



Full Stack Application Generation for Insurance Sales based on Product Models

CARLOS FILIPE DA SILVA LIMA

Outubro de 2016

Full Stack Application Generation for Insurance Sales based on Product Models

Carlos Filipe da Silva Lima

Thesis to obtain the Master of Science Degree in
**Computer Science - Information and Knowledge
Systems**

Supervisor: Prof. Dr. Paulo Gandra de Sousa

October 2016

Acknowledgments

I am grateful to my mentor, Dr. Paulo Sousa for his expertise, sincere and valuable guidance during the overall project.

Additionally, I would like to thank my parents, my sister and my girlfriend for their unceasing encouragement and support for completion of this project.

Finally, I also place on record, my sense of gratitude to one and all who, directly or indirectly, have lent their helping hand in this venture. Especially to Eng. Gabriel Santos, Eng. Mario Sousa, and Marketeer Cátia Baião for reviewing this document, and Eng. Nuno Alves for helping me in a brainstorming session to get a solution approach for discovering the root object of a domain model.

Abstract

The insurance market is segregated in various lines-of-business such as Life, Health, Property & Casualty, among others. This segregation allows product engineers to focus on the rules and details of a specific insurance area. However, having different conceptual models leads to an additional complexity when a generic presentation layer application has to be continuously adapted to work with these distinct models.

With the objective to streamline these continuous adaptations in an existent presentation layer, this work investigates and proposes the usage of code generators to allow a complete application generation, able to communicate with the given insurance product model. Therefore, this work compares and combines different code generation tools to accomplish the desired application generation.

During this project, it is chosen an existing framework to create several software layers and respective components such as necessary classes to represent the *Domain Model*; database mappings; *Service* layer; *REST* Application Program Interface (API); and a rich javascript-based presentation layer.

As a conclusion, this project demonstrates that the proposed tool can generate the application already adapted and able to communicate with the provided conceptual model. Proving that this autonomous process is faster than the current manual development processes to adapt a presentation layer to an *Insurance* product model.

Keywords

Insurance, Line-of-Business, Different conceptual models, Adaptation, Code Generation.

Resumo

O mercado segurador encontra-se dividido em várias linhas-de-negócio (e.g. Vida, Saúde, Propriedade) que têm naturalmente, diferentes modelos conceptuais para a representação dos seus produtos. Esta panoplia de modelos leva a uma dificuldade acrescida quando o software de camada de apresentação tem que ser constantemente adaptado aos novos modelos bem como às alterações efetuadas aos modelos existentes.

Com o intuito de suprimir esta constante adaptação a novos modelos, este trabalho visa a exploração e implementação de geradores de código de forma a permitir gerar toda uma aplicação que servirá de camada de apresentação ao utilizador para um dado modelo.

Assim, este trabalho expõe e compara várias ferramentas de geração de código actualmente disponíveis, de forma a que seja escolhida a mais eficaz para responder aos objectivos estabelecidos. É então seleccionada a ferramenta mais promissora e capaz de gerar vários componentes de software, gerando o seu modelo de domínio, mapeamento com as respectivas tabelas de base de dados, uma camada de lógica de negócio, serviços *REST* bem como uma camada de apresentação.

Como conclusão, este trabalho apresenta uma solução que é capaz de se basear num modelo proveniente do sistema de modelação de produto e assim gerar completamente a aplicação de camada de apresentação desejada para esse mesmo modelo. Permitindo assim, um processo mais rápido e eficaz quando comparado com os processos manuais de desenvolvimento e de adaptação de código-fonte existentes.

Palavras Chave

Seguros, Linhas-de-negócio, Diferentes modelos conceptuais, Adaptação, Geradores de código.

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 1 |
| 1.1 | Problem Statement | 4 |
| 1.2 | Project Objectives | 5 |
| 1.3 | Project Constraints | 6 |
| 1.4 | Hypothesis Test | 6 |
| 1.5 | Contribution to Knowledge | 7 |
| 1.6 | Document Organization | 8 |
| 2 | Value proposition | 9 |
| 2.1 | Perceived Value - A Theoretical Introduction | 11 |
| 2.2 | Application Generator - The Perceived Value | 12 |
| 2.3 | Obstacles in Insurance Sector | 13 |
| 2.3.1 | Dynamic Tools | 13 |
| 2.3.2 | Time-to-Market: Launching of Commercial Offers | 14 |
| 2.4 | Business Opportunities | 15 |
| 2.4.1 | Business Opportunities - Sales & Service | 16 |
| 2.4.2 | Business Opportunities - A Dynamic Generated Application | 17 |
| 2.5 | Business Model | 17 |
| 3 | Context | 19 |
| 3.1 | Insurance Industry | 21 |
| 3.1.1 | Property Insurance | 22 |
| 3.1.2 | Life Insurance | 22 |
| 3.2 | Product Machine | 22 |
| 3.2.1 | Modeling Workbench | 24 |
| 3.2.2 | Testing Workbench | 28 |
| 3.2.3 | Data Workbench | 28 |
| 3.2.4 | Reporting Workbench | 29 |
| 3.3 | Sales & Service | 29 |

| | | |
|----------|---|-----------|
| 3.4 | PM and S&S - Communication | 31 |
| 4 | Code Generation | 33 |
| 4.1 | Introduction | 35 |
| 4.1.1 | Passive Code Generators | 35 |
| 4.1.2 | Active Code Generators | 36 |
| 4.2 | Code Generation Approaches | 36 |
| 4.2.1 | Template-based Code Generation | 36 |
| 4.2.2 | Code Generation based on regular expression substitutions | 37 |
| 4.2.3 | Code Generation in a Model-Driven Environment | 37 |
| 4.3 | Related work | 38 |
| 4.4 | Conclusions | 39 |
| 5 | Application Code Generators | 41 |
| 5.1 | Application Code Generators - Frameworks | 43 |
| 5.1.1 | AndroMDA | 43 |
| 5.1.2 | Celerio | 44 |
| 5.1.3 | JHipster | 45 |
| 5.1.4 | ModelJ | 49 |
| 5.1.5 | Sculptor Framework | 49 |
| 5.2 | Frameworks Comparison | 51 |
| 5.2.1 | Desired Features | 51 |
| 5.2.1.A | Miscellaneous Characteristics | 51 |
| 5.2.1.B | Code Generation Capabilities | 52 |
| 5.2.2 | Advantages and Disadvantages | 52 |
| 5.3 | Conclusions | 54 |
| 6 | Proposed solution | 55 |
| 6.1 | Overview | 57 |
| 6.2 | Conception | 57 |
| 6.2.1 | Choosing the PM Model | 57 |
| 6.2.2 | Interpreting the PM Domain Model | 58 |
| 6.2.3 | Generating the JHipster Domain Model | 59 |
| 6.3 | Development | 60 |
| 6.3.1 | Choosing the PM Model | 60 |
| 6.3.2 | PM Model Interpretation | 62 |
| 6.3.3 | Generate Application Skeleton | 68 |
| 6.3.4 | Generating the Domain Model | 71 |

| | | |
|----------|--|-----------|
| 6.3.5 | Database | 73 |
| 6.3.6 | Application generation and initialization | 73 |
| 6.4 | Validation | 75 |
| 6.4.1 | Generated Entities | 75 |
| 6.4.1.A | P&C Model | 75 |
| 6.4.1.B | Life Model | 76 |
| 6.4.2 | Generated REST Services | 77 |
| 6.4.3 | Generated Database Mappings | 78 |
| 6.4.4 | Generated Database Tables | 80 |
| 6.4.5 | Generated Client-Side | 81 |
| 6.4.5.A | Generated Functionalities | 82 |
| 6.4.5.B | Interacting with application domain model | 83 |
| 6.5 | Conclusions | 84 |
| 7 | JHipster - Custom Extensions | 87 |
| 7.1 | JHipster Modules | 90 |
| 7.2 | Inject PM Model Dependency | 90 |
| 7.3 | Applying BOAdaptable to the Generated Entities | 92 |
| 7.4 | Validation | 94 |
| 7.5 | Conclusions | 94 |
| 8 | Experiments | 95 |
| 8.1 | Generation of Multiple Software Layers | 97 |
| 8.2 | Generation Time | 98 |
| 9 | Conclusions | 99 |
| 9.1 | Code Generation vs Current Manual Processes | 101 |
| 9.1.1 | Current Manual Processes | 101 |
| 9.1.1.A | Adapting: New Attribute | 101 |
| 9.1.1.B | Adapting: New Entity | 101 |
| 9.1.1.C | Adapting: New Domain Model | 102 |
| 9.1.2 | Adopting Code Generation | 102 |
| 9.1.2.A | Advantages | 102 |
| 9.1.2.B | Disadvantages | 103 |
| 9.2 | Switching to Code Generation in the Organization | 103 |
| 9.3 | Objectives Assessment | 104 |
| 9.4 | Limitations and Future Work | 105 |

| | |
|--|------------|
| A From PM to S&S Overview | 113 |
| B CANVAS : Business Model | 117 |
| C Related Technologies | 121 |
| C.1 DDD - Domain Driven Design | 123 |
| C.1.1 Domain | 123 |
| C.1.2 Model | 124 |
| C.1.3 Building Blocks | 124 |
| C.1.4 Model Driven Design - Sterotypes | 126 |
| C.1.4.A Entities | 126 |
| C.1.4.B Value Objects | 126 |
| C.1.4.C Services | 127 |
| C.1.4.D Modules | 127 |
| C.1.4.E Aggregates | 127 |
| C.1.4.F Factories | 128 |
| C.1.4.G Repositories | 128 |
| C.2 DSL - Domain Specific Languages | 129 |
| C.2.1 Why use a DSL? | 130 |
| C.2.2 Visualization | 131 |
| D Sculptor: Domain Driven Design sample | 133 |
| E Diagrams | 137 |

List of Figures

| | | |
|-----|--|----|
| 1.1 | Project objective overview | 5 |
| 2.1 | Selling to Customer and Selling Business | 12 |
| 2.2 | Selling Business | 12 |
| 3.1 | PM: Centralization of all Product Rules | 22 |
| 3.2 | Product Machine - Overview | 23 |
| 3.3 | Combining elements to create conceptual models | 26 |
| 3.4 | Unified Product Platform Ecosystem | 27 |
| 3.5 | Sales and Services - State of the Art | 30 |
| 4.1 | An example of design model (a class diagram) | 38 |
| 5.1 | AndroMDA: From UML Model to multiple cartridge generation | 43 |
| 5.2 | Celerio: Overview | 45 |
| 5.3 | JHipster: Generating the application from command-line | 46 |
| 5.4 | JHipster: Generating the application - An example of a domain model design | 48 |
| 5.5 | Example: Health App - Dashboard | 48 |
| 5.6 | Example: Health App - Actions log | 48 |
| 5.7 | Sculptor: Target Implementation | 50 |
| 6.1 | Sequence Diagram: PM model introspection | 63 |
| 6.2 | Sequence Diagram: Relationship analysis | 64 |
| 6.3 | Partial signatures of the BOAdaptableContractBO class: Relationship analysis | 65 |
| 6.4 | Class Diagram - Business Object | 67 |
| 6.5 | Activity Diagram - Building domain model hierarchy | 68 |
| 6.6 | Class Diagram - JHipster Entity | 71 |
| 6.7 | Code Generation Time | 74 |
| 6.8 | System Metrics | 82 |

| | | |
|------|---|-----|
| 6.9 | Application Health Check | 82 |
| 6.10 | System Metrics | 83 |
| 6.11 | Managing Domain Entities | 84 |
| 6.12 | Creating a Domain Entity | 84 |
| 7.1 | Overall Project Overview | 89 |
| A.1 | Unified Product Platform Ecosystem | 115 |
| B.1 | CANVAS Business Model | 119 |
| C.1 | Domain-Driven Design - Patterns and Relationships | 125 |
| D.1 | Sculptor: DDD Sample Model | 135 |
| E.1 | Sequence Diagram - Creating JDL model | 139 |

List of Tables

| | | |
|-----|---|----|
| 5.1 | Advantages and Disadvantages of: AndroMDA | 52 |
| 5.2 | Advantages and Disadvantages of: ModelJ | 53 |
| 5.3 | Advantages and Disadvantages of: Celerio | 53 |
| 5.4 | Advantages and Disadvantages of: JHipster | 53 |
| 5.5 | Advantages and Disadvantages of: Sculptor | 53 |
| 5.6 | Frameworks Comparison - Solution Required Features | 54 |
| 6.1 | P&C Model - Entity Comparison : PM Model vs Generated Entities | 75 |
| 6.2 | Life Model - Entity Comparison : PM Model vs Generated Entities | 77 |
| 6.3 | Generated REST Services : Resource for IllustrationBO Entity | 77 |
| 6.4 | Generated Database Tables | 81 |
| 8.1 | Number of Generated Files and Folders | 97 |
| 8.2 | Generated Source-code Lines | 97 |
| 8.3 | Solution Generation Times | 98 |

List of Algorithms

| | | |
|-----|---|----|
| 6.1 | Translating PM Entities into a structured data format | 58 |
| 6.2 | Getting Model Relationships | 65 |

Listings

| | | |
|------|--|-----|
| 5.1 | JHipster Domain Language - Syntax reference. | 47 |
| 5.2 | Example of a DSL to express REST WebServices | 50 |
| 6.1 | Maven - pom.xml file example | 61 |
| 6.2 | Maven - Exec Plugin | 61 |
| 6.3 | Non-ambiguous relationship | 66 |
| 6.4 | Ambiguous relationship | 66 |
| 6.5 | Configuration file: yo-rc.json | 70 |
| 6.6 | Velocity template for JDL format | 72 |
| 6.7 | Example of CoverageBO entity exposed in JDL format | 72 |
| 6.8 | Generate Entites batch | 73 |
| 6.9 | Generate Application batch | 74 |
| 6.10 | Database - tables initialization | 78 |
| 6.11 | Liquibase changeset - create QuoteBO | 79 |
| 6.12 | Database connection configuration | 80 |
| 6.13 | Application running log | 80 |
| 7.1 | pm-dependency template | 91 |
| 7.2 | pm-dependency template result | 91 |
| 7.3 | Hooking a JHipster module | 91 |
| 7.4 | JHipster hook configuration | 91 |
| 7.5 | Generate Application batch with pm dependency injector | 91 |
| 7.6 | PM Injector source-code | 92 |
| 7.7 | Additional fields | 93 |
| 8.1 | Bash command: count the generated application files | 97 |
| C.1 | Example: internal DSL | 129 |

Acronyms

| | |
|-------------|---|
| AADL | Architecture Analysis & Design Language |
| ACG | Automatic Code Generator |
| API | Application Program Interface |
| B2B | Business-to-Business |
| BA | Business Attribute |
| BI | Business Intelligence |
| BIRT | Business Intelligence and Reporting Tools |
| BO | Business Object |
| BOM | Business Object Model |
| BOP | Business Owner's Policy |
| COP | Component-Oriented Programming |
| CPU | Central Processing Unit |
| CR | Composition Rule |
| CRM | Customer Relationship Management |
| CRUD | Create, Read, Update, and Delete |
| CRV | Composition Rule Version/Variation |
| CSS | Cascading Style Sheets |
| DAO | Data Access Object |
| DB | Database |

| | |
|-------------|------------------------------------|
| DBMS | Database Management System |
| DDD | Domain Driven Design |
| DSL | Domain Specific Language |
| DSM | Domain-Specific Modeling |
| DTO | Data Transfer Object |
| DWB | Data Workbench |
| EJB | Enterprise JavaBeans |
| EMF | Eclipse Modeling Framework |
| ES | Expert System |
| ES6 | ECMAScript 6 |
| GCR | Group Constraint Rule |
| HTML | HyperText Markup Language |
| HTTP | HyperText Transfer Protocol |
| IoC | Inversion of Control |
| IP | Internet Protocol |
| IT | Information Technology |
| J2EE | Java 2 Platform Enterprise Edition |
| JAR | Java ARchive |
| JDK | Java Development Kit |
| JDL | JHipster Domain Language |
| JMS | Java Message Service |
| JPA | Java Persistence API |
| JS | JavaScript |
| JVM | Java Virtual Machine |
| JSF | JavaServer Faces |

| | |
|----------------|-------------------------------|
| JSON | JavaScript Object Notation |
| KBS | Knowledge-Based System |
| KPI | Key Performance Indicator |
| LOB | Line-of-Business |
| MDA | Model-Driven Architecture |
| MDE | Model-Driven Environment |
| MV* | Model View |
| MVC | Model View Controller |
| NPM | Node Package Manager |
| ORM | Object-Relational Mapping |
| PAS | Policy Administration System |
| PDS | Product Definition System |
| PM | Product Machine |
| MWB | Modeling Workbench |
| P&C | Property and Casualty |
| POJO | Plain Old Java Object |
| POM | Project Object Model |
| ROI | Return On Investment |
| RT | Real Time |
| RWB | Reporting Workbench |
| S&S | Sales & Services |
| SPA | Single Page Application |
| SOA | Service-Oriented Architecture |
| SQL | Structured Query Language |
| TTM | Time To Market |

| | |
|-------------|--|
| TWB | Testing Workbench |
| UDP | Unified Distribution Platform |
| UI | User Interface |
| UML | Unified Modeling Language |
| VI | Version Information |
| VP | Value Proposition |
| VPN | Virtual Private Network |
| WAR | Web application ARchive |
| XAML | Extensible Application Markup Language |
| XMI | XML Metadata Interchange |
| XML | Extensible Markup Language |

1

Introduction

Contents

| | | |
|-----|-------------------------------------|---|
| 1.1 | Problem Statement | 4 |
| 1.2 | Project Objectives | 5 |
| 1.3 | Project Constraints | 6 |
| 1.4 | Hypothesis Test | 6 |
| 1.5 | Contribution to Knowledge | 7 |
| 1.6 | Document Organization | 8 |

Insurance companies are responsible for providing protection for financial losses and implements sophisticated risk management rules to provide the right policies and charge reasonable premiums¹ to their policyholders. Insurance Line-of-Business (LOB) (such as Life, Health, Property & Casualty) are usually segregated to simplify the risk management and to allow product engineers focus on the particular relevant rules per line of business.

The typical process to launch an insurance offer is usually very long, especially in the case of new products. At some Insurers, launching a new product can take as long as a year and even changing a tariff of an existing product can be lengthy (RGA, 2014). Usually, most of this time is spent in the preparation phase where the Insurers need to adapt its Information Technology (IT) systems to handle newest or changed products. In this phase, it is often required the custom development of applications to sell this offer for every intended channel, which can have different technological constraints and business requirements. The custom development is often the only way to address this problem, especially if the Insurer desires the combination of existing products. This custom development is usually costly and leads to effort duplication for the maintenance of software portfolio.

Reducing the need for custom developments can also decrease the Time To Market (TTM) and the overall project costs, providing a competitive advantage as well an operative efficiency. Therefore, there is the need for better tools that can minimize the IT dependencies that currently marketing and sales face, to answer the market demands faster. Such tools need to take into account the diversity and particularities of each channel and device that the insurance offer needs to be distributed to, but also and most importantly they need to leverage existing product definitions without the need for double configuration or coding.

msg life is a European technological company, with 35 years of experience and is specialized in the insurance sector. msg life designs and develops solutions that impact different layers of the insurance business, from product design to distribution. With the centralization of business rules within a multi LOB platform that communicates with multiple systems, Product Machine (PM) is a back-office Product Definition System (PDS) that ensures the autonomy, flexibility, and traceability that Insurers need to create and target the rates according to the intended criteria such as client, distribution channel, and device. The reduction of risks provided by PM is complemented by Sales & Services (S&S), an integrated distribution platform which provides the reduction of time and costs as far as the release of new products into the market and its brokers are concerned. S&S focus on the customer and on the sales process where it packs itself on a multi-LOB, multichannel and multi-device that can make the insurance distributions more efficient and scalable (OjE - O Jornal Económico, 2015).

However, to interact with this product definition system, S&S needs to implement a set of *Java* interfaces to enable communications. These interfaces represent a conceptual model that was modeled by

¹ The amount of money that an individual or business must pay for an insurance policy.

a product engineer. In *Product Machine* a conceptual model is implementing an insurance line-of-business whereas it can hold several entities with various attributes attached to describe the domain. Therefore it is required that several interfaces get manually implemented in S&S side. This manual implementation process is a time-consuming task and currently stands as a bottleneck, especially when *Product Machine* releases new customer-specific models.

Consequently, this project aims the creation of presentation layers (front-end applications) for different PM models by using autonomous processes, such as code generation techniques, to obtain an efficient model adaptation and enable communications between these two system components (model definition and presentation layer) without requiring any manual code to adjust the adaptation layer.

1.1 Problem Statement

Sales & Service is a generic presentation layer for various PM models that is being used by different customers. Usually, when a customer works with the same line of business, for example, Property and Casualty (P&C), it is possible to share the same conceptual model (objects, attributes, relationships) i.e. the Business Object Model (BOM). Although customers can reuse an existing PM conceptual model, they can still define their products, formulas, and rules differently from another client. The dynamic interpretation of models, allows S&S to support new customers without much effort because the adapter layer for the model was already developed on S&S side.

However, when the reuse of conceptual models is not possible (e.g. due to specific customer data structure or rules), a new product definition is often designed and created by product engineers. Upon this scenario, S&S will also demand developments so that it can adapt and communicate with the newest PM model and its interfaces.

When a new PM model is designed and generated by the product engineer, Modeling Workbench (MWB), (thoroughly described in subsection 3.2.1) generates a Java ARchive (JAR) to be used by the downstream applications. This JAR, created in MWB, contains the product details, and it enables the model *Runtime Services*. But not being a standalone system that could provide these services under a Client-Server architecture approach, the downstream systems or applications interested in using the model, must implement all the Java *interfaces* within the JAR artifact generated automatically by MWB. With the correct implementation of all PM model interfaces, the downstream applications can then start interacting with the model *Runtime Services*.

The main problem with this process is that it is a time-consuming task and involves development related costs to msg life. Additionally, when a new customer acquires both PM and S&S, and his business demands the development of a new conceptual model that S&S does not support, the front-end team must focus not only on the front-end behavior but also building a back-end adapter layer.

Many times, PM modelers finish their work faster than S&S team. Consequently, project stakeholders

are often waiting for S&S developments completion, not only for validating what was done by the modeling team but also to assess if the overall project is on schedule. Often, stakeholders ask for estimations so that they can validate the modeled products in their screens using S&S, causing stress on the S&S team that needs to adapt the application to this new product design.

1.2 Project Objectives

The main goal of this project is the study and research of solutions able to streamline the sales process using the existing knowledge about the PDS system directly, without the need to develop the adapter layer on S&S. One possible path is exploring code generation techniques that use a Customer's Product model to generate the right artifacts without manual coding. This project scope and consequent proposed solution is limited to the main PDS used by msg life, i.e. the *FJA Product Machine V4* metamodel (PM). The proposed solution shall use a PM model as input and generate the presentation layer, dropping the need to add new adapters so that an end-user can see the modeled products in PM.

As illustrated in Figure 1.1, the solution shall receive different product models as input, where the output will be a generated presentation layer able to communicate with the *Runtime Services* provided by that input model. Each input model represents a distinct LOB's or even a particular Insurer model. Although a LOB represents a concrete business in real world, the domain model implemented to describe that business may differ from customer to customer.

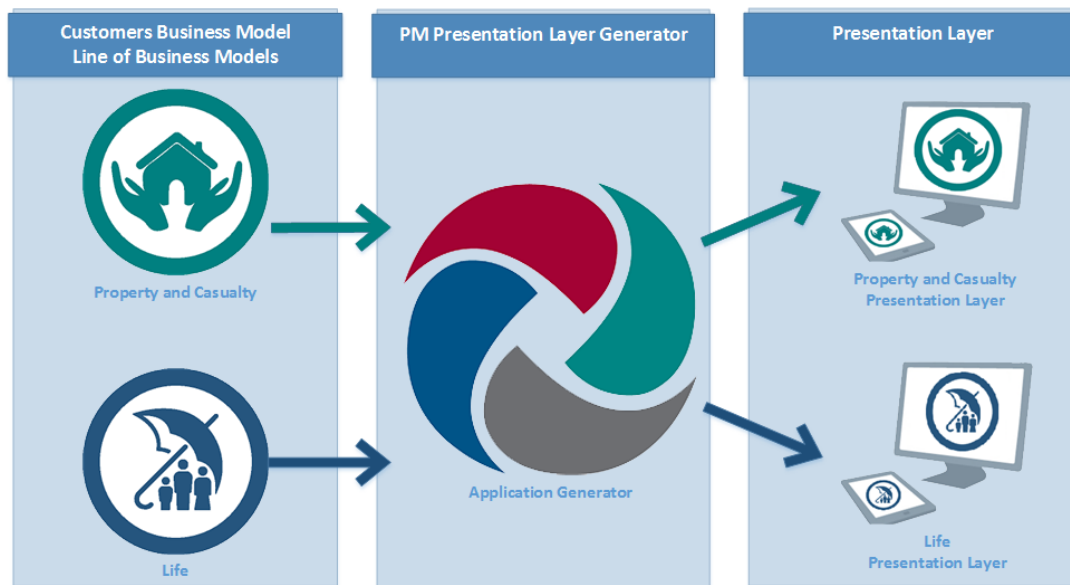


Figure 1.1: Project objective overview

The proposed solution shall generate a presentation layer that follows the modeled conceptual model provided in the PM model, and without any manual coding involved.

The identified sub-goals for this work are:

1. Study similar problems and other approaches to identify alternative solution paths;
2. Define a Domain Specific Language (DSL) or a mapping that enables the re-usage of a PM meta-model independently of customer specific models;
3. Develop code generators;
4. Develop a proof of concept that can be used with at least two different product models (e.g., PM4 Life model and PM4 P&C model);
5. Define general architecture of a channel application (e.g., mobile native app);

1.3 Project Constraints

Due to the existing technology stack used by the overall applications within msg life portfolio, it is expected that the solution follows the same ecosystem. Therefore, it must be an open-source Java-based solution and compatible with *Maven Repository* so that the artifacts can be accessible through Maven (internal msg life Nexus²), enabling Maven dependency management.

1.4 Hypothesis Test

”An *Experimental Design* is the laying out of a detailed experimental plan in advance of doing the experiment. Well chosen experimental designs maximize the amount of information that can be obtained for a given amount of experimental effort.” (NIST, 2016)

To test the efficiency and effectiveness of the adopted solution for this project a testing plan is necessary to support the final conclusions. Therefore, an *empirical analysis*³ using *statistical analysis* shall solidly validate or reject the project claims.

To perform an *empirical analysis* is necessary to identify what are the *claims* or *hypothesis*⁴ that will be relevant to be tested; what are the experience *conditions*; and enumerate what the *dependent* and *independent* variables are. The results will enable the evaluation of actual running time, and operation counts that will support the conclusions about the solution quality on the selected problem instances.

The *hypothesis* that will provide support for the observations are:

- (A) The proposed solution shall generate various application layers (Database mappings, Domain model and Rest API), without any manual coding.

² Sonatype Nexus - Is a Repository Manager - a dedicated server application designed to manage repositories of binary components

³ Method to gain knowledge using direct and indirect observation or experience.

⁴ Proposed explanation made on the basis of limited evidence as a starting point for further investigation. - (Oxford dictionary, 2016)

- (B) The proposed solution shall generate the entire application in an acceptable time. In which an acceptable time is considered to be up to 15 minutes.

To address the previous *hypothesis* will be obtained 20 random samples of trials. For each test, will be performed a new execution of the proposed solution. Where all of the random samples shall be under the same conditions (same machine configuration), featuring 20 valid observations. The retrieved results will serve for future comparisons with other possible solutions or partial changes, such as algorithm refactoring or rewriting.

To address the *hypothesis* B, from the execution samples retrieved, a computational study will be done. Where the Central Processing Unit (CPU) Time⁵ will be taken into account and verified if the generation tool is taken more than 15 minutes to accomplish the desired code generation.

Each PM model has several *interfaces* that in turn represent the domain model. Therefore the *hypothesis* A will be evaluated using a prior interface enumeration so that the results of the generated code get compared to the expected ones. This process will give a clear validation about how many interfaces shall the solution be implementing to enable communications for a given PM model. At this point, the solution accuracy will also be judged.

The solution reliability is another important measurement that must be included in the results, if tests prove that the solution is implementing all interfaces on every execution, then it can be concluded that the solution is reliable⁶.

1.5 Contribution to Knowledge

This research and its developments place a contribution in the *Computer Science* domain, particularly, in the autonomous code generation techniques under a Domain-Specific Modeling (DSM) environment.

The proposed solution for this project (See: chapter 6) contributes with a tool that uses a given input (*PM model*) and translates it into a different format that will serve to generate a complete web application able to run right after the code generation, without any manual coding or any further adjustments. Furthermore, the solution proves that it is possible to generate a complete application from its conception (domain model) to its production-ready deployment without any software developer interaction.

Sub-section 6.4 demonstrates that this project also contributes to msg life solutions portfolio, whereas msg life can use the resultant tool to generate new presentation layers for new PM models in about 5 minutes, and with the entire domain model, database mappings, *RESTful* services, among other infrastructure pieces of software generated. Providing a better time-to-market to get new models ready to ship to their customers.

⁵ The amount of time in which the CPU was busy executing code

⁶ Consistently good in quality or performance; able to be trusted.

Finally, the conclusions of chapter 9 show that the adopted approach also contributes with new ideas for the development processes in msg life, where the ad-hoc and manual development process per each new PM model may be dropped for a development over configuration to adapt the presentation layer quickly with autonomous processes.

1.6 Document Organization

This document is organized as follows: Chapter 2 describes the value and the CANVAS business model related to the development of this project, complemented by the value proposition provided by the existing S&S application to msg life customers.

Chapter 3 describes the project context, where the affected applications are described in detail, i.e. the current PDS system used by msg life to model products and its S&S platform.

Chapter 4 presents the State-of-the-Art about *Code Generation* techniques.

Chapter 5 is also describing state-of-the-art for existing tools capable of a partial or an entire application generation. This chapter performs a comparison between those tools and explains what are the tools that could be suitable as a possible solution path to address the project objectives.

Chapter 6 is the main section of this document. It describes the adopted solution and starts by giving an overview of the solution approach, the conception plan, and the development process. This chapter concludes with the validation of the proposed solution and presents conclusions.

Chapter 7 complements the previous chapter, where it is explained the custom generators that were created to extend the standard behavior of the chosen tool so that the final results are compliant with the project goals.

The validation about this project objectives resides in Chapter 8, here is addressed the identified *Hypothesis Tests* that has established the acceptance criteria for this project describing if the solution approach has successfully met the defined expectations.

Chapter 9 is the last chapter and holds the conclusions about this project, providing a critical thinking about the current manual coding versus the possible adoption of code generation techniques. Furthermore, this section provides a verification about the project objectives and demonstrates if they were completely satisfied. This conclusion section ends with the identification of most relevant future work.

