INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Model-Driven Development for Spring Boot Microservices

BRUNO COSTA MENDONÇA
junho de 2021

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Model-Driven Development for Spring Boot Microservices

Bruno Costa Mendonça

Dissertation to obtain the master's degree in Informatics Engineering, Area of Expertise in Software Engineering

Supervisor: Alexandre Bragança (ATB)

Porto, June 2018

Dedicatory

“I dedicate this work to all those who supported me, family, friends and teachers.”

Abstract

As microservices become more and more common, there is more interest in optimizing their development, making it faster, reducing the complexity and making it less error and bug prone.

This work aims to explore how Model-driven Engineering (MDE) can be used to aid microservices' development, especially Java microservices using Spring Boot.

Firstly, this work starts by presenting its context, with a brief introduction to MDE and microservices, and how MDE can be used to facilitate microservices' development.

Then the State of the Art presents MDE base technologies and MDE-based solutions that can be used to develop microservices.

Furthermore, a value analysis was also done, to explore the benefits of using MDE, and to define its value proposition.

Lastly, a case study was elaborated following three different approaches, traditional development, DSL-based approach, and MDE-based tool approach, which were then compared by code quality and time required for development of the microservices.

This work is especially interesting for someone who wants to develop Java microservices and wants to explore the different approaches and technologies to do so, namely by using MDE.

Keywords: Microservices, Model-driven Engineering, Value analysis, Approach comparison

Resumo

À medida que os microserviços se tornam cada vez mais comuns, há mais interesse em otimizar seu desenvolvimento, otimizando o tempo de desenvolvimento, reduzindo a complexidade e tornando-o menos sujeito a erros e bugs.

Este trabalho tem como objetivo explorar como é que *Model-driven Engineering* (MDE) pode ser usado para auxiliar o desenvolvimento de microserviços, especialmente microserviços Java usando Spring Boot.

Em primeiro lugar, este trabalho começa apresentando seu contexto, com uma breve introdução ao MDE e microserviços, e explicando como é que MDE pode ser usado para facilitar o desenvolvimento de microserviços.

Em seguida é apresentado o Estado da Arte, que apresenta tecnologias base de MDE e soluções baseadas em MDE (ferramentas), que podem ser utilizadas para desenvolver microserviços.

Além disso, também foi feita uma análise de valor para explorar os benefícios do uso de MDE, e para definir sua proposta de valor.

Por fim, foi elaborado um caso de estudo através de três abordagens diferentes, desenvolvimento tradicional, abordagem baseada em DSL, e abordagem utilizando uma ferramenta baseada em MDE, que foram então comparadas através da qualidade do código e pelo tempo necessário para o desenvolvimento dos microserviços.

Este trabalho é especialmente interessante para quem quer desenvolver microsserviços Java e quer explorar as diferentes abordagens e tecnologias para o fazer, nomeadamente através da utilização de MDE.

Palavras-chave: Microserviços, *Model-driven Engineering*, Análise de valor, Comparação de abordagens

Table of contents

Figure List

Table List

Acronyms e Symbols

Acronyms' List

AHP	Analytic Hierarchy Process
AOP	Aspect Oriented Programming
API	Application Programming Interface
AST	Abstract Syntax Tree
ATL	ATL Transformation Language
CASE	Computer-aided Software Engineering
CBO	Coupling between object classes
CR	Consistency Ratio
DDD	Domain-driven Design
DIT	Depth of Inheritance Tree
DSL	Domain-specific Languages
DSML	Domain-specific Modelling Languages
DTO	Data Transfer Object
FFE	Fuzzy Front End
GPL	General-purpose Language
JDL	JHipster Domain Language
JDT	Java Development Tooling
LCOM	Lack of cohesion in methods
MDA	Model-driven Architecture
MDD	Model-driven Development
MDE	Model-driven Engineering
MDSL	Microservice Domain-Specific Language
MPS	Meta Programming System

MSDK	Microsoft SDK for Visual Studio
NCD	New Concept Development
NOC	Number of children
NPD	New Product Development
OCL	Object Constraint Language
QFD	Quality Function Deployment
REST	Representational state transfer
RFC	Response for a Class
SSD	Software Sequence Diagram
WMC	Weighted methods per class

1 Introduction

This chapter provides an introduction on the Monolithic architectural style, the Microservices architectural style, and their differences.

In addition, this chapter introduces Model-driven Engineering (MDE), and how it may be used to support microservices' development.

Furthermore, this chapter presents the problems with traditional microservice development process, and briefly explains the approaches that are analysed and compared in the later sections, as well as the expected results.

1.1 Context

This section introduces Microservices and Model-driven engineering, to ease the understanding of this work.

1.1.1 Microservices

The microservices architectural style is an architectural style that is becoming more and more common. It was inspired by service-oriented computing and is used for development of server-side applications (Dragoni et al., 2017). It is an approach for developing a single application as a suite of small services (microservices), with a bare minimum of centralized management (Martin Fowler & James Lewis, 2014).

A microservice is a component of an application and should be cohesive and independent. However, as microservices may need to communicate with each other, for a microservice architecture application to function correctly it needs to be well coordinated, or the system may be unstable or misbehave.

The Microservices architectural style poses as an alternative to the traditional development architecture (The Monolithic style), where applications are usually built with three main parts, the client-side user interface (frontend), the server-side application (backend) and a database (Martin Fowler & James Lewis, 2014).

The server-side application handles the requests and executes the domain logic. It is common to see server-side applications built as a monolith, a single logical executable, which to update requires the building and deploying of the entire application.

Although monolithic applications can be successful, they have several drawbacks, such as (Dragoni et al., 2017):

- Large-size monoliths are hard to maintain and evolve due to their complexity.
- Monoliths can suffer from "Dependency Hell", in which adding or updating libraries can result in several problems, namely dependencies incompatibility, and/or inconsistent systems.
- Changes require reboot of the whole application which for large projects has a lot of consequences, leading to high downtimes and affecting development itself.
- One-size-fits-all configuration. Although the components of the system may have different resources requirements, in a monolith architecture, the developer must choose a one-size-fits-all configuration, which will likely be sub-optimal with respect to the individual modules.
- Monoliths limit scalability. The most common approach to scale a monolithic application is to start a new instance of the entire application (monolith) and balance the load among the said instances. However, it is possible that only a sub-part of the system is being stress, but it requires a new instance of the whole application, which leads to waste of resources and limits scalability.
- Technology lock-in for developers. Most monoliths are based on a single framework and its bound languages, which makes hard for developers to add new technologies and consequently maintain the project.

These drawbacks have led to a new architectural style, Microservice architecture style, which has characteristics that cope with the previous listed monolithic architecture drawbacks, due to the following characteristics (Dragoni et al., 2017):

- As microservices are cohesive and implement a limited number of functionalities, their code base is smaller, which limits the bug scope. Furthermore, by being independent it is possible to test the microservices in isolation easing the bug investigation.
- It is possible to gradually migrate to a microservice architecture, by gradually converting parts of the monolith to microservices.
- By being independent, to deploy a change in a module of the microservice architecture, a reboot of the entire system is not necessary, only the

microservices of that module need to be reboot, which leads to smaller downtimes.

- Microservices are prone to containerisation, which gives developers more freedom in the configuration of the deployment environment, allowing for resource optimization.
- The scaling of a microservice architecture does not imply the scaling of all its components, as microservices are independent and cohesive they can be individually scaled with respect to their load, improving for resource optimization.
- Another advantage of microservice's independency is that there is no technology lock in (apart from the communication mechanisms of the system). Developers have more freedom and can easily adapt to the technology needs for the implementation of each service, allowing for improved performance.

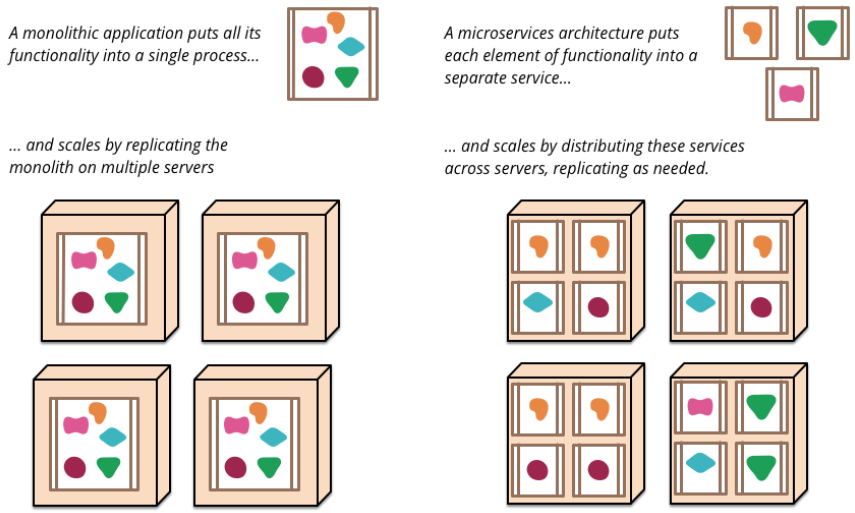

Figure 1 – Monoliths and Microservices (Martin Fowler & James Lewis, 2014)

1.1.2 Model-driven Engineering (MDE)

For a long time, software engineers have been trying to facilitate software development by creating abstractions to help them program in terms of design rather than computer environments and technologies themselves (Schmidt, 2006).

Today, although advances in languages and platforms during the past decades have raised the level of software abstractions, software developers still spend considerable time and effort

developing and maintaining applications using third-generation languages and manually porting application code to different platforms or newer versions of the same platform.

Third-generation languages are high-level computer programming language, more user friendly and machine independent than second generation languages, but with a less specific focus when compared to fourth and fifth generations. Examples of third-generation languages include Java and C#.

One problem with developing software (namely microservices) with third-generation languages is that third-generation languages require developers to pay so much attention to the numerous tactical imperative programming details that they lose focus on the strategic architectural issues such as system-wide correctness and performance.

One possible solution to this problem is to use Model-Driven Engineering technologies that combine Domain-specific Modelling Languages (DSML), transformation engines and generators.

However, it is to note that MDE does not come without drawbacks, there is an initial high cost related to developing or adopting tools and transformations.

MDE is a long-term investment and needs customization of environment, tools and processes, and training (Mohagheghi et al., 2009).

1.1.2.1 MDE, MDD and MDA

There are multiple terms used to describe approaches that focus on models. A good and simple definition is the one by David Ameller (Whittle et al., 2014), which defines them as:

MDD (Model-driven development): MDD is a subset of MDE and focuses on the generation of implementations from models.

MDE (Model-driven engineering): MDE includes other uses of precise models to support the development process, such as model-driven reverse engineering and model-driven evolution.

MDA (Model-driven architecture): MDA is a particular form of MDD that uses the Object Management Group's (OMG) standards.

There are several MDE techniques, for instance (Voelter, 2013):

Model transformation – transform models into models.

Code generation – transform models into text (usually source code, XML, or configuration files).

Model interpretation – an interpreter traverses an abstract syntax tree and directly performs actions depending on its contents.

1.1.2.2 Domain-specific Modelling Languages (DSML)

A domain-specific language (DSL) is a computer language specialized to a particular domain or context, in contrast to a general-purpose language (GPL), which as the name implies is general and applicable across domains.

As defined by Jeff Gray, "DSLs allow a programmer to concisely state a problem using abstractions and notations that closely fit the needs of a specific domain" (Gray et al., 2008), bridging the ap between the domain and the implementation.

There are different types of DSLs, namely domain-specific markup languages, domain-specific modelling languages (DSML), domain-specific programming languages.

Some DSL examples are HTML, Logo, VHDL, Mathematica, SQL.

Examples of GPL are UML, Petri-nets, or state machines.

1.2 Problem

Although microservices are becoming more and more common (especially Java micro-services) their development becomes a repetitive task that can consume lots of time. Also, microservices' development can be complex and subject to several types of errors.

The main problems of the microservice architecture are (Richardson, 2018):

- Finding the right set of services is challenging.
- Distributed systems are complex, which makes development, testing, and deployment difficult.
- Deploying features that span multiple services requires careful coordination.
- Deciding when to adopt the microservice architecture is difficult.

To make microservices' development faster and easier it may be possible to use MDE concepts and technics to develop the microservices with significantly reduced complexity.

Furthermore, as a result of abstraction the developer can pay more attention to the architecture than the underlaying frameworks and languages, focusing more on cross-cutting concerns like performance and correctness.

1.3 Objectives

The main objective of this dissertation is to explore if and how model-oriented approaches (MDE), and in particular domain-specific languages (DSL), can help minimize the complexity inherent in microservices-based solutions (specifically Java micro-services using Spring Boot), supporting the design and code generation.

It is expected to evaluate three different approaches based on domain models and MDE, namely:

- Traditional development, consisting in manual development of the microservices, with no support from DSLs or MDE techniques.
- Developing a DSL (that allows the modeling of microservices) from scratch, and generating code for microservices, using MDE tools (for example the Eclipse Modeling Framework).
- Using MDE-based tool(s) that support code generation of microservices based on a DSL (for example an approach consisting in the usage of ContextMapper and JHipster).

These approaches will focus on generating simple microservices, namely the more common use cases, the create, reading, updating, and deleting of entities.

It would be interesting to generate a mechanism of communication between the microservices, as well as other aspects of microservices, but it is not fit for the time scope available for this work, and it may be done on future refinements.

1.4 Approach Taken

The different approaches are expected to generate microservices for the same scenario, to do so a use case must be designed. This use case must be fit for a microservices-based solution.

Furthermore, the approaches must be analysed and compared, taking into consideration the code quality of the resulting microservices (using the Chidamber and Kemerer's metrics (Chidamber & Kemerer, 1994)), and the time required for the microservices' development.

1.5 Expected Results

This document aims to identify benefits and drawbacks of developing Java microservices using MDE. It is expected this work as interesting results for people who want to develop microservices, and want to know if MDE is a possible solution to facilitate the development for their specific scenario.

To do so, a case study must be developed, using the approaches specified in the Objectives section.

The approaches will be compared and analysed in order to discover if there is a significant different between the approaches in the development time and obtained code quality.

The code quality between the traditional development and DSL-based approach is expected to be similar, as the DSL-based approach attempts to generate code as much similar to the traditional development as possible.

For the MDE-based tool approach there is less control on the code generation process, and so it is expected that the codebase differs significantly, possibly also differing in code quality.

2 State of the Art

This chapter provides a state of the art about model-driven technologies and frameworks that have been developed, as well as MDE-based solutions that support code generation from the domain.

Additionally, this chapter provides some insight on related work, namely Graphical and Textual Model-Driven Microservice Development (Rademacher et al., 2020) and MicroBuilder: A Model-Driven Tool for the Specification of REST Microservice Architectures (Terzić et al., 2017).

2.1 MDE base technologies

MDE base technologies support the development of custom DSLs, as well as code-generation. Different frameworks may have different characteristics and support different MDE techniques.

2.1.1 Modeling SDK for Visual Studio

Modeling SDK for Visual Studio (*Modeling SDK for Visual Studio - Domain-Specific Languages*, 2016) is a tool developed by Microsoft that allows the creation of powerful model-based development tools that can be integrated in Visual Studio.

Modeling SDK for Visual Studio (MSDK) main features are:

- A graphical interface to define a domain-specific language (DSL) in form of a model.
- A model implementation with a strongly-typed API that runs in a transaction-based store.
- A tree-based explorer.
- A graphical editor in which users can view the model or parts of it that it defines.
- Serialization methods that save the models in readable XML.

- Facilities for generating program code and other artifacts using text templating.

The Modeling SDK for Visual Studio is used by creating a Domain-Specific Language Designer project in Visual Studio, which has the layout represented on Figure 2.

The created project has two parts:

- **Dsl project** – The project that contains the code that defines the domain-specific language.
- **DslPackage project** – The project that contains the code that allows instances of the defined domain-specific language to be created and edited (in a separate instance of Visual Studio).

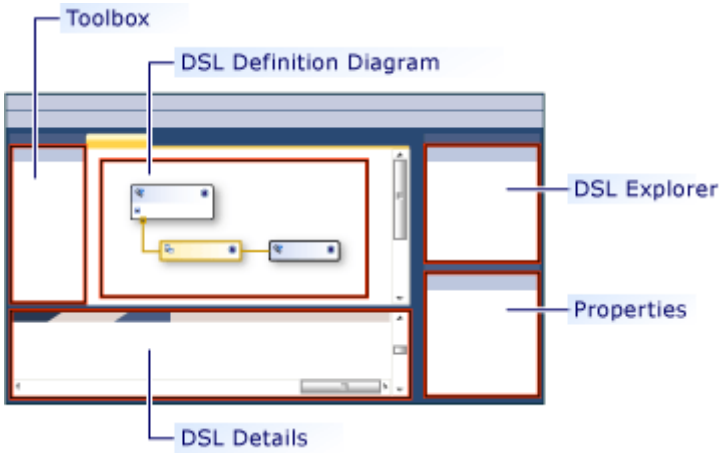

Figure 2 – Modeling SDK for Visual Studio's Layout (*Modeling SDK for Visual Studio - Domain-Specific Languages*, 2016)

A domain-specific language can be defined by dragging items from the toolbox and filling their defined properties, for example a domain class requires a name, and can have domain properties (fields that have name and type).

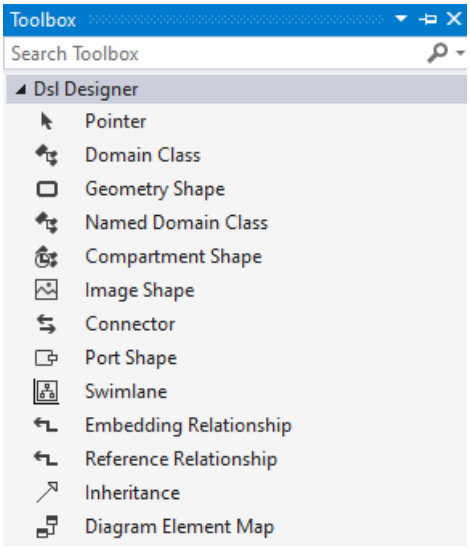

Figure 3 – Modeling SDK for Visual Studio's toolbox

In order to create instances of the domain-specific language it is also necessary to create the shapes of each Domain Class to be instanced, which is done by dragging the Geometry Shape from the toolbox to the Diagram Elements tab. It is also required to link the Domain Class to the Geometry Shape using the Diagram Element Map tool, and mapping the fields in the Decorator Maps tab. An example can be viewed on Figure 4.

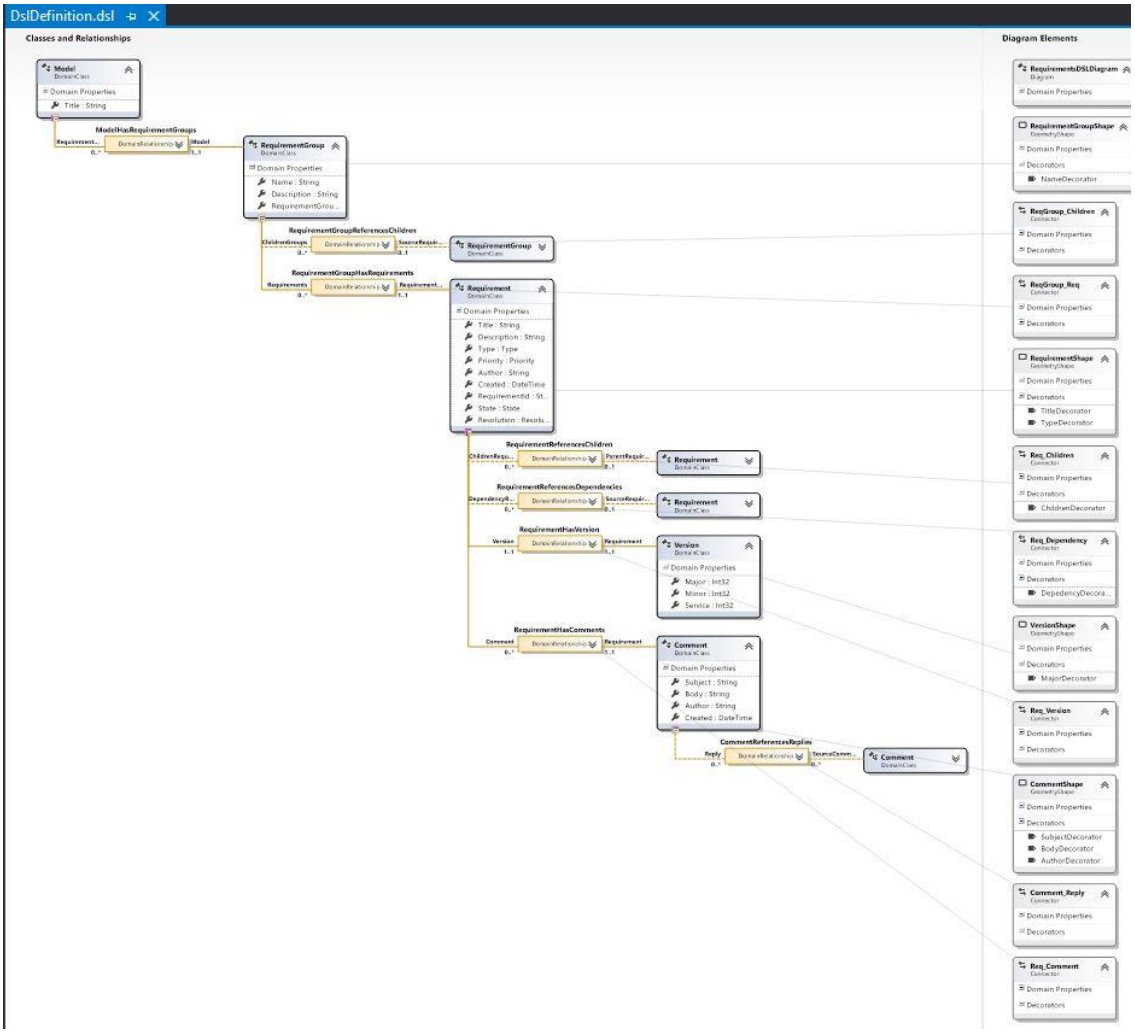

Figure 4 – Requirements DSL using Modeling SDK for Visual Studio

It is possible to add validations to the domain-specific language, by adding a partial class in the Dsl project, of which an example for the Requirement class can be seen in Figure 5.

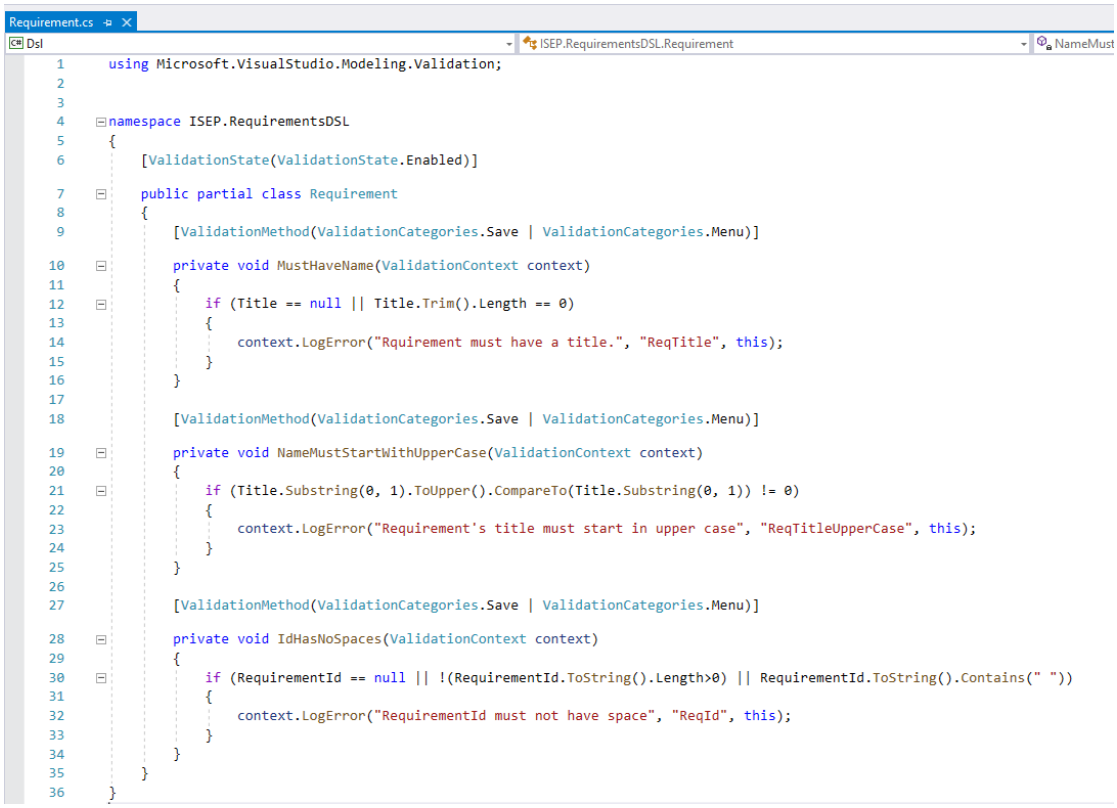

Figure 5 – Requirement's validations using Modeling SDK for Visual Studio.

