain-specific languages (DSL) are explicitly suited for particular set of applications. Such applications are commonly called application domain or business domain. Instead of using a general language capable of solving every possible problem, DSLs are meant to solve very specific problems. Examples of such problems are the definition of user interfaces, transactional management, queries, validation, automatic test definition, security definition, among others. There are many examples of DSLs which are already widely used:

- **CSS**: Style sheets for defining user interface in web applications.
- **SQL**: A language to perform queries over relational databases.
- **XML**: A standardized language used for data transport.
- **WSDL**: A language for describing web services interfaces.
- **JSP**: A language for specifying web user interfaces within the java platform.

According to [27], two types of DSLs can be distinguished. *External DSLs* are languages written in a different language than the main application where they are used. They have their own syntax instead of being built on top of a language. Examples of external DSLs include CSS and regular expressions. Alternatively, *internal DSLs* are written in the same language as the main application’s source code. Examples of internal DSLs are Rails and Django both on top of the Ruby language. Finally, a term introduced by Martin Fowler in [25] is *language workbench*. Language workbenches are tools that support the efficient definition, reuse and composition of languages and their IDEs. Language workbenches make the development of new languages affordable and, therefore, support a new quality of language
2.3 Domain-Specific Languages

engineering, where sets of syntactically and semantically integrated languages can be built with comparably little effort [22]. Examples of language workbenches include Spoofax [5], JetBrains MPS [3] and Xtext [7], being the latest the selected tool for building MoDev (see 3.3.1).

Domain-specific languages offer several advantages explained below [40]:

- **Quality**: DSLs can reduce the errors in created products and can increase the maintainability by eliminating some degree of freedom to developers and automating some of the repetitive and error-prone tasks.

- **Validation and verification**: as DSLs are tailored to particular applications, they have, in some manner, pre-validated and pre-verified some of the rules of such applications.

- **Productivity**: As explained before DSLs removes some of the manually repetitive tasks performed by programmers. The amount of code a developer has to write and read introduces complexity, independent of what the code expresses and how. The ability to reduce that amount while retaining the same semantic content is a huge advantage [30].

- **A communication tool**: Having a domain expressed in a language that uses terms aligned with the domain allows designers and developers to have a clear view of what is being created with no need of implementation details.

- **Data longevity**: As domain-specific language should not retain any platform specific concern, they are expressed in a very meaningful level of abstraction for the domain, therefore, it is possible to analyse and generate code for multiple platforms based on the same model.

There are also still some drawbacks that designers, developers and architects must face when creating applications using domain-specific languages:

- **Expertise and effort**: Creating and maintaining DSLs is not an easy task. People with the necessary expertise about this topic is needed in order to have really useful languages. Besides, the time and costs required to create a DSL have to be taken into account when doing estimations and budgets.

- **Cultural disposition**: Development processes are sometimes really hard to change in software teams. People is not always open to change the way the have been working for years in their companies. Some others are pessimistic about language engineering and the changes involved and many others do not take the risk of using new paradigms in real projects.
• **Tool lock-in**: Many DSLs are created using particular tools and must be created and executed using them. Some development teams are not willing to lock in a specific tool, which no always is an open-source tool but sold by an specific vendor.

### 2.4. MDD benefits and issues

Previous section already mentioned some of the advantages and disadvantages of using domain-specific languages. Many of them are still applicable to the whole model-driven development process and they are explored in this section.

It has been already mentioned that model-driven engineering has a lot of benefits concerning productivity and software quality. Having a MDD tool involved in the development process, a considerable percentage of the code is automatically generated, developers should focus only in creating and performing code constructs for the specific domain where they are working. That means, that only the business logic will be coded by developers and other repetitive code fragments will be generated. Hence, fewer errors in the development process would appear and less time would be invested. But automatic generation does not only affect source code, other artefacts like documentation, tests or configurations could be generated as well. Thus, when the models are updated not only the code will be regenerated but also other artefacts, keeping everything in perfect synchronization.

Starting with documentation, maintaining it right up-to-date is a tedious task and several studies confirm widely that in most projects, software engineers typically do not update documentation as timely or completely as stakeholders or managers advocate [38, 50]. Configuration management in a software projects is also a key topic that can be addressed successfully with an MDD tool. Generating artefacts for an specific environment (e.g. development, staging, QA or production) or for an specific tag or branch in a version control system can be easily abstracted and automated [28]. Configuration for external tools like SCMs (Software Configuration Management Systems) like Maven or VCSs (Version Control Systems) like Git can be obtained *for free* by means of a MDD approach. Testing is another important topic when developing software. Some methodologies like Test-Driven Development (TDD) promotes the creation of tests before the implementation but they have not been widely used in software teams y Using a MDD tool can reduce considerably the effort of creating tests by generating automatically part of them and by forcing developers to complete them, for example, with the help of a continuous integration (CI) system that executes all tests before packaging and deploying. Thus, not allowing to proceed without completing the tests.

As a direct consequence of reducing efforts, development times of software projects using this kind of approach will be lower compared to project manually developed. It can also
help to speed up the return of investment (ROI) of the projects. Therefore, an immediate result of better time to market is the reduction of project costs. This is an important issue in software engineering since many projects do not finish on schedule not only because bad planning but because the team capabilities are not adequate as well [18].

But MDD can also help in software projects with an established and well defined software architecture. Software quality attributes are key topics when defining software architectures. To check how MDD helps, table 2-1 presents a list of the quality feature sets that any piece of software should be characterized by, according to the ISO 9162 standard [9].

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functionality</td>
<td>A set of a attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.</td>
</tr>
<tr>
<td>Reliability</td>
<td>A set of attributes that bear on the capacity of software to maintain its level of performance under stated conditions for a stated period.</td>
</tr>
<tr>
<td>Usability</td>
<td>A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.</td>
</tr>
<tr>
<td>Efficiency</td>
<td>A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.</td>
</tr>
<tr>
<td>Maintainability</td>
<td>A set of attributes that bear on the effort needed to make specified modifications.</td>
</tr>
<tr>
<td>Portability</td>
<td>A set of attributes that bear on the ability of software to be transferred from one environment to another.</td>
</tr>
</tbody>
</table>

The biggest impact of MDD among these quality attributes resides above all in maintainability and portability. Maintainability requires that the MDD tool supports re-generation, that is, the capability of generating code, modify certain pieces of that code manually and generating again without loosing the manually written code. To achieve that, the generation gap pattern or protected regions can be used. The generation gap pattern \(^1\) recommends to split into different folders and classes the automatically generated code and manually written code using object oriented features like abstract classes. A key drawback of this approach is that it can be only used with source code in platforms that support object oriented programming. This pattern is not useful when generating for other platforms or even

\(^1\)[http://heikobehrens.net/2009/04/23/generation-gap-pattern/]
when generating other kind of artefacts like XML or HTML files. Protected regions is the second strategy to support re-generations and is tailored to be used in any plain text file. A protected region consists of an starting point delimited by a unique id and an ending point. They are commonly used as comments within the code or within other type of files. More about protected regions can be found in section 3.3.8. Once re-generation is supported by the MDD tool, adding new functionalities to an application is faster than doing it manually. Depending on the MDD tool, it would be responsible of creating the boilerplate code, the documentation and even some parts of the code for the tests. Sometimes it would be needed to remove parts of the code. This can be done by modifying the model and all the related components in the application will be removed after a re-generation.

In terms of portability, the previous section explained some benefits about DSLs, being data longevity one of them. DSLs are composed by domain related concepts. None of them are tied to an specific platform. Taking this into account, it is possible to create templates for any technology, environment, language or framework. Hence, porting the code easier than manually. Regarding functionality, reliability, usability and efficiency, using an MDD tool has no direct impact in the development of an application. However, some automation can be achieved with the help of architects, designers and developers. Thus, by establishing where can be automated some practices about these characteristics, abstracting them and creating some generators for them would be a good starting point. For example, on the subject of usability, an MDD tool can automate the generation of user interfaces according to the corporate image and identity of any particular company.

On the other hand, one of the drawbacks of MDD is that a new tool have to be developed and maintained. Building a MDD tool requires experts in that field and additional time of the originally considered for the project. Unless a MDD tool already exists and this tool perfectly fits to the project requirements, an MDD approach would not be useful for projects with tight development times and budgets, it is only deserving for large projects and development teams with more than four people. Another disadvantage of MDD is that meta-models and generators are often tightly coupled with very specific domains, making these MDD tools neither reusable nor extensible. That is precisely one of the goals of this work, to make MoDev usable in as many projects or companies as possible.

2.5. Related Work

None of the related work found presents a complete solution to support software architectures in a reusable and modular way using model-driven development. However, many researches threat independently some architectural aspects like architecture conformance, non-functional requirements abstraction, software documentation generation, automatic code generation, integration and configuration modelling and automation. This section examines
some of the most relevant ones and this work is committed to take into account some of the lessons learned.

