UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

CENTRO TECNOLÓGICO

COLEGIADO DO CURSO DE CIÊNCIA DA COMPUTAÇÃO

Daniel Fernandes da Silva

Tonto's Code Compiler: Um Gerador de código Baseado em OntoUML-JS

Vitória, ES

2023

Daniel Fernandes da Silva

Tonto's Code Compiler: Um Gerador de código Baseado em OntoUML-JS

Projeto de Graduação apresentado ao Colegiado do Curso de Ciência da Computação do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito para aprovação na Disciplina Projeto de Graduação II.

Universidade Federal do Espírito Santo – UFES

Centro Tecnológico

Colegiado do Curso de Ciência da Computação

Orientador: Monalessa Perini Barcellos

Coorientador: Paulo Sérgio dos Santos Júnior

Vitória, ES

2023

Daniel Fernandes da Silva
Tonto's Code Compiler: Um Gerador de código Baseado em OntoUML-JS/ Daniel Fernandes da Silva. – Vitória, ES, 2023
62 p. : il. (algumas color.) ; 30 cm.

Orientador: Monalessa Perini Barcellos
Coorientador: Paulo Sérgio dos Santos Júnior

Monografia (PG) – Universidade Federal do Espírito Santo – UFES
Centro Tecnológico
Colegiado do Curso de Ciência da Computação, 2023.

1. Desenvolvimento Orientado a Modelos. 2. Ontologias. 3. Engenharia de *Software* Contínua. 4. Geração de Código. I. Silva, Daniel Fernandes da. II. Universidade Federal do Espírito Santo. IV. Tonto's Code Compiler: Um Gerador de código Baseado em OntoUML-JS

CDU 02:141:005.7

Daniel Fernandes da Silva

Tonto's Code Compiler: Um Gerador de código Baseado em OntoUML-JS

Projeto de Graduação apresentado ao Colegiado do Curso de Ciência da Computação do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito para aprovação na Disciplina Projeto de Graduação II.

Trabalho aprovado. Vitória, ES, 18 de julho de 2023:

Monalessa Perini Barcellos
Orientador

Camila Zacché de Aguiar
Departamento de Informática
Universidade Federal do Espírito Santo

Fabiano Borges Ruy
Departamento de Informática
Instituto Federal do Espírito Santo

Vitória, ES
2023

Dedico este trabalho aos meus pais, pois graças a eles que eu cheguei até aqui, e graças a eles que chegarei a qualquer lugar daqui pra frente.

Agradecimentos

Primeiramente, agradeço à minha família, em especial os meus pais, por todo o apoio e carinho que me deram ao longo da minha vida.

Agradeço também a todos os meus amigos, companheiros fieis que eu fiz ao longo da vida e que espero que sigam comigo até o fim dela.

Dentre os meus amigos, agradeço em especial Gláucio, pelos anos de amizade sincera, e a minha namorada Ana, por todo o apoio e parceria incondicionais dos últimos quatro anos e meio.

Por fim, agradeço meus professores do curso de Ciência da Computação, e meus orientadores Monalessa e Paulo Sérgio, por todo o conhecimento que me passaram, permitindo que eu aplique esse conhecimento no mundo, e eventualmente passe-o adiante.

"Do you believe in gravity?"

Resumo

Num mercado em constante expensão como o de desenvolvimento de software, desafios novos são encontrados constantemente. Um desses desafios é a necessidade de flexibilidade durante o processo de desenvolvimento de software, para permitir que esse processo seja mais adaptativo e responsivo a *feedback*. A ascensão dos métodos ágeis são uma resposta a esse desafio, mas existe uma abordagem mais ampla chamada Engenharia de Software Contínua, uma abordagem que busca aprimorar a qualidade do software através de integração contínua, entrega contínua e feedback contínuo.

Por conta disso, foi desenvolvido, no Núcleo de Estudos em Modelagem Conceitual e Ontologias (NEMO), o *Immigrant*: uma abordagem que visa à integração de dados para apoiar desenvolvimento de software orientado a dados no contexto de ESC. Um dos componentes do *Immigrant* é o *The Band*, responsável pela integração de dados, que propõe o desenvolvimento de serviços web e bancos de dados a partir de ontologias em rede, chamados respectivamente de *Ontology-based Services* (OBS) e *Ontology-based Data Repositories* (OBDR).

Dado este contexto, com o objetivo de agilizar o processo de desenvolvimento desses OBDRs e OBSs, foi desenvolvido neste trabalho o TCC (*Tonto's Code Compiler*), uma aplicação, utilizando de conceitos do Desenvolvimento Orientado a Modelos, com o objetivo de, a partir de um modelo OntoUML, gerar o código correspondente aos OBDRs e OBSs desse modelo, encurtando o espaço temporal entre a confecção do modelo de dados e a codificação dos repositório de dados e serviço correspondentes.

Palavras-chaves: Desenvolvimento Orientado a Modelos; Ontologias; Engenharia de Software Contínua; OntoUML

Sumário

1 Introdução

Este capítulo apresenta a motivação, justificativa e objetivos do trabalho, bem como o método de desenvolvimento a ser adotado e o cronograma de atividades.

1.1 Motivação e Justificativa

Em um mercado em constante expansão como o de desenvolvimento de software, novos desafios surgem a todo momento, incluindo a necessidade de atender novos requisitos para melhor se adequar aos modelos de negócios, além da demanda por mais agilidade nas entregas das soluções contratadas. Ao enfatizar flexibilidade, eficiência e velocidade, as práticas ágeis levaram a uma mudança de paradigma na forma como o software é desenvolvido (OLSSON; ALAHYARI; BOSCH, 2012).

A ampla adoção de métodos ágeis (KURAPATI; MANYAM; PETERSEN, 2012) evidencia a necessidade de flexibilidade e rápida adaptação no atual mercado de desenvolvimento de software. Fitzgerald e Stol (2017), porém, argumentam em favor de uma abordagem mais ampla do que os métodos ágeis, a chamada Engenharia de software Contínua (do inglês, *Continuous software Engineering*) (ESC). ESC consiste em um conjunto de práticas e ferramentas que apoiam uma visão holística do desenvolvimento de software com o objetivo de torná-lo mais rápido, iterativo, integrado, contínuo e alinhado ao negócio (BARCELLOS, 2020).

Para que se possa realizar as atividades que caracterizam ESC são necessárias diversas aplicações[1]. Nesse sentido, organizações costumam usar diferentes aplicações para apoiar diferentes processos do desenvolvimento de software, abordando diferentes domínios da Engenharia de Software. Por exemplo, Azure Devops [2] pode ser utilizada para apoiar o planejamento das tarefas do projeto; o Github[3] pode auxiliar no versionamento de código, parte importante da atividade de integração contínua (CI); e o Clockify[4] e o Tempo[5] podem ser utilizados para gerenciar o tempo investido em cada tarefa. Os dados que são produzidos a partir do uso dessas aplicações e são nelas armazenados podem prover informações úteis para auxiliar na tomada de decisão, visando à melhoria do processo de desenvolvimento de software, do produto ou da organização que está desenvolvendo o software.

[1] Neste trabalho, os termos aplicação, ferramenta e sistema são usados como sinônimos.
[2] <https://azure.microsoft.com/en-us/products/devops>
[3] <https://www.github.com/>
[4] <https://clockify.me/pt/>
[5] <https://www.tempo.io/>

Contudo, essas diferentes aplicações apresentam dados heterogêneos, sendo necessário integrá-los para obter dados mais significativos para apoiar ações de melhoria ou tomada de decisão (FONSECA; BARCELLOS; de Almeida Falbo, 2017). Um dos motivos que fazem organizações falharem em utilizar dados armazenados em aplicações é a dificuldade de acessar, integrar, analisar e visualizar dados produzidos por aplicações heterogêneas. Em geral, cada aplicação implementa seus próprios modelos de dados e foca em aspectos específicos do processo de software, não se preocupando com partilhamento ou integração, gerando conflitos entre ferramentas (CALHAU; FALBO, 2010). Em especial no contexto de desenvolvimento ágil, o desafio é usar dados para apoiar o processo de desenvolvimento de maneira que isso não crie um gargalo de agilidade. Existe a necessidade de extrair informação útil dos dados de aplicativos e apresentá-los ao time no ambiente de desenvolvimento, de maneira efetiva e proativa (WACHE et al., 2001), sem adicionar mais esforço à equipe.

