Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Flexibilização de Regras de Negócio Aplicadas ao Sistema de Dotação de Material do Exército Brasileiro

Pedro Henrique Teixeira Costa

Dissertação apresentada como requisito parcial para conclusão do
Mestrado Profissional em Computação Aplicada

Orientadora
Prof.a Dr.a Edna Dias Canedo

Brasília
2018

Ficha catalográfica elaborada automaticamente,
com os dados fornecidos pelo(a) autor(a)

CP372f

Costa, Pedro Henrique Teixeira
Flexibilização de Regras de Negócio Aplicadas ao Sistema de Dotação de Material do Exército Brasileiro / Pedro Henrique Teixeira Costa; orientador Edna Dias Canedo. -- Brasília, 2018.
132 p.

Dissertação (Mestrado - Mestrado Profissional em Computação Aplicada) -- Universidade de Brasília, 2018.

1. Regras de Negócio. 2. Motor de Regras. 3. Programação Generativa. 4. Logística Militar. 5. Linguagens Específicas de Domínio. I. Canedo, Edna Dias, orient. II. Título.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Flexibilização de Regras de Negócio Aplicadas ao Sistema de Dotação de Material do Exército Brasileiro

Pedro Henrique Teixeira Costa

Dissertação apresentada como requisito parcial para conclusão do
Mestrado Profissional em Computação Aplicada

Prof.a Dr.a Edna Dias Canedo (Orientadora)
CIC/UnB

Prof.a Dr.a Edna Dias Canedo
CIC/UnB

Prof. Dr. Rodrigo Bonifácio de Almeida
CIC/UnB

Prof. Dr. Edilson Ferneda
Universidade Católica de Brasília - UCB

Prof.a Dr.a Aletéia Patrícia Favacho de Araújo
Coordenadora do Programa de Pós-graduação em Computação Aplicada

Brasília, 11 de Outubro de 2018

Dedicatória

Dedico esta dissertação à minha família, pelo apoio incondicional em todos os momentos, principalmente nos de incerteza, pelo constante incentivo e fonte de inspiração.

Dedico também à minha orientadora Prof.a Dr.a Edna Dias Canedo pela confiança, paciência, incentivo, amizade e excelente orientação.

Sem o apoio de vocês, este trabalho não teria sido realizado. A eles, meu muitíssimo obrigado.

Agradecimentos

A realização da presente dissertação de mestrado não teria se tornado uma realidade sem o precioso apoio de várias pessoas, às quais estarei eternamente grato.

Aos meus pais, por me terem dado educação, disciplina, valores e por terem me incentivado a alcançar caminhos cada vez mais distantes. Agradeço a todos os familiares, pelo apoio incondicional, incentivo e compreensão por tudo que não pude fazer junto deles neste longo caminho percorrido até a conclusão do Mestrado.

À minha orientadora Prof.a Dr.a Edna Dias Canedo pela paciência, por todo o conhecimento transmitido, incentivo, disponibilidade, pela amizade e apoio em todos os momentos. Só tenho a agradecer pela oportunidade, pelas orientações, valiosas contribuições para o trabalho, palavras de incentivo, puxões de orelha, paciência, compreensão, competência e dedicação.

Aos professores Rodrigo Bonifácio de Almeida e Edilson Ferneda, que aceitaram compor minha banca de qualificação e de defesa, pelas sugestões e análises significativas. Aqui lhes exprimo a minha gratidão.

Aos professores e funcionários do Mestrado Profissional em Computação Aplicada da UnB. Aos alunos da turma do Mestrado, principalmente aos da linha de Engenharia de *Software*, pela paciência e companheirismo.

A todos que direta ou indiretamente contribuíram para a concretização desta dissertação, pela paciência, atenção e força que prestaram em diversos momentos. O meu profundo e sentido agradecimento.

Resumo

Distribuição de materiais é um tema comum na área de logística. A distribuição de Materiais de Emprego Militar (MEM) no Exército Brasileiro envolve a catalogação de materiais, a definição de regras que associam MEMs a unidades organizacionais, e a execução das regras para derivar os materiais previstos para cada unidade militar, listados no Quadro de Material Previsto (QDM). Atualmente, o QDM é gerado quase que totalmente de forma manual através do preenchimento de planilhas eletrônicas para cada Organização Militar (OM). O objetivo deste trabalho é apresentar a solução desenvolvida para geração automática de QDMs a partir da definição de regras de distribuição e posterior execução em um motor de regras. A simplicidade na definição das regras de distribuição de materiais, transformando definições de alto nível em definições baseadas em regras de motor de inferência, facilita a manutenção e extrai tais definições do código-fonte de uma aplicação. Isso motivou o uso de programação generativa, em termos de meta-programação, e um motor de inferência específico para a linguagem Java. Embora seja específica para a 4ª Subchefia do Estado-Maior do Exército (EME), a solução tende a ser genérica o suficiente para ser adotada, após algumas adaptações, por outras áreas do exército ou até mesmo organizações externas que lidam com regras de distribuição de materiais semelhantes.

Palavras-chave: Regras de Negócio, Motor de Regras, Programação Generativa, Linguagens Específicas de Domínio, Logística Militar.

Abstract

Material distribution is a common theme in the logistics area. The distribution of Military Employment Materials (MEM) in the Brazilian Army involves the cataloging of materials, the definition of rules that associate MEMs with organizational units, and the execution of rules to derive the materials for each military unit listed in the Foreseen Material Table (QDM). Currently, QDM is generated almost entirely manually by filling spreadsheets for each Military Organization (OM). The objective of this work is to present the solution developed for automatic generation of QDMs from the definition of distribution rules and later execution in a rules engine. Simplicity in defining material distribution rules, transforming high-level definitions into definitions based on inference engine rules, facilitates maintenance and extracts such definitions from the source code of an application. This motivated the use of generative programming, in terms of meta-programming, and an inference engine specific to the Java language. Although it is specific to the 4th Army Staff Sub-Committee (EME), the solution tends to be generic enough to be adopted after some adaptations by other areas of the army or even outside organizations dealing with similar materials.

Keywords: *Business Rules, Rule Engine, Generative Programming, Domain Specific Languages, Military Logistics.*

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ADL Architecture Description Language.

ANTLR Another Tool for Language Recognition.

API Application Programming Interface.

AST Abstract Syntax Tree.

BI Batalhão de Infantaria.

BLiP Business Logic integration Platform.

BRMS Business Rules Management System.

CDI Contexts and Dependency Injection.

CEP Complex Event Processing.

DAO Data Access Object.

DRL Drools Rule Language.

DSL Domain Specific Language.

EB Exército Brasileiro.

EMF Eclipse Modeling Framework.

GMF Graphical Modeling Framework.

GP Generative Programming.

IDE Integrated Development Environment.

JAR Java Archive.

JPA Java Persistence API.

JSF JavaServer Faces.

JSR107 Java Temporary Caching API.

JVM Java Virtual Machine.

KB Knowledge Base.

LHS Left-Hand-Side.

MB Managed Bean.

MEM Material de Emprego Militar.

OM Organização Militar.

OSGi Open Services Gateway Initiative.

PBT Property-based Testing.

POJO Plain Old Java Object.

POM Project Object Model.

PROMISE-EB Projeto de Pesquisa para Validação de Práticas e Métodos de Desenvolvimento de Software para o Exército Brasileiro.

QA Quality Attribute.

QC Quadro de Cargos.

QCP Quadro de Cargos Previstos.

QDM Quadro de Dotação de Material.

QDMP Quadro de Dotação de Material Previsto.

RBS Rule-Based System.

RHS Right-Hand-Side.

SIGELOG Sistema de Informações Logísticas.

SISBOL Sistema de Boletins.

SISDOT Sistema de Dotação de Material.

SOA Service-oriented architecture.

SQA Software Quality Assurance.

SQL Structured Query Language.

SWEBOK Software Engineering Body of Knowledge.

UML Unified Modeling Language.

VOs Value Objects.

WAR Web Application Archive.

Capítulo 1

Introdução

Cada vez mais as organizações necessitam de agilidade e velocidade no desenvolvimento de soluções corporativas. Uma alteração ou inclusão de uma regra de negócio em um sistema em produção são geralmente custosas, consomem tempo e envolvem várias pessoas como, por exemplo: analista de negócio, analista de interface, desenvolvedor e testador.

Uma possível abordagem utilizada hoje em dia são os motores de regras, que permitem agilizar o desenvolvimento da solução, diminuindo o risco de manutenção nas regras de negócio e deixando o gestor de negócios à vontade para alterar as regras de acordo com a necessidade da organização sem depender de um analista de sistemas.

Regra de negócio é toda norma ou tudo aquilo que a lei ou o uso comum determina a respeito de qualquer transação que envolve uma determinada organização. Portanto, regra de negócio é uma diretriz destinada a regulamentar o comportamento do negócio [1]. De acordo com Morgan [2], regras de negócio expressam declarações que são elementos-chave na definição das intenções e necessidades do negócio, ou reflexões de como a organização trabalha ou como pretende trabalhar no futuro.

Em muitas organizações, a maioria das regras escritas está desatualizada e muitas vezes em conflito umas com as outras. Regras são geralmente criadas por meio de comunicados que as pessoas podem ou não cumprir, ou adicionadas à pilha crescente de papel na parte da frente ou de trás do manual de políticas. As operações de negócio não têm dado a devida atenção para esse problema e as regras de hoje muitas vezes não proveem suporte às políticas existentes e podem não estar de acordo até mesmo com a legislação.

Segundo Brasil [3], motores de regras são definidos como ferramentas que proveem suporte à identificação, definição, racionalização e qualidade de regras de negócio e regras técnicas. Motores de regras também proveem um repositório que permite que as regras sejam comparadas entre si para definição ou contextualização de problemas, verificação de redundâncias e da qualidade do que foi definido.

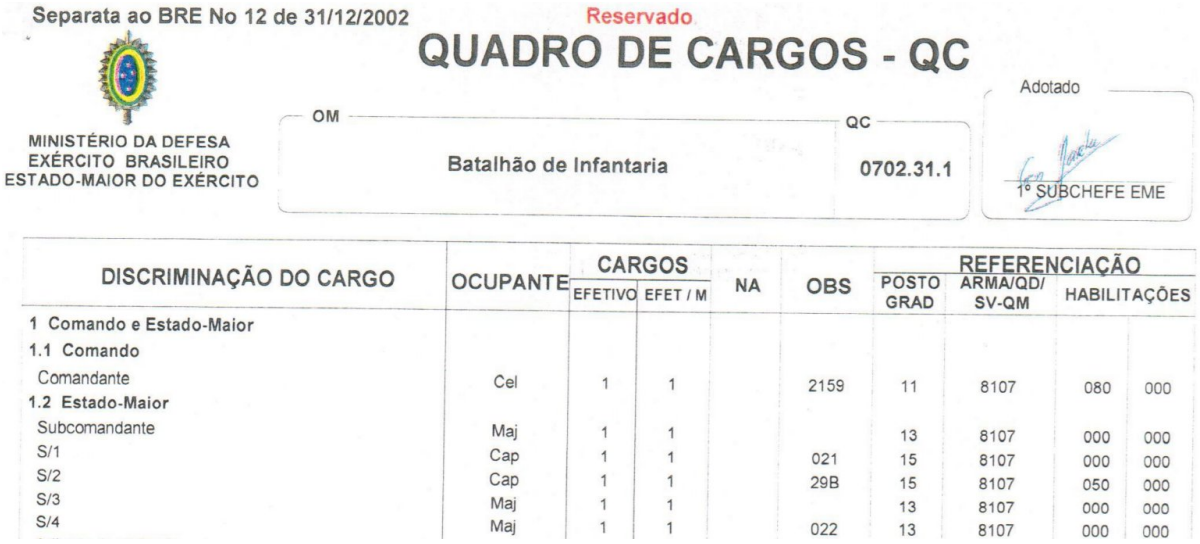
Uma das fases da Logística de Materiais de Emprego Militar das Forças Armadas contempla o planejamento da distribuição de materiais para as unidades militares, a qual pode ser realizada utilizando um motor de regras, como proposto neste trabalho. Tal fase é conhecida como Fase de Dotação de Materiais, e envolve a catalogação de materiais (muitas vezes independente de fornecedor), a definição de regras que associam materiais de emprego militar a unidades organizacionais (por exemplo, diretorias, departamentos, batalhões e cargos militares) e a execução das regras para derivar os materiais inicialmente previstos para cada unidade militar e os materiais que devem ser realmente distribuídos para as organizações militares (que podem sofrer variações dadas as inclusões e supressões de cargos de uma determinada unidade). Esta é a atribuição do Sistema de Dotação de Material.

O Sistema de Dotação de Material (SISDOT) é um sistema gerenciado pela 4ª Subchefia do Estado-Maior do Exército Brasileiro e que tem por finalidade determinar todas as dotações de material das Organizações Militares (OM) do Exército Brasileiro, através da elaboração dos Quadros de Dotação de Material (QDM) e dos Quadros de Dotação de Material Previsto (QDMP) das OM [4],[5]. Tendo por objetivo responder ao seguinte questionamento: **Qual é a quantidade prevista ou necessária de cada Material de Emprego Militar?**.

No processo de elaboração dos QDM e QDMP para uma determinada organização militar são levados em consideração tanto seu Quadro de Cargos (QC) quanto seu Quadro de Cargos Previstos (QCP), os quais indicam os militares efetivos necessários com as respectivas qualificações/habilitações exigidas para o exercício de sua atividade fim.

Várias regras de negócio, conhecidas internamente como "chamadores", são definidas para identificar quais Materiais de Emprego Militar (MEM) devem ser providos a cada OM. Através do cruzamento destas regras negociais com o QC de um tipo de OM é gerado o QDM. Já o QDMP de uma determinada OM é gerado através do cruzamento do QDM e do QCP desta OM.

Um tipo de OM pode ser visto, ao fazer um paralelo com a orientação a objetos, como uma classe e uma OM física é um objeto, uma instância de determinado tipo de OM. Por exemplo: Batalhão de Infantaria (BI) é um tipo de OM que define a estrutura que todos os batalhões de infantaria devem possuir. Essa estrutura é definida no QC específico do BI, indicando os departamentos, conhecidos internamente como frações, e seus cargos em forma de uma árvore [6], [7]. Com isso, as frações podem possuir frações e/ou cargos como filhos e as folhas da árvore são sempre cargos. Cada cargo possui uma série de condições que devem ser atendidas para que um militar possa exercer o cargo. Essas condições podem incluir: a graduação (patente), arma, habilitações ou observações. A Figura 1.1 apresenta o início do QC de um Batalhão de Infantaria (BI).



Separata ao BRE No 12 de 31/12/2002

Reservado

QUADRO DE CARGOS - QC

MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
ESTADO-MAIOR DO EXÉRCITO

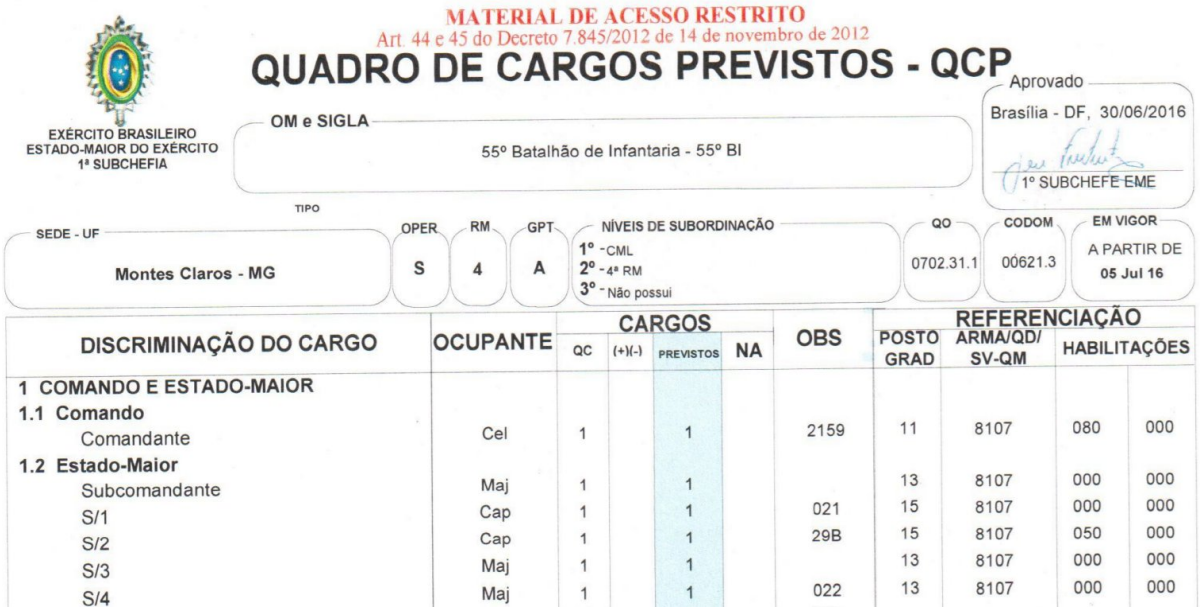
OM: Batalhão de Infantaria — QC: 0702.31.1

Adotado
1º SUBCHEFE EME

DISCRIMINAÇÃO DO CARGO	OCUPANTE	CARGOS		NA	OBS	REFERENCIAÇÃO			
		EFETIVO	EFET / M			POSTO GRAD	ARMA/QD/ SV-QM	HABILITAÇÕES	
1 Comando e Estado-Maior									
1.1 Comando									
Comandante	Cel	1	1		2159	11	8107	080	000
1.2 Estado-Maior									
Subcomandante	Maj	1	1			13	8107	000	000
S/1	Cap	1	1		021	15	8107	000	000
S/2	Cap	1	1		29B	15	8107	050	000
S/3	Maj	1	1			13	8107	000	000
S/4	Maj	1	1		022	13	8107	000	000

Figura 1.1: *Quadro de Cargos*
Fonte: Exército Brasileiro

Uma OM física é uma instância de um tipo de OM, herdando sua estrutura organizacional. Porém, podem haver algumas inclusões ou supressões de cargos de acordo com a realidade da OM física em questão. Essa estrutura específica é determinada no Quadro de Cargos Previstos (QCP) dessa OM. A Figura 1.2 apresenta o início do QCP de uma instância de Batalhão de Infantaria (BI), o 55º Batalhão de Infantaria.



MATERIAL DE ACESSO RESTRITO
Art. 44 e 45 do Decreto 7.845/2012 de 14 de novembro de 2012

QUADRO DE CARGOS PREVISTOS - QCP

EXÉRCITO BRASILEIRO
ESTADO-MAIOR DO EXÉRCITO
1ª SUBCHEFIA

OM e SIGLA: 55º Batalhão de Infantaria - 55º BI

Aprovado
Brasília - DF, 30/06/2016
1º SUBCHEFE EME

SEDE - UF	TIPO OPER	RM	GPT	NÍVEIS DE SUBORDINAÇÃO	QO	CODOM	EM VIGOR
Montes Claros - MG	S	4	A	1º - CML 2º - 4ª RM 3º - Não possui	0702.31.1	00621.3	A PARTIR DE 05 Jul 16

DISCRIMINAÇÃO DO CARGO	OCUPANTE	CARGOS			NA	OBS	REFERENCIAÇÃO			
		QC	(+)(-)	PREVISTOS			POSTO GRAD	ARMA/QD/ SV-QM	HABILITAÇÕES	
1 COMANDO E ESTADO-MAIOR										
1.1 Comando										
Comandante	Cel	1		1		2159	11	8107	080	000
1.2 Estado-Maior										
Subcomandante	Maj	1		1			13	8107	000	000
S/1	Cap	1		1		021	15	8107	000	000
S/2	Cap	1		1		29B	15	8107	050	000
S/3	Maj	1		1			13	8107	000	000
S/4	Maj	1		1		022	13	8107	000	000

Figura 1.2: *Quadro de Cargos Previstos*
Fonte: Exército Brasileiro

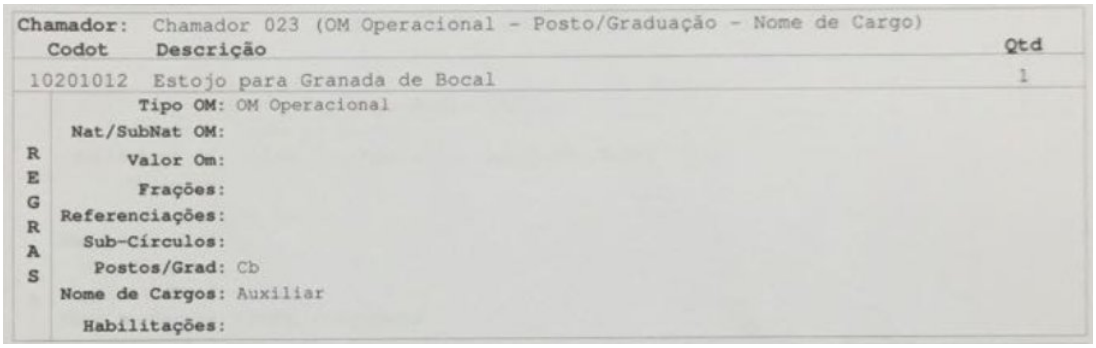
Cada QC, e consequentemente os QCPs derivados, é classificado em: operacionalidade, natureza, subnatureza, grupo e valor. Essa classificação é feita em uma estrutura similar a

uma árvore, onde as folhas dessa árvore são QCs. A Figura 1.3 apresenta a classificação do QC de um Grupo de Artilharia de Campanha de Selva, classificado como: OM operacional; Natureza: Artilharia; Subnatureza: Selva; Grupo: Genérico; e Valor: Grupo (Artilharia).

Figura 1.3: *Classificação de Quadros de Cargos*

Os chamadores, mencionados acima, podem ser declarados usando cada uma dessas condições ou para cargos de um QC específico. A Figura 1.4 apresenta um exemplo de chamador que define que cada militar de uma OM operacional, com a patente de cabo e que exerça um cargo de Auxiliar receberá um estojo para granada de bocal. Existe a possibilidade, não mostrada nesta figura, de negar algumas condições como, por exemplo: poderia existir uma condição habilitando a natureza de Infantaria mas negando para a subnatureza Infantaria/Selva, ou seja, todas as OMs com natureza Infantaria estariam previamente habilitadas, exceto as que possuam natureza Infantaria e subnatureza Selva.

A partir do cruzamento das informações contidas em um Quadro de Cargos (QC) com as condições contidas nos chamadores é gerado o Quadro de Dotação de Material (QDM)



Chamador: Chamador 023 (OM Operacional - Posto/Graduação - Nome de Cargo)

Codot	Descrição	Qtd
10201012	Estojo para Granada de Bocal	1

REGRAS

Tipo OM: OM Operacional
Nat/SubNat OM:
Valor Om:
Frações:
Referenciações:
Sub-Círculos:
Postos/Grad: Cb
Nome de Cargos: Auxiliar
Habilitações:

Figura 1.4: *Regra de Distribuição (chamador)*
Fonte: Exército Brasileiro

para a OM. O QDM é um dos principais artefatos gerados pelo SISDOT, servindo de base para outro importante artefato, o QDMP. A Figura 1.5 apresenta um trecho do QDM para o Batalhão de Infantaria (BI), onde é possível visualizar os materiais e suas quantidades por fração e a totalização desses materiais.

RESERVADO

QUADRO DE DOTAÇÃO DE MATERIAL - QDM

MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
ESTADO-MAIOR DO EXÉRCITO

QO: 0702.31.1 - Batalhão de Infantaria
Nº Ctl: 1
Adotado: 4º Subchefe do EME
Aprovação: Port nº 42 - EME-Res de 16/03/2005 - Publicação: BRE nº 03/05

CODOT / Descrição do Material	1º Cia Fuz		2ª e 3ª Cia Fuz(2)				TOTAL	N Distr	OBS
	2º e 3º Pel Fuz	Pel Ap	Cmdo	1º Pel Fuz	2º e 3º Pel Fuz	Pel Ap			
Classe II - Material de Intendência									
01 - Equipamentos									
10201025 - Caneco	74	25	13	37	74	25	**599**	470	
10201024 - Cantil	74	25	13	37	74	25	**599**	470	
10201013 - Coldre Ambidestro	14	13	5	7	14	13	**182**	471	
10201001 - Colete de Proteção	74	25	13	37	74	25	**599**	093	064
10201002 - Conjunto de Proteção para Atirador de Arma Coletiva	12	10		6	12	10	**102**	465	
10201009 - Equipamento de Uso Individual Completo	74	25	13	37	74	25	**599**	464	
10201022 - Estojo para Bússola	18	10	3	9	18	10	**150**	525	
10201026 - Estojo para Cantil e Caneco	74	25	13	37	74	25	**599**	470	
10201011 - Estojo para Carregador de Fuzil	120	24	16	60	120	24	**834**	526	

Figura 1.5: *Quadro de Dotação de Material*
Fonte: Exército Brasileiro

Este trabalho visa o estudo de motores de regras e sua utilização no Sistema de Dotação de Material do Exército Brasileiro, de forma a facilitar a definição e execução das regras negociais usadas no Sistema.

1.1 Problema de Pesquisa

Existe uma convergência conceitual e semelhanças dos termos nas definições de logística apresentadas por Ballou [8], Bowersox et al. [9] e Gasnier [10], tratando a logística como

o processo de planejar, executar e controlar o fluxo e armazenagem de forma eficaz e eficiente.

O planejamento logístico tenta resolver quatro das maiores áreas-problema: níveis do serviço ao cliente, localização das instalações, decisões de estoques e decisões de transportes [8].

A distribuição de materiais, foco deste trabalho, envolve decisões relacionadas a estoque e transporte. São definidos os materiais (e suas quantidades) necessários para que os funcionários exerçam suas atividades. Ao integrar essas definições com a gestão de estoque e as taxas esperadas de consumo dos materiais, a organização pode obter vantagens estratégicas como, por exemplo, negociar a aquisição de materiais junto aos fornecedores quando o preço destes está em baixa.

A solução proposta, apesar de atender às necessidades específicas do Exército Brasileiro (EB), pode ser utilizada em outras organizações após modificações na solução para adequar à sua realidade.

1.2 Justificativa

A distribuição de materiais de emprego militar pelo Exército Brasileiro (EB) é uma atividade que precisa ser realizada com muita atenção e cautela. Uma distribuição indevida pode acarretar no cancelamento de operações importantes para a nação. Além disso, é preciso distribuir os materiais corretamente para que a execução das atividades seja realizada de maneira correta, sem expor o militar a situações de riscos. Atualmente o processo de distribuição de material é realizado de maneira precária, sem contar com mecanismos de inteligência que verificam as habilidades de um militar e os materiais que são essenciais e necessários para suas atividades diárias, de acordo com suas habilitações.

A distribuição correta e precisa dos materiais de emprego militar a todas as organizações militares é fundamental para que os militares exerçam suas funções e o Estado-Maior consiga ter uma visão geral dos materiais de dotação previstos e os realmente distribuídos a sua corporação.

Devido à quantidade e complexidade dos chamadores elaborados pelo Exército Brasileiro, que impactam na produtividade da equipe para definir as regras que vinculam Materiais de Emprego Militar ao Quadro de Cargos, uma solução bem-sucedida envolve a geração de QDMs a partir das regras de dotação, os chamadores. Com isso pretende-se, além dos benefícios da utilização de motores de regras, reduzir o tempo de geração/disponibilização de um QDM.

A definição de regras negociais específicas vai possibilitar a geração automática de um QDM e QDMP, que hoje são criados manualmente. Mesmo com o uso de motores de

regras, ainda poderá haver alguma intervenção manual, mas grande parte do QDMP deverá ser gerado automaticamente. Essa intervenção manual pode ocorrer quando o QCP inclui frações extras que não existem no QC e, consequentemente, não foram tratadas na geração do QDM. Então essas frações, quando existirem, deverão ser tratadas manualmente no QDMP específico. Outra possibilidade de intervenção manual pode ocorrer quando não são especificados todos os chamadores necessários ou especificados de forma errônea e, com isso, o QDM será gerado de forma incompleta ou com dados errados, exigindo a intervenção manual dos gestores, seja para correção do QDM/QDMP ou para correção/especificação de chamadores.

1.3 Objetivos

1.3.1 Objetivo Geral

O objetivo geral deste trabalho é mapear as regras negociais dos chamadores e normas de dotação para distribuição dos materiais de emprego militar do Exército Brasileiro, utilizando motores de regras para que possa gerar o Quadro de Dotação de Material (QDM) e o Quadro de Dotação de Material Previsto (QDMP) automaticamente.

1.3.2 Objetivos Específicos

Para alcançar o objetivo geral, foram definidos os seguintes objetivos específicos:

- Entendimento das regras negociais do Sistema de Dotação de Material (SISDOT);
- Revisão e compreensão dos motores de regras utilizados na literatura;
- Seleção do motor de regra a ser adotado na solução proposta;
- Construção da solução;
- Testes da solução proposta;
- Implantação da solução proposta;
- Avaliar e descrever os resultados obtidos.

1.4 Resultados Esperados

A transformação de regras de negócio de alto nível em regras de negócio específicas para serem executadas em um motor de regras para a geração automática de QDMs pode

facilitar a manutenção do sistema, além de extrair tais definições do código-fonte da aplicação. Os resultados esperados com a proposta desse trabalho são:

- Investigar o uso de técnicas de meta-programação para o domínio de distribuição de materiais;
- Ampliar os conhecimentos sobre motores de regras;
- Geração automática de QDMs e QDMPs com o uso da solução desenvolvida;
- Validação da solução proposta através de um estudo de caso.

A abordagem proposta será validada em um cenário específico de logística do Exército Brasileiro que é complexa, não sendo um cenário trivial, o que permite que a estratégia poderá ser explorada em outras situações de logística.

1.5 Metodologia de Pesquisa

A ciência é uma forma de conhecer o mundo, de tentar saber cada vez mais, de desvendar os mistérios da natureza [11]. Mas não há ciência sem o emprego de métodos científicos [12]. A Metodologia é a aplicação de procedimentos e técnicas que devem ser observados para construção do conhecimento, com o propósito de comprovar sua validade e utilidade nos diversos âmbitos da sociedade [13], traçando o caminho a ser seguido, detectando erros e auxiliando as decisões do cientista [12].

O conhecimento científico obtido no processo metodológico tem como finalidade, na maioria das vezes, explicar e discutir um fenômeno baseado na verificação de uma ou mais hipóteses, estando diretamente vinculado a questões específicas na qual trata de explicá-las e relacioná-las com outros fatos [14]. Nesta seção é definido o protocolo da metodologia de pesquisa usada neste trabalho, e pode ser vista, de forma resumida, na Figura 1.6.

Do ponto de vista da natureza da pesquisa, este trabalho se classifica como uma pesquisa aplicada que tem por objetivo gerar produtos e/ou processos, com finalidades imediatas, com base em conhecimentos prévios. Quanto à abordagem, este trabalho faz uso da abordagem quantitativa pois os dados a ser analisados durante a validação são numéricos e requerem o uso de recursos e de técnicas estatísticas [15]. Quanto aos objetivos da pesquisa, caracteriza-se como uma pesquisa exploratória, pois visa proporcionar maior familiaridade com o problema investigado a fim de torná-lo explícito [16].

Neste projeto, a coleta e análise dos dados serão realizadas com base em normas, melhores práticas, técnicas, programas, modelos, materiais publicados, constituído principalmente de livros, artigos de periódicos [16] e materiais disponibilizados pelo Exército

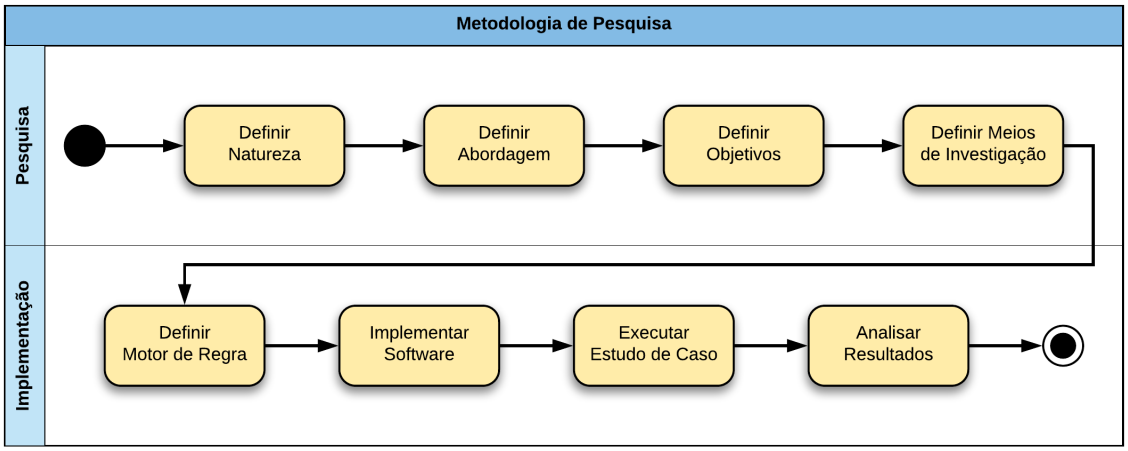


Figura 1.6: *Protocolo da Metodologia de pesquisa*

Brasileiro para o entendimento de sua Norma de Dotação; caracterizando-se, portanto, como uma pesquisa bibliográfica do ponto de vista do procedimento técnico empregado.

Este trabalho também pode ser considerado um estudo de caso prático [17], visto que tem objeto de pesquisa restrito, procurando conhecer seus aspectos, suas características ou reconhecer um padrão científico em que o caso possa ser enquadrado, modelado e implementado [18]. Estudos de caso são largamente usados na investigação de práticas de engenharia de *software* e diversas outras áreas, sendo caracterizados por sua natureza flexível e múltiplas formas de coleta de dados [19]. O métodos de estudo de caso podem ser aplicados e usados de diferentes maneiras e contextos [20], além de ajudar a indústria a avaliar os benefícios de métodos e ferramentas e fornecer uma maneira econômica de garantir que as mudanças no processo forneçam os resultados desejados [21].

Para guiar a seleção dos elementos que irão compor o Chamador e sua Norma de Dotação, o trabalho pautar-se-á pela sequência dos seguintes passos propostos. Em primeiro lugar, são definidos os objetivos e metas que se pretendem alcançar com o motor de regras do chamador. Em seguida, selecionam-se as variáveis/quadro de cargos que devem ser analisadas para responder aos objetivos, formalizando-se as regras e medidas que permitem obter os resultados almejados e enquadrando-se ao Quadro de Dotação de Material (QDM) a política de medição no processo de desenvolvimento de *software*. Por último, será apresentado o sistema que permite criar chamadores, QDM e a partir do chamador gerar as normas de dotação para as Organizações Militares.

1.6 Estrutura do trabalho

Este trabalho está organizado em 5 Capítulos, além deste, consistindo em:

- **Capítulo 2:** Apresenta a fundamentação teórica que apoia este trabalho, tratando conceitos relacionados a Logística de Materiais, Arquitetura de *Software*, Motores de Regras, Técnicas de Programação Generativa e Teste de *Software*.
- **Capítulo 3:** Apresenta o desenvolvimento da solução proposta neste trabalho, com a respectiva modelagem e implementação.
- **Capítulo 4:** Apresenta um estudo de caso prático realizado como forma de validar a solução proposta.
- **Capítulo 5:** Apresenta uma análise geral dos resultados da pesquisa e descreve os trabalhos futuros.

Capítulo 2

Fundamentação Teórica

O objetivo deste trabalho é definir uma arquitetura flexível para lidar com a logística de materiais de uma maneira genérica, que possa ser utilizada em diversas situações. Mas este trabalho encontra-se no âmbito do Exército Brasileiro (EB), mais especificamente na construção do Sistema de Dotação de Material (SISDOT).

Este Capítulo apresenta os conceitos teóricos necessários para o entendimento desse trabalho. As seções deste capítulo estão divididas da seguinte forma: A Seção 2.1 apresenta conceitos relacionados a logística e logística militar, tratando mais especificamente da logística militar no EB, foco da solução proposta. A Seção 2.2 apresenta conceitos relacionados a arquitetura de *software*, servindo de base para as decisões arquiteturais da solução proposta. A Seção 2.3 apresenta os conceitos sobre motores de regras (*rule engine*), sendo de grande importância sua utilização na geração automática de artefatos de saída da solução. A Seção 2.4 apresenta conceitos sobre programação generativa, como *template engine* e linguagens específicas de domínio. Estes conceitos são utilizados no desenvolvimento da solução. A Seção 2.5 apresenta conceitos sobre teste de *software*, sendo utilizados na geração automática de testes como uma forma de validar a solução desenvolvida.

2.1 Logística

A palavra logística tem como significado o raciocínio matemático relativo a lógica [22]. A logística não era vista como uma ciência no mundo clássico [23], com isso não é possível encontrar nesse período referências diretas ao uso da palavra logística, mas elementos em torno dos quais ela se formou, como: transporte, estudo de terrenos, suprimentos, máquinas, cavalos e homens [22].

De acordo com Ballou [24] e Serio et al [25] o papel e a importância da logística mudaram com o desenvolvimento das guerras militares. O trabalho apresentado por

Prebilic [26] assume que essa importância permaneceu praticamente inalterada ao longo da história militar pois as numerosas necessidades dos sistemas militares não mudaram em sua essência como, por exemplo, os militares precisam de água e comida. Assim, os aspectos logísticos estão associados às atividades militares, como a necessidade de suprir tropas militares com alimentos, medicamentos, munições e equipamentos diversos.

O livro "A Arte da Guerra"de Sun Tzu [27] destaca a importância da logística ao incluí-la em um dos fatores essenciais para a condução de uma guerra: "*As leis da organização e da disciplina militar encerram organização e regulamentos, o Tao do comando e a administração da logística*". O livro de Maquiavel [28] sugere a importância do gerenciamento dos suprimentos em trechos como: "*Melhor é vencer o inimigo com a fome do que com a espada, vitória em que pode mais a fortuna do que a virtude*"e "*Quem não prepara os víveres necessários é vencido sem espada*".

O trabalho apresentado por Leibniz [29], identifica a logística como uma das ciências lógicas, a sexta ciência é a logística, que trata do todo e das partes, ou seja, da magnitude em geral, das razões e das proporções. Segundo Brasil [30], o rei Gustavo Adolfo da Suécia, entre 1611 e 1632, ao modernizar a estruturação de suas forças militares com a criação de comboios, que contavam com medidas especiais de proteção, de elementos de suprimento e manutenção, deu origem à noção de um sistema de apoio logístico regular e organizado.

