MINISTÉRIO DA CIÊNCIA, TECNOLOGIA, INOVAÇÕES E COMUNICAÇÕES
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

sid.inpe.br/mtc-m21c/2019/02.19.00.12-TDI

UMA ARQUITETURA PARA UTILIZAÇÃO DE FRAMEWORKS REFLEXIVOS NA PARTE COMPORTAMENTAL DE MODELOS DE OBJETOS ADAPTATIVOS

Antonio de Oliveira Dias

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelo Dr. Eduardo Martins Guerra, aprovada em 25 de fevereiro de 2019.

URL do documento original:
<http://urlib.net/8JMKD3MGP3W34R/3SPKGPL>

INPE
São José dos Campos
2019

PUBLICADO POR:

Instituto Nacional de Pesquisas Espaciais - INPE
Gabinete do Diretor (GBDIR)
Serviço de Informação e Documentação (SESID)
CEP 12.227-010
São José dos Campos - SP - Brasil
Tel.:(012) 3208-6923/7348
E-mail: pubtc@inpe.br

MINISTÉRIO DA CIÊNCIA, TECNOLOGIA, INOVAÇÕES E COMUNICAÇÕES
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

sid.inpe.br/mtc-m21c/2019/02.19.00.12-TDI

UMA ARQUITETURA PARA UTILIZAÇÃO DE FRAMEWORKS REFLEXIVOS NA PARTE COMPORTAMENTAL DE MODELOS DE OBJETOS ADAPTATIVOS

Antonio de Oliveira Dias

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelo Dr. Eduardo Martins Guerra, aprovada em 25 de fevereiro de 2019.

URL do documento original:
<http://urlib.net/8JMKD3MGP3W34R/3SPKGPL>

INPE
São José dos Campos
2019

Dados Internacionais de Catalogação na Publicação (CIP)

Dias, Antonio de Oliveira.

D543a Uma arquitetura para utilização de frameworks reflexivos na parte comportamental de modelos de objetos adaptativos / Antonio de Oliveira Dias. – São José dos Campos : INPE, 2019.

xxiv + 105 p. ; (sid.inpe.br/mtc-m21c/2019/02.19.00.12-TDI)

Dissertação (Mestrado em Computação Aplicada) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2019.

Orientador : Dr. Eduardo Martins Guerra.

1. Modelo comportamental. 2. Meta programação. 3. Adaptative Object Model (AOM). 4. Engenharia de software. 5. Framework. I.Título.

CDU 004.41/.42

Aluno (a): *Antonio de Oliveira Dias*

Título: "UMA ARQUITETURA PARA UTILIZAÇÃO DE FRAMEWORKS REFLEXIVOS NA PARTE COMPORTAMENTAL DE MODELOS DE OBJETOS ADAPTATIVOS"

Aprovado (a) pela Banca Examinadora em cumprimento ao requisito exigido para obtenção do Título de *Mestre* em

Computação Aplicada

Dra. Lúbia Vinhas

Presidente / INPE / São José dos Campos - SP

( ) Participação por Video - Conferência

(X) Aprovado ( ) Reprovado

Dr. Eduardo Martins Guerra

Orientador(a) / INPE / São José dos Campos - SP

( ) Participação por Video - Conferência

(X) Aprovado ( ) Reprovado

Dr. Nilson Sant'Anna

Membro da Banca / INPE / SJCampos - SP

( ) Participação por Video - Conferência

(X) Aprovado ( ) Reprovado

Dr. Ademar Manuel Teixeira de Aguiar

Convidado(a) / FEUP / Porto - PT

(X) Participação por Video - Conferência

(X) Aprovado ( ) Reprovado

Este trabalho foi aprovado por:

( ) maioria simples

(X) unanimidade

São José dos Campos, 25 de fevereiro de 2019

"A imaginação é mais importante do que o conhecimento.
O conhecimento é limitado. A imaginação circunda o mundo.

Albert Einstein

Aos grandes pesquisadores que dedicam suas vidas a evolução de toda a humanidade.

AGRADECIMENTOS

Agradecimentos ao Prof. Dr. Eduardo Guerra pelas orientações e pela Atech por me proporcionar tempo para realizar estudos.

RESUMO

Alguns sistemas têm como característica a frequente mudança de regras, que desenvolvidos sem a tecnologia adequada, demandam horas de desenvolvimento para atualização. Um exemplo dessa necessidade, no contexto do INPE, são aplicações que retornam dados de diferentes tipos de sensores, que frequentemente precisam que seus serviços forneçam novas informações, e mesmo uma nova lógica sobre essas informações. Para atender esta demanda por flexibilidade, foi criado o Modelo Adaptativo de Objetos (AOM), um estilo arquitetural em que tipos de entidades, atributos, relacionamentos e comportamentos são representados por meio de instâncias, criadas a partir de metadados consumidos em tempo de execução. Uma das dificuldades na criação de uma arquitetura AOM está no fato de que, por usar uma estrutura diferente para entidades, não é possível utilizar com frameworks tradicionais. Frameworks AOM existentes proveem uma estrutura genérica e alguns componentes, mas não permitem o uso com outros frameworks. O framework Esfinge AOM Role Mapper possui uma abordagem baseada no mapeamento de modelos que permite o uso com frameworks tradicionais, mas sem dar suporte ao modelo comportamental do AOM, equivalente aos métodos nas classes. Dessa forma, o objetivo desse trabalho é definir um modelo arquitetural capaz de mapear a representação do comportamento de entidades entre modelos de classe estáticos, AOMs específicos de domínio e AOMs independentes de domínio, com a finalidade de possibilitar a utilização de frameworks feitos para modelos de classe estáticos em arquiteturas AOM. O primeiro passo foi a avaliação da abordagem do Esfinge AOM Role Mapper através de um experimento, o qual mostrou que ele possui uma boa aceitação com os desenvolvedores e não tem diferença significativa em tempo de desenvolvimento em comparação com desenvolver sem uso de framework. Em seguida, foram implementadas novas funcionalidades no AOM Role Mapper para adicionar o modelo comportamental em sua base e o seu mapeamento para os outros modelos. Por fim, para avaliar se o objetivo de uso de frameworks tradicionais foi atingido, foi realizado um estudo de caso onde o framework Spring é usado em uma aplicação AOM para gerar serviços web em tempo de execução. Como resultado, com a solução proposta foi possível utilizar a funcionalidade com um baixo acoplamento entre o framework tradicional e o framework AOM. Com isso, esse trabalho cumpre o objetivo de apresentar uma solução arquitetural que possibilita o uso de frameworks tradicionais em aplicações AOM, criando o potencial de permitir a construção desse tipo de aplicação de forma mais eficiente e com menor repetição de código.

Palavras-chave: Modelo Comportamental. Meta Programação. Adaptive Object Model (AOM). Engenharia de Software. Framework.

AN ARCHITECTURE FOR USING REFLEXIVE FRAMEWORKS IN THE BEHAVIORAL PART OF ADAPTIVE OBJECT MODELS

ABSTRACT

Some kind of systems have as characteristic the frequent change of rules, that developed without the appropriate technology, demand hours of development for update. An example of this need, in the context of INPE, are applications that return data from different types of sensors, which often need their services to provide new information, and even a new logic on that information. In order to meet this demand for flexibility, created the Adaptive Object Model (AOM), an architectural style in which entity types, attributes, relationships, and behaviors are represented through instances, created from metadata consumed at runtime. One of the difficulties in creating an AOM architecture lies in the fact that by using a different structure for entities, it is not possible to use with traditional frameworks. Existing AOM frameworks provide a generic framework and some components, but do not allow the use with other frameworks. The Esfinge framework AOM Role Mapper has an approach based on the mapping of models that allows the use with traditional frameworks, but without supporting the behavioral model of AOM, equivalent to the methods in the classes. Thus, the objective of this work is to define an architectural model capable of mapping the representation of the behavior of entities between static class models, domain specific AOMs and domain independent AOMs, in order to enable the use with frameworks made for static class models in AOM architectures. The first step was to evaluate the approach of the AOM Esfinge Role Mapper through an experiment, which showed that it has a good acceptance with the developers and does not have significant difference in development time comparing with not using any framework. Then, new functionalities were implemented in AOM Role Mapper to add the behavioral model in its base and its mapping to the other models. Finally, in order to evaluate if the goal of use with traditional frameworks was reached, a case study was carried out where the Spring framework is used in an AOM application to generate web services at runtime. As a result, with the proposed solution it was possible to use the functionality with a low coupling between the traditional framework and the AOM framework. This work accomplishes the objective of presenting an architectural solution that allows the use with traditional frameworks for AOM applications, creating the potential to allow the construction of this type of application more efficiently and with less code repetition.

Keywords: Behavioral Model. Adaptive Object Model. Software Engineering. Framework. Metaprogramming.

LISTA DE FIGURAS

Pág.

2.1 Entity Mapping 15
2.2 Adaptive Object Model 16
2.3 Exemplo de herança sem o padrão Type Object 17
2.4 Exemplo do Padrão Type Object 18
2.5 Exemplo do de classe sem o padrão Property 19
2.6 Exemplo do Padrão Property 20
2.7 Exemplo do Padrão Type Square 21
2.8 Exemplo do Padrão Accountability 22
2.9 Exemplo de uso do padrão Rule Object 23
2.10 Arquitetura Core AOM 24
2.11 Framework OGHMA 26
2.12 Diagrama AOM do framework Oghma 27
2.13 Diagrama de classe do meta-modelo do framework Ink 29
2.14 Representação do Modelo do Framework AOM 31

3.1 Interfaces do AOM RoleMapper 33
3.2 Criação de Adaptadores Dinâmicos adaptado de 39
3.3 Principais componentes do framework AOM RoleMapper 43

4.1 Tempo de Desenvolvimento por tarefa e abordagem 51

5.1 AOM RoleMapper antes do modelo comportamental 60
5.2 AOM RoleMapper depois do Modelo Comportamental 61

6.1 Arquitetura da solução desenvolvida para o estudo de caso 78
6.2 Exemplo de DSM com oito elementos 80
6.3 Tela de inserção de parâmetros para a criação de um serviço web 89
6.4 Dependência das classes do estudo de caso para criação de serviços web em tempo de execução com os frameworks Spring e Esfinge 91

LISTA DE TABELAS

Pág.

2.1 Características dos frameworks AOM 32

4.1 Experiência dos Participantes 48
4.2 Abordagem do Experimento 49
4.3 Estatística Descritiva 51
4.4 Testando Hipóteses 52
4.5 Dificuldades Encontradas 53
4.6 Benefícios Percebidos 54
4.7 Desvantagens Percebidas 54

6.1 Cenários de teste do estudo de caso de criação de serviços web em tempo de execução 79

LISTA DE ABREVIATURAS E SIGLAS

AOM - Adaptive Object Model
JSON - Javascript Object Notation
XML - Extensible Markup Language
DSM - Design Structure Matrix
EL - Expression Language
DSL - Domain Specific Language
MOF - Meta-Object Facility
UML - Unified Modeling Language

SUMÁRIO

Pág.

1 INTRODUÇÃO 1
1.1 Contexto 2
1.2 Objetivo 3
1.3 Metodologia de Pesquisa 4
1.4 Relevância 5
1.5 Originalidade 6
1.6 Organização do Trabalho 7

2 REFERENCIAL TEÓRICO 8
2.1 Reflexão e Introspecção 8
2.2 Metadados 10
2.3 Frameworks baseados em metadados 13
2.4 Modelo de Objetos Adaptativo 15
2.4.1 Type Object 17
2.4.2 Property 18
2.4.3 Type Square 20
2.4.4 Accountability 21
2.4.5 Rule Object 22
2.4.6 Arquitetura core AOM 23
2.5 Frameworks para AOM 25
2.5.1 Arquitetura do framework Oghma 26
2.5.2 Arquitetura do framework Ink 27
2.5.3 Esfinge AOM Role Mapper 29

3 AOM ROLEMAPPER 33
3.1 Modelo AOM Independente de Domínio 33
3.2 Mapeamento para AOM de Domínio Específico 35
3.3 Mapeamento para Modelo de Classes Estáticas 39
3.4 Transformação de Metadados em Anotações 40
3.5 Estrutura Interna 43
3.5.1 Componente Metadata Handler 44
3.5.2 Componente Gestor do Modelo 45
3.6 Limitações 45

4 EXPERIMENTO COM AOM ROLEMAPPER 46
4.1 Visão Geral 46
4.2 O Experimento 46
4.2.1 Meta 47
4.2.2 Variáveis, Tratamentos e Objetos 47
4.2.3 Hipóteses 48
4.2.4 Participantes 48
4.2.5 Abordagens do Experimento 49
4.2.6 Procedimento 49
4.2.7 Tempo de Desenvolvimento (Q1) 50
4.2.8 Documentação (Q2) 52
4.2.9 Dificuldades encontradas (Q3) 52
4.2.10 Benefícios e desvantagens percebidos (Q4) 53
4.2.11 Nível de aceitação (Q5) 54
4.3 Ameaças à Validade 55
4.3.1 Validade Interna 55
4.3.2 Validade Externa 56
4.3.3 Validade de Construção 56
4.3.4 Validade de Conclusão 57

5 IMPLEMENTAÇÃO E MAPEAMENTO DO MODELO COMPORTAMENTAL DO AOM 58
5.1 Mapeamento do padrão Rule Object 58
5.1.1 Mapeamento de Dependente para Independente de Domínio 58
5.1.2 Mapeamento de Independente de Domínio para Classes Estáticas . . . 59
5.2 Implementação do Rule Object no AOM Role Mapper 59
5.3 Funcionalidades do Modelo Comportamental 62
5.3.1 Adição e execução de comportamento 62
5.3.2 Execução de Comportamento na mudança em uma propriedade 65
5.3.3 RuleObject que executa Expression Language 66
5.4 Mapeamento de regras de um AOM específico de domínio para as interfaces do AOM Role Mapper 68
5.5 Modelo Comportamental no Adaptador para JavaBeans 72

6 AVALIAÇÃO E RESULTADOS 75
6.1 Contexto 75
6.2 Objetivo e Questões de Pesquisa 76
6.3 Procedimento de coleta de dados 77

6.4 Procedimento de análise 79
6.5 Execução do Estudo [Resultados] 80
6.6 Descrição e Execução dos Cenários de Teste 81
6.6.1 Criando Serviço Web Dinâmico (Cenário 1/5) 81
6.6.2 Criação de Serviço Web com Fórmula (Cenário 2/5) 83
6.6.3 Serviços Web Com Parâmetros (Cenário 3/5) 84
6.6.4 Conversão de unidade de medida de um Sensor (Cenário 4/5) 86
6.6.5 Adicionar um novo Serviço Web (Cenário 5/5) 88
6.7 Análise de Modularidade 90
6.8 Análise dos Resultados 92
6.8.1 Utilização de Frameworks Tradicionais (Q1) 92
6.8.2 Acoplamento entre Componentes (Q2) 92
6.9 Limitações 93

7 CONCLUSÕES 94
7.1 Contribuições 96
7.2 Trabalhos Futuros 97

REFERÊNCIAS BIBLIOGRÁFICAS 99

GLOSSÁRIO 105

1 INTRODUÇÃO

No desenvolvimento de sistemas de grande complexidade, o design raramente atende a todas as necessidades arquiteturais, pois podem surgir novos atributos em uma entidade ou um novo método para compor uma funcionalidade importante que não foi identificada nas fases iniciais de um projeto. Alguns destes sistemas tem como característica a frequente mudança de regras, que desenvolvidos sem a técnica adequada, demandam horas de desenvolvimento para a manutenção do software. Normalmente, a arquitetura do software não é flexível o suficiente para permitir a incorporação destes novos atributos ou comportamentos sem modificação no código.

Como alternativa para resolver o problema da flexibilidade, foram documentados vários padrões de projeto (GAMMA et al., 1995) e boas práticas de engenharia de software que possibilitam que o software execute suas regras sem que necessite de mudanças em sua implementação. Contudo, uma boa parte dos sistemas não são desenvolvidos adotando boas práticas e padrões de projeto que acrescentem flexibilidade na solução, como permitir a inclusão de um atributo novo em uma entidade ou a alteração de um relacionamento entre duas entidades. Considere o exemplo de um sistema de informação que recebe dados do usuário por meio de uma interface gráfica e armazena estes dados em banco de dados. Em um dado momento, caso se necessite que um novo atributo seja adicionado, isso gera diversas alterações no software em suas várias camadas. É ainda mais trabalhoso no caso deste novo atributo ser utilizado utilizado em cálculos ou em uma validação complexa.

A necessidade de flexibilidade é importante no contexto do INPE em sistemas científicos, nos quais não é possível ter todos os requisitos na fase inicial do projeto. A medida que o projeto tem a sua evolução, os requisitos tornam-se claros e ajustes são necessários para atendê-los. Um software com uma estrutura mais flexível, permite que o projeto de pesquisa possa se adaptar rapidamente e evolua de forma fluida.

Por exemplo, os sistemas de coleta e processamento de dados de sensores meteorológicos e espaciais do INPE, em uma eventual necessidade de adicionar um sensor com uma nova informação significativa para um projeto de pesquisa, ocasiona a alteração no software existente para obter e processar estes novos dados. Com os paradigmas atualmente utilizados no desenvolvimento desse tipo de software, esta mudança torna-se trabalhosa de ser executada.

1.1 Contexto

Uma alternativa para resolver estas questões é a utilização de um estilo arquitetural mais flexível e dinâmico, que permite a adaptação a mudanças de requisitos em tempo de execução. O estilo arquitetural chamado de Adaptive Object Model (AOM) (YODER et al., 2001) define os tipos de entidades, equivalentes a classes, como instâncias a partir de metadados que são lidos em tempo de execução. Esta arquitetura possibilita que os tipos de entidades possam ser alterados dinamicamente por meio da alteração de seus metadados e a reconstrução das entidades a partir destes metadados, alterando o comportamento da aplicação.

Uma característica da arquitetura AOM é o alto custo de criar componentes para suas camadas, devido ao fato do modelo AOM, na maioria das vezes, ser específico de domínio, ou seja, é criado para atender os requisitos específicos de flexibilidade de uma área de negócio. Outra dificuldade é a integração com frameworks tradicionais, feitos para lidarem diretamente com classes da linguagem de programação, chamadas neste trabalho de “classes estáticas”, pois as entidades AOM tem uma estrutura diferente destas.

Frameworks AOM independentes de domínio fornecem uma estrutura AOM com classes genéricas, ou seja, que não são de um domínio específico. Devido ao alto nível de abstração, eles fornecem uma estrutura bem completa, mas que demanda várias configurações para a sua utilização. Dentre estes podemos citar o Oghma (FERREIRA et al., 2009) e o Ink (ACHERKAN et al., 2011) que além de fornecerem uma estrutura AOM, também disponibilizam componentes, como por exemplo, para persistência das entidades. Porém essas soluções não são compatíveis com os frameworks tradicionais, impedindo sua utilização em conjunto com os frameworks AOM e demandando, muitas vezes, soluções específicas dentro das aplicações.

Para melhorar a utilização com modelos AOM específicos de domínio, foi criado o framework Esfinge AOM Role Mapper. Como os outros frameworks citados, ele também provê uma modelo AOM independente de domínio, porém permite que seja criado um modelo AOM específico de domínio, permitindo seu mapeamento por meio de anotações para o modelo genérico(MATSUMOTO; GUERRA, 2012) (GUERRA EDUARDO; AGUIAR, 2014) . Outra diferença deste framework para os outros frameworks AOM é a funcionalidade de criação de adaptadores em tempo de execução que encapsulam as entidades AOM e disponibilizam uma API similar a utilizada em classes estáticas (GUERRA et al., 2015), com métodos de acesso começados com "get"e "set". Dessa forma, os frameworks tradicionais podem ser utilizados nas entidades

AOM a partir desses adaptadores.

Contudo, antes deste trabalho, o framework Esfinge AOM Role Mapper não implementava o modelo comportamental de uma estrutura AOM, normalmente implementada com o padrão Rule Object. O Rule Object é utilizado para representar comportamento em uma entidade, assim como os métodos o representam em uma classe.

A importância do modelo comportamental para os frameworks AOM é grande, pois é a partir dessa parte que se modela a lógica atrelada às entidades. Esse modelo comportamental também é utilizado para criação e alteração de regras de negócio em sistemas que tem a mudança frequente de requisitos como característica. Para permitir que esse modelo comportamental possa ser implementado no framework Esfinge AOM Role Mapper, seria preciso também implementar seu mapeamento entre os diferentes modelos, para garantir que os frameworks tradicionais possam invocar os Rule Objects através de métodos nos adaptadores.

Considere, como exemplo, uma aplicação que utiliza um framework tradicional, como o Spring (FRAMEWORK SPRING, 2018), que reconhece métodos de uma classe e os disponibiliza como serviços web. A criação de um novo serviço demanda a adição de métodos no código, precisando passar por todo o ciclo de desenvolvimento, testes e deploy. No caso da chamada do serviço estar atrelada a um Rule Object de uma entidade AOM, novos serviços poderiam ser adicionados e modificados a partir da modificação dos metadados dessa entidade em tempo de execução. Seria necessário o suporte ao modelo comportamental AOM e ao mapeamento entre os diferentes modelos de entidades, para tornar essa arquitetura viável.

1.2 Objetivo

O objetivo dessa dissertação é:

- Definir um modelo arquitetural capaz de mapear a representação do comportamento de entidades entre modelos de classe estáticos, AOMs específicos de domínio e AOMs independentes de domínio, com a finalidade de possibilitar a utilização de frameworks que invocam métodos por reflexão em modelos de classe estáticos em arquiteturas AOM.

Neste projeto de pesquisa, foram desenvolvidas funcionalidades no framework Esfinge AOM Role Mapper com o intuito de implementar o modelo comportamental em seu modelo AOM independente de domínio, permitindo seu mapeamento para

classes estáticas e modelos AOM de domínio específico .

O padrão Rule Object foi utilizado para a inclusão de comportamento no modelo AOM independente de domínio do framework. A partir desse modelo, os adaptadores de entidades já existentes no framework foram estendidos para dar suporte a parte comportamental.

1.3 Metodologia de Pesquisa

Após o estudo da arquitetura AOM, foi feito um experimento que focou em avaliar a utilização do framework Esfinge AOM Role Mapper para a utilização com frameworks tradicionais. Foi escolhido um framework que não atua na parte comportamental, mas nos atributos de uma classe. Foi feita uma comparação da implementação de código que foi criado baseado apenas no modelo AOM com outra implementação com o objetivo da utilização do framework tradicional. Foram avaliados itens como o tempo de desenvolvimento e a experiência dos desenvolvedores em relação a ganho de flexibilidade, curva de aprendizado e reaproveitamento de código.

Em seguida, após análise dos resultados deste experimento, foram desenvolvidas novas funcionalidades no framework Esfinge AOM Role Mapper com o intuito de viabilizar a introdução do modelo comportamental nas entidades. O primeiro passo foi a criação de classes que implementam o padrão de projeto Rule Object (YODER; JOHNSON, 2002) no modelo AOM independente de domínio provido pelo framework. Depois, os adaptadores foram atualizados para que os objetos AOM específicos de domínio pudessem ser mapeados para essa estrutura. A partir deste mapeamento, a classe que cria adaptadores para as entidades AOM específicas de domínio consegue reconhecer implementações do Rule Object e fazer o mapeamento de comportamento para a estrutura do framework.

Por fim, o modelo comportamental também foi adicionado na geração de adaptadores para a API de classes estáticas, permitindo a utilização destes objetos adaptados com frameworks tradicionais que também atuam com a invocação de métodos por reflexão. Nesse mapeamento, os metadados dos Rule Objects se transformam em anotações nos respectivos métodos dos adaptadores. Este passo é importante pois os frameworks tradicionais utilizam anotações para localizar métodos para serem invocados.

Para avaliar se é possível atingir os objetivos dessa dissertação com esta abordagem,

foi desenvolvido um estudo de caso que utiliza Esfinge AOM Role Mapper para utilizar o framework Spring. Esse estudo de caso é baseado em aplicações que disponibilizam dados coletados por sensores, aplicáveis para diversos domínio de interesse do INPE, como o de clima espacial (SANT'ANNA et al., 2014). No estudo de caso desenvolvido, foram criados serviços web baseados em entidades AOM criadas em tempo de execução. Os Rule Objects adicionados na entidade são mapeados para métodos com anotações nos adaptadores, os quais são reconhecidos pelo framework Spring(FRAMEWORK SPRING, 2018) e disponibilizados como serviços web.

O estudo de caso baseado em cenários baseou-se em Architecture Tradeoff Analysis Method (ATAM) (KAZMAN et al., 2000). A análise de qualidade da arquitetura baseada em cenários arquiteturais e seu custo-benefício é amplamente utilizada atualmente em trabalhos de pesquisa de engenharia de software como (BEGUM; RAJ, 2018) e (NAAB; ROST, 2018).

Para avaliar o acoplamento do software desenvolvido no estudo de caso, foi utilizado o Design Structure Matrix (DSM) (YASSINE, 2004), utilizado para avaliar o acoplamento entre componentes de software em muitos trabalhos de pesquisa recentes, por exemplo trabalhos como (SANAEI et al., 2015) e (BENKOCZI et al., 2018).

1.4 Relevância

A relevância do trabalho de pesquisa está no uso de aplicações AOM de forma conjunta com os frameworks tradicionais, viabilizando que mais aplicações utilizem este tipo de arquitetura. Uma aplicação desenvolvida utilizando o framework Esfinge AOM Role Mapper permite a flexibilidade de utilização de objetos AOM específicos de domínio sendo mapeados para objetos AOM independentes de domínio e para classes estáticas em uma mesma solução.

Esta flexibilidade permite que alterações de regras de negócio, como por exemplo a criação de um novo método para validar um objeto ou um novo comportamento seja criado devido a um novo atributo coletado de uma fonte externa sejam realizadas sem que haja a necessidade de recompilação de código.

No Instituto Nacional de Pesquisas Espaciais (INPE), a necessidade de sistemas dinâmicos surge no contexto de observatórios virtuais (SANTOS et al., 2013). Um laboratório virtual é um ambiente interativo para a criação e execução de experimentos simulados. No caso do INPE, existem diversas bases de dados a respeito de fenômenos meteorológicos, clima espacial e ciências da terra que são de interesse de

pesquisadores que realizam estudos nessas áreas. Uma abordagem viável seria, ao invés dos pesquisadores baixarem uma quantidade muito grande de dados para executarem seus algoritmos, eles enviarem o código de suas análises para ser executado diretamente em um ambiente controlado do laboratório virtual. Para que isso seja possível, a aplicação servidora deve utilizar mecanismos de adaptação para execução de uma lógica que não é previamente conhecida.

O trabalho apresentado nessa dissertação tem como objetivo servir como base para um projeto de pesquisa que desenvolva a arquitetura de uma plataforma que ofereça esse tipo de funcionalidade. A partir dos resultados, será possível usá-los como base em uma arquitetura que permita a incorporação dinâmica de novas funcionalidades, o que será essencial para que essa plataforma de observatório virtual possa receber o código de pesquisadores e transformar em produtos acessíveis a comunidade em geral. Porém, apesar de ser importante mencionar o projeto dessa plataforma para que a relevância desse trabalho possa ser compreendida dentro do contexto do INPE, sua especificação e desenvolvimento está fora do escopo desse trabalho.

1.5 Originalidade

A originalidade deste trabalho de pesquisa está na criação de uma arquitetura que possibilita o uso de frameworks tradicionais em modelos AOM, principalmente no que diz respeito ao modelo comportamental.

Os frameworks AOM independentes de domínio, Oghma (FERREIRA et al., 2009) e Ink (ACHERKAN et al., 2011) possuem implementação do modelo comportamental. Estes frameworks possuem componentes próprios específicos para o seus frameworks AOM e podem ser reutilizados em aplicações que usam o seus próprios modelos, contudo nenhum deles possibilita o uso de frameworks feitos para classes comuns.

Outro trabalho que tem o comportamento dinâmico como tema de pesquisa em (ARSANJANI, 2001, Disponível em https://hillside.net/plop/plop/plop2k/proceedings/Arsanjani/Arsanjani.pdf) apresenta o termo Rule Object como um padrão de linguagem (Pattern Language), diferente da aplicação do padrão Rule Object, um padrão de modelagem que é aplicado na arquitetura do software.

O trabalho de pesquisa de (DANTAS et al., 2004), informa que AOM pode adicionar comportamento em uma entidade de forma flexível utilizando Aspectos (GRADECKI; LESIECKI, 2003) , mas não apresentou como isso pode ser utilizado para o uso com

os frameworks tradicionais existentes.

1.6 Organização do Trabalho

O trabalho está organizado da seguinte forma:

No capítulo 2, temos o referencial teórico que apresenta conceitos sobre reflexão e metadados, os fundamentos do Modelo de Objetos Adaptativo (AOM) com seus principais padrões de projeto e frameworks AOM que implementam esses conceitos, como o Oghma, Ink e o Esfinge AOM Role Mapper.

O capítulo 3 apresenta em detalhes o framework Esfinge AOM Role Mapper.

O capítulo 4 apresenta o experimento realizado para analisar a utilização do framework Esfinge AOM Role Mapper comparando-o ao desenvolvimento sem utilizar frameworks, mensurando itens como a curva de aprendizado, tempo de desenvolvimento e percepção de ganho na reutilização da solução.

No capítulo 5 apresenta as novas funcionalidades do Esfinge AOM Role Mapper, com as contribuições deste trabalho, para introduzir o modelo comportamental e seu mapeamento entre os modelos.

No capítulo 6 apresenta o estudo de caso para avaliar o uso das funcionalidades do framework Esfinge AOM Role Mapper para a criação de serviços web em tempo de execução adicionando comportamento nas entidades AOM para fornecer objetos para o framework Spring disponibilizá-los como serviços web.

O capítulo 7 apresenta as conclusões deste trabalho de pesquisa, assim como contribuições e trabalhos futuros.

2 REFERENCIAL TEÓRICO

O objetivo deste capítulo é apresentar conceitos importantes no contexto deste trabalho de pesquisa: a seção 2.1 apresenta os conceitos sobre reflexão computacional e introspecção; a seção 2.2 apresenta os conceitos sobre metadados; a seção 2.3 apresenta os frameworks baseados em metadados; a seção 2.4 apresenta o modelo adaptativo de objetos (AOM) e seus padrões de projeto; por fim, a seção 2.5 apresenta os frameworks AOM existentes e suas principais características.

2.1 Reflexão e Introspecção

Reflexão computacional (MAES, 1987) é o comportamento apresentado por sistemas reflexivos quando estes realizam operações sobre si mesmos. Um sistema reflexivo possui, além dos dados que representam o domínio da aplicação, estruturas cujo objetivo é prover sua própria representação. O princípio de reflexão determina que um sistema deve manter uma representação causalmente conectada de seu próprio comportamento, a qual pode ser examinada e modificada pelo próprio sistema.

Metadados são dados sobre os dados. Em uma linguagem de programação orientada a objetos são, por exemplo, os elementos de uma classe, como os atributos, os métodos com os seus parâmetros e sua superclasse (YODER; RAZAVI, 2000). Uma linguagem com suporte a reflexão permite que estes metadados sejam manipulados com intuito de modificar o comportamento do software durante a sua execução.

