UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

CENTRO TECNOLÓGICO

PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Camila Zacché de Aguiar

Interoperabilidade Semântica entre Códigos-Fonte baseada em Ontologia

Vitória, ES

2021

Camila Zacché de Aguiar

Interoperabilidade Semântica entre Códigos-Fonte baseada em Ontologia

Tese de Doutorado submetida ao Programa de Pós-Graduação em Informática da Universidade Federal do Espírito Santo, como requisito para obtenção do Grau de Doutor em Ciência da Computação.

Universidade Federal do Espírito Santo – UFES

Centro Tecnológico

Programa de Pós-Graduação em Informática

Orientador: Prof. Dr. Vítor E. Silva Souza

Vitória, ES

2021

Ficha catalográfica disponibilizada pelo Sistema Integrado de Bibliotecas - SIBI/UFES e elaborada pelo autor

---

Zacché de Aguiar, Camila, 1987-
Z13i Interoperabilidade Semântica entre Códigos-Fonte baseada em Ontologia / Camila Zacché de Aguiar. - 2021.
191 f. : il.

Orientador: Vítor Estêvão Silva Souza.
Tese (Doutorado em Informática) - Universidade Federal do Espírito Santo, Centro Tecnológico.

1. Ontologia. 2. Linguagem de programação (Computadores). 3. Linguagem de programação (Computadores) - Semântica. I.. I. Estêvão Silva Souza, Vítor. II. Universidade Federal do Espírito Santo. Centro Tecnológico. III. Título.

CDU: 004

INTEROPERABILIDADE SEMÂNTICA ENTRE CÓDIGOS FONTE BASEADA EM ONTOLOGIA

Camila Zacché de Aguiar

Tese submetida ao Programa de Pós-Graduação em Informática da Universidade Federal do Espírito Santo como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Aprovada em 24 de novembro de 2021:

Prof. Dr. Vítor Estêvão Silva Souza
Orientador

Prof. Dr. Giancarlo Guizzardi
Membro Interno

Profª. Drª. Monalessa Perini Barcellos
Membro Externo

Profª. Drª. Fernanda Araujo Baião
Membro Externo

Prof. Dr. Rogério Eduardo Garcia
Membro Externo

UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
Vitória-ES, 24 de novembro de 2021.

UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

PROTOCOLO DE ASSINATURA

O documento acima foi assinado digitalmente com senha eletrônica através do Protocolo Web, conforme Portaria UFES nº 1.269 de 30/08/2018, por
VITOR ESTEVAO SILVA SOUZA - SIAPE 2525114
Departamento de Informática - DI/CT
Em 24/11/2021 às 19:47

Para verificar as assinaturas e visualizar o documento original acesse o link:
https://api.lepisma.ufes.br/arquivos-assinados/315348?tipoArquivo=O

UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

PROTOCOLO DE ASSINATURA

O documento acima foi assinado digitalmente com senha eletrônica através do Protocolo Web, conforme Portaria UFES nº 1.269 de 30/08/2018, por
GIANCARLO GUIZZARDI - SIAPE 1485956
Departamento de Informática - DI/CT
Em 08/12/2021 às 12:01

Para verificar as assinaturas e visualizar o documento original acesse o link:
https://api.lepisma.ufes.br/arquivos-assinados/327038?tipoArquivo=O

UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

PROTOCOLO DE ASSINATURA

O documento acima foi assinado digitalmente com senha eletrônica através do Protocolo Web, conforme Portaria UFES nº 1.269 de 30/08/2018, por
MONALESSA PERINI BARCELLOS - SIAPE 1490891
Departamento de Informática - DI/CT
Em 08/12/2021 às 12:18

Para verificar as assinaturas e visualizar o documento original acesse o link:
https://api.lepisma.ufes.br/arquivos-assinados/327066?tipoArquivo=O

Resumo

O código-fonte é uma sequência bem formada de instruções de computador expressas em uma linguagem de programação, composta por um conjunto de símbolos organizados com suas respectivas sintaxes e semânticas. As diferentes representações de código-fonte em linguagens de programação criam um contexto de heterogeneidade, assim como o uso em conjunto de múltiplas linguagens de programação. Esse cenário impede a troca direta de informações entre códigos-fonte de diferentes linguagens de programação, requerendo conhecimento especializado das respectivas linguagens e diversidade de ferramentas e práticas. Nesse sentido, como forma de mitigar a heterogeneidade entre linguagens de programação, aplicamos a interoperabilidade semântica para garantir que as informações compartilhadas tenham seus significados compreendidos e operacionalizados por códigos-fonte das diferentes linguagens de programação. Para isso, adotamos ontologias para garantir interpretações uniformes que compartilham um compromisso consistente comum sobre o domínio de código-fonte. Além de atuar como interlíngua entre diferentes códigos-fonte, ontologias são amplamente aceitas na literatura como ferramentas para prover semântica e interoperabilidade entre entidades com naturezas distintas.

Para aplicar ontologias na interoperabilidade de códigos-fonte, esta pesquisa apresenta uma rede de ontologias de código-fonte denominada SCON — *Source Code Ontology Network* e um método para interoperabilidade de código-fonte baseado em ontologia denominado OSCIN — *Ontology-Based Source Code Interoperability*. Enquanto SCON representa semanticamente os conceitos comuns e consensuais sobre o domínio do código-fonte, independentemente da linguagem de programação, OSCIN visa aplicar esta representação para diferentes propósitos de uma forma unificada. O método se baseia no subdomínio de código-fonte que será representado, na linguagem de programação que é capaz de manipular e no propósito de aplicação que será aplicado.

Objetivando fornecer um conjunto de soluções para apoiar a operacionalização do método OSCIN em diferentes subdomínios de código-fonte, linguagens de programação e propósitos de aplicação, esta pesquisa apresenta o framework OSCINF — *Ontology-based Source Code Interoperability Framework*, que gera os artefatos esperados pelo método OSCIN e define o método SABiOS — *Systematic Approach for Building Ontologies with Scrum* para a construção de ontologias bem fundamentadas. Finalmente, esta pesquisa avalia a interoperabilidade de código-fonte aplicando o método OSCIN para a detecção de *smells*, métricas de software e migração de código em códigos-fonte de diferentes linguagens.

Palavras-chaves: Interoperabilidade Semântica. Interoperabilidade Semântica em Código-Fonte. Ontologia. Engenharia de Ontologia.

Abstract

The source code is a well-formed sequence of computer instructions expressed in a programming language, composed of a set of symbols organized with their respective syntax and semantics. The different representations of source code in programming languages create a heterogeneous context, as does the use of multiple programming languages in a single source code. This scenario prevents the direct exchange of information between source codes of different programming languages, requiring specialized knowledge of their languages and diversity of tools and practices. In this sense, as a way to mitigate heterogeneity between programming languages, we apply semantic interoperability to ensure that shared information have their meanings understood and operationalized by code written in different source programming languages. To do this, we adopt ontologies to ensure uniform interpretations that share a consistent common commitment about the source code domain. In addition to acting as an interlanguage between different source codes, ontologies are widely accepted in the literature as tools to provide semantics and interoperability between entities with different natures.

To apply ontologies to source code interoperability, this research presents a source code ontology network called SCON — *Source Code Ontology Network* and a method for source code interoperability based on ontology called OSCIN — *Ontology-Based Source Code Interoperability*. While SCON semantically represents common and consensual concepts about the domain of source code, regardless of the programming language, OSCIN aims to apply this representation for different purposes in a unified way. The method is based on the source code subdomain that will be represented, the programming language that it is capable of handling, and the application purpose that will be applied.

In order to provide a set of solutions to support the application of the OSCIN method in different source code subdomains, programming languages and application purposes, this research presents the OSCINF framework — *Ontology-based Source Code Interoperability Framework*, which generates the artifacts expected by the OSCIN method and defines the SABiOS method — *Systematic Approach for Building Ontologies with Scrum* for the construction of well-founded ontologies. Finally, this research evaluates source code interoperability by applying the OSCIN method to detect smells, software metrics and code migration in source codes from different programming languages.

Keywords: Semantic interoperability. Semantic Interoperability in Source Code. Ontology. Ontology Engineering.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

AST	Abstract Syntax Tree
ANTLR	ANother Tool for Language Recognition
DBS-O	Database System Ontology
OO	Orientação a Objetos
OOC-O	Object-Oriented Code Ontology
OSCIN	Ontology-based Source Code Interoperability
OSCINF	Ontology-based Source Code Interoperability Framework
ORM-O	Object/Relational Mapping Ontology
OWL	Ontology Web Language
RDF	Resource Description Framework
RDBS-O	Relational Database System Ontology
SABiO	Systematic Approach for Building Ontologies
SABiOS	Systematic Approach for Building Ontologies, Supplemented
SCON	Source Code Ontology Network
SEON	Software Engineering Ontology Network
SPARQL	SPARQL Protocol and RDF Query Language
SPO	Software Process Ontology
SwO	Software Ontology
UFO	Unified Foundational Ontology

Sumário

1 Introdução

Este capítulo apresenta uma visão geral desta tese, descrevendo o contexto e a motivação, as hipóteses de pesquisa, os objetivos, os aspectos metodológicos e, por fim, a estrutura dos capítulos.

1.1 Contexto e Motivação

O código-fonte é uma sequência bem formada de instruções de computador e definições de dados expressas em linguagem de programação (WANG et al., 2014), definida por uma gramática formal, composta por um conjunto de símbolos organizados com suas respectivas sintaxes e semânticas. Enquanto a sintaxe corresponde aos símbolos da linguagem e as regras para arranjar esses símbolos, a semântica estabelece o significado dos símbolos de acordo com uma interpretação e contexto (SOMMERVILLE, 2011). A linguagem de programação adotada no código-fonte pode não apenas influenciar na eficiência do desenvolvimento, mas também interferir na aplicação adequada ao propósito do código-fonte (LU et al., 2020).

Um grande número de linguagens de programação tem surgido nos últimos anos devido à expansão do uso de computadores e da Internet, tratando de questões específicas ou gerais de programação que visam melhorar o relacionamento com outras linguagens, especificar *frameworks* de linguagens ou abordar novos paradigmas. A lista da Wikipedia[1] apresenta aproximadamente 700 linguagens que cobrem linguagens procedurais (ex.: Pascal e C), orientadas a objetos (ex.: Java e C ++), funcionais (ex.: Haskell e Lisp) e outras. Embora esse número de linguagens esteja aumentando, faltam estudos em larga escala e abrangentes que examinem, dentre outras questões, a interoperabilidade das várias linguagens de programação (BISSYANDÉ, 2013).

A interoperabilidade é um conceito multidimensional, abrangendo vários domínios de aplicação e terminologias. Dentre as terminologias utilizadas,[2] nesta pesquisa definimos interoperabilidade como a **capacidade de trocar informações ou usá-las em um contexto heterogêneo**. Diferente das interoperabilidades discutidas em outras pesquisas (e.g., 64 tipos catalogados por Ford et al. (2007)), tais como, interoperabilidade de sistemas, aplicação e sistema de sistemas, definimos a interoperabilidade de código-fonte, aquela **capacidade dos códigos-fonte de trocar informações ou usá-las em um contexto heterogêneo de diferentes linguagens de programação**. Baseado no conceito de heterogeneidade (SHETH, 1999), definimos dois tipos diferentes de heterogeneidade em

[1] <https://en.wikipedia.org/wiki/List_of_programming_languages> (Julho 2021)
[2] <https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:24765:ed-2:v1:en>

código-fonte: heterogeneidade sintática, observada quando sintaxes diferentes das linguagens são atribuídas a conceitos correspondentes, e heterogeneidade semântica, observada quando conceitos correspondentes possuem diferentes interpretações.

Mesmo que algumas linguagens de programação adotem a mesma sintaxe, ou seja, os mesmos termos para se referir às entidades de programação, elas normalmente associam diferentes significados a esses termos, ou seja, usam semânticas diferentes (FARINELLI; MELO; ALMEIDA, 2013). Assim, as diferentes **representações** de código-fonte em linguagens de programação criam um contexto de heterogeneidade. Além disso, o uso de múltiplas linguagens de programação em um único código-fonte (programação poliglota) (FJELDBERG, 2008) traz consigo uma ampla escolha de termos de diferentes linguagens, bibliotecas, *frameworks* de linguagens e tecnologias (NIEPHAUS; FELGENTRE; HIRSCHFELD, 2019). A maioria dos sistemas são multilíngues por natureza e essa tendência está aumentando dia a dia (MUSHTAQ; RASOOL; SHEHZAD, 2017).

Dado que para atingir a interoperabilidade é necessário eliminar a heterogeneidade, identificamos que as heterogeneidades sintática e semântica introduzem obstáculos à interoperabilidade semântica entre os códigos-fonte, uma vez que a troca direta de informações entre o código-fonte das diferentes linguagens de programação requer conhecimento especializado das respectivas linguagens e diversidade de ferramentas e práticas (WAMPLER; TONY, 2010). Portanto, compreender a interação de várias linguagens por meio de artefatos sem integração e suporte de ferramenta é difícil e desafiador (TICHELAAR et al., 2000a).

Nesse sentido, como forma de mitigar obstáculos à interoperabilidade de código-fonte, observamos iniciativas voltadas à padronização de linguagens de programação com a ISO/IEC JTC 1/SC 22,[3] que promove a cooperação internacional em questões relacionadas a linguagens de programação; definição de metamodelos com a ISO/IEC 19506, que é uma especificaçao do *Object Management Group* (OMG) que define um *Knowledge Discovery Meta-model* (KDM) para representar ativos de software existentes, suas associações e ambientes operacionais; construção de ambientes de desenvolvimento multilíngue para prover alto nível de suporte a diferentes linguagens (SIEMUND; TOVESSON, 2018; RASK et al., 2021; NIEPHAUS et al., 2018; PFEIFFER; WASOWSKI, 2012); e análise de código multilíngue (STREIN, 2006; SCHINK et al., 2011; MUSHTAQ; RASOOL; SHEHZAD, 2017) para extração e análise uniforme de diferentes linguages de programação. No entanto, não observamos iniciativas aprofundadas na interoperabilidade semântica de código-fonte para garantir que as informações compartilhadas tenham seus significados compreendidos e operacionalizados por códigos-fonte de diferentes linguagens de programação.

Para isso, ontologias podem ser adotadas a fim de garantir interpretações uniformes que compartilham um compromisso consistente comum sobre o domínio de código-fonte. O

[3] <https://www.iso.org/committee/45202.html>

uso da ontologia reduz o problema de conhecer a estrutura e conceitos de diversas linguagens de programação para o problema de conhecer o conteúdo de ontologias específicas do domínio de código-fonte. Para além de atuar como interlíngua entre diferentes códigos-fonte, ontologias são amplamente aceitas na literatura como ferramentas para prover semântica e interoperabilidade entre entidades com naturezas distintas (GUARINO; OBERLE; STAAB, 2009; GUIZZARDI, 2007). Essas ontologias podem ser organizadas em uma rede de ontologias, a fim de expressar explicitamente como as ontologias estão relacionadas e aproveitar a integração, evolução e reutilização de ontologias (SANTOS, 2017).

Nesse sentido, a partir da representação semântica do código-fonte, na forma de ontologia, é possível mitigar a heterogeneidade das diferentes linguagens de programação e possibilitar a interoperabilidade semântica entre os códigos-fonte, contribuindo para novas soluções de suporte a código multilíngue.

1.2 Hipótese de Pesquisa

Considerando que (i) linguagens de programação não adotam uma conceituação semântica comum; (ii) existentes modelos/padrões de código-fonte apresentam visões particulares e limitadas sobre o domínio, não representando uma conceituação semântica consensual do domínio de código-fonte; (iii) iniciativas de interoperabilidade de código-fonte não se aprofundam na semântica do domínio; (iv) ontologias têm sido reconhecidas como instrumentos para tratar questões semânticas e de interoperabilidade; (v) uma rede de ontologias possibilita a organização, integração, evolução e reúso de ontologias de um dado domínio, definimos como hipótese de pesquisa que:

> **Uma representação semântica do domínio de código-fonte baseada em ontologia (organizada em uma rede de ontologias) fornece uma conceituação consensual, compartilhada e abrangente do código-fonte, que pode mitigar a heterogeneidade das diferentes linguagens de programação e possibilitar a interoperabilidade semântica entre os códigos-fonte.**

1.3 Objetivos

O objetivo geral dessa pesquisa é definir uma fundamentação teórica de código-fonte baseada em uma representação ontológica e aplicada à interoperabilidade de código-fonte de diferentes linguagens de programação.

A partir desse objetivo geral, os seguintes objetivos específicos (OE) são definidos:

- **OE01.** Investigar o estado da arte no domínio do código-fonte em diferentes linguagens de programação, de forma a conhecer e definir os conceitos decorrentes do domínio de forma consensual, compartilhada e abrangente;
- **OE02.** Representar a fundamentação teórica de código-fonte em uma rede de ontologias;
- **OE03.** Estabelecer um método baseado na rede de ontologias de código-fonte para interoperar códigos-fonte de diferentes linguagens de programação;
- **OE04.** Avaliar se o método possibilita a troca de informação entre códigos-fonte de diferentes linguagens de programação.

1.4 Método de Pesquisa

Para assegurar o rigor científico desta pesquisa, adotamos o método de pesquisa *Design Science Research*, que envolve o desenvolvimento de construções inovadoras, destinadas a resolver problemas enfrentados no mundo real, e simultaneamente faz uma espécie de contribuição científica prescritiva (DRESCH; LACERDA; ANTUNES, 2015). *Design Science Research* é baseado nos conceitos dos paradigmas metodológicos *Behavioral Science* e *Design Science* aplicados sobre o **Ambiente**, cujas necessidades e problemas são observados através das pessoas, organizações e suas tecnologias, e sobre a **Base de Conhecimento**, cujas teorias ou artefatos são reconhecidos pela comunidade acadêmica.

O método *Design Science Research* se relaciona com o *Ambiente* por meio do fator **Relevância**, fator definitivo para que os resultados das investigações e o conhecimento gerado sejam usados para resolver problemas práticos (DRESCH; LACERDA; ANTUNES, 2015). Nesse contexto, *Behavioral Science* endereça a pesquisa por meio do desenvolvimento e justificação de teorias e *Design Science* por meio da construção e avaliação de artefatos (HEVNER et al., 2004).

Por outro lado, o método *Design Science Research* se relaciona com a *Base de Conhecimento* por meio do fator **Rigor**, fator essencial para a pesquisa ser considerada válida e confiável. Nesse contexto, *Behavioral Science* tipicamente aplica metodologias relacionadas à coleção de dados e análise empírica e *Design Science* aplica metodologias computacionais e matemáticas para avaliar a qualidade e efetividade dos artefatos (HEVNER et al., 2004).

Assim, o método *Design Science Research* é validado pela forma que as necessidades do negócio são aplicadas ao *Ambiente* e às pesquisas e práticas adicionam conteúdo para a *Base de Conhecimento*. Dado que o conhecimento e a compreensão de um domínio do problema e sua solução são alcançados graças à construção e aplicação de um artefato projetado (HEVNER et al., 2004), a aplicação desse método pode potencialmente reduzir a lacuna existente entre a teoria e a prática.

A Figura 1 apresenta como o método é aplicado nesta pesquisa, cujos ciclos de pesquisa serão explanados a seguir. O **Ambiente** é retratado nas pessoas que atuam como desenvolvedor de software e de linguagem de programação, que adotam processo de desenvolvimento de software e utilizam linguagem de programação. A **Base de Conhecimento** é retratada nos fundamentos por metamodelos e especificações de linguagens de programação, ontologias de código-fonte e artefato de software, ontologia de fundamentação e abordagens de interoperabilidade de código-fonte; nas metodologias pelo método de engenharia de ontologias, pela linguagem de modelagem e codificação de ontologias e pelo método de pesquisa *design science*.

1.4.1 Ciclo de Relevância

O Ciclo de Relevância inicia a pesquisa identificando e representando oportunidades e problemas do ambiente como um contexto de aplicação, a fim de prover requisitos para a pesquisa e critérios de aceitação para os resultados (HEVNER, 2007). Nesta pesquisa, a Relevância é identificada pelo **problema** de interoperabilidade semântica de código-fonte, diante das informações encontradas na literatura sobre o domínio (apresentadas no Capítulo 2) e na prática com experiências de desenvolvedores sobre o suporte ao desenvolvimento de código-fonte. Este assunto está em voga e tem sido abordado na literatura (OUKSEL; SHETH, 1999; VRANDEC et al., 2018) no sentido de interoperar códigos-fontes, porém não semanticamente. A limitação das pesquisas encontradas na literatura deste domínio não diminui a relevância do problema abordado, mas contribui para identificar lacunas que podem ser resolvidas com a aplicação da semântica. A aplicação da semântica no domínio do código-fonte pode levar a uma nova abstração do código-fonte, possibilitando o desenvolvimento de aplicações pioneiras no nível semântico e não sintático.

A **questão de pesquisa** levantada para o problema é definida: como uma representação semântica do domínio de código-fonte baseada em ontologia fornece uma conceituação consensual, compartilhada e abrangente do código-fonte, que pode mitigar a heterogeneidade das diferentes linguagens de programação e possibilitar a interoperabilidade semântica entre os códigos-fonte? Atendendo à questão de pesquisa, o **objetivo** é definir uma fundamentação teórica de código-fonte baseada em uma representação ontológica e aplicada à interoperabilidade de código-fonte de diferentes linguagens de programação. Como **requisitos**, foram estabelecidos que a pesquisa deve: *R1:* prover interoperabilidade de código-fonte de diferentes linguagens de programação; *R2:* adotar engenharia de ontologias para a construção das ontologias; *R3:* prover uma fundamentação téorica de código-fonte, compartilhada, consensual e abrangente, independente de linguagem de programação. A validação dos requisitos deve seguir os **critérios de aceitação**: *C1:* suportar a representação de código-fonte de diferentes linguagens de programação; *C2:* suportar interoperabilidade de código-fonte de diferentes linguagens de programação;

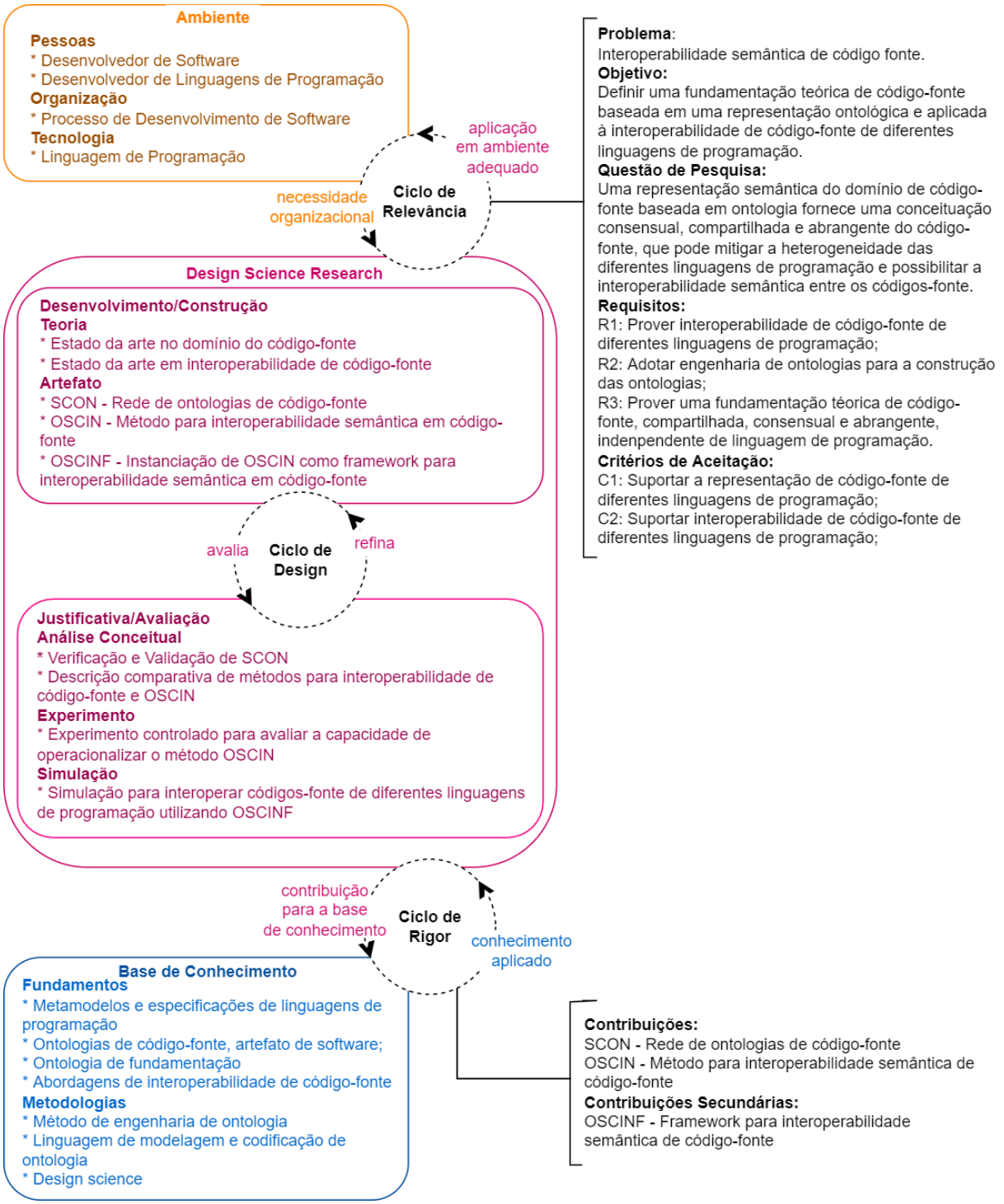


Figura 1 – Visão geral da aplicação do método *Design Science Research*.

C3: fundamentação teórica independente de linguagem de programação.

1.4.2 Ciclo de Rigor

O Ciclo de Rigor faz uso de teorias e métodos apropriados para a construção e avaliação dos artefatos. Nesta pesquisa, o Rigor é alcançado aplicando os conhecimentos relacionados à interoperabilidade semântica, ontologia e código-fonte.

Estudos identificados e analisados sobre interoperabilidade de código-fonte e desenvolvimento multilíngue (Seção 2.1) embasam a proposta desta pesquisa, levando a elaboração da rede de ontologias de código-fonte — SCON (apresentada no Capítulo 3), que é uma das principais contribuições desta pesquisa. O domínio de código-fonte é fundamentado e definido a partir de metamodelos e especificações das linguagens de programação, bem como faz uso de ontologia (Seção 2.2), tipo de recurso em ascensão e aceito para soluções de cunho semântico. A construção da ontologia é apoiada no método SABiOS (apresentado na Seção 5.2 e detalhado no Apêndice A, consiste em uma proposta de evolução do método SABiO (FALBO, 2014)) e apoiada na ontologia de fundamentação UFO – *Unified Foundational Ontology* (GUIZZARDI; WAGNER, 2004). O modelo operacional e de referência da ontologia são representados nas linguagens OWL (MCGUINESS; HARMELEN, 2004) e OntoUML (GUIZZARDI; GRAÇAS; GUIZZARDI, 2011).

Além disso, fundamentado na representação semântica do domínio de código-fonte, o método OSCIN foi elaborado para viabilizar a interoperabilidade de código-fonte de diferentes linguagens de programação baseada em ontologia (apresentado no Capítulo 4). OSCIN também é uma das principais contribuições desta pesquisa e, a partir dele, originase o *framework* OSCINF (apresentado no Capítulo 5) para apoiar a aplicação do método na interoperabilidade de código-fonte.

1.4.3 Ciclo de Design

O Ciclo de Design denota o interno e principal ciclo da pesquisa, cuja natureza está relacionada à construção, avaliação e *feedback* dos artefatos segundo a sua interação com os ciclos de Relevância e Rigor. Nesta pesquisa, o Design é alcançado por meio da construção de diferentes tipos de artefatos e suas avaliações.

SCON — *Source Code Ontology Network* (Capítulo 3), artefato do tipo modelo, rede de ontologias elaborada para representar os conceitos e relações definidos para o domínio de código-fonte. Para atender *R3*, o conhecimento representado foi extraído de metamodelos e especificações de diferentes linguagens de programação, acordadas com especialistas do domínio. O artefato seguiu o método SABiOS — *Systematic Approach for Building Ontologies with Scrum* (Seção 5.2 e Apêndice A) para a construção, verificação e validação das ontologias para atender a *R2*.

OSCIN — *Ontology-based Source Code Interoperability*, método para interoperabilidade de código-fonte baseada em ontologia (Capítulo 4), artefato do tipo método, foi elaborado para viabilizar a interoperabilidade semântica em código-fonte. Para atender *R1*, o método define o conjunto de passos genéricos que pode ser adotado para a interoperabilidade de código-fonte em duas linguagens de programação distintas. Ademais, uma descrição comparativa com outras abordagens baseadas em ontologia reforça as características desse requisito (Seção 7.2.11), bem como experimentos no subdomínio de orientação a objetos e mapeamento objeto/relacional, linguagem de programação Java e Python e perspectiva de *code smell* e métrica de código (Seções 6.1 e 6.2).

Atendendo a *C1*, as ontologias foram instanciadas com dados de projetos de software de duas linguagens de programação distintas. Atendendo a *C2*, o *framework* OSCINF foi aplicado a projetos de software de duas linguagens de programação distintas.

1.5 Estrutura de Capítulos

Este capítulo apresenta a introdução a esta tese, descrevendo seus aspectos gerais. O conteúdo restante é organizado como segue:

- **Capítulo 2. Referencial Teórico:** capítulo que apresenta uma visão geral do conhecimento relacionado ao tema desta pesquisa, introduzindo a ideia de interoperabilidade em código-fonte e descrevendo o papel da ontologia a favor desta interoperabilidade.
- **Capítulo 3. Representação Semântica de Código-Fonte:** capítulo que discute a representação semântica de código-fonte e apresenta SCON – *Source Code Ontology Network*, uma rede de ontologias formada para representar semanticamente os conceitos presentes em um código-fonte, independente de linguagem de programação.
- **Capítulo 4. Método para Interoperabilidade Semântica de Código-Fonte baseada em Ontologia:** capítulo que apresenta OSCIN — *Ontology-based Source Code Interoperability*, um método para interoperabilidade de código-fonte baseada em ontologia, que objetiva representar semanticamente o código-fonte escrito em diferentes linguagens de programação e aplicá-lo a diferentes propósitos de maneira unificada.
- **Capítulo 5. Framework para Interoperabilidade Semântica de Código-Fonte baseada em Ontologia:** capítulo que apresenta OSCINF — *Ontology-based Source Code Interoperability Framework*, um *framework* para interoperabilidade de código-fonte baseada em ontologia, que objetiva fornecer um cojunto de soluções para apoiar a aplicação do método OSCIN em diferentes subdomínios de código-fonte, linguagens de programação e propósitos de aplicação.

- **Capítulo 6. Avaliação da Interoperabilidade de Código-Fonte:** capítulo que apresenta aplicações e avaliações do método OSCIN para diferentes subdomínios de código-fonte, linguagens de programação e propósitos de aplicação a favor da interoperabilidade de código-fonte.
- **Capítulo 7. Trabalhos Relacionados:** capítulo que apresenta trabalhos relacionados a interoperabilidade de código-fonte e ao método OSCIN, como instancias da sua aplicação.
- **Capítulo 8. Considerações Finais:** capítulo que sumariza as ideias apresentadas nos capítulos anteriores, apresenta as contribuições desta pesquisa e direciona os trabalhos futuros.

2 Referencial Teórico

Este capítulo apresenta uma visão geral do conhecimento relacionado ao tema desta pesquisa, introduzindo a ideia de interoperabilidade em código-fonte e descrevendo o papel da ontologia a favor desta interoperabilidade.

2.1 Interoperabilidade de código-fonte

A interoperabilidade é um conceito multidimensional, abrangendo vários domínios de aplicação e terminologias. Dentre as terminologias utilizadas, destacamos as seguintes definições extraídas da ISO/IEC/IEEE 24765:2017[1]: (i) grau em que dois ou mais sistemas, produtos ou componentes podem trocar informações e usar as informações que foram trocadas (ISO/IEC25010, 2011; SEPTEMBER, 1990); (ii) capacidade de dois ou mais *Object Request Brokers* cooperarem para entregar solicitações ao objeto adequado (ISO/IEC19500-2, 2003); (iii) capacidade de comunicar, executar programas e transferir dados entre várias unidades funcionais de uma maneira que requer que o usuário tenha pouco ou nenhum conhecimento das características únicas dessas unidades (ISO/IEC2382, 2015); e (iv) capacidade de colaboração dos objetos, ou seja, capacidade de comunicação mútua de informações para troca de eventos, propostas, solicitações, resultados, compromissos e fluxos (ISO/IEC10746-2, 2009). Baseada no conceito de heterogeneidade (qualidade de ser dissimilar), nesta pesquisa definimos interoperabilidade como a **capacidade de trocar informações ou usá-las em um contexto heterogêneo**.

Além das diferentes terminologias, encontramos na literatura diferentes tipos de interoperabilidade (64 tipos catalogados em (FORD et al., 2007)), tais como: **Interoperabilidade de Sistemas** é definida como a capacidade dos sistemas operarem em conjunto (LAVEAN, 1980), ou seja, troca de informação entre sistemas, independentemente da forma em que eles foram desenvolvidos. **Interoperabilidade de Aplicação** é definida como a capacidade de dois ou mais dispositivos funcionarem juntos em um ou mais aplicativos (KOSANKE, 2005), tal como a adoção de padrões SMTP e SOAP. **Interoperabilidade de Sistemas de Informação** é definida como a capacidade de sistemas trocarem informação ou ter a habilidade de trocar informação. Nesse caso, observamos a adoção de interfaces, esquemas e formatos para a troca de informação (FILETO; MEDEIROS, 2003), tal como *web services*. **Interoperabilidade de Sistemas de Sistemas** é definida como a capacidade de diferentes sistemas se fundirem dinamicamente para formar um sistema holístico, provendo mais que a soma de suas capacidades individuais (DIMARIO, 2006). Nesse caso, observamos a interoperabilidade operacional que diz respeito às relações

[1] <https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:24765:ed-2:v1:en>

entre os sistemas em operação ou interoperabilidade técnica, suas interações entre si, com o ambiente e com os usuários (DIMARIO, 2006).

Diferente das interoperabilidades apresentadas acima, definimos interoperabilidade de código-fonte, aquela **capacidade dos códigos-fonte de trocar informações ou usá-las em um contexto heterogêneo de diferentes linguagens de programação**. Por um lado, a **interoperabilidade sintática de código-fonte** caracteriza-se pela troca e uso de informações no nível sintático da linguagem de programação (gramática), sem lidar com questões semânticas e de interpretação. Para alcançar a interoperabilidade sintática, enfrentamos desafios comuns a outros contextos (VELTMAN, 2001), tais como, identificação de todos os elementos em várias linguagens de programação; estabelecer regras para estruturação desses elementos; mapear e relacionar elementos equivalentes usando esquemas etc.; adotar regras equivalentes para unir diferentes gramáticas de linguagens de programação. Por outro lado, **interoperabilidade semântica de código-fonte** caracteriza-se pela troca e uso de informações no nível semântico da linguagem de programação, lidando com significados pré-estabelecidos, compartilhados e consensuais dos elementos representados na linguagem.

Enquanto a interoperabilidade sintática fornece uma solução tecnológica que pode resolver alguns problemas de interoperabilidade, a interoperabilidade semântica fornece soluções semióticas, linguísticas, filosóficas e sociais (PARK, 2004). Uma vez que esta pesquisa visa a interoperabilidade semântica de código-fonte, as seções a seguir fornecem mais detalhes sobre ontologias, que podem desempenhar um papel importante nesta interoperação (por exemplo, integração, comparação, tradução) de modelos produzidos em linguagens divergentes em sintaxe e semântica, mas cujas conceituações do mundo real se sobrepõem.

2.2 Ontologia

O termo ontologia tem origem no campo da filosofia, cujo interesse é estudar a natureza do ser e sua existência. Por outro lado, o significado do termo na área de Ciência da Computação se diverge, sendo definido como uma especificação explícita de uma conceitualização (GRUBER, 1993), ou conjunto de conceitos e termos ligados entre si (numa rede) que podem ser usados para descrever alguma área do conhecimento ou construir uma representação para o conhecimento (SWARTOUT; TATE, 1999), ou ainda documento/arquivo que define formalmente as relações entre termos (BERNERS-LEE; HENDLER; LASSILA, 2001).

No contexto desta pesquisa, ontologia é uma especificação explícita e formal de uma conceitualização compartilhada (STUDER; BENJAMINS; FENSEL, 1998), em que explícita se refere à maneira clara e bem definida; formal, refere-se a ser legível para

computador; conceituação, refere-se ao modelo abstrato do mundo real acordando para um propósito; e compartilhada, refere-se ao conhecimento consensual de um comunidade sobre um determinado domínio.

Representar a semântica formal para um determinado domínio implica conceituar os objetos do domínio e seus inter-relacionamentos de forma declarativa (JARRAR; MEERSMAN, 2002). Assim, ontologias devem apoiar compromissos ontológicos formais e acordados para o domínio.

Segundo o processo de desenvolvimento, podemos classificar uma ontologia como (FALBO et al., 2013): **de Referência**, quando o objetivo principal é a representação do conhecimento buscando identificar com maior clareza e precisão a natureza semântica das entidades e suas inter-relações no domínio; ou **Operacional**, quando o objetivo principal é garantir propriedades computacionais desejáveis. Uma ontologia operacional é representada em uma linguagem passível de processamento por máquina, tal como Web Ontology Language (OWL),[2] recomendada pela W3C.

Segundo sua generalidade, uma ontologia pode ser classificada como (SCHERP et al., 2011): **de Domínio** — *Domain Ontology*, uma ontologia contendo conhecimento de domínios particulares a fim de representar um mini-mundo específico; **de Núcleo** — *Core Ontology*, uma ontologia que representa diferentes pontos de visão sobre um domínio ou que perpetua sobre diferentes domínios; e **de Fundamentação** — *Foundational Ontology*, uma ontologia que representa um conhecimento geral advindo de diversas áreas do conhecimento.

As ontologias de fundamentação são as tecnologias chave para integrar informações heterogêneas provenientes de diferentes fontes (EUZENAT; SHVAIKO, 2007). Encontramos na literatura diferentes propostas de ontologias de fundamentação, tal como *Basic Formal Ontology* (BFO) (GRENON; SMITH, 2003), baseada no realismo e acordada com os modos de existência no tempo das entidades que populam o mundo; *Descriptive Ontology for Linguistic and Cognitive Engineering* (DOLCE) (MASOLO et al., 2003), baseada em viés cognitivo adotando uma abordagem Descritiva / Multiplicativa; e *Unified Foundational Ontology* (UFO) (GUIZZARDI, 2005), baseada nas teorias de ontologia formal, lógica filosófica, filosofia da linguagem e psicologia cognitiva que, diferente das anteriores, não se aplica a uma área específica e provê uma base para a modelagem conceitual.

Métodos de construção de ontologias têm sido propostos na Engenharia de Ontologias a fim de guiar o seu processo de construção. Encontramos na literatura métodos que adotam diferentes princípios, decisões de *design* e atividades, tal como, Methontology (FERNÁNDEZ-LÓPEZ, 1997), baseado no processo de desenvolvimento de software, ciclo iterativo, prototipação e reuso de ontologias; UPON (NICOLA; MISSIKOFF; NA-

[2] <https://www.w3.org/OWL>

VIGLI, 2005), baseado no processo de software UP – *Unified Process* e suportado pela linguagem UML, contendo ciclo iterativo e incremental; DILIGENT (PINTO; TEMPICH; STAAB, 2009), baseado na evolução de protótipos e desenvolvimento distribuído; OntoloClean (GUARINO et al., 2000), baseado na aplicação de análise ontológica; NEON (SUÁREZ-FIGUEROA et al., 2012), baseado na construção de redes de ontologia com reuso de recursos ontológicos e não ontológicos; e SABiO (FALBO, 2014) baseado na construção de ontologias operacionais e de referência a partir de análise ontológica e reuso de ontologias.

Assim, para a representação semântica do código-fonte, adotamos a representação da conceituação em uma ontologia, seguindo tanto um método de construção de ontologia quanto uma linguagem ontológica que suporta escolhas de representação ontologicamente consistentes, ou seja, explicitação da natureza ontológica dos elementos e de suas relações. A seguir apresentamos uma visão geral da ontologia de fundamentação UFO, a linguagem OntoUML, a linguagem OWL e o método SABiO, adotados no contexto desta pesquisa.

2.2.1 Ontologia de Fundamentação UFO

Uma ontologia de fundamentação define noções básicas de objetos, relações, eventos, processos e outros. Uma ontologia consistente normalmente é construída a partir de uma ontologia de fundamentação (ROUSSEY et al., 2011), provendo os conceitos básicos sobre os quais qualquer ontologia específica do domínio é construída (GUIZZARDI, 2005). O atendimento aos requisitos de interoperabilidade está relacionado à aplicação de uma ontologia de fundamentacao e uma linguagem verdadeiramente ontológica, tal como a ontologia UFO e a linguagem OntoUML (GUIZZARDI, 2019).

UFO — *Unified Foundational Ontology* é uma ontologia de fundamentação baseada nas teorias de ontologia formal, lógica filosófica, filosofia da linguagem e psicologia cognitiva (GUIZZARDI; FALBO; GUIZZARDI, 2008). O núcleo desta ontologia é **UFO-A**, ontologia de objetos (*endurants*) que lida com aspectos de modelagem conceitual estrutural. **UFO-B** é uma ontologia de eventos (*perdurants*) que lida com aspectos como mereologia perdurante, ordenação temporal, participação de objeto em perdurantes, causalidade, mudança e a conexão entre *perdurants* e *endurants* via disposições e outros. **UFO-C** é uma ontologia de entidades intencionais e sociais que lida com noções de crenças, desejos, intenções, objetivos, ações, compromissos e outros.

A Figura 2 apresenta um fragmento de UFO-A relacionado a ***Universal***, padrões de características que podem ser percebidos em uma série de diferentes ***Particulars***. Sob o princípio de aplicação, *Universal* julga se um particular é instância daquele *Universal*. Assim, *Particulars* são instâncias de *Universals* como um tipo de primeira ordem (1stOT), que são instâncias de tipos de segunda ordem (2ndOT). *Universal* pode ser ***Monadic Universal***, padrão de característica aplicado a entidades, ou ***Relation***, aplicado a relações,

conectando entidades.

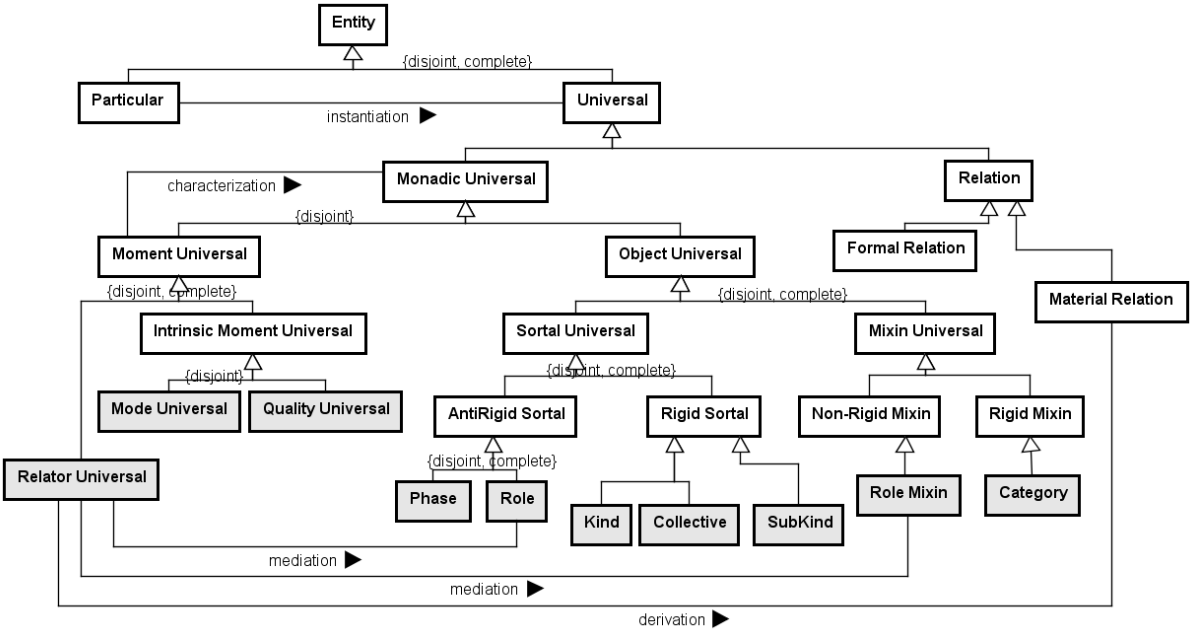


Figura 2 – Fragmento de UFO

Para *Monadic Universal*, ***Object Universal*** é um *particular* existencialmente independente de outros *particulars* que não compartiham uma parte comum, com qualidades espaço-temporais (diretas) e que são fundados na matéria, tal como uma pessoa individual; e ***Moment Universal*** é uma propriedade individualizada existencialmente dependente de outro *particular*, tal como a dor de cabeça de uma pessoa.

Em *Moment Universal*, enquanto um *Intrinsic Moment* é existencialmente dependente de um único indivíduo, tal como uma dor de cabeça; um ***Relator Universal*** é dependente de uma pluralidade de indivíduos, tal como um tratamento médico para dores de cabeça. Para vários *Intrinsic Moment* perceptíveis ou concebíveis há uma dimensão de qualidade associada na cognição humana, tal como a cor da maçã que assume seu valor em um domínio de cor tridimensional constituído pelas dimensões matiz, saturação e brilho, não totalmente independente. ***Quality Universal*** são estes *intrinsic moments* associados a uma *quality structure*, relacionadas às dimensões e domínios da qualidade. ***Mode Universal*** são *intrinsic moments* cujo *universals* não são diretamente relacionados às *quality structures*.

Em ***Object Universals***, enquanto ***Mixin Universal*** reune *Sortal Universal* com diferentes critérios de identidade e não tem instâncias diretas; ***Sortal Universal*** determina a individuação, persistência e identidade de suas instâncias. *Mixin Universal* distingue entre ***Rigid Mixin***, quando representa propriedades essenciais para múltiplos *Rigid Sortal*; e ***Non-Rigid Mixin***, quando representa propriedades essenciais e acidentais para múltiplos *Anti Rigid Sortal*. Um *Rigid Mixin* é definido como ***Category*** (generaliza entidades

rígidas com diferentes princípios de identidade) e um *Non-Rigid Mixin* é definido como ***Role Mixin*** (generaliza entidades anti rígidas com diferentes princípios de identidade).

Assim, todos os *Universals* carregam um princípio de aplicação (julga se um *particular* é uma instância daquele *universal*) e apenas *Sortals* carregam um princípio de identidade (julga se dois particulares são iguais) para suas instâncias. *Sortal Universal* distingue entre ***Rigid Sortal***, quando cada instância é válida para todos os mundos possíveis; e ***Anti Rigid Sortal***, quando deve haver um mundo possível onde a instância não é válida, não oferecendo um critério de identidade. Um *Rigid Sortal* é definido como ***Kind*** (oferece princípio de identidade uniforme para suas instâncias) ou ***Subkind*** (herda princípio de identidade de um único Kind, direta ou indiretamente) ou ***Collective*** (oferece princípio de identidade uniforme para a coleção, não para cada indivíduo da coleção) e um *Anti Rigid Sortal* é definido como ***Phase*** (relacionalmente independentes como possíveis estágios na história de um *particular*) ou ***Role*** (relacionalmente dependente de uma propriedade relacional intrísica).

Por outro lado, ***Relation*** pode ser uma ***Formal Relation***, relação mantida diretamente entre duas ou mais entidades, sem qualquer intervenção individual, tal como *existential dependence*, *part-of*, *subset-of* e *instantiation*; ou uma ***Material Relation***, relação com estrutura material própria mediada a partir de uma entidade *relator*, tal como tratamento médico que conecta o paciente para a unidade médica. A relação entre o *relator* e a entidade relacionada é definida como *mediation*.

2.2.2 Linguagem OntoUML

UFO-A pode ser representada em OntoUML, linguagem de modelagem conceitual concebida como uma versão ontologicamente bem fundamentada do fragmento UML 2.0 de diagramas de classes (GUIZZARDI et al., 2015). OntoUML é um perfil UML que incorpora importantes distinções fundamentais de UFO-A por meio estereótipos aplicados a classes e relações de um diagrama de classes.

Essas distinções de UFO são explicitadas na linguagem de modelagem OntoUML por meio de estereótipos no diagrama de classe UML, resumidos na Tabela 1.

2.2.3 Linguagem OWL

A linguagem OWL (Web Ontology Language) pode descrever formalmente a semântica de um domínio por meio de classes, propriedades e relações que permitem: (i) expressividade, na representação de ideias extremamente complicadas ou muito simples sobre os dados, usando relações hierárquicas e dinâmicas; (ii) flexibilidade, em representar os dados como triplas (sujeito-predicado-objeto) que permitem alterações, incrementos e melhorias no modelo de dados sem grandes desvios; e (iii) eficiência, no suporte ao

Tabela 1 – Estereótipos de OntoUML.

Estereótipo	**Distinção Ontológica**	**Exemplo**
«category»	Tipo rígido cujas instâncias compartilham propriedades intrínsecas comuns, mas obedecem a diferentes princípios de identidade (entidades rígidas não-sortais).	Member
«collective»	Complexo funcional formado por partes iguais, com princípio de identidade próprio que se mantém tanto para suas instâncias como em todos os mundos possíveis.	Database
«kind»	Tipo sortal rígido que é formado por partes distintas (complexo funcional) e fornece um princípio de identidade para suas instâncias.	Class
«role»	Tipo sortal anti-rígido cuja condição de especialização é dada por propriedades extrínsecas (relacionais).	Subclass
«subkind»	Tipo sortal rígido cujas instâncias herdam um princípio de identidade a partir de um *kind*.	Method
«relator»	Conceito que conecta outros conceitos, e portanto existencialmente dependente deles.	Inheritance
«quality»	Tipo cujas instâncias representam propriedades intrínsecas de um indivíduo associado a uma estrutura de qualidade.	Value Type

raciocínio e inferência no modelo de dados, minimizando o armazenamento explícito e a complexidade das consultas que são escritas na linguagem de consulta SPARQL.[3]

OWL é uma extensão do vocabulário de RDF (*Resource Description Framework*) que modela declarações na forma de triplas e adiciona características de propriedades. Tripla é uma entidade de dado atômica em RDF (LASSILA; SWICK, 1999), formada por 3-tuplas no formato sujeito-predicado-objeto, onde predicado e objeto são a propriedade e o valor do recurso denotado pelo sujeito. Toda tupla é unicamente identificada por uma URI — *Uniform Resource Identifier*. A Listagem 2.1 apresenta a tripla correspondente à declaração "*A classe Person é composta pelo método getName*", formada pelo sujeito (*classe people*), predicado (*composta pelo método*) e objeto (*getName*).

Listagem 2.1 – Tripla RDF.

```
1 <http://ex.org#Person http://ex.org#composedByMethod http://ex.org#getName>
```

Estas triplas podem ser manipuladas por meio da linguagem SPARQL, sendo que tanto o sujeito como o predicado e o objeto podem ser representados por uma variável, atribuindo o simbolo "?". Todos os operadores são binários e seus operandos são triplas ou filtros. A Listagem 2.2 apresenta a consulta correspondente aos métodos e suas respectivas classes.

Listagem 2.2 – Consulta SPARQL.

```
1 SELECT ?classe ?metodo
2 WHERE
3   {
4     ?classe http://www.w3.org/1999/02/22-rdf-syntax-ns#type http://ex.org#
      classe .
5     ?classe http://ex.org#composedByMethod ?metodo .
6   }
```

[3] <https://www.w3.org/TR/rdf-sparql-query>

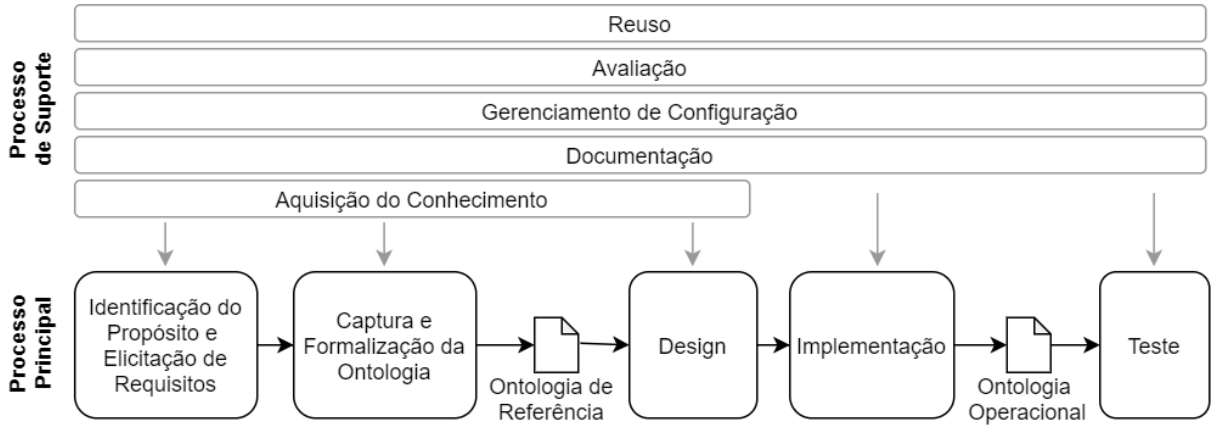


Figura 3 – Processo de SABiO (adaptado de (FALBO, 2014))

2.3 Método SABiO

SABiO (FALBO, 2014) é uma abordagem sistemática para a construção de ontologias operacionais e de referência que aplica análise ontológica e reuso de ontologias. O método é composto de cinco fases principais e processos de apoio, apresentados na Figura 3 e descritos a seguir.

Os processos principais contemplam: **Identificação de Propósito e Elicitação de Requisitos**, que identifica a finalidade e os usos pretendidos da ontologia, definindo seus requisitos funcionais (Questões de Competência) e não funcionais (NFRs) e decompondo a ontologia em partes independentes e interconectadas para facilitar a manutenção e o desenvolvimento; **Captura e Formalização da Ontologia**, que visa registrar de forma objetiva a conceituação do domínio com base em uma análise ontológica utilizando uma ontologia de fundamentação, sugerindo a adoção de um modelo gráfico para representar a ontologia de referência; **Design**, onde define-se o ambiente de implementação e NFRs tecnológicos para a codificação da ontologia de referência em uma linguagem legível por máquina; **Implementação** que segue com a codificação da ontologia na linguagem operacional escolhida; **Teste** que verifica a ontologia operacional usando consultas no ambiente de implementação e valida a ontologia nos aplicativos de software de acordo com seus usos pretendidos.

Além disso, o método SABiO sugere cinco processos de apoio, aplicados da seguinte forma: **Aquisição de Conhecimento**, para reunir o conhecimento do domínio de forma confiável por meio de especialistas e material bibliográfico; **Reutilização**, para aproveitar as conceituações já estabelecidas para o domínio; **Documentação**, para registrar os resultados do processo de desenvolvimento por meio de uma Especificação de Ontologia de Referência; **Gerenciamento de Configuração**, para controlar alterações, versões e entrega por meio de repositório; e **Avaliação**, para avaliar a adequação da ontologia por meio de: *verificação*, garantindo que a ontologia atenda aos seus requisitos; e *validação*, garantindo que a ontologia seja capaz de representar situações do mundo real.

2.4 Considerações Finais

Este capítulo apresentou fundamentos sobre interoperabilidade, código-fonte e ontologia relacionados a esta pesquisa, bem como desafios para a interoperabilidade semântica de código-fonte. Sabemos que uma solução semanticamente interoperável que gerencia vários conflitos semânticos entre diferentes linguagens de programação é uma tarefa ousada, ela deve fornecer a capacidade de detectar e resolver incompatibilidades na semântica e estruturas das linguagens, bem como uma linguagem de consulta padrão sobre o código-fonte. No entanto, em capítulos posteriores, apresentamos uma rede de ontologias de código-fonte (Capítulo 3) e um método de interoperabilidade de código-fonte (Capítulo 4) e seus desdobramentos para demonstrar que os códigos-fonte escritos em linguagens de programação que compartilham um compromisso consistente comum podem interoperar significativamente entre si.

3 Representação Semântica de Código-Fonte

Neste capítulo discutimos a representação semântica de código-fonte e apresentamos SCON – *Source Code Ontology Network*, uma rede de ontologias formada para representar semanticamente os conceitos presentes em um código-fonte, representado em diferentes linguagens de programação.

Para demonstrar melhor o contexto de código-fonte, uma representação adaptada da Ontologia de Software (SwO) e Ontologia de Processo (SPO) (RUY et al., 2016) é apresentada na Figura 4, cujos conceitos são apresentados com o prefixo SwO:: e SPO::.

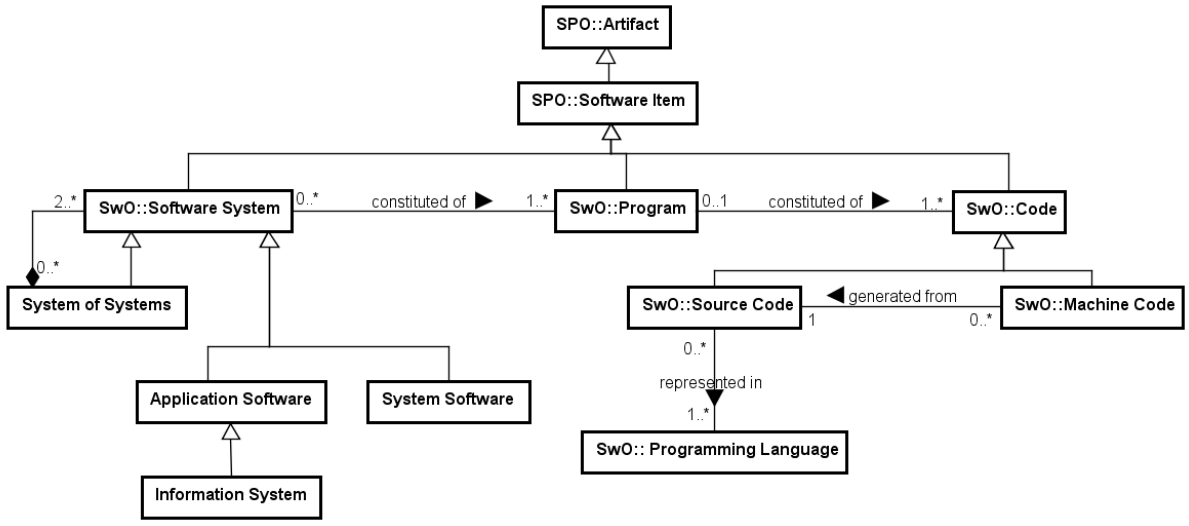


Figura 4 – Representação do contexto de código-fonte - Adaptado a partir de (RUY et al., 2016)

Uma vez que um Artifact é um objeto consumido ou produzido durante o processo de software e é representado em uma linguagem (conjunto de símbolos usados para representar informações), podemos definir o Source Code como um Software Item (pedaço de software produzido durante o processo de software) que é um subtipo de Code (conjunto de instruções de computador e definições de dados) representado em Programming Languages, cuja identidade pode ser dada por sua estrutura sintática.

Por sua vez, o código-fonte constitui Programs que são itens de software que visam produzir um determinado resultado através da execução em um computador, de uma maneira particular, dada por uma especificação de programa. Um programa é constituído por código, mas não é idêntico ao código. O código pode ser alterado sem alterar a identidade de seu programa, cuja identidade é dada por sua especificação (WANG et al., 2014).

Por sua vez, o programa constitui Software Systems, item de software que visa

satisfazer uma especificação do sistema, a respeito de uma mudança desejada em uma estrutura de dados dentro de um computador, abstraindo do comportamento (RUY et al., 2016). Identificamos que sistemas de software podem ser especializados em System Software, quando um sistema de software é designado para realizar um específico conjunto de funções relacionadas a manter e operar outros itens de software, tal como, sistema operacional, *engine* de game e serviço; e Application Software (app), quando um sistema de software é designado para realizar um específico conjunto de funções relacionadas ao próprio sistema de software, tal como, sistema de edição de texto, sistema de comunicação; e Information System, que é designado para realizar um específico conjunto de funções relacionadas a interagir e informar usuários em diferentes contextos organizacionais e sociais (BOELL, 2015). Diferentes sistemas de software podem constituir um System of Systems de modo que sistemas de software com especificações singulares são integrados a um metasistema agregando suas capacidades (TAYLOR et al., 2015).

Nesse sentido, o código-fonte é uma sequência bem formada de instruções de computador e definições de dados expressas em uma linguagem de programação (WANG et al., 2014), sendo definida por uma gramática formal, composta por um conjunto de símbolos organizados com suas respectivas sintaxes e semânticas. As linguagens de programação podem ser vistas como linguagens artificiais definidas por pessoas cujo propósito inicial se refere à comunicação com computadores, mas, tão importante quanto, à comunicação de algoritmos entre pessoas (SLONNEGER; KURTZ, 1995).

Assim como na linguística, as linguagens de programação são formadas pela (i) sintaxe, que se refere a como os símbolos podem ser combinados para formar expressões válidas na linguagem; (ii) semântica, que se refere ao significado de expressões válidas da linguagem; e (iii) pragmática, que se refere à relação dos símbolos da linguagem com as interpretações de seus usuários relacionadas a fenômenos psicológicos e sociológicos (NEURATH, 1938). A sintaxe deve ser especificada antes da semântica, uma vez que o significado pode ser dado apenas a expressões válidas na linguagem; e a semântica deve ser formulada antes de considerar questões pragmáticas, uma vez que a interação com humanos só pode ser considerada em expressões com significado compreendido (SLONNEGER; KURTZ, 1995).

