SENAI CIMATEC

PROGRAMA DE PÓS-GRADUAÇÃO EM MODELAGEM COMPUTACIONAL E TECNOLOGIA INDUSTRIAL

Doutorado em Modelagem Computacional e Tecnologia Industrial

Tese de Doutorado

MoPE - Modelo de Programação Baseado em Exemplos

Apresentada por: Uedson Santos Reis
Orientador: Marcelo Albano Moret Simões Gonçalves
Co-orientador: Eduardo Manuel de Freitas Jorge

Maio de 2018

Uedson Santos Reis

MoPE - Modelo de Programação Baseado em Exemplos

Tese de Doutorado apresentada ao Programa de Pós-graduação em Modelagem Computacional e Tecnologia Industrial, Curso de Doutorado em Modelagem Computacional e Tecnologia Industrial do SENAI CIMATEC, como requisito parcial para a obtenção do título de **Doutor em Modelagem Computacional e Tecnologia Industrial**.

Área de conhecimento: Interdisciplinar

Orientador: Marcelo Albano Moret Simões Gonçalves
SENAI CIMATEC

Co-orientador: Eduardo Manuel de Freitas Jorge
Universidade do Estado da Bahia

Salvador
SENAI CIMATEC
2018

Ficha catalográfica elaborada pela Biblioteca do Centro Universitário SENAI CIMATEC

R375m Reis, Uedson Santos

MoPE – Modelo de programação baseado em exemplos / Uedson Santos Reis. – Salvador, 2018.

106 f. : il.

Orientador: Prof. Dr. Marcelo Albano Moret Simões.

Tese (Doutorado em Modelagem Computacional e Tecnologia Industrial) – Programa de Pós-Graduação, Centro Universitário SENAI CIMATEC, Salvador, 2018.
Inclui referências.

1. Paradigmas de programação. 2. Modelagem de sistemas. 3. Lógica de programação. 4. Engenharia de software. I. Centro Universitário SENAI CIMATEC. II. Simões, Marcelo Albano Moret. III. Título.

CDD: 005.1

NDI - 01

Nota sobre o estilo do PPGMCTI

Esta tese de doutorado foi elaborada considerando as normas de estilo (i.e. estéticas e estruturais) propostas aprovadas pelo colegiado do Programa de Pós-graduação em Modelagem Computacional e Tecnologia Industrial e estão disponíveis em formato eletrônico (*download* na Página Web http://ead.fieb.org.br/portal_faculdades/dissertacoes-e-teses-mcti.html ou solicitação via e-mail à secretaria do programa) e em formato impresso somente para consulta.

Ressalta-se que o formato proposto considera diversos itens das normas da Associação Brasileira de Normas Técnicas (ABNT), entretanto opta-se, em alguns aspectos, seguir um estilo próprio elaborado e amadurecido pelos professores do programa de pós-graduação supracitado.

SENAI CIMATEC

Programa de Pós-graduação em Modelagem Computacional e Tecnologia Industrial

Doutorado em Modelagem Computacional e Tecnologia Industrial

A Banca Examinadora, constituída pelos professores abaixo listados, leram e recomendam a aprovação [com distinção] da Tese de Doutorado, intitulada "MoPE - Modelo de Programação Baseado em Exemplos", apresentada no dia (dia) de (mês) de (ano), como requisito parcial para a obtenção do título de **Doutor em Modelagem Computacional e Tecnologia Industrial**.

Orientador:

Prof. Dr. Marcelo Albano Moret Simões Gonçalves
SENAI CIMATEC

Co-orientador:

Prof. Dr. Eduardo Manuel de Freitas Jorge
Universidade do Estado da Bahia

Membro interno da Banca:

Prof. Dr. Hernane Borges de Barros Pereira
SENAI CIMATEC

Membro interno da Banca:

Prof. Dr. Roberto Luiz Souza Monteiro
SENAI CIMATEC

Membro externo da Banca:

Prof. Dr. Roberto Tadeu Raittz
UFPR - Universidade Federal do Paraná

Membro externo da Banca:

Prof. Dr. Roberto Pacheco
UFSC - Universidade Federal de Santa Catarina

Resumo

Os Paradigmas de Programação influenciam bastante o desenvolvimento e a maturidade de sistemas na área de Software. Eles surgiram para converter problemas reais em soluções computacionais. Após a proposta do Paradigma de Programação Orientado a Objetos, na década de 70, os modelos de programação pouco evoluíram, sob uma óptica paradigmática. Apesar dos benefícios alcançados com a Orientação a Objetos na Engenharia de Software, ainda existem problemas históricos não resolvidos. Por isso, este trabalho foca na problemática de como evoluir regras de negócio mantendo a documentação do projeto alinhada com sua codificação. A proposta desta pesquisa é o desenvolvimento de um Modelo de Programação, que permita a integração dos artefatos de modelagem e de implementação, baseado em um cenário com exemplos de referência. A metodologia Pesquisa-Ação foi utilizada para direcionar a pesquisa, e a construção do modelo foi realizada com base no método Interativo Incremental de desenvolvimento de software. O modelo proposto contribuiu para construção de um projeto de software mais coeso e fácil de manter, além de manter a sincronia entre os modelos de classes desenhados e a regras de negócio implementadas.
PALAVRAS-CHAVE: paradigmas de programação; modelagem de sistemas; lógica de programação; engenharia de software.

Abstract

Programming Paradigms greatly influence the development and maturity of systems in Software area. They have come to convert real problems into computational solutions. After the proposal of Object Oriented Programming Paradigm, in the decade of 70, the models of programming evolved a little bit, under a paradigmatic optic. Despite the benefits of Object Oriented in the Software Engineering, there are still unresolved historical problems. Therefore, this work focuses on the problem of how to evolve business rules keeping the project documentation in line with its coding. The proposal of this research is the development of a Programming Model, which allows the integration of the modeling and implementation artifacts, based on a scenario with reference examples. Action Research methodology was used to direct the research, and the construction of the model was performed based on Incremental Iterative method of software development. The proposed model contributed to the construction of a more cohesive and easy to maintain software project, besides maintaining the synchrony between classes model designed and the implemented business rules.
KEYWORDS: programming paradigm; systems modeling; programming logic; software engineering.

Sumário

Lista de Figuras

Capítulo Um

Introdução

O Pensamento Sistêmico é o entendimento de como funciona um conjunto de entidades e suas relações. Para isso, é necessária uma análise do sistema como um todo, levando em consideração suas partes, como elas se relacionam e como se comportam em conjunto, atuando como um sistema único e integrado [Andrade 2009]. O Pensamento Sistêmico é o que permite a representação aproximada de parte da realidade (domínio de conhecimento). Esta representação é necessária para o entendimento de um problema e para a construção de sua solução. Um Sistema de Representação do Conhecimento é a abstração de um cenário do mundo real extraído por meio do Pensamento Sistêmico.

A construção de um Sistema de Representação do Conhecimento depende de habilidades específicas, tais como entendimento da realidade a ser modelada (também denominada de domínio), conhecimento na linguagem de representação e, principalmente, na capacidade de abstração. O processo de modelagem envolve certo grau de complexidade, exigindo do agente modelador a expertise de definir axiomas (regras) sobre elementos conceituais, determinando restrições e definindo quais conceitos farão parte da representação [Kotiadis e Robinson 2008]. O campo de estudos e propostas de formas de modelagem é muito rico. Nesta pesquisa, o termo modelagem será utilizado sempre que se referir as estratégias que permitem a criação de sistemas de representação do conhecimento a partir da observação do mundo real.

A Matemática é a ciência base para esta atividade, pois apresenta um conjunto de técnicas e linguagens formais (como, por exemplo, a Lógica, os Fractais, as Redes Complexas, etc) apropriadas para codificar uma abstração da realidade. A formalização Matemática garante precisão e permite a sua aplicabilidade em uma diversidade de domínios. A Modelagem Computacional, uma área multidisciplinar com base na Matemática, visa representar o conhecimento de processos sistêmicos a fim de resolver problemas que envolvem diversas áreas como Biologia, Matemática e Computação.

Existem várias estratégias de modelagem com variações de precisão a depender do objetivo da construção da representação. A Rede Semântica e o Mapa Conceitual são utilizados normalmente para representações que exigem um grau menor de formalização [Lima e Alvarenga 2008]. Como por exemplo, elencar e conectar os principais elementos de um texto. Existem modelos com um maior grau de precisão, mais utilizados para criar sistemas de representação do conhecimento com recursos de inferência. Esses modelos estão pautados em lógica matemática, como a Modelagem de Ontologias, os Paradigmas de Programação e o MOBI (Modelo Baseado em Instância) [Jorge et al. 2012]. No caso

dos Paradigmas de Programação, a abordagem é mais ampla do que a modelagem, já que os paradigmas também estão associados a uma linguagem de programação, além da linguagem de modelagem.

O MOBI, base para esta pesquisa doutoral, é utilizado para criar representações e apresenta uma metodologia diferenciada para modelagem de domínios. Observa-se que as metodologias associadas aos principais modelos (Redes Semânticas, Mapas Conceituais, Ontologias e Orientação a Objetos) têm suas especificidades, porém todos seguem um processo *Top-Down*, ou seja, primeiro especificam estruturas genéricas e regras (axiomas) para no final do processo fazer uma validação e uso com indivíduos (instâncias). O MOBI se diferencia, pois utiliza uma estratégia *Bottom-Up*, que parte dos indivíduos e suas relações para a construção das estruturas genéricas [Jorge et al. 2012].

A Ciência da Computação também está imbrincada com o Pensamento Sistêmico e com o processo de modelagem para construção de sistemas de representação do conhecimento. Agregando técnicas de modelagem e de implementação lógica, os Paradigmas de Programação definem metodologias, linguagens e estratégias para o desenvolvimento de software [Tucker e Noonan 2010]. Esses paradigmas têm suas especificidades, porém alguns dos seus conceitos e estruturas podem ser absorvidos por outros [Hurlimann 1998]. O Paradigma Orientado a Objetos, por exemplo, utiliza a ideia de funções apresentada no Paradigma Funcional que por sua vez se inspirou nas funções matemáticas. O Paradigma Funcional, assim como o Orientado a Objetos, também utilizou características do Paradigma Imperativo, como a execução de comandos em sequência [Tucker e Noonan 2010].

Os paradigmas estão associados as técnicas de modelagem, como a UML (*Unified Modeling Language*) que foi idealizada por Grandy Booch, James Rumbaugh e Ivar Jacobson [Booch, Rumbaugh e Jacobson 1998], e é utilizada pelo Paradigma Orientado a Objetos. Ou ainda, o Modelo Relacional proposto por Edgar Frank Codd em 1970 no seu famoso artigo *A Relational Model of Data for Large Shared Data Banks* [Codd 1970], que pode ser utilizado tanto no Paradigma Estruturado quanto no Orientado a Objetos. Essas técnicas ou linguagens nem sempre são excludentes, algumas se complementam, agregando comumente elementos de representação para auxiliar no processo de modelagem.

1.1 Problema

Após a proposta do modelo Orientado a Objetos, que teve sua origem em meados dos anos de 1960 no centro Norueguês de Computação, poucas evoluções foram propostas sobre a temática dos Paradigmas de Programação. A Orientação a Aspectos foi uma dessas propostas, que apesar de benéfica para o aumento de coesão e a redução de acoplamento, limita-se a um problema pontual sem abordar questões estruturais de um Paradigma

de Programação [Alexander 2003]. Um ponto que influencia as questões de coesão e acoplamento dentro dos Paradigmas de Programação é a modelagem, que orienta o projeto e a implementação dos sistemas, definindo a construção de artefatos e a organização do código.

Seguindo a maioria dos processos de desenvolvimento de software, uma equipe de projeto se depara com a fase de análise e modelagem de sistema antes da fase de implementação do software [Sommerville 2011]. A fase de análise na Orientação a Objetos é feita a partir da construção de diagramas UML que definem a estrutura e o processo de modelagem. Porém, é comum no decorrer da construção do software a fase de implementação seguir um caminho distinto do que foi especificado em sua fase de modelagem. Isso pode ocorrer por necessidade de evolução do software ou ainda pela identificação de uma solução melhor por parte da equipe do projeto. O problema é que os diagramas e documentos criados na fase de modelagem ficam dessincronizados.

As técnicas como BDD (*Behaviour-Driven Development*), TDD (*Test-Driven Development*) e o DDD (*Domain-Driven Design*) trazem alternativas para suavizar o problema de dessincronização entre a fase de modelagem e a fase de implementação [Wynne e Hellesoy 2012]. Outra abordagem nesse contexto foi proposta quando Joaquin Miller e Jishnu Mukerji descreveram e especificaram pela primeira vez a MDA (*Model-Driven Architecture*) [OMG 2014]. A MDA é uma arquitetura voltada para a construção de sistemas a partir de modelos. O objetivo é o direcionamento do processo de conversão por meio da parametrização das plataformas alvo que vão definir como os modelos construídos serão convertidos, por exemplo, um mesmo modelo pode dar origem a um sistema web ou a um aplicativo para dispositivo móvel ou ainda uma estrutura de tabelas e suas restrições em um banco de dados. A MDA é uma das principais estratégias para tentar resolver o problema da dessincronização entre a fase de modelagem e a fase de implementação, porém a mesma ainda não resolve o problema por completo pois a fase implementação ocorre após a conversão do modelo em uma plataforma específica. Desta forma, sempre que uma intervenção na regra de negócio é efetuada o código fonte fica dessincronizado do modelo original.

Todas essas questões indicam a necessidade de novos estudos e discussões sobre os Paradigmas de Programação, de forma a propor uma solução que resolva ou reduza essa inconsistência entre a modelagem e a implementação, quando se evoluem regras de negócio em projeto de software. Por isso, o problema desta pesquisa é a falta de sincronia entre os artefatos construídos na fase de modelagem e a implementação realizada na fase de projeto.

1.2 Objetivo

O objetivo desta pesquisa é construir um Modelo de Programação baseado em Exemplos (MoPE), que integre o ambiente de modelagem e o de implementação, tendo como base a inferência de restrições e código a partir de cenários com instâncias de referência. Para atingir este objetivo as seguintes metas foram definidas:

- Construir uma metalinguagem de programação baseada em exemplos (denominada Mar);
- Estender o MOBI para permitir sua integração com uma metalinguagem de programação;
- Construir um compilador responsável pela geração do Modelo de Negócio;
- Definir o processo de conversão do Modelo de Negócio para uma linguagem de programação;
- Desenvolver experimentos práticos para avaliar o modelo proposto.

O MoPE utiliza o MOBI para a construção dos modelos conceitual e permite a implementação de regras de negócio de forma integrada com essa modelagem. Esse artefato unificado (modelo conceitual e regras de negócio) é denominado Modelo de Negócio nesta pesquisa, e pode ser exportado para uma plataforma específica.

1.3 Importância da pesquisa

Apesar dos benefícios das técnicas da Engenharia de Software, existem problemas históricos recorrentes em projetos de sistemas, tais como: forte acoplamento, fraca organização semântica, geração de código desnecessário, e outros. Os pesquisadores têm dispendido esforços para solucionar alguns desses problemas porém, alguns deles ainda persistem, como a sincronia entre a modelagem e a implementação, a mudança de papéis das instâncias na Orientação a Objetos e as várias linguagens utilizadas em um projeto de software [Gamma et al. 2011].

Para minorar esses problemas, a Engenharia de Software sugeriu e aperfeiçoou novas técnicas, como a UML, os Padrões de Projeto, ou o uso de *Frameworks*, porém desde a programação Orientada a Objetos poucos modelos relevantes foram sugeridos ou discutidos. Esta pesquisa busca retomar a discussão em torno dos Paradigmas de Programação e contribuir para a evolução de seus modelos, solucionando problemas recorrentes da temática.

1.4 Limites e limitações

Um modelo de programação pode ser muito abrangente sendo utilizado para desenvolver soluções para diversas áreas, como medicina, finanças ou educação. Dessa forma, faz-se necessário a delimitação de um escopo mais restrito que permita uma futura validação do modelo proposto. O modelo proposto nesta pesquisa aborda apenas a modelagem e a programação das regras centradas no Modelo de Negócio do sistema a ser desenvolvido.

Não faz parte do escopo do MoPE a construção de artefatos específicos de uma plataforma, como telas (*interface* com o usuário) de um sistema ou comandos para salvar dados em um banco de dados. O foco do modelo é a especificação e a escrita da lógica responsável pelo registro e gestão de remédios no estoque de um posto de saúde, por exemplo, ou pelo cálculo do imposto de rende, independente do sistema ser *web*, *mobile* ou *desktop*. O Modelo de Negócio que o MoPE entrega como resultado pode ser exportado e utilizado em qualquer plataforma de desenvolvimento de software Orientado a Objetos, onde poderá ser acessado pelos artefatos ali construídos, como outras classes ou serviços.

1.5 Questões e Hipóteses

É possível que a utilização de cenários e exemplos em um modelo de programação viabilize o desenvolvimento de um sistema? Quais benefícios a integração do processo de modelagem com o processo de implementação pode gerar para a Computação? Questões como essas fundamentam esta pesquisa e por elas as seguintes hipóteses foram formuladas:

- Um modelo de programação que integre o processo de modelagem e o processo de implementação gera um projeto de software com menos problemas de sincronia entre os artefatos de análise, projeto e implementação.
- Um modelo de programação baseado em cenários com exemplos de referência é viável para o desenvolvimento de software.

1.6 Aspectos Metodológicos

O principal método científico adotado neste trabalho é a pesquisa-ação, pois o envolvimento do pesquisador é um fator chave para o desenvolvimento desta pesquisa [Guba e Lincoln 1994]. O modelo proposto foi especificado com base na análise de diversos Paradigmas de Programação, tais como o Orientado a Objetos, Orientado a Aspectos, Funcional, Estrutu-

rado, entre outros, e na experiência, observação e debates entre os membros da equipe de pesquisa.

Esta pesquisa prevê 3 etapas evolutivas para a construção do modelo, onde cada etapa possui ciclos iterativos que refinam o modelo. No primeiro passo, as percepções dos pesquisadores, que conduziram este trabalho (autor, orientador e coorientador), foram utilizadas para iniciar a especificação do modelo. Para pautar e embasar essa especificação inicial, foram realizadas pesquisas na área de Pensamento Sistêmico, Sistemas de Representação do Conhecimento, Modelagem Computacional e Paradigmas de Programação.

Após o embasamento teórico dos fundamentos supracitados, a proposta inicial do modelo foi especificada. Para o segundo ciclo desta etapa um conjunto de problemas de programação foi selecionado e dividido em três níveis de dificuldade. Com base nos princípios do modelo e na observação do que era necessário para resolução dos problemas, originou-se a primeira adequação do modelo.

Na segunda etapa temos a implementação, que seguiu as diretrizes de um processo evolucionário para o desenvolvimento do elementos que compõe o modelo. A terceira etapa é a construção de um experimento prático para avaliar o modelo. No caso da metalinguagem alguns critérios ou características serão levados em consideração, como a legibilidade que é utilizado para definir a facilidade de leitura e entendimento de uma linguagem de programação [Buse e Weimer 2010], a facilidade de escrita, ortogonalidade, entre outros. Outra característica observada é o tempo de aprendizagem de uma linguagem de programação [Sebesta 2011].

1.7 Organização do Documento de Tese de Doutorado

Este documento está dividido em seis capítulos, o primeiro sendo esta introdução (1). O Capítulo 2 trata da revisão bibliográfica da pesquisa, onde seus temas chave, como o MOBI e a MDA, são estudados e discutidos. A metodologia de pesquisa utilizada está no Capítulo 3. No Capítulo 4 o modelo de programação proposto é apresentado junto com os trabalhos correlatos. No Capítulo 5 é apresentado o Experimento Prático. E por fim, as Considerações Finais são propostas no Capítulo 6.

Capítulo Dois

Referencial Teórico

Neste capítulo são apresentados e discutidos os principais aspectos teóricos que embasam essa pesquisa de doutorado. É traçado um panorama que vai desde o Pensamento Sistêmico, passando pelos Sistemas de Representação do Conhecimento até chegar aos dois pilares desta pesquisa que são o MOBI e a MDA.

2.1 Pensamento Sistêmico

O Pensamento Reducionista, formado no Século XVII, abordou o entendimento e a resolução de problemas a partir da divisão de suas partes, visando simplificar o problema a ponto de tornar sua resolução simples e viável. O Pensamento Reducionista ajudou na resolução de tarefas e problemas inclusive fora do campo científico. Militares e imperadores, como Napoleão Bonaparte, costumavam utilizar o termo *dividir para conquistar* inspirado no Pensamento Reducionista. A ideia é, por exemplo, forçar uma divisão do exército inimigo para atacar as partes separadamente ou concentrar ataques em partes do território inimigo, conquistando pouco a pouco todo o território.

Porém, determinados problemas têm características que não são preservadas quando divididas, como por exemplo uma planta que se desenvolve normalmente em um solo mas morre em um outro, neste caso não se pode entender o problema analisando somente a planta ou um dos terrenos, é necessária uma análise da planta em cada um dos solos. Para estudar e trabalhar problemas como este, surge o Pensamento Sistêmico, que aborda a análise do sistema como um todo, observando não só as partes mas também suas interações dentro do sistema [Andrade 2009].

O Pensamento Sistêmico consiste na abstração do cenário do mundo real necessário para o entendimento e a resolução de um problema. Essa abstração pode ser concebida a partir de um Sistema de Representação do Conhecimento, que é um conjunto de técnicas e linguagens capazes de representar um domínio específico do conhecimento humano.

2.2 Sistemas de Representação do Conhecimento

Um Sistema de Representação do Conhecimento permite que um contexto do mundo real seja abstraído e representado de forma sistêmica, podendo por exemplo, ser processado

por algoritmos [Sowa 2000]. Geralmente essa representação é feita por meio da definição de conceitos e seus relacionamentos. Em uma Rede Semântica, um exemplo de Sistema de Representação do Conhecimento, pode-se definir conceitos e conectá-los por meio de um tipo de relacionamento [Grilo et al. 2017]. Um exemplo seria o conceito definido como Pediatra e conectado com (outro conceito chamado) Médico por meio da relação "é um", indicando que nesta representação exemplo *Pediatra é um Médico*. A Inteligência Artificial é uma área que costuma utilizar Redes Semânticas para formalizar suas representações em um contexto mais semântico, pois cada um pode ser processado com base em seus relacionamentos [Sowa 2000].

Outro exemplo de Sistema de Representação do Conhecimento é o Mapa Conceitual que também utiliza conceitos e relacionamentos como a Rede Semântica, porém suas relações são mais descritivas, podendo utilizar frases. É possível, por exemplo, definir que "um Carro é feito a partir da composição e montagem de Carroceria, Rodas, Motor e Componentes Eletro-eletrônicos", onde *Carro*, *Carroceria*, *Roda*, *Motor* e *Componente Eletro-eletrônico* são conceitos, e *feito a partir da composição e montagem* um relacionamento.

Em casos onde a representação precise ser mais formal também podem ser definidas restrições ou regras lógicas. Na construção de uma Ontologia, por exemplo, podem ser definidas restrições nos relacionamentos entre seus conceitos. A relação anterior do conceito *Carro* pode ser restrita à 4 indivíduos do conceito *Roda*, indicando que um Carro só deve possuir 4 rodas. Além de representar um domínio do conhecimento humano, uma Ontologia tem como objetivo formalizar e compartilhar esse domínio do conhecimento [Gruber 1993].

A Computação utiliza Sistemas de Representação do Conhecimento para pautar o desenvolvimento de softwares. A Engenharia de Software define duas importantes etapas para o desenvolvimento de projetos, são as etapas de Modelagem e de Implementação que consistem na construção de representações do comportamento do sistema, por meio das linguagens de modelagem e de programação.

Uma Linguagem de Programação consiste na definição de regras e estruturas que tem como objetivo escrever um conjunto de instruções para programar um computador, ou algoritmo. Qualquer formalização para descrição de um algoritmo ou de uma estrutura de dados por ser chamada de Linguagem de Programação [Kedar 2007].

Um algoritmo é uma representação do comportamento desejado para o computador. Algumas características em uma Linguagem de Programação potencializam a qualidade dessa representação. Autores como [Kedar 2007] e [Sebesta 2011] abordaram algumas dessas características:

- **Simplicidade**: uma linguagem é considerada simples quando seu conjunto base de comandos e estruturas, suficiente para escrever diversos algoritmos, não possui muitas construções básicas ou formas distintas de escrever um mesmo comando;
- **Ortogonalidade**: é a capacidade que as estruturas básicas de um linguagem fornecem para a construção de novas estruturas, aumentando o detalhamento da representação de acordo o necessário;
- **Suporte a Abstração**: é possibilidade de utilização de uma estrutura ignorando sua complexidade, apenas os aspectos relevantes para realizar a chamada ou ativação da estrutura devem ser conhecidos enquanto os detalhes de suas operações ficam escondidos;
- **Verificação**: recurso para checar se as especificações da linguagem estão sendo seguidas de forma a evitar erros na escrita do algoritmo.

Essas características são bem vistas em uma Linguagem de Programação pois permitem com que a mesma seja de boa legibilidade, tenha facilidade de escrita e seja confiável, permitindo a um programador escrever algoritmos com representações factíveis e detalhadas para o funcionamento de um computador.

Como uma Linguagem de Programação não está focada em abordar aspectos em um nível de projeto de software, faz-se necessário a utilização de uma Linguagem de Modelagem [Fowler 2005]. Uma Linguagem de Modelagem serve para especificar, construir e documentar os artefatos de projeto visando organizar os macro componentes do software, definir o comportamento futuro do sistema e representar graficamente esses artefatos em mais alto nível [Ramos 2006].

A utilização de ambas essas representações no desenvolvimento de um sistema computacional é fundamental. Quando uma técnica combina linguagem de modelagem e linguagem de programação a alguns métodos de desenvolvimento tem-se a formação de um Paradigma de Programação.

2.3 Paradigmas de Programação

Os Paradigmas de Programação são modelos de desenvolvimento de software que estruturam aspectos de implementação, assim como a forma de pensar a abstração do sistema e sua representação [Papert 1991]. Alguns paradigmas foram idealizados apenas sob a óptica da implementação, sem a preocupação com a etapa de modelagem de um projeto de software. Como o Paradigma Imperativo que além de ser um dos primeiros representa

a base da maioria dos outros paradigmas, pois estrutura a execução dos comandos de um programa em uma definida ordem [Kedar 2007].