2

Value proposition

Contents

| | | |
|-----|--|----|
| 2.1 | Perceived Value - A Theoretical Introduction | 11 |
| 2.2 | Application Generator - The Perceived Value | 12 |
| 2.3 | Obstacles in Insurance Sector | 13 |
| 2.4 | Business Opportunities | 15 |
| 2.5 | Business Model | 17 |

The Value Proposition (VP) is essential to any company because it describes buyers benefits from using the producer solution. It demonstrates that the target prospects are clear and describe what are they trying to accomplish. The value proposition is essential to the understanding of must-have solutions and creates compelling marketing messages. (techdata, 2016)

To understand the motivation to accomplish a solution for the described problem at Section 1.1, it is first necessary to understand what is the expected *perceived value* for the customer and what *perceived value* is. Some authors defend that “the creation of value is key to any business, and any business activity is about exchanging some tangible and/or intangible good or service and having its value accepted and rewarded by customers or clients, either inside the enterprise or collaborative network or outside“. (Nicola et al., 2012) ”A value proposition articulates the essence of business, defining what the organization fully intends to happen in the customer’s life.” (Barnes et al., 2009)

The value proposition is commonly used as a synonym for *benefits statement* or to describe the offering of a *product* or *service*. However, the *value* cannot be simply represented by benefits and offerings. It must take into account the costs to achieve those benefits. Therefore, the value must be a weighing of gains and costs. Otherwise, if the *value proposition* only focused on delivering benefits to every customer or prospect demand, would lead to bankruptcy. According to (Barnes et al., 2009), the overall product or service value can be driven by the following formula, where the overall value is the sum of all benefits minus its costs.

$$value \equiv \sum_{n=1}^n (Benefit_n - Cost_n) \quad (2.1)$$

Note that *cost* is not only about money but a much richer concept (that might be used in a variety of ways (Lindgreen and Wynstra, 2005)) that also includes other risks and exposures that are involved in any transaction. There are many types of *costs* such as: time, convenience, quality, customizations and of course monetary costs. It is advised that all the *costs* have to be weighed in the balance with any *benefits*. (Barnes et al., 2009)

2.1 Perceived Value - A Theoretical Introduction

According to (Barnes et al., 2009), ”value is specific to a particular instance, because of time, convenience perceived risks and so on are all factors that vary from organization to organization and from individual to individual. Value, like beauty, is in the eye (or mind) of the beholder”. Other authors argue that ”in most cases, value to the producer means something different from value to the user”. (Lindgreen and Wynstra, 2005)

Per (Zeithaml, 1988), ”*perceived value* has been defined as a customer’s overall assessment of the

utility of a product (or service) based on perceptions of what is received and what is given”.

”Value for the Customer (VC) is a personal perception of advantage arising out of a client’s association with an organization’s offering. Results on the combination of benefit and sacrifice; or an aggregation, over time, of any or all these.” (Woodall, 2003)

As described, *value* is vague and differently interpreted by each person. Therefore, the figure 2.1 shows what are the most common characteristics from a *Selling to Customer* and from a *Selling to Business* perspectives. From the consumer perspective, the usability of a product is usually more important (has more value to him) than the environmental impact (which is somewhat important to the producer since it may have economic impacts on their company through environmental taxes).

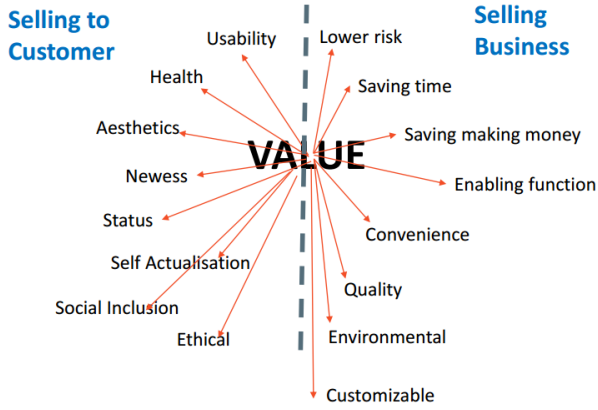


Figure 2.1: Selling to Customer and Selling Business

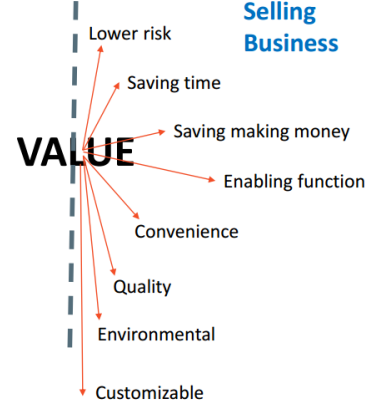


Figure 2.2: Selling Business

Source: Nicola (2016)

2.2 Application Generator - The Perceived Value

This project mainly focuses on the *Selling to Business* value characteristics as illustrated in figure 2.2, where the main perceived value to be achieved by the proposed solution is its efficiency regarding the integration with new PM models. Allowing the product modeling team to define customizations in PM and consequently have a presentation-layer that do not require any manual coding adaptation to work with the applied PM customizations.

Consequently, the solution shall use techniques that improve its *adaptability* (to adapt different PM models), *responsiveness* (it must be fast so that developers can enable its *usability* on their daily development process). The *reliability* of the proposed solution is also imperative, where the adopted techniques shall address the present problem and be precise (for example: given a PM model, all the necessary BOM interfaces must be implemented to enable communications). Another important characteristic to

be achieved with the proposed solution is the relation between time, effort and energy, where the optimal scenario is the reduction of these three costs.

Although it can be hard to achieve, the solution either a new tool, development technique, or a new work framework shall provide utility (easy to use), be more convenient when compared with the current manual development process. Furthermore, it shall also enable financial benefits for the company (e.g. selling for the same price what can be done with less development time – monetary benefits), delivering a better service quality, by answering to customer demands or prospects faster. Customer specific customizations are crucial to enterprise applications, whereas the possibility to extend and override the existing/base behavior is a plus. Therefore, the solution shall expect that it may/will be necessary to add customer customizations to the overall base solution.

2.3 Obstacles in Insurance Sector

When it is identified a gap in the market, it is also recognized a business opportunity. Therefore, this section will describe the obstacles identified by msg life in the insurance sector whereas the most significant gaps to solve are the need for dynamic tools and the poor time-to-market when launching new products into the market.

2.3.1 Dynamic Tools

According to msg life internal researches about the insurance market, the emergence of XXI century consumer's, more informed and demanding (Accenture, 2013a), has a strong impact in the insurance business. An industry that is still strongly oriented to product construction, which is sold through its distribution channels. This "Push" (Cognizant, 2013) logic is being challenged by latest consumer's generations that are increasingly pushing insurers to think that the future lies on the customer (customer-centricity) (Capgemini, 2013), providing them an excellent experience (EY, 2014) on all contacts established with the insurance company, regardless the channels (agencies, banks, Internet, call centers, social networks) or devices (PC, Tablet, Phone). (EY, 2013)

We have entered the era of "Pull Insurance" (Cognizant, 2013) in which informed consumers clearly identify what kind of protections they are interested in to be covered. Consequently, insurance companies are intended to identify within their existing product portfolio, which ones best fit each customer's needs. (IBM, 2013) (EY, 2014) There are a set of pre-conditions to be met to make this possible, Insurers have to design products that can function as "building blocks" (Accenture, 2013b) to compose integrated commercial offers. (IBM, 2013) For such, it is needed highly flexible tools (IBM, 2013), to let insurers setting components (e.g. insurance coverages) of all kinds and reuses them to build insurance products quickly and ensuring the consistency of business rules. Additionally, these tools need to provide services

that can be managed by external systems so that these new products can be used in a dynamic way without additional software developments.

As described in the previous paragraphs, more than ever, the *Sales* and *Marketing* departments require having tools to allow them to build in minutes or hours what once took weeks or months: new commercial offers. The creation of these new offers implies the definition of scenarios, comparisons and bundling of various products and business lines, in applications where the user experience plays the main role and where insurance products are just "building blocks", managed by the company users focused on customer needs. The respect for product rules and its consistency must always be present, but it is not the focus of a Marketing or Commercial department.

Nowadays the insurance products are usually configured, defined and modeled in the insurance companies core systems, whose main concern is the management of product policies. Therefore, products are defined focused on its management optimization and not on the selling process. For this reasons, insurers are facing significant differences between the existing technical and actuarial¹ products approved by regulatory authorities where the commercial perspective and the commercial offers are striving to meet a particular customer profile, channel or a specific client.

Consequently, the problem to solve is how to leverage the definitions of these products to streamline the creation of commercial offers faster and do it on a right tool for business users leveraging the creation of friendly solutions for such distinct lines of business as Life Insurance, Non-Life, and Health.

2.3.2 Time-to-Market: Launching of Commercial Offers

Currently, the process to launch new business offers is time-consuming, especially when launched new insurance products. The launching process of a new insurance product comprises the following steps:

- (1) Design and feasibility evaluation;
- (2) Approval;
- (3) Preparation;
- (4) Launch;
- (5) Sale;
- (6) Maintenance;
- (7) Termination.

In many Insurers, the launching process for new products can take over one year, and even a simple price change may take several months RGA (2014), constituting a significant problem, while it is clear that reducing the time-to-market will significantly constitute an enormous competitive advantage that can make the difference. Analyzing the common duration of the first five stages of this process, typically over 70% of the time is invested in the 3rd phase (preparation). In this phase, the Insurer has to define in detail the product to be sold, the processes that will be used for its management and adapt their

¹ An actuary is a professional dealing with the assessment and management of risk for financial investments, insurance policies, and any other ventures involving a measure of uncertainty. Investopedia (2016a)

information systems to manage this new product. In many cases, whenever is launched a new product, specific software programs are developed to support these new products on the various distribution channels. Even when is not about launching new products, but only to combine existing products into an innovative commercial offer, the problem is similar, i.e. it leads to specific developments.

The inefficient time to market launching new offers derives from the inadequate autonomy of marketing and commercial departments that don't have the right tools that could allow them to be agiler on the evaluation and utilization of the insurer's assets (insurance products and distribution channels). Another significant problem to solve is the lack of technological agility on insurance companies (Accenture, 2013b), which is, in part, responsible for the existing slowness in this industry due to its inability to address in useful time the market needs.

Greater the variety of channels and distribution devices, greater will be the challenge to the Insurers overcome the problem. From a technological point of view, Insurers are facing an accumulated reality over decades, in which different developed solutions with diverse technologies were being used to support the sale of their products.

Furthermore, the nonexistence of an integrated platform for insurance companies, capable of managing the overall sales processes in multiple distribution channels, increase the duplication of efforts on the integration between various sales software and the central management of the insurance business systems (Policy Administration System (PAS)) and adjacent systems, for example: Customer Relationship Management (CRM) and others.

Different and not integrated solutions mean that there are different management procedures for the different channels and products, with numerous system redundancies, organizational structures, tasks, and controls, resulting in service inefficiencies and significant costs to insurance companies.

By solving this IT dependency problem when launching new commercial offers, will also promote the empowerment of business users (Marketing and Sales), which address the issue about the time to market when launching new commercial offers.

2.4 Business Opportunities

The *business opportunities* dealt with a presentation-layer for a given PM model is described in subsection 2.4.1, where it describes the existing S&S business opportunities and its provided benefits.

Then, having in mind the described benefits that a PM presentation-layer delivers to msg Life and its customers, the subsection 2.4.2 describe the additional *business opportunities* and *benefits* of having a fully dynamic application generation that will serve as a PM presentation-layer.

2.4.1 Business Opportunities - Sales & Service²

Sales & Service (S&S), was built to enable faster and better insurance distribution with maximum effectiveness and cost reduction by empowering business users to create a customer-centric, cross-channel and multi-LOB sales approach. Its ability to integrate with several Product Engines (e.g., msgPM, FJA-US PM4, SymAss or others) and leverage existing product definitions enables fast adoption and Return On Investment (ROI).

S&S is a multi-channel, multi-Line-of-Business, Multi-device and Multi-tenant platform that boosts Speed to Market and eliminates redundancies, errors and delays, slashing costs by taking advantage of a common platform for marketing, product management, and field operations. S&S supports a multi-channel approach including agents, distribution partners, service centers, and customer self-service via the Internet in a single system.

S&S simplifies and reduces the time and costs involved with the required steps to bring products to market, enabling insurers to quickly respond to market opportunities, helping to enhance competitiveness and increase market share while improving operational and distribution efficiencies.

With S&S, insurers can model, develop, deploy, and distribute products faster than their competitors, therefore being more responsive to new market opportunities. They can accommodate customers wants and needs, new regulations and legislative trends, market fluctuations, demographic and social changes in their policyholder base, and knowledge they acquire from Business Intelligence (BI) and/or other data about policies, policyholders and the industry at large.

Furthermore, S&S makes the processes of creating, modifying, distributing and updating more collaborative across the entire enterprise. Business users, such as product managers, marketers, actuaries, underwriters, among other business specialists, can collaboratively customize S&S to the needs of each distribution channel to better serve the Salesforce and final customers.

With S&S, the *Salesforce* gets a powerful tool to engage in productive conversations with their customers, understanding their needs and suggesting the most appropriate protection alternatives, while maximizing the Insurance company share of the client wallet. Agents, Brokers, Bank Clerks and Telemarketing operators can find in S&S a customer-centric tool, with powerful pro-active sales features, enabling simple and engaging conversations focused on customer needs that makes the process of selling Insurance much faster and easier.

Seamless support of the entire lifecycle of an insurance policy, including illustration, quoting, new business and endorsements, as well as cancellations, reinstatements, and claims

²This section describes the existing S&S *Business Opportunities*. This information was acquired from msg life internal documentation and may be available in public resources such as company web-site (msg life, 2015) or prospectus.

management, are enabled through Straight-Through Processing features.

With S&S, Insurance companies can guarantee a central part of the Digital Agency mobile model is covered with a powerful tool to sell and serve their partners or customers.

(msg life, 2015)

2.4.2 Business Opportunities - A Dynamic Generated Application

Being a presentation-layer for several PM models, the proposed solution for this project inherits all the *business opportunities* and *benefits* from the existing presentation-layer for a given PM model, the Sales & Service Platform (S&S).

Having a dynamic and entirely generated application, able to integrate and communicate with a given PM model, allows the development team to focus on new functionalities instead of having to support, maintain and develop several *Adapter Layers*.

Beyond the improvement of the time-to-market launching a solution able to communicate with the PM model, msg life can reduce its development costs. Providing better pricing to their customers and therefore standing out from its competitors, and consequently acquiring market share.

2.5 Business Model

This section describes the Business Model applied to S&S using CANVAS. Due to its size, the complete CANVAS model is available at Appendix B.1

As far as the CANVAS front-state is concerned, the solution addressed to solve this project has msg life and its final customers as its *Customer Segments*. Since the solution is a revamp³ of the current S&S application using code generators techniques, it may change the way that developers currently develop S&S platform. Consequently, the final customers will also be affected since their requirements may be answered faster due to the new autonomous programming process implemented. S&S is designed to be a high-performance platform which enables Insurers to solve their time-to-market problems (See: 2.3.2) when launching new products. Another important issue (See: 2.4) that S&S solves is the empowerment of Marketing and Sales departments so that these can focus on their customers. Therefore, these are the S&S *Value Propositions*. Under a Business-to-Business (B2B) model, the *Customer Relationships* happen under personal presentations (such as *Demos* and *Webinar's*), *Sales Force* software tool and msg life Marketing Materials such as the *Inside Insurance Trends*⁴ magazine. S&S reach their customers through business visits, digital marketing, and phone calls. The *Revenue Streams* are composed of software licensing for new clients, development of customer-specific features (i.e. customizations) and the acquisition of new customers.

³ To change or arrange something again, in order to improve it. Cambridge-Dictionary (2016a)

⁴ Available at: <http://www.msg-life.com/pt/inside-insurance-trends/>

Regarding CANVAS backstage, the specialized software development in Insurance Business area is the *Key Activity* of the company as well as the maintenance of portfolio existing solutions. The *Key Resources* are the human resources (such as programmers, managers, marketing and sales people), hardware and software are the key tools for the business. The *Key Partners* are: *Dell*, being the msg life laptop supplier which replaces any damaged laptop if needed; *FJA-US* is the main direct partner. FJA is responsible for S&S Core roadmap⁵ and the producer of PM, the main PDS system that S&S use; *msg life System* is also a direct partner who manages all user accounts and enables the msg life company group communications all around the world with a secure Virtual Private Network (VPN) with *RSA* cryptosystem. The main *Costs Structure* are the same as the *Key Resources* which is normal since the company does not have any stock and the main costs are in fact the people and their equipment that enable programming and managing tasks, i.e. hardware and software.

⁵ "Roadmap is a plan of action for how a product or solution evolves over time. " - (Atlassian, 2016)

3

Context

Contents

| | | |
|-----|--------------------------------------|----|
| 3.1 | Insurance Industry | 21 |
| 3.2 | Product Machine | 22 |
| 3.3 | Sales & Service | 29 |
| 3.4 | PM and S&S - Communication | 31 |

This project is related to the *Insurance Industry*. Therefore the section 3.1 describes the main characteristics of this business. Then it is described two different application modules developed by msg life to enable efficiency in insurance companies. At section 3.2 is described the *Product Machine*, a product definition system. Followed by the section 3.3 that describes the *Sales & Service*, a presentation layer capable of exposing the modeled products in *Product Machine*. This chapter is concluded at section 3.4 which describes the communication between those two systems.

3.1 Insurance Industry

Insurance in simplest terms is about managing risk. For example, in life insurance, the company attempts to manage mortality (death) rates among its clients. The company collects premiums from policyholders, investing the money (usually in low-risk investments), and then reimburses this money once the person passes away, or the policy matures. (Investopedia, 2016b)

A person called *Actuary* constantly crunches demographic data to estimate the life of a person. Therefore, characteristics such as age, sex, smoker, affects the premium that a policyholder must pay. The greater the chance a person to have a shorter life span than the average, the higher the premium that person will have to pay. This process is virtually the same as every other type of insurance. These natures, also known as LOB include: General Liability Insurance, Property Insurance, Business Owner's Policy (BOP), Commercial Auto Insurance, Worker Compensation, Professional Liability Insurance, Directors and Officers Insurance, Data Breach, Homeowner Insurance, Renter Insurance, Life Insurance, Personal Automobile Insurance. (Investopedia, 2016b)

Insurance is the main alternative for businesses and individuals to reduce their financial impact upon risk occurrences. "Risk-transfer mechanism that ensures full or partial financial compensation for the loss or damage caused by events beyond the control of the insured party. Under an insurance contract, a party (the insurer) indemnifies the other party (the insured) against a specified amount of loss, occurring from specified eventualities within a specified period, provided a fee called premium is paid. In general insurance, compensation is typically proportionate to the loss incurred, whereas in life insurance usually a fixed sum is paid". (Merkin and Steele, 2013)

Some types of insurance (such as product liability insurance) are an essential component of risk management and are mandatory in several countries. Insurance, however, provides protection only against tangible losses. It cannot ensure continuity of business, market share, or customer confidence, and cannot provide knowledge, skills, or resources to resume the operations after a disaster. (Business Dictionary, 2016)

During this project will be used two different PM models. One product model that holds data about P&C insurances and another one containing modeled data about Life insurance. For this reason, the following section will explain these LOB's concepts within insurance business.

3.1.1 Property Insurance

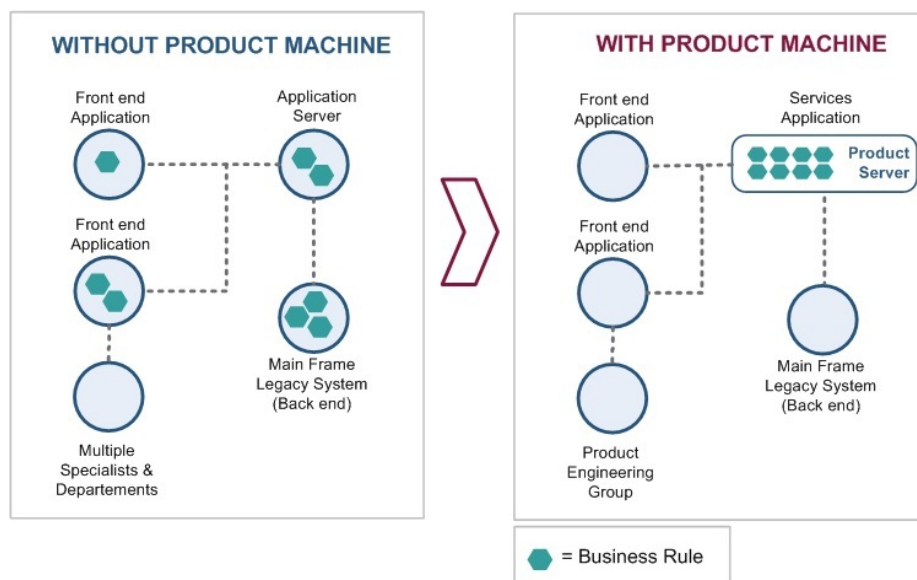
Property insurance is applied to whom own a building or have business personal property, including office equipment, computers, inventory or tools. Purchasing a property insurance policy will protect the insured person in case of fire, vandalism, theft, smoke damage, among other reasons. These owners may also want to consider business interruption/loss of earning insurance as part of the policy to protect the earnings if the business is unable to operate. (Forbes, 2016)

3.1.2 Life Insurance

Life insurance protects an individual against death. If a person holds a life insurance, the insurer pays a certain amount of money to a beneficiary upon his death. The insured person is asked to pay a premium in exchange for the payment of these benefits to the beneficiary. This type of insurance is important because it grants peace of mind. Having a life insurance allows people to know that the family of the policyholder will not be burdened financially upon his death. (Forbes, 2016)

3.2 Product Machine

Product Machine is a software solution designed and developed by FJA-US, a company of the group msg life. PM allows insurance companies to design, model, test and bring their products to the market more efficiently. This tool has multiple services that can be connected to insurance companies legacy systems, unifying them into a unified product platform.

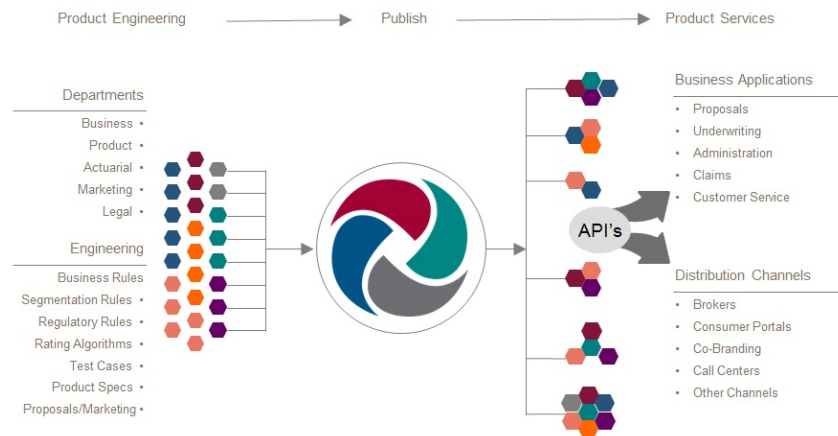


Source: FJA-US (2015) "Product Machine", internal company documentation, July 2014.

Figure 3.1: PM: Centralization of all Product Rules

Figure 3.1 demonstrates the PM centralization capabilities to hold all the details, rules, and formulas about a product. Consequently, turning off all the other distributed rules among insurers applications and making those rules stored in a single location where PM acts as the unique possible source of truth. This centralization allows the reuse of product rules on all downstream applications regardless its channel and device. It also reduces the maintenance time since all the details about the product are in this single location, making easier the traceability of existing errors on the product. (FJA-US, 2015)

Product Machine is a Java-based software. Developed to be employed specifically by product managers, marketers, actuaries, underwriters and other business people. PM transforms product modeling into high-performing runtime services and can be used in a Service-Oriented Architecture (SOA) as well as in a message-oriented middleware using Java Message Service (JMS). Therefore, PM has the necessary API that any business application or distributed channel needs to communicate and get all the required data about the modeled products. Figure 3.2 shows the representation of the product lifecycle, from engineering to its publication where it ends as being product services available to all parties that need to consume product data. (FJA-US, 2015)



Source: FJA-US (2015) "Product Machine", internal company documentation, July 2014.

Figure 3.2: Product Machine - Overview

As illustrated in Appendix A.1, PM is composed of four different modules, MWB, Testing Workbench (TWB), Data Workbench (DWB), and Reporting Workbench (RWB). Each module has its responsibility within the System. Therefore, the next section (3.2.1) describe MWB, being the main PM module this section describes MWB thoroughly, since it provides the ability to design different conceptual models by describing the model rules, product variations, and ends with the generation of all the necessary code to hold and provide the *Product Services* to downstream systems such as S&S.

Then, the following sections, 3.2.2, 3.2.3, and 3.2.4 hold a shallow explanation about the remaining modules: TWB, DWB, and RWB giving a brief description of their responsibilities within the PM as a unified system.

3.2.1 Modeling Workbench

MWB is the *Product Engineer's* main working area. This *Module* allows the modeling and representation of product details. MWB creates an active catalog of products, coverages/benefits, rules and calculations for various situations.

Composed by generic concepts, MWB can build new conceptual models to support new product definitions, delivering flexibility to insurers so that they can produce and configure their products regardless the LOB. Such flexibility is accomplished due to generic concepts that can hold any conceptual model. These concepts are:

- **Business Object (BO), Business Attribute (BA), and Relationship** are components to define the business object structure that in turn represent the structure of the conceptual model.
- **Composition Rule (CR)** enables the product variation and enhances re-usability. It is a rule that joins the parent object to its child object(s). Every Composition Rule (CR) requires at least one *Composition Rule Version/Variation*.
- **Composition Rule Version/Variation (CRV)** manages variations for a CR. If a CRV is activated, then the CR is available. In a CRV it is possible to define the minimum and maximum cardinality restrictions on the parent-child relationship as well as the required, and changeability indicators.
- **Version Information (VI)** is attached to a CRV and defines the effective date, product, and additional availability rules, *Mass Maintenance* Types and reason codes.
- **Group Constraint Rule (GCR)** creates the relationship between a *Domain Object* that shares the same *Domain Object Type*. A GCR can define if a *Domain Object* within a group can be selected without restrictions or if only one element of the group is eligible to be selected (GCR Operation = {ANY or EXACTLYONE}). When the group operation is modeled as EXACTLYONE the other components within the group cannot be selected. If the presentation layer for this data would be HTML based, then it could be easily represented by the following Front-End layout:

| | | |
|-------------------------|---|--------------------------|
| 1. Operation=ANY | ⇒ | <input type="checkbox"/> |
| 2. Operation=EXACTLYONE | ⇒ | <input type="radio"/> |

If the Front-End adds a new object into the group constraint, PM automatically updates the GCR parent object that holds the group.

- **Domain** defines default values as well as attributes and associated possible valid values of components. These valid values are defined in *Continuous Entries* or *Filters* that are attached to the *Domain*. A *Domain* hangs off of CRV's for various CR's.

- **Functional State** corresponds to the *Domains* and describe the behavior of input fields. Although *Domains* are defining attributes default values, and its associated possible values, the functional state (and its associated *Functional State Variations*) defines if the control is:
 1. **Required** describes whether an attribute is mandatory or optional, i.e. the field cannot be left unfilled, and describes if the user is demanded to fill an attribute.
 2. **Relevant** describes whether an attribute is relevant, i.e. the field can be shown or hidden.
 3. **Changeable** describes whether the user can change an attribute value.
 4. **Rating Relevant** describes whether an attribute triggers the *Calculation Service*. Which means that changing a rating relevant attribute would impact the premium.
 5. **Master**: Describe whether an attribute value change will impact the *Product* structure or an attribute domain.
- **Continuous Entry** defines the available valid values for any numerical domain. Multiple continuous entries can be attached to the same domain where the overlap in the possible values will not result in duplicates on the front-end applications. Any default value defined on the *Domain* must exist in the associated *Continuous Entry*. The *Continuous Entry* is composed by:
 1. **Min**: Low-end of range.
 2. **Max**: High-end of range.
 3. **Increment**: Increment range.
 4. **Unit of Measurement**: Type of range defined (e.g. Dollar, Euro, Day, Visit, Percent, etc).
- **Filter** defines the available valid values for any string domain E.g., the Benefit Period domain on an Accumulator CRV would hold the valid values of 'CalendarYear', 'BenefitYear'.
- **Formulas** are rules used by *Information Service*, *Domain Service*, and *Calculation Service* to calculate or derive values. *Formulas* are also used to determine the availability of components i.e. *Formulas* are used on CRV to activate product variations. A CRV can have multiple *Formulas* associated. If a *Formula* returns *true*, then the CRV will be activated and therefore a product variation will occur, applying the new rules described by the activated CRV, i.e. the BO's that doesn't make sense for that rule can be removed, or even a new BO that simply started to be relevant for the product will be added.
- **Key Type** is a unique identifier for an element. If there are several key types, then they combine to create a unique identifier. For example: with *Plan Options* there are two key types: "Plan Option Type" and "Plan Option Name" which would combine to form the unique concrete identifier of: {Plan Option Type},{Plan Option Name} = Acupuncture,AcupCoveredNetNonNet. Usually, each BO is composed by some *Key Types*, such as: *BenefitType*, *AccumulatorType*.

Each *Key Type* is composed by:

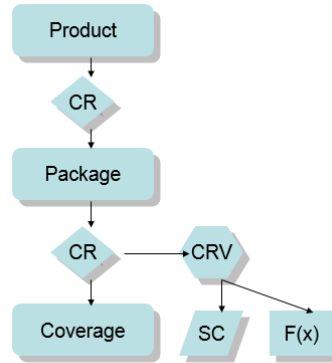
1. **Acronym**: An abbreviation form of other words and pronounced as a term.
2. **Name**: The complete description of the Key Type.
3. **ShortText**: Usually is modeled with the same content of the Name. However, it can be used to act as helper description. Example: If the presentation layer is based on HTML, it may be used as a **title** of an HTML element.

As an example, the PlanOptionName is a *Key Type* where its acronym, name, and shortText is modeled as follows:

- | | | |
|--------------|---|---|
| 1. Acronym | ⇒ | AcupCoveredNetNonNet |
| 2. Name | ⇒ | Acupuncture Covered in Network and out of Network |
| 3. ShortText | ⇒ | Acupuncture is fully covered |

- **Lookup Tables**, are files that are generated by MWB. These are Database (DB) tables used by formulas to retrieve data from the tables (e.g. for rating purposes or to determine the availability of components by state, zip code). More about *lookup tables* can be found at section: 3.2.3.

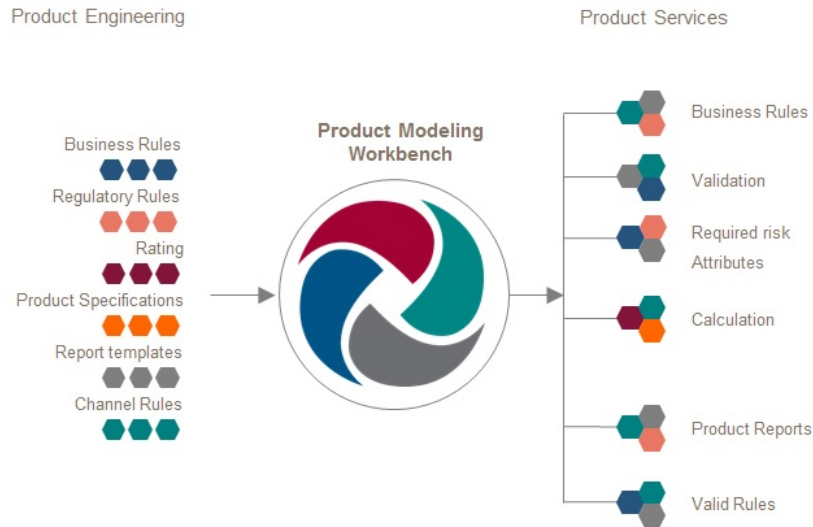
Figure 3.3 exhibits a small example of a model that can be built when all the previous concepts are combined with each other. Together, these elements allow the creation of *Products* with different conceptual models. MWB has achieved what is needed to create any model, with several rules and variations.



Source: FJA-US (2015) "Product Machine", internal company documentation, July 2014.

Figure 3.3: Combining elements to create conceptual models

The work completed in MWB leads to the generation of *Product Services*, constituting the *Product Machine run-time System*. Figure 3.4 illustrates how does the work of *Product Engineers* and the composition of all *Rules* and *Product Specifications* are leading to *Product Services* that serve the downstream systems.



Source: FJA-US (2015) "Product Machine", internal company documentation, July 2014.