Por um lado, nas linguagens de programação, a **sintaxe** é definida por meio de um vocabulário (conjunto de símbolos), que por sua vez são agrupados em expressões válidas de acordo com as regras definidas em uma gramática (SLONNEGER; KURTZ, 1995). A gramática especifica a linguagem por meio de uma descrição estrutural do vocabulário e suas combinações válidas. A linguagem de programação define uma sintaxe concreta e sua implementação adota uma sintaxe abstrata, ou seja, a sintaxe concreta faz parte da definição da linguagem representada na gramática livre de contexto (*context free grammar*) e a sintaxe abstrata faz parte da definição de uma implementação da

linguagem representada na árvore de sintaxe abstrata (*abstract syntax tree*). Assim, a sintaxe lida apenas com a forma e estrutura dos símbolos em uma linguagem, sem qualquer consideração dada ao seu significado.

Por outro lado, a definição **semântica** de uma linguagem consiste em um domínio semântico e um mapeamento semântico da sintaxe para o domínio semântico. Enquanto o domínio semântico ou conceituação de domínio se refere à representação abstrata de certos aspectos de entidades que existem em um determinado domínio, o mapeamento semântico ou função de interpretação refere-se ao significado de cada uma das expressões da linguagem em termos de um elemento do domínio semântico (GUIZZARDI, 2005).

Nesta pesquisa, semântica é entendida como o mapeamento (interpretação) do vocabulário da linguagem de programação para conceitos que representam entidades do mundo real na conceituação do domínio de código-fonte. Visto que as conceituações são entidades abstratas que existem apenas nas mentes das pessoas que usam uma linguagem específica, essas conceituações devem ser capturadas em artefatos concretos (GUIZZARDI, 2005).

Assim, diferentes abstrações e representações de código-fonte em linguagens de programação criam um contexto de heterogeneidade. Baseado no conceito de heterogeneidade (SHETH, 1999), definimos dois tipos diferentes de heterogeneidade em código-fonte: heterogeneidade sintática e semântica. Neste contexto, considere um exemplo em que uma dada realidade apresenta as características de um retângulo que pode ser abstraída em conceitos da linguagem de programação Java, Python e C++, sendo representada em suas respectivas linguagens de programação, como apresentado na Listagem 3.1.

Listagem 3.1 – Exemplo de heterogeneidade em código-fonte.

```
1  ------------------------------------------------------------------------
2  Código fonte em linguagem Java
3  ------------------------------------------------------------------------
4  public class Rectangle extends Shape{
5      private static final int SIDE = 4;
6      private double _width;
7      private double _height;
8      public Rectangle (double width, double height){
9          this._width=width;
10         this._height=height;  };
11 }
12 ------------------------------------------------------------------------
13 Código fonte em linguagem C++
14 ------------------------------------------------------------------------
15 class Rectangle: private Shape{
16 private:
17     static const int SIDE = 4;
18     double _width;
19     double _height;
20 public:
21     Rectangle (double width, double height){
```

```
22          this->_width=width;
23          this->_height=height;   };
24 }
25 ------------------------------------------------------------------------
26 Código fonte em linguagem Python
27 ------------------------------------------------------------------------
28 class Rectangle (Shape):
29     SIDE = 4
30     def __init__(self, width, height){
31          self._width=width
32          self._height=height
33 }
```

A **heterogeneidade sintática** é observada quando linguagens com sintaxes diferentes são atribuídas a conceitos correspondentes. Conforme mostrado na Listagem 3.1, um retângulo é caracterizado por 4 lados, cuja abstração em Java e C++ é dada por uma variável que declara quatro lados não alterados durante a execução e cujo valor é compartilhado pelos objetos retângulos. No entanto, essa conceituação comum é representada de maneira diferente. Como observado na linha 5, em Java, a variável SIDE é declarada com a sintaxe static para representar o seu compartilhamento com todos os objetos Rectangle e com a sintaxe final para representar sua imutabilidade. Já em C++, como observado na linha 17, a variável SIDE é declarada com a sintaxe const para representar sua imutabilidade. Ademais, em Python, a abstração é dada por uma variável declarada com quatro lados que podem ser alterados durante a execução e cujo valor é compartilhado pelos objetos retângulos. Conforme observado na linha 29, a variável SIDE é declarada no escopo da classe (fora do método) para representar que seu valor é compartilhado pelos objetos Rectangle, não declarando nenhuma sintaxe para a sua imutabilidade, uma vez que essa conceituação não é abstraída pela linguagem. Assim, neste exemplo, observamos um caso de heterogeneidade sintática entre as linguagens Java e C++ e, adicionalmente, observamos que a linguagem Python não abstrai uma dada realidade do domínio de código-fonte.

A **heterogeneidade semântica** é observada quando conceitos correspondentes possuem diferentes interpretações. Conforme mostrado na Listagem 3.1, um retângulo é abstraído em Java, C++ e Python como um subtipo de uma forma comum a outras formas além de retângulo, cujas características são herdadas. No entanto, essa conceituação que parece comum é interpretada de maneira diferente, de modo que a conceituação de herança em Java é dada pela capacidade de uma classe herdar características de uma (apenas uma) outra classe; em Python é dada pela capacidade de uma classe herdar características de outras classes; e em C++ é dada pela capacidade de uma classe herdar características de outras classes e alterar o nível de acesso da classe herdada. Assim, neste exemplo, observamos um caso de heterogeneidade semântica entre as linguagens Java, C++ e Python.

Portanto, mesmo que algumas linguagens de programação adotem sintaxes se-

melhantes, ou seja, os mesmos vocabulários para se referir a entidades de programação, geralmente associam significados diferentes a esses vocabulários e diferentes interpretações das conceituações aparentemente comuns. Neste sentido, ontologias podem ser uma representação consistente do mundo real dentro de um contexto, de acordo com a interpretação da realidade, apresentada nas subseções seguintes como uma rede de ontologias de código-fonte.

3.1 SCON - Rede de Ontologias de Código-Fonte

Uma rede de ontologias é definida como uma coleção de ontologias relacionadas entre si por meio de diferentes meta-relacionamentos (PETER et al., 2006). Tais ontologias são organizadas de forma a manter a consistência dos conceitos compartilhados, reduzir sua sobreposição e facilitar sua reutilização. Ademais, as redes podem ser conectadas a fim de formar uma rede de redes de ontologias.

SCON não pretende ser uma "rede de ontologia completa" de código-fonte, mas uma ontologia em evolução que apoia o desenvolvimento e integração de outras ontologias do domínio. Assim, SCON adota três premissas principais (RUY et al., 2016): (i) ser alicerçada em uma base bem fundamentada para o desenvolvimento de ontologias; (ii) oferecer mecanismos para fácil construção e integração de novas ontologias do domínio de código-fonte à rede; e (iii) promover a integração, mantendo uma semântica consistente para conceitos e relações em toda a rede.

Para isso, SCON adota arquitetura de três camadas (RUY et al., 2016): (i) *Foundational Layer:* ontologia de fundamentação UFO que fornece a base necessária para os conceitos e relações de todas as ontologias da rede; (ii) *Core Layer:* ontologias usadas para representar o conhecimento comum do domínio, necessário para as ontologias dos subdomínios específicos; (iii) *Domain-specific Layer:* ontologias dos subdomínios específicos de código-fonte, apoiadas nas *Core Ontologies* e na *Foundational Ontology*. Por fim, SCON se integra com ontologias externas à rede, tanto ontologias de domínio como de núcleo, bem como de outras redes.

SCON está apoiada na ontologia de fundamentação **UFO** – *Unified Foundational Ontology* (detalhada na Seção 2.2.1) na *foundational layer* e na ontologia de núcleo **SCO** – *Source Code Ontology* na *core layer*. Na *domain-specific layer* conta com as ontologias de domínio **OOC-O** — *Object-Oriented Code Ontology*, — ontologia que representa entidades presentes em tempo de compilação no código-fonte orientado a objetos (OO) e **ORM-O** — *Object/Relational Mapping Ontology* — ontologia que representa a semântica das entidades presentes em tempo de compilação no código-fonte que adota *frameworks* de mapeamento objeto/relacional (ORM) (elaborada no contexto do trabalho de um aluno de mestrado (ZANETTI, 2020)). Ademais, estão em construção as ontologias de domínio

DINF-O – *Dependecy Injection Framework Code Ontology* — ontologia que representa a semântica das depências de código-fonte que adota *frameworks* de injeção de dependência — e **FUNC-O** – *Functional Code Ontology* — ontologia que representa a semântica da aplicação de funções em código-fonte.

A Figura 5 apresenta como a rede de ontologias SCON é arquitetada e interligada a outras redes de ontologias, representada graficamente na forma de um grafo. No grafo, os nós representam *Foundational Ontology* (amarelo), *Core Ontology* (rosa) e *Domain Ontology* (verde); as arestas pontilhadas representam as relações de dependência entre as ontologias, indicando uma relação de especialização, reutilização ou outra. Ontologias em construção são identificadas com o nome sublinhado.

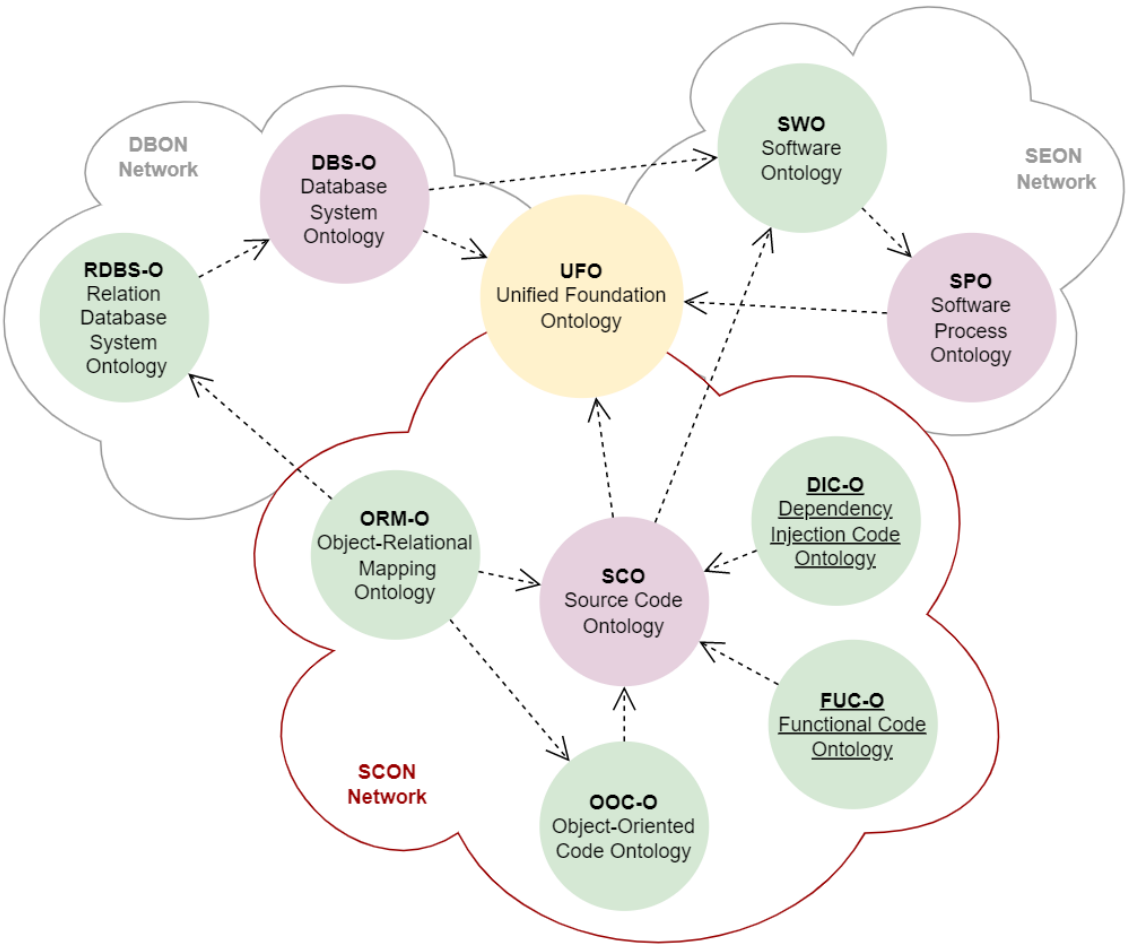


Figura 5 – Rede de Ontologia SCON

Dado que código-fonte é um artefato de software, SCON está relacionada à rede SEON – *Software Engineering Ontology Network* (RUY et al., 2016), rede de ontologias de engenharia de software fundamentada na ontologia UFO que define, entre outros conceitos, código-fonte como um pedaço de software produzido durante o processo de software.

A rede DBON – *Data Base Ontology Network* é uma rede de ontologias que cobre

o domínio de banco de dados e que está relacionada a rede SEON a fim de posicionar banco de dados como um software (elaborada no contexto desse trabalho para apoiar a representação da ontologia ORM-O). Atualmente DBON é formada pelas ontologias DBS-O e RDBS-O, fundamentadas na ontologia UFO. DBS-O representa o domínio de sistema de banco de dados e se relaciona com SwO a fim de contextualizar banco de dados como um sistema. O domínio de banco de dados relacional é representado na ontologia RDBS-O, relacionada a ontologia DBS-O.

A seções seguintes apresentam mais detalhes sobre as ontologias que compõem esta rede.

3.2 SCO - Ontologia de código-fonte

A ontologia de núcleo SCO — *Source Code Ontology* — visa identificar e representar a semântica dos principais conceitos presentes no código-fonte em tempo de compilação.

A fim de construir uma ontologia de código-fonte, seguimos o método SABiOS, método que define sistematicamente as atividades para o desenvolvimento de ontologias de referência e para a sua implementação como ontologias operacionais, apresentado na Seção 5.2 e detalhado no Apêndice A.

Os seguintes requisitos não funcionais (RNF) foram elicitados para SCO: **RNF1** – ser incorporada a uma rede de ontologias para facilitar a reutilização de outras ontologias e, consequentemente, expandir suas própias possibilidades de reuso e associação; **RNF2** – ser definida a partir de fontes de conhecimento reconhecidas na literatura; **RNF3** – ser definida a partir do conhecimento consensual do domínio; **RNF4** – ser fundamentada por uma ontologia de fundamentação a fim de reutilizar os compromissos ontológicos e axiomas já consensualmente estabelecidos.

Para responder ao **RNF1**, a ontologia foi integrada à Rede de Ontologia de Engenharia de Software (SEON), a fim de reutilizar conceitos de engenharia de software relacionados a código-fonte. Duas ontologias da SEON foram reutilizadas: a Software Process Ontology (SPO) e a Software Ontology (SwO), identificadas na SCO com os acrônimos correspondentes (*SPO::* e *SwO::*, respectivamente) e destacadas usando cores diferentes. Em relação ao **RNF2** e **RNF3**, o desenvolvimento da ontologia foi suportado por um processo de aquisição de conhecimento que utilizou fontes consolidadas de conhecimento referentes a diversas linguagens de programação. Finalmente, em resposta ao **RNF4**, a ontologia foi baseada na ontologia de fundamentação UFO-A e na aplicação de análise ontológica.

Para requisitos funcionais (RF), definimos iterativamente o seguinte conjunto de questões de competência (CQs): **RF01**- Quais os principais componentes de um código-

fonte? **RF02**- Como os componentes são organizados no código-fonte? **RF03**- Quais tipos estão presentes em um código-fonte? **RF04**- Quais variáveis estão presentes em um código-fonte? **RF05**- Qual tipo está associado a uma determinada variavel? **RF06**- Quais blocos de código estão presentes em um código-fonte? **RF07**- Quais procedimentos estão presentes em um código-fonte? **RF08**- Quais funções estão presentes em um código-fonte? **RF09**- Qual tipo de retorno de uma determinada função?

Durante a captura e formalização da ontologia, realizamos análises ontológicas baseadas em UFO, representando SCO no modelo OntoUML. Esse processo foi conduzido iterativamente, a fim de abordar diferentes aspectos/refinamentos a cada iteração e de maneira interativa, para que especialistas de domínio e engenheiros de ontologia pudessem discutir a conceituação do domínio.

Inicialmente, SCO foi elaborada no contexto de orientação a objetos, selecionando as três linguagens de programação mais populares de acordo com os índices TIOBE,[1] IEEE Spectrum[2] and Redmonk,[3] a saber: Python, Java e C++. No entanto, com a construção de ontologias em outros subdomínios (tais como mapeamento objeto/relacional e programação funcional), independentes da orientação a objetos, seus conceitos foram separados em uma *Core Ontology* comum ao domínio de código-fonte e outras linguagens de programação foram adicionadas. Assim como o subdomínio de programação funcional que adicionou as linguagens R e Haskel identificadas a partir dos índices mencionados anteriormente.

A Figura 6 apresenta a ontologia SCO e como seus conceitos estão integrados às ontologias SPO (representada na cor verde) e SwO (representada na cor azul) da rede SEON. Uma nova análise ontológica está prevista em trabalhos futuros (Seção 8.3).

A ontologia **SPO** estabelece uma conceituação comum no domínio do processo de software (processos, atividades, recursos, pessoas, artefatos, procedimentos, etc.). Reutilizamos o conceito de **Artifact**, objeto consumido ou produzido durante o processo de software, que é *represented in* **Language** (conceito adicionado pela SCO), um conjunto de simbolos usados para representar informações. Um artefato de software pode ser, entre outras coisas, um **Software Item** como um pedaço de software produzido durante o processo de software.

A ontologia **SwO** especializa ainda mais este conceito, um **Software System** é um Software Item que visa satisfazer uma especificação do sistema que é *constituted of* **Programs**, que são Software Itens que visam produzir um determinado resultado através da execução em um computador, de uma maneira particular, dada por uma especificação de programa. Por sua vez, Programs são *constituted of* **Code**, um Software Item representando um conjunto de instruções de computador e definições de dados que são *represented in* **Program Language** como um **Source Code**.

[1] <https://www.tiobe.com/tiobe-index/>
[2] <https://spectrum.ieee.org/at-work/tech-careers/top-programming-language-2020.>
[3] <https://redmonk.com/sogrady/2020/07/27/language-rankings-6-20/>

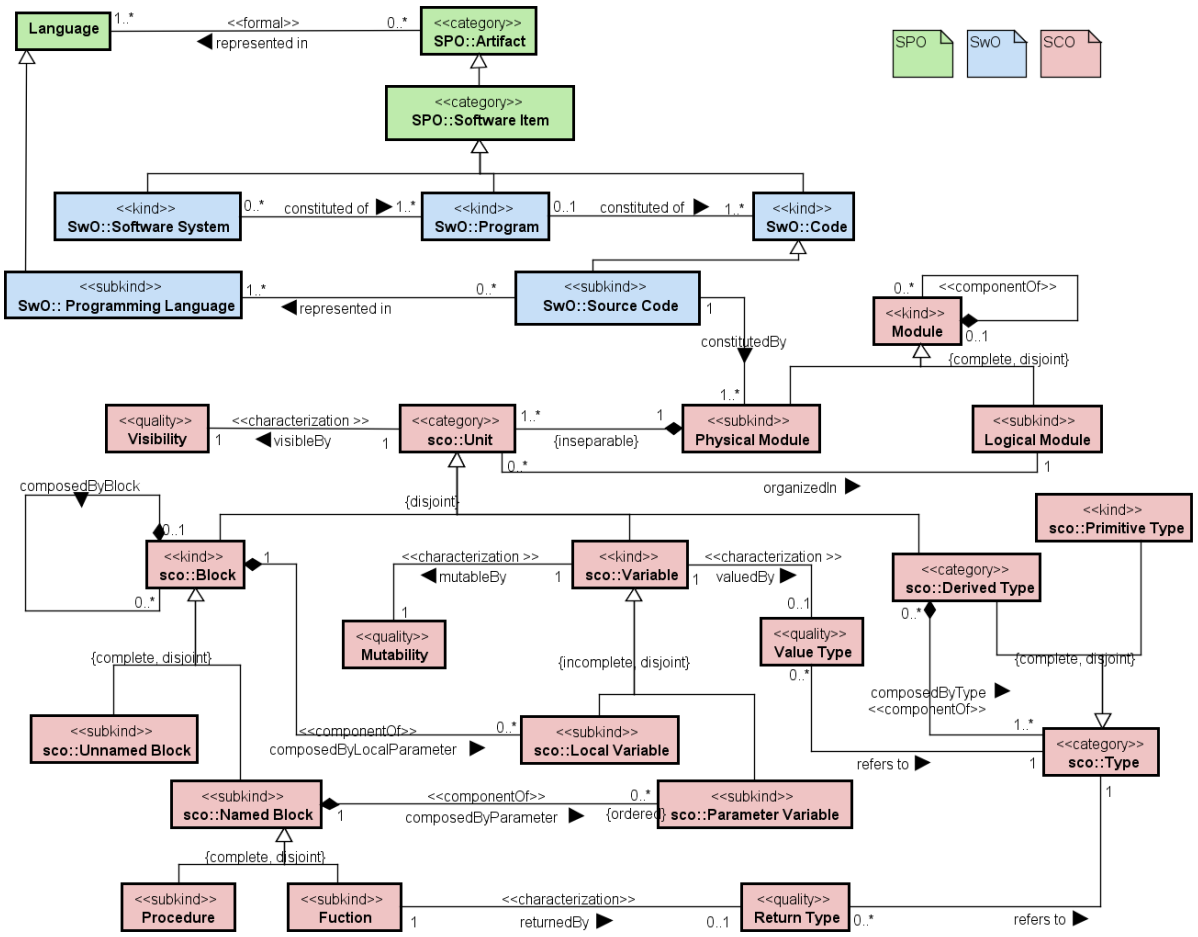


Figura 6 – Ontologia de Referência SCO

A ontologia **SCO** está ancorada no conceito de **Source Code** *represented in* **Programming Language**. Esse código é *constituted of* **Physical Modules**, ou seja, unidades físicas nas quais os arquivos físicos (ex: `.java`) são armazenados (por exemplo, um diretório no sistema de arquivos). Physical Modules são *composed of* **Units** *organized in* **Logical Modules**, ou seja, pacotes ou *namespaces* que agrupam unidades e permitem que os programadores controlem dependências, visibilidade etc. Ambos os **Modules** (Physical ou Logical) podem ser *decomposed in* seus respectivos sub-Modules. No entanto, a decomposição só pode ocorrer entre os módulos do mesmo tipo, ou seja, $\forall m_1, m_2 : Module, PhysicalModule(m_1) \wedge componentOf(m_1, m_2) \rightarrow PhysicalModule(m_2)$ (**A3.1**) e $\forall m_1, m_2 : Module, LogicalModule(m_1) \wedge componentOf(m_1, m_2) \rightarrow LogicalModule(m_2)$ (**A3.2**).

Units são especializadas em **Block**, **Variable** e **Derived Type** que são *characterized by* uma **Visibility**, definindo o seu respectivo tipo de visibilidade.

Um **Block** é uma sequência de instruções definidas por uma dada delimitação, podendo ser *composed by* outros Blocks. Enquanto um **Named Block** é nomeado por um nome identificador, um **Unnamed Block** não tem um nome. Neste contexto, um Named

Tabela 2 – Resultados para verificação de SCO.

RF	**Resposta ao Requisito Funcional (Questão de Competência)**	**Axioma**
RF1	Source Code *constituted of* Physical Module; Unit *component of* Physical Module.	A3.1, A3.2
RF2	Unit *organized in* Logical Module	
RF3	Source Code *constituted of* Physical Module; Unit *component of* Physical Module; + Derived Type *subtype of* Unit; Derived Type *subtype of* Type. + Variable *subtype of* Unit; Variable *valued by* Value Type; Value Type *subtype of* Type. + Block *subtype of* Unit; Named Block *subtype of* Block; Procedure *subtype of* Named Block; Procedure *returned by* Return Type; Return Type *subtype of* Type.	
RF4	Source Code *constituted of* Physical Module; Unit *component of* Physical Module; Variable *subtype of* Unit.	
RF5	Variable *valued by* Value Type.	
RF6	Source Code *constituted of* Physical Module; Unit *component of* Physical Module; Block *subtype of* Unit.	
RF7	Source Code *constituted of* Physical Module; Unit *component of* Physical Module; Block *subtype of* Unit; Named Block *subtype of* Block; Procedure *subtype of* Named Block.	
RF8	Block *subtype of* Unit; Named Block *subtype of* Block; Procedure *subtype of* Named Block; Procedure *returned by* Return Type; Return Type *subtype of* Type.	

Block pode ser uma **Fuction** quando *returned by* **Return Type** ou **Procedure** quando não possui um tipo de retorno.

Variable é uma Unit que mantém um item de informação localizado na memória cujo valor atribuído é *valued by* **Value Type** e pode ser *mutable by* **Mutability** ou não. Variable pode ser uma **Parameter Variable** declarada na assinatura de um Named Block ou **Local Variable** declarada dentro de um Block. Relações *part-of* entre Blocks e Local Variables são transitivas da seguinte maneira: $\forall v : LocalVariable, b_1, b_2 : Block, componentOf(v, b_1) \wedge componentOf(b_1, b_2) \rightarrow componentOf(v, b_2)$ (**A3.3**).

Finalmente, ambos Value Type e Return Type são **Types** de informações que a linguagem é capaz de manipular, seja um **Primitive Type**, predefinido pela linguagem de programação; ou um **Derived Type** *composed by* Type, ou seja, definido a partir de outros tipos.

3.2.1 Verificação de SCO

Para a verificação de ontologia, o método SABiOS sugere identificar se os elementos que compõem a ontologia são capazes de responder às questões de competência levantadas como requisitos funcionais. A Tabela 2 apresenta os resultados para as questões de competência levantadas para a ontologia SCO, mostrando quais conceitos e relações são usados para responder as questões de cada requisito funcional (RF).

3.2.2 Validação de SCO

Para validação de ontologia, a ontologia deve ser instanciada para verificar se ela é capaz de representar situações do mundo real. Para isso, usamos o mesmo fragmento de código escrito nas diferentes linguagens selecionadas para instanciar os conceitos da ontologia. A Tabela 3 mostra alguns resultados da instanciação de SCO.

Tabela 3 – Resultados para validação de SCO.

Código da Linguagem	**Instância de SCO**

C++
`private: int side;`
`public: void perimeter(){};`
	side = Variable
	perimeter = Procedure
	private and public = Visibility
	int = Primitive Type & Value Type
	void = Primitive Type & Return Type

Java
`private int side;`
`public void perimeter(){};`
	side = Variable perimeter = Procedure
	private and public = Visibility
	int = Primitive Type & Value Type
	void = Primitive Type & Return Type

Python
`side = None`
`def perimeter(): ...`
	side = Variable
	perimeter = Procedure

A partir da instanciação de SCO, podemos observar que a **visibility** é explicitamente definida com palavras-chave (`private` e `public` em Java e C++) ou é `public` por padrão (Python). O **value type** é explicitamente definido em algumas linguagens (C++, Java) ou definido pelo valor atribuído (Python).

3.3 OOC-O - Ontologia de Código Orientado a Objetos

A ontologia OOC-O — *Object-Oriented Code Ontology* — visa identificar e representar a semântica das entidades presentes em tempo de compilação no código-fonte orientado a objetos (OO). Programação orientada a objetos (OO) é definida como um método de implementação de software no qual os programas são organizados como coleções cooperativas de **objetos**, cujo objeto representa uma instância de alguma **classe** e cujas classes são membros de uma **hierarquia** de classes ligadas por relacionamentos de **herança**. Uma classe serve como um modelo a partir do qual os objetos podem ser criados, contendo **atributos** e **métodos**. Para que os atributos e métodos de uma classe sejam usados na definição de uma nova classe, a herança é aplicada como um meio de criar **abstrações**.

Abstração é o mecanismo de representar apenas as características essenciais, ignorando os detalhes irrelevantes como forma de ocultar a implementação. Para ocultar dados, o *encapsulamento* aplica um empacotamento de métodos e atributos acessíveis ou modificáveis apenas por meio da interface. Além disso, a abstração pode ser definida por *polimorfismo*, atribuindo a capacidade de assumir várias formas e por *genericidade*, atribuindo a capacidade de assumir vários tipos independentemente da estrutura.

Abstração, encapsulamento, herança e polimorfismo são os princípios básicos da orientação a objetos (CONAWAY; PAGE-JONES; CONSTANTINE, 2000). Em outras palavras, se algum desses elementos estiver faltando, você terá algo menos que uma linguagem OO (BOOCH, 1994). Assim, consideramos uma linguagem de programação OO como uma ferramenta que dá suporte a estes quatro princípios fundamentais: A *abstração* é realizada em um código OO por meio de classes contendo atributos e métodos;

o *encapsulamento* é implementado por métodos de acesso ocultando informações internas da classe de forma a evitar o acesso direto aos seus atributos, e pela visibilidade do elemento evitando o acesso indesejado a esses elementos; a *herança* é representada diretamente como uma relação entre uma subclasse que herda características de uma superclasse; e, finalmente, o *polimorfismo* ocorre por meio dos conceitos de sobreescrita de método, em que uma declaração de método na subclasse modifica o método declarado na superclasse, de classe abstrata, cujos métodos abstratos são implementados de acordo com a subclasse que os herda, e de classe/método genérico, cuja definição pode ser usada por diferentes tipos de dados.

Uma vez que OOC-O faz parte da rede SCON, a ontologia também segue o método SABiOS e a ontologia de fundamentação UFO. Os seguintes requisitos não funcionais (RNF) (apresentados previamente na Seção A.2.3) foram elicitados para OOC-O: **RNF1** – ser modular para facilitar a reutilização por outras ontologias; **RNF2** – ser incorporada a uma rede de ontologias para facilitar a reutilização de outras ontologias e, consequentemente, expandir suas prórpias possibilidades de reuso e associação; **RNF3** – ser definida a partir de fontes de conhecimento reconhecidas na literatura; **RNF4** – ser definida a partir do conhecimento consensual do domínio; **RNF5** – ser fundamentada por uma ontologia de fundamentação a fim de reutilizar os compromissos ontológicos e axiomas já consensualmente estabelecidos.

Em resposta ao **RNF1** e para facilitar a visualização, a ontologia foi dividida em três módulos, a saber: *OOC-O Core*, uma especialização da ontologia SCO com os principais conceitos de orientação a objetos, *OOC-O Class*, detalhamento dos conceitos derivados de *Class*, e *OOC-O Class Member*, detalhamento dos conceitos derivados de *Class Member*, ou seja, *Method* e *Attribute*). Para responder ao **RNF2**, a ontologia é parte da Rede de Ontologia de código-fonte (SCON) a fim de reutilizar conceitos relacionados a código-fonte. Em relação ao **RNF3** e **RNF4**, o desenvolvimento da ontologia foi suportado por um processo de aquisição de conhecimento que utilizou fontes consolidadas de conhecimento referentes a cinco linguagens de programação inseridas no contexto de orientação a objetos. Finalmente, em resposta ao **RNF5**, a ontologia foi baseada na ontologia de fundamentação UFO-A e na aplicação de análise ontológica.

Para requisitos funcionais (RF)(apresentados previamente na Seção A.2.3), definimos iterativamente o seguinte conjunto de questões de competência (CQs): **RF01**- Quais os principais elementos de um código-fonte OO? **RF02**- Quais classes estão presentes em um código OO? **RF03**- Quais elementos compõem uma classe? **RF04**- Qual a herença presente em uma determinada classe? **RF05**- Quais métodos estão presentes em um código OO? **RF06**- Quais elementos compõem um método? **RF07**- Quais os métodos de uma classe? **RF08**- Quais variáveis estão presentes em um código OO? **RF09**- Quais variáveis compõem uma classe? **RF10**- Quais variáveis compõem um método?

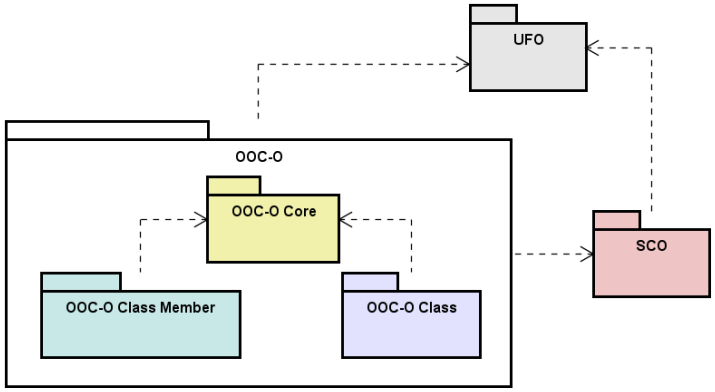


Figura 7 – Módulos da Ontologia OOC-O

Durante a captura e formalização da ontologia, realizamos análises ontológicas baseadas em UFO, representando OOC-O no modelo OntoUML. Esse processo foi conduzido iterativamente, a fim de abordar diferentes aspectos/refinamentos a cada iteração e de maneira interativa, para que especialistas de domínio e engenheiros de ontologia pudessem discutir a conceitualização do domínio. Considerando a quantidade de linguagens de programação existentes, selecionamos linguagens que fornecem construtos para os princípios básicos de OO, a fim de formar a linha de base de nossa pesquisa, a saber: Smalltalk, Eiffel, C++, Java e Python. A seleção considerou duas linguagens de programação relevantes para orientação a objetos e três linguagens OO mais populares de acordo com os índices TIOBE,[4] IEEE Spectrum[5] e Redmonk.[6]

As subseções a seguir apresentam os três módulos de OOC-O: OOC-O Core, OOC-O Class e OOC-O Class Member, apresentados na Figura 7. Uma nova análise ontológica sobre as ontologias de OOC-O está prevista em trabalhos futuros (Seção 8.3).

3.3.1 OOC-O Core

O módulo OOC-O Core é representado na Figura 8 com os conceitos integrados à ontologia SCO.

A ontologia **OOC-O** está ancorada nos conceitos da ontologia **SCO** de modo que **Class** é um subtipo de Derived Type, ou seja, tipo de dado abstrato na linguagem de programação OO e um mecanismo para definir um tipo de dado abstrato em um programa.

Classes são *composed of* **Members**, seja **Method**, um bloco (Function ou Procedure) que pertence à classe e fornece uma maneira de definir o comportamento de um objeto que é invocado quando uma mensagem é recebida pelo objeto (LALONDE; PUGH, 1990); seja **Attribute**, variável que pertence à classe e fornece uma maneira de definir o estado de

[4] <https://www.tiobe.com/tiobe-index/>
[5] <https://spectrum.ieee.org/at-work/tech-careers/top-programming-language-2020.>
[6] <https://redmonk.com/sogrady/2020/07/27/language-rankings-6-20/>

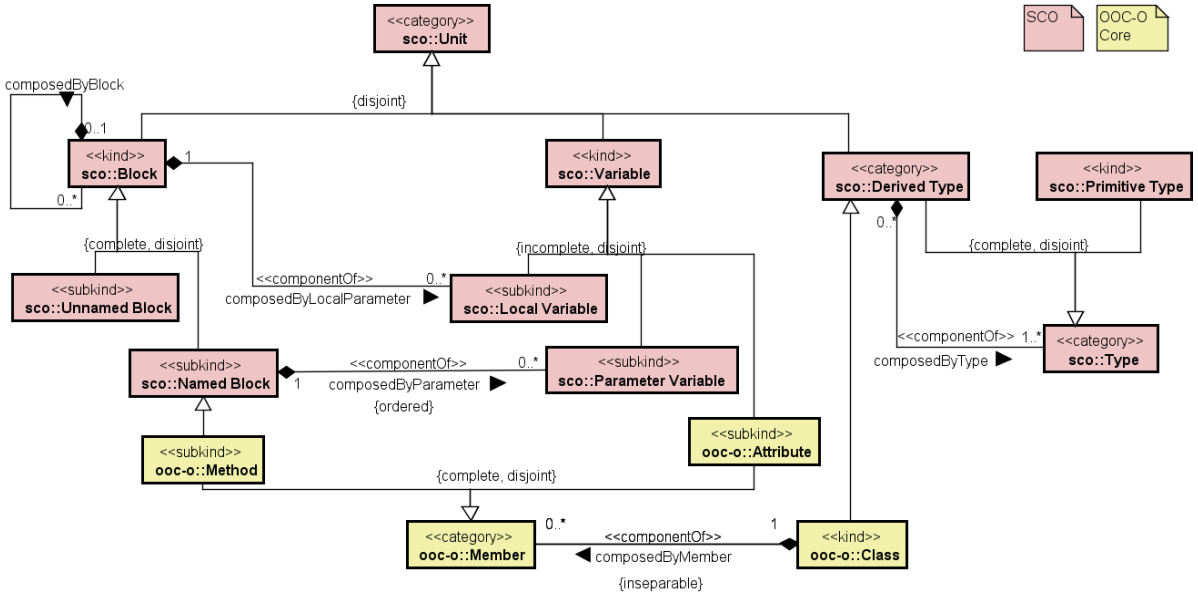


Figura 8 – Ontologia de Referência OOC-O Core

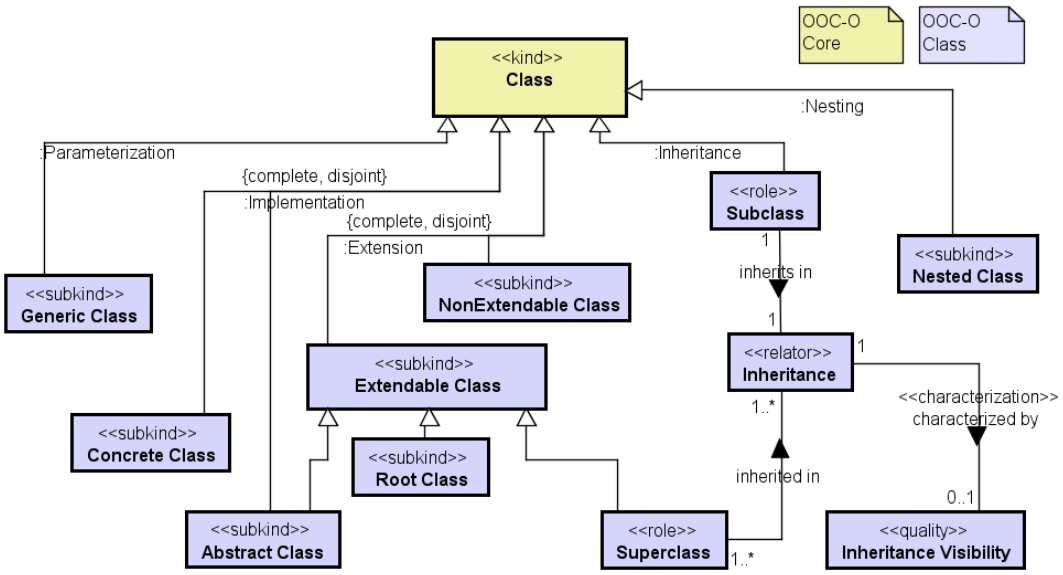


Figura 9 – Ontologia de Referência OOC-O Class

seus objetos. Enquanto Method é subtipo de Named Block, Attribute é subtipo de Variable.

3.3.2 OOC-O Class

O objetivo do módulo OOC-O Class é representar os conceitos relevantes relacionados às classes que estão presentes nas linguagens de programação Orientadas a Objeto. Assim, o módulo OOC-O Class, mostrado na Figura 9, é centrado no conceito Class, já apresentado no módulo OOC-O Core na Seção 3.3.1.

Toda Class deve ser uma **Concrete Class**, classe implementada que pode e pretende ter instâncias, ou uma **Abstract Class**, classe implementada de forma incompleta que será

refinada por seus descendentes. Abstract class, ao contrário de Concrete Class, não deve ter instâncias e deve ser uma Extendable Class. Além disso, toda class deve ser ou uma **Extendable Class**, classe disponível para ser estendida através de Inheritance, ou uma **NonExtendable Class**, o oposto.

Uma Extendable Class pode assumir o papel de **Superclass** quando se relaciona com uma Class que assume o papel de **Subclass** em um relacionamento de **Inheritance**: $\forall c_1, c_2 : Class, i : Inheritance, inheritsIn(c_1, i) \wedge inheritedFrom(c_2, i) \rightarrow subClassOf(c_1, c_2)$ (**A3.4**). A relação entre uma Superclass e uma Subclass é estabelecida principalmente pela existencia de uma relação "is-a" entre elas (TUCKER, 2007).

Neste contexto, **Inheritance Visibility** pode ser configurada para limitar a permissão da Subclass sobre os membros da Superclass. A Extendable Class herdada por todas as classes direta ou indiretamente em um código OO é conhecida como **Root Class** (EIFFEL, 2006) e introduz vários recursos de propósito geral. Quando presente, a Root Class é um ancestral comum para todas as outras classes existentes, ou seja, $\forall c : Class, r : RootClass, c \neq r \rightarrow descendantOf(c, r)$ (**A3.5**), onde $descendantOf$ é definido em termos do predicado $subClassOf$, de acordo com os seguintes axiomas: $\forall c_1, c_2 : Class, subclassOf(c_1, c_2) \rightarrow descendantOf(c_1, c_2)$ (**A3.6**) e $\forall c_1, c_2, c_3 : Class, subclassOf(c_1, c_2) \wedge descendantOf(c_2, c_3) \rightarrow descendantOf(c_1, c_3)$ (**A3.7**).

Finalmente, uma Class pode se relacionar com outra Class por meio de sua declaração que está dentro do corpo daquela Class (GOSLING et al., 2018), denominada **Nested Class**. Além disso, uma Class pode ser uma **Generic Class**, quando descreve um modelo para um possível conjunto de tipos (EIFFEL, 2006).

3.3.3 OOC-O Class Member

O objetivo do módulo OOC-O Class Member é representar os conceitos relevantes presentes nas linguagens de programação OO no que diz respeito aos membros componentes das classes. Como métodos e atributos são os principais componentes de uma classe, o módulo OOC-O Class Member, mostrado na Figura 10, é centrado nos conceitos de Method e Attribute, já apresentados no módulo OOC-O Core na Seção 3.3.1.

Todo Method de uma Class deve ser ou um **Concrete Method**, implementado em sua própria Class (concreta ou abstrata) por meio de Blocks; ou um **Abstract Method**, pertencente a uma Abstract Class e implementado (ou "concretizado") apenas em suas Subclasses.

Um Concrete Method pode ser especializado de acordo com o seu contexto de execução, seja no contexto da classe, invocado pela classe em um **Class Method**, ou no contexto do objeto, invocado pelo objeto em um **Instance Method**. Um Instance Method pode ser especializado em **Accessor Method**, que fornece uma interface entre os dados internos do

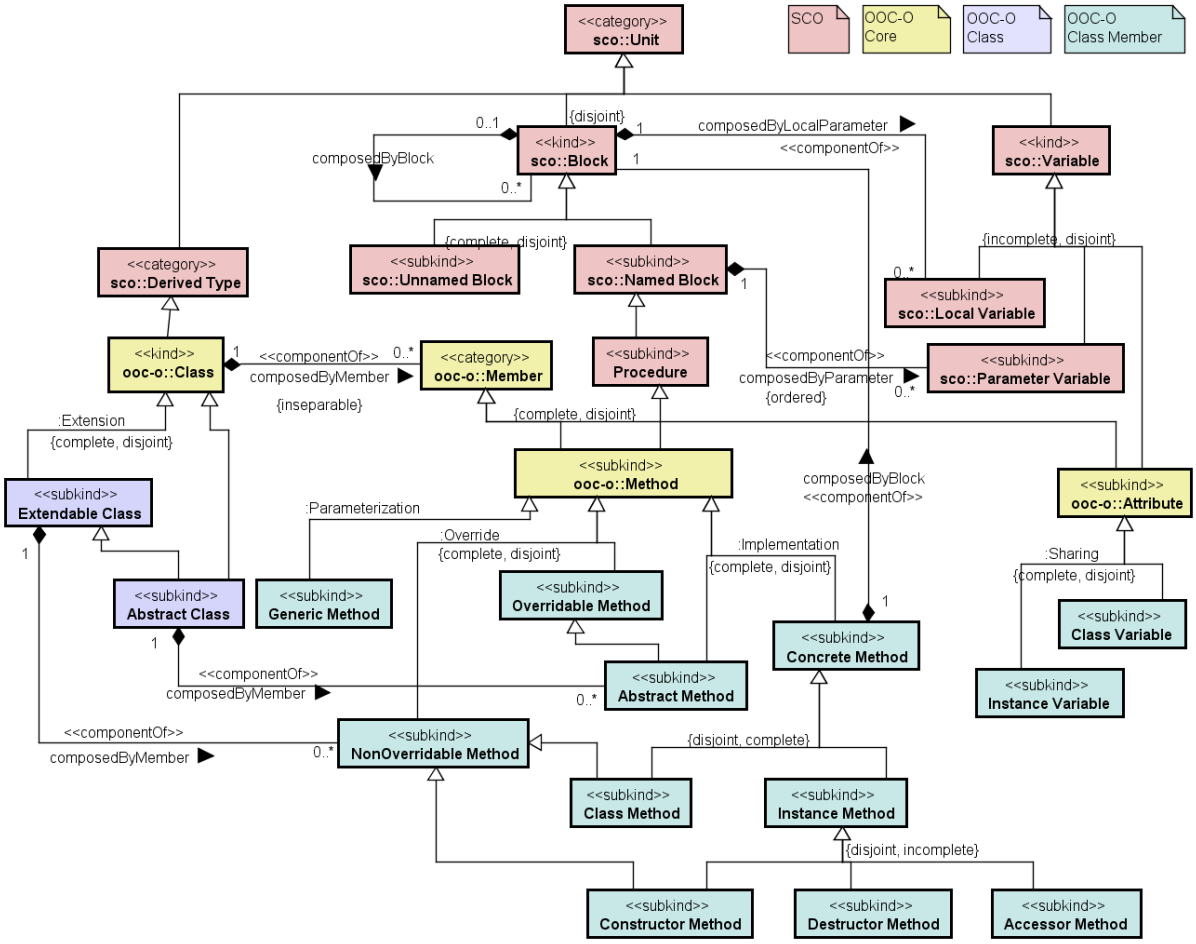


Figura 10 – Ontologia de Referência OOC-O Class Member

objeto e o mundo externo (HUNT, 2002), em **Constructor Method**, que especifica como um objeto deve ser criado e inicializado, ou em **Destructor Method**, que é responsável pela limpeza dos objetos inutilizáveis. Return Type não pode caracterizar nem um Constructor nem um Destructor Method: $\forall rt : ReturnType, m : Method, characterization(rt, m) \rightarrow \neg ConstructorMethod(m) \wedge \neg DestructorMethod(m)$ (**A3.8**).

Além disso, todo Method deve ser ou um **Overridable Method**, método pertencente a uma Extendable Class que pode ser substituído em classes descendentes (SEBESTA, 2012), como um Abstract Method declarado em uma Abstract Class para ser implementado por uma Subclasse; ou um **NonOverridable Method**, método que pode ser herdado mas não pode ser sobrescrito em classes descendentes, como Class Methods e Constructor Methods. Um Method também pode ser um **Generic Method** ao descrever um modelo para um possível conjunto de métodos.

Por sua vez, uma Attribute pode ser uma **Class Variable** quando compartilhada por todos os objetos da Class ou uma **Instance Variable** quando representa o estado particular de cada objeto. Relações *part-of* entre Concrete Methods, Blocks e Local Variables são transitivas de modo que: $\forall v : LocalVariable, b : Block, m : ConcreteMethod, componentOf(v, b) \wedge$

Tabela 4 – Resultados para verificação de OOC-O.

RF	**Resposta ao Requisito Funcional (Questão de Competência)**	**Axioma**
RF1	Object-Oriented Source Code *constituted of* Physical Module; Class *component of* Physical Module.	
RF2	Named Element *characterized by* Element Visibility	
RF3	Class *organized in* Logical Module	
RF4	Member *component of* Class; Attribute and Method *subtype of* Member	
RF5	Subclass *subtype of* Class; Subclass *inherits in* Inheritance; Superclass *inherited in* Inheritance	A3.4, A3.6, A3.7
RF6	Extendable Class *subtype of* Class; Root Class *subtype of* Extendable Class; Subclass *subtype of* Class; Subclass *inherits in* Inheritance; Superclass *inherited in* Inheritance.	A3.4, A3.5, A3.6, A3.7
RF7	Parameter Variable *component of* Method; Local Variable *component of* Block; Block *component of* Block; Block *component of* Concrete Method; Concrete Method *subtype of* Method	A3.3, A3.9
RF8	Variable *characterized by* Mutability	
RF9	Generic Class *subtype of* Class; Concrete Class *subtype of* Class; Abstract Class *subtype of* Class; Non-Extendable Class *subtype of* Class; Extendable Class *subtype of* Class	
RF10	Generic Method *subtype of* Method; Concrete Method *subtype of* Method; Abstract Method *subtype of* Method; Overridable Method *subtype of* Method; Non-Overridable Method *subtype of* Method	

$componentOf(b, m) \rightarrow componentOf(v, m)$ (**A3.9**).

3.3.4 Verificação de OOC-O

Para a verificação de ontologia, o método SABiOS sugere identificar se os elementos que compõem a ontologia são capazes de responder às questões de competência levantadas como requisitos funcionais. A Tabela 4 apresenta os resultados para algumas das questões de competência levantadas para a ontologia OOC-O, mostrando quais conceitos e relações são usados para responder as questões de cada requisito funcional (RF).

3.3.5 Validação de OOC-O

Para validação de ontologia, a ontologia deve ser instanciada para verificar se ela é capaz de representar situações do mundo real. Para isso, usamos o mesmo fragmento de código OO escrito nas diferentes linguagens selecionadas para instanciar os conceitos da ontologia. A Tabela 5 mostra alguns resultados da instanciação de OOC-O. Uma vez que existem conjuntos de generalizações ortogonais que são disjuntos e completos (por exemplo, :Implementation e :Extension no conceito Class), cada instância do conceito (por exemplo, classe Polygon) é classificada em pelo menos cada um destes conjuntos de generalização (por exemplo, Concrete Class ou Abstract Class, e Extendable Class ou Non-Extendable Class).

A partir da instanciação de OOC-O, podemos ver que o código relativo à definição de classe incorpora a semântica de **Concrete Class** e **Extendable Class** na ontologia. A maioria das linguagens, explícita (Smalltalk) ou implicitamente (Eiffel, Java e Python), incorpora a semântica de **Subclass**, uma vez que todas as classes são subclasses de **Root Class** dessas linguagens, como a classe `Object` em Smalltalk, Java e Python, e classe `Any`

Tabela 5 – Resultados para validação de OOC-O.

Código da Linguagem	**Instância de OOC-O**

Smalltalk
`Object subclass: #Polygon`
`  instanceVariableNames: 'side'`
`  perimeter ...`
	Polygon = Concrete Class & Extendable Class & Subclass
	Object = Superclass & Root Class
	side = Instance Variable
	perimeter = Instance Method & Overridable Method

Eiffel
`class Polygon`
`feature{ANY}`
`  perimeter() is  do ... end`
`feature{NONE}`
`  side : INTEGER`
`end`
	Polygon = Concrete Class & Extendable Class & Subclass
	side = Instance Variable
	INTEGER = Value Type
	perimeter = Instance Method & Non-Overridable Method
	NONE and ANY = Visibility

C++
`class Polygon{`
`  private: int side;`
`  public: void perimeter(){};`
`};`
	Polygon = Concrete Class & Extendable Class
	side = Instance Variable
	perimeter = Instance Method
	private and public = Visibility
	int = Primitive Type & Value Type
	void = Primitive Type & Return Type

Java
`public class Polygon{`
`  private int side;`
`  public void perimeter(){};`
`}`
	Polygon = Concrete Class & Extendable Class & Subclass
	side = Instance Variable & Overridable Method
	perimeter = Instance Method
	private and public = Visibility
	int = Primitive Type & Value Type
	void = Primitive Type & Return Type

Python
`class Polygon:`
`  side = None`
`  def perimeter(): ...`
	Polygon = Concrete Class & Extendable Class & Subclass
	side = Instance Variable
	perimeter = Concrete Method & Overridable Method

em Eiffel (C++ não tem uma classe raiz). O código relativo à definição de método em linguagens diferentes incorpora uma semântica altamente mutável, incluindo a semântica de métodos de **instance**, **concrete**, **overridable** e **non-overridable** na ontologia.

Ademais, realizamos uma harmonização entre os elementos das linguagens selecionadas e os conceitos de OOC-O, aplicando relações de equivalência. A Tabela 6 mostra algumas dessas correspondências. Embora os princípios de orientação a objetos estejam bem estabelecidos, a maneira como são tratados nas linguagens de programação não é uniforme. Cada linguagem adota sintaxe e semântica diferentes para seus construtos, resultando em níveis diferentes nos quais esses princípios são tratados. Neste contexto, OOC-O pode ser usada para apoiar a interoperabilidade entre eles.

Portanto, **Abstração** é representada pelo conceito de class nas linguagens, sendo composta por membros como method em Smalltalk, Java e Python, ou routine em Eiffel, ou member function em C++, e por attribute em Eiffel, ou data attribute em Python, ou instance variable em Smalltalk, C++ e Java. A **Herança** é representada por subclass em Smalltalk, Eiffel, Java e Python, ou derived class em C++, e por superclass em Smalltalk, Eiffel, Java e Python, ou base class em C++. O **Encapsulamento** é representado por access em Smalltalk e Eiffel, ou access modifier em C++ e Java, e pela public visibility em Python. O encapsulamento é representado também pelo accessor method em Smalltalk, no entanto, o conceito de accessor method em Eiffel, C++, Java e Python não é equivalente ao método acessor da ontologia porque nessas linguagens há apenas uma convenção para tratar um método de instância como um método acessor. O **Polimorfismo** é representado

Tabela 6 – Equivalência entre as linguagens de programação OO selecionadas e OOC-O.

Ling.	**Conceito da Linguagem**	**Conceito de OOC-O**
Smalltalk	Class	Concrete Class & Extendable Class
	Abstract Class	Abstract Class
	Template	Generic Class
	Method	Concrete Method & Overridable Method
	Accessor Method	Accessor Method
	Instance Variable	Instance Variable
	Access	Visibility
Eiffel	Class	Concrete Class & Extendable Class
	Deferred Class	Abstract Class
	Frozen Class	NonExtendable Class
	Generic Class	Generic Class
	Routine	Instance Method & NonOverridable Method
	Routine Redefinition	Overridable Method
	Accessor Routine	Instance Method
	Attribute	Instance Variable
	Access	Visibility
C++	Class	Concrete Class & Extendable Class
	Abstract Class	Abstract Class
	Final Class	NonExtendable Class
	Template	Generic Class
	Member Function	Instance Method
	Final Member Function	NonOverridable Method
	Virtual Member Function	Overridable Method
	Accessor Member Function	Instance Method
	Instance Variable	Instance Variable
	Access Modifier	Visibility
Java	Class	Concrete Class & Extendable Class
	Abstract Class	Abstract Class
	Final Class	NonExtendable Class
	Generic Class	Generic Class
	Method	Instance Method & Overridable Method
	Abstract Method	Abstract Method
	Final Method	NonOverridable Method
	Accessor Method	Instance Method
	Instance Variable	Instance Variable
	Access Modifier	Visibility
Python	Class	Concrete Class & Extendable Class
	Abstract Class	Abstract Class
	Generic Class	Generic Class
	Method	Concrete Method & Overridable Method
	Accessor Method	Instance Method
	Data Attribute	Instance Variable

pela routine redefinition em Eiffel ou virtual function em C++. Smalltalk, Java e Python incorpora a semântica de overridable method ao conceito de method. O polimorfismo é representado também por generic class/method em Eiffel, Java e Python, ou template em Smalltalk e C++. O polimorfismo também é representado por abstract class em Smalltalk, C++, Java e Python, ou deferred class em Eiffel.

3.4 DBS-O - Ontologia de Banco de Dados

Os sistemas de banco de dados (DBSs) são um componente essencial da vida na sociedade moderna, uma vez que muitas de nossas atividades envolvem algum programa de computador acessando uma base de dados (por exemplo, comprar algo em uma loja, usar uma conta bancária, etc.). Uma base de dados é uma coleção de dados logicamente relacionados que possuem algum significado, acessados por meio de um conjunto de

programas que constituem um sistema de gerenciamento de banco de dados (DBMS). Um DBMS é um sistema de software que facilita os processos de definição, especificando os tipos de dados, estruturas e restrições de dados a serem armazenados; manipulação, execução de consultas para recuperar e alterar os dados armazenados; e compartilhamento, permitindo acesso simultâneo a base de dados (ELMASRI; NAVATHE, 2011).

Essa representação fornece independência de dados por meio da abstração de dados, de forma que as mudanças no nível físico não sejam propagadas para o nível conceitual e vice-versa. Além disso, os DBSs permitem que os dados sejam percebidos em diferentes níveis de detalhes a partir dos modelos de dados. Esse modelo descreve a estrutura da base de dados como tipos de dados, relacionamentos e restrições que se aplicam aos dados (ELMASRI; NAVATHE, 2011).

A ontologia DBS-O — *Database System Ontology* — visa representar os conceitos relevantes para o domínio de sistema de banco de dados. No entanto, dada a extensão deste domínio, esta ontologia requer estudos adicionais para avaliar adequadamente sua generalização. A construção da ontologia seguiu o método SABiOS, apresentado na Seção 5.2 e detalhado no Apêndice A.

Os seguintes Requisitos Não Funcionais (RNF) foram elicitados para a DBS-O: **RNF1** – ser compreensível e extensível, principalmente porque a ontologia será usada para diferentes fins; **RNF2** – ter uma versão operacional da ontologia de referência que pode ser usada em aplicações; **RNF3** – ser modular ou incorporada em uma estrutura modular para facilitar a reutilização de outras ontologias e subsequente reutilização dessa ontologia; **RNF4** – ser baseada em fontes de dados bem conhecidas da literatura.

Os Requisitos Funcionais (RF) são definidos nas seguintes questões de competência: **RF01:** O que é o DBMS de um sistema de banco de dados? **RF02:** Quais são as linguagens usadas por um sistema de banco de dados? **RF03:** O que é o banco de dados de um sistema de banco de dados? **RF04:** Quais são os esquemas de um banco de dados? **RF05:** Qual é o esquema padrão de um banco de dados? **RF06:** Quais são os arquivos de um banco de dados? A ontologia DBS-O é representada na Figura 11 e apresenta como os conceitos de DBS-O estão integrados às ontologias SwO e SPO. Uma nova análise ontológica está prevista em trabalhos futuros (Seção 8.3).

Um **Database System** (DBS) é um **Computer System** cujo objetivo principal é armazenar informações e permitir que os usuários busquem e atualizem essas informações quando solicitadas (DATE, 2004). A vantagem de um DBS é a independência dos dados, ou seja, a estrutura do arquivo é armazenada no **DBMS** (Database Management System), um **Software System** que garante que qualquer alteração na representação física dos dados não afete drasticamente os programas que os utilizam (ELMASRI; NAVATHE, 2011). O DBMS adota uma **Data Language**, uma **Language** declarativa que descreve o problema ao invés da solução e especifica o que deve ser feito (mas não como), para manipular os

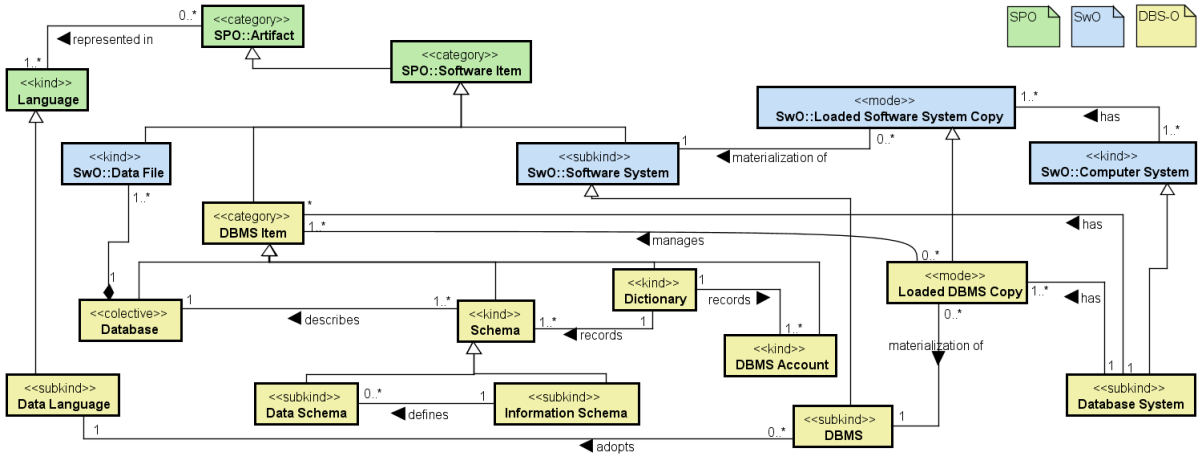


Figura 11 – Ontologia de Referência DBS-O

dados em um **Database**.

O Database System possui uma ou mais **Loaded DBMS Copies** inerentes a uma máquina de computador como parte do Database System. Cada **Loaded DBMS Copy** gerencia **DBMS Itens**, como Database e Dictionary. Um Database é uma coleção de **Data Files** organizados para que seu conteúdo possa ser facilmente acessado e gerenciado (DATE, 2004). Um Dictionary descreve itens de interesse do sistema (DATE, 2004), ou seja, registra a estrutura primária do Database (ELMASRI; NAVATHE, 2011), como **Schemas** e **DBMS Accounts**. Cada Dictionary deve registrar pelo menos uma DBMS Account, usuário físico ou computacional que faz solicitações ao banco de dados, e qualquer número de Schemas, uma coleção de descritores nomeados persistentes (ISO/IEC9075-1, 2008). Existe exatamente um **Informational Schema** que define de forma efetiva e precisa todas as configurações de todos os outros `Schemas` (ditos **Data Schemas**) no Dictionary (DATE, 2004). Coletivamente, os Schemas descrevem o Database.

3.5 RDBS-O - Ontologia de Banco de Dados Relacional

O modelo de dados representacional mais amplamente usado em DBMSs comerciais é o modelo relacional, a base da tecnologia de banco de dados relacional (DATE, 2004). O modelo relacional usa o conceito de relação matemática como estrutura básica e tem sua base teórica na teoria dos conjuntos e na lógica de predicados de primeira ordem (ELMASRI; NAVATHE, 2011). No modelo, as relações são usadas para representar os dados e os relacionamentos entre os dados, as tuplas representam os fatos que normalmente correspondem a entidades ou relacionamentos do mundo real e os atributos especificam como interpretar os valores dos dados em cada tupla adotando um tipo único. Em um banco de dados relacional, as relações são percebidas como tabelas, tuplas como linhas e atributos como colunas. A coleta de dados armazenados no banco de dados em um

determinado momento é chamada de instância (ELMASRI; NAVATHE, 2011).