To create instances of the specified domain-specific language its first required to create the tools used to instance the domain classes, which is done by creating tools in the Editor component of the domain-specific language Figure 6.

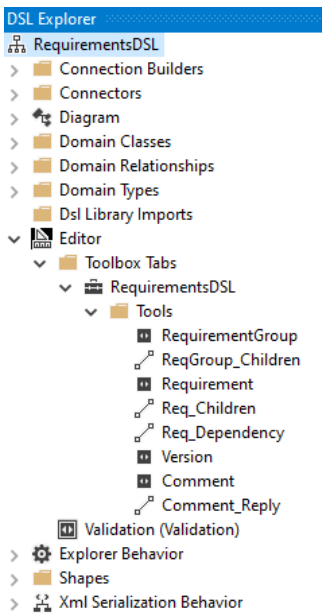

Figure 6 – Modeling SDK for Visual Studio Requirements' tools

A instance of the defined domain-specific language can then be created by building the DSL, using the “Transform All T4 Templates” button and executing the DSL project, launching a new instance of Visual Studio.

On this new instance of Visual Studio its possible to create an instance of the DSL by dragging items from the tools previously defined (Figure 7) and filling the properties for each domain class.

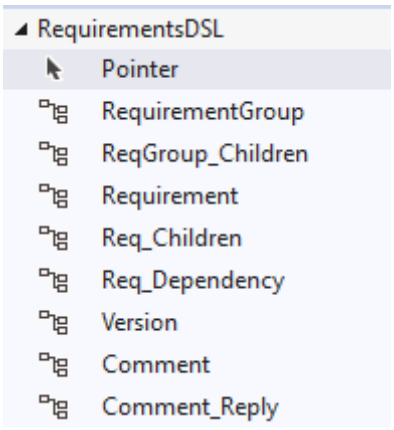

Figure 7 – RequirementsDSL’ tools

An example of a RequirementsDSL’s instance can be seen in Figure 8.

Modeling SDK for Visual studio does not support direct model-to-model transformation, however it supports model-to-text transformation, by creating a Text Template file and saving it (which will execute the defined transformation).

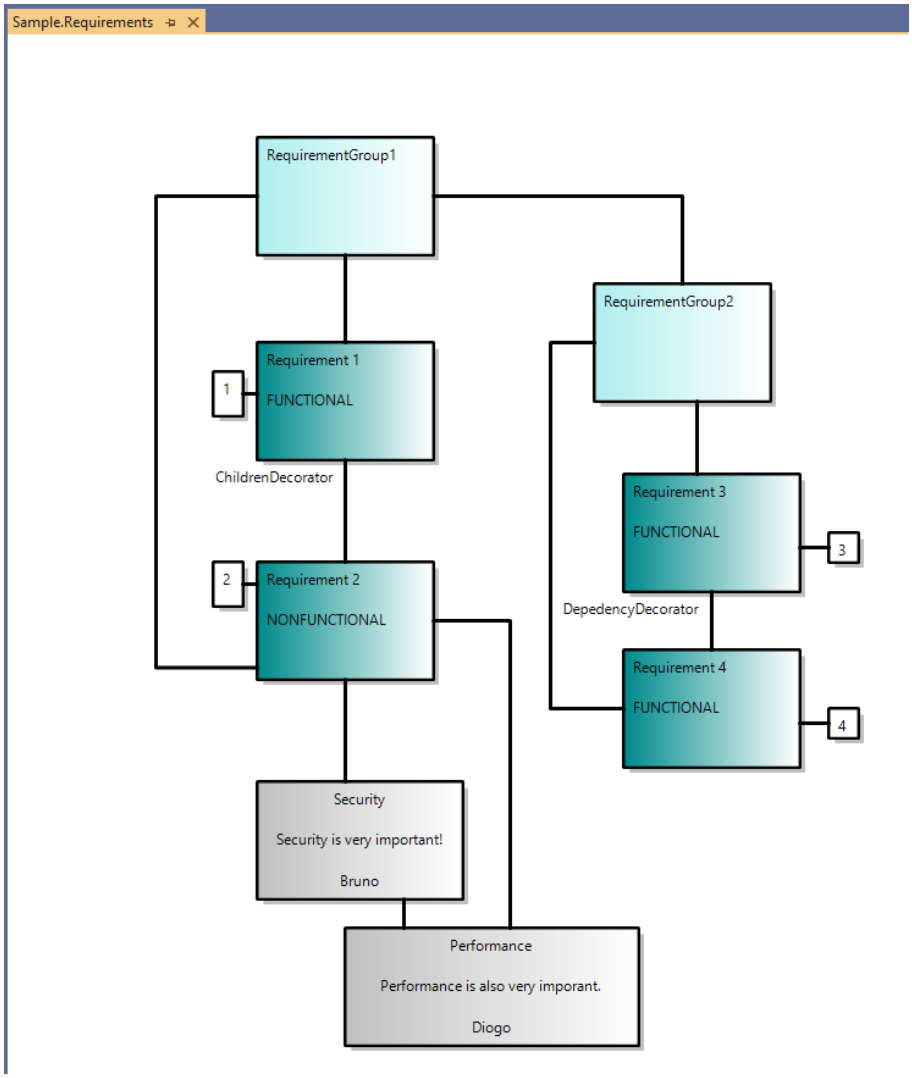

Figure 8 – RequirementsDSL's instance

2.1.2 Meta Programming System (MPS)

MPS is a is a language workbench developed by JetBrains (*MPS*, 2021). MPS is open source with the code available on github (https://github.com/JetBrains/MPS).

MPS focuses on DSL designs and uses projectional editor which allows the use of non-textual notation, including math notations, diagrams, and forms.

MPS differentiates itself with other language workbenches by avoiding the text-form. In MPS programs are always represented as an AST (Abstract Syntax Tree) (*Basic Notions | MPS*, 2021). The code is saved, compiled, and edited as an AST, which avoids the need to define the grammar and building a parser for the language. The language is defined in terms of AST nodes and rules.

The MPS projectional editor Figure 9 directly manipulates the AST, with no parsing required. The editor presents the AST to the user, the way the language author designed.

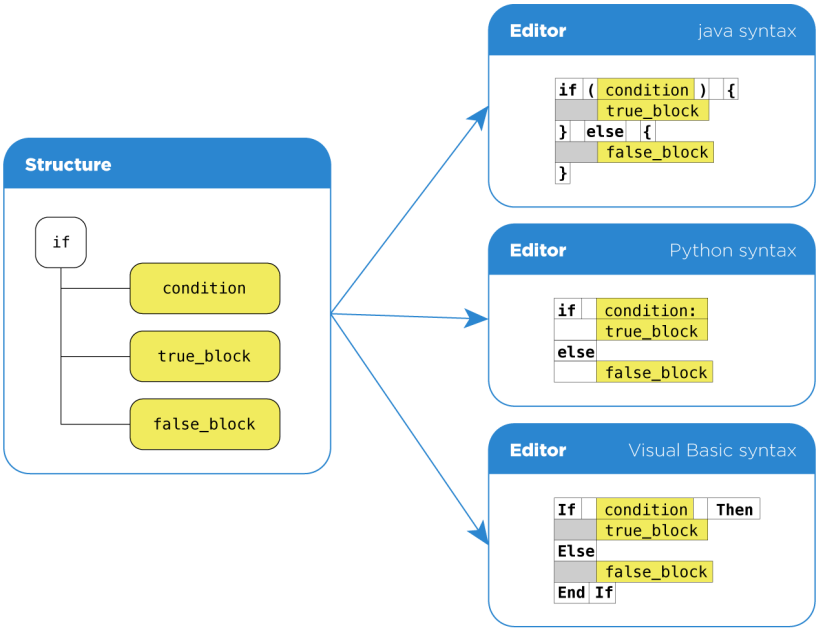

Figure 9 – MPS Projectional Editor (*Basic Notions | MPS*, 2021)

To create a domain-specific language using MPS its necessary to create a Structure project. The Structure defines the types of nodes (similar to domain classes, called Concepts in MPS) that can be instanced in a Model. Concepts have properties, children, and references, and can extend other Concepts and implement ConceptInterfaces. An example of a Requirement's Concept (with similar structure to the Requirement presented on chapters 2.1.1 and 2.1.3) can be viewed in Figure 10.

It's important to note that Requirement implements the "INamedConcept" Concept, which automatically adds a name property to the Requirement, this name property is used to name each instance of this concept created in a model. Furthermore, MPS's references do not support a "0..n" relationship, so in this case, in order to remain as similar to the Requirements defined in other language workbenches a "RequirementReference" concept had to be created, contained a reference to one specific requirement. Then its possible to add a "0..n" relationship from "Requirement" to "RequirementReference", as a children relationship.

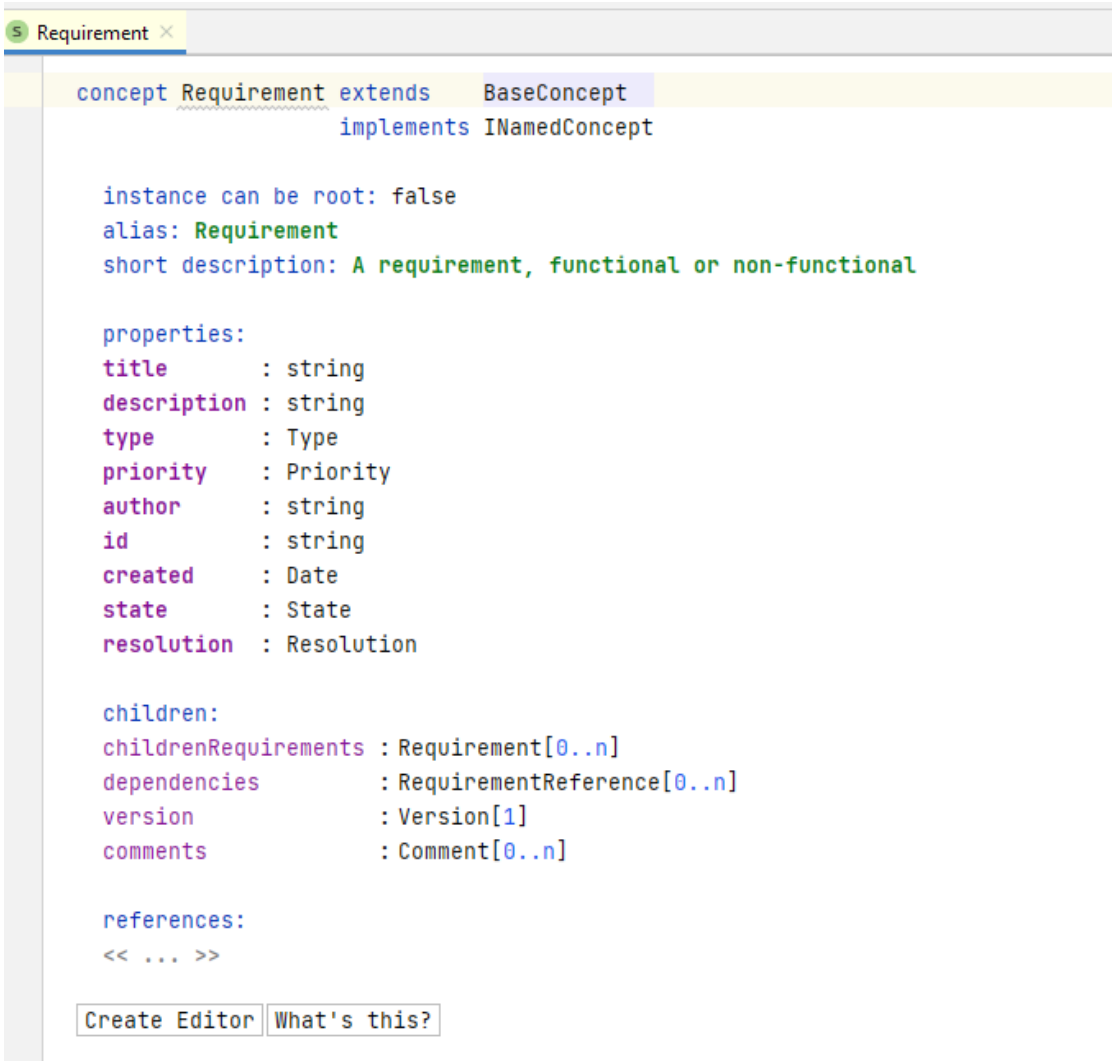

Figure 10 – MPS Requirement Concept

In MPS its possible to add validations (called constraints in MPS) to the structure. Constraints restrict the relationships between nodes, and define:

- The target scope for references.
- The situations in which a node can be child/parent/ancestor of another node.
- The allowed values for each property.
- The property accessor methods (getters and setters).

An example of Requirement's constraints can be seen in Figure 11.

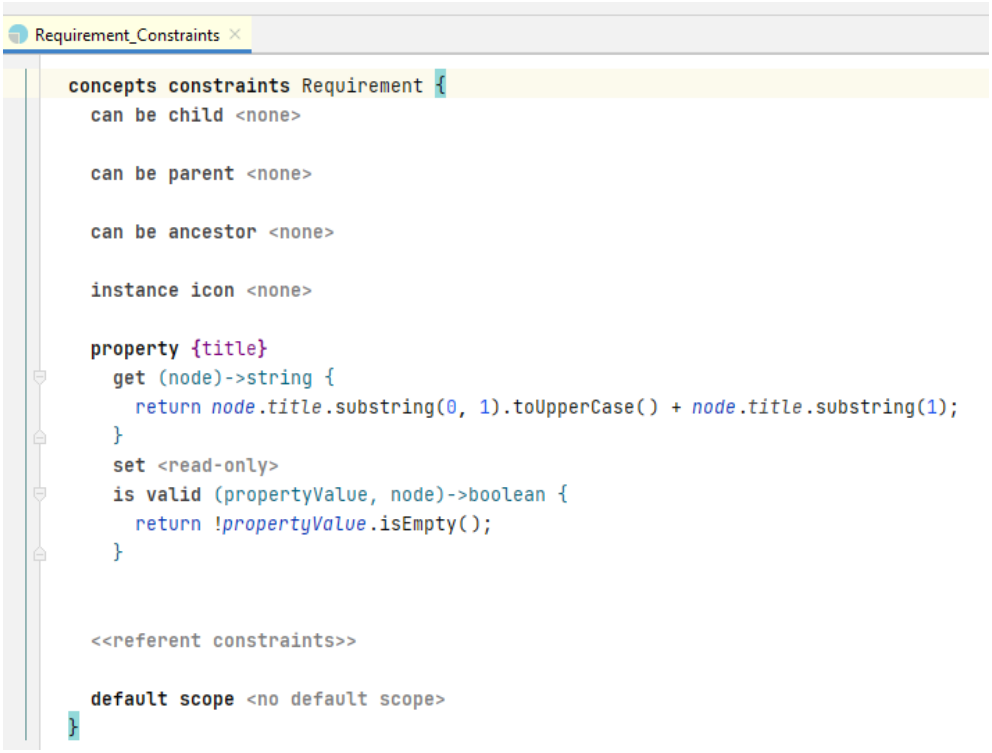

Figure 11 – MPS Requirement's constraints

To create an instance of the Model, is necessary a MPS' Sandbox project is required. In this project there can be several instances of the Model, defined in the language specified by the structure. The models are defined using the MPS' projectional editor (an example can be seen in Figure 12).

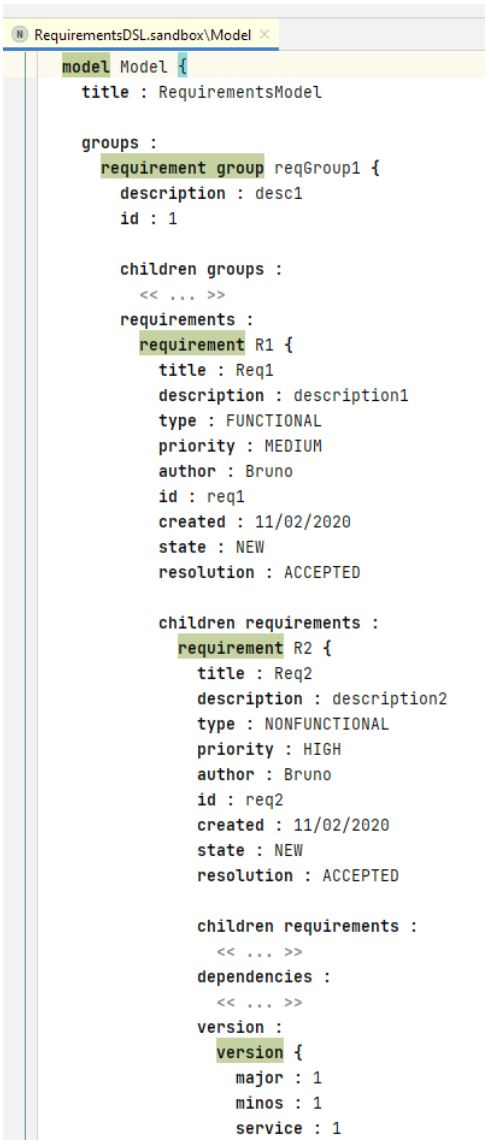

Figure 12 – MPS' Requirements' Model instance

MPS supports the generation of Java code. The code generation is done by sequential mode-to-model transformations, so that repeatedly concepts get reduced to the lower level of abstraction until the bottom-line level is reached (*Basic Notions | MPS*, 2021). The rules for translating concepts and their order are defined in the MPS' Generator.

The MPS'Generator uses mixed compilation/interpretation mode for transformation execution (*Generator | MPS*, 2021). Templates are interpreted and filled at runtime, but functions in rules, macros and scripts must be pre-compiled.

Templates are written in the output language (for example Java) and parametrized by referencing into the input model.

The applicability of each template is defined by the Generator rules (root mapping rule, pattern rule, etc), which are grouped in Mapping Configurations. Mapping Configurations form a single generation step and contain the Generator Rules.

An example of a mapping configuration can be viewed in Figure 13, and the Requirement's Generator template can be viewed in Figure 14.

Figure 13 – MPS' Generator mapping configuration

Figure 14 – MPS' Requirement Generator Template

The template represented in Figure 14 generates a class for each requirement, each with the class name being the name specified on the Model's instance. The two loops are node macros that loop through the requirement's dependencies and children, respectively. For each dependency or children, a new property is added to the Requirement, a property representing the children/dependency requirement. The "$[depReq]" and "$[childRep]" are property macros that compute the value for the property name The macro of "$[childRep]" can be seen in (Figure 15).

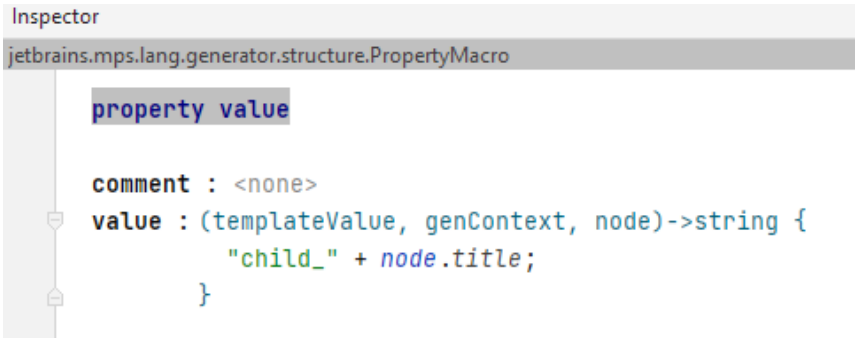

Figure 15 – MPS' property macro

The generation result is a project that contains all the generated classes in the output language (can be viewed in Figure 16). The Requirement generation's result can be viewed in Figure 17.

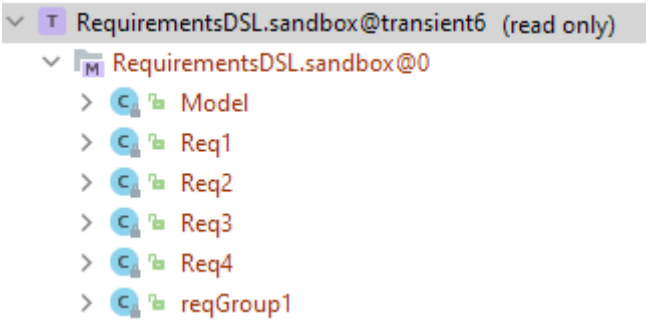

Figure 16 – MPS' generation result

```
public class Req1 {
  private String title;
  private String description;


  public Req2 child_Req2 = new Req2();

  public void Req1(String title, String description) {
    this.title = title;
  }

  public String getTitle() {
    return this.title;
  }

  public void setTitle(String newTitle) {
    this.title = newTitle;
  }

  public String getDescripton() {
    return this.description;
  }

  public void setDescription(String newDescription) {
    this.description = newDescription;
  }


}
```

Figure 17 – MPS' generated Requirement

2.1.3 Eclipse Modeling Framework (EMF)

EMF is an Eclipse-based modeling framework and code generation facility for building tools and other applications based on a structured data model (*Eclipse Modeling Project | The Eclipse Foundation*, 2021).

EMF provides tools and runtime support to produce a set of Java classes for the model, as well as a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor.

EMF (core) is a standard for data models and consists of three fundamental components:

- EMF – The core EMF framework, which includes a meta model (Ecore) for describing models and runtime support for the models.
- EMF.Edit – The EMF.Edit the is a framework used to build editors for EMF models and incudes generic reusable classes.

- EMF.Codegen – The EMF code generation is the component responsible for generating code. It can be used to generate everything needed to build a complete editor for an EMF model. EMF.Codegen includes a GUI and uses the JDT (Java Development Tooling) component of Eclipse.

EMF supports model validation using OCL (Object Constraint Language).

The Eclipse OCL provides parsers, evaluators, validators, code generator and debugger for OCL constraints and expressions on any EMF-based metamodel (*OCL/FAQ*, 2021).

```
class Model
{
    attribute title : String[?];
    property groups : RequirementGroup[*|1] { ordered composes };
    invariant mustHaveTitle: not title.oclIsUndefined();
    invariant idsAreUnique: groups->isUnique(id);
}
```

Figure 18 – OCL validations

On Figure 18 there are two OCL validation examples:

- mustHaveTitle – verifies that the model's title is not undefined.
- idsAreUnique – that verifies the ids of the model's requirements are unique.

Eclipse also supports model-to-model transformations, using ATL (ATL Transformation Language), which is a model transformation language and toolkit.

"ATL provide ways to produce a set of target models from a set of source models" (*ATL | The Eclipse Foundation*, 2021).

An ATL can be used to execute the transformation presented on Figure 19, transforming Requirements to UseCases.