If applied successfully, MDSD can help to an increase in quality, less efforts for developers and software engineers and better return to investment on software projects. Several studies have been made around the adoption of MDE in the industry. Hutchinson et al. [33, 32] reports that success or failure in MDE projects depends on social, technical and organizational factors. This work emphasizes above all on organizational factors where aspects like commitment to organizational changes, highly motivated MDSD users, the integration of MDE in new development processes and the focus on MDE as a solution to new commercial and organizational challenges, can make a huge difference. Burden et al. [17] complements these results by stating that two areas of their data refutes Hutchinson’s observations: how to introduce MDE without permitting engineers to think that they are loosing control and how much engineers have the right training for applying a MDSD approach. Additionally, they illustrate the possibility of involving domain experts within software implementation by means of secondary software or tools specially developed to that end. On the other hand, [20] presents practical experience in two transfer of technology projects on two small companies using MDSD. Results indicated that adoption of MDE in both companies differs. As one company had problems engaging the approach in the company due to technical and organizational factors, the other one adopted successfully MDE as a candidate strategy to build automation tools.

In relation to software architecture conformance, Aldrich et al. [10] proposes a general purpose language called ArchJava which allows programmers to express architectural structure and then seamlessly fill in the implementation with Java code. At every stage of the software lifecycle, ArchJava enforces communication integrity, ensuring that the implementation conforms to the specified architecture. Although this is a great approach and very related to this thesis, it does only contemplate one platform (Java) and one architecture without evidence of how modular, extensible and reusable is.

Regarding software architecture, Mattson et al. [41] reports the use of MDD on a real world software architecture project. They argue that software architecture is a very important design artefact where a set of architectural rules are then followed in the detailed design of a system to achieve certain quality attributes. The question they came across the investigation is which architectural rules could be integrated in a software project using MDSD. In the practical case examined, they found that MDD has been able to automate the step from detailed design to implementation, eliminating time-consuming coding and code reviews, but MDSD was not able to automate enforcement of the architecture on the detailed design due to the inability to model architectural rules. This causes several problems including stalled detailed design, premature detailed design, low review quality and poor communication of
the architecture from architects to design teams and developers.

Despite of Mattsson’s et al. experience automating the enforcement of the architectural rules on the detailed design by means of MDSD, there are still research about using MDD with common elements of software architectures. For example, Cabot et al. [11] explores the state of the art about MDD tools and Non-Functional Requirements (NFR). Two types of approaches were examined, methods using MDD that are not able to manage with NFRs and methods that apply some kind of treatment to NFRs. They considered as an improvement literature findings a formulation of an NFR-aware general framework which allows customization to different settings with their own peculiarities. Test cases generation is another topic widely researched together with MDD. [34] explores how to generate test cases from UML sequence diagrams. This generation, placed in a model-driven methodology, consists of two steps. The first step takes the sequence diagram and performs a model-to-model (M2M) transformation and gets a general test case model or xUnit model which is independent of a particular unit testing framework. Then, in the second step, that model is transformed into platform-specific (e.g. JUnit, SUnit and PHPUnit) test cases that are concrete and executable. Although details of implementation and examples were not enough to fully understand this approach, it is absolutely a good starting point to take into account for this thesis work. Rutherford et al. [47] introduces a case study where a model-driven development tool is capable of generating unit tests and integration test at different layers for a specific application. They exhibit multiple benefits of using MDD in software testing. Regarding documentation, Wang et al. [56] proposes a new methodology for automatic documentation generation, capable of keeping in synchronization a software system and its documentation using model-driven development. With this approach, they are able to generate the requirements specification, the preliminary design specification and the detailed design specification. Moreover, Heinrich et al. [31] develops a model-driven documentation system which is adapted to the MDSD lifecycle and therefore exploits synergies coming along with the alignment of software development and software documentation.

As well as some scientific related and relevant work has been identified, some commercial and open source products are available to supply similar needs. The following are the most representative related to this thesis.

2.5.1. Yeoman

Although Yeoman [8] is not properly a model-driven development tool, it is a extraordinary scaffolding tool used in Node.js environments to generate web applications through a command-line interface (see Figure 2-4). The main concept around Yeoman is a generator. A generator or a set of them are used to generate huge amount of boilerplate code besides from giving great foundation and good practices for building applications.
Apart from being open source, one of the most interesting advantages of Yeoman is the simplicity to create new and customized generators by third-party developers, which contribution has been very important for the adoption of this product. A commonly used generator is the MEAN stack generator. Inspired in the mean.io\textsuperscript{2} framework it simplifies and accelerates the web application development using leading technologies like MongoDB, ExpressJS, AngularJS and Node.js. This framework considers many of the concepts used in section 3.2. Another very known generator is the webapp generator which is useful to generate front-end web applications using technologies and tools like CSS, CoffeScript, SaSS, Bower, Mocha, PhantomJS and Bootstrap.

The main disadvantage of some scaffolding tools like this, is that is impossible to have a global view of the application. That means that while MDD tools provide abstractions of the application through models, with scaffolding the only way to know about the application would be reading the source code. This also happens when changing or removing artefacts from the application. With an MDD tool it is as easy as modifying the models and generating the code again, but with Yeoman any modification must be made directly in the code.

![Welcome to Yeoman, ladies and gentlemen!](image)

**Figure 2-4.** Scaffolding web applications with the Yeoman command-line interface.

### 2.5.2. GenMyModel

GenMyModel [2] is an online UML modelling tool which supports real time collaboration and on-line code generation (see Figure 2-5). Axellience, a french start-up company founded in 2012 and creator of GenMyModel, has raised recently 500.000 EUR to keep developing its

\textsuperscript{2}https://github.com/jrcryer/generator-mean
product. Models are built based on the UML2 meta-model and can be exported in multiple formats. At this moment, this tool supports use case, class, activity and sequence diagrams. Collaborative design is the most remarkable feature of GenMyModel allowing developers and designers to design software architecture together. Another advantage of this tool is that code generation is available on-line by clicking just a button. Generated code can be downloaded as a compressed file or it can be pushed instantly to a remote Git repository. However, code generation cannot be customized and at this moment the Java language, SQL and the Spring framework are only supported, though more languages and technologies will be supported in the next few years.

Figure 2-5.: A screenshot of the GenMyModel application showing some UML diagrams.

2.5.3. Sculptor

Sculptor [4] is a tool developed and maintained by Itemis A.G., the creators of the openArchitectureWare platform, later named as Xtext 1.0. It is a productivity tool that let developers to express a design intent in a textual DSL (see an example in Figure 2-6), from which Sculptor generates high quality Java code and configuration. The concepts from Domain-Driven Design (DDD) like Service, Module, Entity, Value Object and Repository can also
be used. The generated code is based on well-known frameworks, such as JPA, Hibernate, Spring, Google App Engine, JSF, RCP, SmartClient and Java EE. Sculptor takes care of the technical details, the tedious repetitive work and let developers to focus on delivering more business value.

The primary disadvantage of Sculptor is that it focuses exclusively on the DDD paradigm and generates code for very specific frameworks of the Java platform. This turns the tool useless if the requirements of a project does not contemplate such technologies or a different approach.

![A sample blog application modeled with Sculptor using textual representations.](image)

Figure 2-6.: A sample blog application modeled with Sculptor using textual representations.
2.6. Alternatives

Having explored the most related scientific and commercial work, the table 2-2 presents the main problems about them. Once the problems are identified, this work aims to solve or mitigate the impact of some of them. The solution, its implementation and evaluation is explained in the following chapters.

Table 2-2: Problems and alternatives detected in the related work

<table>
<thead>
<tr>
<th>Problem</th>
<th>Description of the problem</th>
<th>Alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Closed solutions</td>
<td>None of the solutions provide a description of how to extend them.</td>
<td>Provide implementations ready to be extended.</td>
</tr>
<tr>
<td>Targeted to a specific company</td>
<td>Some solutions were applicable to determined companies with custom requirements.</td>
<td>Although it is valid to apply solutions to very specific requirements, all companies and software teams have always different requirements. Generic and customizable solutions would be recommended.</td>
</tr>
<tr>
<td>Tools aimed to a very specific platform</td>
<td>Almost every solution explored were aimed to specific technologies (most of them to Java) or vendors without specifying how can be extended or customized.</td>
<td>The creation of more technology independent models/approaches.</td>
</tr>
<tr>
<td>Targeted to a very small problem</td>
<td>Due to they cannot be extended (or do not explain how to do that), some of the are useful in a very small problem. For example, generating JavaDoc or generating code for a very specific project in a determined company.</td>
<td>Providing the easiness of being extended and customized, it would be natural to apply a solution to bigger domains.</td>
</tr>
<tr>
<td>One DSL for everything</td>
<td>When solutions implement a DSL, they introduce all concepts in that single DSL. If any part of that DSL wanted to be reused elsewhere, it would be hard to do.</td>
<td>Like in software development, splitting up domain concepts into different lowly decoupled and cohesive components (DSLs), is a very good practice.</td>
</tr>
</tbody>
</table>
3. **MoDev**

*MoDev* is the chosen name for the tool implemented in this work. This tool allows software developers to define abstractions of software applications in terms of the domain that they are dealing with and in terms of a pre-defined software architecture. In other words, it is a modeling and code generation tool that generates the ”cookie-cutter” code from a structured representation of a customer’s requirements. *MoDev* allows developers to concentrate on understanding the customer’s needs, developing domain-specific logic, and adding to the corporate knowledge base.

This chapter presents at first two stages of the prototyping phase according to the methodology (see 2.2.1). First, the requirements that MoDev should suffice when it is used for creating applications. And then, the definition of a software architecture. As MoDev is intended to be extended and used for any software architecture, an illustrative one is described in section 3.2. Once the architecture, its layers, its technologies and its rules are defined, then a complete explanation o how MoDev was built is presented in section 3.3. There, the rest of the steps of the methodology are explained one by one. Finally, a brief description of how applications are developed using *MoDev* and what to do when new architectures have to be supported by this tool.