A ontologia RDBS-O — *Relational Database System Ontology* — especializa os conceitos de DBS-O para o domínio de bancos de dados relacionais. A construção da ontologia seguiu o método SABiOS, apresentado na Seção 5.2 e detalhado no Apêndice A.

Os seguintes Requisitos Não Funcionais (RNF) foram elicitados para RDBS-O: **RNF1** – ser compreensível e extensível, principalmente porque a ontologia será usada para diferentes fins; **RNF2** – ter uma versão operacional da ontologia de referência que pode ser usada em aplicações; **RNF3** – ser modular ou incorporada em uma estrutura modular para facilitar a reutilização de outras ontologias e subsequente reutilização dessa ontologia; **RNF4** – ser baseada em fontes de dados bem conhecidas da literatura. Seus requisitos funcionais, ou seja, o conhecimento que a ontologia deve representar, são apresentados nas subseções a seguir.

Os Requisitos Funcionais (RF) são definidos nas seguintes questões de competência: **RF01:** Quais são as tabelas de um sistema de banco de dados? **RF02:** Quais dados uma tabela contém? **RF03:** Quais são as colunas de uma tabela? **RF04:** Qual é o tipo de dados de uma coluna? **RF05:** Qual é a chave primária de uma tabela? **RF06:** Quais colunas se referem a uma chave primária? **RF07:** Qual é a chave estrangeira de uma tabela? **RF08:** Quais colunas se referem a uma chave estrangeira? **RF09:** Quais tabelas são relacionadas por meio de uma chave estrangeira? **RF10:** Qual restrição especifica um tipo de dados? **RF11:** Quais restrições especificam uma coluna? **RF12:** Quais são as tabelas básicas de um sistema de banco de dados? **RF13:** Quais são as tabelas derivadas de um sistema de banco de dados? A ontologia RDBS-O é representada na Figura 12 e apresenta como os conceitos de RDBS-O estão integrados à ontologia DBS-O. Uma nova análise ontológica está prevista em trabalhos futuros (Seção 8.3).

O **Relational Database Management System** (RDBMS) é uma especialização de um **Database Management System** (DBMS) cuja abstração de dados é baseada no modelo relacional (DATE, 2004) e, portanto, seu principal **RDBMS Item** é a **Table**. Em um RDBMS, um **Data File** é representado como Tables, definidas por um **Line Type** que é instanciado como **Lines**, que são proposições verdadeiras. Um Line Type é o tipo mais específico de uma Line (ISO/IEC9075-2, 2003), de modo que todas as linhas em uma tabela têm um tipo de linha exclusivo (ISO/IEC9075-1, 2008). Cada Line Type é constituído por um conjunto de **Columns** que representam um campo de tabela (ISO/IEC9075-1, 2008). Em um RDBMS, Line é a menor unidade de dados que pode ser inserida e excluída de uma Table (ISO/IEC9075-2, 2003). Uma Table é uma **Base Table**, que representa os dados armazenados no banco de dados de forma autônoma e independente, como tabelas persistentes, temporárias globais ou temporárias locais (MELTON; SIMON, 2001); ou uma **Derived Table**, tabela não base que pode ser obtida por meio de expressão relacional em uma ou mais Base Table de uma forma não autônoma e dependente (DATE, 2004)

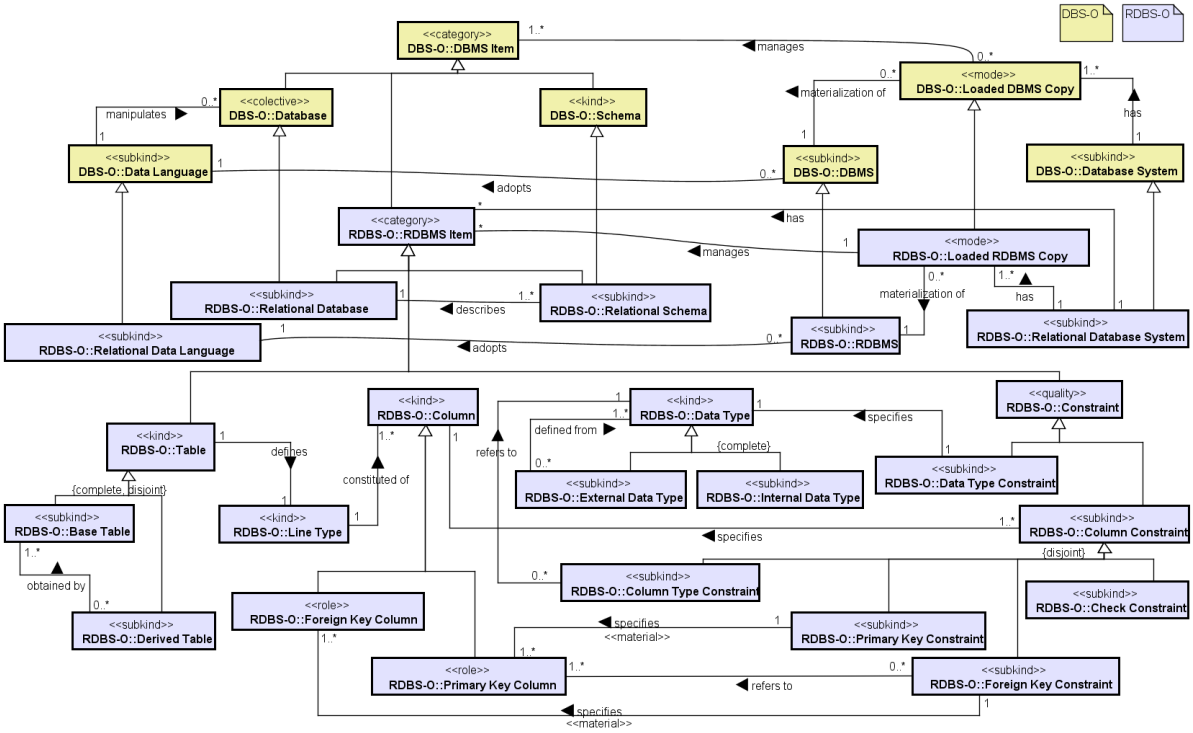


Figura 12 – Ontologia de Referência RDBS-O

(por exemplo, uma *view*).

Uma Column é especificada por **Column Constraints**, sendo exatamente uma **Column Type Constraint**, a fim de restringir os valores que uma Column pode assumir (DATE, 2004) com relação a um **Data Type** (ISO/IEC9075-2, 2003; MELTON; SIMON, 2001; ABBEY; COREY; ABRAMSON, 2002), que por sua vez é um conjunto de valores representáveis especificando o tipo de informação mantida em uma Column. O Data Type pode ser um **Internal Data Type**, definido pelo RDBMS, ou um **External Data Type**, definido pelo usuário com base nos Data Types existentes (DATE, 2004). Além disso, um Data Type T é especificado por uma **Data Type Constraint** que define o conjunto de valores válidos para T. No entanto, uma Column Type Constraint nunca pode ser violada se as Data Type Constraints forem verificadas (DATE, 2004).

Além disso, uma Column Constraint pode ser uma **Check Constraint**, uma **Primary Key Constraint** ou uma **Foreign Key Constraint**. Uma Check Constraint especifica uma condição que deve ser satisfeita para qualquer Line da Table (ISO/IEC9075-2, 2003), como uma condição de valores, um valor nulo, um valor especial usado para indicar a ausência de qualquer valor de dados em uma coluna, um valor único (ou seja, uma coluna não deve ter dois valores não nulos iguais), um valor padrão, etc. Uma Primary Key Constraint é uma restrição de integridade que satisfaz as propriedades de exclusividade e irredutibilidade, ou seja, define o que torna uma linha de dados exclusiva dentro de uma tabela (DATE, 2004; ABBEY; COREY; ABRAMSON, 2002). É a combinação das Check Constraints de valor único e nulo; no entanto, uma Table pode ter no máximo uma Primary Key Constraint e

Tabela 7 – Resultados para verificação RDBS-O.

RF	**Resposta ao Requisito Funcional**
RF01	Loaded RDBMS Copy *materialization of* RDBMS and Loaded RDBMS Copy *manages* RDBMS Item e RDBMS Item *specialized in* Table
RF02	Table *defines* Line Type e Line *instance of* Line Type
RF03	Table *defines* Line Type e Line Type *constituted of* Column
RF04	Column Constraint *specifies* Column; Column Constraint *specialized in* Column Type Constraint e Column Type Constraint *refers to* Data Type.
RF05	Primary Key Constraint *specializes* Column Constraint; Column Constraint *specifies* Column; Table *defines* Line Type e Line Type *constituted of* Column
RF06	Primary Key Constraint *specifies* Primary Key Column e Primary Key Column *specialized of* Column
RF07	Foreign Key Constraint *specializes* Column Constraint; Column Constraint *specifies* Column; Table *defines* Line Type e Line Type *constituted of* Column
RF08	Foreign Key Constraint *specifies* Foreign Key Column e Foreign Key Column *specialized of* Column
RF09	Foreign Key Constraint *specializes* Foreign Key Column; Foreign Key Column *specialized of* Column; Table *defines* Line Type e Line Type *constituted of* Column Foreign Key Constraint *refers to* Primary Key Column; Primary Key Column *specialized of* Column; Table *defines* Line Type e Line Type *constituted of* Column
RF010	Data Type Constraint *specifies* Data Type Data Type Constraint *specifies* Data Type e Data Type *specialized in* Internal Data Type Data Type Constraint *specifies* Data Type e Data Type *specialized in* External Data Type
RF011	Column Constraint *specifies* Column e Column Constraint *specializes* Column Type Constraint, Primary Key Constraint, Foreign Key Constraint e Check Constraint
RF012	Loaded RDBMS Copy *materialization of* RDBMS e Loaded RDBMS Copy *manages* RDBMS Item e RDBMS Item *specialized in* Table e Table *specializes* Base Table
RF013	Loaded RDBMS Copy *materialization of* RDBMS e Loaded RDBMS Copy *manages* RDBMS Item e RDBMS Item *specialized in* Table e Table *specializes* Derived Table

qualquer número de Check Constraints (DATE, 2004). Uma Foreign Key Constraint é uma restrição referencial que especifica uma ou mais Columns de uma tabela de referência que correspondem às Columns de uma tabela referenciada (ISO/IEC9075-2, 2003).

Assim, uma Column pode assumir o papel de **Primary Key Column** — que identifica um valor não nulo exclusivo para qualquer instância de tabela quando associada a uma Primary Key Constraint — ou o papel de **Foreign Key Column** — que pertence a uma tabela de referência e cujos valores correspondem aos de uma Primary Key Column de alguma Table associada a uma Foreign Key Constraint (DATE, 2004).

3.5.1 Verificação de RDBS-O

Para a verificação de ontologia, o método SABiOS sugere identificar se os elementos que compõem a ontologia são capazes de responder às questões de competência levantadas. A Tabela 7 apresenta os resultados para algumas das questões de competência levantadas para a ontologia DBS-O, mostrando quais conceitos e relações são usados para responder cada questão.

3.5.2 Validação de RDBS-O

Para validação de ontologia, a ontologia de referência deve ser instanciada para verificar se ela é capaz de representar situações do mundo real, ou seja, a estrutura de um banco de dados relacional. Esta atividade foi apoiada por uma abordagem semiautomática para construção e publicação de ontologias a partir de bancos de dados relacionais.

Tabela 8 – Resultados da instanciação de RDBS-O usando HR Database.

Conceito	**Instâncias de RDBS-O**
Table	**Instância:** COUNTRIES
	Dump: <rdf:Description rdf:about="#TABLE/COUNTRIES» <rdbs-owl:HASCOLUMN rdf:resource="#COLUMN/COUNTRIES/COUNTRIES_ID"/> <rdbs-owl:HASCOLUMN rdf:resource="#COLUMN/COUNTRIES/COUNTRY_NAME"/> <rdbs-owl:HASCOLUMN rdf:resource="#COLUMN/COUNTRIES/REGION_ID"/> <rdfs:label>#COUNTRIES</rdfs:label> <rdf:type rdf:resource="http://rdbsowl/TABLE"/> </rdf:Description>
Column	**Instância:** COUNTRY_ID
	Dump:<rdf:Description rdf:about="#COLUMN/COUNTRIES/COUNTRY_ID» <rdbs-owl:BELONGSTOTABLE rdf:resource="#TABLE/COUNTRIES"/> <rdbs-owl:DEFINEDBYDATATYPE>#DATATYPE/CHAR</rdbo:DEFINEDBYDATATYPE> <rdfs:label>COLUMN #CONTRY_ID</rdfs:label> <rdf:type rdf:resource="http://rdbs-owl/COLUMN"/> </rdf:Description>
Primary key Column	**Instância:** COUNTRY_ID
	Dump:<rdf:Description rdf:about="#PRIMARYKEYCOLUMN/COUNTRIES/COUNTRY_ID» <rdbs-owl:BELONGSTOTABLE rdf:resource="#TABLE/COUNTRIES"/> <rdfs:label>PRIMARYKEYCOLUMN COUNTRIES COUNTRY_ID</rdfs:label> <rdf:type rdf:resource="http://rdbs-owl/PRIMARYKEYCOLUMN"/> <rdbs-owl:REFERESTOCOLUMN rdf:resource="#COLUMN/COUNTRIES/COUNTRY_ID"/> </rdf:Description>
Foreign key Column	**Instância:** REGION_ID
	Dump:<rdf:Description rdf:about="#FOREIGNKEYCOLUMN/COUNTRIES/REGION_ID» <rdbs-owl:REFERSTOCOLUMN rdf:resource="#COLUMN/COUNTRIES/REGION_ID"/> <rdbs-owl:REFERSTOPRIMARYKEYCOLUMN rdf:resource="#PRIMARYKEYCOLUMN/REGIONS/ REGION_ID"/> <rdbs-owl:BELONGSTOREFERENCINGTABLE rdf:resource="#REFERENCING-TABLE/COUNTRIES"/> <rdbs-owl:REFERSTOREFERENCEDTABLE rdf:resource="#REFEREN-CEDTABLE/REGIONS"/> <rdfs:label>FOREIGNKEYCOLUMN COUNTRIES REGION_ID</rdfs:label> <rdf:type rdf:resource="http://rdbs-owl/FOREIGNKEYCOLUMN"/> </rdf:Description>
Primary key Constraint	**Instância:** PK_COUNTRY
	Dump:<rdf:Description rdf:about="#PRIMARYKEYCONSTRAINT/PK_COUNTRY» <rdbs-owl:DEFINESPRIMARYKEYCOLUMN rdf:resource="#PRIMARYKEYCOLUMN/COUNTRIES/ COUNTRY_ID"/> <rdbs-owl:SPECIFIESTABLE rdf:resource="#TABLE/COUNTRIES"/> <rdfs:label>PRIMARYKEYCONSTRAINT PK_COUNTRY</rdfs:label> <rdf:type rdf:resource="http://rdbs-owl/PRIMARYKEYCONSTRAINT"/> </rdf:Description>
Foreign key Constraint	**Instância:** FK_COUNTRY_REGION
	Dump:<rdf:Description rdf:about="#FOREIGNKEYCONSTRAINT/FK_COUNTRY_REGION» <rdbs-owl:DEFINESFOREIGNKEYCOLUMN rdf:resource="#FOREIGNKEYCOLUMN/COUNTRIES/ REGION_ID"/> <rdbs-owl:REFERESTOPRIMARYKEYCONSTRAINT rdf:resource="#PRIMARY-KEYCONSTRAINT/PK_REGION"/> <rdbs-owl:SPECIFIESTABLE rdf:resource="#TABLE/ COUNTRIES"/> <rdfs:label>FOREIGNKEYCONSTRAINT FK_COUNTRY_REGION</rdfs:label> <rdf:type rdf:resource="http://rdbs-owl/FOREIGNKEYCONSTRAINT"/> </rdf:Description>

As atividades foram realizadas usando um `sample` criado pela Oracle, chamado de **HR Database**, um esquema de aplicação de Recursos Humanos cujo objetivo principal é armazenar os registros dos funcionários de uma organização. Assim, um **Dump RDF** foi gerado, descrevendo as informações extraídas conforme os conceitos definidos na ontologia, como uma instância de RDBS-O, com informações reais de um banco de dados relacional. A Tabela 8 descreve parte da validação do RDBS-O, apresentando alguns dos conceitos instanciados pela ontologia a partir de RH Database.

3.6 ORM-O - Ontologia de Mapeamento Objeto Relacional

A ontologia ORM-O — *Object/Relational Mapping Ontology* — visa identificar e representar a semântica das entidades presentes em tempo de compilação no código-fonte que adota *frameworks* de mapeamento objeto/relacional (ORM). A ontologia ORM-O foi desenvolvida no contexto de uma dissertação de mestrado com o objetivo de reutilizar

as ontologias OOC-O e RDBS-O e, portanto, será brevemente apresentada a fim de relacioná-la com a SCON (mais detalhes da ontologia, bem como sua verificação, validação e aplicação estão disponíveis em (ZANETTI, 2020)).

O Mapeamento Objeto/Relacional reune tanto o paradigma de orientação a objetos quanto o relacional. Enquanto no relacional as informações são armazenadas na forma de tabelas, constituídas por linhas (tuplas) e colunas (atributos) (DATE, 2004), na orientação a objetos as informações são armazenadas em mémoria na forma de objetos contendo caracterísiticas e comportamentos (TEIXEIRA, 2017), definidos por meio de classes e relacionados por referências.

Portanto, uma linguagem OO é utilizada na codificação do programa enquanto o armazenamento das informações é feito em sistemas de banco de dados relacional. Assim, objetos são convertidos em tuplas para serem armazenados em tabelas e consultas são realizadas no banco de dados para que os objetos possam ser recuperados. Para auxiliar o desenvolvimento e minimizar problemas, o mapeamento objeto/relacional normalmente é delegado a um *framework*, conjunto de códigos que fornece solução para algum problema específico, neste caso, mapeamento objeto/relacional.

A construção da ontologia foi baseada no método SABiO (FALBO, 2014). Os seguintes Requisitos Não Funcionais (RNF) foram elicitados para ORM-O: **RNF01:** A ontologia deve considerar *frameworks* de diversas linguagens de programação e diferentes *frameworks* para uma mesma linguagem de programação; **RNF02:** A ontologia deve ser baseada em uma ontologia de fundamentação; **RNF03:** A ontologia deve reutilizar ontologias existentes de assuntos relacionados; **RNF04:** A ontologia deverá ser modularizada para permitir seu reuso.

A captura e formalização da ontologia é apoiada em análise ontológica baseadas em UFO, representando ORM-O no modelo OntoUML. Obedecendo *RNF01* e *RNF02*, foram selecionados os seguintes *frameworks*: Java Persistence API (JPA) (JPA, 2019), que é o padrão de implementação de *frameworks* para Java, como o Hibernate; Django (DJANGO, 2019) e SQLAlchemy (SQLALCHEMY, 2019), que são *frameworks* para a linguagem Python; e QxORM (QXORM, 2019) e ODB (ODB, 2019) que são *frameworks* para C++. Atendendo *RNF03*, foram reutilizadas as ontologias RDBS-O e OOC-O, apresentadas nas seções 3.5 e 3.3 respectivamente; por fim, atendendo *RNF04*, a ORM-O foi dividida em módulos, adaptados aqui para ORM-O Class e ORM-O Variable. Uma nova análise ontológica sobre as ontologias de ORM-O está prevista em trabalhos futuros (Seção 8.3).

Os seguintes requisitos funcionais (RF) foram elicitados para ORM-O: **RF01**: Que classes são mapeadas para o banco de dados? **RF02**: Como os relacionamentos entre classes são mapeados para o banco de dados? **RF03**: Que atributos de uma dada classe são mapeados para o banco de dados? **RF04**: Que atributos de uma dada classe são mapeados para chave primária no banco de dados? **RF05**: Que atributos de uma dada classe são

mapeados para chave estrangeira no banco de dados? **RF06**: Como os relacionamentos de herança entre classes são mapeados para o banco de dados? A seguir, apresentamos os módulos ORM-O Class e ORM-O Variable.

3.6.1 ORM-O Class

A Figura 13 apresenta os conceitos da ORM-O relativos ao mapeamento objeto/relacional de classes e suas possíveis relações, estendendo os conceitos **Class** de OOC-O e **Table** de RDBS-O.

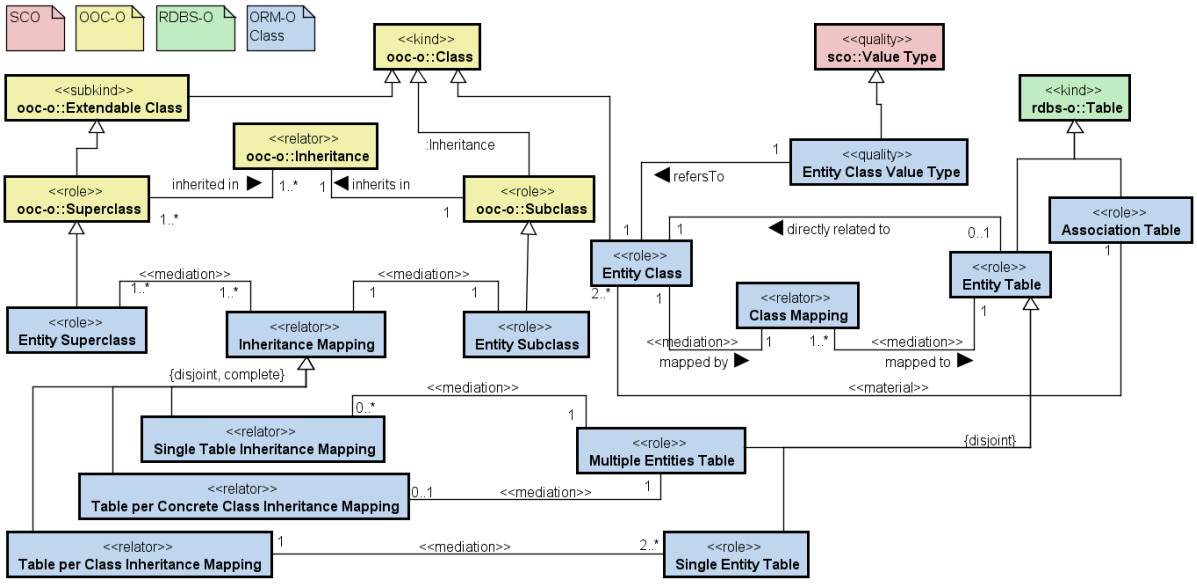


Figura 13 – Ontologia de Referência ORM-O Class adaptada (ZANETTI; AGUIAR; SOUZA, 2019)

O conceito ***Class*** de OOC-O é especializado em **Entity Class**, representação de uma classe que será efetivamente mapeada para o banco de dados. **Entity Class Value Type** é uma especialização de **Value Type** cuja relação *refers to* é feita obrigatoriamente com uma **Entity Class**. O conceito ***Table*** foi especializado em ***Entity Table***, tabelas do banco de dados que persistem dados de instâncias de classes (objetos) e ***Association Table***, tabelas que armazenam as chaves de associação de outras tabelas, geralmente usada na relação many-to-many. O mapeamento de uma *Classe* para uma *Entity Class* é representada pelo conceito ***Class Mapping*** de modo que um *Class Mapping* é exclusivo para uma *Entity Class*.

Existem três diferentes estratégias de mapeamento de herança (AMBLER, 2010), a saber: a estratégia *Single Table* é aquela em que os objetos das classes envolvidas em uma hierarquia são persistidos em uma única tabela no banco de dados; a estratégia *Table per Class* define que cada classe da hierarquia possui sua própria tabela e ali são armazenadas as informações dos atributos dessa única classe. Isso implica dizer que para recuperar informações de uma subclasse é necessário unir informações de diferentes tabelas do banco

de dados; e a estratégia *Table per Concrete Class* que define uma tabela no banco de dados para cada classe concreta da hierarquia (isto é, classes que podem ser instanciadas) e tal tabela deve conter toda informação necessária para recuperar a informação de um objeto, implicando na inclusão de colunas referentes a atributos das superclasses.

Os conceitos **Superclass** e **Subclass** de OOC-O são especializados em **Entity Superclass** e **Entity Subclass** a fim de representar superclasses ou subclasses que são mapeadas para o banco de dados. Este mapeamento de herança é representado pelo conceito **Inheritance Mapping**. Respeitando as diferentes estratégias de mapeamento de herança, **Inheritance Mapping** é especializado em **Single Table Inheritance Mapping**, **Table Per Class Inheritance Mapping** e **Table Per Concrete Class Inheritance Mapping**, que formam um *generalization set* completo e disjunto. Quando uma tabela assume o papel de armazenar informações dos objetos de uma única classe representa-se como **Single Entity Table**. De maneira oposta, em **Multiple Entities Table** são persistidos os atributos de duas ou mais classes da hierarquia.

Sendo assim, como a estratégia *Single Table* implica que todas as classes da hierarquias são mapeadas para uma única tabela, **Single Table Inheritance Mapping** é associado com **Multiple Entities Table** com cardinalidade `1`. Seguindo a análise, **Table per Class Inheritance Mapping** mapeia cada classe da hierarquia para uma tabela específica para ela, portanto é associado com **Single Entity Table** com cardinalidade `2..*`. Por fim, a estratégia *Table per Concrete Class*, que implica que toda a informação de um objeto esteja em uma única tabela, mapeia as instâncias de uma **Entity Subclass** para uma única **Multiple Entities Table**, pois contém também os atributos das **Entity Superclasses**.

3.6.2 ORM-O Variable

A Figura 14 apresenta os conceitos da ORM-O relativos a variável, estendendo o conceito **Instance Variable** de OOC-O e reutilizando os conceitos **Column**, **Primary Key Column** e **Foreign Key Column** de RDBS-O.

O conceito **Instance Variable** de OOC-O é especializado em **Mapped Variable**, variáveis que são efetivamente mapeadas do código OO para o banco de dados relacional. O mapeamento entre **Mapped Variable** e **Column** é representado pelo `<<relator>>` **Variable Mapping** que tem relação *mapped to* com **Column**, dado que atributos são mapeados para colunas do banco de dados.

As variáveis que são mapeadas para **Primary Key Column** e **Foreign Key Column** do banco de dados são representadas pelos conceitos **Mapped Primary Key** e **Mapped Foreign Key**, que especializam **Mapped Variable**. O `<<relator>>` **Variable Mapping** também é especializado em **Primary Key Mapping** e **Foreign Key Mapping**,

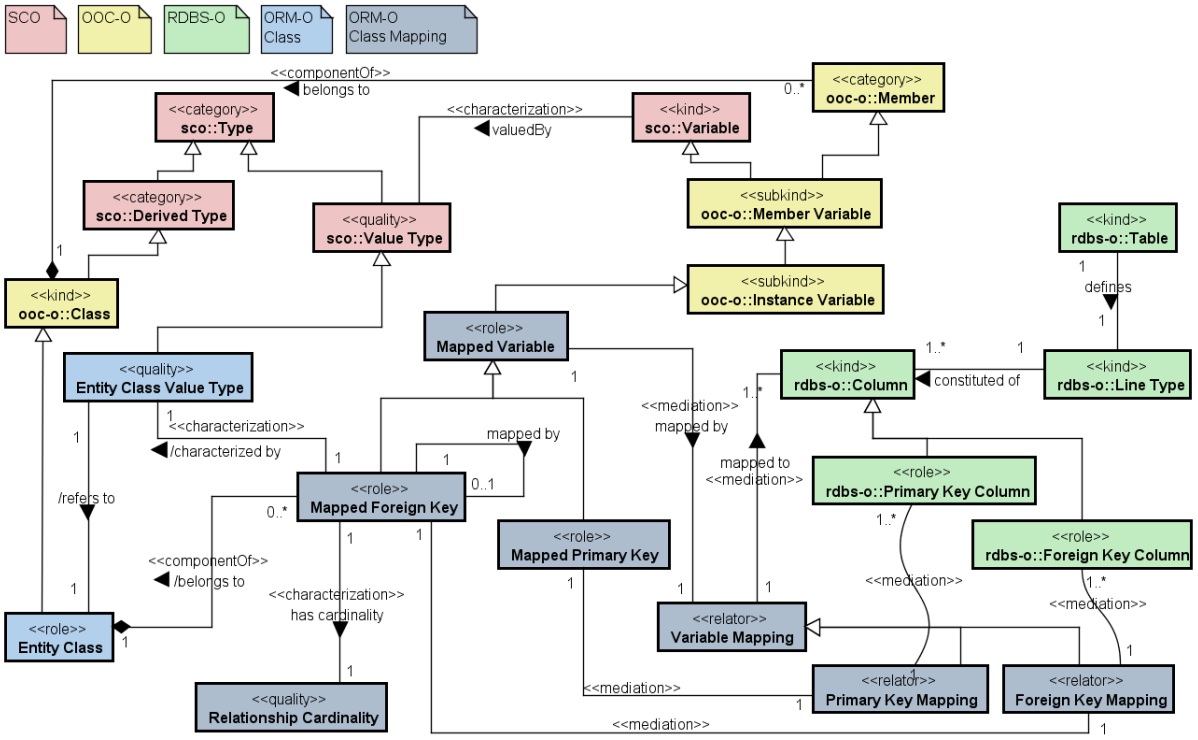


Figura 14 – Ontologia de Referência ORM-O Variable adaptada de (ZANETTI; AGUIAR; SOUZA, 2019)

derivando implicitamente as relações *mapped by* e *mapped to*.

É sabido que, no paradigma relacional, existem quatro diferentes cardinalidades de relacionamentos: um para um (*one-to-one*), um para muitos (*one-to-many*), muitos para um (*many-to-one*) e muitos para muitos (*many-to-many*). Sendo assim, **Mapped Foreign Key** tem uma relação `<<characterization>>` *has cardinality* com **Relationship Cardinality**.

Percebemos ainda que, no lado OO, dizer que uma classe A tem um relacionamento um para um com uma classe B não implica necessariamente em dizer que a classe B tem o mesmo relacionamento com A. No código OO deve ser definido um segundo relacionamento de B para A e, para que o *framework* entenda que se trata do mesmo relacionamento e faça o mapeamento corretamente, essa bidirecionalidade deve ser explicitada. Esse conceito é representado pela relação *mapped by*, que pode associar uma **Mapped Foreign Key** a outra.

3.7 Considerações Finais

Este capítulo apresentou a rede de ontologias de código-fonte SCON, formada por ontologias construídas de acordo com um método de engenharia de ontologia, com base em uma ontologia de fundamentação e suportada por fontes de dados reconhecidas. Nesse

contexto, até onde sabemos, não encontramos na literatura nenhum trabalho que abranja o domínio proposto neste capítulo em profundidade. Assim, a seguir destacamos alguns trabalhos relacionados ao domínio das ontologias que compõem a rede SCON.

Conceitos de orientação a objetos são aplicados em muitas outras áreas como banco de dados (ATKINSON et al., 1990), metodologia de desenvolvimento (PASTOR et al., 1997) e análise de dados (BRUN; RADEMAKERS, 1997). No contexto da ontologia OOC-O, observamos trabalhos que:

- aplicam mapeamento semântico entre conceitos ontológicos da ontologia BWW e construtos de linguagem de programação OO para atribuir semântica e regras no contexto de modelagem de software (EVERMANN; WAND, 2005). Os conceitos BWW (coisa, propriedade e esquema funcional) são mapeados para conceitos UML (objeto, classe, atributo, atributo de classe 'comum' e atributo de classe de associação). Embora a pesquisa tenha aplicado uma análise ontológica para mapear construções orientadas a objetos, ela cobre apenas uma pequena parte do domínio de OOC-O.

- aplicam uma ontologia operacional de linguagem de programação para representar o conhecimento ministrado por um curso a distância em programação de computadores (KOUNELI et al., 2012). Embora a ontologia seja construída seguindo uma metodologia e fontes de informação da linguagem Java, ela não se baseia em nenhuma ontologia de fundamentação. Os conceitos da ontologia são ancorados no conceito *Thing* e hierarquicamente organizados a partir de *Java Element* (Class, Constructor, Data Type, Exception, Interface, Method (AbstractMethod, FinalMethod, ClassMethod e InstanceMethod), Object, Operator, Package, Statement, Thread and Variable (ClassVariable, InstanceVariable, LocalVariable and Parameter)), *Keyword* e *Literal Value*. Ao contrário de OOC-O, esta ontologia representa apenas o domínio da linguagem de programação Java e incorpora conceitos não orientados a objetos, como exceção, operador, instrução e encadeamento.

- elaboram ontologia de referência inspirada na BWW e no framework FRISCO, a ontologia O3, para mapear semanticamente os conceitos do paradigma de programação OO (PASTOR, 1992). Os conceitos de BWW (thing, property, substantial e relation) são especializados para os conceitos do paradigma OO (class (generalization, specialization), domain (primitive type)), attribute (variable, constant), interface). Embora a pesquisa tenha aplicado uma análise ontológica para mapear conceitos de orientação a objetos, ela cobre apenas uma pequena parte desse domínio e incorpora conceitos não orientados a objetos, como restrição, serviço, relação, agente, servidor e outros.

Embora ontologias sobre o domínio do banco de dados estejam disponíveis na

literatura, tais ontologias não representam a mesma visão de RDBS-O. No contexto da ontologia RDBS-O, observamos trabalhos que:

- descreve o esquema de um banco de dados relacional na forma abstrata e gera um formato de representação do próprio banco de dados (LABORDA; CONRAD, 2005). A ontologia Relational.OWL é escrita em Java com JDBC e Jena. Assim, a partir da representação independente de plataforma, um banco de dados extraído de uma plataforma A pode ser importado para uma plataforma B, atualmente com suporte a MySQL e DB2. A ontologia consiste de quatro classes (dbs:Database, dbs:Table, dbs:Column, dbs:PrimaryKey) e sete propriedades (dbs:has, dbs:hasTable, dbs:hasColumn, dbs:isIndentifiedBy, dbs:references, dbs:scale, dbs:length). Uma análise ontológica sobre Relational.OWL resultou em alguns pontos questionáveis: (i) o modelo é representado apenas em OWL, o que é pouco expressivo; (ii) associações de cardinalidade não são apresentadas; e (iii) apresenta cobertura limitada do domínio, não considerando, por exemplo, tipo de dados, chaves estrangeiras e restrições.

- descreve um vocabulário OWL, seus relacionamentos semânticos e restrições do sistema de banco de dados relacional (TRINH; BARKER; ALHAJJ, 2006). A partir desse vocabulário, uma ferramenta gera e publica instâncias da ontologias dinamicamente. A OWL-RDBO é escrita em Java com JDBC para extrair metadados e restrições estruturais do banco de dados, atualmente com suporte a MySQL, PostegreSQL e DB2, e gerar uma ontologia em OWL como uma instância de OWL-RDBO. O artigo descreve que a ontologia é formada por algumas classes rdbo:DatabaseName, rdbo:RelationList, rdbo:Relation, rdbo:AttributeList, rdbo:Attribute) e propriedades (rdbo:hasRelations, rdbo:hasType,rdbo:referenceAttribute, rdo:referenceRelation), mas a ontologia completa não está disponível para acesso. Uma análise ontológica sobre OWL-RDBO resultou em alguns pontos questionáveis: (i) o modelo é representado apenas em OWL, o que não é muito expressivo; (ii) o que OWL-RDBO define como Relação, optamos por utilizar o termo Tabela para diminuir os conflitos semânticos; e (iii) inclui conceitos externos ao domínio, como RelationList para agrupar um conjunto de Relation e AttributeList para agrupar um conjunto de atributos.

A relação existente entre o domínio de orientação a objetos e dados relacionais não é muito explorada de forma ontológica. No contexto da ontologia ORM-O, observamos trabalhos que:

- apresenta uma ontologia para o recurso objeto-relacional do padrão SQL:2003 a fim de clarificar seus elementos e identificar suas inconsistências (CALERO et al., 2006). A ontologia é subdividida in DataTypes (conceitos relevantes a tipo de dados)

e Schema-Objects (conceitos pertinentes ao contexto geral, tabelas, restrições e colunas). Diferentemente de ORM-O, o modelo é representado em UML e regras OCL. Embora essa ontologia objetive representar aspectos objeto/relacionais de um esquema de banco de dados, a ontologia não explora essa relação entre os dois domínios e apenas representa os conceitos relacionais na forma de um diagrama de objeto.

- apresenta a especificação de um metamodelo sobre data warehouse (CHANG; IYENGAR et al., 2001). O modelo é dividido, entre outros, em *Object Model* (conceitos base relacionados a classe e objeto) e *Resource* (conceitos relacionados a representação relacional). Diferentemente de ORM-O, o modelo é representado em UML e define relações de especialização entre os domínios de objeto e dados relacionais. Assim, *Column* e *Data Type* do submodelo *Relational* são especializações de *Attribute* e *Classifier* do submodelo *Object Model*, respectivamente. Em ORM-O tais conceitos são ancorados na ontologia de fundamentação UFO e relacionados por meio de mapeamentos na linguagem de programação.

A seguir, o Capítulo 4 apresenta o método OSCIN para interoperabilidade semântica de código-fonte baseada em ontologia, que pode utilizar as ontologias elaboradas na rede SCON.

4 Método para Interoperabilidade Semântica de Código-Fonte baseada em Ontologia

Neste capítulo apresentamos o OSCIN — *Ontology-based Source Code Interoperability*, método que visa interoperar semanticamente códigos-fonte não cooperantes de diferentes linguagens de programação, proporcionando integração e compartilhamento.

Como apresentado no Capítulo 3, cada artefato de software corresponde ao seu propósito (em níveis diferentes), de modo que Code é a base para os artefatos de software Program e Software System. Embora esses artefatos tenham uma relação muito próxima, esta pesquisa está direcionada ao artefato e interoperabilidade de código-fonte. A Figura 15 apresenta a relação entre os artefatos de software e algumas de suas interoperabilidades, acrescida da interoperabilidade de código-fonte, identificada nesta pesquisa. Neste caso, como já apresentado, definimos interoperabilidade de código-fonte, aquela capacidade dos códigos-fonte de trocar informações ou usá-las em um contexto heterogêneo de diferentes linguagens de programação.

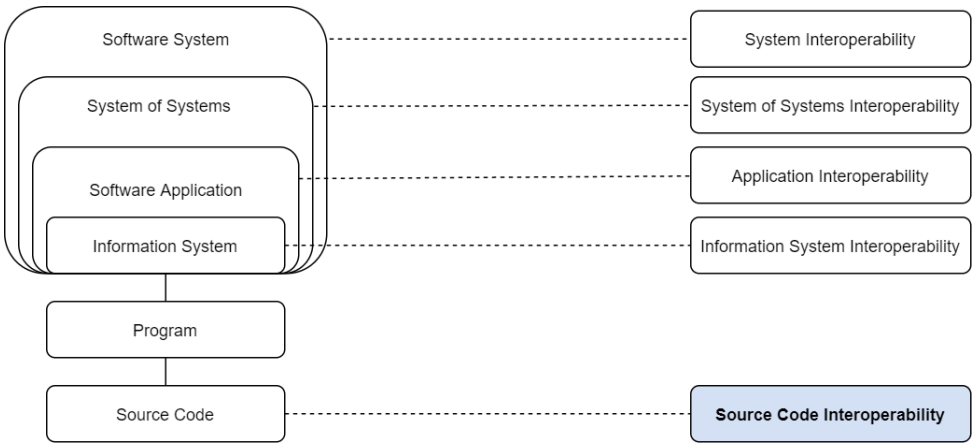


Figura 15 – Relação entre os artefatos e algumas interoperabilidades

OSCIN representa semanticamente o código-fonte na forma de uma ontologia e a utiliza em aplicações ontológicas de forma unificada, possibilitando a troca de informações entre códigos-fonte de diferentes linguagens de programação. Para isso, o método adota três pilares para a interoperabilidade de código-fonte:

- **Subdomínio de código-fonte**, denota a parte do domínio de código-fonte que é tratada pelo método, tal como: paradigma orientado a objeto, *framework* de mapeamento objeto/relacional e outros. Assim, o subdomínio de código-fonte é representado em uma ontologia bem fundamentada e que reúne a conceituação deste

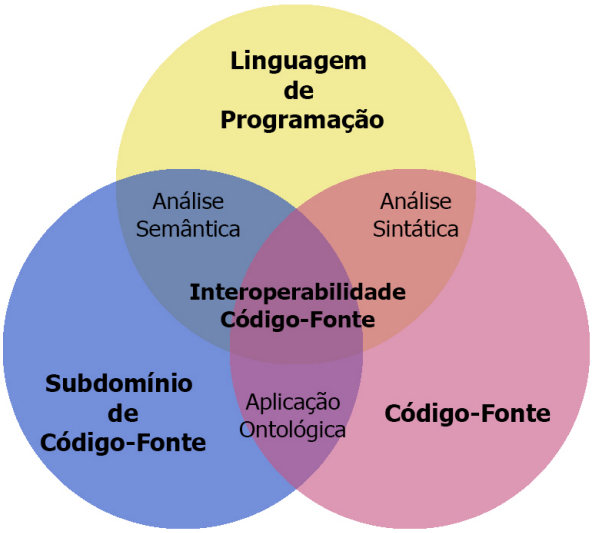


Figura 16 – Pilares do método OSCIN

subdomínio, por exemplo, uma ontologia de código orientado a objetos formada por conceitos como classe, membros de classe, método, etc.

- **Linguagem de programação**, denota a linguagem de programação que o método é capaz de manipular, tal como a linguagem Java.
- **Código-Fonte**, denota o código-fonte que será manipulado pelo método, tal como o código-fonte de uma biblioteca, aplicação ou sistema.

A Figura 16 apresenta como os pilares são combinados de modo a permitir a interoperabilidade do código-fonte, sendo que a **análise semântica** é a interseção entre o subdomínio de código-fonte e a linguagem de programação, mapeando a semântica do subdomínio aos elementos da linguagem de programação, por exemplo, elementos de orientação a objetos da linguagem Java representados como instancias da ontologia de código orientado a objetos; a **análise sintática** é a interseção entre a linguagem de programação e o código-fonte, representando o código-fonte como uma estrutura sintática da linguagem que contempla seus principais elementos, por exemplo, uma árvore sintática abstrata da linguagem Java; e a **aplicação ontológica** é a interseção entre o código-fonte e o subdominio de código-fonte, aplicando a representação semântica do código para um propósito de aplicação, por exemplo, a detecção de *bad smell*. A combinação destes pilares possibilita a interoperabilidade de código-fonte em múltiplas linguagens de programação e diferentes subdomínios de código-fonte.

Para permitir que seus recursos digitais se tornem mais *Findable*, *Accessible*, *Interoperable* e *Reusable* por máquinas e também por humanos, o método OSCIN adota os princípios *FAIR* (GUIZZARDI, 2019):

- *Findable:* facilidade de encontrar os metadados do subdomínio de código-fonte e seus respectivos códigos-fonte tanto por humanos quanto máquinas. Para isso, os subdomínios são representados como ontologias e os códigos-fonte representados como instâncias dessas ontologias, contendo identificadores únicos e pesquisáveis.

- *Accessible:* acesso a metadados do subdomínio, propósitos de aplicação e códigos-fonte. Para isso, os subdomínios e os códigos-fonte são acessados pelos identificadores de forma padronizada e são disponibilizados independentemente dos dados.

- *Interoperable:* integração com outros subdomínios, linguagens de programação e propósitos de aplicação. Para isso, o conhecimento de subdomínios e linguagens de programação é representado em ontologias e de propósitos de aplicação em tecnologias baseadas em ontologia, com uma linguagem formal, acessível e compartilhada.

- *Reusable:* otimização do reuso de metadados do subdomínio de código-fonte e propósitos de aplicação. Para isso, os subdomínios e os propósitos de aplicação são descritos, detalhados e disponibilizados com licença de reuso e seguindo padrões de linguagem de programação e propósitos de aplicação.

4.1 Ciclo de Vida

A Figura 17 apresenta uma visão geral (baseada em notação BPMN) do ciclo de vida do método OSCIN, cujas fases são identificadas verticalmente por cores diferentes, os respectivos papéis por círculos coloridos abaixo de cada fase e os artefatos por símbolo de artefato no final de cada fase. OSCIN é formado pelas fases de:

- [SPEC] Specification Phase (representada pela cor roxa), identifica o contexto no qual o método deve ser aplicado. O *subdomínio de código-fonte* indica a parte do domínio mais geral do código-fonte na qual estamos interessados, por exemplo, código orientado a objetos (OO), código que usa mapeamento objeto/relacional (ORM), código de apresentação em aplicativos Android, etc. A *linguagem de programação* é aquela que pretendemos representar, por exemplo, código OO em Java, código ORM em Python, código de apresentação Android em Kotlin, etc. Finalmente, o *propósito de aplicação* indica o tipo de aplicação que será tratada sobre o código-fonte, por exemplo, detecção de *bad smell*, migração de código, etc;

- [ASES] Subdomain Semantic Abstraction Phase (representada pela cor rosa), define a conceituação do subdomínio dentro do domínio geral de código-fonte e a representa em um modelo semântico, isto é, uma ontologia.

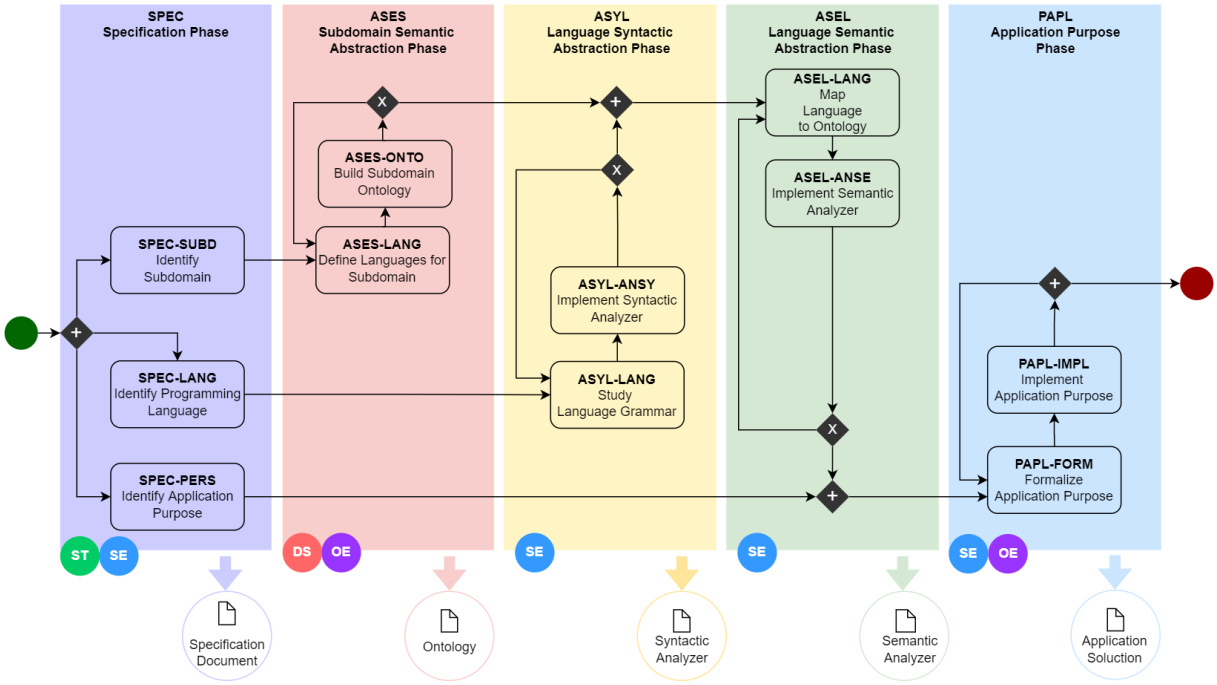


Figura 17 – Ciclo de vida do método OSCIN

- [ASYL] Language Syntactic Abstraction Phase (representada pela cor amarela), analisa o código-fonte em sua forma elementar por meio de um analisador sintático produzido para a linguagem de programação de interesse.
- [ASEL] Language Semantic Abstraction Phase (representada pela cor verde), analisa a estrutura sintática da linguagem de programação e a representa no modelo semântico do subdomínio por meio de um analisador semântico.
- [PAPL] Application Purpose Phase (representada pela cor azul), utiliza a ontologia do subdomínio para um determinado propósito de aplicação ontológica. O método trata o propósito de métrica de código orientado a objeto, detecção de *bad smell* orientado a objeto e migração de código de mapeamento objeto relacional, mas não limita um conjunto específico de propósitos possíveis.

O ciclo de vida é *iterativo* de modo que iterações são realizadas a cada fase do ciclo de vida e ao final de cada fase, o artefato correspondente é gerado, sendo Documento de Especificação para [SPEC] Specification Phase, Ontologia para [ASES] Subdomain Semantic Abstraction Phase, Analisador Sintático para [ASYL] Language Syntactic Abstraction Phase, Analisador Semântico para [ASEL] Language Semantic Abstraction Phase e Solução de Aplicação para [PAPL] Application Purpose Phase.

As fases são definidas de forma progressiva de modo que fases posteriores utilizam artefatos produzidos por fases anteriores no ciclo de vida. O ciclo de vida foi projetado para acomodar subdomínios, linguagens de programação e propósitos de aplicação que evoluem ao longo do tempo, ou seja, o método suporta a evolução de:

- novos subdomínios nas mesmas linguagens de programação, repetindo as fases de [SPEC] Specification Phase, [ASES] Subdomain Semantic Abstraction Phase, [ASEL] Language Semantic Abstraction Phase e, consequentemente, para a adição de novos propósitos de aplicação neste novo subdomínio, a fase de [PAPL] Application Purpose Phase. A fase de [ASYL] Language Syntactic Abstraction Phase não é requerida, uma vez que os artefatos produzidos em ciclos anteriores são reusados neste ciclo;
- novas linguagens de programação no mesmo subdomínio, repetindo as fases de [SPEC] Specification Phase, [ASYL] Language Syntactic Abstraction Phase e [ASEL] Language Semantic Abstraction Phase. As fases de [ASES] Subdomain Semantic Abstraction Phase e [PAPL] Application Purpose Phase não são requeridas, uma vez que os artefatos produzidos em ciclos anteriores são reusados neste ciclo;
- novos propósitos de aplicação ou alterações em propósitos existentes, repetindo as fases de [SPEC] Specification Phase e [PAPL] Application Purpose Phase. As fases de [ASES] Subdomain Semantic Abstraction Phase, [ASYL] Language Syntactic Abstraction Phase e [ASEL] Language Semantic Abstraction Phase não são requeridas, uma vez que os artefatos produzidos em ciclos anteriores são reusados neste ciclo.

Assim, este ciclo evolutivo permite expandir as linguagens de programação selecionadas, os subdomínios tratados e os propósitos de aplicação cobertos pelo método, além de seu refinamento. Neste cenário, o método é idealizado para ser aplicado por desenvolvedores de soluções de desenvolvimento de software (editores de código, ambiente de desenvolvimento integrado, analisador de código e outros), uma vez que a elaboração de soluções de desenvolvimento de software sob medida, em todas as linguagens de programação e para todos os tipos de aplicação, não é uma atividade sustentável e evolutiva.

Assim, os seguintes papéis são identificados durante o ciclo de vida do método:

- ***Stakeholder*:** (representado pelo circulo na cor verde) pessoa interessada no desenvolvimento ou customização de soluções de desenvolvimento de software;
- **Especialista de Domínio:** (representado pelo circulo na cor vermelha) pessoa que mantém o conhecimento/*expertise* do domínio de código-fonte;
- **Engenheiro de Ontologias:** (representado pelo circulo na cor roxa) pessoa com conhecimento em engenharia de ontologias relacionado à construção da ontologia de referência e a implementação da ontologia operacional;
- **Engenheiro de Software:** (representado pelo circulo na cor azul) pessoa com conhecimento em Engenharia de Software relacionado à construção de artefatos de software.

Um determinado indivíduo pode atuar em diferentes papéis e, dependendo da aplicação do método, esses papéis podem sofrer alguma especialização ou generalização. Por exemplo, desenvolvedores de software são o público-alvo que irá usufruir das aplicações para as diferentes linguagens de programação elaboradas de forma interoperável. No entanto, em um dado momento, esses desenvolvedores podem assumir o papel de *Stakeholder* ou Engenheiro de Software a fim de contribuir, estender e evoluir uma determinada aplicação do método.

As seções seguintes apresentam o método OSCIN de maneira detalhada, descrevendo suas fases de acordo com as seguintes características: **how**, como, representa as atividades que devem ser executadas durante o ciclo de vida do método; **who**, quem, representa o papel desempenhado por um indivíduo ou grupo de indivíduos que executa *how* durante o ciclo de vida do método; **what**, o que, representa a informação recebida como entrada (*what-input*) e produzida como saída (*what-output*) a partir de *how*; e **running example**, exemplo ilustrativo, no qual os *stakeholders* desejam detectar *code smell* na camada de apresentação de aplicativos Android. No entanto, o método não é prescritivo em relação aos recursos tecnológicos que devem ser utilizados para guiar essas atividades.

4.2 [SPEC] Specification Phase

A Fase de Especificação ([SPEC] Specification Phase) objetiva identificar as características relacionadas ao subdomínio de código-fonte, a linguagem de programação e o propósito de aplicação. Esta fase é formada pelas atividades de [SPEC-SUBD] Identify Subdomain, [SPEC-LANG] Identify Programming Language e [SPEC-PERS] Identify Application Purpose, descritas a seguir.

4.2.1 [SPEC-SUBD] Identify Subdomain

How: atividade que identifica o subdomínio a ser representado no código-fonte de acordo com o *stakeholder*. Identificar subdomínio é uma tarefa subjetiva e pode ser realizada a partir de diferentes visões sobre o código-fonte ou até mesmo originar diferentes subdomínios a partir de um único código-fonte. Por exemplo, o código escrito de maneira orientada a objetos deve ser analisado de maneira bastante diferente de um escrito usando programação funcional; o mesmo vale para aplicações Web versus software *standalone*, código que usa mapeamento objeto/relacional ou não, e assim por diante.

A ênfase desta atividade está em identificar claramente o subdomínio que será tratado. O nome escolhido para definir o subdomínio merece atenção porque é essencial para sua identificação, reutilização e compartilhamento. Por outro lado, a descrição do subdomínio não precisa ser extensa ou profunda, mas deve apresentar suas principais características. Isso porque mais detalhes e o escopo do subdomínio serão abordados na

[ASES] Subdomain Semantic Abstraction Phase.

Who: engenheiro de software extrai dos *stakeholders* a informação sobre o subdomínio de código-fonte a ser coberto.

What-input: necessidade de interoperar o código-fonte em determinado subdomínio.

What-output: descrição textual do subdomínio identificado, registrado como parte do Documento de Especificação.

Running example: descrição do subdomínio de código-fonte para a camada de apresentação dos aplicativos Android.

4.2.2 [SPEC-LANG] Identify Programming Language

How: atividade que identifica a linguagem de programação manipulada pelo método (em cada ciclo de vida de OSCIN apenas uma linguagem é identificada nesta fase). Em outras palavras, deve-se identificar a linguagem de programação para a qual se deseja interoperar o código-fonte, no contexto do subdomínio identificado. Dado que o objetivo do método é interoperar o código-fonte usando uma representação ontológica do subdomínio, a linguagem de programação identificada deve ser considerada relevante em seu respectivo subdomínio, identificado anteriormente na atividade [SPEC-SUBD] Identify Subdomain.

Além disso, uma linguagem de programação pode abranger diferentes subdomínios e um subdomínio podem ser abrangido por diferentes linguagens de programação. A linguagem identificada orientará as fases de análise sintática e semântica, principalmente no que diz respeito à compreensão dos elementos que compõem a linguagem e à aquisição de conhecimento sobre o subdomínio do código-fonte a ser representado.

A ênfase desta atividade é identificar a linguagem de programação relacionada aos projetos de software que se deseja aplicar a interoperabilidade.

Who: engenheiro de software extrai dos *stakeholders* a linguagem de programação que deve ser considerada para a interoperabilidade de código.

What-input: necessidade de interoperar o código-fonte de uma determinada linguagem de programação.

What-output: linguagem de programação identificada, registrada como parte do Documento de Especificação. Observe que novas linguagens de programação devem ser tratadas em novos ciclos de vida.

Running example: identificação da linguagem Kotlin (Java poderia ser especificada em uma execução posterior, reutilizando artefatos deste).

4.2.3 [SPEC-PERS] Identify Application Purpose

How: atividade que identifica e descreve o propósito de aplicação relacionado ao domínio de código-fonte. Cada propósito de aplicação possui características singulares que devem ser identificadas e descritas nesta atividade.

O método trata o propósito de métrica de código orientado a objeto, detecção de *bad smell* orientado a objeto e migração de código de mapeamento objeto relacional, mas não limita um conjunto específico de propósitos possíveis. Para o propósito de métrica de código, deve-se definir as características relacionadas ao *tipo* de métrica e a *regra* adotada para calcular a respectiva métrica, por exemplo, *tipo*: Número de Classes (NOC) e *regra*: número de classes definidas em um projeto, excluindo classes de biblioteca. Para o propósito de detecção de *smells*, deve-se definir o *tipo* de *smell* e a *regra* adotada para identificar o *smell* no código-fonte, por exemplo, *tipo*: Long Parameter List e *regra*: método de uma classe composta por mais de seis parâmetros. Para a migração de código, deve-se definir o *template* de código a ser adotado na geração de código conforme a linguagem identificada na Seção 4.2.2.

Sugerimos a definição de características a partir da literatura e de novas características para aquelas ainda não identificadas. Obviamente, as organizações são livres para definir características que são particulares ao seu contexto e convenções internas. A ênfase desta atividade é identificar claramente as características da aplicação a partir da literatura e da experiência. Isso é importante porque uma mesma característica pode adotar diferentes definições e interpretações, por exemplo, a literatura define cinco, seis e sete parâmetros para a detecção do *smell Long Parameter List*, dependendo da pesquisa.

Who: engenheiro de software extrai dos *stakeholders* o propósito da aplicação.

What-input: necessidade de interoperar o código-fonte segundo uma determinada aplicação.

What-output: propósito de aplicação identificado e suas respectivas características.

Running example: poderíamos selecionar *smells* a partir de (CARVALHO et al., 2019), por exemplo, *Coupled UI Component*, *Deep Nested Layout*, etc.

4.3 [ASES] Subdomain Semantic Abstraction Phase

A fase de Abstração Semântica do Subdomínio ([ASES] Subdomain Semantic Abstraction Phase) objetiva representar o subdomínio de código-fonte em uma abstração de recursos semânticos, na forma de uma ontologia.

Assim, esta fase lida com diferenças conceituais existentes entre diferentes linguagens

de programação. Nesse caso, é possível observar: (i) linguagens de programação diferentes com tokens e significados idênticos, como o token `class` adotado por Java e C++ para representar um tipo de dado abstrato; (ii) linguagens de programação diferentes com tokens idênticos e significados diferentes, como o token `private`, adotado em Java para representar que uma variável ou método é acessível apenas na própria classe e adotado em C++ para, além disso, também representar herança privada, ou seja, que todos os membros herdados da superclasse são membros privados da subclasse; e (iii) linguagens de programação diferentes com tokens diferentes e significados idênticos, como os tokens `frozen` e `final` adotados por C++ e Java para representar uma classe não extensível.

Observe que esta fase produz um único artefato comum para a representação do subdomínio, ou seja, o subdomínio representado na ontologia é único e independente de linguagem de programação. As atividades [ASES-LANG] Define Linguages for Subdomain e [ASES-ONTO] Build Subdomain Ontology devem ser executadas uma vez para o subdomínio identificado, no entanto, deve-se retornar a estas atividades sempre que forem necessárias alterações no subdomínio.

4.3.1 [ASES-LANG] Define Linguages for Subdomain

How: atividade que define as linguagens de programação que formarão a base de conhecimento para o subdomínio identificado na atividade [SPEC-SUBD] Identify Subdomain. A definição deve incluir a linguagem de programação identificada na atividade [SPEC-LANG] Identify Programming Language, mas também considerar outras linguagens com base em sua relevância e contribuição para o subdomínio de interesse, bem como sua popularidade na comunidade de programação.

Sugere-se identificar as linguagens de programação que incluam o subdomínio de interesse, de acordo com as seguintes diretrizes: (i) *Relevância*, busca por linguagens de programação pela data de criação e considera quais das linguagens foram relevantes para introduzir esse subdomínio na comunidade de programação; e (ii) *Popularidade*, busca por linguagens de programação em pelo menos dois índices (*rankings* de adoção) da comunidade de programação, tal como, TIOBE,[1] IEEE Spectrum[2] e Redmonk.[3] A partir desta lista, deve-se calcular a média das linguagens de programação de acordo com estes índices e identificar quais das linguagens mais bem classificadas representam este subdomínio.

A ênfase desta atividade é definir um conjunto de linguagens de programação fortemente relacionadas ao subdomínio. A definição de um conjunto de linguagens e não apenas uma permite uma análise consensual do subdomínio e dos conceitos que devem fazer parte dele. A definição de linguagens fortemente relacionadas ao subdomínio permite

[1] <https://www.tiobe.com/tiobe-index/>
[2] <https://spectrum.ieee.org/at-work/tech-careers/top-programming-language-2020.>
[3] <https://redmonk.com/sogrady/2020/07/27/language-rankings-6-20/>

uma análise mais precisa, a fim de reduzir a interferência de outros subdomínios que também podem compor essas linguagens e melhorar a percepção sobre os conceitos que compõem esse subdomínio.

Who: engenheiro de ontologia extrai do *ranking* e do especialista de domínio as linguagens de programação a serem consideradas no desenvolvimento da ontologia.

What-input: subdomínio identificado na atividade [SPEC-SUBD] Identify Subdomain.

What-output: linguagens de programação relacionadas ao subdomínio.

Running example: Java e Kotlin poderiam ser selecionados uma vez que são as duas linguagens com suporte na plataforma Android.

4.3.2 [ASES-ONTO] Build Subdomain Ontology

How: atividade que representa a conceituação do subdomínio de interesse na forma de uma ontologia, tanto no nível de referência quanto no operacional. Esta ontologia deve necessariamente representar a conceituação compartilhada do subdomínio e não a estrutura de uma linguagem de programação específica, como ocorre na estrutura sintática da linguagem.