A técnica de programação que utiliza reflexão para a criação de soluções é chamada de metaprogramação (JOSEPH; YODER, 1998). Devido a possibilidade de mudança no software em execução, considera-se que o uso dessa técnica provê ferramentas para a implementação de requisitos de flexibilidade e adaptação.

Continuando a descrição de conceitos importantes, temos a introspecção. Segundo (DOUCET et al., 2003), a introspecção é um subconjunto da reflexão que permite um software obter informações a respeito de si mesmo. A partir das informações obtidas com a introspecção, é possível manipular instâncias acessando os seus atributos e invocando os seus métodos dinamicamente. A linguagem Java não tem suporte completo a reflexão, contudo tem suporte a introspecção. A partir desse recurso pode-se, por exemplo, descobrir os métodos e suas assinaturas, atributos de um objeto, e partir destas informações, invocar métodos e obter seus resultados.

A especificação JavaBeans (SUN MICROSYSTEMS, 1997) foi criada na linguagem Java para facilitar o uso de reflexão. JavaBeans são classes usadas para encapsular

atributos em um único objeto (chamado de bean). As principais características de uma classe que é chamada de "Bean"são um construtor sem argumentos e a presença de métodos de acesso permitem o acesso as suas propriedades, que seguem uma convenção de nomenclatura padrão com prefixo "get"e "set". Como os JavaBeans seguem um padrão de nome para os métodos, fica mais simples utilizar a API de reflexão para encontrar e invocar dinamicamente destes métodos para acessar as propriedades de um objeto.

A Listagem 2.1 apresenta uma classe com o padrão JavaBeans, com os atributos nome e dados da classe Sensor e seus métodos get e set para manipular estes atributos.

Listagem 2.1 - Exemplo de classe no padrão JavaBeans

```
1  public class Sensor {
2     private String nome;
3     private String dados;
4
5     public String getDados() {
6        return dados;
7     }
8     public String getNome() {
9        return nome;
10    }
11    public void setDados(String dados) {
12       this.dados = dados;
13    }
14    public void setNome(String nome) {
15       this.nome = nome;
16    }
17 }
```

Em seguida, é apresentada na Listagem 2.2 um exemplo para ler e imprimir os valores dos atributos de uma classe que implemente esse padrão, como a classe Sensor, por meio da invocação dos métodos começados em "get"utilizando reflexão. O código se beneficia do padrão JavaBeans para encontrar os métodos de acesso aos atributos da classe.

Listagem 2.2 - Imprimindo valores dos atributos a partir dos métodos get

```
1  public class PrintProperties {
2     public static void printClass(Object obj) {
3        for (Method m : obj.getClass().getDeclaredMethods()) {
4           if (m.getName().startsWith("get")) {
5              System.out.print(m + " => ");
6              try {
7                 System.out.println(m.invoke(obj));
8              } catch (Exception e) {
9                 e.printStackTrace();
```

```
10            }
11          }
12        }
13    }
14 }
```

A Listagem 2.3 apresenta um exemplo de uso do método estático printClass, apresentado na Listagem 2.2 . Neste exemplo, é criada uma instância da classe Sensor, são atribuídos valores para os atributos nome e dado e o objeto sensor é passado como parâmetro para o método printClass imprimir os valores contido nos atributos .

Listagem 2.3 - Exemplo de uso de reflexão na classe Sensor

```
1 public static void main(String [] args) {
2    Sensor sensor = new Sensor();
3    sensor.setNome("Magnetometro");
4    sensor.setDados("123m");
5    PrintProperties.printClass(sensor);
6 }
```

A Listagem 2.4 apresenta o resultado dessa execução.

Listagem 2.4 - Resultado da impressão dos métodos por reflexão

```
1 public java.lang.String tarefa.Sensor.getNome() => Magnetometro
2 public java.lang.String tarefa.Sensor.getDados() => 123m
```

2.2 Metadados

Os metadados inerentes a uma classe muitas vezes não são suficientes para identificar o papel deles no contexto do negócio. Deste modo são necessários metadados adicionais para que os frameworks possam identificar os elementos de uma classe utilizando reflexão. Um exemplo deste caso é framework Hibernate (BAUER; KING, 2005), que utiliza metadados adicionais para identificar qual classe será persistida em base de dados, qual o nome da tabela que a classe identifica e quais os nomes dos atributos da classe serão transformados em colunas das tabelas no banco de dados.

Existem várias formas de configurar metadados, como convenções de código, fontes externas como arquivos XML ou utilizar anotações na própria classe.

Anotações são marcações que fazemos em classes, métodos e atributos com o intuito de identificar estes elementos para a execução de um algoritmo de processamento.

Especialmente na linguagem Java, até a versão 4, não tinha suporte nativo a anotações. Os metadados começaram a ser utilizadas em arquivos externos em formato XML. Uma evolução foi o surgimento de componente Xdoclet que usava tags do javadoc para gerar arquivos XML e código java. Porém estes mapeamentos ficavam armazenados em arquivos separados da classe e causavam erros de processamento na vinculação da classe mapeada com as tags e o arquivo XML. A solução para este cenário foi a criação de anotações na própria classe, a partir do java 5, para facilitar este mapeamento e evitando erros na vinculação do mapeamento dos atributos da classe com as tags de mapeamento, mantendo essa definição menos detalhada e mais próxima do código-fonte.

O uso de anotações de código é chamada de programação orientada a atributo. Algumas linguagens de programação, como Java e C#, têm suporte nativo para anotações. Na linguagem C# este recurso chama-se attribute. Na linguagem Python um recurso chamado Decorator também é utilizado para colocar metadados em elementos de código.

A vantagem das anotações em relação as outras formas é que elas são escritas na própria classe, próximas ao elementos que elas marcam, facilitando a identificação de forma visual.

A Listagem 2.5 apresenta um exemplo de como o framework Hibernate utiliza as anotações para identificar os elementos que ele precisa para manipular os dados de uma classe. Neste exemplo de mapeamento de uma classe utilizando anotações do framework Hibernate, a classe de nome Funcionalidade mapeia a tabela funcionalidade. .

Listagem 2.5 - Exemplo de classe com anotação do framework Hibernate

```
1  @Entity
2  @Table(name="funcionalidade")
3  public class Funcionalidade {
4     @Id
5     @Column(name="id")
6     private int funcionalidadeId;
7
8     public int getId() {
9        return funcionalidadeId;
10    }
11
12    public void setId(int id) {
13       this.funcionalidadeId = id;
14    }
15
16    @Column(name="nome", unique=true, nullable=false)
```

```
17     private String nome;
18
19     public String getNome() {
20        return nome;
21     }
22
23     public void setNome(String nome) {
24        this.nome = nome;
25     }
26  }
```

Na linha 1 está a primeira anotação, @Entity, informando que esta classe é um entidade a ser persistida na base de dados. Na linha 2 a anotação @Table informa que esta entidade será armazenada na base de dados na tabela "funcionalidade". Na linha 4 a anotação @Id informa que o atributo funcionalidadeId é o identificador único da tabela funcionalidade. Na linha 5 a anotação @Column informa que o nome da coluna é id para o atributo funcionalidadeId. Na linha 16 a anotação @Column informa que o nome da coluna será nome, que ele será único e não permitirá valores nulos como valor.

A linguagem java permite a criação de anotações. Na Listagem 2.6 é apresentada um exemplo de criação de uma anotação de nome Imprimir.

Listagem 2.6 - Anotação Imprimir

```
1  @Retention(RetentionPolicy.RUNTIME)
2  @Target(value={ElementType.METHOD})
3  public @interface Imprimir {
4  }
```

Pode-se verificar na linha 1 que a política de retenção da anotação é de RUNTIME, ou seja, esta anotação está disponível para ser recuperada em tempo de execução via reflexão. Conforme descrito na linha 2, o elemento @Target com o valor ElementType.METHOD configura a anotação para ser utilizada apenas em métodos.

Para exemplificar o uso de uma anotação, a Listagem 2.7 apresenta o método getNome utilizando a anotação @Imprimir.

Listagem 2.7 - Exemplo de utilização da anotação @Imprimir

```
1  public class Sensor {
2     private String nome;
3     private String dados;
4
5     public String getDados() {
6        return dados;
7     }
```

```
8
9     @Imprimir
10    public String getNome() {
11       return nome;
12    }
13    public void setDados(String dados) {
14       this.dados = dados;
15    }
16    public void setNome(String nome) {
17       this.nome = nome;
18    }
19 }
```

Na linha 9 do exemplo, o método getNome está sendo mapeado com a anotação @Imprimir.

Na Listagem 2.8 é apresentado um exemplo com o objetivo de demonstrar como a anotação é consumida. O código faz a leitura da anotação @Imprimir e invoca o método getNome que contém esta anotação.

Listagem 2.8 - Invocando método com a anotação @Imprimir

```
1  public class LerAnotacao {
2     public static void invokeMethod(Object obj) {
3        for (Method m : obj.getClass().getDeclaredMethods()) {
4           if (m.isAnnotationPresent(Imprimir.class)) {
5              try {
6                 System.out.println(m.invoke(obj));
7              } catch (Exception e) {
8                 e.printStackTrace();
9              }
10          }
11       }
12    }
13 }
```

Na linha 4 é feita a verificação se anotação Imprimir está presente no método. Na linha 6, após confirmar que a anotação está presente, o método é invocado por reflexão e o resultado é impresso na saída padrão do sistema.

2.3 Frameworks baseados em metadados

De acordo com (JOHNSON; FOOTE, 1988), um framework pode ser considerado um software incompleto com alguns pontos que podem ser especializados para adicionar comportamento específico da aplicação, consistindo em um conjunto de classes que representa um design abstrato para uma família de problemas relacionados. Ele fornece um conjunto de classes abstratas que devem ser especializadas e compostas

com outras para criar uma aplicação concreta e executável. As classes especializadas podem ser específicas da aplicação ou importadas de uma biblioteca de classes, geralmente fornecidas juntamente com o framework.

Os frameworks mais modernos fazem uso da introspecção para acessar os metadados das classes em tempo de execução, como suas superclasses, métodos e atributos. Com isto, é reduzido o acoplamento entre as classes de aplicação e a estrutura de classes abstratas e interfaces do framework. Por exemplo, seguindo esta abordagem, o framework pode pesquisar na estrutura de uma classe pelo método correto para invocar. O uso desta técnica fornece mais flexibilidade para a aplicação, uma vez que o framework lê dinamicamente a estrutura das classes, permitindo que a classe possa ser alterada mais facilmente (FOOTE B., 1996).

Quando um framework usa reflexão (MAES, 1987) para acessar e encontrar os elementos da classe, por vezes, a informação intrínseca da classe pode não ser suficiente. Se o comportamento do framework varia para diferentes classes, métodos ou atributos, é necessário acrescentar meta-informação adicional para permitir esta diferenciação.

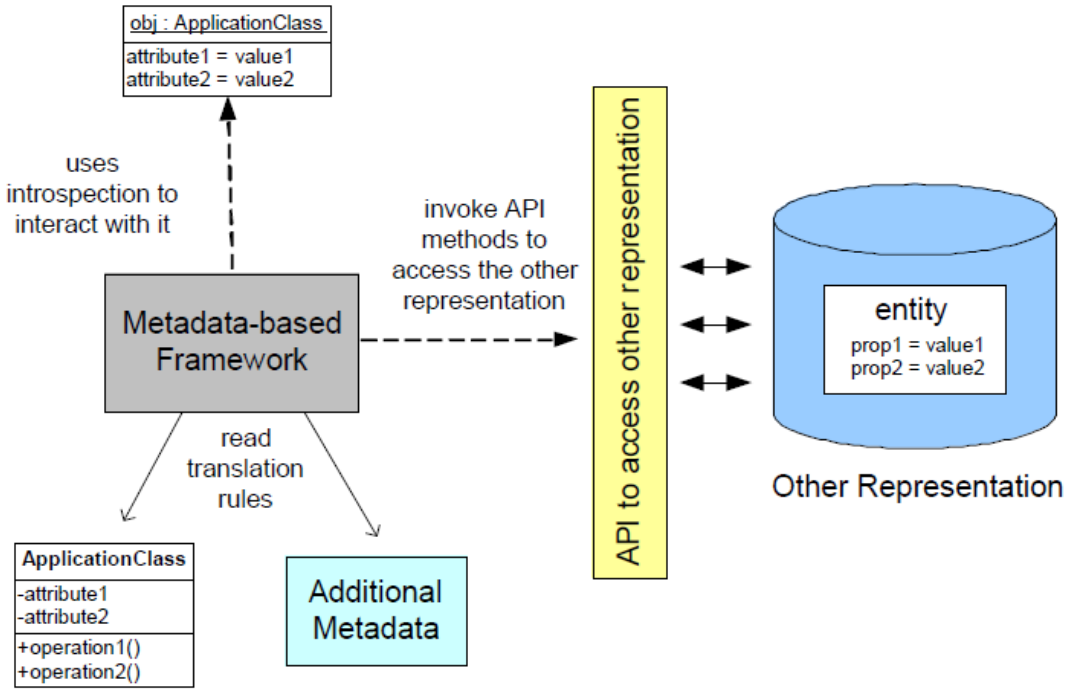
Os frameworks baseados em metadados podem ser definidos como frameworks que processam sua lógica baseada nos metadados de classes (GUERRA et al., 2010). Nesses casos, o desenvolvedor deve definir metadados adicionais específicos de domínio ou da aplicação nas classes de aplicação para serem consumidos e processados pelo framework. O uso de metadados muda a forma como os frameworks são construídos e também seu uso por desenvolvedores de software (O'BRIEN, 2009).

O padrão Entity Mapping (GUERRA et al., 2010) documentou um uso comum de frameworks baseado em metadados, que é o mapeamento de diferentes representações da uma mesma entidade em um sistema.

Por exemplo, uma classe de aplicação pode ser mapeada para um banco de dados para permitir que o framework controle sua persistência, conforme exemplo do framework Hibernate apresentado na Listagem 2.5. Da mesma forma, as entidades podem ser mapeadas para outro esquema de classes ou para um formato XML e um framework que aplica o padrão Entity Mapping, como o Esfinge AOM Role Mapper (ESFINGE FRAMEWORK, 2018), pode traduzir entre as duas representações com base nos metadados de classe.

A Figura 2.1 apresenta a estrutura do padrão Entity Mapping.

Figura 2.1 - Entity Mapping



Fonte: Guerra et al.(2010a)

2.4 Modelo de Objetos Adaptativo

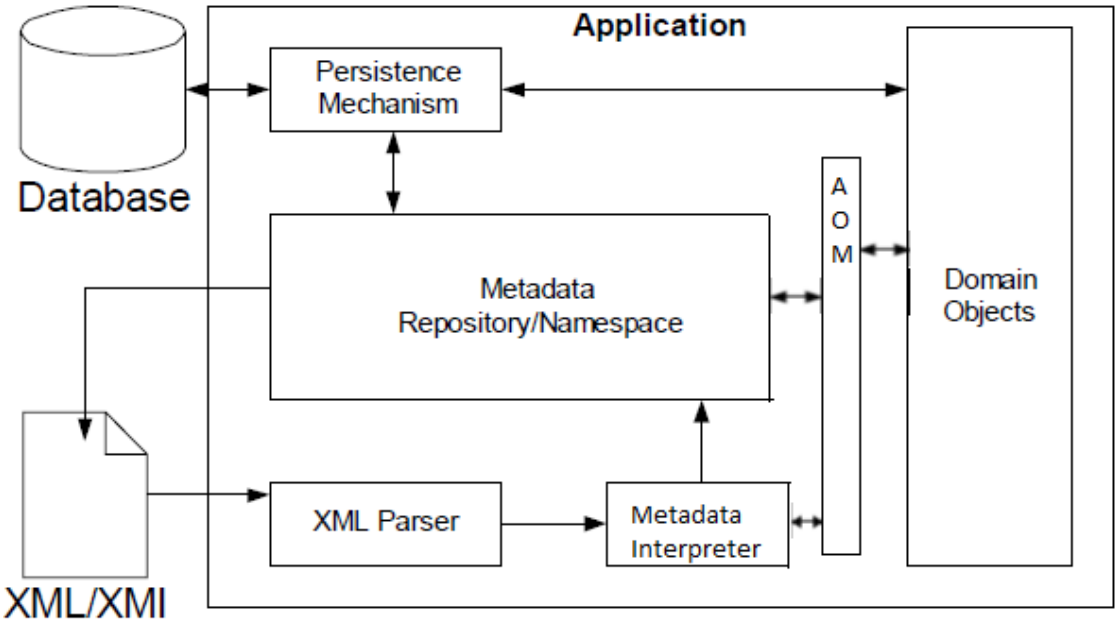
O Modelo de Objetos Adaptativo (Adaptive Object Model - AOM) (YODER; JOHNSON, 2002) é um estilo arquitetural em que entidades, atributos, relacionamentos e comportamentos são representados como instâncias criadas a partir de metadados consumidos em tempo de execução. Novos tipos de entidades podem ser definidos nos metadados, gerando comportamentos diferentes, sem a necessidade de alteração no código fonte.

Isto possibilita que sistemas criados com essa arquitetura sejam flexíveis e seu comportamento possa ser modificado em tempo de execução, diretamente pelos usuários ou especialistas de negócio. Esta flexibilidade permite que as entidades de domínio possam evoluir como parte da utilização do software, sem a necessidade de modificação do código fonte.

Exemplos de uso com sucesso da arquitetura AOM podem ser encontradas nos domínios de seguros (JOHNSON; OAKES, 1998. Disponível em: http://stwww. cs. uiuc. edu/users/johnson/papers/udp) e na área de saúde (YODER; JOHNSON, 2002).

A Figura 2.2 apresenta o primeiro modelo conceitual da arquitetura AOM (YODER; JOHNSON, 2002). Nessa representação, os metadados definidos em XML ou XMI, depois de serem lidos, são enviados ao interpretador de metadados, que armazena essas informações no repositório de metadados. A partir dos dados desse repositório, serão instanciadas as entidades que representam o modelo AOM, nas quais são baseadas as entidades de domínio. Um mecanismo de persistência específico do modelo AOM é utilizado para armazenar as entidades em um banco de dados.

Figura 2.2 - Adaptive Object Model



Fonte: Yoder e Johnson (2002)

No entanto, esta flexibilidade traz como compensação uma complexidade de implementação da arquitetura do sistema. Para sistemas baseados em um modelo de classes estáticas, existem diversos frameworks que facilitam a implementação de diversas camadas do sistema. No caso de sistemas que utilizam a arquitetura AOM, funcionalidades como a persistência, apresentação (WELICKI et al., 2007) e validação, precisam ser implementadas de maneira específica para o modelo criado.

Os componentes das aplicações AOM normalmente são criados de forma a fornecer apenas a flexibilidade necessária para uma aplicação específica. Um modelo AOM muito genérico pode chegar próximo a uma nova linguagem de programação para

a definição dos metadados. Devido a estas diferenças intrínsecas a necessidade de negócio de cada aplicação, os componentes criados para o modelo AOM de uma aplicação são difíceis de serem utilizados em outro sistema.

A estrutura central da arquitetura AOM é composta pelos padrões Type Object, Property, Type Square, Accountability e RuleObject. Essa estrutura central será descrita nas seções seguintes.

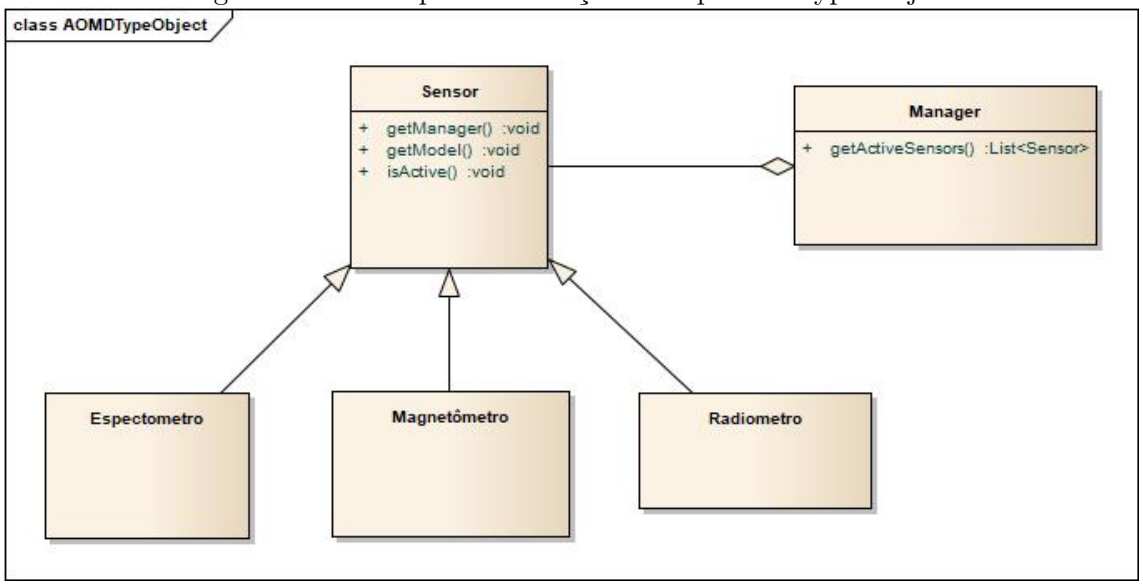
Para ilustrar estes padrões, serão utilizado exemplos com Sensores. A ideia é utilizar um conceito chamado de Running Example, que é um exemplo que é desenvolvido e incrementado no decorrer das seções.

2.4.1 Type Object

O padrão Type Object (JOHNSON R., 1997) é utilizado em situações em que o número de subclasses que uma classe pode aumentar exponencialmente ou não pode ser determinado durante o desenvolvimento do sistema.

Por exemplo, em um sistema de captação e uso de dados de vários sensores (Running Example), é necessário que sensores de diferentes tipos possam ser inseridos no sistema. A solução padrão da orientação a objetos para implementar este requisito, seria criar uma classe Sensor com uma subclasse para cada tipo diferente, conforme a Figura 2.3 .

Figura 2.3 - Exemplo de herança sem o padrão Type Object

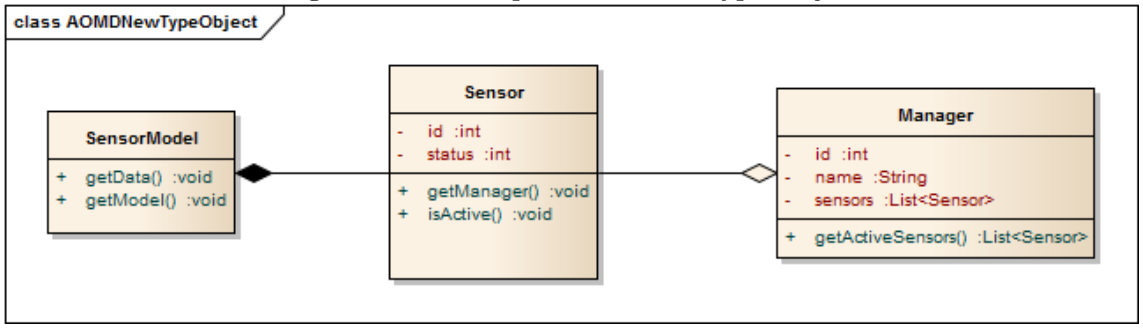


Com essa solução, se for necessário inserir um novo modelo de sensor no sistema, precisaria ser criada uma nova subclasse de Sensor, recompilar o código e instalar a atualização do sistema. Além do trabalho necessário para realizar essa mudança, com a evolução do sistema, é possível que a classe Sensor passe a ter um grande número de subclasses com pequenas diferenças entre elas.

O padrão Type Object resolve essa situação representando as subclasses que não são conhecidas em tempo de desenvolvimento como instâncias de uma classe que representa o tipo. Dessa forma, a classe principal é composta com a instância que representa o tipo correspondente.

Neste exemplo, representado na Figura 2.4, os vários modelos de Sensor podem ser representados com instâncias da classe SensorModel. A relação classe-instância que existia no modelo antigo passa a ser representadas como uma relação de composição, onde as instâncias de Sensor são compostas por uma referência a instância do SensorModel correspondente. Por meio dessas referências é possível determinar o modelo de um sensor. Esta solução permite a adição dinâmica de novos tipos de sensor, representados agora por instâncias, no sistema.

Figura 2.4 - Exemplo do Padrão Type Object

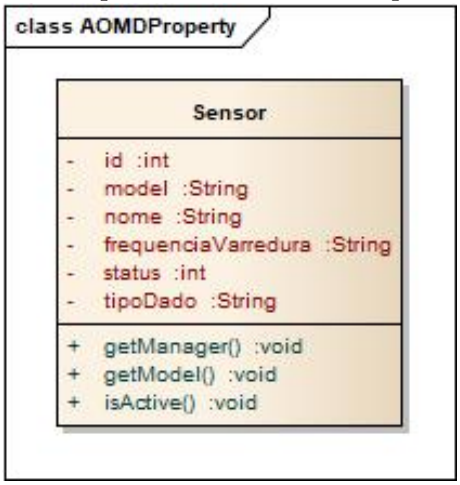


2.4.2 Property

O padrão Property (FOWLER, 1996) é aplicado em situações em que as entidades de um mesmo tipo podem possuir diferentes propriedades. Criar um atributo para representar cada uma dessas propriedades pode não ser a melhor solução, pois muitos podem acabar ficando vazios nas instâncias. Outro problema seria que a cada nova propriedade que fosse necessária, uma modificação no código precisaria ser feita.

Por exemplo, em um sistema de multi sensoriamento (Running Example), pode-se criar uma classe Sensor para armazenar informações de sensores, como nome do sensor, tipo de dados captados, frequência de varredura, dentre outros. A solução segundo a orientação a objetos clássica para representar esses conceitos seria adicionar um atributo para cada tipo de informação necessária para o sensor na classe Sensor, conforme apresentado na Figura 2.5.

Figura 2.5 - Exemplo do de classe sem o padrão Property

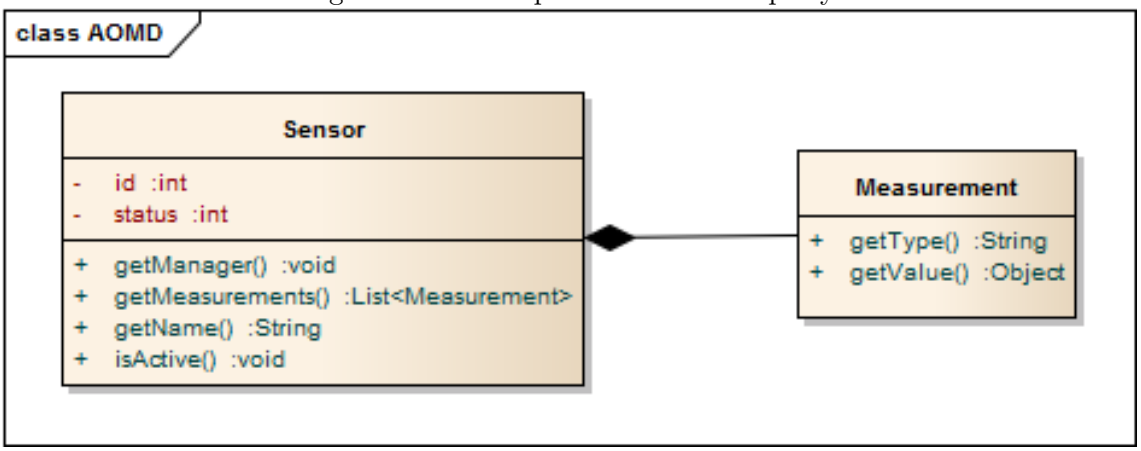


No entanto, este sistema pode ter vários usos em contextos que necessitam de diferentes tipos de informação do sensor. Para fazer com que o sistema possa ser utilizado para todas essas necessidades, poderia ser necessário criar um grande número de atributos na classe Sensor, sendo que apenas uma pequena quantidade destes atributos seria efetivamente utilizado em cada caso (apenas os atributos de interesse de cada um dos contextos seriam efetivamente utilizados).

O padrão Property resolve esse problema representando as propriedades de uma entidade por meio de uma classe e fazendo com que essa entidade possua uma coleção de instâncias dessa classe.

Aplicando o padrão Property neste exemplo, como apresentado na Figura 2.6, uma classe Measurement seria criada para representar uma propriedade de um sensor. Com a mudança, os atributos de Sensor podem ser substituídos por uma coleção de Measurements apenas com as medidas necessárias para cada sensor específico. Dessa forma, para adicionar uma nova propriedade, bastaria adicionar uma nova instância de Measurement na lista da classe Sensor.

Figura 2.6 - Exemplo do Padrão Property



2.4.3 Type Square

No estilo arquitetural AOM, os padrões Type Object e Property são geralmente usados em conjunto, resultando no padrão Type Square (YODER et al., 2001). Nesse padrão, o Type Object é usado duas vezes, uma vez para representar as entidades e tipos de entidade do sistema e novamente para representar as propriedades e tipos de propriedade.

Neste padrão, o tipo de entidade e o tipo de propriedade representam o modelo e, através da sua associação, é possível determinar que tipos de propriedades são aplicáveis a um determinado tipo de entidade.

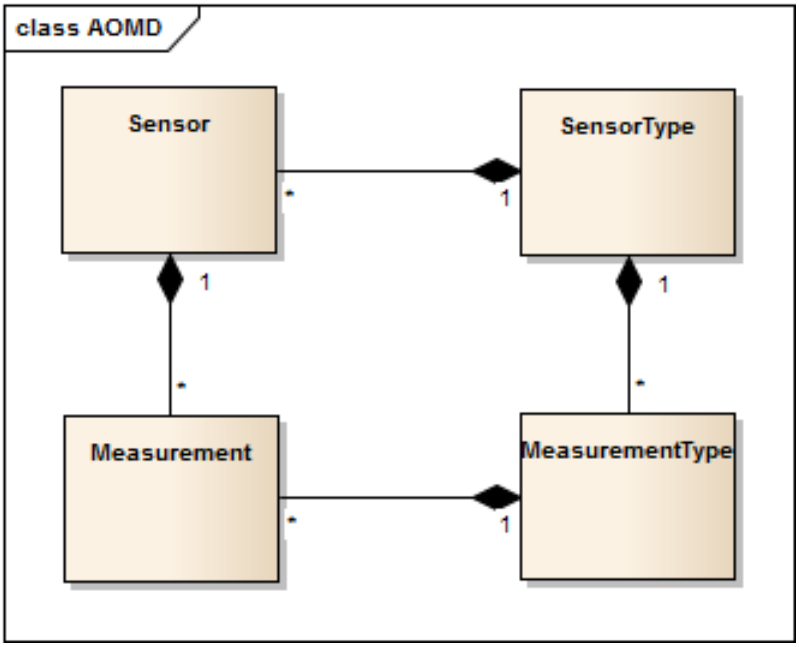
Cada instância da entidade refere-se a uma instância do tipo da entidade. Para cada tipo de propriedade no tipo de entidade, uma propriedade é criada para armazenar o valor do tipo de propriedade na entidade. O tipo de propriedade define as propriedades permitidas para um dado tipo de entidade e também pode definir algumas restrições, tais como os tipo de dados e valores permitidos. Com o Type Square, novos tipos de entidades com diferentes tipos de propriedades podem ser criados. Da mesma forma, os tipos de entidades existentes podem ser alterados em tempo de execução, uma vez que a modelagem é feita no nível de instância.