No âmbito da logística militar, os primeiros a pesquisar sistematicamente a logística e seu papel no sistema de defesa foram Carl von Clausewitz e Henri Antoine Jomini [26]. As áreas de alojamentos, manutenção e abastecimento, bases de operações, e linhas de comunicação, lidavam diretamente com logística na guerra, tanto no nível estratégico como no tático [31]. A logística e linhas de abastecimento foram referidas como centros de gravidade, ou alvos que, se destruídos, podiam derrotar o inimigo sem combate direto [32]. O debate sobre a importância da logística marcou um grande avanço no refinamento gradual do termo e seu conteúdo [26].

Um material de emprego militar (item de suprimento) sem qualquer relação com os engajamentos, servindo apenas para manter as forças, o abastecimento é o que mais diretamente afeta o combate. A distribuição desses itens é realizada quase todos os dias e afeta a todos os indivíduos. Assim, o material impregna totalmente os aspectos estratégicos de todas as ações militares [33]. O planejamento logístico deve levar em consideração quatro fatores na distribuição: demanda, distância, duração e destino [34].

A logística é o método usado para implementar estratégia e táticas militares, sendo um método para ganhar vantagem competitiva [31]. A logística é definida como a arte de mover exércitos, assim como a acomodação e abastecimento dos militares [26]. A estratégia e a tática proporcionam o esquema da condução das operações militares, enquanto

a logística proporciona os meios. Com isso, logística, estratégica e tática foram, pela primeira vez, posicionadas em um mesmo nível dentro da arte da guerra [35].

A logística é um dos elementos do poder de combate das forças terrestres, exercendo papel determinante na amplitude e duração das operações terrestres e contribui para a liberdade de ação dos comandantes táticos, aumentando a gama de opções disponíveis para o cumprimento de suas missões [36]. A previsão e a provisão do apoio necessário para a geração, o desdobramento, a sustentação e a reversão de forças terrestres em operações constitui um processo integrado, conforme apresentado na Figura 2.1.

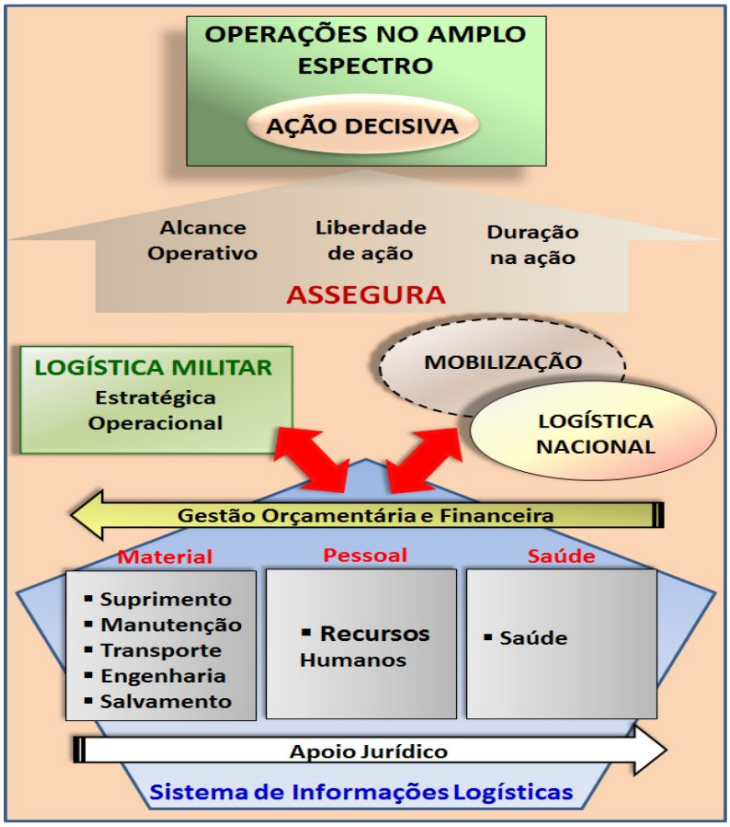


Figura 2.1: *Visão ampla da Logística Militar*
Fonte: [36].

A logística militar engloba três áreas funcionais básicas: material, pessoal e saúde. Essas constituem os eixos de atuação que direcionam o planejamento logístico em todos os níveis de execução, assegurando que as forças operativas terrestres estejam fisicamente disponíveis e apropriadamente equipadas no momento e local oportunos. A logística envolve, ainda, as atividades de Gestão Orçamentária e Financeira e de Apoio Jurídico, que permeiam todas as Áreas Funcionais, tendo por objetivo precípuo assessorar o processo decisório nos diversos níveis de execução do apoio logístico [36].

A logística está presente nos três níveis de condução das operações, assegurando a obtenção e a manutenção da capacidade operativa das forças empregadas. Nos níveis

estratégico e operacional ela condiciona o planejamento e a execução das operações, enquanto no nível tático adapta-se à manobra planejada para torná-la viável. A logística no nível estratégico interage com a Logística Nacional para obtenção e distribuição dos recursos necessários às forças apoiadas. A Logística Nacional é a principal fonte de obtenção de meios logísticos para a logística militar. O estabelecimento de convênios, contratação e terceirização são opções para a obtenção de capacidades logísticas, devendo-se avaliar, em cada caso, os eventuais riscos para a prontidão logística da força operativa a ser desdobrada [36].

A logística militar deve ser coerentemente planejada e executada desde o tempo de paz, bem como estar sincronizada com todas as ações planejadas, estando inerentemente ligada às logísticas conjunta e nacional, ou, em determinadas situações, à logística das operações multinacionais das quais o Brasil esteja participando. Em todas essas situações, deve ser meticulosamente coordenada para assegurar que os recursos sejam disponibilizados aos usuários em todos os níveis. Para tanto, sua organização deve ser pautada pela flexibilidade, adaptabilidade, modularidade, elasticidade e sustentabilidade [36]. A Doutrina Militar Terrestre estabelece os princípios a serem observados pela logística, os quais englobam, além daqueles previstos na Doutrina de Logística Militar, os seguintes preceitos: Antecipação, Integração, Resiliência, Responsividade e Visibilidade [36]. O ciclo logístico é o processo permanente, contínuo e ordenado em fases inter-relacionadas que organiza a sistemática do apoio, compreendendo as fases listadas abaixo [36] e que podem ser vistas na Figura 2.2:

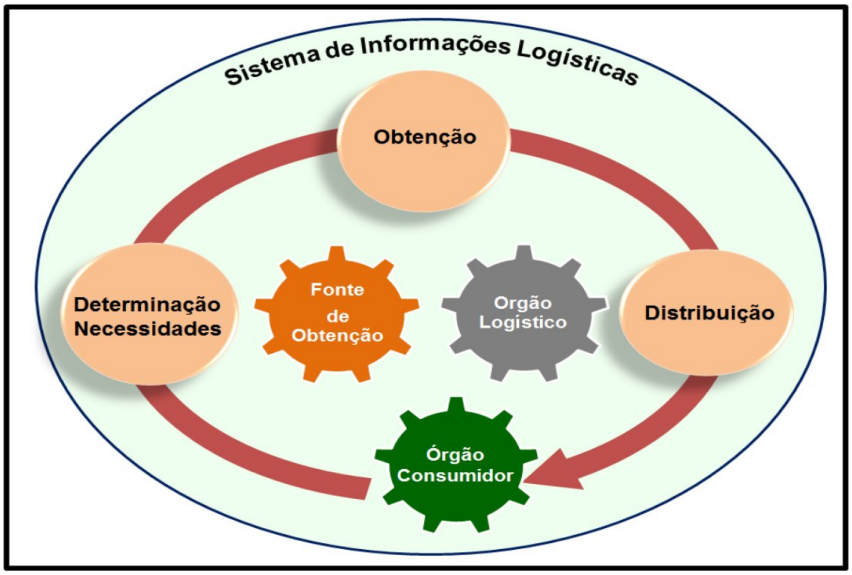


Figura 2.2: *O Ciclo Logístico na Força Terrestre*
Fonte: [36].

- Determinação das necessidades: consiste no exame pormenorizado dos planos propostos e, em particular, das ações e operações previstas, visando a identificar, definir e calcular que recursos logísticos deverão estar disponíveis, quando, em que quantidade, com qual qualidade, e em que local. A complexidade dessa fase decorre da necessidade de se antecipar as demandas, de modo à pré-posicionar os recursos necessários. Compreende o levantamento das necessidades para início das operações (completamento das dotações), sustentação da capacidade operativa (manutenção e reposição das dotações), constituição da reserva (atendimento de demandas específicas) e fins especiais (atendimento de necessidades que não constam das dotações normais). Entre outras, engloba as tarefas de estabelecimento de prioridades, escalonamento de suprimentos, previsão de recursos, estabelecimento de normas e diretrizes e configuração do sistema.

- Obtenção: transforma as necessidades logísticas levantadas em recursos que as satisfaçam. Nesta fase, são identificadas as fontes e tomadas medidas para a disponibilização de pessoal, material e serviços necessários à força apoiada. A obtenção de recursos humanos pode ocorrer por meio de movimentação de pessoal, concurso, formação, convocação, mobilização, contratação e recrutamento. A obtenção de recursos materiais, animais e de serviços ocorre, conforme disposições legais em vigor, por intermédio de doação, compra, contratação de serviço, confisco, contribuição, pedido, requisição, desenvolvimento, troca, empréstimo, arrendamento mercantil, transferência e convênio. Para efeito de planejamento do apoio logístico, devem ser levadas em consideração a existência e a destinação do material salvado e do material capturado.

- Distribuição consiste em fazer chegar aos usuários, oportuna e efetivamente, todos os recursos fixados pela determinação das necessidades. Engloba um sistema de pessoal, instalações, técnicas e procedimentos, visando a receber, acondicionar, movimentar, entregar e controlar o fluxo da cadeia logística entre o ponto de recepção e o ponto de destino.

A integração da cadeia logística por meio de sistemas informacionais é fundamental para a precisão e rapidez do ciclo logístico em todos os níveis de execução da logística, possibilitando aumentar o nível de serviço à força apoiada e apoiar a tomada de decisão. O SISDOT é um subsistema do Sistema de Informações Logísticas (SIGELOG).

Segundo Brasil [30], a logística militar é o conjunto de atividades relativas à previsão e à provisão de recursos humanos, materiais e animais, quando aplicável, e dos serviços necessários à execução das missões das forças armadas. A reunião, sob uma única designação, de um conjunto de atividades logísticas afins, correlatas ou de mesma natureza

é chamada de função logística. Uma atividade logística é um conjunto de tarefas afins, reunidas segundo critérios de relacionamento, interdependência ou de similaridade. Enquanto uma tarefa logística é definida como um trabalho específico e limitado no tempo, que agrupa passos, atos ou movimentos interligados segundo uma determinada sequência e visando à obtenção de um resultado definido.

No Exército Brasileiro, as atividades de logística são agrupadas em sete funções logísticas: Recursos Humanos, Saúde, Suprimento, Manutenção, Transporte, Engenharia e Salvamento. Como o foco deste trabalho é na distribuição de materiais, apenas a função logística suprimento será tratada. A função logística suprimento refere-se ao conjunto de atividades que trata da previsão e provisão do material de todas as classes de material e peças de reparação usadas nos equipamentos necessários ao apoio logístico às organizações e às forças apoiadas. A palavra "suprimento"pode ser, também, empregada com o sentido geral de item, artigo ou material. Tem como atividades o levantamento das necessidades, a obtenção e a distribuição [30].

Com o objetivo de facilitar a administração e controle dos suprimentos, o Exército Brasileiro utiliza dois sistema de classificação [30], [36]: Sistema de Classificação Militar e o Sistema de Classificação por Catalogação. O Sistema de Classificação Militar organiza os itens de materiais de suprimento em classes, conforme a finalidade de emprego [30].

O Sistema de Classificação por Catalogação é baseado no sistema da OTAN [37] para catalogação, classificando os itens em grupos e classes. A catalogação deve ser realizada de forma a ser obtida a identificação de cada item do material de forma precisa, racional e padronizada, evitando omissão, duplicidade ou dúvidas quanto às características de qualquer artigo. A catalogação torna-se, então, um instrumento valioso empregado pelos sistemas de gerenciamento logístico ao permitir, no menor tempo possível, a identificação do item de suprimento procurado, sua localização e quantidades disponíveis em estoque [30].

Sistema de suprimento é o conjunto integrado das organizações, pessoal, equipamentos, princípios e normas técnicas destinado a proporcionar o adequado fluxo do suprimento [30]. A organização e o funcionamento do sistema pressupõem [30]:

- Planejamento e supervisão de todas as ações relacionadas com o suprimento. O planejamento diz respeito à previsão e à provisão das necessidades correntes e futuras, enquanto a supervisão engloba a orientação, a coordenação e o controle de todas as ações de suprimento;
- Normas de solicitação e fornecimento que proporcionem presteza, a fim de atender com oportunidade as necessidades;

- Controles capazes de proporcionar todas as informações pertinentes à situação dos estoques e à comparação das necessidades com as disponibilidades;
- Órgãos executivos, nos diversos escalões de comando, encarregados da obtenção e da distribuição;
- Pessoal e instalações para receber, armazenar e distribuir os diversos itens;
- Utilização do menor número possível de instalações intermediárias, buscando minimizar o manuseio de itens.

Os níveis de estoque previstos serão mantidos por um sistema de suprimento automático. Para tanto, serão utilizados instrumentos de cálculo como quadros de dotação, fatores de consumo, suprimento e reposição, tabelas e outros. Quando surgirem necessidades especiais de reajustamento, serão feitos pedidos [30].

2.2 Arquitetura de Software

Nesta seção são apresentados, de forma resumida, conceitos sobre arquitetura de *software*. Esses conceitos são importantes para a construção de sistemas de *software* com qualidade, norteando assim o desenvolvimento da solução proposta.

Sistemas de *software* são construídos para satisfazer objetivos negociais das organizações. Porém, conforme sugerido por [38], a complexidade do *software* é uma propriedade essencial, não uma acidental. Pode ser muito difícil, senão impossível, para um desenvolvedor entender todas as sutilezas do *design* de um sistema, dependendo do seu tamanho. A arquitetura de *software* é vista como uma ponte entre os objetivos negociais das organizações e o sistema de *software* resultante, abstraindo a complexidade inerente ao sistema. A arquitetura pode ser desenhada, analisada, documentada e implementada usando técnicas conhecidas que oferecem suporte para que os objetivos de negócio sejam atingidos.

Várias decisões arquiteturais devem ser tomadas para que os objetivos negociais sejam atingidos. Algumas das principais decisões são tomadas no início do projeto e podem ter um impacto profundo em todas as outras decisões e no sistema. Outras decisões podem ser adiadas até que sejam necessárias [39].

A história do desenvolvimento de *software* é uma história do aumento do nível de abstração [40]. Cada inovação teórica ou técnica tende a adicionar uma camada de abstração, aumentando o nível de abstração com a qual o desenvolvedor deve lidar. Quando surgiram os primeiros computadores digitais na década de 1950, *software* era escrito em linguagem de máquina. Após certo tempo se tornaram claros alguns padrões de execução comumente usados para: avaliação de expressões aritméticas, invocação de procedimentos, *loops* e declarações condicionais. Com isso, surgiram as primeiras linguagens de programação de

alto nível, permitindo o desenvolvimento de programas mais sofisticados e o surgimento de tipos de dados [41].

No final da década de 1960 foi descrito o valor da utilização de estruturas hierárquicas, em determinados casos, mostrando-se vital para a verificação da solidez lógica do *design* e da corretude de sua implementação [42]. No entanto, a decisão de produzir sistemas hierarquicamente estruturados pode restringir a classe de possíveis sistemas, podendo introduzir desvantagens assim como as vantagens desejadas [43].

Na década de 1970 foi introduzido o conceito de modularidade [44] como um mecanismo para melhorar a flexibilidade e a compreensibilidade de um sistema, ao mesmo tempo que permite a redução do seu tempo de desenvolvimento, aumento do reuso de artefatos entre sistemas e facilita a evolução do sistema [45]. Sistemas grandes podiam ser constituídos de vários pequenos módulos, possivelmente escritos por pessoas diferentes [46].

A primeira referência à arquitetura de *software* ocorreu em 1969 em uma conferência sobre técnicas de engenharia de *software* organizada pela OTAN [47]. Apenas no final da década de 1980 que a arquitetura de *software* emergiu como uma disciplina distinta [48], baseando-se em várias ideias fundamentais que estavam firmemente em vigor, como: *information-hiding*, tipos de dados abstratos, entre outras. A década de 1990 pode ser considerada a "década da arquitetura de *software*"[49] ao serem lançados os primeiros catálogos de estilos arquiteturais [41], desenvolvimento das primeiras linguagens de descrição de arquitetura (Architecture Description Language (ADL)) [50], [51], definição de visões arquiteturais [52], a compreensão da relação entre decisões arquiteturais e atributos de qualidade [53], e o surgimento de técnicas para avaliação da arquitetura de *software*, como o SAAM [54].

Atualmente, a arquitetura de *software* se tornou mais ampla e complexa [55] devido às características dos sistema atuais, como: distribuídos e interconectados, escaláveis horizontalmente e prontos para nuvem, publicados automaticamente em ambientes diversos, atualizáveis com mínimo de *downtime*, resilientes e adaptativos. Há também uma diferença em relação à abordagem tradicional, que trata a arquitetura como um conjunto de componentes e conectores, ao tratar a arquitetura como um conjunto de decisões arquiteturais realizadas no sistema [56], [57].

Existem muitas definições de arquitetura de *software*, mas a ser utilizada neste trabalho é: arquitetura de *software* é um conjunto de estruturas necessárias para raciocinar sobre um sistema, que compreendem elementos de *software*, relações entre eles e propriedades de ambos [58].

A arquitetura de *software* é uma importante subdisciplina da engenharia de *software* e é, de certa forma, o particionamento prudente de um todo em partes, com relações

específicas entre as partes [59]. Este particionamento é o que permite grupos de pessoas, geralmente separadas por fronteiras organizacionais e geográficas, trabalharem juntas cooperativamente e de forma produtiva para resolver problemas grandes que não poderiam ser solucionados de forma individual. Um sistema é quase inevitavelmente particionado simultaneamente de diferentes formas. Cada particionamento resulta na criação de uma estrutura arquitetural. Existem três categorias de estruturas arquiteturais [59], [58]:

- Algumas estruturas particionam o sistema em unidades de implementação, chamadas módulos. Módulos são estruturas estáticas às quais são atribuídas responsabilidades, e servem de base para a distribuição de tarefas às equipes de desenvolvimento. Estruturas de módulo encorporam decisões sobre como o sistema deve ser estruturado como um conjunto de código ou unidades de dados. Examinar as estruturas de módulo é uma excelente forma de raciocinar sobre a capacidade de modificação do sistema. Servem para responder perguntas tais como: Qual a responsabilidade funcional primária de cada módulo? Quais outros elementos de *software* um módulo tem permissão de usar? Quais outros elementos são usados ou dependem deste módulo?
- Outras estruturas são dinâmicas, focando na forma como elementos interagem uns com os outros em tempo de execução. Esses estruturas são chamadas de *component-and-connector* (C&C) e possibilitam raciocinar sobre importantes propriedades de tempo de execução, como: performance, segurança, disponibilidade, entre outras. Ajudam a responder questões como: Quais são os principais componentes em execução e como eles interagem em tempo de execução? Quais os repositórios de dados compartilhados? Quais partes do sistema são replicadas? Como os dados progridem através do sistema? Quais partes do sistema podem ser executadas em paralelo?
- Outro tipo de estrutura descreve o mapeamento entre estruturas de *software* e ambientes organizacional, de desenvolvimento, instalação e execução. Esses mapeamentos são chamados estruturas de alocação e ajudam a responder questões como: em qual processador cada elemento é executado? Em quais diretórios ou arquivos cada elemento é armazenado durante o desenvolvimento, testes e construção do sistema?

A arquitetura de *software* é importante, além de outras características, pois [58]: está relacionada diretamente aos atributos de qualidade do sistema; permite raciocinar sobre o impacto de mudanças enquanto o sistema evolui; a documentação da arquitetura melhora a comunicação entre os interessados e serve como base de treinamento de novos membros da equipe; serve como uma receita para a implementação do sistema; ajuda no planejamento de custos e cronograma de atividades; possibilita o reuso de soluções.

Do ponto de vista organizacional, talvez a característica mais importante esteja diretamente ligada a questões econômicas. Um sistema com uma arquitetura pobre, no qual foi dedicado pouco esforço no seu *design*, possui algumas vantagens iniciais (em comparação com uma arquitetura bem definida) relacionadas a tempo para lançamento no mercado, possibilitando a definição de um conjunto mínimo de *features* de forma rápida. Conforme o sistema vai evoluindo e novas *features* são requeridas, o tempo para implementá-las tende a aumentar gradativamente pois pode envolver a refatoração de várias partes do código-fonte, chegando a um ponto de tornar-se inviável o acréscimo de mais funcionalidades.

Um sistema onde foi gasto suficiente esforço em seu *design* pode levar mais tempo para que uma versão inicial seja lançada no mercado. Porém, como a arquitetura foi bem definida, novas *features* tendem a ser incluídas de forma mais rápida e com o mínimo de impacto nas outras partes do sistema. Com isso, a partir de determinado momento, um sistema com uma arquitetura bem definida oferece vantagens em relação ao um sistema com *design* pobre, devido à facilidade de entregas de novas funcionalidades e de manutenção do mesmo [60].

2.2.1 Atributos de Qualidade

Os requisitos de um sistema são descrições do que o sistema deve fazer, os serviços que ele provê, as restrições em sua operação [61] e como ele deve reagir a estímulos. Os requisitos podem ser classificados em: funcional, não-funcional e restrições. Os requisitos não-funcionais são também conhecidos como atributos de qualidade (QA - *Quality Attribute*).

Um atributo de qualidade é uma propriedade mensurável ou testável de um sistema que é usada para indicar o quão bem o sistema satisfaz as necessidades dos interessados [58]. Atributos de qualidade são qualificações de requisitos funcionais ou do produto como um todo, determinando por exemplo: tempo máximo de resposta de uma determinada função, tempo máximo de *deployment* do sistema, quantidade máxima de cliques na tela para ativar uma funcionalidade. Exemplos de QA: disponibilidade, interoperabilidade, performance, segurança, usabilidade, testabilidade.

A arquitetura de *software* inibe ou permite a obtenção de atributos de qualidade, permitindo raciocinar sobre as consequências de alterações em determinado QA. Os atributos de qualidade são os que mais influenciam a arquitetura de *software* [58], onde as decisões arquiteturais são fortemente influenciadas pela necessidade de atingir os objetivos dos atributos de qualidade [62]. Se o sistema requer uma alta performance, deve-se ter atenção ao gerenciamento do comportamento dos elementos, seus usos de recursos compartilhados, e frequência e volume de comunicação entre os elementos. Se o sistema

deve ter alta disponibilidade, deve-se determinar como o sistema responde a uma falha ou erro.

Algumas técnicas, conhecidas como táticas, podem ser aplicadas para atingir determinado atributo de qualidade, afetando diretamente a resposta do sistema a determinados estímulos, como: incluir o uso de redundância para melhorar a disponibilidade, *cache* para melhorar a performance, e autenticação para melhorar a segurança [63]. Existem catálogos que listam diversas táticas para cada atributo de qualidade. Porém, as táticas não consideram o *tradeoff* com outros QA, podendo afetar positiva ou negativamente outros atributos de qualidade, como: segurança pode afetar negativamente a performance. Cabe ao arquiteto de *software* tomar as decisões levando em conta como estas afetam os atributos de qualidade.

Os atributos de qualidade mais relevantes para a arquitetura proposta nesta dissertação são:

- Manutenibilidade: o conjunto de linguagens e bibliotecas são limitadas às usadas no desenvolvimento do SIGELOG. A divisão do SISDOT em diversos módulos tende a facilitar a manutenção do sistema. Simplicidade na definição das regras de distribuição de materiais (chamadores), transformando definições de alto nível em definições baseadas em regras de motor de inferência, o que facilita a manutenção e extrai tais definições do código fonte de uma aplicação. Isso motivou o uso de programação generativa (em termos de metaprogramação) e um motor de inferência específico para a linguagem Java (Seção 2.3.1);
- Performance: considerando um total de 1000 chamadores, o sistema deve ser capaz de gerar um QDM em menos de 10 segundos;
- Segurança: apenas usuários com perfil de gerente podem gerar QDMs;
- Testabilidade: facilidade para execução de testes automatizados, motivando a definição de uma linguagem específica do domínio para descrever conceitos relacionados à dotação de materiais para o Exército Brasileiro. A partir da DSL, são gerados casos de testes baseados em "especificações executáveis" e classes que instanciam objetos do domínio.

2.2.2 Estilos Arquiteturais

Em alguns casos, elementos arquiteturais são compostos de uma forma específica para resolver um problema em particular. Algumas dessas composições aparecem de forma recorrente em sistemas completamente diferentes. Essas composições se mostraram úteis ao longo do tempo e em diferentes domínios, então foram documentadas e disseminadas

[58], definindo um vocabulário comum a ser usado em cada uma dessas composições e empacotando estratégias que explicam uma abordagem genérica de *design* para o sistema. Essas composições são conhecidas como estilos arquiteturais.

Um estilo arquitetural é uma especialização de tipos de elementos e relações, junto com um conjunto de restrições sobre como devem ser usados [59]. Estilos permitem que seja aplicado conhecimento especializado de *design* a uma determinada classe de sistemas e fornece suporte com ferramentas, análise e implementações específicas para o estilo, possibilitando o reuso em implementações subjacentes[64]. Um estilo arquitetural define, então, uma família de sistemas em termos de um padrão de organização estrutural [41].

Os estilos arquiteturais são um conceito importante na área de arquitetura de *software*, pois: oferecem soluções bem estabelecidas para problemas arquiteturais, ajudam a documentar decisões arquiturais, facilitam a comunicação entre interessados através do uso de um vocabulário comum, e descrevem atributos de qualidade do sistema [65].

Existe na literatura um termo similar a estilo arquitetural: o padrão arquitetural [59], [66]. Um padrão arquitetural foca no problema, seu contexto e em como resolver o problema no contexto determinado. No estilo arquitetural o problema não recebe tanta atenção, apenas a abordagem arquitetural, ou seja, elementos, suas relações e o conjunto de restrições aplicadas a eles. Neste trabalho não será feita uma distinção entre os termos e será usado o termo estilo arquitetural, ou apenas estilo.

Estilos arquiturais podem ser caracterizados de acordo com o tipo de estrutura arquitetural usada [59] e são listados a seguir de forma resumida:

- Módulos
  - Decomposição: mostra como as responsabilidades são alocadas entre módulos e submódulos;
  - Uso: mostra as dependências entre os módulos;
  - Generalização: mostra a generalização ou especialização de módulos;
  - *Layered*: agrupamento lógico de módulos de acordo com suas responsabilidades, de forma coesa;
  - Aspectos: mostra módulos especiais, chamados aspectos, que são responsáveis por *crosscutting concerns* [67], [68], [69];
  - Modelo de dados: descreve como a informação manipulada pelo sistema é estruturada como um conjunto de entidades de dados e seus relacionamentos [70].
- *Component-and-connector*

  - *Pipe-and-filter*: filtros processam a entrada de dados de forma serial e envia o resultado do processamento para o próximo filtro através de *pipes*;
  - Cliente-servidor: componentes cliente realizam requisições síncronas, do tipo *Request/reply*, a serviços dos componentes servidores;
  - *Peer-to-peer*: muitas instâncias do mesmo componente cooperam para atingir o objetivo desejado pela troca de mensagens síncronas, do tipo *Request/reply*;
  - *Service-oriented architecture* (SOA): envolve componentes distribuídos que agem como provedores e/ou consumidores de serviços altamente interoperáveis;
  - *Publish-subscribe*: componentes enviam eventos na forma de mensagens assíncronas para conectores pub-sub que as despacham para todos os componentes que se inscreveram para receber determinado evento;
  - *Shared-data*: mostra como repositórios de dados compartilhados são acessados para leitura/escrita pelos componentes;
  - *Multi-tier*: agrupamento lógico de componentes, sendo uma especialização do estilo cliente-servidor.

- Alocação

  - *Deployment*: descreve o mapeamento em tempo de execução entre componentes e o *hardware* da plataforma onde o *software* é executado;
  - Instalação: descreve a estrutura em árvore dos arquivos e diretórios no ambiente de produção e o mapeamento dos componentes nessa estrutura;
  - Atribuição de tarefas: descreve o mapeamento dos módulos entre pessoas ou equipes responsáveis pelo desenvolvimento desses módulos.

O estilo arquitetural seguido pelo SISDOT é o cliente-servidor, onde os componentes interagem requisitando serviços de outros componentes. Os tipos de componentes são: cliente, que invoca um serviço de um componente servidor, e servidor, que é um componente que provê serviços aos clientes [59]. O tipo de conector utilizado é o *request/reply*, que permite que o cliente faça uma chamada síncrona ao servidor, no caso do SISDOT uma chamada HTTP (*Hypertext Transfer Protocol*) [71].

A estrutura de execução do SISDOT está organizada em agrupamentos lógicos de componentes conhecidos como *tiers*, mecanismo frequentemente utilizado em estilos arquiteturais cliente-servidor. Com isso, o SISDOT pode ser caracterizado como seguidor desse padrão arquitetural. O padrão arquitetural *multi-tier* pode ser classificado como um padrão de C&C (*component and connector*) ou de alocação, dependendo do critério usado para definir as *tiers* [58]. Existe uma variedade de critérios, como o tipo do componente,

o compartilhamento do ambiente de execução, ou possuir o mesmo propósito de *runtime* [59].

Tiers induzem restrições topológicas que restringem quais componentes podem comunicar com outros componentes. Especificamente, conectores podem existir apenas entre componentes da mesma *tier* ou que residem em uma *tier* adjacente [59]. Adicionalmente, *tiers* podem restringir os tipos de comunicação que podem ocorrer em *tiers* adjacentes; facilitam a segurança; otimizam o desempenho e disponibilidade de formas especializadas; aumentam a modificabilidade do sistema e reduz o acoplamento [58]. As maiores desvantagens de arquiteturas *multi-tier* são o seu custo e complexidade, onde seus benefícios podem não ser justificados para sistemas menores e mais simples [58].

Para documentar o uso de um estilo são usadas, em geral, ADLs [72] ou visões arquiteturais.

2.2.3 Visões Arquiteturais

Em sistemas de *software*, as arquiteturas são frequentemente muito complexas para serem compreendidas por completo de uma vez. Por isso, a atenção deve ser dada a uma ou poucas estruturas do sistema em determinados momentos, deixando claro qual visão do sistema está sendo mostrada no momento. Uma visão é uma representação de um conjunto coerente de uma estrutura arquitetural, mostrando seus elementos, as relações entre eles e suas propriedades. Em geral, uma visão é representada na forma de modelos, usando uma notação informal, semiformal ou formal [59].

Modelos tem sido usados pela humanidade desde tempos remotos como uma forma de abstrair a complexidade do mundo real, devido aos limites do intelecto humano, que não permite absorver todas as informações disponíveis no mundo real, mesmo em contextos reduzidos [73].

Um modelo é uma simplificação da realidade [74], usando uma notação bem definida para descrever e simplificar uma estrutura, fenômeno ou relacionamento [75]. Modelos consistem de conjuntos de elementos que descrevem uma realidade física, abstrata ou hipotética [40]. Bons modelos podem (e devem) ser usados como uma forma de comunicação/documentação e são, na maioria dos casos, bem mais baratos de construir do que a solução em si.

A modelagem possui ampla aceitação entre todas as disciplinas de engenharia, principalmente porque a construção de modelos recorre a princípios de decomposição, abstração e hierarquia [76]. Sistemas complexos são hierárquicos e cada nível dessa hierarquia representa diferentes níveis de abstração, construídos um sobre o outro, mas que podem ser compreendidos de forma separada/individual.

Modelagem fornece ao engenheiro de *software* uma abordagem organizada e sistemática para representar aspectos significantes do *software* sendo estudado, facilitando a tomada de decisões e comunicação de decisões significativas entre os envolvidos [77].

2.2.4 Documentação de Arquitetura de Software

Todo sistema de *software* possui arquitetura mesmo que esta não tenha sido especificada e documentada, sendo desconhecida a todos. Mesmo a melhor arquitetura será essencialmente inútil se as pessoas que precisam usá-la não a entendem ou compreendem de forma equivocada, fazendo uso incorreto dela [59]. Então, criar uma arquitetura não é suficiente, ela deve ser documentada de forma a descrever o sistema com detalhes suficientes, sem ambiguidades e organizada de forma que as informações possam ser encontradas rapidamente.

A documentação da arquitetura possui fundamentalmente três usos [59]:

- Serve como meio de educação, facilitando a introdução de novas pessoas no projeto;
- Serve como veículo de comunicação entre os interessados;
- Serve como base para a análise e implementação do sistema

O princípio fundamental da documentação de arquitetura é [59]: documentar as visões arquiteturais relevantes e adicionar documentação que se aplica a mais de uma visão. Um documento de arquitetura é, então, uma coleção de visões selecionadas e documentação relacionada a mais de uma visão.

A escolha das visões que devem ser documentadas depende dos objetivos do projeto e da audiência de cada visão, para quem servirá, com qual propósito e com qual nível de detalhamento. Não existe uma receita que sirva para documentar todos as arquiteturas. Existem algumas sugestões na literatura de quais visões devem ser minimamente utilizadas, como a sugerida por [52], na qual deve-se utilizar minimamente quatro visões: lógica, física, de desenvolvimento e de processos, além de cenários para demonstrar a utilização dos elementos apresentados nas visões.

Nesta Seção foram apresentados conceitos relativos a arquitetura de *software* que servem de base para as decisões arquiteturais tomadas neste trabalho. Uma visão geral do que é arquitetura de software foi feita no início da Seção, incluindo um breve histórico e a definição proposta em [58]. Em seguida foi definido o que é um atributo de qualidade, sua importância para os sistemas de *software* e a influência da arquitetura de *software* na obtenção ou inibição desses atributos. Na sequência foi abordada a importância dos estilos arquiteturais e apresentada uma lista resumida dos principais estilos catalogados

em [59]. Então foram apresentadas as visões arquiturais e sua importância na compreensão de um sistema ou partes específicas deste. Por fim, mas não menos importante, foi abordada a documentação da arquitetura de *software*.

2.3 Rule Engine

Nesta seção são apresentados conceitos sobre motores de regras (*rule engine*). Esses conceitos são importantes para o trabalho devido à similaridade entre chamadores (regras de distribuição de alto nível) e as regras se-então apresentadas abaixo. Com isso, a solução proposta utiliza um motor de regras para a geração automática de Quadro de Dotação de Material (QDM).

Regras estão presentes no dia-a-dia de todos, mesmo que não sejam diretamente percebidas. Toda vez que um motorista para o carro em um sinal vermelho, o faz devido a uma regra de trânsito. Existem regras que determinam uma idade mínima para, por exemplo, tirar a carteira de motorista ou comprar bebidas alcoólicas. Outras são regras da natureza: se você não respirar vai asfixiar; se você pular eventualmente tocará o chão novamente devido à gravidade. Várias dessas regras seguem uma estrutura simples: para um grupo de condições detectadas, certas ações específicas devem ser tomadas.

Esse tipo de estrutura é muito importante para as organizações, já que elas precisam lidar cada vez mais com cenários complexos. Estes cenários são compostos por um grande número de decisões simples individuais, que trabalham em conjunto para fornecer uma avaliação complexa do quadro completo. Esta avaliação complexa começa com avaliações simples usadas para determinar a natureza do ambiente determinado, sendo conhecidas como inferências. Estas inferências podem ser cruzadas com outros dados ou mais inferências até que uma visão complexa do domínio pode ser alcançada, entendida e ações podem ser tomadas para o benefício dos objetivos da organização [78].

Conforme definido em [79], uma regra de negócio é uma declaração compacta, atômica e bem formada sobre um aspecto do negócio que pode ser expresso em termos que podem estar diretamente relacionados com o negócio e seus colaboradores, usando linguagem simples e não ambígua que é acessível a todas as partes interessadas: proprietário da empresa, analista de negócios, arquiteto técnico, cliente e assim por diante. Esse idioma simples pode incluir jargões específicos do domínio.

Sistemas baseados em regras (Rule-Based System (RBS)), também conhecidos como *production systems* ou *expert systems*, são a forma mais simples de inteligência artificial, usando regras como representação do conhecimento [80]. Ao invés de representar o conhecimento de forma declarativa e estática como um conjunto de coisas que são verdadeiras, RBS representa o conhecimento em termos de um conjunto de regras que diz o que fazer

ou o que concluir em diferentes situações [80]. Essas regras são também conhecidas como: condição-ação, produções, situação-ação ou se-então [81].

Sistemas especialistas (*expert systems*) são sistemas que são capazes de oferecer soluções para problemas específicos em um determinado domínio ou que são capazes de dar conselhos, tanto de um modo quanto em um nível comparável ao de especialistas na área [82]. Os problemas nos campos para os quais sistemas especialistas estão sendo desenvolvidos são aqueles que exigem considerável conhecimento humano para sua solução. Exemplos de tais domínios problemáticos são o diagnóstico médico de doenças, aconselhamento financeiro, *design* de produtos, manutenção e diagnósticos de máquinas e equipamentos, manipulação de alarmes, etc. [83].

Um dos primeiros sistemas especialistas que foi reconhecido como tal por especialistas da área de medicina foi o MYCIN [84]. O sistema foi desenvolvido na Universidade de Stanford para diagnosticar infecções sanguíneas e recomendar tratamentos, a partir dos dados de laboratório sobre testes em culturas retiradas do paciente [80]. Com cerca de 450 regras e 1000 fatos sobre medicina [85], o MYCIN foi capaz de executar tão bem quanto alguns especialistas, e consideravelmente melhor do que os médicos juniores [81]. Apesar de nunca ter sido utilizado de forma prática, o MYCIN mostrou ao mundo a possibilidade de substituição de um profissional médico por um sistema especialista [80].

Os especialistas tendem a expressar a maioria de suas técnicas de solução de problemas em termos de um conjunto de regras situação-ação, e isto sugere que sistemas baseados em regras deve ser o método de escolha para a construção de tais sistemas [84]. O apelo intuitivo das regras para resolver problemas mal estruturados resulta em regras que, muitas vezes, são fáceis para os não-programadores lerem e escreverem [86]. Alguns problemas podem ser facilmente solucionados usando linguagens de programação tradicionais, enquanto escrever um sistema baseado em regras é a maneira mais fácil de resolver os outros [87], dando um importante impulso ao desenvolvimento de sistemas especialistas similares em outras áreas, além da medicina [82].