Figure 3.4: Unified Product Platform Ecosystem

There are different *Product Services* created by MWB, allowing other applications to consume the modeled data. Each *Product Service* has its purpose, API, and response structure. The main *Product Service*'s are: the *Information Service*, *Validation Service*, *Functional State Service*, *Domain Service* and *Calculation Service*.

The **Information Service**, also known as *Info Service*, is the main service of PM. It provides the available products, options, and riders. This service returns an entire product definition and its structure, i.e. BO's, BA's, and their child objects with their *Attribute State*. This service allows downstream systems to fetch and get the products available for a given input/criteria, for example, according to an effective date. This service returns not only the product structure but also what are the default values for each returned BA.

The **Validation Service** enforces consistency and accuracy for the changes made by users on downstream systems. This service enables the consistency by validating the product against its configured *Rules* and *Formulas*. It returns an accurate message about what went wrong within the product data, identifying what is the element within the *Product Model* that is related to the validation error.

The most common validations present in PM models are the required fields that were not set by the user and also the validations about min and max values where the minimum shall be less or equal than the maximum value. Although these are common and data scoped validations, this *Service* also allows validations related to the product itself, where it can be validated if the *Coverage A* is not selected when the *Coverage B* is simultaneously selected.

The **Functional State Service** is used to manage attributes changeability. It defines the requiredness and relevance of an attribute. Whenever the functional state declares that an attribute is relevant, it

means the attribute will be displayed on the user screen. On the other hand, if PM states that an attribute is not relevant, then it can simply be defined as an hidden attribute at the presentation layer. Whenever an attribute is required, it means that the attributes cannot be left without the user set one of the valid values within the possible *Domain* values. In this case, if the *Domain* of an attribute does not have a *Default Value* modeled, then the user must choose one value without leaving the attribute empty. If it occurs, when the *Product* is validated against the *Validation Service* it will fail and return a message stating that this attribute is required and cannot be left empty.

The **Domain Service** provides the *Default Value* and the *Domain* for each attribute, i.e. the possible valid values that the user can choose for input fields.

The **Calculation Service** for Life LOB calculates the premium according to the selected coverages within the *Product*. For Health LOB, it calculates the benefits affected by the selected *Plan Options*.

3.2.2 Testing Workbench

TWB is a testing tool where the *Product Modelers* can validate if an updated PM model is correctly defining the *Rules*, *Formulas*, or new Objects added to the model and validate if they are correctly modeled when running the new model with S&S front-end application. This tool has two main configuration parameters:

1. The PM Model version that the modeler wants to test
2. The Front-End (S&S) version that the modeler wants to test his PM model

Usually, the testings using TWB are executed with closed versions of S&S; this avoids the instability of using a SNAPSHOT¹ artifact which may contain ongoing developments and therefore causing errors on modeler's tests leading to false positive issues with the tested PM model.

3.2.3 Data Workbench

Data Workbench is an eclipse perspective within the MWB development environment. Used as a Database connection manager, DWB creates lookup tables compatible with the main database providers.

After defining a connection with the correspondent JDBC Driver, DWB has access to all modeled data allowing its exportation into the established connection. Here, DWB creates DB files that can be eventually used by downstream applications. Acting as read-only data, the created tables are usually used as information catalogs allowing downstream applications to have: filtering mechanisms and front-end translations.

¹If a version number is followed by -SNAPSHOT, then Maven considers it "as-yet-unreleased" version of the associated MajorVersion, MinorVersion, or IncrementalVersion.

3.2.4 Reporting Workbench

Reporting Workbench is built on top of Business Intelligence and Reporting Tools (BIRT), an Eclipse based open-source reporting system for web applications, and mainly used with Java based platforms. BIRT provides a faster way to design and deploy reports with seamless data integration. RWB is a standalone reporting solution that is used to develop and deploy dynamic business reports, performing data analysis such as: Benefit Summaries, Coding documents, Key Performance Indicator (KPI)s, and data validation.

3.3 Sales & Service

S&S is the msg life Unified Distribution Platform (UDP). A platform to enable faster and better *Insurance Distribution* with maximum effectiveness and cost reduction. S&S improve time to market; reduce software development costs that come from *Distribution Channels* and mobile device support; improves sales process by promoting intelligent pro-active sales features, and promote customer and partner self-service, to improve satisfaction and reduce back office costs.

Being a Multi-Channel, Multi Line-of-Business (Life, P&C, Health), Multi-Device and Multi-tenant², Sales & Service is designed to be integrated with other modules. Such as PDS (that holds all business rules), PAS (to processes including rating, quoting, binding, issuing, endorsements, and renewals) and to be agile, and business focused where it can be customizable by business users; therefore it minimizes IT development.

Figure 3.5 exhibit that S&S follow a *Layered Software Architecture* and provides extension points to support different *Authentication* and *Authorization* providers, ID Generators, CRM, Party, Printing, PDS, PAS, and widgets³.

The S&S technology stack, also described in figure 3.5, is composed of a presentation layer that uses HTML5, CSS3 and ECMAScript 6 (ES6) specification (commonly known as JavaScript (JS)). The main libraries used are: **RequireJS**, a file, and module loader, optimized for in-browser use; **BackboneJS** to deliver Model View (MV*) design pattern including routing control for the client-side code (commonly a Web Browser); **JQuery**, to provide easier element selection, JQuery UI components, Event handling, etc; **LESS**, a Style-sheet language that enables the re-use of common styles and allow the usage of variables.

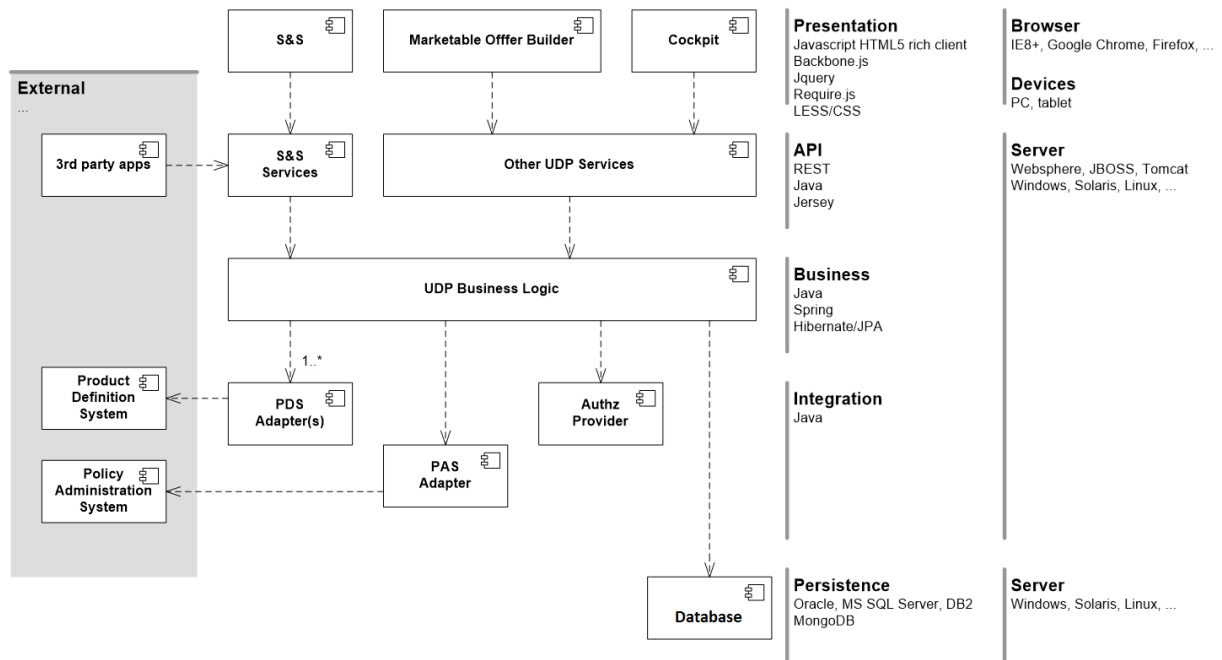
At S&S server-side, the local main web container is the *Apache Tomcat*⁴ and uses **Jersey Server**, an implementation library for Java standard JAX-RS⁵. To enable serialization from Plain Old Java Object (POJO)'s (Java *BO's* or *Data Transfer Object (DTO)*'s) to JavaScript Object Notation (JSON)

²An architecture in which a single instance of a software application serves multiple customers - (Superior Consulting Services, 2016)

³Small application with limited functionality that can be installed and executed within a web page by an end user

⁴Apache Tomcat™ is an open source software implementation of the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies

⁵Java API for RESTful Web Services



Source: msg Life Iberia (2015) "Sales & Service - Project Architecture", internal company documentation, September 2015.

Figure 3.5: Sales and Services - State of the Art

or even Extensible Markup Language (XML) and vice-versa is used **Jackson**, an implementation for the Java standard JAX-B⁶.

Internally, on the *Business Logic Layer*, it is used **Spring Framework** for Inversion of Control (IoC). Spring enables configuration over implementation classes that will run given an application context⁷. Moreover, *Spring* is commonly used to describe a specific customer application context, with its necessary class overrides and customer-specific method implementations.

For persistence is used **Hibernate**, a well-known Object-Relational Mapping (ORM) that also ensure and manage transaction operations with the DB.

For testing, S&S have several **JUNIT** for *unit testing* and use **Selenium** and **Cucumber** for *Functional Tests*.

Maven manages the dependencies and adds the necessary classes and packages into the application *classpath*⁸. Maven compiles, builds and deploy Java ARTifacts (JAR). It also provides a dependency management to build Java artifacts, Web application ARchive (WAR) among other binary files. As far as application artifacts are concerned, maven profiles also enable customer specifications where maven can add or remove files from the final application artifacts according to the specified profile configuration.

Jenkins provides continuous integration and continuous delivery features to applications. Jenkins

⁶Java Architecture for XML Binding

⁷Inside a Spring application, contexts beans that interact with each other to provide the application services

⁸Classpath is a way to tell applications, including the JDK tools, where to look for user classes. - (Oracle, 2016)

build and test software projects continuously making it easier for developers to integrate changes to the project.

3.4 PM and S&S - Communication

To communicate with PM model, S&S must implement the generated interfaces available in a given PM model JAR. These interface implementations are describing the domain, and must be manually developed and following the same structure declared by PM. This domain model needs to be well-defined and correctly mapped on the ORM files so that it gets persisted in the DB. It also needs to be well-configured on JAX-B implementation so that the objects gets correctly serialized/de-serialized between the client and server HyperText Transfer Protocol (HTTP) requests.

Due to PM generic concepts, PM has the required dynamisms to model different domain models. Furthermore, with its code-generation techniques to release a new JAR based on the modeled data in seconds, the PDS adapter layer described in Figure 3.5, needs to be often changed to comply with the model interfaces declared by the PM model JAR. The constant changes in S&S application constitute the main motivation to develop a presentation layer that can be auto-regenerated upon PM changes, without any manual coding and completely following the intended domain model on PM.

To build a proof-of-concept that will generate a presentation layer based on a given PM model, the next section will describe code-generators state-of-the-art.

4

Code Generation

Contents

| | | |
|-----|--------------------------------------|----|
| 4.1 | Introduction | 35 |
| 4.2 | Code Generation Approaches | 36 |
| 4.3 | Related work | 38 |
| 4.4 | Conclusions | 39 |

This chapter presents the State-of-the-Art about *Code Generators*. Here, will be explained what are the types of code generators and the main differences identified.

4.1 Introduction

Code generation is a technique used for rapid software development, where it automates regular coding tasks of software design. (Imam et al., 2014) Furthermore, automatic code generation provides rapid software development as it saves time and effort, enhances both software quality and accuracy, and frees developers from boring routine tasks. (Imam et al., 2014)

By its definition (IEEE, 1990), a code generator is a software tool that accepts as input the requirements or design for a computer program and produces source-code that implements the requirements or design. "Code generators are been proven that they are important tools in software development since they are allowing to automate repetitive coding tasks, and reduce developments costs". (Franky and Pavlich-Mariscal, 2012)

In a context of Model-Driven Architecture (MDA), code generation is a very useful technique to reduce the effort to develop software systems. Code generators have a major role in process automation, in which can automatically implement the models defined by the system designers or modelers. (Franky and Pavlich-Mariscal, 2012) Similarly, on Domain-Specific Modeling (DSM), it is used code generation to implement the domain model. Furthermore, DSM does not expect that all code can be generated from models, but anything that is modeled from the modeler's perspective, generates complete finished code. (Kelly and Tolvanen, 2008)

There are two different types of code generators: passive code generators and active code generators. (CodeSmith Generator, 2016)

4.1.1 Passive Code Generators

Passive code generators are commonly used to generate code once and then giving up all responsibility for it. Wizards and builders that are usually available on modern IDE's are typically passive code generators. This type of code generation can afford a wide head start, ranging from generating code for small object classes to generating code for entire object class hierarchies. A developer can later customize this generated code manually. (CodeSmith Generator, 2016) (Imam et al., 2014)

However, once the code gets generated, a passive code generator cannot regenerate it with the custom changes. Hence, those changes will be lost. (CodeSmith Generator, 2016)

4.1.2 Active Code Generators

According to (CodeSmith Generator, 2016), active code generators are designed to maintain a link with the code that is generated over the long term by allowing the generator to run multiple times over the same code. *Templates* are the source-code for this type of code generator. If an entire class structure is not correctly defined, a single change on the *template* can replicate the correct definition in whole project objects. The previous problem makes an active code generator very useful since it will save an incredible amount of time over fixing the same issue on an entire project structure, i.e. various class files.

4.2 Code Generation Approaches

Code generation frameworks¹ are used to generate the skeleton of a project and also to produce additional functionality progressively for new modules and use-case implementation. The automatic generation of an application skeleton contributes to the cut of overall implementation time, leading to a cost reduction. According to (Franky and Pavlich-Mariscal, 2012) study, "a code generation framework comprises multiple individual code generators that create source-code with specific functionality."

Code generation is being frequently used and has resulted in the development of different types of code generators. (Imam et al., 2014) Therefore, "code generators can be implemented in multiple ways: using template languages, regular expression substitution or as part of a model-driven approach." (Franky and Pavlich-Mariscal, 2012)

4.2.1 Template-based Code Generation

Implementing code generators using *templates* has been the most common approach. These templates are used to describe how the code generator will generate the code based on a given input data. *Template Languages* are used to specify the structure and include mechanisms to reference elements from the input data, to perform code selection and iterative expansion. (Franky and Pavlich-Mariscal, 2012)

In general, a template language has the following characteristics:

1. Mechanisms to include chunks of text written literally in the generated code.
2. When processing the template file, there is a *Context* with variables. This context will hold the input data that is referenced in the template file. The template engine will then populate the data according to the template.
3. A template file can also have conditional and loop statements that will write specific text in the generated code (either once or iteratively), based on logic conditions.
4. Template file can include *macros*, which are substitution functions that facilitate reuse of templates portions.

¹Set of reusable services and components, organized in an extensible structure, to simplify application development.

The advantages and disadvantages identified in (Franky and Pavlich-Mariscal, 2012) work are: although the usage of *Templates* provides flexibility to the code generator, the complexity of a code generator grows, as more templates are required to be maintained. Another problem with a template-based code generator is the debug, where it can be challenging and error-prone since it needs first to generate code from the existing templates, execute and debug the generated code, and finally propagate the corrections back to the affected templates.

The most popular code generator template-based languages are: Velocity, Jelly, FTL, Acceleo, JET, Xpand and MOFScript.

4.2.2 Code Generation based on regular expression substitutions

Regular expressions tools, such as *java.util.regex* library of Java can detect strings that comply with a given regular expression and can also transform those strings. A *regular expression* is a pattern denoting a set of strings of characters. The main components of *regular expressions* are 1) specific characters that a string must contain; 2) character classes donating characters that a string may contain at a specific position, e.g. numbers and letters; 3) quantifiers indicating the presence or absence of certain characters inside a string. (Franky and Pavlich-Mariscal, 2012)

(Franky and Pavlich-Mariscal, 2012) have also proposed a code generator using an *ANT* task *replaceregexp* that uses *regular expressions* as a substitution technique to create source-code based on another source-code from an existing application. In this case, this technique was mainly used to rename classes, packages, and configuration files.

4.2.3 Code Generation in a Model-Driven Environment

In the context of Model-Driven Environment (MDE), software can be developed starting from its conceptual model, i.e. what are the main entities and how they are related to each other.

This type of code generators relies on a *design model* that can be stored using different formats, such as Eclipse Modeling Framework (EMF) or XML Metadata Interchange (XMI). These *design models* describe the system architecture, where the design model may include an abstract representation of reference source-code components if necessary. The code generator takes the design model as input data, generates the project skeleton, and automatically incorporates all of the required components (the ones referenced in the design model). (Franky and Pavlich-Mariscal, 2012)

Often, *stereotypes* are used in MDE code generation. These stereotypes are usually based on Domain Driven Design (DDD) concepts such as *Service*, *Module*, *Entity*, *Value Object*, *Repository* (More about DDD at Appendix: C.1). As described in (Franky and Pavlich-Mariscal, 2012), a study about code generators, each class in the Figure. 4.1 (a simplified class diagram for illustrative purposes) has a stereotype denoting the reference code that will be used to implement that class. Both *Hotel* and



Source: Improving Implementation of Code Generators: A Regular-Expression Approach (Franky and Pavlich-Mariscal, 2012)

Figure 4.1: An example of design model (a class diagram)

Room classes have the *«Entity»* stereotype, which means that the generator responsible for carrying out these objects will be using the persistence entity component with Create, Read, Update, and Delete (CRUD) operations. On the other hand, *SalesReport* and *IncomeReport* have the *«Report»* stereotype, which means the code that implements these classes will be the components that are responsible for implementing reports.

The main difference between the MDE-based generators and the frameworks described in the preceding sections is that an MDE-based framework must include a generator that takes as input the design model. So that the generator creates the skeleton, finds all of the stereotypes that denote reference components (e.g. *«Entity»*, *«Service»*, *«Report»*, among others) and also invoke the correspondent generator for each component.

4.3 Related work

In (Franky and Pavlich-Mariscal, 2012) is suggested an implementation improvement using code generators with a regular-expression approach to convert existing project components into more generic ones. The regular expression substitution technique was used to create the project skeleton, and then each newly generated code was based on the existing project source-code. Generic names replace the names of modular units such as classes and web pages. They have used a component to set some *parameterizations*, including a full description about all substitutions to be performed on the reference source-code. On this project, the usage of a passive code generator is clearly identified by their affirmation:

”Developers of this project could modify the generated code to add new functionalities; whereas code generator cannot be used again to produce the same code, since the changes manually made by developers would be overwritten.” (Franky and Pavlich-Mariscal, 2012)

(Imam et al., 2014) suggested the employment of an Expert System (ES), also known as the Knowledge-Based System (KBS), for developing an Automatic Code Generator (ACG). In their work, it was used

a rule-based system and frames knowledge representation techniques to present the design of a passive code generator. As a case study, it was created a code generator to generate a device driver program.

On (Talab and Jawawi, 2011) work, it was used executable Unified Modeling Language (UML) 2.0 state machines and composite structures that were suitable as inputs for their code generators. The code generators were used to generate Component-Oriented Programming (COP) framework state machines.

(Wang et al., 2012) has proposed an automatic Real Time (RT)-Java code generation approach based on the Architecture Analysis & Design Language (AADL) model for *ARINC635* (*AADL635*) to enable the development of RT-Java ARINC635-based avionics software more productive and trustworthy. The main contributions in this work were: 1) a mapping from the AADL635 model to a high-integrity RT-Java programming model for ARINC635 (RT-Java635); 2) an ARINC635 compliant RT-Java code generation algorithm suitable for complex multi-task collaboration interaction situation. According to (Wang et al., 2012), they have implemented the RT-Java class library and the corresponding code generator.

4.4 Conclusions

As described, code generation is a technique that provides rapid software development due to the automation of the most regular coding tasks. However, to address the requirements of this project, a *code generator* tool either based on templates or regular expression substitution is not enough.

The regular expression approach is not suitable for the desired solution since it requires an existing source and uses it as a template to execute the intended changes (e.g. rename artifacts) which are not the goals of this project. Regarding template based code generators, although these could address the requirements, it would require the creation of multiple templates and definitions since a system contain various types of files, e.g. Java files, Hibernate Mapping files, REST API, Javascript written in a specific web development framework, HTML templates, CSS.

Consequently, to develop a front-end application that gets always generated and fully adapted to a given PM, it will be necessary a tool that could provide features to enable a presentation layer to get completely generated based on the PM model, from its domain model to its front-end screens.

Therefore, the next chapter will describe what are the main existing Java frameworks that use code generation techniques to do a full stack code generation. The frameworks that can partially do it, will also be included to determine if they are suitable to address particular areas within an application.

5

Application Code Generators

Contents

| | | |
|-----|--|----|
| 5.1 | Application Code Generators - Frameworks | 43 |
| 5.2 | Frameworks Comparison | 51 |
| 5.3 | Conclusions | 54 |

5.1 Application Code Generators - Frameworks

This section describes what are the main Java-based application code generation frameworks able to provide more than one type of file generation.

5.1.1 AndroMDA

AndroMDA is an extensible generator framework that adheres to the MDA paradigm. It transforms UML models into deployable components for different platforms, including Java. *AndroMDA* uses a cartridge concept that can generate code for several technologies. This cartridge concept is used to isolate the logic from various technologies within a system application. Figure 5.1 shows that AndroMDA with a given UML model can generate multiple blocks of artifacts needed to an application, i.e. it can generate a cartridge holding Java code, another with hibernate configuration and so on.

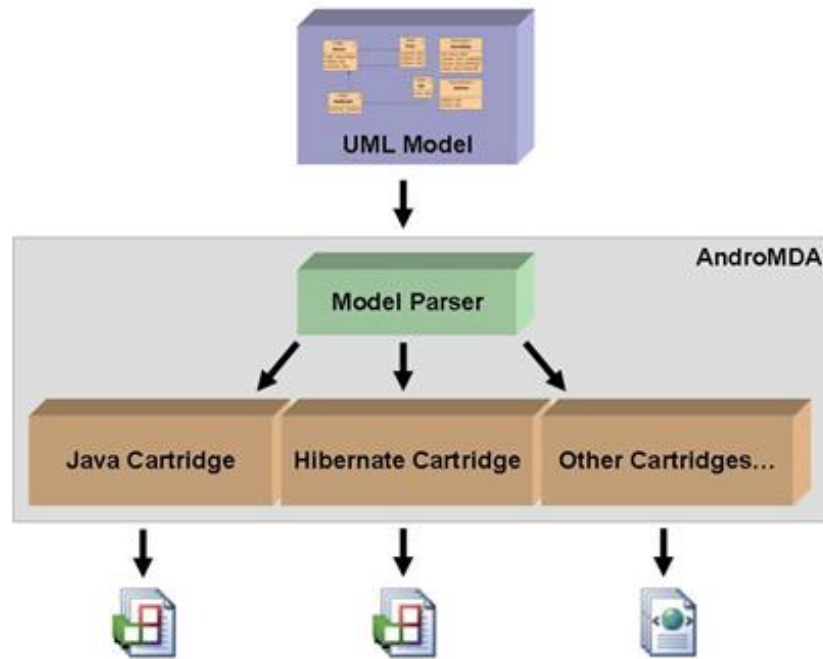


Figure 5.1: AndroMDA: From UML Model to multiple cartridge generation

Source: Getting started Java – Introduction (AndroMDA, 2016)

AndroMDA provides several cartridges out-of-the-box. For example, the Hibernate and Spring cartridges generate the service and data layers for the application. Furthermore, the database schema can be exported to script files, allowing the creation of the application database.

Regarding customizations on top of this tool, AndroMDA can generate custom artifacts from a given model, where developers can write custom cartridges.

According to *AndroMDA* documentation, this tool can generate enterprise quality code that is also

highly customizable to meet the project's particular needs. It also enables the creation of better applications and maintain order on large projects, enforcing best practices and let developers focus on high-level problems instead of wasting time on repetitive code.

"*AndroMDA* takes as its input a business model specified in the UML and generates significant portions of the layers needed to build a Java application." (AndroMDA, 2016) These layers are Presentation Layer, Business Layer, and the Data Access Layer that communicates with Data Stores.

- **Presentation Layer**, AndroMDA currently offers two technology options to build web based presentation layers: Struts and JavaServer Faces (JSF). It accepts UML activity diagrams as input to specify page flows and generates Web components that conform to the Struts or JSF frameworks.
- **Business Layer**, the business layer generated by *AndroMDA* consists primarily of services that are configured using the Spring Framework. These services are implemented manually in *AndroMDA*-generated methods, where business logic can be defined. These generated services can optionally be front-ended with Enterprise JavaBeans (EJB) making the services deployed in an EJB container (e.g. JBoss). These services can also be exposed as Web Services, providing a platform-independent way for clients to access their functionality. *AndroMDA* can even generate business processes and workflows for the jBPM workflow engine (part of the JBoss product line).
- **Data Access Layer**, *AndroMDA* uses Hibernate to generate the data access layer for applications, by generating a Data Access Object (DAO) for each entity defined in the UML model. These data access objects use the Hibernate API to convert database records into Java Objects and vice-versa. *AndroMDA* also supports EJB3/Seam for data access layer.
- **Data Stores**, since *AndroMDA* generates applications using Hibernate to access the data, the databases supported by Hibernate can be used to store the application data.

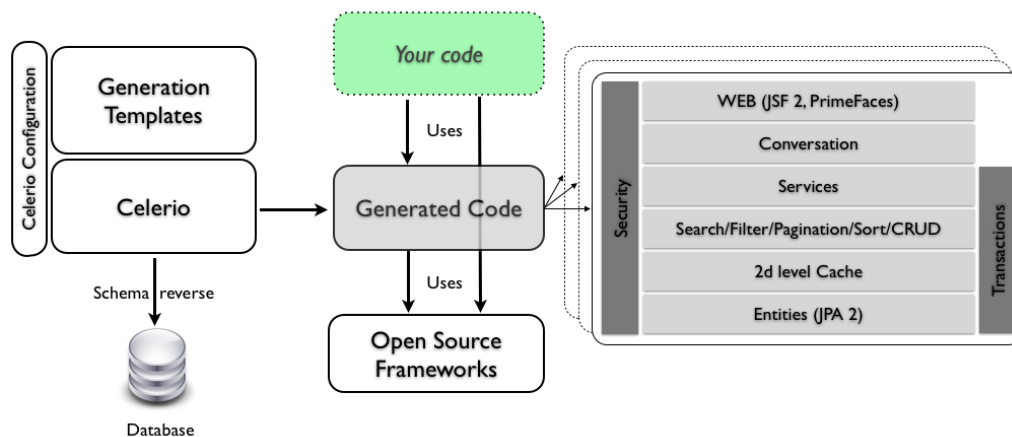
5.1.2 Celerio

Celerio (Jaxio, 2016) is a code generator tool for data-oriented application. Celerio uses as its input the entity-relationship model used by all relational databases. To obtain this model, *Celerio* connects to a given database and performs a reverse engineering of that database schema. *Celerio* supports the most common databases such as Oracle, MySQL, Postgres, and DB2.

The model can be augmented using a configuration file that can hold inheritance settings, variable re-names, bi-directional associations, among other definitions. With this configuration files, *Celerio* executes code generation templates written in *Velocity*.

Celerio comes with ready-to-use code generation templates organized into templates packs ('Backend' pack, 'JSF 2' pack). These templates address most use cases of data-oriented applications.

Regarding customization, *Celerio* allows the definition of new generation templates where the existing ones can be used as inspiration. (Jaxio, 2016)



Source: Celerio documentation (Jaxio, 2016)

Figure 5.2: Celerio: Overview

Figure 5.2 gives an overview how *Celerio* connects to a database, and based on templates, generates code. The generated code is composed of customizable code plus third party libraries.

According to *Celerio* documentation at (Jaxio, 2016), it generates the presentation layer using JSF or *PrimeFaces*. On the Backend layer, it generates services and the data access layer, with its respective DAO's. The Backend technologies used are *Spring* for services and *Hibernate* for Object-Data mapping.

5.1.3 JHipster

JHipster combines three very successful frameworks in web development: *Bootstrap*, *AngularJS*, and *Spring Boot*. At its core, *JHipster* is a *Yeoman* generator. *Yeoman* is a code generator that export and uses the *yo* command to generate complete applications or valuable pieces. *Yeoman* generators promote what the *Yeoman* team calls the "Yeoman workflow", an opinionated¹ client-side stack of tools that can help developers to quickly build web applications, providing the needed infrastructure to get an application working without the normal associated manual setup.

The *Yeoman* workflow is made up of three tools to enhance productivity and satisfaction when building a web application: scaffolding² tool (*yo*); build tool (*Grunt*, *Gulp*, etc.); package manager (*Bower*, *npm*, etc.).

JHipster supports *Liquibase*³, which provides tracking of the DB schema changes over the time,

¹An opinionated person is certain about their beliefs, and expresses strongly their ideas(Cambridge-Dictionary, 2016b)

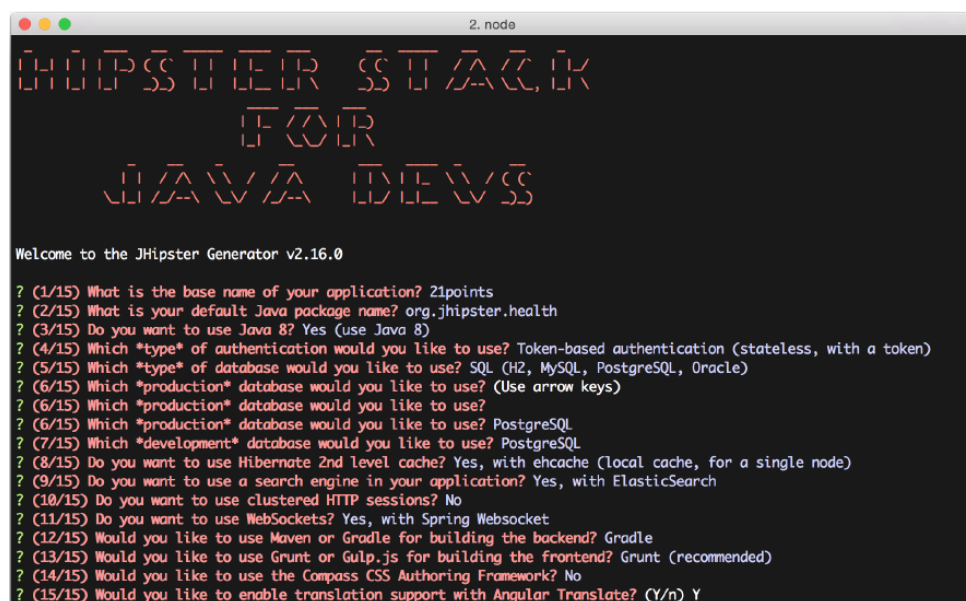
²Scaffolding, in the *Yeoman* sense of the word, means generating files for a web-based application with specific configuration requests.

³A database source control tool and allows DB refactoring

allowing track of the applied changes and therefore allowing a better DB maintenance.

Another useful feature supported by *JHipster* is the `Elasticsearch`, a distributed, open source search and analytics engine, designed for horizontal scalability, reliability, and easy management. It combines the speed of search with the power of analytics via a sophisticated, developer-friendly query language covering structured, unstructured, and time-series data. (Elasticsearch, 2016)

As illustrated by Figure 5.3, *JHipster* builds an application skeleton from the command line where the developer has to choose from the supported options, what components shall *JHipster* be generating for the application. At this point, the domain model was not yet expressed nor created by *JHipster*.