Por exemplo, para representar um Sensor do tipo magnetômetro, seria necessário criar um tipo de entidade e seriam adicionados os tipos de propriedades para as propriedades intensidade, direção e sentido. A partir do tipo de entidade é criada a instância da entidade Sensor que contém as três propriedades adicionadas nos tipos

de propriedades.

A Figura 2.7 apresenta um diagrama como exemplo do padrão Type Square.

Figura 2.7 - Exemplo do Padrão Type Square



2.4.4 Accountability

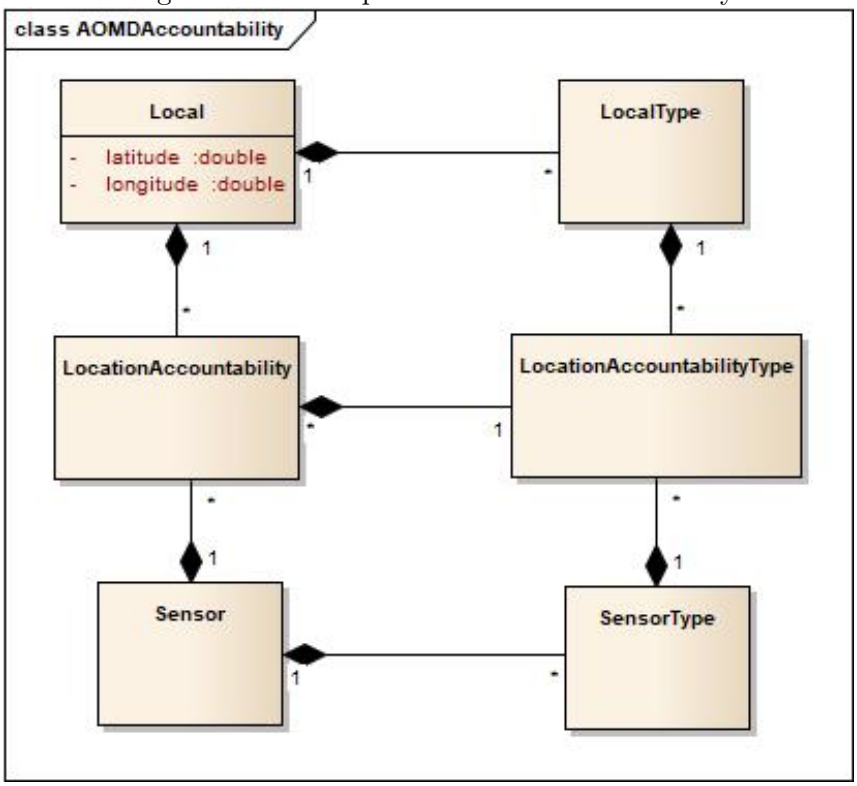
Em uma entidade, existem dois tipos de propriedades: aqueles que se referem a tipos primitivos (atributos) e as que se referem a relacionamentos entre entidades (associações). No modelo AOM, existem diferentes maneiras para diferenciar atributos de associações, como: (a) usar o padrão Property duas vezes, uma vez para atributos e outra para associações; (b) criar duas subclasses para a classe Property, uma para o atributo e outra para a associação; (c) verificar o tipo do valor do objeto Property, sendo que se for uma Entity representa uma associação e se for tipo primitivo corresponde a um atributo; (d) utilizar o padrão Accountability para representar a associação.

O padrão Accountability (YODER et al., 2001) permite que o relacionamento entre entidades possa ser representado por um objeto (em geral uma instância de uma classe Accountability). Cada objeto Accountability está associado a um objeto AccountabilityType, que representa o tipo do relacionamento. Como as associações entre entidades são representadas como instâncias, os tipos de relacionamentos en-

tre as entidades podem ser criados e modificados em tempo de execução, assim como nos outros padrões da arquitetura AOM.

A Figura 2.8 apresenta um diagrama com exemplo do padrão Accountability. No exemplo, a entidade Sensor tem um relacionamento com a entidade Local (por exemplo uma estação de captação de dados).

Figura 2.8 - Exemplo do Padrão Accountability



2.4.5 Rule Object

Rule Object é um padrão arquitetural utilizado para criar abstrações de regras de negócios simples ou complexas.

De acordo com (YODER; JOHNSON, 2002), as regras de negócios para sistemas orientados a objetos podem ser representadas de várias maneiras. Nas classes de entidades, essas regras podem definir restrições básicas, como a cardinalidade das relações ou se um preenchimento de um atributo é obrigatório. No entanto, algumas regras não podem ser definidas desta forma, pois são funcionais ou processuais devido a sua natureza.

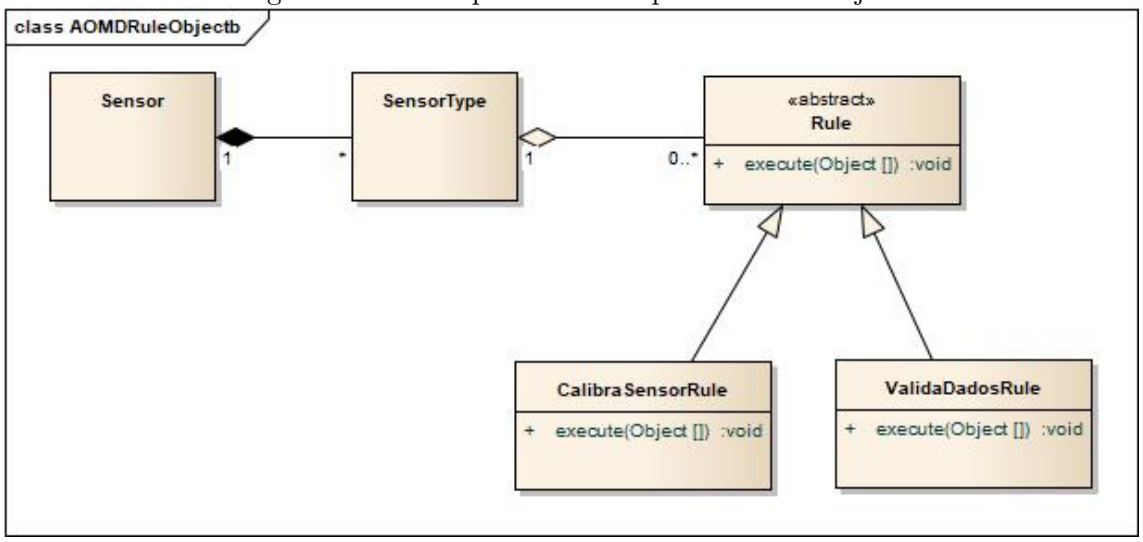
Essas regras de negócios tornam-se mais complexas por sua natureza e os modelos de objetos adaptativos usam o padrão Strategy (GAMMA et al., 1995) e o padrão Rule Object para lidar com eles.

O padrão Rule Object define uma interface padrão para uma família de algoritmos para que os clientes possam trabalhar com qualquer um deles. Se o comportamento de um objeto é definido por uma ou mais instâncias de uma classe do padrão Rule Object, então ele pode ser facilmente alterado trocando a sua instância.

No estilo arquitetural AOM, as classes que utilizam o padrão Rule Object são frequentemente associadas aos tipos de entidades, onde implementam as operações nos métodos de uma classe.

A Figura 2.9 apresenta um exemplo de Rule Object com a entidade Sensor e o tipo de entidade SensorType com as regras CalibraSensorRule e ValidaDadosRule.

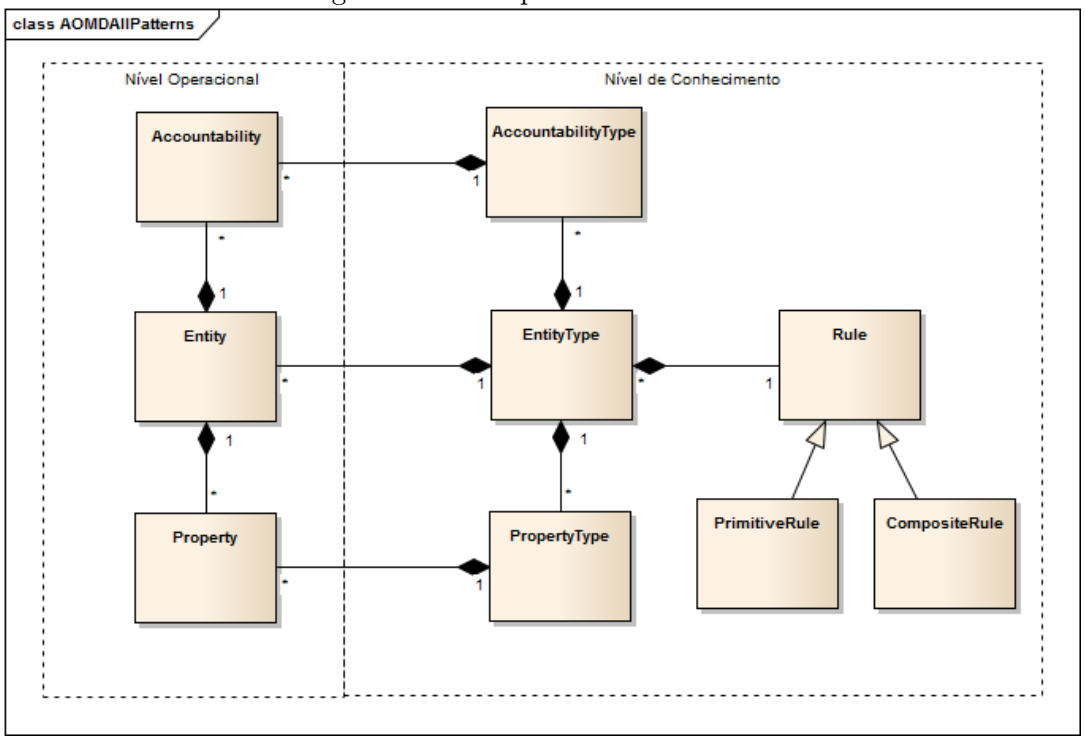
Figura 2.9 - Exemplo de uso do padrão Rule Object



2.4.6 Arquitetura core AOM

O núcleo da arquitetura AOM, apresentado na Figura 2.10 é composta dos principais padrões apresentados, Type Object, Property, Type Square, Accountability e Rule Object.

Figura 2.10 - Arquitetura Core AOM



Fonte: Yoder et al. (2001)

O nível operacional é utilizado para representar as instâncias da aplicação, que contém a informação que é de interesse da aplicação e o nível de conhecimento representa os metadados da aplicação, que descreve as entidades da aplicação.

A Listagem 2.9 apresenta um exemplo de aplicação dos padrões apresentados. Nele é criado um tipo de entidade com seus tipos de propriedades e um RuleObject. A partir do tipo de entidade, é criada a entidade Sensor (Running Example), são atribuídos valores para entidade e é executado o RuleObject .

Listagem 2.9 - Exemplo de criação de um tipo de entidade com tipos de propriedades e regra

```
1  // Cria o tipo de entidade
2  SensorType tipoEntidadeSensor = new SensorType();
3
4  // adiciona as propriedades no tipo de entidade
5  tipoEntidadeSensor.setTipoPropriedade("direcao", String.class);
6  tipoEntidadeSensor.setTipoPropriedade("intensidade", String.class);
7  tipoEntidadeSensor.setTipoPropriedade("sentido", String.class);
8
9  // adiciona o RuleObject no tipo de entidade
10 tipoEntidadeSensor.addRuleObject("nomeRegra", new RuleObjectSensor("sentido"));
11
```

```
12 // cria a entidade baseando-se no tipo de entidade
13 Sensor sensor = tipoEntidadeSensor.criaEntidade();
14
15 // seta valores na entidade
16 sensor.setPropriedade("direcao", "NORTE");
17 sensor.setPropriedade("intensidade", "50");
18 sensor.setPropriedade("sentido", "300");
19
20 // executa a regra
21 Object resultado = sensor.executaRegra("nomeRegra");
```

Na linha 2 é criado o tipo de entidade tipoEntidadeSensor, da liha 5 a 7 são criados três tipos de propriedades, são eles, direcao, intensidade e sentido. Na linha 10 é adicionada uma regra, de nome nomeRegra, que utiliza o RuleObject RuleObjectSensor. Na linha 13 é criado a entidade Sensor baseando-se no tipo de entidade. Da linha 16 a 18 são atribuídos valores para a entidade Sensor. Na linha 21 é executada a regra nomeRegra.

2.5 Frameworks para AOM

Um framework AOM é uma estrutura com classes e pontos de extensão que facilitam o construção de sistemas na arquitetura AOM. Implementar um sistema com a arquitetura AOM sem o apoio de um framework AOM pode ser uma tarefa trabalhosa.

Frameworks AOM têm como principal característica a disponibilização das classes para criação do modelo, assim como componentes que as manipulam. Em geral, eles apresentam um modelo de classes que implementa os padrões de projeto apresentados: Type Object, Property, Type Square, Accountability e Rule Object. Na literatura, exemplos de sistemas que usam o estilo arquitetural AOM (YODER et al., 2001), (YODER; JOHNSON, 2002) apresentam uma implementação para domínios específicos.

Os frameworks AOM, como Oghma (FERREIRA et al., 2009), ModelTalk com seu descendente Ink (HEN-TOV et al., 2010) e Esfinge Role Mapper (ESFINGE FRAMEWORK, 2018), tem o objetivo de serem independentes de domínio, para possibilitar sua utilização em qualquer tipo de sistema. Nas próximas seções são apresentados mais detalhes dos frameworks Oghma, Ink e Esfinge AOM Role Mapper.

2.5.1 Arquitetura do framework Oghma

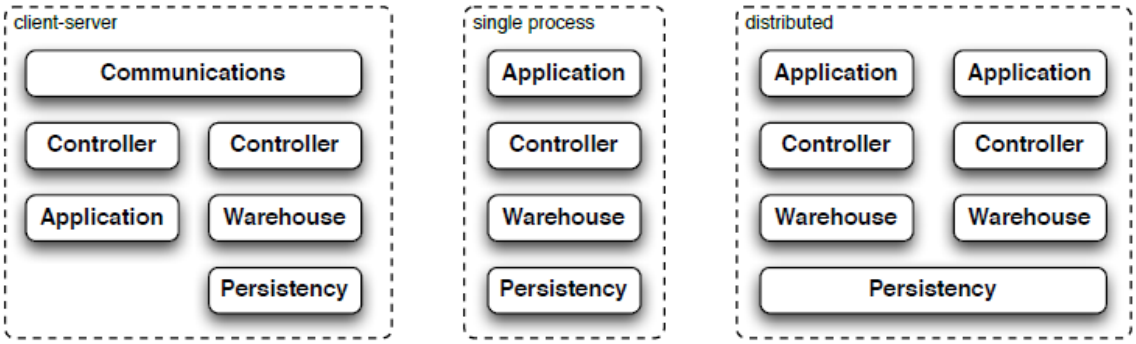
Oghma (FERREIRA et al., 2009) é um framework baseado na arquitetura AOM escrito em C#, que visa abordar várias problemas encontrados na construção de sistemas AOM, principalmente integridade, co-evolução de tempo de execução, persistência, geração de interface de usuário, comunicação e concorrência.

O framework Oghma, cujos principais componentes estão ilustrados na Figura 2.11, é um framework para desenvolver sistemas na arquitetura AOM, que equilibra a adaptabilidade e a reutilização. Ele dá suporte a criação de modelos semelhantes a MOF (Meta-Object Facility) e UML (Unified Modeling Language) e visa cobrir de todo o ciclo de criação e evolução do sistema.

Permite também a introdução de mudanças no sistema durante o tempo de execução, fornecendo assim um tipo de desenvolvimento direcionado ao usuário final. Além disso, o framework tem estrutura para oferecer a evolução do sistema provendo recursos como a auditoria sobre o sistema e viagem no tempo para um ponto arbitrário ao longo de sua evolução (isto é, para setar o sistema em um estado passado).

O framework Oghma inclui um conjunto de componentes intercambiáveis concebidos para ter alto grau de flexibilidade. Ele suporta vários tipos de engines de persistência, incluindo relacional, orientado a objetos, chave-valor e orientado a documentos, e estilos arquiteturais tais como single-process, cliente-servidor e distribuídos.

Figura 2.11 - Framework OGHMA

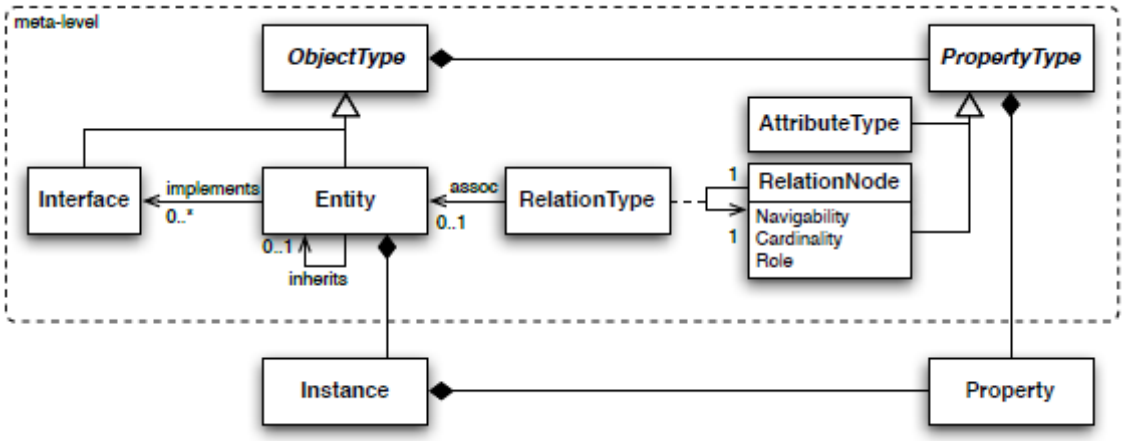


Fonte: Ferreira et al. (2009)

A Figura **??** apresenta o modelo AOM do framework Oghma. O modelo do framework tem muita flexibilidade e por este motivo é complexo e abstrato. Para uti-

lizar o framework em um sistema, mesmo não sendo necessária muita flexibilidade, é necessário instanciar e compreender diversas classes do modelo.

Figura 2.12 - Diagrama AOM do framework Oghma



Fonte: Ferreira et al. (2009)

2.5.2 Arquitetura do framework Ink

Ink (HEN-TOV et al., 2010) é um framework AOM que depende de um interpretador de uma linguagem específica de domínio (DSL) para adicionar adaptabilidade ao modelo.

Ink segue seu predecessor ModelTalk (HEN-TOV et al., 2009) em sua abordagem baseada em DSL. Um arquivo DSL é um componente reusável identificado por um namespace. Ele descreve um domínio específico provendo abstrações e restrições deste domínio.

No framework Ink, duas linguagens são usadas colaborativamente para implementar um sistema AOM:

- Uma linguagem dedicada para a descrição de estruturas orientada a objetos: A estrutura do sistema é expressa utilizando uma linguagem específica baseada em orientação a objetos (OO-like), chamada Ink. A linguagem Ink pode dinamicamente definir e alterar a estrutura de classe em tempo de execução, dando suporte a metaclasses explícitas.
- Uma linguagem dedicada para descrever o comportamento: O compor-

tamento do sistema é expresso com um linguagem orientada a objetos estaticamente tipada (Java). Utilizando duas linguagens em vez de uma, uma linguagem de estrutura dinâmica e uma linguagem estática para o comportamento, são combinados os benefícios de ambos os mundos. Isso promove um estilo de programação com uma estrita separação de estrutura e comportamento.

A Listagem 2.10, obtida de (ACHERKAN et al., 2011), apresenta um exemplo de definição da entidade Movie, com dois atributos, title e rating.

Listagem 2.10 - Exemplo de criação de uma entidade Movie no framework Ink

```
Metaclass id="Movie" class="ink.core:InkClass" super="ink.core:InkClass" {
    java_path ""
    java_mapping "State_Behavior_Interface"
    properties {
      property class="ink.core:StringAttribute" {
      name "title"
      mandatory true
    }
    property class="ink.core:StringAttribute" {
      name "rating"
      mandatory true
    }
  }
}
```

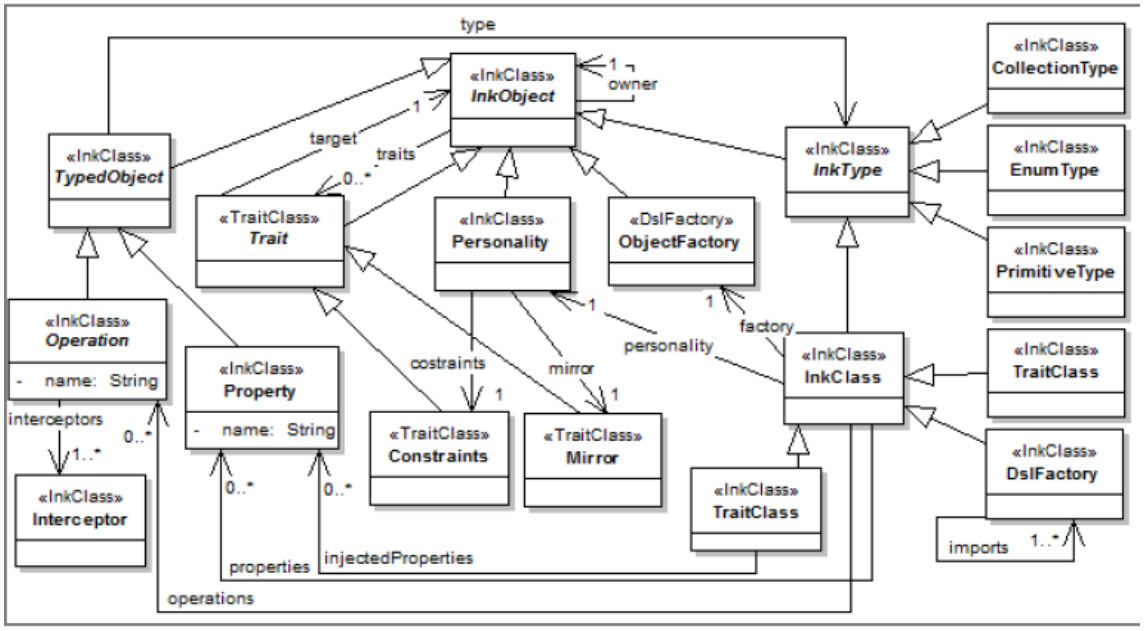
Para evitar um problema de dualidade de tipo entre as duas linguagens, a solução também inclui um Ambiente de Desenvolvimento Integrado (IDE) e uma Maquina Virtual. O IDE impõe regras de mapeamento entre as classes de estrutura e as classes de comportamento em tempo de design. Ele valida se o código de idioma imperativo é compatível com o modelo definido na linguagem de estrutura. O IDE também fornece suporte a ferramentas para a linguagem de estrutura com o mesmo nível de usabilidade que existe para o linguagens orientadas a objeto e estaticamente tipadas, como construtor incremental e funcionalidades de auto-completar o código.

A sua máquina virtual, a Ink VM (IVM), conecta em tempo de execução a estrutura e o comportamento. O IVM fornece uma API para acessar o modelo de estrutura. Ele também fornece um protocolo de meta-objeto (MOP) para manipular o modelo da aplicação em tempo de execução (por exemplo, definindo uma nova classe).

O metamodelo do Ink compreende um conjunto de construções para dar suporte a um sistema de quatro meta níveis, sendo eles tipos, operações, propriedades e restrições. A Figura **??** ilustra um diagrama de classes mostrando os elementos

básicos e as relações dentro do meta-metamodelo do Ink.

Figura 2.13 - Diagrama de classe do meta-modelo do framework Ink



Fonte: Acherkan et al. (2011)

Uma desvantagem apresentada pelo framework Ink é que o desenvolvedor precisa aprender uma nova linguagem de programação. O desenvolvedor também terá que utilizar o ambiente de desenvolvimento específico deste framework.

2.5.3 Esfinge AOM Role Mapper

O framework Esfinge AOM RoleMapper (ESFINGE FRAMEWORK, 2018) foi criado no contexto do projeto Esfinge, que é um projeto open source que compreende vários frameworks baseados em metadados para diferentes domínios. Para simplificação, o framework será referenciado apenas como AOM RoleMapper.

Esta seção descreve as funcionalidades existentes no AOM RoleMapper antes das alterações para adição do modelo comportamental, que é uma das contribuições deste trabalho de pesquisa.

O framework AOM RoleMapper trabalha com três diferentes tipos de modelos. Os modelos AOM específicos de domínio, um modelo independente de domínio e os modelos baseados em JavaBeans (SUN MICROSYSTEMS, 1997). A principal funcio-

nalidade do framework é mapear estruturas AOM de um domínio específico para uma estrutura AOM independente de domínio e da estrutura AOM independente de domínio para um modelo de classes estático baseado na especificação Javabeans.

Para isso, são usadas anotações e arquivos descritores, que fornecem informações para a criação de adaptadores dinamicamente. A idéia é tornar possível um modelo híbrido, que pode ser composto por entidades definidas por cada uma das abordagens. Por exemplo, uma entidade definida como JavaBean poderia ser adicionada como uma propriedade em uma entidade AOM. Isso aumenta a possibilidade de reúso dessas entidades.

Em aplicações reais que utilizam AOM como o estilo arquitetural, geralmente o modelo AOM é implementado apenas em entidades em que a flexibilidade é um requisito, utilizando apenas os padrões necessários. Outras entidades, muitas vezes, seguem um modelo estático adotada pela linguagem de programação alvo, contendo atributos fixos e métodos de acesso. Mesmo em classes que representam entidades AOM, por exemplo, podem haver propriedades estáticas que devem estar presentes em todas as entidades (MATSUMOTO; GUERRA, 2014).

O modelo AOM costuma ser construído para atender os requisitos de flexibilidade e adaptabilidade específicos de um domínio. Por esse motivo que frameworks AOM como o Oghma e o Ink, que proveem apenas a alternativa de modelo AOM mais completo fornecida por eles, podem não ser adequados para aplicações com modelos AOM mais simples ou que serão utilizados em apenas partes do sistema.

Outra questão não atendida pelos outros frameworks AOM apresentados é em relação ao reúso de componentes existentes. Apesar dos frameworks Oghma e Ink fornecerem uma biblioteca própria de componentes, eles não permitem o reúso de frameworks criados para classes da própria linguagem de programação. Por exemplo, um componente de persistência feito para classes C# não pode ser utilizado para persistir entidades AOM do Oghma. Isso limita os desenvolvedores ao reúso apenas dos componentes feitos para o modelo do próprio framework.

Para resolver estas duas questões, o framework AOM RoleMapper apresenta uma solução que atende os seguintes requisitos:

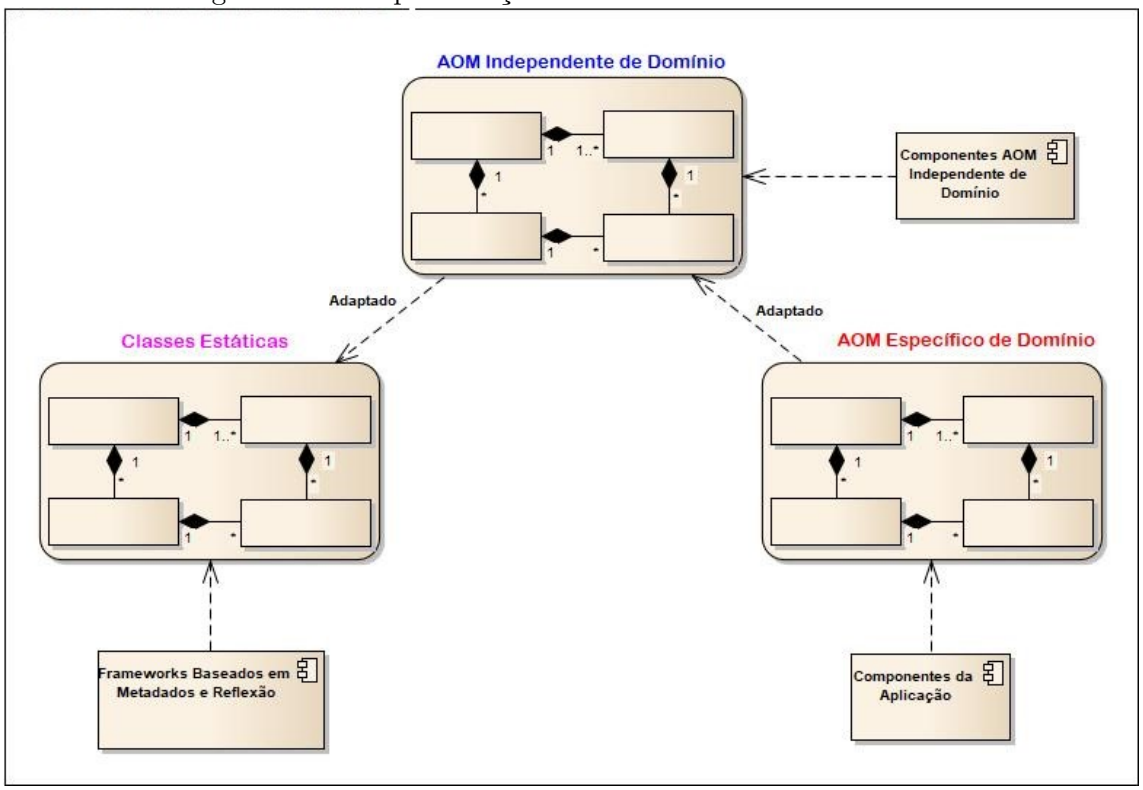
a) Permitir a utilização de componentes feitos para uma aplicação AOM independente de domínio em modelos AOM específicos de domínio;

b) Permitir a utilização de frameworks baseados em reflexão e em anotações

feitos para classes estáticas em entidades AOM independentes ou específicas de domínio.

Uma representação gráfica destes mapeamentos é apresentada na Figura 2.5.3.

Figura 2.14 - Representação do Modelo do Framework AOM



Neste modelo, o desenvolvedor pode definir uma entidade AOM específica de domínio usando suas próprias classes. Essas classes podem conter elementos que reproduzem papéis para os padrões AOM, mas também podem conter elementos específicos do domínio, como métodos e atributos usados pela aplicação. Esse modelo pode ser mapeado para o modelo independente de domínio do AOM RoleMapper utilizando anotações. Dessa forma, o framework provê classes adaptadoras que encapsulam as classes AOM dependentes de domínio e provêem uma API no formato do modelo AOM independente de domínio.

De forma similar, o AOM RoleMapper pode fazer o mapeamento de classes independentes de domínio para classes estáticas, por meio a da criação de adaptadores

de entidade.

A Tabela 2.1 apresenta um comparativo de características dos três frameworks AOM apresentados.

Tabela 2.1 - Características dos frameworks AOM

Característica	Oghma	Ink	Esfinge AOM Role Mapper
Entidade Independente de Domínio	Sim	Sim	Sim
Linguagem de Programação	C#	DSL e Ink	Java
Comportamento	Sim	Sim	Sim
Plugin para IDE	Não	Sim	Não
Mapeamento para classes estáticas	Não	Não	Sim
Mapeamento para entidade independente de domínio	Não	Não	Sim
Utilizado com frameworks tradicionais	Não	Não	Sim

O Capítulo 3 apresenta mais detalhes sobre o framework Esfinge AOM RoleMapper,que foi utilizado como base para esse trabalho de pesquisa.