Embora muitas técnicas diferentes tenham surgido para organizar coleções de regras, todos os RBSs compartilham certas propriedades-chave [84]:

- Incorporam conhecimento humano prático em regras condicionais se-então;
- Sua habilidade aumenta a uma taxa proporcional à ampliação de suas bases de conhecimento;
- Podem resolver uma ampla gama de problemas possivelmente complexos selecionando regras relevantes e então combinando os resultados de forma apropriada;
- Determinam adaptativamente a melhor sequência de regras a serem executadas;

- Explicam suas conclusões, refazendo suas linhas reais de raciocínio e traduzindo a lógica de cada regra empregada na linguagem natural.

Todo RBS consiste de poucos elementos básicos [80], [82], [83], [84], [85], [88]: uma **base de conhecimento** (*knowledge base* - KB) para armazenar o conhecimento, e o **motor de inferência** (*inference engine*), também conhecido como motor de regras (*rule engine*), que possui algoritmos para manipular o conhecimento representado na base de conhecimento.

A base de conhecimento armazena fatos e regras [84]. Fato é uma afirmação usualmente estática sobre propriedades, relações ou proposições [84] e deve ser algo relevante para o estado inicial do sistema [80]. Uma regra é uma declaração condicional que vincula determinadas condições a ações ou resultados [88], podendo expressar políticas, preferências e restrições [89]. As regras podem ser usadas para expressar o conhecimento dedutivo, como relacionamentos lógicos e, assim, suportar tarefas de inferência, verificação ou avaliação [84].

Uma regra se-então assume a forma "se x é A então y é B". A parte condicional (SE) é conhecida como: antecedente, premissa, condição ou LHS (*left-hand-side*). A outra parte é conhecida como: consequente, conclusão, ação ou RHS (*right-hand-side*) [80], [88].

Novas regras podem ser adicionadas à base de conhecimento à medida que aumenta o entendimento sobre o problema, permitindo o desenvolvimento incremental do sistema [83]. Conforme o número de regras do sistema vai aumentando, pode se tornar difícil modelá-las e gerenciá-las. Assim, em sistemas complexos com conjuntos de regras que consistem de milhares de regras, várias formas de representação desses conjuntos de regras são usadas, como: tabelas ou árvores, que são logicamente equivalentes a um conjunto de regras, mas são mais fáceis de entender e manter [90].

Tabelas de decisão são usadas para agrupar conjuntos de regras que são semelhantes com relação a um conjunto de fórmulas presentes nas pré-condições e conclusões ou ações [90]. Essas regras são expressas de forma tabular, possibilitando uma interpretação bastante intuitiva [79].

As árvores de decisão permitem organizar regras de maneira hierárquica. Ao mostrar as dependências entre condições e decisões, isso esclarece o pensamento sobre as consequências de certas decisões serem tomadas [90].

O motor de inferência tenta derivar nova informação sobre determinado problema usando as regras e os fatos [91] residentes na memória de trabalho (*working memory*). Consiste de algoritmos que selecionam e aplicam regras de acordo com os fatos e estratégia de busca selecionada para chegar a uma conclusão [83]. Existem dois tipos gerais de mecanismos de inferência usados em sistemas baseados em regras: *forward chaining* e *backward chaining* [88].

Forward chaining é um mecanismo orientado a dados (*bottom-up*), partindo do conhecimento existente armazenado como fatos e continua até que nenhuma outra conclusão possa ser extraída. O mecanismo verifica as condições prévias (LHS) das regras para ver quais podem ser disparadas. Então as ações no RHS são executadas. Portanto, a principal desvantagem desse modo de inferência é que muitas regras podem ser executadas e não têm nada a ver com a meta estabelecida [90]. Muitos sistemas podem ser definidos de forma a utilizar esse mecanismo, que pode ser implementado de maneira muito eficiente [81].

Backward chaining é um processo reverso de encadeamento e é orientado a objetivos. O sistema tem uma meta (uma solução hipotética) e o mecanismo de inferência tenta encontrar a evidência para provar isso com a ajuda dos fatos armazenados na base de fatos. O mecanismo de inferência processa as regras de maneira inversa (em comparação com o *forward chaining*), ou seja, de RHS para LHS, e procura por essas regras que a RHS corresponda ao objetivo desejado. Se LHS de uma determinada regra pode ser provada para ser verdade em termos dos fatos, o objetivo também é comprovado. Senão, então os elementos do LHS vão para a lista de objetivos e todo o processo se repete [90]. Este mecanismo é a forma mais utilizada de raciocínio automatizado [81].

Qual mecanismo é melhor depende do problema. As vantagens do *forward chaining* incluem sua simplicidade e o fato de que ele pode ser usado para fornecer todas as soluções para um determinado problema. Uma desvantagem é que pode ser ineficiente e também pode parecer sem objetivo. A principal vantagem do *backward chaining* é que ele não busca dados e não aplica regras que não estejam relacionadas ao problema em questão. É o método de escolha se um sistema está sendo usado para verificar uma hipótese específica. Geralmente, se houver muitos dados disponíveis, muitas vezes é mais rápido *chain backwards* de algumas hipóteses prováveis. Se existem muitas hipóteses possíveis, como no diagnóstico médico, provavelmente é melhor encadear os dados iniciais. A procura por regras aplicáveis pode ser restringida permitindo que os consequentes de regras nomeiem conjuntos de regras nos quais outras regras aplicáveis possam ser encontradas [83].

Um algoritmo que tem sido um componente-chave de máquinas de regras é o Rete [81]. O algoritmo Rete foi proposto por [92] como um método eficiente para comparar uma grande coleção de padrões com uma grande coleção de objetos. Ele encontra todos os objetos que correspondem a cada padrão. O algoritmo foi desenvolvido como um interpretador para uso em RBS, podendo ser útil em outros sistemas também, e tem sido usado para sistemas contendo desde algumas centenas ou milhares de padrões (regras) e objetos (fatos). O interpretador executa as seguintes operações [92]:

1. *Match*: avalia as LHS das regras para determinar quais são satisfeitas, dado o conteúdo atual da memória de trabalho;

2. Resolução de conflitos: seleciona uma regra com LHS satisfeito, se nenhuma produção satisfaz os LHSs, interrompe o processamento;

3. Ação: executa as ações definidas no RHS da regra selecionada;

4. Ir para o passo 1.

O Rete cria um grafo direcionado, acíclico e com raiz (ou uma árvore de busca). Cada caminho do nó raiz para uma folha na árvore representa o lado esquerdo de uma regra (parte IF). Cada nó armazena detalhes de quais fatos foram correspondidos pelas regras nesse ponto no caminho. À medida que os fatos são alterados, os novos fatos são propagados através do Rete a partir do nó raiz para as folhas, alterando as informações armazenadas nos nós apropriadamente. Isso significa adicionar um fato novo ou excluir um fato antigo ou alterar informações sobre um fato antigo. Dessa forma, o sistema só precisa testar cada novo fato em relação às regras, e somente contra as regras para as quais o novo fato é relevante, em vez de verificar cada fato em relação a cada regra [80].

Existem outros algoritmos que também são utilizados em sistemas baseados em regras, mas que não serão explorados neste trabalho, como: LEAPS [93], TREAT [94], [95], PHREAK [78], além de variações do Rete, como: Rete-OO [96], [78], Rete-ADH [97] e Rete-ECA [98].

Além da base de conhecimento e do motor de inferência, RBSs podem apresentar outros componentes, como [80]:

- Interface de usuário: comunicação entre o usuário que busca uma solução e o sistema especialista e a interação consiste de algum tipo de processamento de linguagem natural ou um ambiente gráfico;

- Subsistema de explicação: analisa a estrutura do raciocínio realizado pelo sistema e o explica ao usuário, dando ao usuário a possibilidade de indagar os sistemas sobre a maneira pela qual uma conclusão foi alcançada ou sobre os fatos usados.

- Subsistema de aquisição de conhecimento: verifica e atualiza a base de conhecimento, à medida que esta cresce, para possíveis inconsistências e informações incompletas;

- Agenda: lida com estratégias para resolução de conflitos, geralmente usando o algoritmo Rete, priorizando as regras candidatas [84].

O sistema baseado em regras funciona de uma forma muito simples [80]: começa com uma base de conhecimento, que contém todo o conhecimento apropriado codificado em regras se-então, e uma memória de trabalho, que pode ou não conter inicialmente quaisquer dados, asserções ou informações inicialmente conhecidas. O sistema examina todas

as condições da regra (LHS) e determina um subconjunto, o conjunto de conflitos, das regras cujas condições são satisfeitas com base na memória de trabalho. Deste conjunto de conflitos, uma dessas regras é acionada. A escolha da regra é baseada em uma estratégia de resolução de conflitos. Quando a regra é acionada, quaisquer ações especificadas em sua cláusula THEN (RHS) são executadas. Essas ações podem modificar a memória de trabalho, a própria base de regras ou fazer praticamente qualquer outra coisa que o programador do sistema decida incluir. Esse *loop* de regras de disparo e ações de execução continua até que um critério de finalização seja atendido. Esse critério de término pode ser dado pelo fato de que não há mais regras cujas condições são satisfeitas ou uma regra é disparada cuja ação especifica que o programa deve ser finalizado.

O trabalho apresentado por [80] apresenta algumas vantagens de sistemas baseados em regras:

- Os sistemas especialistas carregam a inteligência e as informações encontradas no intelecto de especialistas e fornecem esse conhecimento a outros membros da organização;
- Reduzir o erro devido à automação de tarefas tediosas, repetitivas ou críticas;
- Reduzir a mão de obra e o tempo necessário para testes de sistema e análise de dados;
- Reduzir os custos através da aceleração de observações de falhas;
- Eliminar o trabalho que as pessoas não devem fazer, como: tarefas que consomem muito tempo ou propensas a erros, tarefas em que as necessidades de treinamento são grandes ou custosas;
- Elimina o trabalho que as pessoas preferem não fazer (como tarefas que envolvem tomada de decisão, que não satisfaz a todos; sistemas especialistas asseguram decisões justas sem favoritismo em tais casos);
- Sistemas especialistas têm melhor desempenho que os humanos em determinadas situações;
- Realizar aquisição de conhecimento, análise de processo, análise de dados, verificação de sistema;
- Maior visibilidade do estado do sistema gerenciado;
- Desenvolver requisitos funcionais do sistema e coordenar o desenvolvimento de *software*;
- Para domínios simples, a base de regras pode ser simples e fácil de verificar e validar;

- Os *shells* de sistemas especialistas fornecem meios para construir sistemas simples sem programação;
- Fornecer respostas consistentes para decisões, processos e tarefas repetitivos.

A seguir são listadas algumas das desvantagens de sistemas baseados em regras[80]:

- O conhecimento especializado geralmente não é facilmente codificado em regras;
- Os especialistas geralmente não têm acesso aos seus próprios mecanismos de análise;
- Validação e verificação de sistemas grandes pode ser difícil;
- Quando o número de regras é grande, o efeito de adicionar novas regras pode ser difícil de avaliar;
- Há falta de bom senso humano em algumas tomadas de decisão;
- As respostas criativas que os especialistas humanos podem responder em circunstâncias incomuns não podem ser incorporadas a um sistema especialista;
- Os especialistas em domínio nem sempre conseguem explicar sua lógica e raciocínio;
- Existe falta de flexibilidade e capacidade de adaptação a ambientes em mudança, uma vez que as questões são padrão e não podem ser alteradas;
- O sistema especialista não é capaz de reconhecer quando não há resposta disponível.

2.3.1 Drools

O Drools é uma plataforma de integração de lógica de negócios (BLiP - Business Logic integration Platform), escrito na linguagem Java. É um projeto de código aberto que é apoiado pelo JBoss e Red Hat, Inc[99]. É licenciado sob a Licença Apache, Versão 2.0 [100]. Atualmente encontra-se na versão 7.12.0.

O trabalho no Drools (o motor de regras) começou em 2001. Desde o início, o Drools passou por muitas mudanças. O Drools 1.0 começou com uma pesquisa linear de força bruta. Foi então reescrito na versão 2.0, que foi baseada no algoritmo Rete, que impulsionou o desempenho do Drools. As regras foram escritas principalmente em XML (*Extensible Markup Language*) [101]. A versão 3.0 introduziu um novo formato, o DRL (*Drools Rule Language*), que é uma linguagem específica especialmente trabalhada para escrever regras. Provou ser um grande sucesso e tornou-se o principal formato para escrever regras. A versão 4.0 do motor de regras teve algumas melhorias importantes de desempenho, junto com a primeira versão de um Business Rules Management System (BRMS). Isso formou a base para o próximo grande lançamento (5.0), onde Drools tornou-se uma

plataforma de integração de lógica de negócios [99]. A plataforma consiste de quatro módulos principais [102]:

- Drools Expert: pode ser considerado o núcleo do projeto Drools, sendo usado para especificar, manter e executar suas regras de negócios;
- Drools Fusion: módulo que trata eventos complexos de processamento (*Complex Event Processing* – CEP);
- Drools Flow: combina regras e processos (*workflow*);
- Drools Guvnor: sistema de gerenciamento das regras de negócio (BRMS).

O Drools possui vários módulos opcionais, como: o OptaPlanner para resolver problemas de planejamento e o Drools Chance que adiciona suporte à incerteza. Outra parte muito importante do Drools é o seu plugin do Eclipse [103].

O Drools atualmente fornece um raciocínio claro (*crisp*), mas o raciocínio imperfeito está sendo desenvolvido. Inicialmente, isso será um raciocínio imperfeito com a lógica *fuzzy*; mais tarde, será adicionado suporte a outros tipos de incerteza. Existe trabalho também em andamento para trazer o raciocínio ontológico baseado no OWL (Ontology Web Language). Também continuam sendo realizadas melhorias nos recursos de programação funcional.

Neste trabalho foi usado o módulo Drools Expert, usado para executar as regras, juntamente com o algoritmo PHREAK. Este algoritmo foi introduzido na versão 6 do Drools para abordar alguns dos principais problemas do RETE, incorporando todo o código existente de ReteOO e todos os seus aprimoramentos, além de ter sido inspirado em outros algoritmos, como: LEAPS, Rete/UL [104] e Collection-Oriented Match [105]. Apesar de ser uma evolução do algoritmo RETE, não é mais classificado como uma implementação RETE [106].

A similaridade entre as regras de distribuição definidas no SISDOT e as regras utilizadas nos sistemas baseados em regras (RBS) levou à utilização de motores de regras neste trabalho. Nesta seção foram apresentados conceitos relacionados a motores de regras, os componentes básicos de (RBS) e seu funcionamento. O algoritmo Rete, que tem sido um componente-chave de motores de regras, foi apresentado assim como um resumo de seu funcionamento. Por fim, foi apresentada a ferramenta utilizada neste trabalho para a geração de QDM, o Drools.

2.4 Programação Generativa

Nesta Seção são apresentados conceitos relacionados a programação generativa. Existem várias técnicas de programação generativa e paradigmas com objetivos similares, sendo que as técnicas utilizadas neste trabalho são apresentadas com mais detalhe em subseções específicas. Na solução proposta foi utilizado um *template engine* para a transformação de regras de distribuição de alto nível (chamadores) em regras entendidas pelo Drools, as DRLs mencionadas na Seção 2.3. Foi criada uma linguagem específica de domínio com o objetivo de facilitar a declaração e geração de testes. Os conceitos referentes à definição dessa linguagem podem ser vistos na Subseção 2.4.2.

A maioria das tecnologias orientadas a objeto focam no desenvolvimento de sistemas únicos ao invés de famílias de sistemas. Portanto, elas não suportam adequadamente a reutilização de *software*, devido a deficiências apresentadas por [107], tais como: não distinguir entre engenharia para e com reuso; não existe uma fase para definição do escopo do domínio, com foco no desenvolvimento de um único sistema ao invés de visar na classe de sistemas; não há diferenciação entre a modelagem de variabilidade dentro de uma aplicação e entre várias aplicações.

Design patterns e *frameworks* representam uma contribuição extremamente valiosa da tecnologia orientada a objeto para a reutilização de *software* [108]. *Frameworks* de *software* são uma tecnologia de reuso de *software* que promove a reutilização de arquiteturas inteiras dentro de um domínio de aplicativo estritamente definido [109]. No entanto, elas ainda precisam ser acompanhadas por uma abordagem sistemática de engenharia para e com reuso [107].

Programação generativa (Generative Programming (GP)) está relacionada ao projeto e implementação de componentes de *software* que podem ser combinados para gerar sistemas especializados e altamente otimizados, cumprindo requisitos específicos [110]. Com esta abordagem, em vez de desenvolver as soluções a partir do zero e "reinventar a roda", soluções específicas de domínio são geradas a partir de componentes de *software* reutilizáveis [111].

GP é sobre a fabricação de produtos de *software* a partir de componentes em um sistema de forma automatizada (e dentro de restrições econômicas), ou seja, a forma como outras indústrias têm produzido bens mecânicos, eletrônicos e outros por décadas [112]. Porém, atualmente existem diferenças fundamentais, como por exemplo: os *hardwares* de rede são construídos a partir de componentes de prateleira (COTS - *components off the shelf*), que possuem uma padronização em suas *interfaces*; enquanto, para o *software*, a prática atual é construir os componentes personalizados para cada produto especificamente, seguindo padrões específicos de uma empresa, o que diminui a possibilidade de reuso [113].

A ideia principal na GP é semelhante àquela que levou as pessoas a mudar de linguagens *assembly* para linguagens de alto nível. Isto é, elevando o nível de abstração com o qual um programador ou *designer* deve lidar e automatizar todas as traduções das abstrações de alto nível para os componentes de implementação. O nível de abstração é um dos atributos mais importantes de uma linguagem de programação [114]. Se a linguagem abstrair dos detalhes da implementação, o código pode ser mais facilmente entendido e verificado. Em técnicas GP, os programadores definem um modelo do sistema que apenas define o conteúdo dos dados e a lógica dos aplicativos. Então, programas semelhantes aos compiladores traduzem esses modelos para os programas ou códigos de máquina reais automaticamente. A partir dos compiladores tradicionais, pode haver vários compiladores de modelos que traduzem um único modelo para várias plataformas, dependendo das necessidades de diferentes usuários [113].

De acordo com Barth et al. [112], programação generativa é definida como um paradigma de desenvolvimento de *software* baseado na modelagem de famílias de sistemas de *software* que, dada uma especificação de requisitos, um intermediário ou produto final altamente customizado e otimizado pode ser automaticamente fabricado sob demanda a partir de implementação elementar e reutilizável.

Os objetivos da GP são[110]: diminuir a lacuna conceitual entre o código do programa e os conceitos de domínio; alcançar alta capacidade de reutilização e adaptabilidade; simplificar o gerenciamento de muitas variantes de um componente; e aumentar eficiência (tanto no espaço quanto no tempo de execução).

Para atingir esses objetivos, a GP implementa vários princípios [110]:

- *Separation of concerns*: este termo, cunhado por Dijkstra, refere-se à importância de lidar com uma questão importante de cada vez. Para evitar código de programa que lida com muitos problemas simultaneamente, a programação generativa visa separar cada problema em um conjunto distinto de código. Esses pedaços de código são combinados para gerar um componente necessário;

- Parametrização de diferenças: como na programação genérica, a parametrização permite representar de forma compacta famílias de componentes (ou seja, componentes com muitos pontos em comum).

- Análise e modelagem de dependências e interações: nem todas as combinações de valores de parâmetros são geralmente válidas, e os valores de alguns parâmetros podem implicar os valores de alguns outros parâmetros. Essas dependências são chamadas de conhecimento de configuração horizontal, pois ocorrem entre os parâmetros em um nível de abstração.

- Separar o espaço do problema do espaço da solução: o espaço do problema consiste nas abstrações específicas do domínio com as quais os programadores de aplicativos gostariam de interagir, enquanto o espaço da solução contém componentes de implementação (por exemplo, componentes genéricos). Ambos os espaços têm estruturas diferentes e, portanto, o mapeamento entre eles é feito com conhecimento de configuração vertical. O termo vertical refere-se à interação entre parâmetros de dois níveis de abstração diferentes. Tanto o conhecimento de configuração horizontal quanto vertical são usados para configuração automática.
- Eliminar a sobrecarga e realizar otimizações específicas de domínio: ao gerar os componentes estaticamente (em tempo de compilação), grande parte da sobrecarga devido a código não utilizado, verificações de tempo de execução e níveis desnecessários de indireção podem ser eliminados. Otimizações específicas de domínio complicadas também podem ser executadas (por exemplo, transformações de *loop* para códigos científicos).

A geração de código proporcionada pela GP oferece uma série de benefícios, além de aumentar significativamente a produtividade durante o desenvolvimento e manutenção [115], tais como os apresentados em [116]:

- Qualidade: grandes volumes de código manuscrito tendem a ter qualidade inconsistente porque os engenheiros encontram abordagens novas ou melhores à medida que trabalham. Geração de código a partir de modelos cria uma base de código consistente instantaneamente, e quando os modelos são alterados e o gerador é executado, as correções de *bugs* ou melhorias de codificação são aplicadas de forma consistente em toda a base de código;
- Consistência: o código construído por um gerador de código é consistente no *design* das APIs (*Application Programming Interface*) e o uso de nomes de variáveis. Isso resulta em uma interface consistente, que é fácil de entender e usar;
- Um único ponto de conhecimento: uma alteração em um modelo percola através de todos os geradores em cascata para implementar a mudança no sistema;
- Mais tempo para *design*: com a geração de código, os engenheiros podem reescrever os modelos para modificar como as APIs são usadas e, em seguida, executar o gerador para produzir o código fixo. Os modelos tendem a ser muito mais curtos e específicos do que o código gerado. Além disso, como a geração de código reduz o tempo em determinados projetos, pode-se gastar mais tempo fazendo um *design* adequado e testes de protótipos de especificações alternativas;

- Consistência arquitetural: o gerador de código usado para um projeto é a realização das decisões de arquitetura feitas antecipadamente no ciclo de desenvolvimento, encorajando os programadores a trabalharem na arquitetura. Um gerador de código bem documentado e mantido fornece uma estrutura e abordagem, mesmo quando os membros da equipe deixam o projeto. Além de possibilitar a geração de código por não-programadores [115];
- Abstração: os modelos são definidos em um nível maior de abstração do que o código a ser gerado.

Embora os geradores de aplicativos ofereçam vantagens, eles também incorrem em custos. Suas desvantagens incluem [115]:

- Um único gerador de aplicativos pode ser usado efetivamente apenas em algumas situações;
- Os geradores são difíceis de construir. Exigem linguagens de especificação e interfaces de usuário cuidadosamente projetadas, um conhecimento profundo do domínio do aplicativo e a capacidade de projetar unidades genéricas de *software* confiável no domínio. A criação de um gerador de aplicativos exige conhecimento e habilidade no domínio do aplicativo e na construção de *parsers* e tradutores de linguagens;
- Reconhecer onde uma gerador de código pode ser usado é difícil e geralmente ocorre tarde no ciclo de vida, onde há menos motivação para refazer parte do desenvolvimento;
- A introdução de uma nova tecnologia coloca várias questões organizacionais.

Existem diversas técnicas de programação generativa e outros paradigmas com objetivos similares, como: *Generic Programming, Domain-Specific Languages (DSLs), e Aspect-Oriented Programming (AOP)* [110]. Nas Subseções a seguir são apresentados conceitos relacionados a GP que são usados neste trabalho. A Subseção 2.4.1 apresenta conceitos sobre *template engine*. A Subseção 2.4.2 apresenta conceitos sobre linguagens específicas de domínio.

2.4.1 Template Engine

Conceitos independentes devem ser representados de forma independente e devem ser combinados apenas quando necessário. Quando esse princípio é violado, conceitos não relacionados são agrupados ou dependências desnecessárias são criadas. De qualquer forma, obtém-se um conjunto de componentes menos flexível a partir do qual os sistemas

são compostos. Os *templates* fornecem uma maneira simples de representar uma ampla variedade de conceitos gerais e formas simples de combiná-los [117].

De acordo com He e Zheng [118], *template engine* é uma ferramenta genérica para gerar uma saída textual a partir de arquivos de *templates* e dados. Podem ser usados no desenvolvimento de *software* que necessita da geração automática de código conforme propósitos específicos [119].

A necessidade de páginas da web geradas dinamicamente levou ao desenvolvimento de inúmeros *templates engine* na tentativa de tornar o desenvolvimento de aplicações web mais fácil, melhorar a flexibilidade, reduzir os custos de manutenção, e permitir codificação em paralelo e desenvolvimento em HTML. Estes atraentes benefícios, que têm impulsionado a proliferação de *templates engine*, derivam inteiramente de um único princípio: separar a especificação da lógica de negócios e dos cálculos de dados de uma página, de como uma página exibe tais informações. Com especificações encapsuladas de forma separada, *template engine* promovem reutilização de componentes, aparências (*look*) plugáveis de sites, pontos únicos de mudança para componentes e alta clareza geral do sistema [120].

Além de serem muito utilizados na geração dinâmica de páginas *web*, os *templates* são usados em outras tarefas, tais como: geração de classes Java [121], arquivos de configuração [73], arquivos contendo comandos SQL, entre outras.

Um *template* é um documento de texto que combina marcadores de posição (*placeholders*) e fórmulas linguísticas usadas para descrever algo em um domínio específico [122], [121]. Os *templates* facilitam a comunicação entre os profissionais, contribuem para a disseminação da pesquisa e fornecem um guia útil para iniciantes[122].

O processo de geração de código baseado em *templates* utiliza, como entrada, *templates* e um conjunto de modelos (dados). A linguagem na qual o *template* é escrito é conhecida como metalinguagem [123]. O *template engine* age como um avaliador para gerar código, podendo ser visto como um metaprograma pois manipula código [123]. O modelo é um artefato que descreve o domínio da aplicação em um alto nível de abstração, em uma linguagem de programação ou de modelagem. O modelo é usado pelo avaliador para substituir os marcadores de posição em um *template*. Esses espaços reservados contêm expressões para obter dados do modelo [121].

Existem vários *template engine* [120]: Tapestry, WebMacro, Velocity, PTG, UniT, Tea, WebObjects, FreeMarker, ColdFusion, Template Toolkit, Mason, Thymeleaf, e Pebble. Neste trabalho foi utilizado o template engine Freemarker, que é um *software open-source* projetado para gerar texto a partir de *templates* [124]. O Freemarker pode ser considerado o substituto do Apache Velocity e encontra-se na incubadora de projetos da Apache Software Foundation (ASF), que apoia o seu desenvolvimento e é amplamente usado

nos projetos da família Apache, como o NetBeans [125].

O Freemarker foi escolhido de acordo com as vantagens apresentadas em [119] e [126]: é um *software* de código aberto; é um modelo de propósito geral; é mais rápido que alguns outros; fornece facilidades de uso (por exemplo, suporte a JSON (JavaScript Object Notation), variáveis compartilhadas, carregadores de modelos, entre outros); possui suporte automatizado em Java IDE (Integrated Development Environment); seus *templates* não são compilados para classes, ou seja, um *template* pode ser carregado ou recarregado durante o tempo de execução sem reimplementar o aplicativo; e formata automaticamente valores numéricos, datas e horas de acordo com a localização (*locale*).

2.4.2 Domain Specific Language

Uma linguagem fornece um meio de comunicação por som e escrita de símbolos. Os seres humanos aprendem a linguagem como consequência de suas experiências de vida, mas na linguística (ciência das línguas) as formas e significados das línguas são submetidos a um exame mais rigoroso. Esta ciência também pode ser aplicada a linguagens de programação, que são linguagens artificiais para fins de comunicação com computadores, mas, o mais importante, para comunicar algoritmos entre pessoas[127].

Uma linguagem é um conjunto de sentenças válidas [128]. Uma sentença é composta de frases, uma frase é composta de subfrases e símbolos de vocabulário. Uma linguagem serve como mecanismo para expressar intenções [129]. A criação de uma nova linguagem é uma tarefa demorada, requer experiência e é, portanto, geralmente realizada por engenheiros especializados em linguagens [130]. Para implementar uma linguagem, deve-se construir uma aplicação que leia sentenças e reaja apropriadamente às frases e símbolos de entrada que descobre [131].

A necessidade de novas linguagens para vários domínios em crescimento está aumentando, assim como o surgimento de ferramentas mais sofisticadas que permitem aos engenheiros de *software* definir um novo idioma com um esforço razoável. Como resultado, um número crescente de DSLs estão sendo desenvolvidas para aumentar a produtividade dos desenvolvedores dentro de domínios específicos [130]. Com sua forte relação com as técnicas de engenharia dirigidas por modelos, elas agora estão recebendo mais atenção com o uso em diferentes áreas relacionadas ao *software*, sendo orientadas a negócios ou apenas áreas técnicas [132].

Linguagens Específicas de Domínio (DSL - *Domain Specific Languages*) são idiomas de especificação ou linguagens de programação com alto nível de abstração, simples e concisas [133], focadas em domínios específicos, sendo projetadas para facilitar a construção de aplicativos, usualmente de forma declarativa, com expressividade limitada e linhas reduzidas de código, que resolvem problemas específicos desses domínios [134].

O conceito de DSL existe há muito tempo na comunidade Unix na forma de mini-linguagens [129], como utilitários de *shell*: awk, sed, etc. A linguagem para programar máquinas numericamente controladas APT foi desenvolvida em 1957-58. A BNF (Backus Normal Form) remonta a 1959. Alguns consideram o Cobol como uma DSL para aplicativos de negócios, mas outros argumentam que isso está levando longe demais a noção de domínio de aplicativos [135]. Outros exemplos de DSL que são apresentados nos trabalhos [136], [137], [138], [139], [140], [141], [142]: PIC, SCATTER, CHEM, LEX, YACC, Make, Mathematica, R, SQL, Latex, BibTeX, HTML, CSS, macros em planilhas (Excel), VHDL, linguagens de *script* que permitem construção de aplicações, arquivos de configuração XML.

Uma DSL é uma linguagem de programação de computadores de expressividade limitada, por meio de notações e abstrações apropriadas, focada e geralmente restrita a um domínio de problema específico [143], [139]. DSLs têm o potencial de reduzir a complexidade do desenvolvimento de *software* aumentando o nível de abstração para um domínio. De acordo com o domínio da aplicação, diferentes notações (textual, gráfica, tabular) são usadas [144].

Existem quatro elementos-chave na definição de DSLs [143]:

- Linguagem de programação de computadores: uma DSL é usada por humanos a fim de instruir um computador a fazer algo. Sua estrutura é projetada para facilitar seu entendimento por humanos, mas também deve ser executável por um computador;
- Natureza da linguagem: uma DSL é uma linguagem de programação, e como tal deve ter um senso de fluência, em que a expressividade não vem apenas das expressões individuais, mas também da maneira pela quais elas podem ser compostas;
- Expressividade limitada: uma DSL suporta um mínimo de recursos necessários para útil para tratar um aspecto específico de um sistema;
- Foco no domínio: uma linguagem limitada é útil apenas se tiver um foco claro em um domínio pequeno, sendo uma consequência da expressividade limitada.

Linguagens de propósito geral (GPL - General Purpose Language), como Java ou C, são mais próximas do domínio da computação [134] e são projetadas para serem capazes de lidar com praticamente qualquer tipo de tarefa de programação [145] , como: estruturas de dados, de controle e de abstração variadas [143], não importando a área ou domínio em que esse problema se encaixa [146]. Normalmente, elas são a preferência do programador pois, como elas são de uso geral e amplamente utilizadas, geralmente possuem uma grande comunidade de especialistas. Questões como a maturidade de uma linguagem, a disponibilidade de otimizadores bem testados e a existência de boas ferramentas ou ambientes

de desenvolvimento são mais cruciais para a preferência referida [146]. Por outro lado, as GPLs têm alguns inconvenientes quando se referem a outros aspectos, como escrever, ler e entender os programas, pois [146] a curva de aprendizado geralmente exige um longo tempo de preparação, o que implica em uma vasta perícia em programação para a escrita e leitura do código. As GPLs são difíceis de entender porque, por um lado, sua sintaxe e semântica não são óbvias devido à sua generalidade e, por outro lado, geralmente abordam particularidades de implementação que estão mais próximas da máquina do que da forma de pensar dos humanos [146]. Mesmo seguindo diferentes paradigmas de programação ou exibindo várias construções sintáticas, que visam elevar o nível de abstração, não é suficiente para superar as dificuldades observadas na compreensão dos programas GPL.

As DSLs possuem algumas vantagens em relação às GPLs, conforme apresentado em [133], [139], [147], [148]:

- Facilidade de compreensão, escrita e raciocínio, devido à legibilidade de sua sintaxe;
- Aumenta a produtividade, confiabilidade, testabilidade, portabilidade, manutenibilidade e reutilização;
- Aumenta a corretude do aplicativo desenvolvido e a comunicação entre o especialista em domínio e o programador;
- Permitem validação e otimização no nível de domínio;
- As DSLs reduzem a complexidade ao filtrar as operações complexas internas do sistema. As GPLs exigiriam a codificação manual de todos os detalhes que se tornam complicadas e demoradas;
- Vários paradigmas de programação podem ser combinados e o ruído sintático pode ser rapidamente reduzido;
- Incorporam conhecimento de domínio e, assim, permitem a conservação e reuso desse conhecimento;
- Programas DSL são concisos, auto-documentados, e algumas vezes podem ser reutilizados para fins diferentes;
- Distância semântica reduzida entre o problema e o programa, ao permitir que as soluções sejam expressadas no idioma e nível de abstração do domínio do problema. Consequentemente, os próprios especialistas no domínio podem entender, validar, modificar e, muitas vezes, até mesmo desenvolver programas DSL.

Algumas desvantagens do uso de DSLs são [139], [148]:

- Custos de projeto e desenvolvimento da DSL, e educação de usuários, que pode requerer conhecimento completo das restrições de domínio;
- Disponibilidade limitada;
- Pode levar à cacofonia da linguagem;
- Dificuldade de encontrar o escopo adequado para uma DSL;
- Dificuldade de equilibrar entre especificidade de domínio e construções de linguagem de programação de propósito geral;
- Perda potencial de eficiência quando comparada com *software* codificado manualmente em uma GPL.

Programação orientada a linguagem é um estilo geral de desenvolvimento que opera sobre a ideia de construir *software* em torno de um conjunto de linguagens específicas de domínio [149]. Várias áreas do sistema são identificadas e uma DSL é criada para cada uma dessas áreas identificadas [150].

Language workbench é um termo para descrever uma nova classe de ferramentas de desenvolvimento de *software*, projetadas para construir *software* através de um ambiente rico de múltiplas e integradas DSLs [150]. Essas ferramentas apoiam a definição, reutilização e composição eficientes de idiomas e suas IDEs (Integrated Development Environment), tornando acessível o desenvolvimento de novas linguagens e apoiando a programação orientada a linguagem, na qual conjuntos de linguagens sintática e semanticamente integradas podem ser construídos com um esforço comparativamente pequeno [151]. Isso pode levar a ambientes de programação multi-paradigmáticos e orientados a linguagem que podem endereçar importantes desafios de engenharia de *software* [151].

A maioria das linguagens *workbenches* fornece recursos configuráveis conhecidos de ambientes de programação profissionais, como conclusão automática de código, refatoração e realce de sintaxe. Além disso, eles normalmente fornecem uma coleção de linguagens de especificação internas e personalizadas que abordam preocupações comuns no desenvolvimento da linguagem como, por exemplo, suporte para *pretty-printing*, reescrita, *parsing* e análise ou geração de código [152]. Uma característica de uma linguagem *workbench* é a capacidade de editar a representação abstrata de um programa, em oposição à maneira convencional de manipular arquivos de texto para modificar um programa [153].

Não apenas o uso das linguagens *workbenches* estão crescendo, mas também os número e variedade das próprias *workbenches*. Uma desvantagem desse crescente número de sistemas é que a terminologia usada e os recursos suportados por diferentes *workbenches* são tão diferentes que tanto os usuários quanto os desenvolvedores devem se esforçar para entender os princípios comuns e as decisões de projeto [154].

Exemplos das primeiras linguagens *workbenches* são [153], [151]: SEM, MetaPlex, MetaEdit+, Centaur, Lisa. Estes sistemas foram originalmente baseados em ferramentas para a especificação formal de linguagens de programação de propósito geral. No entanto, muitos deles têm sido usados com sucesso para construir DSLs também. *Workbenches* textuais como JastAdd, Rascal, Spoofax e Xtext podem ser vistos como sucessores desses sistemas, alavancando avanços na tecnologia de edição dos IDEs tradicionais. Ao mesmo tempo, *language workbenches* projecionais como Jetbrains MPS e Intentional estão revivendo e refinando a velha ideia de editores de estrutura, abrindo a possibilidade de misturar notações arbitrárias [154]. Na edição projecional, a representação abstrata é a definição central do sistema. Então, o *workbench* manipula diretamente a representação abstrata e projeta uma representação editável para o programador, exibindo uma ampla gama de ambientes de edição, incluindo estruturas gráficas e tabulares, ao invés de apenas a forma textual [137].

Conforme definido por Bettini [155], Xtext é um *framework* do Eclipse [156], [157] para implementar GPLs e DSLs. Ele permite implementar linguagens rapidamente e, acima de tudo, abrange todos os aspectos de uma infraestrutura de linguagem completa, a partir do *parser*, gerador de código ou intérprete, até uma integração completa do Eclipse IDE com todos os recursos típicos apresentados anteriormente.

Para iniciar uma implementação DSL, o Xtext só precisa de uma especificação gramatical semelhante à Another Tool for Language Recognition (ANTLR). A partir dessa gramática são gerados o *lexer*, *parser*, o modelo AST (Abstract Syntax Tree), a construção do AST para representar o programa analisado e o editor do Eclipse com todos os recursos do IDE. Todas as preocupações do próprio Xtext e do código gerado pelo Xtext podem ser customizadas via injeção de dependência [158]. Para as mais preocupantes, o comportamento padrão do Xtext geralmente é bom. Para as preocupações que precisam de customização (validação, vinculação/escopo, entre outros), o Xtext fornece uma Application Programming Interface (API) fácil de usar.

O Xtext utiliza intensamente o Eclipse Modeling Framework (EMF) [159]. A AST criada pelo *parser* do Xtext é um modelo EMF. O modelo Ecore correspondente pode ser derivado automaticamente da gramática ou ser especificado explicitamente. Assim, o Xtext permite fácil integração com ferramentas do ecossistema Eclipse Modeling, como as linguagens de transformação Model-to-Model ou Model-to-Text. Outro exemplo é integrar o Xtext com ferramentas gráficas como o Graphical Modeling Framework (GMF) [160].