Assim, o conhecimento do subdomínio deve ser extraído das linguagens definidas na atividade [ASES-LANG] Define Linguages for Subdomain, para que os conceitos representados na ontologia reflitam o consenso alcançado entre essas linguagens. Além disso, a ontologia não deve representar as caractéristicas do propósito da aplicação, mas a conceituação compartilhada do subdomínio de código-fonte, uma vez que o propósito é tratado a partir do raciocínio aplicado na ontologia.

A qualidade e a cobertura da ontologia são os principais fatores para a qualidade da interoperabilidade do código-fonte. Por esse motivo, é altamente recomendável a adoção de um método de construção de ontologia que cubra da ontologia de referência à ontologia operacional e seja baseado em uma ontologia de fundamentação, tal como SABiOS, apresentado no Capítulo A. Uma vez que a ontologia de fundamentação define categorias ontológicas independentes de domínio e forma uma base mais elaborada para ontologias específicas de domínio (GUIZZARDI; WAGNER, 2004), sua adoção traz um guia para o desenvolvimento de ontologias consistentes (ROUSSEY et al., 2011).

Por um lado, a ontologia de referência deve ser expressiva e representar com precisão o subdomínio. Por outro lado, a ontologia operacional deve considerar o desempenho do raciocínio e da inferência. Vale ressaltar que a reutilização de ontologias já existentes ou de seus conceitos é incentivada e pode trazer vantagens relacionadas à maturidade da ontologia e ao tempo dedicado à aplicação do método.

A ênfase desta atividade é aplicar um método para a construção da ontologia no subdomínio identificado, considerando análise ontológica. A ontologia deve representar em detalhes todos os conceitos do código-fonte relacionados ao subdomínio, sem se preocupar com a estrutura do código e caraterísticas da perspectiva.

Who: engenheiro de ontologia extrai do especialista de domínio e das linguagens de programação identificadas o conhecimento sobre o subdomínio e o representa em ontologia de referência e operacional.

What-input: subdomínio identificado na atividade [SPEC-SUBD] Identify Subdomain e as linguagens de programação definidas na atividade [ASES-LANG] Define Linguages for Subdomain.

What-output: ontologia de referência e operacional para o subdomínio.

Running example: ontologia sobre a camada de apresentação de aplicativos Android seria desenvolvida, preferencialmente reutilizando ontologias sobre a plataforma Android em geral e sobre código orientado a objetos (por exemplo, (AGUIAR; FALBO; SOUZA, 2019)). A ontologia de referência poderia ser um modelo OntoUML (GUIZZARDI, 2005) e seu documento de especificação, enquanto a ontologia operacional seria um arquivo OWL baseado no modelo de referência.

4.4 [ASYL] Language Syntactic Abstraction Phase

A fase de Abstração Sintática da Linguagem ([ASYL] Language Syntactic Abstraction Phase) objetiva representar o código-fonte em uma abstração de características sintáticas, isto é, uma que considere a estrutura e a forma do código-fonte. As atividades [ASYL-LANG] Study Language e [ASYL-ANSY] Implement Syntactic Analyzer devem ser executadas para a linguagem de programação identificada na atividade [SPEC-LANG] Identify Programming Language.

Como muitas linguagens de programação já possuem um analisador disponível capaz de executar análises sintáticas, a atividade [ASYL-ANSY] Implement Syntactic Analyzer pode ser otimizada reutilizando esses analisadores. No entanto, mesmo reutilizando um analisador disponível, a atividade [ASYL-LANG] Study Language permanece necessária, pois esse conhecimento será usado na [ASEL] Language Semantic Abstraction Phase. Observe que esta fase produz um artefato específico para a linguagem de programação identificada, no entanto, novas linguagens de programação podem ser adicionadas em novos ciclos de vida do método.

4.4.1 [ASYL-LANG] Study Language

How: atividade que estuda o léxico e a sintaxe que compõem a linguagem de programação identificada, ou seja, seus tokens e gramática. Os tokens são usados para designar unidades de significado, ou seja, uma sequência menor de caracteres com significado na linguagem. A gramática é o conjunto de regras da linguagem que descreve sua sintaxe e, portanto, deve ser não ambígua e garantir que cada elemento da linguagem seja entendido de uma maneira única.

A gramática é uma maneira válida de combinar tokens, estabelecendo o relacionamento entre eles e agrupando-os em estruturas intermediárias que podem ser combinadas. Esta estrutura possui no lado esquerdo um símbolo não terminal e no lado direito uma sequência de um ou mais símbolos não terminais e terminais. Os símbolos terminais são definidos a partir dos tokens especificados na linguagem. Como a definição de tokens e da gramática resulta da própria linguagem, sugerimos que esta atividade seja realizada a partir da documentação da linguagem e de suas especificações.

A ênfase desta atividade é entender os elementos que compõem a linguagem de programação e como eles estão estruturados. Sem esse conhecimento, é muito difícil implementar um analisador sintático (atividade [ASYL-ANSY] Implement Syntactic Analyzer) e mapear os elementos da linguagem para a ontologia (atividade [ASEL-LANG] Map Language to Ontology).

No entanto, vale ressaltar que o conhecimento e esforço despendidos nesta fase estão associados ao reaproveitamento de artefatos já elaborados e ao conhecimento do engenheiro de software. Em outras palavras, se a atividade [ASYL-ANSY] Implement Syntactic Analyzer reutilizar um analisador sintático já elaborado para a linguagem, o estudo se concentrará apenas no que é necessário para [ASEL] Language Semantic Abstraction Phase. Além disso, se o engenheiro de software já possuir conhecimento sobre estrutura da linguagem, o esforço despendido nesta atividade é substancialmente reduzido.

Who: engenheiro de software analisa os elementos da linguagem de programação identificada.

What-input: linguagem de programação identificada na atividade [SPEC-LANG] Identify Programming Language.

What-output: análise do conjunto de tokens e gramática definida por essa linguagem.

Running example: a especificação da linguagem Kotlin poderia ser estudada.

4.4.2 [ASYL-ANSY] Implement Syntactic Analyzer

How: atividade que implementa um analisador capaz de mapear os caracteres de um código-fonte na linguagem de programação identificada para o conjunto de tokens dessa linguagem e, a partir desses tokens, mapear para a gramática dessa linguagem. Se a linguagem de programação identificada já tiver um analisador sintático disponível, esta atividade poderá ser otimizada reutilizando este analisador.

Primeiro, deve-se implementar o mapeamento do conjunto de caracteres representados na linguagem de programação para o conjunto de tokens da linguagem, atribuindo seus tipos correspondentes. O mapeamento deve ser realizado usando uma estratégia correspondente à linguagem identificada. Normalmente, o espaço em branco e o terminador de linha são estratégias usadas para esse fim.

Segundo, deve-se implementar o mapeamento do conjunto de tokens para a gramática da linguagem de programação, atribuindo uma estrutura reconhecida. O mapeamento deve ser realizado visitando as regras gramaticais de cada linguagem identificada e vinculando-as à sequência de tokens correspondente.

Terceiro, deve-se implementar o processo de estruturação da linguagem composto pelos símbolos não terminais e terminais da linguagem, normalmente estruturado em uma árvore sintática abstrata (AST). Essa árvore é uma estrutura de dados que compreende um ou mais nós organizados hierarquicamente a partir de um nó raiz, todos os quais, com exceção do nó raiz, têm um único pai. Nós que contêm filhos são definidos como nós internos e os que não contêm filhos como nós folhas.

A ênfase desta atividade é implementar um analisador com alta precisão e, portanto, é altamente recomendável reutilizar um analisador reconhecido na comunidade para otimizar o tempo de aplicação do método e garantir a qualidade da análise.

Who: engenheiro de software reusa ou implementa o analisador sintático para a linguagem de programação identificada.

What-input: tokens e a gramática da linguagem de programação estudada na atividade [ASYL-LANG] Study Language.

What-output: analisador sintático dessa linguagem, que pode ser usado posteriormente para analisar qualquer código-fonte escrito nessa linguagem.

Running example: a ferramenta ANTLR[4] poderia ser utilizada para gerar um analisador sintático (*parser*) baseado na gramática existente de Kotlin.

[4] <https://www.antlr.org>

4.5 [ASEL] Language Semantic Abstraction Phase

A fase de Abstração Semântica da Linguagem ([ASEL] Language Semantic Abstraction Phase) objetiva representar o código-fonte em uma abstração de recursos semânticos, isto é, que considera o significado do código-fonte. Embora a [ASYL] Language Syntactic Abstraction Phase nos permita conhecer os elementos que compõem um código-fonte em determinada linguagem, ela não explica o papel que esses elementos desempenham no código. Por exemplo, o operando `private` em Java é representado pelo operador *classModifier* na [ASYL] Language Syntactic Abstraction Phase, mas isso não significa que a função desse operador seja modificar a visibilidade de uma classe. Embora o nome do operador seja sugestivo, ele nos diz apenas que é um nó folha do nó *classModifier* na árvore AST.

Portanto, esta fase pretende atribuir semântica aos elementos de uma linguagem de programação, observando que um token pode representar, em uma determinada linguagem de programação: (i) um significado único e independente de gramática, por exemplo, o token `class` que representa um tipo de dados abstrato na linguagem Java; ou (ii) significados diferentes dependendo da gramática, por exemplo, o token `final` que pode representar uma classe não extensível, um método não substituível ou uma constante (variável não modificável) em Java, dependendo de onde está sendo usado.

As atividades [ASEL-LANG] Map Language to Ontology e [ASEL-ANSE] Implement Semantic Analyzer devem ser executadas para a linguagem identificada na atividade [SPEC-LANG] Identify Programming Language. Observe que esta fase produz artefatos específicos para a linguagem de programação identificada, no entanto, novas linguagens de programação podem ser adicionadas em novos ciclos de vida do método.

4.5.1 [ASEL-LANG] Map Language to Ontology

How: atividade que mapeia a sintaxe de uma linguagem de programação para o significado do subdomínio, a fim de atribuir semântica ao código-fonte.

Esse mapeamento é realizado de maneira conceitual, vinculando os conceitos definidos na ontologia desenvolvida na atividade [ASES-ONTO] Build Subdomain Ontology com os símbolos da linguagem de programação identificada na atividade [SPEC-LANG] Identify Programming Language. Vale ressaltar que: (i) nem todos os símbolos da linguagem de programação serão mapeados para a ontologia, podendo corresponder exatamente a um, vários ou nenhum conceito semântico da ontologia; e (ii) nem todos os conceitos da ontologia serão mapeados para a linguagem de programação, podendo corresponder exatamente a um, vários ou nenhum símbolo da linguagem.

A ênfase desta atividade é mapear corretamente a representação sintática da linguagem de programação com a representação semântica fornecida pela ontologia. Como

a representação sintática se concentra na estrutura da linguagem, o mapeamento desses elementos para uma representação semântica permite uma maior compreensão do subdomínio.

Who: engenheiro de software mapeia a representação sintática da linguagem para a representação semântica do subdomínio.

What-input: ontologia de referência elaborada para o subdomínio na atividade [ASES-ONTO] Build Subdomain Ontology e a gramática da linguagem de programação estudada na atividade [ASYL-LANG] Study Language.

What-output: mapeamento da ontologia de referência para a linguagem de programação identificada.

Running example: poderia ser produzida uma planilha, associando cada símbolo da linguagem Kotlin aos conceitos da ontologia sobre a camada de apresentação dos aplicativos Android (e demais ontologias reutilizadas).

4.5.2 [ASEL-ANSE] Implement Semantic Analyzer

How: atividade que codifica o mapeamento elaborado na atividade [ASEL-LANG] Map Language to Ontology em um artefato de software, a fim de produzir uma ontologia operacional instanciada a partir de arquivos de código-fonte escritos na linguagem de programação identificada na atividade [SPEC-LANG] Identify Programming Language.

Primeiro, deve-se implementar o mapeamento do conjunto de símbolos da linguagem representado na estrutura sintática para o conjunto de conceitos da ontologia. O mapeamento deve ser realizado usando uma estratégia correspondente ao analisador sintático usado. Normalmente, métodos são usados para percorrer uma árvore AST procurando uma estrutura sintática que corresponde ao conceito da ontologia.

Segundo, deve-se implementar o processo de instanciação da ontologia. Uma vez realizado o mapeamento, o código-fonte correspondente a cada estrutura sintática é mapeado para uma instância desse conceito na ontologia. Observe que o conceito da ontologia pode ser instanciado parcial ou completamente pelo código-fonte, dependendo da linguagem.

A ênfase desta atividade é implementar um analisador semântico fiel ao mapeamento conceitual realizado na atividade [ASEL-LANG] Map Language to Ontology, mas também relacionado ao desempenho desse analisador.

Who: engenheiro de software implementa o analisador semântico para a linguagem de programação identificada.

What-input: tokens e a gramática da linguagem de programação estudada na atividade [ASYL-LANG] Study Language, a ontologia elaborada na atividade [ASES-

ONTO] Build Subdomain Ontology e o seu mapeamento elaborado na atividade [ASEL-LANG] Map Language to Ontology.

What-output: analisador semântico dessa linguagem, que pode ser usado posteriormente para analisar qualquer código-fonte escrito nessa linguagem.

Running example: poderiam ser implementados métodos `visitor` para percorrer o analisador sintático de Kotlin gerado com ANTLR a fim de produzir triplas RDF contendo instâncias da ontologia operacional a partir do arquivo OWL produzido na atividade [ASES-ONTO] Build Subdomain Ontology.

4.6 [PAPL] Application Purpose Phase

A fase de Propósito de Aplicação ([PAPL] Application Purpose Phase) objetiva definir o propósito de aplicação da representação semântica do código-fonte. As atividades realizadas nesta fase devem ser executadas para o subdomínio identificado, produzindo um único artefato, independentemente de linguagem de programação. Dado que as características do propósito não são esgotadas na atividade [SPEC-PERS] Identify Application Purpose, deve-se retornar àquela atividade sempre que alterações nas características forem necessárias. Observe que esta fase produz um artefato específico para o subdomínio identificado, no entanto, novos subdomínios podem ser adicionados em novos ciclos de vida do método.

4.6.1 [PAPL-FORM] Formalize Purpose

How: atividade que formaliza o propósito e as características a serem aplicadas ao código-fonte, seguindo a definição estabelecida na atividade [SPEC-PERS] Identify Application Purpose e a representação ontológica construída na atividade 4.3.

Por exemplo, para o propósito de *bad smell*, pode-se formalizar *smell* na forma de consulta semântica, ou seja, consultas elaboradas sobre os termos da ontologia do subdomínio em uma linguagem de consulta aplicável em ontologia operacional. Como a maioria das regras existentes na literatura não é baseada em uma ontologia, esta atividade pode exigir adaptações ou construção de novas regras.

Uma vez que a formalização é baseada na ontologia, ela é facilmente personalizada, atualizada e independente da linguagem de programação. Por outro lado, o conhecimento da ontologia do subdomínio é indispensável para lidar com a perspectiva.

A ênfase desta atividade é formalizar o propósito da aplicação a partir dos conceitos da ontologia para fornecer interoperabillidade, maior flexibilidade e personalização.

Who: engenheiro de ontologias formaliza as características do propósito.

What-input: ontologia operacional do subdomínio criada na atividade [ASES-ONTO] Build Subdomain Ontology e as características definidas na atividade [SPEC-PERS] Identify Application Purpose.

What-output: propósito de aplicação formalizada.

Running example: consultas SPARQL para cada *code smell* identificado na atividade [SPEC-PERS] Identify Application Purpose poderiam ser escritas.

4.6.2 [PAPL-IMPL] Implement Purpose

How: atividade que implementa uma solução tecnológica a partir do propósito formalizado, a fim de identificar as instâncias dessa ontologia que correspondem a este propósito.

Portanto, uma única ontologia é usada para identificar diferentes propósitos de aplicação e uma única solução de aplicação é usada para identificar o mesmo propósito em diferentes linguagens de programação.

A ênfase desta atividade é desenvolver uma solução tecnológica que utilize a representação semântica do código-fonte para um propósito de aplicação interoperável entre códigos-fonte de diferentes linguagens de programação.

Who: engenheiro de software implementa solução tecnológica segundo o propósito de aplicação.

What-input: analisador sintático construído na atividade [ASYL-ANSY] Implement Syntactic Analyzer, o analisador semântico construído na atividade [ASEL-ANSE] Implement Semantic Analyzer e o propósito formalizado na atividade [PAPL-FORM] Formalize Purpose..

What-output: implementação do propósito de aplicação identificado na atividade [SPEC-PERS] Identify Application Purpose e formalizado na atividade [PAPL-FORM] Formalize Purpose.

Running example: algum tipo de API OWL seria usada para implementar um programa que lê arquivos Kotlin relativos a aplicativos Android, produzindo instâncias da ontologia operacional e executando as consultas sobre essas instâncias, a fim de identificar quais pontos no código apresentam os *smells* especificados.

4.7 Considerações Finais

Este capítulo apresentou o método OSCIN para interoperabilidade de código-fonte baseada em ontologia e que se apoia nos pilares de subdominio de código-fonte, linguagem de programação e código-fonte. As fases definidas pelo método OSCIN permitem que

o código-fonte de diferentes linguagens de programação seja representado em modelos semânticos e aplicado para diferentes propósitos, possibilitando sua interoperabilidade.

Os capítulos seguintes apresentam informações complementares, aplicações práticas e análises do método OSCIN. Na sequência (Capítulo 5) é apresentado o OSCINF, *framework* que visa fornecer um conjunto de soluções para apoiar a aplicação do método OSCIN. O capítulo posterior (Capítulo 6) descreve algumas aplicações e avaliações sobre o método. Finalmente, o Capítulo 7 apresenta trabalhos relacionados ao método OSCIN, considerados instâncias do método, porém com diferenças significativas na sua aplicação.

5 Framework para Interoperabilidade Semântica de Código-Fonte baseada em Ontologia

Neste capítulo apresentamos o *framework* OSCINF — *Ontology-based Source Code Interoperability Framework*, que visa fornecer um conjunto de soluções para apoiar a aplicação do método OSCIN em diferentes subdomínios de código-fonte, linguagens de programação e propósitos de aplicação. O *framework* OSCINF segue as fases definidas pelo método OSCIN (apresentado no Capítulo 4).

A Figura 18 mostra como o *framework* OSCINF aplica o método OSCIN, cujas atividades são destacadas com seus retângulos preenchidos na cor preta. Nesse sentido, a figura é uma extensão da Figura 17 apresentada no Capítulo 4 e mantém a mesma semântica de seus símbolos.

Para [SPEC] Specification Phase, OSCINF utiliza a linguagem natural e a elaboração de um documento de especificação. Para [ASES] Subdomain Semantic Abstraction Phase aplica-se o método SABiOS (apresentado na Seção A) para a construção de uma rede de ontologias de código-fonte (como, por exemplo, a SCON, apresentada no Capítulo 3) em linguagem OWL, baseada na ontologia de fundamentação UFO. Para [ASYL] Language Syntactic Abstraction Phase reutiliza a ferramenta ANTLR e sua gramática para elaborar um gerador de analisador sintático (`ASYLGenerator.java`), baseado na linguagem Java. Para [ASEL] Language Semantic Abstraction Phase define e aplica uma gramática para o mapeamento entre elementos da linguagem e conceitos da ontologia, implementando um gerador de analisador semântico em linguagem Java (`ASELGenerator.java`). Por fim, para [PAPL] Application Purpose Phase utiliza-se a linguagem SPARQL e o *framework* Jena para formalizar o propósito, bem como reutiliza os artefatos elaborados em [ASYL] Language Syntactic Abstraction Phase e [ASEL] Language Semantic Abstraction Phase para gerar um software de aplicação (`OSCINFEXE.zip`), a partir de um gerador de aplicação (`PAPLGenerator.java`), baseado em Java.

A seguir, descrevemos mais detalhes das fases que compõem o *framework* e apresentamos um caso de exemplo em que os *stakeholders* desejam detectar *code smell* de orientação a objetos em projetos Java.

5.1 SPEC - Specification Phase

A Fase de Especificação ([SPEC] Specification Phase) tem como objetivo entender as necessidades dos *stakeholders*, com relação ao subdomínio, linguagem de programação e

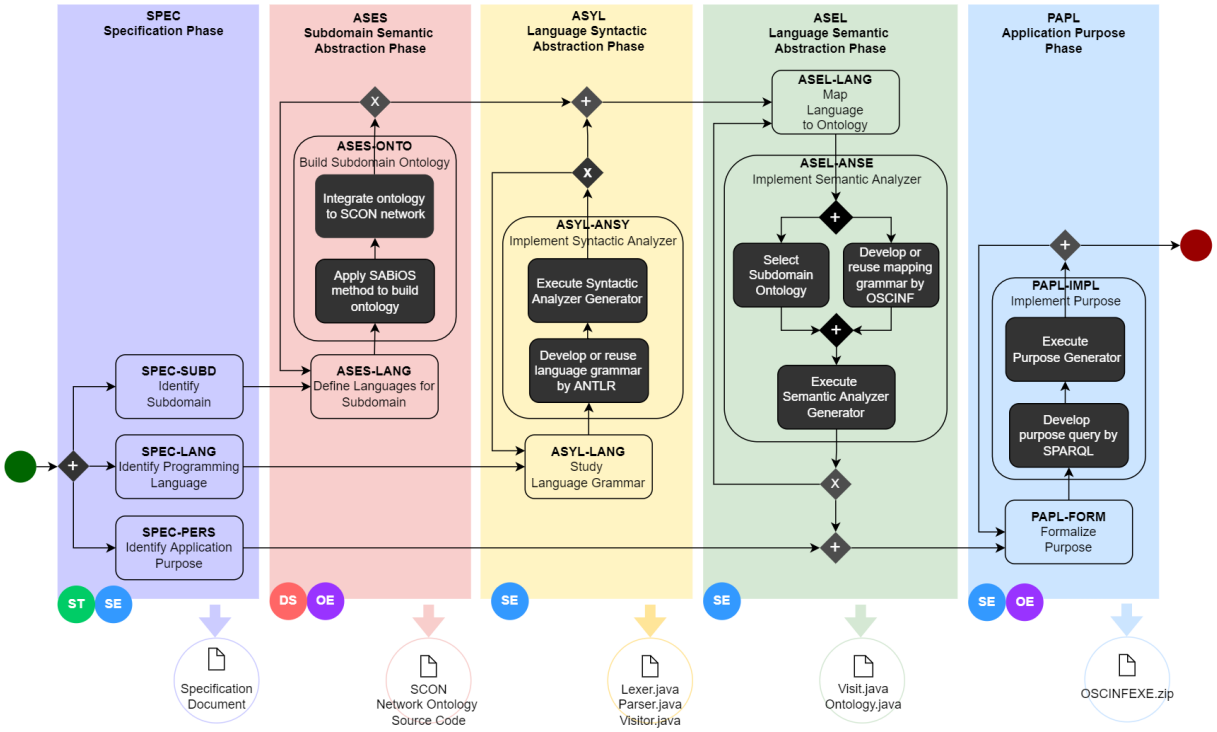


Figura 18 – Framework OSCINF a partir de OSCIN

propósito de aplicação. Assim, o engenheiro de software entrevista os *stakeholders* a fim de identificar o subdomínio de código-fonte a ser coberto (atividade [SPEC-SUBD] Identify Subdomain), a linguagem de programação que será suportada (atividade [SPEC-LANG] Identify Programming Language) e o propósito do qual o código pretende ser aplicado (atividade [SPEC-PERS] Identify Application Purpose).

O Framework OSCINF atualmente cobre os **subdomínios** de orientação a objetos e mapeamento objeto-relacional, a **linguagem de programação** Java e o *framework* de mapeamento objeto/relacional JPA e os **propósitos** de detecção de *code smell* e métricas orientadas a objeto. Ademais, OSCINF objetiva cobrir:

- Subdomínio de código-fonte relacionado aos paradigmas de programação (orientação a objetos, procedural, funcional, etc.), bem como seus *frameworks* (mapeamento objeto/relacional, *front-end*, injeção de dependência, etc.);
- Linguagens de Programação usualmente adotadas pela comunidade e relacionadas aos respectivos subdomínios (Java, C#, Python, JavaScript, etc.);
- Propósitos de Aplicação relacionados à análise estática de código-fonte (detecção de *smell* e detecção de *design patterns*) e medição de código.

OSCINF propõe o modelo apresentado na Tabela 9 como o Documento de Especificação esperado para a Fase de Especificação (preenchido conforme o caso de exemplo).

Tabela 9 – Documento de Especificação

Documento:	Especificação	**Versão:**	v.01

Subdomínio: Orientação a Objetos
A orientação a objetos é definida como um método de implementação de software no qual os programas são organizados como coleções cooperativas de objetos, cada um representando uma instância de alguma classe e cujas classes são membros de uma hierarquia de classes vinculada por relacionamentos de herança.

Linguagens de Programação:
Java

Propósito de Aplicação: Bad Smell

Bad Smell	**Descrição**
Long Parameter List	Método que possui uma lista muito longa de parâmetros, o que dificulta o uso e a compreensão. A longa lista de parâmetros é um smell, porque tudo o que um método precisa é passado através de parâmetros, fazendo com que a lista se torne inconsistente, difícil de manipular e constantemente alterada. Com o uso de objetos, o método pode passar apenas o suficiente por parâmetro e ser responsável por buscar o que precisa a partir dos objetos.

5.2 ASES - Subdomain Semantic Abstraction Phase

A Fase de Abstração Semântica do Subdomínio ([ASES] Subdomain Semantic Abstraction Phase) tem como objetivo representar o subdomínio de código-fonte em uma abstração semântica na forma de uma ontologia.

Para isso, devem ser selecionadas algumas linguagens de programação relacionadas ao subdomínio (atividade [ASES-LANG] Define Linguages for Subdomain) a fim de extrair o conhecimento de forma consensual de diferentes linguagens. OSCINF sugere selecionar as linguagens a partir dos índices sugeridos pelo método OSCIN (cf. Seção 4.3.1).

No caso de exemplo, o engenheiro de software selecionou cinco linguagens de programação relacionadas ao subdomínio de orientação a objetos: as linguagens Smalltalk e Eiffel foram selecionadas de acordo com sua *relevância* para o subdomínio, e as linguagens Java, Python e C++ foram selecionadas porque possuem a maior *popularidade* na comunidade de programação de acordo com os índices TIOBE,[1] IEEE Spectrum[2] e REDMONK.[3]

Para o desenvolvimento das versões de referência e operacional (atividade [ASES-ONTO] Build Subdomain Ontology) da ontologia do subdomínio, OSCINF adota o método SABiOS (apresentado brevemente a seguir) para guiar a construção da ontologia, a ontologia de fundamentação UFO (GUIZZARDI, 2005) para análise ontológica, a linguagem

[1] <http://tiobe.com> (Janeiro 2020)
[2] <https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019> (Setembro 2019)
[3] <https://redmonk.com/sogrady/2020/02/28/language-rankings-1-20/> (Janeiro 2020)

OntoUML (GUIZZARDI, 2005) para a representação gráfica da ontologia de referência e a linguagem OWL para codificação da ontologia operacional. As ontologias construídas são integradas à rede SCON, rede de ontologias de código-fonte (detalhada no Capítulo 3) que reúne e reutiliza fragmentos das ontologias da própria rede e de ontologias externas.

SABiOS - Abordagem Sistemática para Construção de Ontologias com Scrum

SABiOS — *Systematic Approach for Building Ontologies with Scrum* — é proposto com o intuito de guiar mais detalhadamente a construção de ontologias (de qualquer domínio) baseadas no método SABiO (apresentado na Seção 2.3), adotando princípios ágeis. Desta forma, o ciclo de vida de construção de ontologias é iterativo e incremental, com ciclos de curto período de tempo e revisão ao final de cada ciclo. O método adota uma abordagem de conceituação direcionada por questões de competência; por usuário, ao requerer o desenvolvimento colaborativo de especialistas de domínio e engenheiros de ontologia; e por dados, ao requerer o desenvolvimento baseado em fontes de dados consolidadas, assim como SABiO.

SABiOS cobre tanto a construção da ontologia de referência quanto a ontologia operacional, destacando a importância do consenso e argumentação sobre os conceitos a serem representados na ontologia, a construção de ontologias a partir do zero e a partir da reutilização de recursos ontológicos existentes e a aplicação de análise ontológica na construção da ontologia.

O ciclo de vida proposto pelo método SABiOS, representado na Figura 19, é composto por dois tipos de ciclos complementares: **ciclo de vida da ontologia**, apresenta o aspecto dinâmico do processo de construção de ontologia, expresso por meio de fases e iterações; e **ciclo de vida da fase**, apresenta o aspecto dinâmico das fases que compõem o ciclo de vida da ontologia, expresso por meio de processos, atividades e iterações.

O **ciclo de vida da ontologia** é formado por cinco fases que indicam a ênfase dos subprocessos em cada instante do ciclo de vida e o caráter evolutivo da construção de ontologias, a saber: A fase ***Conception*** enfatiza o propósito da ontologia de acordo com as partes interessadas. A fase ***Pre Reference*** enfatiza as questões de modelagem para a elaboração da ontologia de referência. A fase ***Reference*** enfatiza a elaboração da ontologia de referência, buscando enriquecer o conhecimento identificado na fase de concepção. Essa fase é considerada a mais crítica, pois investiga os requisitos a partir das partes interessadas e os representa como um modelo conceitual. Esta fase normalmente mantém o maior número de iterações. A fase ***Pre Operational*** enfatiza as questões de codificação para a elaboração da ontologia operacional. A fase ***Operational*** enfatiza a construção da ontologia operacional, implementando a ontologia em uma linguagem operacional.

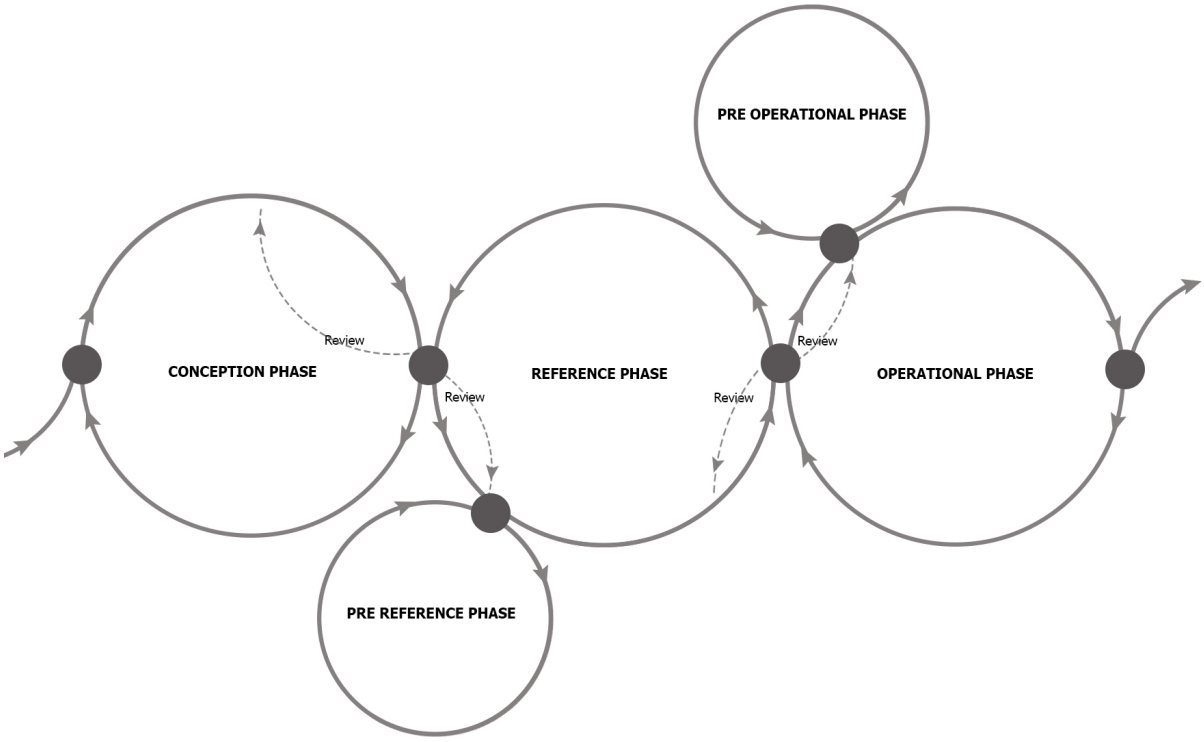


Figura 19 – Ciclo de vida da otologia do método SABiOS

O **ciclo de vida da fase** é formado por atividades que compõem os **Processos Principais**, que definem procedimentos principais para a construção da ontologia e os **Processos de Suporte**, que definem procedimentos auxiliares para os processos principais. Para cada fase do ciclo de vida da ontologia ocorrem diversos ciclos de vida da fase, executados a partir de princípios ágeis e inspirado no *framework* Scrum (SCHWABER; SUTHERLAND, 2020).

A Figura 20 ilustra os processos e atividades de SABiOS dentro do seu ciclo de vida. Dentre os processos principais, o processo **Requirement** (representado pela linha amarela) reconhece as necessidades que a ontologia deve representar, **Capture** (representado pela linha azul e verde) identifica a conceituação para atender aos requisitos, **Design** (representado pela linha rosa) especifica características tecnológicas para a ontologia e **Implementation** (representado pela linha roxa) codifica a ontologia em linguagem operacional.

Dentre os processos de suporte, o processo de **Knowledge Acquisition** extrai o conhecimento de diferentes fontes de conhecimento, **Documentation** registra os resultados das atividades, **Configuration Management** controla a versão e as alterações nos artefatos produzidos, **Evaluation**, avalia os resultados produzidos, **Reuse** reutiliza recursos ontológicos e não ontológicos existentes para construir a ontologia e **Publication** torna disponível a ontologia construída. As atividades dos processos de suporte de *Knowledge Acquisition* (representado pelo círculo pontilhado na cor rosa) e *Reuse* (representado pelo

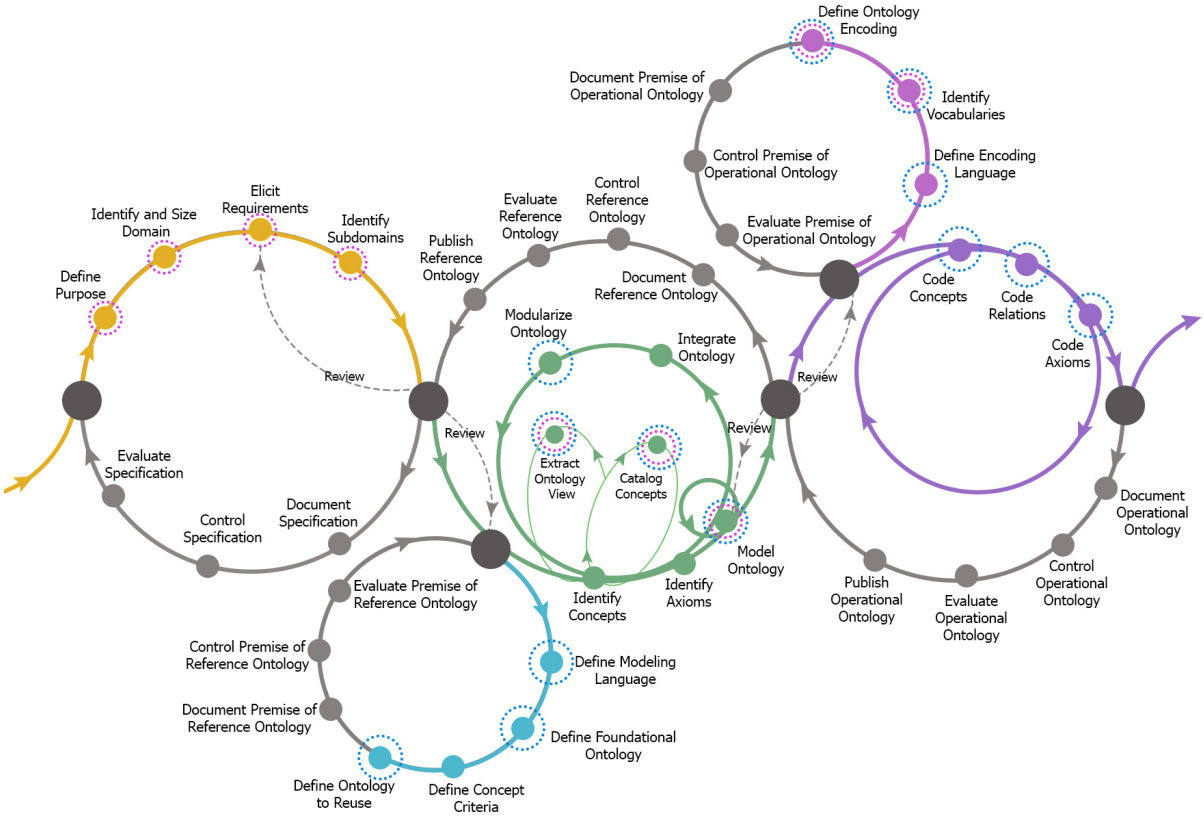


Figura 20 – Visão geral do método SABiOS

círculo pontilhado na cor azul) são realizadas simultaneamente às atividades dos processos principais.

As fases que compõem o método, bem como seus processos e atividades são detalhadas no Apêndice A.

No caso de exemplo, o método SABiOS é aplicado para o desenvolvimento de uma ontologia do subdomínio de orientação a objetos, chamada OOC-O: Ontologia de Código Orientado a Objetos (detalhada na Seção 3.3). Embora esse subdomínio seja um tópico muito estudado, não encontramos nenhuma ontologia que pudesse ser reutilizada de acordo com os requisitos esperados; portanto, nesse caso, apenas reutilizamos ontologias no domínio de software para apoiar a construção de OOC-O. A Figura 21 mostra um fragmento da ontologia de referência de OOC-O, já apresentada na Seção 3.3, prefixado como `ooc-o` e destacado na cor amarela.

A ontologia operacional foi codificada na linguagem OWL e avaliada por meio de procedimentos de verificação e validação. A Listagem 5.1 apresenta um fragmento da ontologia operacional *OOC-O* para os conceitos destacados na Figura 21, prefixados como `ooc-o`.

Vale destacar que a ontologia construída reflete a conceituação do subdomínio de orientação a objetos, representado por conceitos como *Class*, *Member*, *Method* e *Attribute*.

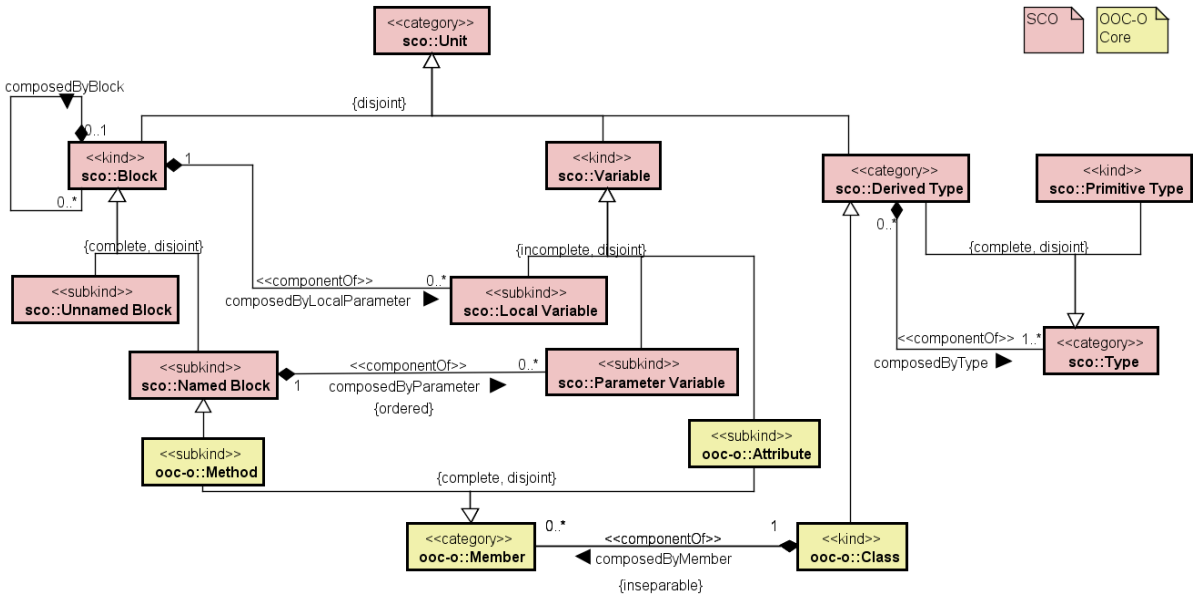


Figura 21 – Fragmento da ontologia de referência OOC-O.

Caraterísticas de *bad smells* não são representadas na ontologia, em vez disso, essas informações serão extraídas dos conceitos do subdomínio, por exemplo, Method *is typeOf* Member e Member *is componentOf* Class pode refletir os métodos que compõem uma classe.

5.3 ASYL - Language Syntactic Abstraction Phase

A Fase de Abstração Sintática da Linguagem ([ASYL] Language Syntactic Abstraction Phase) tem como objetivo representar o código-fonte em uma estrutura sintática conforme sua linguagem de programação.

Para isso, o léxico e a sintaxe da linguagem especificada (atividade [SPEC-LANG] Identify Programming Language) são estudados (atividade [ASYL-LANG] Study Language) a partir de sua especificação.

No caso de exemplo, a linguagem Java foi estudada a partir da sua especificação (GOSLING et al., 2018). As seguintes definições foram adotadas em relação aos tokens: (i) Identificadores, sequência de letras e dígitos que não é uma palavra-chave ou um literal booleano ou nulo; (ii) *Keyword*, string reservada da linguagem, como `private`, `class` e `public`; (iii) Literais, representação de um tipo primitivo, string ou nulo, como `1`, `false` e `null`; (iv) Separadores, representação de pontuação como `(`, `[` e `@`; e (v) Operadores, representação de um operador como `=`, `!` e `?`.

A gramática da linguagem Java na forma de regras sintáticas é definida por uma notação na qual seu símbolo não terminal é representado pelos tokens listados acima e suas regras são representadas pela seguinte notação: (i) símbolos não terminais são definidos por

Listagem 5.1 – Fragmento da ontologia operacional OOC-O.

```
<owl:Class rdf:about="http://ooc-o#Class">
        <rdfs:subClassOf rdf:resource="http://sco#DerivedType"/>
        <rdfs:comment rdf:datatype="http://www.w3.org/2000/01/rdf-schema#
    Literal"> An abstract-definition element in the OO programming language to
     express such definitions, that is, class is an abstract data type and a
    mechanism for defining an abstract data type in a program.  Class
    describes the attributes of its objects as well as the methods they can
    execute.  </rdfs:comment>
</owl:Class>
<owl:Class rdf:about="http://ooc-o#Member">
      <rdfs:comment rdf:datatype="http://www.w3.org/2000/01/rdf-schema#
    Literal"> Element which makes up a class, defined as a variable or method.
     </rdfs:comment>
</owl:Class>
<owl:Class rdf:about="http://ooc-o#Method">
      <rdfs:comment rdf:datatype="http://www.w3.org/2000/01/rdf-schema#
    Literal"> Named Block of a class that provides a way to define the
    behavior of a object, invoked when a message is received by the object. </
    rdfs:comment>
      <rdfs:subClassOf rdf:resource="http://sco#NamedBlock"/>
      <rdfs:subClassOf rdf:resource="http://ooc-o#Member"/>
      <owl:disjointWith rdf:resource="http://ooc-o#Attribute"/>
</owl:Class>
<owl:Class rdf:about="http://ooc-o#Attribute"/>
      <rdfs:comment rdf:datatype="http://www.w3.org/2000/01/rdf-schema#
    Literal"> Variable that belongs to a class. </rdfs:comment>
      <rdfs:subClassOf rdf:resource="http://sco#Variable"/>
      <rdfs:subClassOf rdf:resource="http://ooc-o#Member"/>
</owl:Class>
<owl:ObjectProperty rdf:about="http://ooc-o#componentOfClass">
      <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#
    FunctionalProperty"/>
      <rdfs:domain rdf:resource="http://ooc-o#Member"/>
      <rdfs:range rdf:resource="http://ooc-o#Class"/>
</owl:ObjectProperty>
```

um nome seguido por dois pontos; (ii) definições alternativas para símbolos não terminais são mostrados nas linhas sucessoras ou *any* é usado para listar todas as alternativas de uma alternativa não terminal; (iii) estrutura longa pode ser continuada em uma segunda linha recuando claramente a segunda linha; (iv) *{x}* indica zero ou mais ocorrências de x; (v) *[x]* indica zero ou uma ocorrência de x; (vi) *(one of)* denota uma definição alternativa; e (vii) *but not* denota expansões que não são permitidas.

A Listagem 5.2 ilustra algumas regras para declarar um método em Java, conforme a gramática estudada. Um *NormalClassDeclaration* é composto por zero ou mais ocorrências de um modificador de classe seguido de um token `class`, um tipo, zero ou uma ocorrência de parâmetros, zero ou uma ocorrência de superclasse, zero ou uma ocorrências de superinterface e um corpo de classe. Um *ClassBody* é definido como um *ClassBodyDeclaration*, que inclui, entre outras coisas, *MethodDeclarations*. Um *MethodDeclaration* é composto por zero ou mais ocorrências de um modificador de método seguido pelo cabeçalho e corpo do método. Um *MethodHeader* pode ser composto de parâmetros de tipo, zero ou mais ocorrências de anotação, um resultado, um declarador de método e zero ou uma ocorrência de lançamentos, e assim por diante.

Listagem 5.2 – Fragmento da gramática Java para declarar método.

```
1  NormalClassDeclaration:
2    {ClassModifier} class TypeIdentifier [TypeParameters]
3      [Superclass] [Superinterfaces] ClassBody
4  ClassBody:
5        { {ClassBodyDeclaration} }
6  ClassBodyDeclaration:
7        ClassMemberDeclaration
8        InstanceInitializer
9        StaticInitializer
10       ConstructorDeclaration
11       ClassMemberDeclaration:
12       FieldDeclaration
13       MethodDeclaration
14       ClassDeclaration
15       InterfaceDeclaration
16 ;
17 MethodDeclaration:
18       {MethodModifier} MethodHeader MethodBody
19 MethodHeader:
20       Result MethodDeclarator [Throws]
21       TypeParameters {Annotation} Result MethodDeclarator [Throws]
```

A implementação do analisador sintático (atividade [ASYL-ANSY] Implement Syntactic Analyzer) em OSCINF é realizada de forma automática, de modo que a partir de uma gramática, o respectivo analisador seja gerado automaticamente em linguagem Java, facilitando e otimizando os esforços de implementação.

Para isso, o gerador de analisador sintático (`ASYLGenerator.java`) reutiliza o gerador de parser ANTLR[4] e o respectivo arquivo gramatical da linguagem especificada. A gramática de algumas linguagens de programação são disponibilizadas gratuitamente no próprio repositório de códigos-fonte do ANTLR.[5]

O gerador carrega a gramática da linguagem especificada (conforme gramática ANTLR) e a processa para gerar as classes `Lexer`, `Parser` e `Visitor` da respectiva linguagem, em linguagem Java. A partir da classe `Lexer`, a sequência de caracteres *Unicode* será mapeada para linhas reconhecidas pelos terminadores de linha e para tokens reconhecidos nos espaços em branco, representados de maneira comum. Em seguida, a partir da classe `Parser`, um conjunto de métodos recursivos é definido a fim de procurar as regras gramaticais nesse conjunto de tokens. A partir da classe `Visitor`, a árvore AST será visitada começando pela regra *root* e percorrendo as regras descendentes até atingir os tokens (folhas).

Um único gerador de analisador sintático (`ASYLGenerator.java`) é utilizado para gerar o analisador sintático de diferentes linguagens de programação, conforme o arquivo gramatical de entrada. A Figura 22 apresenta o processo adotado pelo gerador de analisador sintático, baseado na ferramenta ANTLR.

[4] <https://www.antlr.org>
[5] <https://github.com/antlr/grammars-v4>

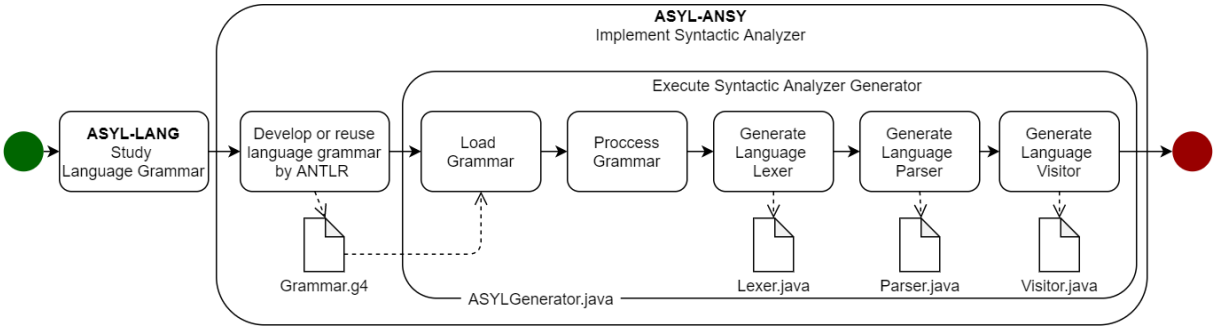


Figura 22 – Geração automatizada do analisador sintático

No caso de exemplo, com base na gramática estudada, o engenheiro de software utiliza o gerador `ASYLGenerator.java`[6] fornecido pelo framework OSCINF e o arquivo gramatical da linguagem Java `Java8.g4`[7] fornecido pela ferramenta ANTLR e exemplificado na Listagem 5.3 para a geração do analisador sintático da linguagem Java. Ao executar o gerador, as classes `Java8Lexer.java`, `Java8Parser.java` e `Java8Visitor.java` serão geradas. O repositório[8] do caso de exemplo apresenta as classes geradas para o caso de exemplo, a partir do gerador `ASYLGenerator.java` disponibilizado no repositório do projeto.[9]

5.4 ASEL - Language Semantic Abstraction Phase

A Fase de Abstração Semântica da Linguagem ([ASEL] Language Semantic Abstraction Phase) tem como objetivo representar o código-fonte em um modelo semântico, conforme o subdomínio especificado (atividade [SPEC-SUBD] Identify Subdomain).

Para isso, o engenheiro de software elabora um mapeamento entre os conceitos da ontologia e as estruturas sintáticas da linguagem de programação especificada (atividade [ASEL-LANG] Map Language to Ontology).

No caso de exemplo, são ilustrados alguns mapeamentos relacionados à declaração de classe em linguagem Java, de acordo com a gramática analisada na atividade [ASYL-LANG] Study Language e a ontologia desenvolvida na atividade [ASES-ONTO] Build Subdomain Ontology: (i) o símbolo terminal *identifier* que compõe um símbolo não terminal *ClassDeclaration* é mapeado para o conceito Class da ontologia; o símbolo terminal *keyword* que compõe um símbolo não terminal *ClassModifier* de um *ClassDeclaration* é mapeado de acordo com o seguinte: (ii) se o símbolo do terminal for igual a *public*, *private* ou *protected*, ele será mapeado para o conceito de Visibility da ontologia; (iii) ou se for igual a *abstract*, será mapeado para o conceito Abstract Class; (iv) ou se for igual a *final*, será mapeado

[6] <https://github.com/camila-aguiar/OSCINF>
[7] <https://github.com/antlr/grammars-v4/tree/master/java/java8>
[8] <https://github.com/camila-aguiar/OSCINF-EXAMPLE>
[9] <https://github.com/camila-aguiar/OSCINF>

Listagem 5.3 – Fragmento da gramática da linguagem Java conforme ANTLR.

```
classDeclaration
      :     normalClassDeclaration
      |     enumDeclaration
      ;

normalClassDeclaration
      :     classModifier* 'class' Identifier typeParameters? superclass?
    superinterfaces? classBody
      ;

classBody
      :     '{' classBodyDeclaration* '}'
      ;

classBodyDeclaration
      :     classMemberDeclaration
      |     instanceInitializer
      |     staticInitializer
      |     constructorDeclaration
      ;

classMemberDeclaration
      :     fieldDeclaration
      |     methodDeclaration
      |     classDeclaration
      |     interfaceDeclaration
      |     ';'
      ;

methodDeclaration
      :     methodModifier* methodHeader methodBody
      ;
```

para o conceito NonExtendable Class.

A implementação do analisador semântico (atividade [ASEL-ANSE] Implement Semantic Analyzer) em OSCINF é realizada de forma automática, de modo que a partir da ontologia e de um arquivo de mapeamento, o respectivo analisador seja gerado automaticamente em linguagem Java. Para isso, o gerador de analisador semântico `ASELGenerator.java` carrega a ontologia `.owl` e gera a classe `Ontology.java`, contendo métodos gerados a partir das classes/propriedades da ontologia e regras para suas instanciações.

O gerador utiliza um arquivo de mapeamento (`Mapper.map`) elaborado manualmente a partir do mapeamento conceitual entre elementos da linguagem e conceitos da ontologia, cuja estrutura do mapeamento é representada na Figura 23. A notação OSCINF para o mapeamento é apresentada na Tabela 10, seguida das respectivas regras.

Tabela 10 – Notação OSCINF

elemento identificador	Definido por um nome único seguido por dois pontos, que retrata a declaração de um método visitor ou não visitor (seguindo por ponto de esclamação) ou chamada de método (seguindo por asterisco *).
premissa	Premissas do elemento identificador são definidas por uma *hash* # seguida pelo conceito referência. Premissas são conceitos referência que devem ser verificados antes do processamento do elemento.

mapeamento classe	Mapeamentos de classes são definidos por colchetes [], seguindo o padrão [conceito referência] [conceito ontologia] [nome elemento] [nome conceito] [restrição] [propriedade]
mapeamento propriedade	Mapeamentos de propriedades são definidos por colchetes [], seguindo o padrão [propriedade] [conceito domínio] [conceito range] [restrição]
subelemento identificador	Elementos declarados que são reutilizados na definição de outro elemento nas linhas sucessoras, contendo o nome do elemento seguido por dois pontos, atuando como chamadas de método.
retorno	Retorno do método é definido pelo operador `=` seguido pela *keyword* `null` ou `child`, quando o método deve percorrer seus nós internos na estrutura da árvore.

Para o mapeamento de propriedades, OSCINF adota a notação apresentada na Tabela 11, seguindo o padrão [propriedade] [conceito domínio] [conceito range] [restrição].

Tabela 11 – Notação OSCINF para mapeamento de propriedade

propriedade	Propiedade da ontologia mapeada para o elemento da linguagem.
dominio	Conceito referência definido como domínio da propriedade.
range	Conceito referência definido como range da propriedade.
restrição	Restrição verificada antes de associar a propriedade ao elemento. A restrição é definida por três partes [tipoRestrição conceito1 conceito2], separadas por espaço, onde o tipoRestrição <VERIFY> verifica se o conceito1 é instância do conceito2.

Para o mapeamento de classe, OSCINF adota a notação apresentada na Tabelas 12, seguindo o padrão [conceito referência] [conceito ontologia] [nome elemento] [nome conceito] [restrição] [propriedade].

Tabela 12 – Notação OSCINF para mapeamento de classe

conceito referência	Conceito da ontologia que será referência para o elemento.
conceito ontologia	Conceito da ontologia mapeado para o elemento identificador da linguagem, podendo ser um ou mais conceitos, separados por vírgula.
nome elemento	Sequencia de chamadas de métodos ou atributos da linguagem que define o nome do elemento, separado por ponto final. O nome do elemento pode conter: [vazio]: indica que será usado o nome do conceito; [+]: indica um loop na sequência, tal como SEQ.SEQ.SEQ+.SEQ.SEQ.SEQ+ onde a sequência tem um loop no terceiro e sexto SEQ; [+options]: indica um loop na sequência com opções condicionais do nome; [*]: indica um loop recursivo na sequência, tal como SEQ*.SEQ onde a sequência tem um loop no primeiro SEQ e visita seus nós sucessores; [/]: indica o valor default que será usado caso o nome declarado não exista;
nome conceito	Sequencia formada por três partes, separadas por vírgula, no formato prefixo, nome, sufixo, que define o nome do conceito na ontologia, ou seja, seu URI. O nome do conceito pode conter: ["string"]: string utilizada no nome do conceito; [conceito referência]: nome do conceito referência; [palavra reservada]: palavra reservada CONT que conta a quantidade de instâncias com esse nome na ontologia e adiciona ao nome do conceito; [ROOT]: nome do conceito referência do próprio elemento.
restrição	Restrições verificadas antes de associar o conceito ao elemento. A restrição é definida por três partes [tipoRestrição propriedade conceitoOntologia], separadas por espaço.

propriedade	Propriedades associadas ao conceito instanciado. A propriedade é definida por duas partes [propriedade conceitoOntologia], separadas por espaço.

O gerador carrega o mapeamento `.map` e o processa para gerar a classe `Visit.java`, contendo os métodos visitors da respectiva linguagem e suas regras de mapeamento para a instanciação da ontologia, em linguagem Java. O gerador utiliza a classe ANTLR `Visitor` para definir um conjunto de métodos *visitors* na estrutura sintática da linguagem e a classe Jena `OntModel` para manipular a ontologia operacional. A partir da classe `Visit`, a árvore AST será visitada começando pela regra *root* e percorrendo as regras descendentes até atingir os tokens (folhas), conforme as regras definidas no mapeamento para a instanciação da ontologia.

Um único gerador de analisador semântico (`ASELGenerator.java`), fornecido pelo OSCINF, é utilizado para gerar o analisador semântico de diferentes linguagens de programação e diferentes subdomínios, conforme o arquivo de mapeamento de entrada. A Figura 24 apresenta o processo adotado pelo gerador de analisador semântico.

No caso de exemplo, o engenheiro de software utiliza o gerador `ASELGenerator.java` para a geração da classe `Java8Ontology.java` a partir da ontologia de orientação a objetos `OOC-O.owl` (atividade [ASES-ONTO] Build Subdomain Ontology). A Listagem 5.4 apresenta a definição do conceito Class e da propriedade visibleBy da ontologia owl (linhas 1-15), que gera automaticamente o método addClass e addvisibleBy em Java (linhas 16-27). O método addClass recebe como entrada o nome do indivíduo (URI) e adiciona o indivíduo como uma instância de Class, retornando o objeto instanciado. O método addvisibleBy recebe como entrada os objetos instanciados na ontologia que atuam como domínio e imagem (*range*) da propriedade visibleBy e adiciona essa propriedade ao objeto domain, retornando o objeto domain.

O engenheiro de software elabora o arquivo de mapeamento `Java8.OOC-O.map` para a linguagem Java e o subdomínio de orientação a objetos, conforme a notação OSCINF. A Listagem 5.5 apresenta um fragmento do mapeamento que define o elemento `NormalClassDeclaration` e `ClassModifier` conforme a notação já descrita nesta seção.

`NormalClassDeclaration` é declarado como um método visitor que percorre os nós internos (=child) da árvore, declarando os mapeamentos sintetizados na Tabela 13. Em sua declaração, chama os métodos `ClassModifier`, `Superclass` e `TypeParameters` que são declarados fora do método `NormalClassDeclaration`.

Tabela 13 – Mapeamento de classes e propriedades do elemento NormalClassDeclaration

elemento identificador	`NormalClassDeclaration`
mapeamento	Classe
conceito referência	`Class`
conceito ontologia	`Class`

nome elemento	`Identifier`, ou seja, o nome do elemento é recuperado pela sequência `NormalClassDeclaration().Identifier()`
nome conceito	`Module.<ROOT>`, ou seja, o nome do indivíduo desse conceito na ontologia é formado pelo nome do conceito referência `Module`, acrescido do ponto final e acrescido do nome do conceito `NormalClassDeclaration`
elemento identificador	`NormalClassDeclaration`
mapeamento	Propriedade
propriedade	`componentOfPhysicalModule`, ou seja, `NormalClassDeclaration` é mapeado para `Class`, que mantém um relação `componentOfPhysicalModule` entre `Class` e `Module`
domínio	`Class`
range	`Module`

`ClassModifier` é declarado como uma chamada de método visitor que não percorre os nós internos, declarando os mapeamentos sintetizados na Tabela 14.

Tabela 14 – Mapeamento de classes e propriedades do elemento ClassModifier

elemento identificador	`ClassModifier`
mapeamento	Classe
conceito referência	`Visibility`
conceito ontologia	`Visibility`
nome elemento	`classModifier.PUBLIC` ou `classModifier.PROTECTED` ou `classModifier.PRIVATE`
nome conceito	`"Visbility".<ROOT>`, ou seja, o nome do indivíduo será formado pela string "Visibility", acrescido do ponto final e acrescido do nome do conceito `ClassModifier`
propriedade	`"visibleBy"Class`, ou seja, define a propriedade `visibleBy` entre `Class` e `ClassModifier`
elemento identificador	`ClassModifier`
mapeamento	Classe
conceito referência	`Class`
conceito ontologia	`AbstractClass` e `ExtendableClass`, ou seja, o elemento será instância de dois conceitos na ontologia
nome elemento	`classModifier.ABSTRACT`
nome conceito	`Class`, ou seja, o nome do indivíduo será o nome adotado para o conceito referência `Class`
elemento identificador	`ClassModifier`
mapeamento	Classe
conceito referência	`Class`
conceito ontologia	`ConcreteClass` e `NonExtendableClass`, ou seja, o elemento será instância de dois conceitos na ontologia
nome elemento	`classModifier.FINAL`
nome conceito	`Class`, ou seja, o nome do indivíduo será o nome adotado para o conceito referência `Class`
elemento identificador	`ClassModifier`
mapeamento	Classe
conceito referência	`Class`
conceito ontologia	`ConcreteClass` e `ExtendableClass`, ou seja, o elemento será instância de dois conceitos na ontologia
nome elemento	vazio, ou seja, o nome do elemento será o nome adotado para o conceito referência deste elemento
nome conceito	`Class`, ou seja, o nome do indivíduo será o nome adotado para o conceito referência `Class`

restrição	`<VERIFY> Class ExtendableClass,NonExtendableClass,NestedClass` adiciona o elemento como istancia de `ConcreteClass` e `ExtendableClass` apenas se a instância for da classe `Class` e não for das classes `ExtendableClass`, `NonExtendableClass` e `NestedClass`

Com base no mapeamento elaborado, o engenheiro de software utiliza o gerador `ASELGenerator.java` para a geração do analisador semântico da linguagem Java e do subdomínio de orientação a objetos, ou seja, `Java8Visit.java`.[10] A Figura 25 apresenta um fragmento do analisador semântico gerado automaticamente por meio do gerador `ASELGenerator.java` para o elemento `NormalClassDeclaration`, detalhado a seguir.