O Paradigma Funcional é outro que não aborda linguagens de modelagem, estruturando apenas a organização e a execução dos comandos de um programa. Neste paradigma, a ideia de funções tem relação direta com a Matemática, pois uma função escrita em um programa tem as mesmas restrições que as funções matemáticas: poder receber parâmetros (informações de entrada) antes do início de sua execução; executar uma sequência de comandos que utilizam esses parâmetros; e retornar um valor que deve ser sempre o mesmo para um determinado parâmetro (exemplo na matemática: $f(x) = y$). Este paradigma dá mais enfase a chamada das funções a passagem de valores entre elas (valor resultante de uma função como valor de entrada em outra) do que na execução sequencial dos blocos de comando [Kedar 2007].

O Paradigma Estruturado foi um dos primeiros a abordar técnicas de modelagem, como por exemplo a Análise Essencial. Este paradigma também abordava questões sobre divisão dos blocos de comandos, a modularização, e a criação de estruturas que agrupam variáveis, formando assim, novos tipos de armazenamento de dados.

O Paradigma Orientado a Objetos agrupou variáveis e métodos (funções) em um mesmo artefato, chamado de objeto, permitindo a manipulação e o armazenamento de dados de forma mais atômica e coesa [Kedar 2007]. Esta nova estruturação é dita mais próxima das estruturas do mundo real, pois os indivíduos desempenham papéis a depender de seu conhecimento, assim como os objetos executam comandos a partir de seus dados. Um objeto, de forma conceitual, é uma instância (indivíduo) de um agrupamento, como *Pessoa* ou *Professor*, também chamadas de classes (conceitos), é nas classes que são definidos quais métodos ou variáveis um objeto pode possuir. Os objetos são sempre criados a partir de uma determinada classe e permanecerão atrelados a ela até o fim de sua utilização. Assim como *Pesquisador* pode ser um exemplo de classe, *Alan Turin* pode ser um exemplo de objeto dessa mesma classe.

A Orientação a Objetos divide seu processo de desenvolvimento em três etapas, Análise, Projeto e Implementação. Onde tanto a fase de Análise quanto a de Projeto abordam estratégias de modelagem. Na fase de Análise uma abordagem mais voltada para a modelagem conceitual é realizada, enquanto a fase de Projeto aborda uma modelagem mais específica já voltada para a fase de Implementação, como a modelagem para definir as camadas de projeto ou para a utilização de alguns Padrões de Projeto [Sommerville 2011].

Uma Linguagem de Modelagem Orientada a Objetos, como a UML, permite a modelagem feita de forma similar a um Mapa Conceitual, ou Rede Semântica, bem como a modelagem voltada a arquitetura do projeto de software. Na etapa de Análise, foco desta pesquisa, é

possível especificar classes e seus relacionamentos, além de definir quais tipos de métodos e atributos o futuro objeto poderá possuir.

A implementação lógica (escrita de blocos de comandos) dos métodos das classes é feita apenas a partir de uma linguagem de programação, porém com a linguagem de modelagem é possível definir a assinatura do método que será futuramente implementado em um classe. A assinatura de um método consiste na identificação de três elementos: tipo de dados que o método irá retornar; nome do método; e os tipos dos parâmetros de entrada. Desta forma, é possível especificar com a modelagem como um método deve se comportar deixando para a etapa de implementação a construção lógica da execução de seus comandos.

Na Computação, e mais precisamente na Engenharia de Software, os Sistemas de Representação do Conhecimento tem papal fundamental, pois é a modelagem (da fase de Análise) que estrutura as classes e o comportamento dos objetos, direcionando os caminhos da implementação.

Ainda na Computação, existem outras estratégias que tratam os Sistemas de Representação do Conhecimento, como as Ontologias que são utilizada para representar um domínio do conhecimento utilizando restrições lógicas no relacionamento de seus conceitos. O objetivo de uma Ontologia está mais centrado na representação formal de um domínio que seja compartilhável, entendida por seres humanos e processada por algoritmos [Gruber 1993]. A linguagem utilizada para construção de uma Ontologia é a OWL (*Web Ontology Language*) que para permitir as escritas de restrições e regras lógicas e facilitar seu compartilhamento entre sistemas é baseada em RDF (*Resource Description Framework*) e XML (*eXtensible Markup Language*) [W3C 2014].

O MOBI inicialmente foi pensado como uma estratégia para modelagem de Ontologias, porém evoluiu durante as pesquisas que o deram origem para se tornar um modelo para criação de Sistemas de Representação do Conhecimento com suas próprias características e premissas.

2.4 MOBI - Modelo de Ontologia Baseado em Instância

O Método de Ontologia baseado em Instância (MOBI) visa à utilização da estratégia *Bottom-Up* [Vet e Mars 1998] para modelagem de Sistemas de Representação do Conhecimento, como as Ontologias, a Orientação a Objetos ou o Modelo Relacional. Essa modelagem é feita a partir da construção de cenários, onde instâncias são relacionadas entre si a fim de fornecer informações que permitam a inferência dos conceitos e seus tipos de relacionamento [Jorge et al. 2012]. Para a construção destes cenários não é necessário

descrever todos as possibilidades, mas somente as que usando instâncias de referência retratam os relacionamentos e classificações relevantes ao contexto da representação.

A estratégia *Bottom-Up* foi inspirada no Diagrama de Instância do modelo Orientado a Objetos onde as instâncias ficam dispostas em 2 colunas e uma coluna se relaciona com a outra por meio de uma propriedade de relacionamento. Um lado representa uma classe e o outro uma segunda classe que possui uma relação com a primeira [Jorge et al. 2012].

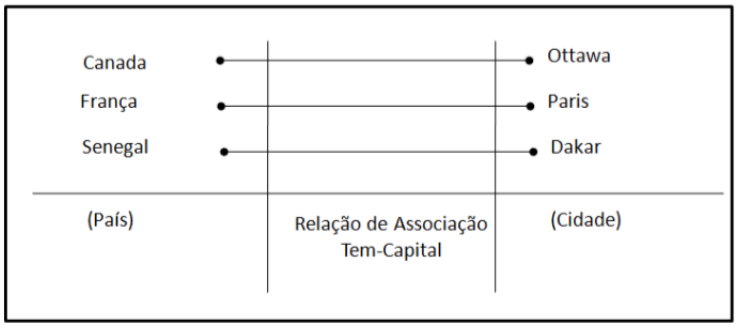


Figura 2.1: Divisão de Cenário do MOBI [Jorge et al. 2012].

Como pode ser visto na Figura 2.1, o MOBI divide um cenário seu em seis partes. Em baixo, ficam o nome da primeira classe, o nome da relação e o nome da segunda classe, respectivamente. Em cima, os nomes das instâncias pertencentes a primeira classe, as ligações entre as instâncias e as instâncias da segunda classe, também respectivamente. A ideia principal é que as restrições das classes e de suas relações sejam inferidas. No exemplo (da Figura 2.1) pode-se verificar que não existe um *País* associado a mais de uma *Cidade*, na relação *Tem-Capital*. Logo, por meio de inferência determina-se que, nesse contexto, um país só pode possuir uma capital.

Outro exemplo é o ilustrado na Figura 2.2, onde algumas instâncias da classe *Posto* (referente a Posto de Saúde) são associados a uma instância da classe *Remedio*. Nesse cenário o MOBI infere que a relação *possui* pode relacionar um posto de saúde a vários remédios. Ainda de acordo com esta representação exemplo, uma instância da classe *Posto* não precisa possuir relacionamento com um *Remédio*, pois no cenário modelado existe um *Posto* (Pernambués) sem relacionamento com qualquer indivíduo da classe *Remédio*.

Para melhorar o entendimento, a Figura 2.3 mostra o resultado de uma possível conversão desse exemplo feito no MOBI para um Diagrama de Classes da UML. Em uma modelagem exemplo de um sistema para registro de remédios em postos de saúde, o resultado é justamente a associação das classes *Posto* e *Remedio*, onde um posto de saúde pode se associar a 0 (zero) ou mais remédios e um remédio tem que se associar a pelo menos 1 (um) posto de saúde.

Seguindo essa estratégia, um analista de sistemas pode construir um cenário a partir

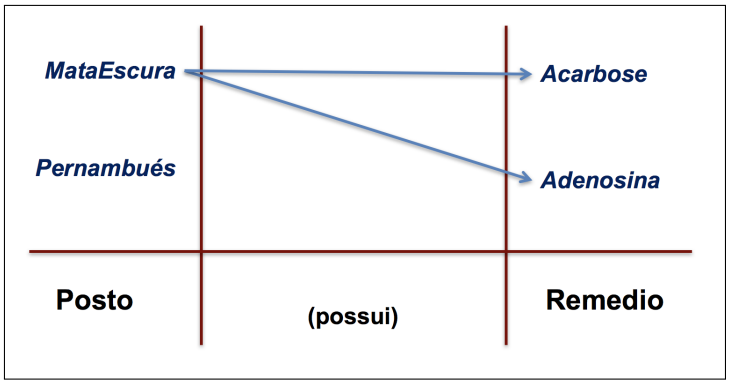


Figura 2.2: Exemplo de Modelagem feita no MOBI.
Fonte: Autor.

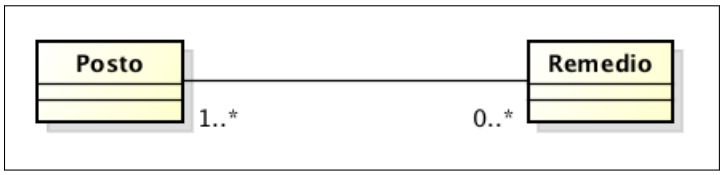


Figura 2.3: Modelagem de Postos de Saúde em UML gerada pelo MOBI.
Fonte: Autor.

de instâncias, de um exemplo real, sem precisar se preocupar com as restrições ou características do modelo. O próprio modelo é construído a partir desse exemplo o que já valida o mesmo, empiricamente. Além disso, por ser feito a partir das instâncias e não das classes, quando o modelo sofrer uma alteração (ex.: um remédio sem associação com um posto de saúde), basta alterar o cenário e todas as restrições ou características inferidas dessa relação serão reconstruídas.

No processo *Top-Down* o analista de sistemas começa pela construção dos conceitos, de suas restrições e de seus relacionamentos, para depois criar instâncias que validam o modelo, como ilustra a Figura 2.4. Na área da Computação o processo é parecido, primeiro as estruturas das classes são definidas para depois o programa ser implementado e testado com o lançamento de dados. Caso algum problema ou inconsistência na modelagem sejam identificados uma correção é realizada na primeira etapa e o processo todo recomeça.

No caso do MOBI, o processo já começa com a montagem de cenários exemplo, onde instâncias são vinculadas umas as outras. Esses cenários são processados pelo compilador para geração das estruturas de classes (ver Figura 2.5). Ao mesmo tempo em que o modelo é inferido já está sendo validado e caso alguma inconsistência seja identificada ela mesma pode ser usada para retroalimentar os exemplos de referência e gerar uma nova versão do modelo desejado.

Além dessas características, o MOBI trata a instância de forma independente de qualquer

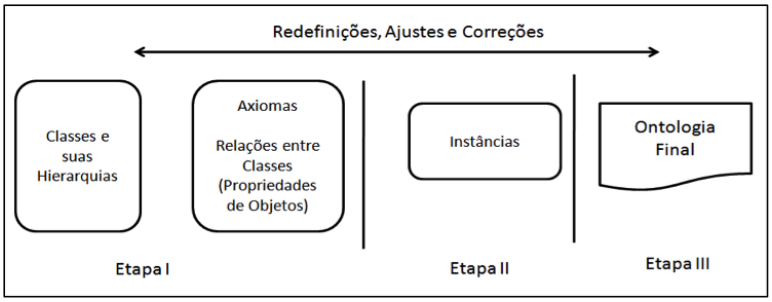


Figura 2.4: Fluxo Convencional dos Processos de Modelagem [Jorge et al. 2012].

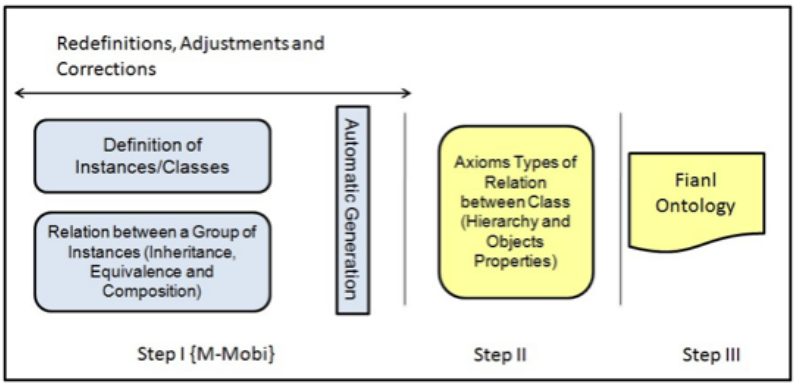


Figura 2.5: Fluxo de Modelagem do MOBI [Jorge et al. 2012].

classe. Na Orientação a Objetos para que se crie uma instância é obrigatório fazê-lo a partir de uma classe e essa instância nunca pode ser mudada de classe. A instância é fortemente vinculada a classe na Orientação a Objetos e na maioria dos Sistemas de Representação do Conhecimento. No MOBI uma instância pode mudar de classe ou agregar novas classes, aumentado as suas funcionalidades, mais próximo do que acontece no mundo real onde os indivíduos podem agregar papeis ou funções a depender das circunstâncias.

Essas características são importantes, pois irão nortear o modelo proposto neste trabalho assim como sua metalinguagem. A partir dessa metalinguagem será possível reutilizar na etapa de desenvolvimento qualquer artefato construído pelo MOBI na etapa de modelagem, assim como vincular classes às instâncias.

Para melhor integração deste modelo com outras plataformas é necessário uma conversão da modelagem conceitual e lógica realizadas no MoPE por meio de um processo de conversão automatizado. Para isso, é fundamental uma estratégia que direcione esse processo. A MDA possui especificações justamente para conversão de modelos em projetos de software o que pode contribuir para especificação e construção do MoPE.

2.5 MDA - Model Driven Architecture

A MDA define uma arquitetura de referência para a geração de código fonte a partir de modelos. Essa especificação é baseada em dois modelos principais. O PIM (*Platform Independent Model*) é um modelo independente de plataforma utilizado para descrever o negócio em alto nível, abstraindo os detalhes sobre a tecnologia de implementação [Kleppe, Warmer e Bast 2004].

O PSM (*Platform-Specific Model*) é um modelo de uma plataforma específica gerado a partir do PIM, um exemplo disso seria uma lógica de programação escrita em uma linguagem específica de determinada tecnologia, como um código *TypeScript* na plataforma *Ionic* (voltada para desenvolvimento *mobile*) [OMG 2014], que seria complementado em projeto de software, com a construção de telas em *HTML* e *CSS*, e o uso do *framework Cordova* para comunicação com os dispositivos móveis.

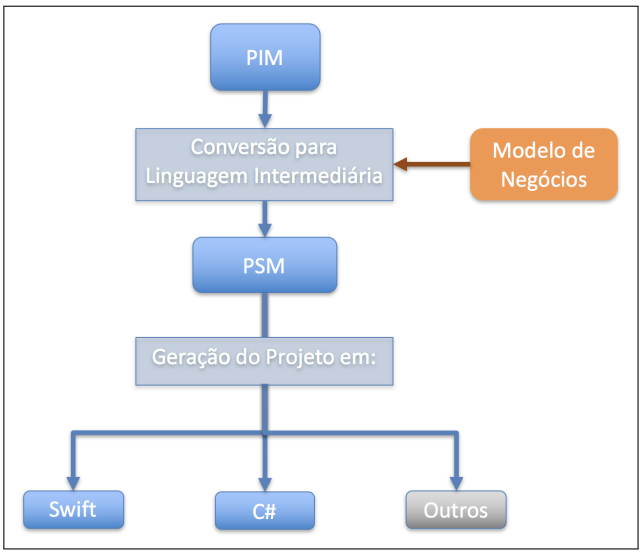


Figura 2.6: Processo de conversão da MDA.
Fonte: Autor.

A Figura 2.6 ilustra o processo de conversão da MDA. Na parte de cima, temos o PIM, onde é feita a modelagem conceitual do sistema (independente de plataforma). A Conversão para o PSM é feita a partir da modelagem e da atribuição de um Modelo de Negócios específico que indica para qual modelo de plataforma o PSM será gerado, como por exemplo as especificações da plataforma Java que resultará em um PSM escrito na linguagem de programação Java. Na etapa de Geração do Projeto, é feita a conversão para uma plataforma específica como por exemplo um projeto para dispositivo móvel implementado na plataforma *Android*, que usa linguagem Java.

O processo de conversão descrito na MDA pode melhorar a produtividade de um projeto de desenvolvimento de software, mas esse não é o único benefício dessa arquitetura. A

portabilidade é outro, uma vez que focando no desenvolvimento do PIM a geração do PSM será direcionada para a plataforma necessitada no momento. Outra vantagem importante é a melhor manutenção da documentação do projeto, uma vez que o modelo é a base para geração do código fonte o mesmo não será abandonado após o inicio da fase implementação e qualquer mudança deverá ser realizada alterando o modelo para depois regerar o PSM [Kleppe, Warmer e Bast 2004].

É importante ressaltar que o MoPE tem como objetivo construir apenas regras de negócio e modelos conceituais para projetos de software Orientados a Objetos, não abordando se a plataforma utilizada é para um sistema *web*, *mobile* ou *desktop*. Dessa forma, o PSM gerado será sempre das classes que representam o modelo conceitual do sistema com os métodos que representam as principais regras de negócio para um sistema Orientado a Objetos.

Seguindo as especificações da MDA e utilizando o MOBI e a metalinguagem Mar como instrumentos para construção dos PIMs, esta pesquisa de doutorado propõe a construção de um Modelo de Programação baseado em Exemplos (MoPE). No próximo capítulo (3), a metodologia utilizada nessa pesquisa é apresentada e detalhada.

Capítulo Três

Metodologia

Neste capítulo são apresentados e discutidos os aspectos metodológicos que conduziram essa pesquisa de doutorado. Etapas foram definidas para a avaliação do modelo, que consistem na formulação e resolução de problemas relacionados a modelagem e a implementação de regras de negócio utilizando o MoPE.

A pesquisa-ação foi o método que escolhido para direcionar o desenvolvimento desta pesquisa [Checkland e Holwell 2007], o motivo é a importância da experiência dos pesquisadores neste trabalho, pois apesar da Engenharia de Software ser uma disciplina consolidada na Ciência da Computação, ainda existem alguns problemas por ela não resolvidos (ver início do Capítulo 4). Nesta etapa, o MOBI foi idealizado como base para o modelo, provendo o ambiente para modelagem e as ideias necessárias para se propor um modelo de programação orientado a exemplos. O modelo proposto passou a incorporar todas as premissas do MOBI, como a instância independente de classes que ajuda a resolver o problema da troca de papéis das instâncias (ver início do Capítulo 4).

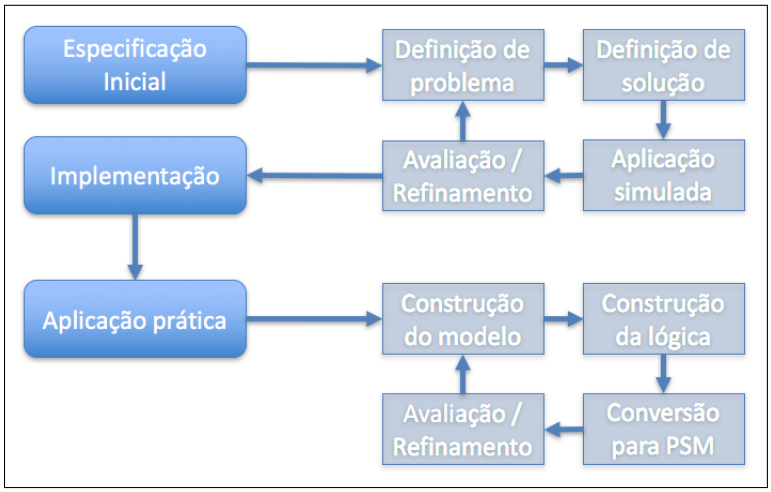


Figura 3.1: Ciclo de Desenvolvimento da Pesquisa.
Fonte: Autor.

Também foram utilizados alguns processos de desenvolvimento ligados a Engenharia de Software, como o Evolucionário [Sommerville 2011] para estruturar o desenvolvimento da pesquisa. Conforme ilustrado na Figura 3.1, o passo a passo de desenvolvimento desta pesquisa tem início no levantamento e na revisão bibliográfica, onde os pesquisadores definiram uma Especificação do modelo. Esta especificação foi evoluída a partir de um ciclo de refinamento, detalhado a seguir.

3.1 1ª Etapa - Especificação

O objetivo desse ciclo é testar e refinar algumas premissas do MoPE, para isso definiu-se os seguintes passos:

3.1.1 Passo I - Definição de Problema

Neste passo são definidos os problemas a serem solucionados neste ciclo. Alguns problemas definidos foram: como simplificar a implementação de algumas estruturas de repetição; e como simplificar a implementação de algumas estruturas condicionais.

Uma estrutura condicional é aquela onde o programador pode executar um determinado comando a depender da verificação de uma proposição lógica, se a proposição for verdadeira executa-se um determinado comando, se não for, pode-se executar outro ou não fazer nada. Estas proposições lógicas podem, por exemplo, verificar se um elemento é igual a um dado valor, se o valor de uma variável é maior ou menor que outra. Já uma estrutura de repetição consiste na repetição de um comando, porém também com base na verificação de uma proposição lógica.

3.1.2 Passo II - Definição de Solução

Neste passo são definidas as premissas, ou lógicas, de solução para os problemas propostos. As premissas definidas aqui para os problemas citados acima foram: utilizar a totalidade das instâncias de uma relação para inferir a iteração (Regra 6 - Subseção 4.5.1.6); utilizar as relações definidas entre as instâncias para inferir a condicionante (Regra 7 - Subseção 4.5.1.7).

3.1.3 Passo III - Aplicação Simulada

Neste passo é feito uma simulação de como o Conversor deve realizar a inferência da lógica implementada na metalinguagem Mar. Após a construção das classes, relacionamentos e das regras de negócio (lógica de programação), é feita a simulação da conversão que consiste na implementação em linguagem de programação Java (escolhida para ser a linguagem intermediária) da inferência resultante que representa o modelo construído. Todos os artefatos são escritos na linguagem intermediária da mesma forma como o Conversor deveria inferi-los, desta forma, pode-se verificar posteriormente se o Conversor processou

a inferência corretamente.

3.1.4 Passo IV - Avaliação / Refinamento

O quarto passo é onde os pesquisadores avaliam os resultados daquele ciclo, refinando as premissas da pesquisa e sua lógica de solução. Neste passo foram idealizados novos problemas, assim como algumas soluções foram melhoradas. Essas evoluções deram origem a novas regras de inferências (mais detalhes na Subseção 4.5.1) além das citadas acima.

3.2 2ª Etapa - Implementação

Na segunda etapa da pesquisa, a Especificação é utilizada como base para a formalização da metalinguagem de programação Mar e para a implementação do Conversor, que é responsável pela inferência da lógica escrita em Mar para a linguagem intermediária (Java).

A implementação do compilador do MoPE foi realizada na plataforma Java, mesma linguagem na qual o MOBI foi escrito, o que facilitou a integração entre ambos. Nos testes realizados, a modelagem foi realizada a partir da implementação do MOBI feita em Java, que foi importada e utilizada no projeto do MoPE. Já a implementação com Mar foi escrita em arquivos do tipo *.mar* os quais são lidos e interpretados por esse compilador.

Esta implementação pode ser analisada em detalhes no Apêndice A.

3.3 3ª Etapa - Aplicação Prática

A terceira etapa utiliza um outro ciclo de refinamento, desta vez para desenvolver uma aplicação utilizando o MoPE. O primeiro passo desse ciclo é a construção do modelo, que consiste na especificação dos cenários, com os relacionamentos de seus exemplos. O segundo passo é implementação das regras de negócio, na metalinguagem Mar, recomenda-se que tanto esse 1º (primeiro) passo quanto o 2º (segundo) sejam realizados por um analista de sistemas (no caso desta pesquisa foi o seu pesquisador principal). No terceiro passo a lógica de conversão implementada na 2ª Etapa (Seção 3.2) processa a inferência que constrói os atributos e os métodos juntamente com suas respectivas classes.

Estes artefatos são convertidos em código de programação da linguagem intermediária (Java), formando o nosso modelo específico de plataforma (PSM), e são comparados com

os resultados que foram simulados no terceiro passo da 1ª Etapa (Aplicação Simulada). Sendo necessário algum refinamento no modelo proposto, o ciclo de refinamento da 1ª Etapa deve ser refeito. Umas vez validado, o PSM é importado em uma plataforma específica para o desenvolvimento de um projeto de software (como o apresentado no Capítulo 5), que pode ser desenvolvido por outros analistas de sistemas que não os mesmos dos passos 1 (um) e 2 (dois).

Capítulo Quatro

Modelo de Programação baseado em Exemplos

Pesquisadores têm dispendido esforços para solucionar alguns dos problemas que envolvem um ambiente de programação. Porém, alguns deles ainda persistem, como a falta de sincronia entre a modelagem e a implementação. Quando o modelo conceitual e as regras de negócio de um projeto de software evoluem de forma dessincronizada entre modelagem e implementação. As estruturas especificadas com a linguagem de modelagem são reconstruídas na linguagem de programação. Isso ocorre, pois, os processos de modelagem e de implementação são realizados de forma separada e desconectada. O problema é que no fim de um projeto de software, os artefatos gerados na etapa de modelagem não representam o sistema construído o que pode gerar alguns problemas, como manutenções feitas de maneira equivocadas ou o aumento da curva de aprendizado de um novo membro da equipe de projeto.

