INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Ferramenta de Mapeamento de Modelos para uma Plataforma de Produtos de Seguros

ISAAC NEVES DE SOUSA
Outubro de 2017

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Ferramenta de Mapeamento de Modelos para uma Plataforma de Produtos de Seguros

Isaac Neves de Sousa

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Sistemas Computacionais

Orientador: Doutor Paulo Gandra de Sousa

Júri:

Presidente:

Doutor Nuno Alexandre Pinto da Silva, ISEP

Vogais:

Porto, Outubro 2017

Aos meus Pais, irmãos e irmãs

Resumo

Atualmente um sistema informático tem normalmente que interagir/comunicar com diversos sistemas externos, que disponibilizam informação em diversos formatos e utilizando diferentes modelos. Na integração deste tipo de sistemas existe a necessidade de transformar diferentes modelos e/ou agregar informação para posterior utilização. Este processo de integração pode ser um processo complexo e demorado, dependendo da complexidade inerente do negócio associado.

Este trabalho pretende apoiar e desenvolver o trabalho previamente realizado pela empresa msg life Iberia que tem vindo a desenvolver uma aplicação de comercialização de seguros denominada Sales & Service, no ramo de Vida e Não Vida, que se distingue pela capacidade de integração de diferentes PDS – *Product Definition System* e pela capacidade de customização dos diferentes produtos existentes nesses sistemas, tanto ao nível dos próprios produtos como à forma de apresentação dos mesmos, sem necessidade de desenvolvimento de uma aplicação de raiz para tal.

Esta capacidade é alcançada com base na existência de um modelo canónico, que representa um produto de seguros, genérico o suficiente para suportar todas essas variações. Os modelos existentes nos diferentes PDS são extensos e com grande nível de especificidade o que torna o processo de integração de um novo Product Definition System num processo demorado e custoso.

Pretende-se desenvolver uma ferramenta de mapeamento entre os diferentes modelos utilizados que permita facilitar e automatizar dentro do possível todo o processo de integração com diferentes PDS. Esta ferramenta deve suportar vários tipos de transformação e apresentar ganhos relevantes comparativamente ao processo manual atualmente em prática.

Para atingir estes objetivos pretende-se tirar partido de técnicas de *Model to Model Transformation* e a utilização das melhores práticas preconizadas pela utilização do paradigma *Model Driven Architecture* e de outras técnicas de mapeamento entre objetos.

Palavras-chave: *Model to Model Transformation, Model Driven Architecture, Product Definition System, Ferramenta Mapeamento, Seguros*

Abstract

Currently a computer system usually has to interact/communicate with several external systems, which provide information in different formats and using different models. In the integration of this type of systems there is a need to transform different models and/or to aggregate information for later use. This integration process can be a complex and time-consuming process, depending on the inherent complexity of the associated business.

This work intends to support and develop the work previously carried out by the company msg life Iberia which has been developing an insurance sales application called Sales & Service in the field of Life and Non Life, distinguished by the integration capacity of different PDS - Product Definition System and the ability to customize the different products in these systems, both in terms of the products themselves and also in the way they are presented, without the need to develop an application from the ground up for achieving that.

This capability is achieved on the basis of the existence of a canonical model, which represents an insurance product, generic enough to support all these variations. The existing models in the different PDS are extensive and with a high level of specificity which makes the process of integrating a new Product Definition System into a time-consuming and costly process.

It is intended to develop a mapping tool among the different models used to facilitate and automate the entire integration process with different PDS. This tool must support various types of transformation and present significant gains compared to the manual process in place nowadays.

To achieve these objectives we intend to take advantage of Model to Model Transformation techniques and the use of the best practices recommended by the use of the Model Driven Architecture paradigm and other mapping techniques between objects.

Keywords: MMT (Model to Model Transformation), MDA (Model Driven Architecture), PDS (Product Definition System), Mapping Tool, Insurance

Agradecimentos

A realização desta dissertação apresentou-se como um desafio com altos e baixos. Várias pessoas estiveram presentes ao longo deste percurso e com o seu apoio foi possível tornar todo o processo mais simples.

Em primeiro lugar quero agradecer aos meus pais, irmãos e irmãs por terem sido incansáveis na motivação e apoio incondicional.

Gostava de agradecer ao meu orientado, Doutor Paulo Gandra de Sousa pela disponibilidade e aconselhamento.

Um agradecimento especial a todos os meus colegas e amigos da organização msg life Iberia pelas palavras de incentivo e por me terem ajudado ao longo deste trabalho.

O meu Muito Obrigado a todos!

Índice

Lista de Figuras

Lista de Tabelas

Acrónimos e Símbolos

Lista de Acrónimos

AMW	*AtlanMod Model Weaver*
API	*Application Programming Interface*
ATL	*Atlas Transformation Language*
BOM	*Business Object Model*
CR	*Composition Rule*
CRUD	*Create, Read, Update and Delete*
CRV	*Composition Rule Variation*
CW	*Customization Workbench*
DSL	*Domain Specific Language*
DTO	*Data Transfer Object*
EMF	*Eclipse Modeling Framework*
EMFT	*Eclipse Modeling Framework Technology*
FFE	*Fuzzy Front End*
GMT	*Generative Modeling Technologies*
MDA	*Model Driven Architecture*
MMT	*Model to Model Transformation*
MO	*Marketable Offer*
MOF	*MetaObject Facility*
MWB	*Modeling Workbench*
NCD	*New Concept Development*
NPD	*New Product Development*
OCL	*Object Constraint Language*
OMG	*Object Management Group*

PAS	*Policy Administration System*
PDS	*Product Definition System*
PIM	*Plataform Independent Model*
PM	*Product Machine*
PSM	*Plataform Specific Model*
QVT	*Query/View/Transformation*
RFP	*Request For Proposal*
S&S	*Sales & Service*
SC	*Selection Criteria*
SOA	*Service-Oriented Architecture*
UML	*Unified Modeling Language*
UUID	*Universally Unique Identifier*
VP	*Value Proposition*
XMI	*XML Metadata Interchange*
XML	*Extensible Markup Language*
XSD	*XML Schema Definition*

1 Introdução

1.1 Contexto

Um dos principais desafios que as companhias de seguros enfrentam atualmente está relacionado com o tempo necessário para o desenvolvimento e comercialização de novos produtos bem como o alinhamento entre as necessidades dos seus clientes e os produtos desenvolvidos. A análise realizada por (RGA, 2014) diz-nos que "globalmente, em média, o tempo necessário para lançar um novo produto desde a sua conceção até à venda é de seis até doze meses." (tradução própria)

De forma geral o desenvolvimento de novos produtos pressupõem a criação de uma aplicação de *software* específica para o produto, o que normalmente implica grandes custos e um grande período de tempo de desenvolvimento. A esta falta de agilidade, normalmente está associada uma perda de competitividade e custos desnecessários. Sendo assim, as companhias de seguros necessitam diminuir a sua dependência junto dos departamentos de *software*, no que se refere à facilidade de desenvolvimento e comercialização de novos produtos.

Nesse sentido a msg life Iberia tem vindo a desenvolver uma aplicação de comercialização de seguros denominada S&S – Sales & Service, que se distingue pela capacidade de integração de diferentes PDS – Product Definition System e pela capacidade de customização dos diferentes produtos existentes nesses sistemas, tanto ao nível dos próprios produtos como à forma de apresentação dos mesmos, sem a necessidade de desenvolvimento de uma aplicação de raiz para tal.

1.2 Problema

O S&S apresenta as vantagens apresentadas através da utilização de um modelo interno único para representar os diferentes conceitos básicos de um seguro. Trata-se de um modelo canónico que contém a informação que representa um produto de seguros. Devido à existência desse modelo é possível generalizar grande parte da aplicação, permitindo a reutilização dos

diferentes componentes e a obtenção de uma camada de independência entre a aplicação e os diferentes PDS. O que permite a adaptação da aplicação para diferentes clientes e necessidades.

Como se pode compreender a integração com cada um dos diferentes Sistemas de Definição de Produtos é um processo complexo e que apresenta grandes desafios, visto cada produto conter diferentes especificidades e regras de negócio. O sistema atual já permite a integração de diferentes modelos da PM – Product Machine[1], no entanto apresenta ainda um custo de integração considerável, que poderia ser reduzido e até certo nível padronizado. Existe espaço para grandes melhorias no que se refere à generalização de algumas tarefas e processos de integração de um novo PDS.

A questão do mapeamento entre os modelos e o modelo interno apresenta vários desafios técnicos e da própria arquitetura da aplicação. Pois é pretendido atingir um nível elevado de independência para com os sistemas externos mas mantendo a capacidade de resposta às necessidades específicas dos vários clientes e dos seus produtos de seguros.

1.3 Motivação

A existência e utilização de diferentes modelos entre aplicações é um dos aspetos que condiciona o desenvolvimento e integração de diferentes sistemas.

A principal motivação deste trabalho é exatamente a de facilitar o processo de integração de um novo PDS no Sales & Service para permitir obter os ganhos de eficiência pretendidos. Permitindo assim aumentar a competitividade e posicionamento do S&S como uma plataforma única de distribuição de produtos de seguros.

O S&S interage com diferentes modelos disponibilizados por um PDS, no entanto surge a necessidade de suportar sistemas completamente distintos e como tal o processo de integração para esses casos apresenta-se como um possível desafio técnico.

1.4 Objetivos e Abordagem

Existe a necessidade de facilitar o processo de integração com os diferentes Product Definition System e um dos principais desafios nesse sentido está relacionado com a dificuldade de mapeamento entre os diferentes modelos utilizados em cada um dos possíveis PDS e o modelo canónico interno, utilizado para permitir uma camada de isolamento e maior autonomia. Para tal pretende-se desenvolver uma ferramenta que permita simplificar o processo de

[1] A Product Machine é uma solução informática que centraliza as regras dos produtos de seguros dos ramos Vida, Não Vida e Saúde. Esta plataforma permite a interação com os sistemas envolvidos no processo de definição, distribuição e gestão dos produtos e contratos.

mapeamento entre os diferentes tipos de modelos utilizados através da definição de técnicas reutilizáveis de transformação. Permitindo diminuir todo o custo associado a esta operação.

Os objetivos propostos podem ser sintetizados nos seguintes pontos:

- Análise dos modelos utilizados e identificação das suas especificidades para obter uma visão global dos mapeamentos necessários;
- Estudo de atuais ferramentas e tecnologias de mapeamento de modelos/objetos passíveis de serem utilizadas para resolver as necessidades de mapeamento;
- Análise do enquadramento do modelo canónico e alinhamento com possíveis necessidades de ajuste/alteração;
- Desenho de uma possível solução tendo em conta os mapeamentos identificados e as necessidades de integração dos diferentes sistemas;
- Implementação (prova de conceito) de uma ferramenta de mapeamento entre diferentes modelos externos e o modelo interno; estes mapeamentos devem suportar vários tipos de transformação, nomeadamente: igualdade; concatenação e *split*; *lookup*; expressões;
- Desenvolvimento de técnicas de geração de código para a realização da transformação dos modelos de acordo com regras de mapeamento previamente definidas;
- Avaliar a solução implementada através da utilização de dois Sistemas de Definição de Produtos distintos. Encontrando-se um totalmente integrado e em funcionamento atualmente e o outro ainda numa fase inicial de integração e como tal passível de demonstrar a adequação/utilidade da solução (prova de conceito) desenvolvida;

A abordagem utilizada baseia-se nas técnicas de MMT - Model to Model Transformation e na utilização das melhores práticas preconizadas pela utilização do paradigma MDA - Model Driven Architecture e ainda na utilização de técnicas e ferramentas de mapeamento de objetos.

Os motores de definição de produtos utilizados são a FJA[2] Product Machine V4 e o msg Symass.

1.5 Estrutura do Documento

O trabalho a desenvolver insere-se numa plataforma de criação e distribuição de seguros já existente. Nesse sentido e para permitir uma melhor compreensão do problema em causa no capítulo 2 é realizada uma breve introdução ao setor dos seguros e a alguns dos conceitos e termos utilizados nesse setor. Neste capítulo são também apresentadas as aplicações Sales & Service, Product Machine e Symass uma vez que estas são as aplicações cuja integração a ferramenta pretende apoiar e facilitar. Neste capítulo alguns detalhes adicionais sobre o problema que se tenta resolver também são apresentados.

[2] Companhia pertencente ao grupo msg life que desenvolve a Product Machine.

No capítulo 1 é apresentada uma análise do valor de negócio e um modelo de negócio da plataforma integrada Sales & Service e Product Machine como produto da msg life Iberia. O valor de negócio do novo componente a desenvolver também é apresentado.

Após a apresentação do contexto e do problema nos capítulos 4 e 5 é apresentado o estado da arte de dois tipos de técnicas possíveis de serem utilizadas para resolver o problema. São avaliadas estas duas técnicas e as ferramentas relevantes dentro de cada área uma vez que no capítulo 6 são apresentadas as abordagens existentes baseadas nessas técnicas e é neste capítulo que se expõe alguns dos critérios que podem ser utilizados para validar qual a abordagem a seguir.

No capítulo 7 são apresentados detalhes sobre a arquitetura da solução a implementar de acordo com as abordagens preconizadas e utilizando as técnicas e ferramentas escolhidas.

Com base no *design* realizado uma prova de conceito da ferramenta de mapeamento foi desenvolvida e toda a informação relevante na execução dessa implementação é apresentada no capítulo 1.

No capítulo 1 são apresentados os critérios de avaliação da solução desenvolvida para validar o seu alinhamento com os requisitos apresentados.

2 Contexto e Problema

A tecnologia e as novas soluções informáticas apresentam um papel relevante em várias áreas do conhecimento e em vários setores comerciais. O setor dos seguros não é exceção, as novas soluções informáticas de desenvolvimento, comercialização e gestão de seguros assumem grande importância no panorama atual. Citação da publicação Insurance Trends (Pereira, 2015), que reforça a importância da tecnologia no setor:

> "A tecnologia tem um papel cada vez mais preponderante no quotidiano das empresas. Se antes o departamento de tecnologias de informação era visto apenas como um facilitador, hoje é encarado como um parceiro, que tem de estar em perfeita sintonia com a estratégia de negócio da organização. E isto é algo que é transversal a toda economia. Como refere Francisco Melo Pereira, o "difícil é identificar algum setor onde não o seja"."

2.1 Seguros

O seguro pode ser entendido como a formalização num contrato da obrigação da seguradora, mediante o pagamento de um prémio, a compensar o segurado pela eventual perda financeira aquando da ocorrência de um sinistro coberto durante o período da apólice.

Tal como refere (Gilberto, 2012) é possível perceber a grande importância dos seguros no dia-a-dia atual:

> "[...] podemos considerar que no mundo atual não se pode ter desenvolvimento económico sem uma indústria seguradora forte que possa garantir a cobertura de riscos que as empresas e/ou particulares não têm capacidade para suportar apenas por sua conta, caso os sinistros se venham a concretizar."

Normalmente um seguro pode ter diferentes fundamentos:

- **Mutualismo** – grupo de pessoas com interesses em comum que constituem uma reserva financeira para cobrir o risco de um sinistro;
- **Incerteza** – possibilidade de ocorrência de um evento;
- **Previdência** – proteção das pessoas em relação a si ou aos seus bens.

De seguida apresentam-se alguns dos principais termos utilizados:

Proposta (*Proposal*) – trata-se do documento preenchido pelo corretor ou segurado no qual se encontram estipuladas as condições do cliente e propostas da seguradora;

Contrato (*Contract*) – formaliza as obrigações das partes envolvidas (segurado e seguradora). Contêm os detalhes das condições aceites e informação como: nome e endereço do segurado e da seguradora, objeto do seguro, valor do prémio, coberturas, condições de pagamento, entre outra informação;

Apólice (*Policy*) – a emissão da apólice implica a aceitação da proposta e do contrato. Trata-se de um documento emitido pela seguradora que formaliza a aceitação do risco;

Prémio (*Premium*) – valor a pagar à seguradora pelo risco que esta vai cobrir. O valor do prémio é calculado com base em vários fatores relacionados com o risco a cobrir;

Sinistro (*Claim*) – trata-se da ocorrência do risco previsto no contrato do seguro, pode originar prejuízo para o segurado ou beneficiário. O sinistro pode ser: total – refere-se à perda total do objeto segurado; parcial – danos parciais no objeto segurado;

Franquia (*Deductible*) – parte da indemnização que fica a cargo do segurado aquando da ocorrência de sinistro ou da ativação do seguro em diversos cenários;

Risco – probabilidade de um evento inesperado ocorrer, normalmente associado a danos materiais e/ou pessoais e prejuízos económicos;

Seguradora – entidade jurídica que assume os riscos definidos no contrato, recebendo para tal um prémio de acordo com o risco e as coberturas. É a entidade legalmente autorizada a exercer a atividade seguradora e que subscreve com o tomador o contrato;

Tomador – é a pessoa singular ou coletiva, que celebra com a seguradora o contrato seguro, sendo o responsável pelo pagamento do prémio;

Segurado ou Pessoa Segura (num seguro do ramo Vida) – juridicamente é a pessoa sobre a qual recai o risco. É a pessoa cuja vida, saúde ou integridade física se segura;

Beneficiário – pessoa singular ou coletiva a favor de quem reverte a indemnização ou parte conforme descrito no contrato de seguro.

Os seguros estão divididos em dois grupos: seguros do ramo vida e seguros do ramo não vida. Os seguros do ramo vida são os seguros de vida risco que cobrem situações de morte e invalidez, os seguros financeiros como Planos de Poupança Reforma e outros produtos de poupança. Os

seguros do ramo não vida são todos os outros. De seguida apresenta-se mais detalhe sobre o ramo vida.

2.1.1 Ramo Vida

Sob o termo geral de Seguro de Vida apenas estão cobertos os riscos de morte e sobrevivência da pessoa segura num prazo determinado. O montante das prestações é exatamente convencionado na apólice e não o valor justo dos danos resultantes do sinistro.

No seguro de Vida admite-se a existência de seguros múltiplos, podendo o segurado acumular vários contratos sobre o mesmo risco e o beneficiário receber a soma de diversas prestações.

Um seguro de vida apresenta alguns elementos fundamentais, como apresentado esquematicamente na Figura 1:

Figura 1 – Elementos fundamentais de um seguro do ramo vida

Fonte da figura: Documentação interna e proprietária do grupo msg life – Seguros

Existem também coberturas complementares – riscos adicionais que podem ser adicionados ao contrato como por exemplo: incapacidade temporária, invalidez permanente por doença, acidente por circulação, entre outros.

2.1.2 Processos

Na preparação e implementação de um novo produto de seguro existem diversas fases e processos (Debbie, 2008). De uma forma geral essas fases podem ser identificadas como sendo:

- **Inicialização do projeto** – constituição da equipa e distribuição de tarefas, definição métodos e processos de trabalho, definição de datas para o projeto;
- **Definição do produto** – durante esta fase a equipa define as especificações do produto e o mercado alvo com base na competição, normas reguladoras e nos possíveis ganhos;

- **Revisão** – a especificação do produto elaborada é revista pelos intervenientes necessários: *underwriters*, atuários, advogados, *marketers* e outros profissionais do ramo;
- **Submissão da proposta de produto** – submissão de todos os documentos e formulários, que especificam o novo produto, para as entidades responsáveis. Existem também tarifas a pagar a essas entidades;
- **Aprovação** – após a submissão do produto para avaliação se o produto obedecer a todas as normas é aprovado e pode ser comercializado.

A modelação de produtos de seguro pode ser entendida como um processo de análise do qual resulta uma especificação precisa e detalhada de um produto pronto a ser comercializado e administrado.

A fase de definição do produto e modelação envolve uma grande componente de análise e é composta pelos seguintes processos: modelar a estrutura; modelar os objetos envolvidos (partes envolvidas e riscos); modelar os dados necessários; modelar as regras aplicáveis; modelar os cálculos (prémios e benefícios).

Alguns dos sistemas de seguro que dependem da especificação do produto são:

- Sistema de administração de apólices;
- Sistemas intermediários de gestão de compensações;
- Sistemas de registo de sinistros;
- Sistemas de gestão de conta corrente;
- Sistemas de gestão de segurados.

Todo o processo de venda e manutenção de um seguro implica a realização de algumas operações desde o momento da venda até ao fim do contrato.

Durante o processo de venda de um produto de seguro uma companhia de seguro ou vendedor de seguros necessita numa fase inicial de recolher os dados mínimos do segurado para ser possível realizar uma simulação do valor de prémio a pagar para cobrir os riscos identificados. Com base nos benefícios e no prémio a pagar o cliente pode optar pela celebração de um contrato, este processo implica a recolha de dados adicionais e a emissão de uma apólice, podendo este processo ser rápido ou mais demorado dependendo do valor do risco em causa.

Durante o período da apólice existe a necessidade de uma gestão dos pagamentos do prémio que pode ser realizado em diferentes formatos. Durante este período podem ocorrer sinistros e seu subsequente registo e processamento para determinação se este está coberto pelo risco e o segurado tem direito a receber a indemnização.

2.2 msg life Iberia

A msg life Iberia é uma entidade que se posiciona na área da consultadoria e implementação de *software* para o setor segurador, e que nasceu com o intuito de agilizar e inovar a simulação, gestão e comercialização de produtos de seguros do ramo vida, não vida e saúde.

Citação do Jornal Económico (Económico, 2015) no qual se responde à pergunta "Como surgiu a msg life em Portugal":

> "Inserida no Grupo msg – líder tecnológico Europeu especializado no setor segurador com mais de 35 anos de experiência – a msg life Iberia concebe, desenha e desenvolve soluções tecnológicas com impacto em diferentes pontos da cadeia do negócio segurador, desde criação do produto até à distribuição de seguros. A operar em Portugal desde Maio de 2012, a msg life Iberia é responsável pela representação comercial das soluções tecnológicas do Grupo msg no mercado Ibérico, assim como pela criação e implementação de soluções tecnológicas para todo o mundo, sobretudo Estados Unidos, Alemanha e América Latina, posicionando-se como centro de competências para a distribuição de Seguros a nível mundial."

As soluções desenvolvidas tentam cobrir as necessidades do ciclo de vida das seguradoras. Um dos grandes produtos trata-se de uma plataforma de gestão de produtos multi-ramo que permite centralizar as regras de negócio e permite a comunicação com diferentes sistemas, a Product Machine. Esta plataforma permite a definição das regras e critérios que constituem um produto de seguros, podendo ser classificada como um *backoffice* para gestão e definição de produtos. Como solução de *frontend* para a Product Machine a empresa desenvolve a aplicação denominada Sales & Service, constituindo uma plataforma integrada para a distribuição de seguros. O Sales & Service centra-se no cliente e no processo de venda e permite tornar a distribuição de seguros eficiente e escalável.

A msg life Iberia encontra-se atualmente a desenvolver vários projetos para apoiar as seguradoras na melhoria do tempo necessário para a comercialização de novos produtos. Tirando partido dos vários canais e dispositivos de distribuição disponíveis de forma a aumentarem as receitas potenciais através de mecanismos de *up-selling*[3] e *cross-selling*[4].

2.3 Product Machine

Um dos produtos desenvolvidos e proprietários da msg life é a Product Machine, desenvolvida pela FJA US ao longo de mais de quinze anos.

[3] Trata-se de uma estratégia utillizada para aumentar as vendas através da venda de um produto mais caro que apresenta outras opções ou recursos tornando-o superior ou com mais qualidade.
[4] Técninca de vendas que tem por objetivo estimular o cliente a comprar outros produtos e serviços que complementam a compra inicial.

A Product Machine é uma solução que centraliza as regras dos produtos de seguros dos ramos Vida, Não Vida e Saúde. Esta plataforma permite a interação com os sistemas envolvidos no processo de definição, distribuição e gestão dos produtos e contratos.

Nesta plataforma é armazenado um catálogo de todos os produtos, coberturas, benefícios, regras e cálculos para todos os tipos de produtos de seguros, desta forma toda esta informação encontra-se disponível e facilmente acessível para todos os sistemas, departamentos e pessoas. O objetivo é simplificar o processo de desenvolvimento, configuração e distribuição de produtos de seguros.

Após esta breve introdução sobre aquilo que a Product Machine tenta disponibilizar torna-se necessário fornecer alguns detalhes sobre a sua arquitetura e funcionamento de forma a esclarecer como os objetivos a que se propõem são atingidos. Toda a informação apresentada baseia-se na documentação interna e proprietária da msg life.

2.3.1 Arquitetura

A Product Machine é uma aplicação desenvolvida na linguagem Java e baseia-se em alguns conceitos dos modelos de desenvolvimento Model Driven Engineering e Model Driven Development. Existe um grande foco na modelação e geração de código a partir desses modelos. A Product Machine utiliza tecnologias como EMF – Eclipse Modeling Framework, OCL – Object Constraint Language e QVT – Query/View/Transformation para tirar partido da geração de artefactos de *software* a partir da modelação e definição de modelos de acordo com um metamodelo previamente definido.

A Figura 2 apresenta as três áreas distintas que formam os sistemas e componentes da Product Machine:

- ***Define*** – esta área corresponde à definição de produto. O sistema da Product Machine que permite a modelação e representação dos detalhes do produto é o MWB - Modeling Workbench. Vários departamentos e profissionais podem utilizar esta ferramenta de forma colaborativa. Este processo tem como resultado a criação de um catálogo de produtos, coberturas, regras de negócio e cálculos para diversos cenários;
- ***Develop*** – como resultado do trabalho desenvolvido no MWB é possível gerar os serviços de produto (Product Services). Estes serviços constituem o sistema executável da Product Machine. Os principais serviços disponibilizados são:
    - **Calculation**: serviço para obtenção de valores do prémio e de benefícios;
    - **Validation**: serviço para validar a consistência e coerência dos dados;
    - **Functional State**: obtenção de dados de um componente do produto de acordo com o contexto atual;
    - **Report**: construção e distribuição de documentos apropriados;
    - **Information**: apresentação de produtos disponíveis e obtenção da especificação de um produto;
    - **Domain**: obtenção de valores válidos para atributos de introdução de dados.

- ***Distribution*** – os serviços da Product Machine são disponibilizados para múltiplas aplicações e para vários canais de distribuição. São disponibilizados utilizando a arquitetura SOA - Service-Oriented Architecture. Os serviços podem ser customizados de acordo com requisitos específicos de cada um dos canais, as customizações podem ser ao nível da disponibilidade de produtos e elementos desses produtos, regras, gamas de valores possíveis e definição de diferentes fórmulas de cálculo.

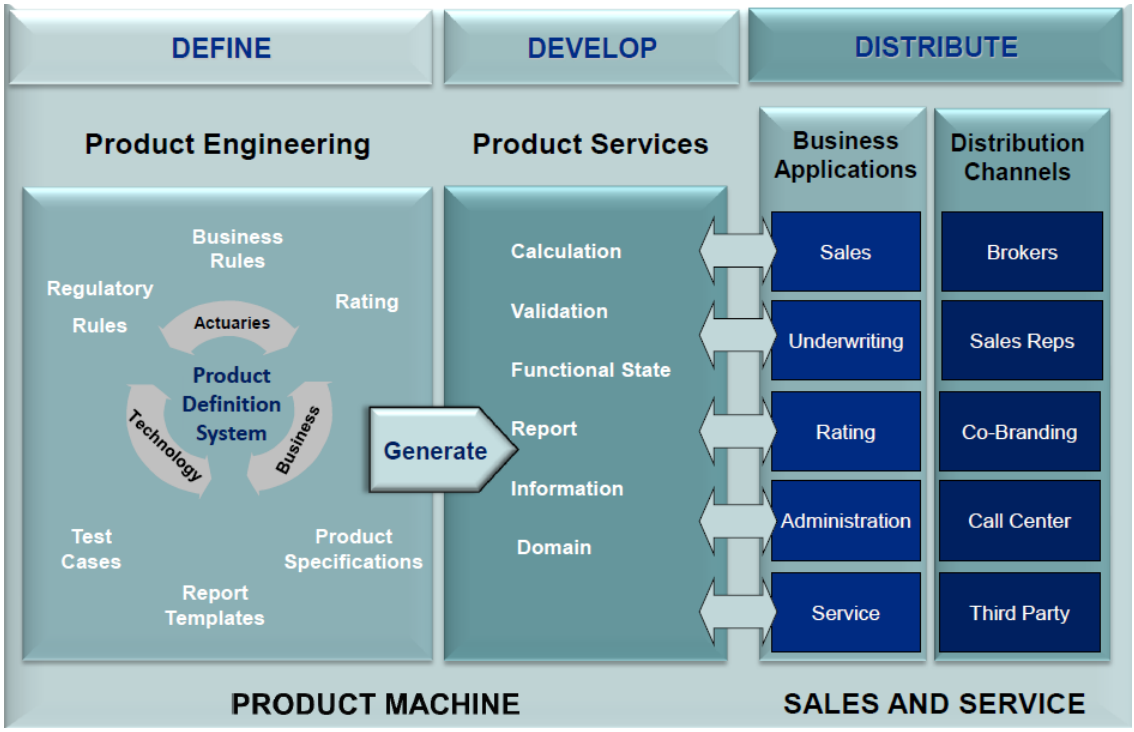


Figura 2 – Arquitetura simplificada da Product Machine

Fonte da figura: Documentação interna e proprietária do grupo msg life – PM4 Training Module 2013

A visão apresentada da arquitetura apresenta a forma como os componentes e sistemas se interligam de forma simplificada. Como se pode observar na Figura 3 os modeladores de produto utilizam o Modeling Workbench para definir o produto, como resultado são geradas classes Java de acordo com o trabalho prévio do *developer* e com os *templates* de código desenvolvidos. Essas classes Java representam os serviços disponibilizados pela Product Machine e que podem ser consumidos por aplicações de vendas e administração para serem utilizadas por agentes de vendas, atuários entre outros profissionais de seguros. Na Figura 4 é apresentada uma perspetiva mais técnica do mesmo processo, de salientar que o Modeling Workbench utiliza um metamodelo de produto previamente definido e permite definir um modelo de produto e um modelo designado de BOM - Business Object Model.

A Product Machine utiliza três tipos de modelos:

- **Product Meta-Model** – especifica o conjunto de ferramentas de modelação (tipos de modelos) utilizado no MWB e disponível para os modeladores utilizarem na definição das especificações dos produtos de seguros;
- **Product Model** – representa a especificação do produto modelada de acordo com o Product Meta-Model;
- **Business Object Model** – representa a informação e os dados de apólices reais entre o segurado e a seguradora, de acordo com as especificações definidas no Product Model.

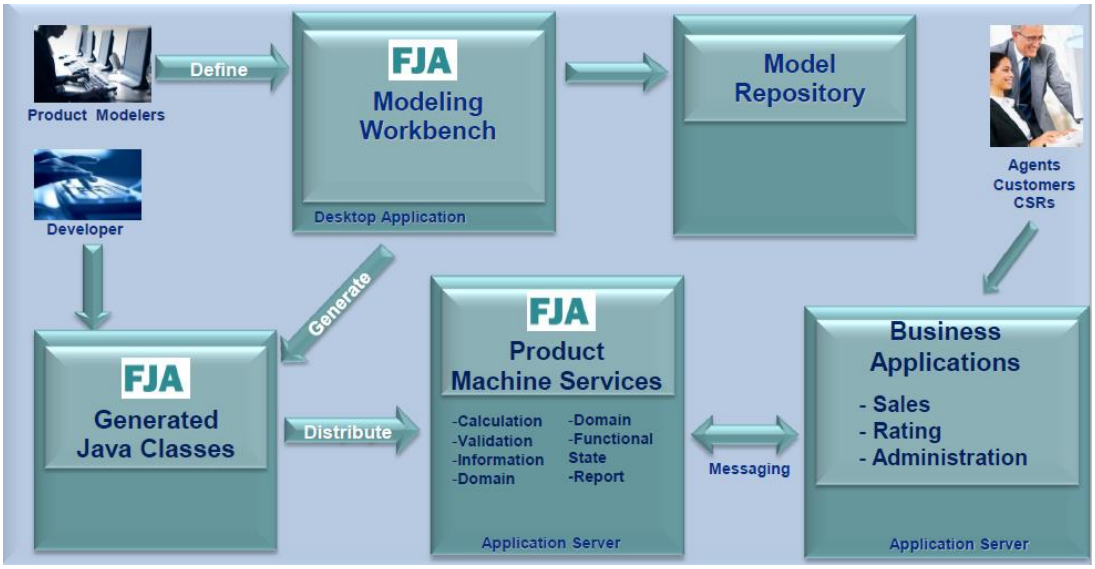


Figura 3 – Detalhe de funcionamento do MWB da perspetiva de um modelador e de um *developer*

Fonte da figura: Documentação interna e proprietária do grupo msg life – PM4 Training Module 2013

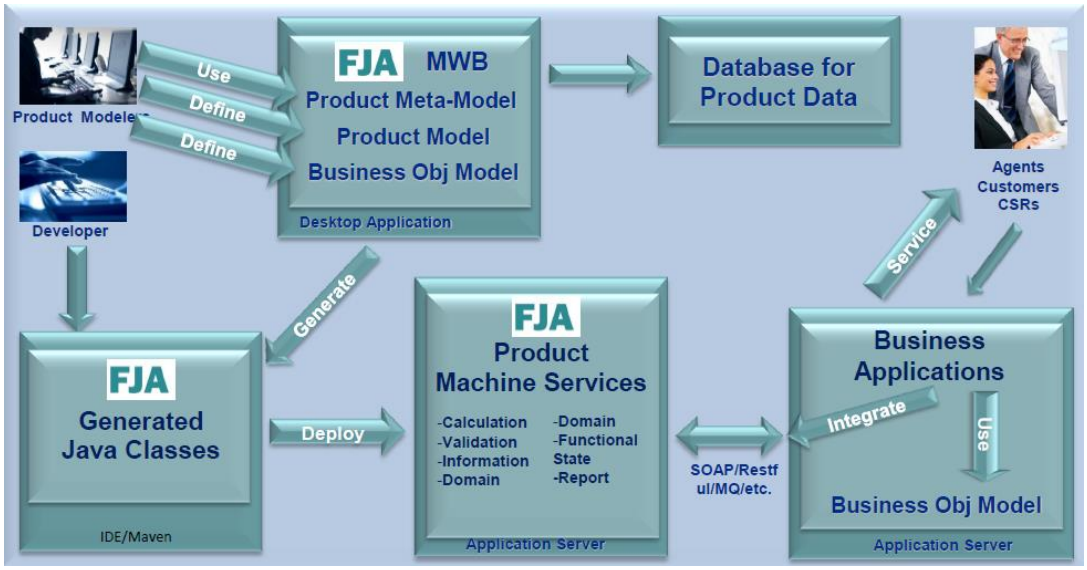


Figura 4 – Detalhe técnico de funcionamento do MWB, particularmente da definição do modelo

Fonte da figura: Documentação interna e proprietária do grupo msg life – PM4 Training Module 2013

2.3.2 Dinamismo e Variabilidade

A Product Machine disponibiliza uma API - Application Programming Interface externa para integração com outros sistemas e aplicações para a comercialização dos produtos definidos.

Como referido anteriormente a Product Machine trabalha com três tipos de modelo diferentes o Product Meta-Model, o Product Model e o Business Object Model. Cada um desses modelos é definido e utilizado em momentos diferentes. O Product Meta-Model é definido na ferramenta CW - Customization Workbench e permite configurar o Modeling Workbench com os tipos de modelos disponíveis. O Product Model é definido através do MWB, representa a especificação do produto e deve obedecer às regras de modelação definidas pelo Product Meta-Model. O Product Model define explicitamente a estrutura possível do BOM. O BOM é o modelo utilizado para permitir a comunicação com os sistemas externos, os objetos (DTO – Data Transfer Object) utilizados na camada de serviços são representações um para um com o BOM e designam-se por BOAdaptable.

Um dos serviços mais relevantes para perceber a capacidade de variação da PM é o Information Service, este serviço permite obter a definição de um determinado produto. Definição essa que pode ser composta por diferentes componentes dependendo do contexto que for enviado para esse serviço. Ou seja, uma aplicação externa invoca o serviço e envia os dados mínimos necessários, de acordo com esse contexto a PM valida se determinado componente do produto deve ser retornado ou não.

Para obter esta variabilidade e dinamismo a PM utiliza os conceitos apresentados na Figura 5 que podem ser descritos:

- **CR - Composition Rule** – permite definir a relação entre componentes do produto e a sua variação. Define a relação pai filho entre os componentes;
- **CRV - Composition Rule Variation** – gere a variação de uma Composition Rule, podem existir várias CRV para uma CR. Define a cardinalidade entre os componentes. Se o CRV se encontrar ativo então a CR está disponível e como tal o componente filho é disponibilizado;
- **SC - Selection Criteria** – critério de seleção de acordo com vários critérios que permite ativar o CRV, o critério pode ser a definição de um intervalo de tempo (data início e fim), localização geográfica, tipo de produto, dependente de valores do contexto, entre outos;
- **Fx - Formula** – execução de uma ou mais fórmulas definidas pelo modelador, se o resultado da sua execução for verdadeiro o CRV correspondente é ativado.

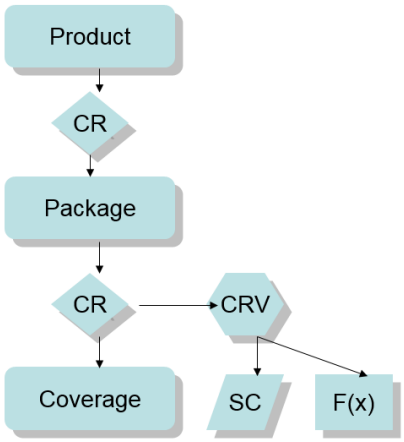


Figura 5 – Conceitos de variação utilizados na Product Machine

Fonte da figura: Documentação interna e proprietária do grupo msg life – PM4 Training Module 2013

Diferentes tipos de valores de domínio existem na PM, permitindo a definição de diversos tipos de dados para um determinado atributo de um componente do produto. Os tipos de domínio são:

- **Continuous Entry** – gama de valores com mínimo, máximo e valor de incremento. Possível definir vários para um mesmo domínio;
- **Filter** – lista de valores possíveis, obtidos de várias fontes possíveis (especificação direta de lista de valores, base de dados);
- **Numeric Entry** – valor numérico;
- **String Entry** – texto possivelmente controlado por expressão regular.

Sobre estes tipos de valores de domínio é possível executar fórmulas para determinar valores por defeito, seleção de valores possíveis de uma lista, expressões regulares e valores mínimos e máximo.

O conceito de Version Information da Product Machine permite suportar a variação da definição do Product Model para um determinado período de tempo e de acordo com vários outros critérios: geográficos; canal de distribuição; qualquer outro critério específico. Esta variação pode ser utilizada em vários níveis do produto: disponibilidade de componentes do produto; valores de domínio; fórmulas de cálculo; regras de validação; entre outros.

Com a utilização dos mecanismos apresentados a PM permite definir e customizar produtos dinâmicos. Como tal o BOM resultante apresenta uma grande variabilidade de componentes.

2.4 Symass

As principais vantagens do Symass são a centralização de informação e a sua administração num sistema único e a redução da necessidade de manutenção. Trata-se de uma aplicação desenvolvida em Java e que utiliza tecnologias *open source.*

A aplicação permite a realização de simulações de seguros do ramo vida e não vida e o cálculo do prémio de um produto, toda a informação da definição e especificação do produto até às regras necessárias para efetuar o cálculo são geridas pelo Symass internamente.

2.4.1 Arquitetura

A arquitetura do Symass compreende diferentes camadas: camada de lógica de negócio; camada de persistência; camada de serviços e camada de apresentação.

A camada mais relevante de analisar é a camada de serviços, pois é com base nesta camada que uma aplicação externa tem a possibilidade de obter a definição do modelo utilizado pelo Symass para representação de um produto de seguros.

Todos os serviços permitem o retorno de mensagens de erro que possam ocorrer durante a sua execução.

Os serviços disponibilizados podem ser separados em três grupos: serviços de manutenção da informação do segurado; serviços de produtos vida e serviços de produtos não vida.

No grupo dos serviços dos produtos de vida existem os seguintes serviços disponíveis:

- Operações CRUD – Create, Reade, Update and Delete para os diferentes componentes de um produto;
- Operações de negócio como:
    - Confirmação de um contrato;
    - Cálculo do prémio;
    - Criação de apólice com base num contrato;
    - Gestão de pagamentos de prémio;
    - Simulação da evolução do produto;
- Pesquisa de contratos.

O Symass disponibiliza o *schema* no formato XSD – XML Schema Definition de todos os modelos que utiliza internamente, ou seja, a estrutura de todos os modelos existentes no Symass pode ser obtido através da chamada a um serviço disponível.

2.5 Sales & Service

Como exposto no subcapítulo 2.3.2 Dinamismo e Variabilidade com base na capacidade de definição de modelos extremamente dinâmicos e genéricos obtida através da utilização da

Product Machine é possível compreender as dificuldades associadas ao desenvolvimento de um *frontend* capaz de apresentar esse mesmo dinamismo e que permita a reutilização de componentes de apresentação mesmo utilizando modelos diferentes.

Para tentar atingir esse objetivo a msg life tem vindo a desenvolver a plataforma Sales & Service. O objetivo principal do S&S é permitir um aumento de produtividade, redução de custos e aumento geral da eficácia no processo de venda e distribuição de seguros em diferentes canais e diferentes ramos de seguros (Vida e Não Vida). Para atingir esses objetivos o S&S permite atualmente a integração de diferentes modelos definidos pelo sistema de definição de produtos Product Machine e desta forma permite a reutilização de componentes e retorno de investimento.

A filosofia por trás da plataforma é a reutilização de código e componentes para permitir o desenvolvimento de um só produto dinâmico e reutilizável. Com uma base de código comum e que possibilite a customização para as várias soluções dos clientes.

Toda a informação apresentada baseia-se em documentação interna e proprietária do grupo msg life.

2.5.1 Conceitos Base

Para facilitar o entendimento da arquitetura e funcionalidades do Sales & Service é necessário compreender alguns conceitos e termos base utilizados.

O Sales & Service inclui uma série de conceitos inovadores que permitem potenciar a definição de produtos tradicional para criar um plataforma de vendas e de serviços mais competitiva. Na Figura 6 são apresentados os conceitos mais importantes utilizados na aplicação e a forma como estes se relacionam. Os conceitos apresentados são:

- **Product** – aquilo que uma companhia de seguros vende. Normalmente é representado por uma estrutura em árvore complexa e com muitos níveis, composta por vários componentes distintos e regras de negócio que formam a definição de um produto;
- **Marketable Offer Component** – possível método de customização de um determinado produto, através da definição de novos valores por defeito, valores possíveis e disponibilidade de certos componentes como por exemplo coberturas;
- **MO - Marketable Offer** – agregação de vários Marketable Offer Component. Existem dois tipos de MO: **Alternative** – para criar uma oferta na qual várias simulações são apresentadas ao utilizador com diferentes opções de coberturas para facilitar a comparação do valor de prémio para cada opção apresentada ao cliente; **Bundled** – para criar uma oferta que pode ser composta por simulações de diferentes produtos que fazem sentido serem vendidos em conjunto e que representam uma oportunidade para o cliente;

- **Sales Process Design** – customização do fluxo de execução do processo de venda, da interação com o utilizador, do *layout* e aspeto da aplicação para os possíveis canais e dispositivos de distribuição;
- **Quote** – simulação de um produto de seguro, a informação necessária sobre o cliente e as suas opções relativamente ao produto em questão. Trata-se de uma estrutura em árvore representativa do produto escolhido. É com base nesta informação que é obtido o valor do prémio;
- **Illustration** – uma oportunidade de negócio. Agrega diferentes *quotes* utilizadas durante o processo de venda para fornecer diferentes opções para o cliente.

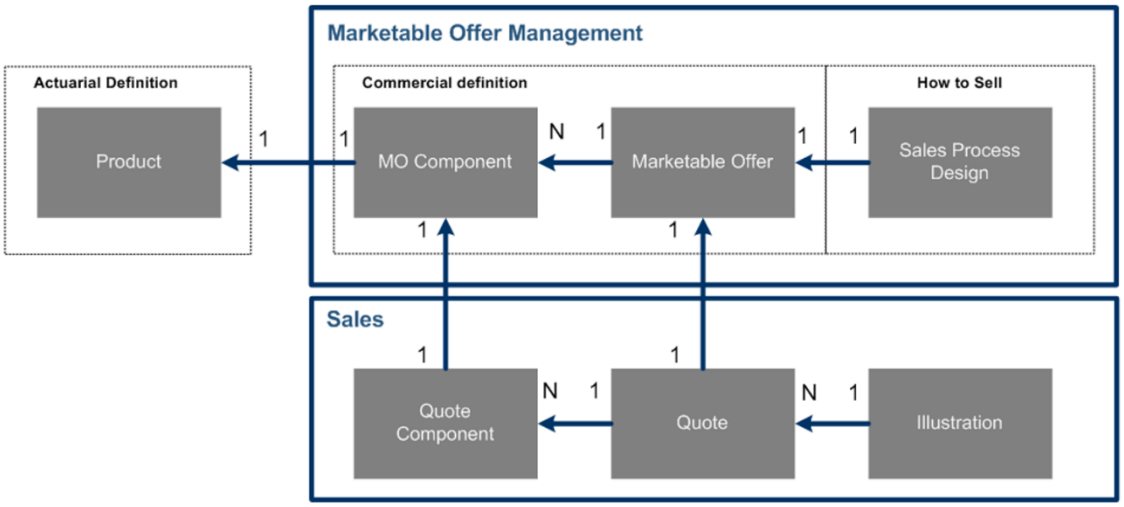


Figura 6 – Conceitos e camadas principais do Sales & Service

Fonte da figura: Documentação interna e proprietária da msg life Iberia – Sales & Service Tech Overview 2016

Os conceitos apresentados agrupam-se em torno de quatro blocos com responsabilidades específicas:

- **Actuarial definition** – a definição atuarial dos produtos disponíveis no *portfolio* da companhia de seguros. Esta é a camada base sobre a qual as restantes camadas e componentes são construídos, trata-se de uma camada externa ao S&S e pode ser o motor de definição de produtos que a companhia de seguros utiliza ou no caso de esta não possuir um PDS ou desejar é utilizada a Product Machine da FJA como PDS;
- **Commercial definition** – camada responsável pela customização e agregação dos produtos atuariais no formato pretendido para alinhar a oferta com a possível procura;
- **How to Sell** – definição da interação com o utilizador, customização dos *layouts*, aspeto e formato de apresentação (formulário ou *wizard*) de forma a alinhar a apresentação do produto com diferentes canais e dispositivos;

- **Sales** – camada na qual as simulações são apresentadas ao cliente e na qual o utilizador interage com a aplicação de forma a preencher os dados necessários para obter a cotação e prosseguir com o processo de venda.

2.5.2 Arquitetura

O S&S segue uma abordagem modular para possibilitar a escolha dos módulos necessários pelo cliente. Tal como se observa na Figura 7 os principais módulos são:

- **Sales Module** – módulo que contém as funcionalidades para venda de seguros, permite simplificar todo o processo de comercialização de seguros. Os principais componentes deste módulo são:
    - **Quotation** – gestão de oportunidades de negócio (Illustration) que pode conter múltiplas quotes de produtos diferentes; comparação de diferentes quotes; gestão de quotes entre vários utilizadores através de mecanismos de autorização e proteção de *portfolio;* impressão e/ou envio de documentos necessários para o processo de venda; emissão de apólices através da integração com sistema de definição de produtos.
    - **Templates** – armazenamento de *quotes* para reutilizar, através da criação de *templates* de *quotes* com campos pré-preenchidos;
    - **Finantial Projections** – possibilidade de projetar a evolução ao longo do tempo do prémio a pagar e dos benefícios de um dado produto. Facilitando a visualização do produto tanto para o cliente como para o vendedor.

  Os ecrãs deste módulo adaptam-se dinamicamente de acordo com a estrutura do produto utilizado (até certo nível). A funcionalidade de Quotation toma em conta o processo de venda configurado para apresentar a informação correta e da forma correta, este aspeto é importante pois o S&S pode ser visualizado em diferentes tipos de dispositivos (*desktop*, *tablets*) e o processo de venda pode necessitar de ser diferente de acordo com o dispositivo;
- **Marketable Offer Management Module** – módulo da versão aumentada do S&S que possibilita a configuração de novos produtos baseados em componentes de produtos existentes. Um novo produto obtido desta forma é designado por Marketable Offer. Uma MO pode ter associado um período de tempo durante o qual se encontra disponível o que permite criar campanhas promocionais de produtos. Os principais componentes deste módulo são:
    - **Marketable Offer Builder** – define quais os produtos que irão constituir uma Marketable Offer. Definição da Marketable Offer como sendo do tipo **Alternative** ou **Bundled**;
    - **Sales Process Designer** – permite que gestor de distribuição ou de marketing configure o aspeto visual e o qual o processo de venda para uma MO, em termos de canais e dispositivos a utilizar. A configuração inclui definir a informação a obter do cliente, quais os ecrãs a apresentar de acordo com a estrutura da Marketable Offer, quais os conceitos de negócio (Coverage,

Insured Person, Premiums) a incluir em cada área do ecrã e qual a forma de apresentação (*card*, *list*, *table*) desses conceitos.

- **Service Module** – módulo que permite a integração com sistemas do cliente para obtenção de funcionalidades pós-venda. As funcionalidades deste módulo implicam a integração com outros sistemas, não existe solução base incluída.

Figura 7 – Principais módulos do Sales & Service

Fonte da figura: Documentação interna e proprietária da msg life Iberia – Sales & Service Tech Overview 2016

Para além dos diferentes módulos o S&S também existe em duas versões CORE (orientado ao produto) e ENHANCED (orientado a Marketable Offer). A versão CORE utiliza diretamente produtos definidos pelo motor de definição de produtos utilizado e como se pode observar na permite:

- Distribuição de produtos em múltiplos canais;
- Cotação de várias *quotes* (simulações) de diferentes produtos;
- Projeções financeiras;
- Avaliação de risco;
- Utilização de *templates* quotes pré-preenchidas;
- Disponibilidade de produtos;
- Pesquisa de informação;
- Geração de diferentes tipos de documentos;
- *Backend* extensível (existência de pontos de extensão de serviços);
- *Frontend* extensível (através da utilização de *plugins*);

- Possibilidade de customizar os diferentes ecrãs, campos apresentados e o aspeto da página. Através da implementação de um projeto específico de cliente sem utilização de editor para utilizador.

A versão ENHANCED permite utilizar os produtos definidos pelo motor de definição de produtos utilizado e criar novas MOs com base nesses produtos. Esta versão acrescenta as seguintes funcionalidades:

- Marketable Offer Builder – possibilita a combinação de produtos, a visibilidade de determinados campos nos ecrãs, entre outras customizações;
- Sales Process Designer – definição de apresentação de ecrã como formulário único ou em modo *wizard,* escolha de diferentes *layouts* e *widgets* para apresentação de conteúdos;
- Cotação de Marketable Offers.

A plataforma S&S necessita de integrar diferentes sistemas por instalação de cliente, ou seja, dependendo dos sistemas que o cliente utiliza é necessário proceder à integração desses sistemas. Nesse sentido existem diferentes tipos de *adapters* que permitem a especificação de interfaces bem definidas e a obtenção de um baixo acoplamento. Tal como representado na Figura 8 os pontos de integração são normalmente ao nível da utilização de soluções existentes de autenticação e autorização através da definição de um Authz Adapter, utilização de PDS de clientes através da utilização de um PDS Adapter ou com um sistema de gestão de apólices definindo um PAS – Policy Administration System Adapter para permitir a transformação de uma *quote* numa apólice através da submissão da apólice para o sistema externo.

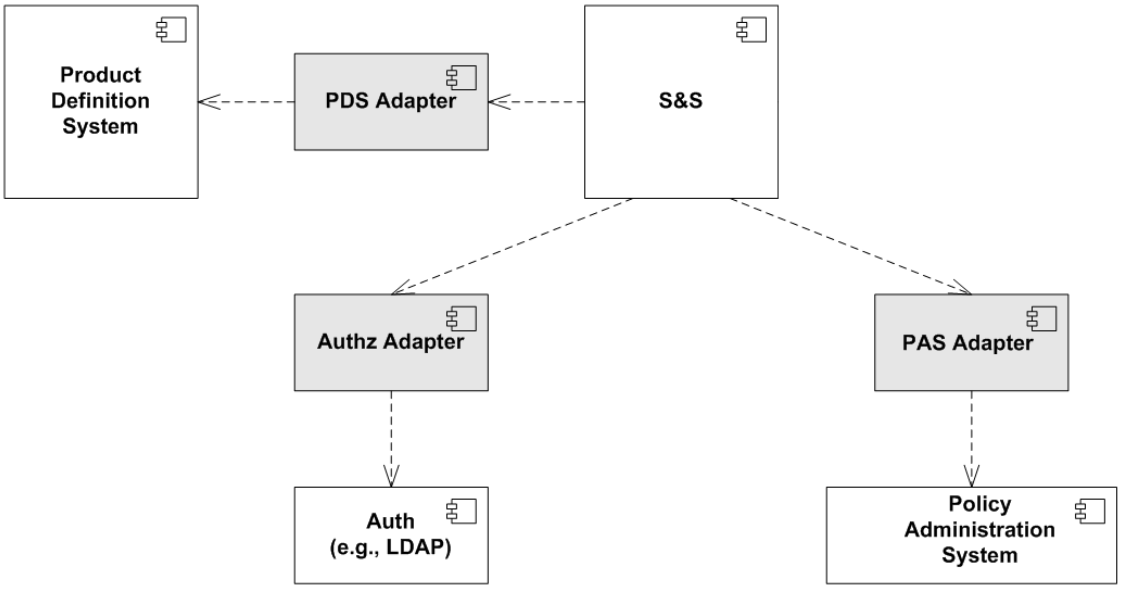


Figura 8 – Principais pontos de integração do Sales & Service

Fonte da figura: Documentação interna e proprietária da msg life Iberia – Sales & Service Tech Overview 2016

O PDS Adapter define um conjunto de interfaces para comunicar com o PDS e é neste *adapter* que é realizada a transformação de e para o modelo canónico do S&S (Internal BOM).