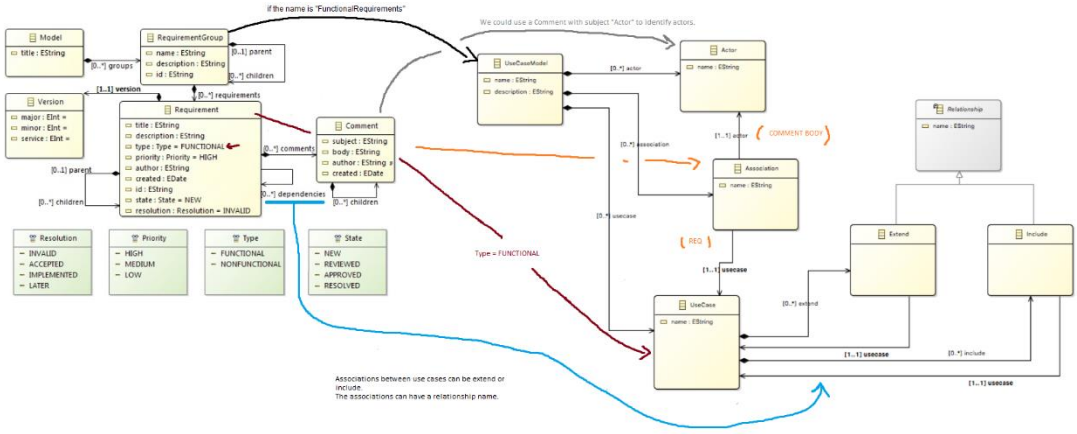

Figure 19 – Requirements to UseCases transformation

The rule displayed in Figure 20 could be used to create UseCases from Requirements.

```
rule UCFromReq {
    from
        req: MM!Requirement ( req.type = #FUNCTIONAL)
    using{
        incs : Sequence(MM1!Include) = Sequence{};
        exts : Sequence(MM1!Extend) = Sequence{};

        act : MM1!Actor = OclUndefined;
        ass: MM1!Association = OclUndefined;
    }
    to
        uc : MM1!UseCase(
            name <- req.title
        )


    do{
        for(reqDep in req.dependencies){
            if(reqDep.description = 'Includes'){
                incs <- incs.append(thisModule.newInclude(req.title, reqDep));
            }else if(reqDep.description = 'Extends'){
                exts <- exts.append(thisModule.newExtend(req.title, reqDep));
            }
        }
        uc.include <- incs;
        uc.extend <- exts;


        for(cm in req.comments -> select(com | com.subject='Actor')){
            act <- thisModule.newActor(cm);
            thisModule.actors <- thisModule.actors.append(act);
            thisModule.model.actor <- thisModule.actors;

            ass <- thisModule.newAssociation(act, uc, req.title+' .s actor is '+cm.body);
            thisModule.associations <- thisModule.associations.append(ass);
            thisModule.model.association <- thisModule.associations;
        }
    }
}
```

Figure 20 – Requirements to UseCases ATL rule

Eclipse also supports code generation or any model-to-text transformation, by using plugins, namely Acceleo (*Acceleo | Home*, 2021). An example of acceleo code generation file can be seen on Figure 21.

```
[module generate('http://www.example.org/usecases', 'http://www.example.org/requirements')]
[query public getRequirement(parm: String) : Requirement =
    invoke('pt.isep.edom.acceleo.inc4.main.ReqModelQuery', 'getRequirement(java.lang.String)', Sequence{parm})
/]
[template public generateUseCaseModel(anUseCaseModel : UseCaseModel)]
[comment @main/]
[file ('UseCaseModel.java', false, 'UTF-8')]
class UseCaseModel implements Executable {
    public void execute() {
        boolean exit=false;
        java.util.Scanner in = new java.util.Scanner(System.in);
        while (!exit) {
            [let actors: OrderedSet(Actor) = anUseCaseModel.actor]
                [if(actors->isEmpty())]
                System.out.println("I do not have any Actors.");
                [else]
                System.out.println("I have the following actors:");
                [/if]
                [for (actor: Actor | actors)]
                    System.out.println("    [actors->indexOf(actor)/]-[actor.name/]");
                [/for]
            [/let]

            System.out.println("0-Exit");
            System.out.println(">>Please enter you option");

            // Read an integer from the input
            int caseInput = in.nextInt();

            switch (caseInput) {
            [let actors: OrderedSet(Actor) = anUseCaseModel.actor]
                [for (actor: Actor | actors)]
                case [actors->indexOf(actor)/]:
                    {[actor.name.replaceAll(' ', '_')/] o=FFactory.getInstance().create[actor.name.replaceAll(' ', '_')/]();
                        o.execute();
                    }
                    break;
            [/for]
            [/let]
                case 0:
                    exit=true;
            }
        }
    }
}
[/file]
[for (actor : Actor | Actor.allInstances())]
    [generateActor(anUseCaseModel, actor)/]
[/for]
[for (uc : UseCase | UseCase.allInstances())]
    [generateUseCase(anUseCaseModel, uc)/]
[/for]
[generatExecutableInterface(anUseCaseModel)/]
[generateFactoryInterface(anUseCaseModel)/]
[generateFactoryImpl(anUseCaseModel)/]
[generateFFactory(anUseCaseModel)/]
[generateMain(anUseCaseModel)/]
[/template]
```

Figure 21 – Acceleo code generation file

2.2 MDE-based solutions

MDE-based solutions are solutions or tools that support MDE techniques, namely code generation, as well as supporting modelling tools (as is the case for ContextMapper).

2.2.1 JHIPSTER

JHipster is a free open-source development platform that is used to generate, develop, and deploy modern web applications and microservice architectures (*JHipster*, 2021).

JHpister supports many frontend technologies, including Angular, React and Vue. On the backend it supports Spring Boot, Micronaut, Quarkus, Node.js, and .NET.

JHipster also has mobile app support for Ionic and React Native.

Figure 22 – JHipster (*JHipster*, 2021)

JHispter's microservice architecture (which overview can be seen in Figure 23) includes:

- A gateway which is a JHipster-generated application that handles the Web traffic and serves as an Angular/React application. There can be several gateways.
- Traefik (*Traefik Labs*, 2021), which is a modern HTTP reverse proxy and load balancer, and can work with a gateway.
- The JHipster Registry which is a runtime application provided by the JHipster that runs the Eureka server (discovery server for applications), the Spring Clound Config server (runtime configuration to all applications), and an administration server with dashboards to manage the applications.
- Consul (*Consul*, 2021) is a service discovery service, as well as a key/value store, that can be used as an alternative to JHipster Registry.
- JHipster UAA which is a JHipster-based User Authentication and Authorization system that uses the OAuth2 protocol.
- Microservices, JHipster-generated applications that handle REST requests. They are stateless and there can be more than one instance of each one, in parallel to handle heavy loads.

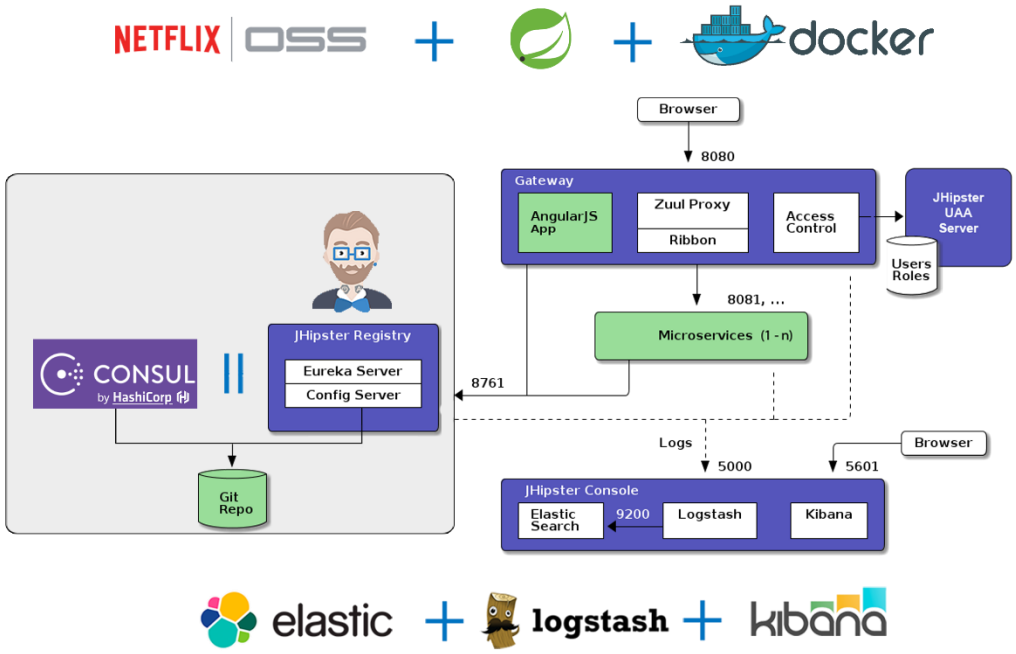

Figure 23 – JHipster microservices architecture overview (*JHipster Microservices Architecture*, 2021)

JHipster also includes JHipster Domain Language (JDL), a JHipster-specific domain language used to describe applications, deployments, entities, and their relationships. JHipster can generate the code for microservices from the JDL (*JDL*, 2021).

2.2.2 Context Mapper

Context Mapper is a modular and extensible modeling framework (which architecture can be seen on Figure 24) for Domain-driven Design (DDD) and its strategic patterns (*ContextMapper*, 2021).

ContextMapper is open source and provides a Domain-specific Language and Tools for Strategic Domain-driven Design (DDD), Context Mapping, Bounded Context Modeling, and Service Decomposition.

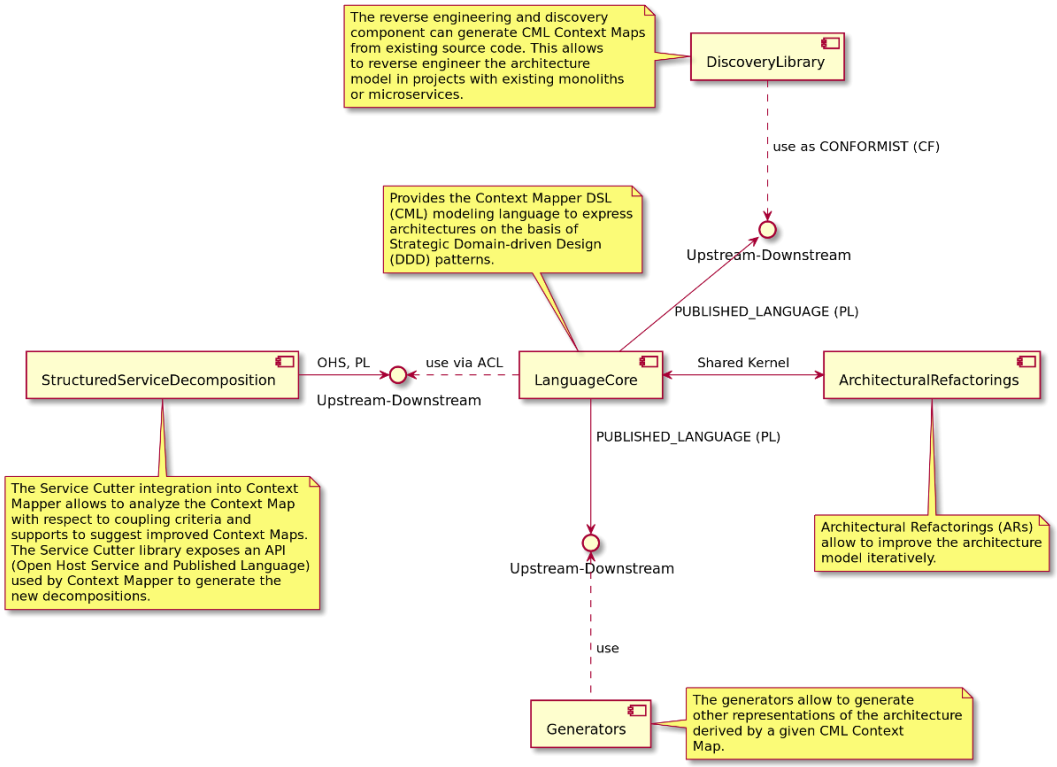

Figure 24 – ContextMapper Framework Architecture (*ContextMapper Home*, 2021)

ContextMapper features include:

- Service decomposition.
- Reverse engineering - recreate a CML context map from the existing code.
- Several generators:
  - Graphical Context Map generator based on Graphviz
  - PlantUML component and class diagram generator
  - Microservice Domain-Specific Language (MDSL) (micro-)service contracts generator
  - Service Cutter input files generators
  - Generic Textual Generator based on Freemarker Templates

It is to note that ContextMapper can be used in pair with JHIPSTER to generate microservices, by generating a JDL (JHipster) from the context map, using the Generic Textual Generator.

2.2.3 AjiL

AjiL (*AjiL*, 2020) is a graphical modelling language and toolkit for model-driven microservice architecture (MSA) engineering, mainly developed by the SEELAB research group of Dortmund's University of Applied Sciences and Arts.

AjiL uses EMF, and more specifically Sirius (*Eclipse Sirius*, 2021) as the graphical modelling framework, and Acceleo (*Acceleo | Home*, 2021) to implement a template-based code generation that leverages the Spring Cloud framework.

Moreover, AjiL is open-source and its code can be accessed on GitHub (https://github.com/SeelabFhdo/AjiL).

AjiL defines a modelling language, named AjiML, that comprises three components, abstract syntax, concrete syntax, and semantics.

AjiML's abstract syntax (of which metamodel can be seen in Figure 25) defines the system as the root element, and a MSA as several microservices which can be classified as functional or infrastructural. Each service consists of a domain model, which aggregates multiple entities, and one or more interfaces (that can provide abilities, for example create, read or manipulating entities of a service) (AjiL, 2020).

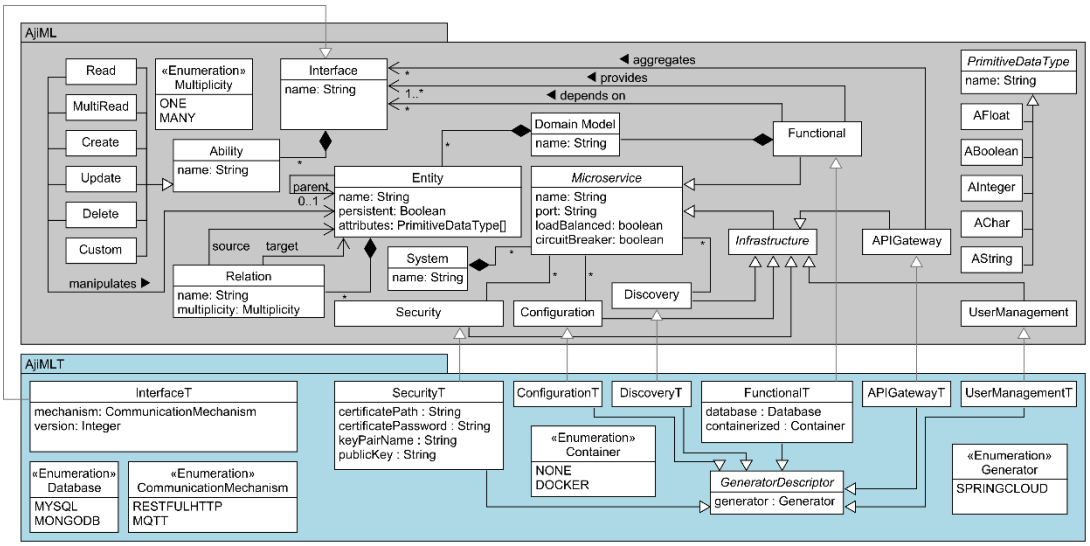

Figure 25 – AjiML Metamodel (*AjiL*, 2020)

AjiL's syntactical semantics and are formulated in the Object Constraint Language (OCL). The more relevant invariants are (*AjiL*, 2020):

- Unique names for the classes Microservice, Interface, Domain Model, Ability.
- Unique ports for each service.
- Entities are only allowed to inherit from Entities of the same Domain Model.
- Entities are only allowed to relate to other Entities of the same Domain Model.
- Prohibition of self-relations for Entities.
- Functional services are not allowed to depend on themselves.

AjiL's concrete syntax is split in two types of diagrams, Overview Diagrams (Figure 26), used to represent the services and their communication structure with each other, and Service Diagrams (Figure 27), that show the inner aspects of a service including its entities, interfaces, and abilities.

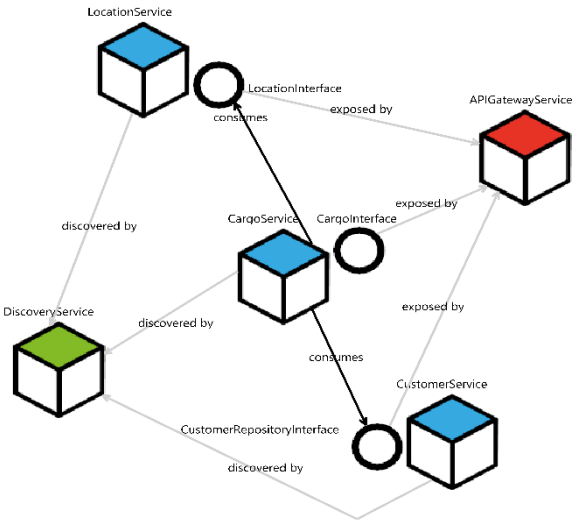

Figure 26 – AjiL's Overview Diagram (*AjiL*, 2020)

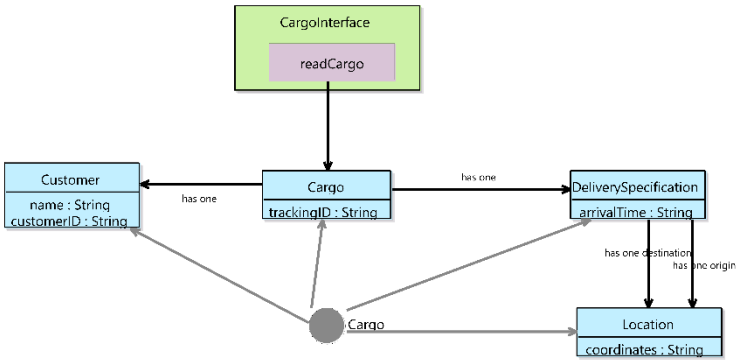

Figure 27 – AjiL's Service Diagram (*AjiL*, 2020)

2.3 Comparison

To classify the DSLs creating using the different frameworks, the following criteria was defined:

- **Focus –** The DSL focus can be vertical or horizontal.
  Vertical if it is aimed to a specific industry or field.
  Horizontal if the DSL has a broad applicability, it may refer to a specific technology but not to a specific industry.
- **Style –** The DSL style can be declarative or imperative.
  Declarative DSLs express the logic of a computation without describing its flow.
  Imperative DSLs define an executable algorithm that manages the steps and the flow that needs to be followed.
- **Notation –** The DSL notation can be graphical or textual.
  Graphical DSLs are displayed with visual models and the development primitives are graphical items, such as blocks, arrows, and so on.
  Textual DSLs describe the DSL with text-based notation, for example using XML or other textual notations.
- **Internality –** The DSL internality can be external or internal.
  External DSLs have their own custom syntax, with a full parser and self-standing, independent models/programs.
  Internal DSLs consist in using a host language and give it the feel of a particular domain or objective, either by embedding pieces of the DSL in the host language or by providing abstractions, structures or functions upon it.
- **Executability –** The DSL executability can be model interpretation or code generation.
  Model interpretation is when the DSL is read and executed at runtime, one statement at a time, the same way as programming languages interpreters does.
  Code generation is when a complete model-to-text transformation is executed at deployment time, producing an executable application, as compilers do for programming languages.

Table 1 – DSL Classification

Criteria	**MSDK**	**EMF**	**MPS**
Focus	Supports both Horizontal and Vertical	Supports both Horizontal and Vertical	Supports both Horizontal and Vertical
Style	Declarative	Declarative	Declarative/Imperative
Notation	Graphical	Textual	Projectional
Internality	External	External	External
Executability	Supports both model interpretation and code generation, but there is more focus on code generation.	Supports both model interpretation and code generation, but there is more focus on code generation.	Supports both model interpretation and code generation.

Table 2 represents a subjective comparison between the MDE base technologies, based on the authors user experience and research, using values from 0 to 10, and evaluating them according to the following criteria:

- **DSL creation** – Whether or not a tool can be used to create and maintain a DSL. And if it can, how intuitively, versatile, and powerful the language workbench is.
- **Model-to-Model transformation** – Whether or not a tool can be used perform a model-to-model transformation. And if it can, how intuitively, versatile, and powerful the transformation workbench is.
- **Code generation** – Whether or not a tool can be used perform a code generation. And if it can, how intuitively, versatile, and powerful the transformation workbench is.
- **Documentation and Community support** – This criterion represents the tool's documentation and community support, which can be time saving in some scenarios.

Table 2 –Tool comparison (0-10)

Criteria	**MSDK**	**EMF**	**MPS**
DSL Creation	9	7	7
Model-to-Model transformation	2	10	7
Code generation	6	8	7
Documentation and Community support	7	9	8

On this comparisons EMF includes the use of Xtext for the DSL creation, ATL for model transformation and Acceleo for code generation.

The DSL creation is more intuitive in MSDK and EMF than MPS, however, it is possible to obtain similar results with these tools.

EMF and MPS both have a good support for Model-to-Model transformation, on the other hand, MSDK does not, only being able to generate text files that may be used as a Model using external tools.

All the tools have good documentation, but EMF has the biggest community support.

Furthermore, although MSDK is more intuitive than EMF and MPS it is also more restrictive.

As for the MDE-based solutions, when it comes to code generation all the tools can generate code from the domain, however JHipster and AjiL are focused on developing Spring Boot microservices, in contrast to the MDE base technologies that are used for any type of code generation.

Although the tools defined as MDE-based solutions do not support DSL development, and thus may be less versatile than the others, it is to note that not every project requires a DSL (of which development may consume lots of time and resources), in which case tools like these can be very useful and time saving.

2.4 Related Work

This section presents some relevant works that are related to this dissertation.

2.4.1 Graphical and Textual Model-Driven Microservice Development

"Graphical and Textual Model-Driven Microservice Development" (Rademacher et al., 2020) is a study, that similarly to this dissertation aims to explore how MDD can be used to support and facilitate microservices' engineering.

This study explores two approaches for employing MDD in microservice architecture (MSA) engineering. The first approach consists in using graphical notation to model the topology and interactions of MSA-based software systems. The second approach emerged from the first, and exploits viewpoint-based modelling, aiming to better cope with MSA's complexity.

To compare the two approaches, a case study (Figure 28) was designed. This case study consists in a MSA-based software system that enables users to rent their electric vehicles' charging points to other electric vehicle drivers.

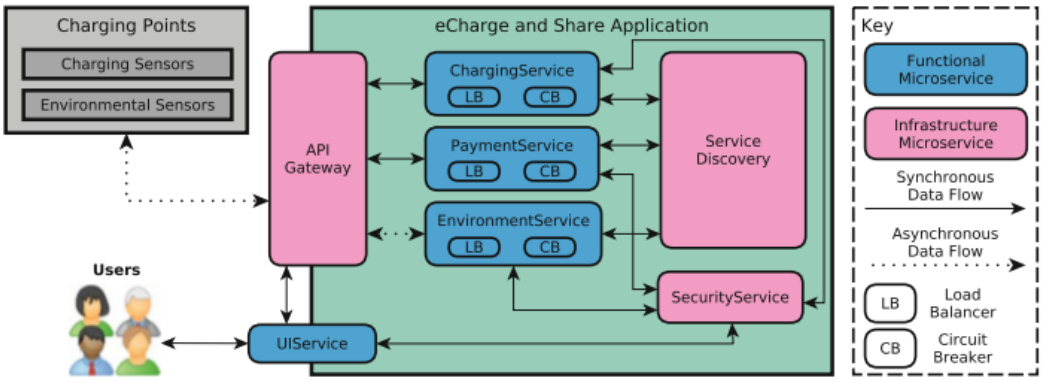

Figure 28 – Case study's architecture (Rademacher et al., 2020)

The first approach used AjiL, as an approach to use graphical MDD for MSA engineering, whereas the second approach used a textual viewpoint-specific approach to model-driven microservice development.

This second approach aimed to "consider technology heterogeneity of microservices, reduce modelling complexity via viewpoints, scale with team distribution, and increase modelling conciseness and efficiency by employing a textual notation"(Rademacher et al., 2020).

The second approach required the development of three metamodels, for the domain data modelling language, the service modelling language, and the operation modelling language.

Furthermore, this second approach considers the stakeholder roles in the workflow with the dedicated modelling languages, namely the domain expert, service developer and service operator.

The study found that although AjiL's graphical modelling approach tends to be more attractive to the human reader, and facilitates the understanding of basic MSA concepts, it may lack precision and expressiveness required to model complex MSA-based software systems.

Lastly, the study presents research questions for subsequent investigation of employing MDD to support and facilitate MSA engineering.

Both "Graphical and Textual Model-Driven Microservice Development" and this work study the use of Model-driven development for microservices. However, the referred study analyses the use of graphical and textual model-driven development, whereas this work analyses and compares three different approaches of microservices development, namely, traditional development, DSL-based approach development, and MDE-based tool approach development.

2.4.2 MicroBuilder: A Model-Driven Tool for the Specification of REST Microservice Architectures

"MicroBuilder: A Model-Driven Tool for the Specification of REST Microservice Architectures" (Terzić et al., 2017) is a paper that presents MicroBuilder, a tool used to specify and generate software that follows Representational State Transfer (REST) microservices design principles.

The paper starts with a brief introduction to microservices and MDE, and then proceeds to present MicroBuilder 's architecture (Figure 29). MicroBuilder was developed using the Eclipse Modeling Framework (EMF) and comprises two main modules, the MicroDSL and MicroGenerator modules.

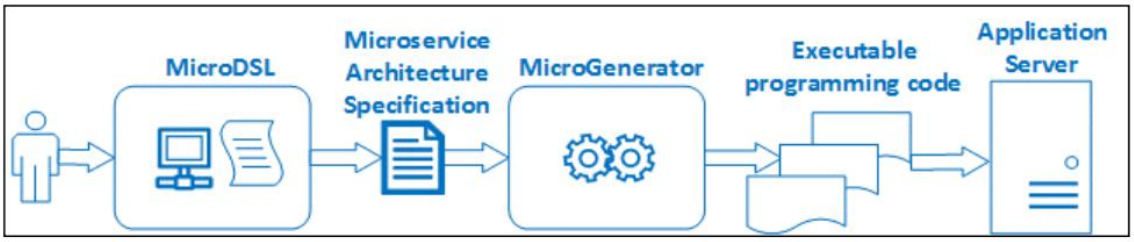

Figure 29 – MicroBuilder's Architecture (Terzić et al., 2017)

The MicroDSL module provides a DSL for the specification of REST microservice software architecture specification. MicroDSL's concrete syntax was developed using the Xtext

language and allows the user to specify the whole microservice architecture at one place, using only domain concepts. The MicroDSL's meta-model can be seen in Figure 30.