3.1. **Requirements**

As stated in previous chapters, many of the current MDD tools are developed to be used in specific domains. This have been useful for companies that work in those domains but the following questions raise: What if one needs to use that tools in new problem domains? What if we need to customize those tools to specific requirements? Commonly MDD tools generate code automatically. What if the generated code does not meet the requirements of my customer or my company? What if developers need to generate code in other language or even in multiple languages and technologies at the same time? The same happens in relation to software architectures. Many of the finished applications developed following a software architecture does not conform to the decisions and rules of the defined architectural model [16].

*MoDev* solves that problems. The main goal of *MoDev* is to abstract software architectures allowing people involved in software projects to reduce efforts and optimize times. As
MoDev is targeted to any software team, it is not forced to use some specific architecture, some precise domain or some particular language. This is accomplished by using language modularization, extension and composition.

MoDev should be also capable to help developers in all stages of software construction process. That implies that model to text (M2T) transformations should not be performed to generate source code only, but it should support things like configuration, documentation, testing and deployment as well. The following is the list of artefacts that MoDev should allow to model and generate:

- **Configuration of the application and integrations with other tools**: Application’s structure is very important. Tools like Maven or Ivy allows developers to organize their applications, manage dependencies and in general, to manage the entire lifecycle. MoDev must provide the basic configuration for such tools. Other tools like Sonar, Jenkins and Unit test frameworks, also require some configuration that MoDev should generate automatically.

- **Source code (classes or modules) according to the architecture**: The software architecture should define how classes or modules are organized and how is their role within the application. MoDev should generate the skeleton of such classes/modules in the appropriate package/folder.

- **Unit test cases for every public method or function created**: Testing is as important as coding business functionalities. A common rule followed in many projects is to create unit test cases for every public method. MoDev should provide at least the skeletons of these unit test with the purpose of improving productivity and force developers to complete them.

- **Documentation for every method and class**: Documenting the code is sometimes a task not performed in a disciplined way by developers. Every update to the model should be reflected in an update of the generated code regarding code documentation when using MoDev.

- **Documentation for REST endpoints**: The REST architectural style has a lot of benefits, one of them about systems integrations. Having REST endpoints with no adequate documentation means that it will not be used appropriately. MoDev should provide mechanisms to update and export such documentation automatically.

- **Protected regions**: MoDev should not be a one-use tool where applications are generated once and then must be completed manually. Modeling and code generation should be an iterative process, that is, supporting re-generations (see 3.3.8).
3.2. The Software Architecture

Although MoDev might be capable of assisting software development teams regardless of the software architecture they are using, it is impossible to take every possible architecture into account. That is why, for illustration purposes of this work, a representative software architecture has been selected in order to develop applications applying it and using MoDev. To define a complete software architecture is a very time-consuming task requiring the participation of different people with expertise in multiple areas. This section, describes only the most relevant items of an example software architecture with the only purpose to help MoDev to be implemented and tested based on it.

As the current trends for software projects go to web and mobile applications, the most common non-functional requirements have been chosen to build the architecture [43, 51, 53, 36].

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reliability</td>
<td>The ability to remain operational over time.</td>
</tr>
<tr>
<td>Usability</td>
<td>Measures how well the application meets the requirements of the user in terms of intuitiveness and a good overall user experience.</td>
</tr>
<tr>
<td>Security</td>
<td>The capability to prevent malicious actions outside of the designed usage, and to prevent information loss.</td>
</tr>
<tr>
<td>Availability</td>
<td>Defines the proportion of time that the system is functional and working.</td>
</tr>
<tr>
<td>Scalability</td>
<td>The ability of a system to either handle increases in load without impact on the performance of the system, or the ability to be readily enlarged.</td>
</tr>
<tr>
<td>Maintainability</td>
<td>The ability of the system to undergo changes with a degree of ease. These changes could impact components, services, features, and interfaces when adding or changing the functionality, fixing errors, and meeting new business requirements.</td>
</tr>
</tbody>
</table>

To meet some of these quality attributes, this architecture will follow a REST architectural style. It means that it will have a set of constraints applied to components, connectors and data elements, within a distributed hypermedia system. Applications will also expose web services designed in terms of the HTTP protocol and Uniform Resource Identifiers (URI). This architectural style was chosen because it has many advantages in terms of decoupling and allows extreme scalability. REST forbids conversational state and promotes stateless
operations, which means that it is possible to scale horizontally very wide without worrying
about state synchronization. The use of the HTTP protocol assures that any tool or tech-
nology can be seamlessly integrated. Moreover, the hypermedia paradigm assures HTTP
endpoint to be always navigable by clients by simply following opaque server provided links.

Regarding functional requirements, it is not necessary to define them here. *MoDev* allows
to define them easily trough a set of *DSLs* (see 2.3). However it is still important to define
the architecture layers, patterns, technologies and rules to support the quality attributes.

### 3.2.1. Layers

Layering is a very known technique used by software engineers to modularize software sys-
tems. One of the benefits of using it, is that any layer can be substituted with alternative
implementations without affecting other layers of the system. Another advantage of layering
is that software developers can understand each layer as a coherent whole without knowing
much about the others. Layers are good starting points for standardization as well [26].

For this software architecture six layers were defined (see Figure 3-1). The *model* layer
is responsible for mapping the persistent objects into their persistent stores regardless of
what kind of persistent store is used. The *repository* layer is in charge of dealing with all
data access functionality using the elements of the *model* layer. This layer would commonly
have the Create-Read-Update-Delete (CRUD) methods but obviously it might contain other
domain specific methods. Then, the *business* layer uses the *repository* elements to define all
the business rules of the application. Finally, the *route* layer defines the REST endpoints
exposing resources by means of Data Transfer Objects (DTO) defined in the *dto* layer.
Finally the *presentation* layer uses the REST endpoints published by the *route* layer to fetch
and post data.

### 3.2.2. Technologies

With the purpose of meeting all the defined quality attributes, this architecture has a spe-
cial selection of technologies. These technologies are, in principle, divided in two groups
depending upon the core language where it will be used. This separation has been made on
purpose to prove that *MoDev* is modular and re-usable as well.

The only set of technologies shared between these two groups are the persistent store engines
and the presentation technologies. MongoDB will be used in case of needing a NoSQL per-
sistent store based on documents. On the other hand, any relational database like MySQL,
PostgreSQL, MariaDB and SQLite can be used when needed. For the presentation layer,
Angular.js and Bootstrap are recommended (see Table 3-2).
3.2 The Software Architecture

Figure 3-1.: Layers for the software architecture.
Table 3-2.: Shared technologies between the two groups of platforms

<table>
<thead>
<tr>
<th>Layer</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model and Repository</td>
<td>MongoDB(^1)</td>
<td>MongoDB is an open-source document database written in C++, and the leading NoSQL database.</td>
</tr>
<tr>
<td></td>
<td>Relational Databases(^2)</td>
<td>A collection of tables of data items organized according to the relational model represent a relational database. Some examples of relational databases are SQLite, MySQL, PostgreSQL and MariaDB.</td>
</tr>
<tr>
<td>Presentation</td>
<td>Angular.js(^2)</td>
<td>Angular is a MV* javascript framework (Model-View-*) maintained by Google which excels in the creation of single page applications (SPAs). Some of the advantages of using it are: data models as POJOs, behaviour with directives, declarative user interface, less code needed, correct DOM manipulation and unit testing ready.</td>
</tr>
<tr>
<td></td>
<td>Bootstrap CSS(^3)</td>
<td>Bootstrap is a open source Javascript framework developed and maintained by a team at Twitter. It is a combination of HTML, CSS and Javascript code designed to help build responsive and mobile-first user interface components.</td>
</tr>
</tbody>
</table>

\(^1\) http://www.mongodb.org/
\(^2\) https://angularjs.org/
\(^3\) http://getbootstrap.com/

The first group of technologies is based on the Node.js platform. Launched on 2009 by Ryan Dahl, Node.js uses the *Google’s V8* runtime for javascript and *libuv* to handle asynchronous events. Since its creation it has been very popular among developers looking for a server side solution, particularly, those handling with HTTP requests. The speed inherited from V8 combined with asynchronous programming mean for developers that they can use it to build applications that *scale* to million users. Although selecting the platform is important, choosing frameworks and libraries is also relevant. Table 3-3 presents an overview of the selected frameworks and libraries for the Node.js platform.
### Table 3-3.: Libraries and frameworks for the Node.js platform