2.5.3 *Design* Técnico e Principais Caraterísticas

Na Figura 9 é possível visualizar a *stack* tecnológica utilizada no S&S. Como se pode verificar a componente web da aplicação utiliza tecnologias como Backbone.js, Marionette, require.js, jQuery, Bootstrap, LESS e a ferramenta pluggable (proprietária da msg life) que permite adicionar comportamento de acordo com uma necessidade específica de um cliente ou diferentes requisitos. Os serviços utilizam Java, Spring e Maven. Outras tecnologias como Selenium, JUnit e Cucumber são utilizadas na criação de testes unitários e funcionais.

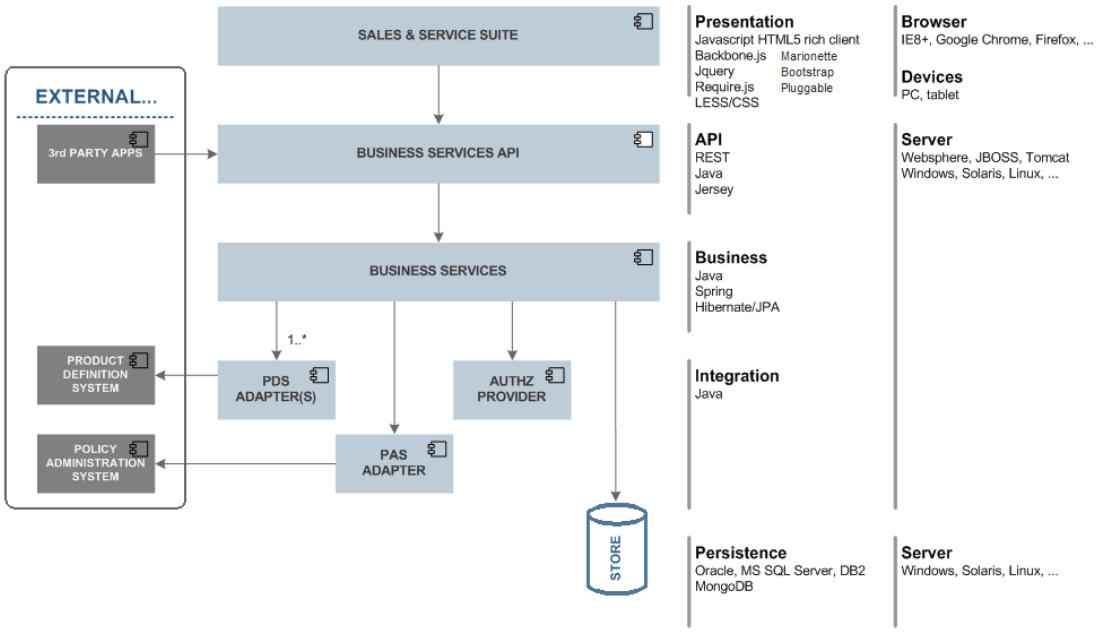


Figura 9 – Tecnologias utilizadas no Sales & Service

Fonte da figura: Documentação interna e proprietária da msg life Iberia – Sales & Service Tech Overview 2016

O *frontend* do S&S foi desenvolvido de forma a permitir um grande nível de flexibilidade no conteúdo a apresentar e na forma de apresentação. O conteúdo a apresentar no ecrã é definido através da definição de metainformação na forma de um DSL – Domain Specific Language em formato XML - Extensible Markup Language designado por Application Flow e Screen Flow. Esta metainformação permite customizar aspetos como:

- Informação a apresentar (*labels*, *inputs*, valores possíveis) e como a apresentar (*select*, *spinner*, *picklist*, entre outras formas);
- Definição de seções para agrupar conteúdos;
- Navegação entre ecrãs e ações possíveis;
- Definição de diferentes layouts através da utilização de diferentes *plugins*.

Na comunicação com o *frontend* é utilizado um DTO designado de Container And Data Holder[5] que contém informação sobre a definição do produto e a metainformação necessária para renderizar um determinado ecrã de acordo com o estado atual do processo de venda. Através da utilização desta informação é possível a apresentação de um ecrã dinamicamente e apresentar as propriedades e atributos dos objetos que constituem o modelo interno.

2.5.4 Limitações identificadas

A arquitetura desenvolvida permite facilmente a definição de um novo Product Definition System *adapter* para permitir a comunicação com outros sistemas de definição de produtos. No entanto o processo de mapeamento entre os possíveis modelos existentes no novo PDS e o modelo interno do S&S apresenta-se como um processo demorado, complexo e que implica trabalho e esforço considerável pelos *developers.* Isto porque normalmente existem diferenças e especificidades consideráveis entre os modelos utilizados.

Neste seguimento é possível compreender que o processo atualmente disponível para integração de novos PDS apresenta espaço para melhoria em termos de simplificação e automatização das suas operações de forma a tornar essa integração mais eficaz e alinhada com a filosofia geral da solução de diminuir o "Time to Market" de novos produtos de seguros.

2.6 Problema

O Sales & Service apresenta as vantagens descritas, através da utilização de um modelo interno único para representar os diferentes conceitos básicos de um seguro. Trata-se de um modelo canónico que contém a informação que representa um produto de seguros. Devido à existência desse modelo é possível generalizar grande parte da aplicação, permitindo a reutilização dos diferentes componentes e a obtenção de uma camada de independência entre a aplicação e os diferentes PDS de forma a permitir a adaptação da aplicação para diferentes clientes e necessidades.

O problema e os requisitos existentes podem ser enumerados de forma resumida:

- **Problema:**
    - Cada PDS possui o seu modelo;
    - Cada especificação de produto é diferente;

[5] Anexos: A Figura 64 – Principais componentes do Container And Data Holder apresenta mais detalhe.

    - Diferentes companhias de seguros que utilizam o mesmo PDS podem ter modelos diferentes.
- **Requisitos:**
    - Utilização de uma base de código comum para todos os clientes;
    - Suportar versões de clientes atuais;
    - Suportar *frontend* dinâmico.

Tal como apresentado no subcapítulo 2.5.4 Limitações identificadas e como se pode compreender a integração com cada um dos diferentes Sistemas de Definição de Produtos é um processo complexo e que apresenta grandes desafios, visto cada produto conter diferentes especificidades e regras de negócio. O sistema atual já permite a integração de diferentes PDS no entanto apresenta ainda um custo de integração considerável, que poderia ser reduzido e até certo nível padronizado.

A questão do mapeamento entre os modelos e o modelo interno apresenta vários desafios técnicos e da própria arquitetura da aplicação. Pois é pretendido atingir um nível elevado de independência para com os sistemas externos mas mantendo a capacidade de resposta às necessidades específicas dos vários clientes e dos seus produtos de seguros.

2.6.1 Integração PDS

O S&S foi desenvolvido, numa fase inicial, focando-se na utilização da Product Machine como motor de definição de produtos. Como exposto anteriormente atualmente a integração de diferentes modelos da PM é realizada através da implementação manual das transformações necessárias num *adapter*.

Com a evolução do projeto tem-se verificado que a integração dos diferentes modelos desenvolvidos na Product Machine apresentam níveis de complexidade diferentes em relação aos mapeamentos necessários. Alguns mapeamentos são idênticos e repetitivos, outros são mais complexos e específicos.

Com a necessidade de integrar um sistema completamente diferente como o Symass é expectável que o tipo de mapeamentos identificados cubram a maioria das necessidades, é possível que novos tipos de mapeamentos possam surgir.

A dificuldade na integração de sistemas diferentes como Symass prende-se na necessidade de suporte de formato diferentes utilizados na definição do modelo e daí surge a necessidade da existência de mecanismos distintos de leitura e tratamento do modelo utilizado no PDS a integrar.

O objetivo principal é permitir simplificar e na medida possível automatizar a leitura do modelo e a execução dos mapeamentos idênticos, permitindo a customização dos mapeamentos mais complexos.

São identificáveis dois tipos de PDS com base na descrição da Product Machine e do Symass, no futuro outros tipos podem ser identificados. Esses dois tipos diferem numa questão fulcral, a Product Machine, devido à sua capacidade de dinamismo, permite a definição de vários modelos distintos. Por seu lado o Symass apresenta um modelo estático (alterações pontuais podem ocorrer). Desta diferença surge a questão: "Será que existem diferentes metodologias de mapeamento mais adequadas a cada um dos cenários ou é possível identificar uma metodologia comum?".

Tendo por base esta questão, os diferentes requisitos apresentados e com o objetivo de apoiar a tarefa de desenho de uma possível solução é realizado um estudo do estado da arte e é realizada uma análise de dois paradigmas diferentes para a resolução do problema. Estas duas tarefas são apresentadas no capítulo 4 Model Driven Architecture e Model to Model Transformation e no capítulo 5 Mapeamento de Objetos.

2.7 Restrições Existentes

A plataforma Sales & Service encontra-se atualmente em ambiente de produção em alguns clientes, ou seja, existe a necessidade de desenvolvimento de novas funcionalidades bem como de suporte e correção de possíveis problemas. Como a base de código utilizada é comum entre as diferentes versões instaladas nos vários clientes, as novas funcionalidades e suporte devem acontecer sobre essa base de código. Igualmente os vários produtos desenvolvidos por esses clientes devem continuar a funcionar. Deste aspeto é possível determinar a primeira restrição, a solução deve permitir o suporte do modelo atualmente existente apenas com pequenas alterações e ajustes.

Outra restrição existente está relacionada com a necessidade de utilizar *software* e/ou tecnologias com determinado tipo de licença que permita a livre alteração e comercialização da solução desenvolvida utilizando essas tecnologias.

3 Análise de Valor

3.1 Introdução

Como refere (Nicola, 2014) a análise de valor "é uma visão global da oferta de produtos e serviços de uma empresa que apresentam valor para o cliente" (tradução própria) sendo assim de forma simples o objetivo de uma análise de negócio clara é identificar a viabilidade e rentabilidade de uma ideia de negócio. Trata-se do ponto de partida para determinar o modelo de negócio a utilizar e desta forma determinar se e quando o negócio irá gerar lucro.

Existem diferentes definições na literatura para o conceito de valor, a seguinte definição de valor dada por (Nicola, 2012) permite perceber um pouco melhor qual o papel do valor no contexto de um negócio:

> "A criação de valor é de extrema importância para qualquer negócio, qualquer atividade de negócio trata-se da troca de um bem ou serviço tangível ou intangível cujo valor é reconhecido e recompensado pelos clientes [...]" (tradução própria)

Como referido na definição de valor, é o cliente que ultimamente avalia o valor e recompensa a criação de valor, no entanto o valor é entendido de forma diferente por diferentes clientes. Ou seja, diferentes clientes têm uma noção diferente do valor do mesmo produto/serviço, a perceção do valor é subjetiva. Tal como (Zeithaml, 1988) sugere o valor percebido pode ser entendido como "a avaliação global do consumidor da utilidade de um produto baseado em perceções daquilo que é recebido e do que é dado" (tradução própria)

Para realizar uma boa análise de negócio ou VP – Value Proposition é necessário agregar vários tipos de informação, como a determinar os clientes alvo, qual o problema que estamos a resolver para esse cliente, qual a solução que oferecemos e qual o valor que estamos a providenciar para o cliente. Com base nesses dados é possível perceber claramente qual o real valor que o produto desenvolvido apresenta e qual o valor percebido pelos diferentes clientes alvo. A partir desse ponto a definição de um modelo de negócio correto e alinhado com a promoção desse valor torna-se bastante mais evidente.

3.2 Processo de inovação

Um processo de negócio e inovação é constituído por diferentes fases e pode ser dividido em três diferentes áreas (Koen, 2002):

- **FFE – Fuzzy Front End:** fase inicial do processo de inovação inclui a identificação de possíveis ideias para um novo produto, é uma fase com um elevado grau de incerteza tanto em termos de prazo como de viabilidade das ideias. Não existe desenvolvimento do produto nesta fase mas sim seleção de ideias viáveis;
- **NPD – New Product Development:** fase de desenvolvimento da ideia identificada na fase anterior, esta fase é caraterizada pela existência de um maior grau de certeza devido à maior organização nas tarefas executadas e com um planeamento definido para atingir uma data de comercialização para o produto a desenvolver;
- **Comercialização:** comercialização do produto desenvolvida durante as fases anteriores.

Durante a fase FFE, tal como descrito, existe um grande grau de incerteza e as práticas e técnicas que permitem uma maior organização na fase NPD não aplicáveis à fase FFE. No sentido de melhorar uma das áreas mais promissoras para a melhoria de todo o processo de inovação, um projeto de investigação realizado em 1998 pela Industrial Research Institute tentou identificar as diferentes técnicas e práticas utilizadas em diferentes organizações durante o processo de inovação. No entanto a falta de uma terminologia comum dificultou esta investigação inicialmente. Para resolver este problema (Koen, 2002) desenvolveu um novo modelo denominado NCD – New Concept Development.

3.2.1 New Concept Development

Para permitir um melhor entendimento do modelo é importante clarificar algumas definições (Koen, 2002):

- **Oportunidade:** uma lacuna tecnológica ou necessidade de negócio identificada por um indivíduo ou organização e que se for corretamente preenchida permite ganhos competitivos ou a resolução de um problema da organização;
- **Ideia:** uma possível solução, pouco detalhada, para o problema ou oportunidade identificado, é a representação inicial e rudimentar do possível produto a desenvolver;
- **Conceito:** representação detalhada do produto a desenvolver, trata-se de uma descrição detalhada das funcionalidades iniciais do produto bem como dos benefícios para o cliente. Existe ainda um conhecimento das necessidades tecnológicas para o desenvolvimento do produto.

O modelo NCD tal como se observa na Figura 10 – Modelo NCD é representado por um círculo constituído por três partes: motor, fatores influenciadores e os cinco elementos internos. Este modelo deve ser interpretado como um modelo relacional e fluído e não um processo linear, isto porque neste modelo as ideais devem fluir, circular e serem iterativamente melhoradas e clarificadas durante a transição entre os cinco elementos internos. No final do processo as ideias são refinadas e evoluem para conceitos.

No modelo existem dois pontos de início para um projeto de inovação, nomeadamente pela **identificação de uma oportunidade** ou pela **geração e enriquecimento de uma ideia**. Os novos conceitos deixam o modelo pelo elemento **definição de conceitos** e entram na fase de desenvolvimento de produto, ou seja, na fase seguinte do processo de desenvolvimento.

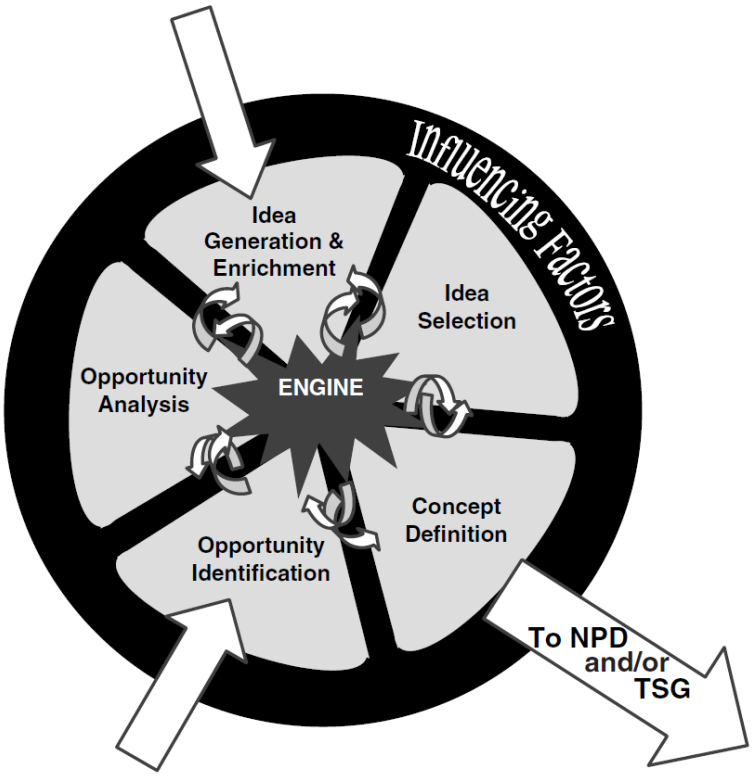


Figura 10 – Modelo NCD (Koen, 2002)

Fonte da figura: (Koen, 2002)

Os cinco elementos chave do modelo são:

Identificação de oportunidades – é neste elemento que as novas oportunidades de negócio são identificadas. As oportunidades identificadas devem estar alinhadas com os objetivos comerciais da organização. As oportunidades podem tentar responder a um problema ou ameaça para a organização, terem por objetivo simplificar processos e operações existentes, tentar aumentar a produtividade ou reduzir os custos operacionais da organização. Adicionalmente as novas oportunidades podem ser derivada por estratégias de marketing para alargar o mercado alvo ou criar novos produtos para introduzir a organização num mercado diferente. No elemento de identificação de oportunidades podem ser aplicados os seguintes métodos ou técnicas (Koen, 2002):

- Definição de roadmaps;
- Análise de tendências tecnológicas;
- Análise de tendências dos clientes;

- Análise da competitividade da oportunidade;
- Estudos de mercado;
- Planeamento de cenários

No contexto da melhoria do mapeamento entre modelos no S&S o elemento de identificação de oportunidades enquadra-se na identificação de uma oportunidade de melhoria do processo de integração de diferentes PDS para permitir um aumento de produtividade e a consequente redução de custos da msg life. Neste sentido a os métodos mais relevantes são:

- Definição de roadmaps;
- Análise de tendências tecnológicas – nomeadamente a identificação de tecnologias que permitem a automatização e melhoria do processo de integração de PDS.

Análise de oportunidades – neste elemento uma oportunidade é analisada para permitir confirmar a sua viabilidade e determinar se a oportunidade se enquadra com os objetivos estratégicos da organização. A análise desenvolvida pode incluir a constituição de grupos de estudo, estudos de mercado e experiências tecnológicas para validar a oportunidade. Os métodos e técnicas a utilizar neste elemento podem ser os mesmos utilizados no elemento de identificação de oportunidades, no entanto para a análise de oportunidades os recursos despendidos com essas metodologias são consideravelmente maiores.

Uma análise para uma oportunidade de grandes proporções normalmente poderia ainda incluir uma equipa de três a cinco pessoas para analisar outros elementos como (Koen, 2002):

- **Enquadramento estratégico:** determinar se a oportunidade se enquadra no mercado da organização e com as suas capacidades, lacunas e fraquezas tecnológicas;
- **Avaliação do segmento de mercado:** trata-se de uma descrição detalhada do potencial segmento de mercado que a oportunidade poderá permitir atingir. Alguns indicadores como tamanho do mercado, taxas de crescimento, quota de mercado dos potenciais competidores são analisados durante este processo;
- **Análise de potenciais competidores:** determinar quais são os maiores competidores identificados no segmento de mercado alvo bem como as suas capacidades e estratégias de desenvolvimento atuais. Analisar quais as características dos produtos necessárias para alcançar vantagem competitiva;
- **Análise das necessidades dos clientes:** tem como objetivo determinar quais as necessidades dos clientes atualmente não satisfeitas pelos produtos existentes.

No contexto do mapeamento de modelos no S&S o elemento de análise de oportunidades consiste na determinação da viabilidade de utilização da nova ferramenta de mapeamento no processo de desenvolvimento atualmente em prática na msg life e também na determinação do alinhamento com a estratégia da empresa. Neste sentido a os métodos a utilizar são os mesmos a utilizar na identificação de oportunidades mas aplicados com maior profundidade de

análise e recursos associados. O enquadramento estratégico seria outra metodologia a utilizar para analisar a ferramenta.

Geração e enriquecimento de ideias – este elemento consiste na criação de ideias e no seu desenvolvimento e maturação através de um processo de enriquecimento que compreende a evolução da ideia, combinação com outras ideias, destruição de ideias, atualização e modificação de ideias. Normalmente uma ideia vai sendo enriquecida com a iteração e obtenção de nova informação durante a execução de todos os outros elementos do modelo NCD. O elemento de geração e enriquecimento pode ser alimentado por ideias provenientes de qualquer pessoa da organização. Neste elemento podem ser aplicados, entre outros, por exemplo os seguintes métodos ou técnicas (Koen, 2002):

- Uma cultura organizacional que incentive os seus colaboradores a partilharem as suas ideias e permita que estes despendam algum tempo na melhoria e validação dessas ideias;
- Diferentes tipos de incentivos para a estimular a criação de ideias (sejam prémios ou reconhecimento);
- Uma plataforma para permitir a partilha das ideias e também facilitar a obtenção de informação de produtos e serviços atuais da organização;
- A existência de uma pessoa com a responsabilidade formal de acompanhar uma nova ideia desde o seu nascimento até à sua análise;
- A existência de algumas métricas capazes de identificar a evolução da geração de ideias;
- Rotação frequente dos elementos das diferentes equipas para maximizar a partilha de conhecimento e diferentes realidades e facilitar o surgimento de novas ideias;
- Mecanismos que permitam a disseminação de competências, conhecimento tecnológico e outras informações importantes dentro da organização.

No contexto do mapeamento de modelos no S&S o elemento de geração e enriquecimento de ideias pode ser entendido como a criação da ideia de tentar automatizar o processo manual de integração de modelos. Neste sentido os métodos mais relevantes são:

- Uma cultura organizacional que incentive os seus colaboradores a partilharem as suas ideias e permita que estes despendam algum tempo na melhoria e validação dessas ideias;
- Diferentes tipos de incentivos para a estimular a criação de ideias (sejam prémios ou reconhecimento);
- Rotação frequente dos elementos das diferentes equipas para maximizar a partilha de conhecimento e diferentes realidades e facilitar o surgimento de novas ideias;
- Mecanismos que permitam a disseminação de competências, conhecimento tecnológico e outras informações importantes dentro da organização.

Seleção de ideias – este elemento consiste na seleção das ideias que apresentam, de acordo com as informações obtidas através dos outros elementos do modelo NCD, melhores perspetivas de se tornarem num conceito de produto viável. Normalmente o maior problema para o processo de seleção relaciona-se com a quantidade de ideias existentes e numa boa capacidade de seleção. Neste elemento podem ser aplicados os seguintes métodos ou técnicas (Koen, 2002):

- Metodologias de análise de *portfolio* baseadas em diversos fatores como:
    - Probabilidade de sucesso tecnológico;
    - Probabilidade de sucesso comercial;
    - Potencial retorno ou recompensa;
    - Alinhamento estratégico.
- Processo formal de seleção de ideias com feedback imediato para os proponentes da ideia;
- Utilização da teoria de opções para avaliar projetos.

No contexto do mapeamento de modelos no S&S o elemento de seleção de ideias do modelo NCD pode se enquadrar na escolha da ideia de automatizar o processo de mapeamento. Neste sentido o método mais relevante é:

- Metodologias de análise de *portfolio* baseadas em diferentes fatores, especialmente probabilidade de sucesso tecnológico, alinhamento estratégico e retorno.

Definição de conceitos – este elemento é o único ponto de saída para o NPD. Para um conceito passar este elemento é necessário obter informação quantitativa e qualitativa que irá permitir que a organização determine se pretende ou não investir no conceito para o desenvolvimento de um novo produto. Diversos tipos de informação tal como (Koen, 2002) indica devem ser compilados, nomeadamente:

> "objetivos; volume da oportunidade e o seu impacto financeiro; um plano de negócios que especifique uma proposta de valor win/win; aspetos de risco tecnológicos e comerciais; aspetos ambientais, de saúde ou de segurança que impeçam a concretização do produto; um plano para o projeto incluindo planeamento de recursos e gestão de tempo" (tradução própria).

Neste elemento podem ser aplicados, entre outros, por exemplo os seguintes métodos ou técnicas (Koen, 2002):

- Abordagens de avaliação de objetivos – definir claramente os objetivos do projeto e os seus resultados;
- Definição de um plano atrativo para a organização, que descreva em que medida o produto pode afetar financeiramente, o volume de mercado e o crescimento de mercado.

No contexto do mapeamento de modelos no S&S o elemento de definição de conceitos apresenta o papel de convencer a msg life das potencialidades da utilização da ferramenta de mapeamento através da clarificação dos potenciais ganhos de produtividade e redução de custos no desenvolvimento e integração com novos PDS. Neste sentido as técnicas apresentadas anteriormente como a definição de um plano atrativo para a organização e a utilização de abordagens de avaliação de objetivos são as mais relevantes.

3.3 Valor para o Cliente

Visto a solução a desenvolver se tratar de um componente da solução global Sales & Service o valor para o cliente é apresentado em duas vertentes, uma como sendo o valor para o cliente da solução Sales & Service e outra como o valor da componente de mapeamento a desenvolver.

3.3.1 Valor da Solução Sales & Service

Os clientes alvo da msg life são companhias de seguros e bancos (que comercializam produtos de seguros).

O setor dos seguros apresenta grandes desafios tais como a complexidade de criação e especificação de um novo produto de seguros, nomeadamente ao nível da definição de todos os seus componentes e regras, na avaliação do risco e na formulação do cálculo do prémio. Para além dessa complexidade durante a criação de um seguro existe a necessidade de comunicação e colaboração entre vários departamentos da companhia de seguros. Tudo isto se resume em demasiado tempo para a criação, distribuição e venda de um novo produto o que implica uma possível perda de competitividade e perdas financeiras.

Sendo assim, as companhias de seguros necessitam de agilizar todo este processo para atingirem um melhor “Time to Market” de forma a responderem às necessidades dos seus clientes de forma competitiva. As companhias necessitam ainda de uma redução de custos operacionais através da melhoria da colaboração e comunicação entre os vários departamentos.

A msg life Iberia através da sua plataforma Sales & Service e Product Machine apresenta uma solução integrada completa para a definição e desenvolvimento de um produto de seguros e a sua comercialização em diferentes canais e dispositivos. A solução permite a centralização de informação num único sistema e facilita a colaboração dos diferentes departamentos uma vez que é orientada a profissionais do negócio e não a pessoal técnico.

Estas caraterísticas da solução permitem a redução do “Time to Market” e a simplificação de todo o processo através da abstração da complexidade inerente ao negócio.

O valor da solução disponibilizada para as companhias de seguros pode ser descrito na forma de benefícios e sacrifícios. São identificáveis duas fases distintas no valor para o cliente: fase de

aquisição e fase de utilização da solução. A fase de aquisição implica um processo de desenvolvimento conjunto para levantamento de requisitos e funcionalidades extra necessárias. Deste modo ao longo do processo de aquisição e numa fase inicial na qual o cliente e a msg life se encontram numa fase de conhecimento, os benefícios são:

- Produto: Solução alternativa – uma vez que a solução desenvolvida pretende responder às necessidades da companhia de seguros através de um acompanhamento próximo;
- Produto: Solução customizável – é durante esta fase que o cliente se apercebe da capacidade de customização da solução e da capacidade da solução para solucionar as especificidades do cliente;
- Serviço: Flexibilidade – o acompanhamento de perto por profissionais com experiência e aptos a responder às necessidades identificadas nesta fase;
- Serviço: Conhecimento técnico – os profissionais apresentam um grande nível de conhecimento da solução e como tal conseguem adaptá-la facilmente aos problemas/necessidades do cliente.

Nesta fase os sacrifícios identificáveis são:

- Valor de aquisição e o preço associado ao desenvolvimento de novas funcionalidades adicionais para responder a requisitos específicos do cliente – para permitir uma solução customizável e alinhada com as necessidades do cliente são necessários desenvolvimentos adicionais e como tal o preço global aumenta;
- Tempo e esforço – nesta fase a participação do cliente no processo de desenvolvimento da solução é considerável e como tal este terá de despender tempo e esforço durante todo o processo.

Numa fase de utilização da solução, os benefícios para o cliente serão um pouco diferentes pois a solução já se encontra desenvolvida e configurada. Durante esta fase o cliente irá obter mais benefícios da sua utilização, como:

- Produto: Solução alternativa e customizável;
- Produto: Produto de qualidade – após a fase inicial de adaptação e utilização da solução o cliente consegue perceber que se trata de um produto confiável e com qualidade pois permite obter melhorias visíveis no seu dia-a-dia;
- Serviço: Capacidade de resposta – resposta rápida e acompanhamento na resolução de problemas na utilização da solução;
- Serviço: Conhecimento técnico.

Os possíveis sacrifícios nesta fase são:

- Custos de manutenção e de novas formações – para obter um nível de acompanhamento e de resolução de problemas rápido e eficaz.

3.3.2 Valor Componente Mapeamento

Os clientes alvo do componente de mapeamento são os *developers* e a própria msg life Iberia.

As necessidades do cliente alvo são o aumento da produtividade no desenvolvimento de *software* e o aumento da sua qualidade. No setor do desenvolvimento de *software* o objetivo final é o desenvolvimento de produtos com qualidade, inovadores e alinhados com as necessidades dos clientes, tentando utilizar o menor número de recursos possíveis, nomeadamente horas de desenvolvimento.

A componente de mapeamento tem como objetivo simplificar e automatizar o processo de desenvolvimento de uma das partes da plataforma que atualmente necessita de uma quantidade significativa de esforço.

Sendo assim o valor da solução desenvolvida pode ser descrito em termos de benefícios e sacrifícios de acordo com a fase de aquisição e a fase de utilização.

Durante a fase de utilização inicial do novo componente os benefícios são: Produto: Solução alternativa e customizável – permite a realização de diferentes tipos de mapeamento e a customização desses mapeamentos. Nesta fase o sacrífico identificável é: Custos de aprendizagem: durante esta fase os *developers* necessitam de um período de adaptação à nova componente e durante esse período o desenvolvimento poderá ser menor e como tal os custos de desenvolvimento podem aumentar.

Na fase de utilização os benefícios são: Produto: Solução alternativa e customizável e Produto. Os únicos sacrifícios identificáveis nesta fase são: custos de formação de novos *developers.*

De forma sucinta a proposta de valor deste trabalho é o desenvolvimento de uma ferramenta de mapeamento semi-automatizada de diferentes modelos no S&S para permitir um aumento de produtividade e a redução de custos durante a fase de integração com novos PDS. Com este benefício a msg life pode investir esses recursos na melhoria do seu produto noutras áreas tornando-o ainda mais competitivo.

3.4 Oportunidades de Negócio

O processo de obtenção de um novo cliente é demorado e complexo. Numa fase inicial é necessário obter um primeiro contato com as companhias de seguro para posteriormente ser possível realizar uma série de apresentações com o objetivo de expor as capacidades da solução. Após este processo e se existir interesse na solução é necessária uma fase de análise das necessidades de customização e novos desenvolvimentos para cumprir os requisitos do novo cliente.

Nesse sentido a prospeção de novas oportunidades de negócio apresenta várias etapas. Um dos pilares para obter novos clientes é a presença em diversos meios do setor (feiras promocionais, conferências, revistas do setor). O objetivo desta presença é conseguir um

primeiro contato para iniciar todo o processo e obter a hipótese de realizar uma apresentação da solução de forma presencial com o potencial novo cliente.

A solução apresenta como vantagens principais a melhoria do "Time to Market" e a possibilidade de utilização da solução por profissionais de negócio. As funcionalidades apresentadas pela solução estão alinhadas com os problemas e necessidades identificadas pelas companhias de seguros. Nesse sentido a plataforma Sales & Service apresenta-se como uma excelente oportunidade de negócio junto dos clientes alvo. Através da apresentação da solução como um mecanismo para redução do "Time to Market" e como uma ferramenta para impulsionar o trabalho de modelação e criação de um produto de seguros pelos profissionais que realmente conhecem as especificidades do negócio.

3.5 Modelo Negócio

Normalmente o modelo de negócio é entendido simplesmente como a forma de fazer dinheiro, no entanto um modelo de negócio é muito mais do que isso. Um modelo de negócio inclui o método utilizado para obtenção e retenção de clientes, conhecimento sobre custos operacionais, identificação de parceiros relevantes e canais de vendas, reconhecimento dos recursos disponíveis, entre ouros aspetos. Neste sentido (Osterwalder, 2005) define modelo de negócio como:

> "Um modelo de negócio descreve o valor que uma organização oferece aos seus clientes e ilustra as capacidades e os recursos necessários para criar, comercializar e entregar este valor e para gerar fluxos de receitas sustentáveis rentáveis." (tradução própria)

Na Figura 69 é possível observar o modelo de negócio para a solução integrada plataforma Sales & Service e Product Machine o encontra-se descrito utilizando o modelo de negócio de Canvas.

3.6 Rede de Valor

Em qualquer negócio existe uma troca de um valor monetário por um qualquer artigo que apresenta valor. Uma das grandes questões que se coloca é como converter recursos intangíveis em ativos com valor. É exatamente isso que (Allee, 2008) transmite no excerto seguinte:

> "A análise da rede oferece uma forma de modelar, analisar, medir e melhorar a forma como um determinado negócio converte recursos tangíveis e intangíveis em diferentes formas de valor negociável e comerciável." (tradução própria)

Uma rede de valor pode ser entendida como um conjunto de interações entre várias pessoas com diferentes papéis e durante essas interações realizam trocas de recursos tangíveis e intangíveis para obter um valor económico ou social (Allee, 2008). Desta forma podemos

entender a rede de valor como um conjunto de indivíduos que interagem e contribuem com diferentes aspetos para a criação de valor.

4 Model Driven Architecture e Model to Model Transformation

4.1 Introdução

As abordagens atuais para o desenvolvimento de *software* que suportam os *developers* na implementação de soluções informáticas empresariais têm demonstrado algumas lacunas no suporte a possíveis alterações de requisitos, alterações tecnológicas, múltiplas plataformas e interoperabilidade entre sistemas (Singh, 2009) e (Tielin, 2010). Model Driven Architecture tenta ajudar a ultrapassar algumas dessas dificuldades através do desenvolvimento com base na definição de modelos, através de um grande nível de abstração e técnicas de automação, transformação e geração de artefactos.

Citação da página web da OMG (OMG, 2015):

> "A arquitetura MDA é a arquitetura empresarial padrão definida pela OMG – Object Management Group[6] [...] que é um consórcio para a definição de especificações para a indústria informática, sem fins monetários e aberto a novos membros. Os membros deste consórcio mantêm a especificação MOF – MetaObject Facility. [...] Empresas de *software* de todo o género desenvolvem ferramentas de modelação que permitem manipular modelos, que cumprem o formato MOF, seja a exportar, importar, armazena, transformar, gerar código entre outro tipo de ações. De salientar que o consórcio OMG não disponibiliza nenhum do software necessário mas disponibiliza as especificações que permitem tornar as aplicações interoperáveis entre si." (tradução própria)

A arquitetura MDA baseia-se na transformação de modelos no formato MOF e cobre todo o processo de desenvolvimento de uma aplicação desde a etapa de definição do modelo inicial como PIM – Plataform Independent Model representando as funcionalidades e comportamentos do negócio. Passando posteriormente por um ou vários PSM – Plataform

[6] Mais informação sobre Object Management Group disponível em: http://www.omg.org/

Specific Model até à geração de código e obtenção de uma aplicação. Desta forma é possível manter o modelo PIM estável e evoluir as tecnologias utilizadas na solução desenvolvida e desta forma maximizar o retorno de investimento da aplicação.

Existem ferramentas para apoiar o desenvolvimento de transformações de modelos e o desenvolvimento de soluções de acordo com MDA, essas ferramentas baseiam-se em várias tecnologias importantes no contexto MDA e MMT. Como tal de seguida apresentam-se as tecnologias relevantes e posteriormente algumas das ferramentas relevantes disponíveis.

4.2 Tecnologias Relevantes

Na área MDA existem diversas tecnologias relevantes para a definição e utilização de modelos. Essas tecnologias permitem serializar, exportar e importar modelos. Existe também a necessidade de pesquisar e navegar pelos diferentes elementos que compõem um modelo. Nesse sentido as tecnologias e padrões como XMI – XML Metadata Interchange, MOF, OCL e QVT servem exatamente esses objetivos. Estas tecnologias cumprem especificações desenvolvidas e mantidas pelo consórcio OMG.

4.2.1 XML Metadata Interchange

A norma ISO/IEC 19509[7] descreve o padrão XMI como sendo o formato de representação de objetos a utilizar para transmitir e armazenar modelos. Define os seguintes aspetos na descrição de objetos utilizando XML:

- A representação em XML de elementos e atributos de um objeto;
- Mecanismos padronizados para relacionar objetos num mesmo ficheiro ou entre diversos ficheiros;
- Identificação de objetos para permitir que estes sejam referenciados noutros objetos utilizando identificadores e UUID - Universally Unique Identifier.

O padrão XMI apresenta soluções para os problemas apresentados através da especificação de regras para a criação de documentos XML e *schemas* que permitem partilhar objetos de forma consistente.

[7] ISO/IEC 19209 – XML Metadata Interchange (XMI) disponível para consulta em: http://www.omg.org/spec/XMI/

4.2.2 MetaObject Facility

Tal como descrito na norma ISO/IEC 19508[8] a especificação MOF é a fundação do Model Driven Architecture definido pela OMG no qual os modelos podem ser exportados de uma aplicação e importados noutra, transmitidos pela rede, armazenados num repositório e posteriormente obtidos desse mesmo repositório em diferentes formatos, transformados e utilizados para gerar código para uma aplicação. As funcionalidades descritas não se encontram restritas a modelos estruturais definidos utilizando UML - Unified Modeling Language, outras definições de modelos também podem ser utilizadas desde que cumpram as especificações MOF (OMG/MOF, 2015).

A existência de *metadata* proprietária e incompatível entre diferentes sistemas é uma das principais limitações no intercâmbio de dados e na integração de diferentes aplicações. *Metadata* são dados acerca de dados, estes dados são normalmente utilizados por diferentes ferramentas, bases de dados, *middleware* entre outros sistemas para descrever a estrutura e o significado dos dados.

As especificações MOF disponibilizam uma *framework* aberta e independente das plataformas para representar e gerir metadata através de uma série de serviços que permitem obter interoperabilidade entre sistemas e desenvolvimento de sistemas a partir de metadata.

4.2.3 Object Constraint Language

Tal como apresentado na norma ISO/IEC 19507[9]:

> "OCL é uma linguagem de especificação, como tal uma expressão OCL garante que não ocorrem efeitos secundários. Como resultado da execução de uma expressão OCL um valor é retornado. A expressão não tem capacidade para alterar o modelo sobre o qual é executada. [...] OCL não se trata de uma linguagem de programação e como tal não é possível desenvolver um programa lógico e com controlo de fluxo de execução. OCL é sim uma linguagem de modelação e as suas expressões não podem ser executada diretamente." (tradução própria de partes da informação disponível na norma ISO)

Quando é necessário especificar restrições e regras adicionais sobre um determinado modelo, normalmente essas regras são descritas através da utilização de linguagens formais. No entanto essas linguagens formais são acessíveis para profissionais com conhecimentos avançados em matemática mas apresentam grandes dificuldades para profissionais com conhecimentos de negócio ou modeladores. A linguagem OCL surge para tentar colmatar esse problema, tratando-

[8] ISO/IEC 19508 – Meta Object Facility (MOF) disponível para consulta em: http://www.omg.org/spec/MOF/

[9] ISO/IEC 19507 – Object Constraint Language (OCL) disponível para consulta em: http://www.omg.org/spec/OCL/

se de um linguagem formal de fácil leitura e escrita. Foi desenvolvida como uma linguagem de modelação de negócio (OMG/OCL, 2014).

4.2.4 Query/View/Transformation

QVT é composto por uma série de linguagens que permitem realizar transformações em modelos e foi definida pelo consórcio OMG. Sendo que as transformações de modelos são uma técnica fulcral utilizada no desenvolvimento MDA, e a linguagem QVT é a linguagem padrão definida para expressar pesquisas, visualizar e transformar modelos MOF (OMG/QVT, 2011). Tal como se pode observar pelo nome, o conjunto de linguagens permite realizar três tipos de operações (Nederpel, 2005):

- ***Transformation*** – permite gerar um novo modelo a partir de um modelo origem;
- ***Query*** – é uma expressão que deve ser executada sobre o modelo de origem e resulta na obtenção de instâncias do modelo de origem. Baseia-se na utilização da linguagem OCL;
- ***View*** – permite obter a representação de um novo modelo completamente derivado de outro modelo, o modelo resultante não pode ser alterado sem implicar a modificação do modelo base.

As três linguagens que coletivamente constituem a linguagem híbrida do QVT são: *Relations, Core* e *Operational Mappings.*

Relations – linguagem declarativa baseada na definição de relações entre elementos de diferentes modelos. Permite a definição de padrões de objetos que podem corresponder às propriedades de uma instância do modelo. Desta forma permite verificar se determinados modelos estão relacionados de alguma forma. Possibilita a realização de transformações unidirecionais ou multidirecionais e atualização incremental de modelos.

Core – linguagem declarativa baseada na definição de relações entre elementos de diferentes modelos, simplificação da linguagem *Relations*. Definição de padrões de objetos mais simples. É mais verbosa que a linguagem *Relations* mas apresenta a mesma expressividade. Trata-se de uma linguagem de transformação simples.

Operational Mappings – linguagem imperativa para transformações, aumenta a linguagem *Relations*. Permite execução de pesquisas sobre o modelo origem, operações sobre objetos, utilização de regras de mapeamento e criação de objetos de destino.

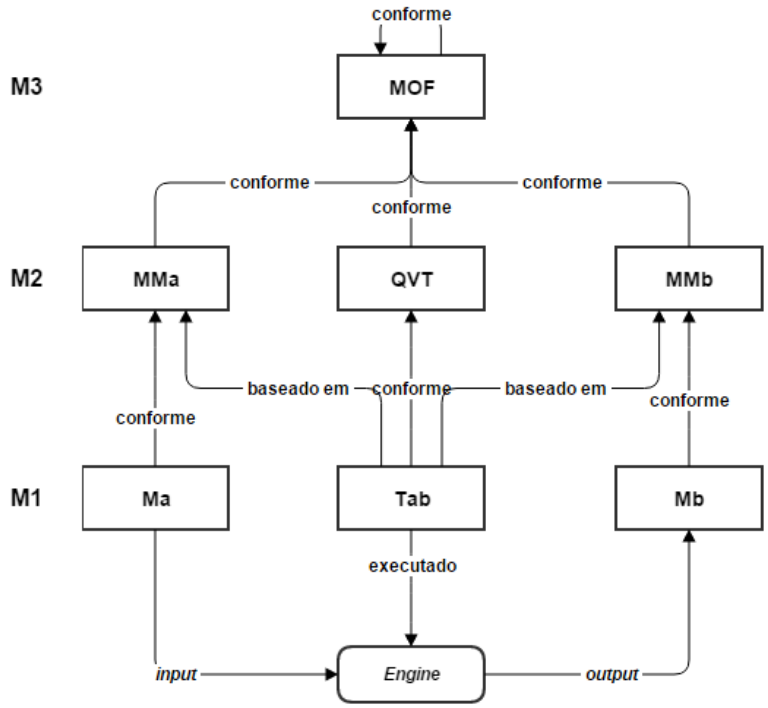


Figura 11 – Contexto de operação da linguagem QVT

Fonte: Figura baseada em conteúdo disponibilizado em (Nederpel, 2005).

Tal como apresentado na Figura 11 a sintaxe abstrata da linguagem é definida de acordo com o metamodelo MOF 2.0. Como se pode observar as transformações são definidas sobre os metamodelos **MMa** e **MMb** e as transformações são executas em instâncias de metamodelos MOF 2.0. A transformação ***Tab*** define as regras necessárias para criar um modelo ***Mb***, concordante com o metamodelo ***MMb***, a partir dos elementos definidos por um modelo ***Ma***, concordante com o metamodelo ***MMa***. Ou seja, o modelo ***Ma*** é o *input* de um Engine que utiliza a transformação ***Tab*** para criar um modelo ***Mb*** como output.

A especificação QVT apresenta os requisitos formulados na proposta QVT RFP – Request For Proposal[10] realizada em 2002 para a qual existiram sete submissões iniciais que acabaram por convergir para uma proposta comum. Alguns dos requisitos formulados no RFP encontram-se na Tabela 1 de forma resumida (Kurtev, 2006).

Fonte da tabela: Tabela baseada em conteúdo disponibilizado no QVT RFP.

Tabela 1 – Requisitos formulados no QVT RFP de 2002

Requisitos Obrigatórios	
Linguagem *Query*	As propostas devem definir uma linguagem para pesquisa de modelos.

[10] QVT RFP disponível em: http://www.omg.org/cgi-bin/doc?ad/2002-4-10

Linguagem Transformation	As propostas devem definir uma linguagem para definição de transformações.
Sintaxe abstrata	A sintaxe abstrata das linguagens QVT devem ser descritas de acordo com o metamodelo MOF 2.0.
Paradigma	A linguagem de definição de transformações deve ser declarativa.
Input e Output	Todos os mecanismos definidos pelas propostas devem operar sobre instâncias de modelos de acordo com o metamodelo MOF 2.0
Requisitos Opcionais	
Directionality	As propostas podem suportar definição de transformações bidirecionais.
Traceability	As propostas podem suportar *traceability* de elementos entre o modelo de origem e destino.
Reusability	As propostas podem suportar mecanismos para permitir a reutilização de definições de transformação.
Atualização do modelo	As propostas podem suportar a execução de transformações que atualizam o modelo existente.

A especificação de requisitos apresentados permite validar se determinada proposta cumpre os requisitos esperados e se se encontra enquadrada nas normas expectáveis.

4.3 Ferramentas Relevantes

Como resposta ao RFP da OMG várias ferramentas foram desenvolvidas que permitem realizar diferentes tipos de transformações entre modelos. São várias as ferramentas relevantes para a solução pretendida. Podemos categorizar as ferramentas com base no seu propósito, desta forma temos:

- Definição de modelos como EMF e Emfatic;
- Geração de código com base no modelo como EMF, Texo e Acceleo;
- Transformação de modelos temos as ferramentas AtlanMod Model Weaver – AMW e Atlas Transformation Language – ATL;
- Modernização de aplicações usando técnicas *model-driven* como a ferramenta Modisco e JaMoPP.

4.3.1 Eclipse Modeling Framework

O projeto EMF é uma *framework* de modelação e geração de código para desenvolver ferramentas e aplicações baseadas em modelos estruturados (Budinsky, 2004). A partir da descrição e especificação de um modelo em XMI as ferramentas de EMF permitem gerar classes

Java que representam esses modelos bem como várias outras classes utilitárias que permitem visualizar e editar esses modelos.

De forma sucinta a *framework* permite, partindo de um determinado modelo, a geração de modelos equivalentes na forma de:

- Código Java anotado;
- Diagrama de classes em UML;
- XML *schema.*

Esta ferramenta possibilita a definição de um metamodelo representativo do modelo de domínio, a grande vantagem da ferramenta encontra-se na facilidade e agilidade de alteração do modelo e a consequente atualização do código gerado a partir do modelo.

A ferramenta EMF permite gerar código representativo dos elementos constituintes de um modelo Ecore previamente definido. Este modelo é definido tendo por base um metamodelo existente, na ferramenta EMF este metamodelo é designado como modelo Ecore. Como apresentado na Figura 12 esse metamodelo é constituído por:

- *EClass* – representa o conceito de uma classe no contexto Ecore, equivalente a uma classe Java;
- *EObject* – representa uma instância, equivalente a uma instância de uma classe em Java;
- *EPackage* – um conjunto de *EClass*, equivalente a um package em Java;
- *EReference* – representa uma associação direta, modela uma relação entre um *EObject* e um ou mais outros *EObject*;
- *EAttribute* – representa um atributo de uma *EClass*;
- *EDataType* – representa um tipo de dados, baseado nos tipos de dados existentes em Java;
- *EOperation* – declaração de um método;
- *EAnnotation* – anotação, equivalente às anotações em Java.

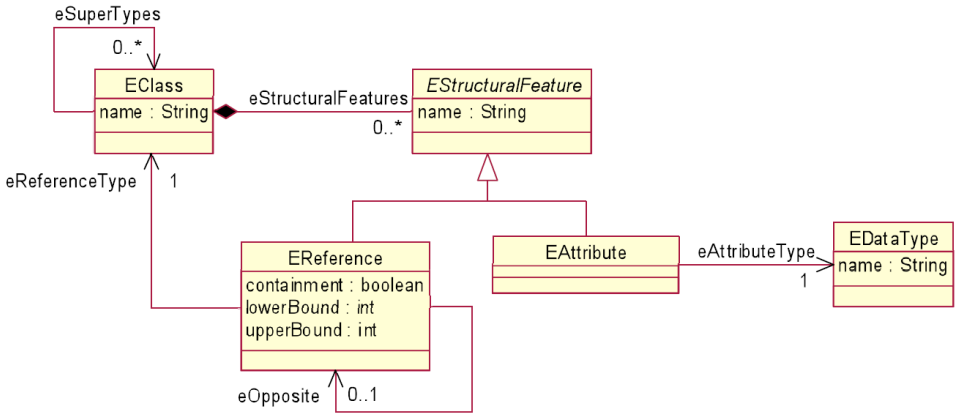


Figura 12 – Metamodelo Ecore

Fonte da figura: (Budinsky, 2004)

O código gerado pela ferramenta EMF disponibiliza capacidades de serialização e persistência de dados, o padrão Observer é utilizado para permitir notificações automáticas de alterações ao modelo. O código gerado esta focado para aplicações Eclipse ou aplicações Java *standalone*.

4.3.2 Emfatic

Emfatic é um editor de texto que suporta navegação, edição e conversão de modelos Ecore através da utilização de uma linguagem semelhante à linguagem Java para definir os elementos constituintes do modelo a definir (Rees, 2012).

Permite converter um modelo Ecore para a representação textual Emfatic e a conversão de uma representação textual Emfatic para um modelo Ecore. Durante essa transformação a ferramenta realiza uma validação de coerência e permite identificar erros na modelação (Daly, 2004). A grande vantagem desta ferramenta é possibilitar a definição de modelos Ecore compatíveis com a ferramenta EMF utilizando um formato textual. Este tipo de formato facilita a definição de modelos grandes comparativamente à utilização de uma ferramenta gráfica como a disponibilizada pela ferramenta EMF.

4.3.3 Texo

Texo é uma ferramenta *open-source* e um componente da EMFT - Eclipse Modeling Framework Technology[11] focado na geração de código para tecnologias *server-side* ao contrário da EMF cujo foco é a geração de código para aplicações ou produtos Eclipse RCP - Rich Client Plataform[12]. Esta diferença no ambiente resulta em diferenças no código gerado por cada uma das ferramentas.

A ferramenta Texo baseia-se fortemente na utilização das técnicas e ferramentas disponibilizadas pela ferramenta EMF. Nomeadamente na definição e suporte do modelo e na capacidade de serialização de XML. No entanto, tendo por base o seu objetivo bem definido de suportar a geração de código para aplicações *server-side* a ferramenta evita ao máximo qualquer tipo de dependência e relação com EMF (Taal, 2015).

Esta ferramenta distingue-se da EMF nos seguintes aspetos:

- No código gerado pela ferramenta Texo não são incluídos os padrões de Adapter e Observer ao contrário do que acontece na ferramenta EMF;
- Texo não utiliza implementações específicas de listas tal como EMF faz;

[11] Eclipse Modeling Framework Technology é um projeto com o objetivo de permitir o desenvolvimento de novas tecnologias que extendam ou complementem a Eclipse Modeling Framework. Mais informação disponível em: http://www.eclipse.org/modeling/emft/

[12] Eclipse RCP trata-se de um conjunto de *plugins* Eclipse que permitem criar aplicações baseadas na estrutura Eclipse mas com propósitos diferentes de um normal IDE. Mais informação disponível em: https://wiki.eclipse.org/Rich_Client_Platform

- A serialização para XML é vista como algo adicional na ferramenta Texo e como tal não faz parte do código gerado;
- As classes geradas pelo Texo não apresenta associações para com o modelo EMF, nem herdam código das classes base EMF.

A ferramenta permite assim a geração de código Java sem dependência para os conceitos EMF o que é desejável em aplicações *server-side* tradicionais.

A geração de código na ferramenta Texo é controlada com base na definição de anotações ao nível do modelo. As anotações permitem controlar os nomes das classes a gerar, os nomes dos métodos, os nomes dos atributos a gerar, condicionar a geração para um determinado elemento do modelo entre outras opções. Durante a geração de código um modelo anotado é automaticamente gerado em memória para cada elemento do modelo original, no entanto é também possível definir este modelo anotado previamente e desta forma definir as regras de geração de código mencionadas. Este modelo anotado manualmente definido não necessita de ser totalmente definido, durante a geração de código a ferramenta complementa o modelo manualmente definido de forma a ter acesso a um modelo anotado completo. Este processo pode ser observado na Figura 13, no qual podemos observar os seguintes passos:

- Leitura do modelo original;
- Definição manual de um modelo anotado completo ou parcial;
- Criação de um modelo completamente anotado, seja através da adição dos componentes em falta no modelo manualmente anotado ou através da criação de raiz de um modelo completamente anotado;
- Utilização de um ou vários Templates para a geração do artefacto utilizando o modelo anotado previamente definido;
- Execução de ações pós geração de código tais como: formatação de código ou adição/organização de artefactos importados.