A microservice architecture specification can be used as an input to the MicroGenerator module to generate the code for the microservice.

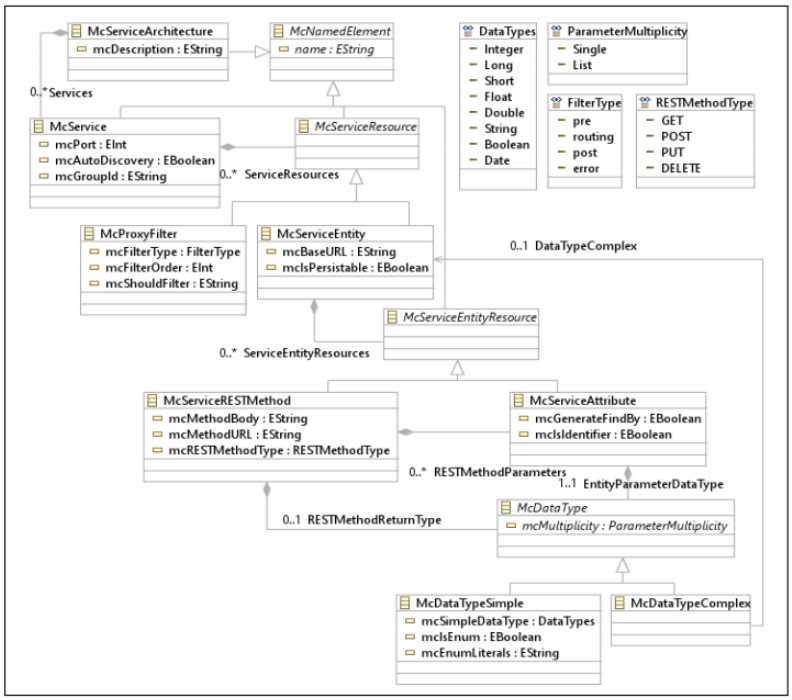

Figure 30 – The MicroDSL meta-model (Terzić et al., 2017)

The MicroGenerator module provides a set of code generators for the Java programming language, however, more languages can be supported by adding new generators as plugins. MicroGenerator's individual code generators are developed using the Xtend language.

This paper compares the number of MicroDSL lines of code with the number of manually written lines of code for each microservice separately, the second being much higher, this leads to a conclusion that MircroBuilder can be suitable for fast prototyping in situations when we need quick solution which requires minimal coding interventions on the generated program code.

Both MicroDSL and the DSL presented by this work allow the specification of REST microservices. However, MicroBuilder focus on developing a Model-Driven Tool for the Specification of REST Microservice Architectures and does not compare different approaches, whereas this work focusses on comparing MDE approaches to develop microservices.

3 Value Analysis and Proposition

This section focusses on the value that is envisioned that this solution provides, using the New Concept Development Model (NCD) to identify and analyse the opportunity, and then define its perceived value, a value proposition and analysing it using the Quality Function Deployment and Analytic Hierarchy Process methods.

3.1 NEW CONCEPT DEVELOPMENT (NCD) MODEL

The innovation process is the process of coming up with an idea and develop, test, and commercialize it.

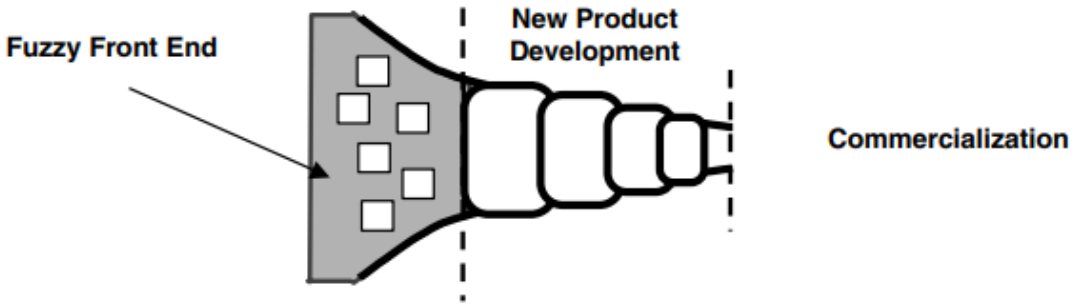

Figure 31 – The Innovation Process (Koen et al., 2002)

The “Fuzzy Front End” is the first stage of the innovation process, and is the part where the opportunities are identified, analysed, and validated, and the concept is developed, prior to entering the product development phase.

The New Concept Development (NCD) Model, defined by Peter Koen (Koen et al., 2002) is a model defined with common language and terminology which aims to help optimize activities

in the FFE(Fuzzy Front End), hopefully resulting in a higher number of profitable concepts entering the NPD (New Product Development).

The NCD model is nonlinear process and consists of three parts (represented in Figure 32):

- the uncontrollable influencing factors
- the controllable engine that drives the activities in the FFE
- the five activity elements of the NCD

The NCD model is represented as a circular shape because the ideas are expected to flow, circulate, and iterate between and among all the five activity elements.

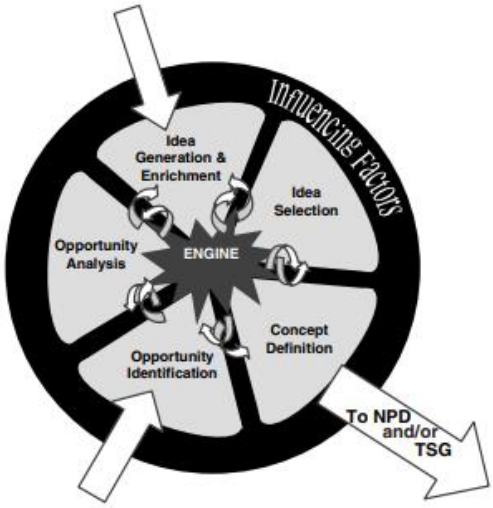

Figure 32 – The NCD Model (Koen et al., 2002)

The influencing factors are the factors that affect the entire innovation process and influence the concept and its viability, namely the organization capabilities, the outside world and the enabling sciences and technologies.

The engine represents the factors that drive the five key elements that are controllable by the corporation, for example the leadership, culture, and business strategy of the organization.

Lastly, there is the inner spoke area which defines the five key elements that are controllable by the organization, namely opportunity identification, opportunity analysis, idea generation and enrichment, idea selection and concept definition.

This chapter focuses on opportunity identification and analysis.

3.1.1 Opportunity identification

Seeing as microservices are becoming more and more common, it makes sense to try to make their development faster and easier, as well as less error and bug prone. One possible solution would be to use MDE with this purpose.

By using MDE to develop microservices several benefits would be achieved, for instance:

- Focus on the model (capture of domain knowledge).
- Reduced gap from business to implementation.
- Up-to-date documentation.
- Long-term cost effective and increased efficiency.
- Increased quality and consistency, less bugs and errors.
- Developers can focus on focus on strategic architectural issues and creativity aspects rather than technical implementations.

3.1.2 Opportunity analysis

Although MDE may be a possible solution to the previously it is necessary to further analyse and define the opportunity.

These techniques achieve different things, and not everyone needs to be used. A study done by Marco Torchiano (Torchiano et al., 2013) analyses the MDE techniques ant its benefits.

Firstly, this study shows that (except for individual companies) the use of modelling is positively correlated with the size of the company, as can be seen in Figure 33.

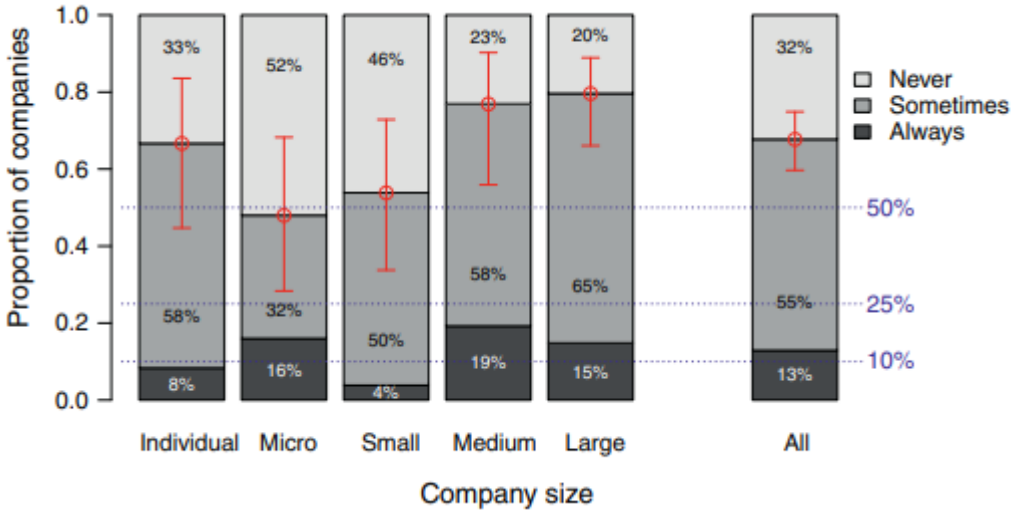

Figure 33 – Proportion of modelling usage per company size (Torchiano et al., 2013)

It is also important to note that “companies that target a particular domain are more likely to use MDE than companies that develop generic software” (Whittle et al., 2014).

The study then proceeds to explore the relevance of the different MD* (model-driven) techniques, as can be seen in Figure 34.

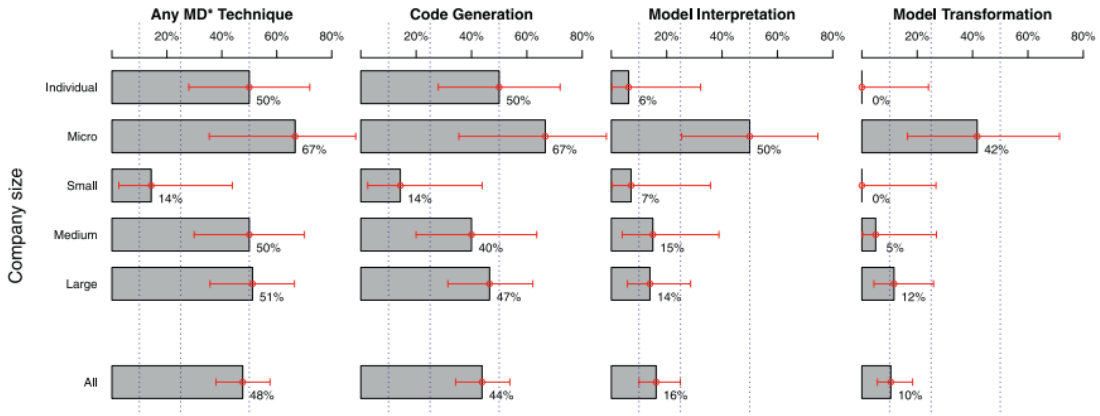

Figure 34 – Diffusion of MD* techniques among modellers per company size.

In the analysed sample, code generation was the most used MD technique, being used by 44% of the 105 modellers, followed by model interpretation which is used by 16% and then model transformation used by 10%. When considered only MD adopters then 92% use code generation, 34% use model interpretation and 20% use model transformation.

The study concludes that micro companies are the only significant adopters of model interpretation and model transformation, whereas code generation is by far the most used technique.

Furthermore, the study shows that MD techniques are mostly used individually (2/3 cases), being the most common toolbox the use of code generation alone (28% of modellers), being the combination of all three techniques the next most frequent use case (with only 7% of modellers). It is also to note that model transformation techniques are never used alone but only together with other techniques.

The study then explores the benefits achieved with the use of MD techniques, as shown in Figure 35. The most significant differences are in *Standardization, Productivity and Platform Independence*, which is consistent with the purpose of modelling and MD.

In addition, when analysing the achieved benefits for the adopters of each technique, “code generation induces a significant difference concerning *Productivity*, which is 3.9 times more likely to be achieved as a benefit when the technique is adopted” (Torchiano et al., 2013). There is also a significant difference in *Flexibility*, *Productivity, Reactivity to changes* (4 times more likely) and *Platform Independence* (5 times) when model interpretation is used.

Moreover, when model transformation is applied there is a significant likelihood achievement increment for *Productivity* (about 8 times) and *Platform Independence* (4 times).

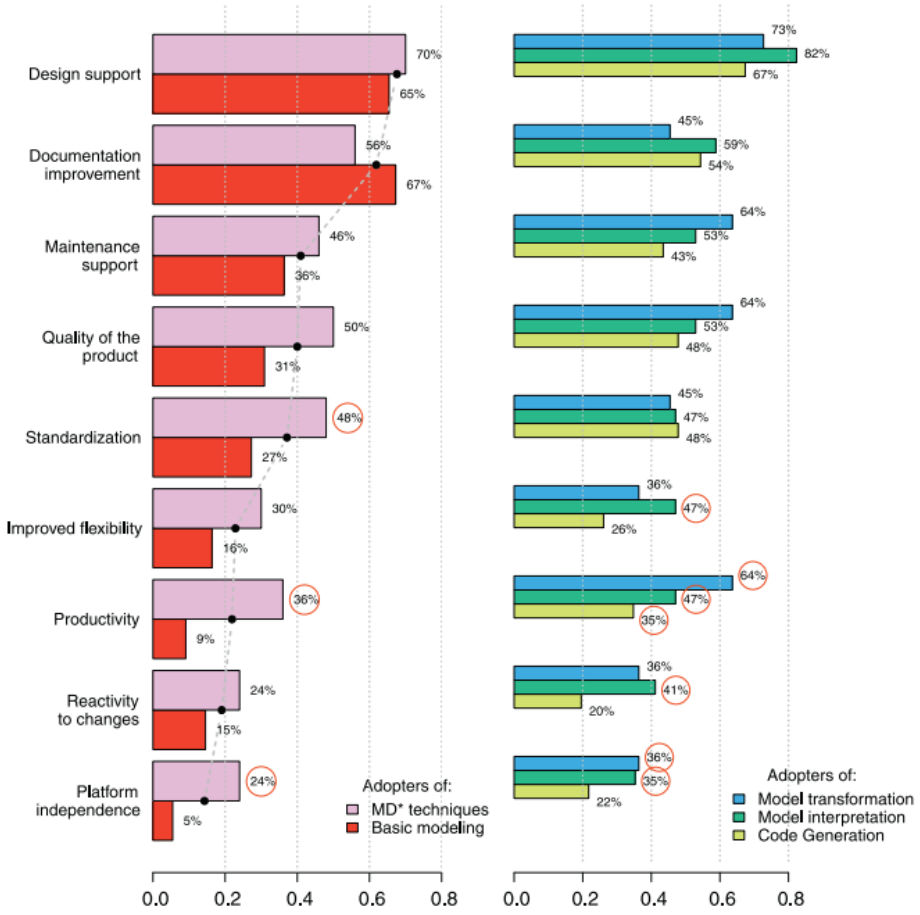

Figure 35 – Benefits achieved with the use of MD* techniques (Torchiano et al., 2013)

3.2 Perceived value

To realize the value of using MDE for microservices' development, it is necessary to identify the stakeholders:

- **Developers**:
  Benefits
  - Developers can increase their productivity, allowing them to focus on strategic architectural issues and creativity aspects rather than technical implementations.

  Sacrifices
  - Developers must be trained to use MD techniques which requires time and resources.

- Not every developer likes/trusts MD technique, which may lead to some initial motivation and productivity decrease. (Hutchinson et al., 2014)

- **Companies**:
  Benefits
  - This approach is long-term cost effective, which allows the company to save money and resources.
  - The company can focus more on business rules and domain and less in the technologies.
  - As the product becomes more reliable and maintainable, the company's service quality is improved.

  Sacrifices
  - There is an initial high cost related to developing or adopting tools and MD techniques, which requires time and resources.
  - Success is dependent on organizational commitment.

- **Users**:
  Benefits
  - The user experience is improved, seeing as the product has increased quality and consistency, and less bugs and errors.

3.3 Value proposition

To define the value proposition its necessary to identify the product, gain creators, pain relievers, gains, pains, and customer jobs, thus developing a value proposition canvas (Figure 36).

The following value proposition represents the value to be delivered by using MD techniques for microservices development.

"The use of MDE in the development of microservices increases the long-term development efficiency, allowing developers and companies to be more productive, and having a more resilient code with easier maintenance, improving the user experience."

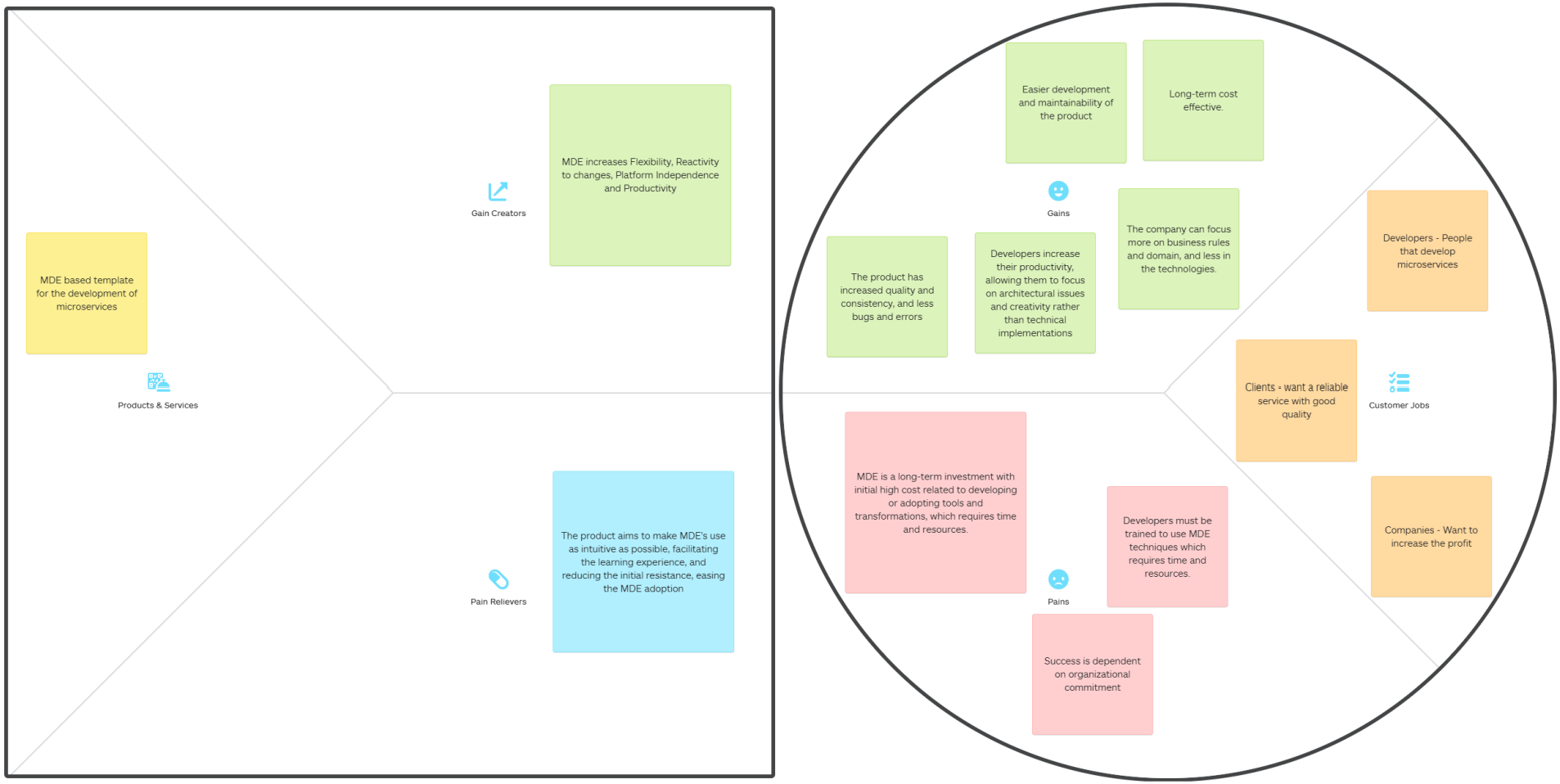

Figure 36 – Value Proposition Canvas (www.strategyzer.com)

3.4 Quality Function Deployment (QFD)

The Quality Function Deployment (QFD) is a method used to translate the customer requirements into engineering characteristics (*QFD*, 2021). QFD is used to produce products with high levels of customer perceived value.

The QFD uses a House of Quality to identify and classify the customer desires (Whats's), identify the desires importance, identify the engineering characteristics that influence the customer desires (How's), correlate the customer desires to the engineering characteristics, to obtain a priority order for the system requirements, which can be analyzed in Figure 37.

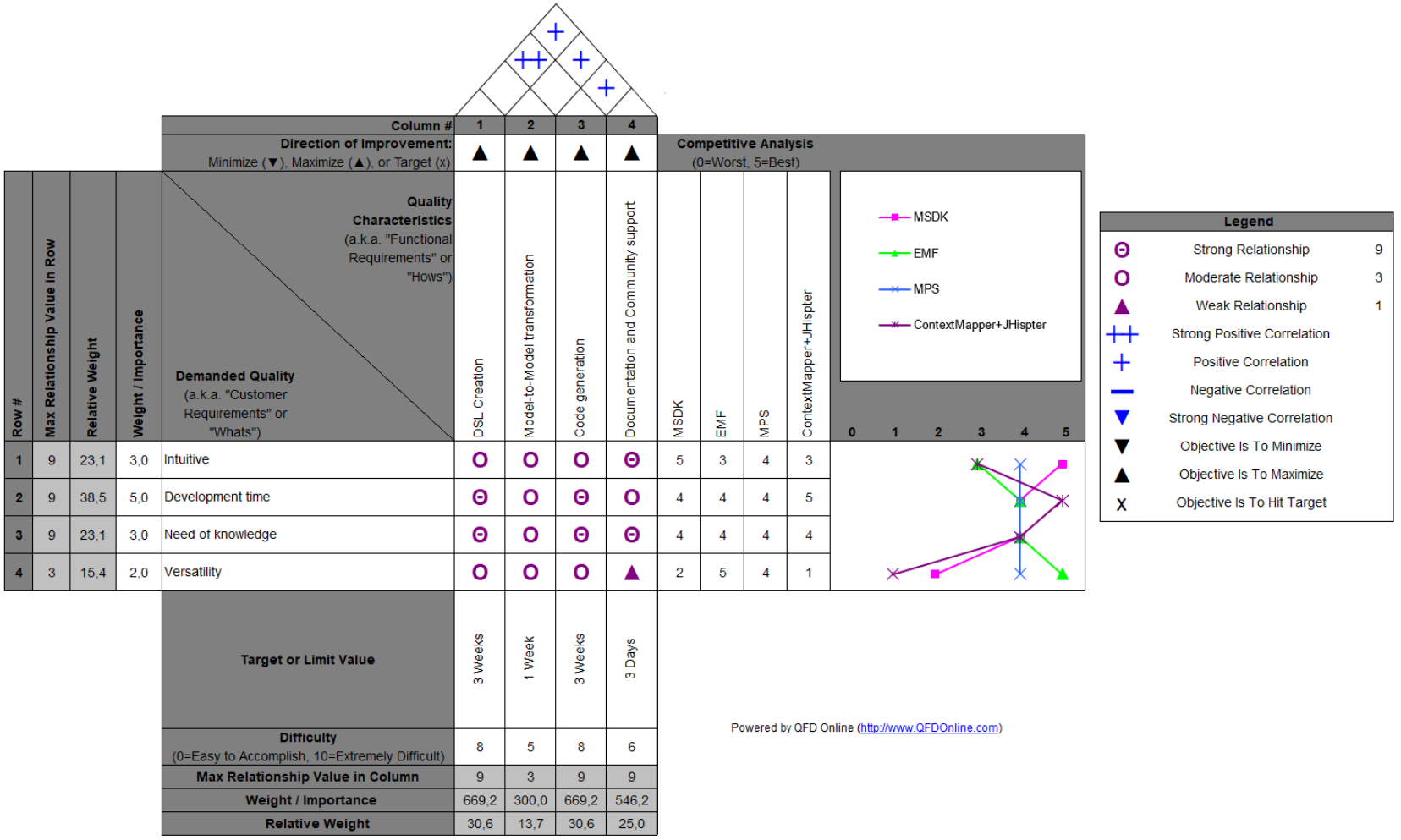

Figure 37 – House of Quality (QFDonline.com)

Through an analysis of Figure 37, it is possible to understand the order of importance of the quality characteristics from the customer's perspective, in this case the Developers.

Firstly, there is the “DSL Creation” and “Code Generation”, the factors that in fact allow the customer to increase his productivity.

Secondly, the “Documentation and Community support”, which has a big influence when choosing the tool to use, as it deeply relates to the ease of use of the framework/application, and saves time in learning and research, ultimately increasing productivity.

Lastly, there is the “Model-to-Model transformation”, that not all tools support, but it can be useful in some scenarios, allowing for productivity increase.