Source: The JHipster Mini-book (Raible, 2015)

Figure 5.3: JHipster: Generating the application from command-line

After creating the application skeleton, *JHipster* is ready to receive the instructions for the intended domain model for the application. For each created entity, *JHipster* is responsible for generating the following components:

- A - Database Table with respective *Liquibase* changelog.
- B - Java Persistence API (JPA) entity class
- C - *Spring* Data JPA Repository interface
- D - *Spring* MVC Rest Controller class
- E - *AngularJS* view, router, controller, service, and the related HyperText Markup Language (HTML) page

Like any other code generators, *JHipster* needs an input describing what shall be generated, whereas *JHipster* supports three different methods for the description of the intended domain model.

1. **jhipster-entity** is a *Yeoman* sub-generator that is also executed by the *yo* command. This entity sub-generator is a command-line tool that prompts developers with a set of questions about the *Entity* to be created. This process requires the description of all fields as well as the possible relationships with other existing *Entities*.
2. **UML** is another alternative for those that want to use visual tools. The supported UML editors by *JHipster* include Modelio, UMLDesigner, GenMyModel and Visual Paradigm.
3. **JHipster Domain Language (JDL)** is a DSL to describe all the intended entities and their relationships in a single *.jh* file (or more than one) with a straightforward and user-friendly syntax. (JHipster, 2016)

JHipster also provides JDL-Studio, an on-line tool to help developers describing the domain model for their applications. This on-line tool also offers a real-time visualization of the described model.

Listing 5.1 shows the JDL domain language syntax that allows the description of the intended domain model.

Listing 5.1: JHipster Domain Language - Syntax reference.

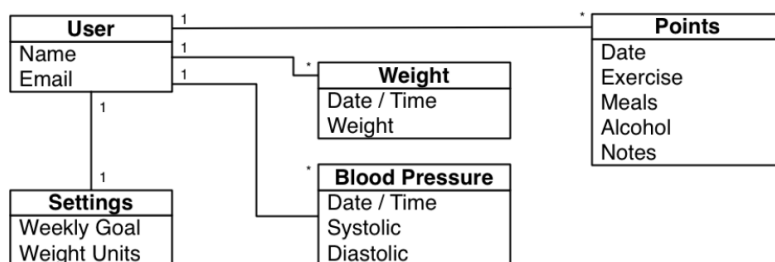
```

1  Entity Declaration
2
3  entity <entity name> {
4    <field name> <type> [<validation>*]
5  }
6      - <entity name> is the name of the entity,
7      - <field name> the name of one field of the entity,
8      - <type> the JHipster supported type of the field,
9      - <validation> the validations for the field          (optional)
10
11 Enum Declaration
12
13 enum <enum name> {
14   <enum values>
15 }
16     - <enum name> is the name of the enum,
17     - <enum values> the comma separated values in uppercase
18
19 Relationship Declaration
20
21 relationship <type> {
22   <from entity>[<relationship name>] to <to entity>[<relationship name>]
23 }
24     - <type> is the type of your relationship
25     - (OneToMany | ManyToOne | OneToOne | ManyToMany)
26     - <from entity> is the name of the entity owner of the relationship,
27     - <to entity> is the name of the entity where the relationship goes to,
28     - <relationship name> is the name of the relationship in the entity.
29     - The possible types and validations are those described here, if the
30     - validation requires a value, simply add (<value>) right after the name
31     - of the validation.
32
```

When *JHipster* is creating an *Entity*, it creates *JSON* and *YAML* files that are describing the created *Entity*, allowing developers to change those files and regenerate the *Entity* again using a simple command-

line instruction. If any conflict occurs, *JHipster* will present a message and prompt some questions so that the developer can solve those conflicts.

On *JHipster* mini-book (JHipster, 2016), the author explains how *JHipster* works, from the design of the intended domain model until the complete application generation. The author has used the command line to generate the application based on the following diagram presented in Figure 5.4.



Source: The JHipster Mini-book (Raible, 2015)

Figure 5.4: JHipster: Generating the application - An example of a domain model design

The author has also described some adjustments in the generated application to get the desired Front-End layout after the default generation, for example, the infinite scroll or pagination layout style. The final results from those adjustments are illustrated in Figure 5.5 and Figure 5.6, an application that allows people to control his health by logging their daily actions, like "did you exercise?", "did you hate healthy food?", among other indicators. These logs are then interpreted in the *Backend Services* which applies a scoring system. These points are then presented in a graphical way under the application dashboard (illustrated in figure 5.5).

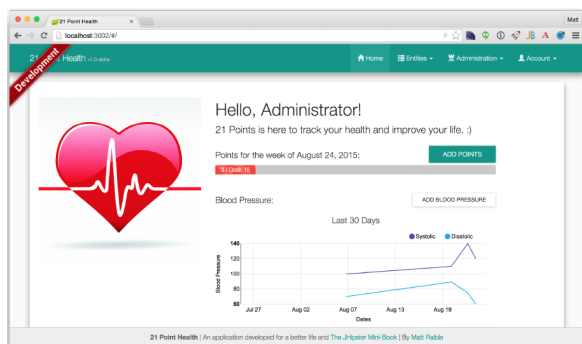


Figure 5.5: Example: Health App - Dashboard

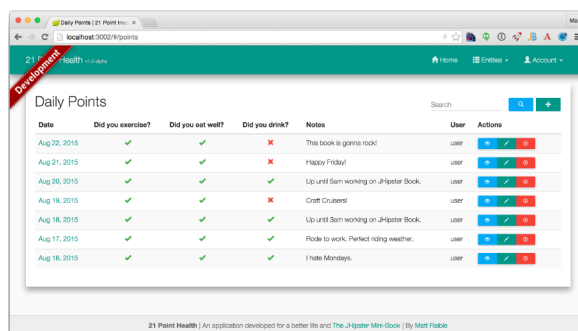


Figure 5.6: Example: Health App - Actions log

Source: The JHipster Mini-book (Raible, 2015)

Any application created by *JHipster* is production ready since it provides monitoring *Metrics*, caching mechanisms such as Ehcache (for local cache) and Hazelcast (for distributive cache system); optimized static resources using GZip and HTTP cache headers; runtime log management using Logback; and uses HikariCP connection pooling, for optimum performance.

5.1.4 ModelJ

ModelJ is a tool that uses code generation to create complete Java 2 Platform Enterprise Edition (J2EE) designs using *Struts* and EJB frameworks. According to (ModelJ, 2016) web-page, developers just need few steps to complete an application ready to be deployable into a *JBoss* web server. *ModelJ* uses proven design patterns to ensure a robust and an easy-to-maintain design. A UML design file is used to generate the domain model. It generates a Swing-based presentation layer. The backend comprises the following technologies: *Castor*, *Velocity*, EJB, *Struts*, *Tiles*, and *JBoss*.

5.1.5 Sculptor Framework

(Sculptor, 2016) is an open source tool that applies the concepts from DDD (See: C.1) and DSL (See: C.2). To generate the application, Sculptor gets a textual specification as input, i.e. a custom DSL is used to describe the design intent and then, it generates Java code and configuration accordingly. On the DSL, the concepts from DDD can be used, e.g. *Module*, *Entity*, *Value Object*, *Service*, and *Repository*.

According to *Sculptor* overview page (Sculptor, 2016): "The DSL and the code generation drives the development and is not a one time shot. The application can be developed incrementally with an efficient round trip loop. The *Sculptor* framework is useful when developing typical enterprise or web applications that benefit from a rich and persistent domain model. Within 15 minutes it is possible to go from scratch to a running application, including build scripts, Eclipse projects, domain model, JPA persistence, services and much more." (Sculptor, 2016) Furthermore, the developer can continue evolving the design, add code manually and regenerate the application. (Sculptor, 2016)

"The generated code is based on well-known frameworks, such as JPA, *Hibernate*, *Spring Framework* and follows J2EE standards" (Sculptor, 2016). Additionally, *Sculptor* takes care of the technical details and repetitive work, empowering developers to focus on delivering more business value.

Another interesting feature (See: C.2.2) that Sculptor provides is the ability to represent the model on different representations (e.g. using diagrams). *Sculptor framework* uses the *DOT* language to describe the generated model and uses *Graphviz* to generate UML diagrams. Different representations are always useful since it provides another perspective and it can be used to validate the application domain model with the PM modelers or other business people.

Regarding customization, "Sculptor framework is not a one-size-fits-all product. Even though it is a good starting point for many systems, sooner or later customization is always needed. Sculptor is designed and documented with this in mind, where the generated result can easily be modified to meet different business needs." (Sculptor, 2016)

The Listing 5.2 is an example of the custom DSL that Sculptor uses to interpret the domain intent and generate the code. Having this listing as an example, *Sculptor* will generate an *Entity* named *Planet*, being an *Entity*, *Sculptor* will also create the respective *Hibernate* mappings so that Planet gets mapped

to be correctly transformed into datasets and therefore stored in a Database. Furthermore, the *Service* layer generates all CRUD operations. A *Resource* with *REST* Web-Services is also created, allowing a web-client to fetch and store *Planet* related data. The *Sculptor* framework also allows the definition of custom DTO objects to be sent as the response of the generated *REST* Web-Services.

Listing 5.2: Example of a DSL to express REST WebServices

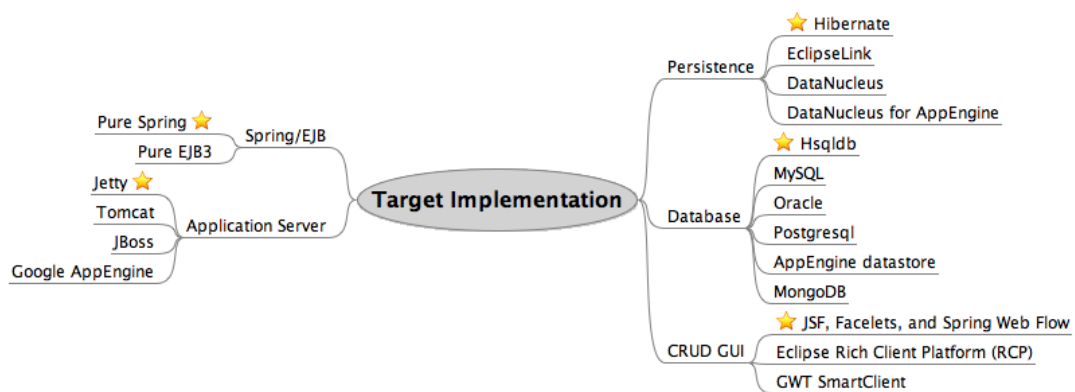
```

1  Resource PlanetResource {
2      show => PlanetService.findById;
3      String createForm;
4      create => PlanetService.save;
5      delete => PlanetService.delete;
6      showAll => PlanetService.findAll;
7  }
8
9  Service PlanetService {
10     findById => PlanetRepository.findById;
11     findAll => PlanetRepository.findAll;
12     save => PlanetRepository.save;
13     delete => PlanetRepository.delete;
14 }
15
16 Entity Planet {
17     String name
18     int diameter
19     Repository PlanetRepository {
20         findById;
21         save;
22         delete;
23         findAll;
24     }
25 }

```

The Appendix D.1 illustrates an example of a domain model used by (Evans, 2003) on his book about DDD. *Sculptor framework* uses this model to illustrate an example about how does Sculptor generate the necessary source-code to support the exactly same model used by (Evans, 2003).

As illustrated in Figure 5.7, Sculptor generates the main application components with their default implementations. Sculptor is very flexible and allows changes about the desired implementation by changing the configuration files.



Source: Sculptor documentation (Sculptor, 2016)

Figure 5.7: Sculptor: Target Implementation

Upon cases, whereas Sculptor does not support a specific implementation natively, this tool allows full control by enhancing the DSL Syntax by changing the Sculptor *Meta Model* (e.g. to define a new language element for the DSL) and make Code generation *templates* changes to generate new types of artifacts.

5.2 Frameworks Comparison

As described in the previous section 5.1, there are some tools already available to be considered as the chosen tool to address partially or completely this work aims. However, each tool has its own pros, cons and value propositions. Therefore, to provide a fair comparison about what are the desired characteristics for this work versus these frameworks capabilities, this section, will announce what the main desired characteristics are, that the chosen tool must have, as well as the features that are not strictly required but that are also important to be considered, e.g. its documentation and extendability.

Therefore, the sub-section 5.2.1 will enumerate what are the main features to be taken into account when compared with this work objectives. Then, each outlined tool in section 5.1 will be analyzed in the sub-section 5.2.2, where it is exposed their advantages and disadvantages when compared with the desired features for this project.

5.2.1 Desired Features

As described in section 5.1, the identified tools are composed of different technology stacks, has different capabilities, and provides different extendability levels.

Therefore, to provide a fair comparison between such different tools, the present sub-section enumerates what are the desired characteristics to be examined in each tool. Providing a guide-line to clearly identify what is(are) the tool(s) that stood out from those characteristics and consequently identifying the tool to provide a complete or a partial solution mechanisms to this work.

5.2.1.A Miscellaneous Characteristics

As already announced in the project constraints at section 1.3, there are general characteristics to respect when choosing the tool to address the proof-of-concept for this work.

The tool must be *maven* compliant whereas all its dependencies and build process are handled by *maven* since it is used on the overall projects in msg life. Additionally, it must be an open-source *Java* based solution to avoid monetary costs to the organization. Another important aspect to be considered is that the tool must be well documented and provide public developer guides to enable an easy and fast adaptation to msg life developers. Furthermore, the tool shall have an active support and an up-to-date bug fixing process.

Another relevant characteristic to consider is the usage of a custom DSL to describe the intended system since it allows the system description without a usage of additional software.

Usually, web-applications are composed of several files, with different syntaxes. For example *Java*, HTML, Cascading Style Sheets (CSS), JS, among others. Therefore, the selected tool shall have the capability to generate all the necessary files by holding various templates already implemented, so that all the different application files can be generated and consequently provide an application ready to run.

5.2.1.B Code Generation Capabilities

Regarding the backend, it is expected from the selected tool that the *Entities* gets generated. Additionally, it is desired that along with the system *Entities* generation, the tool also generates their database mappings, to enable persistence with the most common database providers. Furthermore, the stored information in the application is meant to be visible by the end-user in a frontend layer - typically a web browser. Therefore, the chosen tool shall also generate a RESTful API to enable a frontend interaction with the data stored in the backend.

Therefore, the chosen tool must generate a pleasant and up-to-date frontend application, implemented in a JS Single Page Application (SPA) framework. Where from this generated frontend layer, it is expected the interaction with the generated backend, whereas at least CRUD operations can be available and interacting with the generated RESTful API.

5.2.2 Advantages and Disadvantages

In the previous section, it was outlined what are the general and must have characteristics for the chosen tool to build a proof-of-concept for a full application code generation. Therefore, this section compiles all the information available in section 5.1 and compare each tool with the desired features described in the sub-section 5.2.1.

Having in consideration the desired characteristics, the following tables, will express the pros and cons side-by-side for *AndroMDA* 5.1, *ModelJ* 5.2, *Celerio* 5.3, *JHipster* 5.4 and *Sculptor* 5.5.

Table 5.1: Advantages and Disadvantages of: AndroMDA

| Advantages | Disadvantages |
|--|--|
| Generates Entities | Needs a UML Model as input to generate the application. |
| Generates Data Access Layer using Hibernate. | The Presentation Layer is not using a rich client side javascript framework. |
| Generate CRUD Operations. | No project updates since 2014. |

Table 5.2: Advantages and Disadvantages of: ModelJ

| Advantages | Disadvantages |
|---------------------|--|
| Generates Entities. | Requires a UML Model as input to generate the application. |
| | Lack of Documentation. |
| | Out-of-date project. |
| | Presentation Layer is Swing-Based. |
| | Not integrated with Maven. |

Table 5.3: Advantages and Disadvantages of: Celerio

| Advantages | Disadvantages |
|--|--|
| Generates Entities. | It needs an existing Database Schema as input to generate the application. |
| Generates CRUD Operations. | Lacks on a different kind of file templates to support the entire application. |
| Generates Data Access Layer using Hibernate. | The Presentation Layer is not using a rich client side javascript framework. |
| | Do not generate Web-Services. |
| | Lack of documentation and developer guides. |

Table 5.4: Advantages and Disadvantages of: JHipster

| Advantages | Disadvantages |
|--|--|
| The Presentation Layer uses a rich client side javascript framework (AngularJS). | JDL is not as flexible as Sculptor's DSL |
| Has a DSL to define the intended domain | |
| Good documentation. | |
| Generates Entities. | |
| Generates ORM and Data Access Layer | |
| Generates Web-Services with CRUD Operations. | |
| Integrated with Maven. | |

Table 5.5: Advantages and Disadvantages of: Sculptor

| Advantages | Disadvantages |
|--|--|
| It uses a DSL as input. Good documentation. | The Presentation Layer is not using a rich client side javascript framework. |
| Generates Entities. | |
| Generates Data Access Layer using Hibernate. | |
| Generates Web-Services with CRUD Operations. | |
| Integrated with Maven. | |

5.3 Conclusions

To determine which tool is best to address entirely or partially the project aims, it is necessary to compare what are the desired basic features to be supported. As described under project constraints 1.3, the tool must be accessible using *Maven*. Furthermore, it is also expected that the tool can generate at least the *Entities*, backend *Services*, and data access mechanisms, i.e. it shall be able to store and fetch data from a Database. That being said, the following table 5.6 provides a survey of these features and relates them with the desired characteristics for this project, giving a compiled summary of the previous section (5.2).

Table 5.6: Frameworks Comparison - Solution Required Features

| | AndroMDA | Celerio | JHipster | ModelJ | Sculptor |
|--------------------|----------|----------|----------|----------|----------|
| Maven integration | ✓ | ✓ | ✓ | ✗ | ✓ |
| Entities | ✓ | ✓ | ✓ | ✓ | ✓ |
| RESTful Services | ✗ | ✗ | ✓ | - | ✓ |
| DB Mappings | ✓ | ✓ | ✓ | - | ✓ |
| FrontEnd: JS-Based | ✗ | ✗ | ✓ | ✗ | ✗ |
| Total | 3 | 3 | 5 | 1 | 4 |

As demonstrated by Table 5.6, *ModelJ* is not an option for this project since it does not support most of the required features needed for this project. Moreover, *ModelJ* documentation lacks information about this tool, where no information was found about Services, REST API nor DB mappings.

Regarding *AndroMDA* and *Celerio*, both do not generate the *REST* API nor a presentation layer based on a rich JS framework.

Therefore, *Sculptor* and *JHipster* are the most suitable tools to provide the desired code generation for this project, where *JHipster* stands out with the ability to generate the front-end code using AngularJS, one of the most popular front-end javascript frameworks.

Consequently, the next Chapter proposes *JHipster* to address this project goal since it has proved to be the most suitable tool due to its code generation capabilities when weighted with the desired ones.

6

Proposed solution

Contents

| | | |
|-----|-----------------------|----|
| 6.1 | Overview | 57 |
| 6.2 | Conception | 57 |
| 6.3 | Development | 60 |
| 6.4 | Validation | 75 |
| 6.5 | Conclusions | 84 |

6.1 Overview

JHipster and Sculptor are both illegible tools to provide the capability to generate an application able to interact with a given PM model. However, knowing that Sculptor does not provide a rich javascript based front-end generation, *JHipster* stands as the preferred tool to generate the application since it provides all the generation capabilities announced at the project objectives.

To use *JHipster* to generate the application, it is first necessary to describe the intended domain model. As outlined in section 5.1.3, *JHipster* supports three different input formats to represent the domain model.

The domain model will be driven by a particular PM that will be configurable, i.e. the developer will have the opportunity to choose what is the PM version that will drive the application generation. Therefore, to get the domain for a particular PM model is necessary to develop a tool that will read down the domain model from the provided PM JAR. Then, it will be required a transformation from that domain model to one of the three available *JHipster* domain model representations (See the Section: 5.1.3).

After inferring the domain model from PM and translate it into a valid *JHipster Entity* model representation (that in turn will allow *JHipster* generate the entire application), it will be necessary to develop the following components using code-generation techniques and/or *JHipster sub-generators*:

- 1 - Inject the PM JAR as a dependency of the generated application, so that PM *Runtime Services* get available in the *classpath*, and consequently, the generated application can have access to the *Runtime Services*.
- 2 - Every existing *Entity* must implement *BOAdaptable* - So that the domain model can them interact the with PM Runtime Services (See the Section: 3.2.1).

6.2 Conception

This section will briefly describe what are the main actions to get *JHipster* to generate the application for a given PM model, which means that a specific PM must be selected. Therefore, the sub-section 6.2.1 holds a shallow description of how the selection of a PM model was designed. Then at sub-section 6.2.2, is described the necessary steps to translate the domain model into a valid *JHipster* input representation, so that finally, *JHipster* can generate the application as presented in section 6.2.3.

6.2.1 Choosing the PM Model

There are multiple PM models available within the msg Life portfolio such as *Life*, *Health*, *P&C*, *Group Life*. The generated application shall work at least with *Life* and *P&C* models as described at project objectives (See: 1.2).

Before the application generation takes place, the developer will have to have a way to select the desired PM (artifact and version) to be generated by *JHipster* and/or any other needed sub-generators.

Since the project is a *Maven* project that handles artifact dependencies, it can be used the *Maven* pom.xml file to set the proper PM dependency and let *Maven* pull the correct *JAR* from the internal msg life *Maven Repository* server. Only then, the developed tool can read and interpret the model from the *JAR* that should be already available on the classpath.

6.2.2 Interpreting the PM Domain Model

The domain model lives in a customer specific PM model, and the business objects present on that domain model extends the interface *BOAdaptable*. During the analysis of a model in a *JAR*, it was concluded that all *Entities* of the model reside in a single *Package* within the *Java classpath*. Therefore, it is possible to use the *Java Reflection API* to allow the inspection of any existing object at run time and extract useful information such as Class constructors, methods, and fields. However, *Reflection* should be carefully used due to known performance drawbacks. But for this particular case, its usage makes sense since the interpretation of the model will occur just once, translating the model into a structured data format. Then, this structured data can be translated into another format. In this case, it will be a valid *JHipster* input representation that will take place without any usage of *Java Reflection API*.

As described in the previous paragraph, every *Entity* of the domain model extends the interface *BOAdaptable*, and is located on a single *Java Package* where the algorithm to extract the model from a given PM model artifact could follow Algorithm 6.1.

Algorithm 6.1: Translating PM Entities into a structured data format

```

Initialize entityCollection collection;
entities iterator := all inherit objects from the interface com.fja.pm.BOAdaptable;
while iterator has next do
    entity := get next entity;
    if this entity was not yet processed then
        Create newEntity object;
        properties := read all getter methods from the current entity;
        for property ← 0 to properties.length do
            if property is not another entity type then
                newEntity add regular field;
            else
                add field type to newRelationship;
                set the relationship cardinality type;
        set the newRelationshipCollection into the newEntity;
        Add newEntity to entityCollection;
    else
        go to next entity;
return entityCollection;

```

As described in Algorithm 6.1, the model provided by the PM artifact can be interpreted using Java Reflexion API. The algorithm starts by searching for all implementations of *BOAdaptable* available in the classpath, which in turn represent a PM domain model element, i.e. a BO. Then the algorithm starts iterating all these Classes to infer their fields. If a field is a non-Java standard type (e.g. String, Double, Integer), it may represent another Entity object within the domain. In this case, this particular field will mean a relationship (one-to-one, one-to-many) and therefore will need to be treated accordingly, i.e. it may require that this child *Entity* needs to be created first so that *JHipster* can create the relationship between these two Entities.

6.2.3 Generating the JHipster Domain Model

As described in Section 5.1.3, *JHipster* can generate the domain model for the generated application with three different approaches: i) by the command-line using the `jhipster-entity` sub-generator; ii) using a UML class diagram; or iii) using their DSL called JDL.

The UML method approach is not ideal since it would require drawing the model on another UML tool. The command-line option is simple to use, it only requires the execution of a command: **yo jhipster:entity author**, here the Author *Entity* will be created. Then the developer is required to indicate what are the attributes for this *Entity*. Having all *Entities* created, the developer needs to set their relationships. However, the developer will have to answer all of those questions per each *Entity* and will require time to map each *Entity* Class that was found in PM model, leading to a non-ideal option since it will need some time to write down manually the *Entities* part of the domain model. Furthermore, it also requires interpretation of the model precedences, i.e. which *Entities* required to be first created so that a relationship owner can "know" what are the other *Entities* to establish the relation.

The usage of a DSL that could represent the entire domain model would be the ideal option. *JHipster* has developed the JDL, a text file which has its syntax to represent and hold whole conceptual model description. This file is interpreted by the *JHipster* engine to create the entire domain model by executing one single command: **yo jhipster:import-jdl jdl-model.jh** where `jdl-model.jh` is the file that follows the JDL syntax (see section 5.1). *JHipster* engine will analyze the *Entity* precedences and will sort them in a way that the relationships can be available for all the *Entities*. When *JHipster* detect that the described model is ambiguous, and the relationships end up on a circular dependency, it stops the process and alerts the developer to revise the model.

As seen on previous section (6.2.2), it is possible to affirm that the domain model can be generated in two different steps. The first one is the domain model interpretation from PM using Java Reflexion API and the second, is to re-write this interpreted domain model in a text file that follows the JDL syntax format so that *JHipster* can generate the application.

6.3 Development

This chapter describes the development process that allowed the dynamic creation of an application following the same domain model described under a given PM model. The sub-section 6.3.1, describes how the developer can choose and load the desired PM artifact. Sub-section 6.3.2 describes an ad-hoc¹ tool that will be responsible for interpreting the domain model PM artifact. Sub-section 6.3.3 is explained how did the *JHipster framework* was used to build down the entire infrastructure code to handle, Logging, ORM, REST Services, Authorization, among others handy feature generated by *JHipster*. At this point, the generated code will just represent a scaffolded project with a set of files. At sub-section 6.3.4 exposes the building of the *Domain Model*. The domain model will contextualize the generated application where the generated files start to be considered an *Application* with a purpose, since there is already a specific domain to work with.

6.3.1 Choosing the PM Model

As described in Section 3.1, Insurance business embraces various line-of-business, each one with their models and business rules. Due to this wide range of branches, PM models are often developed to be very specific and according to one single business model.

The existence of multiple models and the constant adaptation of S&S application to interact with these new models has motivated the development of this application generator. That being said, the code generator has to support at least two different PM models from various line-of-business, proving the potential of this project adaptability to distinct models by avoiding specific model code generation.

Therefore, it is crucial to develop a process to pick the correct model artifact to serve as input for the generator.

All the developed projects at msg life are compliant with Apache Maven, a build tool that uses a Project Object Model (POM) file to manage projects. *Maven* also has dependency management capabilities that are perfect to ensure the correct placement of the PM model in the Java classpath.

Listing 6.1 presents an example of a possible *pom.xml* to describe which PM model will be included in the Java classpath of the generated application. Declaring a dependency using *Maven* is very simple. It just needs to declare a proper identification of the artifact to be loaded using maven groupId, artifactId, and version. *Maven* will then download the correspondent JAR from the Sonatype Nexus which is the internal Repository Manager at msg life.

¹Created or done for a particular purpose as necessary

Listing 6.1: Maven - pom.xml file example

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0"
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4     http://maven.apache.org/xsd/maven-4.0.0.xsd">
5
6   <modelVersion>4.0.0</modelVersion>
7   <groupId>entityGenerator</groupId>
8   <artifactId>entityGenerator</artifactId>
9   <version>0.0.1-SNAPSHOT</version>
10
11   <properties>
12     <fja.groupId>com.fja</fja.groupId>
13     <pm.artifactId>metis.pm.life</pm.artifactId>
14     <pm.version>1.0.0-SNAPSHOT</pm.version>
15   </properties>
16
17   <dependencies>
18     <dependency>
19       <groupId>${fja.groupId}</groupId>
20       <artifactId>${pm.artifactId}</artifactId>
21       <version>${pm.version}</version>
22     </dependency>
23   </dependencies>
24 </project>
```

The groupId, artifactId, and version are declared as properties. This way it is possible to configure which PM version shall be used. Maven Properties can be declared implicitly in the pom.xml file and therefore being resolved immediately to their proper values. These properties can also be declared later at the runtime of a *Maven* lifecycle (e.g. clean, compile, install, deploy) or at the execution of any Maven plugin. To inject properties using Maven it is used the **-D** instruction followed with the property name and its value.

Listing 6.2: Maven - Exec Plugin