Uma fonte de dificuldade para integração de dados é a heterogeneidade semântica, que resulta em conflitos sempre que a mesma informação é dada interpretações distintas, algo que pode até passar despercebido (WACHE et al., 2001). Negligenciar esses "conflitos semânticos" pode levar a soluções mal integradas que falham em cumprir seu objetivo (ex.: apresenta informação errada ou inacurada) (POKRAEV, 2009). Para reduzir esses conflitos, a integração não pode ignorar problemas semânticos(CALHAU; FALBO, 2010; FONSECA; BARCELLOS; de Almeida Falbo, 2017).Um jeito de se alcançar isso é utilizar de ontologias para estabelecer uma conceitualização comum sobre o domínio das aplicações, permitindo integração e comunicação entre aplicações. Uma ontologia "*é uma especificação explícita e formal de uma conceitualização compartilhada*" (GRUBER, 1993). Logo, ela pode ser usada como a língua franca de diferentes aplicações, permitindo que elas se entendam (CALHAU; FALBO, 2010). De fato, na última década, ontologias vêm se tornando a maneira dominante de lidar com aspectos semânticos em iniciativas de integrações semânticas (NARDI; FALBO; ALMEIDA, 2013).

Este trabalho apresenta um conjunto de transformações, utilizando técnicas de *Model-Driven Development* (MDD)(OMG, 2014) que permitem criar repositórios de dados (*Ontology-Based Data Repository* - OBDR) e Serviços (*Ontology-Based Services* - (OBS) baseado em modelos de ontologias, descritos em OntoUML[6], para integração de dados que estão em aplicações. Ele foi desenvolvido no contexto de uma pesquisa de doutorado realizada no NEMO[7] (Núcleo de Estudos em Modelagem Conceitual e Ontologias) pelo doutorando Paulo Sérgio dos Santos Júnior, a qual aborda o problema da integração semântica no contexto de ESC. A pesquisa de doutorado visa à criação de *Immigrant*, uma abordagem de integração semântica baseada em redes de ontologias. Um dos componentes de *Immigrant* é *The Band* (SANTOS JÚNIOR, 2023), uma arquitetura de integração

[6] Linguagem de modelagem baseada em ontologias, descrita em 2.5.1

[7] <https://nemo.inf.ufes.br/>

de dados baseada em redes de ontologias que fornece dados integrados a partir de dados armazenados em aplicações usadas por uma organização para atender suas necessidades de informação.

Uma rede de ontologias é uma coleção de ontologias relacionadas através de uma variedade relacionamentos, como alinhamento e dependência (SUÁREZ-FIGUEROA et al., 2012). Em *The Band* ontologias são utilizadas para atribuir semântica aos dados das aplicações. Além disso, a arquitetura da rede de ontologias serve como base para a arquitetura da solução de integração e cada ontologia da rede é usada para a construção de serviços para lidar com os dados das aplicações e repositórios para armazená-los de acordo com uma semântica comum.

Este trabalho visa implementar uma biblioteca baseada no princípio de Desenvolvimento Orientado a Modelos para realizar transformações em modelos OntoUML para um conjunto de códigos em Java que ajudam no desenvolvimento de OBDRs e OBSs. Essas transformações são baseadas naquelas definidas por Guidoni, Almeida e Guizzardi (2021).

1.2 Objetivos

Este trabalho tem como objetivo criar uma biblioteca que implementa um conjunto de transformações para transformar modelos OntoUML em código Java, para a implementação de OBDRs e OBSs.

Os objetivos específicos são:

i. Desenvolver uma funcionalidade que consiga processar a representação textual de um modelo OntoUML, no formato OntoUML Schema;

ii. Desenvolver uma funcionalidade que realize transformações em cima do modelo de entrada, gerando um novo modelo;

iii. A partir desse novo modelo, gerar código Java que permita criar uma estrutura de banco de dados (OBDR) correspondente com o modelo de entrada, e código representando um serviço web (OBS) para criar, acessar e manipular dados nesse banco de dados;

iv. Disponibilizar o conjunto dessas funcionalidades como um pacote público, para que possa ser utilizado em outros projetos.

1.3 Metodologia de Desenvolvimento

Este trabalho será conduzido de acordo com as seguintes atividades:

i. Revisão da Literatura: consistiu na revisão bibliográfica sobre os temas que fundamentam este trabalho: OntoUML, Desenvolvimento Orientado a Modelos, transformações convertendo ontologias para bancos de dados. Para isso foi realizada a leitura de materiais (trechos de livros, teses, dissertações e artigos científicos) pertinentes ao assunto;

ii. Estudo de Tecnologias: Nesta atividade foram estudadas as tecnologias utilizadas neste trabalho. O projeto foi desenvolvido utilizando tecnologias que viabilizam: armazenamento de dados em bancos relacionais, geração programática de código e criação de OBDR e OBS;

iii. Design, Implementação e Testes: Nesta etapa foi definido o projeto de arquitetura da ferramenta, o projeto de seus componentes, e por fim a implementação e teste da ferramenta;

iv. Escrita da monografia: Consistiu na escrita deste documento.

1.4 Organização

Além desta introdução, esta monografia conta também com os seguintes capítulos:

- **Capítulo 2 - Fundamentação Teórica e Tecnologias Utilizadas**: Apresenta o conteúdo teórico necessário para o entendimento do trabalho, e quais tecnologias foram utilizadas em seu desenvolvimento;
- **Capítulo 3 - Tonto's Code Compiler**: Descreve o software produzido nesse trabalho, explicando partes do seu funcionamento e demonstrando exemplos das transformações feitas por ele;
- **Capítulo 4 - Conclusão**: Elabora sobre as conclusões alcançadas com esse trabalho, as dificuldades passadas e quais limitações tiveram de ser aceitas, apresentando também propostas de melhorias futuras para diminuir ou, idealmente, eliminar essas limitações.

2 Fundamentação Teórica e Tecnologias Utilizadas

2.1 Engenharia de Software Contínua

Engenharia de Software Contínua (ESC) consiste em um conjunto de práticas e aplicações que apoiam uma visão holística do desenvolvimento de software com o objetivo de torná-lo mais rápido, iterativo, integrado, contínuo e alinhado ao negócio (BARCELLOS, 2020).

ESC envolve práticas e aplicações que visam estabelecer um fluxo de ponta a ponta entre a demanda do cliente e a entrega rápida de um produto ou serviço. O quadro geral pelo qual isso pode ser alcançado vai além dos princípios ágeis e traz à tona um conjunto holístico de atividades contínuas (FITZGERALD; STOL, 2017). De acordo com Johanssen et al. (2019), em ESC, os clientes são proativos, e os usuários e outras partes interessadas estão envolvidas no processo, aprendendo com o uso de dados e feedback. O planejamento é contínuo, assim como engenharia de requisitos, que se concentra em recursos, arquitetura modularizada e design, e rápida realização de mudanças.

Existem vários trabalhos que dão uma visão geral de ESC ((OLSSON; ALAHYARI; BOSCH, 2012), (FITZGERALD; STOL, 2017), (JOHANSSEN et al., 2019) (BARCELLOS, 2020)). O modelo *Stairway to Heaven* (StH) (OLSSON; ALAHYARI; BOSCH, 2012) define descreve cinco estágios pelos quais organizações passam na evolução de um desenvolvimento de software tradicional para um desenvolvimento de ESC e orientado a dados:

i. Desenvolvimento Tradicional (do inglês, *Traditional Development*): o primeiro estágio consiste no desenvolvimento tradicional. Trata-se de uma abordagem para o desenvolvimento de software caracterizado por ciclos de desenvolvimento lentos, fases sequenciais e uma fase de planejamento rigoroso onde requisitos são estabelecidos antecipadamente. Normalmente, a abordagem é caracterizada por uma interação em cascata entre gerenciamento de produto, desenvolvimento, teste do sistema e o cliente. Projetos que adotam essa abordagem sofrem de longos ciclos de feedback e dificuldades para integrar o feedback do cliente no processo de desenvolvimento do produto.