<table>
<thead>
<tr>
<th>Layer</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model and Repository</td>
<td>Mongoose¹</td>
<td>In case of needing a document-oriented database, Mongoose provides MongoDB object mapping similar to ORM with a familiar interface within Node.js.</td>
</tr>
<tr>
<td></td>
<td>Sequelize²</td>
<td>In case of a relational database needed, the Sequelize library provides easy access to MySQL, MariaDB, SQLite or PostgreSQL databases by mapping database entries to objects and vice versa. To put it in a nutshell, it’s an ORM (Object-Relational-Mapper).</td>
</tr>
<tr>
<td>Business and Route</td>
<td>Express.js³</td>
<td>Express.js is a flexible web application framework which helps building web applications. Currently is the most used framework for the Node.js platform. Express help developers to manage routes, views and handling HTTP requests.</td>
</tr>
<tr>
<td>All</td>
<td>Passport.js⁴</td>
<td>Passport is an authentication middleware for Node.js. Extremely flexible and modular, Passport can be unobtrusively dropped in to any Express-based web application.</td>
</tr>
<tr>
<td></td>
<td>Mocha⁵</td>
<td>Mocha is a feature-rich JavaScript test framework running on node.js and the browser, making asynchronous testing simple and fun. Mocha tests run serially, allowing for flexible and accurate reporting, while mapping uncaught exceptions to the correct test cases.</td>
</tr>
</tbody>
</table>

¹ [http://mongoosejs.com/](http://mongoosejs.com/)
³ [http://expressjs.com/](http://expressjs.com/)

The second group of technologies for this software architecture is based on Java, which is a very representative platform within the enterprise world providing a concurrent, class-based and object oriented language. Along with Node.js, Java is one of the most suitable platforms for the kind of non-functional requirements and quality attributes selected for this software architecture. Figure 3-4 shows the list of recommended frameworks and libraries for this software architecture.
Table 3-4.: Libraries and frameworks for the Node.js platform

<table>
<thead>
<tr>
<th>Layer</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model and Repository</td>
<td>Spring Data(^1)</td>
<td>Spring Data makes it easy to use new data access technologies, such as non-relational databases, map-reduce frameworks, and cloud based data services. Spring Data also provides improved support for relational database technologies. This is an umbrella project which contains many subprojects that are specific to a given database. For this architecture it will be used to support MongoDB and relational databases.</td>
</tr>
<tr>
<td></td>
<td>JPA(^2)</td>
<td>JPA is the de facto application programming interface specification to handle and manage relational data in applications using the Java Platform.</td>
</tr>
<tr>
<td>Business and Route</td>
<td>Spring(^3)</td>
<td>The Spring Framework is an open source application framework and inversion of control container for the Java platform. The framework’s core features can be used by JVM-based system helping developers to build simple, portable, fast and flexible applications.</td>
</tr>
<tr>
<td>All</td>
<td>Apache Shiro(^4)</td>
<td>Apache Shiro is a powerful and easy-to-use Java security framework that performs authentication, authorization, cryptography, and session management. With Shiro’s easy-to-understand API, it is possible to quickly and easily secure any application, from the smallest mobile applications to the largest web and enterprise applications.</td>
</tr>
<tr>
<td></td>
<td>JUnit(^5)</td>
<td>JUnit is a simple framework to write repeatable tests. It is an instance of the xUnit architecture for unit testing frameworks.</td>
</tr>
</tbody>
</table>

\(^1\) [http://projects.spring.io/spring-data/](http://projects.spring.io/spring-data/)


\(^3\) [http://spring.io/](http://spring.io/)

\(^4\) [http://junit.org/](http://junit.org/)


### 3.2.3. Rules

It is also important to highlight some of the most important rules specified by the architecture. These rules must be then taken into account when developing not only the applications that are using it, but also when developing the generators (see 3.3.6) of MoDev.

Organization is the first rule. It is decisive to set up a standard organization for all the artefacts that compose any application using this architecture. Organization means, where and how to put all elements of the application. Table 3-5 explains this organization and
Figure 3-2 shows an example.

Table 3-5.: Rules for organizing the artefacts on applications using this architecture

<table>
<thead>
<tr>
<th>Element</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documentation</td>
<td>Documentation should be located at the doc folder. Specific documentation like REST endpoints documentation (WADL), code documentation like the JavaDoc and other type of documentation should be located in separate folders.</td>
</tr>
<tr>
<td>Back-end code</td>
<td>Back-end code is allocated in the server folder. Modules and other kind of special organization for the application should be sorted within subfolders as well.</td>
</tr>
<tr>
<td>Front-end code</td>
<td>Web client code should be located in a web-client folder. Mobile code (in case it exists) should be allocated into the mobile-client folder</td>
</tr>
<tr>
<td>Tests</td>
<td>Tests for back-end code should be located in the test folder within the server folder. Unit and integration tests must be located in their respective subfolders. On the other hand, tests for the front-end code should be also located in their respective folder inside the web-client or mobile-client folders.</td>
</tr>
<tr>
<td>MoDev models</td>
<td>MoDev models should be located in the model folder.</td>
</tr>
</tbody>
</table>

The second rule is about testing. Both front-end and back-end code must be tested. For each module or class which has exported or public methods there should be at least one test case for the expected behaviour and the exceptional behaviour (i.e. errors or exceptions).

The following rule is about documentation. As the architecture follows a REST architectural style, all rest endpoints should be documented with tools like Swagger\(^1\) in order to provide developers a guide to consume all services. The code should be documented as well, following specifications like JavaDoc for Java or JSDoc for Javascript is considered a good practice. Other recommendations consist of using several tools for version control, continuous integration and delivery and code metrics like Git\(^2\), Jenkins\(^3\) and SonarQube\(^4\) respectively.

### 3.3. Implementing MoDev

#### 3.3.1. Technologies

To create MoDev the following set of tools are used:

\(^1\)https://github.com/wordnik/swagger-core
\(^2\)http://git-scm.com/
\(^3\)http://jenkins-ci.org/
\(^4\)http://www.sonarqube.org/
Figure 3-2: Example of artefact organization for applications using the architecture.
3.3 Implementing MoDev

Eclipse Modeling Framework (EMF)

EMF is a framework for MDD based on Eclipse that works as a basis for a lot of interesting tools. Its core is based on eCore, an implementation of meta-meta-model that is related to eMOF, the meta-meta-model of UML. EMF allows the definition of meta-models using different techniques like tree-based editors. From there it is possible to generate implementation classes that provide a concrete API to build instances of that meta-model. EMF presents some integration with code generators which allow to create model editors [1].

Xtext

Xtext is a language workbench based on Eclipse for developing domain-specific languages in a textual manner. It covers all aspects of a complete language infrastructure, from parsers, over linker, compiler or interpreter [7]. It also supports features like syntax coloring, content assist, validation, quick fixes, advanced java integration and some other IDE features. Xtext has been chosen over other language workbenches [3, 5] because of its simplicity and powerful features [22].

Xtend

Xtend is one of the most common programming languages for the JVM. Xtend uses the exact type system of java and compiles directly to Java code. It also has natural integration with Xtext helping out, among other things, as a powerful template engine [6].

3.3.2. Main concepts

To explain how MoDev was built, it is important to define some relevant concepts. (see Figure 3-3). As MoDev aims to abstract software architectures and this is actually the main concept to highlight. A software architecture is about high level software structures, their relations to each others and the properties involved. Those structures can be abstracted into models which describe them from a generalized point of view. With models representing architectural elements, it is possible to create domain-specific languages (DSLs) with proper semantics and syntax. Each DSL would then have an editor which allows the user of MoDev to create instances of the DSL with features like auto-completion and syntax highlighting. Once the models are specified, MoDev can perform model-to-text transformations (M2T) in order to have software artefacts like source code, documentation, configuration files, etc. Each model would have one or more generators and a generator has only a direct dependency with one model and its possible inherent relationships. In a nutshell, software architectures are abstracted using a set of models, each of one having one editor and several generators. This approach allows MoDev to support any software architecture and any architectural element within them, besides from the ability to reuse some of these elements among multiple software architectures.
Figure 3-3.: General concepts about MoDev and its building
That is why, it was emphasized that a representative software architecture was needed. Having a basic definition of a software architecture it is possible to prove that the mentioned approach to build *MoDev* is actually feasible. The following sections describe all of these elements for this particular architecture.

### 3.3.3. Reference Implementation and Analysis

The reference implementation step is very important. It consists of creating an illustrative application manually with all rules and rationales exposed by a software architecture. In this case, the reference implementation was made about an authorization and authentication system for REST APIs. Indeed, functional and non-functional requirements are described in section 4.1 where instead of doing it manually, *MoDev* was used. Summarizing, this manual implementation was made by one developer, taking approximately 60 hours of work and resulting in 30 files created with more than 1000 lines of code.

Once the reference implementation is finished, is a relatively easy task to separate what code is generic, repetitive and individual. *Generic code* comprises code fragments that are very specific to the platform, in this case Node.js. Configuration files (e.g. a package.json file), mandatory files (the main file, server.js which creates the server) and module imports are just some examples (see table 3-6).