O analisador semântico gerado contém o método visitor para o elemento identificador `NormalClassDeclaration`, que atribui o texto do símbolo `Identifier` de um `NormalClassDeclaration` para o conceito de `Class` da ontologia, utilizando a classe `Java8Ontology` gerada a partir da ontologia. Adicionalmente, declara as chamadas para os subelementos `ClassModifier`, `Superclass` e `TypeParameters`.

A Figura 26 apresenta um fragmento do analisador semântico gerado automaticamente por meio do gerador `ASELGenerator.java` para o elemento `ClassModifier`, detalhado a seguir.

`ClassModifier` é um algoritmo que percorre as opções de `ClassModifier` verificando os mapeamentos definidos. Se a opção for `PUBLIC`, `PROTECTED` ou `PRIVATE`, o elemento é adicionado para a classe `Visibility`, com a propriedade `visibleBy` entre o conceito `Class` e `Visibility`. Se a opção for `ABSTRACT`, o elemento é adicionado para a classe `AbstractClass` e `ExtedableClass`. Se a opção for `FINAL`, o elemento é adicionado para a classe `ConcreteClass` e `NonExtedableClass`. Por fim, se o conceito instanciado for da classe `Class` e não for da classe `ExtendableClass`, `NonExtendableClass` e `NestedClass`, o conceito é instanciado para `ConcreteClass` e `ExtendableClass`.

5.5 PAPL - Application Purpose Phase

A Fase de Propósito de Aplicação ([PAPL] Application Purpose Phase) tem como objetivo representar o propósito de aplicação e suas catacterísticas.

Para isso, o engenheiro de ontologias formaliza o propósito como uma consulta SPARQL aplicável sobre a ontologia do subdomínio a fim de detectar instâncias da ontologia (atividade [PAPL-FORM] Formalize Purpose).

Para apoiar a formalização, OSCINF define um arquivo de declaração de consultas SPARQL (`Query.sparql`) contendo o identificador e a respectiva consulta.

No caso de exemplo, o engenheiro de ontologias formaliza o *smell Long Parameter*

[10] <https://github.com/camila-aguiar/OSCINF-EXAMPLE>

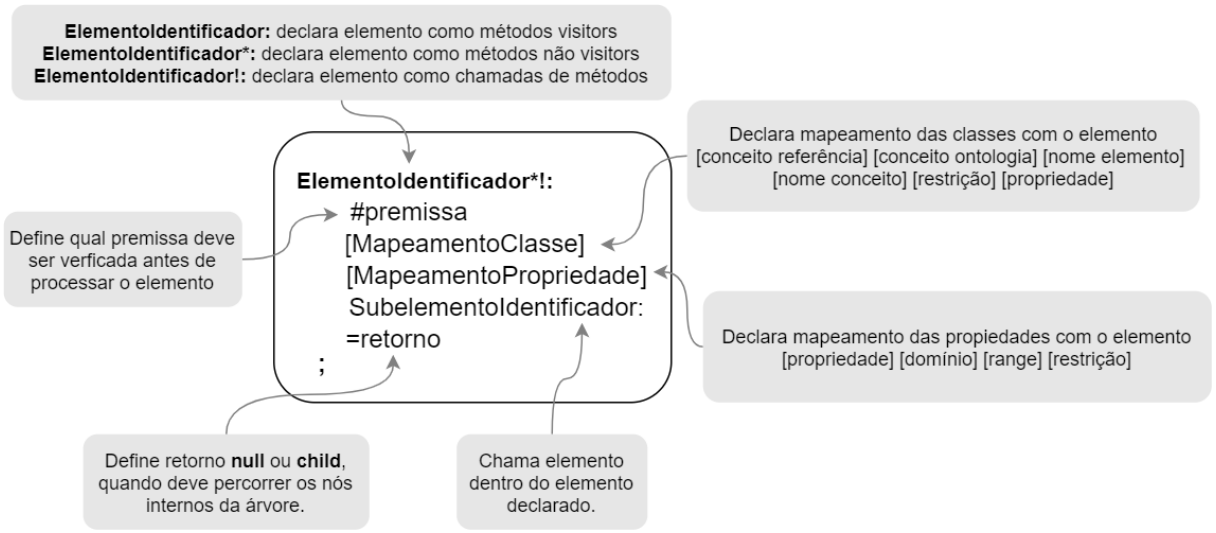


Figura 23 – Formato do arquivo de mapeamento

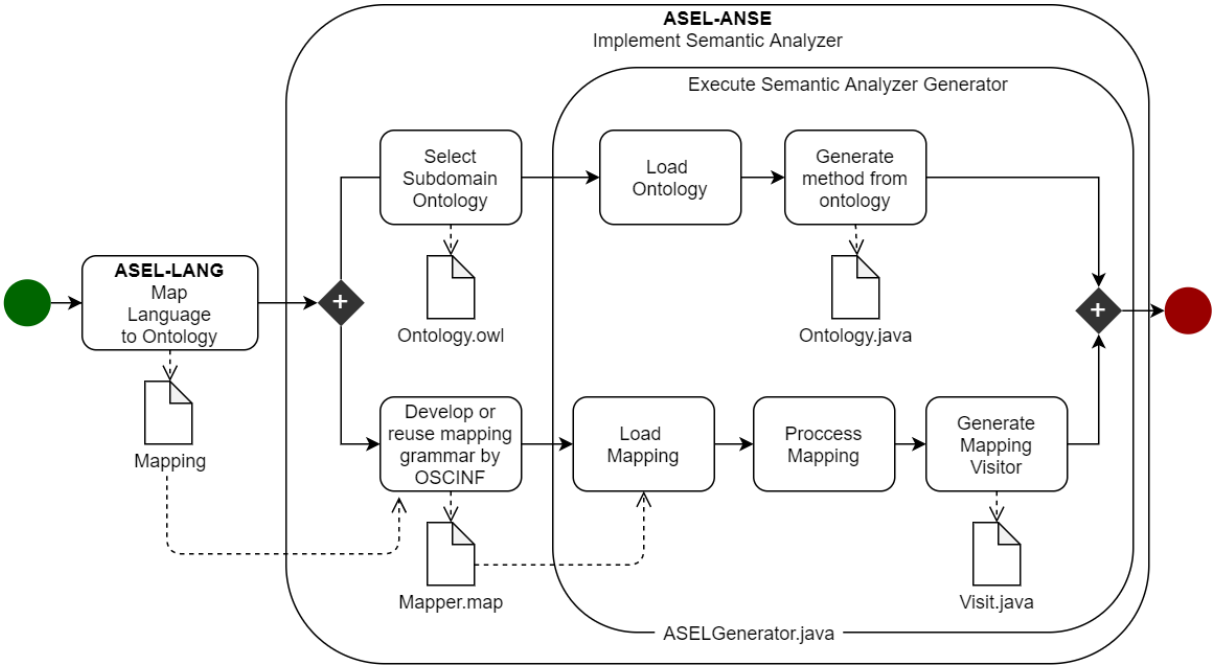


Figura 24 – Geração automatizada do analisador semântico

Listagem 5.4 – Fragmento da classe Java8Ontology.java.

```
1  ------------------------------------------------------------------------------
2  OOC-O.owl
3  ------------------------------------------------------------------------------
4  <owl:Class rdf:about="http://ooc-o#Class">
5          <rdfs:subClassOf rdf:resource="http://sco#DerivedType"/>
6          <rdfs:comment rdf:datatype="http://www.w3.org/2000/01/rdf-schema#Literal">
       An abstract-definition element in the OO programming language to express such
       definitions, that is, class is an abstract data type and a mechanism for
       defining an abstract data type in a program.    Class describes the attributes
       of its objects as well as the methods they can execute.    </rdfs:comment>
7  </owl:Class>
8  <owl:ObjectProperty rdf:about="http://sco#visibleBy">
9          <rdfs:domain rdf:resource="http://sco#Unit"/>
10         <rdfs:range rdf:resource="http://sco#Visibility"/>
11 </owl:ObjectProperty>
12 ------------------------------------------------------------------------------
13 Java8Ontology.java
14 ------------------------------------------------------------------------------
15 public static Individual addClass(String nameIndividual){
16        Individual objIndividual = null;
17        objIndividual = OSCINIndividual.findOrAddIndividual("Class", nameIndividual);
18        return objIndividual;
19    }
20 public static Individual addvisibleBy(Individual objDomain, Individual objRange){
21        objDomain = OSCINIndividual.addObjectProperty("visibleBy", objDomain,
       objRange);
22        return objDomain;
23    }
```

Listagem 5.5 – Fragmento do mapeamento da linguagem Java e ontologia OOC-O.

```
1  NormalClassDeclaration:
2        [Class] [Class] [Identifier] [Module.<ROOT>] [] []
3        ["componentOfPhysicalModule"] [Class] [Module] []
4        ClassModifier:
5        Superclass:
6        TypeParameters:
7        =child
8        ;
9
10 classModifier!:
11       [Visibility][Visibility][classModifier+{PUBLIC,PROTECTED,PRIVATE}] ["
       Visibility".<ROOT>] [] ["visibleBy" Class]
12       [Class] [Abstract_Class, Extendable_Class] [classModifier+{ABSTRACT}] [Class]
       [] []
13       [Class] [Concrete_Class, NonExtendable_Class] [classModifier+{FINAL}] [Class]
       [] []
14       [Class] [Concrete_Class, Extendable_Class] [] [Class] [<VERIFY> Class
       Extendable_Class,NonExtendable_Class,Nested_Class] []
15       =null
16       ;
```

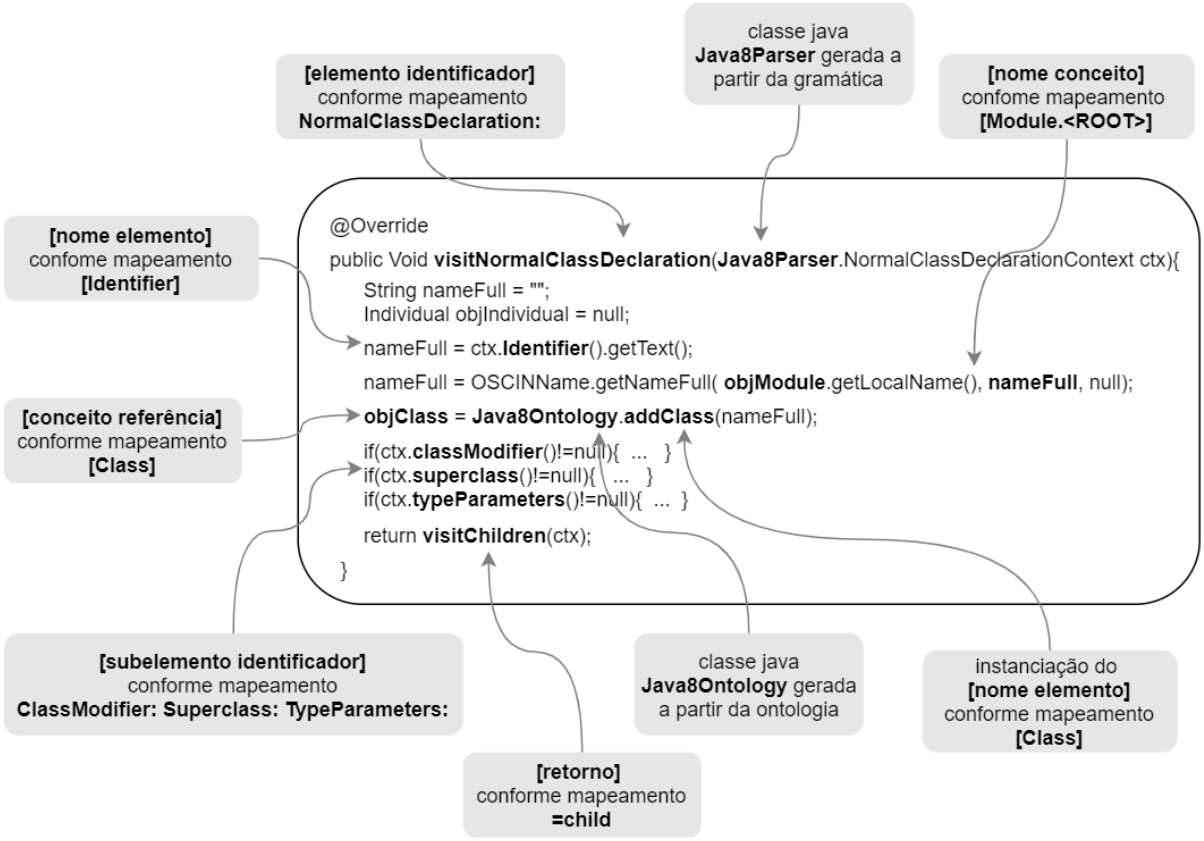


Figura 25 – Fragmento do analisador semântico gerado

List como sendo uma função membro de uma classe composta por mais de seis parâmetros. Mesmo para um *smell* muito discutido como *Long Parameter List*, encontramos diferentes regras de detecção na literatura, como método composto por mais de cinco (KIEFER; BERNSTEIN; TAPPOLET, 2007; DANPHITSANUPHAN; SUWANTADA, 2012) ou sete (KUMAR; SINGH, 2016) parâmetros.

As regras são traduzidas para a linguagem SPARQL usando os conceitos da ontologia, conforme mostrado na Listagem 5.6. A consulta usa os seguintes prefixos: `rdf` (<http://www.w3.org/1999/02/22-rdf-syntax-ns#>), `rdfs` (<http://www.w3.org/2000/01/rdf-schema#>) , `xsd` (<http://www.w3.org/2001/XMLSchema#>), `owl` (<http://www.w3.org/2002/07/owl#>) and `ooc-o` (<http://ooc-o#>). Esta consulta procura por função membro (linha 4) de uma classe (linha 3 e 6) cujo número de variáveis de parâmetro (linhas 5 e 7) é maior que seis (linha 10).

A Implementação do Propósito em OSCINF é realizada de forma automática (atividade [PAPL-IMPL] Implement Purpose) pelo gerador de aplicação `PAPLGenerator.java`,[11] fornecido pelo OSCINF. Para isso, o gerador reutiliza os artefatos `Lexer.java`, `Parser.java` e `Visitor.java` produzidos em ASYL - Language Syntactic Abstraction Phase e os artefatos `Visit.java` e `Ontology.java` produzidos em ASEL - Language Semantic Abstraction Phase para gerar a classe `Load.java` e a solução de aplicação `OSCINFEXE.zip`.

[11] <https://github.com/camila-aguiar/OSCINF>

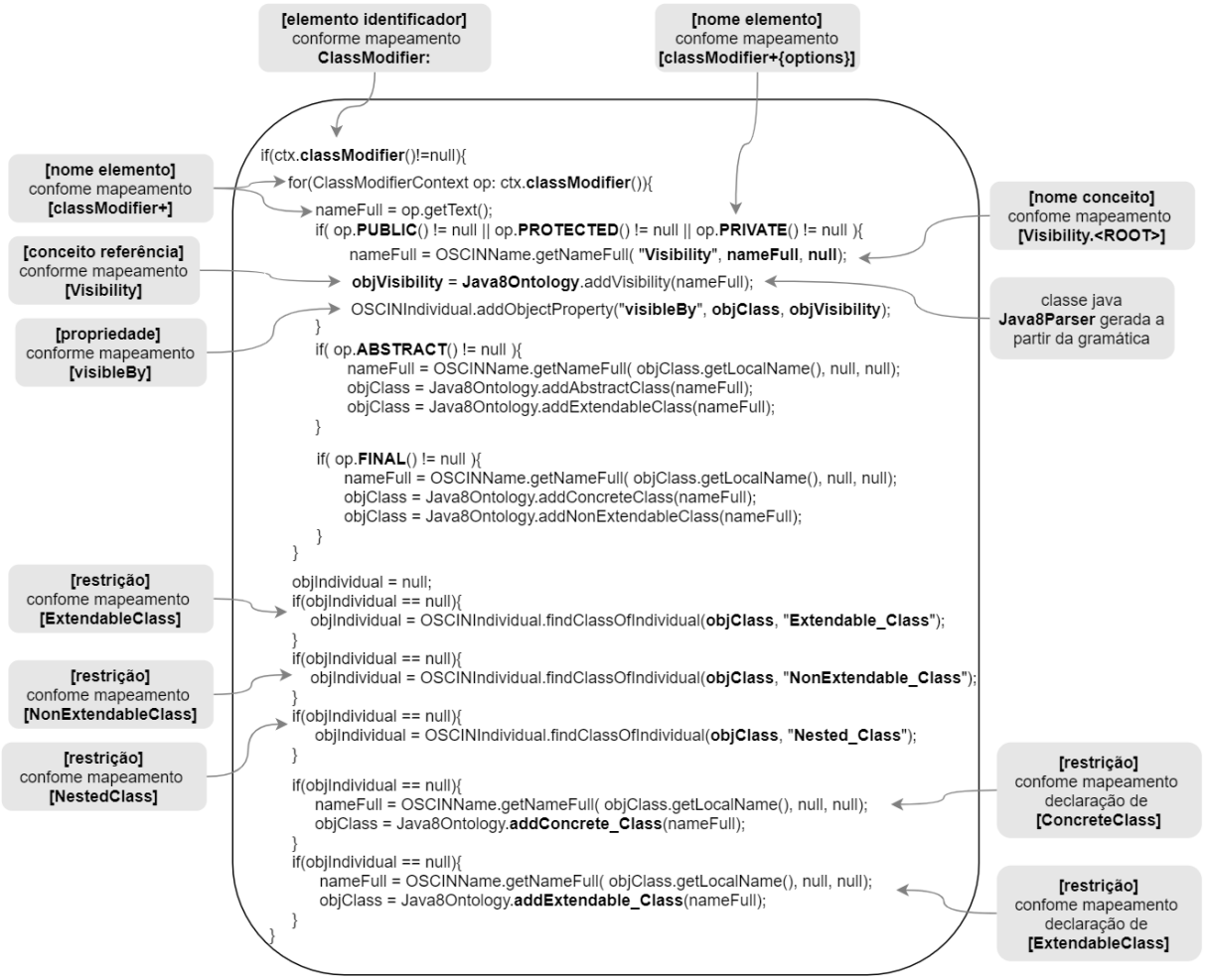


Figura 26 – Fragmento do analisador semântico gerado para o elemento ClassModifier

Listagem 5.6 – Formalização do smell *Long Parameter List* em SPARQL.

```
1 SELECT   ?class ?method (count( ?variable) as ?countVariable)
2 WHERE {
3   ?class rdf:type ooc-o:Class .
4   ?method rdf:type ooc-o:Method  .
5   ?variable rdf:type ooc-o:ParameterVariable .
6   ?method ooc-o:componentOfClass ?class .
7   ?variable ooc-o:componentOfMethod ?method
8  }
9 GROUP BY ?class ?method
10 HAVING (?countVariable > 6)
```

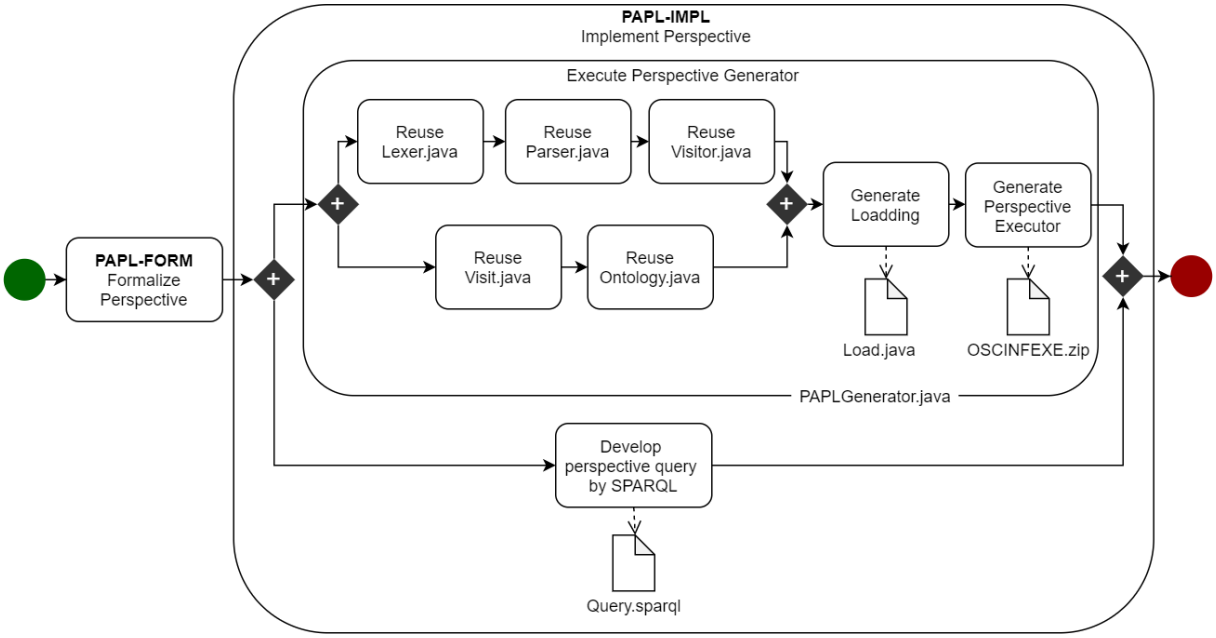


Figura 27 – Geração automática da Solução de Aplicação

A Figura 27 apresenta o processo adotado pelo gerador de aplicação. Um único gerador de aplicação (`PAPLGenerator.java`) é utilizado para gerar a solução de aplicação de diferentes subdomínios, linguagens de programação e propósitos, conforme o arquivo de entrada.

No caso de exemplo, o engenheiro de software reutiliza os artefatos `Java8Lexer.java`, `Java8Parser.java`, `Java8Visitor.java`, `Java8Visit.java` e `Java8Ontology.java` produzidos nas fases anteriores, elabora o arquivo de consultadas SPARQL `Query.sparql` e reutiliza o gerador `PAPLGenerator.java` fornecido pelo OSCINF. Ao executar o gerador, a solução de aplicação `OSCINFEXE.zip` será gerada.

5.6 Processando código-fonte com OSCINF

Dado que as fases OSCINF foram executadas e seus respectivos artefatos elaborados, é possível analisar e manipular diferentes códigos-fonte que atendam aos requisitos de SPEC - Specification Phase sem qualquer necessidade de programação por parte dos *stakeholders*.

A Figura 28 apresenta o processo de execução realizado pelo *stakeholder* para processar um dado código-fonte. Para isso, o *stakeholder* deve selecionar o código-fonte, a gramática da linguagem de programação, a ontologia do subdomínio e as consultas SPARQL do propósito de acordo com a linguagem de programação, o subdomínio e o propósito definidos na SPEC - Specification Phase.

A solução de aplicação `OSCINFEXE.zip` gerada em PAPL - Application Purpose Phase é compilada no arquivo `OSCINFEXE.jar` para execução dos códigos-fonte. Ao executar `OSCINFEXE.jar`, o código-fonte é carregado utilizando os artefatos produzidos durante

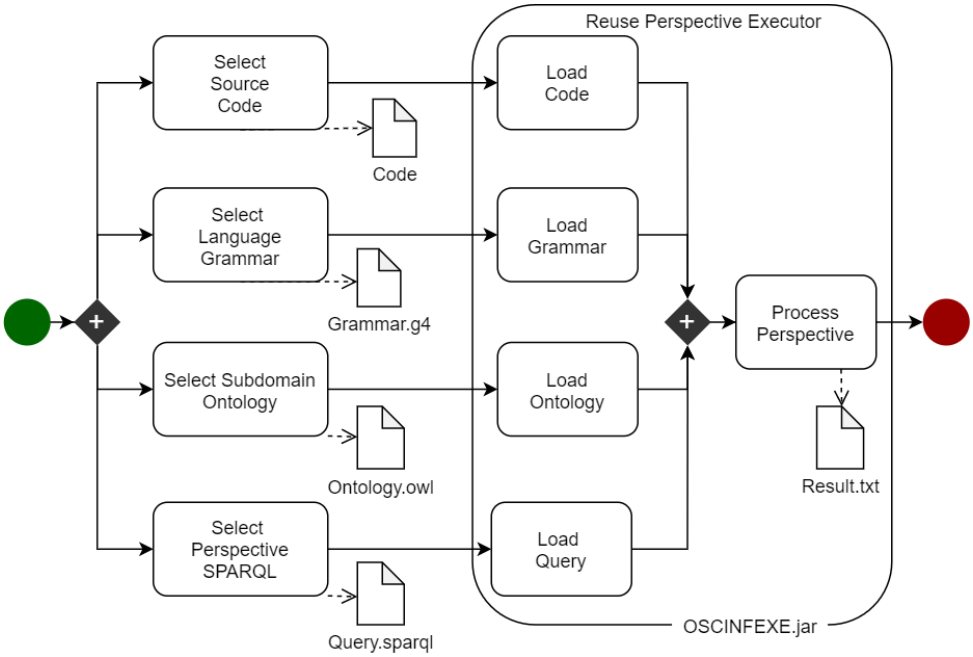


Figura 28 – Execução da Solução de Aplicação

as fases, a gramática é carregada utilizando ANTLR, a ontologia é carregada na classe `OntModel` utilizando Jena Framework[12] e o arquivo de consultas SPARQL é carregado com a classe `QueryExecution`. Assim, o código-fonte é carregado na estrutura sintática da linguagem, que é mapeada para a ontologia como instâncias de ontologia e, por fim, são aplicadas consultas na ontologia instanciada para analisar o propósito de aplicação no código-fonte.

Uma única solução de aplicação (`OSCINFEXE.jar`) é utilizada para processar diferentes códigos-fonte, conforme a especificação. Além disso, novas consultas sobre o propósito podem ser inseridas durante esta fase de execução.

5.7 Considerações Finais

Este capítulo apresentou o *framework* OSCINF para dar suporte à aplicação do método OSCIN. As soluções tecnológicas disponibilizadas pelo *framework* foram codificadas na linguagem Java, mas tais soluções suportam as várias linguagens de programação, dependendo dos artefatos de entrada de cada fase.

A partir do *framework* OSCINF é possível manipular e analisar códigos-fonte de diferentes linguagens de programação e subdomínios de código-fonte. Na ASES - Subdomain Semantic Abstraction Phase, a construção da ontologia é apoiada pelo método SABiOS (apresentado na Seção A) e reutiliza ontologias já construídas na rede de ontologias de código-fonte – SCON (apresentada no Capítulo 3), incluindo a ontologia de núcleo de

[12] <https://jena.apache.org>

código-fonte – SCO, que se faz importante para guiar as novas ontologias. Na ASYL - Language Syntactic Abstraction Phase, o gerador de analisador sintático abstrai as preocupações com a estrutura sintática da linguagem, possibilitando o reaproveitamento de gramáticas já escritas e a geração automática do analisador, sem a necessidade de qualquer codificação. Na ASEL - Language Semantic Abstraction Phase, o gerador de analisador semântico abstrai as preocupações do mapeamento da estrutura sintática da linguagem e dos conceitos da ontologia, possibilitando o mapeamento por meio de uma notação que gera automaticamente o analisador, sem a necessidade de qualquer codificação. Na PAPL - Application Purpose Phase, o gerador de aplicação disponível permite a definição de consultas SPARQL que são executadas na ontologia instanciada, sem a necessidade de qualquer codificação.

As seguir, o Capítulo 6 avalia a interoperabilidade de código-fonte com a aplicação do método OSCIN.

6 Avaliação da Interoperabilidade de Código-Fonte

Neste capítulo, apresentamos aplicações e avaliações a cerca do método OSCIN — *Ontology-based Source Code Interoperability*, descrito no Capítulo 4, para a interoperabilidade de código-fonte.

Neste sentido, apresentamos aplicações (i) de um experimento de laboratório para a aplicação de OSCIN na detecção de *bad smells*; (ii) da execução desta aplicação de detecção *smells* em um conjunto de projetos de software desenvolvidos pela comunidade; (iii) da aplicação de OSCIN no propósito de métricas de código; e (iv) da aplicação de OSCIN no propósito de migração de código. Por fim, a partir destas aplicações, avaliamos se o método OSCIN pode (i) ser operacionalizado em soluções de software, (ii) ser extensível para a inclusão de novos subdomínios, linguagens de programação e propósitos de aplicação, (iii) ser aplicado em projetos de software desenvolvidos pela comunidade e (iv) interoperar código-fonte de diferentes linguagens de programação.

As seções seguintes apresentam detalhes destes resultados.

6.1 Aplicação de OSCIN para Detecção de Smell

Esta seção apresenta os resultados de um experimento de laboratório no qual dois estudantes de pós-graduação (um deles a autora deste trabalho) aplicaram o método OSCIN para o propósito de detecção de *smell* de código em três cenários diferentes: código orientado a objetos em Java, código orientado a objetos em Python e código de mapeamento objeto/relacional em JPA/Java. Ou seja, OSCIN aplicado a diferentes subdomínios (orientação a objetos e mapeamento objeto/relacional) e linguagens de programação (Java e Python) no propósito de aplicação de detecção de *code smells*.

As seções 6.1.1, 6.1.2 e 6.1.3 descrevem, respectivamente, cada um destes cenários. Em seguida, a Seção 6.1.4 apresenta os resultados produzidos por estes três cenários, cada um em uma base de código selecionada para demonstração. Por fim, a Seção 6.1.5 apresenta resultados da execução do detector de *smells* dos dois primeiros cenários em 10 bases de código de projetos do mundo real, com grandes quantidades de arquivos, classes e métodos, para avaliação da escalabilidade do detector produzido.

6.1.1 Smells de Orientação a Objetos em Java

Para a aplicação de OSCIN no subdomínio de orientação a objetos, linguagem de programação Java e propósito de aplicação *bad smells*, utilizamos o *framework* OSCINF e os artefatos descritos como caso de exemplo no Capítulo 5.

Na SPEC - Specification Phase, identificamos o subdomínio orientado a objetos, a linguagem Java e o propósito de aplicação *bad smells*, observando que Java é a linguagem mais utilizada na literatura científica sobre detecção de *smells* em código-fonte. Na ASES - Subdomain Semantic Abstraction Phase, a ontologia de código-fonte orientado a objetos (OOC-O) é elaborada, baseada no método SABiOS, fundamentada em UFO e apoiada na ontologia de código-fonte (SCO) da rede de ontologias de código-fonte (SCON). Na ASYL - Language Syntactic Abstraction Phase, o analisador sintático é gerado a partir da gramática da linguagem Java. Na ASEL - Language Semantic Abstraction Phase, o analisador semântico é gerado a partir da ontologia e do mapeamento definido entre os elementos da linguagem e os conceitos da ontologia. Por fim, na PAPL - Application Purpose Phase, o executor de perspectiva para detectar *bad smell* a partir da ontologia de orientação a objetos instanciada com código-fonte em linguagem de programação Java é gerado.

6.1.2 Smells de Orientação a Objetos em Python

Para detectar *smell* orientados a objetos em Python, grande parte do trabalho realizado no cenário de Java (Seção 6.1.1) foi reutilizado. Na SPEC - Specification Phase, apenas a linguagem de programação foi alterada para Python. A ASES - Subdomain Semantic Abstraction Phase foi ignorada, pois OOC-O é reutilizada neste cenário.

Em seguida, na ASYL - Language Syntactic Abstraction Phase, o engenheiro de software estudou a especificação Python e, de forma análoga ao cenário Java, implementou um analisador usando ANTLR e um arquivo de gramática disponível para Python.

Também como antes, na ASEL - Language Semantic Abstraction Phase foi elaborado um mapeamento entre OOC-O e Python. Por exemplo: *Class* é mapeada para o símbolo terminal *name*; *Visibility* não é mapeada, uma vez que Python não trata esse aspecto; e *Member Procedure* é mapeada para o símbolo terminal *name* que constitui um símbolo não terminal de *funcdef*. Em seguida, um analisador semântico foi implementado em Java, novamente baseado em ANTLR e Jena.

Feito isso, a PAPL - Application Purpose Phase pode ser ignorada, pois a aplicação do propósito de detecção de *smells* para *Long Parameter List* já foi criada no cenário anterior.

6.1.3 Smells de Mapeamento Objeto/Relacional em Java

Neste cenário, na [SPEC] Specification Phase o subdomínio mudou para Mapeamento Objeto/Relacional (ORM) e a linguagem Java permaneceu, mas, mais especificamente, o padrão Java Persistence API (JPA) foi considerado. O propósito de aplicação de *code smell* permaneceu, mas o engenheiro de software especificou um *code smell* ORM a ser detectado, mais especificamente o *Multi Directed Table*. A Tabela 15 apresenta o documento de Especificação elaborado na fase de especificação.

Tabela 15 – Documento de Especificação

Documento:	Especificação	**Versão:**	v.01

Subdomínio: Mapeamento Objeto/Relacional
O mapeamento objeto/relacional define como a comunicação entre os paradigmas orientado a objetos e relacionais é realizada, automatizando a persistência dos dados do objeto em tabelas e colunas do banco de dados e a construção de objetos a partir dos dados recuperados por consultas ao banco de dados.

Linguagens de Programação:
Java

Propósito de Aplicação: Bad Smell

Bad Smell	**Descrição**
Multi Directed Table	Smell que aparece quando classes de entidade que não fazem parte da mesma hierarquia compartilham a mesma tabela de banco de dados. Este *smell* não causa erros de compilação ou execução, mas permite soluções que violam as regras do banco de dados, ou seja, ter algumas colunas na tabela se referindo a atributos de uma classe e outras colunas a atributos de outra classe não relacionada.

Na [ASES] Subdomain Semantic Abstraction Phase, *frameworks* ORM foram selecionados para o desenvolvimento da ontologia do subdomínio. Como no subdomínio de Orientação a Objetos, o engenheiro de ontologia decidiu se concentrar nas três linguagens mais populares do cenário OO (C++, Java e Python). Portanto, além do JPA, o engenheiro de ontologia selecionou os *frameworks* ORM Django e SQLAlchemy de acordo com sua popularidade para Python; e QxORM e ODB de acordo com sua popularidade para C++. Aplicando o método SABiO (FALBO, 2014), a Ontologia de Mapeamento Objeto/Relacional (ORM-O) (descrita na Seção 3.6) foi construída com base na ontologia OOC-O.

Como a linguagem Java já havia sido estudada e mapeada para OOC-O em um cenário anterior, nas [ASYL] Language Syntactic Abstraction Phase e [ASEL] Language Semantic Abstraction Phase o engenheiro de software estudou a especificação JPA[1] e a mapeou para ORM-O, usando uma arquitetura diferente do cenário de Orientação a

[1] <http://jcp.org/en/jsr/detail?id=338>

Objetos (descrito na Seção 6.1.1), adotando bibliotecas JavaParser[2] e OWLAPI.[3] Como o JPA usa anotações Java para realizar mapeamento objeto/relacional, uma vez que essas anotações são encontradas no código-fonte, os conceitos de ORM-O foram instanciados junto com os de OOC-O.

Então, na [PAPL] Application Purpose Phase, o engenheiro de ontologia formalizou o *smell Multi Directed Table*, conforme mostrado na Listagem 6.1. A consulta procura por uma classe de entidade (linha 3) que está relacionada (linha 4) a mais de uma classe (linha 7).

Listagem 6.1 – Formalização do smell *Multi Directed Table* em SPARQL.

```
1 SELECT ?table (count(?class) as ?countClasses)
2 WHERE {
3   ?table rdf:type orm-o:Entity_Table .
4   ?table orm-o:directly_related_to ?class   .
5  }
6 GROUP BY ?class
7 HAVING (?countClasses > 1)
```

Por fim, a aplicação do propósito foi implementada em Java reutilizando a biblioteca OWLAPI. A aplicação carrega a ontologia instanciada e aplica consultas SPARQL para detectar *smells* de *Multi Directed Table* em bases de código Java que usam JPA.

6.1.4 Detecção de *Smells* nos Cenários

Esta seção apresenta os resultados produzidos com o propósito de aplicação de detecção de *smells* para os cenários de orientação a objetos em Java (descrita na Seção 6.1.1), orientação a objetos em Python (descrita na Seção 6.1.2) e mapeamento objeto/relacional em Java (descrita na Seção 6.1.3).

Para isso, selecionamos arbitrariamente o projeto de software *Jena 2.6.3*,[4] escrito com orientação a objetos e linguagem Java; projeto Django,[5] escrito com orientação a objetos e linguagem Python; e projeto SchoolSample,[6] escrito com mapeamento objeto/relacional e linguagem Java.

A aplicação foi executada para o propósito de detecção *bad smells* sobre os códigos-fontes dos projetos de acordo com os *smells Long Parameter List* e *Multi Directed Table*. A Tabela 16 apresenta os resultados da detecção. A aplicação encontrou 6 smells do tipo *Long Parameter List* em Jena (Java) e 119 smells em Django (Python), dos quais, respectivamente, 83% e 50% dos casos são metódos construtores. No cenário de mapeamento

[2] <https://www.javaparser.org>
[3] <http://owlcs.github.io/owlapi>
[4] <https://github.com/JavaQualitasCorpus/jena-2.6.3>
[5] <https://github.com/django/django>
[6] <https://nemo.inf.ufes.br/projetos/o-scan>

Tabela 16 – Parte dos resultados da execução.

Projeto: Jena		**Versão:** 2.6.3
Arquivos: 914 | **Classes:** 1.315 | **Métodos:** 9.900 | **Tempo:** 5.226 seg		
Bad Smell: Long Parameter List		**Total:** 6
Classe	**Metodo**	**Parâmetros**
DBPropDatabase	DBPropDatabase	10
ArgDecl	ArgDecl	7
ModelLoader	loadModel	7
TokenMgrError	TokenMgrError	7
ElementLexer	ElementLexer	8
UUID_V1_Gen	generate	7
Projeto: Django		**Versão:** 3.1
Arquivos: 2.674 | **Classes:** 7.886 | **Métodos:** 24.995 | **Tempo:** 35.760 seg		
Bad Smell: Long Parameter List		**Total:** 119
Classe	**Método**	**Parâmetros**
InlineAdminFormSet	___init___	11
OperationTestCase	alter_gis_model	7
BooleanFieldListFilter	___init___	7
Serializer	serialize	9
Engine	___init___	11
Projeto: SampleSchool		**Versão:** 1.0
Arquivos: 11 | **Classes:** 13 | **Métodos:** 39 | **Tempo:** 16 seg		
Bad Smell: Multi Directed Table		**Total:** 1
Tabela	**Classe**	**Classes**
People	Class e Person	2

objeto/relacional, o analisador encontrou 1 smell do tipo *Multi Directed Table*.

Usando a abordagem baseada em ontologia, o *stakeholder* pode analisar e entender o subdomínio de interesse de diferentes propósitos, permitindo até mesmo a customização de *smells* específicos para seu contexto.

6.1.5 Detecção de *Smells* em Projetos Reais

O detector de *smells* do tipo *Long Parameter List* em código-fonte orientado ao objetos produzido nas seções 6.1.1 e 6.1.2 foi aplicado a um conjunto de projetos de software escritos em Java e Python, desenvolvidos pela comunidade e selecionados arbitrariamente. A partir do Corpus Java Qualitas,[7] que é uma grande coleção de projetos Java de código aberto (TEMPERO et al., 2010) com diferentes tamanhos e propósitos, selecionamos cinco projetos, a saber: (i) o *framework* de testes JUnit; (ii) o *framework* de *logging* Log4j; (iii) o editor de código-fonte jEdit; (iv) o *framework linked data/semantic Web* Jena; e (v) a ferramenta de gerenciamento e compreensão de projetos de software Maven. A partir do

[7] <http://qualitascorpus.com>

Tabela 17 – Resultados para a análise de código em projetos do mundo real.

Projeto	**Arquivos**	**Classes**	**Métodos**	**Long Parameter List Smell**	**Tempo (s)**
JUnit 5.0	1137	2496	8984	4	5101
Log4j 2.0	503	982	3277	39	2944
jEdit 4.3.2	519	1479	6830	48	3125
Jena 2.6.3	914	1315	9900	6	5226
Maven 3.0.5	761	1439	6048	48	4625
Django 3.1	2674	7886	24995	119	35791
Numpy 1.19.1	487	1480	10835	40	7616
Pillow 7.2	272	266	2932	13	323
Requests 2.24	35	76	608	6	95
Theano 1.0.5	380	1268	9653	179	9065

GitHub, cinco projetos de software de código aberto escritos em Python foram selecionados, a saber: (i) a biblioteca HTTP Requests;[8] (ii) a biblioteca de imagens Pillow;[9] (iii) a biblioteca de processamento de *array* Numpy;[10] (iv) o *framework* de aplicação Web Django; e (v) a biblioteca de avaliação de expressões matemáticas Theano.[11]

Cada projeto de software foi processado separadamente pela aplicação e um resumo dos resultados é apresentado na Tabela 17. Para cada projeto é apresentado o número de arquivos .java e .py processados, o número de classes identificadas nesses arquivos, o número de métodos identificados nessas classes, o número de *smells* de *Long Parameter List* identificados nesses métodos e o tempo de processamento em segundos.

O tempo gasto pelo analisador de código é proporcional ao número de arquivos processados considerando a complexidade e qualidade do código, entre 1 a 7 segundos por classe em uma máquina de configuração modesta (processador de 2,70 GHz, SSD de 256 GB e 8 GB de RAM). Este tempo é dedicado 12,06% para o analisador sintático do código, 87,79% para a instanciação da ontologia e 0,15% para a análise de código sobre *bad smell*. Uma vez que a porcentagem de tempo dedicada a análise é baixa, supomos que a inclusão de outros *smells* tem um efeito muito baixo no tempo geral da análise de código.

Os resultados mostram que a solução de aplicação para o propósito de detecção de *bad smell* produzida pelo método OSCIN pode ser aplicada a projetos de software do mundo real de tamanho considerável. Uma vez que outros *smells* podem ser formalizados em consultas SPARQL mais complexas, testes adicionais usando um catálogo mais completo de *smells* são recomendados.

[8] <https://github.com/psf/requests>
[9] <https://github.com/python-pillow/Pillow>
[10] <https://github.com/numpy/numpy>
[11] <https://github.com/Theano/Theano>

6.2 Aplicação de OSCIN para Métrica de Código

Apresentamos brevemente como o método OSCIN poderia ser usado com um propósito de aplicação diferente — a saber, métricas de código —, embora reconheçamos que isso requer uma investigação mais aprofundada.

Reutilizando o subdomínio de orientação a objetos e as linguagens de programação dos cenários Java e Python (incluindo artefatos construídos para eles), especificamos duas métricas a serem analisadas: *Number of Classes* (NOC), métrica que conta o número de classes definidas em um software, excluindo classes de biblioteca; e *Number of Operations* (NOM), métrica que conta o número total de operações definidas pelo usuário (métodos) dentro do software. Ambas as métricas são formalizadas como consultas SPARQL sobre OOC-O, conforme mostrado na Listagem 6.2.

Listagem 6.2 – Formalização de métricas em SPARQL.

```
1 - Number of Classes (NOC)
2 SELECT (count(?class) as ?countClasses)
3 WHERE { ?class rdf:type ooc-o:Class . }
4
5 - Number of Operations (NOM)
6 SELECT ?class (count(?method) as ?countMethod)
7 WHERE { ?class rdf:type ooc-o:Class .
8   OPTIONAL{
9     ?method rdf:type ooc-o:Member_Function .
10    ?method ooc-o:componentOfClass ?class .
11  }
12 } GROUP BY ?class
```

Implementamos este propósito de aplicação a partir de consultas SPARQL aplicáveis sobre a ontologia do subdomínio a fim de calcular métricas de código-fonte a partir das instâncias da ontologia. Esta aplicação foi executada sobre o subdomínio de orientação a objetos (descrito na Seção 6.1.1 e Seção 6.1.2) e nos mesmos projetos da Seção 6.1.4, calculando NOC = 1.315 e NOM = 9.900 para Jena; NOC = 7.886 e NOM = 24.995 para Django e NOC = 13 e NOM = 39 para SampleSchool.

Ressaltamos que incluir esse novo propósito em um subdomínio e linguagens que já haviam sido especificadas para OSCIN exigiu muito pouco esforço de nossa parte, limitando-se apenas a elaborar as consultas apresentadas na Listagem 6.2.

6.3 Aplicação de OSCIN para Migração de Código

Apresentamos brevemente como o método OSCIN poderia ser usado para o propósito de migração de código, ou seja, migrar um código-fonte de uma linguagem de programação para outra. Esta aplicação foi desenvolvida em conjunto com um aluno de mestrado (ZANETTI; AGUIAR; SOUZA, 2019) e já reportada também em sua disserta-

ção (ZANETTI, 2020).

Reutilizando o subdomínio de mapeamento objeto/relacional e as linguagens de programação dos cenários Java e Python (incluindo artefatos construídos para eles, apresentados na Seção 6.1.3), aplicamos OSCIN para migrar código-fonte codificado com o *framework* Hibernate (Java) para o *framework* Django (Python).

Implementamos este propósito a partir do código-fonte instanciado na ontologia ORM-O. A aplicação desenvolvida analisa as instâncias dos conceitos ORM-O e gera as strings de código correspondentes em um único arquivo com a extensão .py. Por exemplo, a Listagem 6.3 apresenta a classe *Person* do projeto escrito em Hibernate/Java que foi usada como entrada para a aplicação e a Listagem 6.4 apresenta o código Django/Python gerado automaticamente como saída para esta aplicação.

Listagem 6.3 – Código da classe Person em Hibernate/Java produzido manualmente.

```
1  @Entity
2  @Table(name="PersonTable")
3  @Inheritance(strategy = InheritanceType.JOINED)
4  public class Person {
5
6          @Id
7          private Long id;
8          private String name;
9          @OneToOne
10         private PersonalAddress address;
11
12 }
```

Listagem 6.4 – Código da classe Person em Django/Python gerado automaticamente.

```
1  class Person(models.Model):
2          address = models.OneToOneField('PersonalAddress', on_delete=models.
       CASCADE)
3          id = models.BigIntegerField(primary_key=True)
4          name = models.CharField(max_lenght=255)
5
6          class Meta:
7                  pass
```

Ressaltamos que incluir esse novo propósito de aplicação em um subdomínio e linguagens que já haviam sido especificadas para OSCIN exigiu esforço específico apenas para a codificação relacionada à linguagem que estava sendo migrada (Django/Python).

6.4 Avaliação de OSCIN

Esta seção descreve as avaliações extraídas a partir das aplicações de OSCIN apresentadas nas seções anteriores, validando se o método OSCIN pode ser operacionalizado em soluções de software, ser extensível para a inclusão de novos subdomínios, linguagens de

programação e propósitos de aplicação, ser aplicado em projetos de software desenvolvidos pela comunidade e interoperar código-fonte de diferentes linguagens de programação.

A fim de avaliar o **Critério I** — se o método OSCIN pode ser operacionalizado em uma solução de software, aplicamos o método para a elaboração do *framework* OSCINF que facilita sua operacionalização por meio da adoção de um método de construção de ontologias, gerador de analisador sintático, gerador de analisador semântico e gerador de aplicação. Utilizando o *framework* OSCINF, operacionalizamos o método OSCIN com a adoção do método SABiOS na construção de ontologias, do analisador sitático baseado em ANTLR, analisador semântico baseado no *framework* Jena e propósito de aplicação baseado em *query* SPARQL com Jena, aplicando para o subdomínio de orientação a objetos, linguagem de programação Java e propósito de detecção de *smell* na Seção 6.1.1, para o subdomínio de orientação a objetos, linguagem de programação Python e propósito de detecção de *Smell* na Seção 6.1.2 e para o subdomínio de orientação a objetos, linguagem de programação Java e propósito de métrica de código na Seção 6.2.

Além do *framework*, o método OSCIN também foi operacionalizado adotando o método SABiO na construção de ontologias, do analisador sintático baseado em JavaParser, do analisador semântico baseado em OWLAPI e do propósito de aplicação baseado em *query* SPARQL com Jena, aplicando para o subdomínio de mapeamento objeto-relacional, linguagem de programação Java e propósito de detecção de *smell* na Seção 6.1.3 e para o subdomínio de mapeamento objeto/relacional, linguagem de programação Java e propósito de migração de código na Seção 6.3.

Desta avaliação podemos concluir que o método OSCIN pode ser operacionalizado pelo *framework* OSCINF e por outras soluções tecnológicas para o processamento de códigos-fonte.

A fim de avaliar o **Critério II** — se o método OSCIN suporta o reuso e a adesão de novas linguagens de programação, subdomínios e propósitos, utilizamos o *framework* OSCINF para a elaboração da solução de aplicação para o subdomínio de orientação a objetos, linguagem de programação Java e propósito de detecção de *smell* na Seção 6.1.1 e em ciclos posteriores adicionamos: novo subdomínio de mapeamento objeto/relacional (Seção 6.1.3), reutilizando o analisador sintático da linguagem já especificada; nova linguagem de programação Python (Seção 6.1.2), reutilizando a ontologia do subdomínio e o propósito de aplicação; e novo propósito de métrica de código (Seção 6.2), reutilizando a ontologia do subdomínio de orientação a objetos e o analisador sintático da linguagem.

Desta avaliação podemos concluir que o método OSCIN pode suportar a adesão de novas linguagens de programação, subdomínios e propósitos que compartilham e/ou reutilizam artefatos produzidos em ciclos anteriores.

A fim de avaliar o **Critério III** — se o método OSCIN operacionalizado permite a

interoperabilidade de código-fonte de diferentes linguagens de programação, utilizamos o *framework* OSCINF para a elaboração da solução de aplicação para o subdomínio de orientação a objetos e propósito de detecção de *smell* tanto para linguagem Java como linguagem Python, descritos na Seção 6.1.1 e na Seção 6.1.2, respectivamente. Com a solução de aplicação, executamos consultas SPARQL independentes de linguagem de programação para a detecção de *smell* nos projetos de software Jena (Java) e Django (Python), detalhados na Seção 6.1.4.

Desta avaliação podemos concluir que o método OSCIN pode permitir a interoperabilidade de código-fonte de diferentes linguagens de programação através de uma representação semântica do subdomínio de código-fonte, na forma de uma ontologia.

A fim de avaliar o **Critério IV** — se o método OSCIN pode ser aplicado em projetos de software desenvolvidos pela comunidade, utilizamos a solução de aplicação elaborada para o subdomínio de orientação a objetos, linguagem de programação Java (Seção 6.1.1) e Python (Seção 6.1.2) para a dedetecção de *smells* em 10 projetos desenvolvidos pela comunidade nessas linguagens, como apresentado na Seção 6.1.4.

Desta avaliação podemos concluir que o método OSCIN pode ser aplicado em projetos de software desenvolvidos pela comunidade para permitir a interoperabilidade de código-fonte de diferentes linguagens de programação. Estudos mais aprofundados sobre os resultados alcaçados com propósito de aplicação de detecção de *smells* serão abordados em trabalhos futuros.

6.5 Ameaças à Validade

Esta seção discute as principais ameaças à validade de nossos experimentos, de acordo com quatro tipos de validade: (i) interna, (ii) externa, (iii) conclusão e (iv) construto (WOHLIN et al., 2012).

A Validade Interna ameaça a conclusão do experimento sobre uma possível relação de causa e efeito entre o método e o resultado. Os artefatos produzidos na aplicação do método estão totalmente relacionados ao seu resultado e, portanto, a qualidade da ontologia e da *query* podem ser uma ameaça ao resultado do método, uma vez que a ontologia pode não representar os conceitos necessários para o propósito de aplicação e a consulta é personalizável pelo autor. Para mitigar essas ameaças, o método define que a ontologia deve ser construída seguindo um método de engenharia de ontologia estabelecido e que a consulta deve ser formalizada utilizando os conceitos da ontologia e da literatura.

A Validade Externa ameaça a generalização do método para uso na indústria. O número de subdomínios e linguagens de programação usados no experimento pode não ser suficiente para generalizar o método para qualquer código-fonte. Para reduzir o impacto

dessa ameaça, instanciamos o método com dois subdomínios de código-fonte e linguagens de programação diferentes. Além disso, para eliminar a ameaça da análise de *toy software*, selecionamos projetos de software reconhecidos pela comunidade de vários propósitos e tamanhos.

A Validade de Conclusão ameaça a conclusão correta dos resultados. A interpretação do leitor sobre a aplicação do método pode ser uma ameaça e, portanto, o método foi descrito com instruções detalhadas e exemplos de execução. Para minimizar ameaças relacionadas à definição do método como um padrão, o método foi utilizado por dois sujeitos diferentes, embora do mesmo grupo de pesquisa e em um contexto acadêmico.

A Validade do Construto ameaça a generalização do método. Para minimizar a ameaça de cobertura das fases definidas pelo método, as fases foram definidas de forma customizável e avaliadas em diferentes linguagens de programação, subdomínios e propósitos, embora não contemple todas as diversidades existentes. Para eliminar a ameaça de projetos influenciados pelo propósito da aplicação especificado, os projetos analisados foram capturados de um repositório de código aberto, desenvolvido em um processo normal de desenvolvimento de software e composto por diferentes programadores. Além disso, os resultados obtidos com a execução do propósito nos projetos analisados foram analisados superficialmente e as ameaças à sua precisão e personalização devem ser abordadas em trabalhos futuros.

6.6 Considerações Finais

Este capítulo apresentou resultados para interoperabilidade de código-fonte baseada em ontologia aplicando o método OSCIN, demonstrando que o método pode ser operacionalizado em soluções de software, ser extensível para a inclusão de novos subdomínios, linguagens de programação e propósitos de aplicação, interoperar código-fonte de diferentes linguagens de programação e ser aplicado em projetos de software desenvolvidos pela comunidade.

O método é aplicado para o propósito de detecção de *smells* a partir de uma ontologia de orientação a objetos e de mapeamento objeto/relacional para analisar projetos de software escritos em linguagem de programação Java e Python. Este propósito de aplicação é executado em projetos de software conhecidos pela comunidade, processando cerca de 7.682 arquivos, 18.687 classes, 84.062 métodos e identificando 502 smells. O método é aplicado para o propósito de métrica de código e executado sobre os projetos selecionados no propósito de detecção de *smells*. Ademais, o método é aplicado para o propósito de migração de código no subdomínio de ORM a fim de migrar código escrito em Hibernate/Java para código Django/Python.

Neste contexto, identificamos pouquíssimos trabalhos que propõem um método

para realizar detecção de *smells*. O método DECOR (MOHA et al., 2010) apresenta as etapas para definir uma técnica de detecção de *smells* por meio da especificação de regras de *smell* em uma *Domain Specific Language* (DSL) para gerar algoritmos de detecção automaticamente. Os *smells* de código são especificados em regras DSL que são reificadas em um modelo de *smell* a partir de um metamodelo orientado a objetos e um analisador sintático. Este modelo deve ser instanciado individualmente para cada *smell* e, a partir dos métodos *visitors*, gerar um algoritmo de detecção específico para aquele *smell* usando o modelo. Por outro lado, os arquivos de código-fonte são reificados no modelo de código-fonte com base na engenharia reversa. Em seguida, os algoritmos de detecção gerados a partir do modelo de *smells* de código são aplicados no modelo de código-fonte. Embora o método DECOR seja capaz de detectar diferentes *smells* de código em uma abstração de alto nível usando uma DSL, podemos observar que o método representa o código-fonte em um modelo sintático e os *smells* de código em um metamodelo independente da linguagem que gera algoritmos específicos. Tendo uma visão diferente, OSCIN representa o código-fonte em um modelo semântico (uma ontologia) e o *smell* em uma linguagem de consulta nesse modelo.

Outros trabalhos não propõem um método para derivar detectores de *code smell*, mas sim um detector para tipos específicos de *smell*. Em particular, alguns desses trabalhos utilizam ontologias para representar o código-fonte ou uma regra de detecção comum para diferentes linguagens, como em iSPARQL (KIEFER; BERNSTEIN; TAPPOLET, 2007) (apresentado na Seção 7.2.1) e OCEAN (Da Silva Carvalho et al., 2017) (apresentado na Seção 7.2.7).

Da mesma forma, o uso de regras comuns de detecção é observado no MLSD (RASOOL; ARSHAD, 2017), *Multiple Language Smells Detector*, que apresenta uma representação intermediária do código-fonte como banco de dados para identificar o *smell* do código em consultas SQL. O código-fonte é estruturado em tabelas a partir de engenharia reversa usando análise sintática, expressões regulares e métricas de software. A partir do banco de dados, os *smells* de código são detectados aplicando consultas SQL no esquema do banco de dados, permitindo a aplicação em múltiplas linguagens.

Para o propósito de migração de código, identificamos pouquíssimos trabalhos que se aprofundaram na relação existente entre os domínios dos paradigmas de orientação-objeto e relacional. Schaub e Malloy (2016) propõe uma linguagem interoperável, denominada iJava, para a migração de código entre as linguagens Java, Python e C++. Diferente de OSCIN, iJava não possui um modelo conceitual formal e explícito e não inclui na conversão entre as linguagens e os mapeamentos objeto/relacionais.

Embora o foco desta pesquisa não inclua adentrar nas especificidades dos propósitos de aplicação, os trabalhos mencionados anteriormente se diferenciam de OSCIN por: (i) apresentarem uma ferramenta computacional e não um método completo de

interoperabilidade de código-fonte (detecção de *smells*, métrica de código, migração de código, etc); (ii) representarem o código-fonte como uma ontologia, limitada a apenas alguns elementos orientados a objetos, objeto/relacional ou como um banco de dados, sem características semânticas do código-fonte; (iii) representarem métricas de código na ontologia ou no banco de dados calculado a partir da estrutura sintática do código-fonte; e (iv) limitarem o escopo da ontologia em paradigma orientado a objetos ou à linguagem Java.

7 Trabalhos Relacionados

Neste capítulo apresentamos trabalhos relacionados ao tema desta pesquisa. No entanto, não identificamos na literatura outros trabalhos relacionados ao mesmo propósito do método OSCIN, a saber, sistematizar o processo de representar semanticamente código-fonte escrito em diferentes linguagens de programação para interoperá-los, aplicando diferentes perspectivas (*bad smell*, medição, etc.). Hipotetizamos, portanto, que este seja o primeiro trabalho nesta área específica.

Sendo assim, apresentamos trabalhos relacionados em duas categorias mais abrangentes. Em primeiro lugar, discutimos os trabalhos que visam a interoperabilidade do código-fonte e, em seguida, trabalhos que podem ser considerados similares a aplicações do método OSCIN.

7.1 Interoperabilidade de código-fonte

A maioria dos sistemas são multilíngues por natureza e essa tendência está aumentando dia a dia (MUSHTAQ; RASOOL; SHEHZAD, 2017). Nesse contexto, uma revisão sistemática analisou 3.820 pesquisas sobre análise de código-fonte multilíngue entre os anos de 2001 e 2016 e constatou que apenas 4% das pesquisas tratam de análise semântica, sendo 26% análise estática, 15% compreensão de programas, 15% engenharia reversa, 13% detecção *cross language link*, 11% de análise dinâmica, 11% análise de aplicações corporativas, 11% de ambiente de desenvolvimento integrado e outas porcentagens menores (MUSHTAQ; RASOOL; SHEHZAD, 2017). A maioria das linguagens de programação tratadas pelas abordagens estudadas são de orientação a objetos 57%, Java 53%, HTML 26%, DSLs 21% e C/C ++/C# 17%, abrangendo de 2 a 9 linguagens por abordagem.

Propostas de análise de código multilíngue visam processar sistemas compostos por códigos-fonte de diferentes linguagens de programação além dos limites de uma única linguagem. Por exemplo, sistemas Web que são compostos de código-fonte HTML e CSS para o *frontend*, JavaScript para lidar com o comportamento do *frontend*, Java para o *backend* e outras. Uma vez que a natureza desses sistemas é complexa e dinâmica, compreender a interação de várias linguagens por meio de artefatos sem integração e suporte de ferramenta é difícil e desafiador (TICHELAAR et al., 2000a).

Para o *reuso de código*, destaca-se o desafio de adotar componentes reutilizáveis de diferentes linguagens e buscar trechos de código relevantes sem o suporte de repositórios e mecanismos de recomendação (MUSHTAQ; RASOOL; SHEHZAD, 2017). A escassez de *detecção de cross-language links* afeta a produtividade e a estabilidade do sistema (TER-

CEIRO; COSTA; MIRANDA, 2010), uma vez que mesmo pequenas mudanças podem impactar o comportamento de todo o sistema, acrescido pela escassez de uma abordagem padrão adequada para vincular artefatos multilíngues (MARINESCU, 2006). Para *refatoração*, uma abordagem geral é difícil, senão impossível, de automatizar (SCHINK et al., 2011), uma vez que a refatoração para um artefato de um tipo geralmente não descreve as mudanças que devem ser feitas em artefatos de outros tipos. Extrair informações a partir de código de diferentes linguagens é essencial para a compreensão de sistemas multilíngues (GIFFHORN; HAMMER, 2008), cujo nível arbitrário de granularidade dificulta a análise, compreensão e engenharia reversa (MUSHTAQ; RASOOL; SHEHZAD, 2017).

Propostas de ambientes de desenvolvimento multilígue são especificados usando *reference attribute grammars* (SIEMUND; TOVESSON, 2018) (atributo que referencia nós da árvore sintática), por meio de um *Language Server Protocol* (LSP) que padroniza a troca de mensagens entre o cliente (ambiente de desenvolvimento) e o servidor (servidor da linguagem de programação), que interpreta a mensagem e processa na sua respectiva linguagem de programação.

Neste caso, saimos do cenário onde o suporte de recursos para uma determinada linguagem requer um esforço significativo e passamos para o cenário onde o suporte pode ser reusado para diferentes ambientes (RASK et al., 2021), ou seja, do cenário onde cada ambiente de desenvolvimento precisa lidar particularmente com cada linguagem de programação (1:1) para o cenário onde qualquer ambiente de desenvolvimento pode lidar com qualquer servidor de linguagem de programação (N:N). Embora este cenário tenha sido adotado por várias linguagens de programação e surgido com vários servidores de linguagens de programação, extensões foram realizadas de maneira ad-hoc com funcionalidades adicionais não normalmente encontradas para linguagens de programação e, portanto, também sem suporte no LSP (MASCI; MUÑOZ, 2019).