ii. Organização Ágil (do inglês, *Agile Organization*): para superar as limitações impostas pelo modelo tradicional, emprega-se o desenvolvimento ágil. As práticas ágeis são caracterizadas por pequenas equipes multifuncionais de desenvolvimento, *sprints* curtos de desenvolvimento resultando em software funcional e planejamento contínuo

no qual o cliente está envolvido para permitir feedback do cliente. Em organizações ágeis, no entanto, o gerenciamento de produtos e verificação do sistema ainda funcionam de acordo com a abordagem tradicional.

iii. Integração Contínua (do inglês, *Continuous Integration*): neste estágio a empresa consegue estabelecer práticas que permitem integração frequente de trabalho, *builds* diárias (compilação dos artefatos de software) e *commit* de alterações (elaboração por escrito das mudanças feitas em uma aplicação), por exemplo, *builds* automatizados e testes automatizados. Neste ponto, tanto a equipe de desenvolvimento de produtos quanto a equipe de teste e verificação trabalham de acordo com práticas ágeis com ciclos curtos de feedback e integração do trabalho.;

iv. Entrega Contínua (do inglês, *Continuous Deployment*): a entrega contínua implica que se envie continuamente as alterações feitas em código para o produto ao invés de planejar lançamentos de grandes funcionalidades. Isso permite o feedback contínuo, a capacidade de aprender com os dados de uso e elimina o trabalho que não produz valor para o cliente. Neste ponto, P&D (Pesquisa e Desenvolvimento), gerenciamento de produtos e clientes estão todos envolvidos em um ciclo de desenvolvimento rápido e ágil no qual o tempo de resposta é menor se comparado ao tradicional.

v. P&D como um Sistema de Inovação (do inglês, *R&D as an Innovation System*): no estágio final, a organização coleta dados de seus clientes e usa a base de clientes instalada para executar experimentos frequentes de recursos para apoiar o desenvolvimento de software orientado a dados do cliente.

2.2 Ontologias

Uma ontologia "*é uma especificação explícita e formal de uma conceitualização compartilhada*" (GRUBER, 1993). Aqui, "conceitualização" se refere a um modelo abstrato de algum fenômeno do mundo real que identifique os conceitos relevantes desse fenômeno; "explícito" significa que os tipos de conceitos usados e restrições impostas no seu uso são definidas explicitamente; "formal" se refere ao fato que uma ontologia deve ser interpretável por máquinas; e "compartilhada" reflete que ontologias devem capturar o conhecimento baseado em consenso de uma comunidade (STUDER; BENJAMINS; FENSEL, 1998).

De acordo com Scherp et al. (2011), ontologias podem ser organizadas em uma arquitetura com três camadas, divididas em ontologias fundacionais, ontologias de base e ontologias de domínio. Ontologias fundacionais almejam modelar o conceitos mais básicos e gerais que compõem o mundo (como objetos, eventos e participações). Eles são genéricos para qualquer área e altamente reutilizáveis em diferentes cenários de modelagem. Ontologias de base provêm um refinamento sobre ontologias fundacionais, adicionando

relações e conceitos detalhados de uma área específica (como serviços, processos, estruturas organizacionais) que ainda assim abrangem vários domínios. Ontologias de domínio focam em um domínio particular da realidade, tal como uma ontologia específica de domínio médico descrevendo a anatomia do corpo humano. Ontologias de domínio podem usar de, ou serem baseadas em, ontologias fundacionais e ontologias de base, especializando seus conceitos.

Outra importante distinção separa ontologias como modelos conceituais, chamadas de ontologias de referência, de ontologias como artefatos computacionais, chamadas de ontologias operacionais (GUIZZARDI, 2007). Uma ontologia de referência é construída com o objetivo de fazer a melhor descrição possível do domínio na realidade, representando um modelo consensual em uma comunidade, independente de sua propriedades computacionais. Os usuários tendo concordado com a conceitualização, versões operacionais (ontologias legíveis por máquina) de uma ontologia de referência podem ser implementadas. Diferente de ontologias de referência, ontologias operacionais são desenvolvidas com foco em garantir propriedades computacionais desejáveis (FALBO, 2014).

2.2.1 UFO - Unified Foundation Ontology

A Unified Foundational Ontology (UFO) foi desenvolvida baseada em teorias de Ontologias Formais, Lógica Filosófica, Filosofia da Linguagem, Linguística, e Psicologia Cognitiva (GUIZZARDI, 2005). UFO é dividida em três partes:

- **UFO-A:** uma ontologia de *Endurants* (objetos) (GUIZZARDI, 2005; GUIZZARDI et al., 2015; GUIZZARDI et al., 2022);
- **UFO-B:** uma ontologia de *Perdurants* (eventos) (GUIZZARDI et al., 2013; ALMEIDA; FALBO; GUIZZARDI, 2019);
- e **UFO-C:** uma ontologia de entidades sociais (GUIZZARDI; FALBO; GUIZZARDI, 2008; GUIZZARDI; GUIZZARDI, 2010).

A seguir serão introduzidos os conceitos da UFO pertinentes a esse trabalho, denotados em itálico. Detalhes mais aprofundados sobre UFO e seus usos podem ser vistos em (GUIZZARDI, 2005), (GUIZZARDI; FALBO; GUIZZARDI, 2008), (GUIZZARDI et al., 2013) e (GUIZZARDI et al., 2022).

UFO apresenta uma distinção fundamental entre as categorias ontológicas de *Types* e *Individuals*. *Types* são padrões de características, replicáveis por diferentes entidades (GUIZZARDI et al., 2022). *Individuals* são entidades particulares, que não podem ser instanciadas.

UFO-A é uma ontologia de *Endurants*. *Endurants* (e.g., Mick Jagger, a Lua, a febre

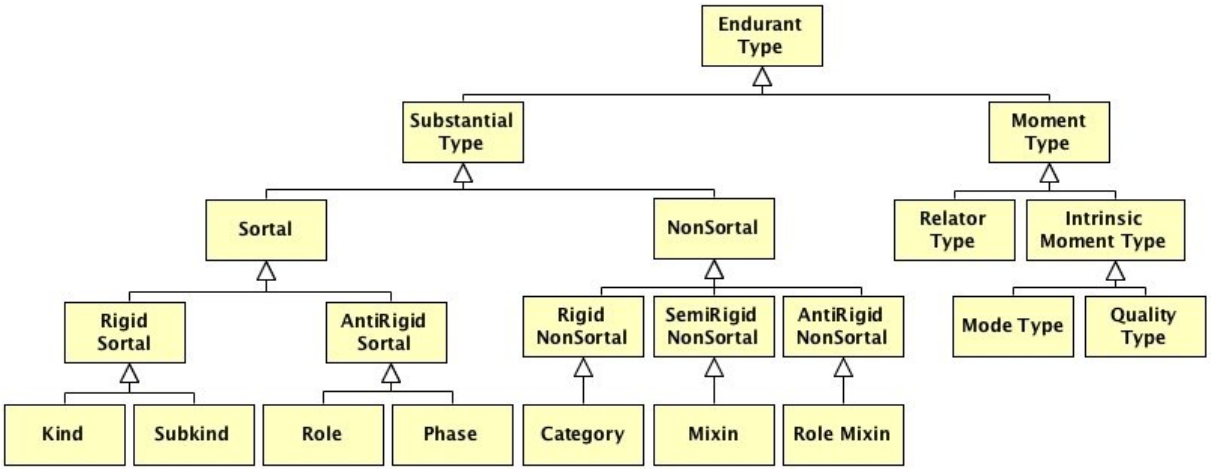

Figura 1 – Taxonomia dos *Endurant Types* na UFO (GUIZZARDI et al., 2018)

de John) são indivíduos que existem no tempo com todas as suas partes (GUIZZARDI et al., 2022). Eles tem propriedades essenciais e acidentais e, portanto, podem mudar qualitativamente mantendo sua identidade – permanecendo o mesmo indivíduo. Os tipos de mudanças que um *endurant* pode ser submetido e ainda assim manter sua identidade são definidas pelo *Kind* que o instancia (GUIZZARDI et al., 2022).