<table>
<thead>
<tr>
<th>Generic component</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration files</td>
<td>A best practice in Node.js applications is to have a package.json file describing the project and including all necessary dependencies</td>
<td>package.json file</td>
</tr>
<tr>
<td>Main file</td>
<td>Every application has an entry point. In this case, the server.js file is responsible for stating serving requests.</td>
<td>server.js file</td>
</tr>
<tr>
<td>Module imports</td>
<td>Importing external modules is made with the keyword <em>require</em>. A best practice is to have imports always in the top of the file.</td>
<td>in almost every file</td>
</tr>
<tr>
<td>Function declaration</td>
<td>Function declarations are made with the keywords <em>module.exports</em>.</td>
<td>module.exports.hi</td>
</tr>
</tbody>
</table>

*Repetitive code* has always the same structure or is even the same in every application. Table 3-7 shows some examples. On the other hand, *repetitive code* corresponds to that particularities that cannot be abstracted nor automated (see Table 3-8).
Table 3-7.: Some repetitive components detected from the reference implementation

<table>
<thead>
<tr>
<th>Repetitive component</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package structure</td>
<td>The package structure is always the same according to the architecture and the implemented modules.</td>
<td>see 3-5</td>
</tr>
<tr>
<td>The repository factory</td>
<td>The software pattern is applied in order to retrieve repositories at persistence layer.</td>
<td>RepositoryFactory file</td>
</tr>
<tr>
<td>The main routes file</td>
<td>All routes are created or imported in a single file in the route package.</td>
<td>index.js file</td>
</tr>
<tr>
<td>Routes</td>
<td>Routes have always the same structure and the are compliant with the HTTP protocol.</td>
<td>app.get('/roles', roleB.allRoles);</td>
</tr>
<tr>
<td>Separation of modules</td>
<td>The domain of an application can be split up into multiple modules. Such modules are distinguished by folders or packages and can be modeled easily.</td>
<td>modules within the repository layer</td>
</tr>
<tr>
<td>Function signatures</td>
<td>The way functions are declared is compliant with the JavaScript language. Those functions signatures can be easily abstracted and modeled.</td>
<td>var deleteRole = function(id)</td>
</tr>
</tbody>
</table>

Table 3-8.: Some individual components detected from the reference implementation

<table>
<thead>
<tr>
<th>Individual component</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Particular imports/require</td>
<td>It is difficult to anticipate every dependency between modules.</td>
<td>every module implemented</td>
</tr>
<tr>
<td>Body of the repository functions</td>
<td>Behaviour of repository (data access) modules cannot be abstracted in an easy way.</td>
<td>every repository module</td>
</tr>
<tr>
<td>Body of the business functions</td>
<td>Implementation of business modules cannot be abstracted in an easy way.</td>
<td>every business module</td>
</tr>
</tbody>
</table>

3.3.4. Meta-models

Table 3-9 shows the meta-models developed for the software architecture and their purpose. Each model abstracts one element of the architecture. These models could be an abstraction of a layer, a component, a particular technology like a database or even elements like documentation and configuration for deployment and testing. Unless necessary for automation purposes, the models have no dependency with any specific platform, version, system or
3.3 Implementing MoDev

domain. They should be as generic as possible.

Table 3-9.: Models developed for the architecture

<table>
<thead>
<tr>
<th>Model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document</td>
<td>The document model abstracts entirely the model layer in case of using a document-oriented NoSQL database. It includes concepts like schemas, indexes, properties and types like arrays and mixed types. It also supports sub-documents or nested documents (see Figure A-1).</td>
</tr>
<tr>
<td>Entity</td>
<td>The entity model represents the model layer in case of having relational databases. It does not keep in mind specific features of particular engines but the relational model in general. It comprises notions like entity/tables, columns, primary keys, relationships, types of relationships (HasOneRelation, HasManyRelation and BelongsToRelation) and data types (see Figure A-4).</td>
</tr>
<tr>
<td>DB Config</td>
<td>This model abstracts the connection parameters that any relational database could have, such as host, port, user, password, database and schema. It also provides grouping of these data by environments, for example, development, staging and production (see Figure A-2).</td>
</tr>
<tr>
<td>Mongo Config</td>
<td>This is the abstraction of the connection parameters to a document-oriented database. Elements like host, port, user, password and database name are modelled. It also provides grouping of these data by environments, for example, development, staging and production (see Figure A-3).</td>
</tr>
<tr>
<td>Repository</td>
<td>This is an abstract model that provides abstractions for the repository layer of the architecture which provides access to the persistent storage elements such as tables or documents. It includes concepts like operations, parameters and return types. All CRUD operations could be modelled here for example (see Figure A-5).</td>
</tr>
<tr>
<td>RelationalRepository</td>
<td>This is the first concrete implementation of the repository model. As the repositories have a dependency to the model layer, here the relational entities are established as a dependency (see Figure A-7).</td>
</tr>
<tr>
<td>DocumentRepository</td>
<td>This is the second concrete implementation of the repository model. Here the belongsTo property would have a relation to the document model (see Figure 3-5 and A-6).</td>
</tr>
</tbody>
</table>

Continued on next page
Table 3-9 – continued from previous page

<table>
<thead>
<tr>
<th>Model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business</td>
<td>The <em>business</em> model abstracts all the business rules for the application, which are modelled like methods or functions with parameters and return types (see Figure A-8).</td>
</tr>
<tr>
<td>Route</td>
<td>The route model allows to define HTTP endpoints in terms of URIs, HTTP verbs and the definition of which operation of the business layer would perform the logic (see Figure A-9).</td>
</tr>
<tr>
<td>DTO</td>
<td>The DTO model provides an abstraction of elements like POJOs in Java. DTOs are used in the route layer to receive and return data from the business layer. This is an abstract model as it could have an optional dependency with <em>documents</em> or <em>entities</em> (see Figure A-10).</td>
</tr>
<tr>
<td>DTOFromDocument</td>
<td>This is a concrete model of DTO that provides a relationship to <em>documents</em>.</td>
</tr>
<tr>
<td>DTOFromEntity</td>
<td>This is a concrete model of DTO that provides a relationship to <em>entities</em>.</td>
</tr>
<tr>
<td>LayeredApp</td>
<td>This model describes entirely the complete architecture. Through composition it uses all the previous described models and organizes them in a logical way. As more models are defined, they should be included in this one for this architecture or in other ones for any different architecture definition (see Figure 3-4). Other definitions were not created outside this main model because they are only useful for this architecture. This is the case of the <em>ProjectConfig</em> model that defines locations of the source files and the technology to be used (Node.JS or Java).</td>
</tr>
</tbody>
</table>

Some of the models are abstractions over abstractions in itself. That means, in terms of Object Oriented Programming (OOP), that sometimes it is necessary to provide abstract implementations and deriving implementations through inheritance. This is the case for example, when abstracting a repository. A repository in this architecture provide access functions to data in a persistent store. However, the architecture supports two different types of persistent stores: a NoSQL based on documents (MongoDB) and a relational approach (relational databases). To achieve that, a repository model has been created with common elements like operations, parameters and return types. Nevertheless, repositories must have a relationship with model elements like documents (NoSQL) and entities (Relational) and thus two additional concrete model implementations of the repository model have been created (see Fig 3-5). In case of needing a new way of persisting data, for example a key-value database, it would be easy to create a new model for it. This approach makes
Figure 3-4.: Overview of the models developed for this architecture
MoDev a very reusable and modular tool.

### 3.3.5. DSLs

Every domain-specific language developed for MoDev is external. They are not created on top of any language and therefore the are as independent as possible. DSLs are meant to provide an abstraction to a particular domain. For each meta-model developed, a DSL was also created. Each DSL has an *abstract syntax* which is commonly a tree or graph holding semantically relevant information. This *abstract syntax* is provided by the meta-model. A DSL defines a *concrete syntax* as well. This syntax defines de notation with which users can express programs. Another important element of these DSLs is the *static semantic*, a set of rules to which all programs must conform in addition to be structurally correct [40].

A good example of a DSL is the corresponding to the document model. The idea of this meta-model is to set up the main concepts and its meaning of the architecture’s model layer having a document-oriented persistent storage. That is why elements like documents/schemas, sub-documents, properties and data types appear. Besides from that, an specification have to be defined of how the language should be used. For that reason syntactical elements like documents-module, schema and the curly brackets around them were surrounded (see Figure 3-6).
Figure 3-6.: The document DSL. An example of the DSL that conforms to the document meta-model.
3.3.6. Generators

Code generation in MoDev is a straightforward task of using instances of DSLs (or models) to transform them into plain text files. All of the design and implementation decisions that go into the creation of the finished product are codified in artefact templates, which are Xtend classes performing model-to-text (M2T) transformations. There is a different artefact template for each type of artefact that is generated automatically. Moreover, the structures that are used in the model specification are significantly different from those that are embodied in the generated artefacts. In most cases the template that creates a particular artefact pulls data from many different models of the specification.

One example of template generating a specific artefact from exactly one model is the document template (see Figure 3-8), which generates the artefact corresponding to a MongoDB document. An example about a template using multiples models is the route template. It takes as references the route model and the business model to build the routes and it documentation (see Figure 3-7).

```java
def generate(RoutesModule routesModule) {
    "FOR" business : routesModule.allInvolvedBusinesss {
        var business.name = require("../business/(business.cContainer as BusinessModule).name="/business.name.toFirstUpper="");
    "ENDFOR"
    
    
    module.exports = function (app, passport) {
        "FOR" routeContext : routesModule.resourcesContexts {
            */
            *
            * Following are the routes for <routeContext.name>
            * Use express to express the element from express
            */
            module.exports = function (app, passport) {

            "FOR" route : routeContext.routes {
                var business = route.operation.entity as Business
                app.<route.httpVerb.toString.toLowerCase>({route.url, business.name.toFirstLower, route.operation.name});
            "ENDFOR"
        } "ENDFOR"
    }
    ...

Figure 3-7.: The route template implemented in Xtend 2
3.3 Implementing MoDev

Figure 3-8.: An extract of the document template implemented in Xtend 2

3.3.7. Editors

As explained in section 3.3.1, MoDev uses the Xtext language workbench. After the definition of the domain-specific languages Xtext creates automatically a set of Eclipse plug-ins supporting these languages with features like syntax colouring, content assist, validation, among others (see Figure 3-9). Each DSL have an editor supporting its particular syntax. Having such characteristics make the creation of models a really easy task for developers, apart from validating the models before generating the artefacts.
Figure 3-9.: An example of some editor’s features like syntax colouring, validation and content-assist.
3.3.8. Supporting re-generations with protected regions

A protected region is a portion of a text file which remains intact after a re-generation. Every protected region has an start point with a unique identifier and an end point indicating the end of the region. Protected regions are used in MoDev due to the need of keeping abstractions in the models and particularities in the generated artefacts while supporting regeneration. It is the case for example, when business rules or data access methods are defined in the repository layer through functions. Every function has a body with its implementation, and here protected regions are created in order to keep manually written code untouched after a regeneration. In this case, every line of code written outside protected regions will be deleted after a regeneration. Listing 3.1 shows an example of a protected region used in the function deletePermission. This protected region has the unique id PermissionBusiness_deletePermission_body, starts at line 4 and finishes at line 18.