Assim, novas propostas têm surgido para padronizar a extensão LSP em ambientes de desenvolvimento, a fim de atender às necessidades particulares das linguagens de programação sem gerar novas reimplementações (RASK et al., 2021). Além disso, ambientes de desenvolvimento com suporte a várias linguagens geralmente são incapazes de fornecer uma experiência de programação consistente devido aos diferentes recursos de tempo de execução específicos de cada linguagem (NIEPHAUS et al., 2018), tais como integrações de linguagens de programação por meio de camadas de abstração relacionada ao sistema operacional ou conexão de rede.

Além de servidores de linguagem, observamos a construção de ambientes de desenvolvimento multilígue especificados usando um modelo universal de linguagem (PFEIFFER; WASOWSKI, 2012) que representa a estrutura sintática comum de diferentes linguagens de programação, tal como Element, Text, Block, Paragraph, WordBlock e Word. Por exemplo, um código JavaScript ‘if(event.command ==’ pode ser representado

como Block(Paragraph[WorkBlock(Word[WordPart('if'), SeparatorPart(content:'('), WordPad('event'), SeparatorPart(':',WordPart('command'), WhiteSpace(' ')]), ...]).

Proposta para padronização é observada na ISO/IEC JTC 1/SC 22,[1] que promove a cooperação internacional em questões relacionadas a linguagens de programação, tal como linguagens de programação clássicas (Fortran, C, C ++ e Ada) e documenta vulnerabilidades de várias linguagens de programação. O padrão 'silenciosamente' e 'invisivelmente' atende às necessidades de programadores de TI e desenvolvedores de compiladores, concentrando-se em ambientes de programação de alto nível para melhorar a portabilidade de aplicativos, a produtividade e a mobilidade dos programadores, bem como a compatibilidade de aplicativos ao longo do tempo (LANGUAGES, 2001). O padrão pretende apoiar o investimento global em sistemas de software, mantendo e melhorando as linguagens de programação padronizadas, melhorando continuamente a padronização do ambiente de programação por meio de documentação e lições aprendidas e respondendo a oportunidades tecnológicas emergentes (LANGUAGES, 2001).

Proposta para definição de metamodelos é observada na ISO/IEC 19506,[2] que é uma especificaçao do *Object Management Group* (OMG) que define um *Knowledge Discovery Meta-model* (KDM) para representar ativos de software (módulos de código-fonte, descrições de banco de dados, scripts, etc) existentes, suas associações e ambientes operacionais, a fim de facilitar a interoperabilidade e troca de dados entre ferramentas de diferentes fornecedores. Uma vez que cada ferramenta produz uma parte do conhecimento sobre os ativos de software, esse conhecimento específico da ferramenta pode estar implícito ('embutido no código' na ferramenta), restrito a linguagem e/ou transformação e/ou ambiente operacional (ISO/IEC19506, 2012). Assim, este metamodelo para descoberta de conhecimento é uma representação intermediária que fornece uma estrutura comum para representar ativos físicos e lógicos em vários níveis de abstração. Nesse caso, o pacote *Code* representa elementos comuns suportados por várias linguagens de programação, como *data types*, *data items*, *classes*, *procedures*, *macros*, *prototypes* e *templates*. O metamodelo é formado por elementos como *CodeModel*, *AbstractCodeElement*, *AbstractCodeRelationship*, *CodeItem*, *ComputationalObject*, *Module*, *Datatype* e suas especializações.

7.2 Trabalhos Similares a uma Aplicação de OSCIN

Alguns trabalhos propõem modelos conceituais e ferramentas com propósito similar às ontologias e/ou às soluções da perspectiva implementadas num ciclo de execução do método OSCIN. A seguir, sintetizamos tais trabalhos relacionados e, em seguida, categorizamo-los seguindo os três pilates do método OSCIN, para comparação.

[1] <https://www.iso.org/committee/45202.html>
[2] <https://www.iso.org/standard/32625.html>

7.2.1 iSPARQL

O primeiro trabalho que identificamos nesta área é o de análise de software com iSPARQL (KIEFER; BERNSTEIN; TAPPOLET, 2007), trabalho que apresenta EvoOnt, um conjunto de ontologias de *design de software*, e iSPARQL, uma *engine* de consulta semântica baseada em SPARQL. EvoOnt é baseada no meta modelo FAMIX (TICHELAAR et al., 2000b), sendo independente da linguagem de programação e modelando aspectos do código-fonte orientado a objetos no que diz respeito ao *design de software* (por exemplo, *metric*, *entity*, *directory*, *package*, *class*, *method* e *localVariable*), *bug* (por exemplo, *product*, *attachment*, *comment* e *person*) e versionamento (por exemplo, *file*, *revision* e *release*). iSPARQL é baseado em SPARQL, executando consultas em ontologias OWL e agregando um conjunto de medidas de similaridade, como *Levenshtein Measure*, *TreeEditDistance Measure* e *Graph Measure*. O código-fonte Java é analisado (*parsing*) para produção de instâncias da ontologia usando a IDE Eclipse (IDE, 2020a) e armazenando em arquivo OWL com o *framework* Jena (MCBRIDE, 2002). Consultas iSPARQL são aplicadas na ontologia EvoOnt instanciada, destinadas à medição da evolução do software com o auxílio das medidas de similaridade, medição de métricas orientadas a objeto, medição de densidade de software e detecção de *smell*.

7.2.2 BugDetector

BugDetector (YU et al., 2008) propõe uma ferramenta para a detecção de bugs em projetos Java. A ontologia de programa Java é baseada na especificação da linguagem (por exemplo, *class*, *method*, *interface*, *clause*, *operator*, *object* e *primaryValue*) enquanto a de padrão de bug é baseada em mais de duzentos padrões de bug identificados em projetos Java (por exemplo, *NullPointerBug* e *MaliciousCodeBug*), abstraídos em regras SWRL (*Semantic Web Rule Language*). O código-fonte Java é analisado (*parsing*) para produção de uma árvore AST por meio da biblioteca ANTLR (PARR, 2013) e instanciado na ontologia a partir do Protégé API (MUSEN, 2015), gerando representação OWL do código-fonte. A sintaxe OWL é convertida para a sintaxe Jess (Java Expert System Shell) com a ajuda da ferramenta SWRL Bridge do Protégé. Jess engine e as regras SWRL são aplicadas sobre a ontologia instanciada com o código Java a fim de detectar bugs no codigo fonte.

7.2.3 Semantic Stimulus Tool

Semantic Stimulus Tool (GANAPATHY; SAGAYARAJ, 2011) é um *framework* de extração de código-fonte para pesquisar e armazenar metadados de código Java a fim de facilitar a reutilização de código. A ontologia é baseada na especificação da linguagem Java (por exemplo, *project*, *packages*, *classes*, *methods* e *parameters*) e populada em arquivos OWL individuais para cada pacote Java. O código-fonte Java é analisado (*parsing*) para

produção de strings de metadados usando QDox (QDOX, 2021) a partir de objetos JavaBuilder. Tais strings de metadados são manipuladas pelo *framework* Jena (MCBRIDE, 2002) e armazenadas em OWL, vinculadas ao próprio código-fonte, que fica armazenado em um repositório HDFS (Hadoop Distributed File System), podendo ser recuperado por meio de consultas sobre a ontologia OWL.

7.2.4 SeCold

SeCold (KEIVANLOO; RILLING; CHARLAND, 2012) é o primeiro e maior conjunto de dados interligados (*Linked Data*) sobre código-fonte disponível publicamente e incluído na LOD Cloud (CYGANIAK; JENTZSCH, 2021) com 1,5 bilhão de triplas extraídas de 18.000 projetos Java de código aberto, com o objetivo de fornecer pesquisa sobre código-fonte e detecção de código clone. A família de ontologias SECON (Source Code Ecosystem Ontology Family) reúne ontologia de código-fonte, versionamento, rastreabilidade, licenciamento, metadados e clone de código. O código-fonte Java é recuperado por um Web *crawler* e pré-processado com normalização e lematização para remover o ruído do código, atribuindo um *hash* a cada linha de código. O código Java pré-processado é analisado (*parsing*) para produção de *tokens* a partir da árvore AST e instanciado na ontologia, sendo armazenado em um *Triple Store*. Consultas SPARQL são aplicadas ao repositório junto com funções de similaridade e clusterização a fim de pesquisar código-fonte clonado.

7.2.5 RDFizer

RDFizer (PAYDAR; KAHANI, 2012) é uma abordagem proposta para detectar padrões de design em código-fonte Java. A ontologia é baseada na ontologia SIMILE (SIMILE..., 2021) Java2RDF e na especificação da linguagem Java (por exemplo, *class*, *member*, *constructorCount*, *methodCount*, *fieldCount*, *SwitchCaseStmt* e *WhileStmt*). O código-fonte Java é analisado (*parsing*) para produção de instâncias da ontologia a partir da árvore AST da IDE Eclipse (IDE, 2020a), gerando representação RDF do código-fonte. Os arquivos RDF são armazenados e manipulados no RepositoryManager, um repositório implementado com API Jena (MCBRIDE, 2002) que permite a aplicação de inferências e consultas SPARQL com base nos padrões de design identificados.

7.2.6 PATO

PATO (ZHAO et al., 2016) é um *framework* para análise de programas que introduz a ideia de aplicar o mesmo *framework* conceitual para diferentes análises. A ontologia é baseada no padrão C99 da linguagem C, contendo 178 conceitos e 68 propriedades (por exemplo, *CProgram*, *Statement*, *Type*, *Expression*, *Variable*, *Literal*, *StorageClass*, *Scope*, *Declaration*, etc). O código-fonte C é analisado (*parsing*) para produção de instâncias

da ontologia a partir da árvore AST da ferramenta ROSE (QUINLAN; LIAO, 2011), armazenado como um arquivo OWL na base de conhecimento. Os conceitos são nomeados conforme o padrão C99 e as instâncias são nomeadas com *scoped IRI*, onde os construtos são nomeados pelo escopo do construto adicionado ao seu nome e, quando aplicável, adicionada a localização dentro de outro construto. Por exemplo, *wri::foo()::1:6,1:10* representa o identificador de uma variável declarada na primeira linha (coluna 6 até 10) da função *foo()* no arquivo *wri*. A base de conhecimento pode ser acessada através da interface de consulta em linguagem C++ ou através da API PATO com regras lógicas e *reasoner* em SWI-Prolog (WIELEMAKER et al., 2012a), aplicado sobre a base de conhecimento carregada na memória com a ferramenta Semantic Web Lib (WIELEMAKER et al., 2012b).

7.2.7 OCEAN

OCEAN (Da Silva Carvalho et al., 2017) é um analisador de *code smell* para código-fonte Java. A ontologia ONTOCEAN é baseada em conceitos orientados a objetos e métricas de software (por exemplo, *Clazz*, *Method*, *Metric*, *ClassOrientedeMetric*, *SLOC*, *Codesmell*, *ClassOrientedSmell*, *BrainClass*, *Comitter*, *Commit*, *Repository*, etc). O código-fonte Java é analisado (*parsing*) para construção de uma árvore AST a partir da biblioteca JDT (IDE, 2020b) da IDE Eclipse, onde as métricas do software são calculadas com a biblioteca Visminer (VISMINER, 2018) e armazenadas em arquivos VCS. Os arquivos VCS são convertidos em arquivos OWL com a biblioteca Java OWL API (API, 2020), contendo as instâncias dos *code smells* identificados. A ontologia OWL é carregada na ferramenta Protégé executando o *reasoner* Hermit OWL a fim de visualizar as instâncias dos *code smells*.

7.2.8 Design Flaws Detection

Design Flaws Detection (MEKRUKSAVANICH, 2017) é uma abordagem para detecção de falhas de design de software. A ontologia é baseada em problemas de código-fonte, formas de detecção e seus relacionamentos (por exemplo, *Anti-pattern*, *Errors*, *PoorCode*, *Bad Code Smells*, *Metrics*, *Heuristics*, etc). O código-fonte Java é analisado (*parsing*) para produção de uma árvore de nós, contendo operadores e declarações em nós internos e variáveis e constantes em nós folha. As regras de detecção de falhas são representadas em Description Logics (DL), que são convertidas em lógica de primeira ordem e aplicadas à árvore por meio de um mecanismo de correspondência de padrões (entre fatos e regras) e um mecanismo de encadeamento reverso para obter os resultados.

7.2.9 CodeOntology

CodeOntology (VRANDEC et al., 2018) é um *framework* para suporte de consultas sobre código-fonte Java. A ontologia é baseada na reengenharia da sintaxe abstrata da linguagem Java, contendo 65 conceitos e 97 propriedades (por exemplo, *CodeElement*, *Class*, *BlockStatement*, *CatchBlock*, *Constructor*, *ContinueStatement*, *ControlFlowStatement*, *Enum*, *Expression*, etc). O codigo fonte Java ou bytecode é analisado (*parsing*) para produção de uma árvore AST a partir da biblioteca SPOON (PAWLAK et al., 2016) e Java *reflection*, sendo serializado em RDF a partir do *framework* Jena (MCBRIDE, 2002). Comentários são extraídos do codigo fonte e ligados com a DBPedia por meio do TagMe (FERRAGINA; SCAIELLA, 2011). Os arquivos RDF são armazenados como *Linked Data*, permitindo a aplicação de consultas SPARQL.

7.2.10 OPAL

OPAL (PATTIPATI; NASRE; PULIGUNDLA, 2020) é um *framework* para análise de programas em linguagem C, construído sobre o *framework* PATO (ZHAO et al., 2016) e inicialmente aplicado para detecção de bugs. A ontologia é baseada na ontologia PATO e na estrutura sintática da linguagem C (por exemplo, *ProgramPointInstance*, *ProgramPath*, *RelationClause*, *SpecClause*, etc). O código-fonte C é processado pelo *framework* PATO e analisado (*parsing*) para produção de instâncias da ontologia a partir da árvore AST da ferramenta ROSE (QUINLAN; LIAO, 2011), gerando *program triples* em RDF. As *program triples* são combinadas para gerar *control flow triples*, triplas RDFs que representam a informação do fluxograma do programa. Regras RDF são aplicadas sobre as *program triples* e *control flow triples* para extrair informações intraprocedurais sensíveis ao caminho, gerando *inferred triples* a partir do *reasoner* do *framework* Jena (MCBRIDE, 2002). As especificações de análise do usuário são representadas como expressões SPARQL e convertidas em restrições de análise do programa, processadas por uma solução de Satisfiability Modulo Theories (SMT) para retornar os bugs identificados no programa.

7.2.11 Instanciação em OSCIN

Os trabalhos descritos anterormente podem ser considerados similares a aplicações do método OSCIN. A **Fase de Especificação** define os três pilares do método OSCIN: Subdomínio de código-fonte, Linguagem de Programação e Perspectiva de Aplicação. Para efeito de comparação, a Tabela 18 instancia os trabalhos relacionados como se fossem execuções do método OSCIN, indicando qual teria sido a escolha em cada um dos seus pilares.

Analisando o pilar de **Subdomínio de código-fonte**, os trabalhos relacionados se limitam a alguns subdomínios específicos de código-fonte, principalmente relacionados

Tabela 18 – Instanciação da Fase de Especificação do Método OSCIN para os trabalhos relacionados

Nome	Ano	Subdomínio de código-fonte	Linguagem de Prog.	Perspectiva de Aplicação
iSPARQL	2007	Orientação a objetos, independente de linguagem de programação	Java	Medição de software, de evolução de software, de densidade de software e code smell
Bug Detector	2008	Sintática da linguagem Java	Java	Detecção de bug
Semantic Stimulus Tool	2011	Orientação a objetos	Java	Pesquisa de código-fonte para reuso
SeCold	2012	Sintática do código-fonte, versionamento, rastreabilidade e licenciamento	Java	Pesquisa de código-fonte e detecção de código clone
RDFizer	2012	Orientação a objetos e sintática da linguagem Java	Java	Detecção de padrões de design
PATO	2016	Sintática da linguagem C	C	Análise de programa com análise de loop canônico, análise de ponteiro, construção de gráfico de fluxo de controle, análise de padrão de acesso a dados e orientação de colocação de dados de GPU
OCEAN	2017	Orientação a objetos, bad smell de código e métricas	Java	Detecção de code smell
Design Flaws Detection	2018	Falhas de código	Java	Detecção de falhas de código
Code Ontology	2018	Orientação a objetos e sintática da linguagem Java	Java	Publicação e pesquisa de Linked Data de código-fonte
OPAL	2020	Sintática da linguagem C	C	Detecção de bug

à sintática da linguagem Java e C (40.0%) e ao paradigma de orientação a objetos (33.3%), como observado na Figura 29. Dentre os subdomínios tratados pelos trabalhos relacionados, o subdomínio de *Orientação a Objetos* corresponde mais fielmente ao pilar subdomínio definido no método OSCIN, pois representa um conceito compartilhado do subdomínio que está presente em várias linguagens de programação, tal como outros paradigmas de programação e *frameworks*. O subdomínio de *Sintática da Linguagem* representa as particularidades de uma linguagem de programação específica, ou seja, a ontologia construída representa elementos da estrutura sintática de uma linguagem, mais próximos de um metamodelo da linguagem do que de uma ontologia. Já os subdomínios de *Métrica de Código*, *Problema de Código* e *Controle de Código* representam mais as perspectivas de aplicação do que as características do código-fonte em si.

Analisando o pilar de **Linguagem de Programação**, os trabalhos relacionados se limitam a linguagem Java (80%) e linguagem C (20%), como observado na Figura 30. Embora o subdomínio de *Sintática da Linguagem* seja endereçada em 40% dos trabalhos, todos os trabalhos (100%) são direcionados a linguagens de programação específicas e não são compartilhados por outras linguagens de programação no mesmo subdomínio. Destacamos que as linguagens (Java e C) tratadas nos trabalhos são amplamente utilizadas pela comunidade, mas que as linguagens emergentes como Python, C# e JavaScript não são consideradas.

Analisando o pilar de **Perspectiva de Aplicação**, os trabalhos relacionados cobrem

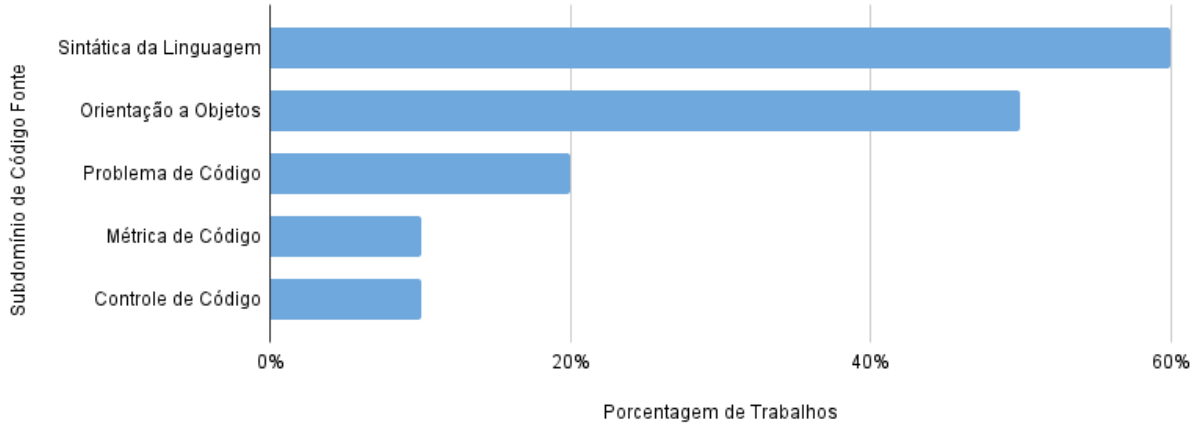


Figura 29 – Subdomínio de código-fonte apresentado nos trabalhos relacionados

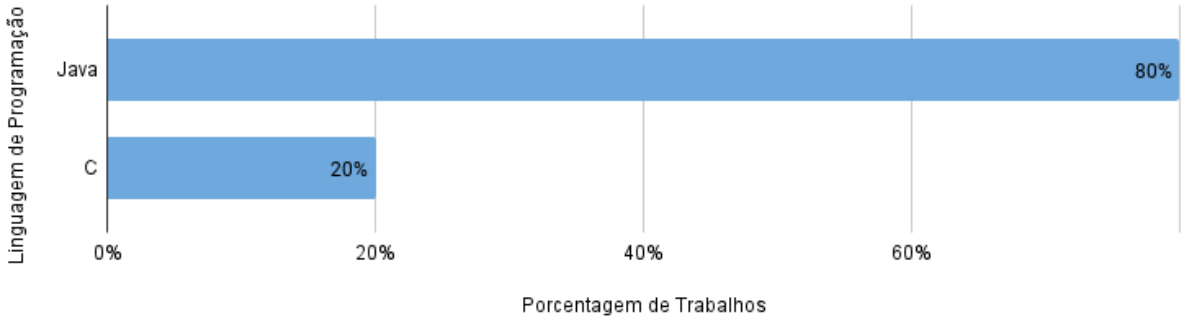


Figura 30 – Linguagem de Programação apresentada nos trabalhos relacionados

diversas perspectivas de aplicação, principalmente *Publicação de LinkedData/Dataset* (25%), *Detecção de Code Smell* (16,7%) e *Detecção de Bug* (16,7%), como observado na Figura 31. Sumarizando a análise podemos considerar: *Análise de Código* com 58,3%, agregando *Detecção de Falha*, *Design Pattern*, *Code Smelll*, *Clone Code* e *Bug*; *Publicação de LinkedData ou Dataset* com 25,0%; e *Análise de Programa* com 16,7%, agregando *Medicação de Software* e *Análise de Programa*.

A Tabela 19 apresenta a instanciação dos trabalhos relacionados segundo as fases de Abstração Semântica do Subdomínio, Abstração Sintática da Linguagem, Abstração Semântica da Linguagem e Implementação da Perspectiva de Aplicação.

A **Fase de Abstração Semântica do Subdomínio** é retratada na forma de ontologia em linguagem OWL e RDF. Essa representação é compreensível por máquina, flexível na alteração do modelo/indivíduo e passível de raciocínio.

A **Fase de Abstração Sintática da Linguagem** é aplicada por um *parser* baseado na estrutura sintática da linguagem de programação, geralmente reutizado de outras ferramentas já consolidadas como a IDE Eclipse, ANTLR, QDox, ROSE e SPOON. Destacamos que o reuso de *parser* sintático é recomendado e contribui para a maturidade da aplicação.

Tabela 19 – Instanciação do Método OSCIN para os trabalhos relacionados

Nome	**Abstração Semântica do Subdomínio**	**Abstração Sintática da Linguagem**	**Abstração Semântica da Linguagem**	**Implementação da Perspectiva de Aplicação**
iSPARQL	Subdomínio representado em OWL	Parser que representa o código-fonte em árvore AST com a IDE Eclipse	Ontologia OWL instanciada com Jena Framework a partir da árvore AST	Regras semânticas e medidas de similaridade em iSPARQL aplicadas sobre a ontologia OWL
Bug Detector	Subdomínio representado em OWL	Parser que representa o código-fonte em árvore AST com a biblioteca ANTLR	Ontologia OWL instanciada com Jena Framework a partir da árvore AST e convertida para sintaxe Jess	Regras SWRL aplicadas sobre a ontologia em sintaxe Jess
Semantic Stimulus Tool	Subdomínio representado em OWL com ferramenta Protégé	Parser que representa o código-fonte em árvore AST com a ferramenta QDox	Ontologia OWL instanciada com Jena Framework a partir da árvore AST	Ontologia OWL carregada na ferramenta Protégé ou Altova Semantics, lincada ao repositório HDFS de código-fonte
SeCold	Subdomínio representado em OWL	código-fonte representado em árvore AST com a IDE Eclipse	Ontologia OWL instanciada com Jena Framework a partir da árvore AST	Ontologia RDF armazenada em repositório e disponibilizada para inferência e consulta SPARQL com algoritmos de similaridade e clusterização
RDFizer	Subdomínio representado em RDF	Parser que representa o código-fonte em árvore AST com a IDE Eclipse	Ontologia RDF instanciada com Jena Framework a partir da árvore AST	Ontologia armazenada em repositório e disponibilizada para consultas SPARQL com Jena Framework
PATO	Subdomínio representado em OWL	Parser que representa o código-fonte em árvore AST com a ferramenta ROSE	Ontologia OWL instanciada a partir da árvore AST, utilizando Scoped IRI como identificadores	Ontologia armazenada em base de conhecimeno OWL e disponibilizada para consultas com interface C++ ou SWI-Protolog (com inferência)
OCEAN	Subdomínio representado em OWL	Parser que representa o código-fonte em árvore AST com a biblioteca JDT da IDE Eclipse	Ontologia OWL instanciada com a biblioteca Java OWL API a partir da árvore AST e das métricas do software calculadas com a biblioteca Visminer	Ontologia OWL carregada na ferramenta Protégé, executando Hermit OWl Reasoner para inferir code smells
Design Flaws Detection	Subdomínio representado em OWL com a ferramenta Protégé	Parser que representa o código-fonte em árvore de nós, contendo operadores e declarações em nós internos e variáveis e constantes em nós folha	Ontologia OWL instanciada a partir da árvore e de regras em Description Logic (DL)	Regras DL são convertidas em fatos lógicos de primeira ordem para que o mecanismo de pattern matching seja aplicado entre o fato e a regra para detecção de falhas, utilizando o mecanismo de backward chaining.
Code Ontology	Subdomínio representado em OWL com a ferramenta Protégé	Parser que representa o código-fonte em árvore AST com a biblioteca SPOON, sendo previamente convertido com Java Reflection nos casos de bytecode	Ontologia RDF instanciada com o framework Jena a partir da árvore AST, cujos comentários são lincados com a DBPedia através da biblioteca TagMe	Ontologia armazenada em Linked Data e disponibilizada para consultas SPARQL
OPAL	Subdomínio representado em OWL com a ferramenta Protégé	Parser que representa o código-fonte em árvore AST com a ferramenta PATO	Ontologia RDF instanciada com o framework Jena a partir da árvore AST, adicionado às informações de controle de fluxo e intrapocedurais sensíveis de caminho.	Consultas SPARQL são aplicadas sobre a ontologia e processadas com solução de Satisfiability Modulo Theories (SMT) em busca de bugs

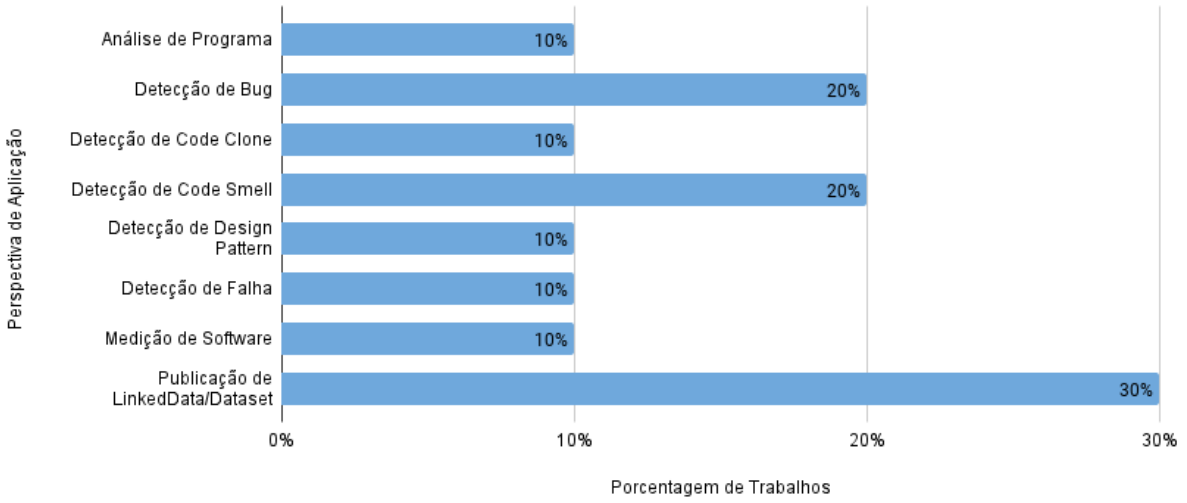


Figura 31 – Perspectiva de Aplicação apresentada nos trabalhos relacionados

A **Fase de Abstração Semântica da Linguagem** instancia a ontologia em OWL, RDF ou Jess a partir da árvore sintática com o *framework* Jena ou com OWL API. Esta instanciação requer a definição de um identificador único para as instâncias (conforme discutido em PATO) e algum processamento relacionado aos conceitos da ontologia, como cálculo de métricas (OCEAN), regras em Description Logics (Design Flaws Detection), link para DBPedia (CodeOntology) e algoritmos de estrutura de código (OPAL). Em alguns casos, a ontologia é convertida para a sintaxe adotada na aplicação, como a sintaxe Jess (BugDetector).

A **Fase de Implementação da Perspectiva de Aplicação** abrange a visualização (20%) e disponibilização (40%) da ontotologia em *dataset* ou *linked data*, acrescido da disponibilização do código-fonte (10%) original em alguns casos. Sobre a ontologia gerada, normalmente são aplicadas consultas semânticas (80%), além de algoritmos Similarity (20%), Custering (10%) e Matching/Satisfiability (20%), como observado na Figura 32.

7.3 Considerações Finais

Este capítulo apresentou trabalhos relacionados ao método OSCIN no que diz respeito às abordagens para a interoperabilidade do código-fonte e aplicações do método. A análise das abordagens de interoperabilidade demonstra que a interoperabilidade do código-fonte é uma área em descoberta e que tem recebido maior atenção e contribuição nos últimos anos. Embora observemos iniciativas de protocolos de servidores de linguagem, análise multilíngue, padronização e metamodelos para o domínio do código-fonte, nenhuma das abordagens se aproxima da proposta desta pesquisa no que diz respeito à semântica do código-fonte. Além disso, a análise da instanciação do método mostra que tanto os subdomínios do código-fonte quanto as linguagens de programação ainda são tratados de

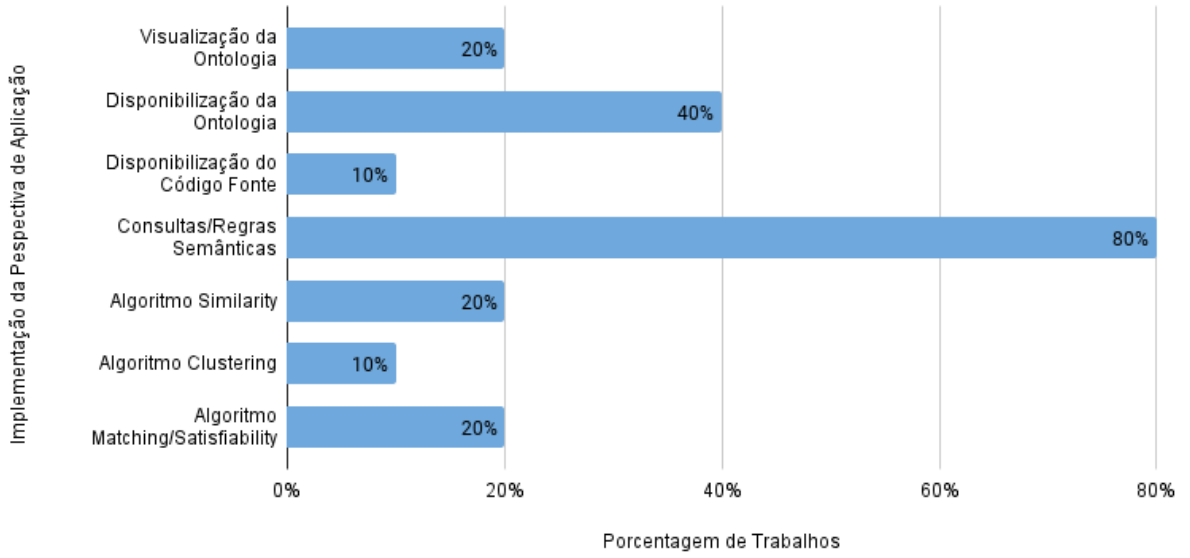


Figura 32 – Implementação da Perspectiva de Aplicação apresentada nos trabalhos relacionados

forma limitada pelas aplicações.

OSCIN difere das propostas existentes, permitindo que as perspectivas de aplicação sejam formalizadas de diferentes maneiras (por exemplo, como consultas SPARQL ou mapeamento de conceitos) sobre um modelo unificado que representa um determinado subdomínio do código-fonte (por exemplo, o subdomínio do código-fonte orientado a objeto ou mapeamento objeto/relacional). Esta formalização permite a aplicação da perspectiva em qualquer linguagem de programação, desde que esteja incluída no subdomínio fornecido e os analisadores necessários tenham sido construídos (por exemplo, Java e Python). É importante notar que a representação semântica do código-fonte torna possível explorar e desenvolver diferentes perspectivas no futuro.

Nesse sentido, em nossas pesquisas, não observamos trabalhos próximos ao método OSCIN no que se refere a um método comum para a representacao e interoperabilidade de código-fonte baseada em ontologia, e nem ao *framework* OSCINF no que se refere a um *framework* que facilite a adoção dessas técnicas por pessoas não especialistas na área e que permita a flexibilidade de novas linguagens, domínios e perspectivas.

8 Considerações Finais

Este capítulo apresenta as principais conclusões desta pesquisa. A Seção 8.1 sumariza a visão geral desta pesquisa, a Seção 8.2 destaca as contribuições e a Seção 8.3 discute os trabalhos futuros.

8.1 Visão Geral da Pesquisa

Conforme já discutido nos capítulos anteriores, esta pesquisa aborda o problema da interoperabilidade semântica em código-fonte, uma vez que o código-fonte de diferentes linguagens de programação podem ser representados de diferentes maneiras. Portanto, as diferenças sintáticas e semânticas das linguagens de programação tornam difícil a troca direta de informações entre códigos-fonte de diferentes linguagens.

Para mitigar a heterogeneidade das diferentes linguagens de programação, definimos uma representação semântica do domínio de código-fonte baseada em ontologia (organizada em uma rede de ontologias), denominada **SCON** — *Source Code Ontology Network*, que é atualmente formada pela OOC-O — *Object-Oriented Code Ontology*, RDBS-O — *Relational Database System Ontology* e ORM-O — *Object/Relational Mapping Ontology*. Essa rede de ontologias adotou o método SABiOS — Systematic Approach for Building Ontologies with SCRUM, uma extensão do método SABiO inspirada em princípios ageis que define as fases de *Conception*, *Pre Reference*, *Reference*, *Pre Operational* e *Operational*, detalhando os processos principais de *Requirement*, *Capture*, *Design* e *Implementation* e processos de suporte *Knowledge Acquisition*, *Reuse*, *Configuration Management*, *Evaluation*, *Documentation* e *Publish*. A avaliação da rede de ontologias foi realizada com as atividades de verificação e validação da ontologia conforme o método SABiOS.

Para aplicar a representação semântica de código-fonte a favor da interoperabilidade de código-fonte, propomos o método **OSCIN** – *Ontology-based Source Code Interoperability*, um método de interoperabilidade de código-fonte baseado em ontologia que visa representar semanticamente o código-fonte de diferentes linguagens de programação e aplicá-lo sob diferentes propósitos de forma unificada. O método é baseado (i) no subdomínio que indica a parte do domínio mais geral do código-fonte a ser representado em uma ontologia bem fundamentada e que reúne a conceituação desse subdomínio; (ii) na linguagem de programação que pretendemos representar em uma estrutura sintática da linguagem que inclui seus principais elementos. Tais representações são mapeadas de forma a atribuir a semântica do subdomínio aos elementos da linguagem de programação; e (iii) no propósito da aplicação que indica o tipo de aplicação que será tratada sobre o código-fonte usando a representação semântica do código. A partir de OSCIN, implementamos o *framework*

OSCINF a fim de fornecer um conjunto de soluções para apoiar a aplicação do método OSCIN em diferentes subdomínios de código-fonte, linguagens de programação e propósitos de aplicação.

Como forma de avaliação, o método OSCIN, por meio do *framework* OSCINF e de outras aplicações, é utilizado de forma a representar diferentes subdomínios (orientação a objetos e mapeamento objeto/relacional), utilizando diferentes linguagens de programação (Java e Python) e aplicação em diferentes propósitos (detecção de *bad smell*, métricas de código e migração de código), validando a hipótese de aplicação da interoperabilidade semântica entre códigos-fonte.

8.2 Contribuições da Pesquisa

Problemas relacionados à interoperabilidade semântica têm sido abordados na área de Engenharia de Ontologias e relacionados ao código-fonte, na área de Engenharia de Software. No entanto, embora esses problemas sejam constantemente tratados nessas áreas, pouco se vê sobre a interseção desses problemas e como lidar com a interoperabilidade semântica em código-fonte.

Assim, para verificar a hipótese de que "uma representação semântica do domínio de código-fonte baseada em ontologia (organizada em uma rede de ontologias) fornece uma conceituação consensual, compartilhada e abrangente do código-fonte, que pode mitigar a heterogeneidade das diferentes linguagens de programação e possibilitar a interoperabilidade semântica entre os códigos-fonte", definida na Seção 1.2, cumprimos os objetivos definidos na Seção 1.3.

Investigamos o estado da arte do domónio de código-fonte em diferentes linguagens de programação, conhecendo e mapeando suas sintaxes específicas a fim de definir os conceitos comuns do domínio, atendendo ao objetivo **OE01**. A partir dessa investigação, elaboramos a rede de ontologias SCON representando os conceitos do código-fonte de forma consensual e independente da linguagem de programação, atendendo ao objetivo **OE2**. Com a representação semântica do domínio do código-fonte, estabelecemos o método OSCIN que utiliza essa representação consensual para trocar informações entre códigos-fonte de diferentes linguagens de programação, atendendo ao objetivo **OE03**. Por fim, para aplicar o método e a rede de ontologias em prol da interoperabilidade do código-fonte, OSCIN foi operacionalizado no *framework* OSCINF e usado em códigos-fonte de diferentes linguagens de programação, validando que é possível trocar informações entre códigos-fonte de diferentes linguagens de programação a partir de uma representação semântica única e consensual, atendendo ao objetivo **OE04**.

Embora a pesquisa contribua para o estado da arte e prática na interoperabilidade semântica do código-fonte, reconhecemos que os resultados alcançados merecem uma

exploração e validação adicionais para confirmar a generalidade do método OSCIN para outros domínios, linguagens e propósitos, bem como para confirmar a precisão do *framework* OSCINF na geração desses artefatos. Atualmente, o método e o *framework* cobrem os subdomínios de orientação a objetos e mapeamento objeto/relacional, linguagens de programação Java e Python e propósito de detecção de *smell* e métrica de código. Além dessas limitações, as ameaças relacionadas à validação da pesquisa já foram discutidas em seção anterior (Seção 6.5).

Ao longo da pesquisa, cujos resultados foram compilados nesta tese, as seguintes contribuições foram publicadas em conferências com revisão por pares (*peer-review*):

- AGUIAR, C. Z. D.; FALBO, R. D. A.; SOUZA, V. E. S. **Ontological Representation of Relational Databases**. In: Proc. of the 11th Seminar on Ontology Research in Brazil (ONTOBRAS 2018). CEUR, 2018. p. 140-151.
- AGUIAR, C. Z. D.; FALBO, R. D. A.; SOUZA, V. E. S. OOC-O: **A reference ontology on object-oriented code**. In: International Conference on Conceptual Modeling. Springer, Cham, 2019. p. 13-27.
- ZANETTI, F. L.; AGUIAR, C. Z. D.; SOUZA, V. E. S. **Representação ontológica de frameworks de mapeamento objeto/relacional**. In: Proc. of the 12th Seminar on Ontology Research in Brazil (ONTOBRAS 2019). CEUR, Porto Alegre, RS, Brasil. 2019.
- AGUIAR, C. Z. D.; ZANETTI, F. L.; SOUZA, V. E. S. **Source Code Interoperability based on Ontology**. In: XVII Brazilian Symposium on Information Systems (SBSI). 2021. p. 1-8.

8.3 Trabalhos Futuros

Trabalhos futuros estão direcionados ao amadurecimento e expansão das contribuições apresentadas na Seção 8.2, destacadas a seguir:

- **SCON:**
  - Expansão da rede SCON a partir da construção de ontologias de diferentes paradigmas de programação, tais como, estruturado, procedural e orientado a eventos, bem como diferentes *frameworks*, tais como, *front-end*. Atualmente, as ontologias *Dependency Injection Framework Code Ontology* – **DINF-O** e *Functional Code Ontology* – **FUNC-O**, que retratam os domínios de injeção de depência e paradigma de programação funcional (respectivamente) estão em construção por outros estudos do mesmo grupo de trabalho;

  - Aplicação de nova análise ontológica nas ontologias de SCON referente à definição de cardinalidade em relações de *characterization* e *componenteOf*, bem como da definição de relações semânticas. Novas versões da rede SCON podem ser acompanhadas pelo site do projeto <https://nemo.inf.ufes.br/projetos/oscin/scon>.
  - Avaliação das ontologias de SCON através de simulação em Alloy.

- **OSCIN:**
  - Amadurecimento do método OSCIN a partir de sua aplicação e análise, bem como sua avaliação para outros subdomínios, linguagens de programação e propósitos;
  - Investigação do propósito de *bad smell* a fim de adicionar os *smells* definidos em catálogos da literatura e avaliá-lo com outros trabalhos e ferramentas;

- **OSCINF:**
  - Aplicação do *framework* OSCINF a ambientes reais de desenvolvimento de software, bem como sua evolução para outros subdomínios, linguagens de programação e propósitos;
  - Evolução de OSCINF como uma ferramenta configurável que visa a interoperabilidade do código-fonte em diferentes linguagens de programação, bem como sua compração com outras ferramentas disponibilizadas pela comunidade;
  - Análise e comparação dos resultados de OSCINF para o propósito de detecção de *smells* com outras ferramentas disponibilizadas pela comunidade;

- **SABiOS:**
  - Amadurecimento do método SABiOS a partir de sua aplicação em outros domínios além de código-fonte, bem como sua comparação com os outros métodos de Engenharia de Ontologias da literatura;
  - Disponibilização de um guia SABiOS para facilitar sua utilização e de um repositório para compartilhar e evoluir ontologias baseadas em SABiOS.

Referências

ABBEY, M.; COREY, M.; ABRAMSON, I. Oracle9i–guia introdutório–aprenda os fundamentos do oracle 9i. *Editora Campus, Rio de Janeiro–RJ*, 2002. Citado na página 57.

ABDELGHANY, A. S.; DARWISH, N. R.; HEFNI, H. A. An agile methodology for ontology development. *International Journal of Intelligent Engineering and Systems*, v. 12, n. 2, p. 170–181, 2019. ISSN 21853118. Citado na página 148.

AGUIAR, C. Z. D.; FALBO, R. D. A.; SOUZA, V. E. S. OOC-O : A Reference Ontology on Object-Oriented Code. In: *International Conference on Conceptual Modeling*. [S.l.]: Springer International Publishing, 2019. p. 13–27. Citado na página 77.

AMBLER, S. W. *Mapping objects to relational databases*. 2010. <https://www.ibm.com/developerworks/library/ws-mapping-to-rdb/>. Acessado em : 05-05-2019. Citado na página 61.

API, O. *Java OWL API*. 2020. <http://owlcs.github.io/owlapi/>. Acessado em : 15/04/2021. Citado na página 125.

ATKINSON, M. et al. The object-oriented database system manifesto. In: *Deductive and object-oriented databases*. [S.l.]: Elsevier, 1990. p. 223–240. Citado na página 64.

BERNERS-LEE, T.; HENDLER, J.; LASSILA, O. The Semantic Web. *Scientific American*, 2001. Citado na página 27.

BISSYANDÉ. Popularity , Interoperability , and Impact of Programming Languages in 100,000 Open Source Projects. In: *IEEE 37th Annual Computer Software and Applications Conference*. [S.l.: s.n.], 2013. p. 303–312. Citado na página 17.

BOELL, S. K. What is an Information System ? 2015. Citado na página 36.

BOOCH, G. Coming of age in an object-oriented world. *IEEE Software*, IEEE, v. 11, n. 6, p. 33–41, 1994. Citado na página 45.

BRUN, R.; RADEMAKERS, F. Root—an object oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, Elsevier, v. 389, n. 1-2, p. 81–86, 1997. Citado na página 64.

CALERO, C. et al. An ontological approach to describe the SQL: 2003 object-relational features. *Computer Standards & Interfaces*, Elsevier, v. 28, n. 6, p. 695–713, 2006. Citado na página 65.

CARVALHO, S. G. et al. An empirical catalog of code smells for the presentation layer of Android apps. *Empirical Software Engineering*, v. 24, n. 6, p. 3546–3586, dec 2019. ISSN 1382-3256. Disponível em: <http://link.springer.com/10.1007/s10664-019-09768-9>. Citado na página 74.

CHANG, D.; IYENGAR, S. et al. Common warehouse metamodel (CWM) specification. *Object Management Group*, v. 1, 2001. Citado na página 66.

CONAWAY, C. F.; PAGE-JONES, M.; CONSTANTINE, L. L. *Fundamentals of object-oriented design in UML*. [S.l.]: Addison-Wesley Professional, 2000. Citado na página 45.

CYGANIAK, R.; JENTZSCH, A. *Linking Open Data cloud diagram*. 2021. <https://github.com/paul-hammant/qdox>. Acessado em : 15/04/2021. Citado na página 124.

Da Silva Carvalho, L. P. et al. An ontology-based approach to analyzing the occurrence of code smells in software. *ICEIS 2017 - Proceedings of the 19th International Conference on Enterprise Information Systems*, v. 2, n. Iceis, p. 155–165, 2017. Citado 2 vezes nas páginas 118 e 125.

DANPHITSANUPHAN, P.; SUWANTADA, T. Code smell detecting tool and code smell-structure bug relationship. In: *2012 Spring Congress on Engineering and Technology*. [S.l.]: IEEE, 2012. p. 1–5. Citado na página 102.

D'AQUIN, M. et al. Ontology Modularization for Knowledge Selection: Experiments and Evaluations. In: *International Conference on Database and Expert Systems Applications*. [s.n.], 2007. ISBN 9783540744696. Disponível em: <http://link.springer.com/10.1007/s10664-019-09768-9>. Citado na página 174.

DATE, C. J. *Introdução a sistemas de bancos de dados*. [S.l.]: Elsevier Brasil, 2004. Citado 6 vezes nas páginas 54, 55, 56, 57, 58 e 60.

DIMARIO, M. J. System of Systems Interoperability Types and Characteristics in Joint Command and Control. In: *IEEE/SMC International Conference on System of Systems Engineering*. [S.l.: s.n.], 2006. p. 236–241. ISBN 1424401887. Citado 2 vezes nas páginas 26 e 27.

DJANGO. *Django Documentation*. 2019. <https://docs.djangoproject.com/en/2.2/>. Acessado em : 05-05-2019. Citado na página 60.

DRESCH, A.; LACERDA, D. P.; ANTUNES, J. A. V. *Design Science Research*. Springer International Publishing, 2015. ISBN 978-3-319-07373-6. Disponível em: <http://link.springer.com/10.1007/978-3-319-07374-3>. Citado na página 20.

D'AQUIN, M. Modularizing ontologies. In: *Ontology Engineering in a Networked World*. [S.l.]: Springer, 2012. p. 213–233. Citado 2 vezes nas páginas 174 e 175.

EIFFEL, E. Analysis, design and programming language. *ECMA Standard ECMA-367, ECMA*, 2006. Citado 2 vezes nas páginas 49 e 168.

ELMASRI, R.; NAVATHE, S. *Database systems*. [S.l.]: Pearson Education Boston, MA, 2011. v. 9. Citado 3 vezes nas páginas 54, 55 e 56.

EUZENAT, J.; SHVAIKO, P. *Ontology matching*. [S.l.: s.n.], 2007. 1–333 p. ISBN 9783540496113. Citado na página 28.

EVERMANN, J.; WAND, Y. Ontology based object-oriented domain modelling:

fundamental concepts. *Requirements engineering*, Springer, v. 10, n. 2, p. 146–160, 2005. Citado na página 64.

FALBO, R. A. et al. Ontology patterns: Clarifying concepts and terminology. *CEUR Workshop Proceedings*, v. 1188, 2013. ISSN 16130073. Citado 2 vezes nas páginas 28 e 161.

FALBO, R. d. A. SABiO: Systematic approach for building ontologies. *CEUR Workshop Proceedings*, v. 1301, 2014. ISSN 16130073. Citado 12 vezes nas páginas 9, 23, 29, 33, 60, 109, 148, 156, 161, 173, 175 e 189.

FARINELLI, F.; MELO, S.; ALMEIDA, M. B. O Papel das Ontologias na Interoperabilidade de Sistemas de Informação: Reflexões na esfera governamental. In: *Encontro Nacional de Pesquisa em Ciência da Informação (ENANCIB)*. [S.l.: s.n.], 2013. ISBN 9788578110796. ISSN 1098-6596. Citado na página 18.

FERNÁNDEZ-LÓPEZ, M. METHONTOLOGY: From Ontological Art Towards Ontological Engineering. *AAAI Technical Report*, p. 33–40, 1997. Citado na página 28.

FERNÁNDEZ-LÓPEZ, M.; GÓMEZ-PÉREZ, A. Overview and analysis of methodologies for building ontologies. *Knowledge Engineering Review*, v. 17, n. 2, p. 129–156, 2002. ISSN 02698889. Citado 3 vezes nas páginas 148, 163 e 171.

FERRAGINA, P.; SCAIELLA, U. Fast and accurate annotation of short texts with wikipedia pages. *IEEE software*, IEEE, v. 29, n. 1, p. 70–75, 2011. Citado na página 126.

FILETO, R.; MEDEIROS, C. *A Survey on Information Systems Interoperability*. [S.l.], 2003. Citado na página 26.

FJELDBERG, H.-c. Polyglot Programming. n. June, 2008. Citado na página 18.

FORD, T. C. et al. A Survey on Interoperability Measurement. In: *12th ICCRTS*. [S.l.: s.n.], 2007. Citado 2 vezes nas páginas 17 e 26.

GANAPATHY, G.; SAGAYARAJ, S. To Generate the Ontology from Java Source Code. *International Journal of Advanced Computer Science and Applications*, v. 2, n. 2, p. 111–116, 2011. ISSN 2158107X. Citado na página 123.

GASEVIC, D.; DJURIC, D.; DEVEDZIC, V. *Model Driven Architecture and Ontology Development Model Driven Architecture and Ontology Development*. [S.l.: s.n.], 2006. ISBN 9783540321804. Citado na página 171.

GIFFHORN, D.; HAMMER, C. Precise analysis of Java programs using JOANA. *Proceedings - 8th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2008*, p. 267–268, 2008. Citado na página 121.

GÓMEZ-PÉREZ, A.; ROJAS-AMAYA, M. D. Ontological reengineering for reuse. *International Conference on Knowledge Engineering and Knowledge Management*, v. 1621, p. 139–156, 1999. ISSN 16113349. Citado na página 170.

GOSLING, J. et al. *The Java language specification: Java SE 10 edition, 20 February 2018*. 2018. Citado 3 vezes nas páginas 49, 91 e 168.

GRENON, P.; SMITH, B. SNAP and SPAN: Prolegomenon to geodynamic ontology. *Spatial Cognition and Computation*, v. 1, n. January, p. 69–105, 2003. Disponível em:

<http://ontology.buffalo.edu/smith/courses03/tb/SNAP{_}SPAN.> Citado na página 28.

GRUBER, T. R. *A translation approach to portable ontology specifications*. 1993. 199–220 p. Disponível em: <http://ac.els-cdn.com/S1042814383710083/1-s2.0-S1042814383710083-main.pdf?{_}tid=28c5c43c-df11-11e2-b19c-00000aacb35f{&}acdnat=1372327738{_}c3518b0aab22af436aef0c5>. Citado na página 27.

GRÜNINGER, M.; FOX, M. S.; GRUNINGER, M. Methodology for the Design and Evaluation of Ontologies. *International Joint Conference on Artificial Inteligence (IJCAI95), Workshop on Basic Ontological Issues in Knowledge Sharing*, p. 1–10, 1995. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.44.8723>. Citado na página 156.

GUARINO, N. et al. Towards a Methodology for Ontology Based Model Engineering. *Proceedings of International Workshop on Model Engineering (IWME-2000): Nice, France: 2000, June, 13*, 2000. Citado na página 29.

GUARINO, N.; OBERLE, D.; STAAB, S. What Is an Ontology ? p. 1–17, 2009. Citado na página 19.

GUIZZARDI, G. *Ontological foundations for conceptual modeling with applications*. 695–696 p. Tese (Doutorado), 2005. Citado 6 vezes nas páginas 28, 29, 37, 77, 87 e 88.

GUIZZARDI, G. On ontology, ontologies, conceptualizations, modeling languages, and (meta)models. *Frontiers in Artificial Intelligence and Applications*, v. 155, n. January 2006, p. 18–39, 2007. ISSN 09226389. Citado 2 vezes nas páginas 19 e 180.

GUIZZARDI, G. Ontology, Ontologies and the "I" of FAIR Giancarlo. *Data Intelligence*, v. 23, n. November, p. 0–2, 2019. Citado 2 vezes nas páginas 29 e 68.

GUIZZARDI, G.; FALBO, R.; GUIZZARDI, R. S. Grounding software domain ontologies in the Unified foundational ontology (UFO): The case of the ODE software process ontology. *Memorias de la 11th Conferencia Iberoamericana de Software Engineering - CIbSE 2008*, n. i, 2008. Citado na página 29.

GUIZZARDI, G.; GRAÇAS, A. P.; GUIZZARDI, R. S. Design patterns and inductive modeling rules to support the construction of ontologically well-founded conceptual models in OntoUML. In: *International Conference on Advanced Information Systems Engineering*. [S.l.: s.n.], 2011. v. 83 LNBIP, p. 402–413. ISBN 9783642220555. ISSN 18651348. Citado 2 vezes nas páginas 23 e 161.

GUIZZARDI, G.; WAGNER, G. A Unified Foundational Ontology and some Applications of it in Business Modeling. *Proc. of the 2004 Open InterOp Workshop on Enterprise Modelling and Ontologies for Interoperability*, p. 161–177, 2004. Citado 3 vezes nas páginas 23, 76 e 161.

GUIZZARDI, G. et al. Towards ontological foundations for conceptual modeling: The unified foundational ontology (UFO) story. *Applied Ontology*, v. 10, n. 3-4, p. 259–271, 2015. ISSN 18758533. Citado na página 31.

HEVNER, A. R. A Three Cycle View of Design Science Research. *Scandinavian*

Journal of Information Systems, v. 19, n. 2, p. 87–92, 2007. Disponível em: <https://www.researchgate.net/publication/254804390{_}A{_}Three{_}Cycle{_}View{_}of{_}Desig>. Citado na página 21.

HEVNER, A. R. et al. Design science in information systems research. *MIS Quarterly: Management Information Systems*, v. 28, n. 1, p. 75–105, 2004. ISSN 02767783. Citado na página 20.

HUNT, J. *Java and object orientation: an introduction*. [S.l.]: Springer Science & Business Media, 2002. Citado na página 50.

HUNT, J. *Smalltalk and object orientation: an introduction*. [S.l.]: Springer Science & Business Media, 2012. Citado na página 168.

IDE, E. *Eclipse IDE*. 2020. <http://eclipse.org>. Acessado em : 15/04/2021. Citado 2 vezes nas páginas 123 e 124.

IDE, E. *Eclipse Java development tools (JDT)*. 2020. <https://projects.eclipse.org/projects/eclipse.jdt>. Acessado em : 15/04/2021. Citado na página 125.

ISO/IEC10746-2. *Information technology — Open distributed processing — Reference model: Foundations — Part 2*. [S.l.], 2009. Citado na página 26.

ISO/IEC19500-2. *Information technology — Open Distributed Processing — Part 2: General Inter-ORB Protocol (GIOP)/Internet Inter-ORB Protocol (IIOP)*. [S.l.], 2003. Citado na página 26.

ISO/IEC19506. Information technology - Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM). v. 2012, n. April, 2012. Citado na página 122.

ISO/IEC2382. *Information technology — Vocabulary*. [S.l.], 2015. Citado na página 26.

ISO/IEC25010. *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. [S.l.], 2011. Citado na página 26.

ISO/IEC9075-1. *Information technology–Database languages–SQL–Part 1: Framework (SQL/Framework)*. [S.l.]: International Organization for Standardization Geneva„ Switzerland, 2008. Citado 2 vezes nas páginas 55 e 56.

ISO/IEC9075-2. Information technology–database languages–sql–part 2: Foundation (sql/foundation). *ISO/IEC*, 2003. Citado 3 vezes nas páginas 56, 57 e 58.

JARRAR, M.; MEERSMAN, R. Formal Ontology Engineering in the DOGMA Approach. *OTM Confederated International Conferences - On the Move to Meaningful Internet Systems*, v. 2519 LNCS, p. 1238–1254, 2002. ISSN 03029743. Citado na página 28.

JPA. *Java Persistence API Documentation*. 2019. <https://download.oracle.com/otn-pub/jcp/persistence-2_2-mrel-spec/JavaPersistence.pdf>. Acessado em : 05-05-2019. Citado na página 60.

KEIVANLOO, I.; RILLING, J.; CHARLAND, P. Semantic Web - The Missing Link in Global Source Code Analysis ? 2012. Citado na página 124.

KIEFER, C.; BERNSTEIN, A.; TAPPOLET, J. Analyzing Software with iSPARQL. In: *International Workshop on Semantic Web Enabled Software Engineering (SWESE)*. [S.l.: s.n.], 2007. Citado 3 vezes nas páginas 102, 118 e 123.

KOSANKE, K. *ISO Standards for Interoperability : a Comparison*. [S.l.], 2005. Citado na página 26.

KOTIS, K. I.; VOUROS, G. A.; SPILIOTOPOULOS, D. Ontology engineering methodologies for the evolution of living and reused ontologies: Status, trends, findings and recommendations. *Knowledge Engineering Review*, v. 35, 2020. ISSN 14698005. Citado na página 148.

KOUNELI, A. et al. Modeling the knowledge domain of the java programming language as an ontology. In: SPRINGER. *International Conference on Web-Based Learning*. [S.l.], 2012. p. 152–159. Citado na página 64.

KUMAR, R.; SINGH, J. A unique code smell detection and refactoring scheme for evaluating software maintainability. *International Journal of Latest Trends in Engineering and Technology*, SN Education Society, v. 7, p. 421–436, 2016. Citado na página 102.

LABORDA, C. P.; CONRAD, S. Relational.OWL - A data and schema representation format based on OWL. In: *Asia-Pacific Conference on Conceptual Modelling (APCCM2005)*. [S.l.]: Australian Computer Society, 2005. v. 43, p. 89–96. ISBN 1920682252. ISSN 14451336. Citado na página 65.

LAFORE, R. *Object-oriented programming in C++*. [S.l.]: Pearson Education, 1997. Citado na página 168.

LALONDE, W. R.; PUGH, J. R. *Inside smalltalk*. [S.l.]: Prentice Hall, 1990. v. 2. Citado na página 47.

LANGUAGES, P. ISO/IEC 22 Bulletin 2001. p. 2–4, 2001. Citado na página 122.

LASSILA, O.; SWICK, R. R. Resource description framework (RDF) model and syntax specification. World Wide Web Consortium Recommendation. n. October, 1999. Disponível em: <http://www.w3.org/TR/REC-rdf-syntax>. Citado na página 32.

LAVEAN, G. E. Interoperability in Defense Communications. C, n. 9, 1980. Citado na página 26.

LU, D. et al. Analysis of the popularity of programming languages in open source software communities. In: *International Conference on Big Data and Social Sciences*. [S.l.: s.n.], 2020. p. 111–114. ISBN 9781728197517. Citado na página 17.

MARINESCU, C. A Meta-Model for Enterprise Applications. In: *International Symposium on Symbolic and Numeric Algoritms for Scientific Computing*. [S.l.: s.n.], 2006. ISBN 076952740X. Citado na página 121.

MASCI, P.; MUÑOZ, C. A. An Integrated Development Environment for the Prototype Verification System. In: *Integrated Development Environment*. [S.l.: s.n.], 2019. v. 2019, p. 35–49. Citado na página 121.

MASOLO, C. et al. Wonderweb deliverable d18 - ontology library (final). Technical Report

D18: http://wonderweb.man.ac.uk/deliverables/documents/D18.pdf. n. June, p. 343, 2003. Disponível em: <http://wonderweb.man.ac.uk/deliverables/documents/D18.pdf>. Citado na página 28.

MCBRIDE, B. Jena: A semantic web toolkit. *IEEE Internet computing*, IEEE, v. 6, n. 6, p. 55–59, 2002. Citado 3 vezes nas páginas 123, 124 e 126.

MCGUINESS, D. L.; HARMELEN, F. V. *OWL Web Ontology Language Overview*. [S.l.], 2004. 1–22 p. Disponível em: <http://www.w3.org/TR/owl-features/>. Citado 2 vezes nas páginas 23 e 180.

MEKRUKSAVANICH, S. Identifying behavioral design flaws in evolving object-oriented software using an ontology-based approach. *Proceedings - 13th International Conference on Signal-Image Technology and Internet-Based Systems, SITIS 2017*, p. 424–429, 2017. Citado na página 125.

MELTON, J.; SIMON, A. R. *SQL1999 understanding relational language components*. [S.l.]: Elsevier, 2001. Citado 2 vezes nas páginas 56 e 57.

MITRA, P.; WIEDERHOLD, G. An Ontology-Composition Algebra. In: *Handbook on Ontologies*. [S.l.: s.n.], 2004. Citado na página 173.

MOHA, N. et al. DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, v. 36, n. 1, p. 20–36, 2010. ISSN 00985589. Citado na página 118.

MUSEN, M. A. The protégé project: a look back and a look forward. *AI matters*, ACM New York, NY, USA, v. 1, n. 4, p. 4–12, 2015. Citado na página 123.

MUSHTAQ, Z.; RASOOL, G.; SHEHZAD, B. Multilingual Source Code Analysis: A Systematic Literature Review. *IEEE Access*, v. 5, p. 11307–11336, 2017. ISSN 21693536. Citado 3 vezes nas páginas 18, 120 e 121.

NEURATH, O. *Foundations of the Unity of Science*. 1938. Citado na página 36.

NICOLA, A.; MISSIKOFF, M. A lightweight methodology for rapid ontology engineering. *Communications of the ACM*, v. 59, n. 3, p. 79–86, 2016. ISSN 15577317. Citado na página 167.