3 AOM ROLEMAPPER

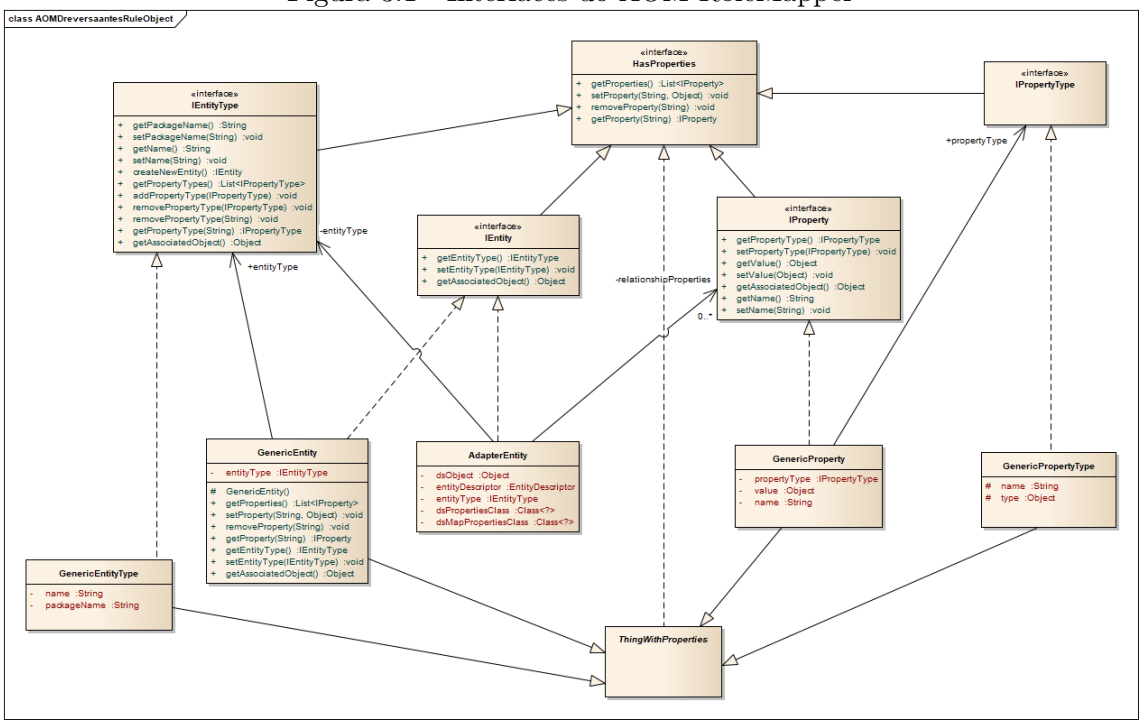
O AOM RoleMapper é utilizado como base para esse trabalho e foi escolhido por ter implementado em suas funcionalidades, o mapeamento do modelo AOM de domínio específico para o modelo AOM independente de domínio e para o modelo de classes estáticas. O objetivo desse capítulo é detalhar as funcionalidades do AOM RoleMapper e sua arquitetura.

3.1 Modelo AOM Independente de Domínio

O modelo independente de domínio segue, de forma genérica, os padrões básicos apresentados da arquitetura AOM. Com ele, pode-se definir tipos de entidades com suas propriedades e criar entidades a partir destes tipos.

A figura 3.1 apresenta as interfaces e suas implementações do AOM RoleMapper.

Figura 3.1 - Interfaces do AOM RoleMapper



A estrutura do framework é composta pelas interfaces IEntity, IEntityType, IProperty, IPropertyType. Estas Interfaces são implementadas por classes em dois pacotes diferentes, um que contém implementações relacionadas com a adaptação das

estruturas do núcleo AOM específicas de domínio para a estrutura do núcleo comum fornecido pelo framework (apresentadas na seção 3.2) e outro pacote que contém classes genéricas de AOM que podem ser usadas para criar um nova aplicação AOM usando o framework.

A estrutura fornece classes de fábrica que são capazes de decidir qual classe instanciar de acordo com parâmetros passados para os métodos de criação. Na Listagem 3.1 é apresentado um exemplo de criação de uma entidade AOM com a API independente de domínio do AOM RoleMapper.

Listagem 3.1 - Exemplo de criação de uma entidade AOM com API independente de domínio

```
1  @Test
2  public void testDomainIndependent() throws Exception {
3     // criando o tipo de entidade
4     IEntityType tipoSensor = new GenericEntityType("Magnetometro");
5
6     // criando os tipos de propriedades
7     GenericPropertyType direcaoPropertyType = new GenericPropertyType("direcao", Integer.class);
8     direcaoPropertyType.setProperty("notempty", true);
9     GenericPropertyType intensidadePropertyType = new GenericPropertyType("intensidade",
          Integer.class);
10    intensidadePropertyType.setProperty("notempty", true);
11    GenericPropertyType sentidoPropertyType = new GenericPropertyType("sentido", String.class);
12    sentidoPropertyType.setProperty("notempty", true);
13
14    // adicionando tipos de propriedades no tipo de entidade
15    tipoSensor.addPropertyType(direcaoPropertyType);
16    tipoSensor.addPropertyType(intensidadePropertyType);
17    tipoSensor.addPropertyType(sentidoPropertyType);
18
19    // Criando a entidade Magnetometro
20    IEntity magnetometro = tipoSensor.createNewEntity();
21
22    // setando valores nas propriedades
23    magnetometro.setProperty("direcao", 350);
24    magnetometro.setProperty("sentido", "N");
25    magnetometro.setProperty("intensidade", 50);
26 }
```

Na linha 4 é criada um tipo de entidade independente de domínio com o nome de Magnetometro. Nas linhas 7 a 12 são criados os tipos de propriedades de nome direcao, intensidade e sentido. Na linha 20 é criada a entidade do tipo Magnetometro a partir das definições da variável tipoSensor e nas linhas subsequentes (23, 24 e 25) são atribuídos valores para os atributos da entidade magnetometro.

O Modelo AOM independente de domínio possui o suporte a adição de metadados.

Pode-se adicionar informações adicionais sobre um tipo de entidade ou um tipo de propriedade. Isto é possível devido ao fato das classes que criam estes tipos implementarem a interface HasProperties. Esta implementação foi criada para a configuração de informações para o consumo de frameworks baseados em metadados que forem processar o modelo AOM. Por exemplo, é possível adicionar a unidade como um metadado de uma propriedade. A propriedade distancia pode ter o metadado "unidade"= "Km".

A Listagem 3.2 apresenta um exemplo de configuração de metadado no tipo de entidade sensorType.

Listagem 3.2 - Exemplo de configuração de metadados no modelo independente de domínio

```
1  @Test
2  public void testDomainIndependentMetadata() throws Exception {
3     // criando o tipo de entidade
4     IEntityType tipoSensor = new GenericEntityType("Magnetometro");
5
6     // criando os tipos de propriedades
7     GenericPropertyType direcaoPropertyType = new GenericPropertyType("direcao", Integer.class);
8     GenericPropertyType distanciaPropertyType = new GenericPropertyType("distancia", String.class);
9
10    // Adiciona metadados para validar maximo e minimo para a direcao
11    direcaoPropertyType.setProperty("direcao.min", 1);
12    direcaoPropertyType.setProperty("direcao.max", 360);
13
14    // Adiciona metadado para o tipo de propriedade distancia
15    distanciaPropertyType.setProperty("unidade", "KM");
16
17    // adicionando tipos de propriedades no tipo de entidade
18    tipoSensor.addPropertyType(direcaoPropertyType);
19    tipoSensor.addPropertyType(distanciaPropertyType);
20
21    // Criando a entidade Magnetometro
22    IEntity magnetometro = tipoSensor.createNewEntity();
23
24    // setando valores nas propriedades
25    magnetometro.setProperty("direcao", 350);
26    magnetometro.setProperty("distancia", "5KM");
27 }
```

Nas linhas 11 e 12 são adicionados metadados para o tipo de propriedade direcao. Na linha 15 é adicionado o metadado unidade no tipo de propriedade distancia.

3.2 Mapeamento para AOM de Domínio Específico

Mapeamento é a forma como o desenvolvedor, ao criar as suas próprias classes que implementam o padrão AOM, faz com que elas sejam reconhecidas pelo framework.

O framework AOM RoleMapper possui classes que implementam as interfaces independente de domínio que encapsulam as classes de um AOM específico de domínio. Para isso, as aplicações AOM específicas de domínio precisam ter os elementos em suas estruturas centrais marcados com metadados fornecidos pelo framework AOM Role Mapper. Para integrar aplicações AOM específicas de domínio com estruturas independente de domínio, o framework AOM RoleMapper usa anotações de código nas classes da aplicação.

A seguir é apresentada a descrição das anotações com o seu significado, juntamente com o tipo de elemento que pode ser anotado:

a) @EntityType: (Classe) Identifica classes que desempenham a função de Tipo de Entidade na arquitetura AOM;

b) @Entity: (Classe) Identifica classes que desempenham a função Entidade na arquitetura AOM;

c) @PropertyType: (Classe) Identifica classes que desempenham a função de Tipo de Propriedade na arquitetura AOM;

d) @EntityProperties: (Classe) Identifica o atributo que se refere a Propriedades de uma Entidade;

e) @FixedEntityProperty: (Atributo em classes de Tipo de Entidade) (Opcional) Identifica atributos correspondentes a propriedades fixas em uma classe Entidade;

f) @Name: (Atributo em classes de Tipo de Entidade ou Tipo de Propriedade) Identifica o atributo que contém o nome de um Tipo de Entidade ou de um Tipo de Propriedade;

g) @PropertyTypeType: (Atributo em classes de Tipo de Propriedade) Identifica o atributo que contém o tipo de um tipo de propriedade;

h) @PropertyValue: (Atributo em classes de Tipo de Propriedade) Identifica o atributo que contém o valor de uma propriedade;

i) @CreateEntityMethod: (Método em classes de Tipo de Entidade) Identifica o método de uma classe da Entidade que efetua a criação de uma Entidade com este tipo. Se nenhum método for anotado, o método createNewEntity da interface IEntityType lançará uma exceção quando invocado a partir do objeto.

j) @Metadata: (Atributo) Identifica uma lista de metadados de um tipo de entidade ou um tipo de propriedade.

k) @FixedMetadata: (Atributo) Identifica o metadado fixo de um tipo de entidade ou um tipo de propriedade.

l) @MetadataMap: (Atributo) Identifica o mapa de metadados de um tipo de entidade ou um tipo de propriedade.

A Listagem 3.3 apresenta um exemplo de mapeamento de uma classe específica de domínio que representa um tipo de entidade, de nome SensorType.

Na linha 1 é feito o mapeamento do tipo de entidade com a anotação @EntityType. Na linha 3 é mapeado o atributo propertyTypes com a anotação de tipo de propriedade @PropertyType. Na linha 6, o atributo metadata é mapeado com a anotação de lista de metadados do tipo de entidade. Na linha 9, o método createSensor é mapeado com a anotação @CreateEntityMethod que cria a entidade.

Listagem 3.3 - Exemplo de classe específica de domínio SensorType mapeada com anotações

```
1  @EntityType
2  public class SensorType {
3     @PropertyType
4     private Set<SensorPropertyType> propertyTypes = new HashSet<SensorPropertyType>();
5
6     @Metadata
7     private List<MetadatasAccountType> metadata = new ArrayList<MetadatasAccountType>();
8
9     @CreateEntityMethod
10    public ISensor createSensor() {
11       ISensor sensor = null;
12       sensor = (ISensor) new Sensor();
13       sensor.setSensorType(this);
14       return sensor;
15    }
16 }
```

A Listagem 3.4 apresenta um exemplo de criação da entidade, de nome Sensor, como uma classe específica de domínio mapeada com anotações.

Na linha 1 do exemplo é feito o mapeamento da entidade Sensor. Na linha 3 é realizado o mapeamento do tipo de entidade SensorType, da Listagem 3.3. Na linha 6 é feito o mapeamento de uma propriedade fixa de nome owner. Na linha 9 é feito o mapeamento da propriedade de nome properties.

Listagem 3.4 - Exemplo de uma classe específica de domínio Sensor mapeada com anotações

```
1  @Entity
2  public class Sensor {
3     @EntityType
4     private SensorType SensorType;
5
6     @FixedEntityProperty
7     private String owner;
8
9     @EntityProperty
10    private List<SensorProperty> properties = new ArrayList<SensorProperty>();
11 }
```

A Listagem 3.5 apresenta a criação do adaptador para a entidade AOM específica de domínio a partir da classe SensorType.

Na linha 3 é criada a instancia de SensorType. Da linha 4 até a linha 7 é criado um tipo de propriedade de nome varredura. Esta propriedade varredura também será adaptada quando é chamado o método getAdapter, para o tipo de entidade, na linha 15. Na linha 10 é criada a entidade Sensor. Na linha 11 é atribuído o tipo de entidade sensorType para a entidade sensor. Na linha 15 é criado o adaptador do tipo de entidade sensorType. Na linha 18 é invocado o método getAdapter para retornar o adaptador da entidade Sensor. Na linha 19 é atribuído o valor 100.0 para a variável varredura, a entidade original, sensor, que está recebendo o valor desta propriedade. Na linha 20 é realizado o teste comparando o valor da variável varredura com o valor atribuído a entidade.

Listagem 3.5 - Criação de um adaptador para entidade AOM específica de domínio

```
1  public void testeAdapterSensorType() throws Exception {
2      // cria o tipo de entidade
3     SensorType sensorType = new SensorType();
4     SensorPropertyType sensorPropertyType = new SensorPropertyType();
5     sensorPropertyType.setName("varredura");
6     sensorPropertyType.setPropertyType(double.class);
7     sensorType.addPropertyTypes(sensorPropertyType);
8
9     // cria a entidade
10    Sensor sensor = new Sensor();
11    sensor.setSensorType(sensorType);
12    double valorVarredura = 100.0;
13
14    // cria o tipo de entidade adaptado
15    AdapterEntityType adaptedEntityType = AdapterEntityType.getAdapter(sensorType);
16
17    // cria a entidade adaptada
18    AdapterEntity entity = AdapterEntity.getAdapter(adaptedEntityType, sensor);
19    entity.setProperty("varredura", valorVarredura);
```

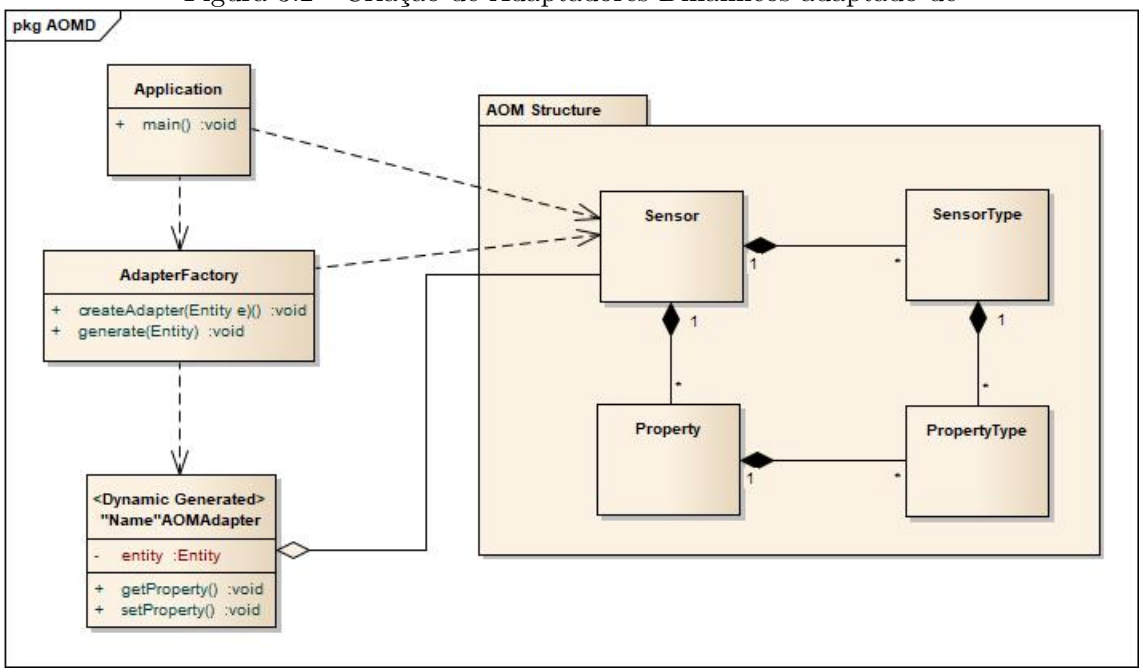
```
20      Assert.assertEquals(valorVarredura, entity.getProperty("varredura").getValue());
21  }
```

3.3 Mapeamento para Modelo de Classes Estáticas

Para Adaptar a API independente de domínio do AOM RoleMapper para uma API estática, foi adotada uma solução de gera em tempo de execução classes adaptadoras com estrutura que contenha métodos de acesso as entidades, ou seja, métodos get e set. O objetivo destes adaptadores é permitir que as entidades AOM possam ser acessadas por frameworks baseados em API estática. Estes adaptadores, quando são invocados, chamam métodos correspondentes na entidade AOM.

A Figura 3.2 apresenta o diagrama de classes com os principais participantes desta solução. A classe denominada Application representa uma aplicação que possui uma instância de uma entidade AOM que segue o padrão JavaBeans. Ele invoca o AdapterFactory passando a entidade como um parâmetro. O AdapterFactory acessa os metadados da entidade e suas respectivas propriedades e tipos de propriedades. Com base nestes metadados, ele gera dinamicamente o adaptador para esta entidade. O adaptador é retornado para a aplicação, que pode acessar seus métodos por reflexão como se fosse um JavaBean.

Figura 3.2 - Criação de Adaptadores Dinâmicos adaptado de



Fonte: Guerra et al. (2015)

A Listagem 3.6 apresenta um exemplo de um teste utilizando a geração de adaptadores para a entidade Sensor com os atributos name e type.

Na linha 4 do código é criado o tipo de entidade sensorType utilizando a API independente de domínio. Nas linhas 7 e 8 são adicionados dois tipos de propriedade, name e type ambas do tipo String. A partir do tipo de entidade é criada a entidade sensor, na linha 11 do teste. A seguir são adicionados valores para as propriedades name e type, nas linhas 14 e 15 respectivamente. Na linha 18 é gerado o adaptador dinâmico sensorAdapter. A partir do adaptador sensorAdapter, é feita a invocação dos métodos getName e getType, utilizando reflexão, nas linhas 21 e 22, e após obtidos os valores de retorno dos métodos é feito a verificação para confirmar que o valores são iguais ao valores da entidade sensor, nas linhas 25 e 26 respectivamente.

Listagem 3.6 - Exemplo com teste de geração de adaptadores dinâmicos utilizando a entidade Sensor

```
1  @Test
2  public void createAdapterSensorProperty() throws EsfingeAOMException {
3      // cria o tipo de entidade
4      IEntityType sensorType = new GenericEntityType("Sensor");
5
6      // cria os tipos de propriedades
7      sensorType.addPropertyType(new GenericPropertyType("name", String.class));
8      sensorType.addPropertyType(new GenericPropertyType("type", String.class));
9
10     // cria a entidade
11     IEntity sensor = sensorType.createNewEntity();
12
13     //atribui valores as propriedades
14     sensor.setProperty("name", "radiometerX");
15     sensor.setProperty("type", "radiometer");
16
17     // gera o adaptador
18     Object sensorAdapter = af.generate(sensor);
19
20     // invoca os mtodos do adaptador
21     String phone = (String) personAdapter.getClass().getMethod("getName").invoke(sensorAdapter);
22     String type = (String) personAdapter.getClass().getMethod("getType").invoke(sensorAdapter);
23
24     // testa os valores obtidos da invocacao dos metodos
25     assertEquals(phone, person.getProperty("name").getValue());
26     assertEquals(type, person.getProperty("type").getValue());
27 }
```

3.4 Transformação de Metadados em Anotações

No processo de geração do adaptador, os metadados da entidade são transformados em anotações nos objeto adaptado, utilizando as configurações do arquivo JSON.

Um exemplo do arquivo JSON é apresentado na Listagem 3.7.

Listagem 3.7 - Exemplo de arquivo JSON para mapear objeto adaptado

```
{
  "entity":[
    {"target":"class"},
   {"annotationPath":"javax.persistence.Entity"}
  ],
  "id":[
    {"target":"method"},
   {"annotationPath":"javax.persistence.Id"}
  ],
  "column":[
    {"target":"method"},
   {"annotationPath":"javax.persistence.Column"},
   {"parameter_1": "name"},
   {"parameter_2": "nullable"}
   ],
  "oneToOne":[
    {"target":"method"},
   {"annotationPath":"javax.persistence.OneToOne"}
   ]
}
```

Um exemplo da geração do adaptador com mapeamento de anotações é apresentado na Listagem 3.8.

Na linha 2 é criado o tipo de entidade tipoSensor. Da linha 5 a 8 são criados dois tipos de propriedades, de nome id e type. Da linha 11 a 15 é feito a configuração para o mapeamento da entidade sensor. Da linha 18 a 22 é realizada a configuração do mapeamento da propriedade type. Na linha 24 é criada a entidade sensor. Na linha 25 é criada a fábrica do adaptador, utilizando o arquivo JSON apresentado na Listagem 3.7. Na linha 28 é criado o adaptador com o objeto sensorAdaptado mapeando com as anotações. Na linha 29 são impressos os métodos e anotações do objeto adaptado.

Listagem 3.8 - Transformando metadados em anotações

```
 //cria o tipo de entidade
IEntityType tipoSensor = new GenericEntityType("Sensor");

// cria os tipos de propriedades
IPropertyType idPropertyType = new GenericPropertyType("id", Long.class);
tipoSensor.addPropertyType(idPropertyType);
IPropertyType typePropertyType = new GenericPropertyType("type", String.class);
tipoSensor.addPropertyType(typePropertyType);

//parametros das annotations de sensor
Map<String, Object> parametersPerson = new HashMap<String, Object>();
parametersPerson.put("name", "tb_sensor");
```

```
13 //annotations da classe sensor
14 tipoSensor.setProperty("table", parametersPerson);
15 tipoSensor.setProperty("entity", true);
16
17 //parametros das annotations de type
18 Map<String, Object> typeParamters = new HashMap<String, Object>();
19 typeParamters.put("name", "tipo");
20 typeParamters.put("nullable", false);
21 //annotations da propriedade type
22 typePropertyType.setProperty("column", typeParamters);
23
24 IEntity sensor = tipoSensor.createNewEntity();
25 AdapterFactory af = AdapterFactory.getInstance("JsonMap.json");
26
27 // gera o objeto adaptado
28 Object sensorAdaptado = af.generate(sensor);
29 ObjectPrinter.printClass(sensorAdaptado);
```

A Listagem 3.9 apresenta um método para ler e imprimir as anotações do objeto adaptado.

Listagem 3.9 - leitura das anotações do objeto adaptado

```
1  public static void printClass(Object obj) {
2     System.out.println("classe name: " + obj.getClass().getName());
3
4     for (Annotation annotation : obj.getClass().getAnnotations()) {
5        System.out.println(annotation);
6     }
7
8     for (Method m : obj.getClass().getMethods()) {
9        for (Annotation annotation : m.getAnnotations()) {
10          System.out.println(annotation);
11       }
12       System.out.println(m);
13    }
14 }
```

A Listagem 3.10 apresenta a impressão das anotações do objeto adaptado.

Listagem 3.10 - log de leitura das anotações do objeto adaptado

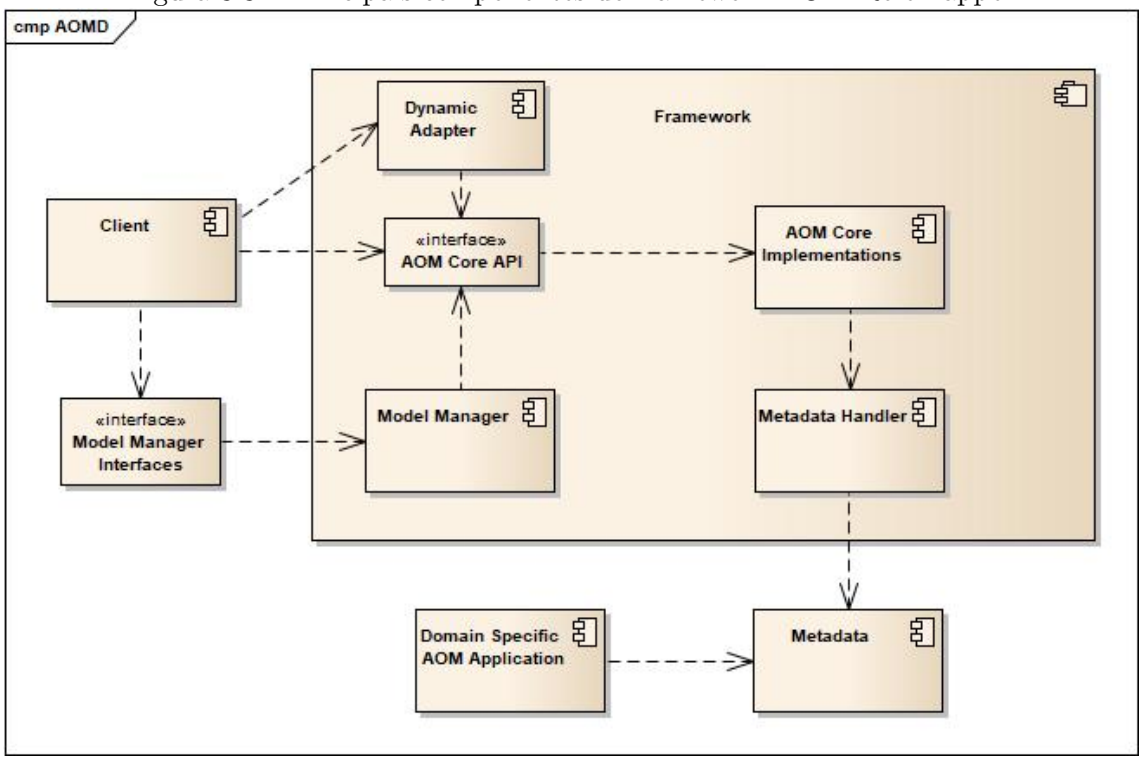
```
1 classe name: sensorAOMBeanAdapter
2 @javax.persistence.Table(name=tb_sensor)
3 @javax.persistence.Entity(name=)
4 @javax.persistence.Id()
5 public java.lang.Long sensorAOMBeanAdapter.getId()
6 @javax.persistence.Column(nullable=false, unique=false, precision=0, name=tipo, length=255,
      scale=0, updatable=true, columnDefinition=, table=, insertable=true)
7 public java.lang.String sensorAOMBeanAdapter.getType()
8 public void sensorAOMBeanAdapter.setType(java.lang.String)
9 public void sensorAOMBeanAdapter.setId(java.lang.Long)
```

Na linha o nome do objeto adaptado sensorAOMBeanAdapter. Nas linhas 2 e 3 têm as anotações do objeto adaptado (@javax.persistence.Table e @javax.persistence.Entity. Na linha 4 tem a anotação do método getId (@javax.persistence.Id()). Na linha 6 é impressa a anotação do método getType (@javax.persistence.Column).

3.5 Estrutura Interna

A Figura 3.3 descreve a relação entre os principais componentes do framework AOM RoleMapper. Esses componentes internos do framework estão descritos em seguida:

Figura 3.3 - Principais componentes do framework AOM RoleMapper



Fonte: Matsumoto e Guerra (2014)

a) O Metadata Handler é responsável por recuperar metadados das classes da aplicação. Ele implementa padrões de leitura de metadados (GUERRA et al., 2009) para desacoplar operações de manipulação de metadados do resto do framework.

b) A API AOM Core inclui um conjunto de interfaces que representam a estrutura comum do núcleo AOM fornecido pelo framework, descritos na

seção 3.1.

c) O componente AOM Core Implementations contém implementações de Interfaces definidas pelo componente API AOM Core.

Existem dois tipos de implementações neste componente, uma implementação básica e geral da estrutura do núcleo do AOM e uma implementação que adapta estruturas do núcleo AOM específicas de domínio usando o componente Metadata Handler, conforme apresentados nas seções 3.1 e 3.2.

d) O Model Manager ou Componente Gestor do Modelo é responsável por instanciar o modelo e gerenciar as instâncias da API AOM Core criadas pelo framework.

e) Dynamic Adapter : contém as classes para gerar o adaptador de classes estáticas a partir das entidades independente de domínio. Também faz o mapeamento das anotações baseando-se em arquivos JSON.

3.5.1 Componente Metadata Handler

O componente de metadados implementa alguns dos padrões como Metadata Reader, Metadata Container e Metadata Repository (GUERRA et al., 2009) e pode ser dividido nas seguintes partes:

(a) Descritores: implementa o padrão Metadata Container. Cada papel em uma arquitetura AOM é representado por um descritor que contém referências para os métodos get/ set / add / remove para cada campo revelante dos objetos.

(b) Metadata Readers: implementa o padrão Metadata Reader Strategy. Atualmente, o framework suporta apenas anotações para determinar os papéis dos elementos AOM em aplicações específicas de domínio, mas uma vez que o padrão metadata reader strategy foi implementado , ele suporta extensões relacionadas com o apoio de outros tipos de metadados .

(c) Metadata Repository: implementa o padrão Metadata Repository, fornecendo um cache de memória dos metadados já recuperados.

(d) Anotações : contém as anotações Java que permitem a identificação dos papéis dos elementos AOM nas aplicações AOM de domínio específico para a estrutura independente de domínio.

3.5.2 Componente Gestor do Modelo

A responsabilidade do componente Gestor do Modelo é orquestrar as instâncias criadas pelo framework AOM RoleMapper. A classe principal deste componente é o ModelManager. Todas as operações que envolvem a manipulação do modelo, incluindo o modelo de persistência, carregamento e de consulta, deve ser feito por meio desta classe . Para acessar a base de dados, o ModelManager faz uso da interface IModelRetriever, que pode ser implementado por frameworks de persistência (MATSUMOTO; GUERRA, 2014).

Uma das principais responsabilidades da classe ModelManager é garantir que um elemento lógico não seja instanciado duas vezes no framework. A fim de controlar isso, o ModelManager contém dois Mapas de objetos, um para guardar as entidades carregados por seus IDs e outra para armazenar os tipos de entidade carregados por seus IDs.

O framewok tem componentes de persistência em MongoDB, Neo4J e CouchDB que persiste dados e metadados das entidades e dos tipos de entidades. Ele também pode carregar modelos de arquivos em formato XML e JSON para criação dos objetos.

3.6 Limitações

Uma limitação da solução atual do framework AOM RoleMapper é que, apesar do fato de que novos adaptadores podem ser criados quando há alterações de um tipo de entidade AOM, isso não vai mudar os adaptadores AOM existentes que já foram criados.

Outra limitação é não permitir herança nas entidades AOM. Esta funcionalidade é importante para que seja possível herdar de classes abstratas e mesmo de classes comuns. Em frameworks tradicionais, é uma prática comum a necessidade de herdar de uma classe específica do framework para compor uma funcionalidade.