3.5 Analytic Hierarchy Process (AHP)

The Analytic Hierarchy Process (AHP) is a multi-criteria decision method, developed by Thoma L. Saaty in 1980. AHP represents an accurate approach to quantifying the weights of decision criteria, and can be used with qualitative, as well as quantitative criteria.

AHP aims to divide the problem in hierarchic decision levels, facilitating its comprehension.

The first step of the AHP method is to build the hierarchic decision three, with three levels representing the problem, the criteria, and the alternatives, respectively. The defined hierarchic three can be seen in Figure 38.

In this case the AHP method aims to determine which is the best tool de develop microservices using MDE.

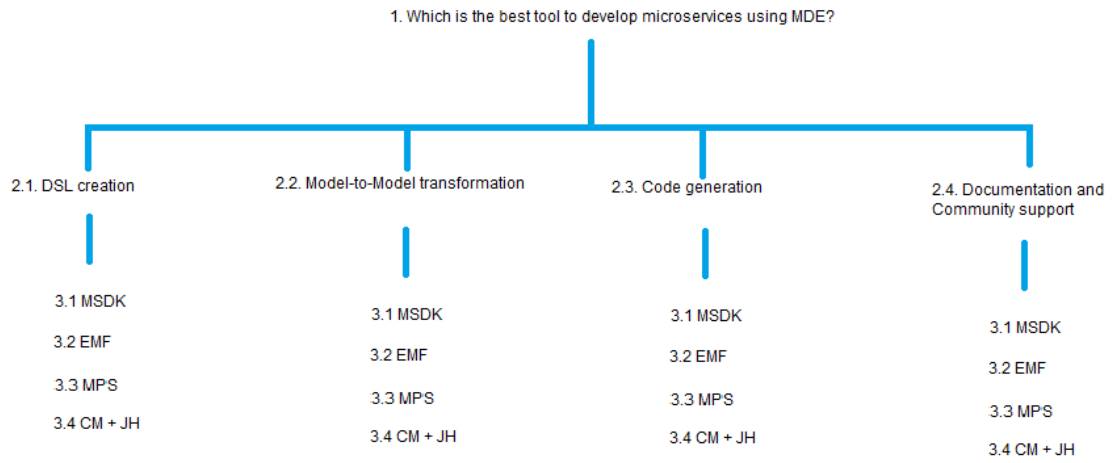

Figure 38 – Hierarchy tree

The level 1 represents the problem, in this case "What is the best tool to develop microservices using MDE?".

The level 2 represents the criteria, namely:

- **DSL creation** – Whether or not a tool can be used to create and maintain a DSL. And if it can, how intuitively, versatile, and powerful the language workbench is.
- **Model-to-Model transformation** – Whether or not a tool can be used perform a model-to-model transformation. And if it can, how intuitively, versatile, and powerful the transformation workbench is.
- **Code generation** – Whether or not a tool can be used perform a code generation. And if it can, how intuitively, versatile, and powerful the transformation workbench is.
- **Documentation and Community support** – This criterion represents the tool's documentation and community support, which can be time saving in some scenarios.

The level 3 represents the alternatives, namely:

- MSDK
- EMF
- MPS
- ContextMapper + JHipster

The second step is to establish a priority between the elements of each hierarchic level, using a comparison matrix, using the scale represented on Table 3.

Table 3 – Fundamental scale - Levels of importance of comparisons

Importance Level	**Definition**	**Explanation**
1	Equal importance	Both activities contribute equally to the objective
3	Weak importance	The experience and the judgement favor an activity over the other
5	Strong importance	The experience and the judgement strongly favor an activity over the other
7	Very strong importance	One activity is much strongly favored over the other

9	Absolute importance	The evidence favors one activity over the other, with the highest degree of certainty
2,4,6,8	Intermediate values	Intermediate values represent a compromise between two definitions

The defined criteria pairwise comparison can be seen in Table 4.

Table 4 – Criteria pairwise comparison

	DSL	MtM	CodeGen	DocCom
DSL	1	7	1	4
MtM	1/7	1	1/7	1/3
CodeGen	1	7	1	3
DocCom	1/4	3	1/3	1
Sum	2 2/5	18	2 1/2	8 1/3

The next step is to obtain the relative priority of each criteria, which is done by normalizing the pairwise comparison matrix (Table 4), and then obtaining the priority vector, by calculating the arithmetical average of the values in each line of the normalized matrix. The obtained result can be seen in Table 5.

Table 5 – Priority vector

DSL	0,422661
MtM	0,053237
CodeGen	0,392661
DocCom	0,13144

After obtaining the priority vector its necessary to evaluate the consistency of the relative priorities, which is done by calculating the Consistency Ratio (CR), in order to measure how consistent the judgments were in relation to large samples of completely random judgments.

According to AHP method the judgements are based on the assumption that the decision maker is rational, so that if A is preferred to B, and B is preferred to C, then A is preferred to C.

If CR's value is superior to 0,1 then the judgements are not trustworthy, and seemingly random, requiring an adjustment of the comparison matrix.

In this case a CR of 0,009163 was reached, assuring the consistency of the criteria comparison matrix.

After creating a consistent criteria comparison matrix the same process must be done on a comparison table for each criteria, considering the selected alternatives, until all the criteria's comparison matrix are consistent. (The full AHP analysis can be seen in Appendix A).

The last step is to obtain the alternatives' priority composite, by multiplying the priority vector of each criteria's comparison matrix with the criteria's relative priority (obtained previously from the criteria comparison matrix). This process can be seen in Table 6.

Table 6 – Alternatives' composite priority

	DSL	MtM	CodeGen	DocCom				
MSDK	0,526042	0,118104	0,073003	0,121873		0,422661		0,27331
EMF	0,217014	0,607358	0,286265	0,557892	x	0,053237	=	0,309792
MPS	0,217014	0,233719	0,121302	0,263345		0,392661		0,186411
CM+JH	0,039931	0,040819	0,519429	0,05689		0,13144		0,230488

The alternative with the highest relative priority represents the AHP's final result, in this case represented in Blue in Table 6, EMF, which according to the results is the tool best suited to develop microservices using MDE, taking into consideration the defined criteria.

4 Case Study

This chapter introduces the case study defined to compare the different microservices' development approaches, containing its description, and specifying the requirements and design.

4.1 Case Study Description

The case study used to compare the different approaches of microservices development, must allow its subdivision in different bounded context, to allow the development of more than one microservice. Also, the microservices must have some relationship with each other, and yet, the use case has to be simple enough for anybody to understand.

The defined case study is a property booking application, where users can rent properties on given dates.

Firstly, a user can register a property, which will then be available to book by other users, (there can only be one booking per property on a given date).

After the stay a user can leave a review of the respective booking, rating it and specifying a comment that should contain relevant information for other users that may be interesting in booking the same property.

A booking has two states, confirmed and unconfirmed. A booking is only confirmed after a payment is issued, after which the booking is locked, and no other user can rent the property on the given dates.

4.2 Functional Requirements

The functional requirements are defined in Table 7.

Table 7 – Functional requirements

Identifier	Requirement
FR1	Users can manage their properties.
FR2	Users can manage their bookings.
FR3	Users can query a property's availability on a given period.
FR4	Users can add reviews to their bookings.
FR5	Users can pay for their bookings.

It is important to note that to create a booking its required to validate that the property exists and is available.

Furthermore, for the requirements RF4 and RF5 it is necessary to verify if the user is allowed to execute such a request. Users can only pay and review their own valid bookings.

Identifier	Requirement
NFR1	The solution must be developed using Spring Boot.
NFR2	The design solution must follow a microservice architecture.
NFR3	An In-memory database per microservice must be used.

4.3 Design

This section explains the design of the proposed use case. It is expected that all approaches follow this design, however as the MDE-tool based approach as less control over the resulting microservices' code than the other, there can be some discrepancies, but the design must be followed whenever possible, or as similar as possible.

4.3.1 Domain

The domain can be seen in Figure 39 and is composed of the following classes:

- **Property** – The Property represents a physical property than can be booked, and its price per night.
- **Address** – The Address represents an address and is used for property locations.
- **Booking** – The Booking represents a booking of a specific property on specific dates.
- **Status** – A Booking can have 2 states "UNCONFIRMED" (default), or "CONFIRMED" (after the payment is validated).
- **Review** – The Review represents a review of a specific booking and can only be done by the respective user.
- **Payment** – The Payment represents the payment of a specific booking and can only be done by the respective user.

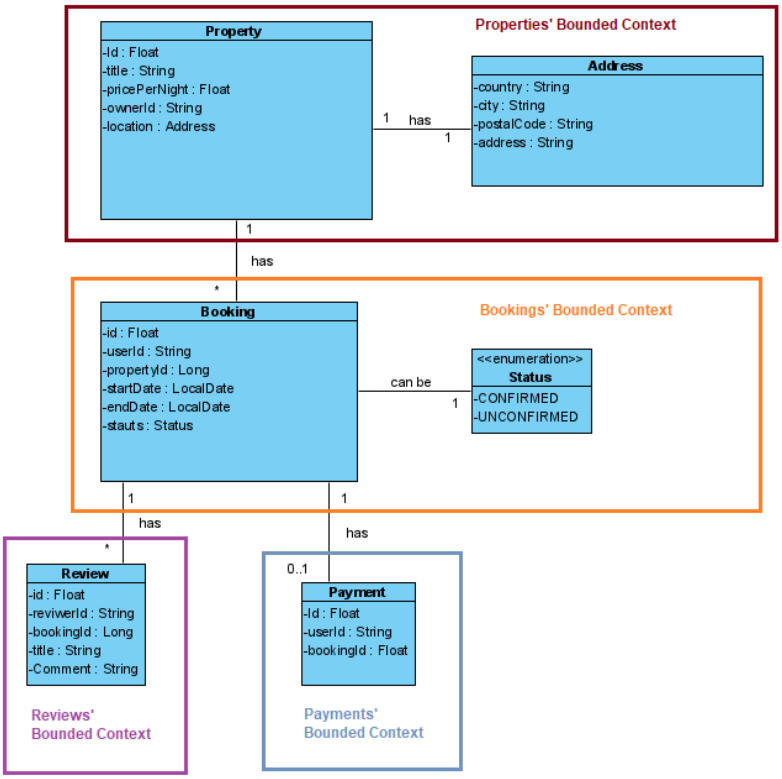

Figure 39 – Domain model

4.3.2 Solution Design and Deployment

In order to divide the solution in several microservices the "decompose by subdomain" (Richardson, 2021) pattern was used.

According to this pattern we should divide our domain in different subdomains, where each subdomain corresponds to a different part of the business, and then classified as follows (Richardson, 2021):

- Core - key differentiator for the business and the most valuable part of the application
- Supporting - related to what the business does but not a differentiator. These can be implemented in-house or outsourced.
- Generic - not specific to the business and are ideally implemented using off the shelf software.

Following this pattern, we can identify 4 subdomains (also visible on Figure 39 as bounded contexts):

- Property management
- Booking management
- Review management
- Payment management

All the subdomains are considered “Core”, except for payment, which is a “Supporting” domain, and represents something that can be outsourced.

For each subdomain a microservice must be developed, leading to the architecture that can be seen in Figure 40.

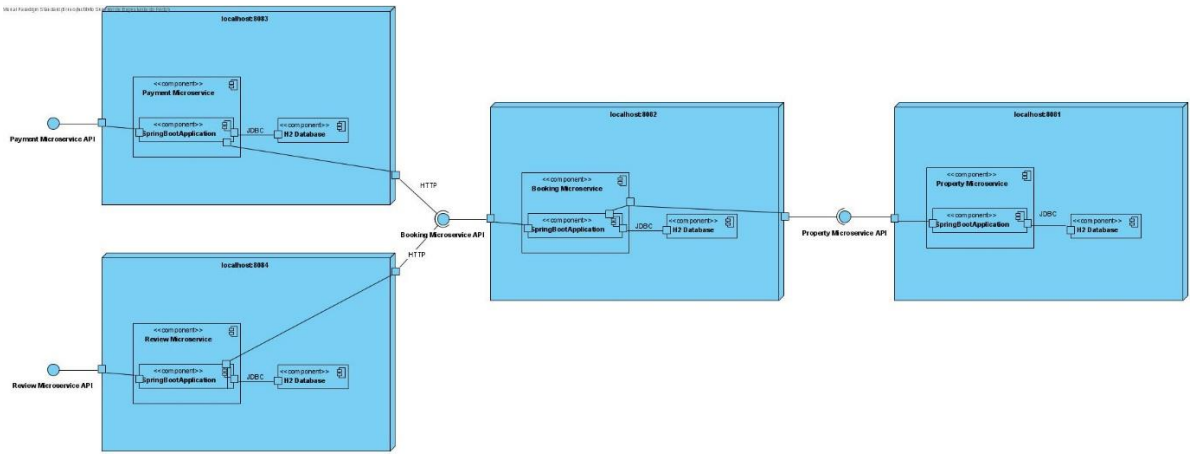

Figure 40 – Deployment Diagram

As explained before, the Booking’s Microservice uses the Property Microservice when creating a new booking to verify the Property exists. The Review and Payment microservices use the Booking’s Microservice to confirm the user is authorized to pay or review a specific booking.

4.3.3 Microservice components

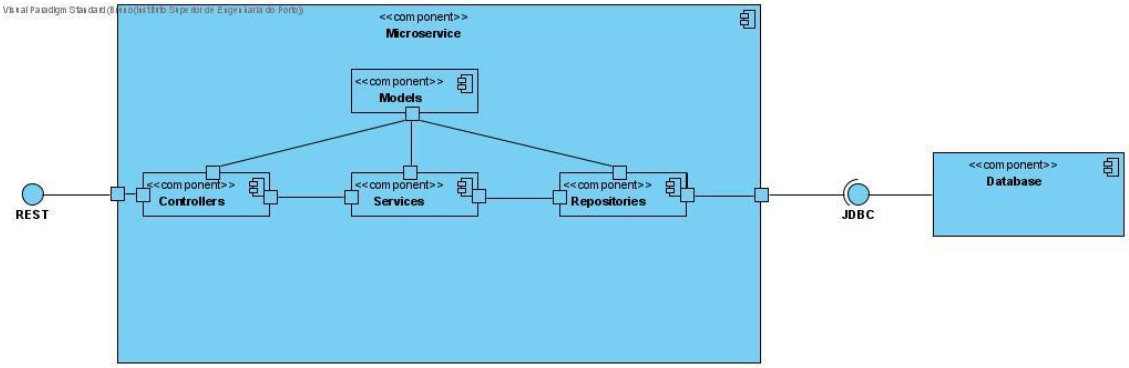

Figure 41 – Microservice component diagram

A can be seen in the Figure 41, there is a database per microservice, and each microservice implements a REST API (application programming interface).

Each microservice has 4 components, namely “Models”, “Controllers”, “Services” and “Repositories”:

- **Models** – Models represent the domain, and map the domain entities to objects, that are used by the other components.
- **Controllers** – The controllers represent the microservice interface, in this case a REST API, that works as a gateway to the services.
- **Services** – Services implement the business logic and can use the repository to manage the data.
- **Repositories** – The repositories are responsible for the isolation of the persistency layer and control the access to the data.

4.3.4 Requirement’s design

This section presents the design of the functional requirements’ process, using software sequence diagrams (SSDs).

4.3.4.1 FR1 and FR2, Properties and Bookings management (CRUD)

The only major difference between this requirement is that for the FR2 (Booking's management), in order to create a booking, the Booking's controller must execute a GET request to the Properties' microservice to validate the existence of the specified property.

The functional requirements FR1 and FR2, which consist in Properties and Bookings management respectively, involve the Create, Read, Update, and Delete of their entities, namely Property and Booking. The respective diagrams can be seen in Figure 42, Figure 43, Figure 44, and Figure 45

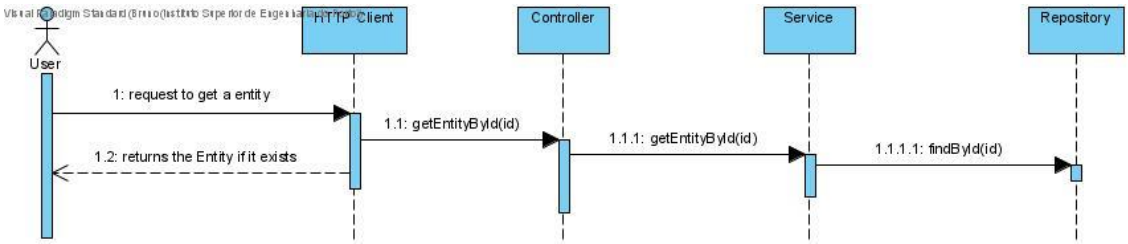

Figure 42 – Create SSD

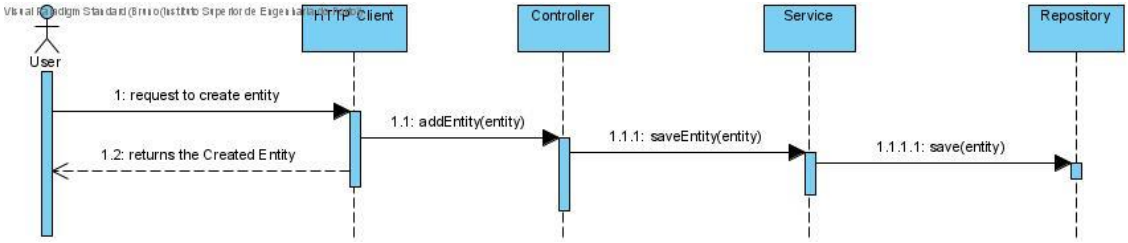

Figure 43 – Read SSD

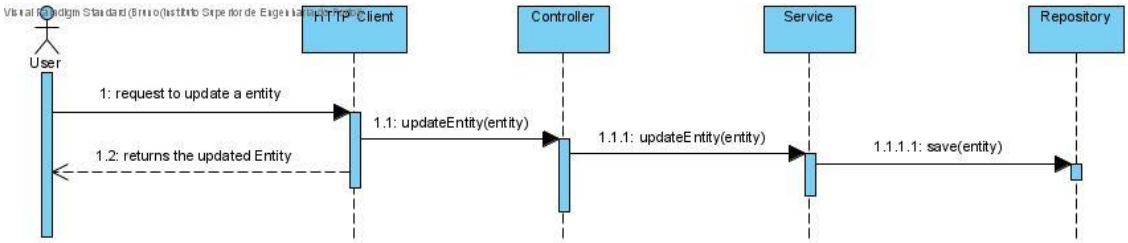

Figure 44 – Update SSD

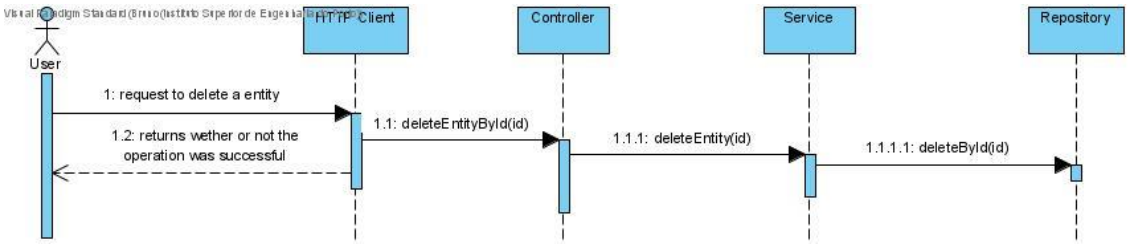

Figure 45 – Delete SSD

4.3.4.2 FR3 - Users can query a property's availability on a given period.

This requirement consists in a user query of a property's availability on between a set of given dates, to do so the Bookings Microservice must check if there is any booking in the respective dates. If no bookings are found, then the Property is available. The sequence diagram for this requirement can be seen in Figure 46.

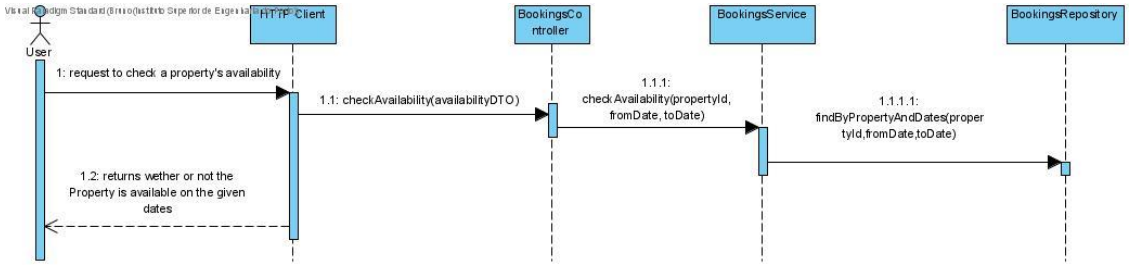

Figure 46 – Query Availability SSD

4.3.4.3 FR4 - Users can add reviews to their bookings.

Users can add reviews to their bookings (using the Reviews' microservice), however, to do so the Reviews' microservice has to check if the user is authorized to submit a review, and it does so by contacting the Bookings Microservice to validate the exitance of a given booking, and the correspondence of the booker, in this case the user that is trying to submit a review. The sequence diagram for this requirement can be seen in Figure 47.

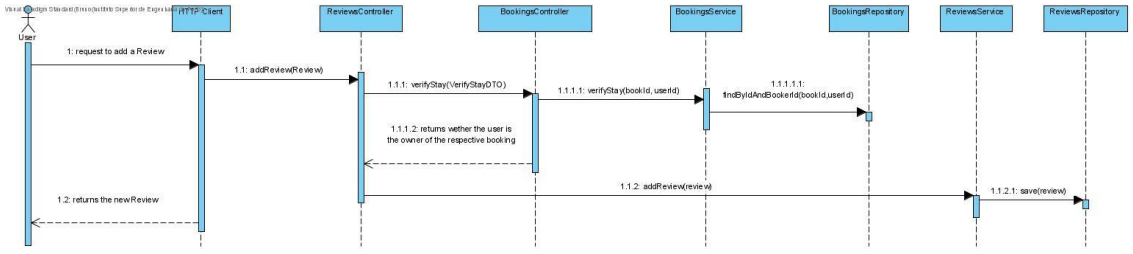

Figure 47 – Add Review SSD

4.3.4.4 FR5 - Users can pay for their bookings.

This requirement is similar to the previous one, however, instead of a user trying to submit a review, the user is trying to submit a payment, and the Payments' microservice has to contact the Bookings microservice to validate the existence of the given booking, and the correspondence of the booker, in this case the user that is trying to pay. The sequence diagram for this requirement can be seen in Figure 48.

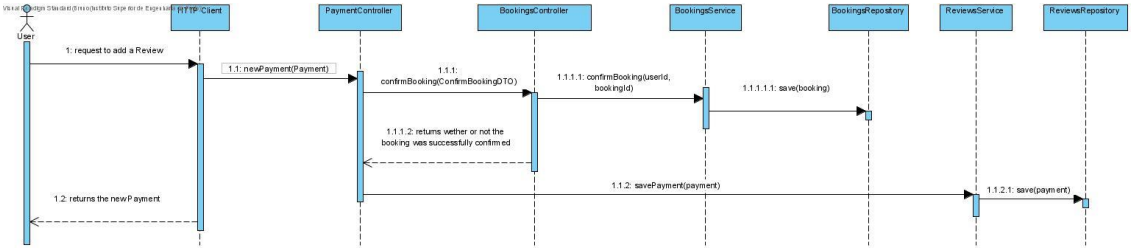

Figure 48 – Pay for booking SSD

5 Implementation

This section presents the development process of the use case for the different approaches.

The use case was developed in 3 different approaches, which will then be analysed and compared, the Traditional Development, the DSL-based approach and the MDE-based tool approach.

5.1 The Approaches

As explained before, these three different approaches will be used to develop the same case study, following the Design whenever possible:

- Traditional development, consisting in manual development of the microservices, with no support from DSLs or MDE techniques.
- Developing a DSL (that allows the modeling of microservices) from scratch using EMF and Xtext, and generating code for microservices using Acceleo.
- Using MDE-based tool(s) that support code generation of microservices based on a DSL in this case using ContextMapper for the microservice specification and JHipster for the code generation.

5.2 Traditional Development

For this approach the case study was manually developed using Spring Boot (*Spring Boot*, 2021), as it is one of the more common approaches for developing Java microservices.