No final deste processo são criados os artefactos definidos.

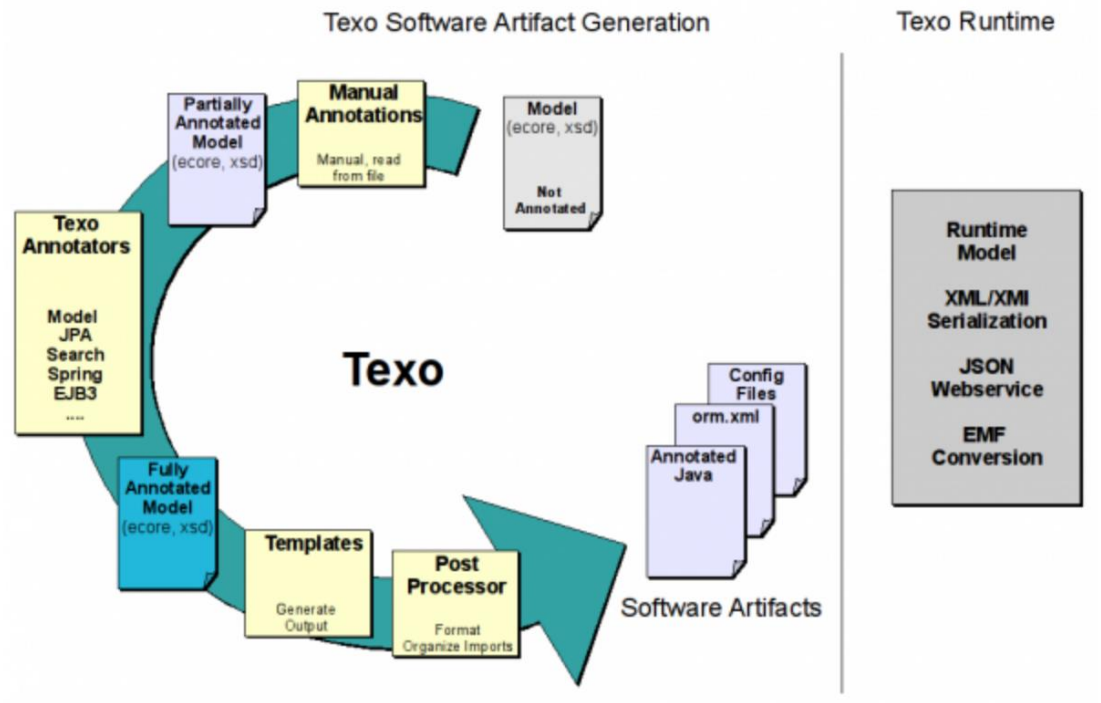


Figura 13 – Processo de geração de código da ferramenta Texo

Fonte da figura: (Taal, 2015)

Algumas das vantagens de utilização da ferramenta Texo são:

- A utilização de modelos anotados para configurar a geração de código permitem configurar o processo de geração com um nível de controlo elevado;
- Apesar de potenciar a geração de artefactos sem dependências diretas para EMF continua a potenciar a utilização de ferramentas disponibilizadas pela ferramenta EMF ao permitir condicionalmente a geração de classes utilitárias para aceder a essas capacidades EMF;
- Permite alterações manuais ao código gerado e a manutenção dessas alterações manuais em futuras gerações de código;
- Permite a customização do código gerado através da customização dos Templates de geração utilizados;
- Permite a formatação de código após a geração.

4.3.4 Acceleo

Acceleo é uma ferramenta de geração de código criada em 2005 e que implementa a especificação *Model-to-text*[13] definida pela OMG (Madiot, 2017). A ferramenta distingue-se de outras ferramentas de geração de código pelas funcionalidades disponibilizadas e pelo seu foco em suportar a geração de código para modelos definidos usando tecnologias EMF. O código

[13] Mais informação disponível em: http://www.omg.org/spec/MOFM2T/1.0/

gerado pode ter como objetivo qualquer linguagem de programação, a ferramenta baseia-se na utilização de *templates* para gerar o código correspondente tendo por base o modelo de entrada.

As funcionalidades a destacar são a sua capacidade de permitir facilmente a geração de código de uma forma incremental, visto permitir manter alterações manuais ao código gerado. Permite definir *overrides* de forma estática ou dinâmica. A afinidade existente com modelos definidos usando EMF é uma das características a destacar. É essa afinidade que facilita a navegação pelos elementos do modelo Ecore permitindo a definição dos *templates* de código a gerar através do acesso as propriedades do modelo como nome de atributos, tipo de dados, cardinalidade, tipo de relação entre elementos.

4.3.5 AMW Modelling Weaving

A ferramenta AMW é um componente do projeto Generative Modeling Technologies – GMT[14] cujo objetivo é produzir ferramentas ilustrativas das capacidades e diversidade de aplicações da utilização de modelos abstratos no desenvolvimento de *software* (Hoisl, 2012). A ferramenta AMW é então um desses componentes e é desenvolvido pelo grupo AtlanMod e faz parte da plataforma ATLAS Model Management Architecture – AMMA[15]. Esta plataforma é composta por outros componentes, entre os quais a ferramenta Modisco que será posteriormente apresentada no capítulo 4.3.7.

Esta ferramenta suporta a criação de diferentes tipos de relações entre elementos de modelos ou metamodelos. Estas relações são armazenadas num modelo de relação (este modelo é normalmente denominado como *weaving model)* definido de acordo com um metamodelo extensível. Este tipo de modelo de relação pode depois ser utilizado em diferentes cenários tais como interoperabilidade de aplicações, definição de transformações, comparação de modelos e junção de modelos.

De forma simplificada a ferramenta AMW permite definir relações entre diferentes elementos de dois ou mais modelos, desta forma é possível definir que um determinado atributo A de um modelo M1 corresponde ao atributo B de um modelo M2 e ainda definir que tipo de relação existe entre esses atributos, por exemplo: *Equality*, *SourceToTarget*, *Concatenation*, *IntToStr*.

As relações entre os modelos são armazenadas num modelo de relação que obedece a um metamodelo. O metamodelo define diferentes tipos de elementos base tal como apresentado na Figura 14, onde podemos identificar os seguintes elementos (Didonet Del Fabro, 2006) (Didonet Del Fabro, 2008): *WElement* – trata-se do elemento base que contém um nome e uma descrição, todos os outros elementos utilizam este elemento; *WModel* – elemento que contém todos os elementos do modelo; *WLink* – representa o tipo de relação, o elemento *WLink* contém uma ou várias referências para o elemento *WLinkEnd*; *WLinkEnd* – relaciona-se com o

[14] Mais informação disponível em: http://wiki.eclipse.org/GMT
[15] Mais informação disponível em: http://wiki.eclipse.org/AMMA

elemento *WElementRef* que contém a referência para o elemento *WModelRef* do modelo a relacionar. Não existe uma relação direta entre *WElementRef* e *WLink* pois assim é possível referenciar o mesmo elemento através de múltiplos elementos *WLinkEnd* e desta forma o mesmo elemento do modelo pode estar envolvido em várias relações.

Este metamodelo base pode ser aumentado para permitir a criação de modelos de relação adaptáveis a diferentes casos de uso. Um dos casos de uso suportados pela ferramenta é a interoperabilidade entre aplicações existentes, neste tipo de cenário existe a necessidade de uma aplicação utilizar o modelo definido numa outra aplicação, surgindo a necessidade de realizar um mapeamento entre os dois modelos e a consequente transformação do modelo de origem num modelo alinhado com o modelo de destino.

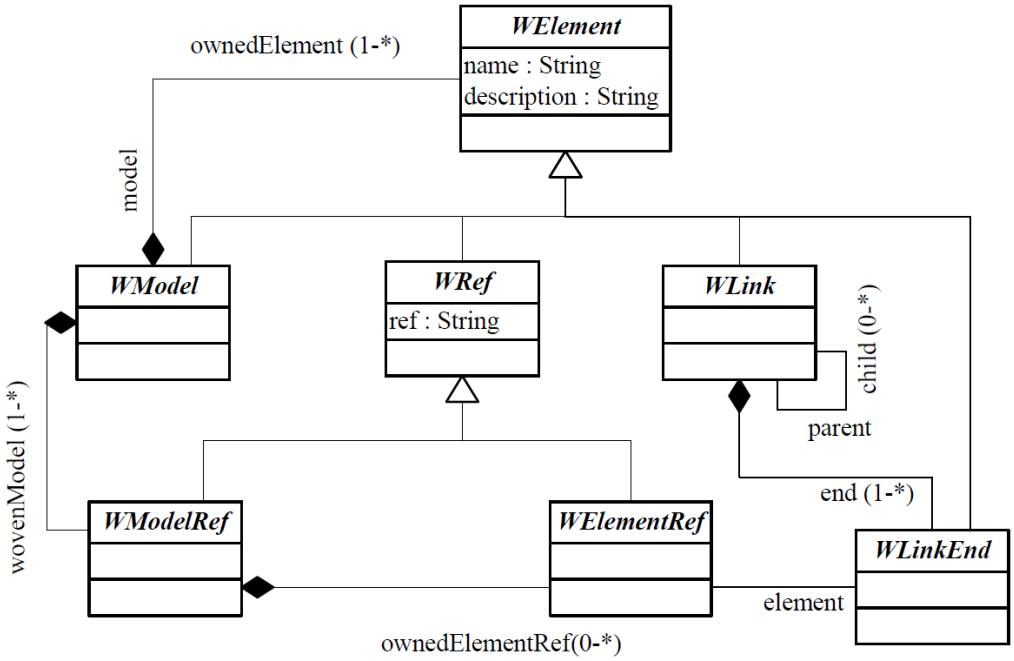


Figura 14 – AMW metamodelo base de relação (*core weaving metamodel)*

Fonte da figura: (Didonet Del Fabro, 2006)

A definição do modelo de relação entre modelos é realizada através de um processo interativo que alterna entre processos semiautomáticos e tarefas manuais de refinamento dos mapeamentos. As tarefas manuais de mapeamento são realizadas através da utilização da interface gráfica disponibilizada pela ferramenta AMW. Tal como apresentado na Figura 15 a interface permite definir diferentes tipos de relações entre os elementos dos modelos.

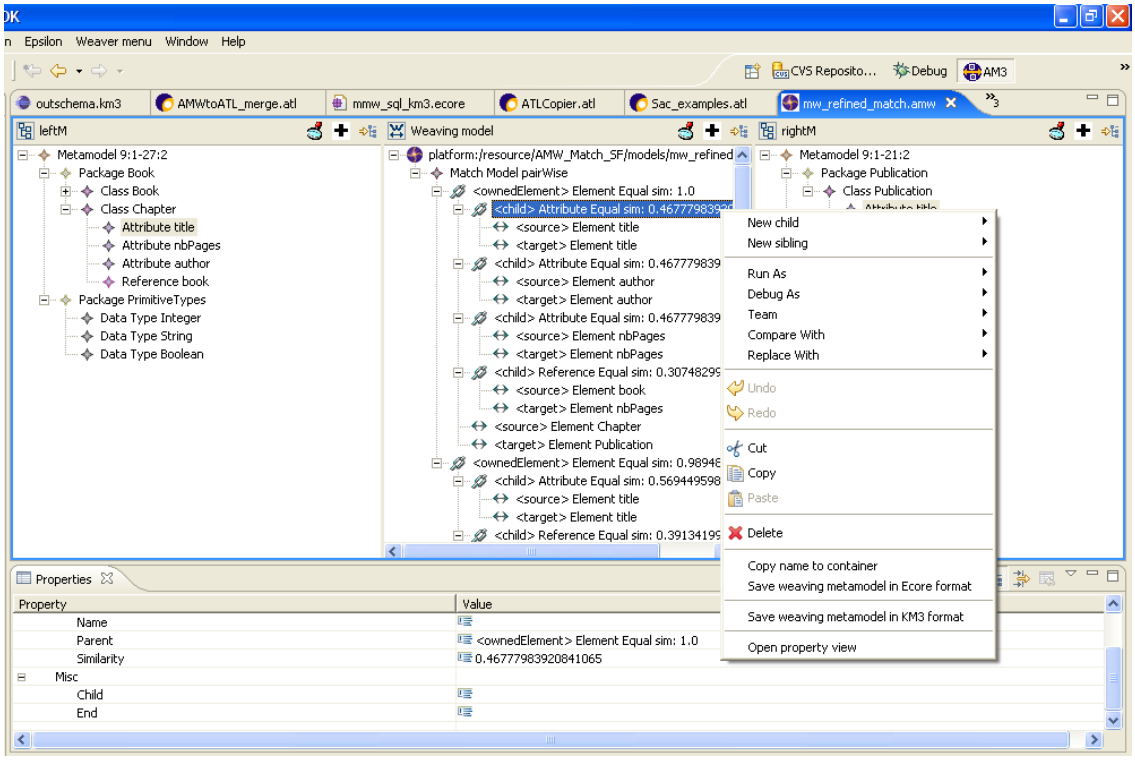


Figura 15 – Interface mapeamento manual de relações entre elementos de modelos na ferramenta AMW

Fonte da figura: (Hoisl, 2012)

Os processos semiautomáticos baseiam-se na utilização de heurísticas de mapeamento existentes ou previamente customizadas. Este processo semiautomático de mapeamento executa um conjunto de heurísticas de correspondência que tentam encontrar conceitos semelhantes nos modelos de entrada. Estes algoritmos podem ser distinguidos em duas categorias (Didonet Del Fabro, 2006) (Didonet Del Fabro, 2010):

- ***Element-to-element*** – um valor de similaridade é calculado para cada elemento dos modelos de entrada. Esses valores são depois utilizados para criar ligações entre esses elementos. Uma ligação com um valor elevado de similaridade significa que existe uma grande probabilidade desses elementos partilharem um significado semântico. O valor de similaridade pode ser calculado utilizando diferentes técnicas, como por exemplo: *String similarity* – os nomes dos elementos do modelo são comparados e um valor de similaridade é atribuído.
- ***Structural*** – um valor de similaridade é calculado para cada elemento do modelo tendo por base propriedades internas de cada modelo, nomeadamente tipo de dados de um elemento, a cardinalidade e a relação entre os elementos.

Após a definição do modelo de relação entre os modelos de entrada é possível realizar operações de transformação. Estas transformações podem ser realizadas através da utilização de transformações ATL e permitem a geração de novos modelos.

4.3.6 Atlas Transformation Language

Citação do manual do utilizador ATL (Wagelaar, 2015):

> "ATL é uma linguagem específica de domínio (*domain-specifc language*) para especificar transformação de modelos, descrita e especificada tanto como metamodelo e como uma sintaxe textual concreta. Trata-se de uma das respostas ao RFP[16] da OMG para MOF e QVT, a proposta ATL foi desenvolvida pelo grupo de pesquisa AtlanMod"[17] (tradução própria).

A linguagem tem por base os formalismos definidos pela linguagem OCL, tentando tirar partido da grande adoção e suporte da OCL pela maioria dos fabricantes de ferramentas para a arquitetura MDA e da familiaridade de utilização existente (Jouault, 2008).

Através da utilização de ATL é possível definir os mecanismos necessários para produzir vários modelos de destino a partir de um ou vários modelos de origem. A linguagem pode ser descrita como um híbrido da programação declarativa[18] e imperativa[19], a forma preferível de descrição das regras de transformação é a declarativa, no entanto também é possível definir transformações, difíceis de descrever declarativamente, de forma imperativa.

Um programa de transformações ATL é composto por várias regras que descrevem a forma de navegação e obtenção dos elementos do modelo de origem que serão utilizados para criar e inicializar os elementos do modelo de destino.

Desenvolvido sobre a plataforma Eclipse[20] o ATL Integrated Development Environment fornece várias ferramentas e utilitários para apoiar o *developer* na definição de regras de transformação, tal como descrito no manual do utilizador (Wagelaar, 2015):

> "[...] fornece uma série de ferramentas utilitárias (*syntax highlighting*, *debugger*) para facilitar a descrição de transformações ATL e também ferramentas adicionais para manusear modelos e metamodelos, como sendo uma notação textual simples para especificar metamodelos e alguns mapeamentos entre sintaxes de linguagem textual e as correspondentes representações de modelo." (tradução própria)

---

[16] Informação adicional do RFP disponível em http://www.omg.org/cgi-bin/doc?ad/2002-4-10

[17] AtlanMod (Inria, Ecole des Mines de Nantes & LINA) mais informação disponível em: http://web.emn.fr/x-info/atlanmod/index.php?title=Main_Page

[18] Paradigma Declarativo: abordagem na qual se descrevem as ações que um programa executa e não como os procedimentos funcionam. Não existe ideia de estado atual do programa.

[19] Paradigma Imperativo: consiste na definição de procedimentos para executar sequências das ações que compõem um programa. Ou seja, sequência de instruções que manipulam valores de variáveis.

[20] Mais informação disponível em: https://projects.eclipse.org/projects/eclipse.platform

De forma geral utilizando ATL um *developer* pode definir três tipos de operações:

- **Transformações entre modelos** – através da especificação de ATL *modules*;
- **Transformações para obtenção de valores primitivos de modelos** – através da especificação de transformações designadas por *queries* e permitem determinar um valor primitivo como uma *string* ou *integer*;
- **Definição de *libraries* ATL independentes** – a linguagem ATL permite desenvolver *libraries* independentes que podem ser importadas e utilizadas em múltiplas operações ATL incluindo outras *libraries*. Isto permite uma melhor organização e reutilização de operações de transformação.

4.3.7 Modisco

Modisco é uma ferramenta *open-source* projetada para permitir a realização de tarefas de obtenção de modelos de aplicações existentes (*model driven reverse engineering*). O seu desenvolvimento iniciou-se em 2006 num esforço conjunto entre a equipa AtlanMod e Mia-Software[21]. Esta ferramenta disponibiliza uma série de componentes para a elaboração de soluções de *reverse-engineering* de modelos a partir do código fonte de uma aplicação (Bruneliere, 2011).

Tal como apresentado na Figura 16 é possível perceber que a ferramenta Modisco possibilita a leitura e obtenção de representações de modelos a partir de vários tipos de artefactos existentes, tais como código fonte, bases de dados, ficheiros de configuração entre outros. A ferramenta baseia-se nas capacidades e ferramentas disponibilizadas pela Eclipse Modeling Framework para construir as suas próprias ferramentas de *reverse engineering*, transformação e geração de artefactos e modelos (Bruneliere, 2014).

[21] Mia-Software (Sodifrance group) mais informação disponível em: http://www.mia-software.com/en/#&panel1-1

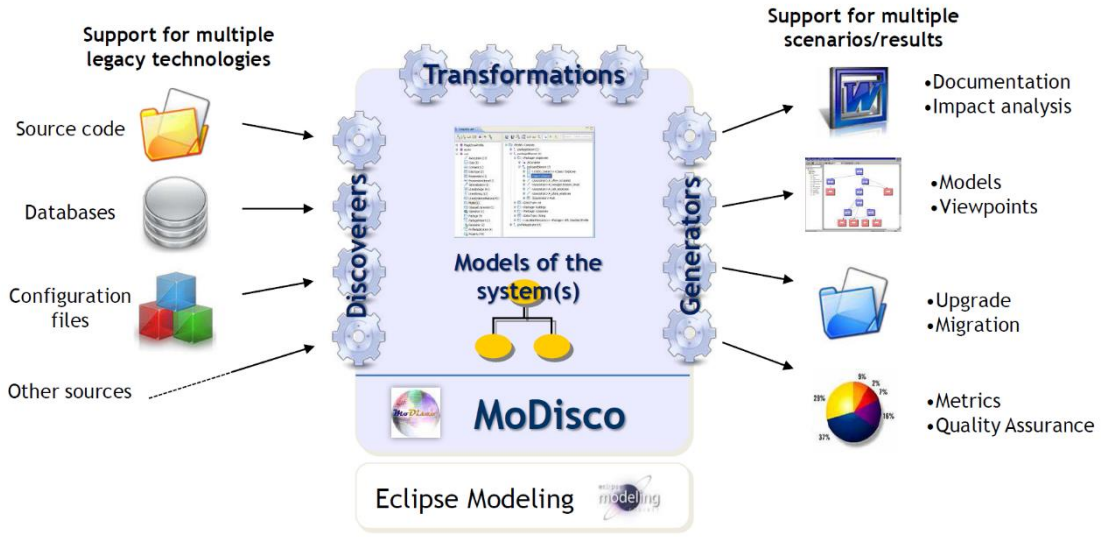


Figura 16 – Visão geral da ferramenta Modisco

Fonte da figura: (Bruneliere, 2014)

A ferramenta é estruturada com base em diferentes *plugins* Eclipse, com responsabilidades bem definidas e que permitem uma fácil extensão e adaptabilidade para novos casos de uso. Como é possível observar na Figura 6 a ferramenta assenta em três camadas, nomeadamente:

- ***Infrastructure*** – trata-se da camada base da ferramenta e contém os componentes genéricos e reutilizáveis, tais como o componente de navegação em modelos Model Browser e o componente de gestão de obtenção de modelos Discovery Manager*;*
- ***Technologies*** – esta camada utiliza os componentes da camada anterior e implementa o suporte para as tecnologias a descobrir através da definição dos metamodelos, descobridores de modelo, geradores e transformações. Esta camada é definida de forma a permitir a sua extensão e especialização para suportar os casos de uso necessários;
- ***Use Cases*** – esta camada representa os casos de uso reais para a ferramenta e permitem a realização das tarefas de obtenção de modelos através da conjugação dos componentes das camadas anteriores.

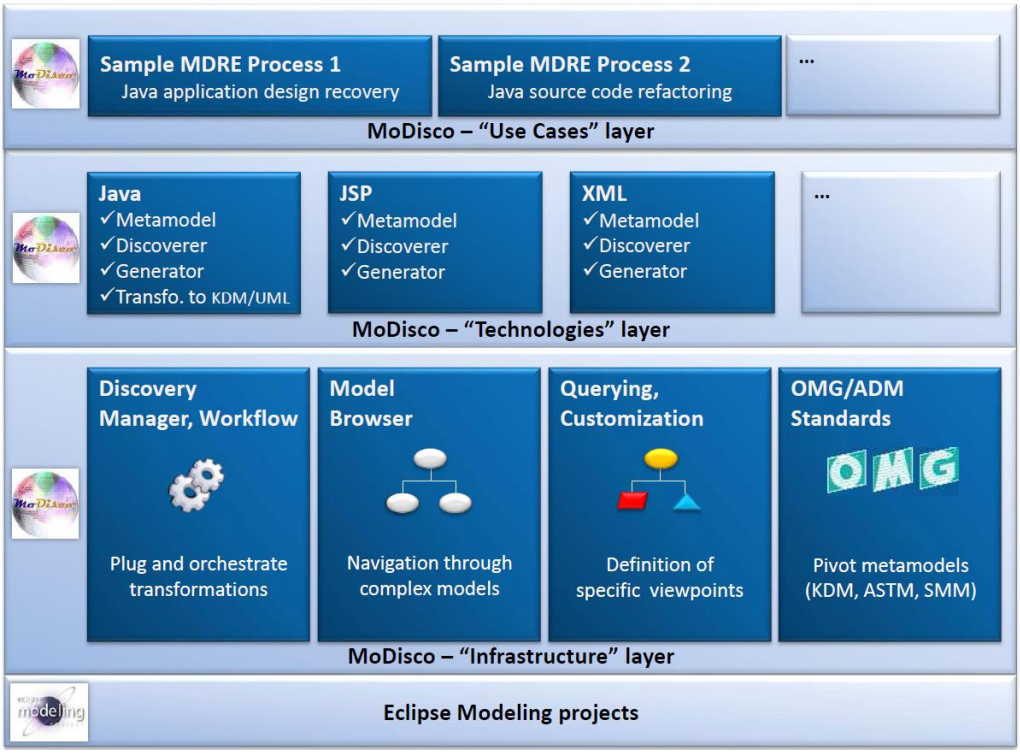


Figura 17 – Camadas e componentes constituintes da ferramenta Modisco

Fonte da figura: (Bruneliere, 2014)

A linguagem de programação Java é uma das linguagens suportadas nativamente por esta ferramenta. Para suportar uma determinada linguagem ou tecnologia é necessário definir o metamodelo correspondente. No caso da linguagem Java o metamodelo implementado na ferramenta Modisco apresenta um elevado nível de cobertura dos conceitos da linguagem Java. Isto permite obter um modelo completo representativo de uma aplicação Java existente. A ferramenta apresenta assim um grande suporte da linguagem Java e permite facilmente a obtenção de modelos de aplicações contruídas usando esta linguagem.

4.3.8 JaMoPP

JaMoPP[22] é um conjunto de *plugins* Eclipse que permitem obter a representação em modelos de acordo com o formato EMF de conteúdo Java (Heidenreich, 2009). O projeto JaMoPP é constituído por:

- Um metamodelo Ecore completo da linguagem Java5;
- Representação completa em formato EMFText[23] da linguagem Java5;
- Uma implementação de análise semântica da linguagem Java5.

[22] Mais informação disponível em: http://www.jamopp.org/index.php/JaMoPP
[23] Mais informação sobre EMFText disponível em: http://www.emftext.org/index.php/EMFText

Com a ferramenta JaMoPP é possível realizar a análise de qualquer aplicação Java e obter uma representação em formato EMF do seu modelo.

5 Mapeamento de Objetos

5.1 Introdução

Em muitas aplicações que interagem com sistemas externos existe a necessidade de realizar um mapeamento e transformação entre os diferentes modelos utilizados pelos diferentes sistemas. No desenvolvimento Orientado a Objetos esses modelos são representados através de diferentes objetos. O mapeamento de objetos consiste na identificação de elementos similares nos objetos e desta forma modelos distintos e desenvolvidos separadamente podem ser utilizados e percebidos por uma aplicação (An, 2007) e (Bonifati, 2010).

O conceito mapeamento de objetos aqui utilizado advém da relação com o conceito de Schema Mapping que se refere à identificação de similaridades semânticas entre diferentes *schemas,* sendo este conceito mais utilizado no contexto de bases de dados (Bonifati, 2010) o conceito mapeamento de objetos é aqui utilizado para referir mapeamentos entre objetos no contexto de desenvolvimento Orientado a Objetos.

5.2 Ferramentas Relevantes

De seguida apresentam-se duas ferramentas relevantes de acordo com o contexto e tendo em conta alguns requisitos como o tipo de licença permitir a alteração e comercialização de *software* que utilize essas ferramentas e aspetos técnicos como não utilização de *reflection*[24] para permitir uma manutenção de desempenho.

5.2.1 MapStruct

Trata-se de uma ferramenta com o objetivo de simplificar a implementação de mapeamentos entre objetos Java dando maior importância à utilização de convenções e não tanto à necessidade de configuração. A sua filosofia passa pela geração de código com base nos

[24] Mais informação disponível em http://docs.oracle.com/javase/6/docs/technotes/guides/reflection/

mapeamentos previamente definidos. O código gerado utiliza simples invocações de métodos e como tal é simples e apresenta um bom desempenho. A ferramenta utiliza definições por omissão mas dá liberdade de customização para comportamentos específicos.

A grande vantagem apresentada pela ferramenta comparativamente à escrita de código de mapeamento manual está na geração automática do código repetitivo de manuseamento e mapeamento dos objetos.

De acordo com (Morling, 2015) em termos comparativos com *frameworks* dinâmicas de mapeamento a ferramenta oferece as seguintes vantagens:

- Execução rápida através do use de simples invocações de métodos sem necessidade de recorrer à utilização de *reflection*;
- Validação de tipos durante fase de compilação – apenas objetos e atributos mapeados são tidos em conta no código gerado, não existe o risco de mapeamentos acidentais;
- Geração de relatório de erros durante o processo de compilação sobre entidades ou atributos que não são possíveis de serem mapeados.

5.2.2 Selma

O nome da ferramenta Selma significa Stupid Simple Statically Linked Mapper. Pode ser descrita como um processador de anotações em código Java e a respetiva geração de código durante compilação para processar o mapeamento entre objetos e como uma biblioteca que fornece os métodos necessários para executar o mapeamento durante a execução da aplicação (Mesle, 2015).

As funcionalidades apresentadas pela ferramenta são:

- Geração de código para execução do mapeamento dos campos dos diferentes objetos;
- Suporte de objetos *nested*;
- Mapeamento entre diferentes tipos de coleções (Map, Collection);
- Mapeamento de Enum com utilização de valores por defeito;
- Suporte de mapeamentos entre tipos customizável;
- Relatório de erros durante compilação.

A ferramenta valoriza a possibilidade de customização e a facilidade de configuração seguindo uma abordagem que tenta simplificar a definição dos mapeamentos necessários através de anotações. O código gerado permite a obtenção de um bom desempenho uma vez que não utiliza *reflection*.

5.2.3 Comparação Ferramentas

Para determinar qual das ferramentas MapStruct ou Selma deve ser escolhida foram tidos em conta vários critérios como: desempenho da ferramenta na execução das transformações; documentação existente e a sua utilidade e facilidade de leitura; complexidade de utilização da ferramenta; adequação das funcionalidades apresentadas e possibilidade de customização da ferramenta.

Estes critérios foram utilizados como fatores de decisão e um peso ou importância foi definido para cada um de acordo com os requisitos específicos da ferramenta a desenvolver. Tomando em conta que o a degradação de desempenho é inaceitável e que a ferramenta necessita de apresentar as funcionalidades necessárias para o desenvolvimento da ferramenta pretendida.

De forma a perceber o desempenho de cada uma das ferramentas foi utilizado um pequeno *benchmark* existente para comparação de ferramentas de mapeamento (Rey, 2015) e foram feitas pequenas alterações nesse *benchmark* para alargar o número de execuções e obter um resultado com maior confiança. Os resultados obtidos na execução do *benchmark* foram tidos em conta na comparação das ferramentas.

A comparação das principais funcionalidades entre Selma e MapStruct foi realizada com base na documentação de cada uma das ferramentas.

Como se pode observar na Tabela 2 para cada um dos fatores existe um valor diferente para o seu peso de acordo com a importância definida. O valor toral apresentado permite concluir que a ferramenta MapStruct para o problema em causa deve ser a ferramenta a utilizar.

Tabela 2 – Comparação entre ferramentas de mapeamento com base na definição de critérios de comparação com diferentes pesos

		Desempenho	Documentação	Complexidade Utilização	Adequação Funcionalidades	Customização Disponível	Total
	Peso	10	6	8	10	6	
MapStruct	Valor	10	8	7	9	7	
	Total	100	48	56	90	42	336
Selma	Valor	8	5	8	9	5	
	Total	80	30	64	90	30	294

6 Soluções e Abordagens Existentes

6.1 Introdução

Para melhorar o processo utilizado na implementação dos diferentes *adapters* responsáveis pela integração de diferentes PDS é necessário melhorar a metodologia utilizada no mapeamento entre os diferentes modelos utilizados por cada um dos diferentes PDS. Para tentar atingir esta melhoria é possível identificar duas abordagens de acordo com as técnicas e tecnologias preconizadas. Essas abordagens podem ser classificadas como:

- Utilização de Model to Model Transformation – transformações entre diferentes modelos, pressupõe o conhecimento da informação representativa dos modelos em causa, ou seja, a existência de uma representação de cada um dos modelos num formato compatível com Model Driven Architecture;
- Utilização de mapeamento de objetos – mapeamento entre atributos de diferentes objetos através da utilização da ferramenta MapStruct que permite a definição do mapeamento entre objetos através de anotações e da geração de código durante compilação para execução dos mapeamentos necessários.

Uma vez que o problema em questão pretende responder à integração de diferentes tipos de sistemas, nomeadamente a Product Machine que permite o acesso à definição do modelo utilizado num formato compatível com MMT e o Symass que utiliza um modelo num formato não compatível com MMT.

Com base nessas diferenças a solução pode inclusive implicar a utilização das duas abordagens numa tentativa integrada de resolver diferentes problemas, uma vez que a aplicação de cada uma das abordagens individualmente pode revelar-se insuficiente. Existe no entanto um aspeto em comum entre as duas abordagens, a necessidade de implementação de técnicas de leitura e carregamento do modelo externo a integrar e ainda a necessidade de uniformizar o modelo canónico interno (BOM interno).

6.2 Leitura Modelo Externo

Numa fase inicial do processo de mapeamento, antes da realização de qualquer outra operação, é necessário ler o modelo externo que se pretende mapear.

Como referido anteriormente já foi realizada a integração com diferentes modelos disponibilizados pela Product Machine. Para a realização dessa integração as classes que representam o modelo externo encontram-se num ficheiro *jar* disponibilizado pela PM e o processo de mapeamento é realizado manualmente através da recolha de informação dessas classes pelo *developer*.

Além dessa operação em paralelo também é necessária a geração de código utilitário utilizado durante o mapeamento. Esse código é gerado através da leitura dinâmica do conteúdo dessas classes. Essa leitura é realizada através de técnicas de *reflection* que permitem, através das classes do BOM da PM, a obtenção de informação sobre os vários atributos existentes nessas classes.

A questão da leitura do modelo externo a partir de diferentes formatos assume maior importância uma vez que dependendo do PDS a integrar podem ser necessárias técnicas diferentes.

O objetivo final da leitura do modelo é a obtenção de um formato compatível com as técnicas de MMT. Para atingir esse objetivo a abordagem pensada passa pela criação de um componente capaz de ler diferentes tipos de formatos, sejam ficheiros *jar* ou representações XML. De seguida, através de técnicas de *reverse engineering* obter a representação do modelo no formato pretendido. Esta é uma das hipóteses que se pretende validar e caso se comprove que não é possível obter a representação pretendida sem perda de informação pode ser então necessária a utilização das duas abordagens preconizadas.

6.3 Uniformização Modelo Interno

Tal como a necessidade de leitura do modelo externo também a uniformização do modelo interno é um requisito transversal às duas abordagens.

Para atingir esta uniformização a abordagem preconizada passa pela utilização das ferramentas EMF e Emfatic para realizar a definição do modelo interno da aplicação no formato Ecore. A ferramenta Texo será utilizada para possibilitar a geração do código Java de acordo com o modelo definido.

Sem este passo a realização da validação da possibilidade de utilização de MMT como abordagem possível não é exequível. Para além disso esta alteração possibilita a obtenção de uma melhoria no processo de alteração do modelo e geração do código correspondente.

6.4 Transformação entre Modelos

A abordagem de utilização das técnicas de MMT assenta na possibilidade da obtenção da representação do modelo externo num formato compatível. Tal como exposto anteriormente alguns PDS permitem a obtenção do formato necessário, outros não e nesses casos existe a necessidade de validar a possibilidade de realização de *reverse engineering* para obter o modelo num formato compatível e sem perda de informação.

Assumindo esses pressupostos, na abordagem baseada na utilização de técnicas MMT o mapeamento entre o modelo interno e externo passaria pela utilização das ferramentas de transformação AMW e ATL.

Nesta abordagem diferentes transformações idênticas e repetitivas seriam desenvolvidas utilizando o conceito de biblioteca de transformações disponível na ferramenta ATL. Esse conjunto de transformações permitiria a sua reutilização e obtenção de um ganho de produtividade.

6.5 Mapeamento de Objetos

A abordagem baseada na utilização da ferramenta MapStruct para a realização do mapeamento entre objetos passa pela utilização da representação obtida pelo componente de leitura de modelos no formato de objetos Java.

As classes do modelo externo obtidas pelo componente de leitura são utilizadas durante a fase de definição do mapeamento com as classes do modelo interno previamente geradas.

A definição das relações entre os atributos e as transformações necessárias são conseguidas através da utilização da ferramenta MapStruct. Depois de configuradas essas regras e durante o processo de compilação do código o MapStruct gera as classes responsáveis pela execução do mapeamento propriamente dito.

Uma vez que a ferramenta permite um grande nível de customização, nesta abordagem a obtenção de um maior nível de automatização passa pela criação de tipos de transformação adicionais aos existentes no MapStruct para suportar casos mais complexos identificados e permitir a sua reutilização.

6.6 Conclusões

Como apresentado a escolha de uma das abordagens encontra-se condicionada pela validação da hipótese de leitura e obtenção do modelo num formato utilizável na abordagem MMT. Desta forma numa fase inicial é necessário desenvolver o componente de leitura e validação da utilização de técnicas de *reverse engineering* disponíveis, de acordo com os resultados obtidos é possível determinar três possíveis abordagens a seguir:

- Utilização exclusiva de mapeamentos MMT utilizando as ferramentas AMW e ATL;
- Utilização exclusiva de mapeamentos utilizando a ferramenta MapStruct;
- Utilização das duas ferramentas de mapeamento numa solução integrada para responder a diferentes tipos de mapeamentos.

7 *Design* da Solução

7.1 Design

Como apresentado na Figura 18 a implementação da prova de conceito da ferramenta de mapeamento pretende auxiliar a integração do motor de definição de produtos Symass e substituir o método de mapeamento atualmente utilizado na integração com a Product Machine.

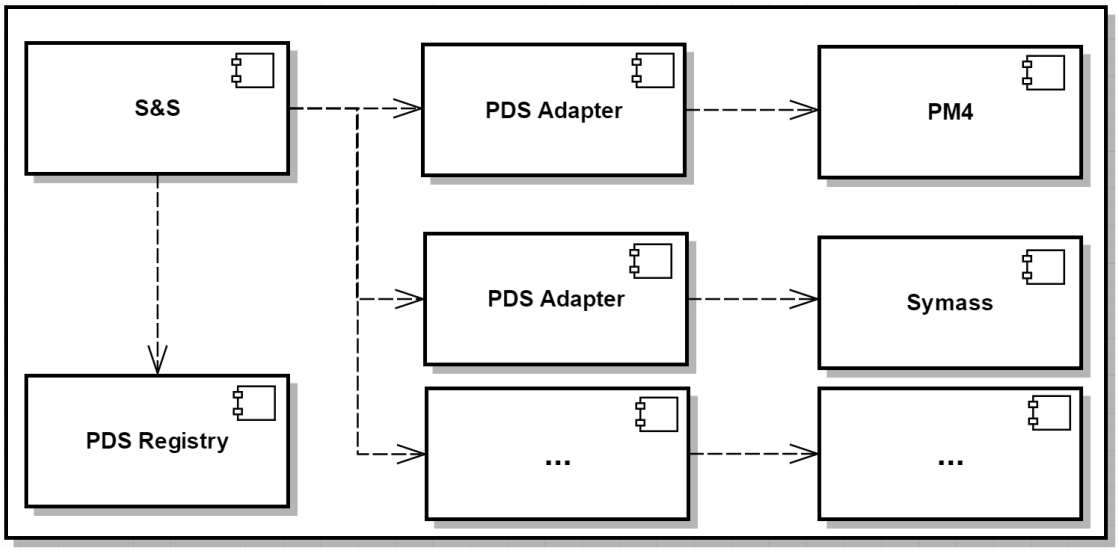


Figura 18 – Integração PDS com PM e Symass

Fonte da figura: Documentação interna e proprietária da msg life Iberia – Sales & Service Tech Overview 2016

Para o funcionamento da ferramenta de mapeamento é possível identificar claramente a necessidade de obtenção da representação do modelo interno e externo. A partir dessas representações um *developer* deve realizar a concretização da definição das regras de mapeamento existentes entre cada modelo procedendo à introdução dessas regras na ferramenta de mapeamento através da utilização de diferentes tipos de transformações.

Idealmente numa fase de maturidade da ferramenta o processo de introdução dessas regras seria feito sobre à representação gráfica dos modelos.

Na Figura 19 podemos observar as interações entre o *developer* e a ferramenta de mapeamento nomeadamente: identificação da fonte do modelo externo; configuração da técnica de leitura a utilizar e definição dos mapeamentos e regras de transformação a utilizar.

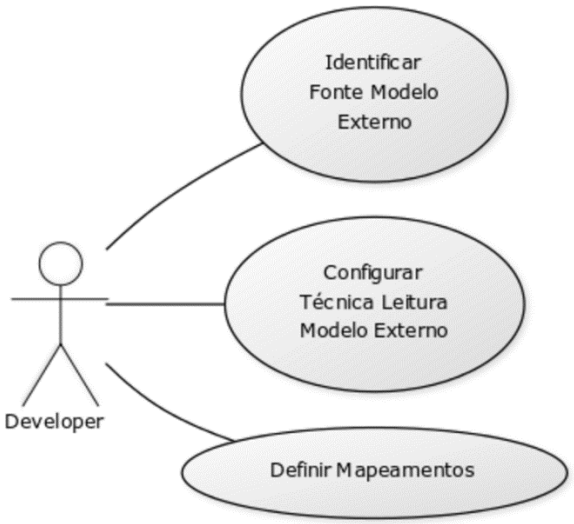


Figura 19 – Interações entre o developer e a ferramenta de mapeamento

Após a realização destas operações o *developer* necessita apenas de programar manualmente a chamada ao método de execução da transformação propriamente dita.

7.2 Arquitetura

A Figura 20 apresenta os diferentes componentes que compõem a ferramenta de mapeamento de modelos:

- Componente de leitura - inclui a leitura do modelo externo de diferentes PDS e a possível transformação por *reverse engineering* para um formato utilizável por MMT;
- Componente de apresentação dos modelos para realização do mapeamento – não é expectável que este componente venha a ser desenvolvido na implementação da prova de conceito;
- Componente de execução do mapeamento – este componente será utilizado pelos diferentes *adapter* criados para a integração de diferentes PDS.

Na Figura 20 podemos visualizar uma visão global da arquitetura do Sales & Service com a inclusão dos componentes identificados. Como se pode observar o componente de leitura dos modelos obtém os dados da Product Machine que se encontra incluída no Sales & Service e

obtém os dados através dos serviços do Symass. O componente de configuração de mapeamentos utiliza a informação recolhida no componente de leitura para permitir a configuração dos mapeamentos necessários. Após realizada essa operação o código necessário para executar o mapeamento dentro do PDS *adapter* é gerado para cada um dos diferentes *adapters.*

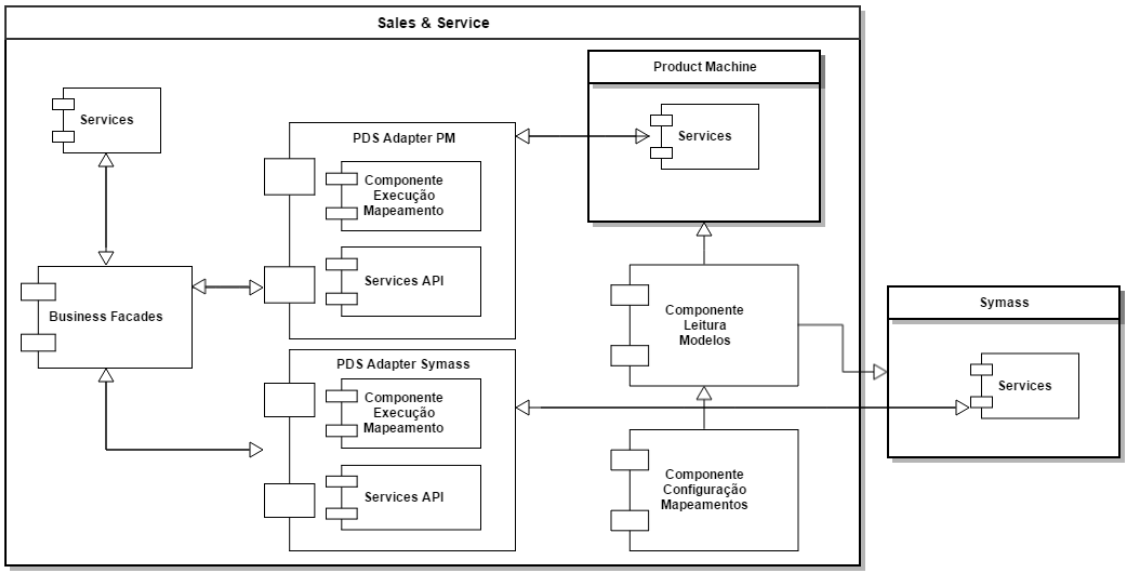


Figura 20 – Visão global arquitetura Sales & Service com a inclusão dos componentes da ferramenta de mapeamento

7.3 Detalhe

A implementação concreta da ferramenta de mapeamento pode variar de acordo com a abordagem escolhida como apresentado no capítulo 6, nesse sentido o componente de execução do mapeamento pode utilizar diferentes técnicas e tecnologias de acordo com os resultados obtidos no componente de leitura.

7.3.1 Componente Leitura

O componente de leitura necessita de ser desenvolvido de forma a permitir a leitura de modelos de diferentes fontes, como tal a sua implementação pode utilizar padrões de *design* como Adapter e Strategy (TutorialsPoint, 2016) para permitir a implementação de diferentes métodos de leitura do modelo.

O objetivo deste componente é permitir obter o modelo representativo da aplicação com a qual é necessário interagir. Como resultado final deve ser possível obter um modelo Ecore para ser utilizado como modelo de entrada para o componente de definição de mapeamento.

Foram identificadas três possíveis alternativas de *design* para o componente de leitura:

1. *Reverse engineering* usando técnicas de *reflection* para obter a informação necessária de modelos em formato não Ecore. Nesta alternativa seria necessário utilizar diferentes mecanismos de mapeamento. Para modelos de entrada Ecore poderíamos realizar os mapeamentos utilizando ferramentas MMT e para modelos de entrada não Ecore poderíamos utilizar as ferramentas de mapeamento de objetos;
2. *Reverse engineering* usando técnicas de reflection para obter informação de modelos em formato não Ecore, de seguida com base nessa informação um modelo Ecore correspondente seria criado. Nesta alternativa seria possível utilizar as mesmas técnicas de mapeamento MMT tanto para modelos de entrada Ecore e outros formatos de modelo. A transformação da informação obtida durante o processo de reflection seria realizada com código desenvolvido à medida para esta tarefa;
3. *Reverse engineering* usando ferramentas de modernização de aplicações já existentes. Nesta alternativa seria possível utilizar as mesmas técnicas de mapeamento MMT. O modelo Ecore seria obtido das aplicações existentes utilizando as capacidades disponibilizadas por ferramentas existentes como Modisco ou JaMoPP.

A alternativa 1 apresenta um grande inconveniente pois seria necessário implementar diferentes mecanismos de definição de mapeamentos para suportar os diferentes formatos de modelo de entrada. Nesse sentido esta alternativa apresenta-se como a menos desejada neste contexto. Já as alternativas 2 e 3 permitiriam a utilização de um mecanismo comum de definição de mapeamentos, visto suportarem a obtenção do modelo de entrada em formato Ecore. Realizando um exercício de comparação entre a alternativa 2 e 3 podemos identificar uma mais-valia da alternativa 3 no que se refere a reutilização de ferramentas existentes e na redução do esforço necessário para obter os mesmos resultados que a alternativa 2 permite obter. A necessidade de desenvolver um mecanismo de captura do modelo Ecore totalmente de raiz e específico para este cenário apresenta-se como uma grande desvantagem da alternativa 2 pois teríamos um custo de desenvolvimento e manutenção elevado.

Como conclusão, das três alternativas apresentadas aquela que se apresenta como sendo a mais vantajosa e a mais alinhada com o objetivo identificado para este componente é a alternativa 3.

Tendo definido a alternativa de *design* mais adequada surge agora a necessidade de detalhar a arquitetura preconizada para este componente.

A utilização da ferramenta Modisco apresenta-se como uma das abordagens alinhadas com os objetivos identificados para este componente. Tal como apresentado no capítulo 4.3.7 esta ferramenta permite a criação de um processo de *reverse engineering* e a obtenção do modelo representativo de uma aplicação. Visto apresentar suporte nativo para a linguagem Java corresponde as necessidades deste componente. A ferramenta apresenta um grande nível de customização e extensão e é com base nessas características que se apresenta a arquitetura específica deste componente na Figura 21. Podemos observar que para atingir os objetivos deste componente será necessária a criação de dois novos *plugins* Eclipse que modificam a funcionalidade base disponibilizada pelos *plugins* base existentes na ferramenta Modisco.

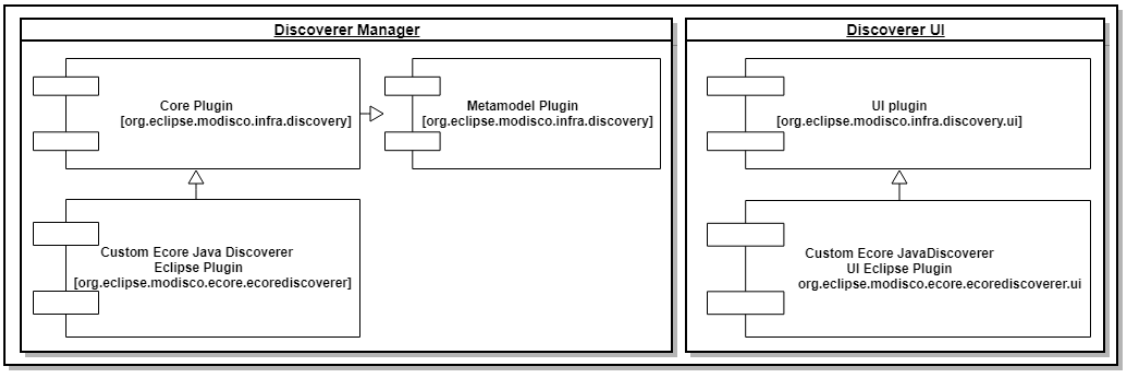


Figura 21 – Diagrama de componentes do componente de leitura

O *plugin* Core Plugin define a API que um *discoverer* deve implementar e disponibiliza um registo de todos os *discoverers* existentes. O *plugin* UI Plugin disponibiliza um menu de contexto customizável e que permite apresentar os *discoverers* aplicáveis para um determinado recurso selecionado.

Os dois novos *plugins* Eclipse a criar:

- Custom Ecore Java Discoverer – *org.eclipse.modisco.ecore.ecorediscoverer*;
- Custom Ecore Java Discoverer UI – org.eclipse.modisco.ecore.ecorediscovere.ui.

Têm por objetivo adicionar um novo Ecore *discoverer* e os elementos UI correspondentes para permitir alinhar o processo de *reverse engineering* com o cenário de utilização pretendido. Estes dois *plugins* irão tirar partido dos mecanismos de extensão disponibilizados na ferramenta Modisco.

7.3.2 Componente Definição Mapeamento

O componente de definição de mapeamento deve permitir que o *developer* defina as regras de transformação entre os diferentes modelos. A implementação deste componente encontra-se restringida pela definição da abordagem concreta a utilizar.

O *developer* irá interagir com o componente de definição de mapeamento na execução de seis tipos de ações, na Figura 22 podemos observar essas interações: executar transformações mapeamento automáticas; adicionar mapeamentos manualmente; alterar mapeamentos manualmente; remover mapeamentos manualmente e gerar modelo mapeamento Ecore.

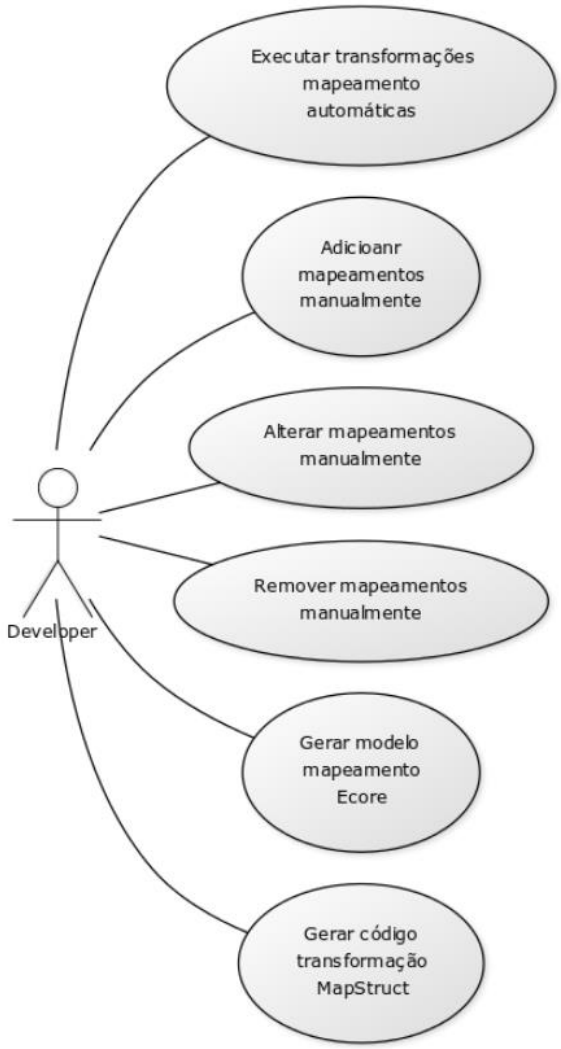


Figura 22 - Interações entre o developer e o componente de mapeamento

Após definir a utilização das ferramentas AMW e ATL em conjunto com a ferramenta MapStruct numa solução integrada para responder a diferentes tipos de mapeamentos como sendo a abordagem a seguir surge a necessidade de definir a arquitetura global do componente de definição de mapeamento.