NICOLA, A.; MISSIKOFF, M.; NAVIGLI, R. A proposal for a unified process for ontology building: UPON. In: *International Conference on Database and Expert Systems Applications*. [S.l.]: Springer International Publishing, 2005. v. 3588, p. 655–664. ISSN 03029743. Citado na página 29.

NIEPHAUS, F.; FELGENTRE, T.; HIRSCHFELD, R. GraalSqueak Toward a Smalltalk-Based Tooling Platform for Polyglot Programming. In: *ACM SIGPLAN Interna- tional Conference on Managed Programming Languages and Run- times (MPLR '19)*. [S.l.: s.n.], 2019. ISBN 9781450369770. Citado na página 18.

NIEPHAUS, F. et al. Live Multi-language Development and Runtime Environments. *arXiv*, v. 2, n. 3, 2018. ISSN 2473-7321. Citado 2 vezes nas páginas 18 e 121.

ODB. *ODB Documentation*. 2019. <https://www.codesynthesis.com/products/odb/doc/manual.xhtml>. Acessado em : 05-05-2019. Citado na página 60.

OUKSEL, A. M.; SHETH, A. Semantic Interoperability in Global Information Systems: A brief introduction to the research area and the special section. *SIGMOD Record*, v. 28, n. 1, p. 5–12, 1999. ISSN 01635808. Citado na página 21.

PARK, J. Information Systems Interoperability : What Lies Beneath ? v. 22, n. 4, p. 595–632, 2004. Citado na página 27.

PARR, T. *The definitive ANTLR 4 reference*. [S.l.]: Pragmatic Bookshelf, 2013. Citado na página 123.

PASTOR, O. *Diseño y Desarrollo de un Entorno de Producción Automática de Software basado en el modelo orientado a Objetos*. Tese (Doutorado) — Tesis doctoral dirigida por Isidro Ramos, DSIC, Universitat Politècnica de . . . , 1992. Citado na página 64.

PASTOR, O. et al. Linking Object-Oriented Conceptual Modeling with Object-Oriented Implementation in Java. In: *Database and Expert Systems Applications, 8th International Conference, DEXA'97, Toulouse, France, September 1-5, 1997, Proceedings*. [S.l.: s.n.], 1997. p. 132–141. Citado na página 64.

PATTIPATI, D. K.; NASRE, R.; PULIGUNDLA, S. K. OPAL: An extensible framework for ontology-based program analysis. *Software - Practice and Experience*, v. 50, n. 8, p. 1425–1462, 2020. ISSN 1097024X. Citado na página 126.

PAWLAK, R. et al. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, Wiley Online Library, v. 46, n. 9, p. 1155–1179, 2016. Citado na página 126.

PAYDAR, S.; KAHANI, M. A Semantic Web based approach for design pattern detection from source code. *2012 2nd International eConference on Computer and Knowledge Engineering, ICCKE 2012*, p. 289–294, 2012. Citado na página 124.

PETER, H. et al. *NeOn: Lifecycle Support for Networked Ontologies*. [S.l.], 2006. 1–60 p. Citado na página 39.

PFEIFFER, R.-H.; WASOWSKI, A. TexMo A Multilanguage Development. In: *European Conference on Modelling Foundations and Applications*. [S.l.: s.n.], 2012. Citado 2 vezes nas páginas 18 e 121.

PHILLIPS, D. *Python 3 object oriented programming*. [S.l.]: Packt Publishing Ltd, 2010. Citado na página 169.

PINTO, H. S.; TEMPICH, C.; STAAB, S. Ontology Engineering and Evolution in a Distributed World Using DILIGENT. *Handbook on Ontologies*, n. May, 2009. Citado na página 29.

QDOX. *QDOX*. 2021. <https://github.com/paul-hammant/qdox>. Acessado em : 15/04/2021. Citado na página 124.

QUINLAN, D.; LIAO, C. The ROSE source-to-source compiler infrastructure. In: CITESEER. *Cetus users and compiler infrastructure workshop, in conjunction with PACT*. [S.l.], 2011. v. 2011, p. 1. Citado 2 vezes nas páginas 125 e 126.

QXORM. *QxOrm Documentation*. 2019. <https://www.qxorm.com/qxorm_en/manual.html>. Acessado em : 05-05-2019. Citado na página 60.

RASK, J. K. et al. The Specification Language Server Protocol : A Proposal for Standardised LSP Extensions Decoupling Using Language-Neutral Protocols. v. 2021, p. 3–18, 2021. Citado 2 vezes nas páginas 18 e 121.

RASOOL, G.; ARSHAD, Z. A Lightweight Approach for Detection of Code Smells. *Arabian Journal for Science and Engineering*, v. 42, n. 2, p. 483–506, 2017. ISSN 21914281. Citado na página 118.

ROUSSEY, C. et al. An Introduction to Ontologies and Ontology Engineering. In: *Ontologies in Urban Development Projects*. [s.n.], 2011. v. 1, p. 9–38. ISBN 978-0-85729-723-5. Disponível em: <http://www.springerlink.com/content/w668411565608515/>. Citado 2 vezes nas páginas 29 e 76.

RUY, F. B. et al. SEON: A software engineering ontology network. In: *European Knowledge Acquisition Workshop*. [S.l.]: Springer International Publishing, 2016. v. 10024 LNAI, p. 527–542. ISBN 9783319490038. ISSN 16113349. Citado 5 vezes nas páginas 9, 35, 36, 39 e 40.

SANTOS, F. Network of Ontologies – A Systematic Mapping Study and Challenges Comparison Network of Ontologies – A Systematic Mapping Study and Challenges Comparison *. 2017. Citado na página 19.

SCHAUB, S.; MALLOY, B. A. The design and evaluation of an interoperable translation system for object-oriented software reuse. *Journal of Object Technology*, v. 15, n. 4, p. 1–33, 2016. ISSN 16601769. Citado na página 118.

SCHERP, A. et al. Designing core ontologies. *Applied Ontology*, v. 6, n. 3, p. 177–221, 2011. ISSN 15705838. Citado na página 28.

SCHINK, H. et al. Hurdles in multi-language refactoring of hibernate applications. *ICSOFT 2011 - Proceedings of the 6th International Conference on Software and Database Technologies*, v. 2, p. 129–134, 2011. Citado 2 vezes nas páginas 18 e 121.

SCHWABER, K.; SUTHERLAND, J. *The Scrum Guide*. [S.l.], 2020. Citado 2 vezes nas páginas 89 e 150.

SEBESTA, R. W. *Concepts of programming languages*. [S.l.]: Boston: Pearson„ 2012. Citado na página 50.

SEPTEMBER, A. IEEE Standard Glossary of Software Engineering Terminology. v. 121990, 1990. Citado na página 26.

SHETH, A. P. Changing Focus on Interoperability in Information Systems: from Systems, Syntax, Structures to Semantics. *Interoperating Geographic Information Systems*, 1999. Citado 2 vezes nas páginas 17 e 37.

SIEMUND, F.; TOVESSON, D. Language Server Protocol for ExtendJ. In: . [s.n.], 2018. p. 1–5. Disponível em: <https://microsoft.github.io/language-server-protocol/>. Citado 2 vezes nas páginas 18 e 121.

SIMILE Widgets. 2021. <https://www.simile-widgets.org>. Acessado em : 15/04/2021. Citado na página 124.

SLIMANI, T. A Study Investigating Knowledge-based Engineering Methodologies Analysis. *International Journal of Computer Applications*, v. 128, n. 10, p. 6–14, 2015. Citado 2 vezes nas páginas 148 e 182.

SLONNEGER, K.; KURTZ, B. L. *Formal Syntax and Semantics of Programming Languages A Laboratory Based Approach*. [S.l.: s.n.], 1995. ISBN 0201656973. Citado na página 36.

SOMMERVILLE, I. *Software engineering 9th Edition*. [S.l.: s.n.], 2011. Citado na página 17.

SQLALCHEMY. *SQLAlchemy Documentation*. 2019. <https://docs.sqlalchemy.org/en/13/>. Acessado em : 05-05-2019. Citado na página 60.

STREIN, D. Cross-Language Program Analysis and Refactoring. In: *International Workshop on Source Code Analysis and Manipulation*. [S.l.: s.n.], 2006. ISBN 0769523536. Citado na página 18.

STUDER, R.; BENJAMINS, V. R.; FENSEL, D. Knowledge Engineering: Principles and methods. *Data and Knowledge Engineering*, v. 25, n. 1-2, p. 161–197, 1998. ISSN 0169023X. Citado na página 27.

SUÁREZ-FIGUEROA, M. C.; GÓMEZ-PÉREZ, A. Ontology requirements specification. In: *Ontology Engineering in a Networked World*. [S.l.]: Springer, 2012. p. 93–106. Citado 2 vezes nas páginas 155 e 157.

SUÁREZ-FIGUEROA, M. C. et al. *Ontology Engineering in a Networked World*. [s.n.], 2012. v. 53. 9–34 p. ISSN 1098-6596. ISBN 9788578110796. Disponível em: <file:///C:/Users/User/Downloads/fvm939e.pdf>. Citado na página 29.

SWARTOUT, W.; TATE, A. Ontologies. *IEEE Intelligent Systems and Their Applications*, v. 14, n. 1, p. 18–19, 1999. ISSN 16877438. Citado na página 27.

TAYLOR, P. et al. System of Systems Engineering System of Systems Engineering. *Engineering Management Journal*, n. July 2015, 2015. Citado na página 36.

TEIXEIRA, M. H. T. H. Impedância objeto relacional — O atrito natural entre os dois mundos. *Tecnologias em Projeção*, v. 8, n. 1, p. 11–19, 2017. Citado na página 60.

TEMPERO, E. et al. Qualitas corpus: A curated collection of java code for empirical studies. In: *Proc. of the 2010 Asia Pacific Software Engineering Conference (APSEC2010)*. [S.l.]: IEEE, 2010. p. 336–345. Citado na página 111.

TERCEIRO, A.; COSTA, J.; MIRANDA, J. Analizo: an extensible multi-language source code analysis and visualization toolkit. *Brazilian Conference on Software: Theory and Practice (CBSoft)–Tools*, n. January, p. 1–6, 2010. Citado na página 121.

TICHELAAR, S. et al. A meta-model for language-independent refactoring. *International Workshop on Principles of Software Evolution (IWPSE)*, v. 2000-Janua, p. 154–164, 2000. Citado 2 vezes nas páginas 18 e 120.

TICHELAAR, S. et al. A meta-model for language-independent refactoring. In: IEEE. *Proceedings International Symposium on Principles of Software Evolution*. [S.l.], 2000. p. 154–164. Citado na página 123.

TRINH, Q.; BARKER, K.; ALHAJJ, R. Rdb2ont: A tool for generating owl ontologies from relational database systems. In: IEEE. *Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services (AICT-ICIW'06)*. [S.l.], 2006. p. 170–170. Citado na página 65.

TUCKER, A. B. *Programming languages> Principles and Paradigmas*. [S.l.]: Tata McGraw-Hill Education, 2007. Citado na página 49.

USCHOLD, M.; KING, M. Towards a Methodology for Building Ontologies. *Workshop on Basic Ontological Issues in Knowledge Sharing*, n. July, 1995. Citado na página 166.

VELTMAN, K. H. Syntactic and semantic interoperability: New approaches to knowledge and the semantic web. *New Review of Information Networking*, v. 7, n. November 2014, p. 159–183, 2001. ISSN 13614576. Citado na página 27.

VISMINER. *VisminerTD Analyzer*. 2018. <https://github.com/visminer>. Acessado em : 15/04/2021. Citado na página 125.

VRANDEC, D. et al. CodeOntology: RDF-ization of Source Code. *17th International Semantic Web Conference*, v. 10588, p. 175–183, 2018. Disponível em: <http://link.springer.com/10.1007/978-3-319-68204-4>. Citado 2 vezes nas páginas 21 e 126.

WAMPLER, D.; TONY, C. Multiparadigm Programming. *IEEE Software*, p. 20–24, 2010. Citado na página 18.

WANG, X. et al. Towards an ontology of software: A requirements engineering perspective. *Frontiers in Artificial Intelligence and Applications*, v. 267, n. September, p. 317–329, 2014. ISSN 09226389. Citado 3 vezes nas páginas 17, 35 e 36.

WIELEMAKER, J. et al. SWI-Prolog. *Theory and Practice of Logic Programming*, v. 12, n. 1-2, p. 67–96, 2012. ISSN 1471-0684. Citado na página 125.

WIELEMAKER, J. et al. Swi-prolog. *Theory and Practice of Logic Programming*, Cambridge University Press, v. 12, n. 1-2, p. 67–96, 2012. Citado na página 125.

WOHLIN, C. et al. *Experimentation in software engineering*. [S.l.]: Springer Science & Business Media, 2012. Citado na página 116.

YU, L. et al. Ontology model-based static analysis on java programs. *Proceedings - International Computer Software and Applications Conference*, p. 92–99, 2008. ISSN 07303157. Citado na página 123.

ZANETTI, F. L.; AGUIAR, C. Z. D.; SOUZA, V. E. S. Representacão Ontológica de Frameworks de Mapeamento Objeto/Relacional. In: *12th Seminar on Ontology Research in Brazil (ONTOBRAS)*. Porto Alegre - RS: [s.n.], 2019. Citado 4 vezes nas páginas 9, 61, 63 e 113.

ZANETTI, L. *Representação Ontológica de Frameworks de Mapeamento Objeto/Relacional*. Tese (Doutorado) — Universidade Federal do Espírito Santo, 2020. Citado 3 vezes nas páginas 39, 60 e 114.

ZHAO, Y. et al. Towards ontology-based program analysis. *Leibniz International Proceedings in Informatics, LIPIcs*, v. 56, n. 26, p. 261–2625, 2016. ISSN 18688969. Citado 2 vezes nas páginas 124 e 126.

Apêndices

APÊNDICE A – Abordagem Sistemática para Construção de Ontologias com Scrum

Conforme abordado na Seção 2.2, a Engenharia de Ontologias deve seguir um método bem definido, tratando aspectos práticos que permitam que comunidades de especialistas e ontologistas cheguem a um consenso sobre o domínio da ontologia, além de mantê-la viva e em evolução (KOTIS; VOUROS; SPILIOTOPOULOS, 2020). Assim, métodos de Engenharia de Ontologias visam proporcionar maior qualidade e confiabilidade às ontologias, estabelecendo um padrão para sua construção. No entanto, um método maduro para a construção de ontologias ainda não foi definido, uma vez que importantes atividades e técnicas de engenharia são perdidas (FERNÁNDEZ-LÓPEZ; GÓMEZ-PÉREZ, 2002), assim como detalhes suficientes do ciclo de vida de ontologia (ABDELGHANY; DARWISH; HEFNI, 2019; SLIMANI, 2015).

Neste contexto, identificamos o método SABiO — *Systematic Approach for Building Ontologies* (FALBO, 2014), apresentado na Seção 2.3, método proposto para a construção de ontologias de referência e operacional previamente analisadas à luz de uma ontologia de fundamentação. O método é formado por cinco processos principais: Identificação do Propósito e Elicitação de Requisitos, Captura e Formalismo da Ontologia, Design, Implementação e Teste, e cinco processos de suporte: Aquisição do Conhecimento, Documentação, Gerenciamento de Configuração, Avaliação e Reuso.

A partir da experiência de aplicação do método SABiO sobre o domínio de código-fonte por diferentes engenheiros de ontologia deste grupo de pesquisa e diante das lacunas identificadas no amadurecimento e detalhamento dos métodos de engenharia de ontologias (ABDELGHANY; DARWISH; HEFNI, 2019), observamos a necessidade de mais informações sobre os processos e atividades do método SABiO. Por isso, estendemos o método definindo mais detalhes sobre o seu processo e, adicionalmente, seguindo princípios ágeis, a fim de induzir um processo de construção orientado, mais colaborativo e flexível.

Assim, SABiOS — *Systematic Approach for Building Ontologies with Scrum* — é proposto com o intuito de guiar mais detalhadamente a construção de ontologias (de qualquer domínio) baseadas no método SABiO, adotando princípios ágeis. Desta forma, o ciclo de vida de construção de ontologias é iterativo e incremental, com ciclos de curto período de tempo e revisão ao final de cada ciclo. Os processos e atividades são baseados no método SABiO e ajustados com base em nossa experiência na construção de ontologias de núcleo, domínio e rede de ontologias, de modo que alguns processos e atividades foram movidos e outras atividades adicionadas e refinadas, bem como as relações entre elas.

SABiOS adota uma abordagem de conceituação direcionada por questões de competência; por usuário, ao requerer o desenvolvimento colaborativo de especialistas de domínio e engenheiros de ontologia; e por dados, ao requerer o desenvolvimento baseado em fontes de dados consolidadas, assim como SABiO.

O método cobre tanto a construção da ontologia de referência quanto a ontologia operacional, destacando a importância do consenso e argumentação sobre os conceitos a serem representados na ontologia, a construção de ontologias a partir do zero e a partir da reutilização de recursos ontológicos existentes e a aplicação de análise ontológica na construção da ontologia.

A.1 Ciclo de Vida

O ciclo de vida proposto pelo método SABiOS é composto por dois tipos de ciclos complementares: **ciclo de vida da ontologia**, apresenta o aspecto dinâmico do processo de construção de ontologia, expresso por meio de fases e iterações; e **ciclo de vida da fase**, apresenta o aspecto dinâmico das fases que compõem o ciclo de vida da ontologia, expresso por meio de processos, atividades e iterações.

O **ciclo de vida da ontologia** (apresentado Figura 33) é formado por cinco fases que indicam a ênfase dos subprocessos em cada instante do ciclo de vida e o caráter evolutivo da construção de ontologias. A cada ciclo de vida da ontologia, ou seja, a cada passagem pelas cinco fases do ciclo, uma nova versão completa da ontologia é construída, cujo período de realização não é previamente definido. O primeiro ciclo de ontologia é definido como o *ciclo de desenvolvimento* e os subsequentes, como *ciclos de evolução*. O ciclo de desenvolvimento é desencadeado pela necessidade de construir uma ontologia e os ciclos de evolução são desencadeados por alterações ou melhorias na ontologia já construída.

A fase ***Conception*** enfatiza o propósito da ontologia de acordo com as partes interessadas. A fase ***Pre Reference*** enfatiza as questões de modelagem para a elaboração da ontologia de referência. A fase ***Reference*** enfatiza a elaboração da ontologia de referência, buscando enriquecer o conhecimento identificado na fase de concepção. Essa fase é considerada a mais crítica, pois investiga os requisitos a partir das partes interessadas e os representa como um modelo conceitual. Esta fase normalmente mantém o maior número de iterações. A fase ***Pre Operational*** enfatiza as questões de codificação para a elaboração da ontologia operacional. A fase ***Operational*** enfatiza a construção da ontologia operacional, implementando a ontologia em uma linguagem operacional.

As fases são definidas de forma progressiva de modo que cada fase use o resultado da fase anterior para fornecer enriquecimento gradual no ciclo de vida, sendo Documento de Especificação para a fase *Conception*, Ontologia de Referência para a fase *Pre Reference*

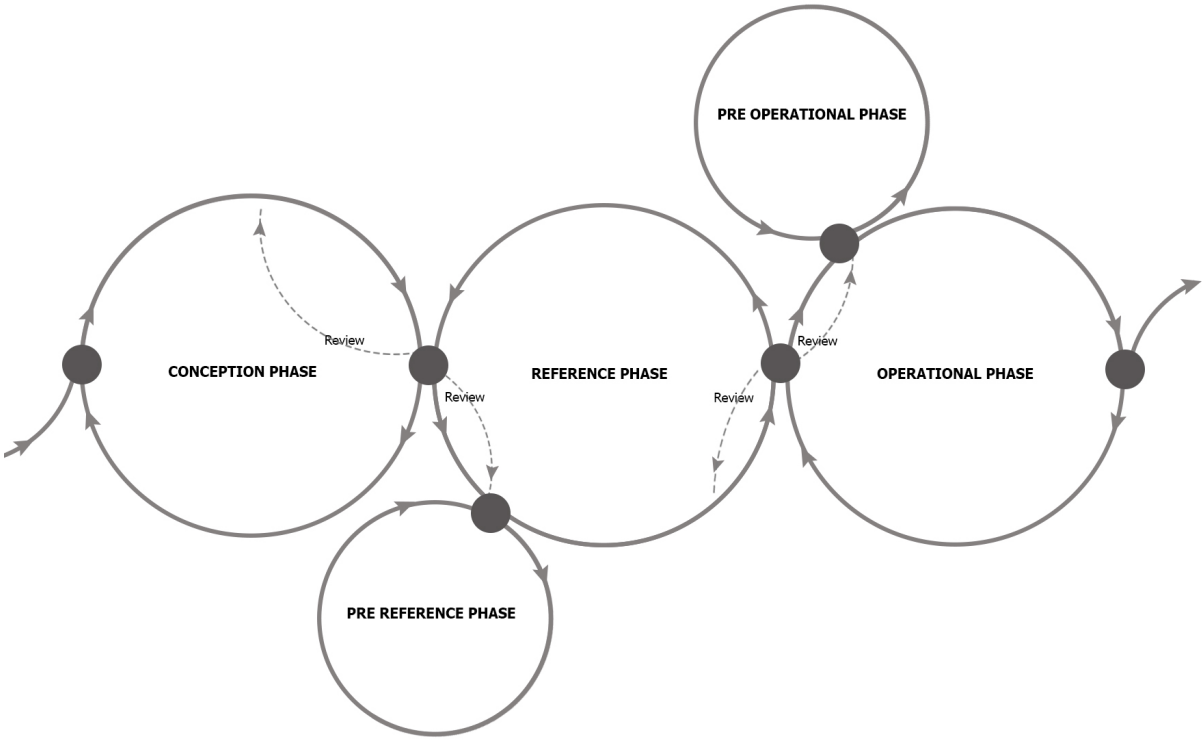


Figura 33 – Ciclo de vida da otologia do método SABiOS

e *Reference* e Ontologia Operacional para a fase *Pre Operational* e *Operational*. Portanto, a fase *Reference* inicia a partir do resultado da fase *Conception* e a fase *Operational* a partir da fase de *Reference*.

De maneira inversa, o resultado de cada fase pode ser revisado (representado pela linha tracejada), de modo que o resultado da fase *Operational* pode originar uma revisão na fase *Pre Operational* e *Reference*, cujo resultado pode originar uma revisão na fase *Pre Reference* e *Conception*. Além disso, o ciclo de vida foi projetado para acomodar a construção de ontologias que evoluem ao longo do tempo, ou seja, os ciclos de evolução geram novos incrementos da ontologia.

O **ciclo de vida da fase** é formado por atividades que compõem os **Processos Principais**, que definem procedimentos principais para a construção da ontologia e os **Processos de Suporte**, que definem procedimentos auxiliares para os processos principais. Para cada fase do ciclo de vida da ontologia ocorrem diversos ciclos de vida da fase, executados a partir de princípios ágeis.

Inspirado no *framework* Scrum (SCHWABER; SUTHERLAND, 2020), apresentamos a seguir a aplicação desse *framework* no ciclo de vida da fase de SABiOS, ilustrado na Figura 34.

Cada ciclo de vida que ocorre dentro da fase é denominado **Sprint**, que passa pelos processos e atividades definidos para a respectiva fase, num período de 2 a 4 semanas. Cada fase define os tipos de processos e atividades de seu sprint, bem como define sprints

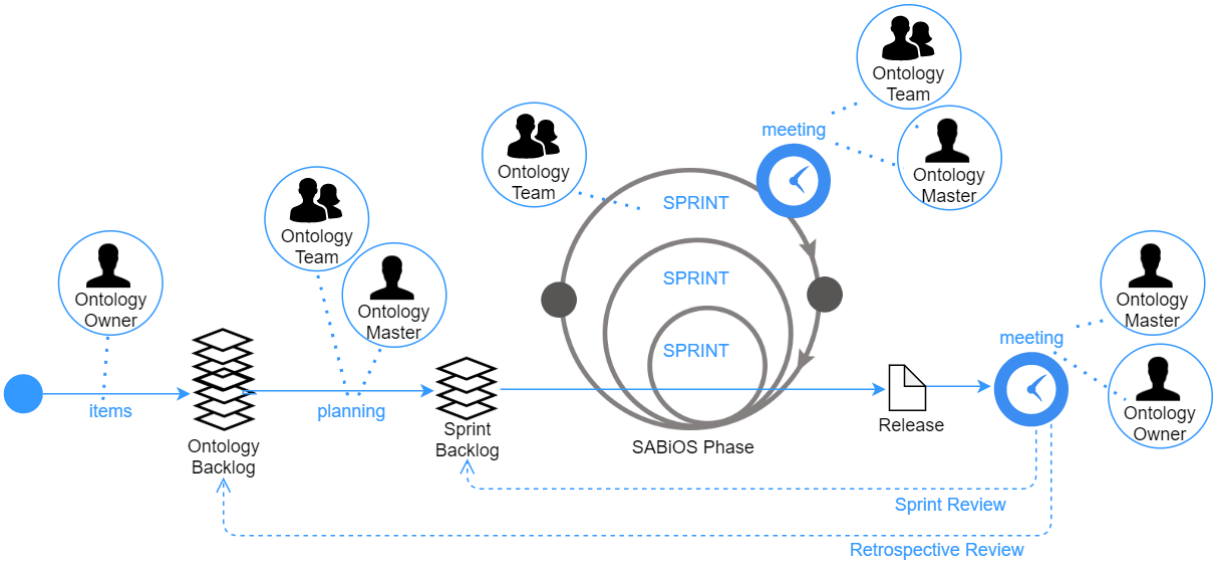


Figura 34 – Ciclo de vida da fase do método SABiOS aplicando o Scrum.

com atividades diferentes dentro da mesma fase (representadas pelos circulos internos do ciclo da fase). Por exemplo, um sprint da Reference Phase pode ser formado pelas atividades [CAPT-CONC] Identify Concepts, [CAPT-AXIM] Identify Axioms, [CAPT-MODE] Model Ontology, [DOCM-MODE] Document Reference Ontology, [CONF-MODE] Control Reference Ontology, [EVAL-MODE] Evaluate Reference Ontology e [PUBL-REFE] Publish Reference Ontology (detalhadas na Figura 35).

O ciclo de vida da fase inicia com a lista de **items** registrados no **Ontology Backlog**, guiado por questões de competência e compartilhado por todas as fases. Ao longo do desenvolvimento da ontologia, o Ontology Backlog sofrerá modificações e priorização. Por exemplo, o Ontology Backlog de uma ontologia de orientação a objetos pode conter items como: (i) Quais os principais elementos de um código-fonte OO? (ii) Quais classes estão presentes em um código OO? (iii) Quais elementos compõem uma classe? (iv) Quais métodos estão presentes em um código OO? e (v) Quais variáveis estão presentes em um código OO?, identificados a partir do Ontology Owner.

No inicio de cada sprint, são priorizados items do Ontology Backlog e definidas as atividades que serão realizadas na sprint relacionada ao ciclo de vida da ontologia, compondo o **Sprint Backlog**. Reuniões diárias são realizadas para acompanhar o trabalho realizado, solucionar obstáculos e alinhar prioridades do dia. Por exemplo, um dado Sprint Backlog pode conter os items (i) e (ii) priorizados para Reference Phase, considerando que estes items já foram tratados pela Conception Phase. Neste momento, o Ontology Master e Ontology Team devem priorizar os items respeitando o ciclo de vida da ontologia.

Cada fase pode executar vários sprints (iterações) para chegar ao artefato final. Ao final de cada iteração, o release do artefato correspondente à fase é esperado, sendo constantemente avaliado em uma **Sprint Review** para garantir *feedback* mais cedo e

permitir melhorias na próxima iteração, evitando a propagação de erros. Por fim, para identificar os pontos positivos e negativos do sprint e melhorá-los nos próximos sprints, é realizada uma **Sprint Retrospective** para a atualização do Ontology Backlog. Por exemplo, ao final do sprint que tratou os items (i) e (ii), pode-se identificar ajustes aos items ou adicionar items complementares ao Ontology Backlog, que será tratado em novos sprints.

Assim, o método SABiOS suporta o envolvimento ativo e colaborativo de diferentes papéis desempenhados por um indivíduo ou grupo de indivíduos durante o processo de construção da ontologia, classificados em:

- **Ontology Owner:** pessoa interessada na construção da ontologia para um dado propósito. Responsável por definir que tipo de ontologia deverá ser construída e quais os requisitos que serão respondidos. Participa do processo de construção da ontologia de modo a priorizar e definir mudanças. Na exemplificação, Ontology Owner trabalha com os itens do Ontology Backlog.
- **Ontology Team:** time de pessoas efetivamente responsável pela construção da ontologia e por selecionar os itens priorizados que serão executados. Na exemplificação, os itens são priorizados a partir do Ontology Backlog e são executados durante a Sprint. Esse time deve ser multidisciplinar e preferencialmente pequeno, composto pelas seguintes especialidades:
  - **Domain Expert:** pessoa que mantém o conhecimento/expertise do domínio;
  - **Ontology Engineer:** pessoa com conhecimento em engenharia de ontologias relacionado à construção da ontologia de referência e a implementação da ontologia operacional;
- **Ontology Master:** pessoa conhecedora do método SABiOS que atua como facilitadora para a construção da ontologia junto ao Ontology Team, participando das reuniões de acompanhamento e de revisão.

Como já apresentado, o método SABiOS é definido por cinco fases, que são descritas por seus processos principais e de suporte. A Figura 35 ilustra tais processos principais (linhas coloridas), processos de suporte (linha cinza) e suas respectivas atividades (pontos das linhas).

Dentre os processos principais, o processo **Requirement** (representado pela linha amarela) reconhece as necessidades que a ontologia deve representar, **Capture** (representado pela linha azul e verde) identifica a conceituação para atender aos requisitos, **Design** (representado pela linha rosa) especifica características tecnológicas para a ontologia e **Implementation** (representado pela linha roxa) codifica a ontologia em linguagem operacional.

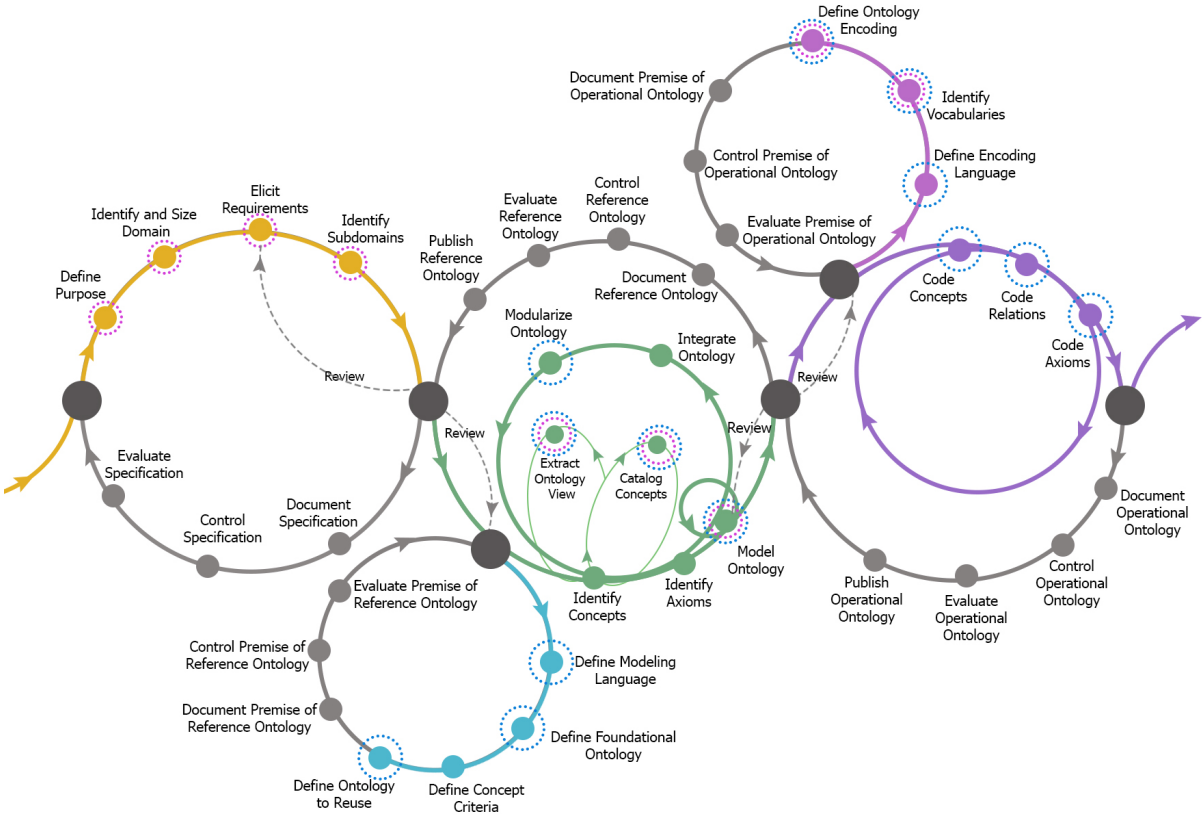


Figura 35 – Visão geral do método SABiOS

Dentre os processos de suporte, o processo de **Knowledge Acquisition** extrai o conhecimento de diferentes fontes de conhecimento, **Documentation** registra os resultados das atividades, **Configuration Management** controla a versão e as alterações nos artefatos produzidos, **Evaluation**, avalia os resultados produzidos, **Reuse** reutiliza recursos ontológicos e não ontológicos existentes para construir a ontologia e **Publication** torna disponível a ontologia construída. As atividades dos processos de suporte de *Knowledge Acquisition* (representado pelo círculo pontilhado na cor rosa) e *Reuse* (representado pelo círculo pontilhado na cor azul) são realizadas simultaneamente às atividades dos processos principais.

O objetivo do método é orientar os papéis envolvidos na construção da ontologia por meio de descrições detalhadas, a fim de possibilitar sua reprodução.

A.2 Conception Phase

A Fase de Concepção (Conception Phase) enfatiza o propósito da ontologia, objetivando tanto identificar o propósito da ontologia dentro de um domínio definido quanto descobrir as necessidades que a ontologia deve satisfazer quanto ao seu conteúdo e suas características.

Esta fase está baseada no processo principal de **requirements**, que é apoiado

nos processos de suporte de **knowledge acquisition**, a fim de extrair o conhecimento do *ontology owner* por meio de técnicas de *brainstorming*, entrevista, questionário e mapeamento conceitual; de **documentation**, a fim de registrar esse conhecimento como uma especificação de ontologia; de **configuration management**, a fim de controlar as alterações, versões e entregas das informações registradas na especificação de ontologia; e de **evaluation**, a fim de validar com o *ontology owner* se as informações extraídas e registradas na especificação de ontologia atendem às suas expectativas. As atividades desses respectivos processos são apresentadas nas subseções seguintes.

A.2.1 [REQI-PURP] Define Purpose

Atividade do processo principal de requirement que define e descreve o objetivo principal da ontologia a fim de responder: **como** a ontologia pretende representar o domínio, **para que** a ontologia será útil e **por que** a ontologia deve ser construída.

O processo de suporte de knowledge acquisition objetiva extrair as expectativas do *ontology owner* a fim de responder estas questões, seguindo a atividade:

KNOW-PURP Capture Ontology Purpose : atividade em que o *knowledge engineer* extrai do *ontology owner* o conhecimento relacionado às tres questões (o que, para que e por que) da ontologia. Este conhecimento normalmente é adquirido por meio de técnicas de *brainstorming* e entrevista.

Esta atividade define: **Who**, *knowledge engineer* como responsável e *ontology owner* como participante; **What-Input**, necessidade de construir uma ontologia; **What-Output**, resposta às questões o que, para que e por que.

No *running example* de Orientação a Objetos, o propósito da ontologia define: **como** representar os conceitos da orientação a objetos presentes em um código-fonte; **para que** os conceitos de orientação a objetos sejam interpretados e identificados por diferentes linguagens de programação de forma única; **porque** cada linguagem de programação define sua própria sintaxe e semântica, não havendo normatização dos conceitos de código-fonte e nem iniciativas maduras que atendam esses requisitos, resultando em códigos-fonte com heterogeneidade e dificuldade de interoperabilidade.

A.2.2 [REQI-DOMN] Identify and Size Domain

Atividade do processo principal de requirement que identifica o domínio que a ontologia pretende representar segundo o seu propósito e dimensiona seus limites. Deve-se identificar o conhecimento que a ontologia deve abranger, bem como o nível de detalhes que a ontologia deve cobrir do domínio e quais os detalhes ou partes não devem ser cobertos. Embora o desenvolvimento de uma ontologia seja principalmente motivado pelos cenários

relacionados à aplicação dessa ontologia (SUÁREZ-FIGUEROA; GÓMEZ-PÉREZ, 2012), o objetivo desta atividade não é listar os cenários de aplicação mas, sim, extrair o domínio de conhecimento inserido nesses cenários.

O processo de suporte de knowledge acquisition objetiva extrair o conhecimento do *ontology owner* sobre o domínio a ser representado na ontologia, seguindo as atividades:

KNOW-DOMU Understanding Ontology Domain : atividade em que o *Knowledge Engineer* objetiva compreender o domínio relacionado ao propósito da ontologia, considerando o conhecimento prévio do *ontology owner*, *domain expert* e fontes da literatura sobre o domínio.

KNOW-DOMS Size Ontology Domain : atividade em que o *knowledge engineer* objetiva dimensionar os limites do domínio a ser representado na ontologia. Essa atividade é um desafio, pois os domínios no mundo real estão interconectados e não é possível definir um limite claro entre eles. Portanto, deve-se considerar o conhecimento prévio do *ontology owner* sobre as dimensões limiares desse domínio.

Os limites devem ser definidos para a *dimensão horizontal*, a fim de limitar as áreas externas que fazem parte do domínio e *dimensão vertical*, a fim de limitar o nível de detalhes que deve ser aprofundado no domínio.

Esta atividade define: **Who**, *knowledge engineer* como responsável e *ontology owner* como participante; **What-Input**, propósito da ontologia definido na atividade [REQI-PURP] Define Purpose; **What-Output**, domínio a ser representado na ontologia e seus limites.

No *running example*, o domínio é definido como programação orientada a objetos (OO), um método de implementação de software no qual os programas são organizados como coleções cooperativas de objetos, cujos objetos representam uma instância de alguma classe e cujas classes são membros de uma hierarquia de classes ligadas por relacionamentos de herança. Uma classe serve como um modelo a partir do qual os objetos podem ser criados, contendo atributos e métodos. Para que os atributos e métodos de uma classe sejam usados na definição de uma nova classe, a herança é aplicada como um meio de criar abstrações.

Na dimensão horizontal, o domínio é definido apenas para o desenvolvimento de software orietado a objetos; para a dimensão vertical, o domínio é limitado na representação do código-fonte em tempo de compilação e não cobre a perspectiva de execução, tal como objetos e mensagens trocadas entre os objetos.

A.2.3 [REQI-ELIC] Elicit Requirements

Atividade do processo principal de requirement que objetiva elicitar o conhecimento a ser representado na ontologia, ou seja, identificar as expectativas do *ontology owner* segundo o propósito e domínio da ontologia.

Para os *requisitos funcionais*, SABiO sugere o uso de questões de competência informais, que são questões não expressadas em linguagem formal (GRÜNINGER; FOX; GRUNINGER, 1995). Tais questões são representadas por frases interrogativas e devem ser definidas de maneira estratificada, com questões de nível superior exigindo a solução de questões de nível inferior (GRÜNINGER; FOX; GRUNINGER, 1995).

As questões servem como restrições sobre o que a ontologia pode ser, ou seja, restringe o que é ou não relevante para a ontologia. Além disso, estas questões são usadas futuramente para avaliar o compromisso ontológico da ontologia a fim de verificar se a ontologia atende aos requisitos.

Para os *requisitos não funcionais*, SABiO sugere requisitos de qualidade da ontologia (performance de raciocínio, disponibilidade, usabilidade, manutenibilidade); de projeto (definição de linguagem de implementação, consenso e conceitos, aderência a modelo de processo); e de uso pretendido (seleção de fonte de dados, agrupamento de ontologia, etc.) (FALBO, 2014), ou seja, requisitos não relacionados ao conteúdo da ontologia.

O processo de suporte de knowledge acquisition objetiva extrair o conhecimento do *ontology owner* sobre os requisitos da ontologia, seguindo as atividades:

KNOW-REQI Capture Ontology Requirements : atividade em que o *knowledge engineer* objetiva descobrir as necessidades ou conhecimentos que devem ser representados na ontologia. Deve-se considerar que dificilmente as necessidades serão esgotadas e, por isso, os requisitos devem ser descobertos, analisados, negociados e atualizados continuamente, ou seja, deve-se retornar a esta atividade sempre que necessário. Este conhecimento normalmente é adquirido por meio de técnicas de *brainstorming*, entrevista e mapeamento conceitual.

KNOW-REQN Negotiate Ontology Requirements : atividade em que o *knowledge engineer* objetiva negociar com o *ontology owner* quais requisitos devem ser considerados para a construção da ontologia. Deve-se considerar que nem toda necessidade inicial do *ontology owner* é relevante para o domínio da ontologia.

Esta atividade define: **Who**, *knowledge engineer* como responsável e *ontology owner* como participante; **What-Input**, domínio da ontologia identificado na atividade [REQI-DOMN] Identify and Size Domain; **What-Output**, requisitos funcionais e não funcionais que a ontologia deve responder.

No *running example*, os seguintes requisitos funcionais são definidos: Quais os principais elementos de um código-fonte OO?; Quais classes estão presentes em um código OO?; Quais elementos compõem uma classe?; Qual a herança presente em uma determinada classe? ; Quais métodos estão presentes em um código OO?; Quais elementos compõem um método?; Quais os métodos de uma classe?; Quais variáveis estão presentes em um código OO?; Quais variáveis compõem uma classe?; Quais variáveis compõem um método?. Para os requisitos não funcionais são definidos: ser modular para facilitar a reutilização por outras ontologias; ser incorporada a uma rede de ontologias para facilitar a reutilização de outras ontologias e, consequentemente, expandir suas prórpias possibilidades de reuso e associação; ser definida a partir de fontes de conhecimento reconhecidas na literatura; ser definida a partir do conhecimento consensual do domínio; e ser fundamentada por uma ontologia de fundamentação a fim de reutilizar os compromissos ontológicos e axiomas já consensualmente estabelecidos.

A.2.4 [REQI-SUBD] Identify Subdomains

Atividade do processo principal de requirement que objetiva identificar subdomínios relacionados às questões de competência definidas para o domínio, a fim de facilitar a identificação das partes do domínio presentes no propósito da ontologia.

Tais subdomínios permitem a construção direcionada ou distribuída da ontologia, ou seja, subdomínios podem ser tratados de forma paralela nas fases posteriores. Para identificar os subdomínios, sugere-se o uso de categorias pré estabelecidas no domínio ou a criação de categorias que refletem os termos de alta frequência presentes nas questões de competência (SUÁREZ-FIGUEROA; GÓMEZ-PÉREZ, 2012).

Esta atividade define: **Who**, *knowledge engineer* como responsável; **What-Input**, requisitos funcionais identificados na atividades [REQI-ELIC] Elicit Requirements; **What-Output**, subdomínios do domínio a ser representado na ontologia.

No *running example*, são definidos os subdmomínios *Core* para representar os conceitos gerais de orientação a objetos, *Class* para representar os conceitos relacionados a classe e *Class Member* para representar os conceitos relacionados aos membros das classes.

A.2.5 [DOCM-SPEC] Document Specification

Atividade do processo de suporte documentation que objetiva documentar a especificação da ontologia por meio do Documento de Especificação de Ontologia, que deve incluir os requisitos levantados para a ontologia (tanto de referência como operacional), bem como seguir as considerações:

- Propósito da Ontologia: texto de linguagem natural com o padrão: *O propósito da*

ontologia é representar <como> <para que> <porque>;

- Domínio da Ontologia: texto descritivo de linguagem natural descrevendo o domínio a ser representado na ontologia;
- Dimensão da Ontologia: texto descritivo de linguagem natural destacando claramente os limites horizontais e verticais da ontologia;
- Requisitos Funcionais: texto em forma de frase interrogativa com identificador único para cada requisito;
- Requisitos Não Funcionais: texto descritivo de linguagem natural com identificador único.
- Subdomínio: nome do subdomínio identificado no domínio a partir das questões de competência.

Esta atividade define: **Who**, *knowledge Eegineer* como responsável; **What-Input**, atividades realizadas no processo principal de requirement; **What-Output**, documento de especificação de ontologia.

No *running example*, um fragmento do Documento de Especificação da Ontologia para o domínio de orientação a objetos é apresentado na Tabela 20.

Tabela 20 – Documento de Especificação de Ontologia

Documento:	Especificação de Ontologia	**Versão:**	v.01
Ontologia:	OOC-O Object Oriented Code	**Data:**	13/09/2018

Propósito da Ontologia:
O propósito da ontologia é representar os conceitos da orientação a objetos presentes em um código-fonte para que os conceitos de orientação a objetos sejam interpretados e identificados por diferentes linguagens de programação de forma única porque cada linguagem de programação define sua própria sintaxe e semântica, não havendo normatização dos conceitos de código-fonte e nem iniciativas maduras que atendam esses requisitos, resultando em códigos-fonte com heterogeneidade e dificuldade de interoperabilidade.

Domínio da Ontologia:
Programação orientada a objetos (OO) é definida como um método de implementação de software no qual os programas são organizados como coleções cooperativas de objetos, cujos objetos representam uma instância de alguma classe e cujas classes são membros de uma hierarquia de classes ligadas por relacionamentos de herança. Uma classe serve como um modelo a partir do qual os objetos podem ser criados, contendo atributos e métodos. Para que os atributos e métodos de uma classe sejam usados na definição de uma nova classe, a herança é aplicada como um meio de criar abstrações.

Dimensão da Ontologia:
Na dimensão horizontal, o domínio é definido apenas para o desenvolvimento de software orietado a objetos; para a dimensão vertical, o domínio é limitado na representação do código-fonte em tempo de compilação e não cobre a perspectiva de execução, tal como objetos e mensagens trocadas entre os objetos.

Requisitos Funcionais:	
RF	**Descrição**
Subdomínio: Core	
RF01	Quais os principais elementos de um código-fonte OO?
Subdomínio: Class	
RF02	Quais classes estão presentes em um código OO?
RF03	Quais elementos compõem uma classe?
RF04	Qual a herença presente em uma determinada classe?
Subdomínio: Class Member	
RF05	Quais métodos estão presentes em um código OO?
RF06	Quais elementos compõem um método?
RF07	Quais os métodos de uma classe?
RF08	Quais variáveis estão presentes em um código OO?
RF09	Quais variáveis compõem uma classe?
RF10	Quais variáveis compõem um método?

Requisitos Não Funcionais:	
RNF	**Descrição**
RNF01	Ser modular para facilitar a reutilização por outras ontologias;.
RNF02	Ser incorporada a uma rede de ontologias para facilitar a reutilização de outras ontologias e, consequentemente, expandir suas prórpias possibilidades de reuso e associação.
RNF03	Ser definida a partir de fontes de conhecimento reconhecidas na literatura.
RNF04	Ser definida a partir do conhecimento consensual do domínio.
RNF05	Ser fundamentada por uma ontologia de fundamentação a fim de reutilizar os compromissos ontológicos e axiomas já consensualmente estabelecidos.

A.2.6 [CONF-SPEC] Control Specification

Atividade do processo de suporte configuration management que objetiva controlar as alterações, versões e entregas das informações registradas no Documento de Especificação de Ontologia, descrito na atividade [DOCM-SPEC] Document Specification.

Para isso, sugere-se o uso do cabeçalho do documento para registrar a versão e sua respectiva data, armazenando o histórico de todas as versões para eventuais comparações ou restaurações de versões anteriores.

Esta atividade define: **Who**, *knowledge engineer* como responsável; **What-Input**, documento de especificação de ontologia elaborado na atividade [DOCM-SPEC] Document Specification; **What-Output**, controle do documento de especificação de ontologia.

No *running example*, o cabeçalho do Documento de Especificação da Ontologia é adotado para controlar as versões do documento de especificação da ontologia, como apresentado na Tabela 20.

A.2.7 [EVAL-SPEC] Evaluate Specification

Atividade do processo de suporte evaluation que objetiva avaliar se as informações extraídas e registradas no Documento de Especificação de Ontologia atendem às expectativas do *ontology owner*. Para isso, deve-se validar as informações registradas junto ao *ontology owner* e retornar ao início da fase sempre que necessário.

Esta atividade define: **Who**, *ontology owner* como responsável; **What-Input**, documento de especificação de ontologia elaborado na atividade [DOCM-SPEC] Document Specification; **What-Output**, avaliação do documento de especificação de ontologia.

No *running example*, reuniões são realizadas com *ontology owner* a fim de validar o documento de especificação de ontologia, apresentado na Tabela 20.

A.3 Pre-Reference Phase

A Fase Pré-Referência (Pre-Reference Phase) objetiva definir questões que irão guiar a elaboração da ontologia de referência, ou seja, decisões dependentes da modelagem conceitual.

Esta fase está baseada no processo principal de **capture**, que é apoiado nos processos de suporte de **knowledge acquisition**, a fim de extrair o conhecimento dos recursos; de **reuse**, a fim de reutilizar recursos ontológicos e não ontológicos reconhecidos; de **documentation**, a fim de registrar esse conhecimento como parte do Documento da Ontologia de Referência; de **configuration management**, a fim de controlar as alterações, versões e entregas das informações documentadas; e de **evaluation**, a fim de verificar se as questões definidas para a ontologia de referência atendem ao seu propósito. As atividades desses respectivos processos são apresentadas nas subseções seguintes.

A.3.1 [CAPT-LANG] Define Modeling Language

Atividade do processo principal de capture que objetiva definir a linguagem de representação a ser aplicada na construção da ontologia de referência, ou seja, no modelo conceitual. A escolha da linguagem influencia diretamente na integração e reuso da ontologia em construção, uma vez que ontologias desenvolvidas em diferentes linguagens podem requerer grande esforco de adaptação.

O processo de suporte de reuse objetiva aproveitar a linguagem de modelagem já estabelecida para representação de ontologias de referência, seguindo a atividade:

REUS-LANG Adopt Modeling Language : atividade que objetiva adotar linguagens de modelagem existentes para a construção da ontologia de referência. Há várias linguagens para modelar, comunicar e negociar os conceitos da ontologia, tal como, UML e

OntoUML (GUIZZARDI; GRAÇAS; GUIZZARDI, 2011).

Esta atividade define: **Who**, *knowledge engineer* como responsável; **What-Input**, documento de especificação da ontologia elaborado na atividade [DOCM-SPEC] Document Specification; **What-Output**, linguagem de modelagem a ser adotada na ontologia de referência.

No *running example*, a linguagem OntoUML é definida para a modelagem da ontologia de referência de orientação a objetos.

A.3.2 [CAPT-FOUN] Define Foundational Ontology

Atividade do processo principal de capture que objetiva definir a ontologia de fundamentação a ser adotada para guiar o processo de modelagem da ontologia em construção e das visões das ontologias reusadas. Para isso, deve-se considerar a popularidade, usabilidade e aderência da ontologia de fundamentação para o domínio definido, bem como o *know-how* do engenheiro de conhecimento.

SABiO destaca a importância de conceitos e relações serem previamente analisados à luz de uma ontologia de fundamentação. O reuso de ontologias de fundamentação pode ser feito por especialização ou analogia, quando os conceitos e relações não são explicitamente extendidas mas, sim, implicitamente usadas para derivar a ontologia em construção (FALBO, 2014).

O processo de suporte de reuse objetiva aproveitar as conceitualizações já estabelecidas em ontologias de fundamentação para apoiar a ontologia em construção, seguindo as atividades:

REUS-FOUN Adopt Foundational Ontology : atividade que objetiva adotar ontologia de fundamentação existentes para a construção da ontologia de referência, tal como a *Unified Foundational Ontology* (UFO) (GUIZZARDI; WAGNER, 2004).

REUS-PATT Adopt Ontology Pattern : atividade que objetiva definir os padrões de ontologia a serem adotados no processo de construção da ontologia. Tais padrões são classificados como (FALBO et al., 2013): *Padrão Conceitual*, fragmentos reusáveis de ontologias de fundamentação ou de referência de domínio; *Padrão Arquitetural*, padrões que descrevem como organizar a ontologia em termos de subontologias ou módulos; *Padrão de Design*, padrões relacionados a raciocínio e construtos lógicos de implementação; e *Padrão de Programação*, padrões relacionados à linguagem de implementação de ontologia.

Esta atividade define: **Who**, *knowledge engineer* como responsável; **What-Input**, domínio da ontologia e linguagem de modelagem identificada nas atividades [REQI-DOMN]

Identify and Size Domain e [CAPT-LANG] Define Modeling Language; **What-Output**, ontologia de fundamentação e padrões conceituais a serem aplicados na ontologia em construção.

No *running example*, a ontologia UFO e seus padrões são definidos para a modelagem da ontologia de referência.

A.3.3 [CAPT-CRIT] Define Concept Criteria

Atividade do processo principal de capture que objetiva definir os critérios que serão adotados para identificar se um conceito é aderente ao domínio da ontologia em construção. Para isso, uma lista de critérios objetivos, com informações quantitativas, ou subjetivos, com informações qualitativas, é definida. Estes critérios podem ser definidos para análise individual de cada fonte de conhecimento ou análise conjunta sobre todas as fontes catalogadas.

Dada a profundidade do conhecimento do domínio neste momento, a definição de critérios pode ser um desafio. Assim, à medida que o conhecimento é enriquecido e as decisões sobre o domínio são tomadas, deve-se retornar a esta atividade para atualizar estes critérios.

Esta atividade define: **Who**, *knowledge engineer* como responsável e *domain expert* como participante; Para otimizar o processo, esta atividade pode ser conduzida unilateralmente pelo *domain expert*; **What-Input**, documento de especificação de ontologia elaborado na atividade [DOCM-SPEC] Document Specification; **What-Output**, critérios definidos para os conceitos do domínio.

No *running example*, os seguintes critérios definem se um conceito é aderente ao domínio de orientação a objetos da ontologia em construção: conceito referente à programação em tempo de compilação; conceito presente em mais de 50% das linguagens de programação analisadas; e conceito relacionado aos principais elementos da orientação a objetos: classe, método e atributo.

A.3.4 [CAPT-REUS] Define Ontology to Reuse

Atividade do processo principal de capture que objetiva definir as ontologias de referência a serem reusadas, ou seja, ontologias ou fragmentos de ontologias que correspondem ao propósito da ontologia em construção.

O processo de suporte de reuse objetiva aproveitar as conceitualizações já estabelecidas em ontologias de núcleo e de domínio para apoiar a ontologia em construção, seguindo a atividade:

REUS-ONTO Adopt Reused Ontology : atividade que objetiva adotar ontologias de núcleo e

de domínio relacionadas ao propósito da ontologia em construção, independente do seu nível de fundamentação ontológico. O reuso de ontologias é um desafio, pois envolve heterogeneidade de formalismo, diversidade de linguagens, problemas lexicais e semânticos, suposições implícitas de representação, perda de conhecimento consensual, falta de documentação e outros (FERNÁNDEZ-LÓPEZ; GÓMEZ-PÉREZ, 2002).

O reuso de uma ontologia de núcleo irá posicionar a ontologia em construção dentro de um domínio mais abrangente, enquanto o de uma ontologia de domínio irá disponibilizar conceitos já formalizados para o seu domínio. Para isso, deve-se buscar por ontologias candidatas em mecanismos de busca, repositórios e ontologias conhecidas.

Para selecionar ontologias de núcleo, deve-se analisar se a ontologia candidata representa conceitos apropriados para ancorar a ontologia em construção. Para selecionar ontologias de domínio, deve-se analisar a cobertura da ontologia candidata quanto às questões de competência da ontologia em construção.

Além destas análises, deve-se considerar (i) se alguma análise ontológica foi aplicada sobre as ontologias candidatas, (ii) se a ontologia de fundamentação adotada na ontologia candidata é aderente à da ontologia em construção e (iii) se a perspectiva do domínio representada na ontologia candidata corresponde à da ontologia em construção. Ressalta-se que a ontologia candidata pode se referir a uma ontologia individual ou a uma rede de ontologias.

Esta atividade define: **Who**, *knowledge engineer* como responsável; **What-Input**, documento de especificação da ontologia elaborado na atividade [DOCM-SPEC] Document Specification; **What-Output**, ontologias a serem reusadas na ontologia em construção;

No *running example*, ontologias existentes que tratam de código-fonte e de orientação a objetos são buscadas em mecanismos de busca e em ontologias conhecidas. Cada ontologia encontrada é analisada de acordo com o seu propósito e aderência ao domínio da ontologia em construção, a saber:

- SWO: ontologia de software que representa os artefatos de software de natureza complexa, incluindo sistemas, programas e códigos de software. Identifica-se que um código OO pode ser representado na ontologia SWO como uma especificação de código-fonte que compõe um software;
- SPO: ontologia de processo de software que descreve processos e atividades utilizando ativos de processo, tais como, Artefato. Identifica-se que um código OO pode ser representado na ontologia SPO como um artefato de software;
- SCO: ontologia de código-fonte que descreve o conceitos gerais de um código-fonte,

independente de linguagem de programação. Identifica-se que um código OO pode ser representado na ontologia SCO como uma especialização;

- JAVAOWL: ontologia da linguagem de programação Java que descreve os principais elementos da linguagem capturados a partir da classe JavaElement. Identifica-se que a ontologia JAVAOWL representa singularmente a estrutura da linguagem Java e não é aderente ao propósito da ontologia em construção;
- O3: ontologia que representa os conceitos do paradigma orientado a objetos e de programação. Identifica-se que a ontologia O3 representa uma perspectiva diferente do domínio da ontologia em construção, tal como, Classe pode ser Agente ou Servidor e é composta de serviços, podendo ser generalizada com um generalizonSet.

Das ontologias analisadas, a ontologia SCO foi selecionada a fim de especializar o domínio de código-fonte para o domínio de orientação a objetos, assim como posicionar o domínio de código-fonte dentro do domínio de processo de software, dado que SCO especializa SPO e SWO. As demais ontologias são desconsideradas, pois não apresentam a mesma perspectiva do domínio de código-fonte da ontologia em construção.

A.3.5 [DOCM-REFE] Document Premise of Reference Ontology

Atividade do processo de suporte documentation que objetiva documentar as premissas da ontologia de referência por meio do Documento de Ontologia de Referência, que deve incluir as questões definidas na fase pré-referência, bem como seguir as considerações:

- Linguagem de Modelagem: descrição da linguagem de modelagem;
- Ontologia de Fundamentação: descrição da ontologia de fundamentação e seus padrões ontológicos;
- Ontologia de Reuso: apresentação da ontologia de núcleo e/ou de domínio a ser reusada na ontologia em construção.

Esta atividade define: **Who**, *knowledge engineer* como responsável; **What-Input**, atividades realizadas no processo principal de capture, da Pre-Reference Phase; **What-Output**, Documento de Ontologia de Referência com suas respectivas premissas.

No *running example*, o Documento de Ontologia de Referência é elaborado com as premissas estabelecidas para a ontologia de referência de orientação a objetos, conforme apresentado na Tabela 21.

Tabela 21 – Documento de Ontologia de Referência

Documento:	Ontologia de Referência	**Versão:**	v.01
Ontologia:	OOC-O Object-Oriented Code	**Data:**	11/01/2019

Linguagem de Modelagem:
OntoUML

Ontologia de Fundamentação:
UFO - Ontologia caracterizada em lógica modal e psicologia cognitiva que pretende fornecer maior nível semântico sobre o mundo em um modelo conceitual de dado domínio de conhecimento, sistematiza questões como noções de tipos e suas instâncias; objetos, e suas propriedades intrínsecas; a relação entre identidade e classificação; distinções entre tipos e suas relações; relações parte-todo entre outros.

Critérios:

CR01	Conceito referente à programação em tempo de compilação.
CR02	Conceito presente em mais de 50% das linguagens de programação analisadas.
CR03	Conceito relacionado aos principais elementos da orientação a objetos, classe e método.

Ontologias de Reuso:

Prefixo	**Definição**	**Base**	**Tipo**	**Análise**
SCO	Ontologia de código-fonte que descreve o conceitos gerais de um código-fonte, independente de linguagem de programação.	UFO	Core	Código OO é uma especialização de código-fonte.

A.3.6 [CONF-REFE] Control Premise of Reference Ontology

Atividade do processo de suporte configuration management que objetiva controlar as alterações, versões e entregas das informações registradas no Documento de Ontologia de Referência, descrito na atividade [DOCM-REFE] Document Premise of Reference Ontology.

Para isso, sugere-se o uso do cabeçalho do documento para registrar a versão e sua respectiva data, armazenando o histórico de todas as versões para eventuais comparações ou restaurações de versões anteriores.

Esta atividade define: **Who**, *knowledge engineer* como responsável; **What-Input**, Documento de Ontologia de Referência elaborado na atividade [DOCM-REFE] Document Premise of Reference Ontology; **What-Output**, controle do Documento de Ontologia de Referência.

No *running example*, o cabeçalho do Documento de Ontologia de Referência é adotado para controlar suas versões, como apresentado na Tabela 21.

A.3.7 [SAVA01] Evaluate Premise of Reference Ontology

Atividade do processo de suporte evaluation que objetiva avaliar se as informações registradas no Documento de Ontologia de Referência atendem às expectativas do *ontology owner* e estão de acordo com o *domain expert*. Para isso, deve-se validar as informações registradas junto ao *ontology owner* e retornar ao início da fase sempre que necessário.

Esta atividade define: **Who**, *ontology owner* e *domain expert* como responsável; **What-Input**, Documento de Ontologia de Referência elaborado na atividade [DOCM-REFE] Document Premise of Reference Ontology; **What-Output**, avaliação do Documento de Ontologia de Referência.

No *running example*, reuniões são realizadas com *ontology owner* e *domain expert* a fim de validar o documento de ontologia de referência, apresentado na Tabela 21.

A.4 Reference Phase

A Fase de Referência (Reference Phase) objetiva representar de forma bem fundamentada a conceituação do domínio com base na especificação elaborada na Conception Phase e nas questões definidas na Pre-Reference Phase, que devem ser revisadas a cada iteração, a fim de adicionar, detalhar e excluir requisitos e questões relacionados às necessidades do domínio. Para este contexto, conceituação é uma estrutura semântica intencional que codifica as regras implícitas que restringem a estrutura de uma parte da realidade (USCHOLD; KING, 1995).