The four microservices were developed following the design (section 4.3), consisting of:

- Models

  The Models were developed using annotations, namely, to generate the Id, and the Entity annotation to mark it as a persistable entity. As an example, the Booking's Model can be seen in Figure 49.

```java
@Entity
public class Booking {

    @Id
    @GeneratedValue
    private Long id;
    private String bookerId;
    private Long propertyId;
    private Date startDate;
    private Date endDate;
    private Status status = Status.UNCONFIRMED;

    public Booking(){

    }

    public Booking(String bookerId, Long propertyId, Date startDate, Date endDate, Status status) {
        this.bookerId = bookerId;
        this.propertyId = propertyId;
        this.startDate = startDate;
        this.endDate = endDate;
        this.status = status;
    }

    public Long getId() { return id; }
    public String getBookerId() { return bookerId; }
    public void setBookerId(String bookerId) { this.bookerId = bookerId; }
    public Long getPropertyId() { return propertyId; }
    public void setPropertyId(Long propertyId) { this.propertyId = propertyId; }
    public Date getStartDate() { return startDate; }
    public void setStartDate(Date startDate) { this.startDate = startDate; }
    public Date getEndDate() { return endDate; }
    public void setEndDate(Date endDate) { this.endDate = endDate; }
    public Status getStatus() { return status; }
    public void setStatus(Status status) { this.status = status; }
}
```

Figure 49 – Traditional Development Booking's Model

It is important to note that the communication between microservices was done using Data Transfer Objects (DTOs), as can be seen in Figure 50.

```java
package com.bcm.bookings.models.DTO;

public class VerifyStayDTO {

    private Long bookingId;
    private String userId;

    public VerifyStayDTO(){

    }

    public VerifyStayDTO(Long bookingId, String userId) {
        this.bookingId = bookingId;
        this.userId = userId;
    }

    public Long getBookingId() {
        return bookingId;
    }

    public void setBookingId(Long bookingId) {
        this.bookingId = bookingId;
    }

    public String getUserId() {
        return userId;
    }

    public void setUserId(String userId) {
        this.userId = userId;
    }
}
```

Figure 50 – Traditional Development VerifyStayDTO

- Controllers

  The developed controllers are REST controllers and use two annotations, the “Request Mapping” annotation to specify the request route, and the “RestController” annotation for spring to auto-detect implementation classes through the classpath scanning. As an example, the Booking’s Controller can be seen in Figure 51.
  Dependency injection is used for the “BookingsService” and the “RestTemplate”.

```java
@RequestMapping("/api/bookings")
@RestController
public class BookingsController {

    private static final String propertiesUrl = "http://localhost:8081/api/property/";

    private final BookingsService bookingsService;
    private final RestTemplate restTemplate;

    @Autowired
    public BookingsController(BookingsService bookingsService, RestTemplateBuilder restTemplateBuilder) {
        this.bookingsService = bookingsService;
        this.restTemplate = restTemplateBuilder.build();
    }

    @PostMapping
    public Booking addBooking(HttpServletRequest request, @RequestBody Booking booking){
        // send GET request
        ResponseEntity<PropertyDTO> propertyDTO = restTemplate.getForEntity( url: propertiesUrl+booking.getPropertyId(), PropertyDTO.class);
        if(propertyDTO.getStatusCode()!= HttpStatus.OK || !propertyDTO.hasBody()) return null;
        return bookingsService.addBooking(booking);
    }

    @GetMapping
    public List<Booking> getBookings(){
        return bookingsService.getBookings();
    }

    @PostMapping(path = "checkAvailability")
    public boolean checkAvailability(@RequestBody AvailabilityDTO availabilityDTO){
        return bookingsService.checkAvailability(availabilityDTO.getPropertyId(), availabilityDTO.getFromDate(), availabilityDTO.getToDate());
    }

    @PostMapping(path = "verifyStay")
    public boolean verifyStay(@RequestBody VerifyStayDTO verifyStayDTO){
        return bookingsService.verifyStay(verifyStayDTO.getBookingId(), verifyStayDTO.getUserId());
    }

    @PostMapping(path = "confirmBooking")
    public boolean confirmBooking(@RequestBody ConfirmBookingDTO confirmBookingDTO){
        return bookingsService.confirmBooking(confirmBookingDTO.getUserId(),confirmBookingDTO.getBookingId());
    }
}
```

Figure 51 – Traditional Development Bookings' Controller

The BookingsController (Figure 51) handles REST requests from the "/api/bookings" route, and can add and get bookings, as well has check the availability of a property on some given dates, verify if a stay was really booked (to let the user add a comment to that specific booking), and confirm bookings (update status after the payment).

- Services

  The developed services implement the business logic, and use the "Service" annotation (for spring to auto-detect implementation classes through the classpath scanning). Dependency injection is used for the "BookingsRepository".

```java
@Service
public class BookingsService {

    private final BookingsRepository bookingsRepository;

    @Autowired
    public BookingsService(BookingsRepository bookingsRepository) { this.bookingsRepository = bookingsRepository; }

    public Booking addBooking(Booking booking) {
        if (booking.getStartDate().after(booking.getEndDate())) return null;

        if (checkAvailability(booking.getPropertyId(), booking.getStartDate(), booking.getEndDate())) {
            return this.bookingsRepository.save(booking);
        } else {
            return null;
        }
    }

    public List<Booking> getBookings() {
        return bookingsRepository.findAll();
    }

    public boolean checkAvailability(Long propertyId, Date fromDate, Date toDate) {
        //Search Bookings from that property that match that time frame
        if (fromDate.after(toDate)) return false;

        List<Booking> conflicts = this.bookingsRepository.findByPropertyAndDates(propertyId, fromDate, toDate);
        return conflicts.isEmpty();
    }

    public boolean confirmBooking(String userId, Long bookingId) {
        Optional<Booking> optBooking = bookingsRepository.findById(bookingId);
        if (optBooking.isPresent()) {
            Booking booking = optBooking.get();

            if (!booking.getBookerId().equals(userId)) return false;

            booking.setStatus(Status.CONFIRMED);
            bookingsRepository.save(booking);
            return true;
        } else {
            return false;
        }
    }

    public boolean verifyStay(Long bookId, String userId) {
        Optional<Booking> optBooking = bookingsRepository.findByIdAndBookerId(bookId, userId);
        return optBooking.isPresent();
    }
}
```

Figure 52 – Traditional Development Bookings' Service

The BookingsService (Figure 52) handles handles the business logic related to the bookings. The requests come from the BookingsController, and are then processed by this service, which uses the BookingsRepository to manage the bookings' data.

- Repositories

The data persistence was done using H2 in-memory databases, and the Repositories extend the JPARepository, that provides a simple way to manage data using the Entity annotation.

```java
package com.bcm.bookings.repository;

import com.bcm.bookings.models.Booking;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;

import java.util.Date;
import java.util.List;
import java.util.Optional;

@Repository
public interface BookingsRepository extends JpaRepository<Booking,Long> {

    @Query("select b from Booking b where b.propertyId= ?1 and b.startDate <= ?3 and b.endDate>= ?2")
    List<Booking> findByPropertyAndDates(Long propertyId, Date fromDate, Date toDate);

    //JPA Query https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.named-queries
    Optional<Booking> findByIdAndBookerId(Long id, String bookerId);
}
```

Figure 53 – Traditional Development Bookings' Repository

The BookingsRepository (Figure 53) manages the Bookings and provides queries to find bookings by property and dates (to check a property's availability), and to find bookings by id and booker id (to verify the existence of a given booking, and its respective booker).

5.3 DSL-based approach Development

The DSL-based approach was developed using Eclipse Modeling Framework (EMF), Xtext, and Acceleo, and can be divided in 2 parts:

- The DSL that allows the microservices specification (EMF and Xtext)
- The code generation component that uses the DSL (Acceleo)

The main objective of this approach is to develop microservices as similar as possible to the Traditional Development approach, generating the code from the domain specification.

A visual representation of the developed DSL can be seen in Figure 54 and contains the following classes:

- Application
- BoundedContext
- Microservice
- Properties and Property – The specified properties will be placed in application.properties.
- Entity, Field, BaseField and SubClass – The fields can be of type BaseField (for example integers, booleans and strings) or of type SubClass (which represents another existing Entity).
- Enum – Will generate an enum helper class.
- Dependencies and Dependency – The dependencies will be added to the pom.xml.

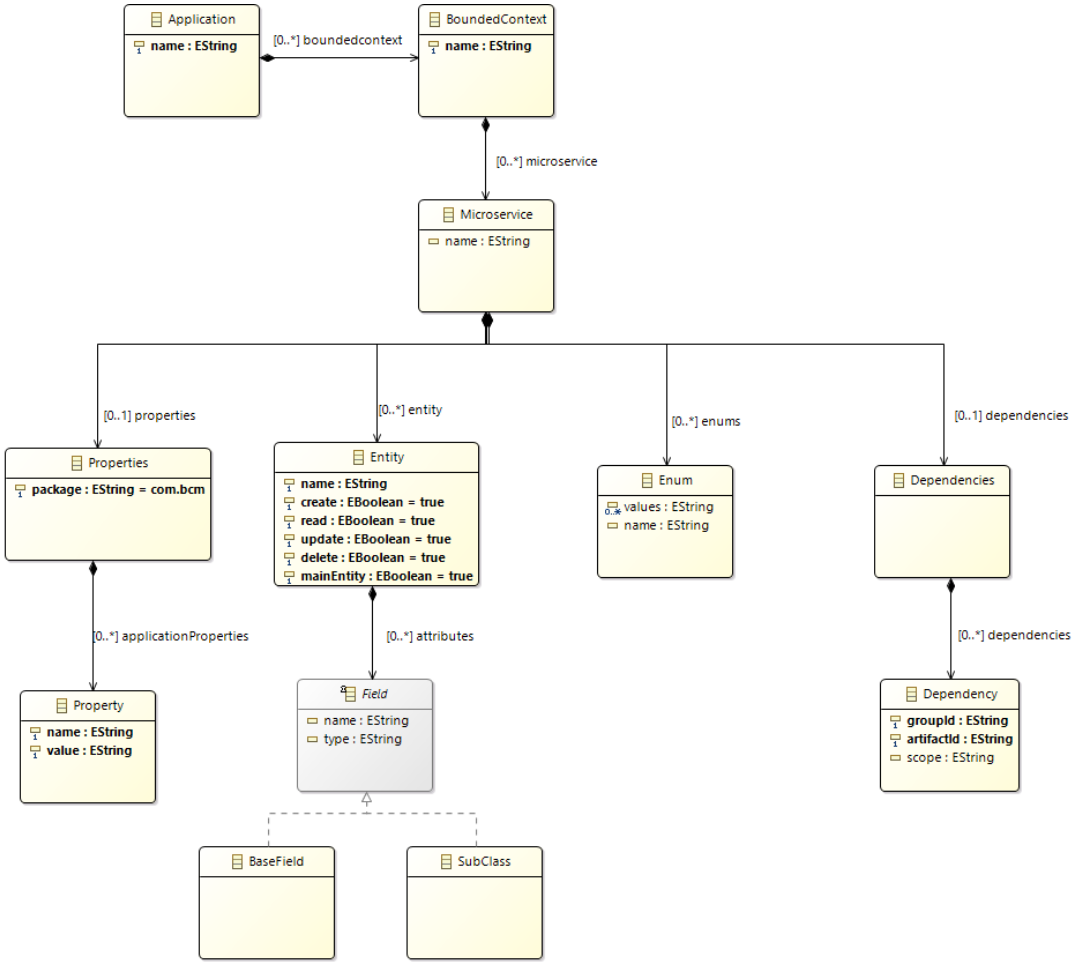

Figure 54 – DSL visual representation (Ecore model)

The DSL was developed using Ecore models, that be seen in Figure 54, and using them to generate an Xtext Project with an Xtext DSL (*Xtext*, 2021), that can be seen in Figure 55.

The generated Xtext project contains both the Xtext DSL and a workflow specification (*Xtext - MWE 2*, 2021, p. 2) that can be used to configure theDSL base package and file extension, the DSL formatter, plugins and artefacts generation.

The developed DSL was designed according to the objectives and requirements, namely to develop simple microservices, supporting the creating, reading, updating and deleting of Entities.

MicroservicesDSL.xtext

```
// automatically generated by Xtext
grammar dei.isep.ipp.pt.xtext.microservicesDSL.MicroservicesDSL with org.eclipse.xtext.common.Terminals

import "http://www.dei.isep.ipp.pt/microservicesDSL"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Application returns Application:
    {Application}
    'Application'
    name=EString
    '{'
        ('boundedcontext' '{' boundedcontext+=BoundedContext ( "," boundedcontext+=BoundedContext)* '}' )?
    '}';


Field returns Field:
    BaseField | SubClass;



EString returns ecore::EString:
    STRING | ID;

BoundedContext returns BoundedContext:
    {BoundedContext}
    'BoundedContext'
    name=EString
    '{'
        ('microservice' '{' microservice+=Microservice ( "," microservice+=Microservice)* '}' )?
    '}';

Microservice returns Microservice:
    {Microservice}
    'Microservice'
    name=EString
    '{'
        ('entity' '{' entity+=Entity ( "," entity+=Entity)* '}' )?
        ('enums' '{' enums+=Enum ( "," enums+=Enum)* '}' )?
        ('properties' properties=Properties)?
        ('dependencies' dependencies=Dependencies)?
    '}';

Entity returns Entity:
    'Entity'
    name=EString
    '{'
        'create' create=EBoolean
        'read' read=EBoolean
        'update' update=EBoolean
        'delete' delete=EBoolean
        'mainEntity' mainEntity=EBoolean
        ('attributes' '{' attributes+=Field ( "," attributes+=Field)* '}' )?
    '}';

Enum returns Enum:
    {Enum}
    'Enum'
    name=EString
    '{'
        ('values' '{' values+=EString ( "," values+=EString)* '}' )?
    '}';

Properties returns Properties:
    'Properties'
    '{'
        'package' package=EString
        ('applicationProperties' '{' applicationProperties+=Property ( "," applicationProperties+=Property)* '}' )?
    '}';

Dependencies returns Dependencies:
    {Dependencies}
    'Dependencies'
    '{'
        ('dependencies' '{' dependencies+=Dependency ( "," dependencies+=Dependency)* '}' )?
    '}';

EBoolean returns ecore::EBoolean:
    'true' | 'false';

BaseField returns BaseField:
    {BaseField}
    'BaseField'
    name=EString
    '{'
        ('type' type=EString)?
    '}';

SubClass returns SubClass:
    {SubClass}
    'SubClass'
    name=EString
    '{'
        ('type' type=EString)?
    '}';

Property returns Property:
    'Property'
    name=EString
    '{'
        'value' value=EString
    '}';

Dependency returns Dependency:
    'Dependency'
    '{'
        'groupId' groupId=EString
        'artifactId' artifactId=EString
        ('scope' scope=EString)?
    '}';

```

Figure 55 – Xtext DSL

The case study's microservices' code can then be generated using Acceleo, with a dynamic instance or with a DSL instance, in this case a DSL instance that can be seen on Figure 56 and Figure 57.

msdsl.msdsl

```
Application BookingsApp {
    boundedcontext {
        BoundedContext ^Property {
            microservice {
                Microservice ^Properties {
                    entity {
                        Entity ^Property {
                            create true
                            read true
                            update true
                            delete true
                            mainEntity true
                            attributes {
                                BaseField title { type String } ,
                                BaseField pricePerNight { type float } ,
                                BaseField ownerId { type String } ,
                                SubClass address { type Address }
                            }
                        } ,
                        Entity Address {
                            create false
                            read false
                            update false
                            delete false
                            mainEntity false
                            attributes {
                                BaseField country { type String } ,
                                BaseField country { type String } ,
                                BaseField postalCode { type String } ,
                                BaseField address { type String }
                            }
                        }
                    } properties Properties {
                        package "com.bcm"
                        applicationProperties {
                            Property "server.port" { value "8081" } ,
                            Property "spring.h2.console.enabled" { value ^true } ,
                            Property "spring.datasource.platform" { value h2 } ,
                            Property "spring.datasource.url" { value "jdbc:h2:mem:mydb" }
                        }
                    }
                }
            }
        } ,
        BoundedContext Booking {
            microservice {
                Microservice Bookings {
                    entity {
                        Entity Booking {
                            create true
                            read true
                            update true
                            delete true
                            mainEntity true
                            attributes {
                                BaseField bookerId { type String } ,
                                BaseField propertyId { type Long } ,
                                BaseField startDate { type Date } ,
                                BaseField endDate { type Date } ,
                                SubClass status { type Status }
                            }
                        }
                    }
                    enums {
                        Enum Status { values { CONFIRMED , UNCOFIRMED } }
                    }
                    properties Properties {
                        package "com.bcm"
                        applicationProperties {
                            Property "server.port" { value "8082" } ,
                            Property "spring.h2.console.enabled" { value ^true } ,
                            Property "spring.datasource.platform" { value h2 } ,
                            Property "spring.datasource.url" { value "jdbc:h2:mem:mydb" }
                        }
                    }
                }
            }
        } ,
```

Figure 56 – Case Study's DSL instance (Property and Booking)

```
        BoundedContext Review {
            microservice {
                Microservice Reviews {
                    entity {
                        Entity Review {
                            create true
                            read true
                            update true
                            delete true
                            mainEntity true
                            attributes {
                                BaseField userId { type String } ,
                                BaseField bookingId { type Long } ,
                                BaseField title { type String } ,
                                BaseField comment { type String } ,
                                BaseField rating { type int }
                            }
                        }
                    }
                    properties Properties {
                        package "com.bcm"
                        applicationProperties {
                            Property "server.port" { value "8083" } ,
                            Property "spring.h2.console.enabled" { value ^true } ,
                            Property "spring.datasource.platform" { value h2 } ,
                            Property "spring.datasource.url" { value "jdbc:h2:mem:mydb" }
                        }
                    }
                }
            }
        } ,
        BoundedContext Payment {
            microservice {
                Microservice Payments {
                    entity {
                        Entity Payment {
                            create true
                            read true
                            update true
                            delete true
                            mainEntity true
                            attributes {
                                BaseField userId { type String } ,
                                BaseField bookingId { type Long }
                            }
                        }
                    }
                    properties Properties {
                        package "com.bcm"
                        applicationProperties {
                            Property "server.port" { value "8084" } ,
                            Property "spring.h2.console.enabled" { value ^true } ,
                            Property "spring.datasource.platform" { value h2 } ,
                            Property "spring.datasource.url" { value "jdbc:h2:mem:mydb" }
                        }
                    }
                }
            }
        }
    }
}
```

Figure 57 - Case Study's DSL instance (Review and Payment)

The defined instance follows the Design, defining the four microservices (Properties, Bookings, Reviews, and Payments) with their respective entities.

The properties for each microservice is also defined in the given instance and must be used in the code generation to define the microservice's properties.

Acceleo uses a mtl file to specify the code generation (model-to-text transformation), which can be seen on Appendix B.

The developed transformation creates a maven project for each microservice, creating the pom.xml using the microservices' specified dependencies.

For each microservice the transformation creates:

- A SpringBootApplication to execute the microservice.
- One application.properties file that contains the specified microservice's properties.
- One domain class for each microservice's entity or enum.
- One REST controller for each microservice's main entity.
- One service for each microservice's main entity, that handles the business logic.
- One JPARepository for each microservice's main entity, that manages the data persistency of a specific entity.

The resulting microservices follow a similar structure to the one of the Traditional Development, and the classes are identical, and support the create, read, update, and delete of the specified entities. However, the communication between microservices and other microservice specific business logic must be done manually.

An example of a generated REST controller can be seen in Figure 58. As mentioned before, the generated REST controllers are like the ones obtained on the Traditional Development, but do not include communication between microservices and specific business logic.

PropertyController.java

```java
package com.bcm.properties.controllers;

import ...

@RequestMapping("/api/property")
@RestController
public class PropertyController {

    private final PropertyService propertyService;

    @Autowired
    public PropertyController(PropertyService propertyService) { this.propertyService = propertyService; }

    @PostMapping
    public Property addProperty(@RequestBody Property property) { return propertyService.saveProperty(property); }

    @GetMapping
    public List<Property> getAllProperty() { return propertyService.getAllProperty(); }

    @GetMapping(path = "{id}")
    public Optional<Property> getPropertyById(@PathVariable("id") Long id){
        return propertyService.getPropertyById(id);
    }

    @PutMapping
    public Property updateProperty(@RequestBody Property property) { return propertyService.updateProperty(property); }

    @DeleteMapping(path = "{id}")
    public void deletePropertyById(@PathVariable("id") Long id) { propertyService.deleteProperty(id); }
}
```

Figure 58 – DSL-based approach REST Controller

The generated services (of which an example can be seen in Figure 59) are also like the ones obtained on the previous approach.

PropertyService.java

```java
package com.bcm.properties.services;

import com.bcm.properties.models.Property;
import com.bcm.properties.repositories.PropertyRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;

@Service
public class PropertyService {

    private PropertyRepository propertyRepo;

    @Autowired
    public PropertyService(PropertyRepository propertyRepo) { this.propertyRepo = propertyRepo; }

    public Property saveProperty(Property property) { return propertyRepo.save(property); }

    public List<Property> getAllProperty() { return propertyRepo.findAll(); }

    public Optional<Property> getPropertyById(Long id) { return propertyRepo.findById(id); }

    public void deleteProperty(Long id){
        if(id!=null){
            if(getPropertyById(id).isPresent())
                propertyRepo.deleteById(id);
        }
    }

    public Property updateProperty(Property property) { return propertyRepo.save(property); }
}
```

Figure 59 – DSL-based approach Service

As for the generated repositories, they are an extension of the JPARepository, just like in the previous approach, the main difference being this one does not include any additional queries.

PropertyRepository.java

```java
package com.bcm.properties.repositories;

import com.bcm.properties.models.Property;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface PropertyRepository extends JpaRepository<Property,Long> {

}
```

Figure 60 – DSL-based approach Repository

The generated domain classes are identical to the Traditional Development ones, and include fields, constructors, and getters and setters. An example of a generated domain class can be seen in Figure 61.

Property.java

```java
package com.bcm.properties.models;

import ...

@Entity
public class Property {

    //variables
    @Id
    @GeneratedValue//(generator="system-uuid")
    private Long id;

    private String title;
    private float pricePerNight;
    private String ownerId;
    private Address address;

    //constructors
    public Property(){

    }

    public Property(
        String title,
        float pricePerNight,
        String ownerId,
        Address address

    ) {
        this.title = title;
        this.pricePerNight = pricePerNight;
        this.ownerId = ownerId;
        this.address = address;
    }


    //gets and sets
    public Long getId() {
        return id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String  title) {
        this.title = title;
    }
    public float getPricePerNight() {
        return pricePerNight;
    }

    public void setPricePerNight(float  pricePerNight) { this.pricePerNight = pricePerNight; }
    public String getOwnerId() { return ownerId; }

    public void setOwnerId(String  ownerId) { this.ownerId = ownerId; }
    public Address getAddress() { return address; }

    public void setAddress(Address  address) { this.address = address; }

}
```

Figure 61 – DSL-based approach Domain class

5.4 MDE-based tool approach Development

This approach consists in using MDE-based tools to develop the microservices code from the domain, using ContextMapper (*ContextMapper*, 2021) and JHipster (JHipster, 2021).

Context Mapper allows model description in a context map (*Context Map*, 2021), that can then be converted into a JHipster Domain Language (JDL) file, using the generic generator (templating based on Freemarker) provided by Context Mapper.

The designed context map that describes the case study can be seen in Figure 62 and Figure 63.

The defined context map aims to follow the domain model (Figure 39) as similarly as possible, and just like in the domain model it defines four bounded contexts, the Properties Bounded Context, the Bookings Bounded Context, the Reviews Bounded Context, and the Payments Bounded Context. The entities on each bounded context are the same as specified in the domain model.

```
/* Example Context Map written with 'ContextMapper DSL' */
ContextMap BookingsContextMap {
    type = SYSTEM_LANDSCAPE
    state = TO_BE

    /* Add bounded contexts to this context map: */
    contains PropertiesContext
    contains BookingsContext
    contains PaymentContext
    contains ReviewContext

    /* Define the context relationships: */

}

/* Bounded Context Definitions */
BoundedContext PropertiesContext implements PropertiesDomain {
    type = APPLICATION
    domainVisionStatement = "The property microservice is responsible for managing the properties."
    implementationTechnology = "Spring Boot Microservice"
    responsibilities = "Properties, Addresses"

    Aggregate Properties {
        Entity Property {
            aggregateRoot

            String title
            - Address location
            Float pricePerNight
            String propertyOwnerId
            def AddressId createAddress(@Address address);
        }
        Entity Address {
            String country
            String city
            String postalCode
            String address
        }

    }
}

BoundedContext BookingsContext implements BookingsSubDomain {
    type = APPLICATION
    domainVisionStatement = "The property microservice is responsible for managing the bookings."
    implementationTechnology = "Spring Boot Microservice"
    responsibilities = "Bookings"

    Aggregate Bookings {
        Entity Booking {
            aggregateRoot

            String bookerId
            Long propertyId
            DateTime startDate
            DateTime endDate
            - Status status

        }

        enum Status {
            CONFIRMED,
            UNCONFIRMED
        }
    }
}
```

Figure 62 – ContextMap – Properties and Bookings Context

```
67
68  BoundedContext PaymentContext implements PaymentDomain {
69      type = APPLICATION
70      domainVisionStatement = "The property microservice is responsible for managing the payments."
71      implementationTechnology = "Spring Boot Microservice"
72      responsibilities = "Payments"
73
74      Aggregate Payments {
75          Entity Payment {
76              aggregateRoot
77
78              String paymentUserId
79              Long bookingId
80          }
81      }
82  }
83
84  BoundedContext ReviewContext implements ReviewDomain {
85      type = APPLICATION
86      domainVisionStatement = "The property microservice is responsible for managing the reviews."
87      implementationTechnology = "Spring Boot Microservice"
88      responsibilities = "Reviews"
89
90      Aggregate Reviews {
91          Entity Review {
92              aggregateRoot
93
94              String reviewerId
95              Long bookingId
96              String title
97              String comment
98
99          }
100     }
101 }
102
103 /* Domain & Subdomain Definitions */
104 Domain BookingsDomain {
105     Subdomain PropertiesDomain {
106         type = CORE_DOMAIN
107         domainVisionStatement = "Subdomain managing contracts and policies."
108     }
109     Subdomain BookingsSubDomain {
110         type = CORE_DOMAIN
111         domainVisionStatement = "Subdomain managing contracts and policies."
112     }
113     Subdomain PaymentDomain {
114         type = SUPPORTING_DOMAIN
115         domainVisionStatement = "Subdomain supporting the payment."
116     }
117     Subdomain ReviewDomain {
118         type = CORE_DOMAIN
119         domainVisionStatement = "Subdomain responsible for the reviews."
120     }
121 }
122
```

Figure 63 – Context Map – Payment and Reviews Context

The result from the generic generator transformation is the JDL file that can be used on JHipster to generate the microservices and can be seen in Appendix C.