O Paradigma Orientado a Objetos também tem alguns problemas relacionados a projetos que usam a herança de classes (sem disjunção). Como na orientação a objetos uma instância só pertence a uma classe, não podendo mudar e nem adquirir outra classe para aumentar suas funcionalidades, as representações acabam ficando imprecisas gerando problemas na implementação. Quando o objeto muda de papéis (alterna entre subclasses), sempre que codificada, a herança de classes não resolve o problema completamente. Um exemplo para este cenário seria um sistema Acadêmico onde as classes *Pessoa*, *Aluno* e *Professor* são modeladas com herança. Neste exemplo, a classe *Aluno* e *Professor* são subclasses de *Pessoa*. Assim, quando ocorrer um Professor que seja aluno (Professor de graduação e Aluno de pós-graduação por exemplo) o sistema não fornecerá uma solução adequada, em uma das opções o cadastro dessa pessoa seria duplicado, pois o sistema não poderia ter uma instância pertencente a duas classes, e caso o sistema faça uma restrição para não duplicar registros, como as feitas por CPF por exemplo, o usuário não poderia ser cadastrado no sistema. Na Orientação a Objetos esse problema poderia ser solucionado utilizando herança múltipla, onde poderia ser criada uma classe (*ProfessorAluno*) que herdaria tanto de Aluno quanto de Professor, porém essa solução é complexa, pois para cada combinação de classes seria necessário a criação de uma classe que represente essa combinação [Gamma et al. 2011].

O objetivo desta pesquisa é a construção de um modelo de programação que integre a modelagem e a implementação de sistemas. Para isso é proposto o MoPE, um modelo que agrega um ambiente de modelagem (baseado no MOBI), um ambiente de implementação (baseado na metalinguagem Mar) e um processo de conversão (baseado na MDA), capaz de gerar um Modelo de Negócio reutilizável, que pode ser utilizado em um projeto de

software de uma plataforma específica.

Esse Modelo de Negócio é concebido a partir de modelagens construídas no MOBI e de implementações escritas em Mar, representando a integração entre ambos os artefatos. Desta forma, o MoPE permite a construção de modelagem e regras de negócio, assim como seu uso de maneira integrada.

4.1 Discussões e Trabalhos Correlatos

Com o intuito de auxiliar os profissionais no desenvolvimento de sistemas, algumas plataformas acrescentaram interessantes melhorias em suas linguagens, como a tipagem dinâmica (ex.: Python, Perl e Ruby) ou a chamada de métodos de forma verbosa (Objetive-C) [Apple 2012]. Tipagem dinâmica é quando uma variável assume o tipo do primeiro valor atribuído a ela. Outras linguagens foram especificadas para ser multiparadigmas, podendo conter chamadas de *script* ou programação lógica junto com a orientação a objetos. Plataformas como *Visual Studio* e *Xcode* buscaram reduzir o tempo de desenvolvimento e suavizar a complexidade do processo de desenvolvimento por meio de editores com ferramentas e recursos de apoio ao desenvolvedor, como autocompletar código, reuso de biblioteca e componentes, engenharia reversa, construção visual de telas, etc. Ferramentas com essas característica são denominadas de *Rapid Application Development* (RAD), e como mencionado, o objetivo é aumentar a produtividade e tornar a construção de software uma tarefa menos árdua [Sommerville 2011].

O problema é que estes novos recursos não alteram questões paradigmáticas sobre a filosofia de como desenvolver software. Houveram, por exemplo, poucas mudanças relacionadas à maior proximidade das estruturas com o mundo real, a não utilização no ambiente de programação dos artefatos definidos no ambiente de modelagem, ou ainda a melhora dos aspectos de organização semântica de código.

Outra questão importante é que o paradigma Orientado a Objeto não resolve de forma satisfatória alguns problemas relacionados a projetos que usam a herança de classes sem disjunção. Como na orientação a objetos uma instância só pertence a uma classe não podendo mudar e nem adquirir outra classe para aumentar suas funcionalidades, ocorrem problemas de desenvolvimento, como o caso citado por [Gamma et al. 2011], quando o objeto muda de papel (alterna entre subclasses). Nestes casos, sempre que codificada, a herança de classes não resolve o problema completamente. Um exemplo para este cenário seria um sistema Acadêmico onde as classes Pessoa, Aluno e Professor são modeladas com herança. Neste exemplo, a classe Aluno e Professor são subclasses de Pessoa. Assim, quando ocorrer um Professor que seja também Aluno (Professor de graduação e Aluno de pós-graduação por exemplo) o sistema não fornecerá uma solução adequada, em uma

das opções o cadastro dessa pessoa seria duplicado, pois o sistema não poderia ter uma instância pertencente a duas classes, e caso o sistema faça uma restrição para não duplicar registros, como as feitas por CPF por exemplo, o usuário não poderia ser cadastrado no sistema. Na Orientação a Objetos esse problema poderia ser solucionado utilizando herança múltipla, onde poderia ser criada uma classe (ProfessorAluno) que herdaria tanto de Aluno quanto de Professor, porém essa solução é complexa, pois para cada combinação de classes seria necessário a criação de uma classe que represente essa combinação.

A Programação Orientação à Aspectos (POA) foi uma abordagem que tentou contribuir na organização semântica do código fonte, agrupando funcionalidades por seu grau de importância dentro do escopo do projeto. Porém apesar da abordagem resultar em benefícios, como o aumento na coesão dos objetos e a redução no acoplamento entre eles, a POA acaba apresentando problemas na dependência de dados, controle de mudanças e entendimento da lógica escrita, já que a mesma altera a implementação original removendo partes da codificação, para realocar em outro agrupamento, e inserindo trechos de código [Alexander 2003].

O Baixo acoplamento é um termo que indica uma menor dependência da estrutura para com outras o que facilita sua reutilização e reduz o impacto de mudanças na mesma [Gamma et al. 2011], já a alta coesão é obtida quando as tarefas realizadas por uma estrutura estão sempre relacionadas, não se tornando muito genérica, pois isso dificultaria sua compreensão, reutilização, manutenção e tenderia a aumentar seu acoplamento [Larman 2007].

Na tentativa em aumentar a integração de modelagem e implementação em projetos de aplicações Web 2.0, Ching Hsu propôs um trabalho baseado na MDA. O foco era permitir o desenvolvimento de serviços que agrupam e integram funcionalidades da Web 2.0 por meio de modelos definidos em UML gerando código fonte a partir de processos de conversão e de técnicas da MDA [Hsu 2012].

Com objetivos similares, Rongliang Luo propôs uma arquitetura para modelagem e implementação de aplicações web para dispositivos móveis [Luo, Peng e Lv 2013]. A ideia da arquitetura é prover a construção de aplicativos para dispositivos móveis também a partir da conversão de modelos pautada pela MDA. Os aplicativos seriam gerados em tecnologias web, como (*HyperText Markup Language*) HTML 5, (*Cascading Style Sheets*) CSS 3 e *JavaScript*. As aplicações seriam suportadas por um serviço web que proveria informações e algumas funcionalidades, como banco de dados, distribuição da aplicação nas lojas virtuais, entre outros.

O que ambos os trabalhos têm em comum é a utilização da MDA para pautar a conversão do modelo em código fonte. Isso ajuda na integração entre modelagem e implementação,

além de automatizar a geração, mesmo que parcial, do código fonte. Porém, somente o uso da MDA, não resolve os problemas de questões paradigmáticas, como por exemplo, a questão citada acima sobre a herança múltipla, ou a evolução das regras de negócio, pois a implementação complementar à modelagem deverá ser feita após a geração em código fonte das estruturas definidas no modelo e isso pode causar aumento no acoplamento e redução na coesão, sempre que as regras de negócio precisem ser alteradas.

Além da MDA, essas propostas também tem a Orientação a Objetos como estratégia em comum com o MoPE (proposta desta pesquisa). Porém como essas propostas não abordam o uso de uma linguagem, ou metalinguagem, de programação que permita completar o modelo criado com a escrita das regras de negócio, essa implementação deve ser realizada na plataforma específica o que dificulta a sincronia entre modelagem e implementação, e acopla o modelo de negócio a tecnologias de plataformas específicas (e.g.: um aplicativo *iOS* com serviços em *.NET*), o que também dificulta o reuso do mesmo em outras plataformas (e.g.: um sistema *desktop* com *Java Swing*). Havendo a necessidade de uma alteração no modelo de classes, por exemplo, sempre haverá um impacto na plataforma específica, pois as regras de negócio estão escritas lá e tem forte dependência com essa modelagem.

Tabela 4.1: Resumo Comparativo

PROPOSTAS	Alta Coesão / Baixo Acoplamento	Legibilidade	Sincronia entre Modelo e Código	Abstração do Modelo de Negócios
Orientação a Aspectos	Facilita	Dificulta	Não	Depende do analista
Orientação a Objetos	Facilita	Facilita	Não	Depende do analista
[Luo, Peng e Lv 2013]	Dificulta	Dificulta	Sim	Não
[Hsu 2012]	Dificulta	Dificulta	Sim	Não
MoPE	**Facilita**	**Facilita**	**Sim**	**Sim**

A Tabela 4.1 compara as abordagens da Orientação a Aspectos, Orientação a Objetos, de [Luo, Peng e Lv 2013] e de [Hsu 2012], com o MoPE.

O primeiro critério utilizado é se a proposta facilita ou dificulta a construção de estruturas com alta coesão e baixo acoplamento. O segundo critério leva em consideração a Legibilidade dos artefatos entregues em cada abordagem, se de forma geral é um código de fácil leitura e entendimento por parte dos analistas de sistemas. O terceiro critério verifica se a sincronia entre os artefatos de modelagem e de implementação é mantida quando uma ou mais mudanças são realizadas. Por fim, o quarto critério analisa se o Modelo de Negócio é abstraído dos componentes específicos da plataforma alvo, o que possibilita a sua reutilização em outros tipos de plataformas ou projetos.

Como a Orientação a Aspectos é uma extensão da Orientação a Objetos, a primeira possui características similares a segunda. Ambas facilitam a construção de estruturas com Alta

Coesão e Baixo Acoplamento, assim como não tratam a sincronia entre modelagem e implementação e deixam a cargo do Analista de Sistemas a abstração do Modelo de Negócio do sistema. Porém, por alterar o comportamento de um código já escrito, a Orientação a Aspectos se diferencia da Orientação a Objetos e acaba dificultando a Legibilidade do código de programação.

Por serem ambas pautadas na MDA, as propostas de [Luo, Peng e Lv 2013] e [Hsu 2012] tiveram resultados equivalentes. A MDA mantem a sincronia entre a modelagem e a implementação, por conta da geração de parte do código fonte, porém não prevê a implementação das regras de negócio em sua arquitetura, por conta disso, essa implementação tem de ser realizada na plataforma alvo o que impede a abstração do Modelo de Negócio, aumenta o Acoplamento e dificulta a Legibilidade do código.

A estratégia do MoPE é utilizar o MOBI como ferramenta de modelagem para permitir a construção de Modelos Conceituais, e a metalinguagem de programação *Mar* para permitir a escrita das Regras de Negócio. O resultado final é o Modelo de Negócios que pode ser exportado para um projeto de software onde possa ser complementado com artefatos específicos da plataforma alvo, como por exemplo a implementação de uma conexão com o banco de dados ou a construção das telas de sistema para interação com os usuários. Com esta estratégia pretende-se preservar a sincronia entre modelagem e implementação do sistema, assim como permitir a reutilização do Modelo de Negócios em outros projetos, sempre que necessário.

4.2 Requisitos do Modelo

Para direcionar o desenvolvimento da metalinguagem Mar e sua integração com o MOBI, assim como o processo de conversão de seus artefatos, os seguintes Requisitos Funcionais (RQF) foram especificados para o MoPE:

- **RQF1**: Prover um Ambiente Integrado para construção de software onde a Modelagem é realizada a partir do MOBI;
- **RQF2**: Prover um Ambiente de Programação e uma Metalinguagem compatíveis com o MOBI;
- **RQF3**: Construir um Modelo de Negócios a partir da modelagem realizada no MOBI e das regras de negócio escritas em Mar;
- **RQF4**: Permitir a exportação do Modelo de Negócio por meio de um PSM.

O diagrama de casos de uso do MoPE (Figura 4.1) descreve como seria a interação de um Analista de Sistema com o MoPE. Esse analista tem como papais descrever o cenário exemplo (RQF1), implementar as regras de negócio da futura aplicação (RQF2) e solicitar ao modelo a exportação do modelo de negócios (RQF4).

Ao passo que um Analista de Sistemas descreve o cenário exemplo, o MOBI constrói o modelo de classes por inferência [Jorge et al. 2012]. Da mesma forma, o MoPE deve construir o modelo de negócios (RQF3) a partir do modelo de classes do MOBI e da lógica escrita em metalinguagem Mar. Tendo como base o cenário exemplo descrito pelo Analista de Sistemas, o processo de geração desse modelo de negócios realiza algumas inferências no Ambiente de Programação assim como no Ambiente de Modelagem.

Seguindo a MDA, esse Modelo de Negócio é o nosso PIM (Modelo Independente de Plataforma), e a partir dele é gerado o PSM (Modelo Específico de Plataforma). Essa conversão do PIM em PSM permite que o Modelo de Negócio possa ser utilizado por outros programadores em outras plataformas para construção de sistemas específicos (RQF4), como por exemplo, um sistema web que utilizou um Modelo de Negócio construído no MoPE para controlar o estoque de remédios, ou uma aplicação *mobile* que utilizou um Modelo de Negócio para cálculo do imposto de renda.

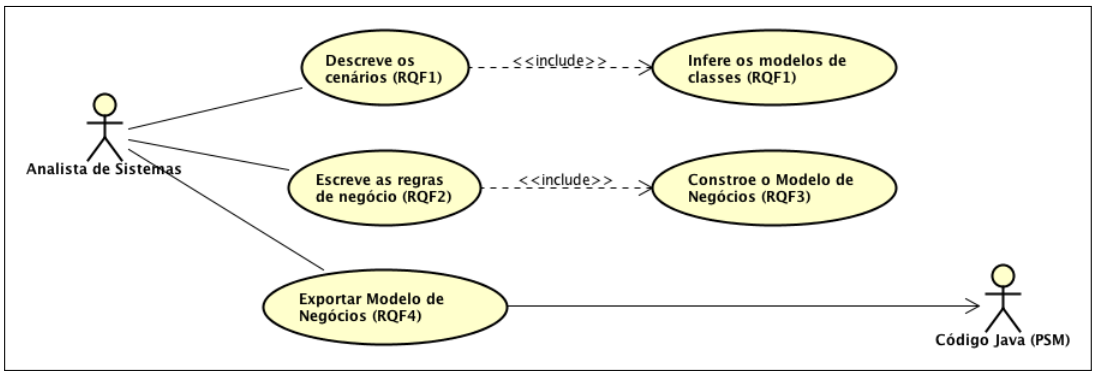


Figura 4.1: Diagrama de Casos de Uso do MoPE.
Fonte: Autor.

4.2.1 Processo de Conversão

O processo de conversão do MoPE segue as especificações da MDA. Uma espécie de PIM formado pelo modelo conceitual, construído no MOBI, e as regras de negócio, implementadas com Mar, são convertidos em um PSM, que pode ser utilizado em uma outra conversão para uma outra plataforma específica, como a linguagem *TypeScript* ou *C#* (RQF4). A Figura 4.2 ilustra esse fluxo no processo de conversão.

Neste caso, o PSM é utilizado como uma linguagem intermediária, pois tem como obje-

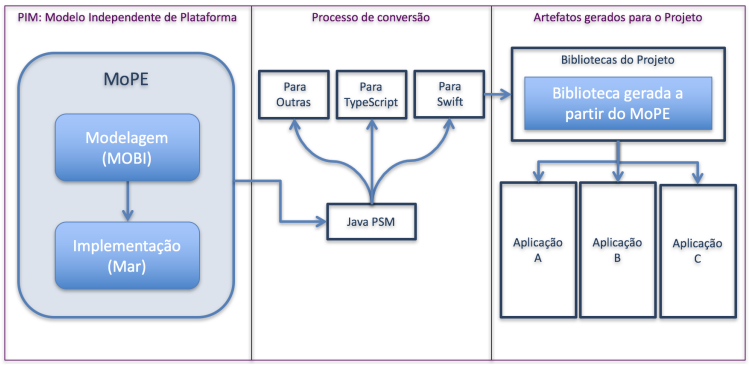


Figura 4.2: Processo de Conversão do MoPE.
Fonte: Autor.

tivo simplificar o processo de conversão para outras plataformas Orientadas a Objetos, tendo de possuir uma sintaxe equivalente com a de uma linguagem Orientada a Objetos. Outro aspecto importante, é que as inferências lógicas serão realizadas pelo compilador na etapa de conversão, assim a sintaxe gerada nessa linguagem intermediária (PSM) se torna viável de conversão, em outras linguagens Orientadas a Objetos. O MoPE utiliza em sua especificação a linguagem (Orientada a Objetos) *Java* como PSM.

No processo de conversão para uma plataforma específica, o Java PSM deve ser gerado como um artefato não editável (biblioteca), o programador poderá apenas utilizar as classes e os métodos construídos no MoPE, mas não editá-los. De forma que qualquer alteração nas principais regras ou no modelo de negócio do projeto deva ser realizada a partir do MoPE, mantendo a consistência e a sincronia entre o que foi especificado na modelagem e construído na implementação.

4.3 Arquitetura do MoPE

A arquitetura do MoPE é composta basicamente pelos ambientes de modelagem e de programação, e o seu compilador, que atua sobre os cenários construídos no MOBI juntamente com a lógica descrita com Mar. Essa arquitetura tem duas características principais: ambiente de modelagem e programação funcionando de forma integrada; e processo de construção baseado em cenários com exemplos de referência (RQF1 e RQF2).

A maioria dos modelos de programação realiza o processo de modelagem separado do processo de programação, mesmo os que envolvem a construção de um modelo conceitual (como o diagrama de classes) e são imprescindíveis para a implementação. Esse tipo de prática gera retrabalho, pois sempre que ocorrer alterações no código implementado a modelagem do sistema precisará ser atualizada, ou vise e versa.

No MoPE o ambiente de implementação está conectado diretamente com o de modelagem, de forma que o código de programação reconhece e reutiliza todos os recursos definidos na modelagem do sistema. Se, por exemplo, já existe a definição de uma determinada classe de negócio no modelo o programador não precisa recriá-la em seu código. O modelo deve permitir que o ambiente de programação acesse todos os recursos pertinentes já criados lá, evitando duplicação de estruturas ou até mesmo atualizações desnecessárias.

A característica fundamental desta arquitetura é o uso do princípio da indução, ou seja, a partir de exemplos de programação realizados com instâncias de um determinado domínio, um código fonte pode ser gerado em uma linguagem contendo suas classes, atributos e métodos.

Além disso, como o MOBI é utilizado como base para a modelagem de classes do modelo de programação proposto, a modelagem de classes é feita a partir da ideia de cenários do MOBI, onde instâncias são relacionadas entre si e um compilador constrói suas estruturas e definições.

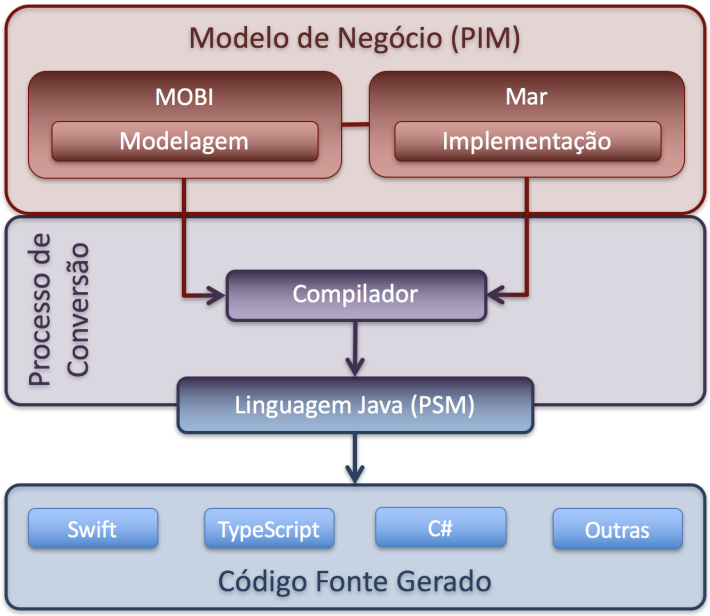


Figura 4.3: Arquitetura do MoPE.
Fonte: Autor.

Como representado na Figura 4.3 a arquitetura define uma Ambiente de Modelagem integrado com o de Programação, onde o MOBI é utilizado para construir o modelo e a metalinguagem Mar para a implementação das regras de negócio. O Conversor é o responsável pela conversão do Modelo de Negócio (modelagem e regras de negócios) em um PSM, mais fácil de ser processada em uma futura conversão para outra linguagem de programação, como por exemplo *C#*, *Swift* ou *TypeScript*.

Esse ambiente de modelagem permite ao usuário criar e definir as estruturas das classes assim como suas relações. O ambiente de programação tem visibilidade dessas classes, dispensando ao usuário programador recriar com a linguagem de programação as classes modeladas. Mais do que isso, qualquer classe necessária no modelo deverá ser criada no ambiente de modelagem, a partir dos cenários. Eliminando assim redundâncias que necessitem de atualização ou sincronização posterior.

Os atributos das instâncias devem ser definidos no ambiente de modelagem, a partir dos cenários. Já os métodos serão codificados, na metalinguagem Mar, dentro do ambiente de implementação. Com isso, pretende-se aumentar a simplicidade do modelo e reduzir a curva de aprendizagem, fazendo com que a programação não aborde aspectos ligados a modelagem e vice-versa.

Como o cenário é montado com base em exemplos, pode-se afirmar que toda a estrutura de classes, ou semântica do projeto de software, será pautada no exemplo modelado. Assim como uma modelagem malfeita pode prejudicar o projeto, um exemplo mal descrito também. Porém, a vantagem de se basear a modelagem do software em exemplos do mundo real é que se o exemplo mudar a modelagem muda, e por consequência o software também, pois a implementação está pautada na modelagem.

A Figura 4.4 ilustra a separação do escopo abordado pelo MoPE. O resultado da conversão deve sempre representar o modelo de negócio da aplicação e poder ser importado e utilizado em qualquer tipo de projeto de software orientado a objetos. A implementação do MoPE realizada nesta pesquisa só contempla a conversão do PSM para uma plataforma alvo, a plataforma Java, que foi implementada para viabilizar os testes com o experimento prático realizado (Capítulo 5).

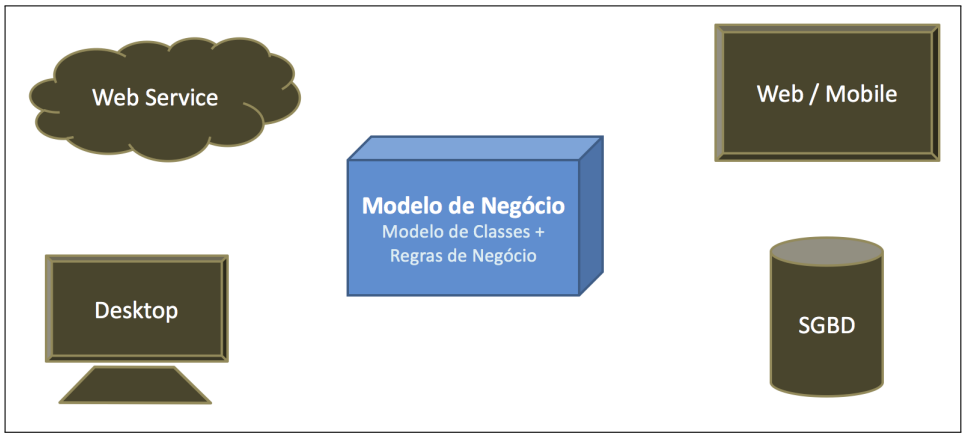


Figura 4.4: Integração do Modelo de Negócio gerado pelo MoPE.
Fonte: Autor.

Também é válido ressaltar que lógicas específicas de plataforma, como a construção de um *web service*, um sistema *web* ou *desktop*, ou ainda um comando para um banco de

dados (SGBD), não são suportadas no ambiente de programação do MoPE, o objetivo do mesmo é apenas a implementação das regras de negócio de um sistemas, genéricas o suficiente para serem utilizadas em qualquer plataforma. Se o objetivo é a construção de uma aplicação *mobile*, por exemplo, o modelo de negócios gerado pelo MoPE deve ser importado em um projeto específico deste tipo e utilizado lá como uma biblioteca. Se uma mudança na modelagem ou nas regras de negócio é necessária, deve ser feita na plataforma integrada do MoPE e depois gerada novamente para a plataforma específica.

Todas essas características tornam a mudança estrutural do software menos impactante, uma plataforma de desenvolvimento que siga essas especificações estaria preparada para alterar a modelagem automaticamente quando o modelador mudar o exemplo. Neste ponto, o MOBI entra para refazer a modelagem com base no novo cenário descrito. Vale ressaltar que um dos principais problemas, ainda presentes e identificados na chamada Crise do Software, é o alto impacto causado por mudanças no decorrer do projeto [Naur 1969].

Para melhor entendimento dos conceitos da metalinguagem a próxima seção (4.4) apresenta 2 (dois) exemplos de como implementar em Mar, na Seção 4.5 a lógica por trás desse funcionamento é explicada e detalhada.

4.4 Como Desenvolver um Modelo de Negócio no MoPE

Esta seção descreve os passos para construção de um exemplo de Modelo de Negócio por meio do MoPE. Esse Modelo de Negócio é voltado ao controle de pedidos de venda.

O primeiro passo é a construção dos cenários que vinculam os exemplos de referência. A Figura 4.5 ilustra os 4 (quatro) cenários construídos, o cenário do Quadrante 1 modela os atributos que a classe *Produto* possui, neste caso, *codigo nome* e *preco*. No Quadrante 2, os produtos são vinculados a um item do pedido, que por sua vez são associados a uma quantidade, no Quadrante 3. O *Item* indica quantas unidades daquele produto estão no pedido de venda, isso é definido a partir da modelagem realizada no Quadrante 4 onde um exemplo da classe *Pedido*, o Pedido de Natal, é vinculado com alguns dos exemplos da classe *Item*, as linhas 1 e 2 que se referem a 2 monitores e 1 Notebook respectivamente.