Para este trabalho foi desenvolvida uma DSL utilizando Xtext, visando a declaração de regras de distribuição de materiais para facilitar a criação de testes manuais e automáticos.

Nesta Seção foram apresentados conceitos relacionados a programação generativa. Conceitos em torno das duas principais técnicas utilizadas neste trabalho também fo-

ram apresentados: *template engine* e linguagens específicas de domínio. Então, resumidamente, o usuário cadastra os chamadores, que são transformados em DRL por um *template engine*, o freemarker. A geração do QDM é realizada pela execução das DRLs pelo Drools.

Para facilitar a definição de testes, o desenvolvedor define os chamadores usando a DSL proposta, a fim de aumentar a expressividade da definição do que está sendo testado e, possivelmente, aumentar a produtividade do desenvolvedor. Durante a execução dos testes os chamadores definidos na DSL são convertidos para uma representação em Java pelo Xtext. A partir daí a execução é igual à definida acima: o *template engine* transforma o chamador em DRL, que é executada pelo Drools, gerando o QDM.

2.5 Teste de Software

Nesta Seção são apresentados os conceitos sobre garantia da qualidade e teste de *softwares*. Esses conceitos são importantes para a geração automática de testes, utilizados como uma forma de validação da qualidade da solução proposta.

Controle e garantia de qualidade são atividades essenciais para qualquer empresa que produza produtos para serem usados por outros. A história da garantia de qualidade no desenvolvimento de *software* é paralela à história da qualidade na fabricação de *hardware*. Nos primórdios da computação a qualidade era responsabilidade exclusiva do programador. Padrões para garantia da qualidade de software foram introduzidos no desenvolvimento de *software* de contrato militar durante a década de 1970 e se espalharam rapidamente no desenvolvimento de *software* no mundo comercial [39].

Toda profissão tem um corpo de conhecimento composto de princípios geralmente aceitos. Para obter conhecimentos mais específicos sobre uma profissão, é necessário: ter concluído um currículo reconhecido ou ter experiência no domínio. Para a maioria dos engenheiros de *software*, conhecimento e experiência em qualidade de *software* é adquirida ao trabalhar em várias organizações. O Software Engineering Body of Knowledge (SWEBOK) [77] constitui o primeiro consenso internacional desenvolvido sobre o conhecimento fundamental exigido por todos os engenheiros de *software*, tornando-se um guia de uso e aplicação das melhores práticas de Engenharia de *Software*, de maneira sensata e razoável.

O SWEBOK é organizado em 10 áreas de conhecimento (Knowledge Areas - KA) e um tópico referente às disciplinas relacionadas com a Engenharia de *Software*. A KA de Qualidade de *Software* cobre técnicas estáticas, aquelas que não requerem a execução do *software* sendo avaliado, enquanto técnicas dinâmicas são cobertas na KA Teste de *Software*.

Os processos de Garantia da Qualidade de Software (Software Quality Assurance (SQA)) garantem que os produtos de *software* e os processos no ciclo de vida do projeto estejam em conformidade com seus requisitos de especificação, pelo planejamento, ordenamento e execução de uma série de atividades para fornecer a confiança adequada de que a qualidade está sendo incorporada ao *software*. Isto significa assegurar que o problema está clara e adequadamente declarado e que os requisitos da solução estão corretamente definidos e expressados [161]. É importante estar ciente de que a SQA não é responsável pela qualidade. A responsabilidade pela qualidade está no próprio projeto. O papel da SQA é monitorar a maneira como o projeto executa suas responsabilidades e fornecer à gerência e à equipe técnica os dados necessários sobre a a qualidade do produto, obtendo assim uma visão e confiança que as ações para alcançar a qualidade do produto estão funcionando. Existem três ferramentas para o SQA [162]:

- Revisão: inspeção dos documentos do projeto. Revisões são aplicadas em vários pontos durante a engenharia de *software* e servem para descobrir erros e defeitos que podem ser removidos. É uma ferramenta importante pois ao encontrar um erro no início do processo, será mais barato corrigi-lo. As revisões economizam tempo, reduzindo a quantidade de retrabalho que será exigida no final do projeto [39].
- Auditoria: auditorias planejadas e espontâneas para checar a aderência dos procedimentos de trabalho e para verificar o progresso do projeto;
- Medições: tirar conclusões sobre a adesão ao processo com base em medições. Observe que o grupo de SQA não está coletando dados, mas usando dados brutos e analisados.

A KA Teste de *Software* consiste na verificação dinâmica do comportamento de um programa com um conjunto finito de casos de testes, selecionados de um domínio geralmente infinito de execuções, para confirmar o comportamento especificado esperado.

Os processos de verificação e validação determinam se os produtos de desenvolvimento de uma determinada atividade estão em conformidade com os requisitos dessa atividade e se o produto satisfaz seu uso pretendido e as necessidades do usuário. A verificação é uma tentativa de garantir que o produto seja construído corretamente, no sentido de que os produtos de saída de uma atividade atendem às especificações impostas a eles em atividades anteriores. A validação é uma tentativa de garantir que o produto certo seja construído, ou seja, o produto atenda ao seu propósito específico [77]. Teste de *software* é uma das atividades de verificação e validação. Existem outros métodos para garantir a qualidade, como verificação baseada em modelo ou prova de teorema, mas a maioria dos desenvolvedores prefere o teste porque é mais simples em conceito e mais fácil de implantar [163].

A motivação do teste de *software* é afirmar a qualidade do *software* com métodos que possam ser aplicados de maneira econômica e eficaz tanto em larga escala como em pequena escala [39]. O teste de *software* é uma área muito ampla, que envolve muitas outras áreas técnicas e não técnicas, como especificação, design e implementação, manutenção, processos e gerenciamento de problemas em engenharia de *software* [164].

Durante o desenvolvimento, os testes podem ser realizados em diferentes níveis de granularidade [61], [164]:

- Teste de unidade: onde unidades de programa individuais ou classes de objeto são testadas, concentrando-se em testar a funcionalidade de objetos ou métodos;
- Teste de componentes: onde várias unidades individuais são integradas para criar componentes, concentrando-se no teste das *interfaces* dos componentes;
- Teste de sistema: onde alguns ou todos os componentes de um sistema são integrados e o sistema é testado como um todo, concentrando-se na interações entre os componentes;
- Teste de aceitação: é feito quando o sistema completo é entregue pelos desenvolvedores aos clientes ou usuários. O objetivo do teste de aceitação é dar confiança de que o sistema está funcionando, em vez de encontrar erros.

A ampla aplicação da Internet e da computação móvel aumentou significativamente a dependência de sistemas habilitados por *software*. Essa dependência gera preocupações críticas sobre confiabilidade e segurança de *software*, porque uma falha de *software* pode levar a consequências desastrosas [165]. O teste destina-se a mostrar que um programa faz o que se pretende fazer e descobrir defeitos do programa antes de ser colocado em uso [61]. Porém, é importante citar uma frase de Edsger W. Dijkstra [166]: "testes de programas podem ser usados para mostrar a presença de *bugs*, mas nunca para mostrar sua ausência!". Embora essa frase indique um limite da utilidade dos testes, também indica uma das propriedades mais úteis dos testes: encontrar *bugs* [167].

A detecção tardia de falhas e erros normalmente leva a enormes custos de correção e manutenção. Os esforços de garantia de qualidade, especialmente os esforços de teste, geralmente consomem mais de 50% dos esforços gerais de desenvolvimento [168], [165], [164], [169], [170], [171]. Em um estudo citado por Harman e McMinn [172], informa que em 2002 o National Institute of Standards and Technology (NIST) estimou o custo de falha de *software* para a economia dos EUA em 60 bilhões de dólares, o que correspondia a 0,6% do PIB na época. O mesmo relatório informa que mais de um terço desses custos de falha de *software* poderiam ter sido eliminados com uma infraestrutura de testes adequada. Isso reforça a importância dos esforços de garantia de qualidade dos *softwares* desenvolvidos.

O estudo realizado por Elberzhager et al [168], identifica abordagens existentes para reduzir o esforço de teste sem que o *software* final perca qualidade. A maioria dos artigos analisados, cerca de 50%, discute a automação de testes. Este resultado não é surpreendente, porque ao aplicar abordagens de automação de teste, muito tempo de teste pode ser salvo, reduzindo o custo do teste de *software* [169]. A automação permite mais ciclos de teste devido a testes repetíveis e execuções de teste mais frequentes. Também facilita a verificação rápida e eficiente de mudanças de requisitos e correções de erros e minimiza os erros humanos [165], [169].

Na literatura são descritas diversas técnicas para geração de casos de testes e dados para testes, como [173], [169], [174], [170], [172], [175], [171]: técnicas de geração aleatória de dados de teste, que escolhem entradas arbitrariamente até que entradas úteis sejam encontradas. Técnicas de geração de dados de teste simbólico, que criam expressões algébricas para várias restrições no programa, atribuindo valores simbólicos à variável. A execução simbólica para geração automática de testes possui dois tipos de grupos: geração de teste estático, feita sem executar o programa; e técnica de geração dinâmica, que determina quais casos de teste atendem ao requisito durante a execução do programa. Existe também a técnica de geração automática de testes baseada em busca, onde o domínio de entrada do sistema sendo testado é usado como espaço de pesquisa e são selecionados os dados que atendem ao objetivo da geração. O processo para encontrar a melhor solução possível entre todas as disponíveis é conhecido como otimização. Esta técnica de geração de testes pode utilizar diversos algoritmos, como: subida de encosta, algoritmos genéticos (GA), Ant Colony Optimization (ACO), Bee Colony Algorithm (BCA), Harmony Search (HS), Particle Swarm Optimization (PSO), programação dinâmica, entre outros algoritmos utilizados para solucionar problemas de busca, que geralmente utilizam heurísticas.

Para a geração de testes desenvolvida neste trabalho foi usada a técnica Property-based Testing (PBT), que tornou-se uma técnica padrão para melhorar a qualidade do *software* em uma ampla variedade de linguagens de programação [176]. PBT é uma técnica de teste de caixa-preta, semi-automática, de alto nível na qual, ao invés de escrever muitos testes de unidade manualmente, o testador simplesmente especifica propriedades gerais que o sistema sob teste (System under test - SUT) deve satisfazer, e geradores que produzem entradas aleatórias bem distribuídas para as partes do sistema que são testadas. Como em todas as técnicas de testes aleatórios, a chance de encontrar um erro e a confiança no SUT aumentam com o número de testes gerados [177].

PBT é uma técnica que está recebendo muita atenção recentemente, é fortemente relacionada a testes baseados em modelos, mas onde o foco está na geração de testes individuais automaticamente a partir de especificações na forma de testes de propriedades

de alto nível [178]. PBT é uma estratégia que leva a uma família de poderosas ferramentas de teste automáticas. Geralmente, são ferramentas para geração e execução de casos de teste com base nas especificações. Geralmente trabalhando como uma biblioteca, elas permitem que o desenvolvedor/testador anote as especificações do programa, usando uma linguagem de programação de sua escolha, na forma de propriedades que devem ser satisfeitas. A partir dessas especificações, as ferramentas geram, executam e verificam automaticamente os resultados de um grande número de casos de teste aleatórios para ver se as propriedades são mantidas [179]. Existe uma variedade de ferramentas PBT que surgiram ao longo dos anos inspiradas no QuickCheck original para Haskell [180], como: JUnit-QuickCheck para Java, theft para C, ScalaTest and ScalaCheck para Scala [181].

Ao contrário dos métodos de teste convencionais, em que o comportamento de um programa é testado apenas em alguns casos pré-determinados, o PBT enfatiza a definição de propriedades e, em seguida, testa sua validade em relação a pontos de dados amostrados aleatoriamente. Como o teste baseado em propriedade (PBT) usa um pequeno número de pontos de dados amostrados aleatoriamente, ele ainda fornece apenas uma resposta aproximada à questão de saber se uma propriedade é satisfeita em todos os pontos de dados. No entanto, pode fornecer mais confiança do que o teste unitário convencional, porque os pontos de dados amostrados aleatoriamente podem cobrir casos problemáticos que não foram previstos pelos desenvolvedores [181].

PBT apresenta três vantagens sobre o teste manual [163]. Primeiro, escrevendo apenas propriedades e omitindo dados de teste, os desenvolvedores podem reduzir significativamente o tamanho e complexidade do código de teste. Segundo, ao executar testes baseados em propriedade contra entradas geradas aleatoriamente, os desenvolvedores podem encontrar mais *bugs* do que testes manuais, pois os testes baseados em propriedade podem verificar propriedades com muitas entradas, então a chance de encontrar *bugs* é aumentada. Finalmente, as propriedades podem ser vistas como uma especificação formal leve do programa. Escrever especificações formais promove o entendimento da especificação e da implementação e é eficaz para compartilhar as especificações entre a equipe. É importante que essa especificação seja executável. Essa especificação executável pode evitar incompatibilidade entre especificação e implementação, o que geralmente é o caso se a especificação for gravada em um documento separado.

O PBT é uma abordagem poderosa para testes de *software*, mas pode ser tão desafiadora para desenvolvedores que a usam pela primeira vez quanto para os *stakeholders* não técnicos entenderem, em geral, os artefatos gerados. Abordagens ágeis para o desenvolvimento de *software* demonstram que o envolvimento de todas as partes interessadas no ciclo de vida do *software* é um fator-chave para o sucesso do projeto. Especificamente, permitir que clientes ou usuários validem os requisitos de *software* por inspeção direta

ou mesmo interação com artefatos de garantia de qualidade de *software*, como casos de testes e conjuntos de testes, traz benefícios efetivos [182]. Neste trabalho, para facilitar o envolvimento dos *stakeholders*, o PBT é usado para gerar especificações de teste baseadas em Cucumber [183], possibilitando o entendimento de todos os envolvidos sobre o que está sendo testado.

Capítulo 3

Arquitetura Proposta

O processo de elaboração dos Quadros de Dotação de Material (QDM) para uma determinada Organização Militar (OM) leva em consideração o seu Quadro de Cargos (QC), que é o documento que detalha os cargos que preenchem a estrutura organizacional de cada OM operativa [184]. Várias regras de distribuição de materiais, conhecidas internamente como chamadores, são definidas para identificar quais Materiais de Emprego Militar (MEM) devem ser providos a cada OM. Através do cruzamento destas regras negociais com o QC de um tipo de OM é gerado o QDM.

Outro artefato importante gerado pelo Sistema de Dotação de Material (SISDOT) é o Quadro de Dotação de Material Previsto (QDMP). Este documento indica quais materiais cada militar de determinada OM física deverá receber. Sua geração leva em consideração o Quadro de Cargos Previstos (QCP) e o QDM da OM selecionada.

Neste capítulo serão apresentados detalhes da implementação do Sistema de Dotação de Material (SISDOT) e encontra-se dividido nas seguintes seções: a Seção 3.1 apresenta as decisões arquiteturais tomadas durante o processo de desenvolvimento, assim como algumas das principais visões arquiteturais de forma a facilitar o entendimento do sistema. A Seção 3.2 apresenta detalhes da implementação de chamadores. A Seção 3.3 apresenta informações sobre a implementação de QDM. A Seção 3.4 apresenta informações sobre a implementação de QDMP. A seção 3.5 apresenta detalhes da implementação de testes, incluindo a geração automática de testes.

3.1 Arquitetura

O processo de desenvolvimento utilizado foi baseado na metodologia ágil [185], [61], [39]. Foram impostas algumas restrições arquiteturais para o desenvolvimento do SISDOT, entre elas:

- Uso da linguagem de programação Java (versão 8) [186], [187], [188], seguindo o padrão Java EE [189],[190],[191] (JSF, CDI, EJB, JPA);
- Uso do Primefaces [192], [193] como biblioteca de componentes JSF [194], [195];
- Uso do Sistema Gerenciador de Banco de Dados Oracle [196], [197];
- Uso do servidor de aplicações Wildfly [198], [199], [200];
- Uso do Maven como ferramenta de *build* [201], [202];
- Uso do Git como ferramenta de controle de versão [203],[204].

O estilo arquitetural do SISDOT é o cliente-servidor, onde os componentes interagem requisitando serviços de outros componentes. A estrutura de execução do SISDOT está organizada em agrupamentos lógicos de componentes conhecidos como *tiers*, mecanismo frequentemente utilizado em estilos arquiteturais cliente-servidor. Com isso, o SISDOT pode ser caracterizado como seguidor desse padrão arquitetural. Uma visão arquitetural é uma representação de um conjunto coerente de uma estrutura arquitetural, mostrando seus elementos, as relações entre eles e suas propriedades. A seguir serão mostradas as visões de casos de uso, de módulos e de *runtime* do SISDOT.

Visão de Casos de Uso

De acordo com Larman [205], casos de uso são requisitos, embora nem todos os requisitos do sistema. Os casos de uso são principalmente requisitos funcionais, que indicam o que o sistema fará. Casos de uso são documentos de texto, não diagramas, e a modelagem de caso de uso é primariamente um ato de escrever texto, não de desenhar [205]. No entanto, a Unified Modeling Language (UML) define um diagrama de casos de uso para ilustrar os nomes de casos de uso, atores e seus relacionamentos.

A Figura 3.1 apresenta um diagrama de casos de uso onde é possível observar os principais casos de uso do SISDOT. Os casos de uso foram especificados em documentos próprios e não serão detalhados neste trabalho, apenas um resumo dos principais casos de uso será apresentado.

O caso de uso Manter Cadastro compreende diversos casos de uso destinados à manutenção do cadastro de entidades utilizadas no sistema, como: Tipo de Material, Classe de Material e Família de Material. Deve ser possível associar uma Classe de Material a um ou mais Tipos de Material cadastrados. Deve ser possível associar uma Família de Material a uma Classe de Material e um Tipo de Material.

O caso de uso Manter Material de Emprego Militar (MEM) é destinado ao cadastro, edição, exclusão e pesquisa de um Material de Emprego Militar (MEM). O conceito de

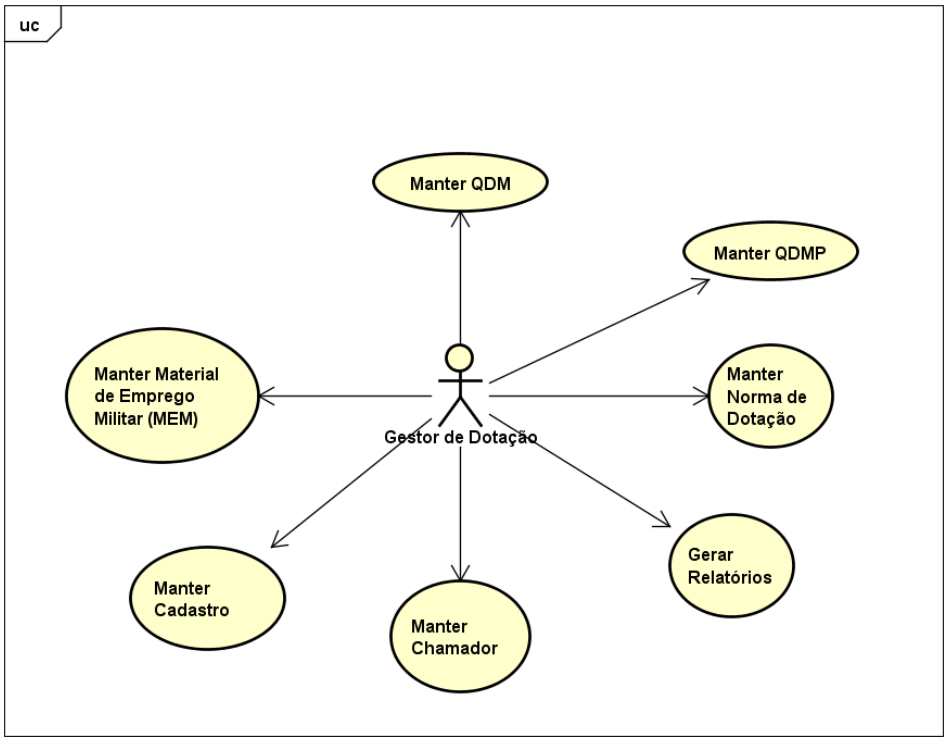


Figura 3.1: *Visão de Casos de Uso do SISDOT*

MEM envolve obrigatoriamente um código, uma descrição, um Tipo de Material, uma Classe de Material e uma Família de Material; e opcionalmente, Palavras-chave, Forma de Uso e MEM Essencial.

O caso de uso Manter Chamador permite a manutenção do cadastro de chamadores, que serão utilizados para a geração de Quadros de Dotação de Material (QDM). Os casos de uso Manter QDM e Manter QDMP especificam como deverá ser feita a manutenção do cadastro de Quadro de Dotação de Material (QDM) e Quadro de Dotação de Material Previsto (QDMP), respectivamente.

O caso de uso Manter Norma de Dotação é destinado ao cadastro, edição, exclusão e pesquisa de uma Norma de Dotação. As Normas de Dotação são regras que orientam o estabelecimento das dotações dos diversos MEMs por cargos e frações. Quando esta Norma de Dotação é padronizada, ou seja, tem um alcance generalizado para um cargo e/ou fração, a Norma de Dotação deve descrever essa intenção.

O caso de uso Gerar Relatórios é destinado, principalmente, à geração de relatório de Material de Emprego Militar (MEM). Permite a seleção de filtros (Tipo de Material, Classe de Material e Família de Material) e a geração de relatórios nos formatos PDF, planilha e documento.

A Figura 3.2 apresenta a tela inicial do Sistema de Dotação de Material (SISDOT), possibilitando o acesso rápido às principais funcionalidades implementadas no sistema.

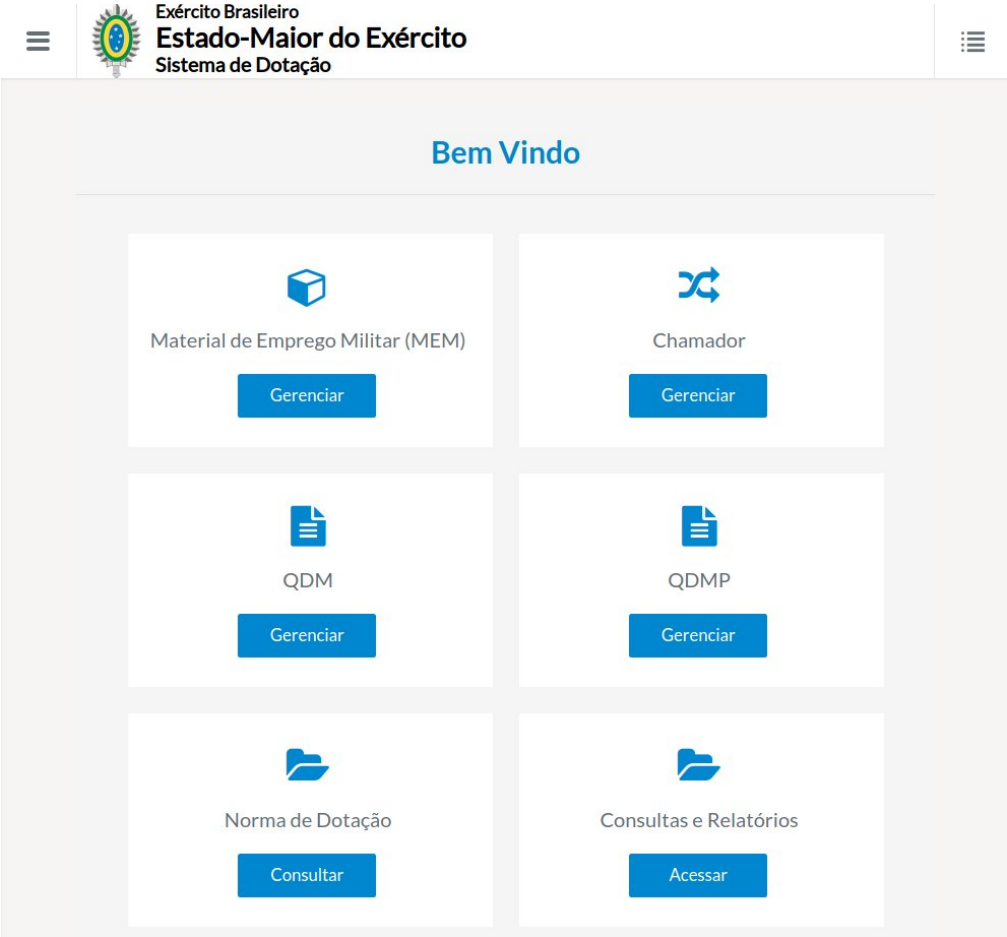


Figura 3.2: *Tela inicial do SISDOT*

Visão de Módulos

O desenvolvimento do sistema foi dividido em vários módulos, conforme a Figura 3.3 e que serão descritos a seguir. Os módulos representados com a cor amarela fazem parte do SISDOT, enquanto os módulo com a cor verde são módulos externos usados pelo SISDOT e pelo Sistema de Boletins (SISBOL), ambos sistemas no âmbito do Projeto de Pesquisa para Validação de Práticas e Métodos de Desenvolvimento de Software para o Exército Brasileiro (PROMISE-EB).

O módulo unb-core possui abstrações que visam facilitar a declaração de Entidades, Data Access Object (DAO) e Serviços. O módulo unb-cache possui abstração para o uso de *cache*, de acordo com a Java Temporary Caching API (JSR107), que especifica a Application Programming Interface (API) e a semântica para armazenamento temporário em *cache* de memória de objetos Java, incluindo criação de objetos, acesso compartilhado, *spool*, invalidação e consistência na Java Virtual Machine (JVM). O módulo unb-jsf possui abstrações que visam facilitar a utilização de JavaServer Faces (JSF).

O módulo sisdot-comum possui classes utilitárias, interfaces e anotações usadas pelos

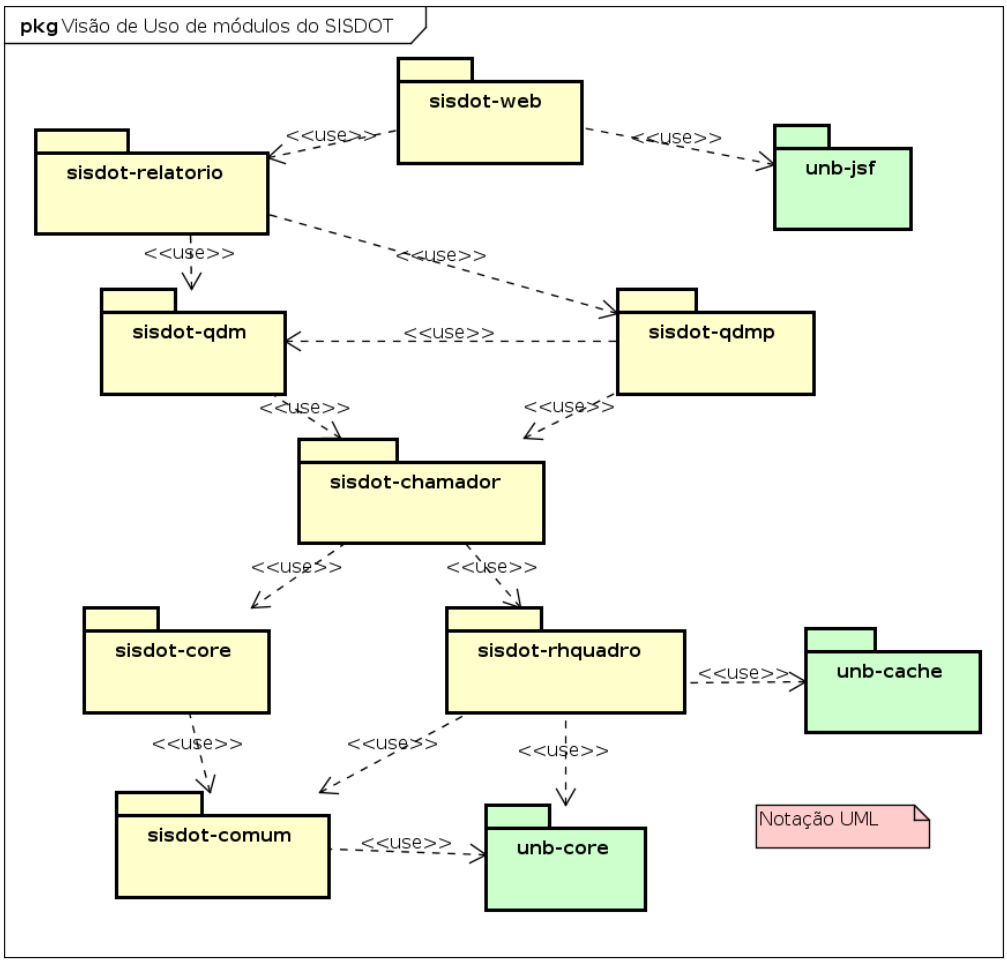


Figura 3.3: *Visão de Uso de Módulos do SISDOT*

outros módulos. Entre as mais importantes estão: anotações que servem como qualificadores na utilização do Contexts and Dependency Injection (CDI) para distinguir as unidades de persistência do Java Persistence API (JPA); anotação para persistir automaticamente o histórico de alterações nas entidades determinadas; classe para formatação de códigos numéricos.

O módulo sisdot-core contém o núcleo do sistema. Todas as entidades são definidas nesse módulo pois o JPA não reconhece, de forma fácil, entidades definidas em outros Java Archive (JAR) implantados no mesmo Web Application Archive (WAR), além de facilitar a implementação de testes de integração. Neste módulo, são definidos repositórios e serviços relativos a algumas entidades, como: Tipo de Material, Classe de Material, Família de Material e Material de Emprego Militar (MEM), incluindo todas as entidades relacionadas.

O módulo sisdot-rhquadro serve para recuperar informações contidas no banco de dados corporativo do Exército Brasileiro (EB), em geral relacionadas a Organizações

Militares, Quadro de Cargos e Quadro de Cargos Previstos. Este módulo serve como uma abstração ao acesso aos dados que inicialmente está sendo realizada por acesso direto ao banco de dados, mas que futuramente deverá ser realizada através de *web services*. Assim, o módulo foi projetado de forma a facilitar essa alteração já planejada previamente, bastando alterar a forma de recuperação dos dados, mas mantendo intacto o contrato com os outros módulos.

Foram desenvolvidas consultas para possibilitar o resgate de informações essenciais ao desenvolvimento dos outros módulos. Essas consultas são realizadas em SQL (Structured Query Language) nativo e os resultados são transformados em VOs (Value Objects), que são objetos utilizados para representar uma entidade retornada pela consulta. Para utilizar os serviços criados deve ser declarada a dependência com o sisdot-rhquadro no arquivo POM (Project Object Model) do módulo que irá utilizar as consultas. Após a declaração da dependência, os serviços do sisdot-rhquadro podem ser injetados, através do uso do CDI, e utilizados nos serviços do módulo dependente.

O módulo sisdot-chamador é responsável pela manutenção de chamadores e disponibilização de consultas para outros módulos. Foram desenvolvidos dois tipos de chamadores: um específico para uma OM tipo e outro genérico, que pode servir para todas as OMs tipo que atenderem às regras especificadas no chamador. Detalhes sobre chamadores podem ser vistos na Seção 3.2.

O módulo sisdot-qdm é responsável pela manutenção do cadastro e geração de QDMs. Um QDM é específico para uma OM-tipo, caso ocorra alguma mudança na estrutura dessa OM-tipo um novo QDM deve ser gerado. Um QDM é composto pela lista de MEM, com sua quantidade e qual cargo deve receber esse material. Detalhes sobre Quadro de Dotação de Material (QDM) podem ser vistos na Seção 3.3.

O módulo sisdot-qdmp é responsável pela manutenção do cadastro e geração de QDMPs. Assim como os módulos sisdot-chamador e sisdot-qdm, este módulo define VOs, DAOs e serviços específicos. Embora não exista uma restrição de uso explícita, como é comum em módulos Open Services Gateway Initiative (OSGi) ou a partir do sistema de módulos do Java 9, há uma restrição ao uso apenas de serviços e VOs dos módulos dependentes. A Seção 3.4 apresenta detalhes sobre Quadro de Dotação de Material Previsto (QDMP)

O módulo sisdot-relatorio possibilita a geração de relatórios customizados, de acordo com os filtros selecionados. Além de receber os filtros selecionados como parâmetro, o serviço de geração de relatórios permite a escolha, através de outro parâmetro, do formato de saída do relatório, que pode ser: PDF, planilha ou documento.

O módulo sisdot-web contém as paǵinas JSF, *scripts* javascript, folhas de estilo (CSS), imagens e outros artefatos necessários para a correta renderização das páginas no *browser* do usuário. Contém também os MBs responsáveis pelo controle das ações dos usuários

nas páginas. O artefato final deste módulo é um arquivo WAR contendo todos os arquivos necessários para a implantação do sistema em um servidor de aplicações, o Wildfly no caso.

Visão de Runtime

Conforme mencionado na Seção 2.2, algumas estruturas arquiteturais são dinâmicas, chamadas de *component-and-connector* (C&C), e focam na forma como os elementos interagem uns com os outros em tempo de execução (*runtime*). De forma a detalhar e ilustrar a arquitetura do SISDOT, será apresentada a visão de *runtime* do caso de uso Cadastrar QDM, conforme a Figura 3.4.

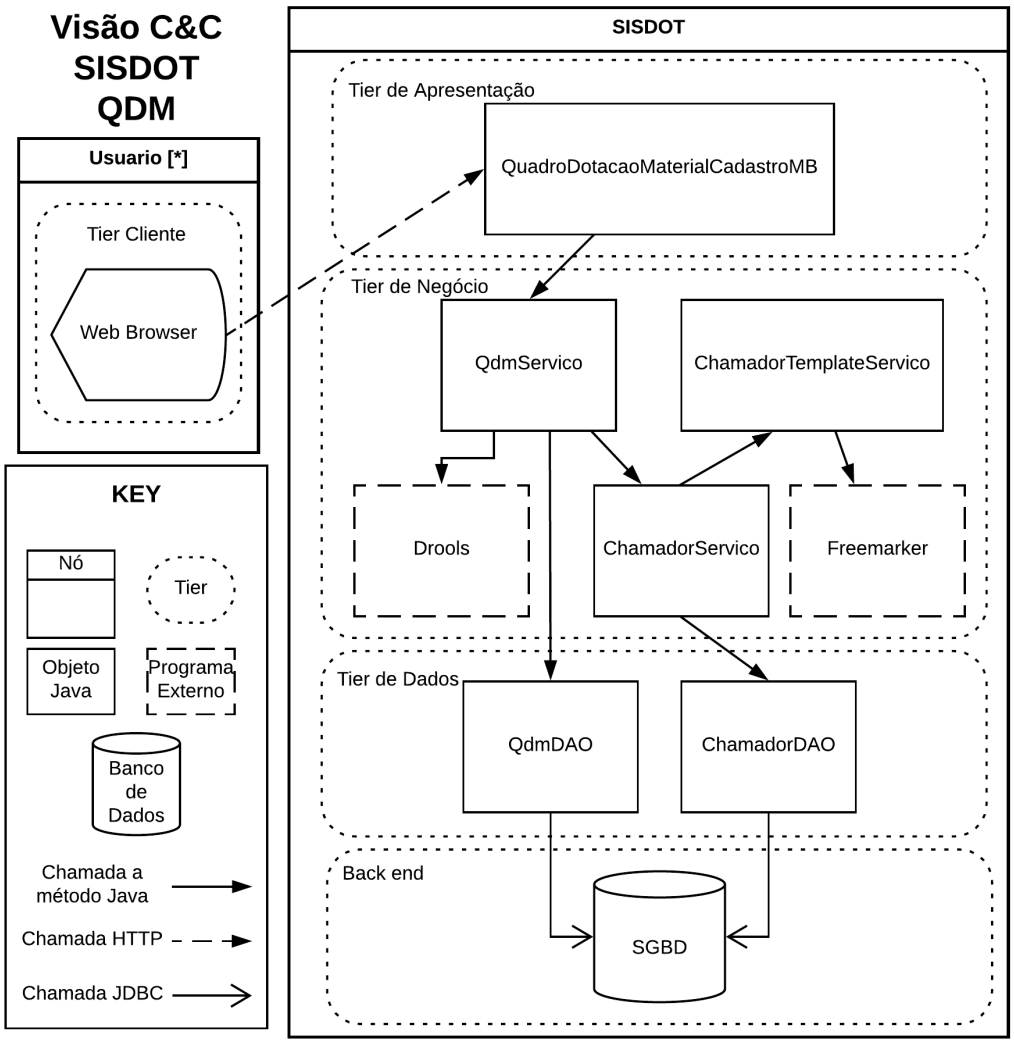


Figura 3.4: *Visão de Runtime de geração de QDM*

Considerando a interação do usuário com o SISDOT em seu navegador *web*, um especialista em distribuição de materiais do Exército Brasileiro (EB) deve estar autenticado e com o perfil que permite o cadastro de QDM. Ao acessar a página JSF de cadastro de QDM, o usuário deve selecionar para qual Quadro de Cargos (QC) o QDM deve ser gerado. Ao selecionar a opção para gerar o QDM com o uso de chamadores previamente cadastrados, uma requisição é enviada para o servidor. Estas ações são realizadas na *tier* cliente. A requisição é recebida pela *tier* de apresentação do SISDOT, mais especificamente por um *managed bean*, uma espécie de *controller* do JSF. Podem ser realizadas algumas validações nos dados recebidos antes de serem repassados para a *tier* de negócio. Nessa *tier* o serviço que possui as regras negociais de QDM é invocado para que este controle o processo de geração de QDM, que inclui a recuperação dos chamadores cadastrados, transformação destes em Drools Rule Language (DRL), a execução dessas regras de forma a popular o QDM e a persistência do QDM em banco de dados.

O processo de geração de QDM é iniciado com a recuperação de uma lista de chamadores previamente cadastrados. O componente QdmServico realiza uma chamada ao método de recuperação de chamadores do componente ChamadorServico, que por sua vez utiliza o ChamadorDAO para realizar a consulta no banco de dados e retornar a lista dos chamadores recuperados. O acesso ao banco de dados é feito com o auxílio do JPA.

A lista de chamadores cadastrados deve ser então convertida em um arquivo DRL, de maneira a ser interpretado pelo *rule engine* (ver Seção 2.3). Para isso, cada chamador é convertido individualmente, utilizando o serviço ChamadorTemplateServico, em uma regra no formato DRL correspondente ao chamador sendo convertido. Este componente utiliza um *template engine* (ver Seção 2.4.1), o Freemarker, para realizar essa conversão e retorna o arquivo contendo todos os chamadores convertidos e outras declarações essenciais contidas no *template*, como a importação das classes necessárias.