4 EXPERIMENTO COM AOM ROLEMAPPER

Esse capítulo descreve um experimento realizado para avaliar o modelo de framework implementado pelo AOM Role Mapper em relação a utilização de componentes e estruturas criadas para modelos de classes estáticas. O experimento foi realizado antes de introduzir o comportamento dinâmico no framework AOM RoleMapper.

No contexto dessa dissertação, essa avaliação foi realizada para avaliar como é a experiência dos desenvolvedores utilizando um framework feito para classes estáticas para entidades AOM. Foi baseado nos resultados desse experimento que chegou-se a conclusão de que foi percebido valor agregado pelos desenvolvedores e pode-se investir em esforços para continuar com a abordagem no mapeamento de Rule Objects para métodos.

4.1 Visão Geral

Os participantes do experimento foram divididos em dois grupos, sendo o primeiro grupo realizando a primeira atividade de criação e validação de uma entidade (Pessoa) sem a utilização de framework e a segunda atividade, também criando e executando a validação de uma entidade (Cobranca) com a utilização do AOM RoleMapper e do Hibernate Validator (HIBERNATE VALIDATOR, 2016) , mapeando a entidade AOM para uma classe estática no formato JavaBeans com anotações e reusando a funcionalidade do Hibernate Validator. O segundo grupo fez a atividade de validação de outra entidade (Cobranca) sem utilizar framework e depois a criação e validação da entidade (Pessoa) utilizando o Framework AOM RoleMapper e o Hibernate Validator.

Esta técnica de experimentação, que divide os participantes em dois grupos é chamada de Crossover design (VEGAS et al., 2016), é amplamente utilizada em experimentos de exercícios práticos em engenharia de software, tem como principais benefícios, não ter a necessidade de ter muitos participantes no experimento e a redução da variabilidade dos resultados devido a diferenças entre os indivíduos.

4.2 O Experimento

Este experimento foi realizado com base nas diretrizes fornecidas por (WOHLIN et al., 2012). A partir deste experimento, forma investigadas as seguintes questões de pesquisa:

- Q1: Como o tempo de desenvolvimento difere comparando as duas abor-

dagens?

- Q2: Os desenvolvedores conseguem entender a abordagem proposta com base na documentação fornecida e implementar com êxito a funcionalidade necessária?

- Q3: Quais foram as principais dificuldades encontradas pelos desenvolvedores nas duas abordagens?

- Q4: Quais são os benefícios e desvantagens da abordagem proposta para utilizar a funcionalidade da estrutura do ponto de vista dos desenvolvedores?

- Q5: Qual é o nível de aceitação dos desenvolvedores com relação à abordagem proposta para utilizar a funcionalidade de uma estrutura que usa o adaptador gerado para a entidade AOM?

A questão de pesquisa Q1 é respondida com a ajuda do experimento, analisando a quantidade de tempo gasto na conclusão de tarefas entre as duas abordagens. As questões Q2, Q3, Q4 e Q5 são respondidos por meio da análise das respostas do questionário aplicado aos participantes do experimento.

4.2.1 Meta

Seguindo Meta-Pergunta-Métrica-GQM (BASILI; ROMBACH, 1988), o objetivo desta experiência é: analisar uma abordagem para utilizar a funcionalidade de uma estrutura que usa o adaptador gerado para a entidade AOM, com o propósito de avaliá-la em relação a quantidade de tempo gasto na conclusão de uma tarefa, a facilidade de uso percebida, utilidade e benefícios do ponto de vista de desenvolvedores de software, no contexto de projeto de software e curso de pós-graduação do Instituto Nacional de Pesquisas Espaciais.

4.2.2 Variáveis, Tratamentos e Objetos

Existem duas variáveis independentes: as duas abordagens de validação (tratamentos) e os objetos experimentais (Tarefa 1 e Tarefa 2). Existe uma variável dependente de objetivo: a quantidade de tempo gasto na conclusão de uma tarefa (variável de resposta). Além disso, existem duas variáveis dependentes subjetivas: a facilidade de uso e a utilidade percebidas pelos desenvolvedores, que são calculadas por perguntas fechadas e perguntas abertas para obter feedback dos participantes.

4.2.3 Hipóteses

As seguintes hipóteses são apresentadas para investigar a questão de pesquisa Q1:

- Hipótese nula: Não há diferença significativa no tempo gasto entre as duas abordagens para concluir as tarefas.
- Hipótese alternativa: Executar a funcionalidade de utilização das tarefas de uma estrutura que usa o adaptador gerado para a entidade AOM consome menos tempo do que executar a mesma tarefa usando um código imperativo.

4.2.4 Participantes

Foram selecionados 21 participantes que estavam entre os estudantes matriculados em cursos de pós-graduação na área de design de software do Instituto Nacional de Pesquisas Espaciais. Como requisito, para participar do experimento, o participante precisa ter experiência em programação usando a linguagem Java e ter conhecimento sobre reflexão e anotações de código. Apesar de haverem alunos de mestrado e doutorado participando, a maioria dos participantes naquele momento estava trabalhando na indústria e fazendo o curso como alunos especiais. Dividimos os 21 participantes em dois grupos, dos quais 12 foram designados para o Grupo 1 e os restantes 9 foram atribuídos ao Grupo 2.

A Tabela 4.1 mostra algumas estatísticas sobre a experiência dos participantes em termos de anos de experiência como programadores e os anos de experiência com a linguagem Java. A estatística inferencial mostra que os dois grupos estão equilibrados em termos de suas experiências, porque não foram encontradas diferenças significativas entre os grupos em relação a esses dois requisitos.

Tabela 4.1 - Experiência dos Participantes

Experiência	Grupo	Min	Média	Max	StDev (Desvio Padrão)
Programação	G1	1	9.83	20	6.18
	G2	2	7.89	25	7.24
Linguagem Java	G1	0	6.25	16	5.05
	G2	1	4.11	10	3.10

Quanto à experiência dos participantes em frameworks e anotações, a maioria deles, cerca de 80% (16), tinha alguma experiência sobre isso. No entanto, todos os

participantes receberam um treinamento sobre reflexão e anotações de código, a fim de homogeneizar seus conhecimentos. Este treinamento envolveu aulas teóricas e exercícios de código prático.

4.2.5 Abordagens do Experimento

Realizamos um experimento em que unidades de estudo foram designadas para grupos experimentais de forma não aleatória, portanto, é considerado um quase-experimento (KAMPENES et al., 2009). Como mostrado na Tabela 4.2, o experimento foi projetado de forma que cada grupo de assuntos usa os dois tratamentos. O primeiro usando código imperativo (Abordagem 1) e outro usando AOM RoleMapper e Hibernate Validator (Abordagem 2) em dois objetos. A entidade Pessoa (Tarefa 1) e Entidade de Cobrança (Tarefa 2).

Tabela 4.2 - Abordagem do Experimento

Grupo	No	Abordagem 1 Task 1	Abordagem 1 Task 2	Abordagem 2 Task 1	Abordagem 2 Task 2
G1	12	✓			✓
G2	9		✓	✓	

4.2.6 Procedimento

Para projetar as atividades para o experimento, consideramos os seguintes requisitos:
(a) a funcionalidade em que seria utilizada do framework existente deveria ser fácil de ser executada usando código diretamente imperativo;
(b) o framework usado deve ser baseado em metadados e bem conhecido pela comunidade de desenvolvimento.

Com base nesses requisitos, foi decidido trabalhar a funcionalidade de implementação de restrições de validação para valores de propriedades para uma entidade. O framework escolhido para ser utilizado foi o Hibernate Validator, que usa anotações nas propriedades da classe para definir tais restrições. Como exemplo, a anotação @Max pode ser usada para definir uma restrição sobre o valor máximo de uma propriedade numérica.

As tarefas para os participantes foram projetadas com o seguinte formato: foi definido um tipo de entidade com sete propriedades e uma restrição de validação para

cada uma delas. Para cada entidade, foi criado previamente um conjunto de testes automatizados que verificam as restrições e as interfaces necessárias para a API da solução. Duas tarefas diferentes com formato semelhante, mas com entidades e restrições diferentes, foram especificadas.

Em resumo, os participantes do experimento desenvolveram o código-fonte para verificar as restrições em seus atributos para os dois tipos de entidades AOM. Para uma entidade, eles desenvolveram a validação acessando os valores da propriedade e verificando as restrições com código imperativo (Abordagem 1). Para a outra entidade, eles configuraram os metadados no tipo de entidade AOM, geraram o adaptador usando a funcionalidade do AOM RoleMapper e invocaram a funcionalidade do Hibernate Validator no adaptador (Abordagem 2).

Após a conclusão das tarefas, os participantes tiveram que preencher um questionário para responder sobre sua experiência usando ambas as abordagens. Na primeira parte do questionário, os participantes tiveram que responder aos seguintes tópicos referentes à execução das tarefas nas duas abordagens:

- o tempo gasto para completar cada tarefa;
- as necessidades para interromper as atividades e por quê;
- as dificuldades para completar as tarefas;
- as necessidades de informações adicionais e que tipo de informação.

Na última parte do questionário, os participantes tiveram que destacar os benefícios e desvantagens de cada abordagem. Além disso, eles precisavam informar qual das duas abordagens escolheriam para implementar um dado recurso de validação, dando razões para sua escolha. Havia também um campo neste formulário em que os participantes poderiam adicionar outros comentários.

4.2.7 Tempo de Desenvolvimento (Q1)

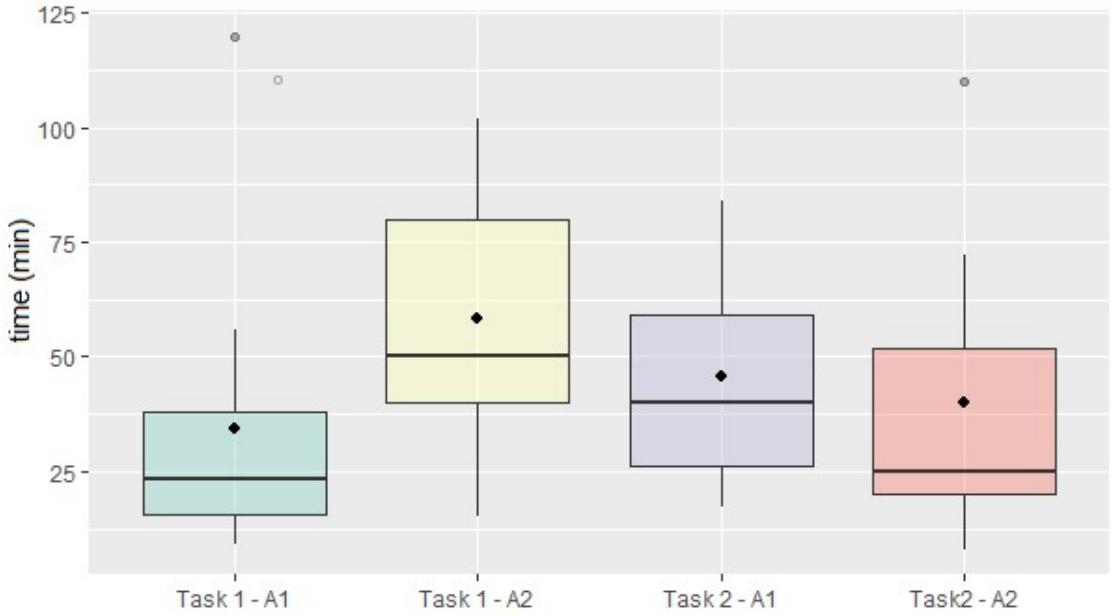
Os resultados do tempo gasto (em minutos) em cada tarefa por grupo são relatados na Tabela 4.3. Apresentamos os valores médios (Média), desvio padrão (StDev), valor mínimo (Min), valor mediano (Mediana) e valor máximo (Max). Examinando o tempo gasto por tarefa, notamos que a Tarefa 1, em média, requer mais tempo quando realizada com a abordagem (A2) do que com a abordagem usando código imperativo (A1). No entanto, a Tarefa 2 foi realizada em menor tempo com a abordagem proposta (A2).

Tabela 4.3 - Estatística Descritiva

Grupo	Task	Abordagem	Média	StDev	Min	Mediana	Max	NP
G1	T1	A1	34.25	30.23	9.00	23.50	120.00	12
G2	T1	A2	58.44	27.60	15.00	50.00	102.00	9
G1	T2	A2	39.83	29.76	8.00	25.00	110.00	12
G2	T2	A1	45.67	25.32	17.00	40.00	84.00	9

Dos boxplots apresentados na Figura 4.1, podemos comparar os valores do tempo gasto graficamente. Os gráficos mostram dois valores que fogem da normalidade (outlier), que optamos por não excluir, pois é um evento que pode ocorrer novamente se sujeitos com pouca experiência forem recrutados.

Figura 4.1 - Tempo de Desenvolvimento por tarefa e abordagem



Existem algumas diferenças entre duas abordagens, como podemos ver nos valores da tabela e boxplots. No entanto, não podemos afirmar se tais diferenças são significativas. Assim, realizamos uma análise estatística mais profunda para testar estas hipóteses. Primeiramente, usamos o teste de Shapiro-Wilk para verificar se os dados estavam ou não distribuídos normalmente. A partir dos resultados do teste de normalidade, comparamos duas amostras usando o Teste t pareado, e duas outras usando o teste de Wilcoxon. Os resultados da análise são apresentados na tabela 4.4.

Tabela 4.4 - Testando Hipóteses

Grupo	Task	Abordagem	Shappiro-Wilk	Teste t	Wilcoxon
G1	T1	A1	.001	N/A	.556
	T2	A2	.038		
G2	T2	A1	.299	.215	N/A
	T1	A2	.777		

Como mostrado na Tabela 4.4, o resultado do teste t (p-valor = 0,215) e o resultado do teste de Wilcoxon (p-valor = 0,556) não revelaram diferenças estatisticamente significantes ao nível alfa de 0,05. Isso indicou que a hipótese nula não poderia ser rejeitada. Portanto, o tempo necessário para concluir as tarefas nas duas abordagens é semelhante.

As pequenas diferenças entre as duas abordagens podem estar relacionadas a outros fatores, como o tipo de tarefa e/ou experiência dos participantes.

4.2.8 Documentação (Q2)

Para concluir a tarefa usando um código imperativo (A1), cerca de 48% dos participantes (10 de 21) afirmaram que precisaram consultar informações sobre métodos de validação e/ou algum conteúdo em expressões regulares. Em relação à abordagem proposta (A2), apenas um terço dos participantes pesquisou informações adicionais sobre os frameworks AOM RoleMapper e Hibernate.

Vale a pena notar que um único participante afirmou que não foi capaz de concluir a tarefa com a abordagem 1 (A1) devido à falta de experiência em programação. No entanto, o mesmo participante foi capaz de completar a tarefa com a abordagem proposta (A2) com base na documentação fornecida e em uma curta pesquisa complementar. Portanto, a maioria dos participantes poderia entender a abordagem proposta com base na documentação fornecida para implementar com êxito a funcionalidade necessária, inclusive pelos desenvolvedores menos experientes.

4.2.9 Dificuldades encontradas (Q3)

Quanto às dificuldades encontradas pelos desenvolvedores, 14 participantes relataram algum tipo de dificuldade em utilizar a abordagem A1 e 18 participantes relataram alguma dificuldade em utilizar a abordagem proposta - A2. A Tabela 4.5 apresenta as principais dificuldades encontradas por eles.

Tabela 4.5 - Dificuldades Encontradas

Tipo de dificuldade	A1	A2
Entender a especificação e a documentação	28.57%	33.33%
Entender os testes de unidade	4.76%	0.00%
Trabalhar com as classe do AOM RoleMapper	28.57%	23.08%
Conhecer métodos da classe String	14.29%	N/A
Conhecer métodos da classe Date	28.57%	N/A
Trabalhar com o framework Hibernate Validator	N/A	33.33%
Definir metadados em entidades AOM	N/A	52.38%
Entender o adaptador de JavaBeans	N/A	28.57%

Conforme mostrado na Tabela 4.5, cerca de 29 % dos participantes encontraram dificuldades para entender a especificação e documentação na abordagem 1 e mais de 33 % dos participantes encontraram dificuldade de entender a documentação na abordagem 2.

Especificamente em relação à abordagem proposta, 52,38 % dos participantes (11 de 21) afirmaram que a principal dificuldade foi definir metadados em entidades AOM. Dificuldades para conhecer métodos relacionados a String (28.57 %) e Date (28.57 %) foram problemas encontrados exclusivamente quando alguns participantes usavam o código imperativo (A1), enquanto lidar com o Adaptador de JavaBeans foi um problema encontrado quando 6 participantes (28,57 %) estavam usando a abordagem proposta (A2).

4.2.10 Benefícios e desvantagens percebidos (Q4)

A Tabela 4.6 apresenta os benefícios de ambas as abordagens que foram destacadas pelos participantes. Em relação a abordagem 1 (A1), o principal benefício destacado por 8 participantes foi a simplicidade, que não foi mencionada por nenhum participante com referência à abordagem proposta (A2). No entanto, alguns participantes apontaram como benefícios da abordagem 2 são a facilidade de manutenção (5 de 21), o fato de reutilizar uma estrutura existente (6 de 21) e a possibilidade de armazenar e alterar rapidamente os metadados de validação (6 de 21).

Tabela 4.6 - Benefícios Percebidos

Benefícios	A1	A2
Simplicidade	38.10%	0.00%
Velocidade de Desenvolvimento	4.76%	19.05%
Facilidade de manutenção	4.76%	23.81%
Não necessitar de conhecimento prévio	28.57%	N/A
Flexibilidade	14.29%	N/A
Reusar um framework existente	N/A	28.57%
Metadados de validação ser armazenados e alterados rapidamente	N/A	28.57%

A Tabela 4.7 apresenta as desvantagens percebidas pelos participantes. Cerca de 43 % dos participantes (9 de 21) relataram que a principal desvantagem é o código repetitivo gerado para várias entidades na abordagem 1 (A1). Além disso, 6 participantes (28,57 %) relataram que a informação de validação é hard-coded. Quanto à abordagem 2 (A2), cerca de 52 % dos participantes (11 de 21) relataram que a principal desvantagem se deveu a uma curva de aprendizado necessária para entender como fazer para criar a entidade AOM; e 6 participantes (28,57 %) relataram que a tarefa de definir metadados é complexa.

Tabela 4.7 - Desvantagens Percebidas

Desvantagens	A1	A2
O código gerado é mais complexo	9.52%	14.29%
Velocidade de Desenvolvimento	4.76%	4.76%
Código repetitivo seria gerado em várias entidades	42.86%	N/A
Os valores das validações são hard-coded	28.57%	N/A
Dificuldade de reúso	9.52%	N/A
Difículdade de entender o Adaptador de JavaBean	N/A	4.76%
Complexidade de definir metadados	N/A	23.81%
Curva de Aprendizado	N/A	52.04%

4.2.11 Nível de aceitação (Q5)

Como mencionado anteriormente, questionamos os participantes sobre qual das duas abordagens eles escolheriam para implementar um dado recurso de validação. Além disso, pedimos que fornecessem o motivo de suas escolhas. Pretendemos a partir desta questão saber qual o nível de aceitação por parte dos programadores relativamente à abordagem proposta para utilizar a funcionalidade de um framework tradicional que usa o adaptador gerado para a entidade AOM. A análise dos dados qualitativos consistiu em mapear os motivos relacionados pelos participantes com

as dificuldades, vantagens e desvantagens mencionadas nas sessões anteriores.

Apenas três participantes escolheram a abordagem 1 (A1). Um dos participantes justificou sua escolha destacando duas dificuldades da abordagem 2 (A2): (i) dificuldades para lidar com o JavaBean; e (ii) dificuldades na definição e mapeamento de metadados. Como justificativa, o segundo participante destacou dois benefícios da abordagem 1 (A1) - a flexibilidade e o maior controle no desenvolvimento. E o terceiro participante justificou sua escolha também por causa da dificuldade em definir e mapear metadados. Tais razões podem estar relacionadas ao tempo de experiência profissional dos participantes, uma vez que a produtividade para eles parece ser uma medida fundamental.

Os outros 18 participantes (85,7 %) escolheram a abordagem proposta (A2). Constatamos que, apesar de uma curva de aprendizado necessária para entender como fazer e outras dificuldades encontradas, alguns participantes perceberam a utilidade da abordagem devido à reutilização do código. Após o período de aprendizagem, alguns participantes perceberam que o resultado poderia ser uma manutenção mais fácil. Vários participantes mencionam a simplicidade e eficiência da abordagem para inserir e alterar validações de forma consistente, bem como flexibilidade de código com a criação de adaptadores e sua legibilidade. Um dos participantes afirmou que, a partir da compreensão de como fazer a transformação para JavaBean, a escrita das especificações tornou-se muito intuitiva. Então, ele levou apenas 8 minutos - o menor tempo relatado - para completar a tarefa usando a abordagem 2 (A2). Além disso, um dos participantes sugeriu a criação de um conjunto de regras de validação para evitar as parametrizações tanto quanto possível.

4.3 Ameaças à Validade

Nesta seção, discutimos possíveis ameaças à validade do experimento em termos de validade interna, validade externa, validade de construção e validade de conclusão, de acordo com Wohlin. (WOHLIN et al., 2012).

4.3.1 Validade Interna

Ameaças à validade interna dizem respeito a análises se, de fato, o tratamento causa os resultados (o efeito) (WOHLIN et al., 2012). Evitamos algumas ameaças sociais à validade interna - por exemplo, rivalidade compensatória e desmoralização ressentida - uma vez que todos os participantes experimentaram ambos os tratamentos.

Elaboramos duas tarefas diferentes que foram aplicadas de forma cruzada para dis-

tinguir os grupos e mitigar o efeito de difusão ou imitação de tratamentos, que ocorre quando um grupo aprende sobre o tratamento de outro grupo. A amostra inclui participantes com conhecimento semelhante - alunos em disciplina de pós-graduação com conhecimento em programação Java, mas com experiências diferentes. No entanto, verificamos que os dois grupos estavam equilibrados em termos de conhecimento prévio. Todos os participantes foram qualificados para o experimento. Foi fornecido o mesmo treinamento para alinhar seu conhecimento sobre reflexão e anotações de código. Além disso, todos os participantes foram guiados pelo mesmo procedimento experimental, incluindo as instruções para alocar um período de tempo buscando evitar interrupções e escolher um ambiente confortável para realizar suas tarefas.

4.3.2 Validade Externa

Ameaças para construir validade dizem respeito à generalização dos resultados para a prática industrial (WOHLIN et al., 2012). Todos os participantes são estudantes regulares de mestrado ou doutorado que frequentaram o curso Design Patterns. Não obstante, naquele momento, a maioria deles trabalhava na indústria. Além disso, o experimento ocorreu no ambiente do desenvolvedor para reproduzir uma situação mais realista e confortável. Embora acreditemos que os resultados possam ser generalizados para a prática industrial, é ideal que o experimento possa ser replicado em ambiente industrial envolvendo desenvolvedores profissionais e em comunidades de desenvolvedores Java. As tarefas eram pequenas, porque o objetivo era criar e validar duas entidades simples, permitindo um experimento mais simples, a fim de evitar a fadiga dos participantes (validade interna). Tais tarefas, no entanto, representam apenas uma parte da atividade real de construção de software. Para ter um cenário mais próximo de construir um software completo, podemos criar outros experimentos mais completos, como fazer uma funcionalidade completa do sistema ou implementar um caso de uso completo com seus fluxos básicos e alternativos sendo testados.

4.3.3 Validade de Construção

Ameaças para validade de construção do experimento é a análise sobre a relação entre teoria e observação (WOHLIN et al., 2012). Foi mitigado o viés de mono-operação através da elaboração de duas tarefas diferentes com um número equivalente de requisitos e recursos. No entanto, o experimento foi sujeito ao viés de mono-método porque envolveu uma única medida objetiva - tempo gasto para executar as tarefas. Outras medidas poderiam ser consideradas, como produtividade e eficácia. No entanto, o código para usar a estrutura e os testes de integração já haviam sido

preparados anteriormente e as tarefas eram pequenas. Assim, o número de linhas de código produzidas pelos participantes foi muito semelhante. Em relação ao delineamento experimental, poderíamos ter também adotado o cruzamento de tratamentos para verificar se a ordem de aplicação é um fator que pode influenciar os resultados.

4.3.4 Validade de Conclusão

Ameaças à validade de conclusão dizem respeito a questões que poderiam afetar a análise sobre os resultados do experimento, por exemplo, escolha dos tamanhos das amostras, escolha dos testes estatísticos, cuidados tomados na implementação e mensuração dos testes de um experimento (WOHLIN et al., 2012). Este experimento possui baixo poder estatístico dos testes, uma vez que o número de participantes não é suficiente para obter um efeito significativo. Por isso, consideramos os dados como indicadores, em vez de conclusivos. Quanto à escolha dos testes estatísticos, verificamos a normalidade dos dados antes de escolhê-los. Assim, usamos tanto estatística paramétrica quanto não paramétrica para testar as hipóteses, levando em conta que a análise foi realizada em amostras pareadas. Além disso, escolhemos um nível apropriado de significância ao testar hipóteses nulas.

5 IMPLEMENTAÇÃO E MAPEAMENTO DO MODELO COMPORTAMENTAL DO AOM

Neste capítulo é apresentado o modelo desenvolvido para uma proposta de abordagem para a implementação e mapeamento do modelo comportamental na arquitetura AOM. Para isso, foram desenvolvidas novas funcionalidades no framework AOM RoleMapper. O principal objetivo é permitir a utilização de frameworks tradicionais para entidades criadas em tempo de execução em uma arquitetura AOM.

5.1 Mapeamento do padrão Rule Object

O padrão Rule Object, apresentado na seção 2.4.5, é uma abstração das regras de negócio nas entidades na arquitetura AOM. A implementação desse padrão normalmente é composta de uma classe com um método para executar as regras que ele abstrai e propriedades que são utilizadas na execução destes métodos. Para adicionar uma nova regra, é necessário informar o nome a regra e a classe que implementa esta abstração, que é armazenada em uma mapa. Durante a execução da regra, é utilizado o nome da regra como chave para descobrir qual regra será executada.

As próximas seções apresentam as abordagens propostas para o mapeamento do Rule Object do modelo independente de domínio para os outros modelos.

5.1.1 Mapeamento de Dependente para Independente de Domínio

Para realizar o mapeamento de entidades AOM específicas de domínio para entidades independente de domínio foram criadas anotações que são utilizadas para mapear as classes, métodos e atributos da entidade específica de domínio para que o framework identifique os elementos que serão utilizados neste mapeamento.

De forma similar, para utilizar o Rule Object neste mapeamento, a criação de anotações de regra torna-se uma alternativa para que estes elementos sejam marcados para serem reconhecidos pelo framework e utilizados na criação dos adaptadores.

O modelo AOM dependente de domínio cria as regras de duas formas. Ele pode ter métodos fixos ou ter um mapa de Rule Objects, utilizando a sua própria interface. Como o modelo independente de domínio tem uma única interface, para adaptar as regras dos métodos do modelo dependente, existem dois adaptadores que utilizam reflexão para realizar a invocação das regras. Um adaptador encapsula uma implementação do Rule Object com a interface dependente de domínio e o outro possui a lógica para a invocação do método fixo na classe do tipo de entidade.

5.1.2 Mapeamento de Independente de Domínio para Classes Estáticas

Em classes estáticas, a representação de comportamento é feita com métodos. Deste modo para realizar o mapeamento de entidades AOM independentes de domínio para classes estáticas é necessário incluir um método para cada Rule Object presente.

Este mapeamento é realizado no momento da criação dos adaptadores pela manipulação de bytecode. Para que estes métodos sejam criados na classe adaptadora, é necessário acessar os Rule Objects presentes no tipo de entidade AOM. Nessa operação são recuperados o nome da regra que será executada e uma instância do RuleObject que será utilizado para executar esta regra. No caso do RuleObject precisar de parâmetros para ser executado, eles devem ser passados quando ele for invocado.

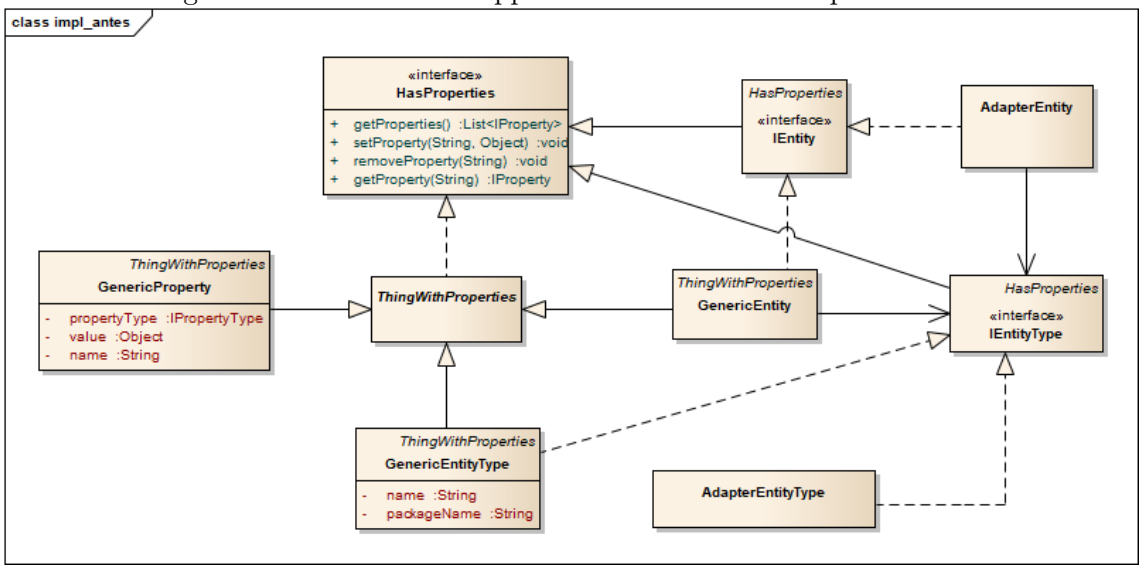
Em casos que houver a necessidade de realizar um mapeamento, devem ser adicionadas metadados no Rule Object, representados por suas propriedades, para que seja feita a transformação dessas propriedade em anotações, baseadas em configurações do arquivo JSON como o da seção 3.4. Esta possibilidade é importante para os frameworks baseados em metadados e reflexão, pois em sua maioria, eles localizam os métodos para invocar através de suas anotações.

5.2 Implementação do Rule Object no AOM Role Mapper

Nesta seção será apresentado como o padrão Rule Object foi introduzido no modelo independente de domínio do framework AOM RoleMapper. Essa implementação é a base para que posteriormente seja feito o mapeamento.

Inicialmente, apresentamos a Figura 5.1 com o diagrama do AOM RoleMapper antes do desenvolvimento da funcionalidade de comportamento.