Com esses cenários descritos, o MOBI já pode inferir que a classe *Produto* possui 3 (três) atributos (um código do tipo numérico, um nome do tipo texto e um preço do tipo decimal). A classe *Item* por sua vez possui relacionamento com apenas 1 (um) *Produto* e com 1 (um) valor numérico que indica a quantidade desse produto. E a classe *Pedido* possui relacionamento com 1 (um) ou mais elementos do tipo *Item*.

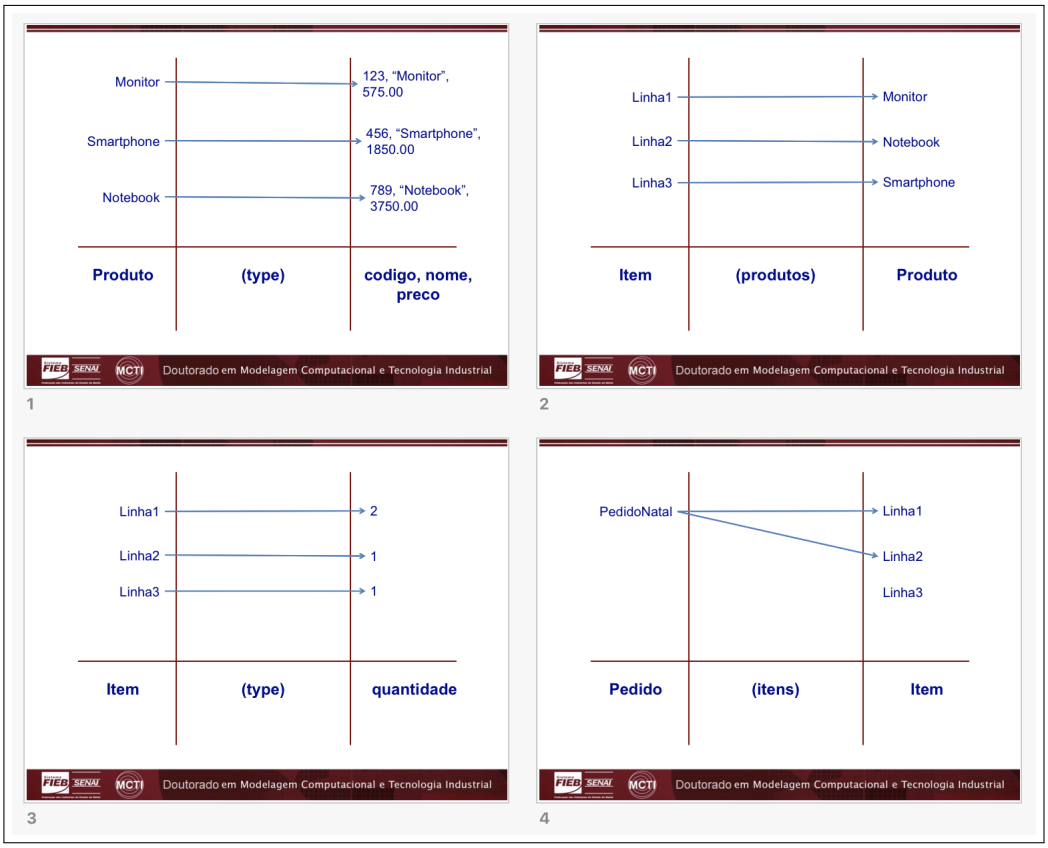


Figura 4.5: Cenários para Modelagem do Pedido de Venda.
Fonte: Autor.

O segundo passo é a implementação das regras de negócio necessárias a futura aplicação para Registro de Pedidos de Venda. Neste exemplo, tem-se a necessidade de implementação de 2 (dois) métodos, uma para calcular o valor total de um item e a outra para calcular o valor total do pedido. Essas implementações devem ser realizadas no ambiente de programação que já reconhece e fornece todas as estruturas descritas nos cenários e inferidas pelo MOBI.

Para realizar a implementação do método *total* na classe *Item*, o Analista de Sistema deve primeiro escrever o nome de uma das instâncias da classe *Item*, neste caso foi a instância *Linha1* (ver linha 2 da Figura 4.6). Depois o mesmo deve escrever o nome do método uma linha abaixo e com uma tabulação a mais em relação ao nome da instância (ver linha 3 da Figura 4.6), isso indica que o método pertence aquela instância, da mesma forma que a implementação (na linha 4) também pertence ao método *total* já que também tem uma tabulação a mais do que o nome do método na linha anterior.

A implementação escrita (na linha 4) multiplica o valor da variável *preco* da instância *Monitor* com o valor da variável *quantidade*, ao mesmo tempo que indica esse resultado como o retorno do método *total*. Como a instância *Monitor* está relacionada com a instância *Linha1*, no Quadrante 2 da Figura 4.5, o compilador consegue inferir que o

valor total de um item sempre será a multiplicação do preço do produto associado a esse item e a sua quantidade.

```
1
2 Linha1
3         total
4                 return Monitor_preco * quantidade
5
```

Figura 4.6: Implementação da classe Item para o exemplo Pedido de Venda.
Fonte: Autor.

Para a implementação do método *total* da classe *Pedido*, o Analista de Sistemas precisaria escrever um código que percorre-se todos os itens de um pedido, somando o total de cada um deles, para depois retornar o resultado desse somatório. Com a metalinguagem Mar, o Analista de Sistema pode retornar apenas o somatório dos preços do item *Linha1* e do item *Linha2*, como ilustrado na Figura 4.7. Como as instâncias *Linha1* e *Linha2* são as únicas relacionadas a instância *PedidoNatal*, o compilador do MoPE consegue inferir que uma vez que todos os elementos de uma relação fazem parte de uma operação, essa operação deve ser replicada a todos os elementos quando o PSM for gerado (mais detalhes na Subseção 4.5.1).

```
1
2 PedidoNatal
3         total
4                 somatorio = (Linha1 total) + (Linha2 total)
5                 return somatorio
6
```

Figura 4.7: Implementação da classe Pedido para o exemplo Pedido de Venda.
Fonte: Autor.

Juntos, esse modelo de classes obtido a partir dos exemplos descritos nos cenários do MOBI, e essas regras de negócio escritas a partir da metalinguagem Mar, formam o Modelo de Negócio desse sistema exemplo para Registro de Pedidos. O PSM gerado a partir desse Modelo de Negócio é apresentado na Figura 4.8, destaque para o código gerado do método *total* na classe *Pedido* (linha 14 a 21). A partir da lógica de inferência explicada acima, é gerado um código que percorre todos os itens da lista somando o total de cada um deles.

As inferências no MoPE são realizadas a partir do cenário exemplo, se este muda, muda-se também o resultado da inferência. Supondo que o cenário exemplo do Quadrante 4 (Figura 4.5) seja alterado, com o vinculo adicional da instância *Linha3* a instância *PedidoNatal*, e mantendo-se a implementação do método *total* da classe *Pedido* (Figura 4.7) o resultado da inferência também mudará.

```java
package br.com.pedido.mope.model;


public class Produto {

    private Integer codigo;
    private String nome;
    private Double preco;

    public Integer getCodigo() {
        return codigo;
    }

    public void setCodigo(Integer codigo) {
        this.codigo = codigo;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public Double getPreco() {
        return preco;
    }

    public void setPreco(Double preco) {
        this.preco = preco;
    }
}
```

```java
package br.com.pedido.mope.model;


public class Item {

    private Integer quantidade;
    private Produto produto;

    public Integer getQuantidade() {
        return quantidade;
    }

    public void setQuantidade(Integer quantidade) {
        this.quantidade = quantidade;
    }

    public Produto getProduto() {
        return produto;
    }

    public void setProduto(Produto produto) {
        this.produto = produto;
    }

    public Double total() {
        return this.produto.getPreco() * this.quantidade;
    }
}
```

```java
package br.com.pedido.mope.model;

import java.util.ArrayList;
import java.util.List;

public class Pedido {

    private final List<Item> itens = new ArrayList<Item>();

    public List<Item> getItens() {
        return itens;
    }

    public Double total() {
        Double somatorio = 0.00;

        for (Item item: this.itens) {
            somatorio = (somatorio+item.total());
        }
        return somatorio;
    }
}
```

Figura 4.8: Classes Java geradas no exemplo Pedido de Venda.
Fonte: Autor.

Na Figura 4.9 tem-se uma ilustração da mudança citada, onde a instância *PedidoNatal* está vinculada às 3 (três) instâncias, *Linha1*, *Linha2* e *Linha3*. Neste caso, a implementação do método *total* da classe *Pedido* passa a se referir apenas aos 2 primeiros elementos da lista, mudando o resultado do código gerado no PSM (ver Figura 4.10).

4.5 Mar: a metalinguagem de programação baseada em exemplos

A metalinguagem de programação Mar foi desenvolvida para codificar lógica de programação em cenários construídos pelo MOBI. A independência de suas instâncias em relação as classes é um dos importantes aspectos que este modelo propõe (RQF2). Na maioria das linguagens de programação orientadas a objetos só é possível criar uma instância a partir de uma classe, o que amarra a instância a classe e não permiti que esta assuma outras classificações dentro do modelo. Isso pode acarretar em problemas de modelagem como citado na introdução deste documento.

O MoPE propõe que a instância seja independente de classes, podendo ser criada sem a necessidade dessa classificação inicial. Outra a abordagem, seria a classificação da instância em qualquer classe, permitindo agregar novas funcionalidades ou características. Desta maneira, as instâncias poderiam agregar funcionalidades dentro do sistema. Como no problema do Professor de graduação que resolve ingressar como aluno em um curso de pós-graduação na mesma universidade que ele leciona. Neste caso, a instância no sistema se comportaria de forma similar ao indivíduo, agregando as funcionalidades da classe Professor, sem necessitar a criação de outra instância.

Outra facilidade da independência da instância é o compartilhamento dos atributos da

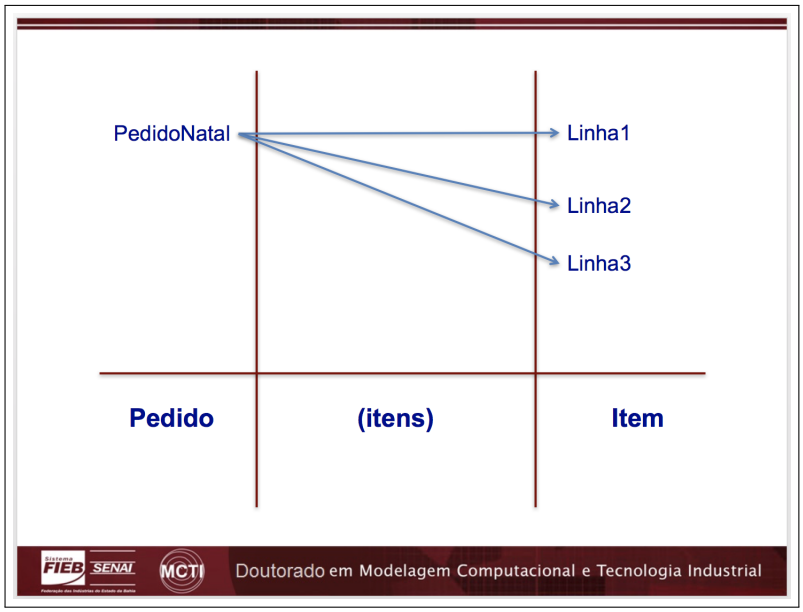


Figura 4.9: Variação do exemplo Pedido de Venda.
Fonte: Autor.

```
1  package br.com.pedido.mope.model;
2
3  import java.util.ArrayList;
5
6  public class Pedido {
7
8      private final List<Item> itens = new ArrayList<Item>();
9
10     public List<Item> getItens() {
11         return itens;
12     }
13
14     public Double total() {
15         Double somatorio = this.itens.get(0).total() + this.itens.get(1).total();
16         return somatorio;
17     }
18
19 }
20
```

Figura 4.10: Classe gerada com a alteração no exemplo Pedido de Venda.
Fonte: Autor.

instância, como nome, data de nascimento, CPF, RG, endereço entre outros. Se a instância referente ao Professor já possuía atributos próprios como nome ou CPF esses podem ser utilizados pelos métodos da nova classe Aluno.

Para viabilizar isso, o modelo prevê que os atributos serão vinculados às instâncias, enquanto os métodos, às classes. Desta forma, um indivíduo não poderá possuir dois atributos de mesmo nome, mas poderá executar dois ou mais métodos de mesma assinatura, já que estes estarão vinculados a classe e serão distinguidos a partir dela. Assim, características vantajosas da orientação a objetos serão mantidas, como a sobrecarga e o polimorfismo.

Além das inferências já realizadas pelo MOBI [Jorge et al. 2012] no Ambiente de Modelagem, o MoPE também realiza inferências em seu Ambiente de Programação. Os detalhes

desses tipos de inferências e da lógica de conversão estão descritos na próxima seção.

4.5.1 Compilador

O compilador do MoPE utiliza 2 (dois) componentes responsáveis pela conversão do modelo de classes, gerado pelo MOBI, e do código escrito em linguagem Mar. Esses 2 (componentes) trabalham de forma complementar, o Motor de Inferência é responsável por processar os aspectos de inferência da metalinguagem Mar, enquanto o Conversor se encarrega de fornecer os artefatos gerados no MOBI para o Motor de Inferência, assim como direcionar o processo de conversão como um todo. O Motor de Inferência da metalinguagem Mar é parte integrante do Conversor que por sua vez é o único a acionar diretamente o motor.

Ambos os componentes têm como objetivo principal construir o Modelo de Negócio resultante da modelagem e da implementação descritos no MoPE. Para viabilizar a construção desse Modelo de Negócios (RQF3), além de todas as características detalhadas neste capítulo, também foram definidos cinco tipos de conversão e duas regras lógicas. Os tipos de conversão são algumas formas de processamento e conversão para a sintaxe da linguagem, enquanto as regras determinam em quais casos podem ocorrer inferência na metalinguagem Mar.

4.5.1.1 Tipo de Conversão 1 - Endentação

A endentação do código *Mar*, feita a partir de tabulação, é utilizada para definir as marcações de início e fim dos blocos de comandos. A maioria das linguagens de programação utiliza marcadores, como *begin* e *end* ou as chaves ({ }), para isso.

A endentação de um comando em relação a outro é o que define o início e fim dos blocos de comandos escritos em Mar. Observa-se na Figura 4.7, que todo código escrito está endentado em relação a palavra *Linha1*, isso indica que esse código pertence a classe *Item*, uma vez que a instância *Linha1* está classificada como *Item*. O mesmo ocorre com a implementação do método *total* (linha 3), como a linha abaixo (4) está com uma tabulação a mais em relação ao método o compilador define essa linha como parte integrante da implementação do método.

Para maiores detalhes de implementação dessa parte da inferência ver no Apêndice A.2 o método *breakInBlocks* na linha 788 da classe *Mar.java*.

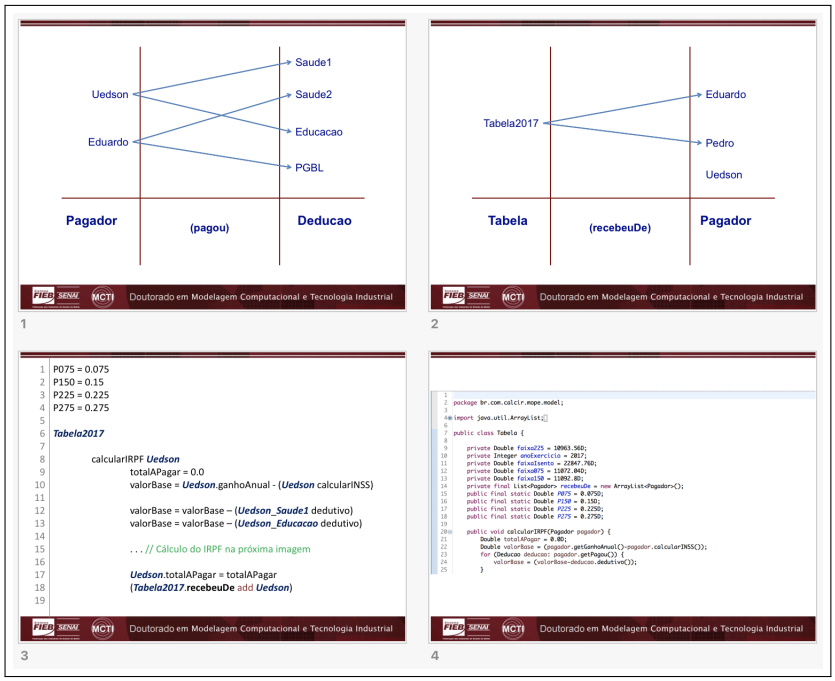


Figura 4.11: Exemplo Imposto de Renda.
Fonte: Autor.

4.5.1.2 Tipo de Conversão 2 - Declaração de variáveis locais

A declaração de uma variável pode ser realizada no momento em que a mesma tem um valor atribuído a si. Na linha 9 (Quadrante 3) da Figura 4.11 tem-se um exemplo, onde a variável *totalAPagar* recebe um valor (0.0) como um número fracionário. Por conta dessa atribuição, o seu tipo foi inferido como *Double* na linguagem *Java* (Quadrante 4).

Para maiores detalhes de implementação dessa parte da inferência ver no Apêndice A.2 o método *whichClass* na linha 596, e o trecho entre as linhas 503 e 516, da classe *Mar.java*.

4.5.1.3 Tipo de Conversão 3 - Tipo de retorno dos métodos

Similar aos tipos de variáveis, o tipo de retorno de um método também pode ser inferido. O compilador considera sempre o tipo de valor utilizado como retorno no método para definir o seu tipo. Na linha 5 da Figura 4.7, por exemplo, o comando retorna uma variável do tipo *Double* indicando que o tipo de retorno desse método também é *Double*, ver Figura 4.8.

Para maiores detalhes de implementação dessa parte da inferência ver no Apêndice A.2 o método *getClassType* na linha 656, e o método *createMethod* na linha 318, da classe *Mar.java*.

4.5.1.4 Tipo de Conversão 4 - Variáveis estáticas

O nome da instância, escrito em código Mar, delimita o inicio do bloco de código pertencente a classe dessa instância, como visto nas linhas de 6 a 18 no Quadrante 3 da Figura 4.11. Caso uma variável seja escrita antes do nome de uma instância esta será inferida como estática, uma vez que está fora do escopo da instância, ver linhas de 1 a 4.

Para maiores detalhes de implementação dessa parte da inferência ver no Apêndice A.2 o trecho entre as linhas 98 e 134 da classe *Mar.java*.

4.5.1.5 Tipo de Conversão 5 - Declaração de variáveis globais na implementação

Da mesma forma que é possível declarar uma variável local apenas atribuindo valor a mesma, também é possível declarar uma variável global seguindo a mesma lógica, a diferença é que neste caso a variável deve ser referencia da a partir de uma instância.

Como exemplo tem-se a variável *valorFaixaIsento* na linha 35 da Figura 5.19 (página 58) que acessada a partir da instância *Uedson*, o que permite ao MoPE inferir que a variável pertence ao escopo global da classe *Pagador*, uma vez que *Uedson* é uma instância desse tipo. O resultado dessa inferência pode ser visto na linha 36 da Figura 5.20 (página 59).

Para maiores detalhes de implementação dessa parte da inferência ver no Apêndice A.2 o método *createIfNotExist* na linha 254 da classe *Mar.java*.

4.5.1.6 Regra 1 - Para todo elemento da relação

Sempre que todas as instâncias de uma relação forem utilizadas em uma operação é possível inferir a necessidade de todos os elementos da lista. O exemplo descrito na Seção 4.4, onde é necessária a implementação lógica de um método para calcular o valor total de um pedido de venda com base nos itens desse pedido. Com apenas 2 (dois) itens lançados no cenário do MOBI, basta calcular o somatório destes 2 (dois) itens e o MoPE infere que o total de um pedido equivale ao somatório de todos os seus itens.

No exemplo citado acima, temos 2 (dois) métodos com o nome de *total*, uma para a classe *Item* e outro para a classe *Pedido*. O método da classe *Item* possui uma implementação mais simples, apenas multiplica o preço do produto pela quantidade (do item), o resultado é o total daquele item. Já no método *total* de *Pedido* temos o somatório do *total* dos 2

(dois) itens associados a este pedido no cenário exemplo, a partir disso é realizada a inferência que gera um código para percorrer todos os elementos da lista, acumulando seu valor total, já que "todos"os itens somados estão formando o valor total do pedido (de natal). É possível visualizar na Figura 4.8 a mesma implementação de ambos os métodos *total* (em *Item* no 2º quadro e em *Pedido* no 3º) geradas em Java PSM.

Para maiores detalhes de implementação dessa parte da inferência ver no Apêndice A.2 o método *convertFor* na linha 545 da classe *Mar.java*.

4.5.1.7 Regra 2 - Se o elemento já pertence a relação

A metalinguagem Mar permite a escrita de 2 (dois) métodos com a mesma assinatura desde que utilizem instâncias diferentes nos parâmetros recebidos. Sempre que isso ocorrer o compilador montará uma estrutura condicional onde o código a ser executado depende se a instância pertence ou não a relação da classe.

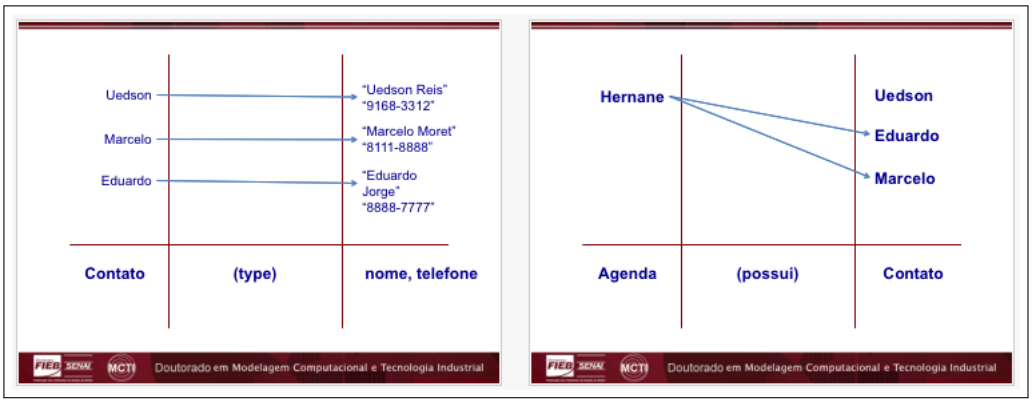


Figura 4.12: Modelagem das classes do exemplo Agenda.
Fonte: Autor.

Para explicar melhor a regra, um exemplo envolvendo uma agenda de contatos telefônicos foi construído. Na Figura 4.12 tem-se a modelagem das classes *Contato* (esquerda) e *Agenda* (direita), onde um contato possui *nome* e *telefone*, e uma agenda se relaciona com pelo menos 1 (um) contato.

A Figura 4.13 demonstra a implementação da funcionalidade adicionar, onde um contato só é registrado se não tem associação com aquela agenda, para evitar duplicidades. Pode-se notar dois métodos com o nome *adicionar*, um deles recebendo como parâmetro a instância *Uedson* e no outro *Eduardo*, onde não se aplica o conceito de sobrecarga, pois as instâncias pertencem a mesma classe (*Contato*). Sempre que o compilador do MoPE processar esse tipo de assinatura de método irá verificar as relações em que essas instâncias estão associadas, havendo diferenças uma regra será inferida a partir dela.

```
1  Hernane
2
3          adicionar Uedson
4                  (Pessoal_possui add Uedson)
5                  return YES
6
7          adicionar Eduardo
8                  return NO
9
10         remover Eduardo
11                 (Pessoal_possui remove Eduardo)
12                 return YES
13
14         remover Uedson
15                 return NO
16
```

Figura 4.13: Métodos *adicionar* e *remover* da classe *Agenda* escritos em Mar.
Fonte: Autor.

No cenário exemplo descrito no Quadrante 4 da Figura 4.12, a *Agenda Hernane* não *possui* o *Contato Uedson*, mas *possui* o *Contato Eduardo*. Por isso, o compilador utilizará a implementação do método *adicionar Uedson* sempre que o *Contato* passado como parâmetro não estiver associado a instância da *Agenda*. Assim como, sempre que um *Contato* já associado for passado como parâmetro a implementação utilizada será a do método *adicionar Eduardo*.

Desta forma, só contatos não associados podem ser associados a uma agenda, evitando duplicidade, assim como só contatos associados podem ser desassociados (regra do método *remover*). A Figura 4.14 mostra o código gerado pelo Conversor do MoPE em linguagem Java (o PSM do MoPE).

Para maiores detalhes de implementação dessa parte da inferência ver no Apêndice A.2 o método *createObjectMethods* na linha 289 da classe *Mar.java*.

4.5.2 Sintaxe

Para especificar a metalinguagem de programação Mar, foi utilizado o padrão EBNF (*Extended Backus–Naur Form*). Os elementos abaixo representam o núcleo da metalinguagem utilizado na construção das aplicações apresentadas no Experimento Prático desta pesquisa (Capítulo 5). Estas aplicações foram desenvolvidas com o objetivo de avaliar o modelo proposto e para melhor entendimento das especificações da metalinguagem uma destas aplicações é utilizada como exemplo.

```java
package br.com.agenda.mope.model;


public class Contato {

    private String nome;
    private String telefone;

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getTelefone() {
        return telefone;
    }

    public void setTelefone(String telefone) {
        this.telefone = telefone;
    }

}
```

```java
package br.com.agenda.mope.model;

import java.util.ArrayList;
import java.util.List;

public class Agenda {

    private final List<Contato> possui = new ArrayList<Contato>();

    public List<Contato> getPossui() {
        return possui;
    }

    public boolean adicionar(Contato contato) {
        if (this.getPossui().contains(contato)) {
            return false;
        } else {
            this.getPossui().add(contato);
            return true;
        }
    }

    public Boolean remover(Contato contato) {
        if (this.getPossui().contains(contato)) {
            this.getPossui().remove(contato);
            return true;
        } else {
            return false;
        }
    }

}
```

Figura 4.14: Métodos *adicionar* e *remover* da classe *Agenda* gerados em linguagem Java. Fonte: Autor.