Após o arquivo DRL ter sido gerado, o arquivo de regras é compilado pelo Drools (ver Seção 2.3.1), uma sessão é criada. O QC para o qual o QDM está sendo gerado é inserido como um fato na *working memory* e as regras são disparadas. À medida que as regras são acionadas o QDM vai sendo populado. O QDM gerado é então persistido no banco de dados pelo QdmDAO e uma mensagem de confirmação é mostrada na tela do usuário que iniciou a requisição.

3.2 Chamador

As regras de distribuição de materiais são regras de negócio do Sistema de Dotação de Material (SISDOT) que identificam quais Materiais de Emprego Militar (MEM) devem

ser providos a cada unidade organizacional (por exemplo, diretorias, departamentos, batalhões e cargos militares). Essas regras são conhecidas internamente como chamadores.

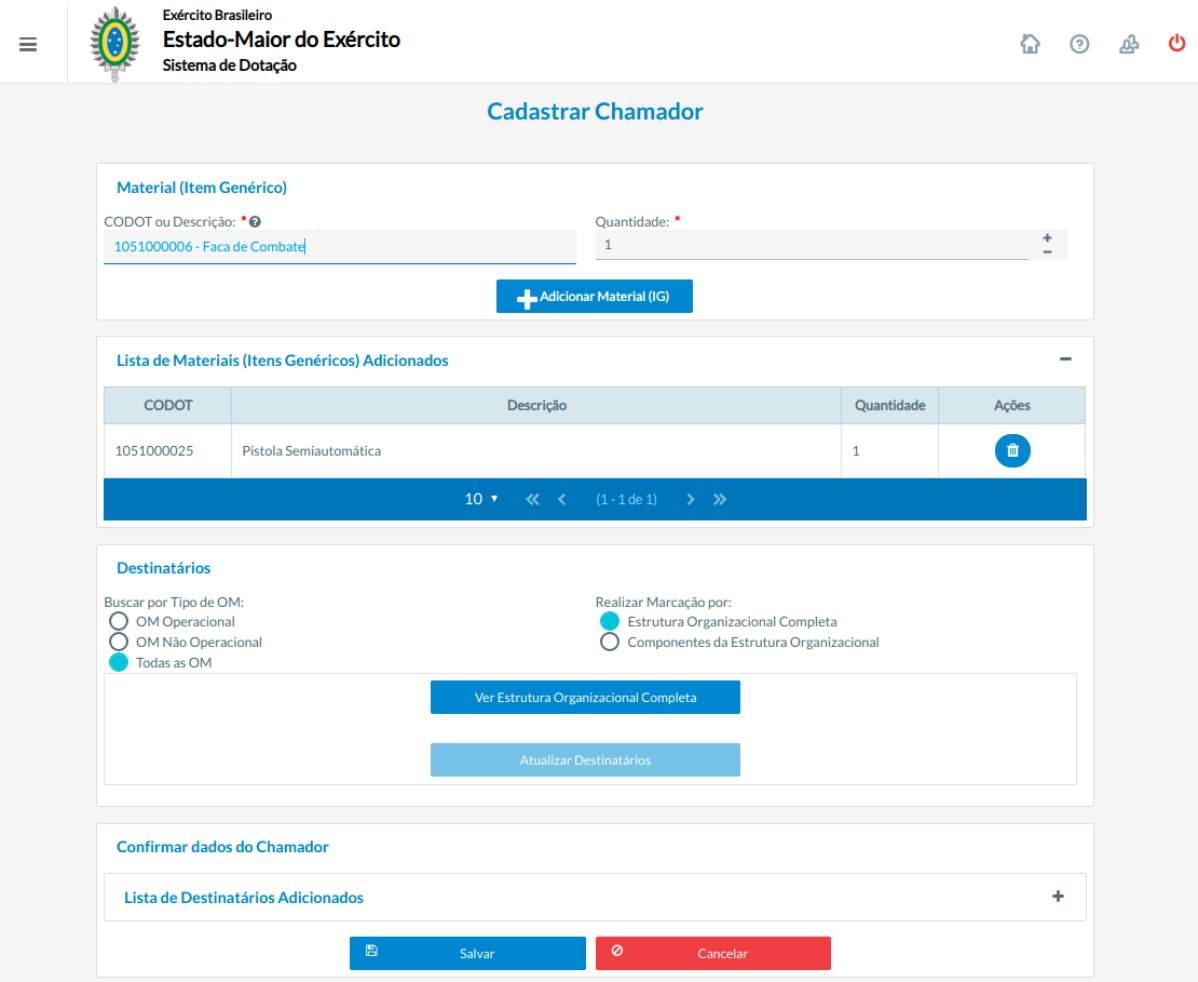


Figura 3.5: *Tela de cadastro de Chamador*

Um chamador, independente de seu tipo, está relacionado a um ou mais MEMs e define suas respectivas quantidades. Um chamador possui uma ou mais regras de distribuição. Cada regra possui um tipo, um valor, uma descrição associada e uma informação que determina se a regra é uma afirmação ou negação. Por exemplo: uma regra do tipo fração possui como valor o identificador da fração e como descrição o nome da fração. Existem vários tipos de regra, como: tipo de OM, natureza, sub-natureza, valor da OM, fração, arma, sub-círculo, patente, habilitação. Alguns tipos de regra referem-se apenas a dados do QC, outros fazem referência apenas a dados de frações ou de cargos.

Para uma melhor utilização do banco de dados, ao evitar a definição de inúmeras colunas que ficariam inevitavelmente em branco em cada regra, a regra é gravada no banco de dados no formato JSON (JavaScript Object Notation) e convertida para objeto

novamente quando da sua recuperação. O conjunto dessas regras define exatamente quem deve receber os MEMs especificados no chamador.

Na Figura 3.5 é apresentada a tela de cadastro de chamador. O usuário precisa estar autenticado e possuir as devidas permissões para acessar esta página. Esta tela permite que o usuário defina quais os MEMs, e suas quantidades, fazem parte do chamador sendo criado. No primeiro momento, o usuário tem acesso a materiais tanto de uso coletivo quanto individual. Ao selecionar o primeiro item, recebe um aviso na tela informando que a partir de então todos os MEMs selecionados deverão ser da mesma forma de uso (coletivo ou individual) dessa primeira escolha, então o sistema só trará opções de MEMs da mesma forma de uso já escolhida. O usuário deve selecionar um dos dois tipos de chamador existentes, que estão resumidos abaixo e serão detalhados nas subseções seguintes:

- Estrutura Organizacional Completa: este tipo de chamador, também conhecido como chamador em árvore, permite que o usuário navegue pela árvore de classificação de QCs para selecionar os cargos;
- Componentes da Estrutura Organizacional: este tipo de chamador, também conhecido como chamador em abas, permite que o usuário especifique de uma maneira mais genérica as regras do chamador.

Na Figura 3.6 é possível observar um trecho da tela de listagem de chamadores, onde é apresentado um chamador em abas previamente cadastrado. Este chamador determina que os militares que se adequem às regras impostas recebam os MEMs especificados. As regras que estão na cor vermelha são regras de negação, como: os militares que possuam a habilitação "Motorista de Viatura Blindada de Combate"não estão aderentes a este chamador e não receberão, por meio deste chamador, os MEMs especificados. Foram também especificadas regras de afirmação, como: o militar que possuir a graduação de Segundo-Sargento está aderente a este chamador. Em resumo: apenas os militares com patentes de sargento, cabo ou soldado, mas que não possuam as referenciações (Arma/QD/SV-QM), habilitações e cargos especificados receberão os MEMs especificados neste chamador. Os militares que não estão aderentes às regras especificadas neste chamador podem até chegar a receber os MEMs declarados, mas por meio de outro chamador que porventura venha a ser criado.

Chamador em Árvore

O chamador do tipo 'Estrutura Organizacional Completa', conhecido como chamador em árvore, possibilita que o usuário navegue pela árvore de classificação de QCs para selecionar os cargos. Com isso, o usuário pode definir chamadores bem específicos para

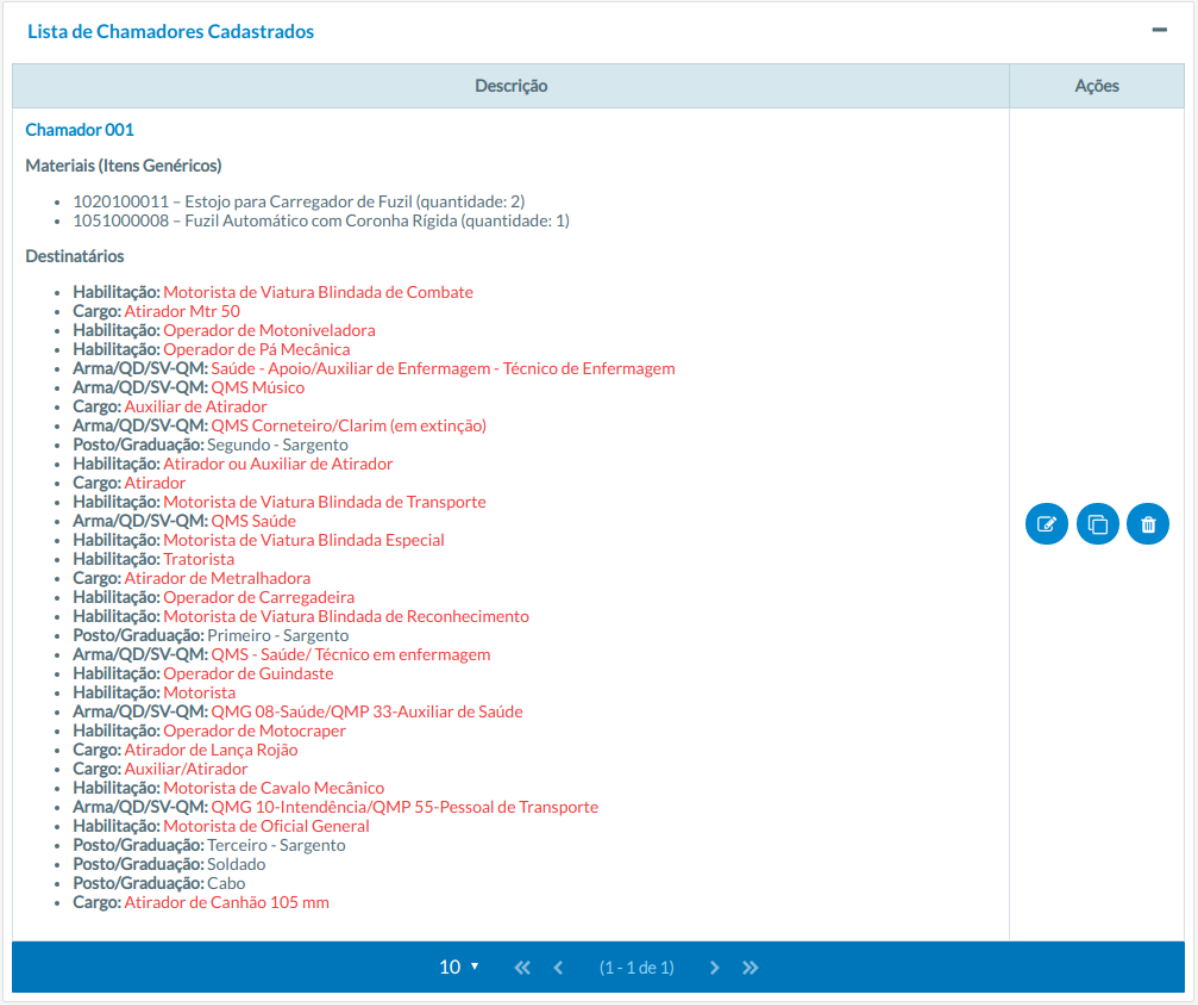


Figura 3.6: *Tela de listagem de Chamadores cadastrados*

cargos de determinados QCs. Este tipo de chamador permite apenas regras de afirmação, não havendo a possibilidade, então, de definir uma regra negando os MEMs para um cargo.

Para cada cargo selecionado é criada uma regra que contém o identificador do cargo dentro do QC e o identificador da fração, também dentro do QC, na qual o cargo está alocado. Existe uma diferença entre o cargo de um QC e um cargo genérico: fazendo um paralelo com a orientação a objetos, um cargo genérico é uma classe e um cargo do QC é uma instância de uma classe que é uma generalização do cargo genérico, herdando seus atributos e comportamento, mas incluindo novos atributos e comportamentos. Com isso, o cargo de QC, além de possuir um código herdado do cargo genérico, é identificado por uma chave composta pelos identificadores, específicos do QC, do cargo e da fração onde o cargo está contido.

Na Figura 3.7 é apresentado um trecho da tela de cadastro de chamador em árvore. Esta árvore representa a classificação de um QC, com a raiz sendo o tipo operacional e

passando pela natureza, subnatureza, grupo e valor, até chegar na estrutura organizacional do QC pretendido. Ao selecionar uma fração todos os cargos, inclusive de subfrações, são selecionados. Mas é possível que o usuário selecione apenas um subconjunto dos cargos de uma fração.

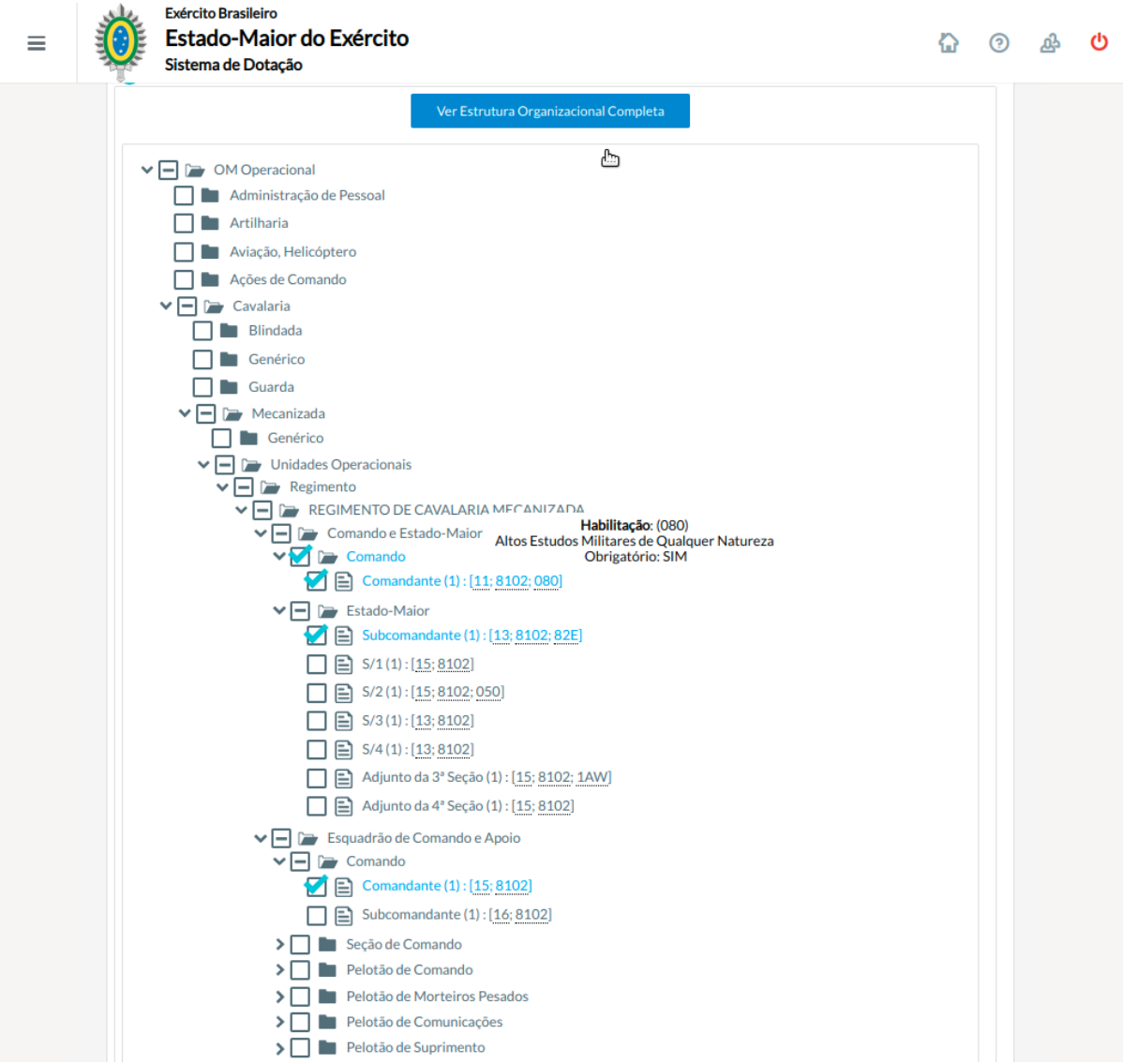


Figura 3.7: *Tela de Chamador em árvore*

Chamador em Abas

O chamador do tipo 'Componentes da Estrutura Organizacional', conhecido como chamador em abas, possibilita a especificação das regras de um chamador de uma forma genérica. Então, ao invés de definir um chamador específica para um QC, como no chamador em árvore, o usuário especifica regras que podem abranger cargos contidos em diversos QCs.

Neste tipo de chamador, ao contrário do chamador em árvore, é possível definir regras de afirmação assim como regras de negação também, quando necessário.

Na Figura 3.8 é apresentado um trecho da tela de cadastro de chamador em abas. Cada aba apresenta um tipo de regra que pode ser adicionado ao chamador. Existem diversos tipos, divididos em abas: natureza/subnatureza, valor, fração, cargo, sub-círculo, observações, posto, arma e habilitação. Todas as abas estão vinculadas entre si, por exemplo: ao definir uma regra afirmando que apenas OMs com natureza de Infantaria devem receber os MEMs especificados, nas outras abas só podem aparecer valores que existam nas OMs com natureza de Infantaria. Com isso, as possibilidades de valores que possam ser selecionados tendem a afunilar de acordo com as regras sendo definidas. Isso foi feito como uma forma de reduzir as chances de erro na hora de cadastro das regras, impossibilitando a criação de um chamador com regras inconsistentes.

Na Figura 3.8 é possível observar a definição de 3 regras: uma afirmando que as OMs com natureza de Infantaria devem receber os MEMs e as outras duas (em vermelho) negando subnaturezas específicas da natureza de Infantaria, no caso: Montanha e Selva. Com isso, apenas as OMs com natureza de Infantaria estão habilitadas a receber os MEMs definidos, exceto as que possuam subnatureza de Montanha ou Selva. Como esta figura apresenta apenas uma trecho da tela não é possível observar o restante da página, na qual é mostrada o restante das regras conforme o formato de lista definido.

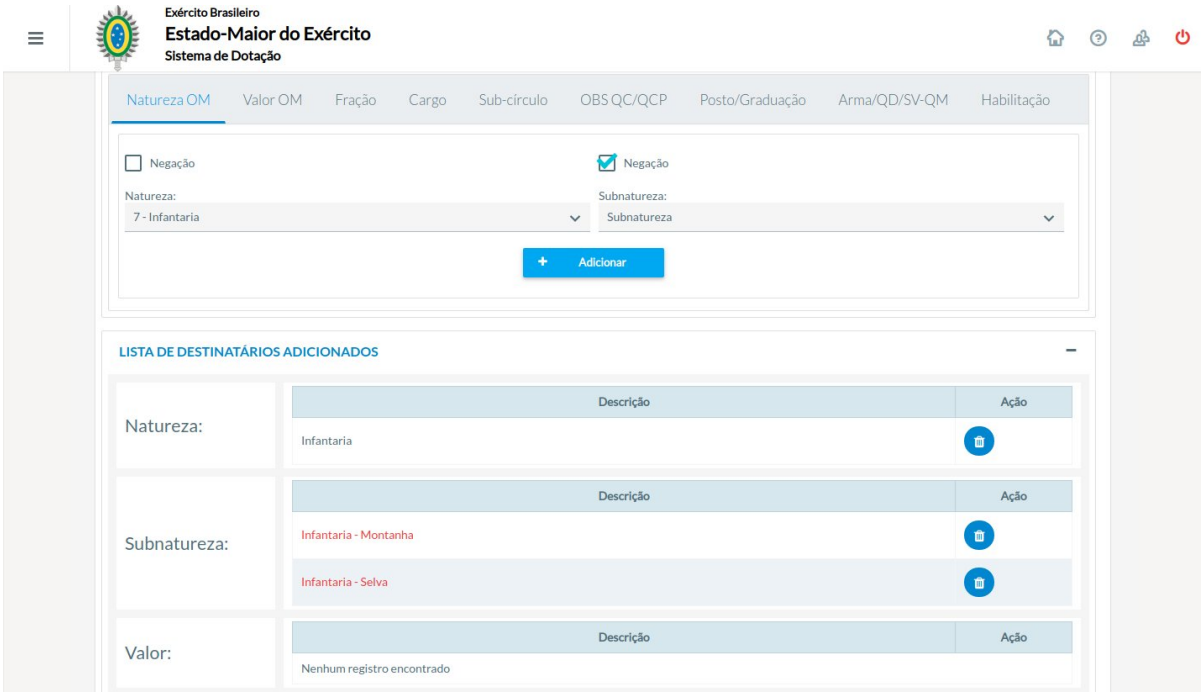


Figura 3.8: *Tela de Chamador em abas*

DSL para Chamador

Conforme apresentado na Subseção 2.4.2, foi desenvolvida uma Domain Specific Language (DSL) utilizando Xtext, visando a declaração de chamadores com a intenção de facilitar a criação de testes manuais e automáticos. A declaração de um chamador na linguagem Java tende a exigir a codificação de mais linhas de código do que a declaração do mesmo chamador utilizando a DSL desenvolvida. Esta DSL foi criada como um projeto separado do SISDOT e a sua utilização no sistema ou em outros projetos relacionados é condicionada à declaração das dependências necessárias nos respectivos POMs.

Para iniciar a implementação da DSL com o Xtext foi necessário definir uma gramática semelhante à ANTLR, conforme apresentada no Código 3.1. A partir dessa gramática são gerados o *lexer*, *parser*, o modelo AST (Abstract Syntax Tree), a construção da AST para representar o programa analisado e o editor do Eclipse com todos os recursos do Integrated Development Environment (IDE).

```
1  grammar br.unb.cic.sisdot.chamador.ChamadorDsl with org.eclipse.xtext.common.Terminals
2
3  generate chamadorDsl "http://www.unb.br/cic/sisdot/chamador/ChamadorDsl"
4
5  Model:
6     chamadores+=Chamador*;
7
8  Chamador:
9     'chamador' codigo=INT '{'
10       ('tipoOM' '=' tipo=TipoOM ',')?
11       'materiais' '=' '['itens+=Item
12          (',' itens+=Item)*
13       '],'
14       'regras' '=' '['regras+=Regra
15          (',' regras+=Regra)*
16       ']'
17     '}'
18 ;
19
20 Item:
21    '(' codot=STRING (':' nome=STRING)? ':' quantidade=INT ')'
22 ;
23
24 Regra:
25    '(' ('tipoChamador' '=' tipoChamador=TipoChamador ',')?
```

```
26      'tipo' '=' tipo=TipoRegra ','
27      'valores' '=' '[' valores+=STRING (',' valores+=STRING)* ']'
28   ')'
29 ;
30
31 enum TipoOM:
32    AMBOS | OPERACIONAL | NAO_OPERACIONAL;
33
34 enum TipoChamador:
35    AFIRMACAO | NEGACAO;
36
37 enum TipoRegra:
38    NATUREZA | SUB_NATUREZA | VALOR | FRACAO | ARMA | CARGO_QC
39    SUB_CIRCULO | POSTO | CARGO | HABILITACAO| OBSERVACAO;
```

Código 3.1: *Gramática Xtext que define a estrutura da DSL*

Um exemplo da definição de um chamador utilizando a DSL desenvolvida é apresentado no Código 3.2. É possível observar que a declaração deste chamador exigiu a codificação de 12 linhas de código, enquanto a declaração do mesmo chamador na linguagem Java exige 50 linhas de código, contando as 4 linhas em branco usadas para facilitar a leitura do mesmo.

O componente responsável pela geração de QDM trabalha apenas com objetos Java. Então foi desenvolvido um gerador de código no projeto de desenvolvimento da DSL. Esse gerador, que é uma generalização da classe AbstractGenerator do Xtext, é responsável pela conversão do modelo EMF contendo os chamadores declarados na DSL em uma classe Java que contém a representação Java desses chamadores. Este gerador é automaticamente executado quando o usuário edita a DSL no Eclipse ou quando da execução do *plugin* maven do Xtext. Como esse gerador está contido na biblioteca (JAR) da DSL, é possível sua invocação de outras formas também, conforme implementado pelo desenvolvedor.

```
1 chamador 4 {
2   materiais=[
3     ('1051000008':"Fuzil Automatico com Coronha Rigida":1),
4     ('1020100011':"Estojo para Carregador de Fuzil":2)
5   ],
6     regras=[
7     ( tipoChamador = NEGACAO , tipo = CARGO , valores = [ "2205" , "528" , "539" , "
        ↪ 364" , "2204" , "121" , "539" ] ),
8     ( tipo = POSTO , valores = [ "22" , "42" , "23" , "24" , "44" ] ),
```

```
9     ( tipoChamador = NEGACAO , tipo = HABILITACAO , valores = [ "748" , "903" , "751"
          ↪ , "769" , "782" , "765" , "921" , "749" , "767" , "793" , "750" , "762" , "
          ↪ 920" , "768" , "756" ] ),
10    ( tipoChamador = NEGACAO , tipo = ARMA , valores = [ "15" , "833" , "729" , "5110"
          ↪  , "5390" , "10" , "5393" , "12" , "5112" , "1055" , "5308" ] )
11  ]
12 }
```

Código 3.2: *Definição de chamador utilizando a DSL desenvolvida*

3.3 Quadro de Dotação de Material

Conforme apresentado na Seção 2.1, a logística militar é o conjunto de atividades relativas à previsão e à provisão de recursos humanos, materiais e animais, quando aplicável, e dos serviços necessários à execução das missões das forças armadas. No Exército Brasileiro (EB), as atividades de logística são agrupadas em sete funções logísticas. Este trabalho tem como foco a função logística suprimento, que refere-se ao conjunto de atividades que trata da previsão e provisão do material de todas as classes de material e peças de reparação usadas nos equipamentos necessários ao apoio logístico às organizações e às forças apoiadas.

Entre as atividades que apoiam a função logística suprimento está o planejamento de todas as ações relacionadas com o suprimento. O planejamento diz respeito à previsão e à provisão das necessidades correntes e futuras. A previsão dos suprimentos tem por objetivo responder à seguinte pergunta: Qual é a quantidade prevista ou necessária de cada Material de Emprego Militar?.

Este questionamento pode ser respondido através de um documento conhecido como Quadro de Dotação de Material (QDM), que contém informações sobre todos os MEM, e suas quantidades, que devem ser providos para cada militar de determinada Organização Militar (OM). Os departamentos determinam suas necessidades correntes de material tendo como referência a previsão existente no QDM [206]. O QDM é um dos principais artefatos gerados pelo SISDOT, servindo de base para outro importante artefato, o Quadro de Dotação de Material Previsto (QDMP).

A geração de um QDM, de forma resumida, dá-se da seguinte forma: o usuário seleciona o Quadro de Cargos (QC) para o qual deseja gerar o QDM. Então o sistema carrega os dados deste QC e também todos os chamadores ativos cadastrados. Cada chamador é transformado em uma regra Drools Rule Language (DRL). Um *container* do Drools (ver 2.3.1) é então instanciado, as regras DRL são compiladas e é aberta uma sessão do Drools. O objeto QC é inserido como um fato na *working memory*, uma classe auxiliar é definida

como global e as regras são então disparadas. Para cada regra efetivamente acionada é populada uma lista com os cargos que devem receber os MEM especificados no chamador referente à regra acionada. Então a classe auxiliar, definida como global, é usada para popular o QDM. Após a execução das regras o QDM está preenchido, de acordo com os chamadores cadastrados, e é persistido no banco de dados.

Esse processo depende da criação de forma correta dos chamadores, pois as regras referentes a cada chamador só serão acionadas se cumprirem todas as condições especificadas no chamador. Com isso, podem ocorrer conflitos entre chamadores (dotando um militar de uma quantidade errada de determinado MEM pois houve a soma das quantidades oriundas de dois ou mais chamadores) ou chamadores podem acabar não sendo acionados quando deveriam ser caso tivessem sido corretamente especificados.

Template para criar DRL

De acordo com o processo de geração de QDM informado acima, um dos passos contempla a transformação de chamadores em regras DRL, explicada a seguir.

Cada chamador é transformado em tempo de execução para o formato DRL, entendido pelo Drools. Essa transformação é feita através do uso de um *template engine* (ver 2.4.1), o Freemarker, que é uma biblioteca java capaz de gerar uma saída formatada aplicando dados dinâmicos seguindo regras pré-configuradas em um *template* estático. Foram definidos dois *templates* para realizar essa transformação.

Um dos *templates* é responsável pela definição da estrutura de um arquivo DRL, definindo o pacote, importações de classes necessárias, definição da variável global usada para auxiliar o preenchimento do QDM e as declarações dos chamadores, já convertidos. Um pacote é uma coleção de regras e outras construções relacionadas, como importações e *globals*. Os membros do pacote são tipicamente relacionados uns aos outros, por exemplo regras de um mesmo domínio. Um pacote representa um *namespace*, que idealmente é mantido exclusivo para um determinado agrupamento de regras. O nome do pacote em si é o *namespace* e não está relacionado a arquivos ou pastas de forma alguma [106].

O outro *template*, apresentado no Código 3.3, é responsável por converter cada chamador em uma regra no formato DRL. O conjunto de chamadores convertidos é usado no *template* apresentado anteriormente para a geração do arquivo DRL. Os dados utilizados por este *template* são previamente processados de forma a agrupar os tipos de regras existentes no chamador em um mapa, além de realizar tratamentos específicos para o QC em questão, como a recuperação de códigos de frações filhas no caso de haver uma regra do tipo fração. Então, o chamador e o mapa são passados para o Freemarker, que executa o *template*. O *template* verifica a presença de determinadas entradas no mapa e preenche a

regra de acordo com as chaves encontradas. O resultado da execução deste *template* pode ser visto no Código 3.2.

```
<#if chamador.tipoOM == "AMBOS">
  <#assign operacional = "(operacional == TipoOrganizacaoMilitar.OPERACIONAL ||
      ↪ operacional == TipoOrganizacaoMilitar.NAO_OPERACIONAL)">
<#else>
  <#assign operacional = "operacional == TipoOrganizacaoMilitar.${chamador.tipoOM}">
</#if>

<#assign possuiRegraQc = false >
<#if mapa['NATUREZA']?? || mapa['SUB_NATUREZA']?? || mapa['VALOR']?? >
  <#assign possuiRegraQc = true >
</#if>

rule "Chamador ${chamador.codigo}"
  when
    <#if possuiRegraQc >
    $qc: QuadroDeCargosVO(
        ${operacional}
        <#if mapa['IDs_QCs']??>
        , ${mapa['IDs_QCs']}
        </#if>
        <#if mapa['NATUREZA']??>
        , ${mapa['NATUREZA']}
        </#if>
        <#if mapa['SUB_NATUREZA']??>
        <#if mapa['NATUREZA']??>||<#else>,</#if> ${mapa['SUB_NATUREZA']}
        </#if>
        <#if mapa['VALOR']?? >
        , ${mapa['VALOR']}
        </#if>
      )
    <#else>
    $qc: QuadroDeCargosVO( ${operacional}
        <#if mapa['IDs_QCs']??>
        , ${mapa['IDs_QCs']}
        </#if>)
    </#if>
    $cargosFiltrados: List( size > 0 ) from accumulate (
```

```
37      $fracao: FracaoQcVO( <#if mapa['FRACAO']?? > ${mapa['FRACAO']}</#if> <#if mapa['
            ↪ IDs_Fracoes_QCs']?? > ${mapa['IDs_Fracoes_QCs']}</#if> ) from $qc.fracoes
38      and
39      $cargo: CargoVO(
40        cargoFuncao == "CARGO"
41        <#if mapa['CARGO_QC']?? >
42        , ${mapa['CARGO_QC']}
43        </#if>
44        <#if mapa['CARGO']?? >
45        , ${mapa['CARGO']}
46        </#if>
47        <#if mapa['SUB_CIRCULO']?? >
48        , ${mapa['SUB_CIRCULO']}
49        </#if>
50        <#if mapa['POSTO']?? >
51        , ${mapa['POSTO']}
52        </#if>
53        <#if mapa['ARMA']?? >
54        , ${mapa['ARMA']}
55        </#if>
56      ) from $fracao.cargos <#if !mapa['HABILITACAO']?? && !mapa['OBSERVACAO']?? >;</#
            ↪ if>
57    <#if mapa['HABILITACAO']?? >
58      and
59      HabilitacaoVO(codigoStr ${mapa['HABILITACAO']}) from $cargo.habilitacoes <#if !
            ↪ mapa['OBSERVACAO']?? >;</#if>
60    </#if>
61    <#if mapa['OBSERVACAO']?? >
62      and
63      ObservacaoVO(codigo ${mapa['OBSERVACAO']}) from $cargo.observacoes;
64    </#if>
65
66      collectList( $cargo )
67    )
68  then
69        for(int i=0; i < $cargosFiltrados.size(); i++){
70          CargoVO c = (CargoVO) $cargosFiltrados.get(i);
71          helper.adicionarMateriais(${chamador.id?string.computer}L, c.getIdFracaoQc()
                ↪ , c.getPerfilCgoNumero());
72        }
```

```
73 end
```

Código 3.3: *Definição do template usado para criar DRL*

Geração do QDM

Outro passo do processo de geração de QDM definido acima está relacionado à execução das regras DRL, cuja criação foi detalhada na Subseção 3.2, pelo *rule engine* (ver 2.3) Drools. O trecho de código Java responsável pela execução das regras pelo Drools está listado no Código 3.4. Neste código é possível observar a criação de um *file system* que abrigará o arquivo de regras, representando os chamadores, passado como parâmetro. As regras são então compiladas e é realizada uma verificação de erros. Caso as regras tenham sido compiladas com sucesso, uma sessão do Drools é criada, a classe auxiliar recebida como parâmetro é definida como uma variável global, o QC para o qual o QDM está sendo gerado é inserido como fato na *working memory* e as regras são executadas.

```
1  private void executarRegras(String regras, QuadroDotacaoMaterialHelper helper) {
2          logger.info("Executando regras ....");
3      KieServices kieServices = KieServices.Factory.get();
4
5      KieFileSystem kfs = kieServices.newKieFileSystem();
6      kfs.write("src/main/resources/regras/qdm/arquivoRegras.drl", kieServices.
           ↪ getResources().newReaderResource(new StringReader(regras)));
7
8      KieBuilder kieBuilder = kieServices.newKieBuilder(kfs).buildAll();
9      Results results = kieBuilder.getResults();
10     if (results.hasMessages(Message.Level.ERROR)) {
11         List<Message> messages = results.getMessages(Message.Level.ERROR);
12         for (Message m : messages) {
13             logger.error(String.format("[%s] - %s[%s,%s]: %s", m.getLevel(), m.getPath
                   ↪ (), m.getLine(), m.getColumn(), m.getText()));
14         }
15         throw new IllegalStateException("Erros de compilacao foram encontrados.");
16     }
17
18     KieContainer kieContainer = kieServices.newKieContainer(kieServices.getRepository
           ↪ ().getDefaultReleaseId(), getClass().getClassLoader());
19
20     KieSession kSession = kieContainer.newKieSession();
21     kSession.setGlobal("helper", helper);
```

```
22      kSession.insert(helper.getQc());
23
24      logger.debug("Disparando regras ...");
25      int regrasDisparadas = kSession.fireAllRules();
26      logger.debug("regrasDisparadas=" + regrasDisparadas);
27
28      kSession.dispose();
29  }
```

Código 3.4: *Trecho de código responsável pela execução das regras no Drools*

Para cada regra disparada é chamado um método da classe auxiliar, passada como parâmetro, para prover os MEMs definidos no chamador para os cargos selecionados nesta regra. Esta classe possui estruturas de dados específicas para agilizar acessos a chamadores, MEMs e cargos, além de estruturas e métodos que facilitam o preenchimento do QDM sendo criado. Algumas das estruturas definidas podem ser usadas também para verificar a consistência dos chamadores, ou seja, se nenhum cargo recebeu o mesmo MEM por dois ou mais chamadores diferentes. Após a finalização da execução do Drools o QDM gerado pode ser recuperado a partir dessa classe auxiliar para ser persistido.

Documento do QDM

Após a geração do QDM, o usuário tem acesso a algumas opções na tela do sistema, como: editar o QDM, enviar o QDM para homologação, validar o QDM, excluir e exportar os dados do QDM. A exportação dos dados pode ser feita nos formatos: PDF, planilha ou documento. Na Figura 3.9 é apresentado um trecho do QDM gerado para um Batalhão de Infantaria (BI). Para a geração deste QDM foi criado apenas um chamador simples que determina a distribuição dos MEMs relacionados para OMs de natureza Infantaria, servindo apenas como um exemplo da aparência de um QDM exportado no formato PDF.

O QDM é organizado em três seções distintas. A primeira seção mostra a dotação por fração (Figura 3.9), ou seja, apresenta quais MEMs devem ser distribuídos para cada fração. A segunda seção contém a dotação por OM, apresentando a totalização dos MEMs que devem ser providos para a OM relacionada ao QDM gerado. A Figura 3.10 apresenta um trecho da seção de dotação por OM do QDM gerado. Nas duas primeiras seções os MEMs são agrupados por classe e família de material (ver 2.1), e ordenados pelo código de dotação (CODOT). Como o SISDOT é gerido pela 4ª Subchefia do Estado-Maior do Exército Brasileiro e esta trabalha apenas com MEMs do tipo 1 (Material Operacional Relacionado em QDM), não foi apresentada no QDM informação relacionada ao tipo de

MATERIAL DE ACESSO RESTRITO

Art. 44 e 45 do Decreto 7.845/2012 de 14 de Novembro de 2012

MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
ESTADO-MAIOR DO EXÉRCITO

QUADRO DE DOTAÇÃO DE MATERIAL

Batalhão de Infantaria	Nr Ctl	Aprovado
QO: 0702.31.1	1	4º SCh EME
Aprovação: Port [pendente] / Publicação: BARE [pendente]		(SEPARATA)

DOTAÇÃO POR FRAÇÃO

CODOT	Descrição	Qnt	Obs	N Dot
1 Comando e Estado-Maior				
1.1 Comando				
Classe 2	Intendência			
Família 1	Equipamentos			
1020100013	Coldre Ambidestro	1		4
1020100017	Estojo para Carregador de Pistola	2		2
Classe 5	Armamento e munição			
Família 10	Armamento			
1051000006	Faca de Combate	1		3
1051000025	Pistola Semiautomática	1		1
1.2 Estado-Maior				
Classe 2	Intendência			
Família 1	Equipamentos			
1020100013	Coldre Ambidestro	5		4
1020100017	Estojo para Carregador de Pistola	10		2
Classe 5	Armamento e munição			
Família 10	Armamento			
1051000006	Faca de Combate	5		3
1051000025	Pistola Semiautomática	5		1

Figura 3.9: *QDM gerado: Dotação por Fração*

material, já que apenas um tipo é utilizado. Na terceira seção são apresentadas as normas de dotação utilizadas por este QDM.