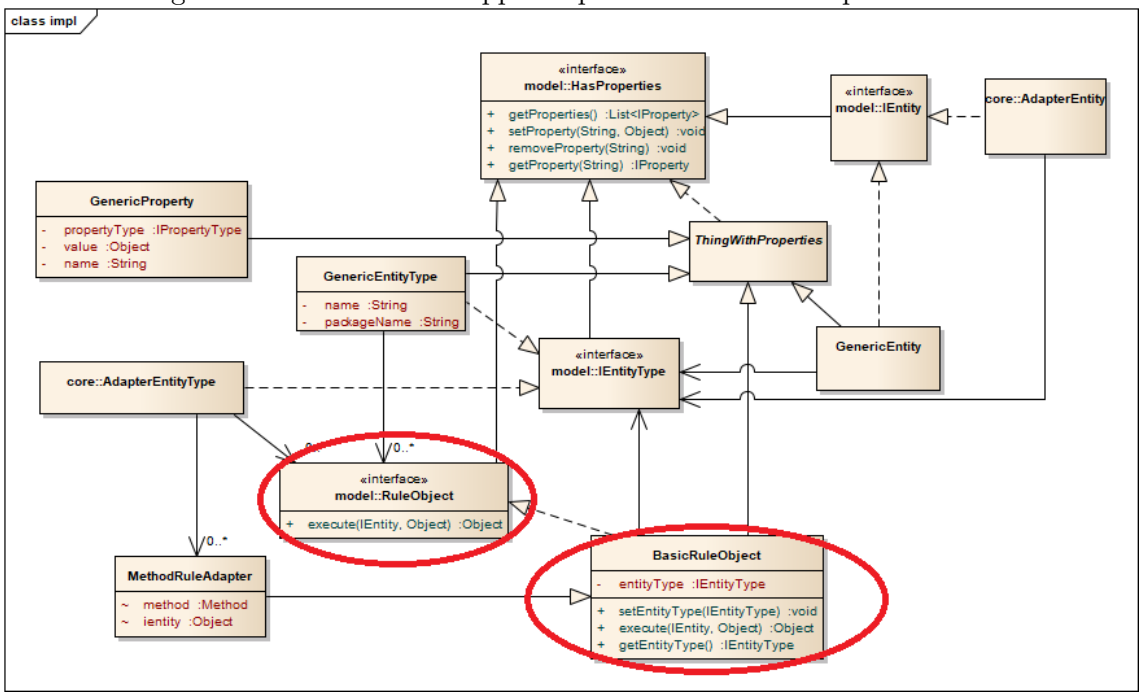
Figura 5.1 - AOM RoleMapper antes do modelo comportamental



Para introduzir o modelo comportamental no AOM RoleMapper, foram feitas algumas alterações no framework.

A primeira alteração foi a criação da interface RuleObject com o intuito de estabelecer um contrato para todo comportamento. A interface RuleObject estende a interface HasProperties, assim como as interfaces IEntity e IEntityType, para que se tenha metadados associados a ele. Em seguida foi criada a classe BasicRuleObject que implementa a interface RuleObject e tem herança da classe ThingWithProperties para que o modelo comportamental tenha a suas propriedades, conforme o diagrama apresentado na Figura 5.2 .

Figura 5.2 - AOM RoleMapper depois do Modelo Comportamental



A interface RuleObject, conforme a Listagem 5.1, deve ser implementada por classes que representam o comportamento de entidades AOM, implementando o método execute. Esse método recebe dois parâmetros, sendo que o primeiro parâmetro é a entidade na qual o método será executado, e o segundo parâmetro um array do tipo Object com os parâmetros que o método recebe para sua execução.

Listagem 5.1 - Definição do método execute da Interface RuleObject

```
1 public interface RuleObject extends HasProperties {
2    public Object execute(IEntity obj, Object... params);
3 }
```

Para permitir a adição de um RuleObject em um tipo de entidade AOM, foi adicionada uma nova operação na interface IEntityType, o método addOperation. Ele recebe como parâmetro o nome da regra e uma instância de uma classe que implementa a interface RuleObject, conforme apresentado na Listagem 5.2 . O RuleObject será armazenado em um mapa, para sua execução a partir do seu nome.

Listagem 5.2 - Exemplo de adicao de RuleObject em uma entidade AOM

```
1       IEntityType tipoProduto = new GenericEntityType("Produto");
2       tipoProduto.addOperation("anosFabricacao", new CalculaAnos("dataFabricacao"));
```

Outra alteração no modelo foi a inclusão do método executeOperation na interface IEntity, apresentado na Listagem 5.3. Esse método é usado para executar o comportamento em uma entidade. Ele possui dois parâmetros, que são o nome regra que será executada e um array do tipo Object com os parâmetros que o método precisa receber para ser executado. O nome da regra é o mesmo nome que foi utilizado para adicionar a operação na interface IEntityType.

Listagem 5.3 - Definição do método de regra executeOperation na Interface IEntity

```
1  public interface IEntity extends HasProperties {
2   //... outros mtodos omitidos
3     public Object executeOperation(String name, Object... params);
4  }
```

As classes GenericEntity e AdapterEntity, que implementam a interface IEntity, na implementação do método executeOperation, obtém o RuleObject do tipo de entidade correspondente e invoca o método execute com os seus parâmetros, para executar a regra.

5.3 Funcionalidades do Modelo Comportamental

Esta seção apresenta as funcionalidades adicionadas no framework AOM RoleMapper e exemplos de sua utilização para o funcionamento do modelo comportamental das entidades AOM.

5.3.1 Adição e execução de comportamento

Para adicionar um RuleObject em uma entidade AOM, o primeiro passo é criar o objeto do tipo IEntityType. A Listagem 5.4 apresenta a criação da IEntityType de nome Produto com os atributos, dataFabricacao do tipo Date e o atributo nome do tipo String. Ambos atributos estão mapeados como obrigatórios.

Listagem 5.4 - Adicionar um RuleObject em uma entidade - parte 1

```
1  public static IEntityType getTipoEntidadeCobranca() throws EsfingeAOMException{
2     IEntityType produto = new GenericEntityType("Produto");
3     //criando property types
4     GenericPropertyType dataNascPropertyType = new GenericPropertyType("dataFabricacao", Date.class);
5     dataNascPropertyType.setProperty("notempty", true);
6     GenericPropertyType nomePropertyType = new GenericPropertyType("nome", String.class);
7     nomePropertyType.setProperty("notempty", true);
8     //adicionando property types no tipo de entidade
9     produto.addPropertyType(dataNascPropertyType);
10    produto.addPropertyType(nomePropertyType);
11    return produto;
12 }
```

O próximo passo é adicionar o RuleObject no tipo da entidade. A Listagem 5.5 cria o RuleObject do tipo CalculaAnos, no seu método construtor está um parâmetro do tipo String, que é o nome do atributo que ele deverá acessar na entidade ao executar o método do RuleObject. O nome da operação que será executada, neste exemplo, chama-se anosFabricacao, este nome é a chave para encontrar o RuleObject para a sua execução posteriormente.

Listagem 5.5 - Adicionar um RuleObject em uma entidade - parte 2

```
1 public static void criarTipoEntidade() throws EsfingeAOMException{
2    IEntityType tipoProduto = FabricaTiposProduto.getTipoEntidadeCobranca();
3    tipoProduto.addOperation("anosFabricacao", new CalculaAnos("dataFabricacao"));
4 }
```

A listagem 5.6 apresenta o RuleObject de nome CalculaAnos. Para que para o objeto tornar-se um RuleObject ele deve estender a classe BasicRuleObject que implementa a interface RuleObject com seu método execute.

Listagem 5.6 - Código do RuleObject CalculaAnos

```
1  public class CalculaAnos extends BasicRuleObject {
2     String paramName;
3
4     public CalculaAnos(String paramName) {
5        this.paramName = paramName;
6     }
7
8     @Override
9     public Object execute(IEntity obj, Object... params) {
10       try {
11          IProperty property = obj.getProperty(paramName);
12          GregorianCalendar gregorianCalendar = new GregorianCalendar();
13
14          if (property.getValue() instanceof String) {
15             try {
16                String value = (String) property.getValue();
17                Date parse = new SimpleDateFormat("dd/MM/yyyy").parse(value);
18                gregorianCalendar.setTime(parse);
19             } catch (ParseException e) {
20                e.printStackTrace();
21             }
22          } else if (property.getValue() instanceof Date) {
23             if (property instanceof AdapterFixedProperty) {
24                AdapterFixedProperty adp = (AdapterFixedProperty) property;
25                Date dataObj = (Date) adp.getValue();
26                gregorianCalendar.setTime(dataObj);
27             } else {
28                GenericProperty gn = (GenericProperty) property;
29                Date dataObj = (Date) gn.getValue();
30                gregorianCalendar.setTime(dataObj);
31             }
32          }
```

```
33
34          LocalDate today = LocalDate.now();
35          int year = 1;
36          int month = 2;
37          int day = 5;
38          LocalDate birthday = LocalDate.of(gregorianCalendar.get(year), gregorianCalendar.get(month)
                + 1,
39               gregorianCalendar.get(day));
40          Period p = Period.between(birthday, today);
41
42          return p.getYears() + " anos";
43        } catch (EsfingeAOMException e) {
44          e.printStackTrace();
45        }
46        return null;
47      }
48  }
```

Na linha 1 pode se verificar que a classe é um RuleObject pois ela herda da classe BasicRuleObject. Na linha 2 o atributo de nome paramName que vai receber o nome da propriedade da qual será obtido o valor para executar a regra é declarado. Na linha 4 tem o método construtor da classe CalculaAnos. Na linha 9 tem a implementação do método execute que realiza a cálculo de diferença de anos entre a data de hoje e a data para na propriedade dataFabricacao. Na linha 42 é retornado o resultado do cálculo realizado pelo método.

A Listagem 5.7 apresenta a criação da entidade produto e são preenchidos os valores para os atributos nome e dataFabricacao.

Listagem 5.7 - Adicionar um RuleObject em uma entidade - parte 3

```
1  public void criarProdutoCorreto() throws EsfingeAOMException {
2    IEntity produto = tipoProduto.createNewEntity();
3    produto.setProperty("nome", "Notebook DELL");
4    GregorianCalendar dataFabr = new GregorianCalendar();
5    dataFabr.set(2010, 11, 23);
6    produto.setProperty("dataFabricacao", dataFabr.getTime());
7  }
```

O último passo é obter o resultado é a execução do RuleObject. Para isso é necessário invocar o método executeOperation, passando como parâmetro uma String com o nome na chave para executar o RuleObject correto. A Listagem 5.8, apresenta a execução do RuleObject com a chave anosFabricacao. O resultado do tipo Object é convertido em tipo inteiro e feito um assertTrue para validação do teste.

Listagem 5.8 - Executando a regra em uma entidade

```
1  public void validaProdutoBase() throws Exception{
```

```
2     Object resultOperation = produto.executeOperation("anosFabricacao");
3     int years = (int) resultOperation;
4     boolean result = years < 7;
5     assertTrue("produto valido ", result);
6  }
```

5.3.2 Execução de Comportamento na mudança em uma propriedade

Dentre as funcionalidades implementadas no AOM RoleMapper está a de permitir que regras sejam executadas quando ocorre a mudança de valor em uma propriedade.

Para esta funcionalidade, foi incluído um mapa com os atributos que serão monitorados e a regra que será executada como uma reação a mudança do valor do atributo.

Dessa forma, quando o valor de um atributo é alterado, é feita uma verificação se o atributo está sendo monitorado e, em caso afirmativo, é executada a regra e o resultado armazenado para ser recuperado posteriormente pelo método getResultOperation.

A Listagem 5.9 apresenta o teste desenvolvido com uma regra de que no caso da data de fabricação do produto for maior do que hoje ela arremessa uma exceção. Na linha 22 do teste é colocado o valor de 23 de novembro de 2017 e a regra executa com sucesso e retorna a quantidade de dias entre hoje e a data de fabricação. Na linha 31 a propriedade dataFabricacao é adicionada como um parâmetro monitorado, invocando o método addPropertyMonitored e na linha 35 o valor de dataFabricacao foi alterado para 23 de novembro de 2030. Desta forma ao verificar o valor retornado no método getResultOperation, será detectado que valor do atributo monitorado foi alterado e será feita a execução internamente pelo próprio framework do RuleObject e uma exceção será arremessada.

Listagem 5.9 - Executando a regra em uma entidade

```
1  @Test(expected = EsfingeAOMException.class)
2  public void testChangePropertyValue() {
3     IEntityType tipoProduto = new GenericEntityType("ProdutoGelado");
4
5     GenericPropertyType dataNascPropertyType = new GenericPropertyType("dataFabricacao", Date.class);
6     dataNascPropertyType.setProperty("notempty", true);
7
8     GenericPropertyType nomePropertyType = new GenericPropertyType("nome", String.class);
9     nomePropertyType.setProperty("notempty", true);
10
11    // adicionando property types no tipo de entidade
12    tipoProduto.addPropertyType(dataNascPropertyType);
```

```
13      tipoProduto.addPropertyType(nomePropertyType);
14
15      // adiciona a operacao de comportamento
16      tipoProduto.addOperation("menorqhoje", new MenorQHoje("dataFabricacao"));
17
18      IEntity produto = tipoProduto.createNewEntity();
19      produto.setProperty("nome", "Yogurt X");
20      GregorianCalendar dataFabr = new GregorianCalendar();
21      dataFabr.set(2017, 11, 23);
22      produto.setProperty("dataFabricacao", dataFabr.getTime());
23
24      // executa o comportamento
25      Object resultado = produto.executeOperation("menorqhoje");
26      Long days = (Long) resultado;
27      System.out.println(days);
28      assertTrue("@@@ Data de Fabricao menor do que hoje ", days > 0);
29
30      // monitora a propriedade dataFabricacao
31      produto.addPropertyMonitored("dataFabricacao", "menorqhoje");
32
33      dataFabr.set(2030, 11, 23);
34      // muda o valor da propriedade monitorada
35      produto.setProperty("dataFabricacao", dataFabr.getTime());
36      Object resultado2= produto.getResultOperation("menorqhoje");
37
38      days = (Long) resultado2;
39      assertTrue("@@@ Data de Fabricao menor do que hoje ", days > 0);
40  }
```

5.3.3 RuleObject que executa Expression Language

Expression Language (EL) (ORACLE, 2018) é uma linguagem utilizada em projetos Java onde se criam expressões simples que tem acesso direto aos atributos dos objetos que estão no padrão JavaBeans (SUN MICROSYSTEMS, 1997). No framework AOM RoleMapper foi desenvolvido um Rule Object que executa uma Expression Language passada como parâmetro.

A utilização deste tipo de RuleObject tem sua importância na configuração de cálculos e validação de atributos que precisam ser alterados em tempo de execução. Tendo seu comportamento baseado em uma expressão configurada, a lógica pode ser facilmente modificada através da mudança nesta expressão.

Para isso, foi criada a classe ELContextAOM que representa o contexto onde a EL será executada. Esta classe tem três atributos, sendo a primeira chamada de FunctionMapper para mapear as funções da EL, VariableMapper para mapear variáveis e a terceira chamada de CompositeELResolver para interpretar as variáveis e resolver o seu valor. A Listagem 5.18 apresenta a classe ELContextAOM.

Listagem 5.10 - Classe ELContextAOM para execução de Expression Language

```
1  public class ELContextAOM extends ELContext {
2     private FunctionMapper functionMapper;
3     private VariableMapper variableMapper;
4     private CompositeELResolver elResolver;
5
6     public ELContextAOM(FunctionMapper functionMapper, VariableMapper variableMapper, ELResolver...
          resolvers) {
7        this.functionMapper = functionMapper;
8        this.variableMapper = variableMapper;
9        elResolver = new CompositeELResolver();
10       for (ELResolver resolver : resolvers) {
11          elResolver.add(resolver);
12       }
13    }
14    public static EvaluationContext criarContexto(Class<?> functionClass, Map<String, Object>
          attributeMap) {
15       VariableMapper vMapper = mapearVariaveis(attributeMap);
16       FunctionMapper fMapper = mapearFuncoes(functionClass);
17       ELContextAOM context = new ELContextAOM(fMapper, vMapper, new ArrayELResolver(), new
             ListELResolver(),
18           new MapELResolver(), new BeanELResolver());
19       return new EvaluationContext(context, fMapper, vMapper);
20    }
21    public static Object execute(String expr, Class<? extends Object> objectClass, Map<String,
          Object> map) {
22       EvaluationContext ec = ELContextAOM.criarContexto(objectClass, map);
23       ValueExpression result = new ExpressionFactoryImpl().createValueExpression(ec, expr,
             Object.class);
24       return result.getValue(ec);
25    }
26 }
```

Para inserir regras de EL na entidade AOM foi criada uma classe do tipo RuleObject específico para expression language. A classe de nome ExpLangRuleObject implementa a interface RuleObject para ter as mesmas abstrações dos objetos de regra e principalmente o método para executar as expressões regulares.

Para executar uma EL é necessário um parâmetro que é o nome da regra que será executada.

Um exemplo de execução de EL apresentado na Listagem 5.11. Neste exemplo é realizado um cálculo de mediana utilizando valores da entidade produto.

Listagem 5.11 - Exemplo de teste de uma Expression Language

```
1  @Test
2  public void testELMediana() {
3     try {
4        IEntityType tipoProduto = new GenericEntityType("Produto");
5
6        // criando property types
```

```
7       GenericPropertyType nomePropertyType = new GenericPropertyType("nome", String.class);
8       nomePropertyType.setProperty("notempty", true);
9
10      GenericPropertyType msMin = new GenericPropertyType("medidaMinima", Double.class);
11      GenericPropertyType msMax = new GenericPropertyType("medidaMaxima", Double.class);
12
13      // adicionando property types no tipo de entidade
14      tipoProduto.addPropertyType(nomePropertyType);
15      tipoProduto.addPropertyType(msMin);
16      tipoProduto.addPropertyType(msMax);
17
18      String expr = "${ (medidaMinima+medidaMaxima) / 2 }";
19      String nameRule = "mediana";
20      tipoProduto.addOperation(nameRule, new ExpLangRuleObject(expr));
21
22      IEntity produto = tipoProduto.createNewEntity();
23      produto.setProperty("nome", "Sensor");
24      produto.setProperty("medidaMinima", 15);
25      produto.setProperty("medidaMaxima", 10);
26
27      Object result = produto.executeOperation(nameRule);
28      System.out.println(" Operao  retornou " + result);
29
30    } catch (Exception e) {
31      e.printStackTrace();
32      assertTrue(false);
33    }
34  }
```

Na linha 4 é criado o tipo de entidade, de nome tipoProduto. Da linha 7 a linha 11 são criados os tipos de propriedades. Da linha 14 a 16 os tipos de propriedades são adicionados aos tipos de entidade. Na linha 18 é criada a Expression Language de nome expr. Na linha 20 o RuleObject é adicionado ao tipo de entidade. Na linha 22 é criada a entidade produto. Da linha 23 a 25 são inseridos valores na entidade produto. Na linha 27 é executada a regra mediana. Na linha 28 é exibido o resultado do RuleObject.

5.4 Mapeamento de regras de um AOM específico de domínio para as interfaces do AOM Role Mapper

Para fazer o mapeamento de uma entidade de domínio específico é necessário utilizar as anotações. Por meio delas, o framework identifica e realiza a criação de adaptadores que fazem as ligações entre as entidade de domínio específico e as entidades independentes de domínio.

Para realizar o mapeamento de comportamento, foram criadas três anotações:

- RuleClass - Realiza o mapeamento de classes de comportamento e deve anotar as interfaces para Rule Objects específicas de domínio;
- RuleMap - Realiza o mapeamento de mapa de atributos ligando nome da regra e a classe de comportamento e deve mapear o atributo no Entity Type que armazena os seus Rule Objects;
- RuleMethod - Realiza o mapeamento de um método que implementa um comportamento. Pode ser utilizado para marcar o método de uma Interface quanto métodos fixos em entidades;

Um exemplo de mapeamento da entidade específica de domínio será apresentado nas próximas três listagens. Nela são utilizadas as anotações para fazer o mapeamento da entidade SensorType para uma entidade independente de domínio e executar o RuleObject. A Listagem 5.14 apresenta a Interface OperacaoSensor. Na linha 1 pode-se observar que a classe foi mapeada com a anotação RuleClass ou seja será adaptado como uma classe de comportamento. Na linha 3 observa-se o método operação mapeado com a anotação RuleMethod, ou seja, será utilizado como um método de comportamento.

Listagem 5.12 - Interface específica de domínio OperacaoSensor

```
1 @RuleClass
2 public interface OperacaoSensor{
3    @RuleMethod
4    Object operacao(Sensor s, Object... params);
5 }
```

A Listagem 5.13 apresenta a classe OperacaoRegraSensor que implementa a Interface OperacaoSensor. Este é um exemplo de um RuleObject que implementa uma interface específica de domínio.

Listagem 5.13 - Classe específica de domínio OperacaoRegraSensor

```
1 public class OperacaoRegraSensor implements OperacaoSensor {
2    @Override
3    public Object operacao(Sensor s, Object... params) {
4       return 10;
5    }
6 }
```

A Listagem 5.14 apresenta o mapeamento da entidade específica de domínio, de nome SensorType. Nela são utilizadas as anotações para fazer o mapeamento desta classe para uma entidade independente de domínio e executar o RuleObject.

Listagem 5.14 - Mapear um RuleObject

```
1  @EntityType
2  public class SensorType {
3     @RuleMap
4     private Map<String, OperacaoSensor> operations = new HashMap<>();
5
6     @PropertyType
7     private Set<SensorPropertyType> propertyTypes = new HashSet<SensorPropertyType>();
8
9     public void addPropertyTypes(SensorPropertyType propertyType) {
10       propertyTypes.add(propertyType);
11    }
12
13    @RuleMethod
14    public int converterUnidade(String propriedade, String unidade) {
15       return 1;
16    }
17
18    @CreateEntityMethod
19    public ISensor createSensor() {
20       Sensor sensor = new Sensor();
21       sensor.setSensorType(this);
22       if (operations == null) {
23          operations = new HashMap<>();
24       }
25       return sensor;
26    }
27 }
```

Na linha 3 pode-se observar o mapeamento do atributo operations com a anotação @RuleMap. Esta anotação mapeia as operações de regra que serão inseridas nesta entidade. Na linha 13 é realizado o mapeamento do @RuleMethod conveterUnidade. Esta anotação mapeia um método fixo que será invocado pela entidade adaptada.

Na Listagem 5.15 é feito o teste para criar os adaptadores da entidade de domínio específico Sensor, verificando se os métodos foram executados corretamente. Este exemplo demonstra as duas formas de mapeamento de RuleObject, utilizando método fixo e com RuleObject específico de domínio.

Listagem 5.15 - Testes de regras anotadas em um RuleObject

```
1  @Test
2  public void testAdaptarSensor() {
3     try {
4        SensorType sensorType = new SensorType();
5        sensorType.addOperacao("regraSensor", new OperacaoRegraSensor());
6
7        SensorPropertyType prop = new SensorPropertyType();
8        prop.setName("unidade");
9        prop.setPropertyType(String.class);
10       sensorType.addPropertyTypes(prop);
11
```

```
12      SensorPropertyType prop2 = new SensorPropertyType();
13      prop2.setName("propriedade");
14      prop2.setPropertyType(String.class);
15      sensorType.addPropertyTypes(prop2);
16
17      Sensor sensor = new Sensor();
18      sensor.setSensorType(sensorType);
19
20      AdapterEntityType adaptedEntityType = AdapterEntityType.getAdapter(sensorType);
21      AdapterEntity entity = AdapterEntity.getAdapter(adaptedEntityType, sensor);
22
23      // vai executar a operacao adicionada no modelo dependente de
24      // dominio
25      Object resultDepend = entity.executeOperation("regraSensor", sensor, null);
26      Assert.assertEquals(10, resultDepend);
27
28      entity.setProperty("unidade", "K");
29      entity.setProperty("propriedade", "grau");
30
31      // vai executar a operacao fixa no modelo independente de dominio
32      Object resultIndep = entity.executeOperation("converterUnidade", "propriedade", "unidade");
33      Assert.assertEquals(1, resultIndep);
34    }catch (Exception e) {
35      e.printStackTrace();
36      Assert.assertTrue(false);
37    }
38  }
```

Na linha 4 é instanciado o tipo de entidade. Na linha 5 é adicionada a operação regraSensor no tipo de entidade sensorType. Da linha 7 a 15 são criadas tipos de propriedades e adicionadas a entidade sensorType. Na linha 17 é instanciada a classe Sensor. Na linha 18 é atribuído o sensorType para o sensor. Na linha 20 é invocado o adaptador do tipo de entidade. Na linha 21 é invocado o adaptador da entidade Sensor. Na linha 25 é executado o método de comportamento regraSensor, invocando o método operacao da classe OperacaoRegraSensor. Nas linha 28 e 29 são atribuídos valores na entidade Sensor. Na linha 32 é invocado o método fixo converterUnidade, a partir da entidade.

Uma extensão deste teste é apresentado na Listagem 5.16 para mostrar que também funciona em entidades dependentes de domínio adaptadas. Na linha 4 é obtido o método para invocar a regra regraSensor a partir no adaptador de classe estática. Na linha 5 é executada a regra de nome regraSensor. Na linha 6 é exibido o resultado da execução.

Listagem 5.16 - Testes de regras anotadas em um RuleObject

```
1  // gerar o adaptador e invocar o metodo regraSensor
2  AdapterFactory af = AdapterFactory.getInstance("JsonMapTest.json");
3  Object adapted = af.generateAdapted(entity, "sensorAdaptado");
```

```
4 Method declaredMethod = adapted.getClass().getDeclaredMethod("regraSensor", String.class,
      Object[].class);
5 Object resultOperation = declaredMethod.invoke(adapted, "regraSensor", new Object[2]);
6 System.out.println("adaptada retornou: " + resultOperation);
7 Assert.assertEquals(10, resultDepend);
```

5.5 Modelo Comportamental no Adaptador para JavaBeans

Outra funcionalidade desenvolvida para o modelo de comportamento foi a inserção do comportamento do RuleObject como métodos no adaptador criado para manter a compatibilidade com o frameworks que trabalham com o padrão JavaBeans.

Por meio da ajuda do componente ASM (JAVA BYTECODE MANIPULATION, 2017), ao gerar a classe de adaptador em tempo de execução, são adicionados os bytecodes para o método que representa o Rule Object, cujo comportamento consiste em invocá-lo na entidade encapsulada.

No exemplo da Listagem 5.17, é criado o tipo de entidade de nome tipoProduto, adicionada a operação chamada de periodoConsumo e é feita a criação da entidade produto. Essa entidade será usada como base para o exemplo.

Listagem 5.17 - Geração da entidade com RuleObject para o exemplo

```
1 IEntityType tipoProduto = new GenericEntityType("Produto");
2 tipoProduto.addOperation("periodoConsumo", new PeriodoConsumo("dataFabricacao"));
3 IEntity produto = tipoProduto.createNewEntity();
4 produto.setProperty("validade", 90);
```

A Listagem 5.18 apresenta um exemplo de teste para criar o adaptador e verificar que o método de comportamento foi incluído do objeto adaptado. Para isso é obtido o método de comportamento na linha 2 e na linha 3 é realizada a invocação do método por reflexão, similar a forma como os frameworks tradicionais fazem.

Listagem 5.18 - Teste de método inserido no adaptador para classe estática

```
1 Object personAdapter = af.generate(produto);
2 Method declaredMethod = personAdapter.getClass().getDeclaredMethod("periodoConsumo", String.class,
      Object[].class);
3 Object resultOperation = declaredMethod.invoke(personAdapter, "periodoConsumo", null);...
```

Durante a criação do adaptador, o framework adiciona o método que executa a regra de comportamento, utilizando o bytecode do método gerado pela biblioteca ASM, conforme a Listagem 5.19 .

Listagem 5.19 - Bytecode para adicionar método no adaptador

```
1  public static MethodVisitor createMethod(String name, String methodName, ClassWriter cw) {
2    MethodVisitor mv = cw.visitMethod(ACC_PUBLIC + Opcodes.ACC_VARARGS, methodName,
3         "(Ljava/lang/String;[Ljava/lang/Object;)Ljava/lang/Object;", null,
4         new String[] { "org/esfinge/aom/exceptions/EsfingeAOMException" });
5    mv.visitCode();
6    Label l0 = new Label();
7    mv.visitLabel(l0);
8    mv.visitLineNumber(44, l0);
9    mv.visitVarInsn(ALOAD, 0);
10   //... linhas omitidas
```

Outra funcionalidade desenvolvida no AOM RoleMapper para o mapeamento comportamental foi a adição de propriedades no RuleObject para que possa ser realizado mapeamento de métodos de regra em anotações de frameworks tradicionais, que utilizam anotações em métodos para reconhecer os seus serviços. Este mapeamento é realizado na criação do adaptador, utilizando as configurações de um arquivo JSON.

A Listagem 5.20 apresenta um exemplo de adição de uma propriedade no RuleObject de nome ConverteCelsiusToFah.

Na linha 1 é criada a instancia do RuleObject ConverteCelsiusToFah. Na linha 2 é criado um mapa de propriedades. Nas linhas 3 é inserida a propriedade nomeendpoint no mapa de propriedades. Na linha 4 é inserido o mapa de propriedades no RuleObject. Na linha 5 é inserida a operação de nome converteCelsiusToFahrenheit e o RuleObject conversor no tipo de entidade tipoSensor.

Listagem 5.20 - Inserindo propriedades no RuleObject

```
1  ConverteCelsiusToFah conversor = new ConverteCelsiusToFah("grausCelsius");
2  Map<String, Object> propriedades = new HashMap<>();
3  propriedades.put("nomeendpoint", "/magnetometro/conv2fah");
4  conversor.setOperationProperties(propriedades);
5  tipoSensor.addOperation("converteCelsiusToFahrenheit", conversor);
```

Na geração do adaptador é realizado o mapeamento do objeto adaptado utilizando as configurações do arquivo JSON. Um exemplo do arquivo JSON é apresentado na Listagem 5.21.

Listagem 5.21 - Exemplo de arquivo JSON para mapear objeto adaptado

```
1  {
2    "restcontroller":[
3      {"target":"class"},
4     {"annotationPath":"org.springframework.web.bind.annotation.RestController"}
5    ],
6    "autowired":[
```

```
7      {"target":"attribute"},
8     {"annotationPath":"org.springframework.beans.factory.annotation.Autowired"}
9    ],
10   "nomeendpoint":[
11     {"target":"method"},
12    {"annotationPath":"org.springframework.web.bind.annotation.RequestMapping"},
13    {"parameter_1": "value"}
14   ]
15 }
```

A próxima etapa é efetuar a geração do adaptador, de nome sensorAdaptado, conforme a Listagem 5.22. Na linha 1 é criada a entidade sensor. Na linha 2 é criado o AdapterFactory que tem o arquivo JSON apresentado na Listagem 5.21, com as configurações de mapeamento, no construtor desta classe. Na linha 3 é feita a geração do adaptador da entidade sensor.

Listagem 5.22 - Gerando Adaptador de um entidade com RuleObject

```
1 IEntity sensor = tipoSensor.createNewEntity();
2 AdapterFactory af = AdapterFactory.getInstance("JsonMap.json");
3 Object sensorAdaptado = af.generate(sensor);
```

A Listagem 5.23 apresenta um teste para ler a anotações do objeto adaptado.

Listagem 5.23 - leitura das anotações do objeto adaptado

```
1   public static void printClass(Object obj) {
2    for (Method m : obj.getClass().getMethods()) {
3         for (Annotation annotation : m.getAnnotations()) {
4            System.out.println(" anotacao: " + annotation + " do metodo: " + m.getName());
5       }
6    }
7 }
```

Como resultado é realizado o mapeamento do objeto adaptado.