```
1 mvn exec:java -D exec.mainClass="entityGenerator.JDLGenerator"
2               -D fja.groupId="com.fja"
3               -D pm.artifactId="metis.pm.life"
4               -D pm.version="1.0.0-SNAPSHOT"
```

Listing 6.2 illustrates the usage of *exec maven plugin*. In this specific command, the plugin will run a Java -jar instruction where the *exec.mainClass* is the property that will indicate to the plugin what is the main Java class to run.

Before start the execution of this main class, Maven will first ensure that the project code is compiling and to do that Maven needs to fetch all of the project dependencies declared in the *pom.xml* file. Furthermore, Maven will use the provided -Dfja.groupId, -Dpm.artifactId, -Dpm.version properties to correctly update its PM model dependency, and once *Maven* finishes the download from the Repository Manager and the inclusion of that JAR in the project classpath, a compilation routine using the new model starts.

Right after a successful compilation the java classpath is ready and fully updated with the correct PM model, which means the code can now use any class from the PM model.

6.3.2 PM Model Interpretation

Right after *Maven* ensures that a given PM model is available in the classpath, it is possible to start analyzing the domain model structure that was modeled by the *Product Engineers*. To processed with this domain model analysis, it was used the *Java Reflexion API*, a set of utility instructions that makes part of the Java Development Kit (JDK) and provides introspection² capabilities.

Reflection is commonly used by programs which require the ability to examine applications running in the Java virtual machine. Reflexion is a relatively advanced feature and should be used only by developers who have a strong grasp of the fundamentals of the language. With that caveat in mind, reflection is a powerful technique and can enable applications to perform operations which would otherwise be impossible. Oracle (2016)

According to its documentation, Java Reflexion API has several drawbacks, and it "should not be used indiscriminately since it involves types that are dynamically resolved and where certain Java virtual machine optimizations can not be performed. Consequently, reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are frequently called in performance-sensitive applications". Oracle (2016)

Although Java Reflexion API must be carefully used, this technique is still a valid and a reliable option to provide PM models introspection, since this model interpretation will be just executed once. Then, the gathered information will be loaded into regular Java Objects that will hold all the needed data about the model. These objects do not require reflexion techniques and can then be used in a second step so that the model data can be written into a specific output format.

Figure 6.1 illustrates a sequence diagram describing the introspection process of a PM model. At *Step 2* the *Reflexions* class is created to inspect a given *Java Package* (for this specific case is the "com.fja" package). This object can retrieve all subtypes of *BOAdaptable* class that are simultaneously located at any named package that contains "com.fja".

All domain objects available in a PM model implement the *BOAdaptable* interface; therefore the *Reflexions* object infers what are the objects that are inherited from *BOAdaptable*.

Every PM model has an implementation of his model, i.e. it is possible to find a specific implementation of all the *BOAdaptable* interfaces in a sub-package named: "com.fja.model.impl". These classes were also determined by the *Reflexions* object, but the *interfaces* are the only objects that are relevant to be analyzed by the introspection tool. Consequently, at *Step 4* there is an additional filtering routine

²"Introspection is the automatic process of analyzing properties, events, and methods." Java Sun (2008)

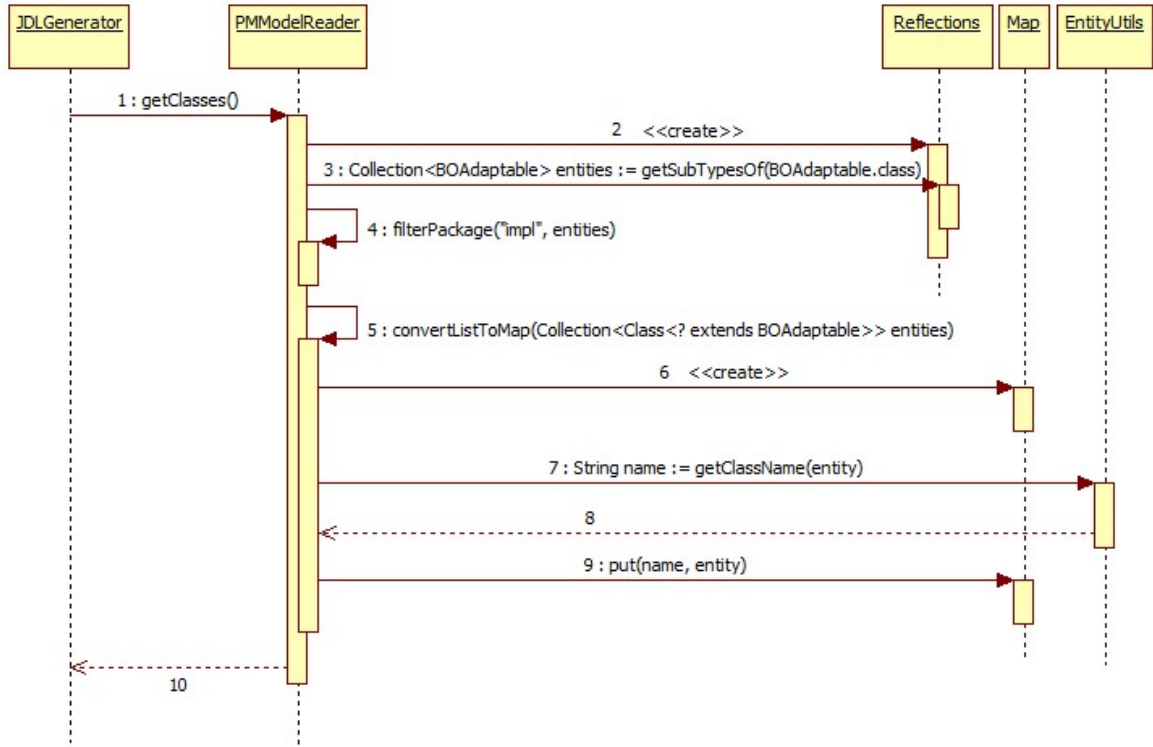


Figure 6.1: Sequence Diagram: PM model introspection

executed to make sure that the returned classes are only composed of the needed interfaces. These are the *Entities* that will drive the domain of the generated application.

Step 10 is the final step of this model introspection, where is returned a *Map* that holds the available *Entities* in the PM model. The key of the returned Map is composed by the *Entity* simple name, where at *Step 7* it is being removed the *BOAdaptable* prefix and the *BO* suffix, e.g. from a named class *BOAdaptableIllustrationBO* it is being translated into *Illustration*.

However, at this point, the information found in the model were only the entity names and the correspondent Java classes. Therefore, a second routine must be performed to infer what are the attributes that compose a given *Entity*.

There are attributes within the domain model that are not just common attributes. Some attributes of an *Entity* may have a type that refers to another *Entity*. In this cases we are not facing a mere attribute but a domain relationship. It is vital that the relationship information gets correctly inferred because it will be used to link and describe the entire domain model properly.

Every PM model has a root object that holds the entire domain structure. The root entity of *Life* model is the *Illustration*, whereas for P&C model is the *Proposal*. Either way, the correct retrieval of these relationships are crucial so that the relationship owner can also be accurately established after this

introspection process.

Therefore, to store the domain model hierarchy, a data structure was developed to hold the entire model and recursively attach the children of each entity. This parent-child hierarchy has allowed the dynamic discovery of the model root object. Furthermore, using a top-down strategy, it is always possible to correctly set the relationship owner, under the parent object.

Consequently, the diagram 6.2 describes the process of inferring an *Entity* children object. The first step is the retrieval of all the *Entities* from PM model that were previously outlined in the sequence diagram 6.1. Then, a *TreeMap* is created. It will hold all the *Entity* names and their correspondent children that also make part of the domain and shall be linked with the current *Entity*, forming a relationship. Inferring a child *Entity* is not simple nor a straightforward process, since multiple types of relationships may exist, such as: one-to-one, one-to-many, many-to-one and many-to-many.

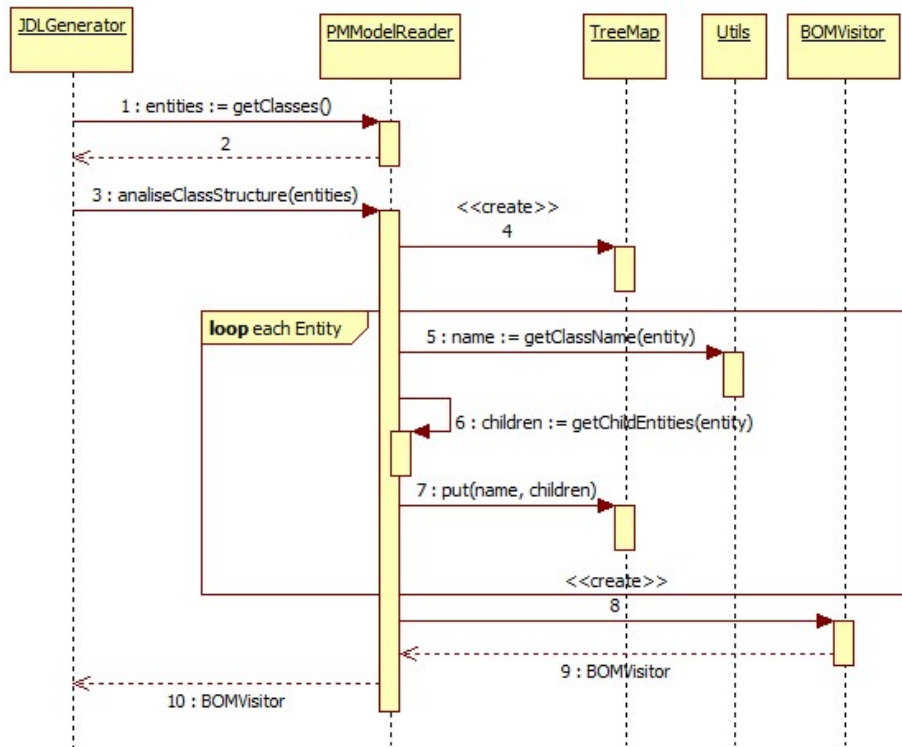


Figure 6.2: Sequence Diagram: Relationship analysis

Before diving into the algorithm details, it is necessary to analyze how does the domain model interfaces in a PM model artifact are exposed. Figure 6.3 shows a set of signatures that were snipped from the *BOAdaptableContractBO* interface. According to these method signatures, the children getter methods are also present on these method signatures. Therefore, is possible to see that any child object of a given *Entity* has two possible nomenclatures to be identified.

1. **getAll«AnotherEntityName»BO** - The return type is an Array, i.e. a Collection, meaning that the relationship between the current *Entity* and this child object will be inferred as being an ONE_TO_MANY relationship;
2. **get«AnotherEntityName»BO** - When the return type is not a Collection, the inferred relationship will be ONE_TO_ONE;

```

    ^ getBusinessTransaction() : BOAdaptableBusinessTransaction
    ^ getAllAdjustmentBO() : BOAdaptableAdjustmentBO[]
    ^ getAllPremiumComponentBO() : BOAdaptablePremiumComponentBO[]
    ^ getProducerBO() : BOAdaptableProducerBO
    ^ getAllPropertyBO() : BOAdaptablePropertyBO[]
    ^ getAllRateComponentBO() : BOAdaptableRateComponentBO[]
    ^ getAllRoleBO() : BOAdaptableRoleBO[]
    ^ getAllPackageBO() : BOAdaptablePackageBO[]
    ^ getUserBO() : BOAdaptableUserBO
    ^ getAllDocumentBO() : BOAdaptableDocumentBO[]
    ^ getAllExperienceBO() : BOAdaptableExperienceBO[]

```

Figure 6.3: Partial signatures of the BOAdaptableContractBO class: Relationship analysis

Having identified the method nomenclature that is being generated by PM on the model *interfaces*, it is possible to develop an algorithm that will retrieve the child objects and the respective relationship cardinality for a given *Entity*.

Algorithm 6.2 describes the logic to retrieve those *Entity* relationships as well as their relationship cardinalities. The algorithm is using *Reflexion* so it can infer which methods are available in the *Interface* that represents the given *Entity*. Then, a collection is populated with the methods that have the "BO" suffix in its return type. At this point, an iteration to those methods starts and is verified if the return type of the method is an *Array*. If so, the *relationship type* will be set as ONE_TO_MANY; otherwise, it will be set as ONE_TO_ONE.

Algorithm 6.2: Getting Model Relationships

```

For a given Entity ... ;
Initialize childrenRelationShips Map;
Method iterator := get all methods from Entity that contains "BO" in its return type;
while iterator has next do
    method := get next method;
    initialize a new relationshipType;
    if method return type is an Array then
        | relationshipType := ONE_TO_MANY ;
    else
        | relationshipType := ONE_TO_ONE ;
    anotherEntityName := methodName without "BO" and without "getAll" ;
    put the anotherEntityName and the correspondent relationshipType into the
    childrenRelationShips Map;
return childrenRelationShips;

```

At this point, the relationships are identified, but there are still ambiguous cases that must be resolved to correctly identify the domain model hierarchy. These cases occur when relationships have a *ONE_TO_ONE* cardinality in both *Entities*. To illustrate this ambiguous situation, the following listings illustrate a snippets JSON that is being saved in a file named *BO_Relations.json* by the *PMModelReader* for debugging purposes.

Listing 6.3 shows an example of a non-ambiguous relationship, where the *Contract* BO is related with many *Adjustment* BO *Entities* and each *Adjustment* has the parent relationship (*ONE_TO_ONE*) with his parent, the *Contract*. These cases have no ambiguity since the relationship owner is the *Entity* that holds the *ONE_TO_MANY* relationship cardinality.

Listing 6.3: Non-ambiguous relationship

```
AdjustmentBO: {
  ContractBO: "ONE_TO_MANY",
  CoverageBO: "ONE_TO_MANY",
  PackageBO: "ONE_TO_MANY"
},
ContractBO: {
  AdjustmentBO: "ONE_TO_MANY",
  DocumentBO: "ONE_TO_MANY",
  ExperienceBO: "ONE_TO_MANY",
  IllustrationBO: "ONE_TO_MANY",
  PackageBO: "ONE_TO_MANY",
  PremiumComponentBO: "ONE_TO_MANY",
  ProducerBO: "ONE_TO_MANY",
  PropertyBO: "ONE_TO_MANY",
  RateComponentBO: "ONE_TO_MANY",
  RoleBO: "ONE_TO_MANY",
  UserBO: "ONE_TO_MANY"
}
```

Listing 6.4: Ambiguous relationship

```
IndividualBO: {
  InsuredPersonBO: "ONE_TO_MANY",
  PartyBO: "ONE_TO_MANY",
  ProducerBO: "ONE_TO_MANY",
  PropertyBO: "ONE_TO_MANY",
  UserBO: "ONE_TO_MANY"
},
InsuredPersonBO: {
  IncomeBO: "ONE_TO_MANY",
  IndividualBO: "ONE_TO_MANY",
  PhoneBO: "ONE_TO_MANY",
  PropertyBO: "ONE_TO_MANY"
}
```

On the other hand, Listing 6.4 illustrates an example of ambiguous relationships between *Individual* BO and the *InsuredPerson* BO, where the parent-child is not possible to be directly inferred. These ambiguous cases occur because all PM models has the entire domain model defined with bidirectional relationships, i.e. each *Entity* has a one-to-one relationship with another child-*Entity* which also have a one-to-one relationship to its parent. For this reason, knowing which *Entity* is the parent is not straightforward.

To correctly get the entire domain model hierarchy, this flat study about each *Entity* and their direct relationships and its relationship cardinalities is not enough. Instead of a flat study, the introspection algorithms must infer the domain model on a hierarchically way, so that all the domain entities get correctly assigned to their parents without any ambiguity.

At the previous diagram 6.2, in the described *Step 8*, the *BOMVisitor* is created. This contextualized class is initialized with the already resolved flat *Entity* map and its relationships. Then it will build the necessary hierarchy and find the root *Entity*. Having this root *Entity* discovered it is possible to move down along the hierarchy, marking the currently visited *Entities* as the parent ones. The algorithm

continues moving down to a lower level, identifying these levels as their children and consequently avoiding any relationship ambiguity.

Therefore, to store the entire PM domain model hierarchy, it was created the *BO* class. This Class is illustrated in Figure 6.4 and will recursively hold the whole domain model hierarchy. Each node contains the *Entity* name, a reference to its parent *BO* and a collection of children *BO*s. This Class will also provide some helper methods so that this hierarchical structure can be navigable.

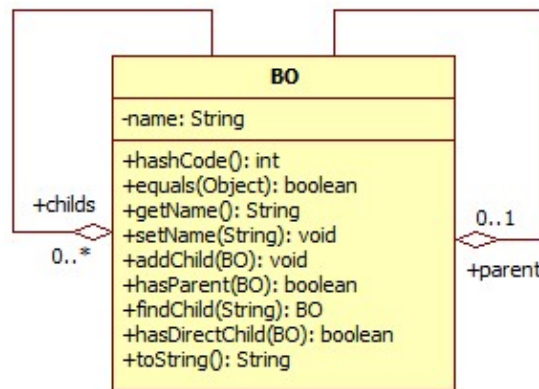


Figure 6.4: Class Diagram - Business Object

The *BOMVisitor* Class will be responsible for creating this *BO* Class that will hold the entire recursive model hierarchy. Therefore, this Class is initialized with the already founded *Entity* map. With this map, the constructor of the *BOMVisitor* will initialize the investigation about the domain model hierarchy. The methods executed by *BOMVisitor* are somewhat procedural. Therefore, the following Activity Diagram 6.5 shows what are the main actions taken by the *BOMVisitor* to infer which *Entity* is the root object for a given PM model.

The *BOMVisitor* Class starts by iterating over the map that contains the PM model *Entities* in a flattened form. This Class starts by building all the *Entities* available in the map and transforms them into a *BO* Class form. All the handled *Entities* are stored in a collection (*visited Entities*) to track the *Entities* that were already built and therefore avoiding infinite loops. At this point, *BOMVisitor* hold various *BO*'s with their drill-down children, i.e. each *Entity* is linked to its children objects.

The next step is the removal of *Entities* that don't relate to any other domain model *Entity*. Although these were found being subtypes of *BOAdaptable*, they do not relate nor are used by the *Runtime Services* in PM model. For this reason, these *Entities* are being disregarded.

To find the *Root Entity* for a PM model, so that the parent-child relationship can be correctly inferred for those that has one-to-one cardinality, the *BOMVisitor* has an algorithm to find the best eligible *Root Object*. Knowing that each PM model has only one root object that defines the entire BOM, and this information could be configurable using a *Maven Property* when setting the PM version dependency,

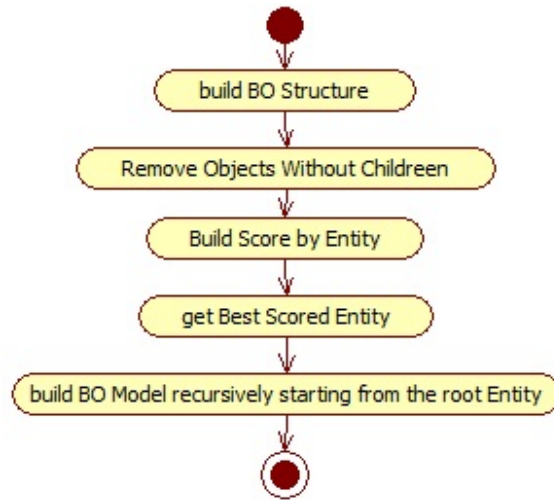


Figure 6.5: Activity Diagram - Building domain model hierarchy

it is better having the root *Entity* being inferred since it avoids input dependencies and user mistakes. Furthermore, it makes the process of generating the application being a simple "push of a button", more streamlined and clean.

The algorithm to retrieve the most eligible root *Entity* is based on the following statement: "The Entity that can visit more Entities in depth is the most suitable Entity to be the root object of the domain." (Alves, 2016) However, it is known that this statement does not contemplate all the possible scenarios where a draw in the score can occur if eventually, the model has the same amount of bi-directional one-to-one relationships. In this cases, it can always be used the *Maven Property* configuration approach since a human intervention is needed to investigate the domain model and set/configure which parent makes sense for the given domain model and its structural dependencies.

Having the *root Entity* inferred correctly, it is now possible to resolve the entire domain model as well as the ambiguity found in the bi-directional one-to-one relationships. Therefore, the final step in *BOMVisitor* is building the final root BO Class that will attach the entire BOM together recursively.

At this point, it is possible to read the domain model stored in a given PM model, since this information is now structured and can be later transformed into any output format that will serve as input for the chosen code generator tool.

6.3.3 Generate Application Skeleton

Before starting using *JHipster* specific commands to generate the application, is necessary to install Java 8, NodeJS, and GIT. Then, using Node Package Manager (NPM) the following packages will be retrieved from on-line repositories using the following NPM commands:

1. Install Yeoman: **npm install -g yo**

Yeoman is a generation tool that helps to kick-start new projects. Using the 'yo' command, Yeoman can scaffold complete projects or just useful parts.

2. Install Bower: **npm install -g bower**

Bower manage components that contain HTML, CSS, Javascript, fonts or even image files. Bower handle the installation of packages and their dependencies.

3. Install Gulp: **npm install -g gulp**

Gulp is a task runner that automates time-consuming tasks in development workflow. For example, performing Javascript minification; Move files to different directories; Translate LESS to CSS;

4. Install JHipster: **npm install -g generator-jhipster**

The generator-jhipster is a Yeoman generator developed by JHipster team. This tool allows the generation of applications written in Java for the back-end, and it generates AngularJS files for a rich client-side front-end. It also generates all the necessary infrastructure code such as Maven, Spring Boot, and Hibernate mappings.

After the installation of these required packages, it is possible to open a command-line and execute the following command: **yo jhipster**. The execution of this instruction will start the generator. This generation tool starts by prompting some questions about what are the desired modules to be included in the generated project. These questions are:

1. Which type of application would you like to create?
2. What is the base name of your application?
3. What is your default Java package name?
4. Which type of authentication would you like to use?
5. Which type of database would you like to use?
6. Which production database would you like to use?
7. Which development database would you like to use?
8. Do you want to use Hibernate 2nd level cache?
9. Do you want to use a search engine in your application?
10. Do you want to use clustered HTTP sessions?
11. Do you want to use WebSockets?
12. Would you like to use Maven or Gradle?
13. Would you like to use the LibSass stylesheet preprocessor for your CSS?
14. Would you like to enable translation support with Angular Translate?
15. Which testing frameworks would you like to use?

With these questions answered correctly, the generation tool starts generating all the necessary files for the chosen modules and according to the provided answers.

After this code generation, it was performed a shallow analysis of what was generated. One interesting file that was created by this tool was the **.yo-rc.json**. As illustrated in Listing 6.5, this JSON file contains the previously answered questions.

Listing 6.5: Configuration file: yo-rc.json

```
1 {
2   "generator-jhipster": {
3     "jhipsterVersion": "3.4.2",
4     "baseName": "generated-salesandservice",
5     "packageName": "com.msg.ib",
6     "packageFolder": "com/msg/ib",
7     "serverPort": "8080",
8     "authenticationType": "session",
9     "hibernateCache": "ehcache",
10    "clusteredHttpSession": "no",
11    "websocket": "no",
12    "databaseType": "sql",
13    "devDatabaseType": "mysql",
14    "prodDatabaseType": "mysql",
15    "searchEngine": "no",
16    "buildTool": "maven",
17    "enableSocialSignIn": false,
18    "rememberMeKey": "2d7719b7273d815b6bffb03dee87cefe2a1f5367",
19    "useSass": false,
20    "applicationType": "monolith",
21    "testFrameworks": [
22      "gatling"
23    ],
24    "jhiPrefix": "jhi",
25    "enableTranslation": true,
26    "nativeLanguage": "en",
27    "languages": [
28      "en",
29      "nl",
30      "fr",
31      "de",
32      "pl",
33      "pt-pt",
34      "es"
35    ]
36  }
37 }
```

The first time the generator was executed it took about 5 minutes to answer all of these questions because they were carefully interpreted to avoid mistakes and unnecessary code generation. It was found that the generator has generated the file **.yo-rc.json** and this file is containing all the answered questions. Motivated by this question answering being such repetitive and a time-consuming process, it was tested if the **yo jhipster** command would first check for the existence of the **.yo-rc.json** file and therefore escaping all those questions.

Therefore, it was removed all the previous generated files except the **.yo-rc.json**. Then it was executed the **yo jhipster** command again. The result was that the questions were not asked at this time, and all the previous generated files and folders were regenerated without any problem. As a verdict, this has simplified the process of creating the application skeleton using *JHipster* with just executing a simple command-line instruction and without user interaction.

To streamline even further the application generation process using *JHipster* and other necessary tools that may be included in the routine to create the application, it was created a bash file named

generate_everything.bat. This bash file has the objective to provide an easier way to generate the application like if it was as simple as a push of a button. Therefore, this file will hold all the necessary commands and sub-commands that may be needed to generate the entire application.

6.3.4 Generating the Domain Model

Having the PM model completely interpreted (See: 6.3.2), and an application skeleton in place (See: 6.3.3), it is now possible to transform the interpreted model into a correct *JHipster* input format. So that *JHipster* engines can generate all the necessary: Java *Entities*; JPA annotations; *REST Services*; *AngularJS Routers, Views, Controllers*; and other utilities for the application such as *Logging, Authentication, and Testing Frameworks*.

As described at conception section: 6.2.3, the chosen *JHipster* input format to expose the intended domain model for the application is the JDL format, a DSL provided by *JHipster* team to express the entire domain model with a single instruction. Therefore, to transform the interpreted domain model from the *BO* hierarchical structure into the JDL format, it was created a set of Classes that represents a *JHipster Entity* as presented in Figure 6.6. Each built *JHipster Entity* will be translated into a JDL text format using *Velocity* engine: a template-driven code generator.

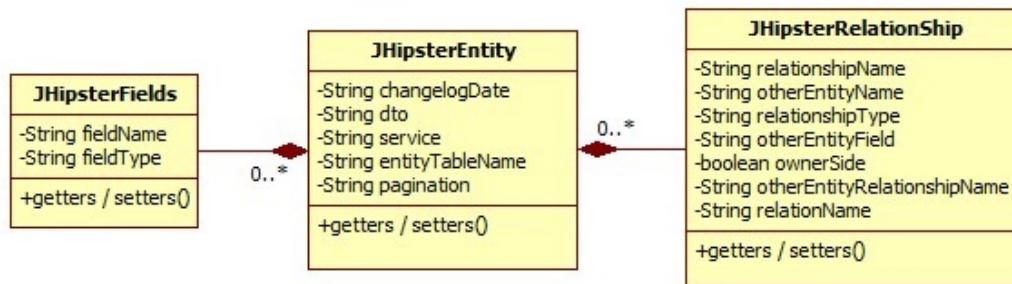


Figure 6.6: Class Diagram - JHipster Entity

The process to create these *JHipster Entities* is described in a diagram at Appendix E.1. As illustrated, each *BO* is translated into a *JHipster Entity* which is then sent to the *Velocity* engine using the *Velocity Context*. Then the context is merged with a *Velocity Template*, in which defines the layout of the generated text file and must follow the JDL format. Therefore, the listing 6.6 illustrates the developed template that follows the *JDL* structure described at Listing 5.1. The template illustrated at Listing 6.6 instructs *Velocity* framework to create the *JDL* file that will be describing the entire domain model, composed of all the necessary *Entities* and their relationships.

Listing 6.6: Velocity template for JDL format

```

#set( $startbrace = "{" )
#set( $endbrace = "}" )
/**
 * The @Auto-Generated $entity.entityTableName entity .
 */
entity $entity.entityTableName {
    #foreach( $field in $entity.fields )
        $field.fieldName $field.fieldType#if( $foreach.hasNext() ),#end
    #end
}
#foreach( $relationship in $entity.relationships )
    relationship $relationship.getUpperCamelRelationType() {
        $entity.entityTableName$startbrace$relationship.getRelationName()
        $endbrace to $relationship.otherEntityName$startbrace$relationship
        .otherEntityRelationshipName$endbrace
    }
#end

```

The result of *Velocity* engine is represented in Listing 6.7. Where it is possible to see the *CoverageBO* Entity as an example of a generated Entity that will make part of the final JDL file containing all Entities description. This file contains all the domain model and its relationships expressed under a format that will serve as input of the *hipster-entity* sub-generator so that it can generate these Entities on top of the already created/scaffolded application structure.

Listing 6.7: Example of CoverageBO entity exposed in JDL format

```

1  /**
2  * The @Auto-Generated CoverageBO entity .
3  */
4  entity CoverageBO {
5      max Double ,
6      groupConstraintType String ,
7      baseComponentFlag Boolean ,
8      coverageEffectiveDate ZonedDateTime ,
9      isSelected Boolean ,
10     coverageId String ,
11     min Double ,
12     excess Double ,
13     sumAssured Double ,
14     objectIdentifier String
15 }
16 relationship OneToMany {
17     CoverageBO{allAdjustmentBO} to AdjustmentBO{coverageBO}
18 }
19 relationship OneToMany {
20     CoverageBO{allPremiumComponentBO} to PremiumComponentBO{coverageBO}
21 }
22 relationship OneToMany {
23     CoverageBO{allRateComponentBO} to RateComponentBO{coverageBO}
24 }
25 relationship OneToMany {
26     CoverageBO{allRoleBO} to RoleBO{coverageBO}
27 }
28 relationship OneToMany {
29     CoverageBO{allPropertyBO} to PropertyBO{coverageBO}
30 }
31 relationship OneToMany {
32     CoverageBO{allCoverageBO} to CoverageBO{coverageBO}
33 }

```

6.3.5 Database

To persist data, *JHipster* natively supports the following database providers: *MySQL*, *MariaDB*, *PostgreSQL*, *MongoDB* or *Cassandra*. Using *Liquibase*, *JHipster* will be responsible for setting the correct database connector driver for the chosen database provider, and create the necessary files to map all database tables with the *Domain Entities*.

In this project, was used *MySQL* since it is a free and widely used database. Furthermore, this configuration can always be easily changed at file `.yo-rc.json` if necessary.

6.3.6 Application generation and initialization

To streamline the application generation and its initialization, it was produced some batch files to execute all the necessary sub-commands. The objective of these executable files is hiding the complexity of all necessary commands to get the application generated and running. Having such executables files, which hide all commands and its complexity, allow this tool to be used by non-technical users.

1. `generate_everything.bat`

This batch file will create a folder named: *Generated* where the entire application will get generated. Then it will execute the batch file *generate_entities.bat* that uses the developed PM introspection project to interpret the PM model and translate the domain model into a JDL format. Finally, having the JDL file describing the domain model under the generated folder, it will run the batch file *generate_application.bat* that will generate the application using *JHipster*.

2. `generate_entities.bat`

The Listing 6.8 is a batch file that starts by creating the new folder named *Generated* if this specific location does not exist in the system. Then, the *mvn clean compile* instruction is executed. This *Maven* command will make sure that the project located at the folder *entityGenerator* is compiling correctly. Then, is executed a *Maven Plugin* capable of executing *Java Main classes* from the command-line.

Listing 6.8: Generate Entites batch

```
1 echo Starting Entity Generator ...
2 call mkdir Generated
3 call mvn clean compile -f ./entityGenerator
4 call mvn exec:java -Dexec.mainClass="entityGenerator.JDLGenerator" -f ./entityGenerator
5 :EOF
```

Once the main class *JDLGenerator* finishes, all *Entities* related with the PM configured under the *pom.xml* file will be available, in the *Generated* folder, in a file named: *jdl-model.jh*.

3. generate_application.bat

Once exposed the domain model under a JDL format, it is possible to instruct *JHipster* to generate an application able to work with that domain model. Therefore, the Listing 6.9 illustrates the set of commands executed so that the application get generated.

Listing 6.9: Generate Application batch

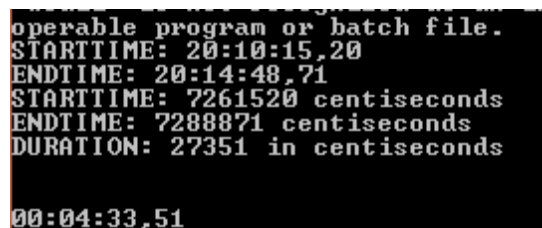
```
1 echo Starting Generate Application...
2 call mkdir Generated
3 call copy .yo-rc.json Generated\
4 call cd Generated
5 call yo jhipster
6 call yo jhipster:import-jdl jdl-model.jh --force
7 cmd /k
8 :EOF
```

Note that the file *.yo-rc.json* is already defined, and it is copied over to the *Generated* folder so that *JHipster* does not ask any questions about the intended system (See: 6.3.3). Then, it is just used the proper command so that *JHipster* knows that the domain model is described under the JDL syntax and at *jdl-model.jh* file.

4. run_generated_application.bat

Among the generated files it is possible to see a *pom.xml* which has a *Spring Boot* plugin to start the generated application. To launch the application is executed the command: **mvn spring-boot:run**.

As illustrated in Figure 6.7, the execution time of the batch file that generates the entire application is taking about 4 minutes and 33 seconds. It is important to notice that most of this time is being expended over the network, because most of the generated files use third party libraries that are stored in NPM remote repositories, such as *AngularJS*, *Bower*, *Gulp*, and many others libraries used in web-development.