Esta fase está baseada no processo principal de **capture**, que é apoiado nos processos de suporte de **knowledge acquisition**, a fim de extrair o conhecimento do domínio; de **reuse**, a fim de reutilizar recursos ontológicos e não ontológicos reconhecidos; de **documentation**, a fim de registrar esse conhecimento como modelo conceitual e parte do Documento da Ontologia de Referência; de **configuration management**, a fim de controlar as alterações, versões e entregas das informações documentadas; de **evaluation**, a fim de verificar se a ontologia está sendo construída conforme os requisitos elicitados e validar se ela atende ao seu propósito; e de **publication**, a fim de disponibilizar a ontologia de referência.

Observa-se que uma única especificação da ontologia pode originar processos distribuídos na fase de referência, conforme os subdomínios definidos na atividade [REQI-SUBD] Identify Subdomains. As atividades desses respectivos processos são apresentadas nas subseções seguintes.

A.4.1 [CAPT-CONC] Identify Concepts

Atividade do processo principal de capture que identifica quais conceitos do domínio fazem parte do propósito da ontologia segundo as fontes de conhecimento. Esta atividade

trata *conceito* como uma concepção ou ideia de uma realidade e, por isso, tipos, propriedades e exemplos não são identificados como conceitos, mas como parte da descrição dos conceitos.

Para isso são definidos as seguintes subatividades:

A.4.2 [CAPT-CATA] Catalog Concepts

Atividade do processo principal de capture que objetiva catalogar os conceitos relacionados ao domínio da ontologia, segundo os critérios definidos na atividade [CAPT-CRIT] Define Concept Criteria.

Os conceitos podem ser catalogados por meio de três diferentes fontes de conhecimento: fonte de dados, *domain expert* e *ontology owner*. Considera-se como fonte de dados os livros, padrões, especificações, artigos e fontes reconhecidas pelas comunidades relacionadas ao domínio. Tais conceitos formam uma lista simples de termos e significados extraídos das diferentes fontes, ou seja, contendo conflitos, sobreposições e sinônimos.

O processo de suporte de knowledge acquisition objetiva extrair o conhecimento das fontes de conhecimento, seguindo a atividade:

KNOW-CONC Capture Concepts : atividade que objetiva captar os conceitos relacionados ao domínio da ontologia. Para a captação de conceitos a partir dos *ontology owner* e *domain expert*, sugere-se a aplicação de técnicas de *brainstorming*, entrevista, formulário e mapeamento conceitual para extrair os conceitos com maior evidência no domínio. Para a captação de conceitos a partir de fontes de dados, sugere-se a busca por fontes de dados em mecanismos de busca e repositórios de dados a fim de aplicar análise de texto para extrair os conceitos com maior evidência no domínio. Ressalta-se que nesta atividade é importante captar todos os conceitos relacionados ao domínio sem se preocupar com as suas sobreposições ou relações.

O processo de suporte de reuse objetiva aproveitar as definições já estabelecidas em fonte de dados, seguindo a atividade:

REUS-DATA Reuse Data Source : atividade que objetiva adotar fontes de dados relacionadas ao domínio da ontologia para a captura dos conceitos. Uma boa prática de engenharia não é "inventar" descrições, mas importá-las de fontes autorizadas (NICOLA; MISSIKOFF, 2016).

Embora esta atividade seja guiada por critérios, permanece subjetiva e dependente da análise dos papéis envolvidos. Apenas critérios definidos a partir da análise individual da fonte de conhecimento são aplicados nesta atividade. Critérios que requerem análise conjunta das fontes de conhecimento serão aplicados na atividade [CAPT-MODE] Model Ontology.

Vale ressaltar que esta atividade é importante para otimizar a atividade [CAPTMODE] Model Ontology, uma vez que o conhecimento do domínio pode ser identificado por diferentes papéis envolvidos de modo que as instruções básicas do domínio sejam identificadas e organizadas.

Esta atividade define: **Who**, *knowledge engineer* como responsável e *domain expert* e/ou *ontology owner* como participante. Para otimizar o processo, esta atividade pode ser conduzida de forma distribuída pelos papéis, de modo que a captação de conceitos pode ser conduzida pelo *knowledge engineer* e/ou *domain expert* e/ou *ontology owner*. **What-Input**, documento de especificação de ontologia elaborado na atividade [DOCMSPEC] Document Specification; **What-Output**, conceitos do domínio extraídos a partir das fontes de conhecimento.

No *running example*, livros e documentos de especificação das linguagens de programação que aplicam orientação a objetos foram definidos como fonte de conhecimento e utilizados para a captura dos conceitos, conforme os critérios estabelecidos (atividade descrita na Seção A.4.1). Uma vez que há uma grande quantidade de linguagens de programação relacionadas ao domínio, a busca por fonte de conhecimento foi direcionada às linguagens de programação Eiffel, Smalltalk, Java, C++ e Python. A Tabela 22 apresenta um fragmento do catálogo de conceitos elaborado para o domínio de orientação a objetos.

Tabela 22 – Fragmento do Catálogo de Conceitos

Fonte:	Smalltalk and Object Orientation (HUNT, 2012)	
Conceito	**Definição**	**Instância**
Class	Bloco de construção básico que atua como template para a construção de instâncias.	`Object subclass: #Polygon`
Instance Variable	Variável que mantém valor próprio para cada instância da classe.	`instanceVariableNames: 'side'`

Fonte:	Eiffel: Analysis, Design and Programming Language (EIFFEL, 2006)	
Conceito	**Definição**	**Instância**
Class	Implementação de um tipo de dado abstrato com o objetivo de ser a unidade modular de decomposição do software e fornecer a base para o sistema de tipos.	`class Polygon end`
Attribute	Tipo de feature que será associado a cada instância da classe.	`side : INTEGER`

Fonte:	Object-Oriented Programming in C++ (LAFORE, 1997)	
Conceito	**Definição**	**Instância**
Class	Classe serve como um blueprint que especifica quais dados e quais funções irão compor os objetos dessa classe.	`class Polygon{ };`
Data Member	Item de dado incluído dentro de uma classe.	`private: int side;`

Fonte:	Especificação da Linguagem Java (GOSLING et al., 2018)	
Conceito	**Definição**	**Instância**

Class	Define novos tipos de referência e descreve como eles são implementados.	`public class Polygon{ }`
Instance Variable	Variável definida para cada instância da classe.	`private int side;`

Fonte:	Python 3 Object Oriented Programming (PHILLIPS, 2010)	
Conceito	**Definição**	**Instância**
Class	Classe é como blueprint para a criação de objetos.	`class Polygon:`
Data Attribute	Atributo que define um valor de dados para cada instância.	`side = None`

Referências:
EIFFEL, E. Analysis, design and programming language. ECMA Standard ECMA-367, ECMA, 2006.
GOSLING, J. et al. The Java language specification: Java SE 10 edition, 20 February 2018. 2018.
HUNT, J. Smalltalk and object orientation: an introduction. Springer Science & Business Media, 2012.
LAFORE, R. Object-oriented programming in C++. Pearson Education, 1997.
PHILLIPS, D. Python 3 object oriented programming. Packt Publishing Ltd, 2010.

A.4.3 [CAPT-VIEW] Extract Ontology View

Atividade do processo principal de capture que objetiva extrair a visão das ontologias de referência a serem reusadas na modelagem da ontologia em construção.

O processo de suporte de knowledge acquisition objetiva extrair o conhecimento sobre a ontologia a ser reusada, seguindo a atividade:

KNOW-VIEW Capture View Concepts : atividade que objetiva identificar os conceitos da ontologia que formarão a visão da ontologia reusada como parte da ontologia em construção. Para ontologia reusada de núcleo, deve-se identificar os conceitos que podem ancorar os conceitos da ontologia em construção. Para ontologia reusada de domínio, deve-se identificar os conceitos que respondem às questões de competência da ontologia em construção.

Para a definição da ontologia reusada, deve-se utilizar uma estratégia de modularização sobre a ontologia original, uma vez que será elaborada uma visão da ontologia que irá refletir uma parte ou um conjunto de conceitos extraídos a partir dela. À medida que a visão vai sendo construída, uma análise ontológica deve ser aplicada a fim de garantir a compatibilidade ontológica com a ontologia em contrução.

Assim, deve-se construir a visão da ontologia reusada, seguindo algumas orientações: (i) se a ontologia reusada corresponde por completo ao propósito da ontologia em construção,

a ontologia reusada completamente deve compor a visão; (ii) se apenas uma parte da ontologia reusada corresponde ao propósito da ontologia em construção, a ontologia reusada deve ser particionada e somente essa parte deve compor a visão; e (iii) se apenas alguns conceitos da ontologia reusada correspondem ao propósito da ontologia em construção, os conceitos relevantes da ontologia reusada devem ser identificados para que deles possam ser visitadas recursivamente suas relações, a fim de reunir os conceitos para compor a visão.

Vale ressaltar que esta atividade está mais relacionada a modularização e aplicação de análise ontológica do que reengenharia ontológica, processo de recuperar e mapear um modelo conceitual de uma ontologia implementada para outro modelo conceitual mais adequado, que é reimplementado (GÓMEZ-PÉREZ; ROJAS-AMAYA, 1999).

Esta atividade define: **Who**, *knowledge engineer* como responsável; **What-Input**, ontologias a serem reusadas na ontologia em construção, identificadas na atividade [CAPT-REUS] Define Ontology to Reuse; **What-Output**, visão das ontologias a serem reusadas na ontologia em construção;

No *running example*, a visão da ontologia SCO foi elaborada a fim de representar apenas os conceitos oriundos ao domínio de orientação a objetos, apresentados na Figura 36. Tal visão é um recorte da ontologia original de SCO (apresentada na Seção 3.2).

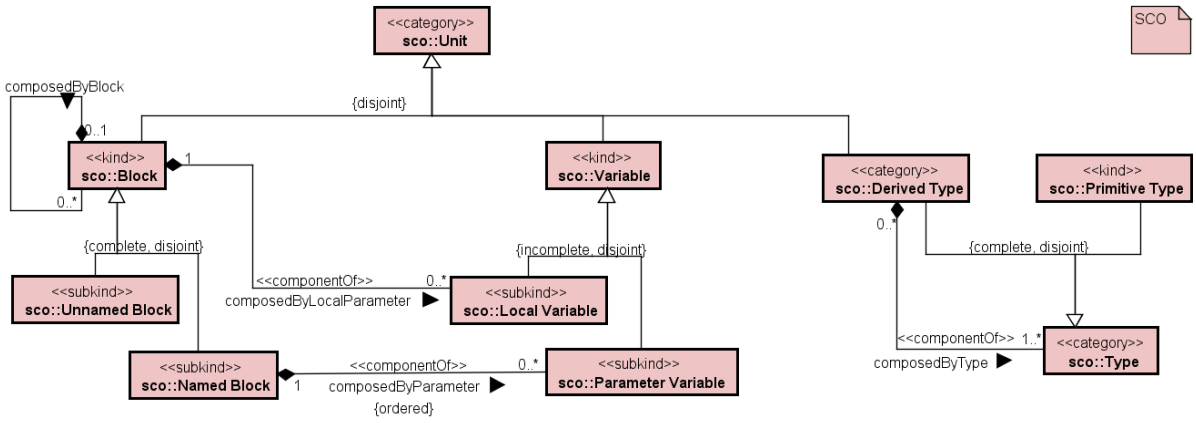


Figura 36 – Visão da Ontologia SCO

A.4.4 [CAPT-AXIM] Identify Axioms

Atividade do processo principal de capture que identifica os axiomas de restrição e inferência que o modelo conceitual da ontologia em construção deve considerar. Deve-se identificar as restrições que o modelo conceitual não consegue representar e registrá-las como axiomas informais, ou seja, em linguagem natural. À medida que o modelo conceitual é elaborado, deve-se retornar a esta atividade para garantir um modelo mais completo e desambíguo.

Esta atividade define: **Who**, *knowledge engineer* como responsável e *domain expert* como participante; **What-Input**, conceitos identificados na atividade [CAPT-CATA] Catalog Concepts ; **What-Output**, axiomas informais da ontologia em construção.

No *running example*, axiomas foram definidos para estabelecer restrições do modelo, tal como a relação de herança, que é estabelecida quando uma subclasse é herdada a partir de uma superclasse, declarando o seguinte axioma: $\forall c_1, c_2 : Class, i : Inheritance, inheritsIn(c_1, i) \wedge inheritedFrom(c_2, i) \rightarrow subClassOf(c_1, c_2)$

A.4.5 [CAPT-MODE] Model Ontology

Atividade do processo principal de capture que modela conceitos e relações cobertos pelo propósito da ontologia em uma linguagem de representação de ontologia, independente de tecnologia. Esta modelagem deve aplicar tanto análise ontológica, a fim de categorizar os conceitos segundo uma ontologia de fundamentação, quanto padrões ontológicos conceituais, a fim de aproveitar fragmentos de ontologias de fundamentação. Esta representação do conhecimento na forma de um modelo conceitual estabelece compromissos ontológicos sobre o domínio, uma vez que deve-se representar parte das informações sobre o mundo e ignorar outras (GASEVIC; DJURIC; DEVEDZIC, 2006).

A modelagem da ontologia pode seguir a abordagem (i) *top-down*, iniciando com os conceitos mais gerais do domínio e, em seguida, especializando-os; (ii) *bottom-up*, iniciando com os conceitos mais específicos do domínio e, em seguida, agrupando-os em conceitos mais gerais; ou (iii) *middle-out*, iniciando com os conceitos mais relevantes e, em seguida, generalizando-os e especializando-os combinando as abordagens *top-down* e *bottom-up*. Sugere-se a aplicação da abordagem *middle-out*, que pode diminuir o retrabalho e o esforço de modelagem, encontrando equilíbrio nos níveis de detalhe, começando com os conceitos mais relevantes e somente quando necessário, especializando-os ou generalizando-os (FERNÁNDEZ-LÓPEZ; GÓMEZ-PÉREZ, 2002).

O processo de suporte de knowledge acquisition objetiva definir um consenso sobre o domínio da ontologia, seguindo as atividades:

KNOW-CONS Concepts Consensus : atividade que objetiva estabelecer consenso semântico sobre os conceitos do domínio da ontologia em construção, a fim de definir sua conceituação e esclarecer dúvidas, ambiguidades e conflitos. Esta atividade está interessada em mitigar as diferenças semânticas das fontes de conhecimento, independentemente das diferenças lexicais. Assim, para cada conceito catalogado para o domínio, deve-se trocar informações, defender diferentes percepções e chegar a uma conceituação compartilhada.

Ademais, deve-se aplicar os critérios referentes ao conjunto de fontes de conhecimento, definidos na atividade [CAPT-CRIT] Define Concept Criteria. Para isso, os conceitos

do catálogo de diferentes fontes são validados de acordo com os critérios e discutidos com os especialistas de domínio. O consenso pode ser criado a partir de uma síntese das fontes de conhecimento ou, concordando com o propósito da ontologia e a discussão dos especialistas, favorecendo uma ou outra fonte.

KNOW-TERM Concepts Terminology : atividade que objetiva definir o termo representativo para cada conceito consensual definido na atividade KNOW-CONS Concepts Consensus . *Termo* é definido como a representação do conceito do domínio na ontologia.

Para isso, deve-se revisitar os conceitos catalogados, considerando: (i) se o termo é consensual entre as diferentes fontes, então este deve ser definido como o termo representativo do conceito da ontologia; (ii) se o termo não é consensual, então o termo representativo do conceito da ontologia deve ser definido subjetivamente de acordo com a maior relevância para o domínio.

O processo de suporte de reuse objetiva adotar a linguagem de modelagem definida na atividade [CAPT-LANG] Define Modeling Language. Para cada conceito modelado, deve-se definir sua categoria acordando com a ontologia de fundamentação definida na atividade [CAPT-FOUN] Define Foundational Ontology. Para cada relação estabelecida, deve-se definir o seu nome, direção, tipo e cardinalidade. Ademais, à medida que o modelo vai sendo construído deve-se aplicar padrões ontológicos conceituais para garantir a qualidade padrão do modelo. Esta atividade requer tomada de decisão sobre o domínio e sua representação e deve considerar o conhecimento adquirido a partir das fontes de conhecimento.

Esta atividade pode ser realizada de forma direcionada ou distribuída de acordo com os subdomínios identificados na atividade [REQI-SUBD] Identify Subdomains. Caso seja realizada de forma distribuída, esta atividade irá originar visões da ontologia em construção que deverão, posteriormente, ser integradas.

Deve-se considerar que a modelagem é evolutiva e, por isso, o modelo deve sofrer ajustes, aplicar análise ontológica, integrar visões de ontologias e modularizar continuamente, ou seja, deve-se retornar a esta atividade sempre que necessário.

Esta atividade define: **Who**, *knowledge engineer* como responsável e *domain expert* como participante; **What-Input**, conceitos identificados na atividade [CAPT-CONC] Identify Concepts a partir das fontes de conhecimento e das ontologias reusadas; **What-Output**, modelo conceitual da ontologia em construção.

No *running example*, o modelo conceitual do domínio de orientação a objetos foi costruído, sendo um fragmento dele apresentado na Figura 37.

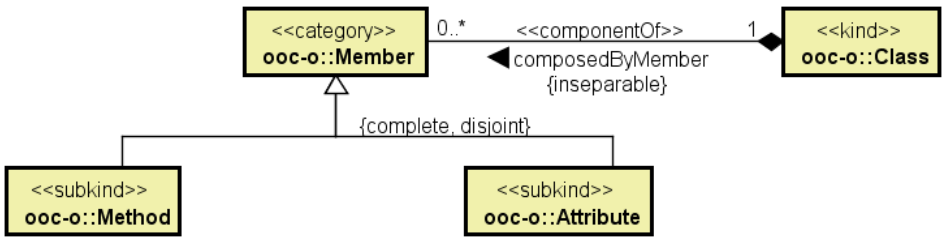


Figura 37 – Modelo conceitual dos principais conceitos de orientação a objetos

A.4.6 [CAPT-INTE] Integrate Ontology

Atividade do processo principal de capture que objetiva integrar tanto as visões da ontologia em construção quanto as visões das ontologias reusadas. Se a ontologia for construída de forma distribuída conforme os subdomínios identificados, cada subdomínio dará origem a uma visão da ontologia. Se a ontologia for construída reusando ontologias existentes, cada ontologia existente dará origem a uma ou mais visões da ontologia. Tais visões deverão ser integradas à ontologia em construção e, portanto, essa atividade é executada de forma iterativa com a atividade [CAPT-MODE] Model Ontology, a fim de elaborar um modelo mais coeso e completo.

Para visões construídas a partir de ontologias de fundamentação, a integração pode ser feita por meio de especialização (conceitos são explicitamente extendidos) ou analogia (conceitos são implicitamente usados para derivar a ontologia em construção) (FALBO, 2014), de modo que os conceitos da ontologia reusada fundamentem os conceitos representados na ontologia em construção. Ademais, a integração com a ontologia de fundamentação explicita o compromisso ontológico e produz um modelo conceitual uniforme para integrar diferentes visões de ontologias. Para visões construídas a partir de ontologias de núcleo, a integração pode ser feita principalmente por meio de especialização, de modo que conceitos e relações da ontologia de núcleo são estendidos para representar conceituações mais específicas do domínio da ontologia em construção (FALBO, 2014). Para visões construídas a partir de ontologias de domínio, a integração pode ser baseada na ideia de composição (MITRA; WIEDERHOLD, 2004), por meio da operação de união. Nesse caso, uma ontologia O é formada pela união dos conceitos da ontologia em construção O_1 e da visão da ontologia reusada O_2 mesclando os conceitos comuns. Ademais, relações de associação, extensão e especialização entre as entidades de O_1 e O_2 podem ser necessárias para estabelecer a união das ontologias. Para as visões da ontologia em construção, a integração pode seguir a combinação destas estratégias.

Esta atividade define: **Who**, *knowledge engineer* como responsável e *domain expert* como participante; **What-Input**, visões da ontologia em construção e as visões das ontologias reusadas, elaboradas nas atividades [CAPT-MODE] Model Ontology e [CAPT-VIEW] Extract Ontology View. **What-Output**, modelo conceitual da ontologia em

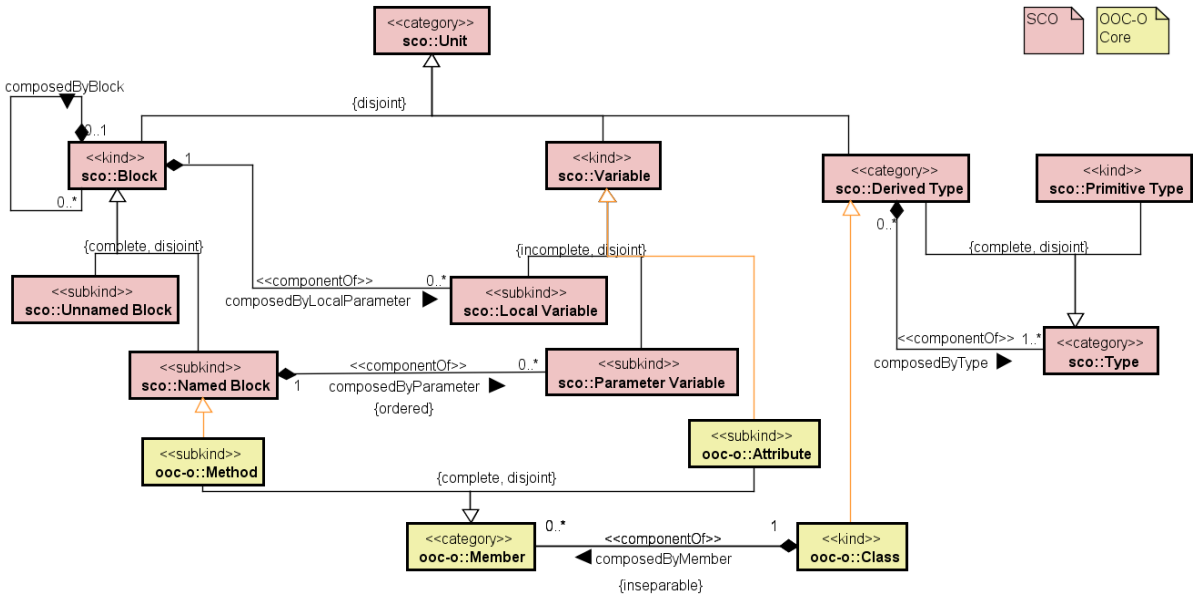


Figura 38 – Integração da ontologia reusada com a ontologia em construção

construção.

No *running example*, a integração foi realizada de forma iterativa de modo que a visão da ontologia SCO reusada (cor vermelha) ancorasse os conceitos da ontologia de orientação a objetos (cor amarela) com relações de especialização (cor amarela), como apresentado na Figura 38.

A.4.7 [CAPT-MODU] Modularize Ontology

Atividade do processo principal de capture que objetiva identificar os módulos nos quais a ontologia pode ser decomposta e que podem ser considerados separadamente enquanto estão interligados com outros módulos (D'AQUIN, 2012).

Uma ontologia de domínio complexa e composta por muitos conceitos não pode ser construída e mantida facilmente, sendo dificilmente reusada em sua totalidade. Assim, o objetivo de utilizar a modularização inclui (D'AQUIN, 2012): (i) melhorar o desempenho, permitindo processamento distribuído ou direcionado; (ii) facilitar o desenvolvimento e a manutenção da ontologia, dividindo-a em componentes independentes e pouco acoplados; e (iii) facilitar a reutilização de partes da ontologia, uma vez que adaptações relevantes já foram aplicadas.

Esta atividade é executada de forma iterativa a fim de elaborar um modelo mais compreensível e desacoplado, priorizando maior coesão e menor dependência. Diferentes cenários e aplicações requerem diferentes maneiras de modularizar (D'AQUIN et al., 2007) e, por isso, deve-se aplicar uma estratégia de modularização adequada, considerando: (i) critério de compreensão e tamanho da ontologia e (ii) preservação da completude dos módulos, atribuindo relações entre os módulos associados e replicando conceitos necessários

para a sua compreensão e visualização.

O processo de suporte de reuse objetiva dispor das estratégias de modularização já discutidas na literatura para apoiar a modularização da ontologia, tais como estratégias de particionamento e extração transversal, seguindo a atividade:

REUS-MODU Adopt Modularization Strategy : atividade que objetiva definir a estratégia de modularização a ser adotada na modelagem da ontologia em construção e das visões das ontologias reusadas. Para isso, sugere-se considerar duas principais estratégias de modularização (D'AQUIN, 2012): (i) se a ontologia agrega muitos conceitos que dificultam sua compreensão e manutenção recomenda-se aplicar a estratégia de particionamento para que os subdomínios sejam tratados de forma independente e mantenham sua completude na forma de módulos, ou seja, particionar a ontologia; (ii) se alguns conceitos são tratados na ontologia de forma conjunta, recomenda-se aplicar o método de extração transversal para que, a partir destes conceitos, seus relacionamentos sejam visitados recursivamente e, a partir deles, seja identificado o conjunto de conceitos e relações necessários para serem reunidos.

Esta atividade define: **Who**, *knowledge engineer* como responsável e *domain expert* como participante; **What-Input**, modelo conceitual da ontologia em construção, elaborado nas atividades [CAPT-MODE] Model Ontology e [CAPT-INTE] Integrate Ontology; **What-Output**, modelo conceitual da ontologia em construção.

No *running example*, a ontologia orientada a objetos foi modularizada adotando a estratégia de particionamento, onde *Core* representa os conceitos principais do código orientado a objetos, *Class* representa os conceitos derivados a partir de classe e *Class Member* representa os conceitos derivados a partir de membros de classe, ou seja, atributos e métodos. A Figura 39 apresenta a modularização adotada na ontologia de orientação a objetos, bem como sua relação com as ontologias reusadas.

A.4.8 [DOCM-MODE] Document Reference Ontology

Atividade do processo de suporte documentation que objetiva documentar o modelo conceitual da ontologia em construção por meio do Documento de Ontologia de Referência, complementando o documento elaborado na atividade [DOCM-REFE] Document Premise of Reference Ontology, bem como seguir as considerações:

- Modularização: representação dos módulos definidos para a ontologia em construção. Sugere-se utilizar um diagrama de pacotes UML cujos pacotes são relacionados por meio de dependências (FALBO, 2014) e padronizados por cores.
- Modelo Conceitual: apresentação do modelo conceitual da ontologia;

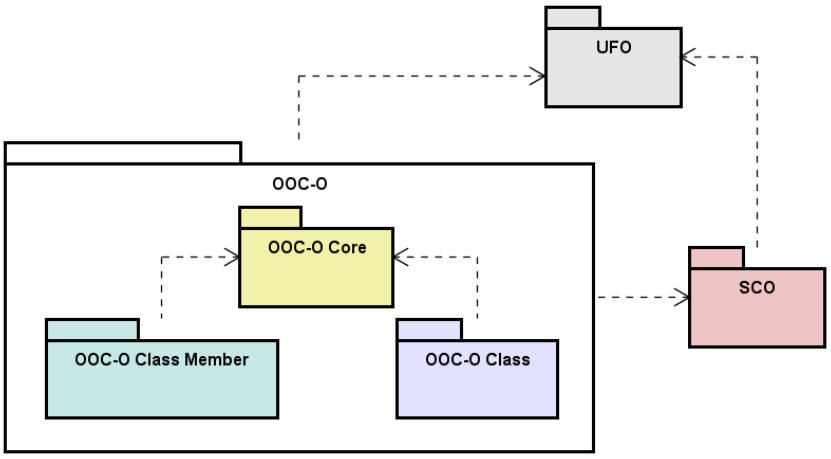


Figura 39 – Modularização da ontologia

- Axiomas do Modelo: apresentação das restrições definidas nos axiomas informais para axiomas escritos em linguagem de primeira ordem ou de restrição de objeto. Axiomas formais apoiam as descrições dos axiomas informais, não o substituindo;
- Descrição do Modelo: texto em linguagem natural contendo uma descrição do modelo e destacando seus conceitos e relações.
- Dicionário do Modelo: apresentação dos conceitos e seus significados na forma de um dicionário.

No *running example*, o Documento de Ontologia de Referência apresentado na Tabela 21 é complementado com a documentação do modelo conceitual, apresentado na Tabela 23.

Tabela 23 – Fragmento da Ontologia de Referência OOC-O

Documento:	Ontologia de Referência	**Versão:**	v.01
Ontologia:	OOC-O Object-Oriented Code	**Data:**	11/04/2019

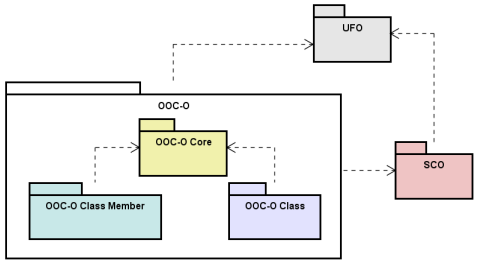
Linguagem de Modelagem:	
OntoUML	
Ontologia de Fundamentação:	
UFO	Ontologia caracterizada em lógica modal e psicologia cognitiva que pretende fornecer maior nível semântico sobre o mundo em um modelo conceitual de dado domínio de conhecimento, sistematiza questões como noções de tipos e suas instâncias; objetos, e suas propriedades intrínsecas; a relação entre identidade e classificação; distinções entre tipos e suas relações; relações parte-todo entre outros.
Critérios:	
CR01	Conceito referente à programação em tempo de compilação.
CR02	Conceito presente em mais de 50% das linguagens de programação analisadas.
CR03	Conceito relacionado aos principais elementos da orientação a objetos, classe e método.

Ontologias de Reuso:		
Prefixo	**Definição**	**Análise**

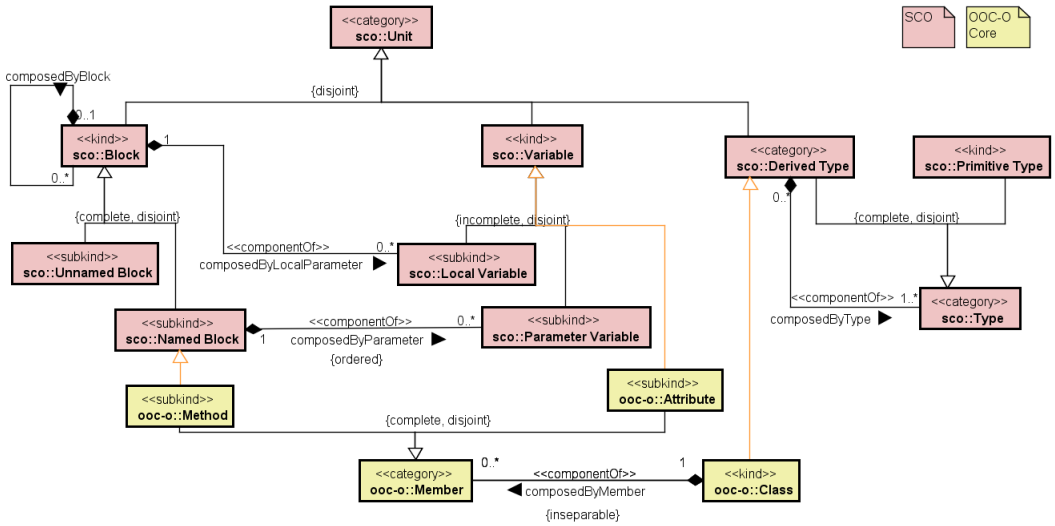
SCO	Core Ontology - UFO Ontologia de código-fonte que representa os conceitos principais de um código, idenpendente da linguagem de programação.	Código de orientação a objetos é uma especialização da ontologia.

Modularização:

Subontologia	**Definição**
Core	Ontologia que representa os conceitos principais do código orientado a objetos.
Class	Ontologia que detalha os conceitos derivados a partir de classe.
Class Member	Ontologia que detalha os conceitos derivados a partir de membros de classe, ou seja, atributos e métodos.



Ontologia: OOC-O Core



Axiomas:

Axioma	**Definição**

Dicionário de Conceitos:

Conceito	**Definição**
Class	Tipo de dado abstrato e mecanismo para definir um tipo de dado abstrato em uma linguagem de programacao OO. Classe descreve os atributos de seus objetos, bem como os métodos que eles podem executar.
Member	Membros que compõem as classes, tal como Method e Attribute.
Method	Named Block (Function ou Procedure) que pertence à classe e fornece uma maneira de definir o comportamento de um objeto que é invocado quando uma mensagem é recebida pelo objeto.
Attribute	Variável que pertence à classe e fornece uma maneira de definir o estado de seus objetos.

A.4.9 [CONF-MODE] Control Reference Ontology

Atividade do processo de suporte configuration management que objetiva controlar as alterações, versões e entregas das informações registradas no Documento de Ontologia de Referência, descrito na atividade [DOCM-MODE] Document Reference Ontology, complementando os registros realizados na atividade [CONF-REFE] Control Premise of Reference Ontology.

Para isso, sugere-se o uso do cabeçalho do documento para registrar a versão e sua respectiva data, armazenando o histórico de todas as versões para eventuais comparações ou restaurações de versões anteriores.

Esta atividade define: **Who**, *knowledge engineer* como responsável; **What-Input**, Documento de Ontologia de Referência elaborado na atividade [DOCM-MODE] Document Reference Ontology; **What-Output**, controle do Documento de Ontologia de Referência.

No *running example*, o cabeçalho do Documento de Ontologia de Referência é adotado para controlar suas versões, complementando os controles registrados na atividade [CONF-REFE] Control Premise of Reference Ontology, conforme apresentado na Tabela 23.

A.4.10 [EVAL-MODE] Evaluate Reference Ontology

Atividade do processo de suporte evaluation que objetiva avaliar se as informações registradas no Documento de Ontologia de Referência atendem às expectativas do *ontology owner* e estão de acordo com o *domain expert*, aplicando avaliação estática sobre os requisitos e propósito da ontologia. Para isso, deve-se validar as informações e retornar ao início da fase sempre que necessário.

Para **verificação** da ontologia, deve-se avaliar por meio de revisão técnica se os requisitos funcionais levantados são respondidos pelos conceitos e relações representados no modelo conceitual da ontologia. Para isso, sugere-se que as questões de competência levantadas na atividade [REQI-ELIC] Elicit Requirements sejam respondidas exclusivamente pelos conceitos e relações da ontologia, relacionando-os no intuito de construir um caminho de raciocínio.

O padrão $c_1 > r > c_2$ transcreve um relacionamento onde, c_1 é o conceito de origem da relação, c_2 é o conceito destino da relação e r é a relação existente entre os conceitos c_1 e c_2. A transcrição de uma dada relação r deve seguir o seu tipo de relação: relação de associação é representada pelo nome da relação, relação de composição é representada por *componentOf*, relação de especialização é representada por *subtypeOf* e relação de generalização é representada por *specializedBy*.

Para **validação** da ontologia, deve-se avaliar se a ontologia atende às situações

do mundo real por meio da instanciação de seus conceitos. Para isso, sugere-se que as instâncias dos conceitos catalogados na atividade [CAPT-CATA] Catalog Concepts sejam mapeadas para instâncias dos conceitos da ontologia.

Esta atividade define: **Who**, *knowledge engineer* como responsável; **What-Input**, Documento de Ontologia de Referência elaborado na atividade [DOCM-MODE] Document Reference Ontology; **What-Output**, avaliação do Documento de Ontologia de Referência.

No *running example*, a verificação da ontologia é dada pelas respostas às questões de competência, tal como o requisito funcional RF03 respondido por: Member *componentOf* Class; Attribute e Method *subtypeOf* Member.

A validação da ontologia é dada pela instanciação dos seus conceitos, tal como o código em linguagem Java (apresentado na Listagem A.1), onde: Polygon *instanceOf* Class; side *instanceOf* Attribute; perimeter *instanceOf* Method; private and public *instanceOf* Visibility; int *instanceOf* Primitive Type & Value Type; e void *instanceOf* Primitive Type & Return Type.

Listagem A.1 – Fragmento de código Java.

```
1 public class Polygon{
2       private int side;
3       public void perimeter(){};
4 }
```

A.4.11 [PUBL-REFE] Publish Reference Ontology

Atividade do processo de suporte publication que objetiva publicar a Ontologia de Referência, definida na atividade [DOCM-MODE] Document Reference Ontology.

Esta atividade define: **Who**, *knowledge engineer* como responsável; **What-Input**, Documento de Ontologia de Referência elaborado na atividade [DOCM-MODE] Document Reference Ontology; **What-Output**, ontologia de referência disponibilizada.

No *running example*, o Documento de Ontologia de Referência foi publicado no site do projeto e está acessível pelo endereço <https://nemo.inf.ufes.br/projetos/oscin>.

A.5 Pre-Operational Phase

A Fase Pré-Operacional (Pre-Operational Phase) objetiva definir questões tecnológicas que irão guiar a implementação da ontologia operacional, ou seja, decisões dependentes de ambiente tecnológico. Neste contexto, a ontologia de referência, elaborada na Reference Phase, fornece as informações iniciais para especificar o *design* que pode ser originado por diferentes escolhas tecnológicas.

Este processo visa preencher a lacuna entre a modelagem conceitual de ontologias de referência e a codificação delas em termos de uma linguagem de ontologia operacional (GUIZZARDI, 2007). Para isso, define um conjunto de atividades que devem ser executadas de forma iterativa até definir o padrão de qualidade esperado. Dado o carater técnico do processo, sugere-se que os requisitos não funcionais definidos na atividade [REQI-ELIC] Elicit Requirements sejam revisitados a cada iteração, a fim de adicionar, detalhar e excluir requisitos não funcionais relacionados às questões tecnológicas da ontologia operacional. As atividades desses respectivos processos são apresentadas nas subseções seguintes.

A.5.1 [DESG-LANG] Define Encoding Language

Atividade do processo principal de design que objetiva definir a linguagem de representação a ser aplicada na construção da ontologia operacional, ou seja, na codificação. A escolha da linguagem influencia diretamente na integração e reuso da ontologia em construção, uma vez que ontologias desenvolvidas em diferentes linguagens podem requerer grande esforco de adaptação.

O processo de suporte de reuse objetiva aproveitar a linguagem de codificação já estabelecida para representação de ontologias operacionais, seguindo a atividade:

REUS-ELAN Adopt Encoding Language : atividade que objetiva definir a linguagem de codificação a ser adotada na construção da ontologia operacional. Há várias linguagens para representar ontologias, como KIF, com base na lógica de primeira ordem; RDF(S), com base em redes semânticas e primitivas baseadas em frames; e OWL, com base em propriedades e atributos dos vocabulários RDF(S) correspondentes às *description logics* (MCGUINESS; HARMELEN, 2004). Dada a popularidade, a descrição de significados bem definidos e relações ricas, sugere-se o uso da linguagem OWL.

Esta atividade define: **Who**, *knowledge engineer* como responsável; **What-Input**, Documento de Ontologia de Referência elaborado na atividade [DOCM-MODE] Document Reference Ontology; **What-Output**, linguagem de codificação a ser adotada na ontologia operacional.

No *running example*, a linguagem OWL é definida para a codificação da ontologia operacional de orientação a objetos.

A.5.2 [DESG-VOCA] Identify Vocabularies

Atividade do processo principal de design que visa identificar os vocabulários a serem adotados na codificação da ontologia em construção, tanto vocabulários base para

auxiliar na construção da ontologia como vocabulários das ontologias reutilizadas.

Tais vocabulários devem adotar linguagem de codificação compatível com a linguagem adotada na ontologia em construção, definida na atividade [DESG-LANG] Define Encoding Language. Nesse contexto, *vocabulário* é definido como o conjunto de conceitos e relações de um determinado domínio, sem necessariamente adotar um formalismo complexo como o de uma ontologia.

O processo de suporte de knowledge acquisition objetiva extrair o conhecimento dos vocabulários, seguindo a atividade:

KNOW-VOCA Understand Vocabulary : atividade que objetiva compreender as classes e propriedades dos vocabulários adotados para a construção da ontologia operacional. Uma vez que nem todos os elementos do vocabulário serão reusados, deve-se definir aqueles que correspondem à ontologia em construção.

O processo de suporte de reuse objetiva aproveitar vocabulários disponíveis que auxiliem a construção da ontologia, seguindo as atividades:

REUS-BVOC Adopt Base Vocabularies : atividade que objetiva selecionar vocabulários para auxiliar a codificação da ontologia em construção, ou seja, vocabulários que não definem conceitos e relações do domínio da ontologia em construção.

Para isso, deve-se buscar por vocabulários em mecanismos de busca e repositórios. Tais vocabulários normalmente incluem, entre outros, <http://www.w3.org/2002/07/owl#>, <http://www.w3.org/1999/02/22-rdf-syntax-ns#> e <http://www.w3.org/2000/01/rdf-schema#>.

REUS-RVOC Adopt Vocabularies from Reused Ontologies : atividade que objetiva selecionar vocabulários das ontologias reusadas na ontologia de referência, ou seja, vocabulários que definem conceitos e relações do domínio da ontologia em construção.

Para isso, o *ontology engineer* deve buscar os recursos disponíveis das ontologias reusadas. Caso o vocabulário da ontologia reusada não seja acessível, ele será construído no processo de *implementation*, junto com a ontologia em construção.

Esta atividade define: **Who**, *knowledge engineer* como responsável; **What-Input**, Documento de Ontologia de Referência elaborado na atividade [DOCM-MODE] Document Reference Ontology; **What-Output**, vocabulários a serem reusados na ontologia operacional.

No *running example*, os seguintes vocabulários foram reusados: owl - <http://www.w3.org/2002/07/owl#>; rdf - <http://www.w3.org/1999/02/22-rdf-syntax-ns#>; xml - <http://www.w3.org/XML/1998/namespace>; xsd - <http://www.w3.org/2001/

XMLSchema#>; rdfs - <http://www.w3.org/2000/01/rdf-schema#>; sco - <http://nemo.inf.ufes.br/projetos/oscin/sco#>.

A.5.3 [DESG-CODE] Define Ontology Encoding

Atividade do processo principal de design que objetiva definir a codificação a ser aplicada na ontologia operacional, ou seja, a estrutura de elementos que representa a ontologia de referência em linguagem operacional.

Esta atividade indica como a arquitetura da ontologia operacional será derivada da ontologia de referência, considerando a linguagem de codificação definida. Uma ontologia pode ser definida como um conjunto de primitivas representacionais que modelam um domínio de conhecimento, caracterizadas como conceitos/classes, relações/propriedades e atributos/propriedades de tipos de dados (SLIMANI, 2015). No caso da linguagem OWL, a arquitetura será baseada em classes e propriedades.

A partir do modelo conceitual, deve-se definir os conceitos e relações que serão representados como classes e propriedades na ontologia operacional. A partir das classes, deve-se definir uma taxonomia da ontologia, ou seja, organizar os conceitos de forma hierárquica. No caso da linguagem OWL, podem ser criadas relações do tipo *subClassOf*, herdada a partir de RDFS.

Deve-se definir a nomenclatura e o uso dos vocabulários a serem adotados na codificação da ontologia. Para a nomenclatura, sugere-se a adoção de regras que permitam a associação com os conceitos e relações definidos na ontologia de referência. Para o vocabulário, sugere-se definir claramente como os conceitos e relações serão usados na ontologia operacional. Esta atividade é relevante para garantir o padrão de codificação da ontologia, especialmente quando a Operational Phase é realizada de forma paralela ou distribuída.

A arquitetura deve adotar modularização para facilitar a compreensão e reuso. Para isso, deve-se visitar a modularização definida para a ontologia de referência de modo a utilizá-la como base para a modularização da ontologia operacional e aplicar ajustes para a sua operacionalização. A arquitetura deve considerar as ontologias reusadas na construção da ontologia de referência na forma de vocabulários, bem como considerar características relacionadas ao raciocínio e performance.

O processo de suporte de knowledge acquisition objetiva extrair o conhecimento relacionado à ontologia operacional, seguindo a atividade:

Map Reference Ontology : atividade que objetiva mapear os conceitos e relações da ontologia de referência que serão representados como classes e propriedades na ontologia operacional.

Para linguagem OWL, cada classe é mapeada como membro da classe *Thing* de OWL e instância de *Class* e cada propriedade como *ObjectProperty* ou *DatatypeProperty*, herdadas a partir da classe *Property* de RDF. Ademais, podem ser definidas restrições de valores e cardinalidade por meio de propriedades.

Esta atividade define: **Who**, *ontology engineer* como responsável; **What-Input**, Documento de Ontologia de Referência elaborado na atividade [DOCM-MODE] Document Reference Ontology; **What-Output**, codificação da ontologia operacional.

No *running example*, os conceitos e relações da ontologia de referência de orientação a objetos são representados como classes e propriedades de OWL. A URI do indivíduo é formada pelo padrão `<module>_<class>_<method>_<variable>` de forma a assegurar o identificador único na ontologia.

A.5.4 [DOCM-OPER] Document Premise of Operational Ontology

Atividade do processo de suporte documentation que objetiva documentar as premissas da ontologia operacional por meio do Documento de Ontologia Operacional, que deve incluir as questões definidas na fase pré-operacional, bem como seguir as considerações:

- Linguagem de Codificação: identificação da linguagem de codificação a ser usada na implementação da ontologia;
- Vocabulários: prefixo identificador do vocabulário e sua definição;
- Regras de Codificação: descrição das regras a serem aplicadas na codificação da ontologia;
- Arquitetura: descrição ou representação da arquitetura a ser adotada na codificação da ontologia.

Esta atividade define: **Who**, *ontology engineer* como responsável; **What-Input**, atividades realizadas no processo principal de design, da Pre-Operational Phase; **What-Output**, Documento de Ontologia Operacional com suas respectivas premissas.

No *running example*, o Documento de Ontologia Operacional com as premissas estabelecidas para a ontologia de operacional de orientação a objetos é apresentado na Tabela 24.

Tabela 24 – Documento de Ontologia Operacional

Documento:	Ontologia Operacional	**Versão:**	v.01
Ontologia:	OOC-O Object-Oriented Code	**Data:**	26/04/2019

Linguagem de Codificação:
OWL

Vocabulários:

Prefixo	**Definição**
owl	http://www.w3.org/2002/07/owl#
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
xml	http://www.w3.org/XML/1998/namespace
xsd	http://www.w3.org/2001/XMLSchema#
rdfs	http://www.w3.org/2000/01/rdf-schema#
sco	http://nemo.inf.ufes.br/projetos/oscin/sco#

Regras de Codificação:

Regra	**Definição**
Nome de Classe	Classes são nomeadas a partir da ontologia de referência seguindo o padrão *Pascal Case*, letra inicial de cada palavra em maiúsculo, sem espaço, hífen, underscore ou pontuação.
Nome de Propriedade	Propriedades são nomeadas a partir da ontologia de referência seguindo o padrão *Camel Case*, letra inicial da primeira palavra em minúsculo e letra inicial das demais palavras em maiúsculo, sem espaço, hífen, underscore ou pontuação. Propriedades com mesmo nome na ontologia de referência são nomeadas adicionando o conceito de destino ao nome da relação.
URI Indivíduo	Individuos são nomeados seguindo o padrão `<module>_<class>_<method>_<variable>` conforme o tipo de indivíduo.
Descrição de Classe	As classes são descritas utilizando a propriedade *comment* de rdfs.
Relação de Especialização	A especialização entre as classes são definidas utilizando a propriedade *subClassOf* de rdfs.
Relação de Disjunção	A disjunção entre as classes são definidas utilizando a propriedade *disjointWith* de owl.
Tipo de Dado String	Dados do tipo string são definidos como *XMLSchema#string* de rdfs.

Arquitetura:
Ontologias:

Prefixo	**Descrição**
ooc-o	Ontologia que corresponde para a subontologia OOC-O Core de referência.
ooc-o-class	Ontologia que corresponde para a subontologia OOC-O Class de referência.
ooc-o-classmember	Ontologia que corresponde para a subontologia OOC-O Class Member de referência.

Taxonomia:
- Conceitos do tipo *kind* e *category* são ancorados na classe *Class* de owl;
- Conceitos do tipo *subkind* são ancorados nos *kinds* respectivos do domínio conforme o modelo conceitual;
- Conceitos do tipo *quality* são ancorados na propriedade *ObjectProperty* de owl. Exceto o conceito *name* que é ancorado na propriedade *DatatypeProperty* de owl.

A.5.5 [CONF-OPER] Control Premise of Operational Ontology

Atividade do processo de suporte configuration management que objetiva controlar as alterações, versões e entregas das informações registradas no Documento de Ontologia Operacional, descrito na atividade [DOCM-OPER] Document Premise of Operational Ontology.

Para isso, sugere-se o uso do cabeçalho do documento para registrar a versão e sua respectiva data, armazenando o histórico de todas as versões para eventuais comparações ou restaurações de versões anteriores.

Esta atividade define: **Who**, *ontology engineer* como responsável; **What-Input**, Documento de Ontologia Operacional elaborado na atividade [DOCM-OPER] Document Premise of Operational Ontology; **What-Output**, controle do Documento de Ontologia Operacional.

No *running example*, o cabeçalho do Documento de Ontologia Operacional é adotado para controlar suas versões, como apresentado na Tabela 24.

A.5.6 [EVAL-OPER] Evaluate Premise of Operational Ontology

Atividade do processo de suporte evaluation que objetiva avaliar se as informações registradas no Documento da Ontologia Operacional atendem às expectativas do *ontology owner* e estão de acordo com os requisitos da ontologia e da qualidade definidos na Conception Phase. Para isso, deve-se validar as informações registradas junto ao *ontology owner* por meio de revisão técnica e retornar ao início da fase sempre que necessário.

Esta atividade define: **Who**, *ontology engineer* e *ontology owner* como responsável; **What-Input**, Documento de Especificação da Ontologia e o Documento de Ontologia de Referência elaborados nas Conception Phase e Reference Phase; **What-Output**, avaliação do Documento de Ontologia Operacional.

No *running example*, reuniões são realizadas com *ontology owner* e *ontology engineer* a fim de validar o documento de ontologia operacional, apresentado na Tabela 24.

A.6 Operational Phase

A Fase Operacional (Operational Phase) objetiva implementar a ontologia de referência elaborada na Reference Phase em uma ontologia operacional, de acordo com o documento da ontologia operacional, descrito na Pre-Operational Phase. Esta fase transforma a ontologia de referência em uma ontologia interpretável por máquina, tornando-a disponível para ser usada em aplicações, suporte a decisões e raciocínio do domínio.

A.6.1 [IMPL-CONC] Code Concepts

Atividade do processo principal de implementation que codifica os conceitos da ontologia de referência como elementos da linguagem formal de ontologia operacional, de acordo com as premissas definidas no documento de ontologia operacional. Para linguagem OWL, os conceitos são codificados como classes na ontologia operacional.

O processo de suporte de reuse objetiva adotar a linguagem de codificação definida na atividade [DESG-LANG] Define Encoding Language e aproveitar vocabulários definidos na atividade [DESG-VOCA] Identify Vocabularies.

Esta atividade define: **Who**, *ontology engineer* como responsável; **What-Input**, Documento de Ontologia Operacional elaborado na atividade [DOCM-OPER] Document Premise of Operational Ontology; **What-Output**, conceitos codificados na ontologia operacional.

No *running example*, o conceito Member da ontologia de referência é codificado na ontologia operacional como apresentado na Listagem A.2.

Listagem A.2 – Fragmento dos conceitos da ontologia operacional.

```
1 <owl:Class rdf:about="http://ooc-o#Member">
2         <rdfs:comment rdf:datatype="http://www.w3.org/2000/01/rdf-schema#
      Literal">Element which makes up a class, defined as a attribute or method
      .</rdfs:comment>
3 </owl:Class>
```

A.6.2 [IMPL-RELC] Code Relations

Atividade do processo principal implementation que codifica as relações da ontologia de referência como propriedades e restrições em uma linguagem formal de ontologia operacional, de acordo com as premissas definidas no documento de ontologia operacional. Para linguagem OWL, as relações são codificadas como propriedades e restrições na ontologia operacional.

O processo de suporte de reuse objetiva adotar a linguagem de codificação definida na atividade [DESG-LANG] Define Encoding Language e aproveitar vocabulários definidos na atividade [DESG-VOCA] Identify Vocabularies.

Esta atividade define: **Who**, *ontology engineer* como responsável; **What-Input**, Documento de Ontologia Operacional elaborado na atividade [DOCM-OPER] Document Premise of Operational Ontology; **What-Output**, relações codificadas na ontologia operacional.

No *running example*, a relação componentOfClass da ontologia de referência é codificado na ontologia operacional como apresentado na Listagem A.3.

Listagem A.3 – Fragmento das relações da ontologia operacional.

```
<owl:ObjectProperty rdf:about="http://ooc-o#componentOfClass">
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#
    FunctionalProperty"/>
        <rdfs:domain rdf:resource="http://ooc-o#Member"/>
        <rdfs:range rdf:resource="http://ooc-o#Class"/>
</owl:ObjectProperty>
```

A.6.3 [IMPL-AXIO] Code Axioms

Atividade do processo principal de implementation que codifica os axiomas definidos na ontologia de referência de acordo com a linguagem operacional.

O processo de suporte de reuse objetiva adotar a linguagem de codificação definida na atividade [DESG-LANG] Define Encoding Language.

Esta atividade define: **Who**, *ontology engineer* como responsável; **What-Input**, Documento de Ontologia Operacional elaborado na atividade [DOCM-OPER] Document Operational Ontology; **What-Output**, axiomas codificados na ontologia operacional.

No *running example*, o axioma definindo Method e Attribute como disjoint é apresentado na Listagem A.4.

Listagem A.4 – Fragmento dos axiomas da ontologia operacional.

```
<rdf:Description>
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#
    AllDisjointClasses"/>
        <owl:members rdf:parseType="Collection">
            <rdf:Description rdf:about="http://ooc-o#Method"/>
            <rdf:Description rdf:about="http://ooc-o#Attribute"/>
        </owl:members>
</rdf:Description>
```

A.6.4 [DOCM-OPER] Document Operational Ontology

Atividade do processo de suporte documentation que objetiva documentar a ontologia operacional na linguagem de codificação definida na atividade [DESG-LANG] Define Encoding Language, seguindo as atividades do processo principal implementation, da Operational Phase e conforme as premissas definidas no documento de ontologia operacional, na atividade [DOCM-OPER] Document Premise of Operational Ontology.

Esta atividade define: **Who**, *ontology engineer* como responsável; **What-Input**, atividades realizadas no processo principal implementation, da Operational Phase; **What-Output**, ontologia operacional na linguagem definida.

No *running example*, a ontologia de referência é codificada na ontologia operacional, cujo fragmento é apresentado na Listagem A.5.

Listagem A.5 – Fragmento da ontologia operacional.

```
<?xml version="1.0"?>
<rdf:RDF xmlns="http://ooc-o#"
     xml:base="http://ooc-o"
     xmlns:sco="http://nemo.inf.ufes.br/projetos/oscin/sco#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:xml="http://www.w3.org/XML/1998/namespace"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
    <owl:Ontology rdf:about="http://ooc-o">
        <rdfs:comment rdf:datatype="http://www.w3.org/2000/01/rdf-schema#
    Literal">The Object-Oriented Code Ontology (OOC-O) aims to identify and
    represent the semantics of the entities present at compile time in object-
    oriented (OO) code.</rdfs:comment>
        <rdfs:comment rdf:datatype="http://www.w3.org/2000/01/rdf-schema#
    Literal">Camila Zacché de Aguiar</rdfs:comment>
        <rdfs:isDefinedBy rdf:datatype="http://www.w3.org/2000/01/rdf-schema#
    Literal">https://link.springer.com/chapter/10.1007/978-3-030-33223-5_3</
    rdfs:isDefinedBy>
        <rdfs:isDefinedBy rdf:datatype="http://www.w3.org/2000/01/rdf-schema#
    Literal">https://nemo.inf.ufes.br/wp-content/uploads/2019/04/OOC-
    O_Specification.pdf</rdfs:isDefinedBy>
        <rdfs:label rdf:datatype="http://www.w3.org/2000/01/rdf-schema#
    Literal">OOC-O: Operational Ontology on Object-Oriented Code</rdfs:label>
        <owl:versionInfo rdf:datatype="http://www.w3.org/2000/01/rdf-schema#
    Literal">1.0 - 31/05/2019</owl:versionInfo>
    </owl:Ontology>

    <owl:Class rdf:about="http://ooc-o#Class">
        <rdfs:subClassOf rdf:resource="http://sco#Derived_Type"/>
        <owl:disjointWith rdf:resource="http://ooc-o#Attribute"/>
        <owl:disjointWith rdf:resource="http://sco#Primitive_Type"/>
        <owl:disjointWith rdf:resource="http://ooc-o#Method"/>
        <rdfs:comment rdf:datatype="http://www.w3.org/2000/01/rdf-schema#
    Literal">An abstract-definition element in the OO programming language to
    express such definitions, that is, class is an abstract data type and a
    mechanism for defining an abstract data type in a program. Class describes
     the attributes of its objects as well as the methods they can execute.</
    rdfs:comment>
    </owl:Class>

    <owl:Class rdf:about="http://ooc-o#Member">
        <rdfs:comment rdf:datatype="http://www.w3.org/2000/01/rdf-schema#
    Literal">Element which makes up a class, defined as a variable or method
    .</rdfs:comment>
    </owl:Class>

    <owl:ObjectProperty rdf:about="http://ooc-o#composedByMember">
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#
    FunctionalProperty"/>
        <rdfs:domain rdf:resource="http://ooc-o#Class"/>
        <rdfs:range rdf:resource="http://ooc-o#Member"/>
    </owl:ObjectProperty>

</rdf:RDF>
```

A.6.5 [CONF-OPER] Control Operational Ontology

Atividade do processo de suporte configuration management que objetiva controlar as alterações, versões e entregas das informações registradas na ontologia operacional, descrita na atividade [DOCM-OPER] Document Operational Ontology.

Para isso, sugere-se o uso de metadados na ontologia para registrar a versão e sua respectiva data, armazenando o histórico de todas as versões para eventuais comparações ou restaurações de versões anteriores.

Esta atividade define: **Who**, *ontology engineer* como responsável; **What-Input**, ontologia operacional elaborada na atividade [DOCM-OPER] Document Operational Ontology; **What-Output**, controle da ontologia operacional.

No *running example*, a ontologia operacional de orientação a objetos conta com os metadados rdfs:isDefinedBy e owl:versionInfo para apresentar a autoria da ontologia e sua versão.

A.6.6 [EVAL-OPER] Evaluate Operational Ontology

Atividade do processo de suporte evaluation que objetiva avaliar se as informações registradas na ontologia operacional atendem às expectativas do *ontology owner* e estão de acordo com os requisitos da ontologia e da qualidade definidos na Conception Phase.

Para **validação** da ontologia, deve-se avaliar se a ontologia atende às situações do mundo real por meio da instanciação de seus conceitos como indivíduos da ontologia.

Para **verificação** da ontologia, deve-se avaliar se os requisitos funcionais e não funcionais levantados são respondidos por meio de consultas aplicadas sobre a ontologia instanciada. Para isso, sugere-se que as questões de competência levantadas na atividade [REQI-ELIC] Elicit Requirements sejam traduzidas em linguagem de consulta relativa à linguagem de codificação. Assim, a partir de um conjunto de casos de teste, é possível verificar o comportamento da ontologia operacional de acordo com as consultas executadas e as instâncias retornadas. Para ontologias construídas com linguagem de codificação OWL, sugere-se o uso de linguagem de consulta SPARQL.

A avaliação pode ser realizada como (FALBO, 2014): Teste de Subontologia, avaliação realizada de forma individual para cada subontologia codificada; Teste de Integração, avaliação realizada sobre as conexões entre as subontologias; Teste de Ontologia, avaliação realizada no contexto da ontologia completa, podendo incluir teste de *stress*, performance, inferência e outros.

Esta atividade define: **Who**, *ontology engineer* e *ontology owner* como responsáveis; **What-Input**, Documento de Especificação da Ontologia e a ontologia operacional elaborados nas Conception Phase e em [DOCM-OPER] Document Operational Ontology;

What-Output, avaliação da ontologia operacional.

No *running example*, o código-fonte (apresentado na Listagem A.1) é instanciado na ontologia como apresentado na Listagem A.6. Consultas SPARQL são elaboradas para responder as questões de competência, como a RF02 respondida na Listagem A.7.

Listagem A.6 – Fragmento da ontologia instanciada com código Java.

```
<owl:NamedIndividual rdf:about="http://ooc-o#project_Polygon">
    <rdf:type rdf:resource="http://ooc-o#Module"/>
    <visibleBy rdf:resource="http://ooc-o#visibility_public"/>
</owl:NamedIndividual>
<owl:NamedIndividual rdf:about="http://ooc-o#project_Polygon_perimeter">
    <rdf:type rdf:resource="http://ooc-o#Concrete_Method"/>
    <componentOfClass rdf:resource="http://ooc-o#project_Polygon"/>
    <visibleBy rdf:resource="http://ooc-o#visibility_public"/>
    <returnedBy rdf:resource="http://ooc-o#type_void"/>
</owl:NamedIndividual>
<owl:NamedIndividual rdf:about="http://ooc-o#project_Polygon_side">
    <rdf:type rdf:resource="http://ooc-o#Instance_Variable"/>
    <componentOfClass rdf:resource="http://ooc-o#project_Polygon"/>
    <visibleBy rdf:resource="http://ooc-o#visibility_private"/>
    <valuedBy rdf:resource="http://ooc-o#type_int"/>
</owl:NamedIndividual>
```

Listagem A.7 – Consulta SPARQL em resposta ao RF02.

```
SELECT ?class
WHERE {?class rdf:type ooc-o:Class. }
```

A.6.7 [PUBL-OPER] Publish Operational Ontology

Atividade do processo de suporte publication que objetiva disponibilizar a ontologia operacional, definida na atividade [DOCM-OPER] Document Operational Ontology.

Esta atividade define: **Who**, *ontology engineer* como responsável; **What-Input**, ontologia operacional elaborada na atividade [DOCM-OPER] Document Operational Ontology; **What-Output**, acesso à ontologia operacional.

No *running example*, a ontologia operacional de orientação a objetos está publicada no site do projeto, no endereço <https://nemo.inf.ufes.br/projetos/oscin>.