A Listagem 5.24 apresenta a leitura das anotações do objeto adaptado.

Listagem 5.24 - log de leitura das anotações do objeto adaptado

```
1   anotacao: @org.springframework.web.bind.annotation.RequestMapping(path=[], headers=[], method=[],
        name=, produces=[], params=[], value=[/magnetometro/conv2fah], consumes=[]) do metodo:
        converteCelsiusToFahrenheit
```

6 AVALIAÇÃO E RESULTADOS

A avaliação das funcionalidades criadas no framework Esfinge AOM Role Mapper foi realizada através de um estudo de caso baseado em necessidades do INPE relacionadas à criação de serviços web em tempo de execução.

Esse capítulo descreve este estudo de caso, cujo foco foi verificar como a criação de serviços web em tempo de execução, é realizada através da adição de comportamento em entidades AOM pelo framework Esfinge AOM Role Mapper com o reaproveitamento de funcionalidade do framework Spring, criado para funcionar com classes estáticas. A partir do comportamento inserido nas entidades, será mostrado que é possível utilizar a funcionalidade do framework de gerar serviços web sem que exista um acoplamento direto entre o código que trabalha com as entidades AOM e o framework utilizado para gerar os serviços.

6.1 Contexto

Este estudo de caso não foi aplicado em um projeto real do INPE, pois o que foi desenvolvido no AOM RoleMapper ainda é considerado uma prova de conceito. Ou seja, faltam funcionalidades importantes no framework AOM tais como persistência relacional e herança nas entidades para permitir sua utilização em projetos reais do INPE. No entanto, o presente estudo de caso foi baseado em necessidades comuns presentes em softwares desenvolvidos pelo INPE. Uma das necessidades identificadas refere-se a sistemas que coletam dados utilizando sensores. Muitos destes equipamentos podem necessitar de ajustes nos dados obtidos devido a diferentes fabricantes ou ao tipo de informação coletada. Por exemplo, pode-se necessitar aplicar fórmulas nos dados para disponibilizar pelo sistema informações necessárias às análises dos especialistas.

Nesse cenário, dados coletados por sensores diferentes podem utilizar unidades de medida diferentes, tornando-se necessário aplicar algoritmos de conversão. Dentro deste contexto, destacam-se três importantes requisitos que podem ser atendidos com aplicação da abordagem proposta por esse trabalho de pesquisa:

a) A aplicação deve permitir a utilização de fórmulas para cálculos de valores que serão retornados por serviços web a partir dos dados obtidos pelos sensores.

b) A aplicação deve permitir que dados retornados por serviços web possam utilizar algoritmos para conversão de valores de diferentes unidades de

medida.

c) Serviços para prover informações de novos sensores e novas fórmulas para cálculos e conversões devem poder ser adicionados e modificados pelos usuários através de configurações em tempo de execução.

Como escopo desse estudo de caso, foi desenvolvida uma solução para criação de serviços web feito pelo framework Spring a partir de métodos de classes e anotações. Os novos comportamentos podem ser adicionados em entidades AOM através dos Rule Objects e com a geração dos adaptadores para classes estáticas é possível utilizar a funcionalidade do framework, gerando serviços web em tempo de execução para novos Rule Objects adicionados nas entidades.

Para o desenvolvimento do estudo de caso, foi escolhido o framework Spring (FRAMEWORK SPRING, 2018) por se tratar de um framework baseado em reflexão e metadados que invoca funcionalidade nas classes, possuir licença open-source e ser largamente utilizado pela comunidade de desenvolvedores de software para criar e disponibilizar serviços web.

6.2 Objetivo e Questões de Pesquisa

Seguindo Meta-Pergunta-Métrica-GQM (BASILI; ROMBACH, 1988), o objetivo deste estudo de caso é:

- Analisar o emprego do framework AOM Role Mapper com o propósito de utilizar frameworks baseados em reflexão e metadados para aplicações AOM, no que diz respeito à invocação de comportamento de entidades AOM, do ponto de vista do pesquisador, no contexto de aplicações que disponibilizam os serviços web que devem ser criados e modificados dinamicamente.

Para atingir esse objetivo buscou-se responder às seguintes questões de pesquisa:

a) Q1 – É possível utilizar funcionalidades de frameworks que invocam métodos por reflexão para entidades AOM, modificando o comportamento a partir de alterações na entidade?

b) Q2 – É possível desacoplar o código que gera e manipula as entidades AOM do código que invoca as funcionalidades do framework que está sendo utilizado?

Para responder a questão de pesquisa Q1, foi utilizado o framework Spring (FRAMEWORK SPRING, 2018), que gera serviços web a partir da leitura de metadados de métodos de classes estáticas por reflexão. No estudo de caso, o comportamento adicionado nas entidades AOM em tempo de execução através dos Rule Objects será disponibilizado como serviço web a partir dos adaptadores. Como resultado, espera-se verificar que os adaptadores criados podem ser identificados pelo framework, mostrando que é possível a utilização de suas funcionalidades.

Para responder a questão de pesquisa Q2, a partir do código criado no estudo de caso, será feita uma análise de dependências utilizando o diagrama DSM (YASSINE, 2004) focando nas interações com os frameworks Spring e AOM RoleMapper.

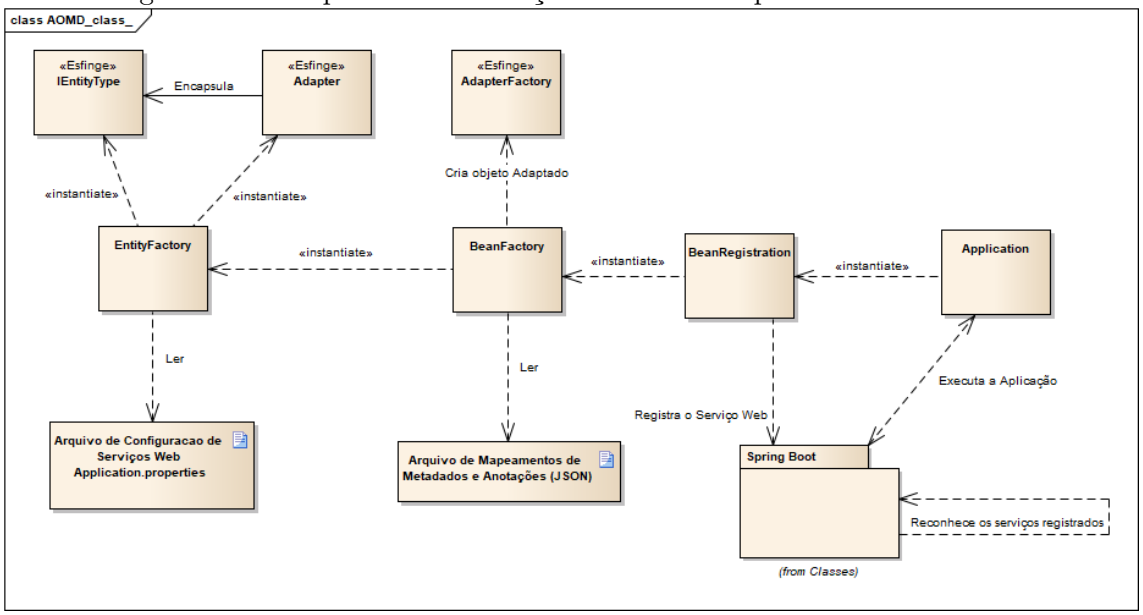
Como resultado, espera-se que não exista dependências entre as classes que manipulam as entidades AOM e as que interagem com o Spring. Isso irá mostrar que a solução desenvolvida não é exclusiva para o framework Spring e que ele não precisou de adaptações em sua estrutura para o processamento das entidades AOM.

6.3 Procedimento de coleta de dados

Para este estudo de caso, foi criado um projeto open source, que está disponível em https://github.com/antoniodiasabc/dynamicwebservice, com a implementação deste estudo de caso.

A Figura 6.1 apresenta a arquitetura proposta para este estudo de caso.

Figura 6.1 - Arquitetura da solução desenvolvida para o estudo de caso



Como mostrado na Figura 6.1, a classe Application executa a aplicação e instancia a classe BeanRegistration, que é responsável por registrar os objetos criados com a arquitetura AOM. Ela instancia a classe BeanFactory onde serão criados os objetos adaptados do AOM. A classe BeanFactory instancia a classe EntityFactory que, de acordo com o tipo de configuração do tipo de entidade AOM que pode ser do tipo EntityType ou AdapterEntytiType, cria a entidade AOM. Em seguida, retorna este objeto criado para a classe BeanFactory que cria o objeto adaptado com o método de comportamento obtido do RuleObject.

O framework AOM RoleMapper utiliza as propriedades contidas no próprio RuleObject para fazer o mapeamento dos metadados em anotações, durante a geração do adaptador. Em seguida, o BeanRegistration registra este objeto no Spring. O Spring reconhece o objeto por meio da anotação @RequestController e do método mapeado com a anotação @RequestMapping, disponibilizando o novo serviço web para atender as requisições dos clientes por meio do protocolo HTTP.

A listagem 6.1 apresenta o arquivo JSON com dois mapeamentos utilizados para as anotações dos objetos no Spring. Na linha 3 tem a chave de nome restcontroller. Ele mapeia a anotação RestController, que serve para anotar classes que serão utilizadas como controladores de requisições HTTP. Na linha 7 tem a chave nomeendpoint que mapeia a anotação RequestMapping, que serve para anotar métodos que serão

utilizados como serviços web.

Listagem 6.1 - Arquivo JSON com o mapeamento das anotações do framework Spring

```
{
[
 "restcontroller":[
   {"target":"class"},
  {"annotationPath":"org.springframework.web.bind.annotation.RestController"}
 ],
 "nomeendpoint":[
   {"target":"method"},
  {"annotationPath":"org.springframework.web.bind.annotation.RequestMapping"},
  {"parameter_1": "value"}
 ]
}
```

6.4 Procedimento de análise

Para avaliar a parte funcional relacionada com o mapeamento do modelo comportamental (Q1) utilizou-se cenários de teste baseados em (KAZMAN et al., 2000), que contempla a avaliação da arquitetura de um software baseada em cenários. Para os vários cenários possíveis de criação de serviços web em tempo de execução, foram desenvolvidos testes com entidades com comportamento adicionado em tempo de execução que utilizam ou não parâmetros de execução, que executam cálculos de fórmulas com Expression Language e que fazem a criação de novos serviços web. A tabela 6.1 apresenta os cenários de testes criados.

Tabela 6.1 - Cenários de teste do estudo de caso de criação de serviços web em tempo de execução

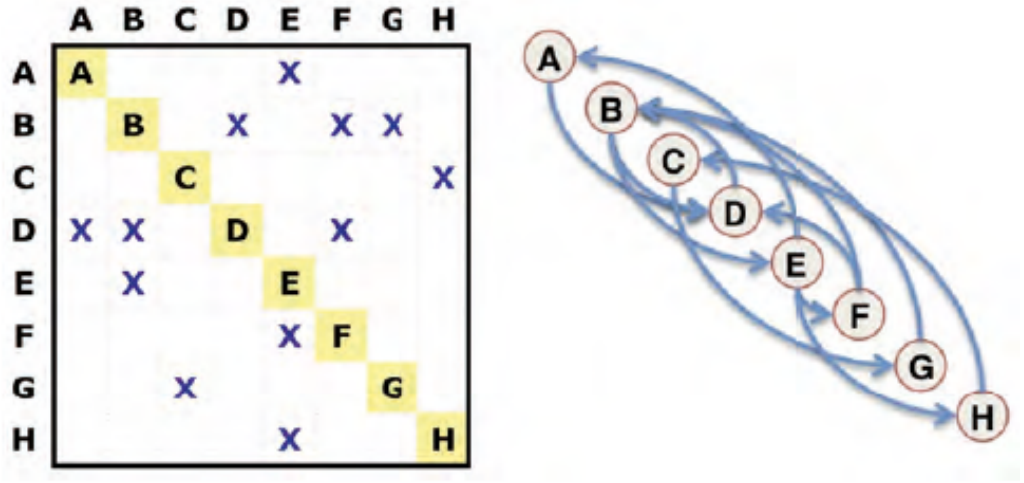
Número	Nome do teste	Objetivo
1	testMapDynamicMethod	Cria serviço web que retorna uma propriedade da entidade
2	testFormula	Utiliza uma formula configurada para retornar resultado de cálculo
3	testParametros	Utiliza parâmetros para fazer o cálculo de volume de um cilindro
4	testConversaoCelsiusToFah	Cria serviço web para a conversão de graus Celsius em Fahreinheit
5	testInsereNovoServicoWeb	Cria novo serviço web por interface gráfica utilizando parâmetros.

Para avaliar a parte de acoplamento (Q2) utilizou-se o diagrama DSM (Design Structure Matrix) (YASSINE, 2004). DSM é uma matriz quadrada, onde as células ao longo da diagonal representam os elementos de um sistema e as células fora da diagonal representam as relações entre aqueles elementos. O DSM é utilizado para avaliar o acoplamento entre componentes de software em muitos trabalhos de pesquisa recentes, por exemplo trabalhos como (SANAEI et al., 2015) e (BENKOCZI et al., 2018).

A Figura 6.2 apresenta um exemplo de DSM. Neste exemplo, o DSM do lado esquerdo da Figura 6.2 modela oito elementos, rotulados de A até H. As marcas X nas células fora da diagonal indicam uma relação direta entre o elemento na coluna com o elemento da linha. Deste modo, olhando para linha são reveladas as fontes de saídas do elemento e olhando para coluna mostra as fontes de entradas do elemento.

Estas relações também podem ser vistas no diagrama de nó com as conexões equivalente (ou grafo orientado) mostrado no lado direito da Figura 6.2.

Figura 6.2 - Exemplo de DSM com oito elementos



Fonte: Browning (2013)

Neste trabalho, o DSM apresentará a matriz de dependências entre as classes criadas neste estudo de caso e os frameworks Esfinge AOM Role Mapper e Spring.

6.5 Execução do Estudo [Resultados]

Como descrito na Tabela 6.1, foram desenvolvidos cinco cenários de teste para avaliar a criação de serviços web a partir do comportamento adicionado em tempo de execução em uma entidade AOM.

O testes desenvolvidos utilizam as funcionalidades de criar adaptadores para entidades AOM do framework Esfinge AOM Role Mapper. Essas entidades possuem metadados ligados aos Rule Objects que são mapeados para gerar as anotações do framework Spring no adaptador. Dessa forma, o framework deverá reconhecer as anotações para criar os serviços web, que irão invocar os Rule Objects para respon-

der as requisições dos clientes.

As entidades e os tipos de entidade são criados a partir de um arquivo de configuração, de nome application.properties. Um serviço do framework Spring faz a leitura das propriedades deste arquivo para criar a entidade a partir do modelo independente de domínio do AOM RoleMapper, encapsula a entidade no adaptador com API de classes estáticas e a partir da leitura das anotações da classe por reflexão disponibiliza seus métodos como serviços web.

O nome do serviço web que atende por esta requisição também é informado no arquivo de configuração. Ele também é um metadado que é transformado em anotação e lido pelo framework Spring. Após estes mapeamentos, são inseridos os valores nas propriedades da entidade que são utilizados na execução do método.

Também foi desenvolvida uma interface gráfica para inserir novos serviços web na aplicação. Por meio desta interface, podem ser manipulados os metadados da entidade para criar um novo serviço web em tempo de execução. Deste modo, para criar um novo serviço web ou alterar um serviço existente basta alterar os parâmetros no arquivo de configuração ou na tela de entrada de dados.

6.6 Descrição e Execução dos Cenários de Teste

A seguir são apresentados os resultados para cada cenário de teste.

6.6.1 Criando Serviço Web Dinâmico (Cenário 1/5)

O primeiro cenário tem o objetivo de testar a criação de um serviço sem utilizar parâmetros, a partir de um Rule Object adicionado em tempo de execução na entidade.

Procedimento de Teste

Para executar o testes os seguintes passos serão executados:

a) Alterar a configuração do arquivo application.properties;

b) Executar o teste de unidade que executa o Spring:
   - Criar o serviço web;
   - Publicar o serviço web;
   - Executar o cliente que faz a requisição ao serviço web;
   - Obter a resposta a requisição;

- Comparar o resultado com o valor esperado e para a execução do Spring.

c) Verificar o log do Spring para confirmar que o serviço foi corretamente mapeado.

Configuração

A Listagem 6.2 apresenta a configuração utilizada para criar a entidade que tem método com comportamento mapeado para criar o serviço web de nome magnetometro/intensidade para executar o primeiro cenário de teste.

Listagem 6.2 - Arquivo de Configuração para cenário 1

```
1 entitytype.name = Magnetometro
2 entitytype.properties = direcao;sentido;intensidade
3 entitytype.ruleobject.rulename = retornaIntensidade
4 entitytype.ruleobject.class = org.inpe.RetornaIntensidade
5 entitytype.ruleobject.metadata.nomeendpoint = magnetometro/intensidade
6 entity.direcao.param = 350
7 entity.sentido = N
8 entity.intensidade = 50
```

Código para avaliação

A listagem 6.3 apresenta o teste desenvolvido para este cenário. Na linha 3 do teste é invocado o método que inicia o processo de criação do tipo de entidade AOM, cria a entidade AOM em seguida, faz o mapeamento do comportamento na entidade e faz o mapeamento do objeto com as anotações do Spring para tornar-se um serviço web. Na linha 5 é realizada a execução de um cliente para fazer a requisição ao serviço web /magnetometro/intensidade, e após obter a resposta da requisição, compara ele com o valor esperado.

Listagem 6.3 - Criando objeto mapeando diretamente o método dinâmico como serviço web

```
1  @Test
2  public void testMapDynamicMethod() {
3     beanRegistration.handleResults2("magnetometro");
4     try {
5        this.mockMvc.perform(get("/magnetometro/intensidade")).andDo(print()).andExpect(status().isOk())
6              .andExpect(content().string("50"));
7     } catch (Exception e) {
8        e.printStackTrace();
9     }
10 }
```

Resultado

Como resultado, durante a execução do Spring, o log apresenta o reconhecimento do serviço web magnetometro/intensidade, conforme a listagem 6.4. O teste foi executado com sucesso e retornou o resultado esperado.

Listagem 6.4 - Log de Execução do Spring mapeando o método executeOperation

```
RequestMappingHandlerMapping : Mapped "{[/magnetometro/intensidade]}" onto public java.lang.Object
    magnetometroAOMBeanAdapter.retornaIntensidade(java.lang.String,java.lang.Object...) throws
    org.esfinge.aom.exceptions.EsfingeAOMException
```

6.6.2 Criação de Serviço Web com Fórmula (Cenário 2/5)

O segundo cenário tem o objetivo de testar a criação de um serviço web utilizando uma fórmula para o cálculo de mediana de um Sensor. Na criação deste serviço web com fórmula, será executada a fórmula (medidaMinima+medidaMaxima) / 2 .

Procedimento de Teste

Para executar o testes os seguintes passos serão executados:

a) Alterar a configuração do arquivo application.properties;

b) Executar o teste de unidade que executa o Spring:
- Criar o serviço web;
- Publicar o serviço web;
- Executar o cliente que faz a requisição ao serviço web;
- Obter a resposta a requisição;
- Comparar o resultado com o valor esperado e para a execução do Spring.

c) Verificar o log do Spring para confirmar que o serviço foi corretamente mapeado.

Configuração

Na listagem 6.5 apresenta a configuração utilizada para criar a entidade que tem método com comportamento mapeado para criar o serviço web de nome magnetometro/formulamediana para executar o segundo cenário de teste. .

Listagem 6.5 - Configuração para criação de serviço web com fórmula

```
entitytype.name=Magnetometro
entitytype.properties = medidaMinima,medidaMaxima
entitytype.ruleobject.rulename = calculaMediana
entitytype.ruleobject.formula=${ (medidaMinima+medidaMaxima) / 2 }
```

```
entitytype.ruleobject.metadata.nomeendpoint = magnetometro/formulamediana
entity.medidaMaxima = 50
entity.medidaMinima = 9
```

Código para avaliação

Na listagem 6.6 apresenta o teste desenvolvido para este cenário. Na linha 3 do teste é invocado o método que inicia o processo de criação do tipo de entidade AOM, cria a entidade AOM em seguida, faz o mapeamento do comportamento na entidade e faz o mapeamento do objeto com as anotações do Spring para tornar-se um serviço web. Na linha 5 é realizada a execução de um cliente para fazer a requisição ao serviço web /magnetometro/formulamediana e após obter a resposta da requisição, compara ele com o valor esperado.

Listagem 6.6 - Teste com serviço web com Formula

```
1  @Test
2  public void testFormula() {
3      beanRegistration.handleResults2("magnetometro");
4     try {
5        this.mockMvc.perform(get("/magnetometro/formulamediana")).andDo(print()).andExpect(status().isOk())
6              .andExpect(content().string("29.5"));
7     } catch (Exception e) {
8        e.printStackTrace();
9        assertTrue(false);
10    }
11 }
```

Resultado

Com esta configuração, o Spring ao ser executado faz o reconhecimento do serviço web, conforme o trecho do log de execução apresentado na listagem 6.7. O teste foi executado com sucesso e retornou o resultado esperado.

Listagem 6.7 - Log de execução do Spring reconhecendo o serviço web testeFormula

```
1  RequestMappingHandlerMapping : Mapped "{[/magnetometro/formulamediana]}" onto public
       java.lang.Object
       magnetometroAOMBeanAdapter.calculaMediana(java.lang.String,java.lang.Object...) throws
       org.esfinge.aom.exceptions.EsfingeAOMException
```

6.6.3 Serviços Web Com Parâmetros (Cenário 3/5)

O terceiro cenário tem o objetivo de testar a criação de um serviço web com parâmetros de execução. Este teste realiza o cálculo de volume de um Sensor.

Procedimento de Teste

Para executar o testes os seguintes passos serão executados:

a) Alterar a configuração do arquivo application.properties;

b) Executar o teste de unidade que executa o Spring:
- Criar o serviço web;
- Publicar o serviço web;
- Executar o cliente que faz a requisição ao serviço web;
- Obter a resposta a requisição;
- Comparar o resultado com o valor esperado e para a execução do Spring.

c) Verificar o log do Spring para confirmar que o serviço foi corretamente mapeado.

Configuração

A listagem 6.8 apresenta a configuração para a criação do serviço web com parâmetros, de nome sensor/volume. O raio do cilindro, na linha 6, tem o valor 10. O parâmetro altura, que será utilizado no cálculo de volume do cilindro, será informado na execução do serviço web .

Listagem 6.8 - Configuração de serviço web dinâmico com parâmetros

```
1 entitytype.name = Sensor
2 entitytype.properties = raio
3 entitytype.ruleobject.rulename = calcularVolume
4 entitytype.ruleobject.class = org.inpe.CalculaVolume
5 entitytype.ruleobject.metadata.nomeendpoint = sensor/volume
6 entity.raio.param = 10
```

Código para avaliação

A listagem 6.9 apresenta o teste para validar a criação de serviço web com RuleObject com parâmetros. Na linha 3 do teste é invocado o método que inicia o processo de criação do tipo de entidade AOM, cria a entidade AOM em seguida, faz o mapeamento do comportamento na entidade e faz o mapeamento do objeto com as anotações do Spring para tornar-se um serviço web. Na linha 5 é realizada a execução de um cliente para fazer a requisição ao serviço web /sensor/volume, passando por parâmetro a altura do cilindro, com valor 20. Após obter a resposta da requisição, compara ele com o valor esperado.

Listagem 6.9 - Teste de serviço web com parâmetros

```
@Test
public void testParametros() {
    beanRegistration.handleResults2("sensor");
   try {
      this.mockMvc.perform(get("/sensor/volume?altura=20")).andDo(print()).andExpect(status().isOk())
           .andExpect(content().string("6283.185307179587 cm3"));
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

Resultado

Com esta configuração, o Spring ao ser executado faz o reconhecimento do serviço web, conforme o trecho do log de execução apresentado na listagem 6.10. O teste foi executado com sucesso e retornou o resultado esperado.

Listagem 6.10 - Log de execução do Spring reconhecendo o serviço web testeParametros

```
RequestMappingHandlerMapping : Mapped "{[/sensor/volume]}" onto public java.lang.Object
    sensorAOMBeanAdapter.calcularVolume(java.lang.String,java.lang.Object...) throws
    org.esfinge.aom.exceptions.EsfingeAOMException
```

6.6.4 Conversão de unidade de medida de um Sensor (Cenário 4/5)

O quarto cenário tem o objetivo de testar a criação de um serviço web que faz conversão de um dado de Sensor de graus Celsius para Fahreinheit.

Procedimento de Teste

Para executar o testes os seguintes passos serão executados:

a) Alterar a configuração do arquivo application.properties;

b) Executar o teste de unidade que executa o Spring:
- Criar o serviço web;
- Publicar o serviço web;
- Executar o cliente que faz a requisição ao serviço web;
- Obter a resposta a requisição;
- Comparar o resultado com o valor esperado e para a execução do Spring.

c) Verificar o log do Spring para confirmar que o serviço foi corretamente mapeado.

Configuração

A configuração da criação do serviço web para fazer a conversão de graus Celsius para Fahreinheit é apresentada na Listagem 6.11. O nome do serviço web é magnetometroAdaptado/conv2fah. A principal diferença deste serviço web para os demais é que ele é criado utilizando o adaptador de entidade específica de domínio.

Listagem 6.11 - Configuração para criação de serviço web com adaptador de entidade

```
1 entitytype.name = Magnetometro
2 entitytype.properties = grausCelsius
3 entitytype.ruleobject.rulename = converteCToFahAdapt
4 entitytype.ruleobject.class = org.inpe.ConverteCelsiusToFah
5 entitytype.ruleobject.metadata.nomeendpoint = magnetometroAdaptado/conv2fah
6 entity.grausCelsius.param = 10
```

Código para avaliação

Para validar esta funcionalidade foi criado o teste apresentado na listagem 6.12. Na linha 3 do teste é invocado o método que inicia o processo de criação do tipo de entidade AOM, cria a entidade AOM em seguida, faz o mapeamento do comportamento na entidade e faz o mapeamento do objeto com as anotações do Spring para tornar-se um serviço web. Na linha 5 é realizada a execução de um cliente para fazer a requisição ao serviço web magnetometroAdaptado/conv2fah, e após obter a resposta da requisição, compara ele com o valor esperado de 50 graus FahreinHeit que equivale aos 10 graus Celsius informado como parâmetro da entidade.

Listagem 6.12 - Teste para criação de serviço web para conversão de dados de Sensor

```
1  @Test
2  public void testConversaoCelsiusToFah() {
3      beanRegistration.handleResults2("magnetometro");
4     try {
5        this.mockMvc.perform(get("/magnetometroAdaptado/conv2fah")).andDo(print()).andExpect(status().isOk())
6              .andExpect(content().string("50.0"));
7     } catch (Exception e) {
8        e.printStackTrace();
9     }
10 }
```

Resultado

Com esta configuração, o Spring ao ser executado faz o reconhecimento do serviço web, conforme o trecho do log de execução apresentado na listagem 6.13. O teste foi executado com sucesso e retornou o resultado esperado.

Listagem 6.13 - Log de execução do Spring reconhecendo o serviço web /magnetometroAdaptado/conv2fah

```
RequestMappingHandlerMapping : Mapped "{[/magnetometroAdaptado/conv2fah]}" onto public
    java.lang.Object
    magnetometroAdaptadoAOMBeanAdapter.converteCToFahAdapt(java.lang.String,java.lang.Object...)
    throws org.esfinge.aom.exceptions.EsfingeAOMException
```

6.6.5 Adicionar um novo Serviço Web (Cenário 5/5)

O quinto cenário tem o objetivo de testar a criação de um novo serviço web em tempo de execução, inserido por meio da interface gráfica e não do arquivo de configuração como apresentado nos outros exemplos. Este novo serviço web faz a conversão de uma unidade medida de graus Fahreinheit para Kelvin.

Para adicionar um novo serviço web, é necessário que sejam fornecidos os dados do nome serviço web, o nome da entidade que vai ser criada, o nome da regra que vai ser executada ou uma fórmula e os nome dos atributos da entidade criada e os seus respectivos valores. Esta criação pode ser feita alterando diretamente o arquivo de configuração application.properties ou inserindo estes dados na tela de entrada de dados criada para este uso.

Após inserção dos dados do novo serviço web, o Spring detecta a alteração e recria todos os objetos e refaz o mapeamento de todas as entidades. Deste modo ele reconstrói o seu contexto e os serviços web estão prontos para receber novas requisições.

Procedimento de Teste

a) Executar o teste unitário que executa o Spring.
   - Invocar o serviço web de nome insere que envia os parâmetros do novo serviço web para alterar o arquivo application.properties;
   - O Spring detecta que houve alteração do arquivo de configuração, cria o serviço web, publica o serviço web, e após esperar alguns segundos para recarregar o contexto do Spring;
   - Executar o cliente que faz a requisição ao serviço web;
   - Obter a resposta a requisição;
   - Comparar o resultado com o valor esperado e para a execução do Spring.

b) Verificar o log do Spring para confirmar que o serviço foi corretamente mapeado.

Configuração

A configuração deste teste é feita diretamente nos parâmetros do teste de criação do serviço web de nome magnetometro/converteF2K. As configurações são apresentadas na listagem 6.14

Listagem 6.14 - Parâmetros para inserção de novo serviço web

```
.param("nome", "magnetometro/converteF2K")
.param("rulename", "converteFahToKelvin")
.param("regra", "org.inpe.ConverteFahToKelvin")
.param("data", "50")
.param("index", "30"))
```

A tela para inserir os parâmetros para criar um novo serviço web é apresentada na figura 6.3.

Figura 6.3 - Tela de inserção de parâmetros para a criação de um serviço web

Código para avaliação

Para testar esta funcionalidade foi desenvolvido o teste apresentado na listagem 6.15 .

Listagem 6.15 - Teste de inserção de novo serviço web

```
@Test
public void testInsereNovoServicoWeb() {
    try {
      this.mockMvc.perform(get("/insere")
            .param("nome", "magnetometro/converteF2K")
            .param("regra", "org.inpe.ConverteFahToKelvin")
            .param("rulename", "converteFahToKelvin")
            .param("data", "50")
            .param("index", "30")).andDo(print()).andExpect(status().isOk());
      Thread.sleep(10000);

      this.mockMvc.perform(get("/magnetometro/converteF2K")).andDo(print()).andExpect(status().isOk())
      .andExpect(content().string("283.15"));
    } catch (Exception e) {
      e.printStackTrace();
```

```
16      }
17  }
```

Na primeira parte do teste, da linha 5 a 9 são enviados os parâmetros, com o nome do serviço web, a classe de comportamento que será executada, o nome do método que será mapeado e os parâmetros da regra. Na linha 10 é feito um sleep para que o Spring reconheça a alteração do arquivo application.properties e recarregue o seu contexto com o novo serviço web. Na linha 12 é feito o teste da requisição do serviço web /magnetometro/converteF2K. Na linha 13 é verificada a resposta da requisição, para confirmar que o serviço web foi criado e retornou o resultado esperado.

Resultado

Com esta configuração, o Spring ao ser executado faz o reconhecimento do serviço web magnetometro/converteF2K, conforme o trecho do log de execução apresentado na listagem 6.16. O teste foi executado com sucesso e retornou o resultado esperado.

Listagem 6.16 - Log de execução do Spring reconhecendo o serviço web magnetometro/converteF2K