```
operable program or batch file.
STARTTIME: 20:10:15,20
ENDTIME: 20:14:48,71
STARTTIME: 7261520 centiseconds
ENDTIME: 7288871 centiseconds
DURATION: 27351 in centiseconds
00:04:33,51
```

Figure 6.7: Code Generation Time

6.4 Validation

The following sub-sections will describe the validations about what has been generated when executed the batch files described in the previous sub-section. It will be reviewed if the generated code is composed by the necessary *Entities*, the correspondent database mappings, *REST Services*, and basic Front-End components for user interaction.

6.4.1 Generated Entities

The main goal of this project is the complete generation of an application able to communicate with PM models, specifically the P&C and *Life* models. As described in Section 1.4, to validate if the generated application follows the same domain model defined in PM; this section will evaluate if all necessary *Entities* were correctly generated for both models, following a prior interface enumeration and compare it with the generated *Entities*.

The validation plan for both PM models has followed the same approach. It was set up an *Eclipse Workspace*, and was created an empty *Maven Project*. Under the *pom.xml* was established the proper PM model dependency.

Once the dependency JAR gets available under the *Java Classpath*, it was possible to use *Eclipse IDE* to verify all the objects that are inherited from *BOAdaptable*; these are all the *Entities* that should have been generated in the application.

6.4.1.A P&C Model

By exploring the *BOAdaptable* inherited interfaces available on the P&C PM model, it is possible to verify that the domain model is composed of the following *Entities* illustrated in Table 6.1. Furthermore, to evaluate if each *Entity* has all its attributes generated, the table also presents the number of expected attributes per *Entity*.

Table 6.1: P&C Model - Entity Comparison : PM Model vs Generated Entities

| (a) PM Model | | | (b) Generated Entities | | |
|--------------|--------------------------------|-------------|------------------------|---------------------|-------------|
| # | Entity Name | N. of Attrs | # | Entity Name | N. of Attrs |
| 1 | BOAdaptableBusinessTransaction | 6 | 1 | BusinessTransaction | 4 |
| 2 | BOAdaptableFormBO | 19 | 2 | FormBO | 14 |
| 3 | BOAdaptablePremiumComponentBO | 13 | 3 | PremiumComponentBO | 9 |
| 4 | BOAdaptablePremiumDetailBO | 17 | 4 | PremiumDetailBO | 10 |
| 5 | BOAdaptableProductComponentBO | 23 | 5 | ProductComponentBO | 19 |
| 6 | BOAdaptablePropertyBO | 37 | 6 | PropertyBO | 26 |
| 7 | BOAdaptableProposalBO | 14 | 7 | ProposalBO | 12 |
| 8 | BOAdaptableQuoteBO | 30 | 8 | QuoteBO | 21 |
| 9 | BOAdaptableQuoteVersionBO | 9 | 9 | QuoteVersionBO | 6 |
| 10 | BOAdaptableRiskBO | 22 | 10 | RiskBO | 17 |
| 11 | BOAdaptableRiskDetailBO | 18 | 11 | RiskDetailBO | 13 |
| 12 | BOAdaptableRoleBO | 21 | 12 | RoleBO | 17 |
| 13 | BOAdaptableScheduledLocationBO | 7 | 13 | ScheduledLocationBO | 7 |
| 14 | BOAdaptableTextBO | 24 | 14 | TextBO | 16 |

As illustrated by the previous tables, all necessary *Entities* were generated for P&C model. However,

there is a gap between the generated attributes and those that compose the PM model. This happens because the model can have three different types of attributes:

1. **Java Standard Attributes:** These are Java native types, e.g. boolean, int, double, char. Or even objects supported by the Java Virtual Machine (JVM). For example Boolean, Double, String.
2. **Non Java Standard Attributes - PM Specific Types:** These are *Value Objects* available in the PM model artifact. Also known as *DiscreteValues*, these types are composed of three Strings: acronym, name, and shortText (see PM Key Type at: 3.2.1).
3. **Domain Model Attributes:** These are the relationships with other *Entities* within the domain model.

Due to the existence of non-standard types, the application presents differences between the expected attributes and those that were generated by *JHipster*. Those differences are related to the PM specific types, where the model introspection tool logs these *Types* as a warning message, since *JHipster* only supports the following *Types*: String; Integer; Long; Float; Double; BigDecimal; LocalDate; ZonedDateTime; Boolean; Enum; Blob.

When possible, the model introspection tool is adapting non-supported standard types, like Calendar into a supported type, in this case, it generates an attribute as being ZonedDateTime instead.

To address the gap between the number of attributes being created by *JHipster*, the Section 7.3 describes the development of a *JHipster Module* that will hook the *jhipster-entity* generator and add additional behavior so that the generated resources can have the necessary *PM Discrete Values* getting generated along with each *Entity*.

6.4.1.B Life Model

Similarly to the preceding sub-section 6.4.1.A, it was analyzed the generated *Entities* using the *Life* model. Table 6.2 presents a comparison between the expected result and the generated *Entities* and its attributes.

In line with the reported results using P&C model, using *Life* model were also found differences in the number of generated attributes, which will be addressed in Section 7.3.

When using *Life* model, some *Entities* were not generated according to the inherited objects of the *BOAdaptable* interface. This is, in fact, normal because the introspection tool is ignoring *Entities* that do not relate with the *Domain Model*, i.e. navigating from the root Object found in the model, the *Entity* is not visitable (see the paragraph at 6.3.2). For this reason, the *Entity* generation is pointless since the *Runtime Services* will not use those objects, neither the generated application need to store them.

Table 6.2: Life Model - Entity Comparison : PM Model vs Generated Entities

| (a) PM Model | | | (b) Generated Entities | | |
|--------------|---------------------------------------|-------------|------------------------|----------------------------|-------------|
| # | Entity Name | N. of Attrs | # | Entity Name | N. of Attrs |
| 1 | BOAdaptableAddressBO | 14 | 1 | AddressBO | 11 |
| 2 | BOAdaptableAdjustmentBO | 13 | 2 | AdjustmentBO | 11 |
| 3 | BOAdaptableBusinessTransaction | 6 | 3 | BusinessTransaction | 5 |
| 4 | BOAdaptableContractBO | 41 | 4 | ContractBO | 28 |
| 5 | BOAdaptableCoverageBO | 21 | 5 | CoverageBO | 18 |
| 6 | BOAdaptableDatePeriod | 4 | 6 | N/A | N/A |
| 7 | BOAdaptableDocumentBO | 10 | 7 | DocumentBO | 6 |
| 8 | BOAdaptableEmailBO | 5 | 8 | BOAdaptableEmailBO | 5 |
| 9 | BOAdaptableExistingCoverageBO | 6 | 9 | ExistingCoverageBO | 5 |
| 10 | BOAdaptableExperienceBO | 26 | 10 | ExperienceBO | 26 |
| 11 | BOAdaptableExperienceDetailBO | 22 | 11 | ExperienceDetailBO | 22 |
| 12 | BOAdaptableIllustrationBO | 19 | 12 | IllustrationBO | 14 |
| 13 | BOAdaptableIncomeBO | 6 | 13 | IncomeBO | 6 |
| 14 | BOAdaptableIndividualBO | 24 | 14 | IndividualBO | 20 |
| 15 | BOAdaptableInsuredPersonBO | 19 | 15 | InsuredPersonBO | 14 |
| 16 | BOAdaptableMultiLifePartnerBO | 11 | 16 | MultiLifePartnerBO | 9 |
| 17 | BOAdaptableOrganizationBO | 8 | 17 | OrganizationBO | 7 |
| 18 | BOAdaptablePackageBO | 14 | 18 | PackageBO | 12 |
| 19 | BOAdaptablePartyBO | 9 | 19 | PartyBO | 9 |
| 20 | BOAdaptablePeriod | 4 | 20 | N/A | N/A |
| 21 | BOAdaptablePerpetual | 2 | 21 | N/A | N/A |
| 22 | BOAdaptablePhoneBO | 13 | 22 | PhoneBO | 11 |
| 23 | BOAdaptablePremiumComponentBO | 14 | 23 | PremiumComponentBO | 8 |
| 24 | BOAdaptablePremiumDetailBO | 9 | 24 | PremiumDetailBO | 7 |
| 25 | BOAdaptablePremiumDetailVector | 5 | 25 | PremiumDetailVector | 5 |
| 26 | BOAdaptablePremiumDetailVectorElement | 6 | 26 | PremiumDetailVectorElement | 6 |
| 27 | BOAdaptableProducerBO | 14 | 27 | ProducerBO | 11 |
| 28 | BOAdaptableProducerOrganizationBO | 5 | 28 | ProducerOrganizationBO | 5 |
| 29 | BOAdaptablePropertyBO | 21 | 29 | PropertyBO | 16 |
| 30 | BOAdaptablePropertyVector | 7 | 30 | PropertyVector | 7 |
| 31 | BOAdaptablePropertyVectorElement | 9 | 31 | PropertyVectorElement | 9 |
| 32 | BOAdaptableRateComponentBO | 9 | 32 | RateComponentBO | 7 |
| 33 | BOAdaptableRateDetailBO | 8 | 33 | RateDetailBO | 7 |
| 34 | BOAdaptableRateDetailVector | 7 | 34 | RateDetailVector | 7 |
| 35 | BOAdaptableRateDetailVectorElement | 6 | 35 | RateDetailVectorElement | 6 |
| 36 | BOAdaptableRoleBO | 12 | 36 | RoleBO | 10 |
| 37 | BOAdaptableUserBO | 6 | 37 | UserBO | 5 |

Legend:
N/A: *Entity* not generated.

6.4.2 Generated REST Services

JHipster generates a *REST Service Layer* where each generated *Entity* receives a *Resource Class* providing basic CRUD operations. Each *Entity Resource* is composed by the respective *Entity Repository*, a compliant JPA Object that is responsible for accessing the database layer and transport data between the generated application and the database.

Taking the *IllustrationBO Entity* from the *Life* PM model as an example, the *REST Services* generated for this entity are illustrated in Table 6.3:

Table 6.3: Generated REST Services : Resource for IllustrationBO Entity

| Method | URL | Description |
|--------|----------------------------|--|
| POST | /api/illustration-bos | Creates a new IllustrationBO |
| PUT | /api/illustration-bos | Update an existing IllustrationBO |
| GET | /api/illustration-bos/{id} | Gets an existing IllustrationBO by its ID |
| DELETE | /api/illustration-bos/{id} | Removes an existing IllustrationBO by its ID |

As anticipated, these *REST Services* are minimalistic regarding its functionalities, where only CRUD operations are available. Although this satisfies the project aims, these *Services* could be improved by adding another *JHipster Custom Module* to add additional *Services*. For example to access or execute the *PM Runtime Services*:

1. **Information Service:** Get and merge *Product Components* related to a given Illustration *Entity*;
2. **Domain Service:** Update values that may no longer be compliant with PM rules and formulas i.e. upon user changes, the *Product* may not be in sync with PM.

6.4.3 Generated Database Mappings

Along with the generated *Entities*, *JHipster* has also created the necessary database mappings using JPA annotations in each *Entity* class. Furthermore, using *LiquidBase* files, the application will create the necessary database tables upon application startup.

The database tables are created due to the file *master.xml*. This file aggregates all the *LiquidBase* partial files that instruct a database to create the necessary tables and gives its consistency through the usage of constraints.

The listing 6.10 illustrates the *master.xml* file that load all the partial XML files responsible for creating a DB table per generated *Entity*.

Listing 6.10: Database - tables initialization

```

1  <databaseChangeLog>
2    <include file="classpath:00000000000000_initial_schema.xml"/>
3    <include file="classpath:20160702185114_added_entity_BusinessTransaction.xml"/>
4    <include file="classpath:20160702185115_added_entity_ProposalB0.xml"/>
5    <include file="classpath:20160702185116_added_entity_QuoteVersionB0.xml"/>
6    <include file="classpath:20160702185117_added_entity_ScheduledLocationB0.xml"/>
7    <include file="classpath:20160702185118_added_entity_QuoteB0.xml"/>
8    <include file="classpath:20160702185119_added_entity_ProductComponentB0.xml"/>
9    <include file="classpath:20160702185120_added_entity_FormB0.xml"/>
10   <include file="classpath:20160702185121_added_entity_RoleB0.xml"/>
11   <include file="classpath:20160702185122_added_entity_PremiumComponentB0.xml"/>
12   <include file="classpath:20160702185123_added_entity_RiskB0.xml"/>
13   <include file="classpath:20160702185124_added_entity_PremiumDetailB0.xml"/>
14   <include file="classpath:20160702185125_added_entity_RiskDetailB0.xml"/>
15   <include file="classpath:20160702185126_added_entity_TextB0.xml"/>
16   <include file="classpath:20160702185127_added_entity_PropertyB0.xml"/>
17 </databaseChangeLog>

```

One important detail discerned during the development of this project was that the order of the *Entities* described under the JDL file is important. Because *JHipster* will create this *master.xml* file using the same order declared in the JDL file. Therefore, it will be also the execution order of the

Structured Query Language (SQL) statements that will create the *Database Schema* for the generated application.

Due to the declared constraints in the SQL statements (to enforce the relationships and provide schema consistency), it is important that each table that contains those constraints have all its dependent tables already created. Otherwise, the generated application won't start properly, and an exception will be thrown to the *System Output* since the Database Management System (DBMS) was not able to assign those constraints.

As an example, Listing 6.11 illustrates the *Liquidbase* file that will create the *QuoteBO* table in the database. This file contains all the necessary columns so that the *Entity QuoteBO* gets stored in a database. Moreover, the file presents two types of constraints: field constraints and foreign-key constraints. As explained in the previous paragraph, it is imperative that the tables that hold the related *Foreign Key Constraints* already exist in the database. In this particular case, it is expected that the *proposal_bo* and the *business_transaction* tables already exist in the database.

Listing 6.11: Liquidbase changeset - create QuoteBO

```
1 <changeSet id="20160702185118" author="jhipster">
2   <createTable tableName="quote_bo">
3     <column name="id" type="bigint" autoIncrement="${autoIncrement}">
4       <constraints primaryKey="true" nullable="false"/>
5     </column>
6     <column name="number" type="integer">
7       <constraints nullable="true" />
8     </column>
9     (... other columns ...)
10    <column name="business_transaction_id" type="bigint">
11      <constraints unique="true"/>
12    </column>
13    <column name="proposal_bo_id" type="bigint"/>
14  </createTable>
15  <addForeignKeyConstraint baseColumnNames="business_transaction_id"
16    baseTableName="quote_bo"
17    constraintName="fk_quotebo_buinesstransaction_id"
18    referencedColumnNames="id"
19    referencedTableName="business_transaction"/>
20  <addForeignKeyConstraint baseColumnNames="proposal_bo_id"
21    baseTableName="quote_bo"
22    constraintName="fk_quoteb_proposalbo_id"
23    referencedColumnNames="id"
24    referencedTableName="proposal_bo"/>
25 </changeSet>
```

Since the *QuoteBO* appears after both dependents, i.e. *BusinessTransaction* and *ProposalBO*, when the system processes the *master.xml* file (at Listing 6.10), the tables will be correctly defined and created in the database.

6.4.4 Generated Database Tables

The generated application holds a configuration to establish a database connection. Since the *.yo-rc.json* file has the database provider configured to be *MySQL*, *JHipster* has generated the correct configuration to run the generated application using a *MySQL* database.

The Listing 6.12 exhibit the *Liquibase Maven Plugin* that is responsible for using the *master.xml* file explained in the previous section and consequently, establishes the connection with the database.

Listing 6.12: Database connection configuration

```
1 <plugin>
2   <groupId>org.liquibase</groupId>
3   <artifactId>liquibase-maven-plugin</artifactId>
4   <version>${liquibase.version}</version>
5   <configuration>
6     <changeLogFile>src/main/resources/liquibase/master.xml</changeLogFile>
7     <driver>com.mysql.jdbc.Driver</driver>
8     <url>jdbc:mysql://localhost:3306/generated-salesandservice</url>
9     <defaultSchemaName>generated-salesandservice</defaultSchemaName>
10    <username>root</username>
11    <password></password>
12    <verbose>true</verbose>
13    <logging>debug</logging>
14  </configuration>
15 </plugin>
```

The generated application was named: *generated-salesandservice* at *.yo-rc.json*. Therefore the database schema will also have the same name. Additionally, to validate if the application creates all the necessary tables in a *MySQL* database, it was installed the *MySQL Workbench*, and created a database server instance that runs by default in localhost on port 3306. Then, it was created a new schema named *generated-salesandservice*. Finally, the generated application was launched using the maven command: *mvn spring-boot:run*.

Upon seeing the following log illustrated in Listing 6.13, the application has started and is available to get accessed using any Browser.

Listing 6.13: Application running log

```
1 -----
2 Application 'generated-salesandservice' is running! Access URLs:
3 Local:      http://127.0.0.1:8080
4 External:   http://192.168.56.1:8080
5 -----
```

At this point, it is expected that the database holds several tables about the existing domain model. To check if the database contains all the domain model entities, it was compared the generated *Entities*

against the database tables that were generated at application start up. Getting the P&C model as an example, Table 6.4 illustrates the generated tables in the database using the *MySQL Workbench*.

Table 6.4: Generated Database Tables

| Entity | | Database Table | |
|---------------------|--------------------|-----------------------|-----------------|
| Name | Num. of attributes | Name | Num. of columns |
| BusinessTransaction | 4 | business.transaction | 4 |
| FormBO | 14 | form_bo | 14 |
| PremiumComponentBO | 9 | premium_component_bo | 8 |
| PremiumDetailBO | 10 | premium_detail_bo | 10 |
| ProductComponentBO | 19 | product_component_bo | 13 |
| PropertyBO | 26 | property_bo | 26 |
| ProposalBO | 12 | proposal_bo | 9 |
| QuoteBO | 21 | quote_bo | 15 |
| QuoteVersionBO | 6 | quote_version_bo | 2 |
| RiskBO | 17 | risk_bo | 13 |
| RiskDetailBO | 13 | risk_detail_bo | 11 |
| RoleBO | 17 | role_bo | 13 |
| ScheduledLocationBO | 7 | scheduled_location_bo | 7 |
| TextBO | 16 | text_bo | 16 |

According to the Table 6.4, it is possible to verify that not every existing attribute in the *Java Entity object* had a correspondent column in the database. Therefore, this was targeted for further investigation, since this missing attribute column could provide data losses to the application. It was then noticed that this is a normal scenario when there are *ONE-TO-MANY* relationships. For example, in the database table `premium_component_bo`, the missing column was the reference to a collection of "allPremiumDetailBOs". And for the `product_component_bo` the suspicious missing columns were: "allPropertyBOs", "allFormBOs", "allTextBOs", "allRoleBOs", "allProductComponentBOs", "allPremiumComponentBOs".

It was then concluded that the fact of those tables not having those foreign key references, is correct and makes sense. Because the table that holds the reference id to establish the relationship, for example, between `PremiumComponentBO` and `PremiumComponentsDetailsBO`, is the `PremiumComponentDetailsBO` table and not the `PremiumComponentBO`.

6.4.5 Generated Client-Side

JHipster has generated a Front-End using *AngularJS*, a popular javascript framework developed by Google to build a SPA. *AngularJS* implements the architectural pattern Model View Controller (MVC) and allows the construction of complex yet organized javascript scripting language.

6.4.5.A Generated Functionalities

Apart from the *Domain Model*, *JHipster* generates various advantageous features that are only accessible using a user that have administration action rights. The generated utility features were:

1. **User Management** - To create new users, set or change user roles (action rights). By default, the generator has initialized two distinct user roles, *ROLE_USER* and *ROLE_ADMIN*.
2. **Metrics** - An application profiling tool that allows the visualization of the current application status. Figure 6.8 shows that is possible to visualize what are the system load, how many memory is being consumed by the system, and how many *Threads* are running.

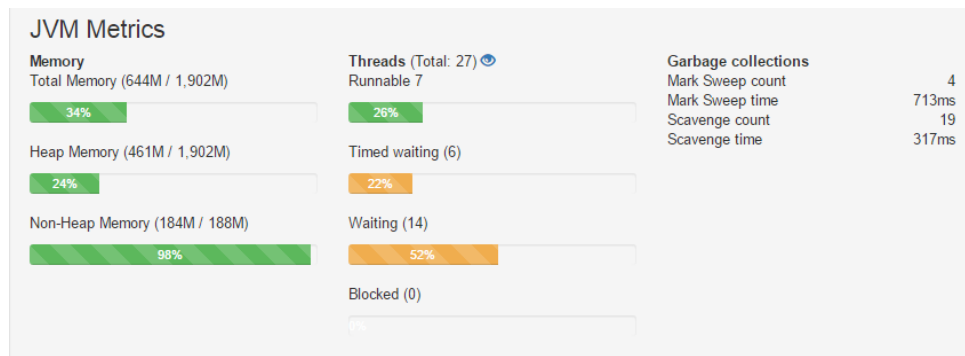


Figure 6.8: System Metrics

In this *Metrics* section, it is also possible to observe the *HTTP requests* and events per second; Get *Service statistics* - how many hits are getting a *Service* or a *Repository*; The *Cache* usage, i.e. how many data records were retrieved from cache mechanism instead of going to the DB.

3. **System Health** - Under the system health, its possible to observe the availability of system resources, such as *Database* and *Disk Space*. To test this feature, it was tried to switch off the database to verify if the application reports back if the database was not available. As illustrated in Figure 6.9, the application did notify the administrator that the connection between the application and the database was not established at that time.

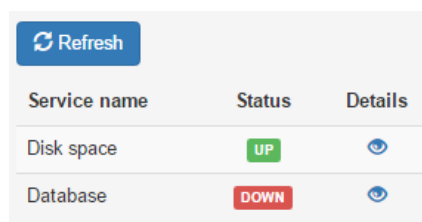


Figure 6.9: Application Health Check

4. **System Audits** - This feature provides access monitorization. The system will audit the login accesses date, the username, and from which Internet Protocol (IP) address the system was accessed.
5. **Logging** - Allows the configuration of logging without the need to stop and start the application.
6. **REST API** - *JHipster* uses *Swagger*, a framework for API's. This feature exposes all the existing REST Services in the *Backend*, allowing a proper and maintainable API documentation. Figure 6.10 shows an example of the available API to access information about the *Resources: Proposal, Quote, and Quote Version*.

| proposal-bo-resource : Proposal Bo Resource | | | Show/Hide | List Operations | Expand Operations |
|---|-----------------------------|--|-----------|-----------------|-----------------------|
| GET | /api/proposal-bos | | | | getAllProposalBos |
| POST | /api/proposal-bos | | | | createProposalBo |
| PUT | /api/proposal-bos | | | | updateProposalBo |
| DELETE | /api/proposal-bos/{id} | | | | deleteProposalBo |
| GET | /api/proposal-bos/{id} | | | | getProposalBo |
| quote-bo-resource : Quote Bo Resource | | | Show/Hide | List Operations | Expand Operations |
| GET | /api/quote-bos | | | | getAllQuoteBos |
| POST | /api/quote-bos | | | | createQuoteBo |
| PUT | /api/quote-bos | | | | updateQuoteBo |
| DELETE | /api/quote-bos/{id} | | | | deleteQuoteBo |
| GET | /api/quote-bos/{id} | | | | getQuoteBo |
| quote-version-bo-resource : Quote Version Bo Resource | | | Show/Hide | List Operations | Expand Operations |
| GET | /api/quote-version-bos | | | | getAllQuoteVersionBos |
| POST | /api/quote-version-bos | | | | createQuoteVersionBo |
| PUT | /api/quote-version-bos | | | | updateQuoteVersionBo |
| DELETE | /api/quote-version-bos/{id} | | | | deleteQuoteVersionBo |
| GET | /api/quote-version-bos/{id} | | | | getQuoteVersionBo |

Figure 6.10: System Metrics

6.4.5.B Interacting with application domain model

As expected, as far as functionalities are concerned the generated Front-End components to allow user interactions with the application *Domain Model* are basic. As illustrated in Figure 6.11, each system *Entity* gets an entry on the *dropdown* menu. There is also an angular router allowing the user to access the correspondent entity view. This view uses a tabular layout to present the data. Additionally, it presents a button to create a new entity of the same type.

Moreover, to create new *Entities*, the user is presented with a modal screen as illustrated in Figure 6.12. This user interface component contains all the *Entity* related attributes. Here, the user can cancel his action or create the entity instance with the provided input.

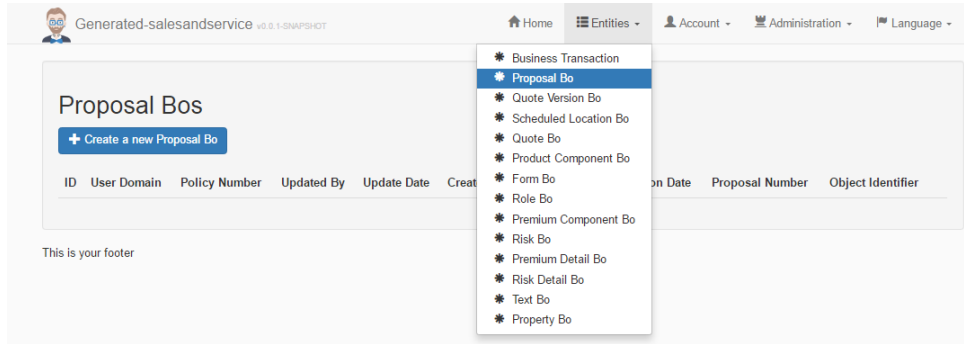


Figure 6.11: Managing Domain Entities

Figure 6.12: Creating a Domain Entity

It is also possible to establish relationships using these generated front-end components. If a *Proposal* already exists, the user can create a new *Quote* and associate it with that existing *Proposal*. Furthermore, the user can open *Quotes* from a *Proposal*, since there will be a link reference to its *Quote*.

6.5 Conclusions

As a conclusion of this development process, where it was applied the *JHipster* generator to scaffold the entire application infrastructure, the generated application is following the model described by the PM model. Here the developed introspection tool is working accordingly, and all the relevant *Entities* for a given PM domain model got generated correctly. However, there is still a known drawback related

with the PM *Discrete Values* that will be addressed in Section 7.3.

Regarding the generated *REST Services*, these were also generated correctly. Where the application got basic CRUD operations that are being used by the client-side code to create, edit and remove any generated *Entity* in the system.

The database related resources were also correctly generated, where the usage of JPA annotations and *Liquidbase* mappings has enabled the persistence layer for the generated application.

As seen, *JHipster* has generated almost everything related to the application infrastructure which takes serious time of development when manually done. This entire application generation using automated processes is a great achievement. However, to establish communication with *PM Runtime Services* this is not enough since the generated domain model does not implement the *interfaces* nor have the necessary *Key Types* associated with the generated *Entities*. Therefore, the next section will describe the custom components created to enable communications between the generated application and the PM model.

7

JHipster - Custom Extensions

Contents

| | | |
|-----|--|----|
| 7.1 | JHipster Modules | 90 |
| 7.2 | Inject PM Model Dependency | 90 |
| 7.3 | Applying BOAdaptable to the Generated Entities | 92 |
| 7.4 | Validation | 94 |
| 7.5 | Conclusions | 94 |

It was identified in the previous section that the generated *Entities* do not have the necessary attributes to establish communication with PM *Runtime Services*. One of the identified causes was that most of the *Entities* have to have some *Discrete Values* that are PM specific objects. Therefore *JHipster* does not recognize those objects to generate the necessary infrastructure for those unknown attributes.

This chapter describes how all the generated *Entities* were improved so that they can hold not only Java standard types but all the necessary and relevant attributes in the domain model to enable PM to recognize them, using the model *interfaces*.

Figure 7.1 illustrates an overview of the evolved components in this project. In this section, it will be particularly described the "PM Dependency Injector" and the "BO Adaptables" which were developed as being new *JHipster Modules*.

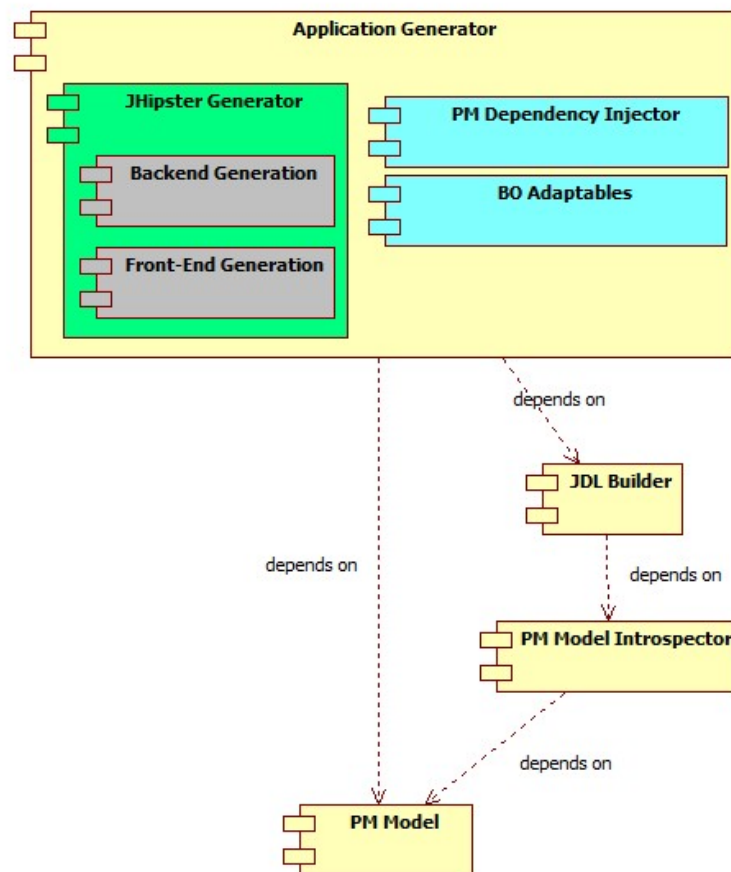


Figure 7.1: Overall Project Overview

Therefore, the following sections describe the *JHipster Modules* that were developed to add customization on top of the base *JHipster entity sub-generator* and together, generate *Entities* that hold all the necessary attributes, interfaces, and relationships to establish communications with the PM *Runtime Services*.

7.1 JHipster Modules

As described, *JHipster* uses *Yeoman* generator, which provides composability for sub-generators.

”The `composeWith` method allows the generator to run side-by-side with another generator (or sub-generator). That way it can use features from the other generator instead of having to do it all by itself.” *Yeoman* (2016)

JHipster has used the *Yeoman* composability to allow other generators being executed along with the main *JHipster entity sub-generator*, where the context and variables are shared between the generator modules. Due to this variable and context sharing mechanism, it is possible to add customizations on top of what has been generated in the first place, by the main *JHipster entity sub-generator*.

That being said, to have multiple generators running side-by-side it is needed to add the module to *JHipster* hook system, where *JHipster* calls certain hooks before and after some of its tasks such as:

- Pre App creation hook [planned]
- Post App creation hook [planned]
- Post Entity creation hook
- Pre Entity creation hook [planned]

To create a *JHipster Module*, its necessary to follow some rules established by *JHipster* and *Yeoman*, where it should follow *Yeoman* extension rules and should not stop the module chain by doing *process.exit* in the developed module. However, this process was streamlined with another existing *JHipster Module*. The sub-generator named: *generator-jhipster-module* allows the creation of a skeleton for new modules, leaving these creation rules being a concern of the *generator-jhipster-module*. Consequently, using this handy tool, it was created the skeleton for all additional modules that are needed to enhance the generated *Entities*.

7.2 Inject PM Model Dependency

The generated *Entities* must be composed not only of Java standard types and domain model relationships but also PM specific types such as *GroupConstraintType*, *GroupConstraintOperationType*, *PremiumDetailType* among many other objects related to the PM model that needs to be recognized by the generated domain model.

Therefore, to recognize all the possible types from the PM model is necessary that the JAR gets available in the generated application classpath. Since the generated application is also a *Maven* project, it has a *pom.xml* file that establishes all the project build process including its dependencies, where these *Maven* capabilities were used to resolve dependencies and add the given PM model into the generated application classpath. Consequently, the *PM Model Introspector* tool was improved to write down which

PM was used to generate the domain model, where a new template file was created. This file is represented in Listing 7.1 and stand as a template for a *Maven Dependency* that will be later included in the generated pom.xml. The template is modified by the *PM Model Introspection* tool using a regular expression substitution technique, generating its final state as illustrated in Listing: 7.3

Listing 7.1: pm-dependency template

```
1 <dependency>
2   <groupId>{0}</groupId>
3   <artifactId>{1}</artifactId>
4   <version>{2}</version>
5 </dependency>
```

Listing 7.2: pm-dependency template result

```
1 <dependency>
2   <groupId>com.fja</groupId>
3   <artifactId>aais.pm.pandc</artifactId>
4   <version>1.1.36.DEVO</version>
5 </dependency>
```

After *PM Model Introspection* tool generates the XML file that holds the correct dependency, its necessary to set the *PM Dependency Injector Module* being one *Post App creation hook*. This is accomplished by having the following code in the PM Injector module:

Listing 7.3: Hooking a JHipster module

```
1 jhipsterFunc.registerModule("jhipster-pminjector", "app", "post", "app");
```

This instruction will generate an additional **.jhipster** folder and a **jhi-hooks.json** file represented in Listing: 7.4, This is interpreted by *JHipster* when the entity sub-generator is running. The *hookFor* and *hookType* designate what is the correct moment to another module starts its execution.

Listing 7.4: JHipster hook configuration

```
1 [
2   {
3     "name": "Jhipster Pminjector generator",
4     "npmPackageName": "jhipster - pminjector",
5     "description": "A JHipster module to generate Jhipster Pminjector",
6     "hookFor": "app",
7     "hookType": "post",
8     "generatorCallback": "jhipster - pminjector:app"
9   }
10 ]
```

At this point, the batch file that was starting the code generation was also adjusted as illustrated in Listing 7.5 so that the new module is executed first, getting registered in the **jhi-hook.json**. Then, when *JHipster* is executed the hook is already set, and consequently, the module will be called by *JHipster* according to the configured task step.

Listing 7.5: Generate Application batch with pm dependency injector