Conforme visto anteriormente, a Figura 5.4 apresenta a implementação da instância CajazeirasIV, pertencente a classe *Posto* (Posto de Saúde). Vale ressaltar que esta implementação é um complemento do modelo construído no Ambiente de Modelagem (MOBI), e está totalmente descrito na Seção 5.1.

Esse primeiro bloco de código (abaixo) define elementos gerais ou mais utilizados pela linguagem. Como os identificadores dos elementos, a quebra de linha que marcará o fim de um comando, e o espaçamento de endentação que indicará se um comando pertence ou não a um determinado bloco.

```
<qualquer> ::= Qualquer elemento definido
<identificador> ::= texto
<nome_*> ::= <identificador>
<tipo_primitivo> ::= <identificador>
<quebra_linha> ::= Quebra de linha
<tab> ::= Espaçamento de endentação
<valores> ::= número | texto | booleano
```

No bloco de código abaixo, tem-se a definição de elementos importantes, Classe e Instância. Onde Classe está definida pela composição de atributos e métodos (também definidos mais a seguir) e uma Instância como um elemento que possui um identificador (o nome da instância) e os elementos definidos nas Classes relacionadas a esta. A Figura 5.4 mostra

os métodos estão estruturados dentro de uma classe, onde apenas são utilizadas quebras de linhas e tabulações para definir quais métodos pertencem a mesma.

```
<Classe> ::= <nome_classe>
        {<quebra_linha> <tab> Atributo {<quebra_linha> <tab> Atributo}}
        {<quebra_linha> <tab> Metodo {<quebra_linha> <tab> Metodo}}
<Instancia> ::= <nome_instancia> {: <Classe> {, <Classe>}}
<instancias> ::= [<nome_instancia>, <nome_instancia> {, <nome_instancia>})]
```

Nesta pesquisa, definimos o elemento Variável apenas como aquele que armazena valores de tipos primitivos (eg.: texto, número, verdadeiro ou falso). O elemento Atributo é utilizado para se referir de forma genérica tanto a uma variável quanto a uma Instância. De forma similar, o elemento Tipo pode se referir tanto a uma Classe quanto a um Tipo Primitivo. Quanto ao elemento Relação, assim como no MOBI, este é definido pela referência de uma Instância para com um Atributo seu, como no exemplo da Figura 5.4 nas linhas 4 e 14 (*CajazeirasIV*_possui). O bloco de código abaixo é a formalização dessas definições.

```
<Variavel> ::= Variavel(<nome_variavel>, <valores>)
<tipo> ::= <Classe> | <tipo_primitivo>
<atributo> ::= <nome_instancia> | <nome_variavel>
<relacao> ::= ( <nome_instancia>"_"<nome_instancia>
            | <nome_instancia>"_"<instancias> )
```

Assim como no exemplo dos métodos dentro da classe, qualquer bloco de código é considerado como pertencente ou não a um estrutura com base em sua endentação, não existindo o uso de palavras-chave para marcar o início ou o fim dos blocos de comando, como visto na Figura 5.4. Da mesma forma, uma linha de comando é finalizada com uma quebra de linha e não com uma determinada palavra-chave. Na linguagem Mar os conceitos para construção de um método são quase os mesmos da Orientação a Objetos, com os elementos Nome e Parâmetros formando sua Assinatura. A Chamada do Método é feita entre parenteses e separando com espaços o Atributo, ou Relação, do Nome do Método e seus Parâmetros. Não é necessário informar o tipo de retorno do método pois o mesmo é inferido a partir do valor retornado na implementação. Como parâmetros de entrada podem ser utilizadas instâncias definidas no ambiente de modelagem. A formalização desses detalhes está no bloco de código abaixo.

```
<bloco> ::= <tab> <qualquer> <quebra_linha>
```

```
            {<tab> <qualquer> <quebra_linha>}
<parametros> ::= <atributo> { <atributo>}
<assinatura_metodo> ::= <nome_metodo> { <parametros> } <quebra_linha>
<retorno> ::= "return" <expressao> <quebra_linha>
<metodo> ::= <assinatura> <parametros> <bloco> {<retorno>}
<chamada_metodo> ::= "[" (<atributo> | <relacao>) " " <nome_metodo>
                  {" " <parametros>}"]"
```

Por fim, no bloco de código abaixo, são definidos elementos de representação para as operações básicas, atribuições e comparações de valor, assim como a estrutura condicional *IF* (utilizada para executar um determinado bloco de comando a depender do valor de um Atributo).

```
<operador> ::= ‘+’ | ‘-‘ | ‘/’ | ‘*’ | ‘^’
<operacao> ::= numero <operador> numero { <operador> numero }
<expressao> ::= <operacao> | texto | numero | <chamada metodo> | <atributo>
<atribuicao> ::= (<atributo> | <relacao> ) ‘=’ <expressao>
<comparacao> ::= <expressao> (‘==’ | ‘>’ | ‘<’ | ‘<=’ | ‘>=’ | ‘!’)
             <expressao> { (and | or) <comparacao>}
<head_if> ::= ‘if’ <comparacao> <quebra_linha>
<else> ::= "else"
<if> ::= <head_if> <bloco>
        {<else> <head_if> <bloco>}
        {<else> <quebra_linha> <bloco>}
```

Vale ressaltar que para maiores detalhes sobre a implementação do modelo proposto, o código fonte responsável pelo compilador do MoPE está disponível no apêndice desde documento.

Para avaliar e testar a viabilidade das ideias apresentadas nesta pesquisa foi definido um Experimento Prático que consiste no desenvolvimento de duas aplicações utilizando o MoPE. Ambas as aplicações estão publicadas, uma na web (https://mymedicine.heroku-app.com) e a outra em uma loja virtual como um aplicativo para dispositivos móveis (https://play.google.com/store/apps/details?id=add.calc_ir), e podem ser acessadas e utilizadas a qualquer momento. No Capítulo 5, o Experimento Prático é apresentado, assim como a avaliação do modelo proposto.

Capítulo Cinco

Experimento Prático

Seguindo a metodologia proposta neste trabalho (Seção 1.6), a qual prevê a resolução (por parte do modelo) de um conjunto de problemas lógicos, este Experimento Prático foi idealizado. O objetivo é avaliar e adequar o modelo ao mesmo tempo que se verifica a sua viabilidade.

Este experimento consiste na construção de 2 (duas) aplicações práticas, uma *web* e outra *mobile* (voltada para dispositivos móveis). O MoPE foi utilizado para construção das regras de ambas as aplicações, não tendo como objetivo a construção de interfaces com o usuário ou conexões com bases de dados.

A primeira aplicação desenvolvida foi um sistema web para gerenciamento do estoque de remédios em diversos postos de saúde, denominado *Remédio Aqui* e pode ser acessada via internet no endereço http://mymedicine.herokuapp.com.

5.1 Estoque de Remédios em Postos de Saúde

Inicialmente, a ideia desta aplicação é prover uma rápida notificação com relação a falta de um remédio em algum dos postos de saúde da prefeitura da cidade de Salvador (Bahia, Brasil). Um agente pode acessar o sistema e informar que um determinado remédio está em falta. Em um segundo momento, algumas novas funcionalidades serão demandadas o que justifica uma modelagem mais abrangente até esta etapa, porém esta pesquisa ficou focada nesta ideia inicial do sistema, detalhada nos requisitos a seguir.

5.1.1 Requisitos

Os requisitos levados em consideração para o desenvolvimento desta aplicação foram divididos em 2 (dois) tipos, os Funcionais (RQF) e os Não Funcionais (RNF), e estão descritos abaixo:

RQF1 - Prover base de dados para armazenamento das informações dos Postos e dos Remédios; RQF2 - Implementar um módulo servidor (*backend*) que centralize as regras de negócio da aplicação; RQF3 - Implementar uma aplicação web (*frontend*) para prover a interface com o usuário; RQF4 - Prover a listagem de todos os Postos cadastrados; RQF5

- Prover a apresentação dessa listagem também em um Mapa; RQF6 - Permitir ao usuário informar se tem ou não tem determinado Remédio em cada Posto de Saúde;

RNF1 - A aplicação deve estar acessível ao usuário via internet; RNF2 - A aplicação só deve permitir acesso de um agente previamente autorizado.

5.1.2 Arquitetura

A aplicação segue a arquitetura de sistemas web dividindo-se em 3 camadas. Uma delas é a Base de Dados onde as informações dos Postos de Saúde e dos Remédios serão armazenadas (RQF1), outra é a camada de *interface*, ou *frontend*, responsável pela comunicação com o usuário. Por fim, tem-se a camada responsável por prover as informações e as funcionalidades do sistema ao *frontend*, centralizando as regras de negócio e o acesso ao banco de dados, essa camada é chamada de *backend*.

Sendo esta uma aplicação *web*, seu *backend* e seu *frontend* se comunicam por meio de um *web service*, tecnologia que utiliza os protocolos de internet e permite que plataformas heterogêneas se comuniquem. Utilizando um *web service* pode-se implementar, por exemplo, um *backend* em linguagem Java e um *frontend* em linguagem *TypeScript*, como feito neste experimento.

Como o escopo do MoPE está centrado na construção e evolução das regras de negócio de um projeto de software, todos os artefatos construídos e abordados nesta aplicação prática dizem respeito as regras de negócio que ficam no *backend*. A comunicação com o Sistema Gerenciador de Banco de Dados (SGBD), a construção das telas utilizadas no *frontend* e o *web service*, foram feitos utilizando-se outras tecnologias (fora do escopo do MoPE).

Os modelos e a implementação, que compõem as regras de negócio, foram construídos e exportados como uma biblioteca para serem utilizados no projeto desse experimento (*Remédio Aqui*). O *backend* da aplicação é composto por essa biblioteca (exportada do MoPE) e por uma implementação complementar, responsável pela comunicação com o SGBD e pelo *web service*, que provê as funcionalidades necessárias ao *frontend* (RQF4 e RQF6). Alterações no modelo ou nas regras de negócio só podem ser realizadas a partir do MoPE, o que garante a consistência dos artefatos.

O *frontend* desta aplicação foi desenvolvido utilizando linguagens e tecnologias *web*, como *HTML5*, *CSS3*, *TypeScript*, entre outras. Nas subseções a seguir são apresentados os artefatos construídos a partir do MoPE para uso nesta aplicação prática.

5.1.3 *Modelagem*

A modelagem desta aplicação consistem na descrição dos cenários envolvendo as classes *Posto* e *Remedio* com seus respectivos atributos, e do cenário descrevendo a relação entre essas duas classes.

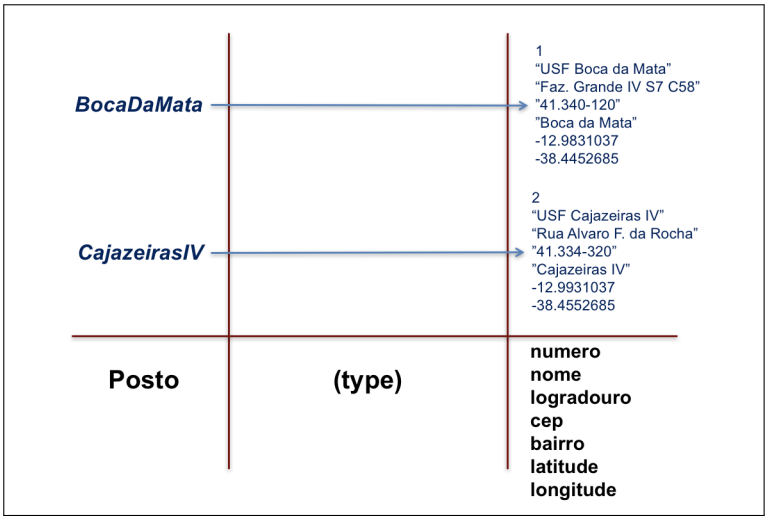


Figura 5.1: Modelagem dos Postos de Saúde no MOBI.
Fonte: Autor.

A Figura 5.1 ilustra o relacionamento da classe *Posto* (Posto de Saúde) com seus atributos, enquanto a Figura 5.2 o relacionamento da classe *Remedio* com os seus respectivos atributos.

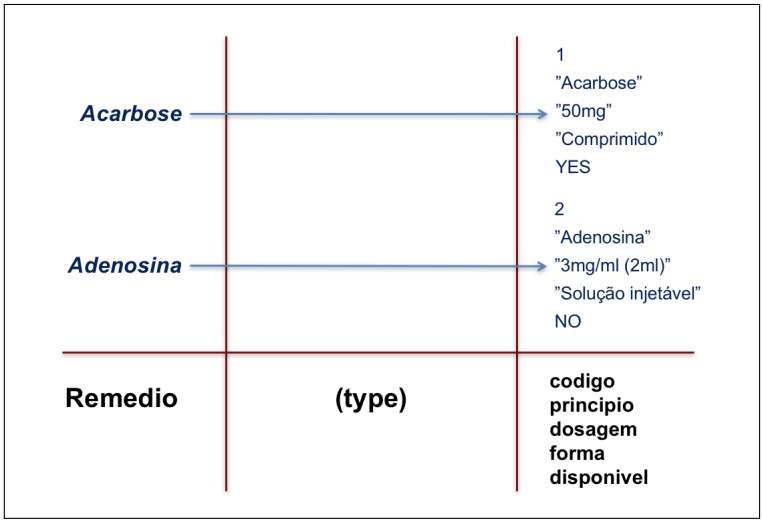


Figura 5.2: Modelagem dos Remédios no MOBI.
Fonte: Autor.

O cenário dos relacionamentos entre as 2 (duas) classes (*Posto* e *Remedio*) está descrito na Figura 5.3. Neste caso pode-se inferir que um posto de saúde deve possuir 0 (zero),

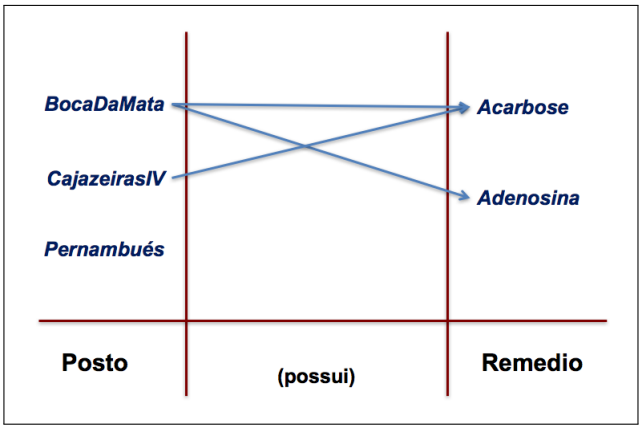


Figura 5.3: Relacionamento entre as classes *Posto* e *Remedios* descrito no MOBI.
Fonte: Autor.

1 (um) ou vários remédios, enquanto um remédio sempre deve está relacionamento a um posto.

5.1.4 Regras de Negócio

A implementação das regras de negócio da classe *Posto* é feita com base na instância *CajazeirasIV* e consiste na construção dos métodos para adicionar e remover um remédio no controle do posto de saúde.

```
1  CajazeirasIV
2
3          adicionar Adenosina
4                  (CajazeirasIV_possui add Adenosina)
5                  return YES
6
7          adicionar Acarbose
8                  return NO
9
10         remover Adenosina
11                 return NO
12
13         remover Acarbose
14                 (CajazeirasIV_possui remove Acarbose)
15                 return YES
16
```

Figura 5.4: Métodos *adicionar* e *remover* da classe *Posto* implementados em Mar.
Fonte: Autor.

O cenário ilustrado na Figura 5.4, se refere a instância (da classe *Posto*) Cajazeiras. E a primeira referência do método *adicionar* utiliza como exemplo a instância (da classe *Remedio*) *Adenosina*, onde a mesma é inserida no relacionamento *possui* e uma confirmação positiva é retornada. A segunda referencia do método *adicionar* utiliza como exemplo a

instância Acarbose (também da classe *Remedio*), porém apenas com um retorno negativo como implementação.

Como a instância *Adenosina* não está relacionada com *CajazeirasIV*, e a instância *Acarbose* está (conforme descrito na Figura 5.3), o conversor do MoPE irá inferir que apenas as instâncias da classe *Remedio* que já não estejam contidas na relação *possui* podem ser adicionadas. Desta forma, o método *adicionar* só adiciona um remédio ao relacionamento da classe *Posto* se este já não estiver relacionado. Assim como, um remédio só será removido se já estiver relacionado.

5.1.5 *Evolução do Modelo e da Implementação*

Para evidenciar a eficiência do modelo na manutenção da sincronia entre os Ambientes de Modelagem e Programação diante de um cenário de evolução das regras de negócio, foram realizadas 2 (duas) alterações no projeto, uma na modelagem e outra na implementação.

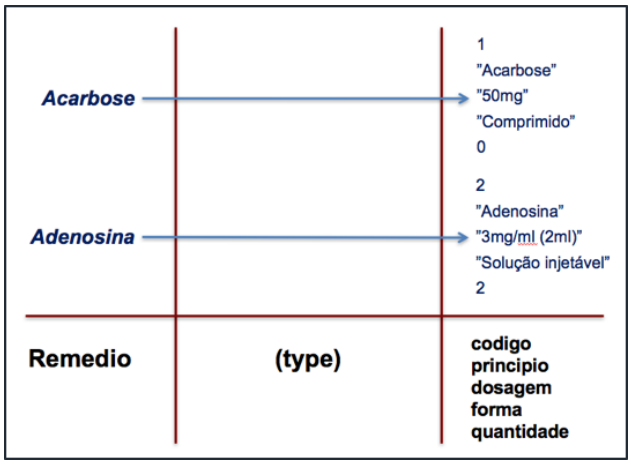


Figura 5.5: Alteração na modelagem da classe *Remedio*.
Fonte: Autor.

Com o objetivo de alterar o sistema para que o mesmo trabalhe com a informação da quantidade de um determinado remédio em estoque, e não apenas com a sua disponibilidade, o atributo *disponivel* foi removido da modelagem da classe *Remedio*, enquanto o atributo quantidade foi acrescentado. Na Figura 5.5, pode-se observar que o novo atributo é do tipo *integer* (número inteiro) enquanto o antigo era um *boolean* (verdadeiro ou falso).

A lógica de programação da classe *Remedio* também foi alterada. Dois métodos novos foram implementados, conforme Figura 5.6, ambos não retornam valores e nem receber parâmetros, um deles tem o nome *maisUm* e o outro *menosUm*. A ideia é que ambos possam ser utilizados para aumentar ou reduzir, um a um, a quantidade de remédios em estoque em um determinado posto de saúde.

```
1  Adenosina
2
3          maisUm
4                  Adenosina_quantidade = Adenosina_quantidade + 1
5
6          menosUm
7                  if Adenosina_quantidade > 0
8                          Adenosina_quantidade = Adenosina_quantidade - 1
9
```

Figura 5.6: Inclusão de Implementação para classe *Remedio*.
Fonte: Autor.

O método *menosUm* foi implementado com uma condição para verificar se o valor do atributo estoque é maior que 0 (zero), só neste caso é que o valor do estoque será reduzido em uma unidade (ver Figure 5.6). Desta forma não será possível atribuir valor abaixo de zero ao estoque de qualquer remédio.

```
52
53      public void maisUm() {
54          this.setQuantidade((this.getQuantidade()+ 1));
55      }
56
57      public void menosUm() {
58          if (this.getQuantidade()> 0) {
59              this.setQuantidade((this.getQuantidade()- 1));
60          }
61      }
62
```

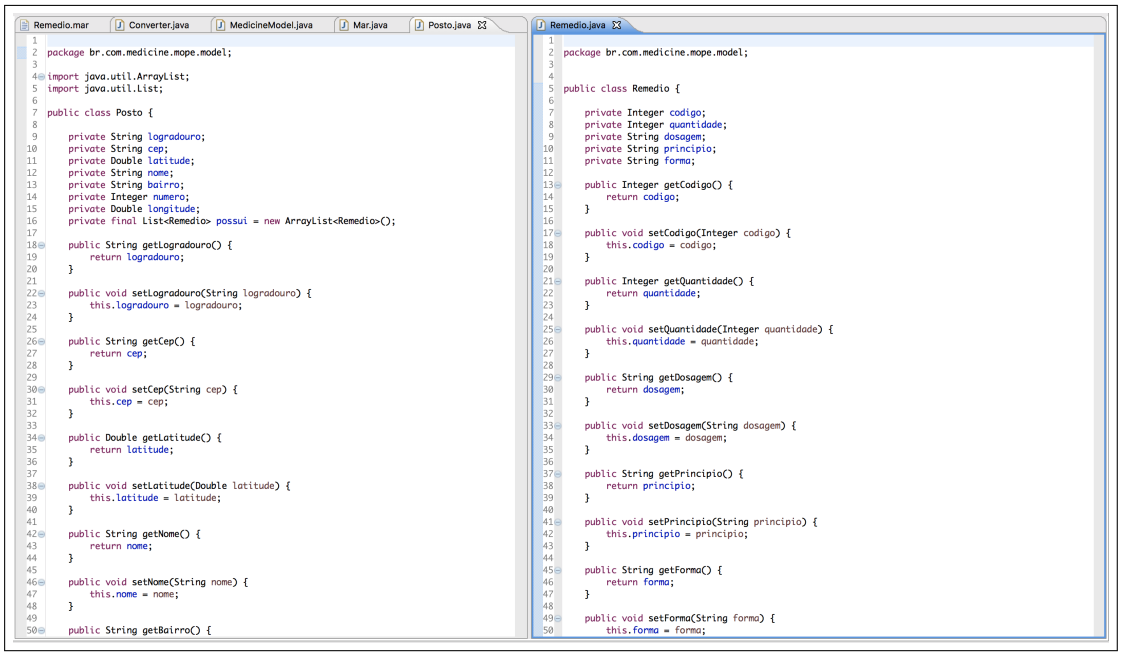
Figura 5.7: Código Java gerado pelo MoPE da classe *Remedio*.
Fonte: Autor.

A Figura 5.7 mostra o código gerado como linguagem intermediária pelo compilador. O método em questão está escrito com um *if* que compara se o valor do atributo quantidade é maior que 0 (zero), caso sim o atributo estoque é atualizado em menos uma unidade, caso contrário nada é executado e o valor se mantém.

5.1.6 Resultado

Ambos os modelos de classes (*Posto* e *Remedio*), assim como a implementação que complementa a classe *Posto*, foram convertidos em 2 (duas) classes escritas na linguagem Java, conforme ilustrado pela Figura 5.8. A linguagem Java é a linguagem intermediária que o MoPE gera após seu compilador converter a lógica escrita na metalinguagem *Mar*. Sendo assim, cada classe é escrita (em Java) com seus respectivos atributos, definidos no ambiente de modelagem (MOBI), assim como os métodos implementados no ambiente de programação (ver Figura 5.9).

As regras de inferência definidas pelo MoPE são aplicadas nessa etapa de conversão (do



Remedio.mar | Converter.java | MedicineModel.java | Mar.java | Posto.java

```
package br.com.medicine.mope.model;

import java.util.ArrayList;
import java.util.List;

public class Posto {

    private String logradouro;
    private String cep;
    private Double latitude;
    private String nome;
    private String bairro;
    private Integer numero;
    private Double longitude;
    private final List<Remedio> possui = new ArrayList<Remedio>();

    public String getLogradouro() {
        return logradouro;
    }

    public void setLogradouro(String logradouro) {
        this.logradouro = logradouro;
    }

    public String getCep() {
        return cep;
    }

    public void setCep(String cep) {
        this.cep = cep;
    }

    public Double getLatitude() {
        return latitude;
    }

    public void setLatitude(Double latitude) {
        this.latitude = latitude;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getBairro() {
```

Remedio.java

```
package br.com.medicine.mope.model;


public class Remedio {

    private Integer codigo;
    private Integer quantidade;
    private String dosagem;
    private String principio;
    private String forma;

    public Integer getCodigo() {
        return codigo;
    }

    public void setCodigo(Integer codigo) {
        this.codigo = codigo;
    }

    public Integer getQuantidade() {
        return quantidade;
    }

    public void setQuantidade(Integer quantidade) {
        this.quantidade = quantidade;
    }

    public String getDosagem() {
        return dosagem;
    }

    public void setDosagem(String dosagem) {
        this.dosagem = dosagem;
    }

    public String getPrincipio() {
        return principio;
    }

    public void setPrincipio(String principio) {
        this.principio = principio;
    }

    public String getForma() {
        return forma;
    }

    public void setForma(String forma) {
        this.forma = forma;
```

Figura 5.8: Classes Posto e Remedio convertidas para linguagem Java.
Fonte: Autor.

PIM para o PSM), conforme descrito no processo da Seção 4.2.1. Desta forma, a linguagem resultante poderá ser convertida em outra com mais facilidade, caso contrário, seria necessário a implementação de um compilador cada vez que um novo módulo de conversão de uma determinada linguagem de programação fosse desenvolvido.

No caso da implementação dos métodos *adicionar* e *remover* (Figuras 5.4 e 5.9), o compilador verificou as relações entre as instâncias (*Adenosina/Acarbose* e *CajazeirasIV*) para escrever a lógica corretamente de ambos os métodos na linguagem intermediária (*Java*).

No momento da criação de outros conversores, para outras linguagens de programação, não será necessária a implementação de um novo compilador em cada um deles, pois a linguagem intermediária já representa a lógica de uma linguagem de programação convencional (não orientada a cenários ou instâncias).

A segunda aplicação consiste em um aplicativo para dispositivos móveis utilizado para simular o cálculo de imposto de renda de uma pessoa.

5.2 Aplicação para Cálculo do Imposto de Renda

O objetivo desta aplicação é realizar cálculos simplificados dos valores a serem pagos referente ao Imposto de Rende (no Brasil). Esse aplicativo está disponível nas lojas

```
78      public Boolean adicionar(Remedio remedio) {
79          if (this.getPossui().contains(remedio)) {
80              return false;
81          } else {
82              this.getPossui().add(remedio);
83              return true;
84          }
85      }
86
87      public Boolean remover(Remedio remedio) {
88          if (this.getPossui().contains(remedio)) {
89              this.getPossui().remove(remedio);
90              return true;
91          } else {
92              return false;
93          }
94      }
```

Figura 5.9: Métodos da Classe Posto convertidos para linguagem Java.
Fonte: Autor.

virtuais *Google Play* (https://play.google.com/store/apps/details?id=add.calc_ir) e *App Store* (https://itunes.apple.com/us/app/calc-ir/id534976561?ls=1&mt=8).