O componente de definição de mapeamento deve suportar a criação de uma representação dos mapeamentos e das relações entre os modelos. Para tal deverá utilizar todas as potencialidades da ferramenta AMW para permitir obter o modelo de relação. A ideia de *design* deste componente assenta na possibilidade de criação de um modelo de transformação a partir do modelo de relação AMW, ou seja, com base nas relações definidas entre os elementos de cada modelo a ideia seria transformar esse modelo num modelo que represente o tipo de transformação a realizar para cada elemento. Com base nesta premissa apresenta-se de seguida a arquitetura específica deste componente utilizando o diagrama de componentes apresentado na Figura 23.

De forma global podemos identificar os quatro componentes mais relevantes neste diagrama:

- *Plugins* AMW – conjunto de *plugins* AMW que permitem realizar os mapeamentos e a obtenção do modelo de relação AMW;
- *MapStructModel* [25] – metamodelo de transformação para identificar o tipo de transformações a realizar entre os modelos;
- *AMWtoEcoreMapStruct* – transformação ATL que deve permitir obter um modelo MapStructModel através do modelo de relação AMW previamente definido;
- *MapStructModel2Text e MapStructModel2TextUI* – *plugins* Acceleo para gerar o código de transformação MapStruct com base no modelo MapStructModel Ecore previamente obtido.

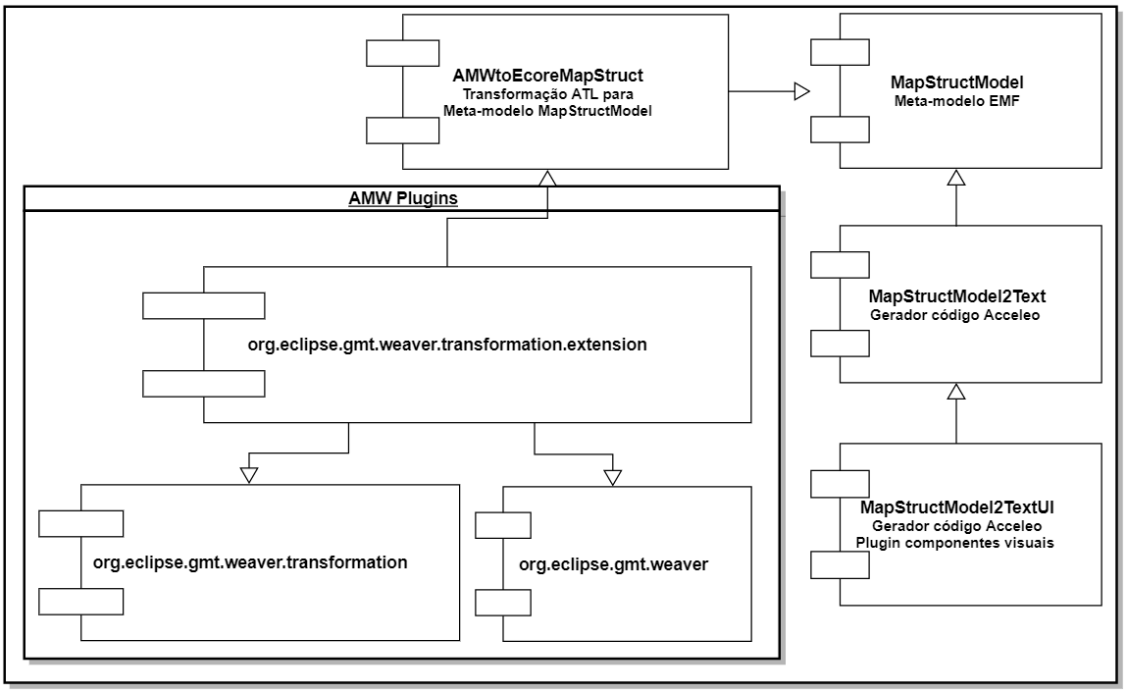


Figura 23 – Diagrama de componentes do componente de definição de mapeamento

De acordo com esta arquitetura a ferramenta AMW e as suas funcionalidades em termos visuais devem ser aproveitadas de forma a suportar as interações necessárias para obter o modelo de mapeamento correspondente.

Um dos aspetos importantes neste *design* está relacionado com o possível metamodelo de transformação a definir. Na Figura 24 apresenta-se um diagrama de classes com o metamodelo preconizado. Neste metamodelo estão definidas apenas três entidades:

[25] O nome *MapStructModel* foi inicialmente definido tendo por base o conhecimento existente na altura de *design*, no entanto este modelo de transformação é genérico e não tem qualquer dependência com a ferramenta MapStruct.

- ***Mapping*** – representa o mapeamento concreto entre o atributo na entidade origem e o atributo na entidade destino. O atributo *expression* tem como objetivo permitir definir tipos de transformação mais complexos;
- ***Mapper*** – representa um mapeamento entre duas entidades, o atributo *mapperName* serve para permitir definir o nome do mapeamento. Esta entidade também deve ter informação sobre a entidade de origem e a entidade de destino. A relação referencedMappers permite relacionar diferentes instâncias *Mapper* a ideia desta relação advém do facto da ferramenta MapStruct necessitar de conhecer os tipos de transformações que uma entidade *parent* pode vir a necessitar de utilizar;
- ***MapperRepository*** – para permitir armazenar várias instâncias da entidade *Mapper* e desta forma possibilitar a definição de mapeamentos para várias entidades.

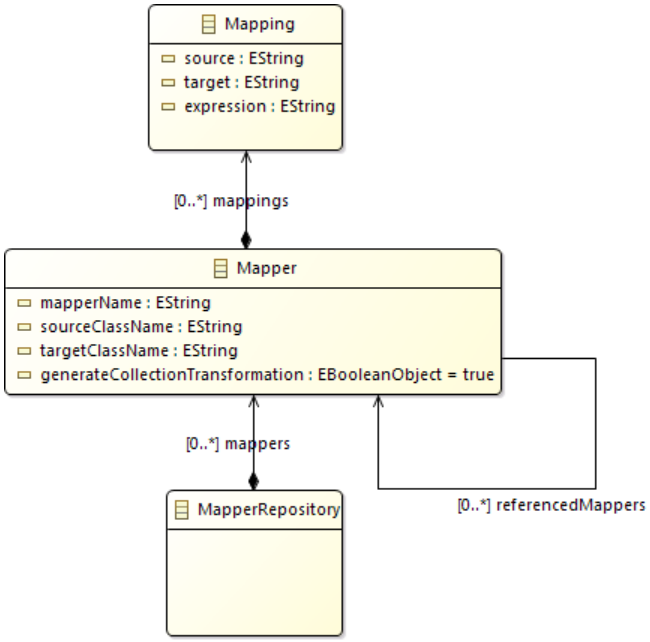


Figura 24 – Metamodelo de transformações para componente mapeamento

Sendo assim a transformação ATL *AMWtoEcoreMapStruct* deverá ser capaz de obter um modelo alinhado com o metamodelo descrito e desta forma permitir que o *plugin* MapStructModel2Text proceda à geração do código de transformação MapStruct para permitir executar a transformação dos elementos entre os modelos mapeados.

7.3.3 Componente Execução Mapeamento

O componente de execução do mapeamento trata-se de um ponto de comunicação entre um PDS *adapter* e o código de transformação gerado de acordo com os mapeamentos definidos. Como se apresenta na Figura 25 este componente deve obedecer a um contrato bem definido.

```
/**
 * Transforms the received SSSBOAdaptable to the appropriate one for Product
 * Definition System.
 *
 * @param originAdaptable
 *            The BOAdaptable to transform.
 * @return The transformed BOAdaptable.
 *
 * @throws PDSTransformationException
 */
public V transformAdaptable(T originAdaptable)
        throws PDSTransformationException;
```

Figura 25 – Excerto de código do possível contrato para o método de execução do mapeamento

A ideia deste componente é permitir a utilização do código de transformação gerado e possibilitar a sua utilização durante a comunicação com o PDS.

8 Construção da Solução

A implementação do protótipo da solução preconizada foi realizado tendo por base um processo iterativo e utilizando metodologias de desenvolvimento ágeis. Este tipo de processo de desenvolvimento permite a validação do progresso e a identificação de possíveis desalinhamentos na solução o mais cedo possível. A implementação dos protótipos de cada um dos componentes individuais tem por objetivo identificar a viabilidade da utilização das técnicas, tecnologias e ferramentas previamente identificadas como sendo as corretas para permitir obter a solução pretendida.

8.1 Implementação Uniformização Modelo Interno

O modelo do S&S é representado por uma série de classes Java e utiliza técnicas de geração de código para permitir a geração de código utilitário de suporte ao modelo. Atualmente nessa geração é utilizada a ferramenta de geração de código Velocity[26].

Tal como discutido previamente existe a necessidade de uniformizar o modelo existente num modelo Ecore que torne possível a utilização transparente do modelo no componente de definição de mapeamentos. Para realizar esta tarefa foram utilizadas as ferramentas Emfatic e EMF para definir o modelo do S&S em formato Ecore.

8.1.1 Modelo e processo geração de código existente

A Figura 26 apresenta um diagrama de classes simplificado de algumas classes do modelo existente do S&S. O modelo existente é composto por cinquenta e duas classes Java com relações entre si. O propósito de apresentar este diagrama e referenciar esse valor serve apenas para demonstrar a necessidade de melhorar o processo de alteração do modelo e subsequente atualização do código gerado atualmente em prática na organização.

[26] Velocity é uma ferramenta de geração de código baseada na utilização de *templates* com acesso a informação providenciada por objetos Java. Mais informação disponível em: http://velocity.apache.org/

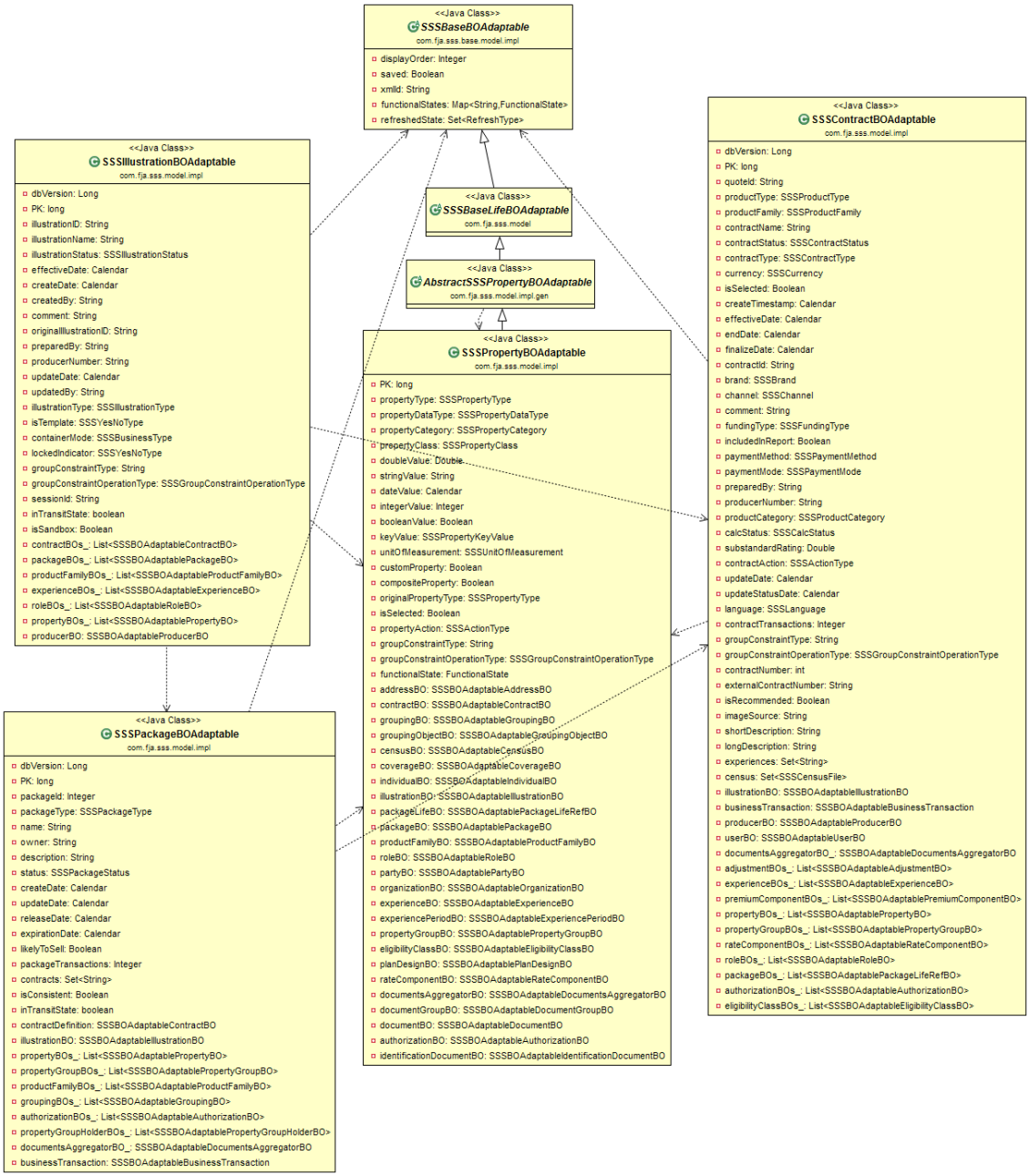


Figura 26 – Diagrama de classes simplificado e incompleto do modelo do S&S

A Figura 27 apresenta os passos realizados durante o processo de geração de código existente. Podemos observar que existem três passos principais. É executado um processo de análise utilizando técnicas de *reflection* para obter a informação necessária do modelo definido por classes Java. Informação sobre os atributos e métodos de cada classe, informação sobre as relações existentes entre as classes e anotações de atributos e métodos. Após obter essa informação é preparado o contexto a utilizar pelos *templates* Velocity, por fim a execução dos *templates* Velocity é realizada e o código é gerado.

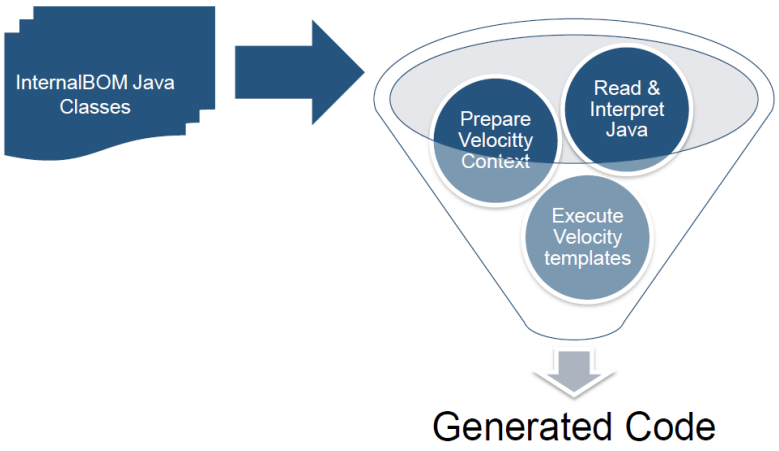


Figura 27 – Processo de geração de código com *reflection* e Velocity

Fonte da figura: Documentação interna e proprietária da msg life Iberia – Sales & Service Tech Overview 2016

O código responsável por obter a informação a partir das classes Java utilizando *reflection* é demasiadamente complexo e devido à sua evolução orgânica com base nos requisitos crescentes apresenta um elevado custo de manutenção. Tal como é possível observar no excerto de código apresentado em Código 1 este tipo de código não é genérico[27] e apresenta por exemplo dependências com nomes de classes.

```
1.  if (methodReturnClass != null && TypeUtils.isAssignable(methodReturnClass, BOAdaptable.class) && !m
    ethodName.endsWith("_")) {
2.      String boName = getBOAdaptableName(methodReturnClass);
3.      String boInterface = methodReturnClass.getSimpleName().replace("[]", "");
4.      String boAdaptableClassName = boInterface.replace("BOAdaptable", "") + ADAPTABLE;
5.      String simpleName = boInterface.replace("SSSBOAdaptable", "");
6.      String directArrayGetter = "get" + simpleName + "s_";
7.      if (isParent(clazz, simpleName) && !isChildren(clazz, simpleName)) {
8.          continue;
9.      }
10.     Map <String, Object> values = new HashMap <> ();
11.     if (methodName.startsWith("getAll")) {
12.         values.put("getter", methodName);
13.         values.put("setter", "set" + methodName.substring(3));
14.         values.put("addMethod", "add" + methodName.substring(6));
15.         values.put("methodBase", methodName.substring(6));
16.         values.put("directArrayGetter", directArrayGetter);
17.         values.put("boInterface", boInterface);
18.         values.put("boName", boName);
19.         values.put("boAdaptableName", boAdaptableClassName);
20.         multipleChildBOs.add(values);
21.         usedMethods.add(methodName);
```

Código 1 - Excerto código extração de informação do modelo definido por classes Java

A Figura 28 e a Figura 29 apresentam a hierarquia de classes do modelo incluindo as classes geradas. Durante o processo de geração de código são criadas classes

[27] Outros exemplos da complexidade deste código podem ser visualizados nos Anexos em Código 31 e Código 32.

Abstract+***BONAME***+Adaptable, são estas classes que contém o código utilitário gerado, como por exemplo: método *cloneBO* para permitir executar a cópia integral de uma determinada instância de um BO; método *copyBO* para permitir executar uma cópia de negócio de um determinado BO, útil para quando um utilizador pretende copiar um determinado contrato de seguro e executar uma série de alterações sobre esse contrato; métodos utilitários para remover e adicionar um determinado BO filho do BO atual; entre outros métodos.

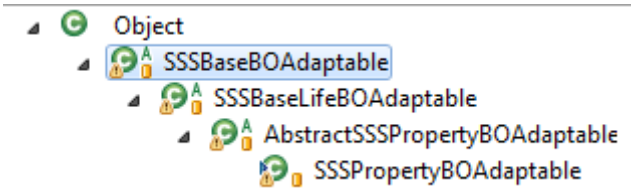


Figura 28 – Hierarquia de classes no modelo existente para a entidade SSSPropertyBO

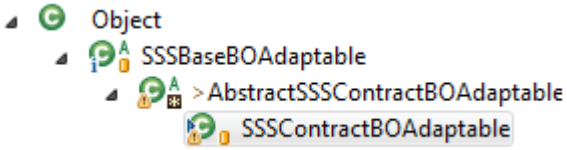


Figura 29 - Hierarquia de classes no modelo existente para a entidade SSSContractBO

Tomando o exemplo apresentado na Figura 29 podemos observar nesta hierarquia que existe uma classe base designada *SSSBaseBOAdaptable*, esta classe contém todo o código comum a todas as entidades do modelo. Como subclasse de *SSSBaseBOAdaptable* temos então a classe *AbstractSSSContractBOAdaptable* e finalmente a classe *SSSContractBOAdaptable* que representa a entidade de modelo neste caso.

O *design* deste tipo de estrutura apresenta um outro problema no que se refere à adição de novas entidades ao modelo, como existe uma classe gerada no meio da hierarquia para adicionar um novo BO é necessário executar um passo intermédio de geração dessa classe. Isto leva à necessidade de definir um *template* Velocity intermédio para gerar uma classe vazia que permita garantir a hierarquia esperada, uma vez que o processo de geração de código não funciona corretamente sem esta hierarquia nem se existirem erros de compilação nas classes Java representativas do modelo. As técnicas de *reflection* falham se existirem erros de compilação.

8.1.2 Criação do novo modelo Ecore

Para a criação do novo modelo Ecore do S&S foram utilizadas as ferramentas EMF e Emfatic. Para tal foi necessário proceder à preparação de um ambiente Eclipse com esses *plugins* instalados. A melhor forma de preparar um ambiente para esta tarefa é instalando uma versão

orientada para modelação[28] que já contém a maioria dos *plugins* necessários. Adicionalmente é necessário instalar o *plugin* Emfatic[29] para obter o ambiente necessário.

Após ter o ambiente necessário preparado para proceder com a criação de um modelo Ecore para o S&S as seguintes tarefas foram realizadas:

1. Análise e levantamento de particularidades do modelo Java existente;
2. Criação de uma representação textual em Emfatic de todas as entidades, atributos e relações identificadas no ponto anterior;
3. Criação do modelo Ecore correspondente utilizando a capacidade de conversão da ferramenta Emfatic para um modelo em formato Ecore;
4. Validação de que todas as particularidades identificadas durante a tarefa um foram preservadas e se encontram representadas no novo modelo.

Durante a execução da primeira tarefa, de levantamento de particularidades, foram identificadas todas as relações entre as diferentes entidades do modelo e catalogadas as anotações existentes utilizadas para identificar a necessidade de comportamento específico durante a geração de código. Para além disso foram também identificados todos os atributos do modelo e o seu tipo de dados de forma a identificar a necessidade de criação de tipos de dados complexos ou distintos.

Após ter esta informação catalogada procedeu-se à realização da tarefa dois. Para realizar esta tarefa numa fase inicial tentou-se definir o modelo Ecore utilizando a ferramenta gráfica existente na ferramenta EMF, como podemos ver na Figura 30 esta ferramenta gráfica disponibiliza todos os elementos do metamodelo EMF Ecore.

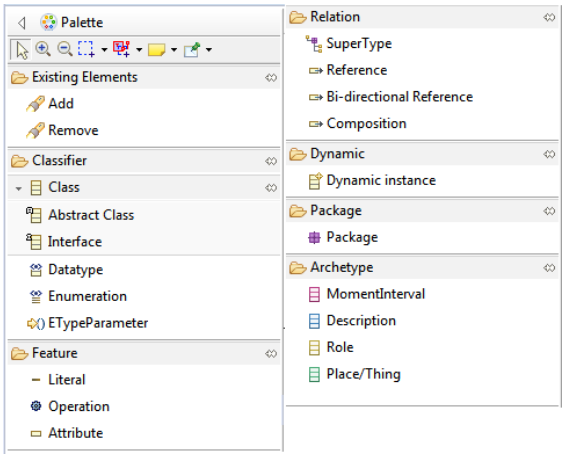


Figura 30 – Elementos e ações da ferramenta gráfica EMF (EcoreTools) de edição de modelos Ecore

[28] Versão Eclipse para modelação disponível em: http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/neon3

[29] *Plugin* Eclipse Emfatic disponível através do *update site*: http://download.eclipse.org/emfatic/update

No entanto, após a utilização da ferramenta gráfica para definir algumas das entidades no novo modelo, a utilização da ferramenta foi-se tornando uma tarefa cada vez mais complicada e pouco gratificante, como se pode observar na Figura 31 pela quantidade de elementos no ecrã. Como tal foi considerada a utilização da ferramenta textual de definição de modelos Emfatic como uma alternativa mais alinhada com esta tarefa. Tanto pela maior facilidade de definição dos elementos constituintes como pelo facto de facilitar a manutenção de um histórico de alterações ao modelo através da utilização de qualquer ferramenta de versionamento[30] como Mercurial[31] ou Git[32].

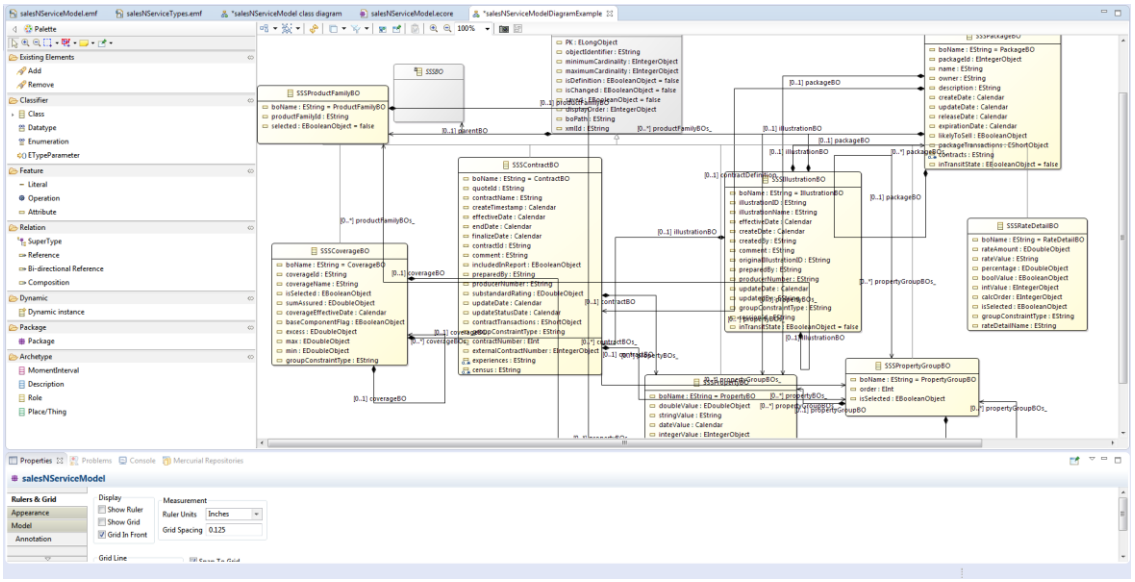


Figura 31 – Utilização da ferramenta gráfica EMF Ecore na definição do modelo Ecore para o S&S

Na definição do modelo utilizando Emfatic foram definidos dois ficheiros ***.emf***:

- ***salesNServiceTypes.emf*** – para representar os tipos de dados utilizados no modelo;
- ***salesNServiceModel.emf*** – para representar as entidades do modelo.

No modelo do S&S é utilizado um tipo de dados designado por *AbstractDiscreteValue* que tal como o nome indica representa um valor discreto de um determinado domínio. Como podemos observar no Código 2 este objeto permite armazenar informação sobre o acrónimo, o nome, a descrição e a ordem de um determinado valor de domínio. Normalmente numa determinada entidade do modelo podem existir um ou mais atributos que utilizem este tipo de dados. Nesse contexto existe um atributo que tem por exemplo um atributo productType do tipo de dados *ProductType*. A classe *ProductType* contém uma lista com todos os valores possíveis para o atributo productType.

[30] *Software* que permite manter um histórico de alterações no código ao longo do tempo. Mais informação disponível em: https://www.atlassian.com/git/tutorials/what-is-version-control
[31] Mais informação disponível em: https://www.mercurial-scm.org/
[32] Mais informação disponível em: https://git-scm.com/

```
abstract class AbstractDiscreteValue {
        attr Integer ~id;
        attr String acronym;
        attr String name;
        attr String shortText;
        attr Integer order;
}
```

Código 2 - Representação textual Emfatic do tipo de dados *AbstractDiscreteValue*

Tomando como exemplo a entidade SSSContractBO do modelo e pela análise da Figura 32 podemos visualizar a existência de vários atributos do tipo *AbstractDiscreteValue* como SSSProductType, SSSProductFamily, SSSContractStatus e SSSContractType.

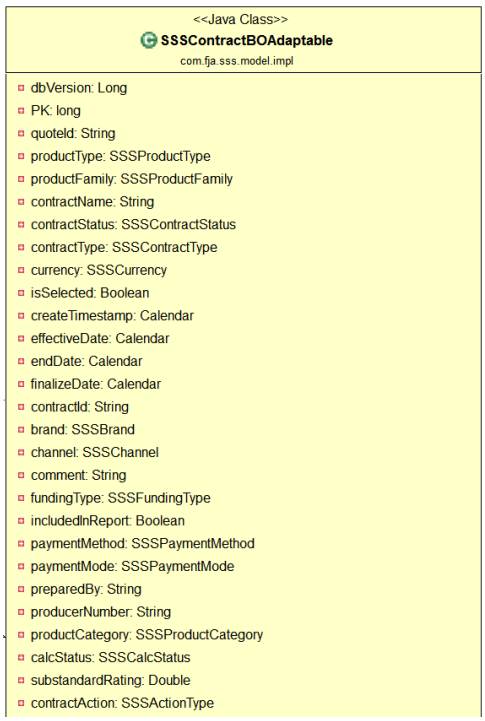


Figura 32 – Lista parcial de atributos da classe SSSContractBO

Na definição do novo modelo continua a ser necessário incluir todos os tipos de dados específicos de um SSSContractBO. Nesse sentido na representação textual Emfatic dos tipos de dados continuamos a ter a definição desses *AbstractDiscreteValues,* tal como é possível observar no Código 3. A representação completa dos tipos de dados para cada tipo de entidade do modelo pode ser consultado nos Anexos no Código 33.

```
// Contract AbstractTypes
class ProductType extends AbstractDiscreteValue {}
class ProductFamily extends AbstractDiscreteValue {}
class ContractStatus extends AbstractDiscreteValue {}
class ContractType extends AbstractDiscreteValue {}
class Currency extends AbstractDiscreteValue {}
class Brand extends AbstractDiscreteValue {}
class Channel extends AbstractDiscreteValue {}
class FundingType extends AbstractDiscreteValue {}
class PaymentMethod extends AbstractDiscreteValue {}
class PaymentMode extends AbstractDiscreteValue {}
class ProductCategory extends AbstractDiscreteValue {}
class CalcStatus extends AbstractDiscreteValue {}
class ActionType extends AbstractDiscreteValue {}
class Language extends AbstractDiscreteValue {}
```

Código 3 - Representação textual Emfatic dos tipos de dados AbstractDiscreteValue necessários para a entidade SSSContractBO

Após termos definido os tipos de dados necessários, temos agora de definir as entidades do modelo, os seus atributos e relações. De forma a exemplificar o funcionamento da definição textual vamos utilizar as classes SSSIllustrationBO, SSSContractBO e SSSPropertyBO.

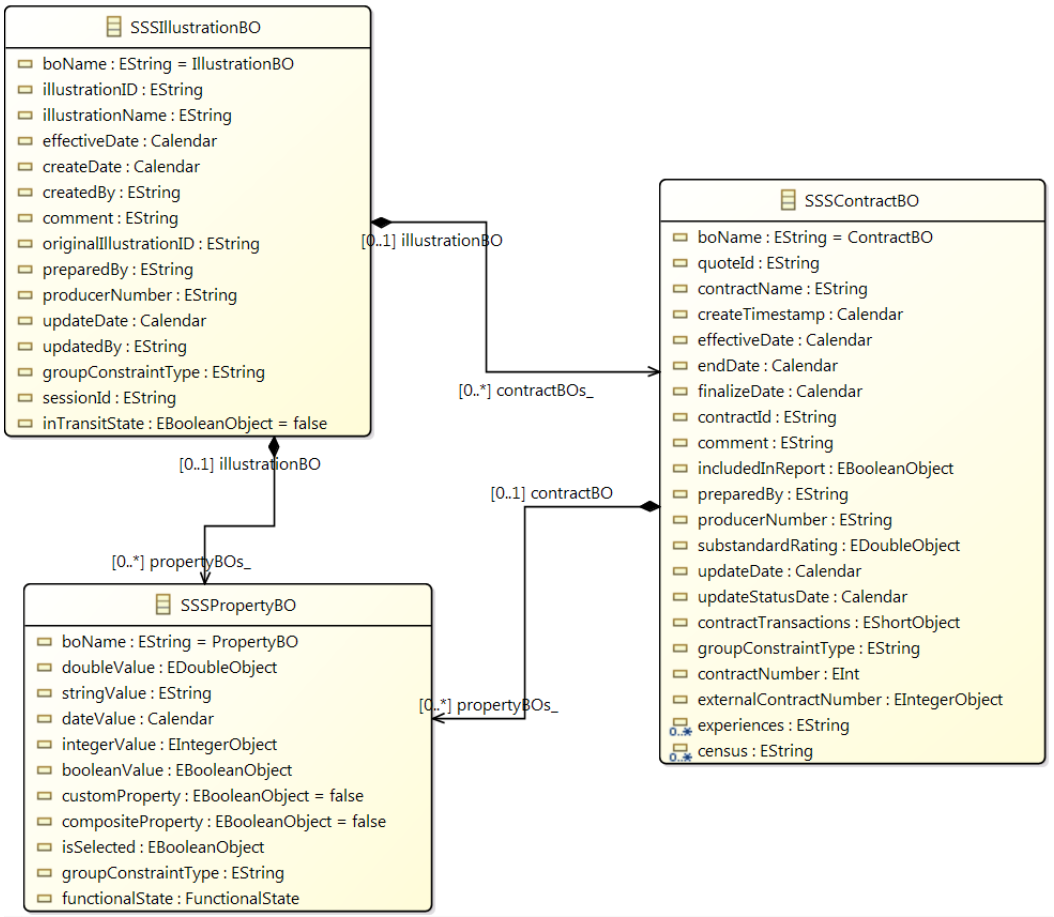


Figura 33 - Diagrama de classes para SSSIllustrationBO, SSSContractBO e SSSPropertyBO

Ao observar a Figura 33 podemos compreender que um objeto SSSIllustrationBO tem uma relação de composição com um objeto SSSContractBO, onde um SSSIllustrationBO pode ter zero

ou múltiplos SSSContractBO e um SSSContractBO tem zero ou no máximo um SSSIllustrationBO. Para representar este tipo de relação em formato Ecore o metamodelo EMF define um tipo de relação Containment EReference neste tipo de agregação estamos a indicar que A tem B, neste tipo de relação existe uma identificação da entidade parent (Budinsky, 2004). Na ferramenta Emfatic este tipo de relação é representado como se apresenta no Código 4 pela utilização da *keyword val* para identificar a entidade Contract contida na entidade Illustration e pela utilização da *keyword ref* para identificar a entidade *parent* na entidade Contract (Daly, 2004).

```
class Illustration extends BaseBO {
        …
        // Children multiple
        @ChildBO
        val Contract[*]#illustrationBO contractBOs;
        …
}
class Contract extends BaseBO {
        …
        // Parent relations
        @ParentBO
        ref Illustration#contractBOs illustrationBO;
        …
}
```

Código 4 – Representação Emfatic da relação Containment EReference de SSSIllustrationBO para SSSContractBO

Um aspeto importante na definição da relação Illustration contém Contract é a necessidade de identificar a entidade oposta da relação, ou seja, se observarmos com atenção temos: `val Contract[*]#illustrationBO contractBOs;` a parte destacada é o identificador da *EReference* oposta que é utilizado no Contract para permitir identificar que um Contract tem um objeto *parent* Illustration. Se não existir uma referência oposta não é necessário definir este tipo de informação (Daly, 2004).

A representação textual completa para a entidade SSSIllustrationBO é apresentada no Código 5, nessa representação temos a definição de todos os atributos constituintes da entidade bem como todas as relações com as entidades contidas. Para permitir a utilização dos tipos de dados *AbstractDiscreteValue* definidos anteriormente é necessário utilizar o nome do *package* correspondente, tomando como exemplo: `val type.IllustrationStatus` a parte destacada representa o nome do package no qual foram definidos os tipos de dados. De salientar ainda a utilização das anotações *IdField* e *ChildBO* que permitem adicionar conhecimento explícito e necessário para o processo posterior de geração de código. A anotação *IdField* tal como o nome indica identifica o atributo a utilizar como identificador único da entidade. Temos as anotações *ChildBO* e *ParentBO* que permitem identificar explicitamente as entidades pai e filho no modelo do S&S.

```
class Illustration extends BaseBO {
	unsettable attr String boName = "IllustrationBO";
	@IdField
	attr String illustrationID;
	attr String illustrationName;
	val type.IllustrationStatus illustrationStatus;
	attr Calendar effectiveDate;
	attr Calendar createDate;
	attr String createdBy;
	attr String comment;
	attr String originalIllustrationID;
	attr String preparedBy;
	attr String producerNumber;
	attr Calendar updateDate;
	attr String updatedBy;
	val type.IllustrationType illustrationtype;
	val type.YesNoType isTemplate;
	val type.BusinessType containerMode;
	val type.YesNoType lockedIndicator;
	attr String groupConstraintType;
	val type.GroupConstraintOperationType groupConstraintOperationType;
	transient attr String sessionId;
	attr Boolean inTransitState = false;
	// Parent relations

	// Children single
	@ChildBO
	val Producer#illustrationBO producerBO;

	// Children multiple
	@ChildBO
	val Contract[*]#illustrationBO contractBOs;
	@ChildBO
	val Package[*]#illustrationBO packageBOs;
	@ChildBO
	val ProductFamily[*]#illustrationBO productFamilyBOs;
	@ChildBO
	val Experience[*]#illustrationBO experienceBOs;
	@ChildBO
	val Role[*]#illustrationBO roleBOs;
	@ChildBO
	val Property[*]#illustrationBO propertyBOs;
}
```

Código 5 – Representação Emfatic completa da entidade Illustration

Podemos observar a utilização de *modifiers* como *unsettable* e *transient* para permitir identificar um campo como não modificável ou cujo valor é temporário e não deve ser persistido. Estes *modifiers* também existem no metamodelo Ecore EMF daí ser possível definilos usando Emfatic.

Outro aspeto a referir é a possibilidade de definir comentários na definição textual que permitem aumentar a facilidade de leitura do modelo.

A representação completa do novo modelo Ecore do S&S em formato textual Emfatic encontra-se disponível nos Anexos no Código 34.

A última tarefa necessária para obter o novo modelo Ecore consiste apenas na utilização da funcionalidade “Generate Ecore Model” como apresentado na Figura 34.

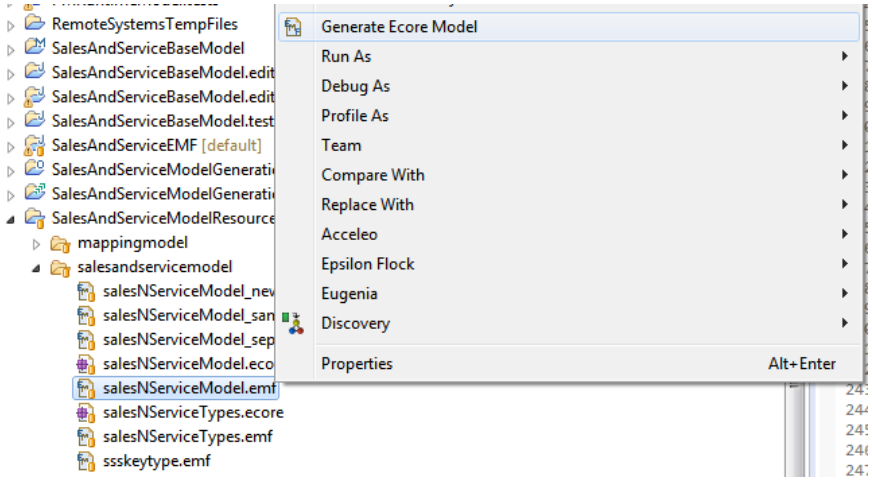


Figura 34 – Funcionalidade de geração do modelo Ecore usando Emfatic

Após realizar esta ação um novo ficheiro *salesNServiceModel.ecore* é criado contendo a representação Ecore do modelo, como apresentado na Figura 35.

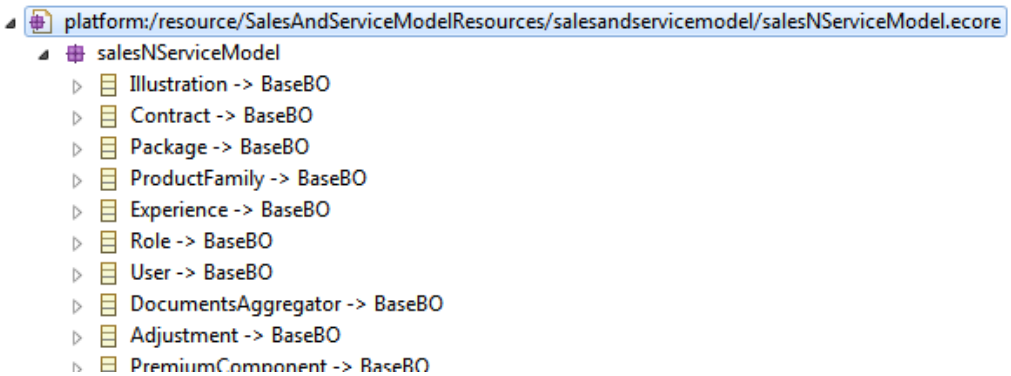


Figura 35 – Visualização parcial do novo do S&S em formato Ecore criado usando Emfatic

8.1.3 Geração de código usando Texo

A tarefa de geração de código de acordo com o novo modelo Ecore definido foi realizada utilizando a ferramenta Texo. Para tal foi necessário preparar um ambiente Eclipse com os *plugins* necessários, nomeadamente Texo[33] e Acceleo[34]. Após instalar esses *plugins* procedeu-se à criação de um novo projeto no qual se colocou o ficheiro *.ecore* do modelo do S&S e procedeu-se à geração de código utilizando as opções disponibilizadas pelo *plugin Texo*[35] como apresentado na Figura 36.

[33] Informação e passos necessários para realizar a instalação do *plugin* Texo disponível em: https://wiki.eclipse.org/Texo/Download_and_Install

[34] Informação e passos necessários para realizar a instalação do *plugin* Acceleo disponível em: https://wiki.eclipse.org/Acceleo/Installation

[35] Informação sobre o processo de geração de código usando o *plugin* Texo disponível em: https://wiki.eclipse.org/Texo/QuickStart

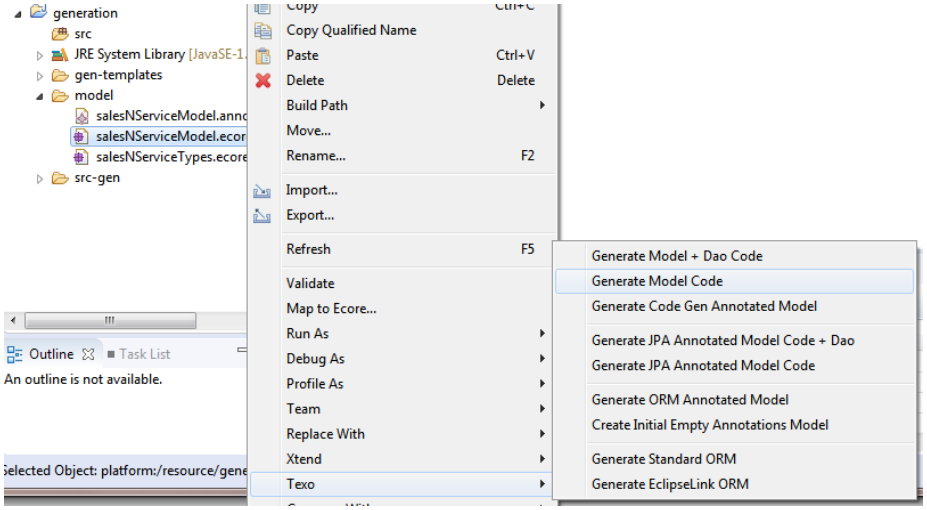


Figura 36 – Geração de código para o modelo do S&S usando o *plugin* Texo

No entanto para responder às necessidades específicas da solução S&S seria necessário permitir facilmente adicionar código extra ao código gerado pelos templates base. Para ultrapassar esta limitação a ferramenta Texo permite facilmente fazer *override* dos seus *templates* de geração ou adicionar novos *templates*. Nesse sentido procedeu-se à especificação do *template entity_addition.xpt* como apresentado na Figura 37. Este *template* é invocado durante o processo de geração de código para cada entidade do modelo e tem por objetivo permitir a adição de código na classe a gerar, ou seja, permite atingir exatamente o objetivo pretendido.

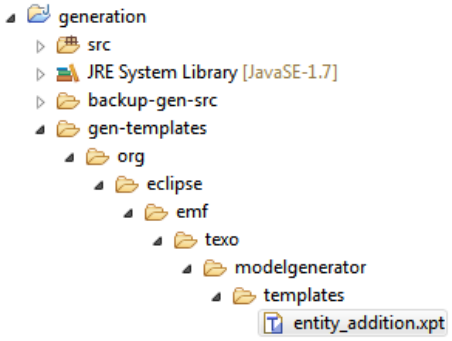


Figura 37 – Adição de novo *template* Texo *entity_adition.xpt*

Numa fase inicial apenas se definiu um novo método simples para validar a inclusão desse método no código gerado. No entanto após realizar esse teste inicial verificou-se que a ferramenta apresentava uma desvantagem que não tinha sido inicialmente identificada. O processo de geração de código para o modelo do S&S demorava consideravelmente mais tempo que o esperado, tendo por base de comparação o processo de geração utilizando *reflection* e *templates* Velocity, previamente apresentado no capítulo 8.1.1, que demorava em média 5 segundos. A geração de código usando Texo demorava em média 50 segundos. Este tempo de execução iria tornar a tarefa de alteração e validação de código gerado num processo demorado nesse sentido procedeu-se a uma análise dos tempos de execução da geração. A conclusão foi que a ferramenta Texo disponibiliza as capacidades de adição de *templates* em

formato *.xpt*, ou seja, estes *templates* devem ser definidos utilizando a ferramenta Xpand[36]. Com base nesta informação e após investigação de outras alternativas (Goubert, 2010) foi decidido que a melhor alternativa seria adicionar a capacidade de definição de novos *templates* utilizando a ferramenta Acceleo.

Para proceder a essa tarefa de extensão obteve-se o código fonte aberto[37] da ferramenta Texo e após análise da arquitetura dos vários *plugins* existentes foi possível identificar a possibilidade de definir um novo *extension-point*[38] (Vogel, 2016) (Blewitt, 2013) que permitiria a contribuição de novos tipos de geradores de código utilizando a ferramenta Acceleo. O novo *extension-point* apresentado no Código 6 foi adicionado ao *plugin* Texo responsável pela gestão do processo de geração: *org.eclipse.emf.texo.generator*

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>
   <extension-point id="org.eclipse.emf.texo.extensionpoint.annotationmodels" name="%annotationModels"
schema="schema/org.eclipse.emf.texo.extensionpoint.annotationmodels.exsd"/>
   <extension-point id="org.eclipse.emf.texo.extensionpoint.modelannotators" name="%modelAnnotators"
schema="schema/org.eclipse.emf.texo.extensionpoint.modelannotators.exsd"/>
   <extension-point id="org.eclipse.emf.texo.extensionpoint.codegenerators" name="%codeGenerators"
schema="schema/org.eclipse.emf.texo.extensionpoint.codegenerators.exsd"/>
</plugin>
```

Código 6 – Novo *extension-point codegenerators* adicionado à ferramenta Texo

O código necessário para permitir executar a leitura dos novos geradores de código foi adicionado à classe *ExtensionPointUtils* já existente no *plugin.* Como apresentado no Código 7 esse código permite obter uma lista de *CodeGenerator* que posteriormente serão executados durante o processo de geração.

```
1.  /**
2.  * Takes care of reading extensions provided by other plugins and setting the information in the relevant Texo
3.  * registries.
4.  * @author <a href="mtaal@elver.org">Martin Taal</a>
5.  */
6.  public class ExtensionPointUtils {
7.      private static List<CodeGenerator> codeGenerators;
8.      public static List<CodeGenerator> readCodeGeneratorsFromExtensions() {
9.          if (codeGenerators == null) {
10.             IConfigurationElement[] config = Platform.getExtensionRegistry().getConfigurationEle
    mentsFor(CODE_GENERATORS_EXTENSION);
11.             codeGenerators = new ArrayList<CodeGenerator>();
12.             for (IConfigurationElement extension: config) {
13.                 try {
14.                     CodeGenerator codeGenerator = (CodeGenerator) extension.createExecutableExte
    nsion(ATT_CLASS);
15.                     codeGenerators.add(codeGenerator);
16.                 } catch (CoreException e) {
17.                     throw new RuntimeException(e);
18.                 }
```

[36] Linguagem especializada na geração de código baseada na utilização de modelos EMF. Mais informação disponível em: http://wiki.eclipse.org/Xpand

[37] Código fonte da ferramenta Texo disponível em: https://wiki.eclipse.org/Texo/Developing#GIT

[38] Eclipse *extension-point* é um conceito de desenvolvimento de *plugins* Eclipse que permite a contribuição de novas funcionalidades para um determinado *plugin*. Mais informação disponível em: http://www.vogella.com/tutorials/EclipseExtensionPoint/article.html

```
            }
        }
        return codeGenerators;
    }
```

Código 7 – Método de leitura de novos geradores de código adicionado ao *plugin* Texo

Foi ainda definida uma nova interface *CodeGenerator* como apresentado em Código 8.

```
public interface CodeGenerator {
    public String generate(Map<String, Object> configurations, ArtifactGenerator artifactGenerator, Object mainObject);
}
```

Código 8 – Nova interface *CodeGenerator*

Depois de adicionar o código de leitura de novos *CodeGenerators* foi também adicionado o código que será invocado durante a geração. Este Código 9 permite obter o conteúdo gerado por vários geradores e procede à sua concatenação retornando finalmente esse conteúdo.

```
/**
* Base type for all xtend2 templates.
* @author <a href="mtaal@elver.org">Martin Taal</a>
*/
public abstract class BaseTemplate {
    protected String executeCodeGenerators(TemplateType templateType, Object mainObject) {
        long start = System.currentTimeMillis();
        List<CodeGenerator> codeGenerators = ExtensionPointUtils.readCodeGeneratorsFromExtensions();
        Map<String, Object> configurations = new HashMap<String, Object>();
        configurations.put(ConfigurationType.TEMPLATE.toString(), templateType.toString());
        StringBuilder strBuilder = new StringBuilder();
        for (CodeGenerator codeGenerator: codeGenerators) {
            strBuilder.append(codeGenerator.generate(configurations, artifactGenerator, mainObject));
        }
        String result = strBuilder.toString();
        if (result.isEmpty()) {
            result = "// Empty String";
        } else {
            result = "//" + result;
        }
        return result;
    }
```

Código 9 – Método de execução dos novos geradores de código obtidos

Este método é invocado durante o processo nativo de geração da ferramenta Texo, para tal foi alterado um dos *templates* base que trata de gerar o código de cada entidade do modelo. O *template EntityTemplate.xtend* foi alterado para substituir o conteúdo apresentado em Código 10 pela nova forma de invocação de geradores adicionais apresentados no Código 11. Desta forma permitindo iniciar o processo de obtenção de conteúdo adicional gerado por outras ferramentas ao invocar o método *execureCodeGenerators*.

```
«executeXPandTemplate("org::eclipse::emf::texo::modelgenerator::templates::entity_addition",
eClassModelGenAnnotation)»
```

Código 10 – Execução original de *EntityTemplate.xtend* responsável pela adição de conteúdo

```
«executeCodeGenerators(org.eclipse.emf.texo.generator.BaseTemplate.TemplateType.ENTITY,
eClassModelGenAnnotation)»
```

Código 11 – Execução alterada de *EntityTemplate.xtend* responsável pela adição de conteúdo

De seguida foi criado um novo *plugin* Eclipse *org.eclipse.emf.texo.generator.acceleo*, este novo *plugin* define uma extensão e implementa o código necessário para realizar a contribuição para o *extension-point org.eclipse.emf.texo.extensionpoint.codegenerators* definido anteriormente. Tal é definido no ficheiro *plugin.xml* através da inclusão de uma entrada *extension* e da declaração de um *codeGenerator*.

A implementação concreta da extensão declarada é realizada pela classe *AcceleoGeneratorLauncher* que tal como podemos observar no Código 12 implementa a interface *CodeGenerator* anteriormente apresentada no Código 8. Esta classe é responsável por iniciar o processo de geração utilizando a ferramenta Acceleo.