```javascript
module.exports.deletePermission = function(req, res){
  var id = req.query.id;

  /* PROTECTED REGION ID(PermissionBusiness_deletePermission_body) ENABLED START */
  var permissionRepository = repositoryFactory.getPermissionRepository();
  var promise = permissionRepository.deletePermission(id);

  promise.then(function (affectedRows) {
    if (!affectedRows || typeof affectedRows === 'undefined') {
      res.status(404).json(Error.resourceNotFound);
    } else {
      res.status(204).json('');
    }
  }).then(null, function (error) {
    logger.error(error.stack);
    res.status(500).json(Error.unknownError(error));
  });
  /* PROTECTED REGION END */
};
```

Listing 3.1: Protected region example

3.4. A development process with MoDev

Having a MDD tool is a factor to perform some changes in a regular software development process. Regularly, in a code-centric approach any substantial change has to be made in the source code and then in documentation, but with MoDev, a model-centric tool, those changes should be made first at model level and then at code level. Figure 3-10 shows a
recommended software process to follow when developing applications using *MoDev*.

When starting a software project functional and non-functional requirements should be defined. Some of the non-functional requirements are solved with this tool, but functional requirements are unique for every application. Using *MoDev*, functional requirements can be modeled easily using the editors. Starting by documents or entities and passing by business rules and presentation components, all elements are split up into modules. For example, the authorization module, the administration module or the accounting module. Once the modules and its elements are established, the next step consists of creating a main model. For the defined architecture, this model corresponds to LayeredApp. This main model takes those separate units and collect them into a whole, allowing to resume the complete architecture. When the main model is complete, a one-click generation function is activated taking no longer than 30 seconds to finish. This generates and leaves all artefacts in the specified place. Checking that every artefact was correctly generated, the subsequent step is to complete the artefacts. *MoDev* is capable of generating about 40% to 70% of the artefacts and their contents depending upon the application. The rest of the code, tests, documentation, configuration or any other type must be completed manually using the respective protected regions. As *MoDev* also generates the configuration for continuous integration (CI) and continuous delivery as well as for code metrics, the next step is to version the artefacts with version control tools like Git and triggering automatically the CI process. If there are still functionalities to develop, the process may start again from the first step, if not the application should be completely finished with code, documentation, tests and possibly other elements perfectly synchronized and up to date.

### 3.5. Facing new software architectures

Previous sections described how *MoDev* faced the software architecture described in 3.2. However, *MoDev* aims to support any abstractable software architecture and when this need comes out some steps should be followed. First of all, identify the models for the software architecture. Then, check if any of the detected models can be reused. For example, many software architectures use relational databases at persistence layer, that means that the *entity* model, DSL, editor and probably the generators can be reused here. The next step is to develop the rest of the models and their corresponding DSLs, editors and generators. Finally, a main model should be created, as it was done with the *layeredApp* model, that aggregates all necessary models, be they reused or new.
Figure 3-10: Suggested development process with MoDev
4. Case study and evaluation

In this work, a model-driven development tool has been developed with the promise of reducing efforts to developers when creating applications. This tool is based on a determined software architecture and the possibility to extend it to multiple ones is open, even reusing and sharing components among each other. Such promise is evaluated in this section by setting up a case study. This case is only for illustration purposes. It has been developed by the authors using MoDev and it is fully compliant to the software architecture described in section 3.2. It is a very light system with very few components and business rules.

4.1. Case study: The authentication and authorization system

Almost every application developed for the web and/or the mobile world considers mechanisms for authenticating and authorizing users. This can be achieved by means of external systems like LDAP or Active Directories and social networks like Facebook or Twitter through standards including OAuth\(^1\) and OpenID\(^2\). However, many applications have their own authorization and authentication. One of the most common ways to implement it, is the Role-Based Access Control (RBAC) method \([46, 57]\) and there are even approaches to implement it using model-driven engineering \([21, 55, 54]\).

This case study pretends to develop an stateless application capable of securing Representational State Transfer (REST) endpoints. Stateless means that keeping information between clients and server is discarded. Each call from client to server must contain all information necessary to understand the request. Ordinary methods like cookie transfer from web-browser to the server are discouraged. On the other hand, securing REST endpoints stands for authenticating clients (e.g. with usernames and passwords) and stands for authorizing clients by verifying who is permitted to execute some action in some resource. All functional requirements are summarized in table 4-1 and some non-functional requirements are described in table 4-2, complementing those ones explained in section 3.2.

\(^1\)http://oauth.net/
\(^2\)http://openid.net/
Table 4-1: Functional requirements for the first study case: The authentication and authorization system

<table>
<thead>
<tr>
<th>Functional requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permissions Management</td>
<td>A permission determines if a specific role is permitted to perform some action at a determined URI. Such actions are compliant with the standard HTTP verbs (e.g. GET, PUT, POST, DELETE, OPTIONS). The application has to provide REST endpoints to perform operations (initially CRUD operations) over permissions and have to provide a user interface for it.</td>
</tr>
<tr>
<td>Roles Management</td>
<td>A role consists of grouping certain users to given them appropriate permissions. Each role has a set of permissions. A permission also can belongs to multiple groups. The application has to provide REST endpoints to perform operations (initially CRUD operations) over roles and have to provide a user interface for it.</td>
</tr>
<tr>
<td>Users Management</td>
<td>A user is a person who has an account to interact with the system. Credentials for users are e-mail and a password. Additional data about the user should be persisted as well. Each user has a role assigned. The system has to provide REST endpoints to perform operations (initially CRUD operations) over users and have to provide a user interface for it.</td>
</tr>
<tr>
<td>Self security</td>
<td>The main purpose of this application is to provide an API to secure external REST endpoints. However, the system may use its own functionalities to provide security to itself.</td>
</tr>
<tr>
<td>Integration with external systems</td>
<td>Implementation of this system must follow a REST architectural style to enable integration with external systems through HTTP resources.</td>
</tr>
</tbody>
</table>
Table 4-2.: Some illustrative non-Functional requirements for the first study case: The authentication and authorization system

<table>
<thead>
<tr>
<th>Functional requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability</td>
<td>As the number of external systems using this API can grow fast in little time, the application must support horizontal scalability. Therefore, using technologies like NoSQL databases are encouraged.</td>
</tr>
<tr>
<td>Maintainability</td>
<td>Future versions of the application may contain new features like integration with social networks. Maintainability of the system is a very important aspect to solve this in short times with low budgets.</td>
</tr>
<tr>
<td>Security</td>
<td>Only authenticated and authorized users/systems are permitted to interact with the application. The system must provide methods for stateless authentication and authorization such as API Keys.</td>
</tr>
</tbody>
</table>

4.1.1. Development of the case study

According to section 3.4 the first step is to create models. Creating models, depending on the methodology used and the complexity of the project, can require previous analysis or modelling using external tools. In this case, taking advantage of the simplicity of the study case, the modelling and analysis phase is made completely on MoDev.

First of all, the models mapping documents in a MongoDB database are created (see result in Figure B-1). Domain concepts like Permission, User and Role are designed according to the MoDev’s document model and DSL semantics and syntactic (see Figure A-1). For MongoDB mapping documents there is no need to write manual. Generated code is fully functional. Then, data access abstractions (i.e. functions for manipulating the database) are created like Java interfaces declaring function names, parameters and data types. For this simple case, the resulting became pure CRUD functions, but it is possible to modify them or create new ones if necessary (see Figure B-2). Repositories require manually written code by developers. To get a fully functional code, the body of the functions must be implemented respecting the respective protected regions. Additionally, skeletons for unit tests are also generated automatically with some default implementation. This default implementation makes all tests to fail when they are executed. This obligates developers to implement them (see Listing 4.1).

```
describe('deleteUser function', function () {
  /* PROTECTED REGION ID(deleteUser_body) ENABLED START */
  it('should be fulfilled', function(){
```
Business methods are similar to data access methods regarding their declaration, but their role is exclusive for application’s rules and validations declaration. Once generated, business methods must be manually coded in order to get them fully functional. Skeletons of tests for business methods are also generated as in repository functions. Listing 4.2 shows an example of a fully implemented business method.