5.2.1 Requisitos

O propósito desta aplicação é permitir ao pagador de impostos brasileiro realizar e gravar uma simulação simplificada do cálculo de Imposto de Renda. Para isso, os seguintes Requisitos Funcionais (RQF) e Não Funcionais (RNF) foram definidos:

RQF1 - Realizar o cálculo de imposto de renda de um cidadão; RQF2 - Armazenar os cálculos realizados localmente (no próprio dispositivo móvel); RQF3 - Prover a listagem dos cálculos realizados por cidadão; RQF4 - Permitir a edição (recálculo) do cálculo de um cidadão;

RNF1 - A aplicação não deve depender de acesso a internet; RNF2 - A aplicação deve funcionar em dispositivos móveis; RNF3 - Os cálculos devem ser armazenados no próprio dispositivo do usuário.

5.2.2 Arquitetura

Como a ideia central desta aplicação é ser prática (RNF1) e estar sempre a mão (RNF2) para realizar um cálculo pessoal de imposto mantendo os dados em fácil acesso (RNF3), sua arquitetura foi idealizada para que a mesma funcione integralmente a partir de um dispositivo móvel, sem a necessidade de um serviço *web* (ou um *web service*).

Esta aplicação segue a arquitetura MVC (*Model View Controller*) [Gamma et al. 2011], sendo o modelo provido pelo MoPE e as camadas de visão e controle implementadas na plataforma específica. A tecnologia (plataforma específica) utilizada para construção desse aplicativo *mobile* (aplicativo voltado para dispositivo móvel) foi o *Ionic Framework* (https://ionicframework.com/).

O *Ionic* é uma tecnologia que permite a construção de um aplicativo *mobile* para mais de uma plataforma (e.g.: *iOS, Android, Windows Phone*, etc) a partir de um mesmo código fonte. Desta forma, todos os detalhes de armazenamento de dados, construção de telas e integração desses componentes da aplicação foram desenvolvidos neste *framework*. Sendo necessária a conversão do PSM (escrito em linguagem Java) para a linguagem *TypeScript*, que é a linguagem utilizada pelo *Ionic*.

No MoPE foram construídos o modelo de classes e a implementação da lógica e das regras para o cálculo do imposto de renda brasileiro (RQF1). Como o escopo definido do MoPE prevê, temos a construção do modelo de negócios sendo realizado no MoPE, e esse modelo de negócios é utilizado em uma plataforma específica, onde são desenvolvidas as *interfaces* de comunicação com usuário ou a lógica de persistência dos dados (RQF2, RQF3 e RQF4), que neste caso foram implementadas no *Ionic Framework*, completando assim a construção do software.

5.2.3 Modelagem

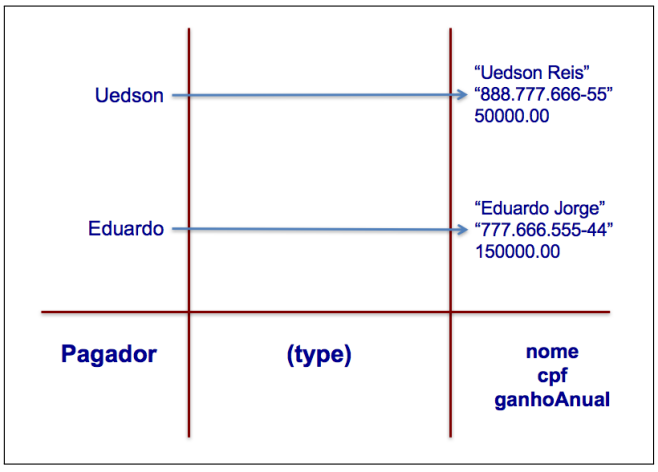


Figura 5.10: Modelagem da classe *Pagador* feita no MOBI.
Fonte: Autor.

A modelagem desta aplicação definiu as classes *Pagador* (pessoa que paga o imposto), *Deducao* (gastos que podem ser deduzidos do imposto) e *Tabela* (tabela de pagamentos do imposto para um determinado ano).

O cenário descrito na Figura 5.10 utiliza 2 (dois) exemplos para define a classe *Pagador* com 3 (três) atributos (*nome*, *cpf* e *ganhoAnual*).

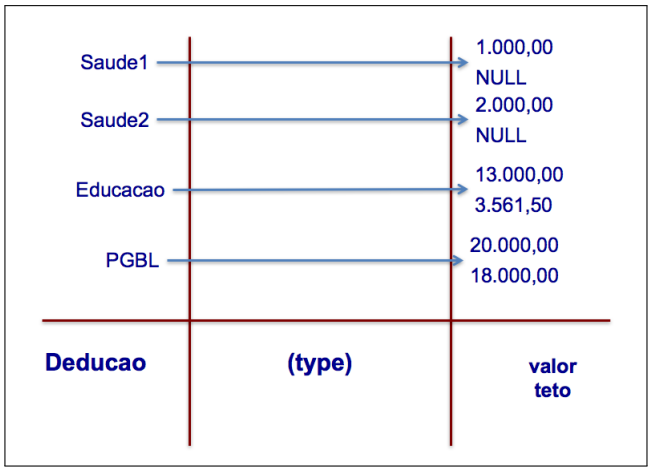


Figura 5.11: Modelagem da classe *Deducao* feita no MOBI.
Fonte: Autor.

No cenário da Figura 5.11 4 (quatro) exemplos são descritos para que a classe *Deducao* seja criada com os atributos *valor* (valor da despesa a ser deduzida) e *teto* (limite máximo de valor a ser deduzido). Os exemplos *Saude1* e *Saude2*, seriam gastos com saúde e por isso não teriam um teto (limite máximo) de valor a ser deduzido. *Educacao* seria um exemplo de gasto com educação, seja colégio, faculdade, pós-graduação, etc. E *PGBL* se refere a um investimento feito em previdência privada.

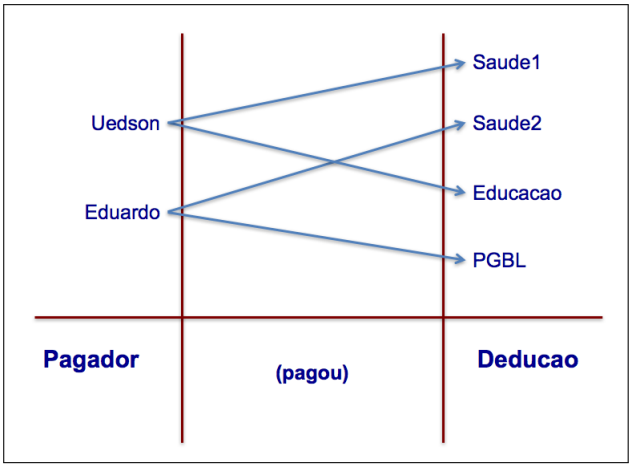


Figura 5.12: Modelagem do relacionamento entre as classes *Pagador* e *Deducao*.
Fonte: Autor.

A Figura 5.12 ilustra o cenário de relacionamento entre as classes *Pagador* e *Deducao*, onde os exemplos de cada classe são vinculados, indicando que ambos os indivíduos realizaram gastos dedutíveis em 2 (duas) oportunidades, *Uedson* com *Educacao* e *Saude1* e *Eduardo* com *PGBL* e *Saude2*.

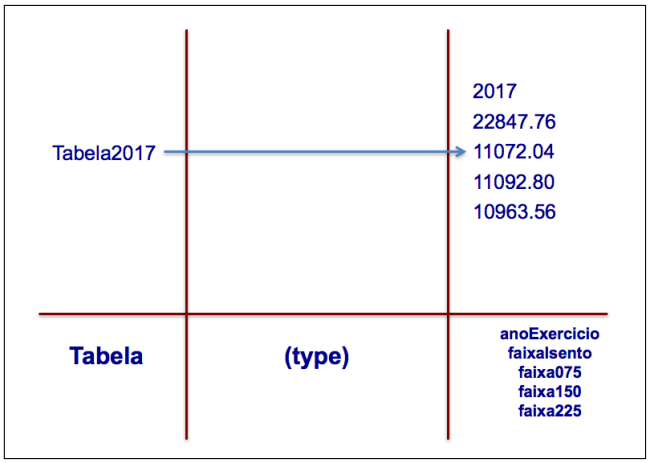


Figura 5.13: Modelagem da classe *Tabela* feita no MOBI.
Fonte: Autor.

Na Figura 5.13 o cenário ilustrado é o da classe *Tabela*, que representa a tabela do imposto de renda de um determinado ano. O exemplo utilizado neste cenário é o da tabela de 2017 que define 4 (quatro) faixas de valores para incidência de diferentes alíquotas no cálculo do imposto de renda, sendo elas as faixas de isento (*faixaIsento*), 7,5% (*faixa075*), 15% (*faixa150*) e 22,5% (*faixa225*). Mais detalhes sobre o cálculo de imposto de renda estão descritos na Seção 5.2.4.

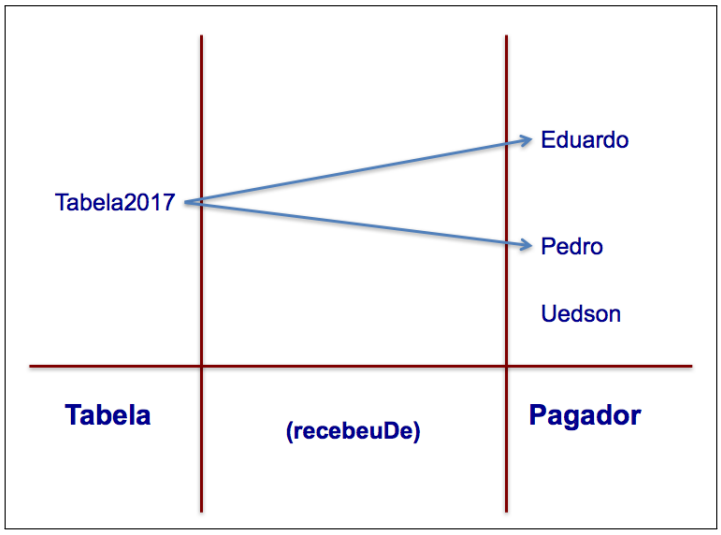


Figura 5.14: Modelagem do relacionamento entre as classes *Tabela* e *Pagador*.
Fonte: Autor.

A Figura 5.14 ilustra o cenário de relacionamento entre as classes *Tabela* e *Pagador*, onde os exemplos de cada classe são vinculados, indicando que os indivíduos *Eduardo* e *Pedro* realizaram o cálculo do imposto de renda utilizando a Tabela de 2017 (*Tabela2017*). Como o indivíduo *Uedson* não foi vinculado a nenhum exemplo da classe *Tabela*, infere-se que o mesmo não realizou o cálculo.

5.2.4 *Regras de Negócio*

Vale ressaltar que todos os valores ou regras aqui descritos com relação aos tributos (impostos) brasileiros estavam em vigência durante o período de desenvolvimento do aplicativo *Calc IR*.

Além das despesas dedutíveis indicadas como exemplo na modelagem dessa aplicação (Figura 5.11), sempre se deve considerar o valor pago ao INSS (Instituto Nacional do Seguro Social) como dedutível. O cálculo para pagamento do INSS leva em consideração três alíquotas de pagamento, quem ganhava até R$ 1.659,38 pagava 8% do seu salário ao INSS, entre R$ 1.659,38 e R$ 2.765,66 9%, entre R$ 2.765,66 e R$ 5.531,31 11%, e para salários acima de R$ 5.531,31 o valor a ser pago era fixado em 11% de R$ 5.531,31.

```
1  taxaPadraoINSS = 0.11
2  inssLimite1 = 1659.38 * 13
3  inssLimite2 = 2765.66 * 13
4  inssLimite3 = 5531.31 * 13
5
6  Uedson
7          calcularINSS
8                  taxa = taxaPadraoINSS
9
10                 if Uedson.ganhoAnual <= inssLimite1
11                         taxa = 0.08
12                 else if Uedson.ganhoAnual <= inssLimite2
13                         taxa = 0.09
14
15                 if Uedson.ganhoAnual > inssLimite3
16                         return inssLimite3 * taxa
17                 else
18                         return Uedson.ganhoAnual * taxa
19
```

Figura 5.15: Implementação da classe *Pagador* escrita em Mar.
Fonte: Autor.

Toda a lógica dessas regras de negócio foram implementadas, em metalinguagem *Mar*, a partir do indivíduo *Uedson* (Figura 5.15), como o mesmo pertence a classe *Pagador*, essa implementação é inferida como método desta classe. Outro ponto de destaque nesta implementação são as variáveis colocadas na parte superior (da linha 1 à 4), antes do texto que indica o inicio do escopo de implementação do indivíduo *Uedson* (linha 6). Como estas variáveis estão fora do escopo do objeto devem ser consideradas como variáveis estáticas, representando os mesmo valores independente dos indivíduos ou exemplos que a utilizem.

A regra de cálculo para os gastos dedutíveis exemplificados, na Figura 5.11, é mais simples, não há faixas para diferentes alíquotas apenas um teto para dedução, quando esse teto é ultrapassado o valor a ser deduzido é o próprio teto. Conforme a lógica ilustrada na Figura 5.16, escrita em metalinguagem Mar a partir do exemplo *Educacao*.

O cálculo do imposto de renda é feito com base no salário anual do cidadão, porém

```
1  Educacao
2
3      dedutivo
4          if Educacao_teto is not NULL and Educacao_teto < Educacao_valor
5                  return Educacao_teto
6          else
7                  return Educacao_valor
8
```

Figura 5.16: Implementação da classe *Deducao* escrita em Mar.
Fonte: Autor.

antes de realizar o calculo os gastos dedutíveis (exemplos da classe *Deducao*) devem ser abatidos desse salário anual, o valor resultante é chamado de base de cálculo, e é ele que será utilizado para calcular o valor a ser pago referente ao imposto de renda. Essa parte da lógica do cálculo está ilustrada na Figura 5.18 da linha 29 até a linha 32.

```
1  P075 = 0.075
2  P150 = 0.15
3  P225 = 0.225
4  P275 = 0.275
5
6  Tabela2017
7
8              adicionar Uedson
9                      (Tabela2017_recebeuDe add Uedson)
10                     return YES
11
12             adicionar Eduardo
13                     return NO
14
15             remover Uedson
16                     return NO
17
18             remover Eduardo
19                     (Tabela2017_recebeuDe remove Eduardo)
20                     return YES
   ...
```

Figura 5.17: Implementação da classe *Tabela* escrita em Mar (parte 1).
Fonte: Autor.

Uma vez definida a base de cálculo, ou valor base, esse valor é dividido em faixas, onde em cada uma incide um tipo diferente de alíquotas (0.0%, 7,5%, 15%, 22,5% e 27,5%). As alíquotas podem ser visualizadas na Figura 5.17 nas linhas de 1 a 4, com excepção da 0.0% correspondente a faixa de isento.

Um cidadão que obteve um valor base de até R$ 22.847,76 foi considerado isento (*faixaIsento*), os que obtiveram um valor acima de R$ 22.847,76 porém abaixo de 33.919,81 (*faixa075*) pagaram 7,5% sobre o valor compreendido entre essas faixas. Por exemplo, se um cidadão obteve um valor base de R$ 23.847,76 irá pagar R$ 75,00 referente a 7,5% de R$ 1.000,00 (que é o valor entre as faixas de isento e de 7,5%). Seguindo esta lógica, os primeiros R$ 22.847,76 do valor base sempre irão para a faixa de isento, assim como os primeiros R$ 11.072,04 (após o valor isento) serão enquadrados na faixa da alíquota de 7,5% (*faixa075*), totalizando os primeiros R$ 33.919,80 do valor base de um cidadão.

```
...
21          getTetoEducacao
22                  return 3561.5
23
24          getTetoPGBL Uedson
25                  return Uedson_ganhoAnual * 0.12
26
27          calcularIRPF Uedson
28                  totalAPagar = 0.0
29                  valorBase = Uedson.ganhoAnual - (Uedson calcularINSS)
30
31                  valorBase = valorBase – (Uedson_Saude1 dedutivo)
32                  valorBase = valorBase – (Uedson_Educacao dedutivo)
33
                    ... // Cálculo do IRPF na próxima imagem
66
67                  Uedson.totalAPagar = totalAPagar
68                  (Tabela2017.recebeuDe add Uedson)
69
```

Figura 5.18: Implementação da classe *Tabela* escrita em Mar (parte 2).
Fonte: Autor.

```
...
34          if valorBase <= Tabela2017.faixaIsento
35                  Uedson.valorFaixaIsento = valorBase
36          else
37                  Uedson.valorFaixaIsento = Tabela2017.faixaIsento
38                  valorBase = valorBase - Tabela2017.faixaIsento
39
40                  if valorBase <= Tabela2017.faixa075
41                          Uedson.valorFaixa075 = valorBase
42                          totalAPagar = valorBase * P075
43                  else
44                          Uedson.valorFaixa075 = Tabela2017.faixa075
45                          totalAPagar = Tabela2017.faixa075 * P075
46                          valorBase = valorBase - Tabela2017.faixa075
47
48                          if valorBase <= Tabela2017.faixa150
49                                  Uedson.valorFaixa150 = valorBase
50                                  totalAPagar = totalAPagar + valorBase * P150
51                          else
52                                  Uedson.valorFaixa150 = Tabela2017.faixa150
53                                  totalAPagar = totalAPagar + Tabela2017.faixa150 * P150
54                                  valorBase = valorBase - Tabela2017.faixa150
55
56                                  if valorBase <= Tabela2017.faixa225
57                                          Uedson.valorFaixa225 = valorBase
58                                          totalAPagar = totalAPagar + valorBase * P225
59                                  else
60
61                                          Uedson.valorFaixa225 = Tabela2017.faixa225
62                                          totalAPagar = totalAPagar + Tabela2017.faixa225 * P225
63                                          valorBase = valorBase - Tabela2017.faixa225
64                                          Uedson.valorFaixa275 = valorBase
65                                          totalAPagar = totalAPagar + valorBase * P275
...
```

Figura 5.19: Implementação da classe *Tabela* escrita em Mar (parte 3).
Fonte: Autor.

De forma a aplicar cada alíquota apenas a sua faixa de valor correspondente e não o valor como um todo, essa mesma lógica de distribuição dos valores é aplicada em cada faixa (*faixaIsento*, *faixa075*, *faixa150* e *faixa225*, caso o valor base seja maior do que o da faixa de 22,5% a alíquota aplicada será a de 27,5% para todo esse valor excedente), como ilustrado na Figura 5.19.

Para melhor visualização de sua implementação, a Figura 5.18 tem parte de seu código abstraído (da linha 34 à 66) e colocado na Figura 5.19, a qual mostra a integra da lógica de cálculo do imposto de renda.

5.2.5 *Resultado*

No caso desta aplicação prática a conversão ocorreu em duas etapas, primeiro o modelo de negócios construído no MoPE (modelagem e implementação) foi convertido em sua linguagem PSM (Java), depois esse PSM foi convertido em linguagem *TypeScript* para ser inserido no projeto do *Ionic Framework*.

pagador.ts

```
1  import { Deducao } from './deducao';
2
3  export class Pagador {
4
5      public static readonly taxaPadraoINSS = 0.11;
6      public static readonly inssLimite1 = 1659.38 * 13;
7      public static readonly inssLimite2 = 2765.66 * 13;
8      public static readonly inssLimite3 = 5531.31 * 13;
9
10     public ganhoAnual: number;
11     public cpf: String;
12     public nome: String;
13     public readonly pagou: Array<Deducao> = new Array<Deducao>();
14
15     public totalAPagar: number;
16     public valorFaixaIsento: number;
17     public valorFaixa075: number;
18     public valorFaixa150: number;
19     public valorFaixa225: number;
20     public valorFaixa275: number;
21
22     public calcularINSS(): number {
23         var taxa: number = Pagador.taxaPadraoINSS;
24
25         if (this.ganhoAnual <= Pagador.inssLimite1) {
26             taxa = 0.08;
27         } else if (this.ganhoAnual <= Pagador.inssLimite2) {
28             taxa = 0.09;
29         }
30
31         if (this.ganhoAnual > Pagador.inssLimite3) {
32             return (Pagador.inssLimite3 * taxa);
33         } else {
34             return (this.ganhoAnual * taxa);
35         }
36     }
37 }
```

Figura 5.20: Código gerado da classe *Pagador*.
Fonte: Autor.

A Figura 5.20 mostra o código gerado da classe *Pagador*, destaque para as variáveis estáticas no topo da imagem (linhas de 5 à 8), como foram definidas acima do escopo de implementação do indivíduo *Uedson* foram inferidas como estáticas pelo MoPE.

Outro destaque importante é a criação, por parte do MoPE, das variáveis *totalAPagar*, *valorFaixaIsento*, *valorFaixa075*, *valorFaixa150*, *valorFaixa225* e *valorFaixa275*. Estas variáveis não foram definidas na modelagem da classe *Pagador*, porém foram escritas na implementação da classe *Tabela* (ver Figuras 5.18 e 5.19, linhas 35, 37, 41, 44, 49, 52, 57, 61, 64 e 67). Como estas são variáveis que apenas tem seu valor atribuído durante alguma ação (não têm valor inicial como as que foram exemplificadas na modelagem), tiveram sua criação inferida e realizada pelo compilador do MoPE.

Na Figura 5.21 temos a conversão da implementação da classe *Deducao* que trás 2 (duas) variáveis (*valor* e *teto*) e o método *dedutivo()* responsável pela verificação do valor dedutível, sendo *valor* se este for menor que *teto*, caso contrário será *teto*.

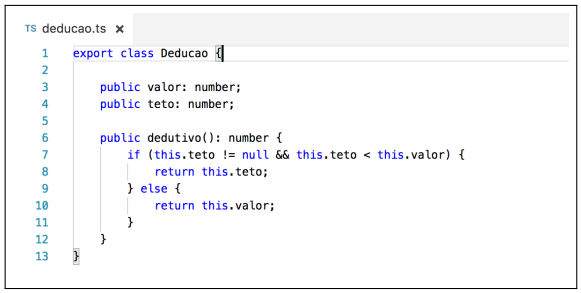


Figura 5.21: Código gerado da classe *Deducao*.
Fonte: Autor.

As Figuras 5.22, 5.23 e 5.24 ilustram a lógica da classe *Tabela*, necessária para o cálculo do imposto de renda, já convertida em *TypeScript*, uma vez implementada no MoPE a partir da metalinguagem Mar. Observa-se que todas as inferências e conversões ocorram como esperado, tanto na conversão para o PSM quanto na conversão para a plataforma específica (*TypeScript*).

tabela.ts

```typescript
import { Pagador } from './pagador';

export class Tabela {

    public static readonly INSS: number = 0.11;
    public static readonly P075: number = 0.075;
    public static readonly P150: number = 0.150;
    public static readonly P225: number = 0.225;
    public static readonly P275: number = 0.275;

    public anoExercicio: number = 2017;
    public faixaIsento: number = 22847.76;
    public faixa075: number = 11072.04; // 22.847,76 - 33.919,80
    public faixa150: number = 11092.80; // 33.919,80 - 45.012,60
    public faixa225: number = 10963.56; // 45.012,60 - 55.976,16
    public tetoEducacao: number = 3561.50;

    public readonly recebeuDe: Array<Pagador> = new Array<Pagador>();

    public getTetoEducacao(): number {
        return 3561.50;
    }

    public getTetoPGBL(pagador: Pagador): number {
        return pagador.ganhoAnual * 0.12;
    }

    public calcularIRPF(pagador: Pagador): void {
        var totalAPagar: number = 0.0;
        var valorBase: number = pagador.ganhoAnual - pagador.calcularINSS();

        pagador.pagou.forEach(deducao => {
            valorBase = valorBase - deducao.dedutivo();
        });

        if (valorBase <= this.faixaIsento) {
            pagador.valorFaixaIsento = valorBase;
        } else {
            pagador.valorFaixaIsento = this.faixaIsento;
            valorBase = valorBase - this.faixaIsento;
```

Figura 5.22: Código gerado da classe *Tabela* (parte 1).
Fonte: Autor.

```typescript
            if (valorBase <= this.faixa075) {
                pagador.valorFaixa075 = valorBase;
                totalAPagar = valorBase * Tabela.P075;
            } else {
                pagador.valorFaixa075 = this.faixa075;
                totalAPagar = totalAPagar + this.faixa075 * Tabela.P075;
                valorBase = valorBase - this.faixa075;

                if (valorBase <= this.faixa150) {
                    pagador.valorFaixa150 = valorBase;
                    totalAPagar = totalAPagar + valorBase * Tabela.P150;
                } else {
                    pagador.valorFaixa150 = this.faixa150;
                    totalAPagar = totalAPagar + this.faixa150 * Tabela.P150;
                    valorBase = valorBase - this.faixa150;

                    if (valorBase <= this.faixa225) {
                        pagador.valorFaixa225 = valorBase;
                        totalAPagar = totalAPagar + valorBase * Tabela.P225;
                    } else {
                        pagador.valorFaixa225 = this.faixa225;
                        totalAPagar = totalAPagar + this.faixa225 * Tabela.P225;
                        valorBase = valorBase - this.faixa225;

                        pagador.valorFaixa275 = valorBase;
                        totalAPagar = totalAPagar + valorBase * Tabela.P275;
                    }
                }
            }
        }

        pagador.totalAPagar = totalAPagar;
        this.adicionar(pagador);
    }

```

Figura 5.23: Código gerado da classe *Tabela* (parte 2).
Fonte: Autor.

```
76
77      public adicionar(pagador: Pagador): boolean {
78          var index: number = this.recebeuDe.indexOf(pagador);
79          if (index > -1) {
80              return false;
81          } else {
82              this.recebeuDe.push(pagador);
83              return true;
84          }
85      }
86
87      public remover(pagador: Pagador): boolean {
88          var index: number = this.recebeuDe.indexOf(pagador);
89          if (index > -1) {
90              this.recebeuDe.splice(index, 1);
91              return true;
92          } else {
93              return false;
94          }
95      }
96  }
```

Figura 5.24: Código gerado da classe *Tabela* (parte 3).
Fonte: Autor.