JHipster can generate microservices from a JDL file by using the command:

```
jhipster import-jdl <jdl file>
```

To execute the microservices it is first required to launch the JHipster Registry (*JHipster Registry*, 2021), that has explain in section 2.2.1, handles the microservice discovery, and provides a Spring Cloud Config Server and an administration server.

JHipster generates microservices and a gateway that handles the Web traffic and serves as a frontend application (using Angular), the projects can be executed using maven.

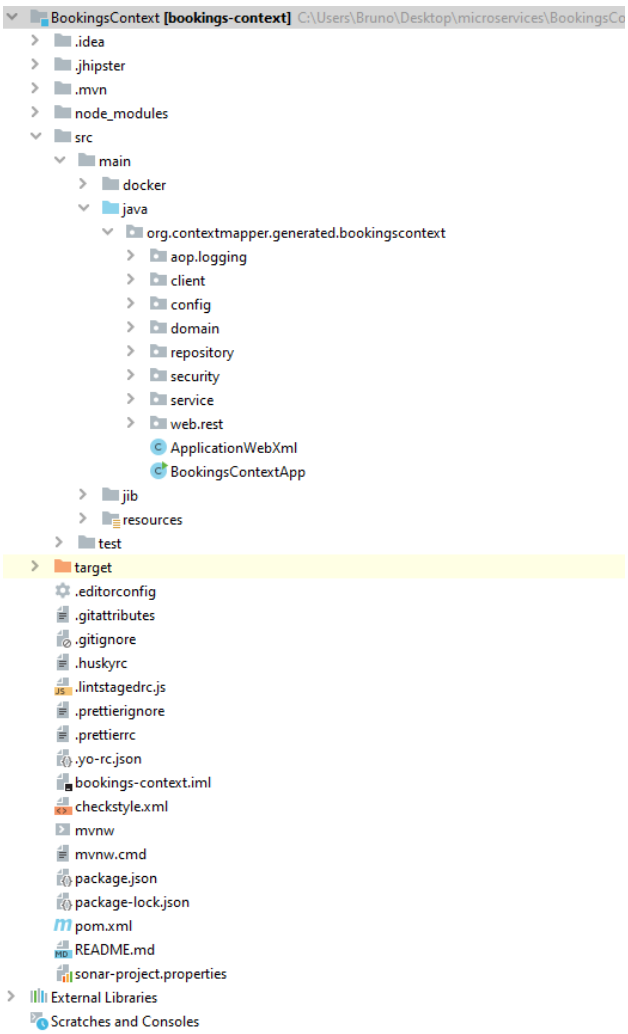

Figure 64 – JHipster generated microservice structure

Each generated microservice follows the structure visible on Figure 64, which includes:

- Docker configurations to run the microservice on a containerized environment
- Aspect Oriented Programming (AOP) logging
- Security (Authentication and Authorization, using JWT tokens)
- REST Controllers
- Domain entities
- Service layer with interfaces, DTOs and mappers (to and from DTOs)
- JPARepositories with in-memory data persistence
- Unit tests

The resulting microservices follow a different structure to the one of the Traditional Development, but they also support the create, read, update, and delete of the specified entities. However, just like in the DSL-based approach Development, the communication between microservices and other microservice specific logic must be done manually.

An example of a generated REST controller can be seen in Figure 65, the REST controllers also include requests to handle the read, update, and delete of the entities.

PropertyResource.java

```java
package org.contextmapper.generated.propertiescontext.web.rest;

import ...

/**
 * REST controller for managing {@link org.contextmapper.generated.propertiescontext.domain.Property}.
 */
@RestController
@RequestMapping("/api")
public class PropertyResource {

    private final Logger log = LoggerFactory.getLogger(PropertyResource.class);

    private static final String ENTITY_NAME = "propertiesContextProperty";

    @Value("${jhipster.clientApp.name}")
    private String applicationName;

    private final PropertyService propertyService;

    public PropertyResource(PropertyService propertyService) { this.propertyService = propertyService; }

    /**
     * {@code POST  /properties} : Create a new property.
     *
     * @param propertyDTO the propertyDTO to create.
     * @return the {@link ResponseEntity} with status {@code 201 (Created)} and with body the new propertyDTO, or with status {@code 400 (Bad Request)}
     * if the property has already an ID.
     * @throws URISyntaxException if the Location URI syntax is incorrect.
     */
    @PostMapping("/properties")
    public ResponseEntity<PropertyDTO> createProperty(@RequestBody PropertyDTO propertyDTO) throws URISyntaxException {
        log.debug("REST request to save Property : {}", propertyDTO);
        if (propertyDTO.getId() != null) {
            throw new BadRequestAlertException("A new property cannot already have an ID", ENTITY_NAME, "idexists");
        }
        PropertyDTO result = propertyService.save(propertyDTO);
        return ResponseEntity.created(new URI("/api/properties/" + result.getId()))
            .headers(HeaderUtil.createEntityCreationAlert(applicationName, true, ENTITY_NAME, result.getId().toString()))
            .body(result);
    }

```

Figure 65 – MDE-based tool approach REST Controller

The generated services (of which an example can be seen in Figure 66) implement an interface and use the specific repository to handle the create, read, update, and delete of entities. It is important to note the use of DTO's and the included logging.

PropertyService.java | PropertyServiceImpl.java

```java
package org.contextmapper.generated.propertiescontext.service.impl;

import ...

/**
 * Service Implementation for managing {@link Property}.
 */
@Service
@Transactional
public class PropertyServiceImpl implements PropertyService {

    private final Logger log = LoggerFactory.getLogger(PropertyServiceImpl.class);

    private final PropertyRepository propertyRepository;

    private final PropertyMapper propertyMapper;

    public PropertyServiceImpl(PropertyRepository propertyRepository, PropertyMapper propertyMapper) {
        this.propertyRepository = propertyRepository;
        this.propertyMapper = propertyMapper;
    }

    @Override
    public PropertyDTO save(PropertyDTO propertyDTO) {
        log.debug("Request to save Property : {}", propertyDTO);
        Property property = propertyMapper.toEntity(propertyDTO);
        property = propertyRepository.save(property);
        return propertyMapper.toDto(property);
    }

    @Override
    @Transactional(readOnly = true)
    public List<PropertyDTO> findAll() {
        log.debug("Request to get all Properties");
        return propertyRepository.findAll().stream()
            .map(propertyMapper::toDto)
            .collect(Collectors.toCollection(LinkedList::new));
    }


    @Override
    @Transactional(readOnly = true)
    public Optional<PropertyDTO> findOne(Long id) {
        log.debug("Request to get Property : {}", id);
        return propertyRepository.findById(id)
            .map(propertyMapper::toDto);
    }

    @Override
    public void delete(Long id) {
        log.debug("Request to delete Property : {}", id);
        propertyRepository.deleteById(id);
    }
}
```

Figure 66 – MDE-based tool approach Service

The generated repositories are similar to the other approaches, consisting in a simple extension of the JPARepository. An example of the generated repositories can be seen in Figure 67.

```
1  package org.contextmapper.generated.propertiescontext.repository;
2
3  import ...
7
8  /**
9   * Spring Data  repository for the Property entity.
10  */
11 /unused/
12 @Repository
13 public interface PropertyRepository extends JpaRepository<Property, Long> {
14 }
```

Figure 67 – MDE-based tool approach Repository

An example of a generated domain class can be seen in Figure 68, in this case the Property class. The generated class includes the fields, constructors, gets and setters, and the equals, hashCode and toString methods.

Property.java

```java
package org.contextmapper.generated.propertiescontext.domain;

import ...

/**
 * A Property.
 */
@Entity
@Table(name = "property")
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Property implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "title")
    private String title;

    @Column(name = "price_per_night")
    private Float pricePerNight;

    @Column(name = "property_owner_id")
    private String propertyOwnerId;

    @OneToOne
    @JoinColumn(unique = true)
    private Address location;

    // jhipster-needle-entity-add-field - JHipster will add fields here
    public Long getId() { return id; }

    public void setId(Long id) { this.id = id; }

    public String getTitle() { return title; }

    public Property title(String title) {
        this.title = title;
        return this;
    }

    public void setTitle(String title) { this.title = title; }

    public Float getPricePerNight() { return pricePerNight; }

    public Property pricePerNight(Float pricePerNight) {
        this.pricePerNight = pricePerNight;
        return this;
    }

    public void setPricePerNight(Float pricePerNight) { this.pricePerNight = pricePerNight; }

    public String getPropertyOwnerId() { return propertyOwnerId; }

    public Property propertyOwnerId(String propertyOwnerId) {
        this.propertyOwnerId = propertyOwnerId;
        return this;
    }

    public void setPropertyOwnerId(String propertyOwnerId) { this.propertyOwnerId = propertyOwnerId; }

    public Address getLocation() { return location; }

    public Property location(Address address) {
        this.location = address;
        return this;
    }

    public void setLocation(Address address) { this.location = address; }
    // jhipster-needle-entity-add-getters-setters - JHipster will add getters and setters here

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (!(o instanceof Property)) {
            return false;
        }
        return id != null && id.equals(((Property) o).id);
    }

    @Override
    public int hashCode() { return 31; }

    // prettier-ignore
    @Override
    public String toString() {
        return "Property{" +
            "id=" + getId() +
            ", title='" + getTitle() + "'" +
            ", pricePerNight=" + getPricePerNight() +
            ", propertyOwnerId='" + getPropertyOwnerId() + "'" +
            "}";
    }
}
```

Figure 68 – MDE-based tool Domain class

As mentioned before, the generated microservices also include DTO's and mappers, of which examples can be seen in Figure 69 and Figure 70 respectively.

PropertyDTO.java

```java
package org.contextmapper.generated.propertiescontext.service.dto;

import java.io.Serializable;

/**
 * A DTO for the {@link org.contextmapper.generated.propertiescontext.domain.Property} entity.
 */
public class PropertyDTO implements Serializable {

    private Long id;

    private String title;

    private Float pricePerNight;

    private String propertyOwnerId;


    private Long locationId;

    public Long getId() { return id; }

    public void setId(Long id) { this.id = id; }

    public String getTitle() { return title; }

    public void setTitle(String title) { this.title = title; }

    public Float getPricePerNight() { return pricePerNight; }

    public void setPricePerNight(Float pricePerNight) { this.pricePerNight = pricePerNight; }

    public String getPropertyOwnerId() { return propertyOwnerId; }

    public void setPropertyOwnerId(String propertyOwnerId) { this.propertyOwnerId = propertyOwnerId; }

    public Long getLocationId() {
        return locationId;
    }

    public void setLocationId(Long addressId) { this.locationId = addressId; }

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (!(o instanceof PropertyDTO)) {
            return false;
        }

        return id != null && id.equals(((PropertyDTO) o).id);
    }

    @Override
    public int hashCode() { return 31; }

    // prettier-ignore
    @Override
    public String toString() {
        return "PropertyDTO{" +
            "id=" + getId() +
            ", title='" + getTitle() + "'" +
            ", pricePerNight=" + getPricePerNight() +
            ", propertyOwnerId='" + getPropertyOwnerId() + "'" +
            ", locationId=" + getLocationId() +
            "}";
    }
}
```

Figure 69 – MDE-based tool DTO

PropertyMapper.java

```java
package org.contextmapper.generated.propertiescontext.service.mapper;


import ...

/**
 * Mapper for the entity {@link Property} and its DTO {@link PropertyDTO}.
 */
@Mapper(componentModel = "spring", uses = {AddressMapper.class})
public interface PropertyMapper extends EntityMapper<PropertyDTO, Property> {

    @Mapping(source = "location.id", target = "locationId")
    PropertyDTO toDto(Property property);

    @Mapping(source = "locationId", target = "location")
    Property toEntity(PropertyDTO propertyDTO);

    default Property fromId(Long id) {
        if (id == null) {
            return null;
        }
        Property property = new Property();
        property.setId(id);
        return property;
    }
}
```

Figure 70 – MDE-based tool Mapper

6 Analysis and Evaluation

As stated in the Expected Results section, the approaches will be compared and analysed in order to discover if there is a significant different between the approaches in the development time and obtained code quality.

For the DSL-based approach only the generated code is taken into consideration, seeing as if the manual alterations (to implement the functional requirements) were taken into consideration, then the result would be the same as the Traditional Development.

It is expected that the code quality between the traditional development and DSL-based approach is similar, as the DSL-based approach attempts to generate code as like the traditional development as possible.

As for the MDE-based tool approach, as there is less control on the code generation process, it is expected that the code base differs significantly, possibly also differing in code quality.

6.1 Development Time

The approaches follow different processes and have different development time, which can be seen in Table 8.

This development time does not include the analysis and design of the case study, nor the time required to study and learn the different tools and technologies, which represent a considerable amount of time.

Table 8 – Case study's development time

Approach	Time (in hours)
Traditional Development	6
DSL based approach	13
MDE tool based approach	5

The time of development differs significantly, however, it is also important to note that these approaches have different results and might be used in different scenarios.

The DSL based approach is the one that required more time, as it includes the elaboration of a DSL, and the code generator. Of the 13 hours required half was for the DSL development and the other half for the code generator development.

6.2 Code Quality

The code quality was analysed using the Chidamber and Kemerer's metrics (Chidamber & Kemerer, 1994). The Chidamber and Kemerer's metrics suite is well known and widely accepted by the software engineering community, as it is based on measurement theory and oriented to OO programming languages (which is the case for the Java microservices).

The Kidamber and Kemerer's metrics suite consists in 6 metrics (*Chidamber & Kemerer Object-Oriented Metrics Suite*, 2021):

- **WMC: Weighted methods per class**
  WMC is simply the number of methods per class. A high WMC has been found to lead to more faults, as it leads to clustered code, less organized and more difficult to read and debug, overall increasing the effort required to develop and maintain the class.

- **DIT: Depth of inheritance tree**
  DIT is the maximum inheritance path from the class to root class. The deeper a class is in the hierarchy tree, the more methods, and variables it is likely to inherit, making it more complex.
  Deep trees indicate greater design complexity, however, they also promote reuse.

- **NOC: Number of children**
  NOC represents the number of children of a given class (immediate sub-classes). NOC measures the breadth of a class hierarchy, whereas DIT measures the depth.

A high NOC is usually a positive indicator, and it can indicate:

- High reuse of the base class (Inheritance).
- Base class may require more testing.
- Improper abstraction of the parent class.
- Misuse of sub-classing.

It is important to note that classes higher up in the hierarchy should have more sub-classes than the lower ones.

- **CBO: Coupling between object classes**
  CBO represents the number of classes to which a class is coupled, including the "uses" and the "used-by" relationships.
  High CBO is undesirable, as it leads to excessive coupling, making classes more difficult to reuse.

- **RFC: Response for a Class**
  RFC represents the set of methods that can potentially be executed in response to a request received by an object of that class.
  A high RFC is a negative indicator, as it points to high complexity.

- **LCOM: Lack of cohesion in methods**
  LCOM represents the lack of cohesion between methods, by counting the sets of methods in a class that are not related through the sharing of some of the class's fields.
  A high LCOM may indicate that the class can be divided into two or more sub-classes.

The Software Assurance Technology Center (SATC) at NASA Goddard Space Flight Center conducted a study on the Chidamber and Kemerer's metrics to define better metrics and thresholds that do a good job of discriminating between "solid" code and "fragile" code. (Rosenberg et al., 1999).

The study analysed object-oriented code written in both C++ and Java, that was collected for over three years, consisting of over 20,000 classes from more than 15 programs.

In the referred study, the following thresholds were defined:

Table 9 – Metric's Thresholds

Metric	Threshold
NOM: Number of methods	≤ 20 preferred ≤ 40 acceptable
WMC: Weighted methods per class	≤ 25 preferred ≤ 40 acceptable
DIT: Depth of inheritance tree	DIT < 2 may represent a poor exploitation of the advantages of OO design and inheritance DIT > 5 could be overkill, taking great advantage of inheritance but paying the price in complexity
NOC: Number of children	No "good" or "bad" number, however it becomes important when a class is found to have high values for other metrics
CBO: Coupling between object classes	< 5
RFC: Response for a Class	≤ 50
LCOM: Lack of cohesion in methods	Not evaluated

The metrics for each microservice were then collected using CKJM (Spinellis, 2019) and can be seen on Table 10, Table 11 and Table 12 respectively for the Traditional Development, DSL-based approach and MDE tool-based approach.

Table 10 – Traditional Development CK metrics

Metrics	Number of classes	Min	1st Quartile	Mean	3rd Quartile	Max
WMC	29	0	2	5,34	7,5	13
DIT	29	1	1	1,03	1	2
NOC	29	0	0	0	0	0
CBO	29	0	0	1,90	2	10
RFC	29	0	4	9	12	25
LCOM	29	0	0	7,86	9,5	48

Table 11 – DSL-based approach CK metrics

Metrics	Number of classes	Min	1st Quartile	Mean	3rd Quartile	Max
WMC	22	0	2	5,18	6	13
DIT	22	1	1	1,05	1	2
NOC	22	0	0	0	0	0
CBO	22	0	1	1,41	2	2
RFC	22	0	4	8,09	12	14
LCOM	22	0	0	7,5	4	48

Table 12 – MDE tool-based approach CK metrics

Metrics	Number of classes	Min	1st Quartile	Mean	3rd Quartile	Max
WMC	203	0	2	4,67	6	21
DIT	203	0	1	0,89	1	2
NOC	203	0	0	0	0	0
CBO	203	0	0	5,60	7	30
RFC	203	0	4	16,83	23	79
LCOM	203	0	0	11,82	10	174

The metrics show that all the approaches have a low DIT (<2), which may represent poor exploitation of the advantages of OO design and inheritance.

The only metric that overtakes the threshold is the CBO of the MDE-tool based approach, which indicate this approach has high coupling between objects, however it is important to mention that it takes into consideration significantly more classes than the other approaches, as it generates a heavier structure that includes, security authentication and authorization, DTOs and mappers, and configuration classes.

In order to check if there is a significant different between the approaches, we must first perform a normality test, so a Shapiro-Wilk normality test was performed.

Table 13 – Shapiro-Wilk normality test

Metrics	Traditional p-value	DSL+Acceleo p-value	CM+JH p-value
WMC	0.05403	0.02936	4.708e-14
DIT	1.315e-11	7.417e-10	2.2e-16
NOC	NA, 0 for all, no difference		
CBO	2.897e-07	6.963e-05	2.2e-16
RFC	0.1198	0.0009691	1.75e-15
LCOM	3.162e-07	4.814e-07	2.2e-16

As can be seen in Table 13, as p-value is mostly not greater than the significance value (0.05) we can conclude the data does not follow a normal distribution.

Since the data does not follow a normal distribution, in order to check if there is a significant difference between the approaches a Kruskal-Wallis rank sum test was performed (Table 14).

Table 14 - Kruskal-Wallis rank sum test

	p-value
WMC	0.3236
DIT	0.009339
NOC	NA
CBO	0.01134
RFC	0.09196
LCOM	0.2372

The Kruskal-Wallis rank sum test indicates a significant difference in DIT and CBO, but in order to know between what approaches a pairwise comparisons using Wilcoxon rank sum test was performed (Table 15).

Table 15 - Wilcoxon rank sum test

WMC

	CMJH	DSL
DSL	0.57	-
Traditional	0.57	0.90

RFC

	CMJH	DSL
DSL	0.16	-
Traditional	0.16	0.93

DIT

	CMJH	DSL
DSL	0.05	-
Traditional	0.05	0.87

LCOM

	CMJH	DSL
DSL	0.35	-
Traditional	0.55	0.55

NOC

	CMJH	DSL
DSL	-	-
Traditional	-	-

CBO

	CMJH	DSL
DSL	0.067	-
Traditional	0.047	0.459

The Wilcoxon rank sum test (Table 15) indicates that there is no significant different between the traditional development and the DSL-based approach. However, between these approaches and the MDE tool-based approach there is a significant different in CBO, namely because the MDE tool-based approach has a CBO above the threshold, indicating high coupling between the objects.

6.3 Evaluation

The metrics are overall within the specified thresholds, which indicates a good code quality for all the approaches, meaning that at least for this case study MDE can in fact be used to develop microservices without negatively impacting the code quality.

The DSL-based approach requires more time than the other approaches, which is to be expected since the amount of necessary work is higher, taking into consideration the need to develop a DSL and a code generator. On the other hand, this approach provides the best control when it comes to code generation, as both the DSL and code generator itself are done by the programmer or development team. This approach can be especially useful for companies that want to generate microservices fit to the company's requirements.

In contrast, the MDE tool-based approach requires the less time to develop the microservices, however it comes at a cost, as it provides the lowest amount of control over the generation process. It is to note that the microservices come with a long list of features that would take a considerable amount of time to develop or generate. This approach can be especially useful for quick proof of concepts or microservices' base to be further developed.

The traditional development provides full control on the resulting microservices, as it is all done manually. Consequently, if microservices' development is a recurrent process, as there is not automation it may end up costing a considerable amount of time more than the other approaches, while also being a more error and bug prone approach.

7 Conclusion

This chapter provides the summary of the results of the work done, depicting the obtained results, limitations, and future improvements. Finally, this chapter ends with a reflection of the author on the final result.

7.1 Summary

The three different approaches were compared (traditional development, DSL-based approach, and MDE-tool based approach), through the development of a case study.

The different approaches follow different processes and have different results, that can be useful in different scenarios, and overall, the code quality is good with all the tested approaches, however, in the traditional development and DSL-based approach it is fully dependent on the development team.

The traditional development takes less time than the DSL-based approach, and consists in developing each microservice from scratch, which as mentioned before is more error and bug prone, and is obviously not time efficient if microservice development is a recurrent process.

The DSL-based approach allows automated code-generation of microservices from the domain, using a hand-made DSL with full control on the generated content, which allows similar code quality to the traditional development, however it requires more time, resources, and training.

The MDE-based tool approach is the one that requires less time, however the code generation process provides a lot less control, which might result in the code being significantly different

from the other two approaches. On the other hand, it provides a lot of *out of the box* functionalities (including logging, security, microservice discovery, a gateway with frontend) with little development time, which might be perfect to develop quick proof of concepts.

7.2 Goals Achieved

This work introduces MDE and microservices, and how MDE can be used to generate Java microservices.

The main goal of this work was to compare different approaches of Java microservice development using Spring Boot, and it was achieved using the three proposed approaches, which were realized analyzed, and compared (traditional development, DSL-based approach and MDE tool-based approach).

The results were according to the expectations, seeing as the traditional development and DSL-based approach have similar code quality, and the MDE-based approach resulted in a significantly different codebase, and slightly different code quality.

This work also highlights the benefits of each approach, and how they can be used for different purposes.

7.3 Limitations and Future Work

The generated microservices was simple, and it would be interesting to generate more complex microservices, including security, documentation, tests, and possibly some communication between microservices, but it would require a considerable amount of time, which was out of the time scope available for this work.

For future work it would also be interesting to test more tools and compare the obtained results.

7.4 Final Remarks

This work allowed the author the explore some of the main concepts of the master's degree in software engineering, namely microservices and MDE.

This work provided a challenge, generating Java microservices from the domain, which could really be useful both from both academic and professional perspective.

Furthermore, seeing as both microservices and MDE are trending technologies it was a great opportunity to learn some more about them, and acquire skills that are relevant from a professional standpoint.