3.4 Quadro de Dotação de Material Previsto

Outro importante documento gerado pelo SISDOT é o Quadro de Dotação de Material Previsto (QDMP). O QDMP é o documento, baseado no Quadro de Cargos Previstos (QCP) e no QDM de cada OM, que estabelece a quantidade de Material de Emprego Militar (MEM) considerada necessária ao adestramento da OM e ao cumprimento de suas missões em tempo de paz [184].

O QCP é o documento específico para cada OM, operativa ou não operativa, que prevê os cargos necessários para seu funcionamento de acordo com suas necessidades. Em uma OM operativa o QCP será baseado no QC [184]. Fazendo um paralelo com a orientação a objetos, o QC é uma classe e o QCP uma instância desta classe. Por exemplo: existe

MATERIAL DE ACESSO RESTRITO
Art. 44 e 45 do Decreto 7.845/2012 de 14 de Novembro de 2012

MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
ESTADO-MAIOR DO EXÉRCITO

QUADRO DE DOTAÇÃO DE MATERIAL

Batalhão de Infantaria	Nr Ctl	Aprovado
QO: 0702.31.1	1	4º SCh EME
Aprovação: Port [pendente] / Publicação: BARE [pendente]		(SEPARATA)

DOTAÇÃO POR OM

CODOT	Descrição	Qnt	Obs	N Dot
Classe 2	Intendência			
Família 1	Equipamentos			
1020100013	Coldre Ambidestro	599		4
1020100017	Estojo para Carregador de Pistola	1198		2
Classe 5	Armamento e munição			
Família 10	Armamento			
1051000006	Faca de Combate	599		3
1051000025	Pistola Semiautomática	599		1

Figura 3.10: *QDM gerado: Dotação por OM*

o QC para Batalhão de Infantaria (BI), que prevê a quantidade de MEM necessária ao cumprimento das atividades estabelecidas na base doutrinária deste tipo de OM. Todas as OM (físicas) que são BI possuem a estrutura organizacional determinada pelo QC. Porém, um QCP pode incluir ou suprimir alguns cargos para refletir sua realidade. Existem vários BI, cada um com suas necessidades específicas, uns com mais militares em alguns dos cargos definidos no QC e outros com menos militares.

No QCP não há a possibilidade de incluir ou excluir cargos ou frações, e sim a possibilidade de alterar a quantidade de militares que venham a ocupar esses cargos. Por exemplo: caso não exista fisicamente uma determinada fração na OM, mas que foi prevista no QC, os cargos desta fração deverão ter quantidade igual a zero no QCP, suprimindo todos os cargos desta fração existentes no QC.

O QCP deve estar aderente à estrutura organizacional imposta no QC, sem a possibilidade de incluir ou excluir cargos ou frações. Entretanto, o QCP pode possuir módulos, vistos como frações extras. Um módulo é uma estrutura organizada e equipada para atender às necessidades específicas de uma OM, constando apenas no seu QCP. Cada módulo do QCP segue a estrutura imposta pelo seu próprio QC, mas as quantidades podem ser alteradas neste QCP. Um exemplo hipotético: existe um QC de uma banda de música com três trompetistas. Um QCP que possui um módulo aderente a este QC pode alterar a quantidade de trompetistas para dois.

A geração de um QDMP leva em consideração QCP e os QDMs da OM e suas frações extras, quando existirem. Para que aconteça a geração do QDMP a presença do QDM da OM é obrigatória, enquanto as presenças dos QDMs das frações extras é apenas desejável,

mas necessária para geração de um QDMP completo.

O processo de geração de um QDMP dá-se da seguinte maneira: o usuário, autenticado e na tela de geração de QDMP, seleciona a OM para a qual deseja-se gerar o QDMP. Para que o usuário possa selecionar a OM, este deve informar algum dado para que seja possível recuperar as informações da OM, como: o código da OM (Codom), ou o código do QC ou um trecho do nome da OM. Após preencher o dado para pesquisa o sistema lista na tela os dados da(s) OM(s) encontrada(s), possibilitando a seleção de uma. Ao selecionar a OM o sistema recupera na base de dados corporativa do EB os dados do QCP desta OM. Recupera também, na base de dados do SISDOT, os QDMs da OM e de suas frações extras. O QDMP é gerado ao cruzar as informações contidas nos QDMs com as informações do QCP, levando em consideração as supressões declaradas no QCP. Após a geração do QDMP o usuário tem a possibilidade de algumas ações, como: edição e exportação dos dados. A Figura 3.11 apresenta um trecho de um QDMP gerado e exportado no formato pdf.

MATERIAL DE ACESSO RESTRITO
Art. 44 e 45 do Decreto 7.845/2012 de 14 de Novembro de 2012

MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
ESTADO-MAIOR DO EXÉRCITO

QUADRO DE DOTAÇÃO DE MATERIAL PREVISTO

55º Batalhão de Infantaria	Nr Ctl	Aprovado
CODOM: 0006.21.3	1	4º SCh EME
Aprovação: Port [pendente] / Publicação: BARE [pendente]		(SEPARATA)

CODOT	Descrição do Material	QDM Completo	QDM Reduzido	Diversos	Total
Classe 2	Intendência				
Família 1	Equipamentos				
1020100013	Coldre Ambidestro	599	364		364
1020100017	Estojo para Carregador de Pistola	1198	728		728
Classe 5	Armamento e munição				
Família 10	Armamento				
1051000006	Faca de Combate	599	364		364
1051000025	Pistola Semiautomática	599	364		364

Figura 3.11: *QDMP gerado*

Existe a possibilidade do QCP possuir uma fração chamada 'Diversos'. Embora seja vista como uma fração extra, não possui um QC e, consequentemente, também não possui um QDM. Quando existir essa fração, com cargos preenchidos, o QDMP gerado não é completo e exige uma edição manual por parte do usuário para completá-lo.

3.5 Teste de Software

Conforme apresentado na Seção 2.5, o teste de *software* consiste na verificação dinâmica de que um programa fornece comportamentos esperados em um conjunto finito de casos de teste, selecionados adequadamente a partir do domínio de execução geralmente infinito [77]. Então, o teste de *software* destina-se a mostrar que um programa faz o que se pretende fazer e descobrir defeitos do programa antes de ser colocado em uso. O teste não é mais visto como uma atividade que começa somente após a conclusão da fase de codificação, com o objetivo limitado de detectar falhas. O teste de *software* deve abranger todo o ciclo de vida de desenvolvimento e manutenção, com seu planejamento iniciando nos estágios iniciais do processo de requisitos de *software*.

Durante o desenvolvimento do Sistema de Dotação de Material (SISDOT) foram criados casos de teste em diferentes níveis de granularidade, incluindo testes unitários e de integração. Porém, o foco desta Seção apresentar a utilização da DSL (Seção 3.2) criada para definição de chamadores junto com a especificação de testes de aceitação, com a utilização da ferramenta Cucumber. Esses testes servem como uma forma de verificar a corretude da geração de QDMs (Seção 3.3), podendo ser definidos de forma manual ou gerados automaticamente.

Foi criado um projeto separado do SISDOT para a implementação desses testes. O desenvolvimento da DSL também foi realizado em um projeto à parte. Ambos projetos possuem repositórios git próprios no bitbucket, assim como o SISDOT, todos eles fazendo parte do mesmo projeto, o Projeto de Pesquisa para Validação de Práticas e Métodos de Desenvolvimento de Software para o Exército Brasileiro (PROMISE-EB).

O projeto para implementação dos testes foi dividido em diversos módulos:

- Cache: possui uma implementação simples de um *cache* em memória utilizado para armazenar dados de QCs buscados no banco de dados de forma a agilizar sua recuperação posterior, já que os QCs são necessários para a geração de QDMs;
- Comum: define classes que auxiliam o acesso ao banco de dados, a execução de regras pelo Drools e classes POJO que representam os resultados esperados após a execução dos testes. Define também, em um JAR específico para testes, uma infraestrutura para execução dos testes, utilizada por outros módulos;
- Maven-plugin: define um *plugin* maven responsável pela geração automática dos testes;
- Automático: configuração e utilização do *plugin* maven desenvolvido, assim como a geração e execução dos testes;
- Manual: definição dos casos de teste de forma manual.

Geração automática de testes

Para a geração automática de testes foi implementado um *plugin* maven. Foi feita a opção pela criação deste plugin para que todos os artefatos gerados fossem criados na fase generate-sources do ciclo de vida de *build* do projeto maven. Com isso, outros projetos ou módulos, neste caso o módulo automático, podem utilizar o *plugin* para gerar os artefatos de teste e utilizá-los nas fases seguintes do ciclo de vida de *build*, como compile, test e verify, juntamente com os códigos já existentes.

Para utilizar o *plugin* basta defini-lo no arquivo POM onde será usado, conforme apresentado no Código 3.5. Apenas a propriedade "listaQcsString"é de preenchimento obrigatório, onde são definidos os códigos dos QCs que devem ser testados. É possível configurar outras propriedades neste *plugin*, como: a "seed"usada para geração de números randomicamente, de forma que os testes possam ser repetidos, ou seja, que gerem sempre os mesmos resultados de maneira que possam ser conferidos e repetidos. Outra propriedade configurável é o caminho do diretório onde os artefatos serão gerados.

```
<plugin>
  <groupId>br.mil.eb.cds</groupId>
  <artifactId>sisdot-teste-qdm-maven-plugin</artifactId>
  <version>${project.version}</version>
  <executions>
    <execution>
      <id>executar</id>
      <configuration>
        <seed>10</seed>
        <listaQcsString>702311,2304310,762314,300401,757311,2203310</listaQcsString>
        <baseDir>${caminho.codigo.gerado.teste.resources}</baseDir>
      </configuration>
      <goals>
        <goal>gerar</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Código 3.5: *Declaração do plugin responsável pela geração de testes*

Para executar este *plugin* é necessário declarar as dependências com o projeto que define a DSL na qual os chamadores gerados são definidos, além do plugin do xtext, responsável pela conversão da DSL em uma representação java dos chamadores, para que possam ser usados na execução dos testes. É necessário também declarar as dependên-

cias com o junit e com o cucumber, este último responsável pela execução dos cenários definidos, utilizando a infraestrutura de testes provida pelo junit.

O *plugin* define diversos cenários que serão gerados para cada QC declarado. Cada cenário é responsável pela geração de um chamador no formato DSL e um arquivo de definição do cenário no formato do cucumber. Para cada cenário é gerado um QDM para o QC, usando o chamador gerado e o resultado deste QDM é conferido de acordo com os dados previstos na definição do cenário do cucumber. Foram criadas diversas combinações de cenários e para cada uma delas alguns dos valores das propriedades são gerados randomicamente, seguindo a técnica Property-based Testing (PBT).

Para um entendimento dessas combinações que formam os cenários é necessário entender melhor a regra Drools Rule Language (DRL). Todos os chamadores são convertidos para o formato DRL para que o Drools possa executar as regras. Uma regra DRL, no âmbito deste projeto, pode ser dividida em três regiões lógicas. Para melhor ilustrar essa divisão, é apresentado um exemplo hipotético de uma DRL no Código 3.6. As três regiões são: QC, nas linhas 3 e 4; fração, na linha 6; e cargo, nas linha de 8 a 17. A parte condicional (LHS) desta regra é avalida de forma sequencial e pode ser entendida, de forma resumida, da seguinte maneira:

1. primeiro é testada a região de QC: quando existir um QC, na *working memory*, que seja operacional ou não-operacional e cujo código do Valor deste QC esteja entre os códigos da lista apresentada, guarde uma referência para este QC na variável "$qc";

2. caso a primeira condição tenha sido satisfeita, é testada a região relativa a frações. É realizada uma iteração sobre todas as frações do QC. Para cada fração que satisfaça a condição apresentada é executada a terceira região, relativa aos cargos dessa fração selecionada;

3. quando houver uma fração selecionada, as condições referentes a cada cargo desta fração são avaliadas. Caso o cargo sendo avaliado satisfaça as condições, este é coletado (linha 19) e adicionado a uma lista previamente definida (linha 5) que contém todos os cargos que estão aderentes a este chamador (regra DRL). Após a iteração sobre todos os cargos selecionados (a partir das frações selecionadas), é feita uma verificação no tamanho da lista de cargos filtrados (linha 5). Caso a lista não esteja vazia, este chamador é válido para os cargos filtrados;

4. por fim, já na parte RHS da DRL, para cada cargo filtrado são disponibilizados os MEMs definidos no chamador. A atribuição dos MEMs para cada cargo é feita por um objeto auxiliar definido como uma variável global na DRL. Esse objeto ajuda a popular o QDM a medida que as regras são disparadas e as condições (LHS) satisfeitas.

```
rule "Chamador 1"
  when
    $qc: QuadroDeCargosVO( (operacional == TipoOrganizacaoMilitar.OPERACIONAL ||
        ↪ operacional == TipoOrganizacaoMilitar.NAO_OPERACIONAL)
                , idValor in (23, 41, 90, 50, 42, 31, 21, 32, 20, 30, 40) )
    $cargosFiltrados: List( size > 0 ) from accumulate (
      $fracao: FracaoQcVO( idFracao not in (14086, 1130, 8383, 9609, 2740, 10947,
          ↪ 9149, 14131, 2265, 2513, 15219) ) from $qc.fracoes
      and
      $cargo: CargoVO(
        cargoFuncao == "CARGO"
        , idCargo not in (364, 539, 121, 2205, 528, 2204, 539)
        , idPosto in (24, 23, 44, 22, 42)
        , idArma not in (1055, 10, 833, 5308, 5393, 729, 5110, 15, 5112, 5390, 12)
      ) from $fracao.cargos
      and
      HabilitacaoVO(codigoStr not in ("782", "756", "793", "768", "765", "751", "921",
          ↪  "749", "767", "748")) from $cargo.habilitacoes
      and
      ObservacaoVO(codigoStr in ("40D","40K")) from $cargo.observacoes;

      collectList( $cargo )
    )
  then
        for(int i=0; i < $cargosFiltrados.size(); i++){
          CargoVO c = (CargoVO) $cargosFiltrados.get(i);
          helper.adicionarMateriais(1L, c.getIdFracaoQc(), c.getPerfilCgoNumero());
        }
end
```

Código 3.6: *Divisão lógica de uma DRL no âmbito do SISDOT*

Os cenários de teste foram também divididos em três grupos, de acordo com as seções da DRL. Isso foi feito pois cada região é testada de forma quase que independente, possuindo dependência apenas da validade da região anterior. Por exemplo: a região de fração só é avaliada se a região de QC tiver sido avaliada como verdadeira, então essas regiões podem ser testadas de forma independente ao assumir que a região anterior é verdadeira e que foram gerados casos de testes que ajudaram a validar o funcionamento das condições da região anterior. Ao assumir a independência dos grupos, para fins de teste,

é possível focar nas combinações das propriedades de cada grupo, facilitando a geração de cenários de teste.

Cada grupo de cenários é responsável por testar várias das combinações das propriedades da entidade relativa ao grupo (QC, fração ou cargo). Em cada cenário é definido um chamador e uma lista com os resultados esperados, que são gerados de acordo com a combinação específica de propriedades deste cenário. Como o MEM definido no chamador não interfere na execução das regras, foi definido um MEM padrão para todos os chamadores gerados, assim como a quantidade igual a um (unitário). Isso foi feito para diminuir a complexidade do algoritmo de geração ao não precisar acessar o banco de dados várias vezes para recuperar MEMs de forma randômica, tendo em vista que não interferem na avaliação das regras envolvidas na geração do QDM.

No grupo de QC existem as seguintes propriedades: operacionalidade, natureza, subnatureza e valor. Foram criados cenários específicos para cada propriedade e para combinações entre elas. Para a propriedade natureza do QC, por exemplo, foram definidos os seguintes cenários:

- natureza válida e tipoOM válido: este cenário gera um QDM válido, com resultados válidos, pois neste cenário é criado um chamador com os mesmos valores de natureza e operacionalidade (tipoOM) do QC para o qual será gerado o QDM, ou seja, natureza e tipoOM válidos;
- natureza válida e tipoOM válido (AMBOS): este cenário funciona de forma parecida com o anterior, mas o tipoOM é definido como "AMBOS"no chamador. Este tipoOM indica que o QC pode ser operacional ou não-operacional;
- natureza válida e tipoOM inválido: o valor da natureza é o mesmo do QC, pois é válido, e o valor do tipoOM é alterado. Por exemplo: se o QC for operacional o valor é alterado para não-operacional e vice-versa;
- natureza inválida e tipoOM válido: o valor da natureza deve ser inválido no chamador, ou seja, um valor diferente da natureza do QC. A geração deste valor inválido de natureza é realizado de forma randômica, usando a *seed* configurada no *plugin* maven, em uma lista de códigos de naturezas reais previamente definida;
- natureza inválida e tipoOM inválido: os valores de natureza e tipoOM são inválidos, e são gerados conforme citado anteriormente: invertendo o valor de operacionalidade e recuperando randomicamente um código de natureza da lista declarada previamente;
- negar natureza, inválido: as combinações citadas anteriormente nesta lista possuem chamadores com regras do tipo "AFIRMACAO", o tipo padrão de uma regra. Mas existe o tipo "NEGACAO", onde a regra é negada, ou seja, a condição (na DRL)

Tabela 3.1: Cenários definidos para Valor de QC.

Cenário	Valor	Subnatureza	Natureza	TipoOM
1	0			0
2	0			1
3	1			0
4	1			1
5	0		0	0
6	0		0	1
7	0		1	0
8	0		1	1
9	1		0	0
10	1		0	1
11	1		1	0
12	1		1	1
13	0	0	0	0
14	0	0	0	1
15	0	0	1	0
16	0	0	1	1
17	0	1	0	0
18	0	1	0	1
19	0	1	1	0
20	0	1	1	1
21	1	0	0	0
22	1	0	0	1
23	1	0	1	0
24	1	0	1	1
25	1	1	0	0
26	1	1	0	1
27	1	1	1	0
28	1	1	1	0

é válida quando o valor testado não pertence à lista de valores especificados na regra. Neste cenário, por exemplo, há uma regra de negação da natureza de forma que o QDM seja inválido. Então foi criada uma regra para "NATUREZA"do tipo "NEGACAO"definindo como valor o código da natureza do QC. Com isso, esta regra não é válida para este QC, sendo válida para todas as outras naturezas exceto a deste QC;

- negar natureza, válido: para este cenário é criado um chamador com uma regra para "NATUREZA"do tipo "NEGACAO", mas gerando um código de natureza diferente da natureza do QC. Com isso, esta regra é válida para este QC.

Seguindo a mesma linha de combinações utilizadas paras cenários envolvendo natureza

do QC, foram criados cenários para subnatureza e valor. Não foram criados cenários para tipoOM pois esta é uma propriedade de preenchimento obrigatório no chamador e a checagem de sua validade já é exercitada em todos os cenários do grupo. Na Tabela 3.1 são apresentados os cenários definidos para o Valor do QC e suas combinações. Na primeira coluna é apresentado o número do cenário, criado apenas para facilitar uma possível referência a uma linha específica da tabela. As outras colunas contém os valores das propriedades do QC sendo combinadas. A células com valor 0 indicam que o valor desta propriedade, indicada pela coluna, é inválido, ou seja, deverá ser selecionado um valor randomicamente de uma lista previamente definida e este valor deve ser diferente do valor que essa propriedade apresenta no QC. As células com o valor 1 indicam que o valor desta propriedade é válido, ou seja, é o mesmo valor definido no QC para esta propriedade. As células sem valor indicam que essa propriedade não faz parte da combinação.

Para a propriedade Valor do QC também foram criados dois cenários para testar as regras "NEGACAO". Em um cenário é criada uma regra de negação para o Valor do QC, ou seja, todos os QCs que não possuam o Valor indicado na regra seriam avaliados como verdadeiros. No outro cenário é criada uma regra para o Valor do QC, onde todos os QCs que possuam o Valor definido seriam avaliados como verdadeiros.

No grupo Cargo existem as seguintes propriedades: código do cargo, subcírculo ao qual o cargo pertence, o posto (patente) necessário para ocupar o cargo, e a arma (Arma, Quadro ou Serviço) necessária para ocupar o cargo. O subcírculo está relacionado às graduações (posto/patente) existentes, servindo como uma forma de agrupamento. Por exemplo: o subcírculo superior agrupa as patentes de coronel, tenente-coronel e major; o subcírculo subalterno agrupa os tenentes.

Existe uma ampla gama de especializações desempenhadas por cada integrante da Força Terrestre, abrangendo os mais diversos campos de atividades, e que, na maioria dos casos, define toda a carreira militar desses indivíduos. A grande divisão dessas especializações é definida pela Arma, Quadro ou Serviço a que pertence um militar do Exército. As Armas englobam o militar combatente por excelência, tradicionalmente a atividade-fim da profissão. As Armas dividem-se em dois grupos: as Armas-Base (Infantaria e Cavalaria) e as Armas de Apoio ao Combate (Artilharia, Engenharia e Comunicações). Os Quadros reúnem os militares que, de origem diversa, aglutinam-se dentro desses quadros com uma finalidade geral própria, por exemplo: Quadro de Engenheiros Militares (QEM), Quadro de Material Bélico (QMB) e Quadro Complementar de Oficiais (QCO). Por fim, há os Serviços que, como o termo indica, têm uma atividade de apoio bem definida, normalmente de cunho logístico. Os Serviços de Intendência e de Saúde (médicos, dentistas e farmacêuticos) trabalham na paz e na guerra para a manutenção do homem, pelo atendimento às suas necessidades de sustento e sanitárias. Os oficiais de Intendência são mestres

no suprimento e nas finanças. O Serviço de Intendência é a parte da logística voltada para as atividades de suprimento. Ele distribui o material de intendência (uniformes, equipamentos individuais, etc) e os diversos tipos de munição e de gêneros alimentícios. Proporciona também, em operações, outros serviços como lavanderia e banho. Nas organizações militares os intendentes assessoram os comandantes na administração financeira e na contabilidade. Cabe ao Serviço de Intendência atender aos objetivos do Exército Brasileiro no que se refere a atividades logísticas que convergem para o planejamento correto e o provimento oportuno, nos locais determinados e nas quantidades e especificações exigidas [207].

Para cada cenário do grupo de cargos foi selecionado um cargo aleatoriamente da lista de cargos do QC, utilizando a *seed* configurada no *plugin* maven. De acordo com o cargo selecionado são geradas as regras conforme a combinação de propriedades especificada para o cenário. Para cada cenário, além das regras, é gerada também uma lista contendo os resultados esperados, ou seja, os dados que se espera encontrar no QDM gerado para o QC.

A Tabela 3.2 apresenta as combinações de propriedades de cargos, onde cada linha representa a combinação usada na definição do cenário. Na primeira coluna é apresentado o número do cenário, criado apenas para facilitar uma possível referência a uma linha específica da Tabela. As outras colunas apresentam as propriedades de cargo sendo combinadas. As células com valor 1 indicam que a regra será criada com o tipo da coluna e com o valor correspondente no cargo. Enquanto que para as células com valor 0 são criadas regras de negação com o tipo da coluna e com o valor correspondente no cargo. Por exemplo: na linha 10 são criadas duas regras no chamador gerado: uma de negação para o código do cargo selecionado e outra especificando o subcírculo do cargo selecionado. Com isso, ao executar esse chamador, todos os cargos da lista de cargos do QC que não possuam o código declarado mas que possuam o mesmo subcírculo declarado estarão aptos a receber o MEM definido no chamador.

Para a propriedade código do cargo também foram criados dois cenários para testar as regras do tipo "NEGACAO". Para cada um dos cenários, assim como para os outros cenários deste grupo, é selecionado um cargo aleatoriamente do QC. Em um cenário é criada uma regra de negação para o código do cargo selecionado, ou seja, todos os cargos do QC são providos com o MEM definido no chamador, exceto os cargos que possuam o mesmo código do cargo selecionado. No outro cenário é criada uma regra para o código do cargo selecionado. Sendo assim, apenas os cargos que possuam o mesmo código do cargo selecionado estão apto a receber o MEM definido no chamador, enquanto os outros cargos não.

Tabela 3.2: Cenários definidos para Cargos.

Cenário	**Código**	**Subcírculo**	**Posto**	**Arma**
1	1			
2	0			
3		1		
4		0		
5			1	
6			0	
7				1
8				0
9	1	1		
10	0	1		
11	1	0		
12	0	0		
13	1		1	
14	0		1	
15	1		0	
16	0		0	
17	1			1
18	0			1
19	1			0
20	0			0
21	0	0	0	
22	0	0	1	
23	0	1	0	
24	0	1	1	
25	1	0	0	
26	1	0	1	
27	1	1	0	
28	1	1	1	
29	0	0	0	0
30	0	0	0	1
31	0	0	1	0
32	0	0	1	1
33	0	1	0	0
34	0	1	0	1
35	0	1	1	0
36	0	1	1	1
37	1	0	0	0
38	1	0	0	1
39	1	0	1	0
40	1	0	1	1
41	1	1	0	0
42	1	1	0	1
43	1	1	1	0
44	1	1	1	1

Para este trabalho foram especificados 100 cenários. Todos os cenários gerados, independente do grupo, contém um chamador, com as regras referentes à combinação de propriedades definida para o cenário, e uma lista de resultados esperados, que serão comparados com os dados do QDM gerado. Para cada QC declarado no *plugin* é gerada uma lista de cenários. Para cada QC é gerado um arquivo cucumber contendo os cenários definido para este QC. A geração do arquivo é realizada com o uso do Freemarker (Seção 2.4.1), usando o *template* definido no Código 3.7.

```
# language: pt

Funcionalidade: QDM ${idQc?string.computer}
  Como usuário quero gerar um QDM a partir de chamadores
  Para que eu não tenha que gerar um QDM manualmente

<#list cenarios as cenario>
  Cenario: ${cenario.nome}
    Dado Quadro de Cargos com código ${idQc?string.computer}
    E Chamadores: ${cenario.chamador.codigo?string.computer}
    Quando Gero QDM
    E os resultados são consistentes
    <#assign executar=cenario.deveExecutar() />
    <#if executar>
    Entao os cargos devem estar populados:
    | idFracao | idCargo | codot      | quantidade |
  <#list cenario.resultados as resultado>
    | ${resultado.idFracao?string.computer?right_pad(8)} | ${resultado.idCargo?string.
        ↪ computer?right_pad(7)} | ${resultado.codot} | ${resultado.quantidade} |
  </#list>
  E o resultado final deve ser:
    | codot      | quantidade |
    <#list cenario.resultadosTotal as resultado>
    | ${resultado.codot} | ${resultado.quantidade} |
  </#list>
    <#else>
    Entao o QDM deve estar vazio
    </#if>

</#list>
```

Código 3.7: *Definição do template usado para criar o arquivo Cucumber*

Enquanto é gerado um arquivo do cucumber para cada QC, apenas um arquivo contendo todos os chamadores é gerado. A separação de cenários do mesmo QC em um arquivo do cucumber está relacionada à declaração da funcionalidade sendo testada, no caso a geração de QDMs para o QC selecionado nos mais variados cenários, descritos acima. Para a geração do arquivo de chamadores no formato de DSL é criada uma lista contendo todos os chamadores gerados em cada cenário, de todos os QCs declarados no *plugin*. O *template* usado para criar este arquivo contendo os chamadores é apresentado no Código 3.8.

```
1  <#list chamadores as chamador>
2  chamador ${chamador.codigo} {
3    tipoOM = ${chamador.tipo},
4    materiais=[
5    <#list chamador.itens as item>
6      ('${item.codot}'<#if item.nome?has_content >:"${item.nome}"</#if>:${item.
         ↪ quantidade})<#sep>,
7    </#list>
8    ],
9    regras=[
10   <#list chamador.regras as regra>
11     (tipoChamador=${regra.tipoChamador}, tipo=${regra.tipo}, valores=[<#list regra.
         ↪ valores as valor>"${valor}"<#sep>,</#list>])<#sep>,
12   </#list>
13   ]
14 }
15 </#list>
```

Código 3.8: *Definição do template usado para criar o arquivo de Chamadores*

O processo de geração automática de testes pode ser resumido da seguinte maneira: o *plugin* maven deve ser declarado e configurado, indicando a lista com os códigos dos QCs para os quais devem ser gerados testes. Então, para cada QC da lista são gerados cenários, com variações em suas propriedades de acordo com o QC sendo testado. Os cenários gerados são transformados em arquivos do cucumber e de DSL, que representam os cenários gerados. Houve então uma conversão dos cenários gerados com base na técnica Property-based Testing (PBT) em casos de teste por exemplo, que serão executados automaticamente durante a fase adequada do ciclo de vida de *build* do maven. É gerado um relatório no formato HTML com o resultado da execução dos testes e um trecho deste relatório é apresentado na Figura 3.12.

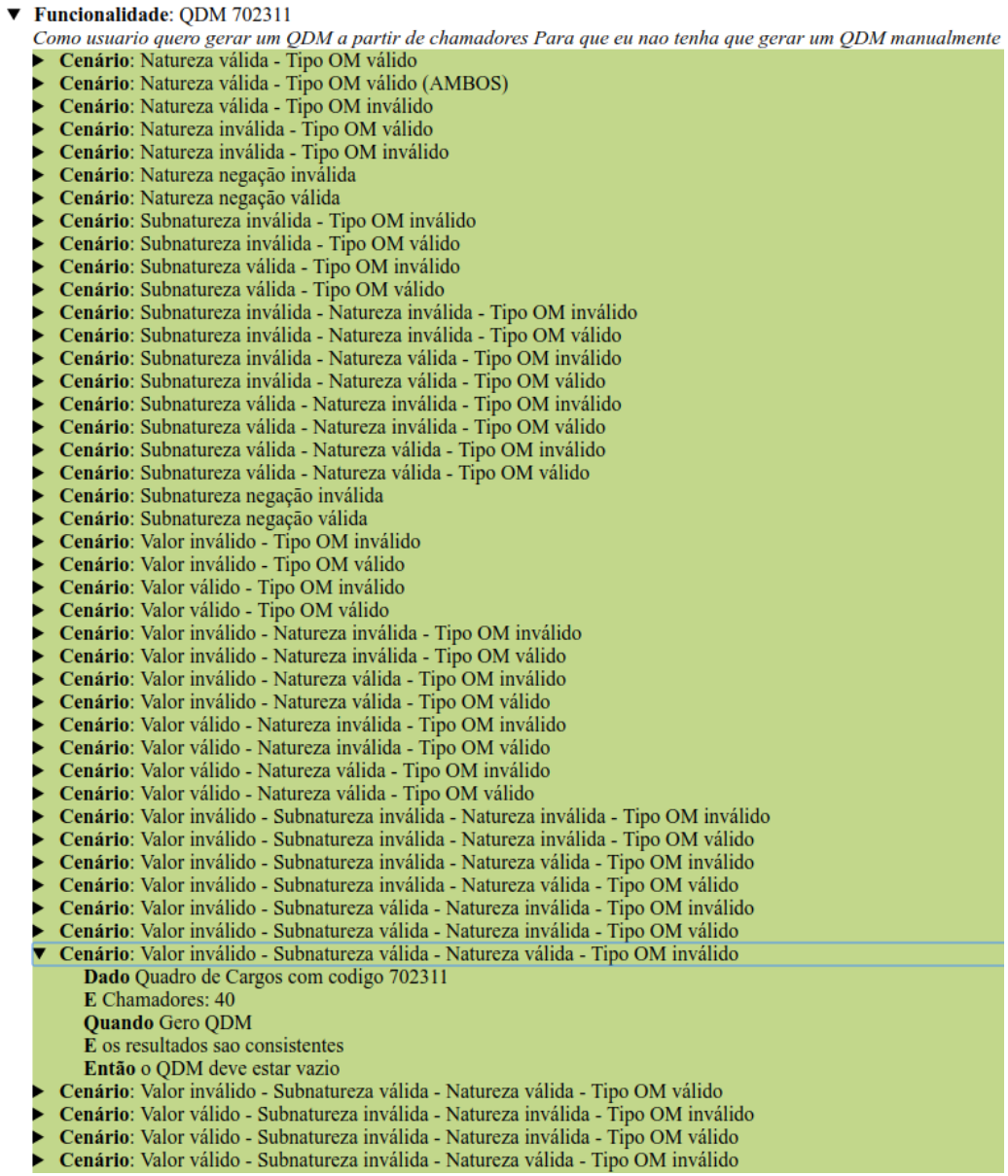


Figura 3.12: *Relatório da execução dos testes gerados automaticamente*

Testes manuais

Além do *plugin* para geração automática de testes, foi desenvolvido, neste projeto, um módulo com o intuito de facilitar a definição manual de testes. Neste módulo o testador é capaz de especificar chamadores usando a DSL criada e utilizar um ou mais chamadores na definição de cenários do Cucumber. A definição dos testes é manual mas são executados automaticamente com o uso de *plugins* específicos do maven.

A definição manual de testes serve para verificar se um conjunto de chamadores declarados resulta nos dados esperados de um QDM gerado para determinado QC. Esse tipo de teste serve também para detecção de inconsistências no conjunto de chamadores de-

clarados, quando um cargo é provido com o mesmo Material de Emprego Militar (MEM) por meio de mais de um chamador.

A especificação de um caso de teste de forma manual exige a definição de um ou mais chamadores declarados com a DSL e a definição de um ou mais cenários em um arquivo do Cucumber, fazendo referência aos chamadores declarados na DSL. Os chamadores declarados são, geralmente, uma combinação dos casos de teste gerados automaticamente, seja em chamadores separados ou juntando as regras em um mesmo chamador. Reforçando assim a importância dos cenários de teste gerados automaticamente.

Para exemplificar a definição manual de testes, foram definidos três chamadores que são exercitados em dois diferentes cenários. Os chamadores declarados são apresentados no Código 3.9. É possível observar que os chamadores 01 e 02 contém regras que estão contidas em cenários de teste gerados automaticamente. Porém, na geração de testes automática esses chamadores são executados de forma separada, enquanto no teste manual é possível testar o comportamento desses chamadores em conjunto.

```
1  chamador 01 {
2    tipoOM = AMBOS,
3    materiais=[
4      ('1051000025':'Pistola Semi-Automática': 1),
5      ('1051000006':1), //Faca de Combate
6      ('1020100017':1), //Estojo para Carregador de Pistola
7      ('1020100013':1)  //Coldre Ambidestro
8    ],
9    regras=[
10     (tipo=POSTO, valores=['2','3','4','11','12','13','15','16','17','21','51','52','53
         ↪ '])
11   ]
12 }
13
14 chamador 02 {
15   tipoOM = AMBOS,
16   materiais=[
17     ('1051000025':'Pistola Semi-Automática': 1),
18     ('1051000006':1), //Faca de Combate
19     ('1020100017':1), //Estojo para Carregador de Pistola
20     ('1020100013':1)  //Coldre Ambidestro
21   ],
22   regras=[
23     (tipo=ARMA, valores=['10','12','729','833','1055','1063','3200','5110','5308','
         ↪ 5390','5391','5392','5393'])
```

```
  ]
}

chamador 03 {
  tipoOM = AMBOS,
  materiais=[
    ('1051000025':'Pistola Semi-Automática': 1),
    ('1051000006':1), //Faca de Combate
    ('1020100017':1), //Estojo para Carregador de Pistola
    ('1020100013':1)  //Coldre Ambidestro
  ],
  regras=[
  (tipo=HABILITACAO, valores=['708','748','749','750','751','756','762','763','765','
      ↪ 767','768','769','782','79A','903','920','921','943','946'])
  ]
}
```

Código 3.9: *Definição de chamadores utilizando a DSL*

Para exercitar os chamadores declarados neste exemplo foram definidos dois cenários de teste, conforme apresentado no Código 3.10. Um dos objetivos da utilização do Cucumber neste projeto é a sua utilização como uma ferramenta de documentação robusta e rica em detalhes que serve de guia para o desenvolvimento do produto. Essa documentação é escrita na forma de funcionalidades e cenários que exercitam essa funcionalidade, escritas em uma linguagem semi-formal (Gherkin), mas de forma concisa e clara com a intenção de facilitar o entendimento por todos os *stakeholders*.

O primeiro cenário tem o objetivo de testar a geração de um QDM para o QC com código 757311, usando os chamadores 1 e 2, definidos no Código 3.9. Ao gerar o QDM os resultados devem ser consistentes e devem conter os valores esperados, listados em forma de tabela. Esses valores esperados devem ser obtidos com o analista de negócio, ou o usuário responsável pela definição de QDM no EB. Apenas esses profissionais sabem exatamente quais os resultados devem ser esperados ao executar um conjunto de chamadores para geração de um QDM para determinado QC. Essa é uma das razões do teste automático gerar cenários mais básicos e com apenas um chamador, pois assim é possível gerar os resultados esperados de forma automática apenas com uma simples filtragem do QC, frações ou cargos. Caso os cenários gerados automaticamente fossem mais complexos haveria a necessidade de desenvolver uma solução paralela para geração de QDM apenas para criar a lista de resultados esperados no cenário. Isto levaria a problemas como: a duplicidade da solução de geração de QDM custaria tempo e dinheiro, e talvez o principal

seja o levantamento de um questionamento: como validar essa outra solução?