```
1 echo Starting Generate Application...
2 call mkdir Generated
3 call copy .yo-rc.json Generated\
4 call cd Generated
5 call yo jhipster-pminjector
6 call yo jhipster
7 call yo jhipster:import-jdl jdl-model.jh --force
8 cmd /k
9 :EOF
```

Listing 7.6 illustrates the source-code of the PM dependency injector, which is responsible for adding the `pm-dependency.xml` previously generated by the introspection tool and will add it to the `pom.xml` file of the generated application by *JHipster*.

Listing 7.6: PM Injector source-code

```
1 var file = "pom.xml";
2 var pmDependencyStr = fs.readFileSync("pm-dependency.xml", { encoding : 'UTF-8' });
3 var pmDependency = new dom().parseFromString(pmDependencyStr);
4
5 fs.readFile(file, 'utf8', function (err, data) {
6     if (err) throw err;
7
8     // Create an XMLDom Element:
9     var doc = new dom().parseFromString(data);
10    // Parse XML with XPath:
11    var dependencies = doc.getElementsByTagName('dependencies')[0];
12    var firstDependency = dependencies.getElementsByTagName("dependency")[0];
13    dependencies.insertBefore(pmDependency, firstDependency);
14
15    var prettyXML = prettyData.xml(doc.toString());
16    fs.writeFile (file, prettyXML, function(err) {
17        if (err) throw err;
18        this.log('End of pminjector generator');
19    }).bind(this)
20    );
21 }.bind(this));
22
23 this.log('\n' + chalk.bold.green('pminjector done'));
```

When the batch file is once again executed so that the application gets regenerated, it is possible to see that the dependency for the PM is now included as a dependency in the `pom.xml` of the generated application. Consequently, the JAR is now in the classpath, and the application can now access and recognize the existence of all PM types available in the JAR. The next step is to improve each generated *Entity* so that all *Entities* implement the correspondent *interface* available in PM.

7.3 Applying BOAdaptable to the Generated Entities

For an *Entity* to be recognized by PM on its *Runtime Services*, each *Entity* must implement the related *interface* that will establish a concrete agreement for all the necessary attribute getters so that

PM can infer about the domain model and its value state in the downstream systems, i.e. the generated application.

The *BOAdaptables* is a *JHipster* module that will add the *"implements"* reference to each generated *Entity* to the co-related interface. Additionally, all the getters in the *Entity* will have to match all method signatures declared in the related interface to avoid compilation errors.

To accomplish this, the introspection tool was once again improved to generate an additional file named *additionalFields.json*. For each *Entity*, this file describes what are the non-standard Java types that must be included, and also the *BOAdaptable* interface that the *Entity* must be implementing.

Listing 7.7 illustrates an example of an *Entity* that needs to have additional fields other than the Java standard types that were already handled by *JHipster* upon the interpretation of the JDL. At the left-side of the illustrated JSON, is possible to identify what are the additional fields that need to be part of the *Entity*. The *BOAdaptable* field is not an additional field neither a new getter to be generated, but it is the *interface* that each *Entity* must be implementing. There is also Boolean types, because *JHipster* is generating a method named as *"public Boolean isIsChange()"* while PM expects to find a *"public Boolean getIsChanged()"*. For this reason, a new getter method was added to match the PM signature where it is reusing the same attribute *"isChanged"* from the *Entity*.

Listing 7.7: Additional fields

```

1  "RiskDetailBo" : {
2    "IsChanged:isChanged" : "java.lang.Boolean",
3    "AllPropertyBO:allPropertyBo" : "[Lcom.fja.pm.aais.model.BOAdaptablePropertyBO;",
4    "categoryOverride" : "com.fja.pm.aais.vo.Category",
5    "riskDetailCategory" : "com.fja.pm.aais.vo.Category",
6    "groupConstraintOperationType" : "com.fja.pm.aais.vo.GroupConstraintOperationType",
7    "groupConstraintType" : "com.fja.pm.aais.vo.GroupConstraintType",
8    "AllTextBO:allTextBo" : "[Lcom.fja.pm.aais.model.BOAdaptableTextBO;",
9    "riskDetailType" : "com.fja.pm.aais.vo.RiskDetailType",
10   "riskDetailClass" : "com.fja.pm.aais.vo.RiskDetailClass",
11   "IsSelected:selected" : "java.lang.Boolean",
12   "BOAdaptable" : "com.fja.pm.aais.model.BOAdaptableRiskDetailBO",
13   "RiskBO:riskBo" : "com.fja.pm.aais.model.BOAdaptableRiskBO"
14 },
15 "OtherEntities...": {
16 }

```

With this information stored in the *additionalFields.json* file, it was possible to develop the *BOAdaptables JHipster* module that will be hooked to the Post Entity hook. Which means that right after *JHipster* finishes the generation of a single *Entity* it will call this *BOAdaptables* module that in turn, will add all of these additional customizations to the generated *Entity*.

As a result, all the *Entities* have now all the necessary getters so that PM can access using the established *interface*, making the generated domain model ready to communicate with the *PM Runtime Services*.

7.4 Validation

The code generation was executed once again, to validate the function of these two modules. As a result, it was opened the generated project using *Eclipse IDE* where it was possible to open all the PM types that are not part of Java. Therefore, the PM dependency injector has set the PM dependency correctly in its pom file.

Additionally, it was confirmed that the *Eclipse Markers* did not identify any compilation error in the source-code. Furthermore, it was manually revised each generated *Entity* in which was validated that all the *Entities* were implementing the correct PM *interface*, and since there were no compilation errors in the source-code, it means that all methods correctly followed the interface signatures established by the PM model.

7.5 Conclusions

Having all the generated *Entities* implementing correctly the related PM *interface* and having all the relationships between all the domain model objects established, it is concluded that the generated application has a domain model fully compatible with the provided PM and its *Runtime Services*.

8

Experiments

Contents

| | | |
|-----|--|----|
| 8.1 | Generation of Multiple Software Layers | 97 |
| 8.2 | Generation Time | 98 |

At Chapter 1.4 - *Hypothesis test*, was enumerated some tests that should be done against the proposed solution. This chapter describes the executed tests and validations previously identified in chapter 1.4.

8.1 Generation of Multiple Software Layers

An application is composed of several software layers such as *Domain*, *Services*, *Business Logic*, *Data Access* among others infrastructure components that are described by distinct files. During the validation phase of this project it was analyzed how many files were generated by the proposed solution. Consequently, Table 8.1a and Table 8.1b illustrates the overall number of generated files and folders by the generator tool.

The proposed solution uses existing modules and dependencies from NPM. When the application is being generated from the first time, those third party dependencies are fetched from a remote repository so that, in generation time, all the dependencies are available. This network operation takes some time to accomplish; therefore Table 8.1a illustrates the number of created files on the host machine by this tool, even though some files are just dependencies. On the other hand, Table 8.1b illustrates only files that are related to the generated application.

Table 8.1: Number of Generated Files and Folders

| (a) With node modules | | | (b) Without node modules | | |
|-----------------------|---------|--------|--------------------------|---------|-------|
| Model | Folders | Files | Model | Folders | Files |
| AAIS | 4.572 | 32.207 | AAIS | 317 | 3.441 |
| LIFE | 4.606 | 32.812 | LIFE | 350 | 4.025 |

Additionally, it was used a bash command to investigate how many lines of code were generated per file type. Listing 8.1 holds the bash command used to get those metrics.

Listing 8.1: Bash command: count the generated application files

```

1 find src/ -name '*.<file-extention>' -type f -exec wc -l {} \;
2 | awk '{ sum += $1 } END { print sum }'
```

Using the previous bash command it was obtained the number of generated source-code lines for the most common files in the application such as Java, JavaScript, HTML, CSS, JSON and property files. The results are exposed in Table 8.2.

Table 8.2: Generated Source-code Lines

| Model | Java | Javascript | HTML | CSS | XML | JSON | properties | Total |
|-------|--------|------------|--------|-----|-------|--------|------------|---------------|
| AAIS | 18.674 | 10.004 | 6.896 | 337 | 2.088 | 12.042 | 264 | 50.305 |
| LIFE | 20.086 | 18.707 | 11.340 | 337 | 3.588 | 18.807 | 264 | 73.129 |

8.2 Generation Time

As described in this project *Hypothesis Test*, the proposed solution shall generate the entire application within the acceptable time of 15 minutes. During this project, was used two different PM artifacts that have different *Domain Models*. Therefore there are a different number of *Entities* to be generated.

To test the generation time, it was executed the code generation 20 times, and the results are presented in Table 8.3.

Table 8.3: Solution Generation Times

| (a) Generate AAIS Model | | (b) Generated LIFE Model | |
|-------------------------|-------------|--------------------------|-------------|
| # | Time | # | Time |
| 1 | 00:03:33,49 | 1 | 00:05:31,40 |
| 2 | 00:03:40,94 | 2 | 00:05:21,68 |
| 3 | 00:03:48,17 | 3 | 00:05:17,13 |
| 4 | 00:03:57,65 | 4 | 00:05:20,49 |
| 5 | 00:03:38,10 | 5 | 00:05:19,36 |
| 6 | 00:03:39,28 | 6 | 00:05:34,93 |
| 7 | 00:03:47,33 | 7 | 00:05:37,26 |
| 8 | 00:03:47,85 | 8 | 00:05:23,93 |
| 9 | 00:03:39,99 | 9 | 00:05:23,91 |
| 10 | 00:03:38,43 | 10 | 00:05:15,30 |
| 11 | 00:03:41,72 | 11 | 00:05:32,17 |
| 12 | 00:03:44,39 | 12 | 00:05:33,80 |
| 13 | 00:03:39,88 | 13 | 00:05:35,02 |
| 14 | 00:03:38,61 | 14 | 00:05:42,80 |
| 15 | 00:03:37,12 | 15 | 00:05:30,30 |
| 16 | 00:03:33,79 | 16 | 00:05:31,00 |
| 17 | 00:03:34,94 | 17 | 00:05:31,23 |
| 18 | 00:03:42,59 | 18 | 00:05:42,86 |
| 19 | 00:03:55,54 | 19 | 00:05:34,35 |
| 20 | 00:03:39,90 | 20 | 00:05:47,01 |

As demonstrated in the previous Table 8.3, depending on the model, the generation time is different. This different generation time is understandable since these models have a different amount of *Entities* to be generated. Using the *AAIS* model in which have 14 *Entities* to be generated, the generation tool takes in average 3 minutes and 42 seconds. Whereas *LIFE* model, having 37 *Entities* to be generated, takes about 5 minutes and 30 seconds.

9

Conclusions

Contents

| | | |
|-----|--|-----|
| 9.1 | Code Generation vs Current Manual Processes | 101 |
| 9.2 | Switching to Code Generation in the Organization | 103 |
| 9.3 | Objectives Assessment | 104 |
| 9.4 | Limitations and Future Work | 105 |

9.1 Code Generation vs Current Manual Processes

This section describes, in a critical thinking approach, the possible application of the proposed solution in the company. Weighing the advantages provided by such autonomous code generation techniques versus its adoption disadvantages.

9.1.1 Current Manual Processes

To compare the impacts of changing to another development process approach, it will be described three different possible scenarios when it comes to changing and adapting the current Sales & Service application.

9.1.1.A Adapting: New Attribute

Adding a new attribute is the most basic scenario when it comes to domain model changes. In this situation, the domain model is known, and already fully implemented by Sales & Services application. Although it is the most basic scenario, it still requires manual changes in the application so that these new attributes gets handled by the application.

When a new attribute is added by a *Product Modeler* in *Product Machine*, Sales & Services developers must proceed with several changes in the application. The necessary changes are:

- 1 - Add/implement the getter method in the *BOAdaptable* implementation Classe
- 2 - Add the attribute to Builder algorithms
- 3 - Add the attribute in the respective *Entity* Transformer algorithm
- 4 - Add attribute to the *Merge* algorithm
- 5 - Add attribute to be included in the *Entity Clone* method
- 6 - Add attribute to be included in the *Entity Copy* method
- 7 - Add the attribute to all required *Flows* so that the attribute gets rendered in the user's browser.
- 8 - Add the attribute to Hibernate Mappings, so it gets stored in the database.

As seen, adding just a new attribute to an existing *Entity* can require various actions from a developer. Leading to a repetitive and time-consuming task just to adapt the application for new attributes. Furthermore, to accomplish this manual process, a developer usually expends 4 hours of his working day.

9.1.1.B Adapting: New Entity

Adding a new *Entity* in a *Product Machine* model is the second worse case scenario when it comes to adapting Sales & Services to work with that model.

When a new *Entity* is modeled in PM, a Sales & Services developer must create new objects to handle that new domain *Entity*. The necessary components required to be produced are:

- 1 - Create *Entity* Transformer
- 2 - Create *Entity* Builder
- 3 - Create *Entity* related Algorithms: Copy, Clone
- 4 - Create and implement the correspondent *BOAdaptable* for the newest *Entity*
- 5 - Adjust the parent *Entity BOAdaptable* to have a getter for this new child object
- 6 - Create new Hibernate Mapping, so that it maps a new *Entity*
- 7 - Map all desired attributes in the *Screen Flows*

The amount of time to accomplish all of these steps to adapt S&S to have a new *Entity* is usually estimated around 1 to 3 working days of a developer.

9.1.1.C Adapting: New Domain Model

The worse case scenario is when S&S needs to be adapted to new domain models. In this case, a Sales & Services developer must go through a meticulous analysis to understand which *Entities* in this new domain model matches with the existing internal BOM.

This analysis is a manual process that involves having object transformations so that they can match existing *Entities* in the S&S internal BOM.

Furthermore, this analysis and consequent developments are very expensive regarding development time. This kind of adaptation can take several weeks or even months to accomplish and requires to run serious regression tests on the application.

9.1.2 Adopting Code Generation

As seen, the current manual process to adapt the presentation-layer with a given PM model can take from 4 hours for a simple attribute change up to several weeks of development time to adjust the application to new domain models.

The usage of code generation would mostly address the identified issues. Particularly the necessary time to make the presentation-layer able to communicate with the PM model and be ready to use.

Therefore, this section describes the advantages and disadvantages of adopting these code generation techniques when compared with the current manual process to adapt a *Product Machine* model to Sales & Services presentation-layer.

9.1.2.A Advantages

There are various benefits when code generation techniques are used. The main advantage is the time to generate the entire application. When compared with the current manual process this scenario fits the

case when it is necessary the adaptation of an entire domain model. Therefore, the comparison between these two processes, i.e. manual vs generated, demonstrates that the autonomous generation process is enormously more preferable, where in just 5 minutes the organization can have the application ready to run.

Another advantage is the simplicity in running the generation engine. Where using an executable batch file, even a non-developer person can start the generator and start the generated application.

For the developers, having such autonomous tools is also very convenient. Since it prevents these developers from being continuously working on such repetitive tasks, such as mapping an attribute in Hibernate configuration files, adjust existing algorithms, adjusting description languages such as *Flow screen definitions*, among others repetitive adaptations.

9.1.2.B Disadvantages

The first identified disadvantage when code generation techniques are being adopted is that it starts to have two different kinds of code in applications, the generated code, and the manual code. The manual code can be added for several reasons, but the most common reason is the necessary customizations that customers sometimes require to their businesses.

Developers must be careful where the manual code are set. Otherwise, a regeneration might override and drop all the manual code previously added, leading the developers starting over again their customizations.

Another problem with applications entirely generated is the debug and small changes, where it is challenging and error-prone. Because it needs first to generate code from the existing DSL, execute the generator, debug the generated code, and finally propagate the corrections back to the affected templates or DSL. (Franky and Pavlich-Mariscal, 2012) With this process, what should just take half a minute to change, can take three or four minutes since it is necessary to run the code generation routine to the developer see his changes applied and working.

9.2 Switching to Code Generation in the Organization

To switch to a code generation process in the organization, it would be necessary to change the current development process. In which more than ever new DSLs would be created to describe particularities within the application that currently are somewhat manually hardcoded.

Another implication of this process change is the way that developers code the application. Where it demands a process change from doing static code that is done once to a completely different approach. The code generation approach demands that instead of coding regular manual code, the developers must invest in programming over configuration, where many DSLs would be needed to describe particular areas

within the application and its correspondent parsers, so that the initial desired code gets generated.

The existing *ContainerAndDataHolder* object structure that is being returned in each *REST* endpoint is currently providing metadata and configuration hints to the javascript in the presentation layer of Sales & Service. This metadata allows the javascript parser to render the correct forms, widgets, and other User Interface (UI) components in the user's browser.

The *ContainerAndDataHolder* is currently a big data structure concerning its dimension, that degrades the network communication time between the client-server requests. Therefore, using the code generation approach, the return would just be the necessary objects to be rendered on the screen. Since the application is not generic anymore and it is particularly working for a specific implementation that was based on the initial DSLs that drove the code generation.

Although the proposed solution generates a complete application, it stands as a poor application as far as its functionalities are concerned. Where just CRUD operations are available. Therefore, most of the existing features in Sales & Service would be required to be ported over to this code generation approach.

9.3 Objectives Assessment

According to the project objectives described at 1.2, it is concluded that the present work has addressed all the objectives. Having studied various solutions approaches, where some, such as template driven and regular expression substitution were partially discarded due to the need for a more general solution path.

Studying various code generation frameworks and compared its features, has proved that *JHipster* has stood out from the others due to the overall code generation capabilities in which does not only generate a partial software layer, but is capable of a multi-software layer generation, including a rich javascript-based presentation-layer generation.

Regarding the definition of a DSL to describe the application generation intent, was not necessary since *JHipster* supports a DSL named JDL that has produced the same result.

Additionally, it is concluded that the objective to "Develop code generators" was achieved. Since it were developed new modules to work with *JHipster* and add extensibility to the default *JHipster entity sub-generator* behavior. The developed modules "PMInjector" and "BOAdaptables" have enhanced the generated *Domain Model* so that it gets compatible with *PM Runtime Services*.

It was also developed a proof of concept that can be used with at least two different product models. During this project, it was used two difference models to prove that the solution is dynamic and able to adapt to different models of various domains, where the product models used were: PM4 Life model and PM4 P&C model.

Regarding a general architecture for a channel application, the generated front-end is a Web-based solution that uses a richer Javascript framework, *HTML5*, and *CSS3*. This technology stack is fully compatible with any Desktop PC, Laptop, Tablet and even Smartphones. Furthermore, due to the usage of *Bootstrap*, the application got a responsive front-end, which means that regardless the user screen resolution, the elements will be reorganized according to the screen viewport.

9.4 Limitations and Future Work

Having a front-end application able to communicate with a PM model is much more than just create a domain model that implements its model interfaces. Each PM model has a necessary *minimal context* that needs to be set under its entities, so that when it gets in the *Information Service*, PM knows which product matches that *minimal context* criteria, and therefore be able to infer which *ProductComponentInfo* will be sent back to the presentation layer.

Therefore, it is identified as future work that this project would need to dynamically infer what is the *minimal context* of a PM model. Hence, the presentation layer should have a form where this *minimal context* should be rendered so that the User can fill the fields and consequently the application could retrieve the correspondent *Product* from PM.

Furthermore, PM *Runtime Services* are an essential component to communicate with, since it is from where all the modeled data comes from. Consequently, the generated application should also establish in which moments the backend code should go over these PM *Runtime Services* and when that is not necessary, for example, when should the system call the *Validate Service*.

Additionally, the returned object of *Information Service* is a *ProductComponentInfo*, each domain *Entity* has one equivalent product component info in PM. But these objects are not *BOAdaptables*. Thus, the generated application would need to have a new code generator module to generate these Builders or Factories, to deliver the capability to build an *Entity* based on a *ProductComponentInfo* object.

The generated presentation-layer is also very basic with just CRUD operations. As future work is also identified that this area would need various enhancements so that it could have an Homepage and/or a Dashboard, a Find screen to find created products with different Criteria's, workflow (Draft, Request Audit, Finalize, In Production and Discontinued).

Bibliography

- RGA. Life insurance product development innovation and optimization, 2014. URL [https://www.rgare.com/knowledgecenter/Documents/Life%20Insurance%20Product%20Development_Final%201-28-15%20\(2\).pdf](https://www.rgare.com/knowledgecenter/Documents/Life%20Insurance%20Product%20Development_Final%201-28-15%20(2).pdf).
- OjE - O Jornal Económico. msg life quer ajudar seguradoras a lancar produtos mais rapido, May 2015. URL <http://www.oje.pt/msg-life-quer-ajudar-seguradoras-a-lancar-produtos-mais-rapido/>. Accessed 11-Nov-2015.
- The National Institute of Standards and Technology NIST. What is experimental design?, Jan 2016. URL <http://www.itl.nist.gov/div898/handbook/pri/section1/pri11.htm>. Accessed 26-01-2016.
- Oxford dictionary. hypothesis - definition of hypothesis in english from the oxford dictionary, Ago 2016. URL <http://www.oxforddictionaries.com/definition/english/hypothesis>. Accessed 30-Aug-2016.
- techdata. Why is a value proposition important?, 2016. URL [http://www.techdata.com/tdagency/vendorconnect/FY14TDAgencyWebsite%20Giveaway\[1\].pdf](http://www.techdata.com/tdagency/vendorconnect/FY14TDAgencyWebsite%20Giveaway[1].pdf).
- S. Nicola, E. P. Ferreira, and J. J. P. Ferreira. A novel framework for modeling value for the customer, an essay on negociation. *International Journal of Information Technology & Decision Making*, 11(661): 1103–1120, 2012. ISSN 1520-9210. doi: 10.1142/S0219622012500162.
- C. Barnes, H. Blake, and D. Pinder. *Creating and Delivering Your Value Proposition: Managing Customer Experience for Profit*. Kogan Page Publishers, 2009. ISBN 0749458593, 9780749458591.
- A. Lindgreen and F. Wynstra. Value in business markets: What do we know? where are we going? *Industrial Marketing Management*, 34(7):732–748, 2005. doi: 10.1016/j.indmarman.2005.01.001.
- V. A. Zeithaml. Consumer perceptions of price, quality, and value: A means-end model and synthesis of evidence. *Journal of Marketing*, 52(3):2–22, 1988. ISSN 00222429. URL <http://www.jstor.org/stable/1251446>.

Tony Woodall. Conceptualising 'value for the customer': an attributional, structural and dispositional analysis. *Academy of marketing science review*, 2003:1, 2003.

Susana Nicola. Apresentação em aula: Análise de valor de negócio. 2016.

Accenture. Accenture 2013 consumer-driven innovation survey, Oct 2013a. URL https://www.accenture.com/t20150918T224011__w__/us-en/_acnmedia/Accenture/Conversion-Assets/Microsites/Documents15/Accenture-Consumer-Driven-Innovation-Survey-2013.pdf. Accessed 22-Oct-2016.

Cognizant. Advice made social, Oct 2013. URL <http://www.cognizant.ch/InsightsWhitepapers/Advice-Made-Social.pdf>. Accessed 22-Oct-2016.

Capgemini. Leveraging social media across the insurance lifecycle, Oct 2013. URL https://www.capgemini.com/sites/default/files/resource/pdf/leveraging_social_media_across_the_insurance_lifecycle.pdf. Accessed 22-Oct-2016.

EY. Consumers on board, Oct 2014. URL [http://www.ey.com/Publication/vwLUAssets/EY-consumers-on-board/\\$FILE/EY-consumers-on-board.pdf](http://www.ey.com/Publication/vwLUAssets/EY-consumers-on-board/$FILE/EY-consumers-on-board.pdf). Accessed 22-Oct-2016.

EY. Insurance in a digital world: the time is now, Oct 2013. URL [http://www.ey.com/Publication/vwLUAssets/EY_Insurance_in_a_digital_world:_The_time_is_now/\\$FILE/EY-Digital-Survey-1-October.pdf](http://www.ey.com/Publication/vwLUAssets/EY_Insurance_in_a_digital_world:_The_time_is_now/$FILE/EY-Digital-Survey-1-October.pdf). Accessed 22-Oct-2016.

IBM. Insurers, intermediaries and interactions, Oct 2013. URL http://www-935.ibm.com/services/multimedia/Insurers_intermediaries_interactions_Avril_2013.pdf. Accessed 22-Oct-2016.

Accenture. Accenture technology vision 2014, Oct 2013b. URL https://www.accenture.com/gr-en/_acnmedia/Accenture/next-gen/reassembling-industry/pdf/Accenture-Technology-Vision-2014.pdf. Accessed 22-Oct-2016.

Investopedia. Actuary, Oct 2016a. URL <http://www.investopedia.com/terms/a/actuary.asp>. Accessed 02-10-2016.

msg life. msg life - sales & service, Ago 2015. URL <http://www.msg-life.com/en/software-consulting-and-cloud-solutions-for-life-insurance-companies-and-pension-scheme-providers/sales-service/>. Accessed 24-Jan-2016.

Cambridge-Dictionary. Revamp, Oct 2016a. URL <http://dictionary.cambridge.org/dictionary/english/revamp>. Accessed 02-10-2016.

Atlassian. Agile roadmaps: build, share, use, evolve, Ago 2016. URL <https://www.atlassian.com/agile/roadmaps>. Accessed 25-Jan-2016.

- Investopedia. The industry handbook: The insurance industry, Feb 2016b. URL <http://www.investopedia.com/features/industryhandbook/insurance.asp>. Accessed 20-02-2016.
- R. Merkin and J. Steele. *Insurance and the Law of Obligations*. OUP Oxford, 2013. ISBN 0191507911, 9780191507915.
- Business Dictionary. What is insurance? definition and meaning, Feb 2016. URL <http://www.businessdictionary.com/definition/insurance.html>. Accessed 20-02-2016.
- Forbes. 13 types of insurance a small business owner should have - forbes, Feb 2016. URL <http://www.forbes.com/sites/thesba/2012/01/19/13-types-of-insurance-a-small-business-owner-should-have/#693858e294fd>. Accessed 20-02-2016.
- FJA-US. Fja-us.pm4, 2015. URL https://www.msg-life.com/fileadmin/Uploads/en/pdfs/ProductMachine_Folder_20140829.pdf.
- Superior Consulting Services. Software multitenancy, Ago 2016. URL <https://www.teamscs.com/2015/11/using-entity-frameworks-access-multi-tenant-data-saas-environment/>. Accessed 30-Aug-2016.
- Oracle. Path and classpath (the java™ tutorials ¿ essential classes ¿ the platform environment), Oct 2016. URL <https://docs.oracle.com/javase/tutorial/essential/environment/paths.html>. Accessed 02-Oct-2016.
- A.T. Imam, T. Rousan, and S. Aljawarneh. An expert code generator using rule-based and frames knowledge representation techniques. In *Information and Communication Systems (ICICS), 2014 5th International Conference on*, pages 1–6, April 2014. doi: 10.1109/IACS.2014.6841951.
- IEEE. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990. doi: 10.1109/IEEESTD.1990.101064.
- M.C. Franky and J.A. Pavlich-Mariscal. Improving implementation of code generators: A regular-expression approach. In *Informatica (CLEI), 2012 XXXVIII Conferencia Latinoamericana En*, pages 1–10, Oct 2012. doi: 10.1109/CLEI.2012.6427199.
- S. Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Press, 2008. ISBN 978-0-470-03666-2.
- CodeSmith Generator. Active vs. passive generation - codesmith generator - confluence, 2016. URL <https://codesmith.atlassian.net/wiki/display/Generator/Active+vs.+Passive+Generation>.

- Velocity. The apache software foundation : Apache velocity - vtl reference, 2016. URL <http://velocity.apache.org/engine/devel/vtl-reference-guide.html>.
- Jelly. The apache software foundation : Jelly - jelly overview, 2016. URL <http://commons.apache.org/proper/commons-jelly/overview.html>.
- FTL. The apache software foundation : Freemarker java template engine, 2016. URL <http://freemarker.incubator.apache.org/>.
- Acceleo. The Eclipse Foundation : Acceleo, 2016. URL <http://www.eclipse.org/modeling/m2t/?project=acceleo>.
- JET. The Eclipse Foundation : JET, 2016. URL <https://eclipse.org/modeling/m2t/?project=jet>.
- Xpand. The Eclipse Foundation : Xpand, 2016. URL <http://www.eclipse.org/modeling/m2t/?project=xpand>.
- MOFScript. The Eclipse Foundation : MOFScript, 2016. URL <http://www.eclipse.org/gmt/mofscript/>.
- M.A.S. Talab and D.N.A. Jawawi. A code generator for component oriented programming framework. In *Open Systems (ICOS), 2011 IEEE Conference on*, pages 225–230, Sept 2011. doi: 10.1109/ICOS.2011.6079314.
- Ying Wang, Dianfu Ma, Yongwang Zhao, Lu Zou, and Xianqi Zhao. Automatic rt-java code generation from aadl models for arinc653-based avionics software. In *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, pages 670–679, July 2012. doi: 10.1109/COMPSAC.2012.94.
- AndroMDA. Getting started java – introduction, 2016. URL <http://andromda.sourceforge.net/andromda-documentation/getting-started-java/>.
- Jaxio. Celerio , a code generation tool for data-oriented application written in java., 2016. URL <http://www.jaxio.com/en/celerio.html>.
- JHipster. Jhipster home, 2016. URL <https://jhipster.github.io/>.
- Cambridge-Dictionary. Revamp, Oct 2016b. URL <http://dictionary.cambridge.org/dictionary/english/opinionated>. Accessed 02-10-2016.
- Elasticsearch. Elasticsearch — elastic, 2016. URL <https://www.elastic.co/products>.
- Matt Raible. *The JHipster Mini-book*. C4Media, 2015. ISBN 9781329638143.

Modelio. Modelio, download uml modeling tool and free products, 2016. URL <https://www.modeliosoft.com/>.

UMLDesigner. Uml designer documentation, 2016. URL <http://www.uml designer.org/>.

GenMyModel. Design software faster than ever, 2016. URL <https://www.genmy model.com/>.

Visual Paradigm. Software design tools for agile teams, with uml, bpmn and more, 2016. URL <http://www.visual-paradigm.com/>.

JDL-Studio. Jdl-studio, 2016. URL <http://jhipster.github.io/jdl-studio/>.

Ehcache. Terracotta : Ehcache, 2016. URL <http://www.ehcache.org/>.

Hazelcast. Hazelcast the leading in-memory data grid - hazelcast.com, 2016. URL <https://hazelcast.com/>.

Logback. Logback home, 2016. URL <http://logback.qos.ch/>.

HikariCP. Hikaricp, 2016. URL <https://github.com/brettwooldridge/HikariCP>.

ModelJ. The model-driven design tool for j2ee, 2016. URL <http://modelj.sourceforge.net/>.

Sculptor. Sculptor - generating java code from ddd-inspired textual dsl, 2016. URL <http://sculptorgenerator.org/>.

E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003. ISBN 978-0321125217.

Maven. Maven – welcome to apache maven, 2016. URL <https://maven.apache.org/index.html>.

Sonatype Nexus. Nexus repository - software component management — sonatype, 2016. URL <http://www.sonatype.com/nexus-repository-sonatype>.

Java Sun. Lesson: Introspection (the java™ tutorials & javabeans(tm)), 2008. URL <http://web.archive.org/web/20090226224821/http://java.sun.com/docs/books/tutorial/javabeans/introspection/index.html>.

Oracle. Trail: The reflection api (the java™ tutorials), 2016. URL <https://docs.oracle.com/javase/tutorial/reflect/>.

Nuno Alves. personal communication, Apr. 04 2016.

Java 8. Java se development kit 8 downloads, 2016. URL <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>.

NodeJS. Download — node.js, 2016. URL <https://nodejs.org/en/download/>.

GIT. Git - downloading package, 2016. URL <https://git-scm.com/download/win>.

Yeoman. Composability — yeoman, Ago 2016. URL <http://yeoman.io/authoring/composability.html>. Accessed 20-Aug-2016.

A. Avram and F. Marinescu. *Domain Driven Design Quickly*. C4Media, 2006. ISBN 978-1-4116-0925-9.

M. Fowler and R. Parsons. *Domain-Specific Languages*. Addison-Wesley, 2012. ISBN 978-0321712943.



From PM to S&S Overview

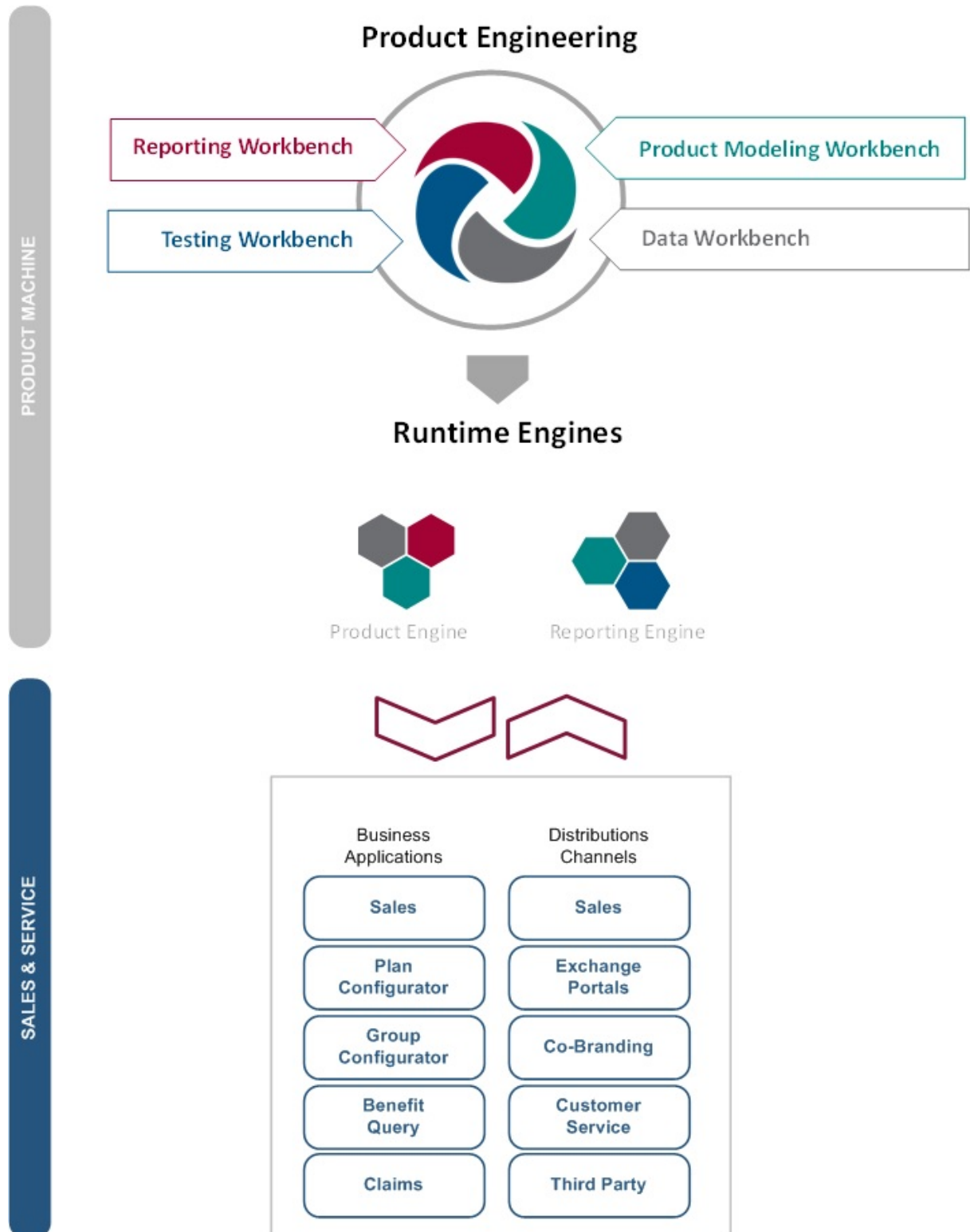


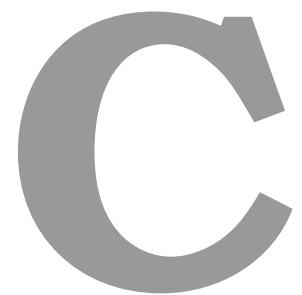
Figure A.1: Unified Product Platform Ecosystem



CANVAS : Business Model

| | | | | |
|--|---|--|--|---|
| Key Partners Suppliers Dell FJA-US msg life System RSA | Key Activities Programming Developing Insurance Solutions Maintenance of existing solutions | Value Propositions S&S Platform Performance Insurer: Time to Market Improve product launch process Users Empowerment Marketing and Sales empowerment Retrieve IT dependencies | Customer Relationship Sales Presentations Demos Webinar Sales Force Email Marketing Platform Insurance Events msg life Marketing Materials Inside Insurance Trends Digital & Paper Magazine Brochures Videos Infographics | Customer Segments msg life Final Customers |
| | | | Channels Business Visits Digital Marketing Internet E-mail Phone | |
| Cost Structure Human Resources Hardware Software Licence Security | | | Revenue Streams Licensing Custom Product Features Existing Customers maintenance New Customers | |
| | | | | |

Figure B.1: CANVAS Business Model



Related Technologies

Contents

| | | |
|-----|---|-----|
| C.1 | DDD - Domain Driven Design | 123 |
| C.2 | DSL - Domain Specific Languages | 129 |

In the enterprise development community, especially the web development community, have been tainted by years of hype that took software engineers and similar positions away from proper object-oriented software development. In the Java community, proper domain modeling was lost in the hype of EJB and the container/component models of 1999-2004. Luckily, shifts in technology and the collective experiences of the software development community are moving us back towards traditional object-oriented paradigms. However, the community is lacking a clear vision for how to apply object orientation on an enterprise scale, in which is why DDD is important. (Avram and Marinescu, 2006)

C.1 DDD - Domain Driven Design

Software development is often applied to automating real-world processes, or providing solutions to real business problems. For this reason, it is important to know from the beginning that software is originated from and deeply related to the domain of the problem that it was designed to work with. (Avram and Marinescu, 2006)

To create complex software, e.g. for insurance or banking business, domain knowledge is the key. Bankers and Insurer's specialists are the ones that understand very well this business; they know all the details, all the catches, all the rules and the possible issues. This is where software development should always start: the domain.

When a software project begins, the focus should reside on its operation domain. The entire purpose of the software is to enhance a specific domain. Therefore, the software has to fit harmoniously with the domain it has been created for. Software needs to incorporate the core concepts and elements of the domain and to realize the relationships between them precisely. (Avram and Marinescu, 2006)

C.1.1 Domain

"A domain model is not a particular diagram; it is the idea that the diagram is intended to convey. It is not just the knowledge in a domain expert's head; it is a rigorously organized and selective abstraction of that knowledge." (Evans, 2003)

A domain describes something in the real world. Therefore, a domain cannot just be taken and poured over the keyboard into the computer to become code. An abstraction of the domain must be created. This domain abstraction is earned with conversations with domain experts. In the beginning, the domain abstraction is unfinished but in time, while the development teams work on it, it gets better and becomes more and more clear to the development team. This abstraction, for the software development team, becomes the model, the model of the domain.

C.1.2 Model

A model is an internal representation of the target domain, and it is very necessary throughout the design and the development process. During the design process, lots of references to the model are made. The world around us is too much for our heads to handle. Even a specific domain could be more than what the human mind can handle at one time. Therefore, the information must be organized, to systematize it, to divide it up into smaller pieces and, to group these pieces into logical modules. (Avram and Marinescu, 2006)

A domain contains just too much information to be included in the model. Some are not even necessary to be included, other are key elements that are needed to be included. The challenge is the analysis about what to keep and what to throw away. This is part of the design, the software creating process. (Avram and Marinescu, 2006)

A model is an essential part of software design. It is needed to be able to deal with complexity. The thinking process about the domain is synthesized into this model. Therefore, it is very common that the model starts to be the bridge between developers and domain/business specialists. It is very common that model elements start to be used in conversations/discussions. For this reason, the model must be exposed to domain experts, designers, and developers. Therefore it is imperative to establish ways to express it, precisely, completely and without ambiguity. There are different ways to do that (Avram and Marinescu, 2006) :

1. **Graphically:** Diagrams, Use Case, Drawings, pictures, etc.
2. **Writing:** Writing down the vision about the domain

C.1.3 Building Blocks

In Object-Oriented programs, it is very common to see UI, database, and other support code written directly into the business objects. On the other hand, some business logic is embedded within the UI widgets and database scripts. This scenario sometimes happens because it is the easiest way to make things work quickly.

However, when the domain-related code is mixed with the other layers, it becomes tough to see and think about. Additionally, superficial changes to the UI can change the internal business logic that will impact another section within the application which leads to a scenario where to change a simple business rule will require meticulous tracing of UI code, database code, or other program elements. Implementing coherent, model-driven objects become impractical. "Automated testing is awkward. With all the technologies and logic involved in each activity, a program must be kept very simple, or it becomes impossible to understand". (Avram and Marinescu, 2006)

Therefore, splitting a complex program into layers is the most common solution. "Develop a design within each layer that is cohesive and that depends only on the layers below. Follow standard architectural

patterns to provide loose coupling to the layers above. Concentrate all the code related to the domain model in one layer and isolate it from the user interface, application, and infrastructure code. The domain objects, free of the responsibility of displaying themselves, storing themselves, managing application tasks and so forth, can be focused on expressing the domain model.” (Avram and Marinescu, 2006) This allows a model to evolve to be rich and clear enough to capture essential business knowledge and put it to work. The Figure C.1 illustrates the map of the most important patterns and its relationships used in a model-driven design.

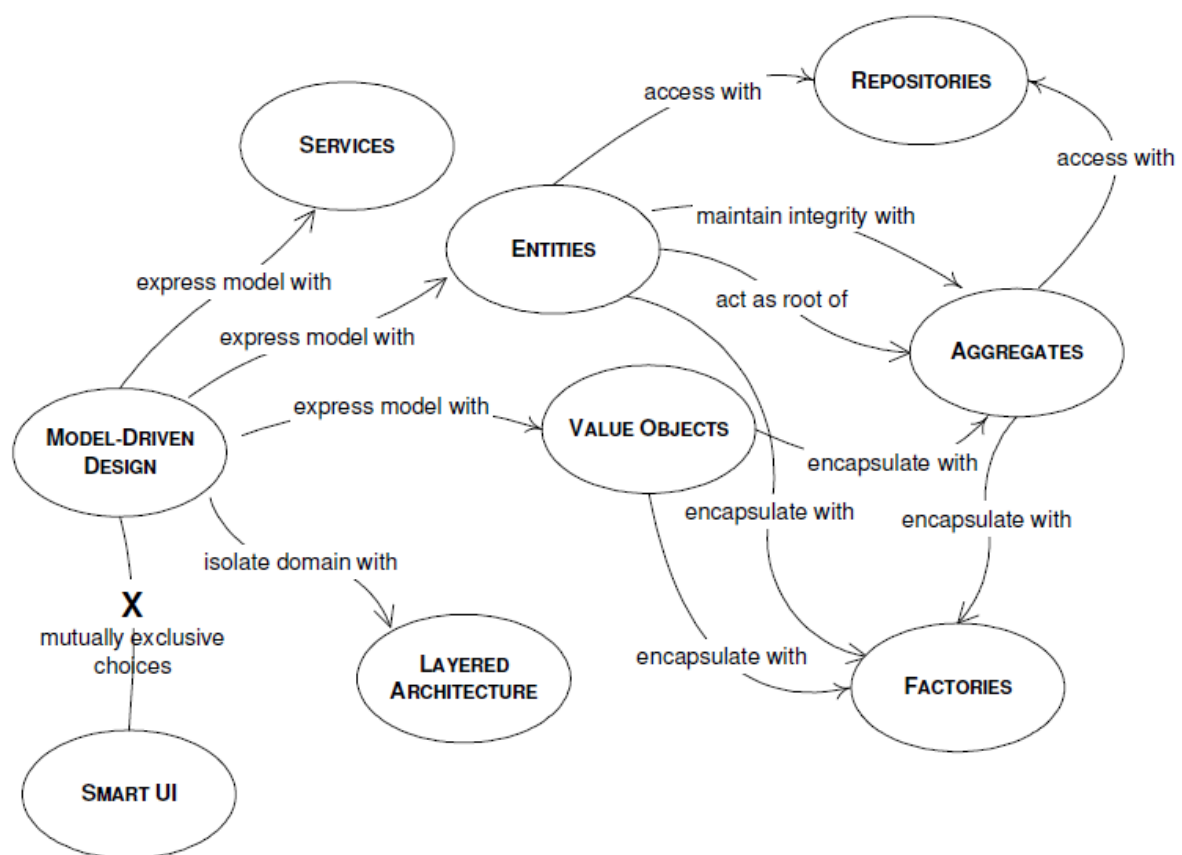


Figure C.1: Domain-Driven Design - Patterns and Relationships

Source: Domain-Driven Design Quickly (Avram and Marinescu, 2006)

It is important to design an application in separate layers and establish rules of interactions between those layers. If the code is not clearly divided into layers, it will soon become so entangled that it becomes tough to manage changes. One simple change in one section of the code may have unexpected and undesirable results in other sections. (Avram and Marinescu, 2006)

C.1.4 Model Driven Design - Sterotypes

The concern of having complex applications divided into various layers, each with their own responsibilities within the system, lead to the definition of these DDD patterns as illustrated in Figure C.1. These stereotypes: *Entities*, *Value Objects*, *Services*, *Modules*, *Aggregates*, *Factories*, *Repositories*; allow the layer segregation, leading to a low-coupled and high-cohesion code per system layer.

C.1.4.A Entities

Entities are important objects of a domain model, and they should be considered from the beginning of the modeling process. An *Entity* is a category of objects which have an identity. Usually, the identity is either an attribute of the object, a combination of attributes or even a behavior. (Avram and Marinescu, 2006)

Although *Entities* are an important object of a domain model, not all objects should be defined as an *Entity* and it is crucial to determine if an object needs to be an *entity* in the domain model. *Entities* are tracked by their identity, but creating and tracking identities come with costs, e.g.: Performance degradation, since there has to be one instance for each *entity* object. "If Customer is an entity object, then one instance of this object, representing a particular bank client, cannot be reused for account operations corresponding to other clients. The outcome is that such instance has to be created for every client. This can result in system performance degradation when dealing with thousands of instances." (Avram and Marinescu, 2006)

C.1.4.B Value Objects

An object that is used to describe certain aspects of a domain but does not have an identity is named *Value Object*. (Avram and Marinescu, 2006) This type of objects are used when their identity are not necessary, but their attributes are. Having no identity, *Value Objects* can be easily created and discarded. This type of object simplifies the design because garbage collector takes care of these objects when they are no longer referenced by any other object.

Value Objects are created with a constructor, and never modified during their life time. When a new *Value Object* is needed, it can be simply created and used.

It is highly recommended that *Value Objects* be immutable. Being immutable and having no identity means that they can be shared, and this can be imperative for some designs. Having immutable sharable objects brings important performance implications.

Value Objects can contain other *Value Objects*, and they can even reference other *Entities*. Although *Value Objects* are used to simply contain attributes of a domain object, that does not mean that it should include a long list of all the necessary attributes. It can simply be composed of another object either *Entities* or *Value Objects*. (Avram and Marinescu, 2006)

C.1.4.C Services

There is an important behavior of the domain that does not seem to belong to any object. "Adding such behavior to an object would spoil the object, making it stand for functionality which does not belong to it." (Avram and Marinescu, 2006) When such behavior is recognized in the domain, the best practice is to declare it as a *Service*. A *Service* is an object without an internal state, and its purpose is to provide functionality for the domain. *Services* can group related functionality which serves *Entities* and the *Value Objects*. The three main characteristics of a *Service* are:

1. The operation performed refers to a domain concept which does not naturally belong to an *Entity* or *Value Object*.
2. The operation performed refers to other objects in the domains.
3. The operation is stateless.

Services act as interfaces which provide operations. A *Service* is not about the object performing the service but is related to the objects operations that are carried out on/for a domain object. A *Service* usually becomes a point of connection for many objects. This connection is one of the reasons why behavior which naturally belong to a *Service* should not be included in domain objects. A *Service* should not replace the operation which normally belongs to domain objects. A *Service* should not be created for every operation needed. However, when such an operation stands out as an important concept in the domain, a *Service* should be set up for it. (Avram and Marinescu, 2006)

C.1.4.D Modules

Enterprise applications are complex, and with time, the model tends to grow. Often the models reach a point where its just hard to talk about as a whole and to understand the particularities about the domain. For these cases, it is necessary to organize the model into *Modules*.

Modules are an efficient way to manage complexity, because looking into project *Modules* and then its relationships; it is easier to get the picture of a large model. Another reason for using *Modules* is related to code quality. Software source-code should have a high cohesion and a low coupling. Where cohesion starts at the class and method level, but it can be applied at the module level. It is recommended to group highly related classes into modules to maximize cohesion. (Avram and Marinescu, 2006)

C.1.4.E Aggregates

An *Aggregate* is a group of associated objects which are considered as one unit about data changes. The *Aggregate* is demarcated by a boundary which separates the objects inside from those outside. Each *Aggregate* has one root object which is an *Entity*, and it is the only object accessible from outside. The root can hold references to any of the aggregate objects, and the other objects can hold references to each other, but an outside object can hold references only to the root object. (Avram and Marinescu, 2006)

Aggregates ensure data integrity and enforce the invariants. Since other objects can hold references only to the root object, it means that they cannot directly change the other objects in the aggregate. All that they can do is modify the root, or ask the root to perform some actions. Moreover, the root will be able to change the other objects, but that is an operation contained inside the aggregate, and it is controllable. (Avram and Marinescu, 2006)

C.1.4.F Factories

"The creation of an object can be a major operation in itself, but complex assembly operations do not fit the responsibility of the created objects. Combining such responsibilities can produce ungainly designs that are hard to understand." (Avram and Marinescu, 2006) Therefore, it is necessary a concept that encapsulates the process of complex object creation.

Factories are used to encapsulate the necessary knowledge for an object creation, and they are especially useful to create *Aggregates* (when the root of the Aggregate is created, all the objects contained by the Aggregate are created along with it, and all the invariants are enforced). It is important for the creation process to be atomic. If it is not, there is a chance for the creation process to be half done for some objects, leaving them in an undefined state. (Avram and Marinescu, 2006)

C.1.4.G Repositories

Databases are part of the infrastructure. A poor solution is having the client to be aware of the details needed to access a database. Therefore, *Repositories* encapsulate all the logic needed to obtain object references. "The domain objects will not have to deal with the infrastructure to get the needed references to other objects of the domain. They will just get them from the Repository, and the model is regaining its clarity and focus." (Avram and Marinescu, 2006)

C.2 DSL - Domain Specific Languages

DSL "is a computer programming language of limited expressiveness focused on a particular domain." (Fowler and Parsons, 2012)

DSL has little expressiveness, therefore it supports a bare minimum of features needed to support its domain which means that an entire system cannot be built or described using one simple DSL. Rather, a DSL can be used to describe one particular aspect of the system. This limited language is only useful if it has a clear focus on a small domain. Here, the domain focus is what makes a limited language worthwhile. (Fowler and Parsons, 2012)

According to (Fowler and Parsons, 2012), there are three main categories of DSL: external DSLs; internal DSLs and language workbench.

1. **Internal DSL** is a particular way of using a *general-purpose language* (Java, C#, C++). A script in an internal DSL is a valid code in its general-purpose language, but only uses a subset of the language's features in a particular style to handle one small aspect of the overall system. The result should have the feel of a custom language, rather than its host language. The classic example of this style is *Lisp*; Lisp programmers often talk about Lisp programming as creating and using DSLs. *Ruby* has also developed a strong DSL culture; Many *Ruby* libraries come in the style of DSLs. In particular, Ruby's most famous framework, *Rails*, is often seen as a collection of DSLs.

In Java, an example of the usage of an internal DSL is *JMock* a Java library for Mock Objects. The Listing C.1 illustrates an example of an *Internal DSL*:

Listing C.1: Example: internal DSL