References

Acceleo | Home. (2021). https://www.eclipse.org/acceleo/

AjiL. (2020). [Java]. SEELAB FHDO. https://github.com/SeelabFhdo/AjiL

ATL | The Eclipse Foundation. (2021). https://www.eclipse.org/atl/

Basic notions | MPS. (2021). https://www.jetbrains.com/help/mps/basic-notions.html

Chidamber & Kemerer object-oriented metrics suite. (2021). https://www.aivosto.com/project/help/pm-oo-ck.html

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, *20*(6), 476–493. https://doi.org/10.1109/32.295895

Consul. (2021). Consul by HashiCorp. https://www.consul.io/

Context Map. (2021). Context Mapper. https://contextmapper.org/docs/context-map/

ContextMapper. (2021). Context Mapper. https://contextmapper.org/docs/home/

ContextMapper Home. (2021). Context Mapper. https://contextmapper.org/docs/home/

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, Today, and Tomorrow. In M. Mazzara & B. Meyer (Eds.), *Present and Ulterior Software Engineering* (pp. 195–216). Springer International Publishing. https://doi.org/10.1007/978-3-319-67425-4_12

Eclipse Modeling Project | The Eclipse Foundation. (2021). https://www.eclipse.org/modeling/emf/

Eclipse Sirius. (2021). [Text]. Projects.Eclipse.Org. https://projects.eclipse.org/projects/modeling.sirius

Generator | MPS. (2021). https://www.jetbrains.com/help/mps/mps-generator.html

Gray, J., Fisher, K., Consel, C., Karsai, G., Mernik, M., & Tolvanen, J.-P. (2008). DSLs: The good, the bad, and the ugly. *Companion to the 23rd ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications - OOPSLA Companion '08*, 791. https://doi.org/10.1145/1449814.1449863

Hutchinson, J., Whittle, J., & Rouncefield, M. (2014). Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming*, *89*, 144–161. https://doi.org/10.1016/j.scico.2013.03.017

JDL. (2021). https://www.jhipster.tech/jdl/

JHipster. (2021). https://www.jhipster.tech/

JHipster Microservices Architecture. (2021). https://www.jhipster.tech/microservices-architecture/

JHipster Registry. (2021). https://www.jhipster.tech/jhipster-registry/

Koen, P. A., Ajamian, G. M., Boyce, S., Clamen, A., Fisher, E., Fountoulakis, S., Johnson, A., Puri, P., & Seibert, R. (2002). Effective Methods, Tools, and Techniques. *The PDMA ToolBook for New Product Development*, 32.

Martin Fowler & James Lewis. (2014). *Microservices*. Martinfowler.Com. https://martinfowler.com/articles/microservices.html

Modeling SDK for Visual Studio—Domain-Specific Languages. (2016). MSDK - Overview. https://docs.microsoft.com/en-us/visualstudio/modeling/modeling-sdk-for-visual-studio-domain-specific-languages

Mohagheghi, P., Fernandez, M. A., Martell, J. A., Fritzsche, M., & Gilani, W. (2009). MDE Adoption in Industry: Challenges and Success Criteria. In M. R. V. Chaudron (Ed.), *Models in Software Engineering* (Vol. 5421, pp. 54–59). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-01648-6_6

MPS: The Domain-Specific Language Creator by JetBrains. (2021). JetBrains. https://www.jetbrains.com/mps/

OCL/FAQ. (2021). https://wiki.eclipse.org/OCL/FAQ

QFD. (2021). https://quality-one.com/qfd/

Rademacher, F., Sorgalla, J., Wizenty, P., Sachweh, S., & Zündorf, A. (2020). Graphical and Textual Model-Driven Microservice Development. In A. Bucchiarone, N. Dragoni, S. Dustdar, P. Lago, M. Mazzara, V. Rivera, & A. Sadovykh (Eds.), *Microservices* (pp. 147–179). Springer International Publishing. https://doi.org/10.1007/978-3-030-31646-4_7

Richardson, C. (2018). *Microservices Patterns With examples in Java*.

Richardson, C. (2021). *Microservices Pattern: Decompose by subdomain*. Microservices.Io. http://microservices.io/patterns/decomposition/decompose-by-subdomain.html

Rosenberg, L. H., Stapko, R., & Gallo, A. (1999). Risk-based Object Oriented Testing. *In: Proceedings of the 24 Th Annual Software Engineering Workshop, NASA, Software Engineering Laboratory*.

Schmidt, D. C. (2006). *COVER FEATURE Model-Driven Engineering*.

Spinellis, D. (2019). *dspinellis/ckjm: Version 1.0*. Zenodo. https://doi.org/10.5281/zenodo.2529928

Spring Boot. (2021). https://spring.io/projects/spring-boot

Terzić, B., Dimitrieski, V., Kordić (Aleksić), S., Milosavljevic, G., & Luković, I. (2017, March 12). *MicroBuilder: A Model-Driven Tool for the Specification of REST Microservice Architectures*. Enterprise Information Systems.

Torchiano, M., Tomassetti, F., Ricca, F., Tiso, A., & Reggio, G. (2013). Relevance, benefits, and problems of software modelling and model driven techniques—A survey in the Italian industry. *Journal of Systems and Software*, *86*(8), 2110–2126. https://doi.org/10.1016/j.jss.2013.03.084

Traefik Labs: Makes Networking Boring. (2021). Traefik Labs: Makes Networking Boring. https://traefik.io/

Voelter, M. (2013). *Designing, Implementing and Using Domain-Specific Languages*. 558.

Whittle, J., Hutchinson, J., & Rouncefield, M. (2014). The State of Practice in Model-Driven Engineering. *IEEE Software*, *31*(3), 79–85. https://doi.org/10.1109/MS.2013.65

Xtext. (2021). https://www.eclipse.org/Xtext/

Xtext—MWE 2. (2021). https://www.eclipse.org/Xtext/documentation/306_mwe2.html

Appendixes

Appendix A. AHP analysis

Pairwise comparison

	DSL	MtM	CodeGen	DocCom
DSL	1	7	1	4
MtM	1/7	1	1/7	1/3
CodeGen	1	7	1	3
DocCom	1/4	3	1/3	1
Sum	2 2/5	18	2 1/2	8 1/3

Normalized pairwise comparison and relative priority

	DSL	MtM	CodeGen	DocCom
DSL	0,4179	0,3889	0,4038	0,4800
MtM	0,0597	0,0556	0,0577	0,0400
CodeGen	0,4179	0,3889	0,4038	0,3600
DocCom	0,1045	0,1667	0,1346	0,1200
Sum	1,0000	1,0000	1,0000	1,0000

	Priority vector
DSL	0,422661
MtM	0,053237
CodeGen	0,392661
DocCom	0,13144

Teste da consistência

	DSL	MtM	CodeGen	DocCom		
DSL	1	7	1	4	0,422661	1,7137
MtM	1/7	1	1/7	1/3	0,053237	0,2135
CodeGen	1	7	1	3	0,392661	1,5823
DocCom	1/4	3	1/3	1	0,13144	0,5277

STEP2	4,05465	
	4,010819	
	4,029691	
	4,014796	
STEP3	4,027489	
STEP4	Consistência	IC (lambdamax-n)/n-1 0,009163
STEP 5	CR=IC/IR	0,010181 consistent

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0.00	0.00	0.58	0.90	1.12	1.24	1.32	1.41	1.45	1.49	1.51	1.48	1.56	1.57	1.59

Alternatives' matrix

DSL	MSDK	EMF	MPS	CM•JH
MSDK	1	3	3	9
EMF	1/3	1	1	7
MPS	1/3	1	1	7
CM•JH	1/9	1/7	1/7	1
Soma	1 7/9	5 1/7	5 1/7	24

MtM	MSDK	EMF	MPS	CM•JH
MSDK	1	1/7	1/3	5
EMF	7	1	4	9
MPS	3	1/4	1	7
CM•JH	1/5	1/9	1/7	1
Soma	11 1/5	1 1/2	5 1/2	22

CodeGen	MSDK	EMF	MPS	CM•JH
MSDK	1	1/5	1/2	1/5
EMF	5	1	3	1/3
MPS	2	1/3	1	1/4
CM•JH	5	3	4	1
Soma	13	4 1/2	8 1/2	1 7/9

DocCom	MSDK	EMF	MPS	CM•JH
MSDK	1	1/5	1/3	3
EMF	5	1	3	7
MPS	3	1/3	1	5
CM•JH	1/3	1/7	1/5	1
Soma	9 1/3	1 2/3	4 1/2	16

Normalized alternatives' matrix

DSL	MSDK	EMF	MPS	CM•JH	Pesos
MSDK	0,5625	0,5833	0,5833	0,3750	0,526042
EMF	0,1875	0,1944	0,1944	0,2917	0,217014
MPS	0,1875	0,1944	0,1944	0,2917	0,217014
CM•JH	0,0625	0,0278	0,0278	0,0417	0,039931
					1

MtM	MSDK	EMF	MPS	CM•JH	
MSDK	0,0893	0,0950	0,0609	0,2273	0,118104
EMF	0,6250	0,6649	0,7304	0,4091	0,607358
MPS	0,2679	0,1662	0,1826	0,3182	0,233719
CM•JH	0,0179	0,0739	0,0261	0,0455	0,040819

CodeGen	MSDK	EMF	MPS	CM•JH	
MSDK	0,0769	0,0441	0,0588	0,1121	0,073003
EMF	0,3846	0,2206	0,3529	0,1869	0,286265
MPS	0,1538	0,0735	0,1176	0,1402	0,121302
CM•JH	0,3846	0,6618	0,4706	0,5607	0,519429

DocCom	MSDK	EMF	MPS	CM•JH	
MSDK	0,1071	0,1193	0,0735	0,1875	0,121873
EMF	0,5357	0,5966	0,6618	0,4375	0,557892
MPS	0,3214	0,1989	0,2206	0,3125	0,263345
CM•JH	0,0357	0,0852	0,0441	0,0625	0,05689

Teste da consistência

STEP1						STEP2	STEP3	STEP4	STEP5
1	3	3	9	0,526042	2,1875	4,158416			
1/3	1	1	7	0,217014	0,888889	4,096	4,091745	0,030582	0,03398
1/3	1	1	7	0,217014	0,888889	4,096			
1/9	1/7	1/7	1	0,039931	0,160384	4,016563			
1	3	1/2	1/5	0,073003	0,2948	4,038076			
5	1	3	1/3	0,286265	1,1883	4,15116	4,135065	0,045022	0,050024
2	1/3	1	1/4	0,121302	0,4926	4,060829			
5	3	4	1	0,519429	2,2285	4,290194			
1	1/5	1/3	3	0,121873	0,4919	4,0362			
5	1	3	7	0,557892	2,3555	4,222175	4,118466	0,039489	0,043876
3	1/3	1	5	0,263345	1,0994	4,174659			
1/3	1/7	1/5	1	0,05689	0,2299	4,040829			

STEP1						STEP2	STEP3	STEP4	STEP5
1	7	1/3	5	0,118104	0,4869	4,122411			
7	1	4	9	0,607358	2,7363	4,505302	4,266445	0,088815	0,098683
3	1/4	1	7	0,233719	1,0256	4,388202			
1/5	1/9	1/7	1	0,040819	0,1653	4,049864			

Matrix with global priorities

	DSL	MtM	CodeGen	DocCom				
MSDK	0,526042	0,118104	0,073003	0,121873		0,422661		0,27331
EMF	0,217014	0,607358	0,286265	0,557892	x	0,053237	=	0,309792
MPS	0,217014	0,233719	0,121302	0,263345		0,392661		0,186411
CM•JH	0,039931	0,040819	0,519429	0,05689		0,13144		0,230488

Appendix B. Acceleo generate.mtl

generate.mtl

```
[comment encoding = UTF-8 /]
[module generate('http://www.dei.isep.ipp.pt/microservicesDSL')]


[template public generateApplication(anApplication : Application)]
[comment @main/]
[for (boundedContext : BoundedContext | anApplication.boundedcontext)]
[   generateBoundedContext(anApplication, boundedContext)/]
[/for]
[/template]

[template public generateBoundedContext(anApplication : Application, boundedContext : BoundedContext)]
[for (microservice : Microservice | boundedContext.microservice)]
[   generateMicroservice(boundedContext, microservice)/]
[/for]
[/template]




[template public generateMicroservice(boundedContext : BoundedContext, microservice : Microservice)]
[generateMicroserviceMain(boundedContext, microservice)/]
[generateMicroserviceDomain(boundedContext, microservice)/]

[for (entity : Entity | microservice.entity)]
    [if(entity.mainEntity)]
    [generateMicroserviceRepository(boundedContext,microservice,entity)/]
    [generateMicroserviceService(boundedContext,microservice,entity)/]
    [generateMicroserviceController(boundedContext,microservice,entity)/]
    [/if]
[/for]
[generateMicroserviceProperties(boundedContext, microservice)/]
[generateMicroservicePom(boundedContext, microservice)/]
[/template]


[template public generateMicroserviceMain(boundedContext : BoundedContext, microservice : Microservice)]
[file (boundedContext.name+'\\src\\main\\java\\'+microservice.properties._package.replaceAll('\\.','\\\\')+'\\'+microservice.name.replaceAll(' ', '_').toLowerCase()
+'\\'+microservice.name.replaceAll(' ', '_').toLowerCase().toUpperFirst()+'Application.java', false, 'UTF-8')]

package [microservice.properties._package+'.'+microservice.name.replaceAll(' ', '_').toLowerCase() /];

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class [microservice.name.replaceAll(' ', '_').toLowerCase().toUpperFirst()/]Application {

    public static void main(String['['/']'/]  args) {
        SpringApplication.run([microservice.name.replaceAll(' ', '_').toLowerCase().toUpperFirst()/]Application.class, args);
    }
}
[/file]
[/template]


[template public generateMicroserviceDomain(boundedContext : BoundedContext, microservice : Microservice)]
[for (entity : Entity | microservice.entity)]
    [generateMicroserviceEntity(boundedContext,microservice,entity)/]
[/for]
[for (enum : Enum | microservice.enums)]
    [generateMicroserviceEnum(boundedContext,microservice,enum)/]
[/for]
[/template]

```

```
[template public generateMicroserviceEntity(boundedContext : BoundedContext, microservice : Microservice,entity: Entity)]
[file (boundedContext.name+'\\src\\main\\java\\'+microservice.properties._package.replaceAll('\\.','\\\\')+'\\'+microservice.name.replaceAll(' ', '_').toLowerCase()
+'\\'+'models\\'+entity.name.replaceAll(' ', '_').toUpperFirst()+'.java', false, 'UTF-8')]
package [microservice.properties._package+'.'+microservice.name.replaceAll(' ', '_').toLowerCase()+'.models' /];

[if(entity.attributes -> exists(type='Date'))]
import java.util.Date;
[/if]
[if (entity.mainEntity)]
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
[else]
import javax.persistence.Embeddable;

@Embeddable
[/if]
public class [entity.name.replaceAll(' ', '_').toUpperFirst()/] {

    //variables
[if (entity.mainEntity)]
    @Id
    @GeneratedValue//(generator="system-uuid")
    private Long id;

[/if]
[for(field : Field | entity.attributes)]
    [let name: String = field.name ]
    [let type: String = field.type ]
    private [type.toString()/] [name.toString()/];
    [/let]
    [/let]
[/for]

    //constructors
    public [entity.name.toUpperFirst()/](){

    }

    public [entity.name.toUpperFirst()/](
    [for( field : Field | entity.attributes)]
        [field.type/] [field.name/][if (not (field = entity.attributes -> last()))],[/if]
    [/for]

    ) {
    [for( field : Field | entity.attributes)]
        this.[field.name/] = [field.name/];
    [/for]
    }


    //gets and sets
[if (entity.mainEntity)]
    public Long getId() {
        return id;
    }

[/if]
[for(field : Field | entity.attributes)]
    [let name: String = field.name ]
    [let type: String = field.type ]
    public [type.toString()/] get[name.toString().toUpperFirst()/]() {
        return [name.toString()/];
    }

    public void set[name.toString().toUpperFirst()/]([type.toString()/]  [name.toString()/]) {
        this.[name.toString()/] = [name.toString()/];
    }
    [/let]
    [/let]
[/for]

}
[/file]
[/template]



[template public generateMicroserviceEnum(boundedContext : BoundedContext, microservice : Microservice,enum: Enum)]
[file (boundedContext.name+'\\src\\main\\java\\'+microservice.properties._package.replaceAll('\\.','\\\\')+'\\'+microservice.name.replaceAll(' ', '_').toLowerCase()
+'\\'+'models\\'+enum.name.replaceAll(' ', '_').toUpperFirst()+'.java', false, 'UTF-8')]
package [microservice.properties._package+'.'+microservice.name.replaceAll(' ', '_').toLowerCase()+'.models' /];

public enum [enum.name.toUpperFirst()/] {
    [for(value : String | enum.values)]
    [value.toUpper()/][if (not (value = enum.values -> last()))],[/if]
    [/for]
}

[/file]
[/template]

[template public generateMicroserviceRepository(boundedContext : BoundedContext, microservice : Microservice,entity: Entity)]
[file (boundedContext.name+'\\src\\main\\java\\'+microservice.properties._package.replaceAll('\\.','\\\\')+'\\'+microservice.name.replaceAll(' ', '_').toLowerCase()
+'\\'+'repositories\\'+entity.name.replaceAll(' ', '_')+'Repository.java', false, 'UTF-8')]
package [microservice.properties._package+'.'+microservice.name.replaceAll(' ', '_').toLowerCase()+'.repositories' /];

[let className : String = entity.name.replaceAll(' ', '_').toUpperFirst()]
import [microservice.properties._package+'.'+microservice.name.replaceAll(' ', '_').toLowerCase()+'.models' /].[className/];
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface [className/]Repository extends JpaRepository<[className/],Long> {

}
[/let]
[/file]
[/template]
```

```
[template public generateMicroserviceService(boundedContext : BoundedContext, microservice : Microservice,entity: Entity)]
[file (boundedContext.name+'\\src\\main\\java\\'+microservice.properties._package.replaceAll('\\.','\\\\')+'\\'+microservice.name.replaceAll(' ', '_').toLowerCase()
+'\\'+'services\\'+entity.name.replaceAll(' ', '_')+'Service.java', false, 'UTF-8')]
package [microservice.properties._package+'.'+microservice.name.replaceAll(' ', '_').toLowerCase()+'.services' /];

[let className : String = entity.name.replaceAll(' ', '_').toUpperFirst()]
[let entityName : String = className.toLowerFirst()]
import [microservice.properties._package+'.'+microservice.name.replaceAll(' ', '_').toLowerCase()+'.models' /].[className/];
import [microservice.properties._package+'.'+microservice.name.replaceAll(' ', '_').toLowerCase()+'.repositories' /].[className/]Repository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;
import java.util.UUID;

@Service
public class [className/]Service {

    private [className/]Repository [entityName/]Repo;

    @Autowired
    public [className/]Service([className/]Repository [entityName/]Repo) {
        this.[entityName/]Repo = [entityName/]Repo;
    }

    public [className/] save[className/]([className/] [entityName/]){
        return [entityName/]Repo.save([entityName/]);
    }

    public List<[className/]> getAll[className/](){
        return [entityName/]Repo.findAll();
    }

    public Optional<[className/]> get[className/]ById(Long id){
        return [entityName/]Repo.findById(id);
    }

    public void delete[className/](Long id){
        if(id!=null){
            if(get[className/]ById(id).isPresent())
                [entityName/]Repo.deleteById(id);
        }
    }

    public [className/] update[className/]([className/] [entityName/]){
        return [entityName/]Repo.save([entityName/]);
    }
}
[/let]
[/let]
[/file]
[/template]
```

```
[template public generateMicroserviceController(boundedContext : BoundedContext, microservice : Microservice,entity: Entity)]
[file (boundedContext.name+'\\src\\main\\java\\'+microservice.properties._package.replaceAll('\\.','\\\\')+'\\'+microservice.name.replaceAll(' ', '_').toLowerCase()
+'\\'+'controllers\\'+entity.name.replaceAll(' ', '_')+'Controller.java', false, 'UTF-8')]
package [microservice.properties._package+'.'+microservice.name.replaceAll(' ', '_').toLowerCase()+'.controllers' /];

[let className : String = entity.name.replaceAll(' ', '_').toUpperFirst()]
[let entityName : String = className.toLowerFirst()]
import [microservice.properties._package+'.'+microservice.name.replaceAll(' ', '_').toLowerCase()+'.models' /].[className/];
import [microservice.properties._package+'.'+microservice.name.replaceAll(' ', '_').toLowerCase()+'.services' /].[className/]Service;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.Optional;
import java.util.UUID;

@RequestMapping("/api/[entityName/]")
@RestController
public class [className/]Controller {

    private final [className/]Service [entityName/]Service;

    @Autowired
    public [className/]Controller([className/]Service [entityName/]Service) {
        this.[entityName/]Service = [entityName/]Service;
    }

    @PostMapping
    public [className/] add[className/](@RequestBody [className/] [entityName/]){
        return [entityName/]Service.save[className/]([entityName/]);
    }

    @GetMapping
    public List<[className/]> getAll[className/](){
        return [entityName/]Service.getAll[className/]();
    }

    @GetMapping(path = "{id}")
    public Optional<[className/]> get[className/]ById(@PathVariable("id") Long id){
        return [entityName/]Service.get[className/]ById(id);
    }

    @PutMapping
    public [className/] update[className/](@RequestBody [className/] [entityName/]){
        return [entityName/]Service.update[className/]([entityName/]);
    }

    @DeleteMapping(path = "{id}")
    public void delete[className/]ById(@PathVariable("id") Long id){
        [entityName/]Service.delete[className/](id);
    }
}

[/let]
[/let]
[/file]
[/template]
```

```
[template public generateMicroservicePom(boundedContext : BoundedContext, microservice : Microservice)]
[file (boundedContext.name+'\\pom.xml', false, 'UTF-8')]
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.4.4</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>[microservice.properties._package/]</groupId>
    <artifactId>[boundedContext.name.replaceAll(' ', '_')/]</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>[boundedContext.name.replaceAll(' ', '_')/]</name>
    <description>[boundedContext.name.replaceAll(' ', '_')/] microservice</description>
    <properties>
        <java.version>11</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-jersey</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>

        [if not(microservice.dependencies.oclIsUndefined())]
        [for(dependencies : Dependencies | microservice.dependencies)]
        [for(dependency : Dependency | dependencies.dependencies)]
        <dependency>
            <groupId>[dependency.groupId/]</groupId>
            <artifactId>[dependency.groupId/]</artifactId>
            <scope>[dependency.scope/]</scope>
        </dependency>
        [/for]
        [/for]
        [/if]
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
[/file]
[/template]
```

Appendix C. Case Study's JDL

```
1
2   /* Bounded Context PropertiesContext */
3
4   entity Property {
5       title String
6       pricePerNight Float
7       propertyOwnerId String
8   }
9
10
11  entity Address {
12      country String
13      city String
14      postalCode String
15      address String
16  }
17
18  microservice Property, Address with PropertiesContext
19
20  application {
21      config {
22          baseName PropertiesContext,
23          packageName org.contextmapper.generated.propertiescontext,
24          applicationType microservice
25          serverPort 8081
26          enableSwaggerCodegen true
27      }
28      entities Property, Address
29  }
30
31  /* Bounded Context BookingsContext */
32
33  entity Booking {
34      bookerId String
35      propertyId Long
36      startDate LocalDate
37      endDate LocalDate
38  }
39
40  microservice Booking with BookingsContext
41
42  application {
43      config {
44          baseName BookingsContext,
45          packageName org.contextmapper.generated.bookingscontext,
46          applicationType microservice
47          serverPort 8082
48          enableSwaggerCodegen true
49      }
50      entities Booking
51  }
52
```

```
52
53  /* Bounded Context PaymentContext */
54
55  entity Payment {
56      paymentUserId String
57      bookingId Long
58  }
59
60  microservice Payment with PaymentContext
61
62  application {
63      config {
64          baseName PaymentContext,
65          packageName org.contextmapper.generated.paymentcontext,
66          applicationType microservice
67          serverPort 8083
68          enableSwaggerCodegen true
69      }
70      entities Payment
71  }
72
73  /* Bounded Context ReviewContext */
74
75  entity Review {
76      reviewerId String
77      bookingId Long
78      title String
79      comment String
80  }
81
82  microservice Review with ReviewContext
83
84  application {
85      config {
86          baseName ReviewContext,
87          packageName org.contextmapper.generated.reviewcontext,
88          applicationType microservice
89          serverPort 8084
90          enableSwaggerCodegen true
91      }
92      entities Review
93  }
94
95  /* relationships */
96  relationship OneToOne {
97              Property{location} to Address
98  }
99
100 /* microservice gateway app */
101 application {
102     config {
103         baseName gateway,
104         packageName org.contextmapper.generated.gateway,
105         applicationType gateway
106         serverPort 8080
107     }
108     entities Property, Address, Booking, Payment, Review
109 }
110
111 /* additional options */
112 dto * with mapstruct
113 service * with serviceImpl
114
```