```
# language: pt
Funcionalidade: QDM
  Como usuário quero gerar um QDM a partir de chamadores
  Para que eu não tenha que gerar um QDM manualmente

  Cenario: Teste 1
  Dado Quadro de Cargos com código 757311
  E Chamadores: 1, 2
  Quando Gero QDM
  E os resultados são consistentes
  Entao os cargos devem estar populados:
  | idFracao | idCargo | codot      | quantidade |
  | 6765748  | 10403   | 1051000025 | 1          |
  | 6765748  | 10403   | 1020100017 | 1          |
  | 6765748  | 10403   | 1020100013 | 1          |
  | 6765748  | 10403   | 1051000006 | 1          |
  | 6765748  | 21328   | 1051000025 | 1          |
  | 6765750  | 10122   | 1051000025 | 1          |
  | 6765793  | 13276   | 1051000025 | 6          |
  | 6765796  | 24655   | 1051000025 | 1          |
  | 6765796  | 24429   | 1051000025 | 2          |
  | 6765798  | 24429   | 1051000025 | 3          |
  | 6765798  | 11433   | 1051000025 | 9          |
  E o resultado final deve ser:
  | codot      | quantidade |
  | 1020100013 | 74         |
  | 1020100017 | 74         |
  | 1051000006 | 74         |
  | 1051000025 | 74         |