UFO distingue *Endurant Types* em *Substantial Types* e *Moment Types*. Naturalmente, esses são os tipos cujas as instâncias são *Substantials* e *Moments* (GUIZZARDI, 2005), respectivamente. *Substantials* são objetos existencialmente independentes, como a Lua, um carro, um cachorro. *Moments*, em contrapartida, são indivíduos existencialmente dependentes, como: (1) a capacidade de Sofia de falar italiano (que depende de Sofia); e (2) o casamento entre John e Yoko (que depende tanto de John quanto de Yoko) (GUIZZARDI et al., 2018). *Moments* do tipo (1) são chamados *Modes*; e do tipo (2) são chamados *Relators*. *Relators* são indivíduos com o poder de conectar entidades, por exemplo, um *relator* de matrícula conecta um indivíduo na função de estudante com uma instituição de ensino. Cada instância de um *relator type* é existencialmente dependente de pelo menos duas entidades distintas. Além disso, existe um terceiro tipo de *moment* chamado *quality*. *Qualities* são *moments* individuais que podem ser mapeados para algum espaço qualitativo, por exemplo a cor de uma maçã que pode mudar de verde para vermelha, mantendo sua identidade (GUIZZARDI, 2005) (GUIZZARDI et al., 2018).

Dentro da hierarquia de *Substantial Type*, tipos *Sortal* são os que ou provêm ou carregam algum princípio uniforme de identidade para suas instâncias. Um princípio de identidade de um *sortal* S torna explícito as propriedades que duas instâncias de S não podem ter em comum, pois tais propriedades identificam unicamente as instâncias de S. Em particular, ele também informa por quais mudanças um indivíduo consegue passar sem alterar sua identidade – permanecendo o mesmo indivíduo (GUIZZARDI et al., 2018). Dentro da categoria dos *sortals*, podemos distinguí-los entre *rigid* e *anti-rigid*. Um tipo

Rigid é um que classifica suas instâncias necessariamente (no sentido modal), ou seja, as instâncias que são desse tipo não podem deixar de ser desse tipo, a menos que deixem de existir. Tipos *Anti-Rigid*, em contrapartida, caracterizam um tipo cujas instâncias podem deixar de ser ou voltar a ser do tipo sem alterarem sua identidade (GUIZZARDI, 2005). Por exemplo, vamos comparar o tipo *rigid* Pessoa e os tipos *anti-rigid* Estudante ou Marido. Um indivíduo João nunca deixa de ser uma instanciação de Pessoa, mas ele pode vir a ser ou deixar de ser um Estudante ou um Marido, dependendo de seu estado acadêmico ou marital, respectivamente (GUIZZARDI et al., 2018).

Kinds são tipos *sortal rigid* que provêm um princípio uniforme de identidade para suas instâncias (ex.: Pessoa). *Subkinds* são tipos *sortal rigid* que carregam o princípio de identidade fornecido por um único *kind* (Ex.: o *kind* Pessoa pode ter os *subkinds* Homem e Mulher que carregam o princípio de identidade provido por Pessoa). (GUIZZARDI et al., 2018) Quanto aos tipos *anti-rigid sortal*, temos a distinção entre *roles* e *phases*. *Phases* são tipos relacionalmente independentes definidos por condições de instanciação contigentes e intrísecas. (GUIZZARDI, 2005). Por exemplo, uma Criança é uma *phase* de Pessoa, instanciada pelas instâncias de pessoas que tem a propriedade intrínseca de terem menos que 12 anos de idade. *Roles*, por outro lado, são tipos relacionalmente dependentes, capturando propriedades relacionais partilhadas por instâncias de um dado *kind*: entidades têm papeis (roles) quando relacionadas com outras entidades por meio de relações materiais (por exemplo, alguém estando no papel de Marido quando conectado via relação material de "estar casado com" com alguém no papel de Esposa) (GUIZZARDI et al., 2018). Como cada indivíduo do universo em questão deve obedecer exatamente um princípio de identidade que, por sua vez, é provido por um *kind*, cada hierarquia *sortal* tem um *kind* único no topo, também chamado de *ultimate sortal* (GUIZZARDI, 2005).

Tipos *Non-Sortal* (também chamados de tipos dispersivos (GUIZZARDI, 2005)) agregam propriedades que são comuns para diferentes *sortals* – que acabarão por classificar entidades de diferentes *kinds.Non-sortals* não provêm um princípio uniforme de identidade para sua instâncias; Ao invés disso, eles classificam coisas que partilham propriedades em comum, mas que obedecem principios de identidade diferentes. Mobília é um exemplo de *non-sortal* que agrega as propriedades de Mesa, Cadeira, e outros. Outros exemplos seriam Obra de Arte (incluindo pinturas, estátuas, composições musicais), Objeto Assegurado (incluindo Obras de Arte, Imóveis, Carros, Partes do Corpo) e Entidades Legais (Pessoas, Organizações, Contratos). As meta-propriedades de rigidez ou anti-rigidez podem ser aplicadas para distinguir diferentes tipos de *non-sortal* (GUIZZARDI et al., 2018). Uma *Category* representa um *non-sortal rigid* e relacionalmente independente – um tipo dispersivo que agrega propriedades essenciais que sejam comuns em diferentes *rigid sortals* (GUIZZARDI, 2005) (Ex.: Objeto Físico agrega propriedades essenciais sobre Mesa, Copo e Carro). Um *Role Mixin* representa um *non-sortal anti-rigid* e relacionalmente dependente – um tipo dispersivo que agrega propriedades que sejam comuns em diferentes *roles*

(GUIZZARDI, 2005) (e.g., o tipo Consumidor que agrega propriedades de consumidores individuais ou corporativos). Apesar de não estar no conjunto original dos *Endurant Types* da UFO, a noção de um *Phase Mixin* emergiu, com o passar dos anos, como uma noção útil que vários autores davam falta (CARVALHO et al., 2017). Um *Phase Mixin* representa um *non-sortal anti-rigid* relacionalmente independente – um tipo dispersivo que agrega propriedades que sejam comuns em diferentes *phases* (Ex.: o tipo Agente Ativo que agrega as propriedades de Pessoa Viva e de Organização Ativa). Finalmente, um *Mixin* é um *non-sortal* que representa propriedades partilhadas por coisas de diferentes *kinds* mas que são essenciais em algumas instâncias, e acidentais em outras (Ex.: o tipo Item Assegurado pode ser essencial em Carro, supondo que todos os carros devem ter seguro, mas serem acidental em Casa, nem toda casa sendo obrigatoriamente assegurada) (GUIZZARDI et al., 2018).

2.3 *Immigrant*: Uma abordagem Baseada em Redes de Ontologias para Integração de Dados

Immigrant é uma abordagem baseada em rede de ontologia para integrar dados de aplicações para apoiar o desenvolvimento de software orientado a dados no contexto de ESC. A abordagem considera, em uma perspetiva *top-down*, as necessidades de informações da organização a serem obtidas a partir do estado atual da organização e das ações de melhoria e implementação de práticas de ESC necessárias. Para isso, *Immigrant* provê dois componentes: *California* (SANTOS; BARCELLOS; CALHAU, 2020), um processo baseado na teoria de sistemas para implementação de práticas de ESC, e *Zeppelin* (SANTOS JÚNIOR et al., 2021), um instrumento de diagnóstico de práticas de ESC.

Em uma perspectiva *bottom-up*, a abordagem considera os dados disponíveis nas aplicações utilizadas pela organização para apoiar seu processo de desenvolvimento. Considerando as necessidades de informação da organização e os dados disponíveis, a abordagem provê o componente *The Band* (SANTOS JÚNIOR, 2023), que utiliza ontologias de *Continuum* como interlíngua para integrar dados de diferentes aplicações e apresentar dados integrados em *dashboards* para permitir o monitoramento e *insights* visando à tomada de decisão orientada a dados. Assim, *The Band* permite que *Immigrant* aborde questões semânticas envolvidas na integração de dados para fornecer dados integrados e significativos considerando as necessidades de informação da organização. A Figura 2 apresenta uma visão geral de *Immigrant*.