```
1  RequestMappingHandlerMapping : Mapped "{[/magnetometro/converteF2K]}" onto public java.lang.Object
       magnetometroAOMBeanAdapter.converteFahToKelvin(java.lang.String,java.lang.Object...) throws
       org.esfinge.aom.exceptions.EsfingeAOMException
```

6.7 Análise de Modularidade

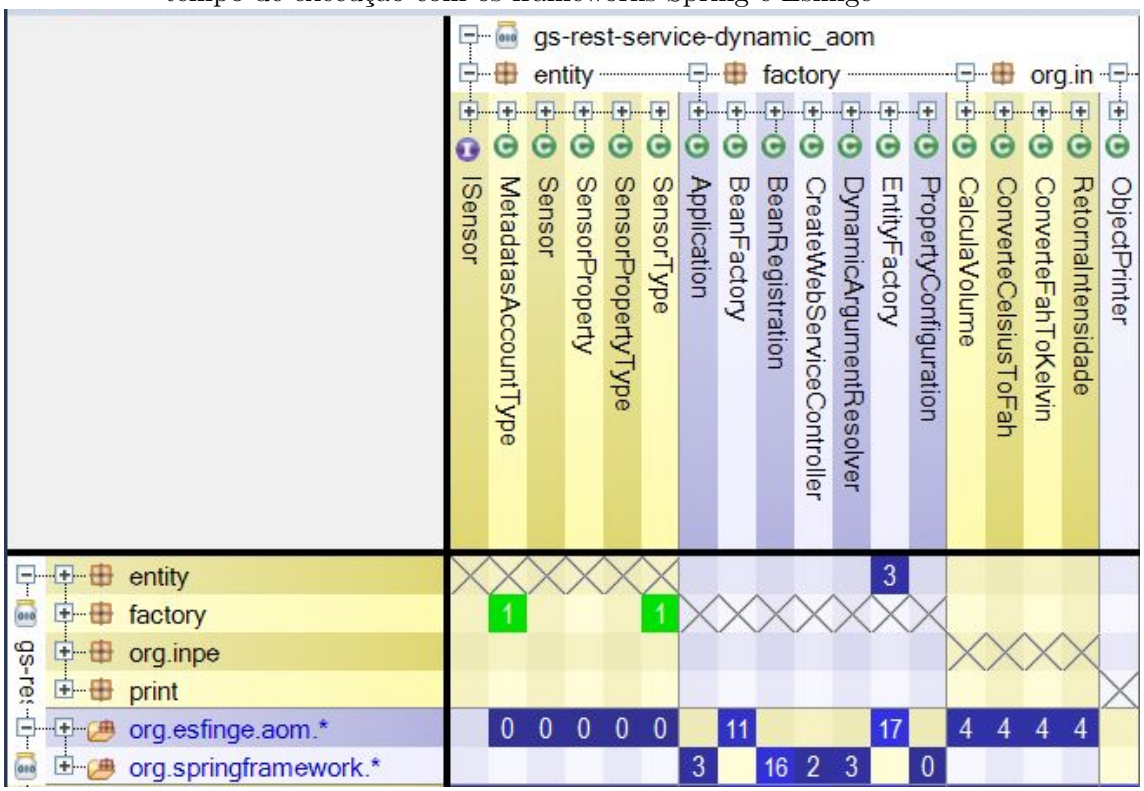
Conforme apresentado nas seções anteriores, foi possível realizar o mapeamento de um serviço web para o Rule Object a partir do adaptador gerado pelo AOM Role Mapper. O objetivo dessa seção é apresentar uma análise do acoplamento do código desenvolvido na solução do estudo de caso com os frameworks Spring e AOM RoleMapper.

A análise do acoplamento entre as classes desenvolvidas para realizar esta atividade foi feita utilizando a técnica Design Structure Matrix (DSM) (EPPINGER; BROWNING, 2012) gerado pela ferramenta JArchitect.

A DSM gerada compara as classes criadas no estudo de caso para criação dos serviços web e os frameworks AOM RoleMapper e Spring. O objetivo é verificar o baixo acoplamento entre estas classes e os frameworks utilizados, confirmando que nenhuma classe do estudo de caso depende ao mesmo tempo do framework Spring e do framework Esfinge AOM Role Mapper.

A figura 6.4 apresenta a DSM entre as classes desenvolvidas no estudo de caso e os dois frameworks utilizados.

Figura 6.4 - Dependência das classes do estudo de caso para criação de serviços web em tempo de execução com os frameworks Spring e Esfinge



Analisando as dependências entre as classes do estudo de caso com o framework AOM Role Mapper, pode-se verificar 17 dependências entre a classe EntityFactory e o framework Esfinge AOM Role Mapper. A classe BeanFactory apresenta 11 dependências com o framework Esfinge AOM Role Mapper. O importante é verificar que nenhuma das duas classes que apresentam dependências com o framework Spring.

Analisando as classes do estudo de caso que interagem com o framework Spring, verifica-se que a classe Application têm 3 dependências com o framework Spring. A classe BeanRegistration têm 16 dependências do framework Spring. A classe CreateWebServiceController têm 2 dependências e a classe DynamicArgumentResolver têm 3 dependências. Nenhumas delas têm dependência com o AOM Role Mapper.

As classes de regras utilizadas nos cenários de testes apresentam 4 dependências

cada uma com o framework AOM RoleMapper. Esta dependência é com a classe BasicRuleObject que as tornam um RuleObject.

Partindo da evidência que no estudo de caso nenhuma classe depende dos dois frameworks, chegamos a conclusão que é possível modularizar o uso do framework AOM RoleMapper e do framework Spring. Sendo assim, o uso do Spring é feito de forma transparente ao fato dele estar, na verdade, invocando a lógica de uma entidade AOM.

6.8 Análise dos Resultados

A análise dos resultados deste estudo de caso será realizada respondendo as perguntas de pesquisa.

6.8.1 Utilização de Frameworks Tradicionais (Q1)

Q1: *É possível utilizar funcionalidades de frameworks que invocam métodos por reflexão para entidades AOM, modificando o comportamento a partir de alterações na entidade?*

Baseado no fato que nos cinco cenários de teste apresentados foram utilizadas funcionalidades do framework Spring utilizando a geração de adaptadores do AOM Role Mapper após inserir o comportamento na entidade AOM, é possível afirmar que essa utilização é possível com a abordagem proposta. Os serviços web foram criados e disponibilizados a partir do comportamento adicionado em tempo de execução, funcionou como esperado em todos os cenários de testes apresentados.

6.8.2 Acoplamento entre Componentes (Q2)

Q2: *É possível desacoplar o código que gera e manipula as entidades AOM do código que invoca as funcionalidades do framework que está sendo utilizado?*

O DSM mostrou que o código que trabalha com o framework AOM RoleMapper não é acoplado com Spring e vice-versa. As classes que têm dependência com um framework não têm dependência com outro. Deste modo, pode-se concluir que é possível ter um baixo acoplamento entre os frameworks AOM e os frameworks que trabalham com reflexão e classes estáticas utilizando a abordagem proposta. Isso mostra que da forma como a solução foi estruturada, o framework Spring é utilizado nas entidades encapsuladas sem depender de detalhes do framework AOM utilizado, assim como o código que lida com framework AOM RoleMapper para preparar o

adaptador da entidade também não depende diretamente de classes do Spring.

6.9 Limitações

Os cenários desenvolvidos para avaliar o criação de serviços web a partir de Rule Objects inseridos em tempo de execução apresentam algumas limitações. Essa seção apresenta essas limitações e sugere como podem ser endereçadas em trabalhos futuros.

Uma limitação do estudo de caso é o fato de que ele foi desenvolvido utilizando apenas o framework Spring. Apesar dele ser amplamente utilizado no mercado de desenvolvimento de sistemas web, é possível que outros frameworks possuam detalhes que não sejam implementados pelo estágio atual da solução. Um trabalho futuro nesse caso, seria ampliar o estudo de caso para outros frameworks.

Outra limitação está ligada a varredura de mapeamentos que o framework Spring realiza apenas durante a carga de seu contexto de execução. Como consequência desta limitação, toda vez que é feito uma alteração da configuração de um serviço web, é necessário recarregar o seu contexto. Uma melhoria para esta situação seria fazer uma alteração no próprio framework Spring para que ele aceite o registro de novos serviços web a qualquer momento da sua execução, sem a necessidade recarregar o contexto. Porém essa alteração seria fora do escopo desse estudo de caso, que tinha o objetivo justamente de avaliar a utilização do framework sem alterações.

Também não foi incluído neste estudo de caso um cenário para o mapeamento dependente de domínio para independente de domínio utilizando as anotações do AOM Role Mapper. Este cenário pode ser incluído em um trabalho futuro.

7 CONCLUSÕES

Esse trabalho foca em uma solução para a necessidade de flexibilidade no desenvolvimento de sistemas nos quais não é possível ter todos os requisitos na fase inicial do projeto. A frequente mudança de regras normalmente implica em horas de desenvolvimento para fazer as alterações no código, para que seu comportamento atenda os novos requisitos. O estilo arquitetural AOM (YODER et al., 2001) é uma abordagem para resolver este problema. Contudo, as entidades AOM tem uma estrutura diferente de objetos comuns, não sendo possível reutilizá-la com os frameworks de padrão de mercado, que trabalham com anotações e classes estáticas.

O framework AOM RoleMapper (ESFINGE FRAMEWORK, 2018), que utiliza a arquitetura AOM, possibilita o reúso de frameworks tradicionais em entidades AOM por meio da criação de adaptadores que encapsulam esta entidade. Porém este framework não havia implementado o modelo de comportamento para a arquitetura AOM, equivalente aos métodos de uma classe. Isso é importante para o reúso de frameworks que invocam comportamento nas classes, como os que geram e disponibilizam serviços web.

O objetivo deste trabalho foi definir um modelo arquitetural capaz de mapear a representação do comportamento de entidades entre modelos de classe estáticos, AOMs específicos de domínio e AOMs independentes de domínio, com a finalidade de possibilitar a reutilização frameworks que invocam métodos por reflexão em modelos de classe estáticas em arquiteturas AOM.

O primeiro passo foi a realização de um experimento para avaliar a facilidade de reúso de um framework com a abordagem do AOM RoleMapper. Foi utilizado o Hibernate Validator, um framework que acessa apenas as propriedades da entidade, para a avaliação de entidades AOM. O experimento foi realizado com alunos do INPE divididos em dois grupos com o objetivo de realizar uma atividade de implementação de uma entidade sem utilizar qualquer framework e depois utilizando o framework.

Como o resultado deste experimento, pode-se verificar que os desenvolvedores tiveram boa aceitação com framework AOM Role Mapper, sendo que mais de 85 % dos participantes responderam que preferem utilizar o framework AOM, pois apesar da curva de aprendizado, perceberam a utilidade devido a reutilização de código. Em relação ao tempo de desenvolvimento, não houve a princípio uma diferença significativa entre os participantes, apesar da abordagem proposta ser mais complexa.

A partir dessa conclusão, foram desenvolvidas novas funcionalidades no framework Esfinge AOM Role Mapper para o suporte ao modelo comportamental de um AOM. As principais funcionalidades desenvolvidas foram:

a) Suporte ao padrão de projeto Rule Object no modelo independente de domínio do framework;

b) Mapeamento do padrão Rule Object de um modelo AOM específico de domínio para o modelo AOM independente de domínio do framework;

c) Mapeamento do padrão Rule Object do modelo AOM independente de domínio do framework para métodos de classe estáticas;

d) Mapeamento de metadados nos Rule Objects do AOM para anotações nos métodos.

Para avaliar estas melhorias, foi desenvolvido um estudo de caso utilizando as funcionalidades desenvolvidas para reutilização de um framework baseado em reflexão e no uso de anotações. Neste estudo, foram criados e modificados serviços web em tempo de execução com framework Spring a partir de entidades AOM com Rule Objects. Os cenários de teste foram baseados em requisitos comuns em sistemas do INPE, como execução de cálculos por meio de fórmulas e conversão de unidades de medida de um Sensor.

O estudo de caso utilizou a metodologia ATAM (KAZMAN et al., 2000), onde foram criados cinco cenários arquiteturais com o objetivo de verificar que é possível reutilizar funcionalidades de frameworks que invocam métodos por reflexão para entidades AOM, modificando o comportamento a partir de mudanças nessa entidade.

Outro objetivo do estudo de caso foi verificar que é possível desacoplar o código que manipula e gera as entidades AOM do código que invoca as funcionalidades do framework reutilizado. Para verificar este objetivo foi gerado uma representação DSM (YASSINE, 2004) do código fonte da aplicação do experimento verificando as dependências entre as classes dos frameworks envolvidos e o código criado no experimento. Nesta análise pode-se verificar que nenhuma classe tinha dependência dos dois frameworks ao mesmo tempo, comprovando o desacoplamento.

Uma importante conclusão do estudo de caso foi que, utilizando a solução proposta por esse trabalho é possível reutilizar as funcionalidades de frameworks que invocam

métodos por reflexão para entidades AOM, modificando o comportamento da aplicação de acordo com mudanças na entidade em tempo de execução. Outra conclusão é que esse objetivo pode ser atingido com o baixo acoplamento entre as classes criadas no estudo de caso e os frameworks utilizados, pois nenhuma classe do estudo de caso dependia dos dois frameworks ao mesmo tempo.

Desta forma, pode-se verificar que foi cumprido o objetivo de definir um modelo arquitetural capaz de mapear a representação do comportamento de entidades entre modelos de classe estáticas, AOMs específicos de domínio e AOMs independentes de domínio, com a finalidade de possibilitar a reutilização em arquiteturas AOM de frameworks que invocam métodos por reflexão em modelos de classes estáticas. Como evidência, no estudo de caso apresentado pode-se verificar que os cinco cenários arquiteturais conseguiram reutilizar com sucesso o framework. O desacoplamento demonstrado pela DSM, provê uma evidência de que, como o código que usa o framework tradicional não depende do framework AOM, e que a solução poderá ser generalizada.

Outros frameworks AOM, como o Oghma (FERREIRA et al., 2009) e o Ink (ACHERKAN et al., 2011), proveem um modelo independente de domínio e componentes e frameworks auxiliares baseados nesse modelo. Nenhuma das abordagens previamente existentes permitia a reutilização de frameworks tradicionais. É no suporte a esse tipo de reúso, principalmente na parte do modelo comportamental, que reside o originalidade deste trabalho.

Como descrito na introdução, no contexto INPE, este projeto é um importante passo para o projeto de uma plataforma de laboratório virtual. A ideia dessa plataforma é que usuários possuam submeter experimentos e análises, baseados em bases de dados pré-existentes, os quais seriam dinamicamente incorporados a plataforma. A solução desenvolvida poderia ser utilizada como base para a incorporação de comportamento dinâmico submetido pelos usuários na plataforma.

7.1 Contribuições

As principais contribuições deste trabalho são:

a) Experimento: O experimento realizado mostrou que os desenvolvedores percebem ganho na reutilização de frameworks tradicionais em aplicações AOM pela abordagem do AOM Role Mapper. Esse resultado motivou a extensão dessa abordagem para o modelo comportamental, com o desen-

volvimento das funcionalidades de comportamento no framework.

b) Funcionalidades do AOM Role Mapper: As funcionalidades desenvolvidas no framework AOM RoleMapper são: criação de classes que implementam o padrão de projeto Rule Object no modelo independente de domínio; o mapeamento do padrão Rule Object de um modelo AOM específico de domínio para o modelo AOM independente de domínio; o mapeamento do padrão Rule Object do modelo AOM independente de domínio do framework para métodos de classe estáticas; o mapeamento de metadados nos Rule Objects do AOM para anotações nos métodos; e a execução de fórmulas por meio de Expression Language.

c) Estudo de caso: O estudo de caso, onde foram criados serviços web com comportamento inserido em tempo de execução, através do uso do Spring e do AOM Role Mapper, gera evidência que foi alcançado o objetivo de possibilitar o reúso de frameworks tradicionais em aplicações AOM. O resultado também mostra, por meio do DSM, que o acoplamento entre as classes do estudo de caso e dos frameworks é fraco, não havendo classe que dependam de ambos.

d) Proposta de arquitetura para serviços web dinâmicos: Como um resultado secundário, a arquitetura criada no estudo de caso mostrou-se efetiva para a criação de serviços web de forma dinâmica por sua simplicidade e por ter acoplamento baixo com relação aos frameworks utilizados. Ela pode ser utilizada como base para a arquitetura com requisito de criação de serviços em tempo de execução.

7.2 Trabalhos Futuros

Dentre os trabalhos futuros, pode-se mencionar a utilização de aspectos para mapear comportamentos de um tipo de entidade com eventos de processamento.

A passagem de parâmetros para RuleObjects é uma limitação, no caso do tipo da entidade ser uma classe, pois poderá necessitar de conversão explícita.

Outra funcionalidade importante a ser implementada é a herança nas entidades AOM para que seja possível trabalhar com classes abstratas e mesmo com classes comuns. Em frameworks tradicionais, é uma prática comum a necessidade de herdar de uma classe específica do framework para compor uma funcionalidade.

A persistência em banco de dados relacional do modelo AOM precisa ser desenvolvida, armazenando tanto os objetos criado como os metadados e objetos de regra.

Realizar novo experimento com o framework AOM RoleMapper com as funcionalidades de comportamento dinâmico para verificar a utilização deste pelos desenvolvedores.

REFERÊNCIAS BIBLIOGRÁFICAS

ACHERKAN, E.; HEN-TOV, A.; LORENZ, D. H.; SCHACHTER, L. The ink language meta-metamodel for adaptive object-model frameworks. In: ACM INTERNATIONAL CONFERENCE COMPANION ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS COMPANION, 2011. **Proceedings...** [S.l.], 2011. p. 181–182. 2, 6, 28, 96

ARSANJANI, A. **A pattern language for adaptive and scalable business rule construction**, Citeseer, 2001, Disponível em https://hillside.net/plop/plop/plop2k/proceedings/Arsanjani/Arsanjani.pdf. 6

BASILI, V. R.; ROMBACH, H. D. The tame project: towards improvement-oriented software environments. **IEEE Transactions on Software Engineering**, v. 14, n. 6, p. 758–773, 1988. 47, 76

BAUER, C.; KING, G. **Hibernate in action**. [S.l.]: Manning Greenwich CT, 2005. 10

BEGUM, A.; RAJ, V. C. Architectural analysis for mobile devices with quality attributes using architecture trade-off analysis method. **Journal of Computational and Theoretical Nanoscience**, v. 15, n. 11-12, p. 3352–3358, 2018. 5

BENKOCZI, R.; GAUR, D.; HOSSAIN, S.; KHAN, M. A. A design structure matrix approach for measuring co-change-modularity of software products. In: INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR), 15., 2018. **Proceedings...** [S.l.]: IEEE, 2018. p. 331–335. 5, 79

DANTAS, A.; YODER, J. W.; BORBA, P.; JOHNSON, R. E. Using aspects to make adaptive object-models adaptable. In: ECOOP WORKSHOP ON REFLECTION, AOP AND METADATA FOR SOFTWARE EVOLUTION (RAM-SE), 2004. **Proceedings...** [S.l.], 2004. p. 9–19. 6

DOUCET, F.; SHUKLA, S.; GUPTA, R. Introspection in system-level language frameworks: Meta-level vs. integrated. In: CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE, 2003. **Proceedings...** [S.l.]: IEEE, 2003. p. 10382. 8

EPPINGER, S. D.; BROWNING, T. R. **Design structure matrix methods and applications**. [S.l.]: MIT press, 2012. 90

ESFINGE FRAMEWORK. **Esfinge Framework**. 2018. Disponível em: <http://esfinge.sf.net>. Acesso em: 20 de maio de 2018. 14, 25, 29, 94

FERREIRA, H. S.; CORREIA, F. F.; AGUIAR, A. Design for an adaptive object-model framework. In: INTERNATIONAL CONFERENCE ON MODEL DRIVEN ENGINEERING LANGUAGES AND SYSTEMS (MODELS'09), 12., 2009. **Proceedings...** [S.l.]: IEEE, 2009. 2, 6, 25, 26, 96

FOOTE B., Y. J. Evolution, architecture, and metamorphosis. In: **Pattern Languages of Program Design**. [S.l.]: Addison-Wesley, 1996. v. 2, p. 295–314. 14

FOWLER, M. **Analysis patterns: reusable object models**. [S.l.]: Addison-Wesley, 1996. 18

FRAMEWORK SPRING. **Spring Framework 5: The right stack for the right job**. 2018. Disponível em: <https://spring.io/>. Acesso em: 20 de Outubro de 2018. 3, 5, 76, 77

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design patterns: elements of reusable object-oriented software**. Boston, MA, USA: Addison-Wesley Longman, 1995. ISBN 0-201-63361-2. 1, 23

GRADECKI, J. D.; LESIECKI, N. **Mastering aspectJ: aspect-oriented programming in Java**. [S.l.]: John Wiley & Sons, 2003. 6

GUERRA, E.; FERNANDES, C.; SILVEIRA, F. F. Architectural patterns for metadata-based frameworks usage. In: CONFERENCE ON PATTERN LANGUAGES OF PROGRAMS, 17., 2010. **Proceedings...** [S.l.]: ACM, 2010. p. 4. 14

GUERRA, E.; SANTOS, J.; AGUIAR, A.; VERAS, L. G. Dynamic generated adapters from adaptive object models to static apis. In: CONFERENCE ON PATTERN LANGUAGES OF PROGRAMS, 22., 2015. **Proceedings...** [S.l.], 2015. p. 12. 2

GUERRA, E. M.; SOUZA, J. T. D.; FERNANDES, C. T. A pattern language for metadata-based frameworks. In: CONFERENCE ON PATTERN LANGUAGES OF PROGRAMS, 16., 2009. **Proceedings...** [S.l.]: ACM, 2009. p. 3. 43, 44

GUERRA EDUARDO; AGUIAR, A. Support for refactoring an application towards an adaptive object model. In: MURGANTE, B.; MISRA, S.; ROCHA, A.

M. A. C.; TORRE, C.; ROCHA, J. G.; FALCÃO, M. I.; ANIAR, D.; APDUHAN, B. O.; GERVASI, O. (Ed.). **Computational Science and Its Applications : ICCSA 2014**. Berlin: Springer, 2014. p. 73–89. 2

HEN-TOV, A.; LORENZ, D. H.; PINHASI, A.; SCHACHTER, L. Modeltalk: when everything is a domain-specific language. **IEEE Software**, v. 26, n. 4, 2009. 27

HEN-TOV, A.; NIKOLAEV, L.; SCHACHTER, L.; WIRFS-BROCK, R.; YODER, J. W. Adaptive object-model evolution patterns. In: LATIN AMERICAN CONFERENCE ON PATTERN LANGUAGES OF PROGRAMS, 8., 2010. **Proceedings...** [S.l.]: ACM, 2010. p. 5. 25, 27

HIBERNATE VALIDATOR. **Application layer agnostic validation**. 2016. Disponível em: <http://hibernate.org/validator>. Acesso em: 20 maio 2016. 46

JAVA BYTECODE MANIPULATION. **ASM**. 2017. Disponível em: <https://asm.ow2.io>. Acesso em: 20 Fevereiro 2017. 72

JOHNSON, R.; OAKES, J. **The user-defined product framework**, Citeseer, 1998. Disponível em: http://stwww. cs. uiuc. edu/users/johnson/papers/udp. 15

JOHNSON, R. E.; FOOTE, B. Designing reusable classes. **Journal of Object-Oriented Programming**, v. 1, n. 2, p. 22–35, 1988. 13

JOHNSON R., W. B. Type object. In: MARTIN, R. C.; RIEHLE, D.; BUSCHMANN, F. (Ed.). **Pattern Languages of Program Design**. [S.l.]: Addison-Wesley, 1997. v. 3, p. 47–65. 17

JOSEPH, B. F.; YODER, J. Metadata and active object-models. In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES AND APPLICATIONS, 1998. **Proceedings...** [S.l.]: ACM, 1998. 8

KAMPENES, V. B.; DYBÅ, T.; HANNAY, J. E.; SJØBERG, D. I. A systematic review of quasi-experiments in software engineering. **Information and Software Technology**, v. 51, n. 1, p. 71–82, 2009. 49

KAZMAN, R.; KLEIN, M.; CLEMENTS, P. **ATAM: method for architecture evaluation**. [S.l.: s.n.], 2000. 5, 79, 95

MAES, P. Concepts and experiments in computational reflection. **ACM Sigplan Notices**, v. 22, n. 12, p. 147–155, 1987. 8, 14

MATSUMOTO, P. M.; GUERRA, E. An architectural model for adapting domain-specific aom applications. In: BRAZILIAN SYMPOSIUM ON SOFTWARE COMPONENTS, ARCHITECTURES AND REUSE, 6., 2012. **Proceedings...** [S.l.]: IEEE, 2012. p. 31–40. 2

_____. An approach for mapping domain-specific aom applications to a general model. **Journal of Universal Computer Science**, v. 20, n. 4, p. 534–560, 2014. 30, 45

NAAB, M.; ROST, D. How to evaluate software architectures: tutorial on practical insights on architecture evaluation projects with industrial customers. In: INTERNATIONAL CONFERENCE ON SOFTWARE ARCHITECTURE COMPANION (ICSA-C), 2018. **Proceedings...** [S.l.]: IEEE, 2018. p. 6–7. 5

O'BRIEN, L. Design patterns 15 years later: an interview with erich gamma, richard helm and ralph johnson. **Indianápolis: Informit, Oct**, v. 22, 2009. 14

ORACLE. **JAVA EE 6 Tutorial**: Expression language. 2018. 66

SANAEI, R.; OTTO, K.; HÖLTTÄ-OTTO, K.; LUO, J. Trade-off analysis of system architecture modularity using design structure matrix. In: INTERNATIONAL DESIGN ENGINEERING TECHNICAL CONFERENCES AND COMPUTERS AND INFORMATION IN ENGINEERING, 2015. **Proceedings...** [S.l.]: ASME, 2015. p. V02BT03A037. 5, 79

SANT'ANNA, N.; GUERRA, E.; IVO, A.; PEREIRA, F.; MORAES, M.; GOMES, V.; VERAS, L. G. Modelo arquitetural para coleta, processamento e visualização de informações de clima espacial. In: SIMPÓSIO BRASILEIRO DE SISTEMAS DE INFORMAÇÃO, 2014. **Proceedings...** [S.l.], 2014. p. 125–136. 5

SANTOS, R. D. C. dos; CORREA, L. A. R.; GUERRA, E. M.; VIJAYKUMAR, N. L. A private cloud-based architecture for the brazilian weather and climate virtual observatory. In: INTERNATIONAL CONFERENCE ON COMPUTATIONAL SCIENCE AND ITS APPLICATIONS, 2013. **Proceedings...** [S.l.]: Springer, 2013. p. 295–306. 5

SUN MICROSYSTEMS. **Javabeans(TM) specification 1.01 final release**. ago. 1997. Disponível em: <http://download.oracle.com/otn-pub/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/beans.101.pdf>. 8, 29, 66

VEGAS, S.; APA, C.; JURISTO, N. Crossover designs in software engineering experiments: benefits and perils. **IEEE Transactions on Software Engineering**, v. 42, n. 2, p. 120–135, 2016. 46

WELICKI, L.; YODER, J. W.; WIRFS-BROCK, R. Rendering patterns for adaptive object-models. In: CONFERENCE ON PATTERN LANGUAGES OF PROGRAMS, 14., 2007. **Proceedings...** [S.l.]: ACM, 2007. p. 12. 16

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in software engineering**. [S.l.]: Springer, 2012. 46, 55, 56, 57

YASSINE, A. An introduction to modeling and analyzing complex product development processes using the design structure matrix (dsm) method. **Urbana**, v. 51, n. 9, p. 1–17, 2004. 5, 77, 79, 95

YODER, J. W.; BALAGUER, F.; JOHNSON, R. Architecture and design of adaptive object-models. **ACM Sigplan Notices**, v. 36, n. 12, p. 50–60, 2001. 2, 20, 21, 25, 94

YODER, J. W.; JOHNSON, R. The adaptive object-model architectural style. In: BOSCH, J.; GENTLEMAN, M.; HOFMEISTER, C.; KUUSELA, J. (Ed.). **Software architecture**. [S.l.]: Springer, 2002. p. 3–27. 4, 15, 16, 22, 25

YODER, J. W.; RAZAVI, R. Metadata and adaptive object-models. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 2000. **Proceedings...** [S.l.]: Springer, 2000. p. 104–112. 8

GLOSSÁRIO

Atributo - Utilizado para declarar variáveis de instância em uma classe.

Classe - Referência para as classes definidas com a linguagem de programação.

Classe Estática - Classes definidas com uma linguagem de programação tradicional.

Entidade - Abstração de uma estrutura de negócio que está sendo representado no software.

Framework Tradicional ou Reflexivo - Frameworks que se baseiam na estrutura de uma classe. Eles utilizam por padrão, reflexão e anotações para encontrar e invocar métodos.

Modelo de Classe Estático - Modelo em que as classes são definidas em linguagem de programação tradicional e não utilizam metadados para manipular atributos e métodos em tempo de execução.

Propriedade - Representa logicamente uma informação sobre uma entidade.

Rule Object - Padrão que define uma abstração para comportamento de regra de negócio.

RuleObject - Entidade de regra que implementa o padrão Rule Object.

Tipo de entidade - Abstração dos metadados de uma entidade de negócio.

PUBLICAÇÕES TÉCNICO-CIENTÍFICAS EDITADAS PELO INPE

Teses e Dissertações (TDI)

Teses e Dissertações apresentadas nos Cursos de Pós-Graduação do INPE.

Manuais Técnicos (MAN)

São publicações de caráter técnico que incluem normas, procedimentos, instruções e orientações.

Notas Técnico-Científicas (NTC)

Incluem resultados preliminares de pesquisa, descrição de equipamentos, descrição e ou documentação de programas de computador, descrição de sistemas e experimentos, apresentação de testes, dados, atlas, e documentação de projetos de engenharia.

Relatórios de Pesquisa (RPQ)

Reportam resultados ou progressos de pesquisas tanto de natureza técnica quanto científica, cujo nível seja compatível com o de uma publicação em periódico nacional ou internacional.

Propostas e Relatórios de Projetos (PRP)

São propostas de projetos técnico-científicos e relatórios de acompanhamento de projetos, atividades e convênios.

Publicações Didáticas (PUD)

Incluem apostilas, notas de aula e manuais didáticos.

Publicações Seriadas

São os seriados técnico-científicos: boletins, periódicos, anuários e anais de eventos (simpósios e congressos). Constam destas publicações o Internacional Standard Serial Number (ISSN), que é um código único e definitivo para identificação de títulos de seriados.

Programas de Computador (PDC)

São a seqüência de instruções ou códigos, expressos em uma linguagem de programação compilada ou interpretada, a ser executada por um computador para alcançar um determinado objetivo. Aceitam-se tanto programas fonte quanto os executáveis.

Pré-publicações (PRE)

Todos os artigos publicados em periódicos, anais e como capítulos de livros.