  Cenario: Teste 2
  Dado Quadro de Cargos com codigo 757311
  E Chamadores: 1, 2, 3
  Quando Gero QDM
  E os resultados são inconsistentes
```

Código 3.10: *Definição de cenários utilizando o Cucumber*

O segundo cenário tem o objetivo de demonstrar a inconsistência entre chamadores ao

gerar um QDM para o QC com código 757311, usando os chamadores 1, 2 e 3, definidos no Código 3.9. Ao gerar o QDM, os resultados devem ser inconsistentes pois ao menos um cargo foi provido com os MEMs declarados por mais de uma chamador. No caso em questão, os chamadores 2 e 3 estão em conflito. Exemplos de cargos que receberam os mesmos MEMs por meio dos dois chamadores são listados abaixo:

- O cargo "Operador de Micro"da fração "Seção de Controle de Suprimento", pois possui arma com código 3200 (Qualquer QMG, exceto as QMG 08-Saúde e 00-Singular), presente no chamador 02; e possui habilitações 79A (Operador de Computador) e 903 (Atirador ou Auxiliar de Atirador), ambas presentes no chamador 03. Com isso, os dois chamadores são válidos para esse cargo, gerando uma inconsistência;
- O cargo "Padioleiro"da fração "1ª, 2ª e 3ª Turmas de Evacuação", pois possui arma com código 833 (QMG 08-Saúde/QMP 33-Auxiliar de Saúde), presente no chamador 02; e possui habilitação 920 (Motorista), presente no chamador 03.

Os testes declarados de forma manual são claramente importantes para ajudar na garantia da qualidade da solução. No exemplo apresentado, o primeiro cenário demonstrou que o QDM gerado apresentava os dados esperados. O segundo cenário, talvez mais importante, permitiu verificar a inconsistência entre chamadores na geração de um QDM. Isso demonstra a cautela que os profissionais do EB responsáveis pela gestão do SISDOT devem ter na definição de chamadores, pois inconsistências acarretam na geração de QDM com dados errados.

Capítulo 4

Estudo de Caso Prático

A aceitação de estudos empíricos em engenharia de *software* e suas contribuições para o aumento do conhecimento está crescendo continuamente [15]. Para a validação da arquitetura proposta, foi realizado um estudo de caso prático, que consiste na avaliação dos principais decisões arquitetônicas do SISDOT relacionadas à geração de QDMs, tido como o principal caso de uso do sistema. O estudo de caso tem como objetivo responder às seguintes perguntas de pesquisa:

- **RQ.1**: A abordagem proposta, baseada na meta-programação, gera QDMs corretos?
- **RQ.2**: O uso de uma DSL reduz o esforço necessário para implementar os casos de teste que visam a geração de QDM?
- **RQ.3**: A abordagem proposta, baseada em meta-programação, gera QDMs dentro de um prazo satisfatório?

RQ.1 lida com o conceito mais importante: corretude em relação à geração de QDM. Pretende-se demonstrar que a solução gera automaticamente as regras, eliminando o trabalho que o desenvolvedor teria para implementar a solução em Java. Com poucas linhas de código, relativas à definição do *template* e sua execução, a solução gera um arquivo de regras com um número significante de DRLs, de acordo com a quantidade de chamadores cadastrados. A resposta a esta questão está relacionada aos resultados dos testes manuais e automáticos.

RQ.2 está relacionada com a produtividade do testador e demonstra o quanto de código o desenvolvedor tem que escrever ao usar a DSL proposta, em comparação com o esforço para especificar chamadores usando a linguagem Java em casos de teste unitários.

Embora não exista nenhum requisito funcional formal que especifique o tempo aceitável para gerar um QDM, foi determinado que um tempo superior a 10 segundos, quando se usam 1000 chamadores, seria inaceitável. Portanto, a **RQ.3** serve para garantir que os QDMs sejam gerados em um tempo satisfatório.

4.1 Coleta e Análise dos dados

Conforme apresentado na Seção 3.5, foram definidos, no SISDOT, testes em diferentes níveis de granularidade, além dos testes baseados na DSL e Cucumber. O teste de software destina-se a mostrar que o sistema faz o que se pretende e descobrir defeitos, mas nunca mostrar sua ausência. Os testes de software foram usados na **RQ.1** como uma forma de aumentar a confiança na corretude da solução para geração de QDMs.

Foram definidos 100 cenários que servem de base para geração automática de artefatos para testar cada QC declarado no *plugin* maven criado. Esses cenários cobrem diversas validações de casos básicos de regras de chamadores, aumentando a confiança de que a geração de QDMs funciona corretamente. Foram gerados automaticamente testes para um conjunto de 20 QCs, selecionados de forma a expressar a variedade dos tipos existentes, e todos os cenários passaram nos testes executados, ou seja, não foram encontrados erros nos resultados esperados.

Nos testes manuais, os chamadores declarados são, geralmente, uma combinação dos casos de teste gerados automaticamente. Neste tipo de teste é necessária a presença de um analista de negócio para auxiliar na definição dos chamadores e dos resultados esperados. Durante a execução dos testes os dados obtidos na geração do QDM para determinado QC são comparados com os resultados esperados pelo analista de negócio. Pela combinação dos casos de teste mais básicos gerados automaticamente e definição dos resultados esperados por um analista de negócio, este tipo de teste demonstra-se importante para ajudar na garantia da qualidade da solução desenvolvida.

A primeira pergunta de pesquisa é respondida qualitativamente. Embora isso possa parecer decepcionante (sob a perspectiva de métodos empíricos), fortes evidências, coletadas dos especialistas do domínio, mostram que a abordagem proposta produziu QDMs corretos. Acredita-se que essa corretude se deve a vários fatores, incluindo o envolvimento dos especialistas do domínio que ajudaram a entender completamente os requisitos, a experiência da equipe de desenvolvimento e os resultados satisfatórios dos testes executados. No entanto, a decisão de não escrever as regras de baixo nível no nível do código-fonte, mas usando uma abordagem de geração de programa que reduziu significativamente o esforço necessário para escrever essas regras, também contribuiu para alcançar essa confiança sobre a corretude. Em uma reunião com os especialistas do domínio, um dos especialistas disse: "Fiquei surpreso com a corretude da solução. Na versão anterior do sistema, tínhamos que iniciar várias interações até ficarmos confiantes sobre o funcionamento correto dos chamadores (regras de alto nível).

Para responder as outras questões de pesquisa, a coleta dos dados foi realizada utilizando as seguintes métricas: linhas de código (LOC) e tempo de execução. A escolha dessas duas métricas está relacionada ao tamanho do código, que pode indicar uma maior

produtividade; e à satisfação do requisito funcional que determina o tempo máximo de execução para a geração de QDM.

Foram selecionados os 100 chamadores declarados no ambiente de desenvolvimento do SISDOT. Esses chamadores são do tipo ABAS (Estrutura Organizacional por Componentes) (Seção 3.2) e correspondem a chamadores reais utilizados pelo Exército Brasileiro (EB), e serviram para a realização de alguns testes na geração de QDMs pela equipe de desenvolvimento. Foram gerados também, de forma automática, 100 chamadores do tipo ÁRVORE (Estrutura Organizacional Completa) (Seção 3.2). Foram selecionados 100 QCs para a coleta de métrica de tempo de geração de QDMs, e para cada QC foi gerado automaticamente um chamador do tipo ÁRVORE. Essa geração deu-se a partir da seleção aleatória de cargos, criando uma regra do tipo CARGO_QC para cada cargo. A quantidade de cargos selecionados deu-se pela seleção aleatória de um número entre 1 e 1/3 da quantidade total de cargos do QC, sendo que esse limite superior não poderia exceder 60.

Os dados analisados para a **RQ.2** foram coletados a partir desses 200 chamadores. Cada um desses chamadores, representados na linguagem Java, foi convertido para o seu correspondente no formato da DSL com o uso do Xtext, e os dados referentes ao número de linhas de código foi coletado para cada uma das representações. Os dados coletados foram exportados para o formato CSV, para que pudessem ser analisados usando o ambiente R [208].

Na análise de dados para a **RQ.2**, que compara o número de linhas de código entre as duas abordagens distintas para representar chamadores, foram criadas duas colunas adicionais: diferença, que mede a diferença no número de linhas de código necessárias especificar chamadores usando Java e a DSL; e percentual de redução, que indica quanto menos código é necessário para especificar um chamador usando a DSL, quando comparado à especificação direta no código Java.

A Tabela 4.1 apresenta uma amostra dos dados coletados, incluindo as colunas adicionais. A primeira coluna contém o código do chamador. Os chamadores com código entre 1 e 20 representam chamadores do tipo ABAS, enquanto os outros chamadores são do tipo ARVORE. A segunda coluna contém a quantidade de linhas de código para representar o chamador na DSL e a terceira coluna contém a quantidade de linhas para representar o mesmo chamador na linguagem Java. A quarta coluna contém a diferença do número de linhas de código entre as representações Java e DSL. A quinta coluna apresenta quantos em quantos por cento a representação DSL é menor que a representação Java. Segue um exemplo hipotético para o cálculo dessa coluna: dado um chamador em Java com 100 linhas e um chamador em DSL com 30 linhas, o chamador em DSL é 70% menor que o chamador em Java.

Tabela 4.1: Comparação do número de linhas de código entre DSL e Java.

Código	**DSL**	**Java**	**Diferença**	**Menor (%)**
1	10	26	16	62
2	10	25	15	60
3	10	32	22	69
4	11	52	41	79
5	15	41	26	64
6	9	16	7	44
7	18	21	3	15
8	12	25	13	52
9	9	11	2	19
10	11	23	12	53
11	7	27	20	75
12	8	41	33	81
13	8	14	6	43
14	10	23	13	57
15	7	45	38	85
16	11	24	13	55
17	7	21	14	67
18	7	9	2	23
19	9	19	10	53
20	8	12	4	34
122	10	48	38	80
123	8	44	36	82
124	10	54	44	82
125	8	39	31	80
126	12	47	35	75
127	9	43	34	80
128	9	39	30	77
129	9	27	18	67
130	12	25	13	52
131	12	45	33	74
132	11	48	37	78
133	11	60	49	82
134	8	38	30	79
135	11	69	58	85
136	8	32	24	75
137	10	59	49	84
138	11	49	38	78
139	11	18	7	39
140	8	67	59	89
141	10	32	22	69

Na Tabela 4.2 são apresentadas algumas estatísticas descritivas dos dados coletados, contendo as seguintes colunas: valor mínimo, mediana, média, desvio padrão e valor máximo. É possível observar, por exemplo, que a menor diferença entre a quantidade de linhas de código foi de 2 linhas enquanto a maior diferença foi de 121 linhas. É possível observar ainda que o chamador declarado em DSL é, em média, 54.66% menor do que o chamador declarado em Java.

Tabela 4.2: Análise dos dados de comparação entre DSL e Java

	Min.	**Mediana**	**Média**	**SD**	**Max.**
DSL	7.00	9.00	9.67	2.01	18.00
Java	9.00	23.00	28.66	18.47	130.00
Diferença	2.00	13.00	18.98	18.25	121.00
Menor (%)	14.29	56.26	54.66	21.68	93.08

No estudo estatístico, a relação entre duas ou mais variáveis denomina-se correlação. Existe uma forte correlação entre as variáveis Java e Diferença, de 0.994. Essa correlação pode ser visualizada no gráfico de dispersão apresentado na Figura 4.1. A partir desse dado é possível identificar que à medida que o tamanho do código Java cresce a diferença para a representação em DSL também aumenta.

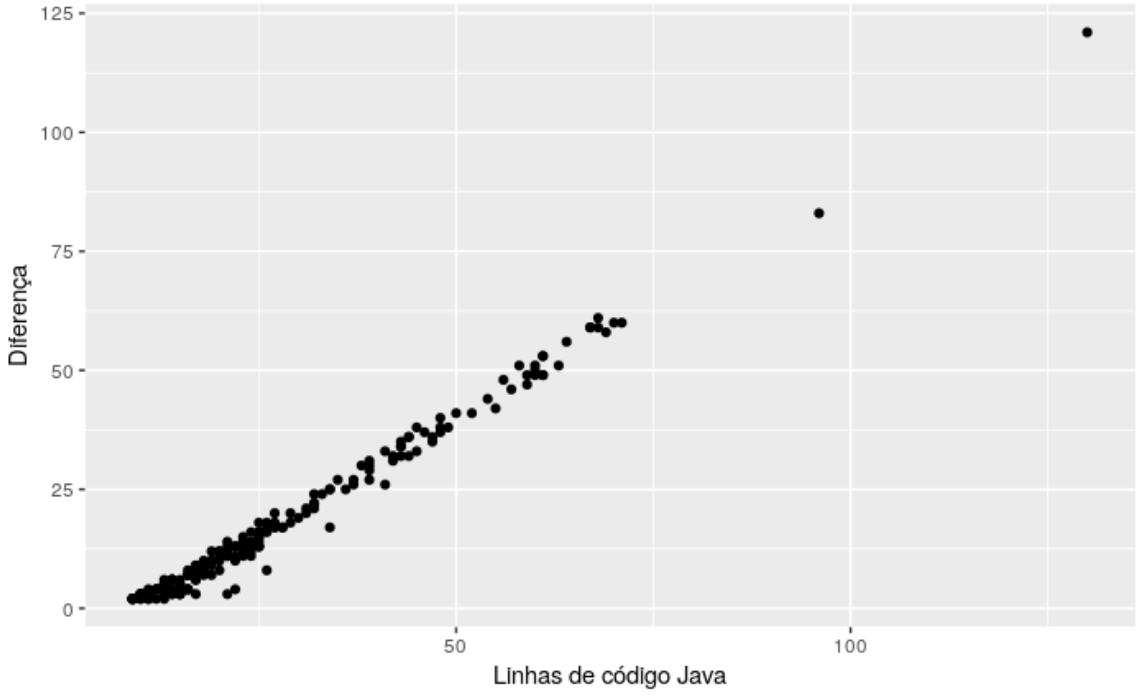


Figura 4.1: *Correlação entre Java e Diferença*

Para a coleta da métrica de tempo de execução, relativa à **RQ.3**, foi utilizado um computador com processador Intel(R) Core(TM) i7-4790, com 16GB de memória RAM, com o sistema operacional Ubuntu 18.04.1 LTS, 64 bits, com kernel 4.15.0-34-generic. A configuração utilizada é, de certa forma, comum em computadores domésticos. Espera-se uma melhoria considerável nos tempos de geração de QDMs ao utilizar máquinas mais potentes, como os servidores do EB. Foram usados 100 QCs, escolhidos de forma a representar os variados tipos de OM, naturezas, subnaturezas e valores. Para cada QC foram gerados QDMs usando as seguintes quantidades de chamadores: 70, 200, 400, 700, 1000, 1500, 2000.

Como só haviam 200 chamadores declarados, os conjuntos com quantidade maior contém chamadores repetidos. A seleção dos chamadores foi realizada de maneira randômica, mas com o mesmo *seed*, para que os mesmos conjuntos fossem utilizados nas gerações dos QDMs de diferentes QCs. A seleção aleatória de chamadores ocorreu para todos os conjuntos, seja para selecionar um subconjunto dos chamadores disponíveis ou para completar um conjunto maior, repetindo alguns dos chamadores. Os resultados obtidos foram armazenados em arquivo no formato CSV para análise no ambiente R.

A Tabela 4.3 apresenta uma amostra dos resultados obtidos com a execução. A primeira coluna contém o código do QC, e da segunda coluna em diante contém o tempo, em milissegundos, para a geração dos QDMs usando a quantidade de chamadores indicada no *header* da respectiva coluna. Por exemplo: a segunda coluna contém os tempos de geração de QDMs usando 70 chamadores, a terceira coluna contém os tempos de geração de QDMs usando 200 chamadores, e assim por diante.

Nesta Tabela é possível observar, por exemplo, que o tempo para geração de um QDM para o QC com código 702311 (Batalhão de Infantaria) usando 1000 chamadores foi de 6185 ms, o maior tempo entre todos aferidos para esta quantidade de chamadores. Para este mesmo QC foram gerados QDMs com tempos de 4301 ms, 7753 ms e 10098 ms para 700, 1550 e 2000 chamadores, respectivamente. Os tempos, da coluna que contém os dados relativos a 1000 chamadores, marcados em negrito representam o maior e o menor valor encontrados nos dados coletados.

Foi realizada uma análise sobre os dados coletados, incluindo informações levantadas sobre cada QC, como: quantidade total de cargos e de militares. Não foi possível encontrar uma relação entre os tempos de execução e a quantidade total de cargos e nem com a quantidade de militares. Por exemplo: a correlação entre a quantidade de cargos e os tempos usando 1000 chamadores é de 0.1565, indicando que não há uma relação clara entre essas variáveis. Outro exemplo: o maior tempo de geração ao usar 1000 chamadores foi de 6185 ms para um QC que possui 242 cargos. Mas para o QC que possui mais cargos, com 1235, o tempo de geração foi de 5003 ms. O QC com o menor tempo de geração

Tabela 4.3: Tempo (ms) para geração de QDMs.

QC	**70**	**200**	**400**	**700**	**1000**	**1500**	**2000**
101410	362	890	1866	3237	4846	6877	9231
206410	384	895	1875	3170	4838	6975	9348
215304	413	960	2026	3358	5235	6851	9416
231303	370	934	1954	3221	4931	6715	9264
300400	385	891	1903	3132	4808	6858	9283
500312	379	944	1992	3495	4709	6637	9213
524311	379	942	1999	3231	4991	7061	9686
609420	369	911	1895	3162	4875	6905	9259
619321	366	917	1945	3269	5010	6976	9163
627320	403	934	2006	3314	5081	7522	9011
656320	371	900	1868	3197	5058	7086	9398
702311	705	1524	2902	4301	**6185**	7753	10098
710312	404	954	1996	3237	4861	6728	9102
725310	380	920	1872	3104	4860	6957	9190
727310	364	922	1949	3206	4722	6795	9222
757311	403	944	2084	3359	5263	6960	9230
950401	393	928	1993	3235	5076	7282	10060
1108400	398	968	2048	3458	4703	6727	9575
1112401	388	894	1880	3165	4837	6873	9270
5716900	373	918	1979	3423	**4662**	6504	9032
7805004	382	1071	1808	2979	4710	6915	9287
1470310	348	879	1861	3091	4711	6824	8996
739310	383	914	1962	3096	4722	6753	9032
8204900	347	847	1841	3052	4731	6907	9348
1501402	348	871	1869	3092	4784	6873	9581
6512900	362	899	1882	3161	4796	6825	9691
2522135	356	902	1890	3115	4819	6928	9153
302401	354	900	1871	3080	4898	6875	9272
1115310	374	890	1857	3131	4904	6891	9201
7038901	391	926	1955	3268	5026	7450	9241
6800901	339	905	1870	3138	4895	6982	8975
6993900	352	868	1861	3090	4741	6793	9445
2205400	390	901	1890	3186	5038	7107	9146
7017901	356	893	1852	3061	4773	6760	9201
2587150	365	887	1854	3147	5132	6902	9034
8417000	388	892	1917	3191	4892	7010	9061
903311	404	890	1910	3180	4843	6950	9346
9015002	369	882	1893	3070	4783	6792	8989
5406194	368	895	1924	3124	4870	6749	8955

Tabela 4.4: Análise dos tempos para geração de QDMs.

Quantidade	Min.	Mediana	Média	SD	Max.
700	2979	3179.5	3196.71	144.2615	4301
1000	4662	4909.5	4925.57	179.7946	6185
1500	6504	6968.5	7002.41	229.0101	7753
2000	8650	9284.5	9337.90	276.2198	10098

(4662 ms) possui 3 cargos, enquanto o QC que apresenta a menor quantidade de cargos (2) teve tempo de geração de 4731 ms.

Na Tabela 4.4 são apresentadas algumas estatísticas descritivas dos dados coletados, contendo as seguintes colunas: valor mínimo, mediana, média, desvio padrão e valor máximo. É possível observar, por exemplo, que o tempo médio para geração de QDMs ao usar 1000 chamadores foi de 4925 ms, enquanto para 2000 chamadores foi de 9337 ms. Estes tempos médios de execução são importantes de se observar devido ao requisito funcional levantado que limita o tempo de geração de um QDM a 10 segundos, quando usar 1000 chamadores, a quantidade esperada no sistema.

A Figura 4.2 apresenta um gráfico dos tempos necessários para geração de QDMs para 6 diferentes QCs. Para cada QC foram gerados 5 QDMs, com diferentes quantidades de chamadores: 400, 700, 1000, 1500, 2000.

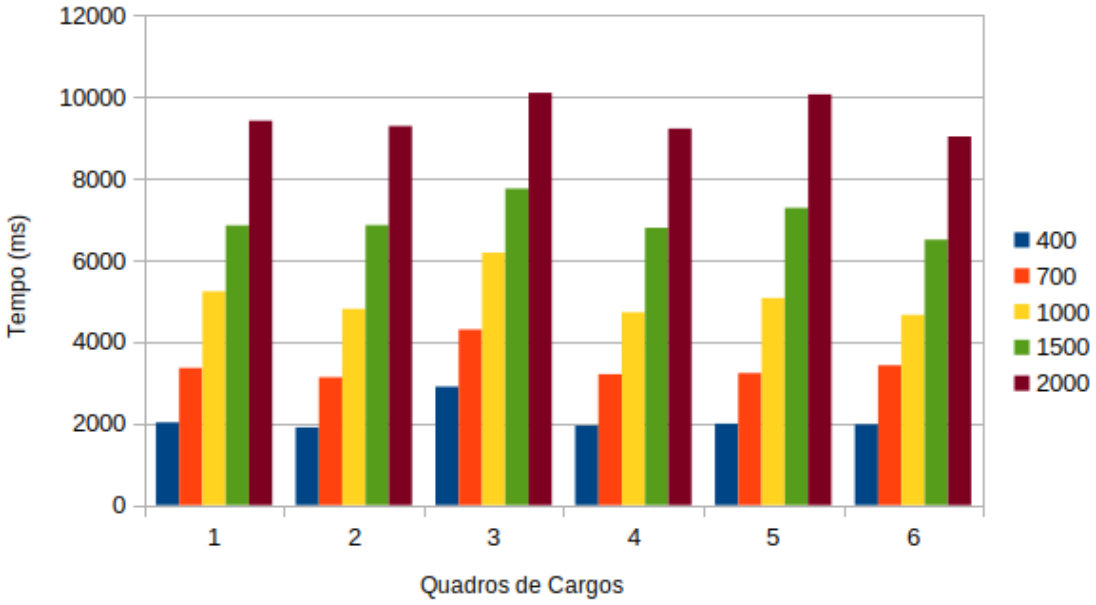


Figura 4.2: *Tempo (ms) de geração de QDM*

4.2 Discussão

Os testes realizados durante o desenvolvimento, juntamente com a geração automática de testes, envolvimento e *feedback* dos especialistas do domínio, serviram para aumentar a confiança na corretude da solução, respondendo satisfatoriamente a **RQ.1**.

Ao analisar os dados relacionados à **RQ.2**, foi possível observar que a representação de um chamador usando a DSL desenvolvida é, em média, 54% menor que a representação do mesmo chamador em Java. Isso indica um possível ganho de produtividade para o desenvolvedor ao escrever casos de teste, já que ela terá que escrever uma quantidade menor de código. Além disso, usando a DSL, é possível especificar chamadores usando uma abordagem declarativa, que é próxima do vocabulário do domínio do problema.

Com relação à terceira questão de pesquisa, o limite imposto pelo requisito funcional de 10 segundos não foi ultrapassado, mesmo considerando 2000 chamadores, que é o dobro do número de chamadores esperados para o sistema em produção.

4.3 Ameaças à validade

Foram considerados apenas 200 chamadores neste estudo, que serviu de base para medir o tempo de geração de QDMs. Espera-se que o SISDOT tenha cerca de 1000 chamadores no ambiente de produção. Para mitigar a ameaça relacionada ao pequeno número de chamadores disponíveis para teste, foi considerada sua repetição ao realizar o teste de desempenho do aplicativo. Desta forma, espera-se que o sistema apresente um comportamento semelhante ao ambiente de produção (em relação ao desempenho). A mesma situação ocorre com a comparação de linhas de código entre as declarações de chamadores em DSL e Java. Como o número de chamadores é relativamente pequeno, pode não ser possível generalizar os resultados encontrados.

Capítulo 5

Conclusões

O objetivo desse trabalho foi descrever as principais decisões arquiteturais e de *design* adotadas para implementar os mecanismos para derivar regras de negócio de alto nível em regras específicas para o motor de regras Drools e a definição de uma DSL que facilita a declaração de regras de negócio de alto nível. Além da geração automática de testes para complementar os testes definidos manualmente.

Foi realizado um estudo de caso prático para validar a arquitetura proposta a partir da avaliação do principal caso de uso do Sistema de Dotação de Material (SISDOT), relacionado à geração de QDMs. Neste estudo de caso foram levantadas três perguntas de pesquisa, que foram respondidas satisfatoriamente.

A partir dos resultados obtidos com o estudo de caso foi possível observar uma diminuição considerável, de 54% em média, da quantidade de linhas de código necessárias para a declaração de um chamador usando a DSL criada em comparação com a representação do mesmo chamador em Java, indicando um possível ganho de produtividade para o testador. Essa diminuição da quantidade de linhas de código na DSL pode ser explicada pela possibilidade de declaração de várias regras do mesmo tipo em apenas uma linha de código, indicando apenas seus valores na lista de valores. Enquanto essas mesmas regras, quando declaradas na linguagem Java, devem ser realizadas uma em cada linha de código, por motivo de legibilidade.

Outro resultado do estudo de caso demonstrou a aderência do sistema ao requisito funcional declarado, de limitar o tempo de geração de um QDM em 10 segundos. Esse limite não foi ultrapassado nem quando foi utilizado o dobro da quantidade de chamadores esperados para o sistema em produção. Para a coleta desses dados foi utilizado um computador com configuração que é, de certa forma, de utilização doméstica. Pretende-se ter uma melhora significativa nos tempos de geração de QDM quando o sistema estiver implantado nos servidores do Exército Brasileiro (EB), que possuem *hardware* especializado.

Além dos testes unitários e de integração implementados durante o desenvolvimento do SISDOT, foram definidos testes que utilizam a DSL para declaração dos chamadores, além da especificação dos cenários de teste com o Cucumber, assim como uma estrutura para suportar suas execuções. Estes testes foram criados exclusivamente para aumentar a confiança na corretude dos QDMs resultantes.

Foram criados dois tipos de testes, um com geração automática dos artefatos usados nos testes e outro onde os cenários de testes foram definidos manualmente. Na geração automática de testes, alguns dos valores das propriedades disponíveis foram gerados randomicamente, seguindo a técnica Property-based Testing (PBT). Os cenários de teste gerados foram então convertidos para artefatos nos formatos DSL e Cucumber. Já os testes manuais necessitam do auxílio de um analista de negócio para que possam ser corretamente especificados e são geralmente compostos pelos cenários básicos definidos nos testes gerados automaticamente. Neste tipo de teste há a possibilidade de utilização de mais de um chamador em cada cenário definido, além de servir para detecção de inconsistências no conjunto de chamadores declarados.

Acredita-se que os 100 cenários de teste gerados automaticamente para cada QDM, juntamente com os casos de teste definidos manualmente, aumentaram a confiança na corretude dos resultados dos QDMs gerados pelo sistema.

A partir do conjunto dos resultados obtidos no estudo de caso prático, assim como dos *feedbacks* positivos recebidos dos analistas de negócio, é possível afirmar que a solução desenvolvida possui qualidade e confiabilidade suficientes para ser utilizada em produção.

5.1 Contribuições

Apesar de a arquitetura alvo deste trabalho focar nas necessidades específicas do EB, acredita-se que os sistemas logísticos de outras instituições podem se beneficiar dessas decisões.

5.2 Trabalhos Futuros

Para refinar os resultados obtidos com a arquitetura proposta foram definidos os os seguintes trabalhos futuros:

- *Refactoring* dos serviços de QDM para otimizar a sua geração;
- Definir mais chamadores reais, com o auxílio dos especialistas do domínio;
- Definir mais cenários para a geração de testes automáticos;

- Coletar e analisar as métricas novamente quando houver uma quantidade expressiva de chamadores, os quais continuam em constante criação pelo Exército Brasileiro, com o intuito de testar a arquitetura proposta;
- Incluir a funcionalidade de autocompletar de valores no editor de DSL para facilitar sua declaração;
- Implementar outro sistema ou estudo de caso, em outro contexto, que utilize as decisões arquiteturais e de *design* adotadas no SISDOT.

Referências

[1] Hay, David e Keri Anderson Healy: *Defining Business Rules*. Relatório Técnico, The Business Rules Group, 2000. http://www.businessrulesgroup.org/first_paper/BRG-whatisBR_3ed.pdf. 1

[2] Morgan, Tony: *Business rules and information systems: aligning IT with business goals*. Addison-Wesley Professional, 2002. 1

[3] BRASIL, ABPMP: *Bpm cbok v3.0: Guia para o gerenciamento de processos de negócio-corpo comum de conhecimento*, 2013. 1

[4] Alencar, Thales Mota de: *A gestão de suprimento classe V (Munição) no Exército Brasileiro adequada ao tempo de paz*. Ciências militares, Escola de Comando e Estado-Maior do Exército, 2014. http://www.eceme.eb.mil.br/images/IMM/producao_cientifica/teses/thales-mota-de-alencar.pdf. 2

[5] Braz, Márcio Alexandre de Lima: *A logística militar e o serviço de intendência: uma análise do programa excelência gerencial do exército brasileiro*. Administração pública, Fundação Getúlio Vargas, 2004. https://bibliotecadigital.fgv.br/dspace/handle/10438/3394. 2

[6] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest e Clifford Stein: *Algoritmos: Teoria e Prática*. Elsevier Editora Ltda., 3ª edição, 2012, ISBN 978-85-352-3699-6. 2

[7] Sedgewick, Robert e Kevin Wayne: *Algorithms*. Addison-Wesley, 4ª edição, 2011, ISBN 978-0-321-57351-3. 2

[8] Ballou, Ronald H: *Gerenciamento da cadeia de suprimentos: planejamento, organização e logística empresarial*. Bookman, 2001. 5, 6

[9] Bowersox, Donald J. e David J. Closs: *Logistica Empresarial: o processo de integração da cadeia de suprimentos*. ATLAS, 2004, ISBN 978-8522428779. 5

[10] Gasnier, Daniel Georges: *A Dinâmica dos Estoques: guia prático para planejamento, gestão de materiais e logística*. IMAM, 2003, ISBN 978-8589824453. 5

[11] Gewandsznajder, Fernando: *O que é o método científico*, 1987. http://bibliotecadigital.fgv.br/dspace/handle/10438/9273. 8

[12] Marconi, Marina de Andrade e Eva Maria Lakatos: *Fundamentos de Metodologia Científica*. Editora Atlas, 5ª edição, 2003, ISBN 85-224-3397-6. 8

[13] Prodanov, Cleber Cristiano e Ernani Cesar de Freitas: *Metodologia do trabalho científico: métodos e técnicas da pesquisa e do trabalho acadêmico*. Editora Feevale, 2ª edição, 2013, ISBN 978-85-7717-158-3. 8

[14] Praça, Fabíola Silva Garcia: *Metodologia da pesquisa cientÍfica: OrganizaÇÃo estrutural e os desafios para redigir o trabalho de conclusÃo*. Diálogos Acadêmicos, 8:72–87, julho 2015, ISSN 0486-6266. http://uniesp.edu.br/sites/_biblioteca/revistas/20170627112856.pdf. 8

[15] Runeson, Per e Martin Höst: *Guidelines for conducting and reporting case study research in software engineering*. Empirical Softw. Engg., 14(2):131–164, apr 2009, ISSN 1382-3256. http://dx.doi.org/10.1007/s10664-008-9102-8. 8, 90

[16] Thiollent, Michel: *Metodologia da pesquisa-ação*, 2011. 8

[17] Runeson, Per, Martin Host, Austen Rainer e Bjorn Regnell: *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012. 9

[18] Travassos, Guilherme Horta, Dmytro Gurov e EAGG Amaral: *Introdução à engenharia de software experimental*. UFRJ, 2002. 9

[19] Cruzes, Daniela S., Tore Dybå, Per Runeson e Martin Höst: *Case studies synthesis: a thematic, cross-case, and narrative synthesis worked example*. Empirical Software Engineering, 20(6):1634–1665, 2015, ISSN 15737616. 9

[20] Cavaye, A. L. M.: *Case study research: a multi-faceted research approach for IS*. Information Systems Journal, 6(1 996):227–242, 1996, ISSN 13501917. 9

[21] Kitchenham, B., L. Pickard e S. L. Pfleeger: *Case studies for method and tool evaluation*. IEEE Software, 12(4):52–62, Jul 1995, ISSN 0740-7459. 9

[22] Júnior, Reinaldo: *A logística no âmbito da história*. REVISTA REVERTE, n. 6:ISSN 2236–1294, março 2008. 11

[23] Lazenby, J.F.: *Logistics in Classical Greek Warfare*. War in History, 1(1):3–18, mar 1994, ISSN 0968-3445. http://journals.sagepub.com/doi/10.1177/096834459400100102. 11

[24] Ballou, Ronald H.: *The evolution and future of logistics and supply chain management*. European Business Review, 19(4):332–348, 2007, ISSN 0955-534X. http://www.emeraldinsight.com/doi/10.1108/09555340710760152. 11

[25] Serio, Luiz Carlos;, Mauro ; Sampaio e Susana Pereira: *A Evolução dos Conceitos de Logística: um estudo na cadeia automobilística no Brasil*. Em *EnANPAD 2006*, páginas 1–15, 2006. http://www.anpad.org.br/enanpad/2006/dwn/enanpad2006-golb-2727.pdf. 11

[26] Prebilič, Vladimir: *Theoretical aspects of military logistics*. Defense & Security Analysis, 22(2):159–177, 2006, ISSN 14751798. https://doi.org/10.1080/14751790600764037. 12

[27] Tzu, Sun e Sun Pin: *A Arte da Guerra*. WMF Martins Fontes, 3ª edição, 2014, ISBN 978-8578277970. Tradutora: Ana Aguiar Cotrim. 12

[28] Maquiavel, Nicolau: *A Arte da Guerra*. L&PM, 1ª edição, 2008, ISBN 978-8525417343. Tradutor: Eugênio Vinci de Moraes. 12

[29] Leibniz, Gottfried Wilhelm: *Projeto a respeito de uma nova enciclopédia que deve ser redigida pelo método da descoberta*. Scientiae Studia, 5:95 – 107, março 2007, ISSN 1678-3166. http://dx.doi.org/10.1590/S1678-31662007000100006. 12

[30] Brasil, Ministério da Defesa: *Manual de campanha c 100-10 - logÍstica militar terrestre*, 2003. http://bibliotecamilitar.com.br/wp-content/uploads/2016/02/02-manual_c_100-10-logistica_militar_terrestre.pdf, 2ª edição, 2003. 12, 15, 16, 17

[31] Rutner, Stephen M., Maria Aviles e Scott Cox: *Logistics evolution: a comparison of military and commercial logistics thought*. The International Journal of Logistics Management, 23(1):96–118, may 2012, ISSN 0957-4093. http://www.emeraldinsight.com/doi/10.1108/09574091211226948. 12

[32] Simon, Steve John: *The art of military logistics*. Communications of the ACM, 44(6):62–66, jun 2001, ISSN 0001-0782. http://doi.acm.org/10.1145/376134.376167. 12

[33] Clausewitz, Carl Von: *Da Guerra*. WMF Martins Fontes, 3ª edição, 2010, ISBN 978-8578272111. Tradução: Luiz Carlos Nascimento e Silva do Valle; Ensaios explicativos: Peter Paret, Michael Howard e Bernard Brodie. 12

[34] Riley, Lieutenant General Jonathon: *Logistics and Supply in Renaissance Armies*. Arms & Armour, 8(2):139–151, oct 2011, ISSN 1741-6124. https://doi.org/10.1179/174962611X13155725685997. 12

[35] Marques, Wagner Rodrigues: *O atual nível de terceirização logística das forças armadas brasileiras*, 2014. http://www.esg.br/images/Monografias/2014/MARQUES.pdf. 13

[36] Brasil, Ministério da Defesa: *Manual de campanha eb20-mc-10.204 - logÍstica*, 2014. http://bdex.eb.mil.br/jspui/bitstream/123456789/434/1/EB20-MC-10.204.pdf, 3ª edição, 2014. 13, 14, 16

[37] OTAN, Logistics Committee: *Nato logistics handbook*, 2012. https://www.nato.int/docu/logi-en/logistics_hndbk_2012-en.pdf, 2nd Edition, 2012. 16

[38] Brooks Jr, Frederick P: *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., 1995. 17

[39] Pressman, Roger S: *Software engineering: a practitioner's approach*. McGraw-Hill, 2005. 17, 44, 45, 46, 50

[40] Mellor, Stephen J: *MDA Destilada: princípios de arquitetura orientada por modelos*. Addison-Wesley Professional, 2004. 17, 24

[41] Garlan, David e Mary Shaw: *An introduction to software architecture*. Knowl. Creat. Diffus. Util., 1(January):1–40, 1994, ISSN 0270-5257. http://portal.acm.org/citation.cfm?id=865128. 18, 22

[42] Dijkstra, Edsger W: *The Structure of the "THE" -Multiprogramming System*. Communications of the ACM, 11(5):341–346, 1968, ISSN 00010782. 18

[43] Parnas, David L.: *On a "buzzword": Hierarchical structure*. IFIP Congress 74, North Holland Publishing Company., páginas 139–148, 1974. 18

[44] Parnas, David L.: *On the criteria to be used in decomposing systems into modules*. Communications of the ACM, 15(12):1053–1058, dec 1972, ISSN 00010782. http://portal.acm.org/citation.cfm?doid=361598.361623. 18

[45] Tarr, Peri e M Sutton: *N degrees of separation : Multi-dimentional separation of concerns*. Em *ICSE '99 Proceedings of the 21st international conference on Software engineering*, páginas 107–119, 1999. 18

[46] Deremer, Frank e Hans Kron: *Programming-in-the large versus programming-in-the-small*. Proceedings of the international conference on Reliable software -, páginas 114–121, 1975, ISSN 03621340. http://portal.acm.org/citation.cfm?doid=800027.808431. 18

[47] Kruchten, P., H. Obbink e J. Stafford: *The past, present, and future for software architecture*. IEEE Software, 23(2):22–30, mar 2006, ISSN 0740-7459. http://ieeexplore.ieee.org/document/1605175/. 18

[48] Shaw, Mary e Paul Clements: *The golden age of software architecture*. IEEE Software, 23(2):31–39, 2006, ISSN 07407459. 18

[49] Perry, Dewayne E. e Alexander L. Wolf: *Foundations for the study of software architecture*. ACM SIGSOFT Software Engineering Notes, 17(4):40–52, 1992, ISSN 01635948. http://portal.acm.org/citation.cfm?doid=141874.141884. 18

[50] Björnander, Stefan: *Architecture Description Languages*, capítulo 5, páginas 97–134. Wiley-Blackwell, 2013, ISBN 9781118602638. https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118602638.ch5. 18

[51] Medvidovic, N. e R.N. Taylor: *A classification and comparison framework for software architecture description languages*. IEEE Transactions on Software Engineering, 26(1):1–24, 2000, ISSN 0098-5589. https://ieeexplore.ieee.org/document/825767/. 18

[52] Kruchten, P.: *The 4+1 view model of architecture*. Software, IEEE, November 1(November):9, 1995, ISSN 0740-7459. http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=469759. 18, 25

[53] Barbacci, Mario, Mark H. Klein, Thomas A. Longstaff e Charles B. Weinstock: *Quality attributes*. Relatório Técnico, Software Engineering Institute, Carnegie Mellon University, 1995. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=12433. 18

[54] Kazman, R., L. Bass, G. Abowd e M. Webb: *Saam: a method for analyzing the properties of software architectures*. Proceedings of 16th International Conference on Software Engineering, páginas 81–90, 1994, ISSN 0270-5257. http://ieeexplore.ieee.org/document/296768/. 18

[55] Hohpe, Gregor, Ipek Ozkaya, Uwe Zdun e Olaf Zimmermann: *The software architect 's role in the digital age*. IEEE Software, 33(6):30 – 39, 2016, ISSN 0740-7459. 18

[56] Bosch, J.: *Software architecture: The next step*. Lecture Notes in Computer Science, 3047:194–199, 2004. https://www.scopus.com/inward/record.uri?eid=2-s2.0-35048903201&partnerID=40&md5=eb8f8f0f70f477937ced17d564731443. 18

[57] Jansen, A. e J. Bosch: *Software architecture as a set of architectural design decisions*. Em *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, volume 2005, páginas 109–120. IEEE, 2005, ISBN 0-7695-2548-2. http://ieeexplore.ieee.org/document/1620096/. 18

[58] Len, Bass, Clements Paul e Kazman Rick: *Software Architecture in Practice*. Addison-Wesley, 2013. 18, 19, 20, 22, 23, 24, 25

[59] Clements, Paul, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord e Judith Stafford: *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2011. 19, 22, 23, 24, 25, 26

[60] Fowler, Martin: *Design stamina hypothesis: Is it worth the effort to design software well?* https://martinfowler.com/bliki/DesignStaminaHypothesis.html, 2007. Online; acessado em 30 de maio de 2018. 20

[61] Sommerville, Ian: *Software engineering*. Addison-Wesley, 2011. 20, 46, 50

[62] Bass, Len, Mark Klein e Felix Bachmann: *Quality Attribute Design Primitives and the Attribute Driven Design Method*. Em *Software Product-Family Engineering*, páginas 169–186. Springer Berlin Heidelberg, 2002, ISBN 978-3-540-47833-1. http://link.springer.com/10.1007/3-540-47833-7{_}17. 20

[63] Garlan, David: *Software architecture: a roadmap*. Proceedings of the Conference on The Future of Software Engineering, páginas 91–101, 2000, ISSN 0164-1212. http://dl.acm.org/citation.cfm?id=336537. 21

[64] Caporuscio, Mauro, Paola Inverardi e Patrizio Pelliccione: *Formal analysis of architectural patterns*. Em *Software Architecture*, EWSA 2004, páginas 10–24, 2004, ISBN 978-3-540-24769-2. https://link.springer.com/chapter/10.1007/978-3-540-24769-2_2. 22

[65] Avgeriou, Paris e Uwe Zdun: *Architectural patterns revisited - a pattern language*. Information Systems Journal, 81:1–39, 2005. http://dblp.uni-trier.de/db/conf/europlop/europlop2005.html{#}AvgeriouZ05. 22

[66] Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad e Michael Stal: *Pattern-oriented software architecture: A system of patterns*. Wiley, 1996. 22

[67] Kiczales, Gregor, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean Marc Loingtier e John Irwin: *Aspect-oriented programming*. Em *ECOOP'97: Object-Oriented Programming*, páginas 220–242, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg, ISBN 978-3-540-69127-3. https://link.springer.com/chapter/10.1007/BFb0053381. 22

[68] Apel, Sven: *How aspectj is used: An analysis of eleven aspectj programs*. Journal of Object Technology, 9(1):117–142, janeiro 2010, ISSN 1660-1769. http://www.jot.fm/contents/issue{_}2010{_}01/article2.html. 22

[69] Laddad, Ramnivas: *AspectJ in Action: practical aspect-oriented programming*. Manning Publications Co, 2003, ISBN 1-930110-93-6. 22

[70] Merson, Paulo: *Data model as an architectural view*. Relatório Técnico CMU/SEI-2009-TN-024, Software Engineering Institute, Carnegie Mellon University, 2009. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=9033. 22

[71] Tanenbaum, Andrew S. e David J. Wetherall: *Computer networks*. Prentice Hall, 5ª edição, 2010, ISBN 978-9332518742. 23

[72] Kamal, Ahmad Waqas e Paris Avgeriou: *Modeling Architectural Patterns' Behavior Using Architectural Primitives*. Em *Software Architecture*, volume 5292 LNCS, páginas 164–179. Springer Berlin Heidelberg, 2008, ISBN 3540880291. https://link.springer.com/chapter/10.1007/978-3-540-88030-1_13. 24

[73] Costa, P.H.T. e E. Dias Canedo: *Using a mdd approach to develop software systems*. Em *2015 10th Iberian Conference on Information Systems and Technologies, CISTI 2015*, 2015, ISBN 9789899843455. https://ieeexplore.ieee.org/document/7170371/. 24, 38

[74] Rumbaugh, James, Ivar Jacobson e Grady Booch: *Unified modeling language reference manual, the*. Addison-Wesley, 2004. 24

[75] Arrington, CT: *Enterprise Java with UML*. John Wiley & Sons, 2001. 24

[76] Booch, Grady: *Object oriented analysis & design with application*. Addison-Wesley, 2007. 24

[77] Bourque, P e RE Fairley: *Swebok - guide to the software engineering body of knowledge, version 3.0. 2014*. 25, 44, 45, 74

[78] Salatino, Mauricio, Mariano De Maio e Esteban Aliverti: *Mastering JBoss Drools 6*. Packt Publishing Ltd, 2016, ISBN 978-1-78328-862-5. 26, 30

[79] Graham, Ian: *Business rules management and service oriented architecture: a pattern language*. John wiley & sons, 2007, ISBN 978-0-470-02721-9. 26, 28

[80] Grosan, Crina e Ajith Abraham: *Rule-Based Expert Systems*, páginas 149–185. Springer Berlin Heidelberg, 2011, ISBN 9783642210037. https://doi.org/10.1007/978-3-642-21004-4_7. 26, 27, 28, 30, 31, 32

[81] Russell, Stuart J e Peter Norvig: *Artificial intelligence: a modern approach*. Pearson, 3ª edição, 2010. 27, 29

[82] Lucas, Peter J. F. e Linda C. Van Der Gaag: *Principles of Expert Systems*. International Computer Science Series. Addison-Wesley, 1991, ISBN 978-0201416404. 27, 28

[83] Gallacher, Joe: *Practical introduction to expert systems*. Microprocessors and Microsystems, 13(1):47–53, jan 1989, ISSN 0141-9331. http://www.sciencedirect.com/science/article/pii/0141933189900343. 27, 28, 29

[84] Hayes-Roth, Frederick: *Rule-based systems*. Communications of the ACM, 28(9):921–932, sep 1985, ISSN 0001-0782. http://doi.acm.org/10.1145/4284.4286. 27, 28, 30

[85] Buchanan, Bruce G. e Richard O. Duda: *Principles of rule-based expert systems*. Em Yovits, Marshall C. (editor): *Advances In Computers*, volume 22 de *Advances in Computers*, páginas 163 – 216. Elsevier, 1983. http://www.sciencedirect.com/science/article/pii/S0065245808601291. 27, 28

[86] Grossner, Clifford, Alun D. Preece, P. Gokul Chander, T. Radhakrishnan e Ching Y. Suen: *Exploring the structure of rule based systems*. Em *Proceedings of the Eleventh National Conference on Artificial Intelligence*, AAAI'93, páginas 704–709. AAAI Press, 1993, ISBN 0-262-51071-5. http://dl.acm.org/citation.cfm?id=1867270.1867375. 27

[87] Friedman-Hill, Ernest: *Jess in Action: Rule-Based system in Java*. Manning Publications Co., 2003, ISBN 1-930110-89-8. 27

[88] Abraham, Ajith: *Rule-Based Expert Systems*, capítulo 130. American Cancer Society, 2005, ISBN 9780471497394. https://onlinelibrary.wiley.com/doi/abs/10.1002/0471497398.mm422. 28

[89] Gilman, Ekaterina: *Exploring the use of rule-based reasoning in ubiquitous computing applications*. Doctoral training committee of technology and natural sciences of the university of oulu, University of Oulu, 2015. http://urn.fi/urn:isbn:9789526209586. 28

[90] Nalepa, Grzegorz J.: *Knowledge Engineering with Rules*, páginas 27–48. Springer International Publishing, 2018, ISBN 978-3-319-66655-6. https://doi.org/10.1007/978-3-319-66655-6_2. 28, 29

[91] Sasikumar, M, S Ramani, S Muthu Raman, KSR Anjaneyulu e R Chandrasekar: *A practical introduction to rule based expert systems*, 2007. https://www.researchgate.net/profile/Srinivasan_Ramani/publication/265038834_A_Practical_Introduction_to_Rule_Based_Expert_Systems/links/551faa1d0cf2a2d9e1407dce/A-Practical-Introduction-to-Rule-Based-Expert-Systems.pdf. 28

[92] Forgy, Charles L.: *Rete: A fast algorithm for the many pattern/many object pattern match problem*. Artificial Intelligence, 19(1):17–37, sep 1982, ISSN 0004-3702. http://www.sciencedirect.com/science/article/pii/0004370282900200. 29

[93] Batory, Don: *The leaps algorithm*. Relatório Técnico, University of Texas at Austin, Austin, TX, USA, 1994. https://dl.acm.org/citation.cfm?id=899216. 30

[94] Wang, Yu Wang e E. N. Hanson: *A performance comparison of the rete and treat algorithms for testing database rule conditions*. Em *[1992] Eighth International Conference on Data Engineering*, páginas 88–97, Feb 1992. https://ieeexplore.ieee.org/abstract/document/213202/. 30

[95] Miranker, D. P. e B. J. Lofaso: *The organization and performance of a treat-based production system compiler*. IEEE Transactions on Knowledge and Data Engineering, 3(1):3–10, Mar 1991, ISSN 1041-4347. https://ieeexplore.ieee.org/abstract/document/75882/. 30

[96] Sottara, Davide, Paola Mello e Mark Proctor: *A configurable rete-oo engine for reasoning with different types of imperfect information*. IEEE Transactions on Knowledge and Data Engineering, 22(11):1535–1548, nov 2010, ISSN 1041-4347. http://ieeexplore.ieee.org/document/5551128/. 30

[97] Kim, Milhan, Kiseong Lee, Youngmin Kim, Taejin Kim, Yunseong Lee, Sungrae Cho e Chan Gun Lee: *Rete-adh: An improvement to rete for composite context-aware service*. International Journal of Distributed Sensor Networks, 10(4):507160, apr 2014, ISSN 1550-1477. https://doi.org/10.1155/2014/507160. 30

[98] Lee, Rachel e Sang young Cho: *Rete-eca: A rule-based system for device control*. Em *Proceedings of the International Workshop on Artificial Neural Networks and Intelligent Information Processing*, páginas 95–102. SciTePress - Science and and Technology Publications, 2014, ISBN 978-989-758-041-3. http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0005140500950102. 30

[99] Bali, Michal: *Drools JBoss Rules 5.0 Developer's Guide*. Packt Publishing Ltd, 2009, ISBN 978-1-847195-64-7. 32, 33

[100] Browne, Paul: *JBoss Drools business rules*. Packt Publishing Ltd, 2009, ISBN 978-1-847196-06-4. 32

[101] Deitel, Harvey M., Paul J. Deitel, Tem R. Nieto, Ted Lin e Praveen Sadhu: *XML How to Program*. Pearson, 2000, ISBN 978-0130284174. 32

[102] Amador, Lucas: *Drools developer's cookbook*. Packt Publishing Ltd, 2012, ISBN 978-1-84951-196-4. 33

[103] Bali, Michal: *Drools JBoss Rules 5.X Developer's Guide*. Packt Publishing Ltd, 2013, ISBN 978-1-78216-126-4. 33

[104] Doorenbos, Robert B.: *Production Matching for Large Learning Systems*. Computer science department, Carnegie Mellon University, 1995. http://reports-archive.adm.cs.cmu.edu/anon/1995/CMU-CS-95-113.pdf. 33

[105] Acharya, Anurag e Milind Tambe: *Collection oriented match*. Em *Proceedings of the Second International Conference on Information and Knowledge Management*, CIKM '93, páginas 516–526, New York, NY, USA, 1993. ACM, ISBN 0-89791-626-3. http://doi.acm.org/10.1145/170088.170411. 33

[106] Team, JBoss Drools: *Drools documentation - version 7.7.0.final*. https://docs.jboss.org/drools/release/7.7.0.Final/drools-docs/html_single/index.html, 2017. Online; acessado em 19 de junho de 2018. 33, 66

[107] Czarnecki, Krzysztof e Ulrich W. Eisenecker: *Components and generative programming*. ACM SIGSOFT Software Engineering Notes, 24(6):2–19, outubro 1999, ISSN 0163-5948. http://doi.acm.org/10.1145/318774.318779. 34

[108] Gamma, Erich, Richard Helm, Ralph Johnson e John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994, ISBN 978-0201633610. 34

[109] Cechticky, V. e A. Pasetti: *Generative programming for space applications*. Em *DASIA 2003 - Data Systems In Aerospace*, volume 532 de *ESA Special Publication*, páginas 19–30, 2003. http://adsabs.harvard.edu/abs/2003ESASP.532E...3C. 34

[110] Czarnecki, Krzysztof: *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. Department of computer science and automation, Technical University of Ilmenau, 1998. https://bit.ly/2JBWYS6. 34, 35, 37

[111] Arora, Ritu, Purushotham Bangalore e Marjan Mernik: *Developing scientific applications using generative programming*. Em *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, páginas 51–58, 2009, ISBN 978-1-4244-3737-5. https://ieeexplore.ieee.org/document/5069162/. 34

[112] Barth, Barbara, Greg Butler, Krzysztof Czarnecki e Ulrich Eisenecker: *Generative programming*. Em Frohner, Ákos (editor): *Proceedings of the Workshops on Object-Oriented Technology*, ECOOP '01, páginas 135–149, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg, ISBN 978-3-540-47853-9. http://dl.acm.org/citation.cfm?id=646781.705926. 34, 35

[113] Azimi, Reza e Farid Hosseini: *Generative programming: A model driven approach*, 2003. http://www.eecg.toronto.edu/~jacobsen/courses/ece1770/index-03.html. 34, 35

[114] Paska, Marek: *Generative programming with support for formal verification*. Em *2009 IEEE International Symposium on Industrial Embedded Systems*, páginas 58–61. IEEE, jul 2009, ISBN 978-1-4244-4109-9. http://ieeexplore.ieee.org/document/5196194/. 35

[115] Cleaveland, J Craig: *Building application generators*. IEEE Software, 5(4):25–33, 1988, ISSN 0740-7459. https://ieeexplore.ieee.org/document/17799/. 36, 37

[116] Herrington, Jack: *Code Generation in Action*. Manning Publications Co., 2003, ISBN 1-930110-97-9. 36

[117] Stroustrup, Bjarne: *The C++ Programming Language*. Addison-Wesley, 3ª edição, 1997, ISBN 0-201-88954-4. 38

[118] He, Zonggang e Junsheng Zheng: *Design and implementation of student attendance management system based on mvc*. Em *2009 International Conference on Management and Service Science*, páginas 1–4. IEEE, sep 2009, ISBN 978-1-4244-4638-4. http://ieeexplore.ieee.org/document/5304685/. 38

[119] Benato, Gabriele Salgado, Frank José Affonso e Elisa Yumi Nakagawa: *Infrastructure based on template engines for automatic generation of source code for self-adaptive software domain*. Em *Proceedings of the International Conference on Software Engineering and Knowledge Engineering, SEKE*, páginas 30–35, jul 2017, ISBN 1891706411. http://ksiresearchorg.ipage.com/seke/seke17paper/seke17paper{_}147.pdf. 38, 39

[120] Parr, Terence John: *Enforcing strict model-view separation in template engines*. Em *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, páginas 224–233, New York, New York, USA, 2004. ACM Press, ISBN 1-58113-844-X. http://doi.acm.org/10.1145/988672.988703. 38

[121] Arnoldus, Jeroen, Jeanot Bijpost e Mark G.J. van den Brand: *Repleo: A Syntax-safe Template Engine*. Em *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, GPCE '07, páginas 25–32, 2007, ISBN 978-1-59593-855-8. http://doi.acm.org/10.1145/1289971.1289977. 38

[122] Segura, Sergio, Amador Durán, Javier Troya e Antonio Ruiz Cortés: *A template-based approach to describing metamorphic relations*. Em *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*, páginas 3–9, 2017, ISBN 9781538604243. 38

[123] Arnoldus, B J, M G J van den Brand e a Serebrenik: *Less is more: Unparser-completeness of metalanguages for template engines*. Em *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*, GPCE '11, páginas 137–146, 2011, ISBN 978-1-4503-0689-8. http://doi.acm.org/10.1145/2047862.2047887. 38

[124] Radjenovic, Jelena, Branko Milosavljevic e Dusan Surla: *Modelling and implementation of catalogue cards using freemarker*. Program, 43(1):62–76, 2009, ISSN 0033-0337. https://doi.org/10.1108/00330330910934110. 38

[125] Wengner, Miro: *A review of java template engines*. https://dzone.com/articles/template-engines-at-one-spring-boot-and-engines-se, 2016. Online; acessado em 4 de junho de 2018. 39

[126] Parr, Terence John: *Web application internationalization and localization in action*. Em *Proceedings of the 6th International Conference on Web Engineering*, ICWE '06,

páginas 64–70, 2006, ISBN 1-59593-352-2. http://doi.acm.org/10.1145/1145581.1145650. 39

[127] Slonneger, Kenneth e Barry L Kurtz: *Formal Syntax and Semantics of Programming Languages*. Addison-Wesley, 1995, ISBN 0-201-65697-3. 39

[128] Parr, Terence: *Language Implementation Patterns: create your own domain-specific and general programming languages*. Pragmatic Bookshelf, 2010, ISBN 978-1-934356-45-6. 39

[129] Bentley, Jon e David Gries: *Programming pearls*. Commun. ACM, 30(4):284–290, abril 1987, ISSN 0001-0782. http://doi.acm.org/10.1145/32232.315727. 39, 40

[130] Karsai, Gabor, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler e Steven Völkel: *Design guidelines for domain specific languages*. CoRR, abs/1409.2378(October):7, 2014. http://arxiv.org/abs/1409.2378. 39

[131] Parr, Terence: *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013, ISBN 978-1-93435-699-9. 39

[132] Collet, Philippe: *Domain specific languages for managing feature models: Advances and challenges*. Em LNCS, Springer (editor): *6th International Symposium, ISoLA 2014*, volume 8802 de *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, páginas 273–288, Corfu, Greece, 2014. ISBN 9783662452349. https://hal.archives-ouvertes.fr/hal-01345656. 39

[133] Raja, Aruna e Devika Lakshmanan: *Domain specific languages*. International Journal of Computer Applications, 1(21):105–111, feb 2010, ISSN 09758887. http://www.ijcaonline.org/journal/number21/pxc387640.pdf. 39, 41

[134] Neeraj, K. R., P. S. Janardhanan, A. B. Francis e R. Murali: *A domain specific language for business transaction processing*. Em *2017 IEEE International Conference on Signal Processing, Informatics, Communication and Energy Systems (SPICES)*, páginas 1–7, Aug 2017. https://ieeexplore.ieee.org/document/8091270/. 39, 40

[135] Mernik, Marjan, Jan Heering e Anthony M. Sloane: *When and how to develop domain-specific languages*. ACM Computing Surveys, 37(4):316–344, dec 2005, ISSN 0360-0300. http://doi.acm.org/10.1145/1118890.1118892. 40

[136] Sánchez Cuadrado, Jesús, Javier Cánovas e Jesus Garcia Molina: *Comparison between internal and external dsls via rubytl and gra2mol*. Em Mernik, Marjan (editor): *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. IGI Global, setembro 2012, ISBN 9781466620926. https://hal.inria.fr/hal-00752687. 40

[137] Bussenot, Robin, Hervé Leblanc e Christian Percebois: *A domain specific test language for systems integration*. Em *Proceedings of the Scientific Workshop Proceedings of XP2016*, XP '16 Workshops, páginas 16:1–16:10, New York, NY, USA, 2016. ACM, ISBN 978-1-4503-4134-9. http://doi.acm.org/10.1145/2962695.2962711. 40, 43

[138] Kurtev, Ivan, Jean Bézivin, Frédéric Jouault e Patrick Valduriez: *Model-based dsl frameworks*. Em *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, páginas 602–616, New York, NY, USA, 2006. ACM, ISBN 1-59593-491-X. http://doi.acm.org/10.1145/1176617.1176632. 40

[139] Deursen, Arie van, Paul Klint e Joost Visser: *Domain-specific languages: An annotated bibliography*. SIGPLAN Not., 35(6):26–36, junho 2000, ISSN 0362-1340. http://doi.acm.org/10.1145/352029.352035. 40, 41

[140] Zdun, Uwe: *A dsl toolkit for deferring architectural decisions in dsl-based software design*. Information and Software Technology, 52(7):733–748, jul 2010, ISSN 0950-5849. http://www.sciencedirect.com/science/article/pii/S0950584910000443. 40

[141] Cunningham, H. Conrad: *A little language for surveys: Constructing an internal dsl in ruby*. Em *Proceedings of the 46th Annual Southeast Regional Conference on XX*, ACM-SE 46, páginas 282–287, New York, New York, USA, 2008. ACM, ISBN 978-1-60558-105-7. http://doi.acm.org/10.1145/1593105.1593181. 40

[142] Ghosh, Debasish: *DSLs in action*. Manning Publications Co., 2011, ISBN 9781935182450. 40

[143] Fowler, Martin e Rebecca Parsons: *DSL: Linguagens Específicas de Domínio*. Bookman, 2013, ISBN 978-85-407-0212-7. 40

[144] Pfeiffer, Michael e Josef Pichler: *A comparison of tool support for textual domain-specific languages*. Em *Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling*, DSM'08, páginas 1–7, 2008. http://www.dsmforum.org/events/dsm08/papers.html. 40

[145] Veldthuis, M.: *Quby: A domain-specific language for non-programmers*, 2012. http://fse.studenttheses.ub.rug.nl/10483/. 40

[146] Oliveira, Nuno, Maria João Pereira, Pedro Henriques e Daniela Cruz: *Domain specific languages: a theoretical survey*. Em *INFORUM'09 Simpósio de Informática, Lisboa*, ESTiG, 2009. http://hdl.handle.net/10198/1192. 40, 41

[147] Heering, J., M. Mernik e A. M. Sloane: *Domain-specific languages*. Em *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*, HICSS'03, páginas 323–323, Jan 2003, ISBN 0-7695-1874-5. https://ieeexplore.ieee.org/document/1174888/. 41

[148] Damyanov, I. e M. Sukalinska: *Domain specific languages in practice*. International Journal of Computer Applications, 115(2):42–45, apr 2015. 41

[149] Fowler, Martin: *Language workbenches: The killer-app for domain specific languages?* https://www.martinfowler.com/articles/languageWorkbench.html, 2005. Online; acessado em 10 de junho de 2018. 42

[150] Fowler, Martin: *Languageworkbench*. https://martinfowler.com/bliki/LanguageWorkbench.html, 2008. Online; acessado em 10 de junho de 2018. 42

[151] Erdweg, Sebastian, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth e Jimi van der Woning: *The state of the art in language workbenches*. Em *Software Language Engineering*, páginas 197–217, Cham, 2013. Springer International Publishing, ISBN 978-3-319-02654-1. https://link.springer.com/chapter/10.1007/978-3-319-02654-1_11. 42, 43

[152] Karol, Sven, Tobias Nett, Jerónimo Castrillón e Ivo F. Sbalzarini: *A domain-specific language and editor for parallel particle methods*. CoRR, abs/1704.00032, 2017. http://arxiv.org/abs/1704.00032. 42

[153] Schmitt, Christian, Sebastian Kuckuk, Harald Köstler, Frank Hannig e Jurgen Teich: *An evaluation of domain-specific language technologies for code generation*. Em *2014 14th International Conference on Computational Science and Its Applications*, ICCSA 2014, páginas 18–26, 2014. https://ieeexplore.ieee.org/document/6976658/. 42, 43

[154] Erdweg, Sebastian, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth e Jimi van der Woning: *Evaluating and comparing language workbenches*. Computer Languages, Systems & Structures, 44(PA):24–47, dec 2015, ISSN 1477-8424. http://dx.doi.org/10.1016/j.cl.2015.08.007. 42, 43

[155] Bettini, Lorenzo: *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2ª edição, 2016, ISBN 978-1-78646-496-5. 43

[156] Gallardo, David, Ed Burnette e Robert McGovern: *Eclipse in Action: A Guide for Web Developers*. Manning Publications Co., 2003. 43

[157] Kulkarni, Ram: *Java EE Development with Eclipse*. Packt Publishing Ltd, 2ª edição, 2015, ISBN 978-1-78528-534-9. 43

[158] Prasanna, Dhanji R.: *Dependency Injection*. Manning Publications Co., 2009, ISBN 978-1-933988-55-9. 43

[159] Blewitt, Alex: *Eclipse 4 Plug-in Development by Example Beginner's Guide*. Packt Publishing Ltd, 2013, ISBN 978-1-78216-032-8. 43

[160] Eysholdt, Moritz e Heiko Behrens: *Xtext: Implement your language faster than the quick and dirty way*. Em *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications*

Companion, OOPSLA '10, páginas 307–309, New York, NY, USA, 2010. ACM, ISBN 978-1-4503-0240-1. http://doi.acm.org/10.1145/1869542.1869625. 43

[161] Santiago, Marcos Rogério: *Ensaio do swebok - software engineering body of knowledge*, 2011. http://www.cin.ufpe.br/~bfs3/UFPE/Engenharia_Software/77017069-SWEBOK-traduzido.pdf. 45

[162] Runeson, Per e Peter Isacsson: *Software quality assurance-concepts and misconceptions*. Em *Proceedings. 24th EUROMICRO Conference (Cat. No.98EX204)*, volume 2, páginas 853–859 vol.2, Aug 1998. https://ieeexplore.ieee.org/document/708112/. 45

[163] Yatoh, Kohsuke, Kazunori Sakamoto, Fuyuki Ishikawa e Shinichi Honiden: *ArbitCheck: A Highly Automated Property-Based Testing Tool for Java*. Em *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, páginas 405–412. IEEE, mar 2014, ISBN 978-1-4799-5790-3. http://ieeexplore.ieee.org/document/6825695/. 45, 48

[164] Luo, Lu: *Software testing techniques*. Relatório Técnico, Software Engineering Institute, Carnegie Mellon University, 2001. https://www.cs.cmu.edu/~luluo/Courses/17939Report.pdf. 46

[165] Xu, Dianxiang, Weifeng Xu, Michael Kent, Lijo Thomas e Linzhang Wang: *An automated test generation technique for software quality assurance*. IEEE Transactions on Reliability, 64(1):247–268, March 2015, ISSN 0018-9529. 46, 47

[166] Dijkstra, Edsger W.: *Structured programming*. capítulo Chapter I: Notes on Structured Programming, páginas 1–82. Academic Press Ltd., London, UK, UK, 1972, ISBN 0-12-200550-3. http://dl.acm.org/citation.cfm?id=1243380.1243381. 46

[167] Yenigün, Hüsnü, Cemal Yilmaz e Andreas Ulrich: *Advances in test generation for testing software and systems*. International Journal on Software Tools for Technology Transfer, 18(3):245–249, Jun 2016, ISSN 1433-2787. https://doi.org/10.1007/s10009-015-0404-z. 46

[168] Elberzhager, Frank, Alla Rosbach, JüRgen MüNch e Robert Eschbach: *Reducing test effort: A systematic mapping study on existing approaches*. Information and Software Technology, 54(10):1092–1106, outubro 2012, ISSN 0950-5849. http://dx.doi.org/10.1016/j.infsof.2012.04.007. 46, 47

[169] Mahadik, Pranali, Debnath Bhattacharyya e Hye jin Kim: *Techniques for automated test cases generation: A review*. International Journal of Software Engineering and Its Applications, 10:13–20, dezembro 2016, ISSN 1738-9984. http://dx.doi.org/10.14257/ijseia.2016.10.12.02. 46, 47

[170] Ali, Shaukat, Lionel C. Briand, Hadi Hemmati e Rajwinder Kaur Panesar-Walawege: *A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation*. IEEE Transactions on Software Engineering, 36(6):742–762, nov 2010, ISSN 0098-5589. http://ieeexplore.ieee.org/document/5210118/. 46, 47

[171] Khurana, Namita e Rajender Singh Chhillar: *A comparison of evolutionary techniques for test case generation and optimization*. Journal of Theoretical and Applied Information Technology, 95(19):5285–5295, 2017, ISSN 1817-3195. 46, 47

[172] Harman, Mark e Phil McMinn: *A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search*. IEEE Transactions on Software Engineering, 36(2):226–247, mar 2010, ISSN 0098-5589. http://ieeexplore.ieee.org/document/5342440/. 46, 47

[173] Hooda, Itti e Rajender Chhillar: *A review: Study of test case generation techniques*. International Journal of Computer Applications, 107(16):33–37, December 2014. https://www.ijcaonline.org/archives/volume107/number16/18839-0375. 47

[174] Elghondakly, Roaa, Sherin Moussa e Nagwa Badr: *A comprehensive study for software testing and test cases generation paradigms*. Em *Proceedings of the International Conference on Internet of Things and Cloud Computing*, ICC '16, páginas 50:1–50:7, New York, NY, USA, 2016. ACM, ISBN 978-1-4503-4063-2. http://doi.acm.org/10.1145/2896387.2896435. 47

[175] Sahoo, Rajesh Kumar, Deeptimanta Ojha, Durga Prasad Mohapatra e Manas Ranjan Patra: *Automated Test Case Generation and Optimization : A Comparative Review*. International Journal of Computer Science and Information Technology, 8(5):19–32, oct 2016, ISSN 0975-4660. http://aircconline.com/ijcsit/V8N5/8516ijcsit02.pdf. 47

[176] Lampropoulos, Leonidas, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C. Pierce e Li yao Xia: *Beginner's luck: a language for property-based generators*. Em *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages - POPL 2017*, páginas 114–129, New York, New York, USA, 2017. ACM Press, ISBN 9781450346603. http://dl.acm.org/citation.cfm?doid=3009837.3009868. 47

[177] Loscher, Andreas e Konstantinos Sagonas: *Automating Targeted Property-Based Testing*. Em *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, páginas 70–80. IEEE, apr 2018, ISBN 978-1-5386-5012-7. https://ieeexplore.ieee.org/document/8367037/. 47

[178] Earle, Clara Benac, Lars Ake Fredlund, Julio Marino e Thomas Arts: *Teaching Students Property-Based Testing*. Em *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, páginas 437–442. IEEE, aug 2014, ISBN 978-1-4799-5795-8. http://ieeexplore.ieee.org/document/6928850/. 48

[179] Castro, Laura M.: *Advanced management of data integrity: property-based testing for business rules*. Journal of Intelligent Information Systems, 44(3):355–380, jun 2015, ISSN 0925-9902. http://link.springer.com/10.1007/s10844-014-0335-2. 48

[180] Claessen, Koen e John Hughes: *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*. Em *Proceedings of the fifth ACM SIGPLAN International*

Conference on Functional Programming - ICFP '00, volume 35, páginas 268–279, New York, New York, USA, 2000. ACM Press, ISBN 1581132026. http://portal.acm.org/citation.cfm?doid=351240.351266. 48

[181] Chepurnoy, Alexander e Mayank Rathee: *Checking laws of the blockchain with property-based testing*. Em *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, volume 2018-Janua, páginas 40–47. IEEE, mar 2018, ISBN 978-1-5386-5986-1. http://ieeexplore.ieee.org/document/8327570/. 48

[182] Castro, Laura M., Pablo Lamela e Simon Thompson: *Making property-based testing easier to read for humans*. Computing and Informatics, 35(4):890–913, 2016, ISSN 1335-9150. http://www.cai.sk/ojs/index.php/cai/article/viewArticle/3381. 49

[183] Wynne, Matt, Aslak Hellesoy e Steve Tooke: *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2017. 49

[184] Brasil, Ministério da Defesa: *Instruções reguladoras do processo de concepção de quadro de organização - eb20-ir-10.004*, 2015. http://bdex.eb.mil.br/jspui/bitstream/123456789/218/1/EB20-IR-10.004.pdf. 50, 71

[185] Beck, Kent, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland e Dave Thomas: *Manifesto for agile software development*, 2001. http://www.agilemanifesto.org/. 50

[186] Deitel, Paul e Harvey Deitel: *Java how to program*. Prentice Hall, 9ª edição, 2012, ISBN 978-0-13-257566-9. 51

[187] Urma, Raoul Gabriel, Mario Fusco e Alan Mycroft: *Java 8 in Action: Lambdas, streams, and functional-style programming*. Manning Publications Co., 2015, ISBN 9781617291999. 51

[188] Spell, Terrill Brett: *Pro Java 8 Programming*. Apress, 2015, ISBN 978-1-4842-0641-6. 51

[189] Goncalves, Antonio: *Beginning Java EE 7*. Apress, 2013, ISBN 978-1-4302-4627-5. 51

[190] Gupta, Arun: *Java EE 7 Essentials*. O'Reilly, 2013, ISBN 978-1-449-37017-6. 51

[191] Juneau, Josh: *Introducing Java EE 7*. Apress, 2013, ISBN 978-1-4302-5849-0. 51

[192] Çalışkan, Mert e Oleg Varaksin: *PrimeFaces Cookbook*. Packt Publishing Ltd, 2ª edição, 2015, ISBN 978-1-78439-342-7. 51

[193] Jonna, Sudheer: *Learning PrimeFaces Extensions Development*. Packt Publishing Ltd, 2014, ISBN 978-1-78398-324-7. 51

[194] Juneau, Josh: *JavaServer Faces: Introduction by Example*. Apress, 2014, ISBN 978-1-4842-0838-0. 51

[195] Salas-Zárate, María del Pilar, Giner Alor-Hernández, Rafael Valencia-García, Lisbeth Rodríguez-Mazahua, Alejandro Rodríguez-González e José Luis López Cuadrado: *Analyzing best practices on web development frameworks: The lift approach*. Science of Computer Programming, 102:1–19, 2015, ISSN 0167-6423. http://www.sciencedirect.com/science/article/pii/S0167642314005735. 51

[196] deHaan, Lex, Tim Gorman, Inger Jorgensen e Melanie Caffrey: *Beginning Oracle SQL*. Apress, 3ª edição, 2014, ISBN 978-1430265566. 51

[197] Hasler, Tony: *Expert Oracle SQL*. Apress, 2014, ISBN 978-1430259770. 51

[198] Ćmil, Michal, Michal Matloka e Francesco Marchioni: *Java EE 7 Development with WildFly*. Packt Publishing, 2014, ISBN 978-1782171980. 51

[199] Vohra, Deepak: *Advanced Java EE Development with WildFly*. Packt Publishing, 2015, ISBN 978-1783288908. 51

[200] Stancapiano, Luca: *Mastering Java EE Development with WildFly*. Packt Publishing, 2017, ISBN 978-1787287174. 51

[201] Anardu, Lorenzo, Roberto Baldi, Umberto Antonio Cicero, Riccardo Giomi e Giacomo Veneri: *Maven Build Customization*. Packt Publishing, 2014, ISBN 978-1783987221. 51

[202] Siriwardena, Prabath: *Mastering Apache Maven 3*. Packt Publishing, 2014, ISBN 978-1783983865. 51

[203] McQuaid, Mike: *Git in Practice*. Manning Publications, 2014, ISBN 978-1617291975. 51

[204] Santacroce, Ferdinando: *Git Essentials*. Packt Publishing, 2017, ISBN 978-1787120723. 51

[205] Larman, Craig: *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, 3ª edição, 2004, ISBN 978-0131489066. 51

[206] Lima, Flávio da Costa: *O processo decisório para obtenção de materiais de emprego militar no exército brasileiro*, 2007. http://bibliotecadigital.fgv.br/dspace/handle/10438/3514. 65

[207] Brasil, Exército Brasileiro: *Armas, quadros e serviços*. http://www.eb.mil.br/armas-quadros-e-servicos, 2018. Online; acessado em 28 de setembro de 2018. 81

[208] Crawley, Michael J: *The R book*. John Wiley & Sons, 2013. 92