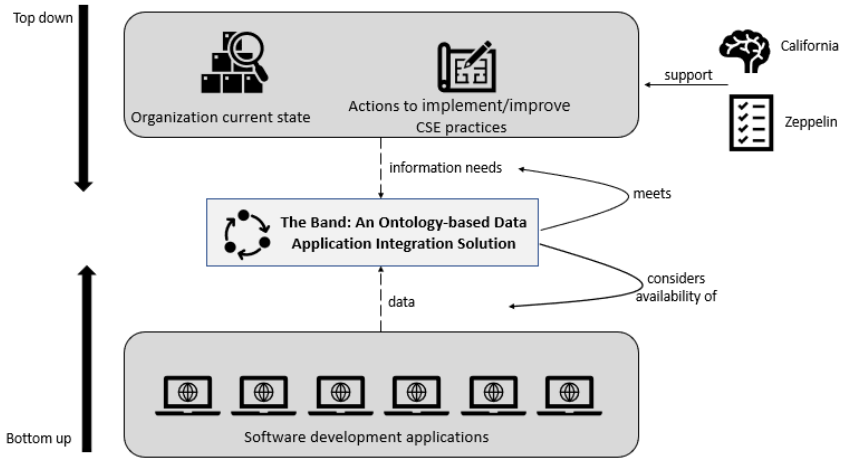

Figura 2 – Visão Geral de *Immigrant* (SANTOS JÚNIOR, 2023).

2.3.1 *The Band*

The Band (SANTOS JÚNIOR, 2023) é uma solução de software de integração de dados baseada em redes de ontologias e *Federated Information System*[1] (FIS). A rede de ontologias fornece as camadas de abstração a serem consideradas (ou seja, *fundamental, core e domain*), os subdomínios e respectivos conceitos, relacionamentos e axiomas a serem abordados. A arquitetura FIS, por sua vez, fornece meios para criar sistemas que compartilham, trocam e combinam dados e uma interface para um cliente acessar dados na federação de sistemas. Em *The Band*, cada ontologia em rede é usada como base para um *ontology-based service* (OBS) que é um sistema da federação *The Band* e possui seu próprio repositório, denominado *ontology-based data repository* (OBDR) (SANTOS JÚNIOR, 2023). Portanto, cada OBS captura, armazena e compartilha dados relacionados à porção de domínio endereçada pela referida ontologia em rede.

Ao criar OBSs baseados em ontologias em rede, é possível observar os relacionamentos entre as ontologias em rede e identificar quais dados precisam ser trocados entre diferentes OBSs. Ao organizar OBSs em um FIS, os critérios FIS relevantes podem ser considerados para contribuir para a definição da arquitetura da solução. Por exemplo, ao aplicar o critério de transparência, *The Band* deve permitir que um cliente pesquise dados sem saber onde eles estão armazenados e usando uma linguagem de consulta baseada em uma conceituação comum (ou seja, conceitos da ON). Ao aplicar o critério de autonomia, os OBSs devem ser capazes de trabalhar independentemente de outros OBSs e lidar com todos os dados necessários para tratar o domínio de interesse. A Figura 4 mostra o fragmento de um OBDR gerado usando a *Scrum Reference Ontology* (SRO) como base.

[1] Um Sistema de Informação Federado (FIS) é um conjunto de componentes de sistemas de informação distintos e autônomos, os participantes de uma federação. Os participantes em primeiro lugar operam de forma independente, mas possivelmente abriram mão de alguma autonomia para participar de uma federação. (BUSSE et al., 1999)

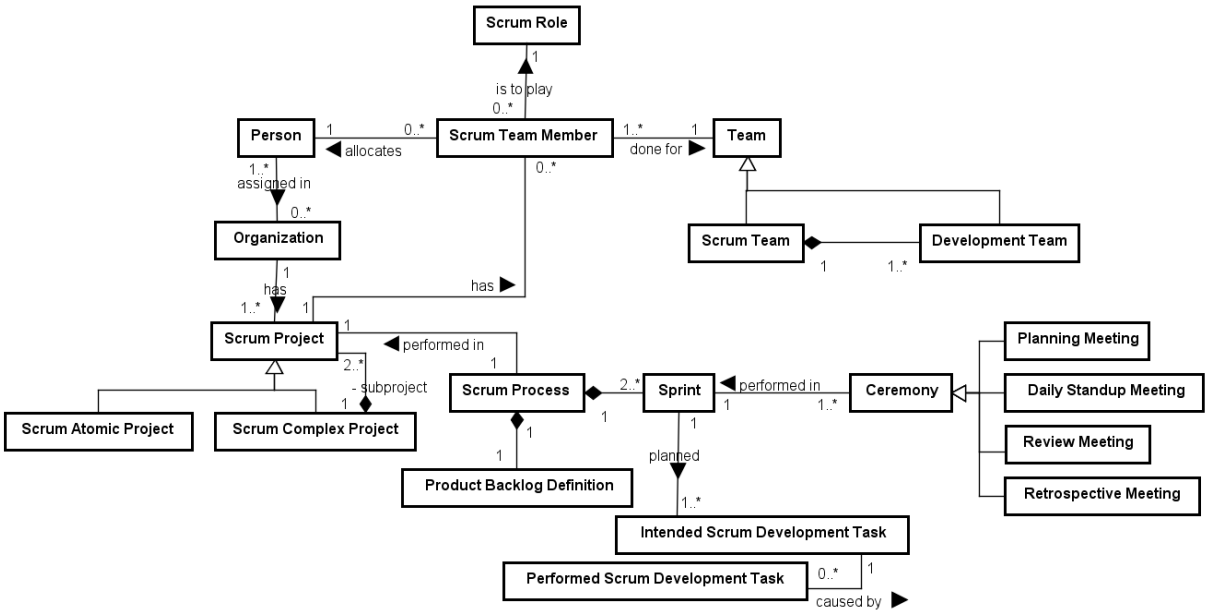

Figura 3 – Fragmento do modelo de SRO

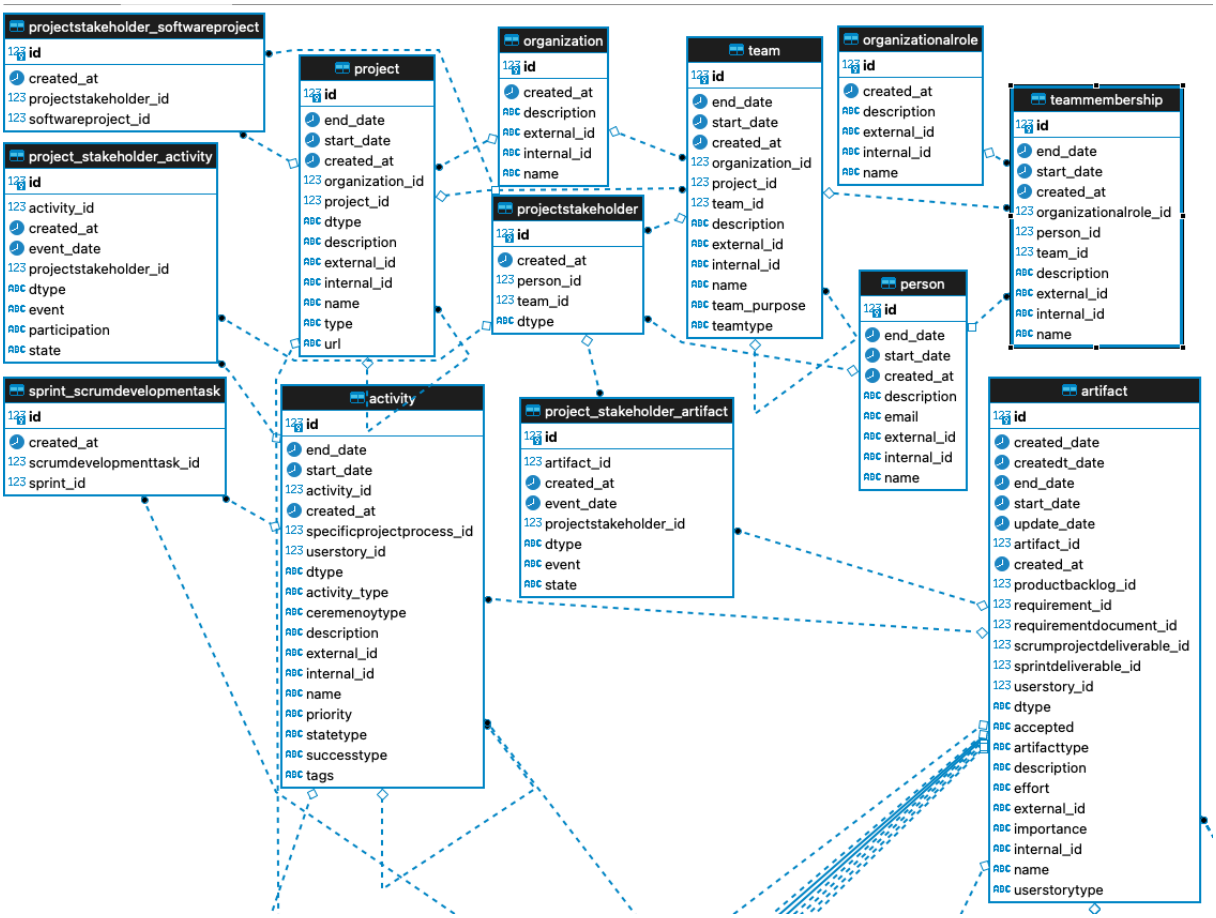

Figura 4 – Fragmento de diagrama Entidade Relacionamento da OBDR baseada na SRO.