```javascript
module.exports.deletePermission = function(req, res){
    var id = req.query.id;

    /* PROTECTED REGION ID(deletePermission_body) ENABLED START */
    var permissionRepository = repositoryFactory.getPermissionRepository();
    var promise = permissionRepository.deletePermission(id);

    promise.then(function (affectedRows) {
        if (!affectedRows || typeof affectedRows === 'undefined') {
            res.status(404).json(new Error('resourceNotFound'))
        } else {
            res.status(204).json('');
        }
    }).then(null, function (error) {
        logger.error(error.stack);
        res.status(500).json(new Error('unknownError(error));
    });
    /* PROTECTED REGION END */
};
```

Listing 4.2: An example of a fully implemented business method
4.1.2. Evaluation

This case study leads us to a very first impression of how MoDev works. In section 3.1 the requirements of the tool were exposed and in section 3.2 the architecture was presented. This section uses them to have some criteria to evaluate the tool. The evaluation is presented in two directions. First of all, this study case was used as a reference implementation to build MoDev (see 3.3.3) and to get an implementation of the software architecture. This reference implementation was built manually following a traditional approach. The most natural way to evaluate MoDev is comparing the traditional approach and the MoDev approach. A traditional approach is code-centric. That means, new features and maintenances are performed directly in the code. Sometimes a design phase comes before by using tools like UML editors. After coding, the documentation and testing phases come through. Figure 4-1 shows an activity diagram with the traditional approach and section 3.4 explains the MoDev approach.

![Activity Diagram](image)

Figure 4-1.: Developing with the traditional approach
At a first glance, there are almost the same quantity of steps to be executed in each approach. The first steps in both approaches are very similar and consist of the design of the application to be developed. Then, in the traditional approach there are four steps which are very time-consuming. Configuring, coding, testing and documenting are the more expensive tasks for developers. However, in the MoDev approach these four steps are practically condensed in the generation step, with some little extra effort in the code and tests completion phase. A big portion of the artefacts are automatically generated, including documentation. The rest of the phases are exactly the same in both approaches. Besides, it must be taken into account that following the MoDev approach, any change in the model causes multiple automatic changes in the rest of the artefacts. Nevertheless, with the traditional approach any change in the design must be performed completely step by step by the developer. An alternate way to compare both approaches would be drawing a comparison in terms of times of development in each case, in order to check the productivity. However, that would be a subjective comparison since the same developers accomplished both approaches, giving up considerations like technology learning curves and previous domain knowledge. An important consideration about the MoDev approach is that it changes completely the traditional way of building software. Model-driven development is not widely adopted in the industry (yet). New ways of making software sometimes causes rejection and cultural problems because developers are stepping out of their comfort zone.

The second direction of this evaluation refers to code metrics. In order to get some metrics automatically, this work uses SonarQube\(^3\). SonarQube analyses the code and presents results including lines of code (LOC), code quality, method complexity, percentage of documented code and unit test coverage. SonarQube supports multiple languages including the one needed for this work: Javascript. Table 4-3 presents a summary of the results produced by SonarQube. The first column indicates the name of the metric generated by SonarQube. The second column shows the results after generating the project with MoDev, without writing manually any code. The third column denotes the result when the application was finished by the developers completing the code manually. Having these results, MoDev can be evaluated in terms of the following criteria:

\(^3\)http://www.sonarqube.org/
Table 4-3: SonarQube metrics for the first study case

<table>
<thead>
<tr>
<th>Metric</th>
<th>Generated code</th>
<th>Finished application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Files analysed</td>
<td>35</td>
<td>36</td>
</tr>
<tr>
<td>LOC</td>
<td>1902</td>
<td>2665</td>
</tr>
<tr>
<td>Total Issues</td>
<td>264</td>
<td>201</td>
</tr>
<tr>
<td>Blocker Issues</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Critical Issues</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Major Issues</td>
<td>232</td>
<td>169</td>
</tr>
<tr>
<td>Minor Issues</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Info Issues</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Technical Debt</td>
<td>2d 4h</td>
<td>2d</td>
</tr>
<tr>
<td>Comments</td>
<td>29.5%</td>
<td>25%</td>
</tr>
<tr>
<td>Total unit tests executed</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>Unit tests passed</td>
<td>0%</td>
<td>100%</td>
</tr>
</tbody>
</table>

- **Productivity**: A good (but maybe not the best) metric for productivity in software projects is counting lines of code (LOC) of the application [23, 37]. Detecting how many LOC are automatically generated by MoDev and how many LOC had the project when finished can give a good factor of how much work the tool saves to the developers. Results shown in Table 4-3 indicates that 71.4% (1902) of the lines were automatically generated by Modev and 28.6% were manually written by the developer. This LOC corresponds to application, tests and documentation code. Although LOC is not a perfect metric for a real estimation on times and efforts, it is true that without MoDev developers must additionally write around 70% of fully functional code without errors; besides from certain quality topics discussed later. Another factor of productivity shown in the results is the percentage of comments. 25% of the comments were generated with little effort. This is a good number taking into account that SonarQube calculates it according to the total number of lines, including blank lines, imports, includes, package declarations, among others. Besides from keeping comments in total synchronization, having them around *for free* definitely increases productivity (see Figure 4-2 and 4-3).

- **Quality**: SonarQube raises an issue every time a piece of code breaks a coding rule. Ideally, a software team would not introduce any new issues, but in real life it is not always possible and sometimes it is not worth it. Five types of issues are reported, being blocker and and critical the ones that need to be resolved as soon as possible. Keeping major and minor issues as low as possible is considered a good practice. SonarQube reported zero critical and blocker issues in the two stages. In the second stage (when

---

4 [http://docs.codehaus.org/display/SONAR/Metric+definitions](http://docs.codehaus.org/display/SONAR/Metric+definitions)
developers completes the code manually ) quality of the project is responsibility of developers while in the first stage, MoDev has the entire responsibility. In terms of code quality, this results show that using MoDev helps developers to keep the code as best as possible. Moreover, every generated artefact is fully compliant with the architecture. That forces developers to respect the architecture as well.

- **Maintainability**: Maintainability refers to the ease with which an application can be maintained in terms of defects, repairs, new requirements meeting and efficiency, safety and reliability maximization. Although, maintainability is determined mostly due to the design of the application, MoDev can help to increase this easiness. For example, creating a new business method implies eight steps performed automatically without errors by MoDev and only by modifying the corresponding model and completing very specific parts:
  
  - create the file for the business module (if not already present)
  - create the business method skeleton
  - create the documentation of the business method
  - create the file for the tests of the business module (if not already present)
  - create the skeletons for business method’s test cases
Figure 4-3.: Example of generated documentation for REST endpoints

- create the corresponding route and its parameters
- create the documentation of the route endpoint
- create the documentation of the REST endpoint.

If these were made manually, it would require a considerable effort. In fact, the knowledge of which artefacts should be created/modified and where to put those artefacts without errors is required as well. Another benefit is that MoDev always knows which artefacts must be created, modified or removed. Sometimes developers forget that, leaving unused code in the base project or even worst, not creating other important artefacts.

When software projects are developed using MDD tools, maintainability refers not only to the application developed, but also to the MDD tool as well. This turns MoDev into a disadvantage because of the need of experts in the MDD field when an adjustment or new feature is needed in the tool. Besides from the additional time and budget required to do that. In any case, textual representations were chosen because of this problem as well. They are easier to maintain compared to graphical representations.

- **Flexibility**: One of the disadvantages of using model-driven development tools is the tool lock-in, that is, tying up an application to be developed and maintained with an
specific tool. Using MoDev can help a software project to be developed faster and better. However, if the team decides not to use it any more, the project can continue without any restriction or problem.

- **Ability to work on teams**: For the development of MoDev, Xtext was selected as the language workbench to develop domain-specific languages. Xtext works only with textual languages and at this moment there is no support for an easy integration with graphical models. There is a large discussion [12, 44, 42, 45] about what type of DSL to use in MDD tools. Despite of that, it is possible to highlight a great benefit of using textual models in this case. With textual representations, the use of a version control systems (VCS) like Git is as easy as it is used with code. Branches, tags, merges and diffs are naturally available for the developers, easing the work in a team. The same does not occurs with graphical notations because some of the use binary data, other use complex textual representations under the hood. This makes versioning and above all merging a difficult task in software teams using VCSs.

- **Testability**: Testability is mostly determined by the design of the application and some rationales exposed by the software architecture. However, MoDev helps developers to not forget implementing tests after (or even before) developing business cases. This help comes if the form of the generation of configuration and skeletons of unit test cases over the main functionalities. In this way, testing an application would not be anymore an optional step.

- **Technology portability**: The models in MoDev were developed to be independent of any specific framework or language. This allows to port implementations into different technologies using the same model and different code templates or transformations. For example, the tool allows to define business implementation through operations, parameters and data types. In MoDev business implementations were made for Node.js and the express framework. However, it is easy to create multiple templates to support it in technologies like Enterprise Java Beans (EJB) or Spring using the same abstractions.

- **Time to adopt this new approach**: This is an issue that should be taken into account when using model-driven development tools applied in real software projects. As there would exist additional languages (DSLs) and tools to learn, this additional learning curve must be assumed by the team. However, as the languages were kept as simple and intuitive as possible, and the editor provide a lot of features to ease language manipulation, this additional training time is considerably low compared with other GPLs and graphical tools.

- **Domain expression**: Although the developed models represent an abstraction of the software architecture to generate applications, it would be better if the models had more terms related to the specific domain. This leads to a direct understanding of the
models by people not necessarily involved in technology details. However, as MoDev aims to abstract proper software architectures for multiple domains, such separation between domain and software architecture could be harmful as well.

- **Architecture conformance**: *MoDev* can also be used as an architectural conformance tool. It forces developers to create architecture components like code, tests, documentation and configuration in the way the architecture have defined. Any modification in the software architecture can be easily implemented in *MoDev* as well.
5. Conclusions

In this work, the applicability of model-driven development to increase, above all, the productivity and quality of applications in software teams was researched. A tool called MoDev was developed following a model-driven software development iterative methodology consisting of the following phases: prototyping, reference implementation, analysis, meta-modeling, DSL design, model transformations building, DSL editor building and testing. MoDev requires well defined software architectures to work. Initially an illustrative software architecture was specified to test MoDev. However, any abstractable software architecture can be implemented using the tool. Moreover, MoDev was created with independent and composable models to enable reusing. That means, that components of some architecture can be used in another one and code generation templates can be customized to support any platform, technology or language. The creation of applications using MoDev is relatively easy. The analysts, architects and developers create a set of models, then create a main model importing the previous ones. The next step is to trigger the generation with a simple click requiring just seconds to get a lot of code automatically generated. Then, developers complete the code and tests manually. Once the code is fully functional, continuous integration and continuous delivery processes are recommended. Compared to a traditional development approach, which is code-centric, developing applications with MoDev involves the creation of models and the completion of artefacts which are automatically generated. It means that the process is model-centric. This change in the paradigm has to be taken into account when creating applications using model-driven engineering in software development teams.

A case study was delivered proving that MoDev leads software teams to significant improvements in multiple aspects and other few drawbacks that should be resolved in future research (see Table 5-1). In terms of productivity, approximately 70% of the code of the finished case study was automatically generated. In addition, other artefacts like configuration files and documentation were generated. Regarding quality, an automated software metrics tool was used to show that the generated code has no important quality issues. Besides, every generated artefact is fully compliant with the architecture, forcing developers to respect them as well. Concerning maintainability, an example of which tasks are necessary to create a new business method and the implications of doing the same work manually were illustrated. MoDev does not only generate artefacts, but it really knows what changes to apply and where. With MoDev the problem of tool lock-in does not occur and as it uses textual languages the ability to model and code in large software teams is increased. In regard to
to testability, it helps by making the tests processes easier as it generates the skeleton of test modules and tests cases for every modelled method. Finally, respecting portability, as models were developed independent of technologies, model-to-text transformations can be performed for any platform.

Contrary to many related work and tools found in the literature, *MoDev* not only serves as a starting point for software projects, that means, it does not only generate the project’s base structure from scratch but it can be used iteratively and multiple times during the entire life cycle of an application. This is achieved thanks to the protected regions which allow re-generations. However, during the development of the case study, it was noted that changing or modifying accidentally these protected regions can be a bit harmful. Developers should pay attention to this because protected regions should keep always the same structure and identifiers.

Two additional negative points were detected in this solution. First, it was not tested in multiple and bigger case studies. This is a weakness because the explained benefits could not be as bigger as with other applications. The same occurs for the people involved in the development of the case study. Maybe other developers could have different impressions of what the tool offers. On the other hand, *MoDev* was implemented with independent models

<table>
<thead>
<tr>
<th>The good</th>
<th>The bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Improvements in productivity, quality, maintainability, flexibility, testability, portability and the ability to work in teams</td>
<td>Some minor drawbacks in terms of maintainability because of the use of protected regions.</td>
</tr>
<tr>
<td>Serves not only as starting point for a project. It is useful in the entire project lifecycle.</td>
<td>Training required to learn new languages (DSL)</td>
</tr>
<tr>
<td>It is a generic solution and it is not heavily scoped to a particular domain.</td>
<td>It was not applied with different software architectures.</td>
</tr>
<tr>
<td>It is (relatively) easy to be extended and reused.</td>
<td>Lack of domain expressiveness</td>
</tr>
<tr>
<td>One model, multiple implementations.</td>
<td></td>
</tr>
<tr>
<td>Architecture conformance.</td>
<td></td>
</tr>
</tbody>
</table>
that allows to be reusable, composable and extensible to different and heterogeneous software architectures. However, the second negative point comes from this because just one illustrative software architecture was taken as a reference. Having multiple and different software architectures could prove that MoDev is really reusable, composable and extensible as expected.

According to these results, MoDev can be initially used in production environments in small software projects and within small software teams. Once tested and evaluated in this kind of projects it can be improved and maintained to help the development of big projects in large teams.

5.0.3. Future work

Two directions are proposed for future research of this work. First of all, to extend MoDev in order to resolve some of the drawbacks explained in the previous section. Applying this tool in real software teams in real companies with many developers could give this research additional support to be implemented in big projects. Secondly, extending the tool to support heterogeneous software architectures would provide additional reasons for the adoption of the tool.

Secondly, and looking forward to more interesting researches about model-driven development, MoDev lays the foundations for more extensible approaches. For example, Cabot et. al. [15] and Shroff et. al. [49] combine MDD with cloud-computing. Taking advantage of the approach followed by MoDev, where models are created independent in order to be reusable, a cloud repository of models can be created. This repository can publish a directory of languages that any developer can use, extend and modify to create customized software architectures. Another topic related to cloud computing has to do with language editors. At this moments, languages have to be manipulated within Eclipse. However, recent approaches use web technologies to provide online editors with features like offline edition, online collaboration, integration with online services like github and tight integration with web browsers and mobile devices. Having online editors and a models directory would be a great start to collaborative modeling and generation services in the cloud.
A. Appendix: Meta-models developed for the software architecture

Figure A-1.: The document meta-model

Figure A-2.: The DB-Config meta-model
Figure A-3.: The Mongo-Config meta-model

Figure A-4.: The entity meta-model

Figure A-5.: The repository abstract meta-model
A Appendix: Meta-models developed for the software architecture

Figure A-6.: The document-repository meta-model. An extension of the repository meta-model.

Figure A-7.: The relational-repository meta-model. An extension of the repository meta-model.

Figure A-8.: The business meta-model.
Figure A-9.: The route meta-model.

Figure A-10.: The dto abstract meta-model.
B. Appendix: DSL instances created for the case study