Capítulo Seis

Considerações finais

Este trabalho teve como objetivo a construção de um novo modelo de programação, visando mitigar os problemas de dessincronia entre a modelagem e a implementação de um projeto de software. A proposta foi baseada na extensão do MOBI, que pautaria o ambiente de modelagem e serviria de base para o desenvolvimento do ambiente de implementação. A pesquisa utilizou a metodologia incremental de desenvolvimento de software, junto com o método científico Pesquisa-ação, para especificar, modelar e construir o modelo proposto, denominado MoPE. O Modelo de Programação orientado a Exemplos foi assim denominado pois segue os princípios do MOBI, que é um método de modelagem baseado em Instâncias.

6.1 Contribuições

Foi desenvolvida uma metalinguagem de programação baseada em exemplos que permite a um analista de sistemas implementar regras de negócio para um projeto de software a partir do modelo de classes construído no MOBI. A metalinguagem *Mar* é integrada ao MOBI tendo acesso aos artefatos modelados nele, além disso, não é possível criar classes no Ambiente de Programação, isso só é possível a partir do Ambiente de Modelagem. Desta forma, a escrita de regras de negócio será sempre complementar ao modelo de classes, impedindo o retrabalho na recriação das classes no Ambiente de Programação e mantendo a sincronia desse com o Ambiente de Modelagem.

Também foi implementado um compilador capaz de processar os artefatos construídos no Ambiente de Modelagem (modelagem feita no MOBI) e no Ambiente de Programação (código escrito em *Mar*) e construir um Modelo de Negócios unificado. Esse componente é responsável por realizar a inferência das regras lógicas previstas para a metalinguagem *Mar* (Seção 4.5.1), além de integrar esse resultado ao modelo de classes já inferido pelo MOBI a partir dos exemplos de referência descritos em seu cenário.

Para integrar esses componentes e estrutura o modelo proposto, foi definido um processo de trabalho para todo o MoPE. Esse processo se inicia com um analista de sistema descrevendo os exemplos de referência e as conexões entre eles no cenário do MOBI. Depois a implementação das regras de negócio do sistema são escritas em *Mar* no Ambiente de Programação. O resultado final desse processo é a conversão do Modelo de Negócio (modelagem das classes mais sua implementação), gerado pelo compilador, em uma lin-

guagem intermediária, ou PSM, que tem como objetivo simplificar a sua conversão em uma linguagem específica de uma determinada plataforma.

Por fim, foram desenvolvidos 2 (dois) sistemas a fim de avaliar o modelo proposto, ambos os projetos utilizaram um Modelo de Negócios construído no MoPE e exportado para sua respectiva plataforma, o primeiro deles um sistema *web* em plataforma *Java* e o segundo um aplicativo para dispositivos móveis em plataforma *Ionic*. Ambos os sistemas estão funcionando e disponíveis na *web* (http://mymedicine.herokuapp.com e https://play.google.com/store/apps/details?id=add.calc_ir respectivamente).

6.2 Conclusão

Conclui-se que o modelo proposto, por meio da integração dos artefatos de modelagem e os de implementação, permite a construção de um projeto de software mantendo a sincronia entre os artefatos de modelagem e os de programação. Essa integração e sincronia entre modelo e regras de negócio, permite a exportação desse Modelo de Negócios para diversas plataformas específicas, além de sua reutilização em outros projetos de software. Esse Modelo de Negócios representa todo o modelo de classes e todas as regras de negócios de um dado tipo de sistema, sem levar em consideração a plataforma específica do sistema (se é *web*, *mobile* ou *desktop*). A redução do retrabalho sempre que uma alteração no modelo de classes for necessária e o melhor entendimento desse modelo também são vantagens do uso do modelo proposto, pois sempre que o modelo de classes é alterado por meio do MoPE essa alteração é passada ao Ambiente de Programação, sem a necessidade de uma replicação no mesmo.

A proposta de um modelo de programação baseado em cenários com exemplos de referência também mostrou-se viável para o desenvolvimento de um projeto de software. Pois simplificou a evolução do modelo de classes e das regras de negócio, por meio do uso de inferência lógica a partir dos exemplos, permitindo descrever toda uma estrutura com poucas linhas de código ou com a criação de um relacionamento entre duas instâncias.

6.3 Atividades Futuras de Pesquisa

Como trabalhos futuros, sugere-se a construção de uma IDE (Ambiente de Desenvolvimento Integrado) para uso do MoPE e baseada em suas premissas. Essa IDE deve fornecer o Ambiente de Modelagem a partir de uma interface gráfica simplificando processo de montagem dos cenários e associação das instâncias. O Ambiente de Programação deve possuir recursos como o autocomplete durante a escrita do código em metalingua-

gem *Mar* e a indicação de erros sintáticos ou semânticos no código já escrito. Uma IDE que suporte o processo e o modelo de desenvolvimento propostos nesta pesquisa também possibilita uma melhor comparação do MoPE com outras plataformas de desenvolvimento de software.

Também faz-se necessária uma evolução no processo de conversão do modelo a fim de implementar alguns Padrões de Projeto no código intermediário. Como são soluções já catalogadas e validadas para vários tipos de problemas padrão, os Padrões de Projeto podem ser utilizados para direcionar a forma como o código a ser gerado será escrito. Esse processo melhoraria a qualidade do código gerado e facilitaria sua manutenção, porém teria o desafio de identificar de forma adequada o problema, para com base nele escolher o padrão recomendado.

Por fim, sugere-se a evolução do conversor da metalinguagem para um compilador. Para isso, deve-se construir um analisador léxico e sintático, transformando a metalinguagem em uma linguagem de programação, além de aumentar sua estabilidade e confiança, com verificação de suas estruturas e um tratamento de exceção.

Apêndice A

Código Fonte

A.1 Código para Conversão do Modelo de Negócios

Converter.java

```
 1 package br.com.mope;
 2
 3 import java.io.File;
27
28 public class Converter {
29
30     private static final String OUTPUT_FOLDER = "output/";
31
32     private final Map<String, JDefinedClass> classes = new
   HashMap<String, JDefinedClass>();
33     private final JCodeModel codeModel = new JCodeModel();
34
35     private final Mobi mobi;
36     private final Mar mar;
37
38     public Converter(Mobi mobi) {
39         this.mobi = mobi;
40
41         String packageName = "br.com."+
   this.mobi.getContext().getUri().toLowerCase() +".mope";
42         this.mar = new Mar(packageName, this.codeModel, this.mobi);
43     }
44
45     /**
46      * Calls all necessary functions to generate Java code
47      * @throws IOException
48      */
49     public void exportCode() throws Exception {
50         this.generateAllDomainClasses();
51         this.generateAllMethods();
52         this.generateJavaFiles();
53     }
54
55     /**
56      * Generates a public [classType] get() method to the given class
57      * @param definedClass : The class to generate the method
58      * @param attribute : The class attribute to be returned
59      */
60     private void generateGetMethod(final JDefinedClass definedClass,
   final JFieldVar attribute) {
61         String name = "get" + attribute.name().substring(0,
   1).toUpperCase() + attribute.name().substring(1,
   attribute.name().length());
62         JMethod getMethod = definedClass.method(JMod.PUBLIC,
```

Page 1

Converter.java

```
attribute.type(), name);
63          getMethod.body()._return(attribute);
64      }
65
66      /**
67       * Generates a public void set([classType]) method to the given
class
68       * @param definedClass : The class to generate the method
69       * @param attribute : The class attribute to be modified
70       */
71      private void generateSetMethod(final JDefinedClass definedClass,
final JFieldVar attribute) {
72          String name = "set" + attribute.name().substring(0,
1).toUpperCase() + attribute.name().substring(1,
attribute.name().length());
73          JMethod setMethod = definedClass.method(JMod.PUBLIC,
this.codeModel.VOID, name);
74
75          setMethod.param(attribute.type(), attribute.name());
76
setMethod.body().assign(JExpr._this().ref(definedClass.fields().get(at
tribute.name())), attribute);
77      }
78
79      private void generateJavaFiles() throws IOException {
80          File file = new File(OUTPUT_FOLDER);
81          file.mkdirs();
82          this.codeModel.build(file);
83      }
84
85      /**
86       * Finds all Domain Classes from MOBI through the
mobi.getAllClasses()
87       * method and generates the necessary Java code to represent them.
Stores
88       * generated class in the HashMap classes for future references.
89       */
90      private void generateAllDomainClasses() {
91          HashMap<String, Class> mobiClasses =
this.mobi.getAllClasses();
92          for (String key : mobiClasses.keySet()) {
93              try {
94                  this.generateDomainClass(key);
95              } catch (JClassAlreadyExistsException e) {
```

Page 2

Converter.java

```
 96                 e.printStackTrace();
 97             } catch (Exception e) {
 98                 e.printStackTrace();
 99             }
100         }
101     }
102 
103     /**
104      * Creates the domain.[uri] class that will contains all the
    Mobi2Java generated code
105      * @param uri : The class name from MOBI
106      * @return The generated class on JCodeModel
107      * @throws Exception
108      */
109     private void generateDomainClass(final String uri) throws
    Exception {
110         this.mar.loadAllCodeFiles();
111         JDefinedClass domainClass =
    this.classes.get(this.mar.getPackagePath(uri));
112 
113         if (domainClass == null) {
114             domainClass =
    this.codeModel._class(this.mar.getPackagePath(uri));
115             this.classes.put(domainClass.name(), domainClass);
116         }
117 
118         try {
119             this.generateClassAttributes(domainClass);
120             this.generateClassProperties(domainClass);
121         } catch (Exception e) {
122             throw e;
123         }
124     }
125 
126     private void generateClassAttributes(final JDefinedClass
    definedClass) throws Exception {
127         System.out.println("generateClassAttributes for " +
    definedClass.name() + " : " + definedClass.fullName());
128 
129         Class mobiClass = this.mobi.getClass(definedClass.name());
130         System.out.println("Class Mobi: " + mobiClass.toString());
131 
132         Iterator<Instance> iterator =
    this.mobi.getClassInstances(mobiClass).iterator();
```

Page 3

Converter.java

```
133         Instance instance = iterator.next();
134         if (iterator.hasNext()) instance = null;
135 
136         for (Attribute attribute :
    this.mobi.getAllClassAttributes(mobiClass)) {
137 
138             System.out.println("field " + attribute.getUri() + " for "
    + definedClass.name() + " - " + definedClass.fullName());
139             JFieldVar dataType = definedClass.field(JMod.PRIVATE,
    this.mar.getClassType(attribute), attribute.getUri());
140 
141             if (instance != null) {
142                 String value =
    instance.getAttributeValueHashMap().get(attribute.getUri()).getValue()
    ;
143 
144 
    dataType.init(this.mar.getLitValue(attribute.getType(), value));
145             }
146 
147             this.generateGetMethod(definedClass, dataType);
148             this.generateSetMethod(definedClass, dataType);
149         }
150 
151         this.generateEqualsMethod(definedClass);
152     }
153 
154     private void generateClassProperties(final JDefinedClass
    definedClass) throws Exception {
155         System.out.println("generateClassProperties for " +
    definedClass.name() + " - " + definedClass.fullName());
156 
157         Class mobiClass = this.mobi.getClass(definedClass.name());
158         System.out.println("ClassMobi: " + mobiClass.toString());
159 
160         for (Relation relation :
    this.mobi.getAllClassCompositionRelations(mobiClass)) {
161 
162             if
    (relation.getClassB().getUri().equals(definedClass.name())) continue;
163 
164             System.out.println("field " + relation.getUri() + " for "
    + definedClass.name() + " - " + definedClass.fullName());
165 
```

Page 4

Converter.java

```
166             JFieldVar objProperty;
167 
168             if (relation.getCardinalityA().getType() ==
    Cardinality.ONE_N || relation.getCardinalityA().getType() ==
    Cardinality.ZERO_N) {
169 
170                 JType type =
    this.mar.getClassType("List-"+relation.getClassB().getUri());
171                 objProperty = definedClass.field(JMod.PRIVATE |
    JMod.FINAL, type, relation.getUri().split("_")[1]);
172 
    objProperty.init(JExpr._new(this.codeModel.ref(ArrayList.class).narrow
    (this.mar.getClassType(relation.getClassB().getUri()))));
173 
174             } else {
175                 JType type =
    this.mar.getClassType(relation.getClassB().getUri());
176                 objProperty = definedClass.field(JMod.PRIVATE, type,
    relation.getUri().split("_")[1]);
177                 this.generateSetMethod(definedClass, objProperty);
178             }
179 
180             this.generateGetMethod(definedClass, objProperty);
181         }
182     }
183 
184     private void generateEqualsMethod(final JDefinedClass
    definedClass) {
185 
186         if (this.mobi.getKeyAttribute(definedClass.name()) == null)
    return;
187 
188         String unique =
    this.mobi.getKeyAttribute(definedClass.name()).iterator().next().getUr
    i();
189         unique = "get" + unique.substring(0, 1).toUpperCase() +
    unique.substring(1, unique.length());
190 
191         JMethod equalsMethod = definedClass.method(JMod.PUBLIC,
    this.codeModel.BOOLEAN, "equals");
192         equalsMethod.annotate(java.lang.Override.class);
193         JVar objParam = equalsMethod.param(Object.class, "obj");
194         JBlock methodBody = equalsMethod.body();
195 
```

Page 5

Converter.java

```
196
    methodBody._if(objParam.eq(JExpr._null()))._then()._return(JExpr.FALSE
    );
197
198
    methodBody._if(JExpr._this().eq(objParam))._then()._return(JExpr.TRUE)
    ;
199
200
    methodBody._if(objParam._instanceof(definedClass).not())._then()._retu
    rn(JExpr.FALSE);
201
202         JVar other = methodBody.decl(definedClass, "other",
    JExpr.cast(definedClass, objParam));
203
    methodBody._if(other.invoke(unique).invoke("equals").arg(JExpr._this()
    .invoke(unique)).not())._then()._return(JExpr.FALSE);
204
205         methodBody._return(JExpr.TRUE);
206     }
207
208     private void generateAllMethods() throws
    JClassAlreadyExistsException {
209
210         for (String classUri : this.mar.getAllImplementedClasses()) {
211             JDefinedClass jDefClass =
    this.codeModel._getClass(this.mar.getPackagePath(classUri));
212             if (jDefClass == null) jDefClass =
    this.codeModel._class(this.mar.getPackagePath(classUri));
213
214             this.mar.createClassCode(classUri, jDefClass);
215         }
216
217     }
218 }
```

Page 6

A.2 Código para Conversão da Metalinguagem

Mar.java

```
 1 package br.com.mope;
 2
 3 import java.io.File;
 4 import java.util.ArrayList;
 5 import java.util.Date;
 6 import java.util.HashMap;
 7 import java.util.HashSet;
 8 import java.util.List;
 9 import java.util.Map;
10 import java.util.Set;
11
12 import com.sun.codemodel.JBlock;
13 import com.sun.codemodel.JCodeModel;
14 import com.sun.codemodel.JConditional;
15 import com.sun.codemodel.JDefinedClass;
16 import com.sun.codemodel.JExpr;
17 import com.sun.codemodel.JExpression;
18 import com.sun.codemodel.JFieldVar;
19 import com.sun.codemodel.JForEach;
20 import com.sun.codemodel.JInvocation;
21 import com.sun.codemodel.JMethod;
22 import com.sun.codemodel.JMod;
23 import com.sun.codemodel.JType;
24 import com.sun.codemodel.JVar;
25
26 import br.com.mope.littlelanguage.LittleLanguage;
27 import br.com.mope.littlelanguage.Token;
28 import mobi.core.Mobi;
29 import mobi.core.common.Relation;
30 import mobi.core.concept.Attribute;
31 import mobi.core.concept.AttributeTypeEnum;
32 import mobi.core.concept.Class;
33 import mobi.core.concept.Instance;
34 import mobi.core.relation.InstanceRelation;
35
36 public class Mar {
37
38     private static final String INPUT_FOLDER = "input/";
39     private static final String EXTENSION = ".mar";
40
41     private static final String DOT = "_";
42     private static final String GET = "get";
43     private static final String SET = "set";
44
```

Page 1

Mar.java

```
45      private static final Token BEGIN_TOKEN = new Token(Token.BEGIN);
46      private static final Token ELSE_TOKEN = new Token(Token.ELSE);
47      private static final Token END_TOKEN = new Token(Token.END);
48      private static final Token EQUAL_TOKEN = new Token(Token.EQUAL);
49      private static final Token FOR_TOKEN = new Token(Token.FOR);
50      private static final Token IF_NOT_TOKEN = new
    Token(Token.IF_NOT_EQUAL);
51      private static final Token IF_TOKEN = new Token(Token.IF);
52      private static final Token LESS_TOKEN = new Token(Token.SUB);
53      private static final Token MAJOR_TOKEN = new Token(Token.MAJOR);
54      private static final Token MINOR_TOKEN = new Token(Token.MINOR);
55      private static final Token MORE_TOKEN = new Token(Token.MORE);
56      private static final Token RETURN_TOKEN = new Token(Token.RETURN);
57      private static final Token TAB_TOKEN = new Token(Token.TAB);
58      private static final Token TIMES_TOKEN = new Token(Token.TIMES);
59
60      private final Map<String, List<Token>> filesTokens = new
    HashMap<String, List<Token>>();
61      private final Set<Instance> allInstances = new
    HashSet<Instance>();
62
63      private Set<Instance> thisInstances;
64      private Mobi mobi;
65
66      private final String packageName;
67      private final JCodeModel codeModel;
68
69      public Mar(String packageName, JCodeModel codeModel, Mobi mobi) {
70          this.packageName = packageName;
71          this.codeModel = codeModel;
72          this.mobi = mobi;
73      }
74
75      public void loadAllCodeFiles() {
76          File files = new
    File(INPUT_FOLDER+this.mobi.getContext().getUri()+"/");
77
78          for (File file : files.listFiles()) {
79              if (file.getName().endsWith(EXTENSION)) {
80                  String className = file.getName().substring(0,
    file.getName().length() - EXTENSION.length());
81                  this.filesTokens.put(className,
    LittleLanguage.loadLanguage(file));
82              }
```

Page 2

Mar.java

```
83          }
84      }
85
86      public List<Token> getFilesTokens(String fileName) {
87          return this.filesTokens.get(fileName);
88      }
89
90      public Set<String> getAllImplementedClasses() {
91          return this.filesTokens.keySet();
92      }
93
94      public String getPackagePath(String name) {
95          return this.packageName +".model."+ name;
96      }
97
98      public void createClassCode(String uri, JDefinedClass
    definedClass) {
99          List<Token> tokens = this.filesTokens.get(uri);
100         if (tokens == null) return;
101
102         int instanceIndex = 0;
103
104         this.thisInstances = this.mobi.getClassInstances(uri);
105         this.allInstances.addAll(this.thisInstances);
106
107         for (int i = 0; i < tokens.size(); i++) {
108             Instance instance = new
    Instance(tokens.get(i).getCaracter());
109             if (this.thisInstances.contains(instance)) break;
110             instanceIndex = i + 1;
111         }
112
113         if (instanceIndex > 0)
    this.createStaticCaracteristics(definedClass, tokens.subList(0,
    instanceIndex));
114         if (instanceIndex < tokens.size())
    this.createObjectMethods(definedClass, tokens.subList(instanceIndex,
    tokens.size()));
115     }
116
117     private void createStaticCaracteristics(JDefinedClass
    definedClass, List<Token> tokens) {
118         if (tokens.size() < 2) return;
119
```

Page 3

Mar.java

```
120         for (List<Token> line : Mar.breakInLines(tokens)) {
121             this.createInvocationLine(definedClass, line);
122         }
123     }
124 
125     private void createInvocationLine(JDefinedClass definedClass,
    List<Token> line) {
126         if (line.size() > 1) {
127 
128             if (line.get(1).getName().equals(Token.EQUAL)) {
129                 JType type =
    this.whichClass(line.get(2).getCaracter());
130                 JFieldVar field = definedClass.field(JMod.PUBLIC |
    JMod.STATIC| JMod.FINAL, type, line.get(0).getCaracter());
131                 field.init(this.createMinAttribution(line.subList(2,
    line.size())));
132             }
133         }
134     }
135 
136     private void createMinAttribution(JBlock block, List<Token>
    tokens) {
137         if (tokens.get(0).equals(BEGIN_TOKEN) &&
    tokens.get(tokens.size()-1).equals(END_TOKEN)) {
138             JInvocation jInvoke =
    block.invoke(this.createMinAttribution(tokens.subList(1, 2)),
    tokens.get(2).getCaracter());
139 
140             if (tokens.size() > 4) {
141 
    jInvoke.arg(this.createMinAttribution(tokens.subList(3, 4)));
142             }
143         }
144     }
145 
146     private JExpression createMinAttribution(List<Token> tokens) {
147 
148         if (tokens.get(0).equals(BEGIN_TOKEN) &&
    tokens.get(tokens.size()-1).equals(END_TOKEN)) {
149             JInvocation jInvoke =
    JExpr.invoke(this.createMinAttribution(tokens.subList(1, 2)),
    tokens.get(2).getCaracter());
150 
151             if (tokens.size() > 4) {
```

Page 4

Mar.java

```
152
    jInvoke.arg(this.createMinAttribution(tokens.subList(3, 4)));
153             }
154
155             return jInvoke;
156         }
157
158         String op = Token.MORE;
159         int index = tokens.indexOf(MORE_TOKEN);
160
161         if (index < 0) {
162             index = tokens.indexOf(TIMES_TOKEN);
163             op = Token.TIMES;
164         }
165
166         if (index < 0) {
167             index = tokens.indexOf(LESS_TOKEN);
168             op = Token.SUB;
169         }
170
171         if (index < 0) {
172             index = tokens.indexOf(MAJOR_TOKEN);
173             op = Token.MAJOR;
174         }
175
176         if (index < 0) {
177             index = tokens.indexOf(MINOR_TOKEN);
178             op = Token.MINOR;
179         }
180
181         if (index < 0) {
182             index = tokens.indexOf(IF_NOT_TOKEN);
183             op = Token.IF_NOT_EQUAL;
184         }
185
186         if (index < 0) {
187             String value = "";
188             for (Token token : tokens) {
189                 if (token.getCaracter() == null) continue;
190                 value += this.replaceInstance(token.getCaracter());
191             }
192
193             JType type = this.whichClass(value);
194
```

Page 5

Mar.java

```
195             if (type == null) {
196                 if (value.contains(DOT)) {
197                     return this.invokeGetAttribute(value);
198                 }
199                 return JExpr.ref(value);
200             } else {
201                 return this.getLitValue(type, value);
202             }
203         } else {
204             JExpression expr1 =
    this.createMinAttribution(tokens.subList(0, index));
205             JExpression expr2 =
    this.createMinAttribution(tokens.subList(index+1, tokens.size()));
206 
207             if (op.equals(Token.MORE)) {
208                 return expr1.plus(expr2);
209 
210             } else if (op.equals(Token.MAJOR)) {
211                 return expr1.gt(expr2);
212 
213             } else if (op.equals(Token.MINOR)) {
214                 return expr1.lte(expr2);
215 
216             } else if (op.equals(Token.IF_NOT_EQUAL)) {
217                 return expr1.ne(expr2);
218 
219             } else if (op.equals(Token.SUB)) {
220                 return expr1.minus(expr2);
221 
222             } else {
223                 return expr1.mul(expr2);
224             }
225         }
226     }
227 
228     private String[] convertInstancesToCalled(String instances) {
229         String[] called = instances.split(DOT);
230 
231         String[] result = {
232                 this.replaceInstance(called[0]),
233                 this.fromInstanceToVarieble(called[1])[1]
234             };
235 
236         return result;
```

Page 6

Mar.java

```
237     }
238
239     private JInvocation invokeGetAttribute(String instances) {
240         String[] called = this.convertInstancesToCalled(instances);
241         called[1] = this.attributeToGetMethod(called[1]);
242         return JExpr.invoke(JExpr.ref(called[0]), called[1]);
243     }
244
245     private JInvocation invokeSetAttribute(String instances,
    JExpression arg) {
246         String[] called = this.convertInstancesToCalled(instances);
247         called[1] = this.attributeToSetMethod(called[1]);
248
249         JInvocation setMethod = JExpr.invoke(JExpr.ref(called[0]),
    called[1]);
250         setMethod.arg(arg);
251         return setMethod;
252     }
253
254     private void createIfNotExist(String instances, JType type) {
255         String[] called = instances.split(DOT);
256         for (Instance instance : this.allInstances) {
257             if (instance.getUri().equals(called[0])) {
258
259                 Set<mobi.core.concept.Class> classSet =
    this.mobi.getAllInstanceClassRelation().get(called[0]);
260                 mobi.core.concept.Class mobiClass =
    classSet.iterator().next();
261                 JDefinedClass definedClass =
    this.codeModel._getClass(this.getPackagePath(mobiClass.getUri()));
262
263                 JFieldVar var = definedClass.fields().get(called[1]);
264                 if (var == null) {
265                     this.createAttribute(definedClass, type,
    called[1]);
266                 }
267             }
268         }
269     }
270
271     private String attributeToGetMethod(String value) {
272         return GET+ value.substring(0, 1).toUpperCase() +
    value.substring(1, value.length());
273     }
```

Page 7

Mar.java

```
274
275     private String attributeToSetMethod(String value) {
276         return SET+ value.substring(0, 1).toUpperCase() +
    value.substring(1, value.length());
277     }
278
279     private String replaceInstance(final String name) {
280         for (Instance instance : this.thisInstances) {
281             if (instance.getUri().equals(name)) {
282                 return "this";
283             }
284         }
285
286         return this.fromInstanceToVarieble(name)[1];
287     }
288
289     private void createObjectMethods(JDefinedClass definedClass,
    List<Token> tokens) {
290         if (tokens.size() < 2) return;
291
292         List<List<Token>> lines = Mar.breakInLines(tokens);
293         List<List<List<Token>>> blocks =
    Mar.breakInBlocks(lines.subList(1, lines.size()));
294
295         for (int i=0; i < blocks.size(); i++) {
296             List<List<Token>> block = blocks.get(i);
297
298             if ((i+1) < blocks.size()) {
299                 List<List<Token>> nextBlock = blocks.get(i+1);
300                 String nextMethodName =
    nextBlock.get(0).get(0).getCaracter();
301
302                 if
    (block.get(0).get(0).getCaracter().equals(nextMethodName)) {
303                     this.createMethod(definedClass, block,
    nextBlock);
304                     i++;
305                     continue;
306                 }
307             }
308
309             if (block.size() > 1) {
310                 this.createMethod(definedClass, block, null);
311             } else {
```