2.3.2 *Ontology-Based Data Repository* e *Ontology-Based Service*

Um Repositório de Dados Baseado em Ontologia (do inglês *Ontology-Based Data Repository*, ou OBDR) é um repositório de dados baseado modelo de informação proveniente de uma ontologia e, consequentemente, representa conceitos dessa ontologia. Ele pode ser implementado de diferentes maneiras, como um banco de dados relacional ou um banco de dados orientado a grafos (SANTOS JÚNIOR, 2023), como exemplificado da Figura 4.

Uma vez que o OBDR está implementado, pode ser gerado gerado o Serviço Baseado em Ontologia (do inglês *Ontology-Based Service*, ou OBS) a partir do modelo de informação da ontologia (SANTOS JÚNIOR, 2023). Esses serviços podem então ser usados para manusear (criar, ler, alterar) as informações do OBDR respectivo.

Para a geração de OBDRs e OBSs de uma ontologia em rede inteira, é necessário que sejam criados os OBDRs e OBSs de cada ontologia da rede.

2.4 Desenvolvimento Orientado a Modelos

Muitas vezes, há uma grande distância entre a etapa de elicitação de requisitos e o desenvolvimento (codificação) de sistemas computacionais (ALMEIDA; ECK; IACOB, 2006; ALMEIDA, 2006), tornando, assim, o desenvolvimento de um aplicativo computacional demorado e custoso. O Desenvolvimento Orientado a Modelos é uma das possíveis soluções para aumentar a velocidade e diminuir o custo do desenvolvimento de uma aplicação computacional.

O Desenvolvimento Orientado a Modelos (do inglês, *Model-Driven Development*) é uma abordagem de desenvolvimento de sistemas computacionais, que aumenta o poder da utilização de modelos para esta finalidade (OMG, 2014). Tal abordagem proporciona um meio de utilizar modelos no curso do entendimento, projeto, construção, implantação, operação, manutenção e modificação de sistemas computacionais (OMG, 2014). Os modelos são expressos em linguagens de modelagem que possuem sua sintaxe abstrata descritas em modelos conhecidos como metamodelos (ALMEIDA et al., 2006) (JUNIOR, 2009).

2.4.1 Processo de Desenvolvimento Orientado a Modelos

O Processo de Desenvolvimento Orientado a Modelos é caracterizado como uma sequência de etapas de projeto que possuem como finalidade o desenvolvimento de um sistema computacional. Para cada etapa de projeto, são executadas atividades de projeto. Cada atividade de projeto contém uma atividade de transformação (do inglês, *transformation activity*) e uma atividade de avaliação (do inglês, *assessment activities*) (ALMEIDA; ECK; IACOB, 2006).

Uma atividade de transformação é uma atividade de projeto genérica que é respon-

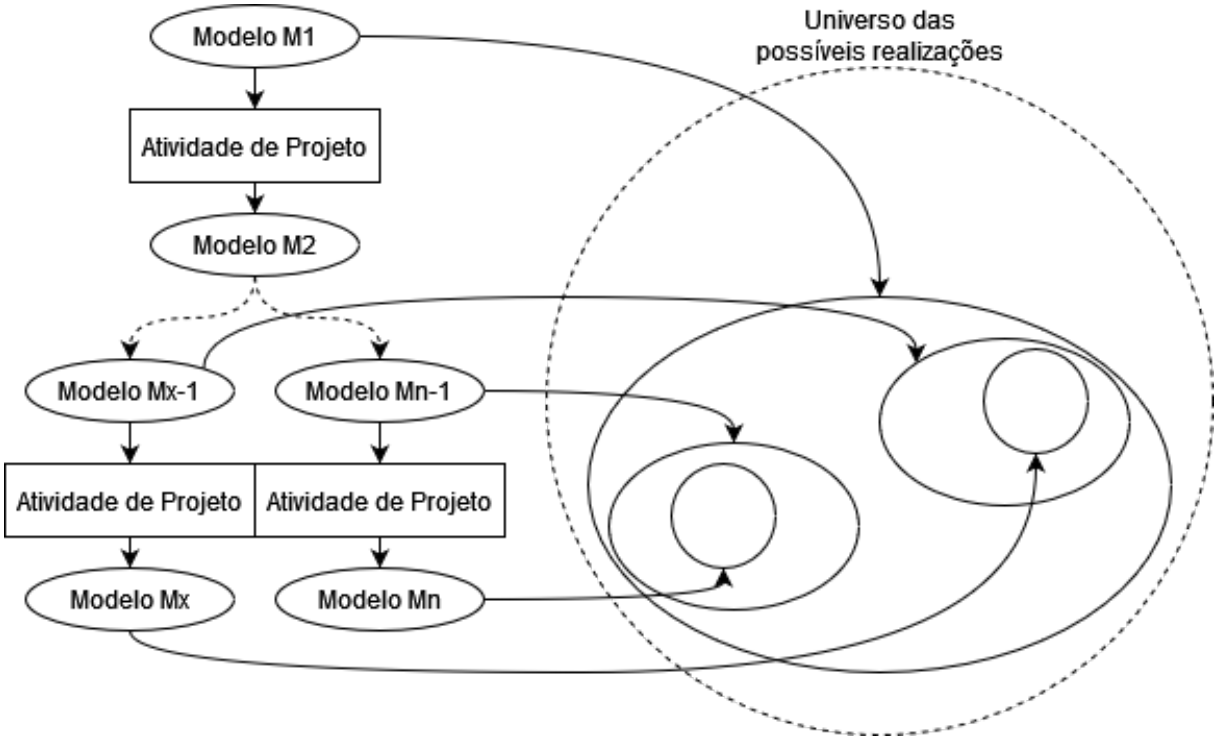

Figura 5 – Redução do espaço de projeto ao longo de múltiplas transformações

sável por produzir um modelo destino (do inglês, *target model*) tendo com base um modelo origem (do inglês, *source design*) e um ou mais requisitos que devem ser satisfeitos. Uma atividade de avaliação é uma atividade de projeto genérica que possui como compromisso avaliar o modelo destino produzido pela atividade de transformação. A noção de modelo origem e destino depende da etapa do projeto (ALMEIDA et al., 2006). Em princípio, as atividades de avaliação devem incluir a avaliação de conformidade (do inglês, *conformance assessment*) para avaliar se o modelo destino está em conformidade com o modelo origem (ALMEIDA, 2006). A teoria sobre o relacionamento entre conformidade e transformações é apresentada na seção 2.4.3.

Durante o processo de desenvolvimento do sistema computacional, as atividades de transformação incorporam diversas decisões de projetos que adicionam características que eventualmente estarão associadas ao sistema em realização. Diferentes decisões de projeto guiam para diferentes alternativas de realização. A redução do espaço de realização imposta pelas sucessivas decisões de projeto é apresentada na Figura 5.

As decisões de projeto aplicadas em cada etapa do projeto devem reunir dois requisitos para que o processo de construção do sistema computacional tenha sucesso:

i. as decisões devem contribuir para satisfazer requisitos do sistema que ainda não foram cumpridos, e

ii. as decisões de projeto devem preservar as características que estão presentes no modelo origem, isto é, o modelo destino deve estar em conformidade com o modelo

origem (ALMEIDA; ECK; IACOB, 2006).