```
1.  public class AcceleoGeneratorLauncher extends BaseTemplate implements CodeGenerator {
2.      protected String acceleo(Object mainObject) {
3.          String result = "";
4.          try {
5.              File folder = new File(artifactGenerator.getOutputFolder());
6.              EObject modelObject = (EObject) mainObject;
7.              List < String > arguments = new ArrayList < String > ();
8.              GenerateEntity generate;
9.              generate = new GenerateEntity(modelObject, folder, arguments);
10.             generate.doGenerate(new BasicMonitor());
11.             result = generate.getGenerationResult().entrySet().iterator().next().getValue();
12.             if (result.isEmpty()) {
13.                 result = "No Exception but no content";
14.             }
15.         } catch (IOException e) {
16.             e.printStackTrace();
17.             result = e.getMessage();
18.         }
19.         return result;
20.     }
21.     @Override
22.     public String generate(Map<String, Object> configurations,
23.     ArtifactGenerator artifactGenerator, Object mainObject) {
24.         setArtifactGenerator(artifactGenerator);
25.         if (configurations.containsValue(TemplateType.ENTITY.toString())) {
26.             return acceleo(mainObject);
27.         }
28.         return "";
29.     }
30. }
```

Código 12 – Código fonte da classe *AcceleoGeneratorLauncher*

Na Figura 38 apresenta-se um diagrama de sequências no qual podemos visualizar o fluxo executado durante o processo de geração relativamente à obtenção dos novos geradores de código. O processo é iniciado anteriormente por uma ação de geração de código que é tratada pela ferramenta Texo até eventualmente chegar à execução do EntityTemplate. Este irá invocar

o método *executeCodeGenerators* passando informação sobre o tipo de *template* e a entidade do modelo a processar. Durante a execução do *executeCodeGenerators* será invocado o método *readCodeGeneratorsFromExtensions* da classe *ExtensionPointUtils* que procede à leitura da lista de geradores de código existentes. Após obter essa lista, para cada um dos geradores de código é invocado o método *generate* passando informação de configuração, o controlador de geração e a entidade do modelo. Internamente cada *CodeGenerator* gera o código e retorna o conteúdo gerado que é concatenado pelo *BaseTemplate* e finalmente adicionado ao restante código gerado.

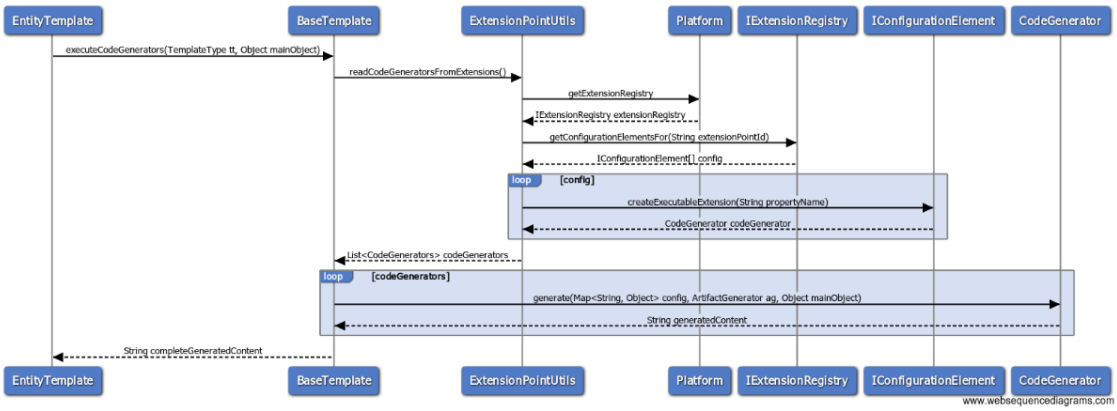


Figura 38 – Diagrama de sequência da obtenção de geradores de código

Um novo projeto Acceleo designado *com.msg.emf.model.generator* foi criado para permitir definir o novo *template*[39] de adição de código a ser utilizado durante a execução de código Texo. Neste novo projeto foi adicionado o *template generateEntity.mtl* que utiliza como modelo de entrada o modelo anotado criado pelo Texo no momento de preparação da geração de código. No Código 13 é possível visualizar um pequeno excerto exemplificativo do *template* Acceleo criado.

[39] Informação sobre o processo de criação de um novo projeto Acceleo pode ser visualizada em: https://wiki.eclipse.org/Acceleo/Getting_Started

```
[**
 * The documentation of the template generateElement.
 * @param anEClass
 */]
[template public generateElement(anAnnotatedEClass : EClassModelGenAnnotation)]
[comment @main/]
[file (anAnnotatedEClass.simpleClassName, false, 'UTF-8')]

/**
* @generated
*/
protected static final List<String> declaredFields = new ArrayList<String>();

/**
* @generated
*/
static {
        [for (featureAnnotation : EStructuralFeatureModelGenAnnotation |
anAnnotatedEClass.getAllFeatures())]
                [if (featureAnnotation.generateCode)]
                        declaredFields.add("[featureAnnotation.validJavaMemberName/]");
                [/if]
        [/for]
}

/**
* @generated
*/
@Override
public List<String> getDeclaredFields() {
        return declaredFields;
}

/**
* @generated
*/
protected static final List<String> idFields = new ArrayList<String>();
[if (not(anAnnotatedEClass.getIdfields()->isEmpty()))]
        /**
        * @generated
        */
        static {
                [for (idField : EStructuralFeatureModelGenAnnotation | anAnnotatedEClass.getIdfields())]
                    idFields.add("[idField.name/]");
                [/for]
        }
[/if]
```

Código 13 – Excerto do *template* Acceleo de adição do código necessário para cada entidade

Após acrescentar a capacidade de utilização de *templates* Acceleo foi novamente gerado o código e foram realizados alguns testes para obtenção de métricas de comparação do tempo de execução, tal como apresentado na Tabela 3 podemos observar que foi possível obter uma melhoria no tempo de execução de 49.38%.

Tabela 3 - Métricas tempos execução da geração código Xpand vs Acceleo

	Acceleo	Xpand
	Segundos	Segundos
Execução	27.862	41.233
1	27.004	42.801
2	25.199	52.795
3	24.132	56.257
4	24.866	56.305
5	20.938	46.965
Média	25.00017	49.39267

Xpand Média	49.39267
Acceleo Média	25.00017
Melhoria	49.38%

Para permitir uma fácil perceção das classes geradas e do tempo de execução foi também adicionada uma nova *dialog* após a geração de código como apresentado na Figura 65 esta nova funcionalidade apresenta uma listagem de todas as classes geradas e contém informação sobre a entidade do modelo, a classe correspondente gerada e o tempo de geração dessa classe. Esta nova *dialog* é apresentada utilizando a informação obtida durante o processo de geração e foi conseguida através da criação de uma nova classe *GenerationDetailsInfoTitleAreaDialog* que faz *override* à classe *TitleAreaDialog* disponibilizada nativamente pela plataforma Eclipse.

Para cada entidade existente no modelo é gerada uma classe que contém todo o código necessário e desta forma não existe qualquer tipo de hierarquia complexa no código gerado. O Código 14 apresenta um pequeno excerto do código gerado para a entidade SSSIllustrationBO.

```
1.  /**
2.  * A representation of the model object '<em><b>SSSIllustrationBO</b></em>'.
3.  * <!-- begin-user-doc --> <!-- end-user-doc -->
4.  * @generated
5.  */
6.  public class SSSIllustrationBO extends SSSBaseBO {
7.      /**
8.      * <!-- begin-user-doc --> <!-- end-user-doc -->
9.      * @generated
10.     */
11.     private String boName = "IllustrationBO";
12.     /**
13.     * <!-- begin-user-doc --> <!-- end-user-doc -->
14.     * @generated
15.     */
16.     private String illustrationID = null;
17.     /**
18.     * <!-- begin-user-doc --> <!-- end-user-doc -->
19.     * @generated
20.     */
21.     private List<SSSContractBO> contractBOs_ = new ArrayList<SSSContractBO>();
22.     /**
23.      * @generated
24.      */
25.     @Override
26.     @ SuppressWarnings({"squid:MethodCyclomaticComplexity", "squid:S1142"})
27.     public Integer getChildBOIndex(SSSBO sssBO) {
28.         if (sssBO instanceof SSSContractBO) {
```

```
29.             return getChildBOIndex(contractBOs_, sssBO);
30.         }
31.         if (sssBO instanceof SSSPackageBO) {
32.             return getChildBOIndex(packageBOs_, sssBO);
33.         }
34.         if (sssBO instanceof SSSProductFamilyBO) {
35.             return getChildBOIndex(productFamilyBOs_, sssBO);
36.         }
37.         if (sssBO instanceof SSSExperienceBO) {
38.             return getChildBOIndex(experienceBOs_, sssBO);
39.         }
40.         if (sssBO instanceof SSSRoleBO) {
41.             return getChildBOIndex(roleBOs_, sssBO);
42.         }
43.         if (sssBO instanceof SSSPropertyBO) {
44.             return getChildBOIndex(propertyBOs_, sssBO);
45.         }
46.         return null;
47.     }
```

Código 14 – Excerto de código gerado pela ferramenta Texo para a entidade SSSIllustrationBO

8.2 Implementação Componente Leitura

Para realizar a implementação do componente de leitura tal como definido anteriormente utilizou-se a ferramenta Modisco. Em primeiro lugar para permitir utilizar esta ferramenta foi necessário instalar o *plugin* Eclipse Modisco[40].

De seguida e com base na informação presente na documentação disponibilizada pela ferramenta Modisco[41] sobre o processo de adição de novos *discoverers* foram identificadas as tarefas:

1. Criação de um novo *plugin* Eclipse *org.eclipse.modisco.ecore.ecorediscoverer* para permitir adicionar o comportamento de obtenção de modelos Ecore;
2. Criação de um novo *plugin* Eclipse *org.eclipse.modisco.ecore.ecorediscoverer.ui* para permitir adicionar o novo *discoverer* criado ao menu de contexto da ferramenta Modisco.

8.2.1 Criação *plugin* obtenção modelos Ecore

Para permitir adicionar as capacidades de *reverse engineering* específicas para criar um modelo Ecore foi criado um novo *plugin* Eclipse. De acordo com a documentação Modisco para permitir que o catálogo de *discoverers* tenha conhecimento de um novo *discoverer* e possibilite a sua utilização é necessário registar o novo *discoverer* utilizando o *extension-point* *org.eclipse.modisco.infra.discovery.core.discoverer*. Como apresentado no Código 15 foi então

[40] Mais informação sobre os passos necessários para proceder à instalação do *plugin* Modisco disponível em: https://wiki.eclipse.org/MoDisco/Installation

[41] A documentação é disponibilizada diretamente no Eclipse. No entanto também se encontra disponível em: http://download.eclipse.org/modeling/mdt/modisco/nightly/doc/org.eclipse.modisco.doc/

definida a extensão no novo *plugin* especificando o identificador da extensão e a classe responsável por executar a operação de obtenção do modelo.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<?eclipse version="3.4"?>
<plugin>
   <extension point="org.eclipse.modisco.infra.discovery.core.discoverer">
      <discoverer class="org.eclipse.modisco.ecore.ecorediscoverer.EcoreModelDiscovererFromProject"
id="org.eclipse.modisco.ecore.ecorediscoverer"/>
</plugin>
```

Código 15 – Definição do novo *discoverer* através do *extension-point* Modisco

O próximo passo foi iniciar a implementação de um novo *discoverer* Modisco. A ferramenta Modisco define uma interface Java IDiscoverer que deve ser implementada pelo novo discoverer criado. Como apresentado no Código 16 esta interface define dois métodos:

- *isApplicableTo* – este método é utilizado para identificar se o objeto a ser manipulado atualmente é suportado pelo *discoverer*. O critério de validação deve ser implementado pelo *discoverer* de acordo com a sua lógica interna. Este método é utilizado pelo mecanismo genérico de gestão de *discoverers* do Modisco, a ideia é condicionar o conteúdo apresentado no menu de contexto que o utilizador visualiza quando clica num determinado elemento no seu editor Eclipse (seja um projeto Java, um package, uma classe Java entre outros elementos);
- *discoverElement* – método utilizado para executar o processo de análise sobre um determinado elemento selecionado pelo utilizador.

```
1.  /**
2.  * A discoverer is a process which discovers information from one input source.
3.  * Such a process can be parameterized through additional input data.
4.  *
5.  * A basic implementation is provided in {@link AbstractDiscoverer}.
6.  * In the context of model-driven reverse-engineering, a common category of
7.  * discoverers is characterized by injecting information into a result model.
8.  * Class {@link AbstractModelDiscoverer} provides a basic implementation for such discoverers.
9.  */
48. public interface IDiscoverer<T> {
10.     /**
11.     * To determine if the source object can be handled by the discoverer. Each
12.     * discoverer has to implement this method with its own criteria to filter   the selected object.
13.     * Some usual implementations are proposed on {@link AbstractDiscoverer}.
14.     * @param source  the selected object.
15.     * @return <code>true</code> if the selected object is managed by this  discoverer, <code>false</code> otherwise.
16.     */
49.     boolean isApplicableTo(T source);
17.     /**
18.     * Generic method to launch a discovery from a source element.
19.     * Additional discovery parameters values (input or output) should be
20.     * managed using fields and methods annotated with a {@link Parameter}
21.     * annotation. See the class {@link AbstractDiscoverer} as an example.
22.     * @param source  the selected object.
23.     * @param monitor  a progress monitor used to report progress and respond to
24.     *    cancellation. May be a {@link NullProgressMonitor} if no  monitor is to be used.
25.     * @throws DiscoveryException  abnormal discovery process termination
26.     */
27.     void discoverElement(T source, IProgressMonitor monitor) throws DiscoveryException;
28. }
```

Código 16 – Interface *IDiscoverer* definida pela ferramenta Modisco

Na Figura 39 apresenta-se um diagrama de classes que representa a hierarquia de classes criadas. Foi criada uma nova classe Java *AbstractEcoreModelDiscovererCustom* para permitir centralizar o código comum do novo tipo de *discoverer* a criar. Esta classe é uma subclasse da *AbstractDiscoverer* disponibilizada pela ferramenta Modisco que implementa a interface IDiscoverer e contém algum código utilitário comum para qualquer tipo de *discoverer*. A classe que realmente implementa a lógica específica do tipo de *reverse engineering* pretendido é a classe *EcoreModelDiscovererFromProject*. Esta classe está direcionada para tratar elementos da interface IAdaptable do Eclipse. Nesta classe temos o código concreto para os métodos *isApplicableTo, discoverElement* e outros métodos utilitários necessários para o processo de análise.

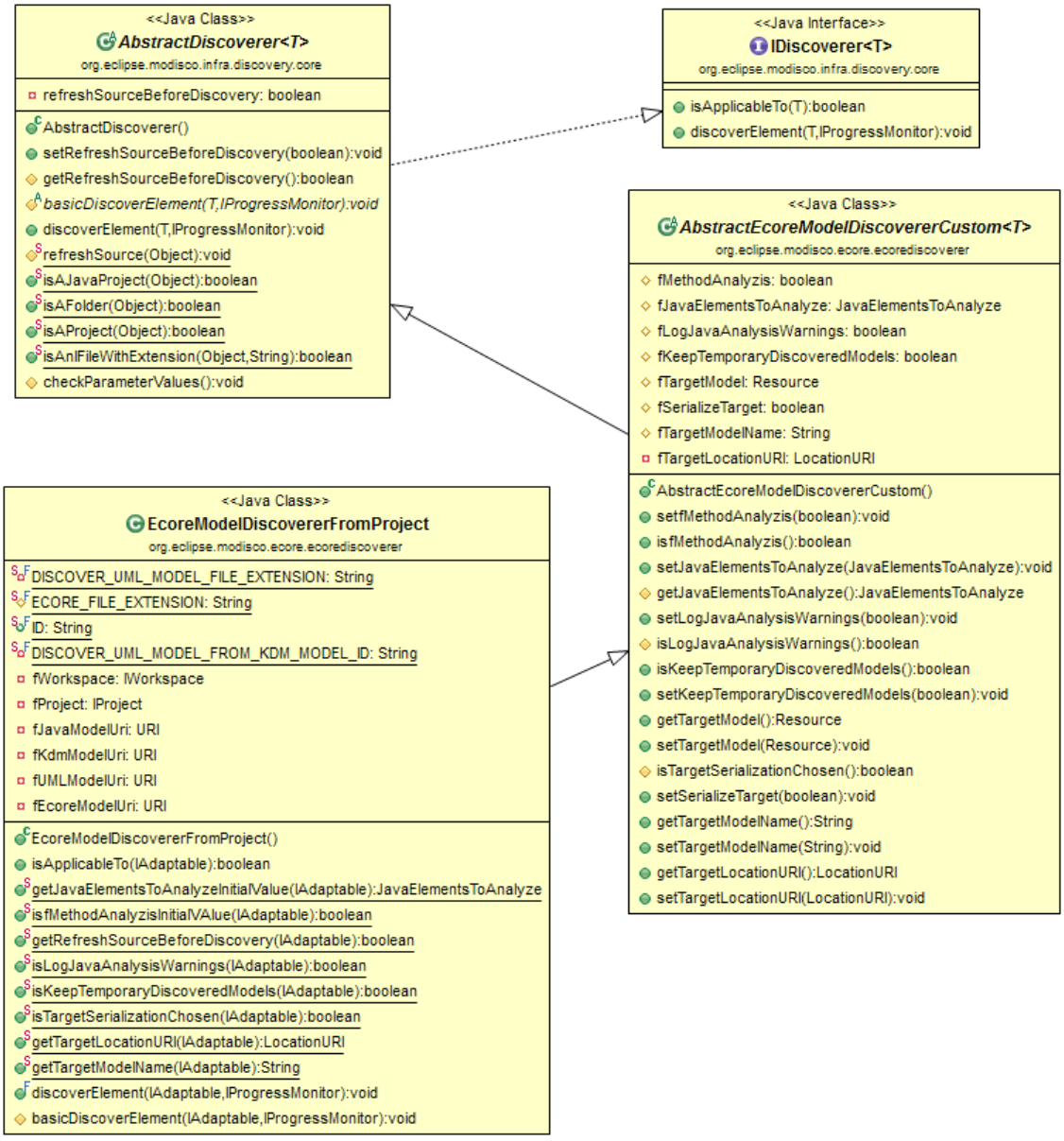


Figura 39 – Diagrama de classes do novo *discoverer* Modisco *EcoreModelDiscovererFromProject*

Como apresentado no Código 17 o método isApplicableTo contém lógica para validar se o elemento a processar é uma instância de um IProject, IJavaProject ou IPackageFragment que são os tipos de elementos de entrada que o novo *discoverer* deve ser capaz de processar. No caso prático de leitura de modelos que este componente tenta suportar esses tipos de elementos são os expectáveis.

```
1.  @Override
2.  public boolean isApplicableTo(final IAdaptable iAdaptable) {
3.      if (iAdaptable instanceof IProject) {
4.          try {
5.              IProject project = (IProject) iAdaptable;
6.              return project.isAccessible() && project.getNature(JavaCore.NATURE_ID) != null;
7.          } catch (CoreException e) {
8.              Logger.logError(e, Activator.getDefault());
9.              return false;
10.         }
11.     }
12.     if (iAdaptable instanceof IJavaProject) {
13.         IJavaProject javaProject = (IJavaProject) iAdaptable;
14.         return javaProject.exists() && javaProject.getProject() != null && javaProject.getProject(
    ).isAccessible();
15.     }
16.     if (iAdaptable instanceof IPackageFragment) {
17.         IPackageFragment packageFragment = (IPackageFragment) iAdaptable;
18.         return packageFragment.exists() && packageFragment.getResource().isAccessible() && packag
    eFragment.getResource() instanceof IContainer;
19.     }
20.     return false;
21. }
```

Código 17 – Método *isApplicableTo* da classe *EcoreModelDiscovererFromProject*

Na Figura 40 é apresentado o digrama de sequência que demostra o fluxo de execução para determinar se um *discoverer* é aplicável para um determinado elemento selecionado pelo utilizador. O processo começa pela seleção de um elemento Eclipse pelo utilizador, esta ação desencadeia um evento que será tratado pela classe *ContributionItemForModiscoMenu*, a lista total dos *discoverers* registados no *DiscoveryManager* é obtida e para cada um dos *discoverers* é invocado o método *isApplicableTo* passando o elemento selecionado pelo utilizador, se o resultado for positivo o *discoverer* é adicionado a uma lista *applicableDiscoverers*. No fim deste processo e de acordo com o elemento selecionado são apresentados os *discoverers* corretos no menu de contexto e o utilizador pode selecionar um deles para proceder com a obtenção do modelo.

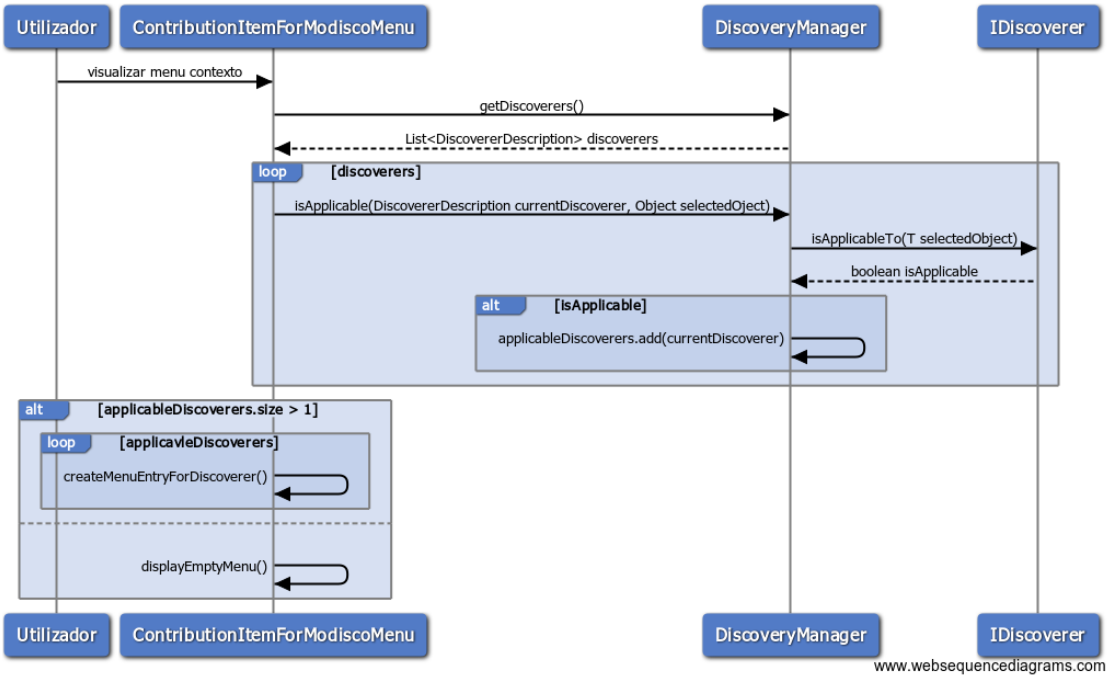


Figura 40 - Diagrama de sequência obtenção dos *discoverers* apresentados no menu contexto

De seguida procedeu-se à implementação do processo concreto de obtenção de um modelo Ecore com base num projeto ou package Java. O método *discoverElement*[42] é responsável pela execução das operações e transformações necessárias para obter a representação Ecore.

A obtenção de um modelo Ecore representativa de um projeto ou de um *package* Java pode ser obtido através da execução de uma série de transformações sucessivas entre formatos diferentes. Inicialmente é obtida uma representação Java de todos os elementos que representam a linguagem Java, esses elementos são representados utilizando um metamodelo da linguagem Java definido pela ferramenta Modisco [43] . De seguida é executada uma transformação para o formato Knowledge Discovery Metamodel – KDM[44]. Posteriormente é executada uma transformação do formato KDM para o formato UML e por fim é obtida a representação Ecore utilizando as capacidades da biblioteca *org.eclipse.uml2.uml.util.UMLUtil.*

Este processo de obtenção do modelo é apresentado em maior detalhe no diagrama de sequência dividido em três partes na Figura 41, Figura 42 e Figura 43. Podemos observar que o processo é iniciado quando o utilizador seleciona um *discoverer* no menu de contexto referente ao elemento Eclipse selecionado. Esta ação é tratada pela classe *ModiscoMenuSelectionListener*

[42] Um excerto do código deste método pode ser consultado nos Anexos no Código 35.

43 Mais informação sobre o metamodelo Java definido na ferramenta Modisco disponível em: http://download.eclipse.org/modeling/mdt/modisco/nightly/doc/org.eclipse.modisco.java.doc/mediawiki/java_metamodel/user.html

[44] KDM é uma especificação EMF de um metamodelo representativo de informação sobre os componentes e os vários elementos de software existente. Mais informação disponível em: http://download.eclipse.org/modeling/mdt/modisco/nightly/doc/org.eclipse.modisco.java.doc/mediawiki/java_metamodel/user.html

que dá início ao processo de *reverse-engineering* ao invocar o método *discoverElement* da classe *EcoreModelDiscovererFromProject*. Nesta classe são executadas algumas ações de preparação (*prepareDiscovery*) e de seguida o método comum *discoverElement* da classe AbstractDiscoverer é chamado passando o elemento a descobrir. Neste método as configurações do processo são validados e o elemento a descobrir é refrescado para garantir que se encontra atualizado. De seguida o método *basicDiscoverElement* da classe *EcoreModelDiscovererFromProject* é invocado e o fluxo de execução chega até ao método *discoverJavaModelFromSource* da mesma classe, neste método são preparadas as configurações necessárias (como: nome do ficheiro a gerar; tipo de análise da transformação a realizar) para executar a invocação do método *discoverElement* da classe *DiscoverJavaModelFromProjectCustom.* Internamente neste método é executada toda a lógica de análise do modelo Java representativo do elemento de entrada a analisar. Esta classe foi criada no contexto do trabalho realizado para permitir adicionar comportamento específico no que se refere à exclusão da análise de métodos Java, ou seja, o comportamento normal deste tipo de análise Modisco é descobrir todo o modelo Java, incluindo os métodos. Para facilitar o processo e tendo em conta que o modelo Ecore que se pretende obter não necessita de informação sobre métodos foi criada uma classe *JavaReaderCustom* que altera o comportamento normal da classe *JavaReader* e permite ignorar a análise dos métodos presentes no código. Nos Anexos é possível visualizar um diagrama de classes com a hierarquia das classes criadas na Figura 66 e o código fonte da classe *JavaReaderCustom* no Código 36.

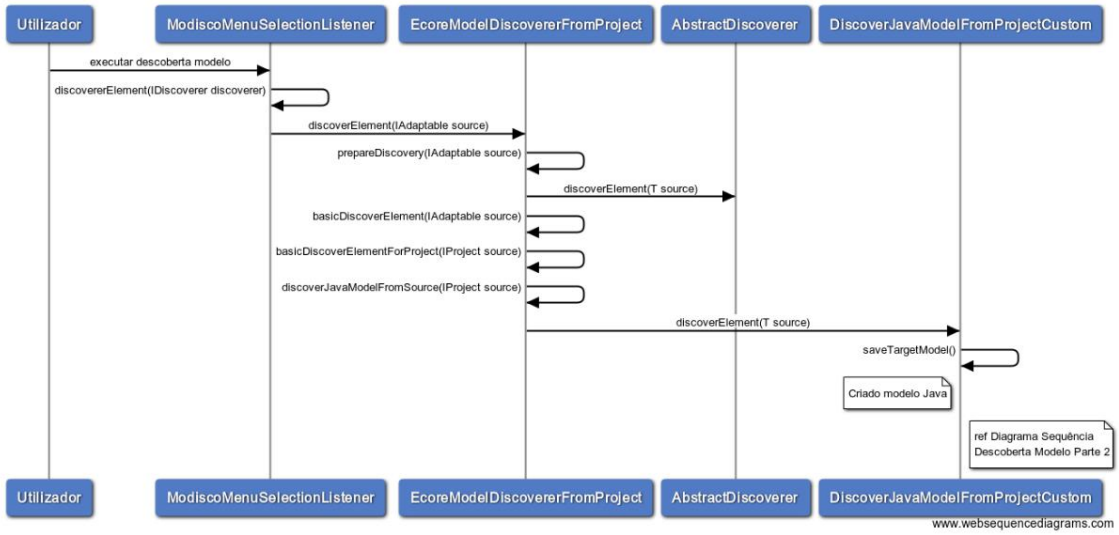


Figura 41 – Diagrama de sequência Descoberta de Modelo Ecore (Parte 1)

No fim do processo um novo modelo Java é criado e guardado em disco. O próximo passo do processo é iniciado pela invocação do método *discoverKdmModelFromJavaModel* da classe *EcoreModelDiscovererFromProject*, nesse método o modelo Java previamente criado é utilizado na chamada do método *discoverElement* da classe *DiscoverKDMModelFromJavaModel*, internamente nesta classe (disponibilizada nativamente pela ferramenta Modisco) é gerada uma representação KDM correspondente ao modelo Java recebido e esse novo modelo KDM é gravado.

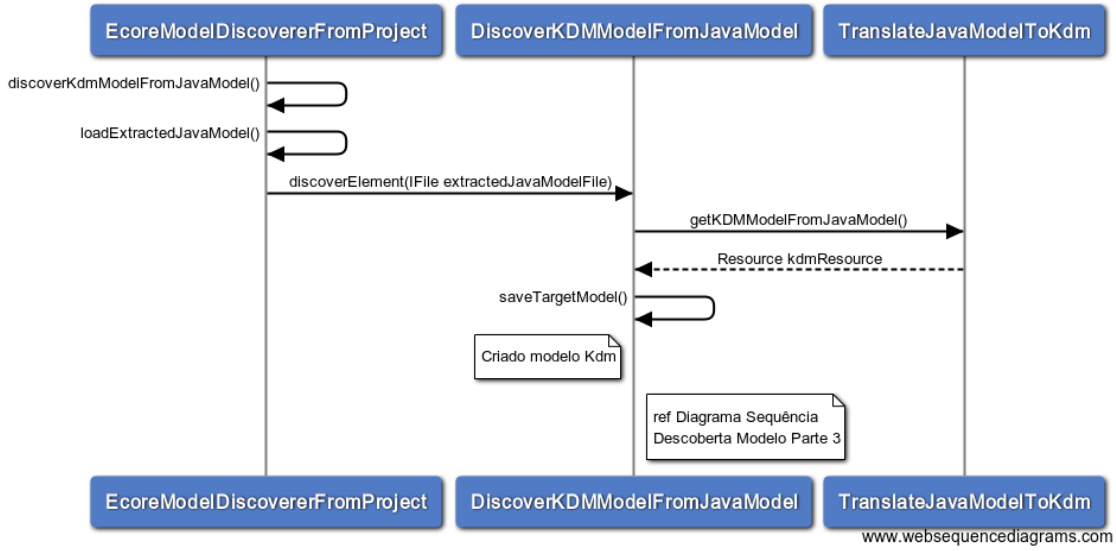


Figura 42 - Diagrama de sequência Descoberta de Modelo Ecore (Parte 2)

Após obter o modelo KDM o processo continua e o método *discoverUmlModelFromKdmModel* é executado, neste método o modelo KDM criado é enviado para o método *discoverElement* da classe *DiscoverUmlModelFromKdmModel.* Esta classe disponibilizada pela ferramenta Modisco obtém um modelo UML do modelo KDM recebido e esse novo modelo é gravado para servir de entrada na última transformação necessária. O modelo UML é enviado para o método *extractEcoreModel* da classe *UML2EcoreUtils* criada para permitir realizar a transformação do modelo UML num modelo Ecore programaticamente utilizando a classe *UMLUtil*[45] disponível na ferramenta Eclipse UML2[46]. O código fonte da classe UML2EcoreUtils pode ser consultado nos Anexos no Código 37. O novo modelo Ecore finalmente obtido é gravado e o processo termina.

[45] A classe *UMLUtil* disponibiliza métodos de transformação entre modelos Ecore e UML. Informação em: http://download.eclipse.org/modeling/mdt/uml2/javadoc/5.2.0/org/eclipse/uml2/uml/util/UMLUtil.html

[46] Informação sobre o projeto Eclipse UML2 disponível em: http://wiki.eclipse.org/MDT/UML2

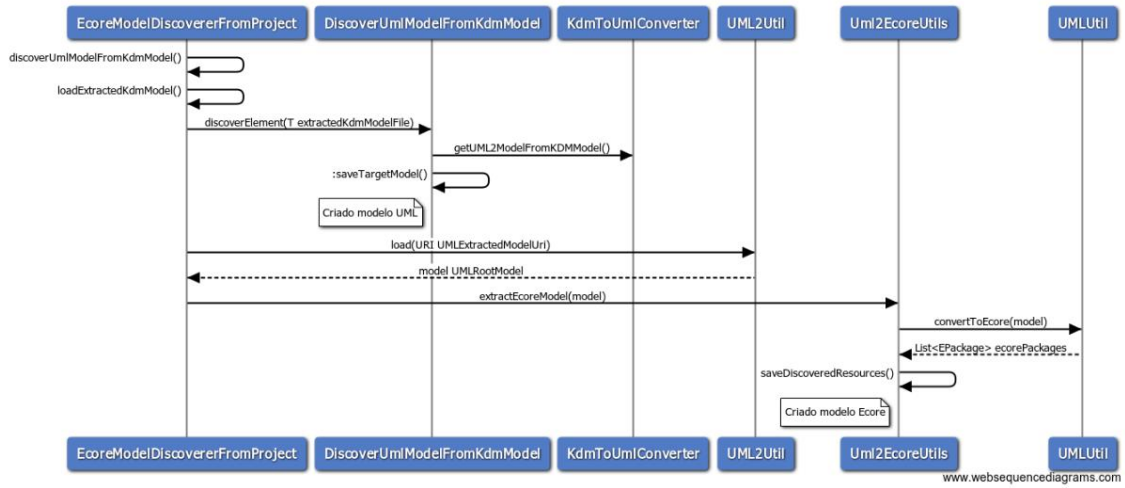


Figura 43 - Diagrama de sequência Descoberta de Modelo Ecore (Parte 3)

8.2.2 Criação *plugin* UI obtenção de modelos Ecore

Para permitir que o novo *discoverer* criado seja apresentado no menu de contexto Modisco foi necessário criar um novo *plugin* Eclipse. Após criar o *plugin* e com base na informação disponibilizada na documentação Modisco é necessário registar o novo *discoverer* utilizando o *extension-point* Modisco *org.eclipse.modisco.infra.discovery.ui.discoverer.* Como se pode visualizar no Código 18 o novo *discoverer* previamente criado foi adicionado para possibilitar a apresentação dos novos componentes visuais.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>
        <extension point="org.eclipse.modisco.infra.discovery.ui.discoverer">
                <discoverer discovererID="org.eclipse.modisco.ecore.ecorediscoverer"
icon="icons/ecore.png" label="Discover Ecore Model from Project"/>
        </extension>
</plugin>
```

Código 18 - Definição dos componentes visuais do novo *discoverer* através do *extension-point*

Após definir o novo *discoverer* na extensão visual da ferramenta Modisco foi possível obter o resultado apresentado na Figura 44, onde podemos visualizar a inclusão da opção “Discover Ecore Model from Project...” tal como definido na extensão apresentada no Código 18.

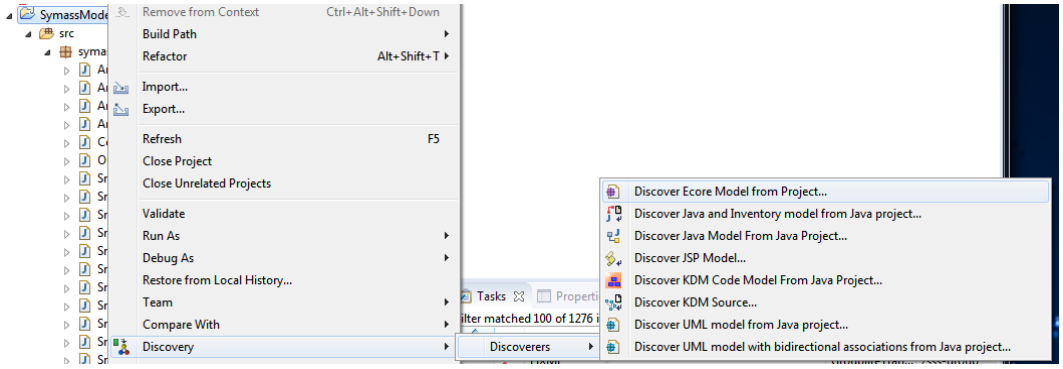


Figura 44 – Menu de contexto Modisco com o novo *discoverer* Ecore

Algumas melhorias visuais foram realizadas ao comportamento base da ferramenta Modisco para permitir apresentar um novo menu de seleção dos elementos a analisar num determinado projeto ou *package* Java. Sem estas alterações não seria possível definir condicionalmente quais as classes a incluir no novo modelo Ecore gerado. Isso iria aumentar a complexidade do modelo desnecessariamente. Desta forma como apresentado na Figura 45 o utilizador tem a capacidade de selecionar as classes Java que deseja incluir na análise.

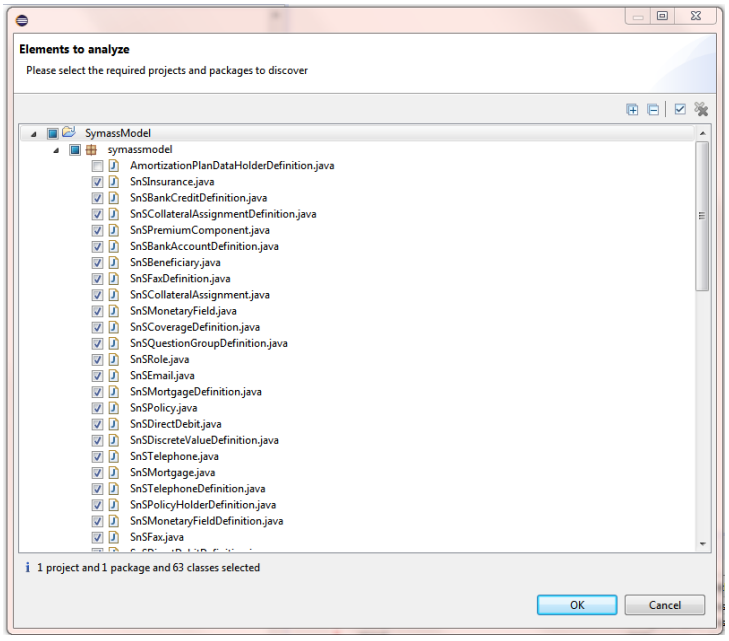


Figura 45 – Janela de seleção de classes a incluir na obtenção do modelo Ecore

Como apresentado na Figura 46, o menu de configuração da operação foi alterado para permitir definir o nome do modelo Ecore e especificar a localização na qual o novo modelo deve ser guardado, utilizando para tal uma janela de seleção da localização tal como apresentado na Figura 47.

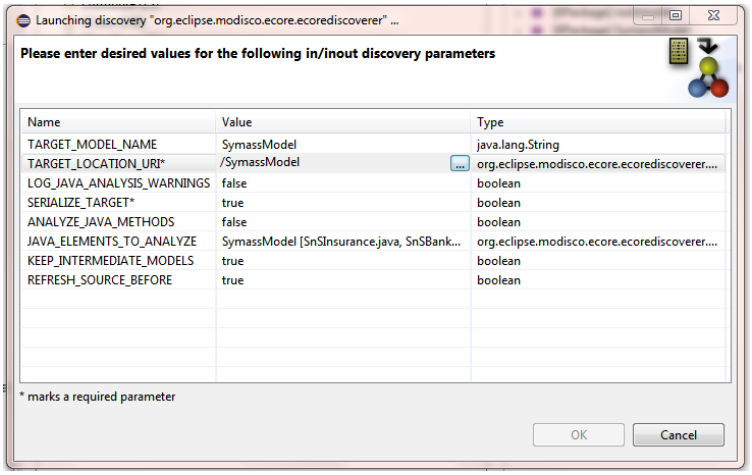


Figura 46 – Menu de configuração da operação de obtenção do modelo Ecore

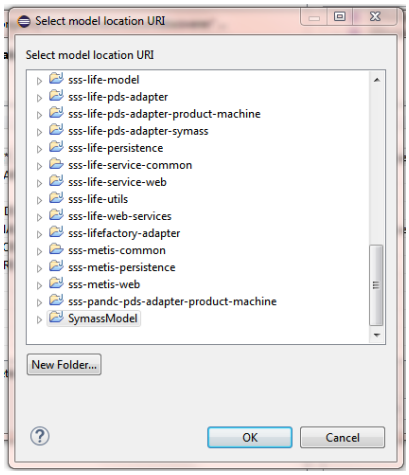


Figura 47 – Janela de seleção da localização para guardar o modelo obtido

Para além dessas alterações foi também adicionada a capacidade de manter os modelos intermédios, gerados durante o processo de transformação realizado, é assim possível manter os modelos Java (formato *.xmi*), KDM (formato *.xmi*) e UML (formato *.uml*).

Para atingir as alterações visuais apresentadas no novo *plugin* criado foram definidas novas classes como se pode observar no diagrama de classes da Figura 48, onde vemos as classes *JavaElementsToAnalyzeDialog* e *JavaElementsToAnalyzeComposite* que alteram e adicionam comportamento aos elementos visuais existentes na ferramenta Modisco.

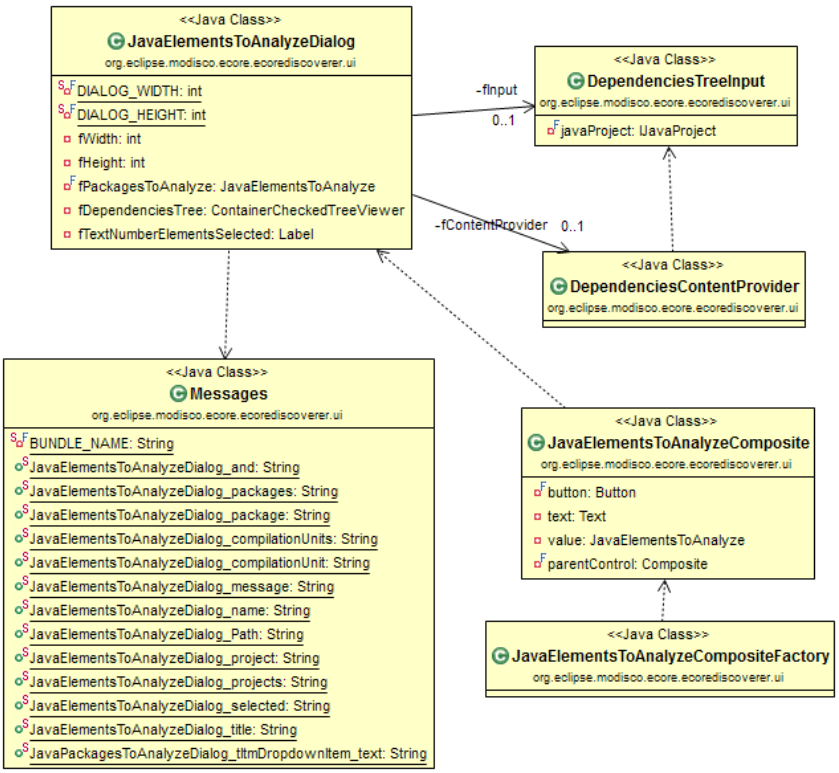


Figura 48 – Diagrama de classes criadas para alterar aspetos visuais da ferramenta Modisco

8.2.3 Obtenção modelo Ecore

Após a criação dos dois *plugins* necessários para adicionar o novo *discoverer* à ferramenta Modisco é possível obter um modelo Ecore representativo de classes Java contidas num determinado projeto Java. Tal como apresentado na Figura 49 onde podemos ver a representação do modelo da aplicação Symass em formato Ecore, contendo os atributos e relações entre entidades identificados durante a operação de análise executada ao código Java.

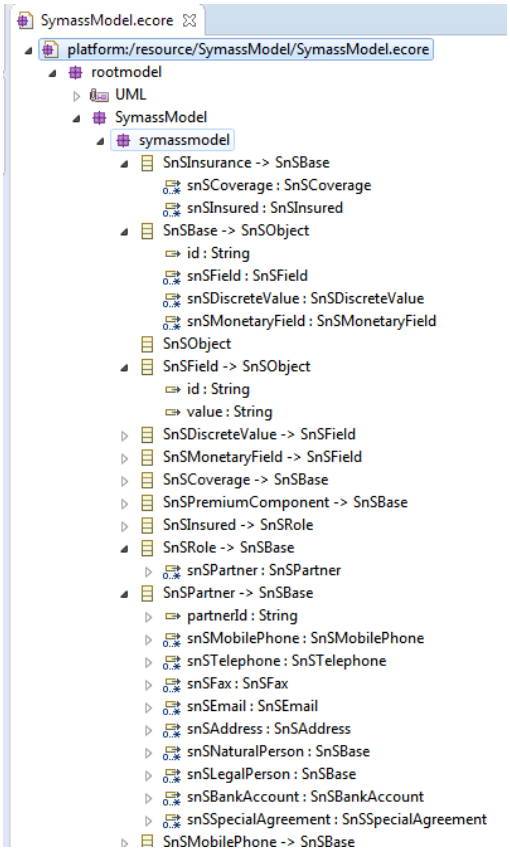


Figura 49 – Excerto do modelo Ecore do Symass

8.3 Implementação Componente Definição Mapeamento

Para realizar a implementação do componente de definição de mapeamento tal como definido anteriormente utilizou-se a ferramenta AMW. Em primeiro lugar para permitir utilizar esta ferramenta foi necessário instalar os *plugins* Eclipse ATL[47] e AMW[48]. A instalação do *plugin* não é um processo de instalação Eclipse normal e envolveu a obtenção do código fonte da ferramenta AMW e a importação de todos os projetos para o Eclipse de forma a permitir obter um ambiente de desenvolvimento Eclipse com acesso a todos os *plugins* necessários.

8.3.1 Mapeamento de modelos utilizando AMW

Após montar um ambiente de desenvolvimento foi possível iniciar uma aplicação Eclipse com acesso às funcionalidades da ferramenta AMW. Para permitir analisar as capacidades de mapeamento foram criados dois modelos Ecore simples como apresentado na Figura 50, apesar de estes dois modelos serem extremamente minimalistas encontram-se alinhados com os casos

[47] Informação sobre os passos necessários para instalar o plugin Eclipse ATL disponível em: http://www.eclipse.org/atl/download/

[48] A instalação do *plugin* AMW não é um processo de instalação padrão Eclipse. Mais informação disponível em: http://www.inf.ufpr.br/didonet/amw/userguide/web/gs/install.html

práticos reais em termos de tipos de mapeamentos passíveis de serem realizados. Temos do lado esquerdo da figura o modelo Ecore *salesAndServiceSampleModel.ecore* para exemplificar o modelo S&S e do lado direito da figura o modelo Ecore *productMachineSampleModel.ecore* para exemplificar o modelo da PM.

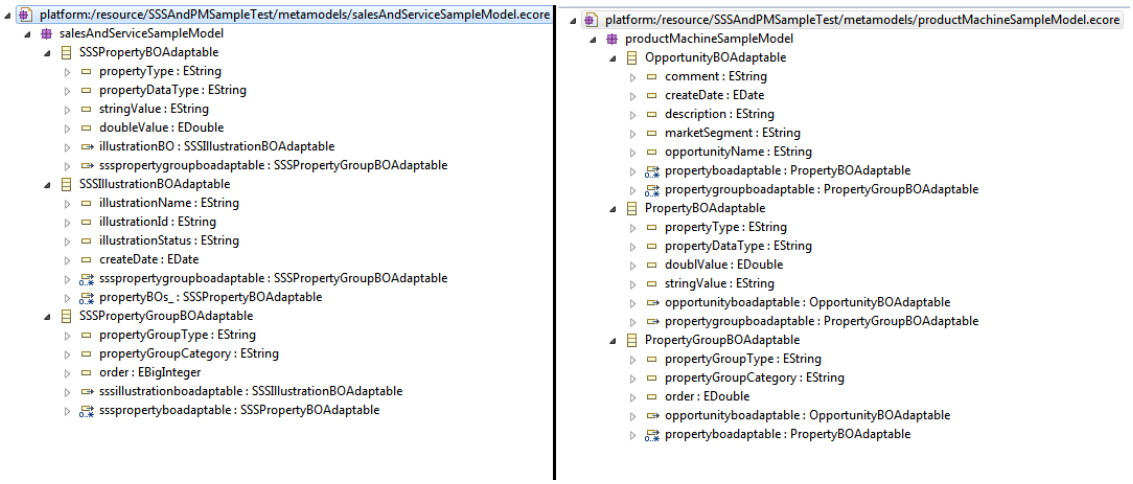


Figura 50 – Modelos Ecore S&S e PM simplificados

Para executar o caso prático de mapeamento com AMW foi criado um novo projeto *SSSAndPMSampleTest1* com a estrutura apresentada na Figura 51, como podemos ver este projeto contém uma pasta *models* onde se colocaram os modelos a utilizar e uma pasta *amw_matches* que será utilizada para armazenar os modelos de relação criados com a ferramenta AMW.

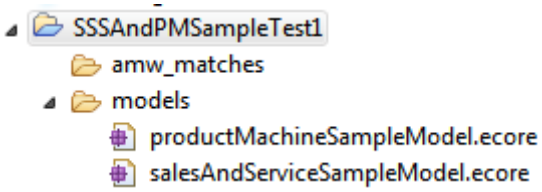


Figura 51 – Estrutura do projeto *SSSAndPMSampleTest1*

De seguida criou-se um novo ficheiro *Weaving Model*[49] utilizando a opção apresentada na Figura 52.

[49] Informação adicional sobre o processo de adição de um novo modelo weaving disponível em: http://www.inf.ufpr.br/didonet/amw/userguide/index.html

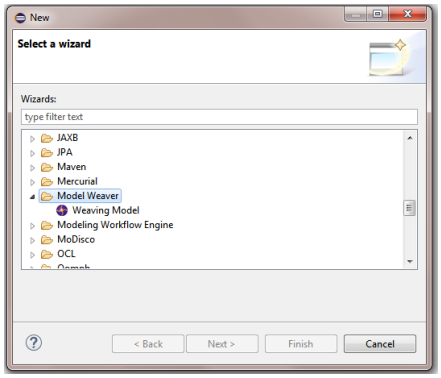


Figura 52 – Adição de novo *Weaving Model* utilizando a ferramenta AMW

De seguida é iniciado o *wizard* de adição de um novo modelo *weaving* da ferramenta AMW, este *wizard* contém três passos, no primeiro é necessário definir a opção "Load KM3 extensions" e selecionar os metamodelos de mapeamento necessários para o cenário pretendido, no nosso caso devem ser selecionados os metamodelos apresentados na Figura 53, estes metamodelos são necessários para permitir a definição das relações entre elementos.

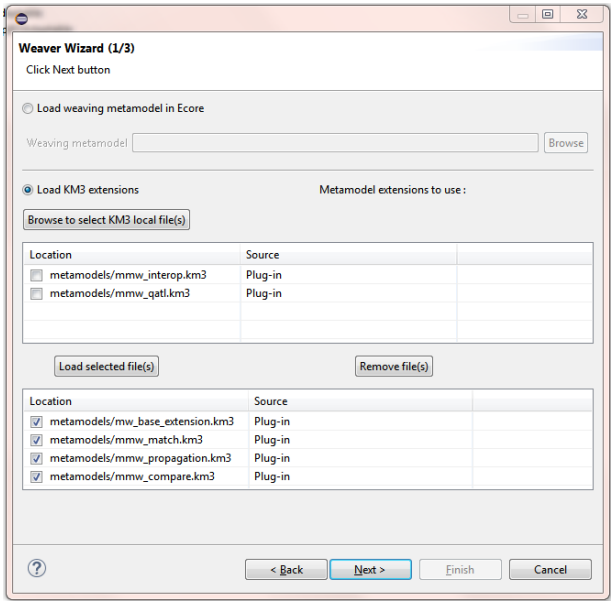


Figura 53 – Configuração de um novo modelo *weaving* passo 1 do wizard

No segundo passo do *wizard* como apresentado na Figura 54 é necessário definir o nome do ficheiro para o modelo com a extensão *.amw* e é também necessário escolher o tipo de painel de mapeamento que pretendemos ter disponível para executar o mapeamento, neste caso o tipo de painel a escolher é a *TransformationWeavingPanelExtension*. Neste passo devemos ainda escolher o tipo de *weaving model* como sendo *MatchModel.*

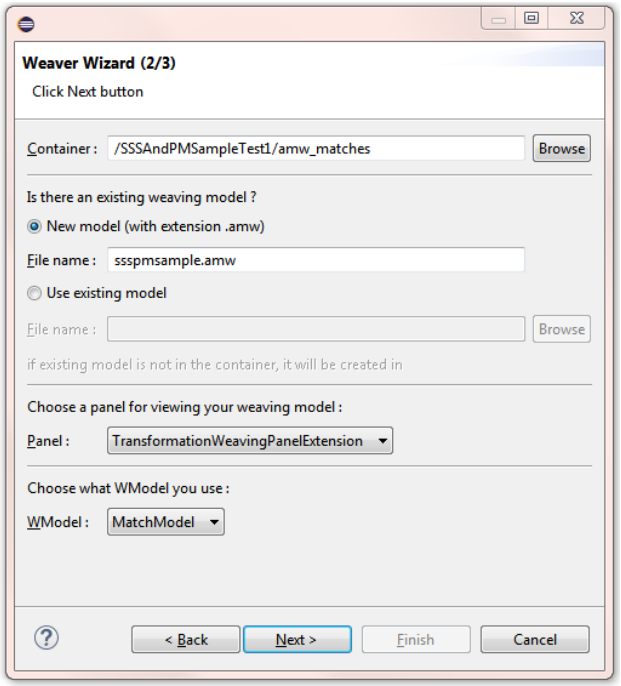


Figura 54 – Configuração de um novo modelo *weaving* passo 2 do wizard

No último passo do *wizard* tal como apresentado na Figura 55 vamos definir os modelos que pretendemos relacionar[50], neste caso colocamos o modelo do S&S como sendo o modelo esquerdo e o modelo da PM como sendo o modelo direito.

[50] Para definir cada um dos modelos esquerdo e direito é necessário realizar sequencialmente os passos apresentados nos Anexos na Figura 67 e Figura 68.

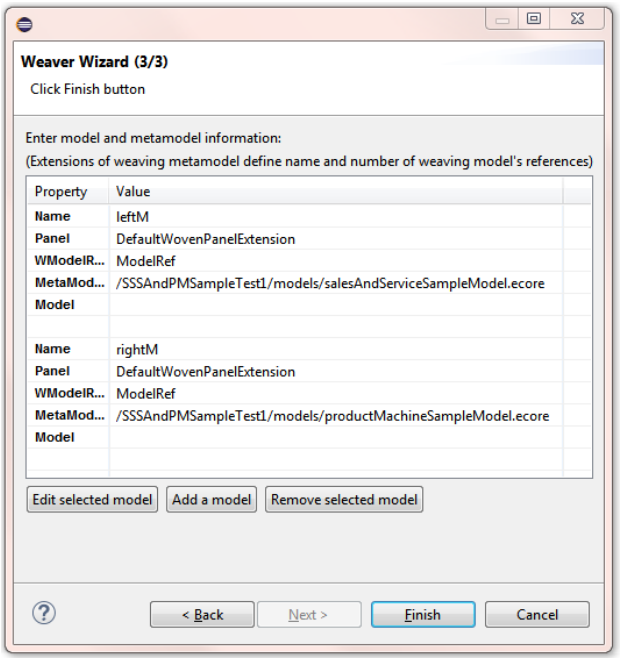


Figura 55 – Configuração de um novo modelo *weaving* passo 3 do wizard

Após concluir este último passo o novo modelo *weaving* é criado e o painel de mapeamento da ferramenta AMW é apresentado (Figura 56). Este painel permite realizar as tarefas de mapeamento de relações manualmente e a execução de heurísticas de mapeamento para proceder a criação de relações de forma automática entre os elementos dos modelos esquerdo e direito.

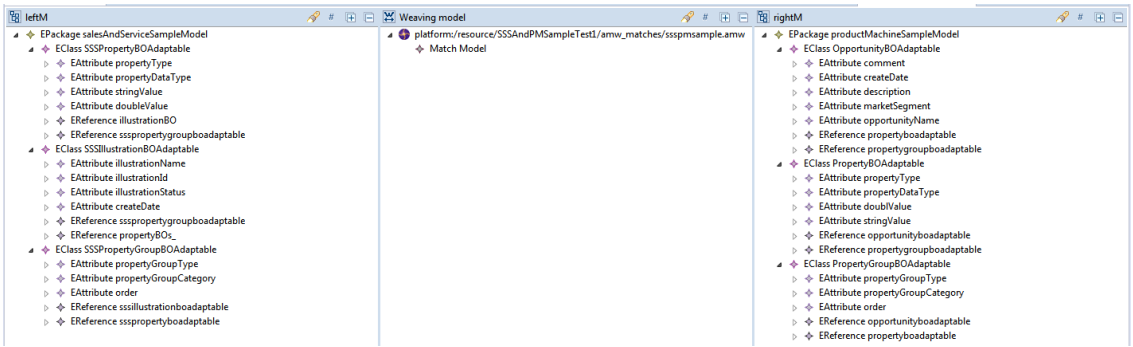


Figura 56 – Painel de mapeamento para os modelos S&S e PM simplificados

Sobre o modelo *weaving* criado executamos três tipos de heurísticas de mapeamento[51]:

- *Link generation* - criação de relações entre elementos. Esta transformação deve ser sempre a primeira a ser executada:
    - *Cartesian product* – permite a criação automática de relações entre todos os pares de elementos entre o modelo esquerdo e o modelo direito, este

[51] Mais informação sobre as transformações de mapeamento da ferramenta AMW disponível em: http://www.inf.ufpr.br/didonet/amw/userguide/web/tasks/matching_exec.html

algoritmo apenas cria relações para elementos do mesmo tipo, como por exemplo: EClass – EClass ou EAttribute-EAttribute;

- *Similarity assignment* – aplicação de algoritmos para determinar um valor de similaridade entre os elementos do modelo. Esta transformação só deve ser executada após a execução da *link generation:*
    - *Name similarity* – aplica métodos de comparação similaridade entre o nome dos elementos criados durante a execução do produto cartesiano;
- *Link selection and rewriting* – interpreta os valores de similaridade atribuídos previamente selecionando e reorganizando as relações existentes. Esta transformação deve ser executada no fim das *similarity assignment:*
    - *Link rewriting* – para cada elemento do modelo da esquerda seleciona a relação com maior valor similaridade e reorganiza as relações tendo em conta as relações de *containment* entre os elementos, ou seja, relações entre entidades e os seus atributos.

Como resultado de cada transformação executada é criado um novo modelo *weaving* que contém as novas relações criadas durante a execução da transformação. No caso prático que temos vindo a realizar após a execução das transformações especificadas é obtido o modelo de relação apresentado na Figura 57, como se pode observar foram criadas várias relações entre elementos do modelo esquerdo e do modelo direito. O modelo de relação resultante pode ser validado manualmente e qualquer relação errada pode ser removida ou em falta pode ser adicionada para garantir o correto mapeamento dos elementos entre os modelos.

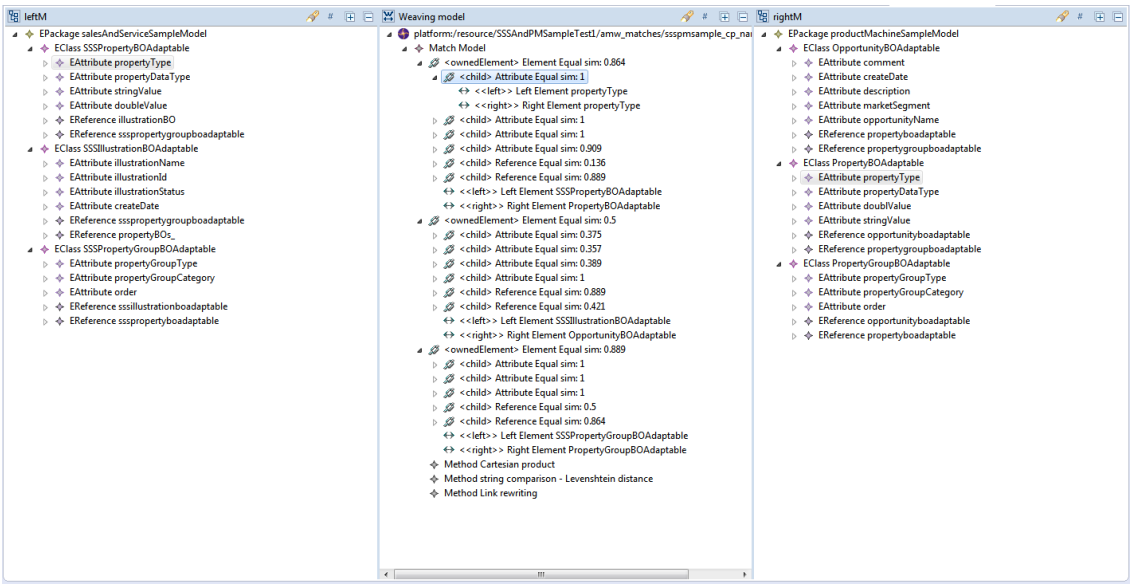


Figura 57 – Modelo de mapeamento para os modelos S&S e PM após transformações automáticas

8.3.2 Criação metamodelo de transformação

O resultado final do componente de definição de mapeamento deve ser um modelo de transformação e o código de transformação MapStruct correspondente. Para suportar esse

resultado final é necessário definir um metamodelo que permita guardar informação necessária sobre cada elemento e a sua relação. Como tal um novo modelo Ecore foi definido de acordo com o *design* apresentado no capítulo 7.3.2. O Código 19 apresenta a representação textual do modelo utilizando a ferramenta Emfatic.