```plaintext
documents-module auth

schema User {
  name String required
  lastName String required
  email String required unique index
  password String required
  salt String required
  tokens schema Token {
    Token String
    salt String
    expiration Number
  } *
  role ref Role
}

schema Permission {
  name String required index
  httpVerb String required
  url String required
  group schema PermissionGroup {
    name String
description String
  }
compound-index uniquePermission {
  attribute httpVerb 1
  attribute url 1
  unique
  }
}

schema Role {
  name String required unique index
  permissions ref Permission *
}
```

Figure B-1.: The document DSL instance for the case study
Figure B-2.: The repository DSL instance for the case study

```plaintext
repositories-module auth {
  repository UserRepository belongsTo auth.User {
    operation getAllUsers []
    operation createUser []
    operation updateUser []
    operation deleteUser []
  }

  repository RoleRepository belongsTo auth.Role {
    operation getAllRoles []
    operation createRole {
      parameter role
    }
    operation updateRole []
    operation deleteRole []
  }

  repository PermissionRepository belongsTo auth.Permission {
    operation getPermissionsByUser {
      parameter user
      parameter offset
      parameter limit
    }
    operation getAllPermissions []
    operation createPermission []
    operation updatePermission []
    operation deletePermission {
      parameter id
    }
  }
}
```

Figure B-3.: The business DSL instance for the case study

```plaintext
business-module auth {
  business PermissionBusiness []

  business RoleBusiness {
    operation getAllRoles returnType auth.RoleDto * []
    operation createRole returnType auth.RoleDto {}
    operation updateRole returnType auth.RoleDto {}
    operation deleteRole []
  }

  business UserBusiness []

    operation getAllUsers returnType auth.UserDto * []
    operation createUser returnType auth.UserDto {}
    operation updateUser returnType auth.UserDto {}
    operation deleteUser []
  }
```
Figure B-4.: The layeredApp DSL instance for the case study
Bibliography


[22] Erdweg, Sebastian; Storm, Tijz; Boersma, Meinte; Bosman, Remi; Cook, WilliamR.; Gerritsen, Albert; Hulshout, Angelo; Kelly, Steven; Loh, Alex; Konat, Gabriël; Molina, PedroJ.; Palatnik, Martin; Pohjonen, Risto; Schindler, Eugen; Schindler, Klemens; Solmi, Riccardo; Vergu, VladA.; Visser, Eelco; Vlist, Kevin; Wachsmuth, GuidoH.
Bibliography


[53] Stefani, Antonia; Xenos, Michalis: *A Model for Assessing the Quality of e-Commerce Systems*. 2001


[56] Wang, Chao; Li, Hong; Gao, Zhigang; Yao, Min; Yang, Yuhao: An automatic documentation generator based on model-driven techniques. In: *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on* Bd. 4, 2010, S. V4–175–V4–179

[57] Xiao-Jun, Ma; Xi, Yan: A web application system development platform based on role-based access control model. In: *Computer Application and System Modeling (ICCASM), 2010 International Conference on* Bd. 12, 2010, S. V12–166–V12–169