As decisões de projeto devem eventualmente guiar o projeto de desenvolvimento do sistema computacional, a fim de definir todas as características que são necessárias para o sistema em realização. A plataforma em que o sistema computacional irá ser desenvolvido, particularmente, define como as decisões de projeto podem ser realizadas. De forma similar, as decisões de projeto definem as possíveis plataformas em que o projeto pode ser realizado (ALMEIDA; ECK; IACOB, 2006).

2.4.2 Plataformas de Realização e Modelos Independentes de Plataforma

Uma plataforma de realização provê ao projetista um conjunto de construtores reusáveis. Estes construtores permitem ao projetista desenvolver um sistema computacional sem que o mesmo tenha a necessidade de se preocupar com a implementação dos construtores (ALMEIDA; ECK; IACOB, 2006). Por prover um conjunto de construtores reusáveis, a plataforma de realização impõe um conjunto de regras no projeto do sistema computacional. Estas regras impostas pela plataforma de realização devem ser incorporadas nas etapas de projeto do sistema computacional em realização (ALMEIDA; ECK; IACOB, 2006).

Quando os sistemas computacionais são descritos de forma a evitar dependências dos construtores de uma plataforma específica, os modelos utilizados são chamados de modelos independentes de plataforma (do inglês, *platform-independent models (PIMs)*) (OMG, 2014). A independência de plataforma evita poluir modelos independentes de plataforma com interesses relacionados a restrições de uma plataforma específica, facilitando o reuso dos mesmos modelos para a construção de um sistema em diferentes plataformas (ALMEIDA; ECK; IACOB, 2006).

2.4.3 Relacionamento entre Conformidade e Transformações

As regras de conformidade determinam o mínimo que deve ser preservado na etapa de projeto, ou seja, determinam o máximo de liberdade para a criação do modelo destino sem perder as decisões de projeto presentes no modelo origem. As especificações de transformação determinam o máximo que pode ser escrito na etapa de projeto, ou seja, determinam o resultado em um específico modelo destino (mínimo de libertada) (ALMEIDA; ECK; IACOB, 2006). Isto é apresentado na Figura 6.

As regras de conformidade determinam implicitamente o conjunto de modelos destino que estão em conformidade com um modelo origem. Estas regras são “faróis” e um modelo origem “ilumina” uma área alvo no nível do modelo destino. Em contraste, uma especificação de transformação (não-parametrizada ou determinística) é descrita como uma seta do modelo origem para o modelo destino (ALMEIDA; ECK; IACOB, 2006).

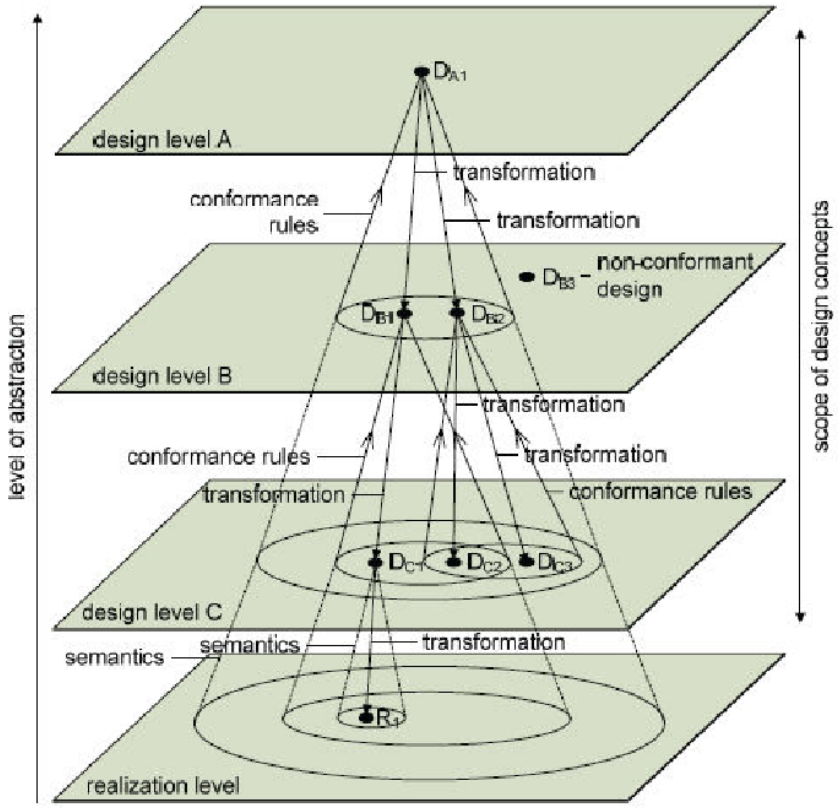

Figura 6 – Relação entre conformidades e transformações (ALMEIDA; ECK; IACOB, 2006)

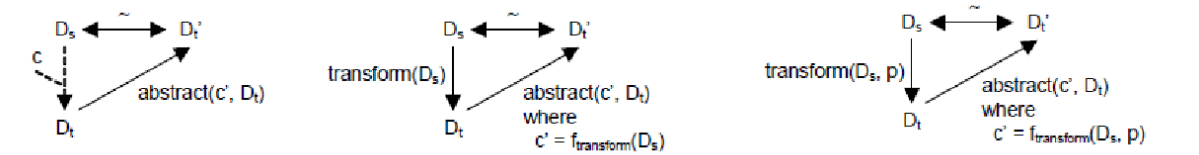

Figura 7 – Transformação e Conformidade

A avaliação da conformidade pode ser realizada como conforme demonstrado na Figura 7. A Figura 7 ilustra o que é feito durante a construção de um modelo destino *Dt* a partir de um modelo origem *Ds*, com a inserção de características *c* (consequência das decisões de projeto que foram tomadas). Para considerar a conformidade de um modelo destino com relação a um modelo origem, devemos abstrair as características do modelo destino resultando assim em uma abstração modelo destino *Dt'*. Então, devemos checar se o modelo destino sem as características (*Dt'*) é equivalente ($\sim$) ao modelo origem (ALMEIDA; ECK; IACOB, 2006).

Para abstrair as características *c* que foram inseridas, é necessário conhecer as decisões de projeto que foram tomadas no passo de projeto que levou *Ds* a *Dt*. Se, por exemplo, durante a construção do modelo destino, uma interação foi inserida, nós devemos saber que a interação deve ser abstraída durante o processo de validação da conformidade

(ALMEIDA; ECK; IACOB, 2006).

Uma transformação do modelo origem *Ds* para o modelo destino *Dt* é um caso especial de construção de um modelo destino. Neste caso, o modelo destino e as características inseridas são completamente determinados pelas regras de transformação e pelo modelo origem *Ds*. Consequentemente, as características *c'* que devemos abstrair durante a avaliação de conformidade podem ser automaticamente determinadas por uma função $f_{transform}$ aplicada sobre o modelo origem *Ds*. Esta função deve ser definida como um complemento da transformação. Se tal função existe, é possível validar automaticamente a conformidade do modelo destino em relação ao modelo origem. A Figura 7 (ii) ilustra este caso.

A Figura 7 (iii) ilustra o caso das transformações parametrizadas. Para as transformações parametrizadas, as decisões de projeto que foram tomadas nas etapas de projeto são determinadas pelas regras de transformação, o modelo origem e os valores dos parâmetros. Consequentemente, *c'* pode ser determinada por uma função $f_{transform}$ no modelo origem *Ds* e valores dos parâmetros p (ALMEIDA; ECK; IACOB, 2006).

Uma abordagem de Desenvolvimento Orientado a Modelos que preserva conformidade em suas transformações sempre produz modelos de destino que estão em conformidade com os modelos de origem. O benefício deste tipo de abordagem é que as atividades de validação de conformidade não tem que ser executadas manualmente para cada aplicação de transformação. Isto é particularmente benéfico para as abordagens de projeto iterativas em que as transformações são re-aplicadas com frequência para lidar com as mudanças nos modelos de origem (ALMEIDA; ECK; IACOB, 2006).