```
@namespace(uri="http://www.example.org/mapstructmodel", prefix="mapstructmodel")
package mapstructmodel;

class MapperRepository {
   val Mapper[*] mappers;
}

class Mapper {
        attr String mapperName;
        attr String sourceClassName;
        attr String targetClassName;
        attr Boolean generateCollectionTransformation = true;
        val Mapping[*] mappings;
        ref Mapper[*] referencedMappers;
}

class Mapping {
        attr String source;
        attr String target;
        attr String expression;
}
```

Código 19 – Representação textual Emfatic do metamodelo de transformação

O modelo EMF correspondente foi criado utilizando a funcionalidade "Generate Ecore model" Emfatic. Para permitir que o novo metamodelo possa ser utilizado no contexto de um ambiente Eclipse é necessário criar os *plugins* EMF corretos (Vogel, 2016). Desta forma tendo como ponto de partida o modelo Ecore foi criado um novo projeto EMF e de seguida utilizando as funcionalidades da ferramenta EMF foram criados dois dos três *plugins* apresentados na Figura 58, nomeadamente o *plugin MapStructModel.edit* e *MapStructModel.editor.* Estes *plugins* tornam possível trabalhar com o metamodelo num ambiente Eclipse.

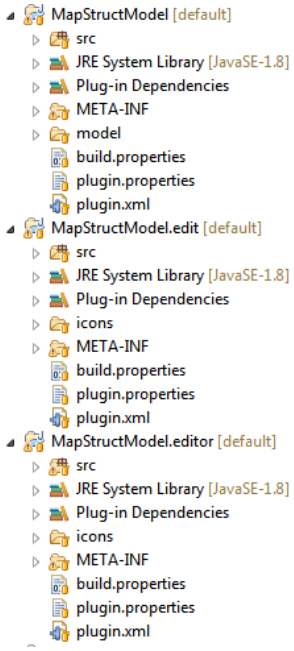


Figura 58 – *Plugins* EMF para o metamodelo de transformação

8.3.3 Transformação ATL para obtenção modelo de transformação

Depois de definido o modelo de relação AMW utilizando as funcionalidades manuais ou automáticas da ferramenta AMW surge a necessidade de executar uma transformação ATL sobre esse modelo para extrair um modelo de transformação MapStructModel. Para atingir esse objetivo foi necessário adicionar uma nova opção ao menu de contexto nativo da ferramenta AMW.

Para permitir adicionar novos tipos de transformações sobre o modelo *weaving* o *plugin org.eclipse.gmt.weaver.transformation* disponibiliza um *extension-point* designado por *org.eclipse.gmt.weaver.transformation.transformationID*. Sobre este *extension-point* foi desenvolvido um novo tipo de extensão designado por *transformationmodel* para permitir definir uma nova categoria de transformações. Para permitir a adição deste novo tipo de transformações foi necessário alterar o comportamento nativo AMW de gestão de transformações disponíveis [52] . Na Figura 59 podemos observar as classes alteradas e adicionadas para permitir modificar o comportamento nativo de gestão de transformações do *plugin* AMW. A classe *TransformationExtensionManager* previamente existente e utilizada na gestão dos tipos de transformação existentes foi alterada para permitir a leitura das transformações de acordo com as alterações feitas à classe *TransformationConfig*. Esta classe foi definida como sendo a classe base de configuração de transformações e três novas

[52] A ferramenta AMW permite definir novas transformações para melhorar o processo de mapeamento existente: https://www.eclipse.org/gmt/amw/usecases/matching/matching_develop.php

subclasses foram adicionadas para permitir especificar a nova divisão entre tipos de transformação:

- *MatchTransformationConfig;*
- *HotTransformationConfig;*
- *TransformationModelTransformationConfig*.

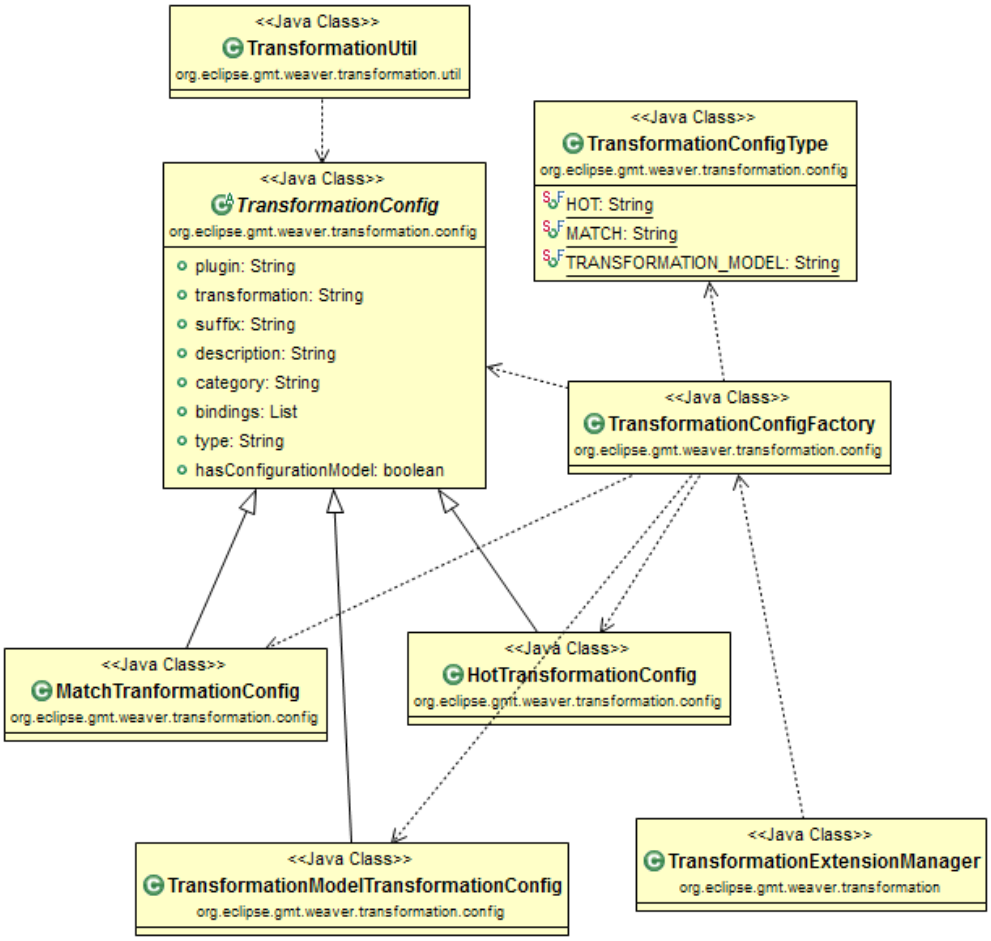


Figura 59 – Diagrama de classes alteradas/adicionadas para suportar novos tipos de transformações

Como se pode observar no Código 20 temos a definição da nova transformação *AMWtoEcore_MapsStruct,* esta extensão foi adicionada ao *plugin* AMW *org.eclipse.gmt.weaver.transformation.extension* utilizando o novo tipo de transformação *transformationmodel* previamente definido. De salientar que na definição desta extensão é especificada a transformação ATL a ser executada bem como a informação sobre os modelos de entrada para a transformação, neste caso o *leftM* e o *rightM* são modelos que obedecem ao metamodelo MOF[53]. Para além dessa disso também são definidas a categoria de transformação como sendo "AMW to Ecore" e a descrição "Matched Weaving to MapStruct Ecore" que será utilizada como o texto a apresentar no menu de contexto.

[53] Mais informação disponível em: http://www.omg.org/mof/

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
         <extension
         id="WeaverTransformations"
         point="org.eclipse.gmt.weaver.transformation.transformationID">

               <transformationmodel transformation="transformations/AMWtoEcore_MapStruct.asm"
description="Matched Weaving to MapStruct Ecore" file_suffix="transformationmodel" category="AMW to
Ecore">
                        <binding weavingReference="leftM" header="left" metamodelHeader="MOF"/>
                        <binding weavingReference="rightM" header="right" metamodelHeader="MOF"/>
               </transformationmodel>

        </extension>
</plugin>
```

Código 20 – Definição da extensão da nova transformação AMWtoEcore_MapStruct

A apresentação da nova opção de menu foi conseguida alterando o método *addTransformationExtensions* da classe *TransformationWeavingPanel*, esta classe já existente na ferramenta AMW é responsável por adicionar componentes visuais ao painel de mapeamento de modelos. No método *addTransformationExtensions* como se apresenta no Código 21 foi adicionado um novo ciclo para iterar todos os tipos de transformações do tipo *transformationmodel* e adicionar novas entradas ao *menuManager*.

```
1.  **
2.  * adds the actions of matching transformations and HOT transformations
3.  * obtained through the "transformation" extension point
4.  * @param menuManager
5.  * /
6.  private void addTransformationExtensions(IMenuManager menuManager) {
7.      menuManagers.clear();
8.      for (Iterator tcs = extensionManager.getHotExtensions().iterator(); tcs.hasNext();) {
9.          TransformationConfig tc = (TransformationConfig) tcs.next();
10.         if (!tc.hasConfigurationModel) {
11.             IMenuManager mngr = (IMenuManager) menuManagers.get(tc.category);
12.             if (mngr == null) {
13.                 mngr = new MenuManager(tc.category);
14.                 menuManager.add(mngr);
15.                 menuManagers.put(tc.category, mngr);
16.             }
17.             createAction(tc, mngr);
18.         }
19.     } // add transformation tasks
20.     for (Iterator tcs = extensionManager.getTransformationModelExtensions().iterator(); tcs.hasN
    ext();) {
21.         TransformationConfig tc = (TransformationConfig) tcs.next();
22.         if (!tc.hasConfigurationModel) {
23.             IMenuManager mngr = (IMenuManager) menuManagers.get(tc.category);
24.             if (mngr == null) {
25.                 mngr = new MenuManager(tc.category);
26.                 menuManager.add(mngr);
27.                 menuManagers.put(tc.category, mngr);
28.             }
29.             createAction(tc, mngr);
30.         }
31.     }
32. }
```

Código 21 - Excerto de código da classe *TransformationWeavingPanel* responsável pela apresentação da nova opção de menu

A ação que dever ser executada quando a opção for selecionada é também definida como sendo do tipo *TAction*, como é possível observar no Código 22 trata-se de um tipo de ação disponível na ferramenta AMW que permite executar uma operação de transformação sobre o modelo *weaving* criado.

```
1.  /**
2.   * Action that executes a transformation in the weaving panel the
3.  * transformations may have two different signatures: create OUT : ATL from
4.  * IN : AMW, left: MOF, right: MOF; for the higher-order transformations
5.  * create OUT : AMW from IN: AMW, left : MOF, right : MOF; for the matching  transformations
6.  * @author Marcos Didonet Del Fabro
7.  */
8.  class TAction extends Action {
9.      private TransformationConfig transformationConfig = null;
10.     public TAction(TransformationConfig tc) {
11.         super(tc.description);
12.         this.transformationConfig = tc;
13.     }
14.     public void run() {
15.         try {
16.             GlobalWeaverEditor editor = globalEditor;
17.             ILauncher launcher = TransformationUtil.setModelsMap(null, editor.getModelManager(),
     transformationConfig, false, null, null);
18.             IModel out = TransformationUtil.executeTransfo(launcher, editor.getModelManager(), g
    lobalEditor.getXmlFile(), transformationConfig); // in the launch configuration, we assume the models conform
    to // MOF and AMW
19.             transformationConfig.saveNewModel(out, editor.getModelManager(), editor.getXmlFile()
    , ""); // refreshes the resource and shows confirmation message
20.             WeaverResourceUtil.refreshResourceFolder(editor.getModelManager().getWeavingModel())
    ;
21.             MessageDialog.openInformation(editor.getEditorSite().getShell(), transformationConfi
    g.description, TransformationActivator.getDefault().getString("_UI_model_created") + ((EMFModel)
     out).getResource().getURI().toString());
22.         } catch (Exception e) {
23.             e.printStackTrace();
24.         }
25.     }
26. }
```

Código 22 – Código fonte da classe TAction que permite definir uma ação de transformação

Finalizadas as alterações descritas ao clicar sobre um modelo *weaving* utilizando o painel AMW é agora possível observar a apresentação da nova opção de transformação tal como se pode visualizar na Figura 60.

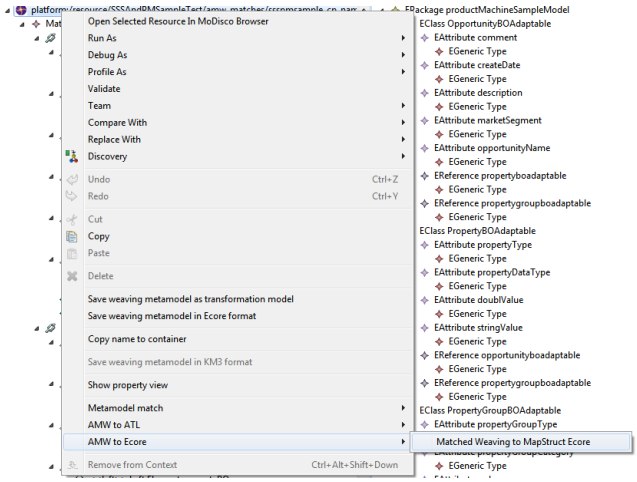


Figura 60 – Nova opção de transformação apresentada no menu de contexto AMW

Após realizadas as alterações expostas para permitir a adição do novo tipo de transformação em termos visuais foi necessário criar a transformação ATL[54] (Jouault, 2013) propriamente dita que realize a conversão de um modelo no *weaving* num modelo no formato *MapSctructModel.* Para realizar essa tarefa foi criado um ficheiro *AMWtoEcore_MapStruct.atl*[55] que define uma transformação ATL. Esta transformação ATL como apresentado no Código 23 recebe como *input* um modelo AMW e os modelos mapeados *left* e *right* e no final cria um novo modelo MAPSTRUCT.

```
-- @path AMW=/TestingATL/mwcore.ecore
-- @path MAPSTRUCT=/TestingATL/mapstructmodel.ecore

module AMWtoEcoreMapStruct;
create OUT: MAPSTRUCT from IN: AMW, left: MOF, right: MOF;
```

Código 23 – Excerto do cabeçalho da transformação ATL AMWtoEcoreMapStruct

Como se pode observar no Código 24 temos um excerto do código de transformação ATL, é possível destacar que a regra *ElementEqual* é executada para todas as relações do modelo AMW do tipo *ElementEqual* de forma a permitir criar uma nova instância *Mapper* no modelo a ser construído. Na execução desta regra os atributos *sourceClassName*, *targetClassName* e *mapperName* são preenchidos com a informação obtida de cada um dos modelos *left* e *right.* A relação de referência de transformações *referencedMappers* também é pré-preenchida com a informação necessária para permitir a execução de uma ação final de preenchimento completo desta referência que deve conter todas as relações com outros mapeamentos existentes. A regra *AttributeEqual* é executada sobre todos os elementos do tipo *AttributeEqual* do modelo AMW recebido e para cada um é criada uma nova entidade *Mapping* no novo modelo de transformação a ser construído. Informação sobre o atributo de origem e de destino é obtida dos modelos *left* e *right*.

[54] Transformações ATL: http://www.eclipse.org/atl/documentation/basicExamples_Patterns/
[55] Conteúdo total da transformação disponível nos Anexos no Código 38.

```
rule ElementEqual {
	from
		amw: AMW!ElementEqual
	using {
		childDebug: AMW!WLink = OclUndefined;
		leftModelRef: MOF!EClassifier = OclUndefined;
		leftModelRefId: String = OclUndefined;
		refers_link: AMW!WLink = OclUndefined;
	}
	to
		mapstructmapper: MAPSTRUCT!Mapper (
			sourceClassName <- amw.left.name,
			targetClassName <- amw.right.name,
			mapperName <- amw.right.name + 'Mapper',
			mappings <- amw.child,
			referencedMappers <- amw.child -> select(e | e.
					oclIsTypeOf(AMW!ReferenceEqual)) -> collect(childReference |
					thisModule.resolveTemp(childReference.getReferencedLink(amw),
					'mapstructmapper'))
		)
	do {
		thisModule.mappersByRelationClasses <- thisModule.mappersByRelationClasses.
				union(Map{(Tuple{leftClassID = amw.left.element.ref, rightClassID = amw.
				right.element.ref},
						mapstructmapper)});
	}
}

rule AttributeEqual {
	from
		amw: AMW!AttributeEqual
	to
		mapstructmapping: MAPSTRUCT!Mapping (
			source <- amw.left.name,
			target <- amw.right.name
		)
}
```

Código 24 – Excerto código da transformação ATL apresentando a regra ElementEqual e AttributeEqual

Existe um aspeto importante da transformação a destacar, durante a execução das regras ATL definidas não é possível completar a definição do atributo *referencedMappers* no momento em que um novo elemento *Mapper* é criado, isto é compreensível visto nesse momento nem terem sido ainda processadas todas as relações existentes entre os modelos *left* e *right*. Para permitir a correta definição dessa informação é definida uma regra a executar apenas no final do processamento de todas as regras de transformação. Como se pode observar no Código 25 esta ação final de transformação irá percorrer todas as instâncias *Mapper* criadas e para cada uma irá obter a informação de todas as instâncias *Mapper* referenciadas na estrutura do novo modelo criado.

```
--execute delayed actions
endpoint rule EndRule() {
	using {
		leftModelReferenceParentId: String = OclUndefined;
		rightModelReferenceParentId: String = OclUndefined;
		currentReferenceKey: TupleType(leftClassID: String, rightClassID: String) =
				OclUndefined;
		currentReferencedMapper: MAPSTRUCT!Mapper = OclUndefined;
		elementMapper: MAPSTRUCT!Mapper = OclUndefined;
	}
	do {
		for (e in MAPSTRUCT!Mapper.allInstances()) {
			elementMapper <- OclUndefined;
		}
		for(elementEqual in AMW!ElementEqual.allInstances()) {
			elementMapper <- OclUndefined;
			for(referenceEqual in elementEqual.child -> select(e | e.
					oclIsTypeOf(AMW!ReferenceEqual))) {
				currentReferencedMapper <- OclUndefined;
				leftModelReferenceParentId <- thisModule.
					getRelationLeftParentRefId(referenceEqual);
				rightModelReferenceParentId <- thisModule.
					getRelationRightParentRefId(referenceEqual);
				currentReferenceKey <- Tuple{leftClassID = leftModelReferenceParentId,
					rightClassID = rightModelReferenceParentId};
				currentReferencedMapper <- thisModule.mappersByRelationClasses.
					get(currentReferenceKey);
				if (not currentReferencedMapper.oclIsUndefined()) {
					elementMapper <- thisModule.mappers.get(elementEqual);
					if (not elementMapper.oclIsUndefined()) {

	elementMapper.referencedMappers.append(currentReferencedMapper);
					}
				}
			}
		}
	}
}
```

Código 25 – Excerto de código da transformação ATL apresentando a regra *EndRule*

No final da execução do módulo ATL é obtido um novo modelo com informação sobre os mapeamentos no formato MapStructModel. Como se apresenta na Figura 61 e de acordo com o exemplo de mapeamento entre o modelo simplificado do S&S e da PM apresentado no capítulo 8.3.1 foi obtido o modelo de transformação correspondente aos mapeamentos previamente definidos entre os elementos dos modelos.

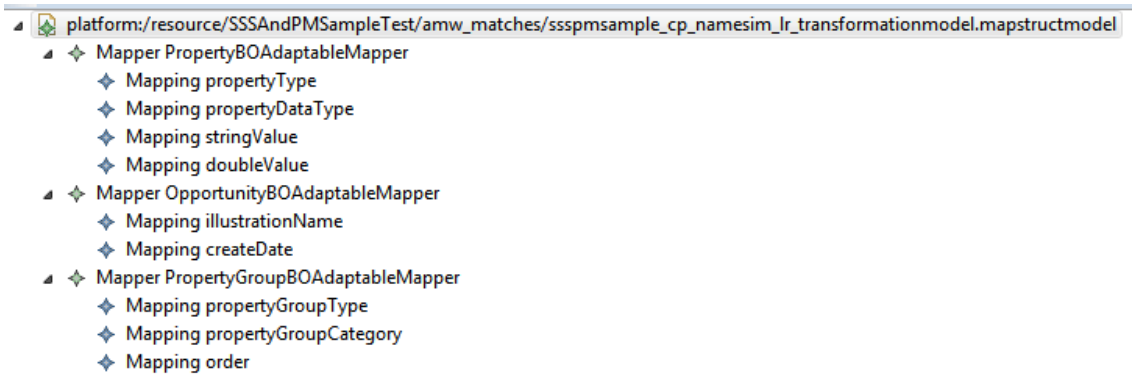


Figura 61 – Modelo de transformação MapStruct obtido através de AMWtoEcore_MapStruct

8.3.4 Geração de código de transformação MapStruct

Como vimos no capítulo anterior foi possível obter um modelo MapStructModel com as transformações necessárias entre os modelos mapeados. Surge agora a necessidade de permitir obter o código de mapeamento MapStruct que permita concretizar as transformações definidas anteriormente. Para permitir atingir esse objetivo como apresentado na Figura 62 foram criados dois novos *plugins* Eclipse Acceleo:

- *MapStructModel2Text* – contém o *template* Acceleo responsável por converter a informação de transformação contida no modelo MapStructModel para código Java de acordo com a ferramenta MapStruct;
- *MapStructModel2TextUI* – permite adicionar os componentes visuais necessários para a execução da geração de código para um modelo MapStructModel através da utilização do menu de contexto Acceleo.

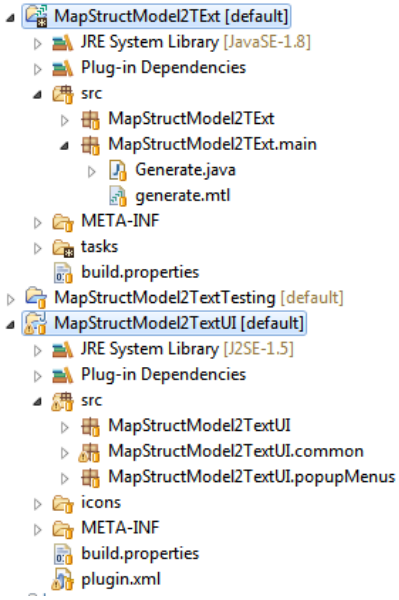


Figura 62 – Novos *plugins* Eclipse Acceleo para gerar código de mapeamento MapStruct

No *plugin MapStructModel2Text* foi criado então um *template* Acceleo para gerar o código Java representativo do mapeamento modelado para um formato em código reconhecido pela ferramenta MapStruct. Como se pode observar no Código 26 com este *template* será criada uma interface Java para cada entidade a mapear entre os modelos. Esta interface define através das anotações da ferramenta MapStruct tipos de mapeamento diretos entre atributos e adicionalmente é declarada a utilização de outras interfaces de mapeamento para permitir identificar que outras entidades devem ser transformadas no contexto da entidade atual.

```
[comment encoding = UTF-8 /]
[module generate('http://www.example.org/mapstructmodel')]
[template public generateElement(aMapper : Mapper)]
[comment @main/]
[file (aMapper.mapperName.concat('.java'), false, 'UTF-8')]
package com.ins.mapstruct;
import java.util.List;
import org.mapstruct.Mapper;
import org.mapstruct.Mapping;
import org.mapstruct.Mappings;
// [protected ('imports')]
// [/protected]
/**
* THIS CLASS IS GENERATED, CHANGE CODE ONLY IN THE INDICATED PLACES
*/
@Mapper(uses = {
[for (aReferencedMapper : Mapper | aMapper.referencedMappers)]
        [aReferencedMapper.mapperName.concat('.class')/],
[/for]
        //[protected ('Invoking other mappers')]
                // Replace this line by any additional mappers to be used.
        //[/protected]
})
public interface [aMapper.mapperName.toUpperFirst()/] {
        [let sourceClass : String = aMapper.sourceClassName.toUpperFirst()]
        [let targetClass : String = aMapper.targetClassName.toUpperFirst()]
        @Mappings({
        [for (aMapping : Mapping | aMapper.mappings)]
                @Mapping([if (not(aMapping.source.oclIsUndefined()) and aMapping.source.trim() <>
'')]source = "[aMapping.source/]",[/if] [if (not(aMapping.target.oclIsUndefined()) and
aMapping.target.trim() <> '')]target = "[aMapping.target/]",[/if] [if
(not(aMapping.expression.oclIsUndefined()) and aMapping.expression.trim() <> '')]expression =
"[aMapping.expression/]",[/if]),
        [/for]
                //[protected ('Additional mappings')]
                        // Replace this line by any additional mappings.
                //[/protected]
        })
        [targetClass/] to[targetClass/]([sourceClass/] [sourceClass.toLowerFirst()/]);
        [if (aMapper.generateCollectionTransformation)]
        /**
        * @generated
        */
        List<[targetClass/]> to[targetClass/]s(List<[sourceClass/]> [sourceClass.toLowerFirst()/]List);
        [/if]
        [/let]
        [/let]
}
[/file]
[/template]
```

Código 26 – *Template* Acceleo de geração de código das interfaces de mapeamento MapStruct

Utilizando o exemplo de mapeamento entre o modelo simplificado do S&S e da PM apresentado no capítulo 8.3.1 e o modelo de transformação previamente obtido é possível utilizar o menu de contexto de geração de código disponibilizado pelo novo *plugin* Eclipse *MapStructModel2TextUI* como apresentado na Figura 63, ao selecionar esta opção o novo *template* Acceleo é executado e o código correspondente é gerado para uma nova pasta *src-gen.*

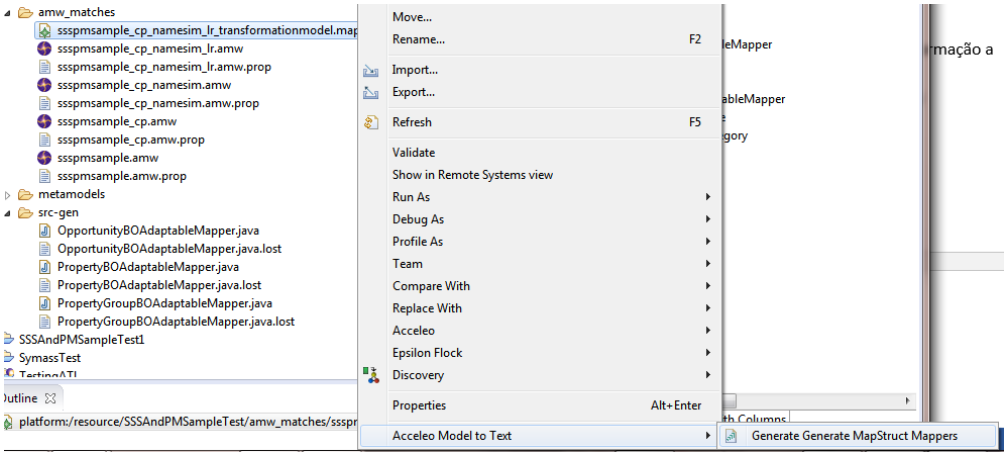


Figura 63 – Menu de contexto de geração de código das interfaces MapStruct

O código gerado após esta ação apresenta os mapeamentos entre os elementos do modelo, tomando como exemplo o mapeamento entre a entidade *SSSIllustrationBOAdaptable* e *OpportunityBOAdaptable* e tal como se observa no Código 27 onde se apresenta o código gerado para a interface de mapeamento MapStruct. Neste código podemos observar que as transformações entre os atributos *illustrationName* - *opportunityName* e *createDate* - *createDate* se encontram corretamente definidas. Para além disso as interfaces de mapeamento *PropertyGroupBOAdaptableMapper* e *PropertyBOAdaptableMapper* foram também corretamente definidas, visto que uma entidade *OpportunityBO* contém as entidades *PropertyBO* e *PropertyGroupBO.*

```
1.  package com.ins.mapstruct;
2.  import java.util.List;
3.  import org.mapstruct.Mapper;
4.  import org.mapstruct.Mapping;
5.  import org.mapstruct.Mappings; // Start of user code imports // End of user code
6.  /*** THIS CLASS IS GENERATED, CHANGE CODE ONLY IN THE INDICATED PLACES*/
7.  @
8.  Mapper(uses = {
9.      PropertyGroupBOAdaptableMapper.class, PropertyBOAdaptableMapper.class,
10. //Start of user code Invoking other mappers
11. // Replace this line by any additional mappers to be used.
12. //End of user code
13. })
14. public interface OpportunityBOAdaptableMapper {@
15.     Mappings({@
16.         Mapping(source = "illustrationName", target = "opportunityName", ), @Mapping(source = "c
    reateDate", target = "createDate", ),
17. //Start of user code Additional mappings
18. // Replace this line by any additional mappings.
19. //End of user code
20.     })
21. OpportunityBOAdaptable toOpportunityBOAdaptable(SSSIllustrationBOAdaptable sSSIllustrationBOAdap
    table);
22.
23. /**
24. * @generated
25. */
26.     List<OpportunityBOAdaptable> toOpportunityBOAdaptables(List<SSSIllustrationBOAdaptable> sSSI
    llustrationBOAdaptableList);
27. }
```

Código 27 – Código gerado da interface OpportunityBOAdaptableMapper

O resultado final da operação de definição de mapeamento entre modelos é a criação das interfaces de mapeamento de acordo com a ferramenta MapStruct.

8.3.5 Alterações à ferramenta AMW

Para além das alterações realizadas ao comportamento nativo da ferramenta AMW previamente apresentadas foram realizadas as seguintes correções e alterações:

- Foi corrigido um erro existente no mecanismo de pesquisa de elementos disponível no painel de mapeamento, basicamente a pesquisa de elementos nos modelos a mapear não funcionava e nenhum resultado era apresentado ao utilizador. O aspeto visual da funcionalidade também foi alterado;
- O aspeto visual das ações de expandir e colapsar os elementos do modelo também foi alterado.

As alterações apresentadas não apresentam um grande valor na utilização da ferramenta mas a sua correção ou alteração foram utilizados como processo de aprendizagem da arquitetura e do código da ferramenta AMW.

8.4 Implementação Componente Execução Mapeamento

O componente de execução de mapeamento tem por objetivo permitir a obtenção do código Java de transformação correspondente aos mapeamentos entre entidades previamente realizado. Com esse objetivo em mente e após analisar a forma de utilização e definição de transformações na ferramenta MapStruct (Morling, 2015) iniciou-se a implementação do componente de execução de mapeamento.

8.4.1 Utilização ferramenta MapStruct

Para possibilitar a utilização da ferramenta MapStruct foi necessário criar um novo projeto Eclipse Maven (Vogel, 2016). Nesse novo projeto Maven foi adicionada a dependência para a MapStruct como se apresenta no Código 28.

```
<dependencies>
        <dependency>
                <groupId>org.mapstruct</groupId>
                <artifactId>mapstruct</artifactId>
                <version>${org.mapstruct.version}</version>
        </dependency>
</dependencies>
```

Código 28 – Dependência Maven para a ferramenta MapStruct

De seguida foi definido que na fase de *build* do novo projeto Maven criado deveria ser executado o processo de geração de código de transformação da ferramenta MapStruct. Este

processo analisa as interfaces MapStruct existentes no projeto e gera o código correspondente de acordo com os mapeamentos definidos nessas interfaces. Para definir a execução do processo de geração durante a fase de *build* foi adicionado o conteúdo apresentado no Código 29 ao ficheiro *pom.xml* do projeto Maven.

```
<build>
        <plugins>
                <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-compiler-plugin</artifactId>
                        <version>3.5.1</version>
                        <configuration>
                                <source>1.7</source> <!-- or higher, depending on your project -->
                                <target>1.7</target> <!-- or higher, depending on your project -->
                                <annotationProcessorPaths>
                                        <path>
                                                <groupId>org.mapstruct</groupId>
                                                <artifactId>mapstruct-processor</artifactId>
                                                <version>${org.mapstruct.version}</version>
                                        </path>
                                </annotationProcessorPaths>
                        </configuration>
                </plugin>
        </plugins>
</build>
```

Código 29 – Processo geração código MapStruct durante a fase de *build* do projeto

Desta forma após adicionar as *interfaces* de mapeamento neste projeto ou após alguma alteração manual realizada sobre as *interfaces* de mapeamento o novo código de transformação MapStruct é automaticamente gerado. O Código 30 apresenta o resultado de geração de código para a entidade *PropertyBOAdaptable.*

```
1.  public class PropertyBOAdaptableMapperImpl implements PropertyBOAdaptableMapper {
2.  @Override
3.  public PropertyBOAdaptable toPropertyBOAdaptable(SSSPropertyBOAdaptable sSSPropertyBOAdaptable) {
4.          if (sSSPropertyBOAdaptable == null) {  return null; }
5.          PropertyBOAdaptable propertyBOAdaptable = new PropertyBOAdaptable();
6.          propertyBOAdaptable.setPropertyType(sSSPropertyBOAdaptable.getPropertyType());
7.          propertyBOAdaptable.setPropertyDataType(sSSPropertyBOAdaptable.getPropertyDataType());
8.          propertyBOAdaptable.setStringValue(sSSPropertyBOAdaptable.getStringValue());
9.          return propertyBOAdaptable;
10.     }
11. @Override
12. public List<PropertyBOAdaptable> toPropertyBOAdaptables(List<SSSPropertyBOAdaptable> sSSPropertyB
    OAdaptableList) {
13.         if (sSSPropertyBOAdaptableList == null) {
14.             return null;
15.         }
16.         List<PropertyBOAdaptable> list = new ArrayList<PropertyBOAdaptable> ();
17.         for (SSSPropertyBOAdaptable sSSPropertyBOAdaptable: sSSPropertyBOAdaptableList) {
18.             list.add(toPropertyBOAdaptable(sSSPropertyBOAdaptable));
19.         }
20.         return list;
21.     }
22. }
```

Código 30 – Código gerado pela ferramenta MapStruct

9 Avaliação da Solução

9.1 Introdução

Devido à natureza da solução a implementar, as grandezas que fazem sentido avaliar são a exatidão das transformações e a adequação dos diferentes mapeamentos suportados pela ferramenta. A exatidão e a satisfação do utilizador são as duas grandezas mais relevantes, visto que o principal objetivo da ferramenta de mapeamento é permitir a transformação correta entre diferentes modelos através da execução de um processo semiautomático com interação humana.

O resultado expectável é que essa transformação permita a obtenção de um modelo final com a estrutura e os dados corretos. Nesse sentido é possível o desenvolvimento de testes unitários para diferentes tipos de mapeamento e diferentes cenários. Através da execução desses testes deve ser possível determinar a exatidão do mapeamento executado.

Para avaliar a adequação dos diferentes mapeamentos suportados o caso prático de integração com o Symass permite realizar essa avaliação com o nível de abrangência pretendido. Ou seja, a avaliação da capacidade da ferramenta no apoio da integração do PDS permite validar esta grandeza.

A metodologia a utilizar para avaliar a solução deverá ser a validação do caso prático de integração com o Symass, baseada num caráter de teste e validação do resultado obtido. Se a ferramenta desenvolvida é adequada para o problema e se existem vantagens na sua utilização comparativamente ao mapeamento manual utilizado previamente.

9.2 Ferramenta Mapeamento vs Mapeamento Manual

A ferramenta de mapeamento de modelos pretende potencializar a produtividade no processo de integração de novos PDS com o S&S. Atualmente este processo de integração é realizado manualmente. Um dos critérios mais importantes a validar é a qualidade e exatidão do código de mapeamento gerado comparativamente ao código desenvolvido manualmente pelos programadores.

Neste sentido é possível identificar duas hipóteses:

1. **Exatidão do código de mapeamento gerado pela ferramenta:** pretende-se testar e validar se o código gerado apresenta um grau de exatidão aceitável comparativamente ao mapeamento manual;
2. **Aumento de produtividade comparativamente ao mapeamento manual:** com esta hipótese pretende-se validar se a inclusão da ferramenta de mapeamento no processo de integração com um novo PDS permite obter um ganho de produtividade face ao processo manual atualmente utilizado.

A validação das hipóteses apresentadas permite determinar se a utilização da ferramenta apresenta os benefícios pretendidos para melhorar a produtividade e reduzir o tempo de desenvolvimento.

Para validar a primeira hipótese pretende-se utilizar testes unitários JUnit que permitam validar o resultado final da transformação realizada utilizando o código de mapeamento gerado e o código de mapeamento manual. Os dados obtidos serão apresentados de forma tabular para permitir uma fácil visualização dos resultados obtidos.

Em relação à segunda hipótese de aumento de produtividade existe uma grande limitação na execução de um número considerável de repetições para permitir uma análise estatística adequada devido ao tempo necessário para realizar um mapeamento completo. Nesse sentido pretende-se reduzir a complexidade do mapeamento a testar, ou seja, a ideia será realizar mapeamentos parciais entre modelos utilizando as duas técnicas e registar o respetivo tempo de execução. A metodologia de avaliação será então a utilização de grupos de teste de diferentes mapeamentos para avaliar o desempenho das duas técnicas.

Para avaliar estatisticamente os resultados obtidos na validação da segunda hipótese pretende-se utilizar o teste estatístico Wilcoxon Signed Rank test, a escolha deste tipo de teste deve-se ao pequeno número de ensaios a realizar, visto que o tipo de teste pretendido apresenta um fator temporal considerável para a sua realização.

Para além das hipóteses apresentadas a satisfação do utilizador é outro aspeto importante para validar a ferramenta uma vez que é necessária interação humana para a execução dos mapeamentos. Nesse sentido para determinar a satisfação do utilizador durante a utilização da ferramenta devem ser realizados inquéritos de satisfação. Estes inquéritos devem ter por objetivo determinar a facilidade de utilização da ferramenta e o alinhamento das funcionalidades com as necessidades de utilização.

9.3 Testes Unitários

Durante a implementação da prova de conceito e para validar a coerência dos mapeamentos e as respetivas transformações diferentes testes unitários devem ser realizados. Os testes a realizar devem permitir validar a qualidade do código gerado pela solução.

Nesse sentido o mesmo teste unitário deve ser executado para o código gerado e para o código desenvolvido manualmente para os diferentes mapeamentos parciais a realizar. Com base na execução destes testes será possível determinar se a solução apresenta a exatidão pretendida.

A ferramenta apresenta a exatidão pretendida se todos os mapeamentos de teste definidos forem bem sucessivos e o resultado final da transformação esperado for cumprido.

9.4 Testes Desempenho

Um dos objetivos da ferramenta de mapeamento é o aumento da produtividade face ao mapeamento manual atualmente realizado.

9.5 Conclusões

Por razões de constrangimentos temporais não foi possível proceder-se a criação dos testes unitários de controlo da qualidade do código de transformação obtido pelo protótipo da ferramenta de mapeamento criado.

10 Conclusão

Atualmente o processo de integração de diferentes PDS com a plataforma S&S continua a apresentar-se como um grande desafio no desenvolvimento de *software* e no aumento da produtividade. A problemática da integração e comunicação entre diferentes aplicações no que se refere à necessidade de transformação e adaptação dos modelos específicos de cada aplicação é uma área atualmente em evolução (Biehl, 2010).

Nesse sentido os esforços exploratórios e de investigação realizados neste trabalho permitiram identificar a existência de várias técnicas, tecnologias e ferramentas direcionadas para este tipo de problema. No entanto no panorama de ferramentas gratuitas atualmente existentes (Biehl, 2010) (e que sejam do conhecimento aquando da realização deste trabalho) não foi possível identificar nenhuma capaz de responder aos requisitos de mapeamento concretos do cenário de integração da aplicação S&S com os diferentes PDS. Tendo obtido essa conclusão o exercício de implementação de um protótipo da ferramenta de mapeamento preconizada permitiu identificar que este tipo de tarefa apresenta diversos obstáculos e complexidades.

Outra observação em relação ao trabalho realizado foi que a implementação dos vários componentes apresentou-se como um desafio muito maior do que o antecipado a vários níveis: a quantidade de novas tecnologias e ferramentas estudadas e a sua aprendizagem apresentou um custo temporal elevado. Para além disso a implementação dos novos *plugins* Eclipse apresentou-se também como um grande desafio no qual existiu a necessidade de aprendizagem de todo o mecanismo de desenvolvimento de *plugins* Eclipse.

Como resultado da implementação do protótipo da ferramenta de mapeamento foi possível validar a viabilidade das tecnologias e ferramentas utilizadas. As técnicas de Model Driven Architecture e Model to Model Transformation (Sendall, 2003) demonstraram um papel relevante na solução implementada. Através da utilização dessas tecnologias e ferramentas foi possível implementar uma solução com as capacidades básicas necessárias para potenciar o interesse na continuação do esforço de implementação mais completo da ferramenta.

Foi possível ainda concluir que a tarefa de uniformização do modelo do S&S utilizando a abordagem e arquitetura MDA (Poole, 2001) (Singh, 2009) permitiria obter melhorias no processo de alteração do modelo do S&S e de igual forma as melhorias aplicadas ao processo

de geração de código poderiam também apresentar resultados muito positivos em termos de produtividade e facilidade de utilização. Atualmente o mecanismo de manutenção do modelo do S&S e a geração de código apresenta um elevado nível de complexidade de utilização e de manutenção do próprio código e *templates* de geração. A utilização do novo modelo Ecore e a correspondente metodologia de geração de código permitem aumentar consideravelmente a produtividade deste processo.

O trabalho realizado na implementação do protótipo da ferramenta de mapeamento não permitiu no entanto a obtenção de uma ferramenta utilizável num cenário real no seu estado atual.

10.1Objetivos Realizados

Os objetivos propostos inicialmente podem ser divididos em duas categorias:

- Análise e investigação – nesta categoria foram cumpridos os objetivos:
    - Estudo e investigação de ferramentas e tecnologias de mapeamento de modelos/objetos passíveis de serem utilizadas para resolver as necessidades de mapeamento de análise dos modelos;
    - Análise do enquadramento do modelo canónico e alinhamento com possíveis necessidades de ajuste/alteração.
- Implementação protótipo da solução – nesta categoria foram cumpridos parcialmente os objetivos:
    - Implementação (prova de conceito) de uma ferramenta de mapeamento entre modelos externos e o modelo interno;
    - Desenvolvimento de técnicas de geração de código para a realização da transformação dos modelos de acordo com regras de mapeamento previamente definidas.

Como aspetos positivos dos objetivos realizados temos o enquadramento do modelo canónico, este objetivo foi atingido e a sua implementação foi muito para além das expectativas inicialmente definidas. Isto porque foi possível implementar completamente o modelo do S&S em formato Ecore e desta forma permitir a utilização de ferramentas de edição textual de modelos como foi o caso da ferramenta Emfatic. Utilizando esta representação textual torna-se possível controlar alterações ao modelo com maior facilidade.

A conversão do modelo em formato Ecore permitiu ainda atingir um objetivo adicional com grande interesse para a msg life Iberia, nomeadamente a possibilidade de melhoria do processo de geração de código representativo do modelo S&S. Utilizando a ferramenta Texo e através de todas as melhorias realizadas sobre essa ferramenta foi possível atingir um processo de geração de código alinhado com as expectativas.

A implementação do protótipo da ferramenta de mapeamento permitiu a aplicação prática de vários conceitos MDA e MMT possibilitando validar o funcionamento em conjunto de várias tecnologias e ferramentas.

Outro aspeto positivo do trabalho realizado foi o desenvolvimento do componente de leitura de modelos. Todas as alterações e melhorias realizadas à ferramenta Modisco permitiram obter uma ferramenta capaz de obter representações Ecore de aplicações Java de forma genérica e de forma automática. Este componente pode ser útil mesmo fora do âmbito de utilização da ferramenta desenvolvida e foi desenvolvido de forma a possibilitar exatamente esse cenário, uma vez que se trata de um *plugin* Eclipse que pode ser facilmente instalado independentemente de todos os outros componentes desenvolvidos.

A análise dos objetivos inicialmente propostos e que foram realizados permite perceber que não foram cumpridos todos os objetivos inicialmente propostos. Um dos objetivos mais ambiciosos inicialmente definidos passava pela possibilidade de validar o protótipo da ferramenta utilizando o caso prático real da integração com o PDS Symass e desta forma permitir avaliar a adequação da ferramenta. No entanto este objetivo não foi cumprido devido a constrangimentos temporais e pelo facto de o protótipo desenvolvido ter revelado a necessidade de investimento no refinamento e melhoria das funcionalidades implementadas.

Posto isto, apesar de alguns objetivos não terem sido atingidos é de referir que os objetivos que foram atingidos revelaram um grau de complexidade e inovação elevado. Permitindo a obtenção de informação e conhecimento extremamente valioso nesta área, tanto a nível pessoal como em concreto para a msg life Iberia.

10.2Limitações e Trabalho Futuro

Ao longo deste trabalho foram sendo identificadas várias limitações da solução preconizada. Sendo a principal limitação relaciona-se com a falta de suporte para diferentes tipos de mapeamento entre elementos. Ou seja, o componente de definição de mapeamentos da ferramenta apenas possibilita definir mapeamentos entre elementos iguais sem permitir a realização de tipos de transformação complexos entre elementos de diferentes modelos.

O componente de definição de mapeamentos apresenta outra limitação que foi identificada durante a utilização do modelo Ecore disponibilizado pela PM. A limitação encontrada relaciona-se com a quebra de performance experienciada quando se tenta mapear um modelo com o tamanho e complexidade como o modelo da PM. Este modelo inclui dependências para outros modelos e metamodelos internos da PM o que torna o processo de leitura pela ferramenta AMW consideravelmente mais lento.

Outra limitação identificada relaciona-se com o componente de leitura de modelos. A integração com aplicações para as quais não seja possível aceder ao código fonte para executar a leitura do modelo utilizando a ferramenta de leitura tornaria todo o processo inválido. No entanto sendo este um cenário pouco realista podemos menosprezar consideravelmente esta

limitação. Continuando a discutir as limitações da componente de leitura é possível identificar um cenário prático mais comum. Na integração com uma aplicação que disponibiliza serviços para serem consumidos, nomeadamente através de uma API SOAP ou REST. A solução idealizada não contempla este cenário, pelo menos de forma transparente.

De forma geral podemos identificar que a solução proposta apresenta um considerável nível de complexidade de utilização. Existem vários processos na utilização da componente de definição de mapeamento que poderiam ser simplificados. Para tornar a solução mais apelativa todo o processo de mapeamento deveria ser simplificado e para alguns dos passos necessários deveria existir um comportamento por defeito. Deixando em aberto a possibilidade de permitir alterações na configuração se necessário.

O componente de execução de mapeamentos poderia ser melhorado para permitir disponibilizar mais facilmente o código gerado MapStruct. Atualmente este componente necessita de interação manual no que se refere à cópia das classes geradas para o *package* Java de destino.

Tendo por base as limitações apresentadas podemos identificar como trabalho futuro as seguintes áreas:

- Melhoria do componente de mapeamento através da inclusão de mais tipos de regras de mapeamento. Atualmente o catálogo de mapeamento é extremamente reduzido e apenas serviu o propósito de um protótipo de solução. Para tornar viável a utilização da ferramenta de mapeamento este catálogo deveria incluir mapeamentos mais complexos;
- Finalizar a implementação do componente de execução de mapeamento para permitir melhorar o processo de transformação concreta entre modelos, este processo ainda depende de interação manual para tarefas que poderiam ser claramente automatizadas;
- Melhorar a integração dos vários componentes de forma a facilitar a utilização da ferramenta como uma só e não delegar para o *developer* a necessidade de conhecimento dos processos para proceder à execução das diferentes tarefas de mapeamento;
- Validar a viabilidade do protótipo criado para o caso concreto previamente identificado de mapeamento com o PDS Symass;
- Melhoria da solução para permitir o suporte de integração com aplicações que apenas disponibilizem serviços através de uma API SOAP ou REST;
- Renomear o metamodelo MapStructModel criado, não existe nenhuma dependência entre o metamodelo e a ferramenta MapStruct.