```
1 mainfram.expects(once())
2     .method("buy").with(eq(QUANTITY))
3     .will(returnValue(TICKET));
```

This example uses partial Java language features to develop an internal DSL. *JMock* uses a mix of *Method Chaining* on the mock object itself (*expects*) and *Nested Function* (*once*). *Object Scoping* is used to allow the Nested Function methods to be bare. *JMock* uses progressive interfaces that allow to *with* be only available after *method* permitting the auto-completion in IDE's, guiding the developer to write the JMock expectation in the right way. (Fowler and Parsons, 2012)

2. **External DSL** is separated language from the main application language that is using the DSL. Often, is used a *custom syntax* but is very common to find DSLs using well-known syntaxes like XML. To interpret an external DSL, a parser is used in the host language code with text parsing

techniques. External DSLs are widely used and include: *regular expressions*, *SQL*, *Awk* and are using XML to describe and configure the behavior of well-known libraries like: *Hibernate* (hbm.xml Describes how the classes are mapped in the Database); *Maven* (pom.xml describes how projects are related to each other, stating its classpath dependencies, exclusions and build process to the application artifacts). Other examples, now used for presentation layers are the well-known: Extensible Application Markup Language (XAML) (Microsoft introduced XAML as a DSL to lay out UIs) and CSS, both are interpreted by browsers to render according to what those DSL languages are describing. (Fowler and Parsons, 2012)

3. **Language Workbench** is a specialized IDE for defining and building DSLs. In particular, a language workbench is used not just to determine the structure of a DSL but also as a custom editing environment for people to write DSL scripts. The resulting scripts intimately combine the editing environment and the language. (Fowler and Parsons, 2012)

C.2.1 Why use a DSL?

DSL are popular for several reasons, but (Fowler and Parsons, 2012) has highlighted two main ones: improving productivity for developers and enhance communication with the domain experts.

This statement is vital because it brings us two essential reasons about why should we be taken DSLs into account for the development of this project. First, and according to the previous statement, a DSL can bring us an improvement of developers productivity which alone is an important statement to address the objectives of this project. Secondly, being a tool that can be used as a communication bridge between front-end development team and domain experts (i.e. *PM modelers*) when the system design is being decided, it can solve communication issues and prevent wrong path decisions.

DSL provides means to communicate the intent of a part of a system more clearly. A system definition in a DSL form is easier to understand what it is doing. Moreover, this is not just an aesthetic desire, because the easier is to read a lump of code, the easier it is to find mistakes, and the easier it is to modify the system. (Fowler and Parsons, 2012)

”The limited expressiveness of DSLs makes it harder to say wrong and easier to see when you’ve made an error.” (Fowler and Parsons, 2012)

The hardest part of a software project and the most common reasons for its failures are the communication with customers and users of that software. (Fowler and Parsons, 2012) A DSL provides a clear and precise language to express the domain in which can help on this communication issues. Its revision should not only include the development team but most importantly, the domain experts. However, if we want to guaranty that the domain experts completely understand the content of the *model* expressed by the DSL, sometimes a DSL alone is not enough. It is also necessary to provide a visual representation

of the model (See: C.2.2). Being able to read and understand the DSL, domain experts can prevent mistakes to happen during the system design phase. (Fowler and Parsons, 2012)

C.2.2 Visualization

Visual representations are a great advantage when working with DSLs, in particular, graphical representations. Even with a textual DSL is possible to obtain a diagrammatic representation of what that DSL is expressing. The most common tool used to express a DSL in a diagram is the *DOT*¹ language. The *DOT* language is part of the *Graphviz*² package. (Fowler and Parsons, 2012)

Graphviz is an open-source tool that allows the description of mathematical graph structures (nodes and edges) and automatically plots them. It also lay out node-and-arc graphs structures. Having various kinds of reports creates different perspectives of what a model looks like and that is very useful.

Using tools such as *Graphviz* is extremely helpful for many DSLs because it gives another representation perspective about the DSL. Another visualization is always very valuable since it gives humans different ways to understand the model. (Fowler and Parsons, 2012)

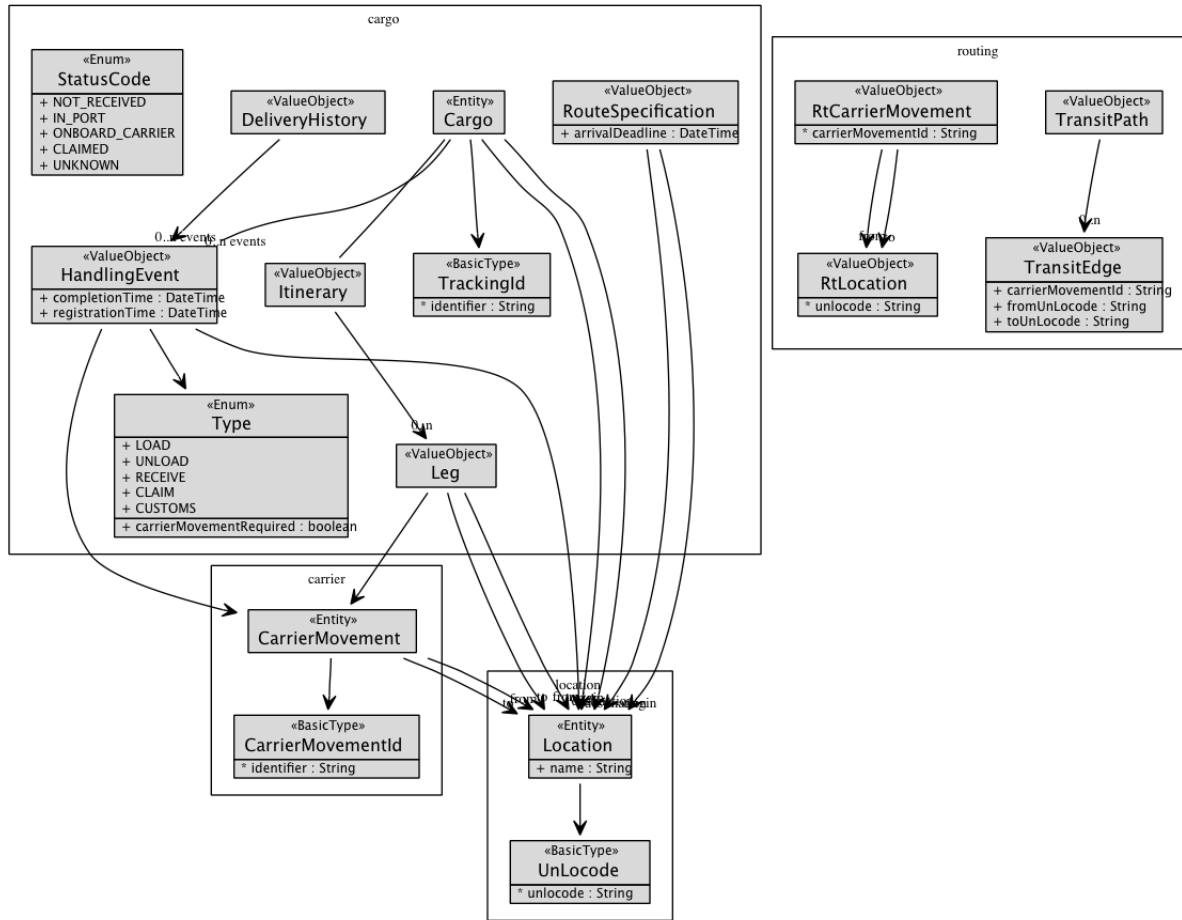
¹DOT is a plain text graph description language

²Graphviz is open source graph visualization software



Sculptor: Domain Driven Design

sample



Source: Sculptor documentation (Sculptor, 2016)

Figure D.1: Sculptor: DDD Sample Model



Diagrams

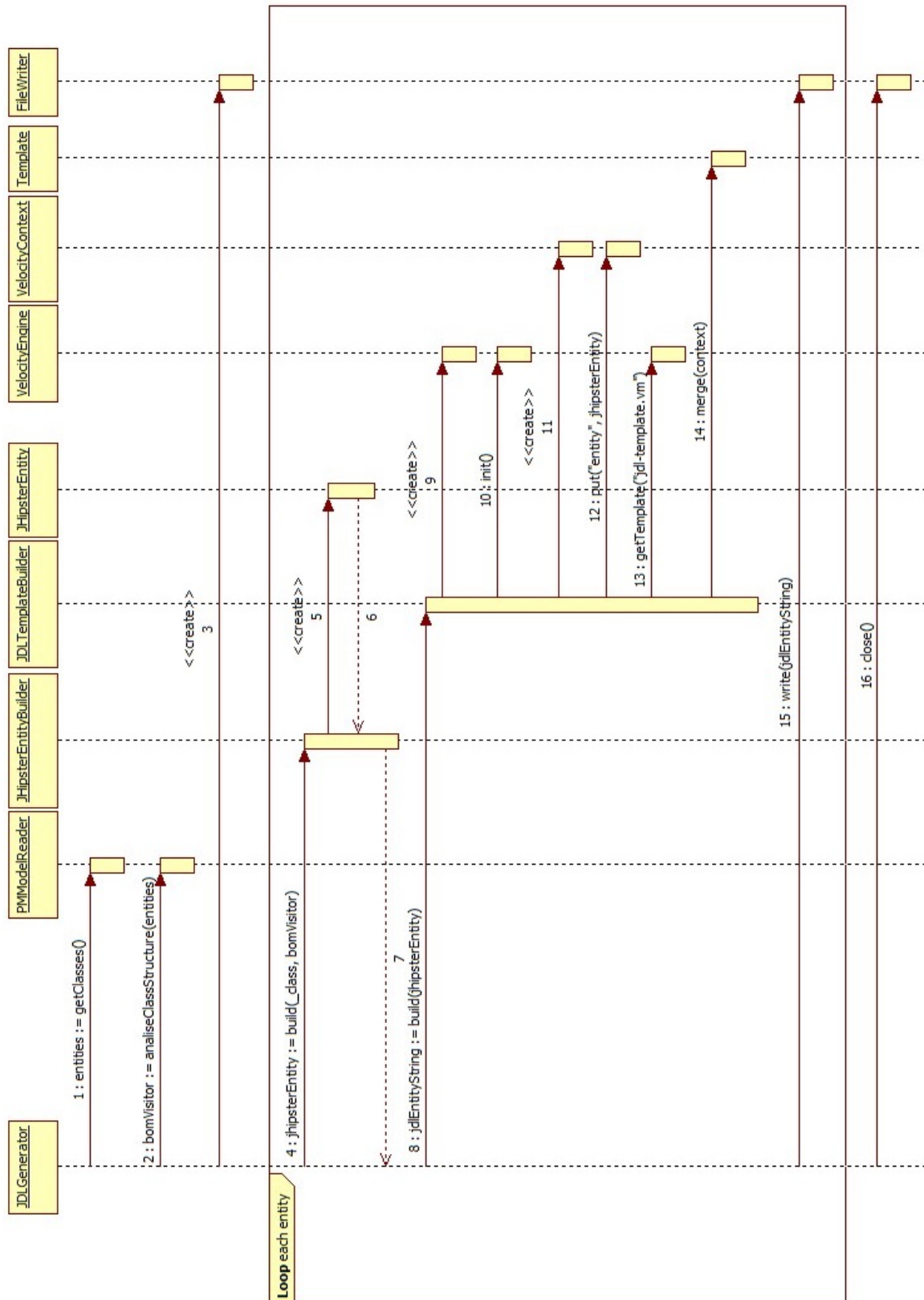


Figure E.1: Sequence Diagram - Creating JDL model