Page 8

Mar.java

```
312                 // Se for attributo vem por aqui
313                 // A fazer
314             }
315         }
316     }
317 
318     private void createMethod(JDefinedClass definedClass,
    List<List<Token>> block, List<List<Token>> nextBlock) {
319         String param = null;
320         String methodName = block.get(0).get(0).getCaracter();
321 
322         if (block.get(0).size() > 1) param =
    block.get(0).get(1).getCaracter();
323 
324         block = block.subList(1, block.size());
325 
326         List<Token> terms = new ArrayList<Token>();
327         JType type = null;
328 
329         for (List<Token> line : block) {
330             int index = line.indexOf(new Token(Token.RETURN));
331             if (index >= 0) {
332                 for (Token token : line.subList(index+1,
    line.size())) terms.add(token);
333             }
334 
335             if (terms.size() < 1) type = this.codeModel.VOID;
336             else type = this.getClassType(terms);
337         }
338 
339         JMethod method = definedClass.method(JMod.PUBLIC, type,
    methodName);
340         if (param != null) {
341             String[] variable = this.fromInstanceToVarieble(param);
342             method.param(this.getClassType(variable[0]), variable[1]);
343         }
344 
345         this.implementMethod(definedClass, method, block, nextBlock);
346     }
347 
348     private String[] fromInstanceToVarieble(String instance) {
349         String[] variable = { "", instance };
350         Set<mobi.core.concept.Class> classSet =
    this.mobi.getAllInstanceClassRelation().get(instance);
```

Page 9

Mar.java

```
351
352         if (classSet == null || classSet.isEmpty()) return variable;
353
354         mobi.core.concept.Class className =
    classSet.iterator().next();
355
356         variable[0] = className.getUri();
357         variable[1] = className.getUri().substring(0,
    1).toLowerCase();
358         variable[1] += className.getUri().substring(1,
    className.getUri().length());
359         return variable;
360     }
361
362     private void implementMethod(JDefinedClass definedClass, JMethod
    method, List<List<Token>> block, List<List<Token>> nextBlock) {
363
364         JForEach forInferred = null;
365         JConditional jIF = null;
366         JBlock pBlock = null;
367         int oldIndex = 0;
368
369         if (nextBlock != null) {
370             InstanceRelation param = new
    InstanceRelation(block.get(0).get(1).getCaracter());
371
372             mobi.core.concept.Class thisClass =
    this.mobi.getClass(definedClass.name());
373
374             JInvocation invoke = null;
375             Relation rr = null;
376
377             for (Relation relation :
    this.mobi.getAllClassRelations(thisClass)) {
378                 if
    (relation.getInstanceRelationMapB().values().contains(param)) {
379                     invoke =
    JExpr._this().invoke(this.attributeToGetMethod(relation.getUri().split
    (DOT)[1])).invoke("contains");
380
    invoke.arg(this.createMinAttribution(nextBlock.get(0).subList(1, 2)));
381                     jIF = method.body()._if(invoke);
382                 }
383                 rr = relation;
```

Page 10

Mar.java

```
384             }
385
386             if (invoke == null) {
387
388                 if (nextBlock.get(0).size() > 1) {
389                     invoke =
    JExpr._this().invoke(this.attributeToGetMethod(rr.getUri().split(DOT)
    [1])).invoke("contains");
390
    invoke.arg(this.createMinAttribution(nextBlock.get(0).subList(1, 2)));
391                 } else {
392                     String attribute =
    block.get(0).get(0).getCaracter().split(DOT)[1];
393                     invoke =
    JExpr._this().invoke(this.attributeToGetMethod(attribute)).invoke("equ
    als");
394
    invoke.arg(JExpr.lit(Integer.parseInt(nextBlock.get(1).get(2).getCarac
    ter())));
395                 }
396                 jIF = method.body()._if(invoke);
397             }
398
399             List<Token> newLine = new ArrayList<Token>();
400             newLine.add(new Token(Token.ELSE));
401             nextBlock.add(newLine);
402
403             block.addAll(0, nextBlock);
404         }
405
406         for (int i = 0; i < block.size(); i++) {
407             List<Token> line = block.get(i);
408             List<Token> nextLine = null;
409
410             int firstIndex = 0;
411             try {
412                 while (line.get(firstIndex).equals(TAB_TOKEN))
    firstIndex++;
413                 if (!line.contains(ELSE_TOKEN) && (oldIndex >
    firstIndex)) {
414                     pBlock = null;
415                     jIF = null;
416                 }
417                 oldIndex = firstIndex;
```

Page 11

Mar.java

```
418             } catch (IndexOutOfBoundsException iobe) {
419                 continue;
420             }
421 
422             if (i < block.size()-1) {
423                 nextLine = block.get(i+1);
424                 List<Token> newLine = this.convertFor(line,
    nextLine);
425                 if (newLine != null) line = newLine;
426             }
427 
428             Token firstToken = line.get(firstIndex);
429 
430             if (firstToken.getCaracter() != null &&
    firstToken.getCaracter().startsWith("--")) continue;
431 
432             JBlock mBlock = method.body();
433 
434             if (jIF != null) mBlock = jIF._then();
435             if (pBlock != null) mBlock = pBlock;
436 
437             if (line.contains(ELSE_TOKEN)) {
438                 if (jIF != null) {
439                     if (line.size() > firstIndex+1) {
440                         jIF =
    jIF._elseif(this.createIfExpression(definedClass,
    line.subList(firstIndex+2, line.size())));
441                         pBlock = null;
442                     } else {
443                         pBlock = jIF._else();
444                         jIF = null;
445                     }
446                 }
447 
448             } else if (line.contains(IF_TOKEN)) {
449 
450                 jIF =
    mBlock._if(this.createIfExpression(definedClass,
    line.subList(firstIndex+1, line.size())));
451                 pBlock = null;
452 
453             } else if (line.contains(FOR_TOKEN)) {
454                 String[] called =
    line.get(1).getCaracter().split(DOT);
```

Page 12

Mar.java

```
455                 String instanceType =
    this.fromInstanceToVarieble(called[0])[0];
456                 String[] attribute =
    this.fromInstanceToVarieble(called[1]);
457
458                 forInferred =
    mBlock.forEach(this.getClassType(attribute[0]), attribute[1],
    this.invokeCollection(instanceType, attribute[0]));
459                 pBlock = forInferred.body();
460
461                 if (nextLine != null) for (Token token : nextLine) {
462                     if (token.getCaracter() != null &&
    token.getCaracter().contains(DOT)) {
463                         token.setName(Token.NO_TOKEN);
464                         token.setCaracter(attribute[1]);
465                     }
466                 }
467
468             } else if (line.contains(EQUAL_TOKEN)) {
469
470                 Token sideA = firstToken;
471                 List<Token> sideB =
    line.subList(line.indexOf(EQUAL_TOKEN)+1, line.size());
472
473                 JFieldVar fieldB = null;
474                 JType type;
475
476                 if (sideB.size() == 1) {
477                     fieldB =
    definedClass.fields().get(sideB.get(0).getCaracter());
478                     if (fieldB == null) {
479                         type =
    this.whichClass(sideB.get(0).getCaracter());
480                     } else {
481                         type = fieldB.type();
482                     }
483                 } else {
484                     type = this.getClassType(sideB);
485                 }
486
487                 if (sideA.getCaracter().contains(DOT)) {
488                     this.createIfNotExist(sideA.getCaracter(),
    this.getClassType(sideB));
489
```

Page 13

Mar.java

```
    mBlock.block().add(this.invokeSetAttribute(sideA.getCaracter(),
    this.createMinAttribution(sideB)));
490                 } else {
491                     JVar fieldA =
    definedClass.fields().get(sideA.getCaracter());
492                     if ((fieldA == null)) {
493                         for (Object obj :
    method.body().getContents()) {
494                             if (obj instanceof JVar) {
495                                 JVar jVar = (JVar) obj;
496                                 if
    (jVar.name().equals(sideA.getCaracter())) {
497                                     fieldA = jVar;
498                                 }
499                             }
500                         }
501                     }
502
503                     if ((fieldA == null)) {
504                         fieldA = method.body().decl(type,
    sideA.getCaracter());
505                         if (fieldB == null) {
506
    fieldA.init(this.createMinAttribution(sideB));
507                         } else {
508
    fieldA.init(JExpr.ref(sideB.get(0).getCaracter()));
509                         }
510                     } else {
511                         if (fieldB == null) {
512                             mBlock.assign(fieldA,
    this.createMinAttribution(sideB));
513                         } else {
514                             mBlock.assign(fieldA,
    JExpr.ref(sideB.get(0).getCaracter()));
515                         }
516                     }
517                 }
518
519                 if (forInferred != null) {
520                     forInferred = null;
521                     pBlock = null;
522                 }
523
```

Page 14

Mar.java

```
524             } else if (line.contains(RETURN_TOKEN)) {
525                 List<Token> sideB = line.subList(firstIndex+1,
    line.size());
526                 mBlock._return(this.createMinAttribution(sideB));
527 
528             } else {
529                 this.createMinAttribution(mBlock,
    line.subList(firstIndex, line.size()));
530             }
531         }
532     }
533 
534     private JExpression invokeCollection(String className, String
    attributeName) {
535         Class mobiClass = this.mobi.getClass(className);
536         for (Relation relation :
    this.mobi.getAllClassCompositionRelations(mobiClass)) {
537             System.out.println("$ C="+ className  +" R="+
    relation.getUri());
538             if (relation.getUri().contains(DOT) &&
    relation.getUri().split(DOT)[2].equals(attributeName)) {
539                 return
    JExpr.ref(className.toLowerCase()).invoke(this.attributeToGetMethod(re
    lation.getUri().split(DOT)[1]));
540             }
541         }
542         return null;
543     }
544 
545     private List<Token> convertFor(final List<Token> line, final
    List<Token> nextLine) {
546 
547         int il = -1; int in = -1;
548 
549         for (int i = 0; i < line.size(); i++) {
550             Token token = line.get(i);
551             if (token.getCaracter() != null &&
    token.getCaracter().contains(DOT)) {
552                 il = i; break;
553             }
554         }
555 
556         for (int i = 0; i < nextLine.size(); i++) {
557             Token token = nextLine.get(i);
```

Page 15

Mar.java

```
558             if (token.getCaracter() != null &&
    token.getCaracter().contains(DOT)) {
559                 in = i; break;
560             }
561         }
562 
563         if (il < 0 || in < 0 || (!line.subList(0,
    il).equals(nextLine.subList(0, in)))) return null;
564 
565         String[] tokenLine = line.get(il).getCaracter().split(DOT);
566         String[] tokenNext =
    nextLine.get(in).getCaracter().split(DOT);
567 
568         if (!tokenLine[0].equals(tokenNext[0])) return null;
569 
570         String[] attribute =
    this.fromInstanceToVarieble(tokenLine[1]);
571         String[] attributeNext =
    this.fromInstanceToVarieble(tokenNext[1]);
572 
573         if (!attribute[0].equals(attributeNext[0])) return null;
574 
575         final List<Token> result = new ArrayList<Token>();
576         result.add(FOR_TOKEN);
577         result.add(line.get(il));
578         result.addAll(line);
579 
580         return result;
581     }
582 
583     private JExpression createIfExpression(JDefinedClass definedClass,
    List<Token> tokens) {
584 
585         int index = tokens.indexOf(new Token(Token.AND));
586 
587         if (index < 0) {
588             return this.createMinAttribution(tokens);
589         } else {
590             JExpression expr1 =
    this.createMinAttribution(tokens.subList(0, index));
591             JExpression expr2 =
    this.createMinAttribution(tokens.subList(index+1, tokens.size()));
592             return expr1.cand(expr2);
593         }
```

Page 16

Mar.java

```
594     }
595
596     private JType whichClass(String attribute) {
597         try {
598             return
    this.codeModel.ref(Integer.valueOf(attribute).getClass());
599         } catch (Exception e) {
600             try {
601                 return
    this.codeModel.ref(Double.valueOf(attribute).getClass());
602             } catch (Exception e1) {
603                 try {
604                     return this.codeModel.ref(new
    Date(Long.valueOf(attribute)).getClass());
605                 } catch (Exception e2) {
606                     if (attribute.equals("YES") ||
    attribute.equals("NO")) {
607                         return
    this.codeModel.ref(Boolean.valueOf(attribute).getClass());
608                     } else if (attribute.startsWith("\"")) {
609                         return
    this.codeModel.ref(String.valueOf(attribute).getClass());
610                     } else {
611                         return null;
612                     }
613                 }
614             }
615         }
616     }
617
618     public JType getClassType(List<Token> tokens) {
619         if (tokens.size() > 0) {
620             JType type = this.whichClass(tokens.get(0).getCaracter());
621             if (type != null) return type;
622         }
623
624         if (tokens.contains(new Token(Token.TIMES)) ||
    tokens.contains(new Token(Token.MORE))) {
625             return this.codeModel.ref(Double.class);
626         } else {
627             for (Token token : tokens) {
628                 if (token.getCaracter() == null) continue;
629
630                 if (token.getCaracter().contains(DOT)) {
```

Page 17

Mar.java

```
631                     String[] attribute =
    token.getCaracter().split(DOT);
632                     String[] instance =
    this.fromInstanceToVarieble(attribute[0]);
633                     JDefinedClass definedClass =
    this.codeModel._getClass(this.getPackagePath(instance[0]));
634                     JFieldVar field =
    definedClass.fields().get(attribute[1]);
635                     if (field == null) {
636                         String[] var =
    this.fromInstanceToVarieble(attribute[1]);
637                         field = definedClass.fields().get(var[1]);
638                         if (field == null) {
639                             for (JFieldVar fieldVar :
    definedClass.fields().values()) {
640                                 if
    (fieldVar.type().name().contains(var[0])) {
641                                     field = fieldVar;
642                                     break;
643                                 }
644                             }
645                         }
646                     }
647                     return field.type();
648                 } else if
    (tokens.get(0).getCaracter().contains("valor")) {
649                     return this.codeModel.ref(Double.class);
650                 }
651             }
652         }
653         return this.codeModel.ref(String.class);
654     }
655
656     public JType getClassType(String name) {
657         if (name.startsWith("List-")) {
658             return
    this.codeModel.ref(List.class).narrow(this.getClassType(name.split("-"
    )[1]));
659         } else if (AttributeTypeEnum.contains(name)) {
660             return this.codeModel.ref(name);
661         } else {
662             JDefinedClass type =
    this.codeModel._getClass(this.getPackagePath(name));
663             if (type == null) {
```

Page 18

Mar.java

```
664                 try {
665                     return
    this.codeModel.ref(this.getPackagePath(name)).elementType();
666                 } catch (Exception e) {
667                     e.printStackTrace();
668                 }
669             }
670             return type;
671         }
672     }
673
674     public JType getClassType(Attribute attribute) {
675         JType type = null;
676
677         switch (attribute.getType()) {
678         case INTEGER:
679             type = this.codeModel.ref(Integer.class);
680             break;
681         case LONG:
682             type = this.codeModel.ref(Long.class);
683             break;
684         case DOUBLE:
685             type = this.codeModel.ref(Double.class);
686             break;
687         case STRING:
688             type = this.codeModel.ref(String.class);
689             break;
690         case DATE:
691             type = this.codeModel.ref(Date.class);
692             break;
693         case BOOLEAN:
694             type = this.codeModel.ref(Boolean.class);
695             break;
696         }
697
698         if (attribute.isMany()) {
699             return this.codeModel.ref(List.class).narrow(type);
700         } else {
701             return type;
702         }
703     }
704
705     private JExpression getLitValue(JType type, String value) {
706         switch (type.name()) {
```

Page 19

Mar.java

```
707         case "Integer":
708             return this.getLitValue(AttributeTypeEnum.INTEGER, value);
709         case "Double":
710             return this.getLitValue(AttributeTypeEnum.DOUBLE, value);
711         case "String":
712             return this.getLitValue(AttributeTypeEnum.STRING, value);
713         case "Date":
714             return this.getLitValue(AttributeTypeEnum.DATE, value);
715         case "Boolean":
716             value = (value.equals("YES")) ? "true" : "false";
717             return this.getLitValue(AttributeTypeEnum.BOOLEAN, value);
718         default:
719             return JExpr._null();
720         }
721     }
722 
723     public JExpression getLitValue(AttributeTypeEnum type, String
    value) {
724         switch (type) {
725         case INTEGER:
726             return JExpr.lit(Integer.valueOf(value));
727         case DOUBLE:
728             return JExpr.lit(Double.valueOf(value));
729         case STRING:
730             return JExpr.lit(value);
731         case DATE:
732             return JExpr.lit(value);
733         case BOOLEAN:
734             return JExpr.lit(Boolean.valueOf(value));
735         default:
736             return JExpr._null();
737         }
738     }
739 
740     public JFieldVar createAttribute(JDefinedClass definedClass, JType
    type, String name) {
741         JFieldVar dataType = definedClass.field(JMod.PRIVATE, type,
    name);
742 
743         this.generateGetMethod(definedClass, dataType);
744         this.generateSetMethod(definedClass, dataType);
745 
746         return dataType;
747     }
```

Page 20

Mar.java

```
748
749     /**
750      * Generates a public [classType] get() method to the given class
751      * @param definedClass : The class to generate the method
752      * @param attribute : The class attribute to be returned
753      */
754     public void generateGetMethod(final JDefinedClass definedClass,
    final JFieldVar attribute) {
755         String name = GET + attribute.name().substring(0,
    1).toUpperCase() + attribute.name().substring(1,
    attribute.name().length());
756         JMethod getMethod = definedClass.method(JMod.PUBLIC,
    attribute.type(), name);
757         getMethod.body()._return(attribute);
758     }
759
760     /**
761      * Generates a public void set([classType]) method to the given
    class
762      * @param definedClass : The class to generate the method
763      * @param attribute : The class attribute to be modified
764      */
765     public void generateSetMethod(final JDefinedClass definedClass,
    final JFieldVar attribute) {
766         String name = SET + attribute.name().substring(0,
    1).toUpperCase() + attribute.name().substring(1,
    attribute.name().length());
767         JMethod setMethod = definedClass.method(JMod.PUBLIC,
    this.codeModel.VOID, name);
768
769         setMethod.param(attribute.type(), attribute.name());
770
    setMethod.body().assign(JExpr._this().ref(definedClass.fields().get(at
    tribute.name())), attribute);
771     }
772
773     private static List<List<Token>> breakInLines(List<Token> tokens)
    {
774
775         final List<List<Token>> lines = new ArrayList<List<Token>>();
776
777         for (int fromIndex = 0; fromIndex < tokens.size(); fromIndex+
    +) {
778             int eolIndex = tokens.subList(fromIndex,
```

Page 21

Mar.java

```
   tokens.size()).indexOf(new Token(Token.EOL)) + fromIndex;
779
780             if (fromIndex <= eolIndex) {
781                 lines.add(tokens.subList(fromIndex, eolIndex));
782                 fromIndex = eolIndex;
783             }
784         }
785         return lines;
786     }
787
788     private static List<List<List<Token>>>
   breakInBlocks(List<List<Token>> lines) {
789
790         final List<List<List<Token>>> blocks = new
   ArrayList<List<List<Token>>>();
791         List<List<Token>> block = null;
792
793         for (List<Token> line : lines) {
794             if (line.size() <= 1) continue;
795
796             if (line.get(0).getName().equals(Token.TAB)) {
797
798                 if (line.get(1).getName().equals(Token.TAB)) {
799                     block.add(line.subList(2, line.size()));
800                 } else {
801                     if (block != null) blocks.add(block);
802                     block = new ArrayList<List<Token>>();
803                     block.add(line.subList(1, line.size()));
804                 }
805
806             } else break;
807         }
808
809         blocks.add(block);
810         return blocks;
811     }
812
813 }
```

Page 22

Referências Bibliográficas

[Alexander 2003]Alexander, R. The real costs of aspect-oriented programming? *IEEE Software*, p. 91–93, Novembro 2003. 1.1, 4.1

[Andrade 2009]Andrade, A. *Pensamento Sistêmico - Caderno de campo: O desafio da mudança sustentada nas organizações e na sociedade*. [S.l.]: Bookman, 2009. 1, 2.1

[Apple 2012]Apple, I. *Programming with Objective-C*. Dezembro 2012. Https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithC CH1-SW1. 4.1

[Booch, Rumbaugh e Jacobson 1998]Booch, G.; Rumbaugh, J.; Jacobson, I. *Unified Modeling Language User Guide*. [S.l.]: Addison-Wesley, 1998. 1

[Buse e Weimer 2010]Buse, R.; Weimer, W. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, v. 36, n. 4, p. 546–558, 2010. 1.6

[Checkland e Holwell 2007]Checkland, P.; Holwell, S. Action research: Its nature and validity. Information Systems Action Research, v. 13, 2007. 3

[Codd 1970]Codd, E. F. Relational completeness of data base sublanguages. Database Systems: 65–98, 1970. 1

[Fowler 2005]Fowler, M. *UML Essencial - Um breve guia para a Linguagem-Padrão de Modelagem de Objetos*. São Paulo: Bookman, 2005. ISBN 0-321-19368-7. 2.2

[Gamma et al. 2011]Gamma, E. et al. *Padrões de Projeto: soluções reutilizáveis de software orientado a objetos*. [S.l.]: Bookman, 2011. ISBN 9788577800469. 1.3, 4, 4.1, 5.2.2

[Grilo et al. 2017]Grilo, M. et al. Robustness in semantic networks based on cliques. *Physica A: Statistical Mechanics and its Applications*, v. 472, April 2017. 2.2

[Gruber 1993]Gruber, T. A translation approach to portable ontology specifications. *Knowledge Acquisition 5, 199-220*, 1993. 2.2, 2.3

[Guba e Lincoln 1994]Guba, E.; Lincoln, Y. Competing paradigms in qualitative research, in handbook of qualitative research. *Thousand Oaks: Sage. Applied Social Research Methods Series*, 1994. 1.6

[Hsu 2012]Hsu, C. Visual modeling for web 2.0 applications using model driven architecture approach. *Simulation Modeling Practice and Theory*, 2012. 4.1, 4.1

[Hurlimann 1998]Hurlimann, T. Modeling languages: A new paradigm of programming. *CiteSeerX*, n. 1217-45922.95, 1998. 1

[Jorge et al. 2012]Jorge, E. et al. *Modelo de Ontologia Baseada em Instâncias.* Tese (Doutorado em Difusão do Conhecimento) — UNEB, LNCC, SENAI e UEFS, Salvador, Bahia, Brasil, 2012. 1, 2.4, 2.1, 2.4, 2.5, 4.2, 4.5

[Kedar 2007]Kedar, S. *Programming Paradigms and Methodology.* Pune: Technical Publications Pune, 2007. 2.2, 2.3

[Kleppe, Warmer e Bast 2004]Kleppe, A.; Warmer, J.; Bast, W. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Boston: Addison-Wesley, 2004. ISBN 0-321-19442-X. 2.5, 2.5

[Kotiadis e Robinson 2008]Kotiadis, K.; Robinson, S. Conceptual modelling: Knowleged acquisition and model abstraction. Proceedings of the 2008 Winter Simulation Conference, p. 952–958, 2008. 1

[Larman 2007]Larman, C. *Utilizando UML e Padrões.* São Paulo: Laser House, 2007. 4.1

[Lima e Alvarenga 2008]Lima, G.; Alvarenga, L. Mapa conceitual como ferramenta para organização do conhecimento em sistema de hipertextos e seus aspectos cognitivos. Perspectivas em Ciência da Computação, 2008. ISSN 1981-5344. 1

[Luo, Peng e Lv 2013]Luo, R.; Peng, X.; Lv, Q. A mda based modeling and implementation for web app. *Journal of Software*, 2013. 4.1, 4.1

[Naur 1969]Naur, P. *Software Engineering: Concepts and Techniques.* [S.l.]: Petrocelli/Charter, Ed., 1969. 4.3

[OMG 2014]Omg, O. M. G. *Model Driven Architecture: MDA Guide rev. 2.0.* [S.l.]: Object Management Group, Junho 2014. Http://www.omg.org/cgi-bin/doc?ormsc/14-06-01. 1.1, 2.5

[Papert 1991]Papert, S. New images of programming: in search of an educationally powerful concept of technological fluency. *National Science Foundation*, 1991. 2.3

[Ramos 2006]Ramos, R. A. *Treinamento Prático em UML.* São Paulo: Digerati Books, 2006. ISBN 85-7702-051-7. 2.2

[Sebesta 2011]Sebesta, R. W. *Conceitos de linguagens de programação.* Porto Alegre: Bookman, 2011. 1.6, 2.2

[Sommerville 2011]Sommerville, I. *Engenharia de Software.* São Paulo: Pearson Prentice Hall, 2011. 1.1, 2.3, 3, 4.1

[Sowa 2000]Sowa, J. *Knowledge Representation: Logical, Philosophical, and Computational Foundations.* California: Brooks/Cole, ISBN 0-534-94965, Pacific Grove., 2000. 2.2

[Tucker e Noonan 2010]Tucker, A.; Noonan, R. *Linguagens de Programação - Princípios e Paradigmas (Segunda Edição).* São Paulo: McGraw-Hill, 2010. ISBN 978-85-7726-044-7. 1

[Vet e Mars 1998]Vet, P. E.; Mars, N. J. Bottom-up construction of ontologies. *IEEE Transactions on Knowledge Data Engineering*, v. 10, p. 513–526, 1998. 2.4

[W3C 2014]W3c, R. C. W. G. *Resource Description Framework (RDF)*. [S.l.]: World Wide Web Consortium (W3C), Fevereiro 2014. Https://www.w3.org/RDF/. 2.3

[Wynne e Hellesoy 2012]Wynne, M.; Hellesoy, A. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. [S.l.]: Pragmatic Bookshelf, 2012. ISBN 1934356808 9781934356807. 1.1