Podemos observar que a lista de trabalho futuro contém vários itens com um esforço temporal associado possivelmente considerável. Como tal a viabilidade da continuidade da implementação da ferramenta fica dependente da disponibilidade de capacidade de desenvolvimento para poder ser terminada.

Referências

Allee, V., 2000. The value evolution - Addressing larger implications of an intellectual capital and intangibles perspective. *Journal of Intellectual Capital,* 1(1), pp. 17-32.

Allee, V., 2008. Value network analysis and value conversion of tangible and intagible assets. *Journal of Intellectual Capital,* 9(1), pp. 5-23.

An, Y. B. A. M. R. J. M. J., 2007. *A Semantic Approach to Discovering Schema Mapping Expressions,* s.l.: IEEE.

Biehl, M., 2010. *Literature Study on Model Transformations.* Stockholm, Royal Institute of Technology.

Blewitt, A., 2013. *Eclipse 4 Plug-in Development by Example.* Birmingham: Packt Publishing Ltd..

Bonifati, A. M. G. P. P. V. Y., 2010. *Discovery and Correctness of Schema Mapping Transformations,* s.l.: Springer.

Bruneliere, H., 2011. *MoDisco in a Nutshell!.* [Online]
Available at: https://jaxenter.com/modisco-in-a-nutshell-103816.html
[Accessed 25 Fevereiro 2017].

Bruneliere, H. C. J. D. G. M. F., 2014. Modisco: a Model Driven Reverse Engineering Framework. *Information adn Software Technology*, pp. 10012-1032.

Budinsky, F. S. D. M. E. E. R. G. T. J., 2004. *Eclipse Modeling Framework: A Developer's Guide.* s.l.:Addison-Wesley Professional.

Daly, C., 2004. *Emfatic.* [Online]
Available at: https://www.eclipse.org/emfatic/
[Accessed 28 Agosto 2016].

Debbie, M., 2008. *Insurance product development: Innovating to win and retain customers.* [Online]
Available at: http://www.jiops.com/01/2008/insurance-product-development-innovating-to-win-and-retain-customers/
[Accessed 10 Janeiro 2016].

Didonet Del Fabro, M., 2010. *Model Weaving Establishing links between model elements.* s.l., MDE Diploma - IBM.

Didonet Del Fabro, M. B. J. V. P., 2006. *Weaving Models with the Eclipse AMW plugin.* Esslingen, Germany, Eclipse Modeling Symposium, Eclipse Summit Europe.

Didonet Del Fabro, M. V. P., 2008. Towards the efficient development of model transformations using model weaving and matching transformations. *Software and Systems Modeling SoSym - Springer Berlin*, 29 Abril.

Económico, O. J., 2015. *Msg Life quer ajudar seguradoras a lançar produtos mais rápido.* [Online]
Available at: http://www.oje.pt/msg-life-quer-ajudar-seguradoras-a-lancar-produtos-mais-rapido/
[Accessed 15 Janeiro 2016].

Farron, B., 2004. *Reinsurance Group of America - Life Insurance Product Development Innovation and Optimization.* [Online]
Available at: https://www.rgare.com/knowledgecenter/Documents/Life%20Insurance%20Product%20Development_Final%201-28-15%20(2).pdf
[Accessed 5 Fevereiro 2016].

Gilberto, F., 2012. *Manual Prático dos Seguros.* 2ª Edição ed. Lisboa: Lidel - edições técnicas.

Goubert, L., 2010. *Acceleo vs Acceleo vs Xpand.* [Online]
Available at: http://eclipsemde.blogspot.pt/2010/12/acceleo-vs-acceleo-vs-xpand.html
[Accessed 5 Abril 2017].

Group, O. M., 2016. *Object Management Group.* [Online]
Available at: http://www.omg.org/
[Accessed 4 Janeiro 2016].

Heidenreich, F. J. J. S. M. W. C., 2009. *Construct to Reconstruct - Reverse Engineering Java Code with JaMoPP.* Dresden, Technische Universitat Dresden.

Hoisl, B., 2012. *AMW.* [Online]
Available at: http://wiki.eclipse.org/AMW
[Accessed 28 Março 2017].

Jouault, F., 2013. *ATL/Tutorials - Create a simple ATL transformation.* [Online]
Available at: https://wiki.eclipse.org/ATL/Tutorials_-_Create_a_simple_ATL_transformation
[Accessed 11 Maio 2017].

Jouault, F. A. F. B. J. K. I., 2008. *ATL: A model transformation tool,* s.l.: Elsevier B.V..

Koen, P. A. B. H. M. J. K. E. J., 2014. Managing the Front End of Innovation - Part I. *Research-Technology Management*, Março.

Koen, P. A. B. H. M. J. K. E. J., 2014. Managing the Front End of Innovation - Part II. *Research-Technology Management*, Maio.

Koen, P. A. e. a., 2002. Fuzzy Front End: Effective Methods, Tools and Techniques. In: S. Somermeyer, ed. *PDMA Toolbook for new product development.* New York: John Wiley & Sons.

Kurtev, I., 2006. *An introduction to the MOF 2.0 QVT standard with focus on the Operational Mappings,* Nantes, France: ATLAS group, INRIA & University of Nantes.

Madiot, M., 2017. *Acceleo.* [Online]
Available at: https://www.eclipse.org/acceleo/
[Accessed 20 Agosto 2016].

Mesle, S. L., 2015. *Selma Java bean mapping that compiles.* [Online]
Available at: http://www.selma-java.org/
[Accessed 5 Fevereiro 2016].

Morling, G., 2015. *MapStruct User Guide.* [Online]
Available at: http://mapstruct.org/documentation/#section-intro
[Accessed 12 Fevereiro 2016].

Musset, J. J. É. L. S., 2008. *Acceleo User Guide.* s.l.:Obeo.

Nederpel, R., 2005. *A QVT model transformation language represented by graph production systems,* Enschede, Netherlands: University of Twente.

Nicola, S., 2014. *A Quantitative Model for Decomposing and Assessing the Value for the Customer,* Porto: Univerdidade do Porto - Faculdade de Engenharia.

Nicola, S. F. E. P. F. J. J. P., 2012. A Novel Framework for Modeling Value for the Customer, an Essay on Negotiation. *International Journal of Information Technology & Decision Making,* 11(03), pp. 661-703.

OMG/MOF, 2015. *MetaObject Facility (MOF).* [Online]
Available at: http://www.omg.org/spec/MOF/2.5/
[Accessed 11 Janeiro 2016].

OMG/OCL, 2014. *Object Constraint Language.* [Online]
Available at: http://www.omg.org/spec/OCL/2.4/
[Accessed 12 Janeiro 2016].

OMG/QVT, 2011. *MOF 2.0 Query/View/Transformation (QVT).* [Online]
Available at: http://www.omg.org/spec/QVT/1.1/
[Accessed 10 Janeiro 2016].

OMG, 2015. *OMG's MetaObject Facility.* [Online]
Available at: http://www.omg.org/mof/
[Accessed 10 Fevereiro 2016].

Osterwalder, A. P. Y. T. C., 2005. Clarifying Business Models: Origins, Present and Future of the Concept. *Communications of the Association for Informaion Systems,* Volume 15.

Pereira, F. M., 2015. O Futuro dos Seguros. *Insurance Trends*, 3 Julho, p. 5.

Poole, J. D., 2001. *Model-Driven Architecture: Vision, Standards And Emerging Technologies.* s.l., ECOOP.

Rees, D., 2012. *Emfatic.* [Online]
Available at: https://wiki.eclipse.org/Emfatic
[Accessed 25 Agosto 2016].

Rey, A., 2015. *Object-to-object mapping framework microbenchmark.* [Online]
Available at: https://github.com/arey/java-object-mapper-benchmark
[Accessed 12 Fevereiro 2016].

RGA, 2014. *Life Insurance Product Development Innovation and Optimization,* s.l.: RGA - Reinsurance Group of America.

Sendall, S. K. W., 2003. *Model Transformation – the Heart and Soul of Model-Driven Software Development.* Lausanne, Swiss Federal Institute of Technology.

Singh, Y. S. M., 2009. *Model Driven Architecture: A Perspective.* Patiala, India, IEEE - International Advance Computing Conference.

Sousa, P. G. M. J., 2015. Integration in an Insurance Distribution Plataform. *IADIS International Conference Information Systems.*

Standardization, I. -. I. O. f., 2014. *XML Metadata Interchange (XMI),* s.l.: ISO - International Organization for Standardization.

Taal, M., 2015. *Texo.* [Online]
Available at: https://wiki.eclipse.org/Texo#texo
[Accessed 12 Março 2017].

Thomas, D., 2004. MDA: Revenge of the Modelers or UML Utopia. *Thought Works*.

Tielin, Q. Y. L. W. P., 2010. *A Model Transformation Plataform Design Based on a Model Driven Architecture.* Changsha, China, IEEE - International Advance Computing Conference.

TutorialsPoint, 2016. *Design Patterns in Java Tutorial.* [Online]
Available at: http://www.tutorialspoint.com//design_pattern/index.htm
[Accessed 14 Fevereiro 2016].

Vogel, L., 2016. *Eclipse Extension Points and Extensions - Tutorial.* [Online]
Available at: http://www.vogella.com/tutorials/EclipseExtensionPoint/article.html
[Accessed 15 Abril 2017].

Vogel, L., 2016. *Eclipse Modeling Framework (EMF) - Tutorial.* [Online]
Available at: http://www.vogella.com/tutorials/EclipseEMF/article.html
[Accessed 22 Janeiro 2017].

Vogel, L., 2016. *Using Maven within the Eclipse IDE - Tutorial.* [Online]
Available at: http://www.vogella.com/tutorials/EclipseMaven/article.html
[Accessed 20 Setembro 2017].

Wagelaar, D. B. H. F. T. R., 2015. *ATL User Guide - Introduction.* [Online]
Available at: https://wiki.eclipse.org/ATL/User_Guide_-_Introduction
[Accessed 9 Fevereiro 2016].

Zeithaml, V. a., 1988. Consumer Perceptions of Price, Quality, and Value: A Means-End Model and Synthesis of Evidence. *Journal of Marketing,* Volume 52, pp. 2-22.

Anexos

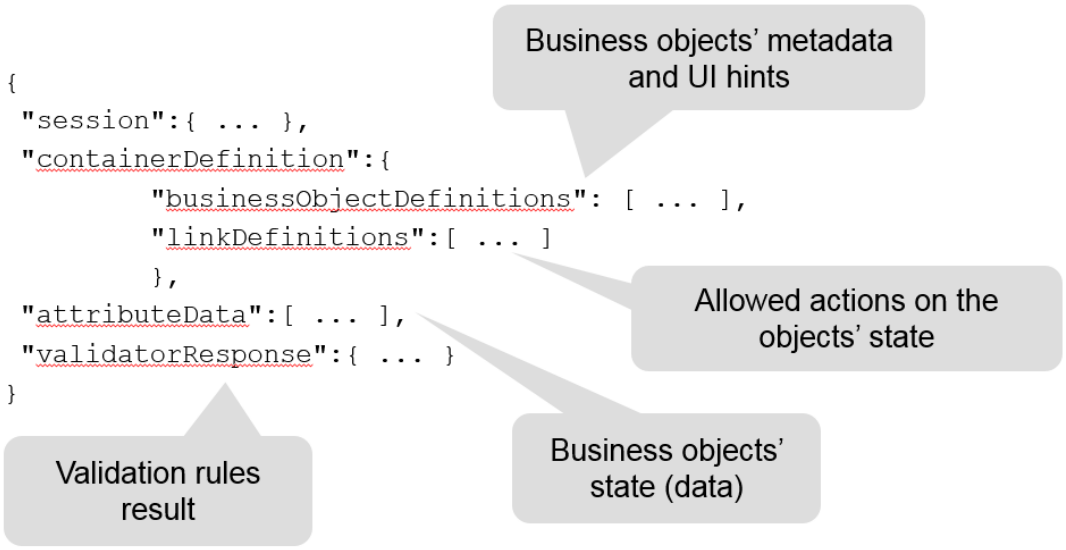


Figura 64 – Principais componentes do Container And Data Holder

Fonte da figura: Documentação interna e proprietária da msg life Iberia – Sales & Service Tech Overview 2016

Excertos de código de leitura informação classes do modelo Java com utilização de *reflection*

```
1.  private static void allFieldsToMap(List<String> methodsNameList, List <Map<String, Object>> attribut
    es, List<Map<String, Object>> frontendNotMergeableAttributes, List<Map<String, Object>> pdsNotMe
    rgeableAttributes, List<Field> fields, Map<String, Map<String, String>> idFieldsMap, List<String
    > mandatoryFieldsForContainerList) {
2.      for (Field field: fields) {
3.          if (!isFieldToExclude(field) && !isParentBOField(field)) {
4.              Map <String, Object> map = new HashMap <> ();
5.              String rawFieldName = field.getName();
6.              String fieldName = rawFieldName;
7.              Class<?> rawFieldTypeClass = field.getType();
8.              Class<?> fieldTypeClass = getFieldClass(field);
9.              boolean isArray = isFieldTypeArrayLike(rawFieldTypeClass);
10.             if (isArray) {
11.                 map.put("isArray", Boolean.valueOf(true));
12.                 map.put("typeArray", fieldTypeClass.getCanonicalName());
13.             }
14.             String fieldType = getTypeDescription(rawFieldTypeClass);
15.             map.put("type", fieldType);
16.             map.put("isSetType", TypeUtils.isAssignable(rawFieldTypeClass, Set.class));
17.             map.put("isListType", TypeUtils.isAssignable(rawFieldTypeClass, List.class));
18.             Boolean isPrimitive = Boolean.valueOf(fieldTypeClass.isPrimitive());
19.             boolean isIdField = isIdField(field);
20.             if (isIdField) {
21.                 Map<String, String> fieldInformation = new HashMap <> ();
22.                 fieldInformation.put("isPrimitive", isPrimitive.toString());
23.                 fieldInformation.put("fieldType", fieldTypeClass.getName());
24.                 fieldInformation.put("upperFieldName", StringUtils.capitalize(fieldName));
25.                 fieldInformation.put("idFieldType", getTypeForIdField(field.getType()));
26.                 idFieldsMap.put(fieldName, fieldInformation);
27.             }
28.             boolean isMandatoryFieldForContainer = isMandatoryFieldForContainer(field);
29.             if (isMandatoryFieldForContainer) {
30.                 mandatoryFieldsForContainerList.add(fieldName);
31.             }
32.             boolean isNotToClone = false;
33.             if (isNotToClone(field)) {
34.                 isNotToClone = true;
35.             }
36.             map.put("isNotToClone", isNotToClone);
37.             boolean isNotToCopyDataFrom = false;
38.             if (isNotToCopyDataFrom(field)) {
39.                 isNotToCopyDataFrom = true;
40.             }
41.             map.put("isNotToCopyDataFrom", isNotToCopyDataFrom);
42.             boolean isNotToReset = false;
43.             if (isNotToReset(field)) {
44.                 isNotToReset = true;
45.             }
46.             map.put("isNotToReset", isNotToReset);
47.             map.put("isPrimitive", isPrimitive);
48. // We have 3 different possible scenarios (which for us are // different): // 1- BOAdaptable //
    1.1- list/array of BOAdaptable // 1.2- single BOAdaptable // 2- Primitive type // 3- AbstractDis
    creteValue
49.             if (TypeUtils.isAssignable(fieldTypeClass, BOAdaptable.class)) {
50.                 map.put("isBOAdaptable", Boolean.valueOf(true));
51.                 if (isArray) {
52. // We are doing this name transformation because the // names used for List of BOAdaptables are
    not aligned // with the respective getters and setters methods. // fieldName = "all" // + WordUt
    ils.capitalize(rawFieldName.substring( // 0, rawFieldName.length() - 2));
53.                 }
```

```
            } else if (TypeUtils.isAssignable(fieldTypeClass, AbstractDiscreteValue.class) || (Type
Utils.isAssignable(fieldTypeClass, SSSAbstractDiscreteValue.class))) {
                map.put("isAbstractDiscreteValue", Boolean.valueOf(true));
            } else if (!SSSAbstractDiscreteValue.class.equals(fieldTypeClass)) {
                map.put("isAttribute", Boolean.valueOf(true));
            }
            map.put("name", fieldName);
            map.put("upperName", StringUtils.capitalize(fieldName));
            String setMethodName = "set" + WordUtils.capitalize(fieldName);
            if (methodsNameList.contains(setMethodName)) {
                map.put("existsSetter", true);
                map.put("setterName", setMethodName);
            } else {
                map.put("existsSetter", false);
            }
            String getMethodName = "get" + WordUtils.capitalize(fieldName);
            if (methodsNameList.contains(getMethodName)) {
                map.put("existsGetter", true);
                map.put("getterName", getMethodName);
            } else {
                map.put("existsGetter", false);
            }
            attributes.add(map);
            if (isFieldNotToMergeFrontEnd(field)) {
                frontendNotMergeableAttributes.add(map);
            }
            if (isFieldNotToMergePDS(field)) {
                pdsNotMergeableAttributes.add(map);
            }
        }
    }
}
```

Código 31 – Excerto de código de obtenção de informação sobre atributos de classes Java do modelo

```
protected static List<Class<?>> findMyTypes(String basePackage) throws IOException, ClassNotFoundExc
eption {
    final ResourcePatternResolver resourcePatternResolver = new PathMatchingResourcePatternResolv
er();
    final MetadataReaderFactory metadataReaderFactory = new CachingMetadataReaderFactory(resource
PatternResolver);
    final List<Class<?>> candidates = new ArrayList<>();
    final String packageSearchPath = ResourcePatternResolver.CLASSPATH_ALL_URL_PREFIX + resolveBa
sePackage(basePackage) + "/" + "**/*.class";
    final Resource[] resources = resourcePatternResolver.getResources(packageSearchPath);
    for (final Resource resource: resources) {
        if (resource.isReadable()) {
            final MetadataReader metadataReader = metadataReaderFactory.getMetadataReader(resourc
e);
            if (isCandidate(metadataReader)) {
                final Class<?> clazz = Class.forName(metadataReader.getClassMetadata().getClassNa
me());
// Do not consider inner classes or subpackages
                if (!clazz.isMemberClass() && clazz.getEnclosingClass() == null && clazz.getPackag
e().getName().equals(basePackage) && !candidates.contains(clazz)) {
                    candidates.add(clazz);
                }
            }
        }
    }
    return candidates;
}
protected List<Method> getAllDeclaredMethodsInclusiveFromSuperClass(Class<?> targetClass) {
```

```
22.         final List<Method> allMethods = new ArrayList<>(Arrays.asList(targetClass.getMethods()));

23.         final List<Method> allMethodsFromAbstractSuperClass = new ArrayList<>();
24.         Class<?> parent = targetClass.getSuperclass();
25.         while (parent != null) { // Exclude the AbstractGenerated Class fields
26.             if (parent.getSimpleName().startsWith(ABSTRACT_GENERATED_BOADAPTABLE_CLASS_PREFIX)) {
27.                 allMethodsFromAbstractSuperClass.addAll(Arrays.asList(parent.getDeclaredMethods(
    )));
28.             }
29.             parent = parent.getSuperclass();
30.         }
31.         allMethods.removeAll(allMethodsFromAbstractSuperClass);
32.         Collections.sort(allMethods, new Comparator < Method > () {@
33.             Override public int compare(Method o1, Method o2) {
34.                 return o1.getName().compareTo(o2.getName());
35.             }
36.         }); // FIXME: PropertyBOAdaptable hierarchy changes made this necessary - we // need to
    review the hierarchy.
37.         handlePropertyBOSpecificHierarchy(targetClass, allMethods);
38.         handleDuplicatedMethods(allMethods); // FIXME: This should not be needed -
    - This should not happen but we are // repeating methods in BOAdaptable classes and in the // Ab
    stractBOAdaptable class.
39.         return new ArrayList < > (new LinkedHashSet < > (allMethods));
40.     }
41.
42. /**
43. * FIXME: PropertyBOAdaptable hierarchy changes made this necessary - we
44. * need to review the hierarchy.
45. *
46. * @param targetClass
47. * @param allMethods
48. */
49. private void handlePropertyBOSpecificHierarchy(Class<?> targetClass, List<Method> allMethods) {
50.     if ("SSSPropertyBOAdaptable".equals(targetClass.getSimpleName()) || "SSSCustomerPropertyBOAda
    ptable".equals(targetClass.getSimpleName())) {
51.         final Iterator<Method> iterator = allMethods.iterator();
52.         while (iterator.hasNext()) {
53.             final Method currentMethod = iterator.next();
54.             if ("getPropertyType".equals(currentMethod.getName())) {
55.                 if (!TypeUtils.isAssignable(getMethodReturnClass(currentMethod), SSSPropertyType.
    class)) {
56.                     iterator.remove();
57.                 }
58.             } else if ("getPropertyCategory".equals(currentMethod.getName())) {
59.                 if (!TypeUtils.isAssignable(getMethodReturnClass(currentMethod), SSSPropertyCateg
    ory.class) && !TypeUtils.isAssignable(getMethodReturnClass(currentMethod), SSSCategory.class)) {
60.                     iterator.remove();
61.                 }
62.             } else if ("getPropertyClass".equals(currentMethod.getName())) {
63.                 if (!TypeUtils.isAssignable(getMethodReturnClass(currentMethod), SSSPropertyClass
    .class)) {
64.                     iterator.remove();
65.                 }
66.             }
67.         }
68.     }
69. }
```

Código 32 – Excerto código de obtenção de informação sobre as classes e métodos a processar do modelo Java

Representação textual Emfatic dos tipos de dados *AbstractDiscreteValue* do novo modelo Ecore do S&S – ficheiro *salesNServiceTypes.emf*

```
@namespace(uri="http://www.example.org/salesNServiceTypesModel", prefix="type")
package type;

abstract class AbstractDiscreteValue {
	attr Integer ~id;
	attr String acronym;
	attr String name;
	attr String shortText;
	attr Integer order;
}

// Illustration AbstractTypes
class IllustrationType extends AbstractDiscreteValue {}
class IllustrationStatus extends AbstractDiscreteValue {}
class YesNoType extends AbstractDiscreteValue {}
class BusinessType extends AbstractDiscreteValue {}

// Illustration, Contract and Adjustment shared AbstractTypes
class GroupConstraintOperationType extends AbstractDiscreteValue {}

// Contract AbstractTypes
class ProductType extends AbstractDiscreteValue {}
class ProductFamily extends AbstractDiscreteValue {}
class ContractStatus extends AbstractDiscreteValue {}
class ContractType extends AbstractDiscreteValue {}
class Currency extends AbstractDiscreteValue {}
class Brand extends AbstractDiscreteValue {}
class Channel extends AbstractDiscreteValue {}
class FundingType extends AbstractDiscreteValue {}
class PaymentMethod extends AbstractDiscreteValue {}
class PaymentMode extends AbstractDiscreteValue {}
class ProductCategory extends AbstractDiscreteValue {}
class CalcStatus extends AbstractDiscreteValue {}
class ActionType extends AbstractDiscreteValue {}
class Language extends AbstractDiscreteValue {}

// Package AbstractTypes
class PackageType extends AbstractDiscreteValue {}
class PackageStatus extends AbstractDiscreteValue {}

// Experience AbstractTypes
class CoverageType extends AbstractDiscreteValue {}
class ClaimsBasis extends AbstractDiscreteValue {}
class IBNRBasis extends AbstractDiscreteValue {}
class EnrollmentType extends AbstractDiscreteValue {}
class ExperienceType extends AbstractDiscreteValue {}
class ExperienceSetStatus extends AbstractDiscreteValue {}

// Role AbstractTypes
class ParticipantType extends AbstractDiscreteValue {}
class RoleType extends AbstractDiscreteValue {}

// User AbstractTypes
class ReferralLevel extends AbstractDiscreteValue {}

// Adjustment AbstractTypes
class AdjustmentProperty extends AbstractDiscreteValue {}
class CalcType extends AbstractDiscreteValue {}
```

```
// PremiumComponent AbstractTypes
class PremiumComponentType extends AbstractDiscreteValue {}
class InputOutputType extends AbstractDiscreteValue {}

// RateComponent AbstractTypes
class RateComponentType extends AbstractDiscreteValue {}
class RateComponentCategory extends AbstractDiscreteValue {}
class RateComponentClass extends AbstractDiscreteValue {}

// EligibilityClass AbstractTypes
class EligibilityClassType extends AbstractDiscreteValue {}

// ExperiencePeriod AbstractTypes
class ExperiencePeriodType extends AbstractDiscreteValue {}

// Coverage AbstractTypes
class CoverageCategory extends AbstractDiscreteValue {}

// Individual AbstractTypes
class NamePrefix extends AbstractDiscreteValue {}
class NameSuffix extends AbstractDiscreteValue {}
class Title extends AbstractDiscreteValue {}
class Gender extends AbstractDiscreteValue {}

// Document AbstractTypes
class DocumentType extends AbstractDiscreteValue {}
class DocumentCategory extends AbstractDiscreteValue {}

// PremiumDetail AbstractTypes
class PremiumDetailType extends AbstractDiscreteValue {}
class UnitOfMeasurement extends AbstractDiscreteValue {}

// PlanDesign AbstractTypes
class PlanDesignType extends AbstractDiscreteValue {}

// RateDetail AbstractTypes
class RateDetailType extends AbstractDiscreteValue {}
class RateDetailCategory extends AbstractDiscreteValue {}

// Organization AbstractTypes
class LegalStructureType extends AbstractDiscreteValue {}

// Address AbstractTypes
class AddrType extends AbstractDiscreteValue {}
class AddrUsage extends AbstractDiscreteValue {}
class StateName extends AbstractDiscreteValue {}

// InsuredPerson AbstractTypes
class PartyType extends AbstractDiscreteValue {}
class OccClass extends AbstractDiscreteValue {}
class ThirdPartySystem extends AbstractDiscreteValue {}

// Producer AbstractTypes
class StatusType extends AbstractDiscreteValue {}

// Property AbstractTypes
class PropertyType extends AbstractDiscreteValue {}
class PropertyDataType extends AbstractDiscreteValue {}
class PropertyCategory extends AbstractDiscreteValue {}
class PropertyClass extends AbstractDiscreteValue {}
class PropertyKeyValue extends AbstractDiscreteValue {}
```

```
// PropertyGroup AbstractTypes
class PropertyGroupType extends AbstractDiscreteValue {}

// BusinsessTransaction AbstractTypes
class TransactionType extends AbstractDiscreteValue {}

// Phone AbstractTypes
class PhoneType extends AbstractDiscreteValue {}
```

Código 33 - Representação textual Emfatic dos tipos de dados *AbstractDiscreteValue* do novo modelo Ecore do S&S

Representação textual Emfatic das entidades do novo modelo Ecore do S&S – ficheiro *salesNServiceModel.emf*

```
@namespace(uri="http://www.example.org/salesNServiceModel", prefix="salesNServiceModel")
package salesNServiceModel;

import
"platform:/resource/SalesAndServiceModelResources/salesandservicemodel/salesNServiceTypes.ecor
e";

class Illustration extends BaseBO {
	unsettable attr String boName = "IllustrationBO";
	@IdField
	attr String illustrationID;
	attr String illustrationName;
	val type.IllustrationStatus illustrationStatus;
	attr Calendar effectiveDate;
	attr Calendar createDate;
	attr String createdBy;
	attr String comment;
	attr String originalIllustrationID;
	attr String preparedBy;
	attr String producerNumber;
	attr Calendar updateDate;
	attr String updatedBy;
	val type.IllustrationType illustrationtype;
	val type.YesNoType isTemplate;
	val type.BusinessType containerMode;
	val type.YesNoType lockedIndicator;
	attr String groupConstraintType;
	val type.GroupConstraintOperationType groupConstraintOperationType;
	transient attr String sessionId;
	attr Boolean inTransitState = false;

	// Parent relations

	// Children single
	@ChildBO
	val Producer#illustrationBO producerBO;

	// Children multiple
	@ChildBO
	val Contract[*]#illustrationBO contractBOs;
	@ChildBO
	val Package[*]#illustrationBO packageBOs;
	@ChildBO
	val ProductFamily[*]#illustrationBO productFamilyBOs;
	@ChildBO
	val Experience[*]#illustrationBO experienceBOs;
	@ChildBO
	val Role[*]#illustrationBO roleBOs;
	@ChildBO
	val Property[*]#illustrationBO propertyBOs;
}

class Contract extends BaseBO {
	unsettable attr String boName = "ContractBO";
	@IdField
	attr String quoteId;
```

```
val type.ProductType productType;
val type.ProductFamily productFamily;
attr String contractName;
val type.ContractStatus contractStatus;
val type.ContractType contractType;
val type.Currency currency;
attr Calendar createTimestamp;
attr Calendar effectiveDate;
attr Calendar endDate;
attr Calendar finalizeDate;
attr String contractId;
val type.Brand brand;
val type.Channel channel;
attr String comment;
val type.FundingType fundingType;
attr Boolean includedInReport;
val type.PaymentMethod paymentMethod;
val type.PaymentMode paymentMode;
attr String preparedBy;
attr String producerNumber;
val type.ProductCategory productCategory;
val type.CalcStatus calcStatus;
attr Double substandardRating;
val type.ActionType contractAction;
attr Calendar updateDate;
attr Calendar updateStatusDate;
val type.Language language;
attr Short contractTransactions;
attr String groupConstraintType;
val type.GroupConstraintOperationType groupConstraintOperationType;
attr int contractNumber;
attr Integer externalContractNumber;
unique ordered attr String[*] experiences;
unique ordered attr String[*] census;

// Parent relations
@ParentBO
ref Illustration#contractBOs illustrationBO;

// Children single
@ChildBO
val BusinessTransaction#contractBO businessTransactionBO;
@ChildBO
val Producer#contractBO producerBO;
@ChildBO
val User#contractBO userBO;
@ChildBO
val DocumentsAggregator#contractBO documentsAggregatorBO;

// Children multiple
@ChildBO
val Adjustment[*]#contractBO adjustmentBOs;
@ChildBO
val Experience[*]#contractBO experienceBOs;
@ChildBO
val PremiumComponent[*]#contractBO premiumComponentBOs;
@ChildBO
val Property[*]#contractBO propertyBOs;
@ChildBO
val PropertyGroup[*]#contractBO propertyGroupBOs;
```

```
        @ChildBO
        val RateComponent[*]#contractBO rateComponentBOs;
        @ChildBO
        val Role[*]#contractBO roleBOs;
        @ChildBO
        val PackageLife[*]#contractBO packageBOs;
        @ChildBO
        val Authorization[*]#contractBO authorizationBOs;
        @ChildBO
        val EligibilityClass[*]#contractBO eligibilityClassBOs;
}

class Package extends BaseBO {
        unsettable attr String boName = "PackageBO";
        @IdField
        attr Integer packageId;
        val type.PackageType packageType;
        attr String name;
        attr String owner;
        attr String description;
        val type.PackageStatus status;
        attr Calendar createDate;
        attr Calendar updateDate;
        attr Calendar releaseDate;
        attr Calendar expirationDate;
        attr Boolean likelyToSell;
        attr Short packageTransactions;
        unique ordered attr String[*] contracts;
        attr Boolean inTransitState = false;
        val Contract contractDefinition;

        // Parent relations
        @ParentBO
        ref Illustration#packageBOs illustrationBO;

        // Children single
        @ChildBO
        val BusinessTransaction#packageBO businessTransactionBO;
        @ChildBO
        val DocumentsAggregator#packageBO documentsAggregatorBO;

        // Children multiple
        @ChildBO
        val Property[*]#packageBO propertyBOs;
        @ChildBO
        val PropertyGroup[*]#packageBO propertyGroupBOs;
        @ChildBO
        val ProductFamily[*]#packageBO productFamilyBOs;
        @ChildBO
        val Grouping[*]#packageBO groupingBOs;
        @ChildBO
        val Authorization[*]#packageBO authorizationBOs;
        @ChildBO
        val PropertyGroupHolder[*]#packageBO propertyGroupHolderBOs;
}

class ProductFamily extends BaseBO {
        unsettable attr String boName = "ProductFamilyBO";
        @IdField
        attr String productFamilyId;
        val type.ProductFamily type;
```

```
        attr Boolean selected = false;
        // Parent relations
        @ParentBO
        ref Illustration#productFamilyBOs illustrationBO;
        @ParentBO
        ref Package#productFamilyBOs packageBO;

        // Children single

        // Children multiple
        @ChildBO
        val Property[*]#productFamilyBO propertyBOs;
        @ChildBO
        val Broker[*]#productFamilyBO brokerBOs;
}

class Experience extends BaseBO {
        unsettable attr String boName = "ExperienceBO";
        @IdField
        attr Integer experienceId;
        attr String experienceSetName;
        val type.CoverageType coverageType;
        attr Calendar originalEffectiveDate;
        attr Calendar asOfDate;
        val type.ClaimsBasis claimsBasis;
        attr Double expectedLossRatio;
        attr Boolean enrollmentCostIncludedInExperience;
        val type.IBNRBasis ibnrBasis;
        val type.EnrollmentType lastEnrollmentType;
        // Not yet used
        attr Double actualWaiverNonGaap;
        attr Double credibilityOverrideAmt;
        attr String description;
        attr Double estimatedManualClaimCount;
        val type.ExperienceType expSetType;
        attr Calendar lastEnrollmentDate;
        attr Boolean pendingClmInd;
        attr Calendar priorCarrEffDate;
        unique ordered val type.ProductType[*] productsIncludedInExperience;
        attr Double uwPoolingCharge;
        val type.ExperienceSetStatus status;
        @GenModel(documentation="Attribute that indicates if the instance is consistent. If it
        is not consistent means that it must be updated.")
        attr Boolean isConsistent = true;

        // Parent relations
        @ParentBO
        ref Illustration#experienceBOs illustrationBO;
        @ParentBO
        ref Contract#experienceBOs contractBO;

        // Children single

        // Children multiple
        @ChildBO
        val ExperiencePeriod[*]#experienceBO experiencePeriodBOs;
        @ChildBO
        val RateComponent[*]#experienceBO rateComponentBOs;
        @ChildBO
        val Property[*]#experienceBO propertyBOs;
        @ChildBO
```

```
        val RateComponent[*]#experienceBO rateComponentBOs;
        @ChildBO
        val Property[*]#experienceBO propertyBOs;
        @ChildBO
        val PropertyGroup[*]#experienceBO propertyGroupBOs;
}

class Role extends BaseBO {
        unsettable attr String boName = "RoleBO";
        @IdField
        val type.ParticipantType participantType;
        @IdField
        attr Integer order;
        val type.RoleType roleType;
        attr Boolean isSelected;
        attr String groupConstraintType;
        val type.GroupConstraintOperationType groupConstraintOperationType;
        attr Boolean policyHolderSameAsInsured;

        // Parent relations
        @ParentBO
        ref Illustration#roleBOs illustrationBO;
        @ParentBO
        ref Contract#roleBOs contractBO;
        @ParentBO
        ref Coverage#roleBOs coverageBO;
        @ParentBO
        ref PackageLife#roleBOs packageBO;

        // Children single

        // Children multiple
        @ChildBO
        val Party[*]#roleBO partyBOs;
        @ChildBO
        val Property[*]#roleBO propertyBOs;
        @ChildBO
        val PremiumComponent[*]#roleBO premiumComponentBOs;
}


class User extends BaseBO {
        unsettable attr String boName = "UserBO";
        val type.ReferralLevel referralLevel;
        attr Integer authorityLevel;
        attr String groupConstraintType;
        val type.GroupConstraintOperationType groupConstraintOperationType;

        // Parent relations
        @ParentBO
        ref Contract#userBO contractBO;
        @ParentBO
        ref Individual#userBO individualBO;

        // Children single

        // Children multiple
}
```

```
class DocumentsAggregator extends BaseBO {
	unsettable attr String boName = "DocumentsAggregatorBO";
	@IdField
	attr String documentsAggregatorType;
	attr String groupConstraintType;

	// Parent relations
	@ParentBO
	ref Contract#documentsAggregatorBO contractBO;
	@ParentBO
	ref Package#documentsAggregatorBO packageBO;

	// Children single

	// Children multiple
	@ChildBO
	val Property[*]#documentsAggregatorBO propertyBOs;
	@ChildBO
	val PropertyGroup[*]#documentsAggregatorBO propertyGroupBOs;
	@ChildBO
	val Document[*]#documentsAggregatorBO documentBOs;
	@ChildBO
	val DocumentGroup[*]#documentsAggregatorBO documentGroupBOs;
}

class Adjustment extends BaseBO {
	unsettable attr String boName = "AdjustmentBO";
	attr Double adjustmentAmount;
	attr String adjustmentType;
	attr Boolean isSelected;
	attr String adjustmentFactor;
	val type.AdjustmentProperty adjustmentMethod;
	attr Double adjustmentFormula;
	val type.CalcType rationale;
	attr String adjustmentComment;
	attr String groupConstraintType;
	val type.GroupConstraintOperationType groupConstraintOperationType;

	// Parent relations
	@ParentBO
	ref Contract#adjustmentBOs contractBO;
	@ParentBO
	ref Coverage#adjustmentBOs coverageBO;
	@ParentBO
	ref PackageLife#adjustmentBOs packageBO;

	// Children single

	// Children multiple
}

class PremiumComponent extends BaseBO {
	unsettable attr String boName = "PremiumComponentBO";
	@IdField
	val type.PremiumComponentType premiumComponentType;
	val type.InputOutputType inputOutput;
	attr Boolean isSelected;
	attr String groupConstraintType;
	val type.GroupConstraintOperationType groupConstraintOperationType;
```

```
        // Parent relations
        @ParentBO
        ref Contract#premiumComponentBOs contractBO;
        @ParentBO
        ref Coverage#premiumComponentBOs coverageBO;
        @ParentBO
        ref PackageLife#premiumComponentBOs packageBO;
        @ParentBO
        ref Role#premiumComponentBOs roleBO;

        // Children single

        // Children multiple
        @ChildBO
        val PremiumDetail[*]#premiumComponentBO premiumDetailBOs;
}

class RateComponent extends BaseBO {
        unsettable attr String boName = "RateComponentBO";
        @IdField
        val type.RateComponentType rateComponentType;
        @IdField
        val type.RateComponentCategory rateComponentCategory;
        @IdField
        val type.RateComponentClass rateComponentClass;
        @IdField
        attr RateComponentLevel level;
        @IdField
        attr String contextId;
        @GenModel(documentation="Indicates which was the original type of the rate component.
        This is necessary since sometimes we map different PM BOs to the same rate component.")
        @IdField
        attr String rateComponentOriginalType;
        @IdField
        val type.InputOutputType inputOutput;
        attr Boolean isSelected;
        attr String rateComponentName;
        attr String groupConstraintType;
        val type.GroupConstraintOperationType groupConstraintOperationType;

        // Parent relations
        @ParentBO
        ref Contract#rateComponentBOs contractBO;
        @ParentBO
        ref Grouping#rateComponentBOs groupingBO;
        @ParentBO
        ref GroupingObject#rateComponentBOs groupingObjectBO;
        @ParentBO
        ref Coverage#rateComponentBOs coverageBO;
        @ParentBO
        ref EligibilityClass#rateComponentBOs eligibilityClassBO;
        @ParentBO
        ref Experience#rateComponentBOs experienceBO;
        @ParentBO
        ref ExperiencePeriod#rateComponentBOs experiencePeriodBO;
        @ParentBO
        ref PackageLife#rateComponentBOs packageBO;
        @ParentBO
```

```
        ref PlanDesign#rateComponentBOs planDesignBO;
        @ParentBO
        ref PropertyGroup#rateComponentBOs propertyGroupBO;

        // Children single

        // Children multiple
        @ChildBO
        val RateDetail[*]#rateComponentBO rateDetailBOs;
        @ChildBO
        val Property[*]#rateComponentBO propertyBOs;
        @ChildBO
        val PropertyGroup[*]#rateComponentBO propertyGroupBOs;
}

class Authorization extends BaseBO {
        unsettable attr String boName = "AuthorizationBO";
        @IdField
        attr String authorizationId;
        attr String state;

        // Parent relations
        @ParentBO
        ref Contract#authorizationBOs contractBO;
        @ParentBO
        ref Package#authorizationBOs packageBO;

        // Children single

        // Children multiple
        @ChildBO
        val Property[*]#authorizationBO propertyBOs;
}

class EligibilityClass extends BaseBO {
        unsettable attr String boName = "EligibilityClassBO";
        @IdField
        val type.EligibilityClassType eligibilityClassType;

        // Parent relations
        @ParentBO
        ref Contract#eligibilityClassBOs contractBO;

        // Children single
        @ChildBO
        val Census#eligibilityClassBO censusBO;

        // Children multiple
        @ChildBO
        val PlanDesign[*]#eligibilityClassBO planDesignBOs;
        @ChildBO
        val CensusClass[*]#eligibilityClassBO censusClassBOs;
        @ChildBO
        val Property[*]#eligibilityClassBO propertyBOs;
        @ChildBO
        val PropertyGroup[*]#eligibilityClassBO propertyGroupBOs;
        @ChildBO
        val RateComponent[*]#eligibilityClassBO rateComponentBOs;
}
```

```
class Grouping extends BaseBO {
	unsettable attr String boName = "GroupingBO";
	attr Integer groupingIndex;
	@IdField
	attr Integer groupingId;
	attr Integer planId;
	val type.ProductType productType;
	@GenModel(documentation="This attribute will not be stored in database. See GIP-2963")
	attr String productDescription;

	// Parent relations
	@ParentBO
	ref Package#groupingBOs packageBO;

	// Children single

	// Children multiple
	@ChildBO
	val RateComponent[*]#groupingBO rateComponentBOs;
	@ChildBO
	val Property[*]#groupingBO propertyBOs;
	@ChildBO
	val PropertyGroup[*]#groupingBO propertyGroupBOs;
	@ChildBO
	val GroupingObject[*]#groupingBO groupingObjectBOs;
}

class PropertyGroupHolder extends BaseBO {
	unsettable attr String boName = "PropertyGroupHolderBO";
	@IdField
	attr Integer planId;

	// Parent relations
	@ParentBO
	ref Package#propertyGroupHolderBOs packageBO;

	// Children single

	// Children multiple
	@ChildBO
	val PropertyGroup[*]#propertyGroupHolderBO propertyGroupBOs;
}

class Broker extends BaseBO {
	unsettable attr String boName = "BrokerBO";
	attr String brokerName;
	attr Double commissionPercentage;

	// Parent relations
	@ParentBO
	ref ProductFamily#brokerBOs productFamilyBO;

	// Children single

	// Children multiple
}

class ExperiencePeriod extends BaseBO {
```

```
        unsettable attr String boName = "ExperiencePeriodBO";
        @IdField
        attr Integer experiencePeriodNum;
        val type.ExperiencePeriodType expPeriodType;
        attr Integer expPeriodId;
        attr String carrierName;
        attr Calendar startDate;
        attr Calendar endDate;
        attr Calendar asOfDate;
        attr Double actualWaiverNonGAAP;
        attr Double conversionCharges;
        attr Integer coveredLives;
        attr Double coveredVolume;
        attr Double duePremium;
        attr Double historicalInforceRate;
        attr Integer numOfPaidClaims;
        attr Double paidClaims;
        attr Double pooledClaims;

        // Parent relations
        @ParentBO
        ref Experience#experiencePeriodBOs experienceBO;

        // Children single

        // Children multiple
        @ChildBO
        val Property[*]#experiencePeriodBO propertyBOs;
        @ChildBO
        val ExperiencePeriodCoverage[*]#experiencePeriodBO experiencePeriodCoverageBOs;
        @ChildBO
        val RateComponent[*]#experiencePeriodBO rateComponentBOs;
}

class Coverage extends BaseBO {
        unsettable attr String boName = "CoverageBO";
        attr String coverageId;
        attr String coverageName;
        @IdField
        val type.CoverageType coverageType;
        val type.CoverageCategory coverageCategory;
        attr Boolean isSelected;
        attr Double sumAssured;
        attr Calendar coverageEffectiveDate;
        attr Boolean baseComponentFlag;
        val type.ActionType coverageAction;
        attr Double excess;
        attr Double max;
        attr Double min;
        attr String groupConstraintType;
        val type.GroupConstraintOperationType groupConstraintOperationType;

        // Parent relations
        @ParentBO
        ref Coverage#coverageBOs coverageBO;
        @ParentBO
        ref PackageLife#coverageBOs packageBO;
        @ParentBO
        ref PlanDesign#coverageBOs planDesignBO;
```

```
        // Children single

        // Children multiple
        @ChildBO
        val Adjustment[*]#coverageBO adjustmentBOs;
        @ChildBO
        val Coverage[*]#coverageBO coverageBOs;
        @ChildBO
        val PremiumComponent[*]#coverageBO premiumComponentBOs;
        @ChildBO
        val Property[*]#coverageBO propertyBOs;
        @ChildBO
        val PropertyGroup[*]#coverageBO propertyGroupBOs;
        @ChildBO
        val RateComponent[*]#coverageBO rateComponentBOs;
        @ChildBO
        val Role[*]#coverageBO roleBOs;
}

class Party extends BaseBO {
        unsettable attr String boName = "PartyBO";
        attr String externPartyId;
        @IdField
        attr String partyId;
        attr String groupConstraintType;
        val type.GroupConstraintOperationType groupConstraintOperationType;
        attr Boolean isOrganization;

        // Parent relations
        @ParentBO
        ref Role#partyBOs roleBO;

        // Children single
        @ChildBO
        val Individual#partyBO individualBO;
        @ChildBO
        val Organization#partyBO organizationBO;

        // Children multiple
        @ChildBO
        val Address[*]#partyBO addressBOs;
        @ChildBO
        val Email[*]#partyBO emailBOs;
        @ChildBO
        val Property[*]#partyBO propertyBOs;
}

class Individual extends BaseBO {
        unsettable attr String boName = "IndividualBO";
        attr String firstName;
        attr String lastName;
        attr String fullName;
        attr String fullNameAndTitle;
        attr String middleInitial;
        attr String middleName;
        val type.NamePrefix namePrefix;
        val type.NameSuffix nameSuffix;
        val type.Title title;
        attr Boolean company;
        attr String status;
```

```
        val type.Gender sex;
        attr Integer age;
        attr Calendar dateOfBirth;
        attr Calendar dateOfDeath;
        attr Integer dobDay;
        attr Integer dobMonth;
        attr Integer dobYear;
        attr String groupConstraintType;
        val type.GroupConstraintOperationType groupConstraintOperationType;

        // Parent relations
        @ParentBO
        ref Party#individualBO partyBO;

        // Children single
        @ChildBO
        val InsuredPerson#individualBO insuredPersonBO;
        @ChildBO
        val Producer#individualBO producerBO;
        @ChildBO
        val User#individualBO userBO;

        // Children multiple
        @ChildBO
        val Property[*]#individualBO propertyBOs;
}

class Document extends BaseBO {
        unsettable attr String boName = "DocumentBO";
        @IdField
        val type.DocumentType documentType;
        attr String documentName;
        attr Integer documentId;
        val type.DocumentCategory documentCategory;
        attr Boolean isSelected;
        val type.YesNoType seenByUser;
        val type.YesNoType changedByUser;
        attr String groupConstraintType;
        val type.GroupConstraintOperationType groupConstraintOperationType;

        // Parent relations
        @ParentBO
        ref DocumentsAggregator#documentBOs documentsAggregatorBO;
        @ParentBO
        ref DocumentGroup#documentBOs documentGroupBO;

        // Children single

        // Children multiple
        @ChildBO
        val Property[*]#documentBO propertyBOs;
        @ChildBO
        val PropertyGroup[*]#documentBO propertyGroupBOs;
}

class PremiumDetail extends BaseBO {
        unsettable attr String boName = "PremiumDetailBO";
        @IdField
        val type.PremiumDetailType premiumDetailType;
```

```
        attr Double amount;
        attr Double percentage;
        val type.UnitOfMeasurement unitOfMeasurement;
        attr Boolean isSelected;
        attr String groupConstraintType;
        val type.GroupConstraintOperationType groupConstraintOperationType;

        // Parent relations
        @ParentBO
        ref PremiumComponent#premiumDetailBOs premiumComponentBO;

        // Children single
        @ChildBO
        val PremiumDetailVector#premiumDetailBO premiumDetailVectorBO;

        // Children multiple
}

class GroupingObject extends BaseBO {
        unsettable attr String boName = "GroupingObjectBO";
        attr Integer objectId;
        @IdField
        attr Integer groupingObjectId;
        attr Integer groupingContextId;
        val type.CoverageType objectType;

        // Parent relations
        @ParentBO
        ref Grouping#groupingObjectBOs groupingBO;

        // Children single

        // Children multiple
        @ChildBO
        val RateComponent[*]#groupingObjectBO rateComponentBOs;
        @ChildBO
        val Property[*]#groupingObjectBO propertyBOs;
        @ChildBO
        val PropertyGroup[*]#groupingObjectBO propertyGroupBOs;
}

class PlanDesign extends BaseBO {
        unsettable attr String boName = "PlanDesignBO";
        @IdField
        val type.PlanDesignType planDesignType;
        @IdField
        attr int order;
        attr String comments;
        attr String experienceSet;
        unique ordered attr String[*] censusClasses;

        // Parent relations
        @ParentBO
        ref EligibilityClass#planDesignBOs eligibilityClassBO;

        // Children single

        // Children multiple
        @ChildBO
```

```
        val Property[*]#planDesignBO propertyBOs;
        @ChildBO
        val PropertyGroup[*]#planDesignBO propertyGroupBOs;
        @ChildBO
        val Coverage[*]#planDesignBO coverageBOs;
        @ChildBO
        val RateComponent[*]#planDesignBO rateComponentBOs;
}

class RateDetail extends BaseBO {
        unsettable attr String boName = "RateDetailBO";
        @IdField
        val type.RateDetailType rateDetailType;
        val type.RateDetailCategory category;
        attr Double rateAmount;
        attr String rateValue;
        attr Double percentage;
        attr Boolean boolValue;
        attr Integer intValue;
        attr Integer calcOrder;
        attr Boolean isSelected;
        attr String groupConstraintType;
        val type.GroupConstraintOperationType groupConstraintOperationType;
        attr String rateDetailName;

        // Parent relations
        @ParentBO
        ref RateComponent#rateDetailBOs rateComponentBO;
        @ParentBO
        ref RateComponentMatrix#rateDetailBOs rateComponentMatrixBO;
        @ParentBO
        ref RateComponentVector#rateDetailBOs rateComponentVectorBO;

        // Children single
        @ChildBO
        val RateComponentVector#rateDetailBO rateDetailVector;

        // Children multiple
}

class Census extends BaseBO {
        unsettable attr String boName = "CensusBO";
        @IdField
        attr String Id;

        // Parent relations
        @ParentBO
        ref EligibilityClass#censusBO eligibilityClassBO;

        // Children single

        // Children multiple
        @ChildBO
        val Property[*]#censusBO propertyBOs;
        @ChildBO
        val PropertyGroup[*]#censusBO propertyGroupBOs;
}

class CensusClass extends BaseBO {
```

```
        unsettable attr String boName = "CensusClassBO";
        attr Integer censClassId;
        attr String censClassName;
        attr Double eligAvgSalAmt;
        attr Double insuredMalePct;
        attr Double insuredAvgSalAmt;
        attr Double insuredAvgAgeNum;
        attr Double eligAvgAgeNum;
        attr Double insuredVol;
        attr Integer insuredLives;
        attr Double eligMalePct;
        attr Integer eligLives;
        attr String sicCode;

        // Parent relations
        @ParentBO
        ref EligibilityClass#censusClassBOs eligibilityClassBO;

        // Children single

        // Children multiple
}

class ExperiencePeriodCoverage extends BaseBO {
        unsettable attr String boName = "ExperiencePeriodCoverageBO";
        @IdField
        attr String Id;

        // Parent relations
        @ParentBO
        ref ExperiencePeriod#experiencePeriodCoverageBOs experiencePeriodBO;

        // Children single

        // Children multiple
}

class Organization extends BaseBO {
        unsettable attr String boName = "OrganizationBO";
        attr String name;
        attr String contactPerson;
        val type.LegalStructureType legalStructure;
        attr String groupConstraintType;
        val type.GroupConstraintOperationType groupConstraintOperationType;

        // Parent relations
        @ParentBO
        ref Party#organizationBO partyBO;
        @ParentBO
        ref ProducerOrganization#organizationBO producerOrganizationBO;

        // Children single
        @ChildBO
        val MultiLifePartner#organizationBO multiLifePartnerBO;

        // Children multiple
        @ChildBO
        val Property[*]#organizationBO propertyBOs;
}
```

```
class Address extends BaseBO {
	unsettable attr String boName = "AddressBO";
	val type.AddrType addressType;
	val type.AddrUsage addressUsage;
	attr String addressLine1;
	attr String addressLine2;
	attr String city;
	attr String externAddressId;
	attr String postcode;
	val type.StateName state;
	attr String suburb;
	attr String town;
	attr String groupConstraintType;
	val type.GroupConstraintOperationType groupConstraintOperationType;

	// Parent relations
	@ParentBO
	ref Party#addressBOs partyBO;

	// Children single

	// Children multiple
	@ChildBO
	val Property[*]#addressBO propertyBOs;
}

class Email extends BaseBO {
	unsettable attr String boName = "EmailBO";
	attr String usage;
	attr String emailAddress;
	attr String groupConstraintType;
	val type.GroupConstraintOperationType groupConstraintOperationType;

	// Parent relations
	@ParentBO
	ref Party#emailBOs partyBO;

	// Children single

	// Children multiple
}

class InsuredPerson extends BaseBO {
	unsettable attr String boName = "InsuredPersonBO";
	val type.PartyType type;
	attr String externEmployerId;
	attr String externPartyId;
	attr Boolean businessOwner;
	attr Integer insuranceAge;
	val type.YesNoType smokerStatus;
	attr String occupation;
	val type.OccClass occupationClass;
	attr Integer retirementAge;
	attr Boolean selfEmployed;
	attr Boolean selfEmployedMoreThan3Years;
	val type.Gender sex;
	val type.ThirdPartySystem source;
	attr String groupConstraintType;
```

```
        val type.GroupConstraintOperationType groupConstraintOperationType;

        // Parent relations
        @ParentBO
        ref Individual#insuredPersonBO individualBO;

        // Children single

        // Children multiple
        @ChildBO
        val Income[*]#insuredPersonBO incomeBOs;
        @ChildBO
        val Phone[*]#insuredPersonBO phoneBOs;
        @ChildBO
        val Property[*]#insuredPersonBO propertyBOs;
}

class Producer extends BaseBO {
        unsettable attr String boName = "ProducerBO";
        val type.PartyType type;
        @IdField
        attr String externPartyId;
        val type.ProductType defaultProduct;
        val type.StatusType defaultContractStatus;
        attr String licenseNo;
        attr String officeId;
        attr String officeList;
        attr String groupConstraintType;
        val type.GroupConstraintOperationType groupConstraintOperationType;

        // Parent relations
        @ParentBO
        ref Contract#producerBO contractBO;
        @ParentBO
        ref Illustration#producerBO illustrationBO;
        @ParentBO
        ref Individual#producerBO individualBO;

        // Children single

        // Children multiple
        @ChildBO
        val Phone[*]#producerBO phoneBOs;
        @ChildBO
        val ProducerOrganization[*]#producerBO producerOrganizationBOs;
}

class DocumentGroup extends BaseBO {
        unsettable attr String boName = "DocumentGroupBO";
        @IdField
        attr String documentGroupType;

        // Parent relations
        @ParentBO
        ref DocumentsAggregator#documentGroupBOs documentsAggregatorBO;

        // Children single

        // Children multiple
```

```
        @ChildBO
        val Document[*]#documentGroupBO documentBOs;
        @ChildBO
        val Property[*]#documentGroupBO propertyBOs;
        @ChildBO
        val PropertyGroup[*]#documentGroupBO propertyGroupBOs;
}

class PremiumDetailVector extends BaseBO {
        unsettable attr String boName = "PremiumDetailVector";
        attr Integer lengthOfPeriod;
        attr String groupConstraintType;
        val type.GroupConstraintOperationType groupConstraintOperationType;

        // Parent relations
        @ParentBO
        ref PremiumDetail#premiumDetailVectorBO premiumDetailBO;

        // Children single

        // Children multiple
        @ChildBO
        val PremiumDetailVectorElement[*]#premiumDetailVectorBO premiumDetailVectorElements;
}

class RateComponentMatrix extends BaseBO {
        unsettable attr String boName = "RateComponentMatrixBO";
        @IdField
        val type.RateComponentType rateComponentType;
        @IdField
        attr Integer x;
        @IdField
        attr Integer y;
        val type.InputOutputType inputOutput;
        attr Boolean isSelected;

        // Parent relations
        @ParentBO
        ref PropertyGroup#rateComponentMatrixBOs propertyGroupBO;

        // Children single

        // Children multiple
        @ChildBO
        val RateDetail[*]#rateComponentMatrixBO rateDetailBOs;
}

class Property extends BaseBO {
        unsettable attr String boName = "PropertyBO";
        @IdField
        val type.PropertyType propertyType;
        val type.PropertyDataType propertyDataType;
        val type.PropertyCategory propertyCategory;
        val type.PropertyClass propertyClass;
        attr Double doubleValue;
        attr String stringValue;
        attr Calendar dateValue;
        attr Integer integerValue;
        attr Boolean booleanValue;
```

```
val type.PropertyKeyValue keyValue;
val type.UnitOfMeasurement unitOfMeasurement;
attr Boolean customProperty = false;
attr Boolean compositeProperty = false;
val type.PropertyType originalPropertyType;
attr Boolean isSelected;
val type.ActionType propertyAction;
attr String groupConstraintType;
val type.GroupConstraintOperationType groupConstraintOperationType;
@InternalBOMField
attr FunctionalState functionalState;

// Parent relations
@ParentBO
ref Address#propertyBOs addressBO;
@ParentBO
ref Contract#propertyBOs contractBO;
@ParentBO
ref Grouping#propertyBOs groupingBO;
@ParentBO
ref GroupingObject#propertyBOs groupingObjectBO;
@ParentBO
ref Census#propertyBOs censusBO;
@ParentBO
ref Coverage#propertyBOs coverageBO;
@ParentBO
ref Individual#propertyBOs individualBO;
@ParentBO
ref Illustration#propertyBOs illustrationBO;
@ParentBO
ref InsuredPerson#propertyBOs insuredPersonBO;
@ParentBO
ref PackageLife#propertyBOs packageLifeBO;
@ParentBO
ref Package#propertyBOs packageBO;
@ParentBO
ref ProductFamily#propertyBOs productFamilyBO;
@ParentBO
ref Role#propertyBOs roleBO;
@ParentBO
ref Party#propertyBOs partyBO;
@ParentBO
ref Organization#propertyBOs organizationBO;
@ParentBO
ref Experience#propertyBOs experienceBO;
@ParentBO
ref ExperiencePeriod#propertyBOs experiencePeriodBO;
@ParentBO
ref PropertyGroup#propertyBOs propertyGroupBO;
@ParentBO
ref EligibilityClass#propertyBOs eligibilityClassBO;
@ParentBO
ref PlanDesign#propertyBOs planDesignBO;
@ParentBO
ref RateComponent#propertyBOs rateComponentBO;
@ParentBO
ref DocumentsAggregator#propertyBOs documentsAggregatorBO;
@ParentBO
ref DocumentGroup#propertyBOs documentGroupBO;
@ParentBO
ref Document#propertyBOs documentBO;
```

```
        @ParentBO
        ref Authorization#propertyBOs authorizationBO;

        // Children single

        // Children multiple
}

class PropertyGroup extends BaseBO {
        unsettable attr String boName = "PropertyGroupBO";
        @IdField
        val type.PropertyGroupType propertyGroupType;
        val type.PropertyCategory propertyCategory;
        val type.PropertyClass propertyClass;
        @IdField
        attr int order;
        attr Boolean isSelected;

        // Parent relations
        @ParentBO
        ref PlanDesign#propertyGroupBOs planDesignBO;
        @ParentBO
        ref RateComponent#propertyGroupBOs rateComponentBO;
        @ParentBO
        ref Contract#propertyGroupBOs contractBO;
        @ParentBO
        ref Grouping#propertyGroupBOs groupingBO;
        @ParentBO
        ref GroupingObject#propertyGroupBOs groupingObjectBO;
        @ParentBO
        ref Census#propertyGroupBOs censusBO;
        @ParentBO
        ref Coverage#propertyGroupBOs coverageBO;
        @ParentBO
        ref EligibilityClass#propertyGroupBOs eligibilityClassBO;
        @ParentBO
        ref ExperiencePeriod#propertyGroupBOs experiencePeriodBO;
        @ParentBO
        ref Experience#propertyGroupBOs experienceBO;
        @ParentBO
        ref Package#propertyGroupBOs packageBO;
        @ParentBO
        ref PropertyGroup#propertyGroupBOs propertyGroupBO;
        @ParentBO
        ref PropertyGroupHolder#propertyGroupBOs propertyGroupHolderBO;
        @ParentBO
        ref DocumentsAggregator#propertyGroupBOs documentsAggregatorBO;
        @ParentBO
        ref DocumentGroup#propertyGroupBOs documentGroupBO;
        @ParentBO
        ref Document#propertyGroupBOs documentBO;

        // Children single

        // Children multiple
        @ChildBO
        val Property[*]#propertyGroupBO propertyBOs;
        @ChildBO
        val PropertyGroup[*]#propertyGroupBO propertyGroupBOs;
        @ChildBO
```

```
        val RateComponent[*]#propertyGroupBO rateComponentBOs;
        @ChildBO
        val RateComponentMatrix[*]#propertyGroupBO rateComponentMatrixBOs;
        @ChildBO
        val RateComponentVector[*]#propertyGroupBO rateComponentVectorBOs;
}

class BusinessTransaction extends BaseBO {
        unsettable attr String boName = "BusinessTransactionBO";
         attr Calendar txnEffectiveDate;
        attr Calendar txnEndDate;
        val type.TransactionType txnType;
        attr String groupConstraintType;
        val type.GroupConstraintOperationType groupConstraintOperationType;

        // Parent relations
        @ParentBO
        ref Contract#businessTransactionBO contractBO;

        // Children single

        // Children multiple
}

class RateComponentVector extends BaseBO {
        unsettable attr String boName = "RateComponentVectorBO";
        attr long parentPK;
        @IdField
        val type.RateComponentType rateComponentType;
        @IdField
        attr Integer x;
        val type.InputOutputType inputOutput;
        attr Boolean isSelected;

        // Parent relations
        @ParentBO
        ref PropertyGroup#rateComponentVectorBOs propertyGroupBO;

        // Children single

        // Children multiple
        @ChildBO
        val RateDetail[*]#rateComponentVectorBO rateDetailBOs;
}

class ProducerOrganization extends BaseBO {
        unsettable attr String boName = "ProducerOrganizationBO";
        attr String licenseNo;
        attr String groupConstraintType;
        val type.GroupConstraintOperationType groupConstraintOperationType;

        // Parent relations
        @ParentBO
        ref Producer#producerOrganizationBOs producerBO;

        // Children single
        @ChildBO
        val Organization#producerOrganizationBO organizationBO;
```

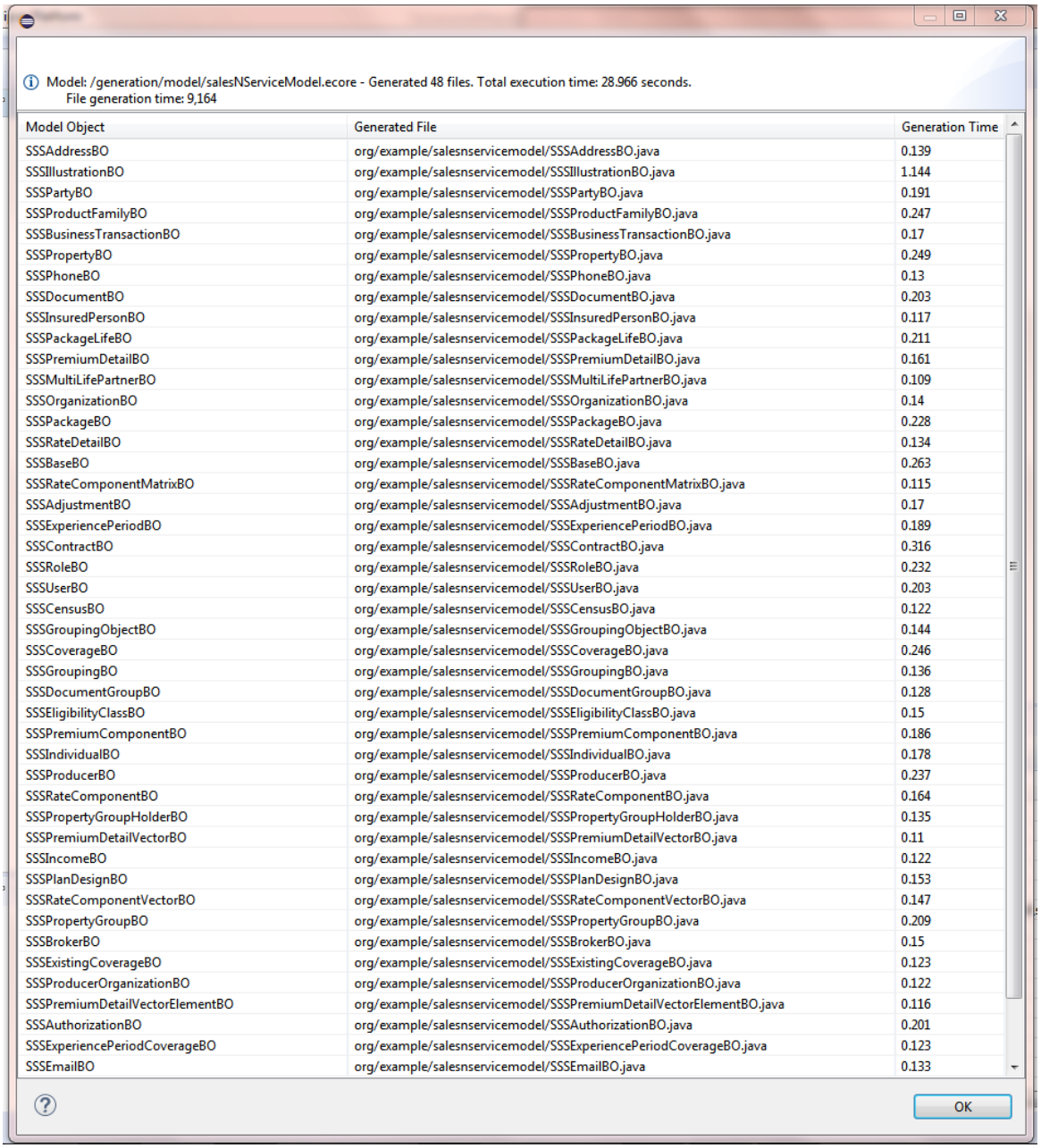
```
        // Children multiple
}

class MultiLifePartner extends BaseBO {
        unsettable attr String boName = "MultiLifePartnerBO";
        val type.PartyType type;
        attr String companyName;
        attr String contactPerson;
        attr String externPartID;
        attr String organizationForm;
        val type.ThirdPartySystem source;
        attr String taxIdNumber;
        attr String groupConstraintType;
        val type.GroupConstraintOperationType groupConstraintOperationType;

        // Parent relations
        @ParentBO
        ref Organization#multiLifePartnerBO organizationBO;

        // Children single

        // Children multiple
        @ChildBO
        val Phone[*]#multiLifePartnerBO phoneBOs;
}

class Income extends BaseBO {
        unsettable attr String boName = "IncomeBO";
        attr String incomeType;
        attr Double income;
        attr Boolean isSelected;
        attr String groupConstraintType;
        val type.GroupConstraintOperationType groupConstraintOperationType;


        // Parent relations
        @ParentBO
        ref InsuredPerson#incomeBOs insuredPersonBO;

        // Children single

        // Children multiple
}

class Phone extends BaseBO {
        unsettable attr String boName = "PhoneBO";
        val type.PhoneType type;
        val type.AddrUsage usage;
        attr String areaCode;
        attr String extension;
        attr String phoneNumber;
        attr String externCommunicationId;
        attr String first3Digits;
        attr String last4Digits;
        attr String groupConstraintType;
        val type.GroupConstraintOperationType groupConstraintOperationType;

        // Parent relations
        @ParentBO
```

```
        ref InsuredPerson#phoneBOs insuredPersonBO;
        @ParentBO
        ref Producer#phoneBOs producerBO;
        @ParentBO
        ref MultiLifePartner#phoneBOs multiLifePartnerBO;

        // Children single

        // Children multiple
}

class PremiumDetailVectorElement extends BaseBO {
        unsettable attr String boName = "PremiumDetailVectorElement";
        attr Integer index;
        attr Double amount;
        attr Double percentage;
        attr String groupConstraintType;
        val type.GroupConstraintOperationType groupConstraintOperationType;

        // Parent relations
        @ParentBO
        ref PremiumDetailVector#premiumDetailVectorElements premiumDetailVectorBO;

        // Children single

        // Children multiple
}

class PackageLife extends BaseBO {
        unsettable attr String boName = "PackageBO";
        @IdField
        attr String packageId;
        @IdField
        val type.PackageType packageType;
        attr Boolean isSelected;
        val type.ActionType packageAction;
        attr String groupConstraintType;
        val type.GroupConstraintOperationType groupConstraintOperationType;

        // Parent relations
        @ParentBO
        ref Contract#packageBOs contractBO;

        // Children single

        // Children multiple
        @ChildBO
        val Coverage[*]#packageBO coverageBOs;
        @ChildBO
        val Adjustment[*]#packageBO adjustmentBOs;
        @ChildBO
        val PremiumComponent[*]#packageBO premiumComponentBOs;
        @ChildBO
        val Property[*]#packageLifeBO propertyBOs;
        @ChildBO
        val RateComponent[*]#packageBO rateComponentBOs;
        @ChildBO
        val Role[*]#packageBO roleBOs;
        @ChildBO
```

```
        val ExistingCoverage[*]#packageBO existingCoverageBOs;
}

class ExistingCoverage extends BaseBO {
        unsettable attr String boName = "ExistingCoverageBO";
        val type.CoverageType existingCoverageType;
        attr String policyId;
        attr Double coverageAmount;
        attr String groupConstraintType;
        val type.GroupConstraintOperationType groupConstraintOperationType;

        // Parent relations
        @ParentBO
        ref PackageLife#existingCoverageBOs packageBO;

        // Children single

        // Children multiple
}

// Generic and abstract reusable classes
abstract class BaseBO {
        //unsettable attr String boName = "BaseBO";
        attr Long PK;
        attr String objectIdentifier;
        attr Integer minimumCardinality;
        attr Integer maximumCardinality;
        attr Boolean isDefinition = false;
        attr Boolean isChanged = false;
        @MandatoryField
        attr Boolean saved = false;
        attr Integer displayOrder;
        attr String boPath;
        val BO parentBO;
        attr String xmlId;
}

abstract interface BO {

}

// Custom DataTypes
datatype Calendar : java.util.Calendar;
datatype FunctionalState : com.fja.sss.core.model;

enum RateComponentLevel {
        DEFAULT;
        PACKAGEDPLAN;
        TEMPORARYBACKUP;
}
```

Código 34 – Representação textual Emfatic das entidades do novo modelo Ecore do S&S

Nova *dialog* de resultados da geração de código usando a ferramenta Texo



Model Object	Generated File	Generation Time
SSSAddressBO	org/example/salesnservicemodel/SSSAddressBO.java	0.139
SSSIllustrationBO	org/example/salesnservicemodel/SSSIllustrationBO.java	1.144
SSSPartyBO	org/example/salesnservicemodel/SSSPartyBO.java	0.191
SSSProductFamilyBO	org/example/salesnservicemodel/SSSProductFamilyBO.java	0.247
SSSBusinessTransactionBO	org/example/salesnservicemodel/SSSBusinessTransactionBO.java	0.17
SSSPropertyBO	org/example/salesnservicemodel/SSSPropertyBO.java	0.249
SSSPhoneBO	org/example/salesnservicemodel/SSSPhoneBO.java	0.13
SSSDocumentBO	org/example/salesnservicemodel/SSSDocumentBO.java	0.203
SSSInsuredPersonBO	org/example/salesnservicemodel/SSSInsuredPersonBO.java	0.117
SSSPackageLifeBO	org/example/salesnservicemodel/SSSPackageLifeBO.java	0.211
SSSPremiumDetailBO	org/example/salesnservicemodel/SSSPremiumDetailBO.java	0.161
SSSMultiLifePartnerBO	org/example/salesnservicemodel/SSSMultiLifePartnerBO.java	0.109
SSSOrganizationBO	org/example/salesnservicemodel/SSSOrganizationBO.java	0.14
SSSPackageBO	org/example/salesnservicemodel/SSSPackageBO.java	0.228
SSSRateDetailBO	org/example/salesnservicemodel/SSSRateDetailBO.java	0.134
SSSBaseBO	org/example/salesnservicemodel/SSSBaseBO.java	0.263
SSSRateComponentMatrixBO	org/example/salesnservicemodel/SSSRateComponentMatrixBO.java	0.115
SSSAdjustmentBO	org/example/salesnservicemodel/SSSAdjustmentBO.java	0.17
SSSExperiencePeriodBO	org/example/salesnservicemodel/SSSExperiencePeriodBO.java	0.189
SSSContractBO	org/example/salesnservicemodel/SSSContractBO.java	0.316
SSSRoleBO	org/example/salesnservicemodel/SSSRoleBO.java	0.232
SSSUserBO	org/example/salesnservicemodel/SSSUserBO.java	0.203
SSSCensusBO	org/example/salesnservicemodel/SSSCensusBO.java	0.122
SSSGroupingObjectBO	org/example/salesnservicemodel/SSSGroupingObjectBO.java	0.144
SSSCoverageBO	org/example/salesnservicemodel/SSSCoverageBO.java	0.246
SSSGroupingBO	org/example/salesnservicemodel/SSSGroupingBO.java	0.136
SSSDocumentGroupBO	org/example/salesnservicemodel/SSSDocumentGroupBO.java	0.128
SSSEligibilityClassBO	org/example/salesnservicemodel/SSSEligibilityClassBO.java	0.15
SSSPremiumComponentBO	org/example/salesnservicemodel/SSSPremiumComponentBO.java	0.186
SSSIndividualBO	org/example/salesnservicemodel/SSSIndividualBO.java	0.178
SSSProducerBO	org/example/salesnservicemodel/SSSProducerBO.java	0.237
SSSRateComponentBO	org/example/salesnservicemodel/SSSRateComponentBO.java	0.164
SSSPropertyGroupHolderBO	org/example/salesnservicemodel/SSSPropertyGroupHolderBO.java	0.135
SSSPremiumDetailVectorBO	org/example/salesnservicemodel/SSSPremiumDetailVectorBO.java	0.11
SSSIncomeBO	org/example/salesnservicemodel/SSSIncomeBO.java	0.122
SSSPlanDesignBO	org/example/salesnservicemodel/SSSPlanDesignBO.java	0.153
SSSRateComponentVectorBO	org/example/salesnservicemodel/SSSRateComponentVectorBO.java	0.147
SSSPropertyGroupBO	org/example/salesnservicemodel/SSSPropertyGroupBO.java	0.209
SSSBrokerBO	org/example/salesnservicemodel/SSSBrokerBO.java	0.15
SSSExistingCoverageBO	org/example/salesnservicemodel/SSSExistingCoverageBO.java	0.123
SSSProducerOrganizationBO	org/example/salesnservicemodel/SSSProducerOrganizationBO.java	0.122
SSSPremiumDetailVectorElementBO	org/example/salesnservicemodel/SSSPremiumDetailVectorElementBO.java	0.116
SSSAuthorizationBO	org/example/salesnservicemodel/SSSAuthorizationBO.java	0.201
SSSExperiencePeriodCoverageBO	org/example/salesnservicemodel/SSSExperiencePeriodCoverageBO.java	0.123
SSSEmailBO	org/example/salesnservicemodel/SSSEmailBO.java	0.133

Figura 65 – Nova dialog de resultados da geração de código na ferramenta Texo

```
1.  /**
2.  * Discovers the model and saves the result model if
3.  * {@link AbstractModelDiscoverer#isTargetSerializationChosen()
4.  * serialization is required}.
5.   */
6.  @Override
7.  public final void discoverElement(final IAdaptable iAdaptableSource, final IProgressMonitor moni
    tor) throws DiscoveryException {
8.      prepareDiscovery(iAdaptableSource);
9.      super.discoverElement(iAdaptableSource, monitor);
10.     if (monitor.isCanceled()) {
11.         return;
12.     }
13.     monitor.subTask("Loading discovered UML Model");
14.     ResourceSet resourceSet = new ResourceSetImpl();
15.     Model root = UML2Util.load(resourceSet, fUMLModelUri, UMLPackage.Literals.PACKAGE);
16.     if (monitor.isCanceled()) {
17.         return;
18.     }
19.     monitor.subTask("Discovering Ecore Model from discovered UML Model...");
20.     Resource ecoreModelResource = new Uml2EcoreUtils().extractEcoreModel(root, fSerializeTarget,
     fEcoreModelUri);
21.     if (monitor.isCanceled()) {
22.         return;
23.     }
24.     monitor.subTask("Refreshing resources");
25.     refreshSource(iAdaptableSource);
26.     if (!fKeepTemporaryDiscoveredModels) {
27.         cleanTemporaryModels();
28.     }
29.     IBrowserRegistry.INSTANCE.browseResource(ecoreModelResource);
30. }
31.
32. @Override
33. protected void basicDiscoverElement(final IAdaptable iAdaptableSource, final IProgressMonitor mo
    nitor) throws DiscoveryException {
34.         basicDiscoverElementForProject(fProject, monitor);
35. }
36. /**
37. * Discover required UML Model by executing following Discoverers:
38. * <ol>
39. * <li>DiscoverJavaModelFromProject</li>
40. * <li>DiscoverKDMModelFromJavaModel</li>
41. * <li>DiscoverUmlModelFromKdmModel</li>
42. * </ol>
43. * @param source IProject
44. * @param monitor IProgressMonitor
45. * @throws DiscoveryException
46. */
47. private void basicDiscoverElementForProject(final IProject source, final IProgressMonitor monito
    r) throws DiscoveryException {
48.     discoverJavaModelFromSource(source, monitor);
49.     if (monitor.isCanceled()) {
50.         return;
51.     }
52.     discoverKdmModelFromJavaModel(monitor);
53.     if (monitor.isCanceled()) {
54.         return;
55.     }
56.     discoverUmlModelFromKdmModel(monitor);
57. }
```

Código 35 - Excerto do código responsável pelo processo de obtenção do modelo Ecore

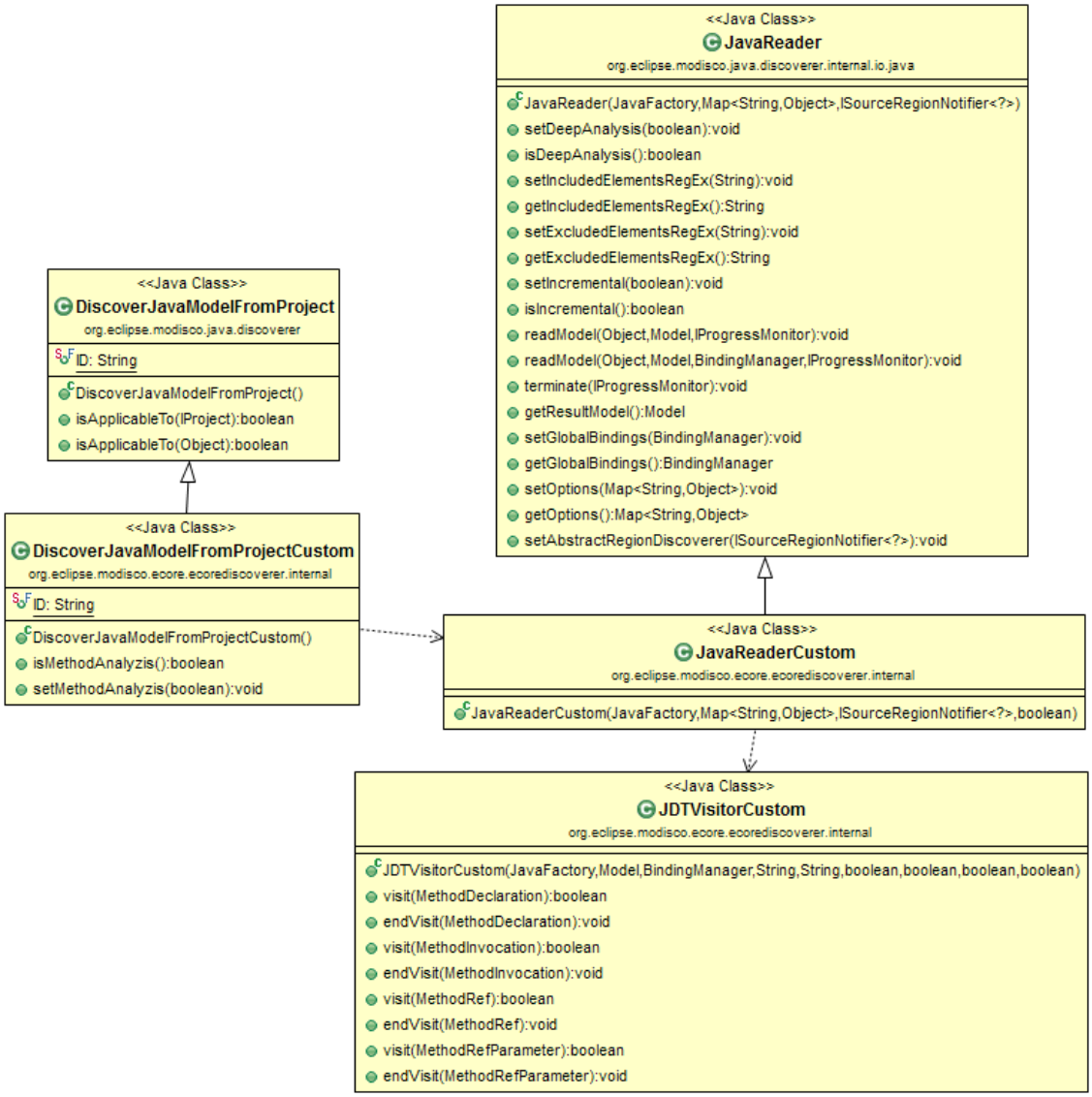


Figura 66 – Diagrama das classes utilizadas na análise do código Java pela classe *DiscoverJavaModelFromProjectCustom*

```
/**
* Custom implementation of an overridden JavaReader version to allow for method
* exclusion from discovered Java Model.
* Attention: This implementation of JavaReader does not supports
* abstractRegionDiscoverer usage. Method visitCompilationUnit is overridden and
* does not calls abstractRegionDiscoverer.notifySourceRegionVisited(...)
*
*/
@SuppressWarnings("restriction")
public class JavaReaderCustom extends JavaReader {
    private JavaFactory factory;
    private boolean fMethodAnalyzis = false;

public JavaReaderCustom(JavaFactory factory, Map<String, Object> options, ISourceRegionNotifier<
?> abstractRegionDiscoverer, boolean methodAnalyzis) {
        super(factory, options, abstractRegionDiscoverer);
        this.factory = factory;
        this.fMethodAnalyzis = methodAnalyzis;
    }

    protected void visitCompilationUnit(final Model resultModel1, final org.eclipse.jdt.core.dom
.CompilationUnit parsedCompilationUnit, final String filePath, final String fileContent) {
        JDTVisitorCustom jdtVisitor = new JDTVisitorCustom(this.factory, resultModel1, getGlobal
Bindings(), filePath, fileContent, getGlobalBindings().isIncrementalDiscovering(), this.isDeepAn
alysis(), false, fMethodAnalyzis);
        parsedCompilationUnit.accept(jdtVisitor);
    }
}
```

Código 36 – Código fonte da classe *JavaReaderCustom*

```
public class Uml2EcoreUtils {
    protected Map<String, String> initAllOptionsToProcess() {
        final Map<String, String> options = new HashMap<String, String>();
        options.put(UML2EcoreConverter.OPTION__ECORE_TAGGED_VALUES, UMLUtil.OPTION__PROCESS);
        options.put(UML2EcoreConverter.OPTION__REDEFINING_OPERATIONS, UMLUtil.OPTION__PROCESS);
        options.put(UML2EcoreConverter.OPTION__REDEFINING_PROPERTIES, UMLUtil.OPTION__PROCESS);
        options.put(UML2EcoreConverter.OPTION__SUBSETTING_PROPERTIES, UMLUtil.OPTION__PROCESS);
        options.put(UML2EcoreConverter.OPTION__UNION_PROPERTIES, UMLUtil.OPTION__PROCESS);
        options.put(UML2EcoreConverter.OPTION__DERIVED_FEATURES, UMLUtil.OPTION__PROCESS);
        options.put(UML2EcoreConverter.OPTION__DUPLICATE_OPERATIONS, UMLUtil.OPTION__PROCESS);
        options.put(UML2EcoreConverter.OPTION__DUPLICATE_OPERATION_INHERITANCE, UMLUtil.OPTION__
PROCESS);
        options.put(UML2EcoreConverter.OPTION__DUPLICATE_FEATURES, UMLUtil.OPTION__PROCESS);
        options.put(UML2EcoreConverter.OPTION__DUPLICATE_FEATURE_INHERITANCE, UMLUtil.OPTION__PR
OCESS);
        options.put(UML2EcoreConverter.OPTION__SUPER_CLASS_ORDER, UMLUtil.OPTION__PROCESS);
        options.put(UML2EcoreConverter.OPTION__ANNOTATION_DETAILS, UMLUtil.OPTION__PROCESS);
        options.put(UML2EcoreConverter.OPTION__INVARIANT_CONSTRAINTS, UMLUtil.OPTION__PROCESS);
        options.put(UML2EcoreConverter.OPTION__OPERATION_BODIES, UMLUtil.OPTION__PROCESS);
        options.put(UML2EcoreConverter.OPTION__COMMENTS, UMLUtil.OPTION__PROCESS);
        options.put(UML2EcoreConverter.OPTION__CAMEL_CASE_NAMES, UMLUtil.OPTION__IGNORE);
        return options;
    }
    public Resource extractEcoreModel(final org.eclipse.uml2.uml.Package profile, boolean serialize
Target, URI ecoreModelUri) {
        final Map<String, String> options = initAllOptionsToProcess();
        final Collection<EPackage> ecorePackages = UMLUtil.convertToEcore(profile, options, null,
null);
        final ResourceSet resourceSet = new ResourceSetImpl();
        final List<Resource> resources = new ArrayList<Resource> ();
        for (final EPackage ePackage: ecorePackages) {
            Resource resource = resourceSet.createResource(ecoreModelUri);
```

```
29.             resources.add(resource);
30.             resource.getContents().add(ePackage);
31.         }
32.         if (serializeTarget) {
33.             saveDiscoveredResources(profile, resources);
34.         }
35.         return resources.get(0);
36.     }
37.     private void saveDiscoveredResources(final org.eclipse.uml2.uml.Package profile, final List<Resou
    rce> resources) {
38.         for (final Resource currentResource: resources) {
39.             try {
40.                 currentResource.save(null);
41.             } catch (final Exception e) {
42.                 Logger.logError(e, "umlToEcore(" + profile.getClass() // $NON-NLS-1$
43.                     + ") not handled", Activator.getDefault());
44.             }
45.         }
46.     }
47. }
```

Código 37 – Código fonte da classe Uml2EcoreUtils

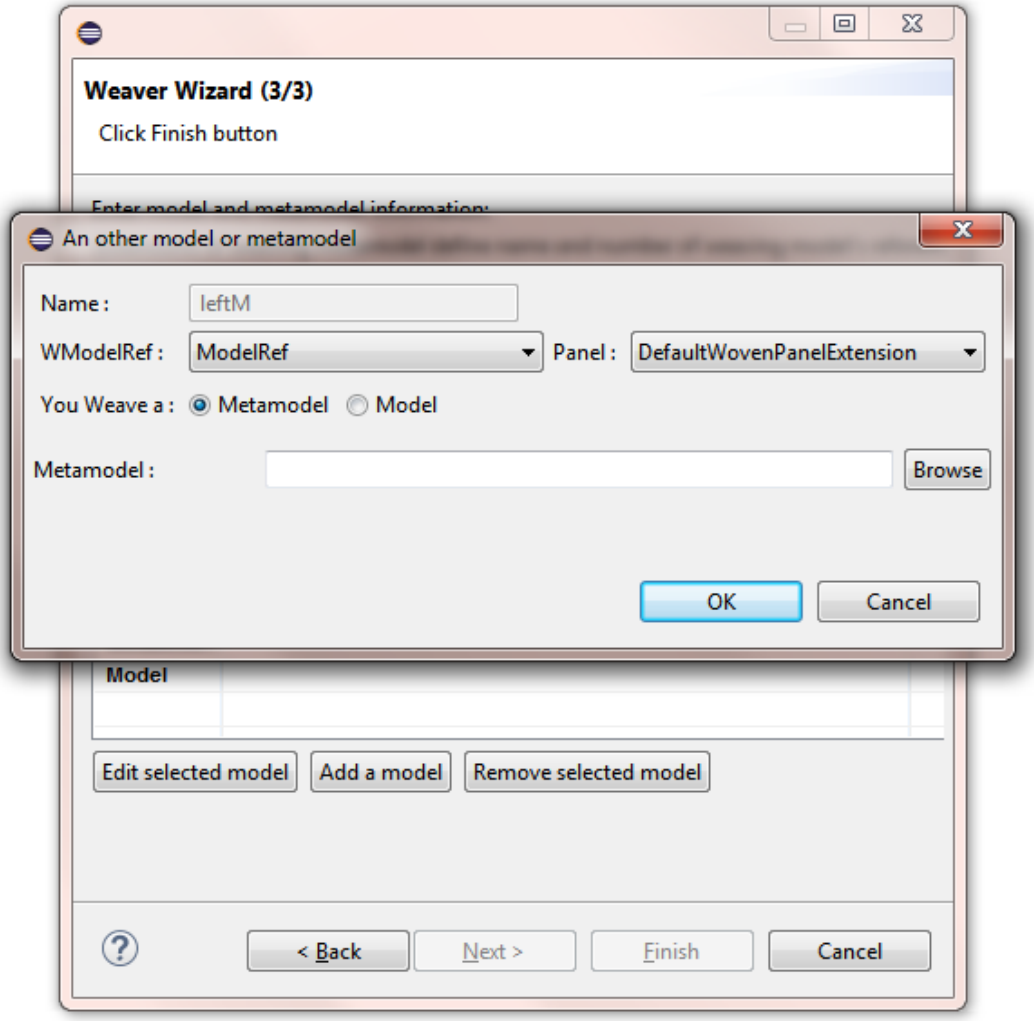


Figura 67 – Configuração de um novo modelo *weaving* passo 3 do wizard seleção do modelo

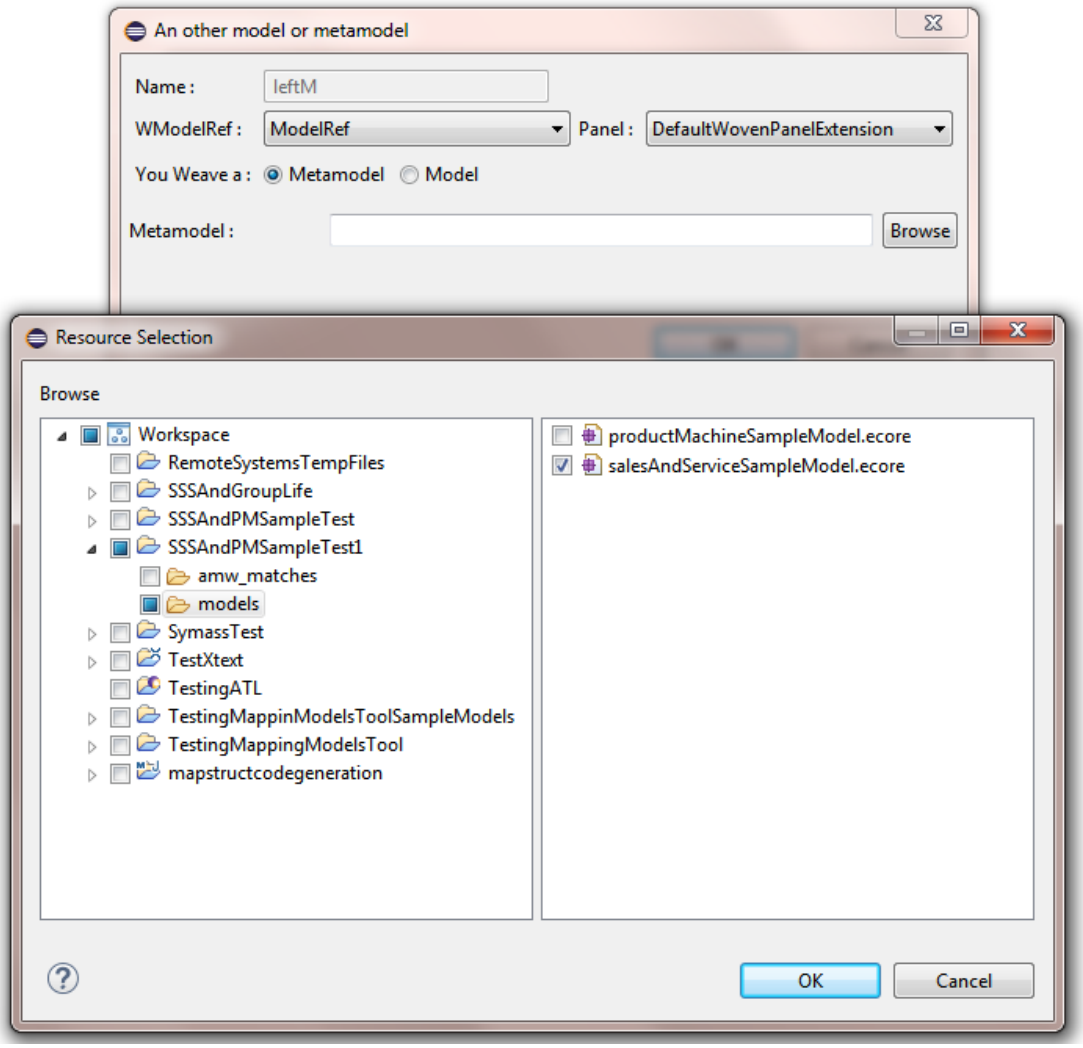


Figura 68 – Configuração de um novo modelo *weaving* passo 3 do wizard seleção do modelo no *workspace*

Código da transformação ATL para criação do modelo MapStructModel

```
-- @path AMW=/TestingATL/mwcore.ecore
-- @path MAPSTRUCT=/TestingATL/mapstructmodel.ecore

module AMWtoEcoreMapStruct;
create OUT: MAPSTRUCT from IN: AMW, left: MOF, right: MOF;

helper def: countRules: Integer =
        1;

helper def: moduleName: String =
        'T_' + if (not AMW!MatchModel.allInstancesFrom('IN') -> select(e | true) -> first().
                        name.oclIsUndefined() ) then
                AMW!MatchModel.allInstancesFrom('IN') -> select(e | true) -> first().name
        else
                'In2Out'
        endif;

helper def: leftModelRef: AMW!WModelRef =
        AMW!MatchModel.allInstancesFrom('IN') -> select(e | true) -> first().leftM;

helper def: rightModelRef: AMW!WModelRef =
        AMW!MatchModel.allInstancesFrom('IN') -> select(e | true) -> first().rightM;

helper context AMW!ElementEqual def: leftRightAreAbstract: Boolean =
        MOF!EClass.getInstanceById('left', self.left.element.ref).abstract and MOF!EClass.
                        getInstanceById('right', self.right.element.ref).abstract;

helper def: leftMM: String =
        MOF!EPackage.allInstancesFrom('left') -> select(e | e.name <> 'PrimitiveTypes') ->
                        first().name;

--'Inschema';
helper def: leftM: String =
        thisModule.leftMM + 'Model';

helper def: rightMM: String =
        MOF!EPackage.allInstancesFrom('right') -> select(e | e.name <> 'PrimitiveTypes') ->
                        first().name;

--'Outschema';
helper def: rightM: String =
        thisModule.rightMM + 'Model';

helper def: sourcePkg: String =
        thisModule.leftMM;

helper def: targetPkg: String =
        thisModule.rightMM;

helper def: getInstanceById(modelName: String, classifierID: String): MOF!EClassifier =
        MOF!EModelElement.getInstanceById(modelName, classifierID);

helper def: getLeftInstance(classifierID: String): MOF!EModelElement =
        thisModule.getInstanceById('left', classifierID);

helper def: getRightInstance(classifierID: String): MOF!EModelElement =
        thisModule.getInstanceById('right', classifierID);

-------------------------------------------------------------------------------
------------------- helpers to identify the origin of the input model elements
--- ----------------------------------------------------------------
helper def: leftPkgs: Sequence(MOF!EPackage) =
        MOF!EPackage.allInstancesFrom('left');

helper context MOF!EClassifier def: isLeft: Boolean =
        thisModule.leftPkgs -> exists(e | e = self.ePackage);

helper context MOF!EClassifier def: isRight: Boolean =
        not self.isLeft;
```

```
helper context MOF!EStructuralFeature def: isLeftsf: Boolean =
        self.eContainingClass.isLeft;

helper context MOF!EStructuralFeature def: isRightsf: Boolean =
        self.eContainingClass.isRight;

helper def: mappersByRelationClasses: Map(TupleType(leftClass: String, rightClass:
                String), MAPSTRUCT!Mapper) =
        Map{};

helper def: mappers: Map(TupleType(leftClass: AMW!ElementEqual), MAPSTRUCT!Mapper) =
        Map{};

helper context MOF!EModelElement def: getLink(right: MOF!EModelElement): AMW!WLink =
        AMW!Equivalent.allInstancesFrom('IN') -> select(e | e.left.element.ref = self.
                        __xmiID__ and right.__xmiID__ = e.right.element.ref) -> first();

helper context AMW!ReferenceEqual def: getReferencedLink(amw: AMW!ElementEqual):
                AMW!ElementEqual =
        self.left.getReferredElement().eReferenceType.getLink(self.right.getReferredElement().eReferenceTy
pe);

rule ElementEqual {
        from
                amw: AMW!ElementEqual
        using {
                childDebug: AMW!WLink = OclUndefined;
                leftModelRef: MOF!EClassifier = OclUndefined;
                leftModelRefId: String = OclUndefined;
                refers_link: AMW!WLink = OclUndefined;
        }
        to
                mapstructmapper: MAPSTRUCT!Mapper (
                        sourceClassName <- amw.left.name,
                        targetClassName <- amw.right.name,
                        mapperName <- amw.right.name + 'Mapper',
                        mappings <- amw.child,
                        referencedMappers <- amw.child -> select(e | e.
                                        oclIsTypeOf(AMW!ReferenceEqual)) -> collect(childReference |
                                        thisModule.resolveTemp(childReference.getReferencedLink(amw),
                                        'mapstructmapper'))
                )
        do {
                thisModule.mappersByRelationClasses <- thisModule.mappersByRelationClasses.
                                union(Map{(Tuple{leftClassID = amw.left.element.ref, rightClassID = amw.
                                right.element.ref},
                                                mapstructmapper)});
                for (ref_link in amw.child -> select(e | e.oclIsTypeOf(AMW!ReferenceEqual))) {
                                childDebug <- ref_link;

                                refers_link <- ref_link.getReferencedLink(amw);

                                thisModule.getLeftInstance(ref_link.left.element.ref).name.println();
                                ref_link.left.element.ref.println();
                                ref_link.left.element.ref.regexReplaceAll('/' + thisModule.
                                                getLeftInstance(ref_link.left.element.ref).name,
'').println();
                                ref_link.right.element.ref.println();
                                ref_link.right.element.ref.regexReplaceAll('/' + thisModule.
                                                getRightInstance(ref_link.right.element.ref).name,
'').println();
                        }
        }
}

rule AttributeEqual {
        from
                amw: AMW!AttributeEqual
```

```
to
                mapstructmapping: MAPSTRUCT!Mapping (
                        source <- amw.left.name,
                        target <- amw.right.name
                )
}

helper def: extractParentRefId(completeRefId: String, refId: String): String =
        completeRefId.regexReplaceAll('/' + refId, '');

helper def: getRelationLeftParentRefId(referenceEqual: AMW!ReferenceEqual): String =
        thisModule.extractParentRefId(referenceEqual.left.element.ref, thisModule.
                        getLeftInstance(referenceEqual.left.element.ref).name);

helper def: getRelationRightParentRefId(referenceEqual: AMW!ReferenceEqual): String =
        thisModule.extractParentRefId(referenceEqual.right.element.ref, thisModule.
                        getRightInstance(referenceEqual.right.element.ref).name);

--execute delayed actions
endpoint rule EndRule() {
        using {
                leftModelReferenceParentId: String = OclUndefined;
                rightModelReferenceParentId: String = OclUndefined;
                currentReferenceKey: TupleType(leftClassID: String, rightClassID: String) =
                                OclUndefined;
                currentReferencedMapper: MAPSTRUCT!Mapper = OclUndefined;
                elementMapper: MAPSTRUCT!Mapper = OclUndefined;
        }
        do {
                for (e in MAPSTRUCT!Mapper.allInstances()) {
                        elementMapper <- OclUndefined;
                }
                for(elementEqual in AMW!ElementEqual.allInstances()) {
                        elementMapper <- OclUndefined;
                        for(referenceEqual in elementEqual.child -> select(e | e.
                                        oclIsTypeOf(AMW!ReferenceEqual))) {
                                currentReferencedMapper <- OclUndefined;
                                leftModelReferenceParentId <- thisModule.
                                                getRelationLeftParentRefId(referenceEqual);
                                rightModelReferenceParentId <- thisModule.
                                                getRelationRightParentRefId(referenceEqual);
                                currentReferenceKey <- Tuple{leftClassID = leftModelReferenceParentId,
                                                rightClassID = rightModelReferenceParentId};
                                currentReferencedMapper <- thisModule.mappersByRelationClasses.
                                                get(currentReferenceKey);
                                if (not currentReferencedMapper.oclIsUndefined()) {
                                        elementMapper <- thisModule.mappers.get(elementEqual);
                                        if (not elementMapper.oclIsUndefined()) {

        elementMapper.referencedMappers.append(currentReferencedMapper);
                                        }
                                }
                        }
                }
        }
}
```

Código 38 – Código da transformação ATL para criação do modelo MapStructModel

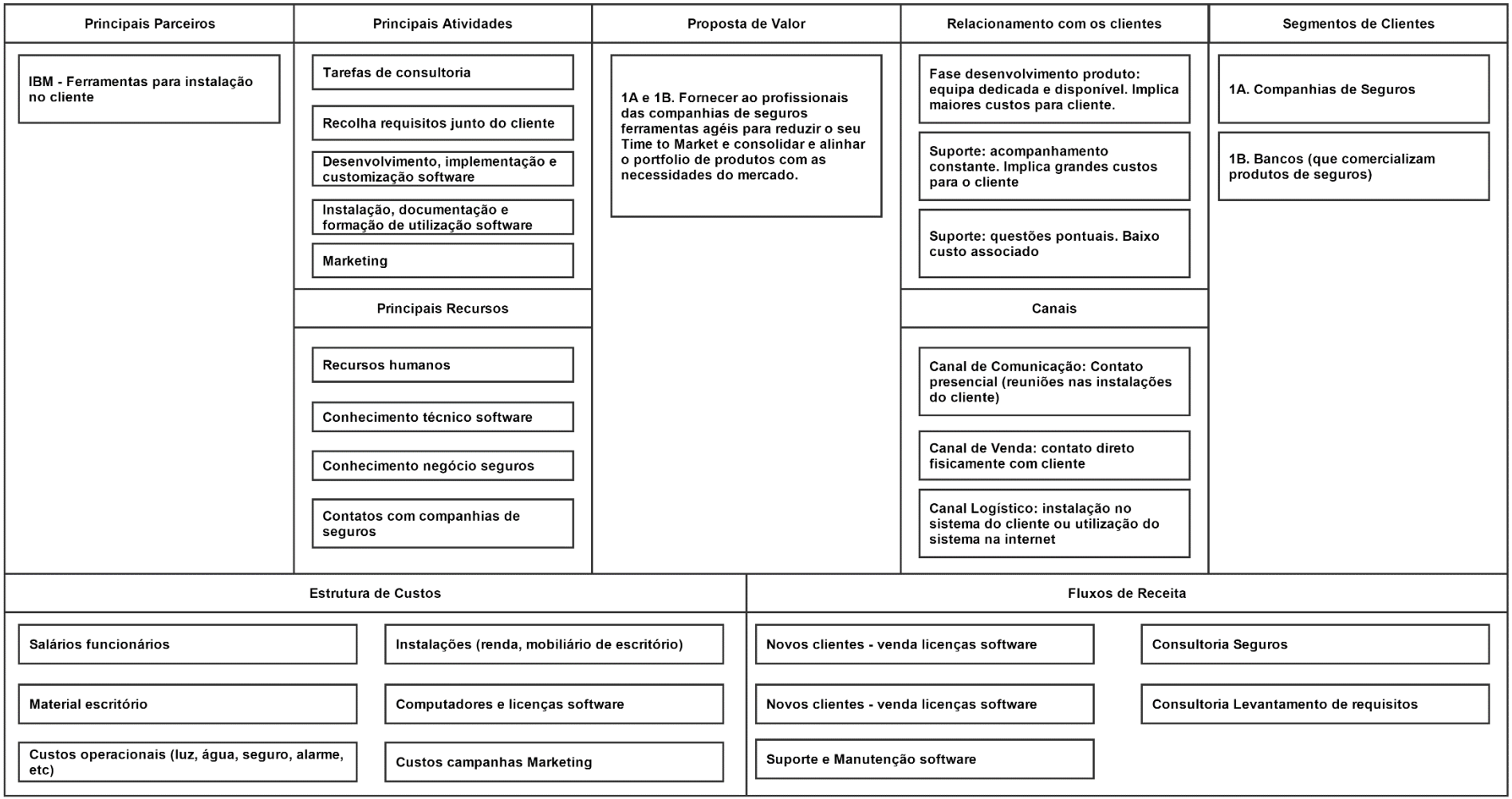


Figura 69 – Modelo de negócio de Canvas para a plataforma integrada Sales & Service e Product Machine