A fim de demonstrar que uma transformação de modelos garante a conformidade, é necessário demonstrar que para cada modelo origem *Ds*:

$$abstract(transform(D_s), f_{transform}(D_s)) \sim D_s$$

Para as transformações parametrizadas, é necessário demonstrar que para cada modelo origem *Ds* e cada valor admissível do parâmetro de *Ds*:

$$abstract(transform(D_s), f_{transform}(D_s, p)) \sim D_s$$

Nesse trabalho, as principais transformações feitas sobre modelos serão as operações chamadas de *Flattening* e *Lifting*, especificadas em (GUIDONI; ALMEIDA; GUIZZARDI, 2021). A seguir, essas serão apresentadas em detalhe, descrevendo suas ações e quais suas consequências para o modelo.

2.4.4 *Flattening* e *Lifting*

Na operação de *flattening*, mostrada na linha superior da Tabela 8, uma superclasse (mostrada em cinza) será removida do modelo. Os atributos dessa classe serão migrados

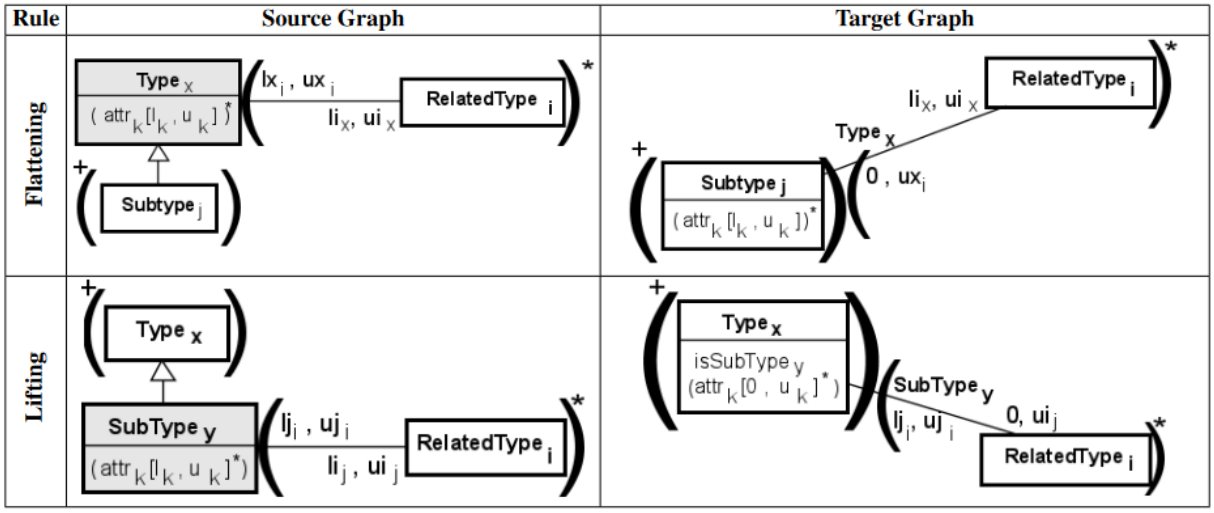

Figura 8 – Padrão das transformações (GUIDONI; ALMEIDA; GUIZZARDI, 2021)

para cada uma de suas especializações. Pontas de relações que estejam conectadas na classe também são migradas para as especializações, mas com seus limites inferiores relaxados para zero, pois o limite inferior anterior não precisa ser satisfeito para cada subclasse (GUIDONI; ALMEIDA; GUIZZARDI, 2021).

Na operação de *lifting*, mostrada na linha inferior da Tabela 8, uma subclasse (mostrada em cinza) será removida do modelo. Os atributos dessa classe serão migrados para a sua superclasse imediata, com suas cardinalidades inferiores relaxadas para zero. Pontas de relações que estejam conectadas na classe também migram para a superclasse imediata. A cardinalidade inferior da outra ponta da relação ($RelatedType_i$) é relaxada para zero (GUIDONI; ALMEIDA; GUIZZARDI, 2021).

Na ausência de conjuntos de generalização, um atributo booleano é adicionado na superclasse, usado para identificar se uma instanciação sua está instanciando a subclasse levantada ($SubType_y$) ou não. Caso haja um conjunto de generalização, é criado um enum discriminador com um valor para cada subclasse presente no conjunto. Um atributo do tipo desse discriminador é criado na superclasse, com a cardinalidade sendo definida pelas características do conjunto de generalização: Se o conjunto é incompleto, a cardinalidade é opcional, do contrário é obrigatória; e se o conjunto é disjunto, sua cardinalidade é multi-valorada, do contrário ela é mono-valorada. (GUIDONI; ALMEIDA; GUIZZARDI, 2021)

2.5 Tecnologias Utilizadas

2.5.1 OntoUML

OntoUML foi concebida como uma versão ontologicamente fundada da parte de diagramas de classes da UML 2.0. A ideia era empregar um método de engenharia de

linguagens baseado em ontologias para desenvolver uma linguagem para modelagem conceitual estrutural que tivesse duas principais características. Primeiramente, a visão de mundo integrada na linguagem através de suas primitivas deve ser isomórfica às distinções ontológicas definidas na UFO-A. Segundamente, o metamodelo da linguagem deve incorporar restrições sintáticas formais que delimitem o conjunto de modelos gramaticalmente válidos na linguagem aos modelos que representem o estados de casos permitidos pela ontologia usada. Ao fazer isso, foi construído uma linguagem de modelagem com semânticas formal e realista explicitamente definidas. Essa linguagem seve como ferramenta para permitir o uso de teorias formais de ontologias na construção de modelos conceituais e ontologias de domínio. (GUIZZARDI et al., 2015)

A primeira versão desse metamodelo UML foi proposta por Guizzardi em 2005; implementada completamente como um perfil UML concreto por Carraretto em 2010; e implementada como ferramenta baseada em modelos por Guizzardi e Benevides em 2009, chamada de *OntoUML Editor*[2]. Nesse editor de OntoUML original, tinha-se suporte para a construção de modelos e verificação formal (comparando contra as restrições formais de metamodelo que refletem o UFO-A). Com os anos, um número de contribuições foram agregadas na proposta de um novo editor que, além das funcionalidades previamente mencionadas, incorporasse funcionalidades adicionais (GUERSON et al., 2015) (GUIZZARDI et al., 2015):

i. **Uma abordagem baseada em padrões para construção de modelos:** Como mostrado por Guizzardi em 2014, OntoUML é uma linguagem baseada em padrões no sentido de que suas primitivas de modelagem são padrões. Não só isso, esses padrões tem natureza ontológica, por diretamente refletirem as micro-teorias ontológicas da linguagem. Além do mais, o editor também implementa um número de padrões topológicos que permitem isolar o escopo da transitividade das relações todo-parte(GUIZZARDI, 2009). Finalmente, o editor permite a extração de Padrões relacionados a Modelos de Ontologias de Base pare que tais padrões possam ser reutilizados para a construção de Ontologias de Domínio;

ii. **Verbalização de modelo:** Verbalização de modelo significa gerar uma renderização do modelo em linguagem natural. Esse processo é muito útil, por exemplo, para permitir que *experts* no domínio, que não sejam versados na linguagem de modelagem, consigam analisar o que está representado no modelo conceitual. O editor incorpora uma funcionalidade para automaticamente gerar verbalização de modelo em Inglês seguindo uma versão levemente modificada da proposta SBVR (Semântica para Vocabulários e Regras de Negócio) da OMG[3];

[2] https://code.google.com/archive/p/ontouml/
[3] http://www.omg.org/spec/SBVR/1.2/

iii. **Suporte para a representação de restrições formais específicas de domínio:** Para cobrir restrições de domínio que não possam ser representadas na notação diagramática da linguagem, o editor atual suporta a especificação de restrições formais OCL (GUERSON; ALMEIDA; GUIZZARDI, 2014) e OCL temporais (GUERSON; ALMEIDA, 2016). O editor fornece suporte para coloração de sintaxe, compleção de código e verificação de sintaxe para restrições textuais;

iv. **Validação de modelo:** Essa abordagem utiliza validação de modelos conceituais usando simulação visual. Em particular, na estratégia proposta por Braga em 2010 e Benevides em 2011, e posteriormente implementada no editor, temos a geração automática de instâncias visuais (exemplares) de um dado modelo conceitual tal que o modelador é visualmente confrontado pelo que o modelo está de fato representando. Em outras palavras, a estratégia é contrastar sistematicamente o conjunto de instâncias formalmente válidas de um dado modelo conceitual (automaticamente gerado pelo simulador visual) com o conjunto esperado de instâncias do modelo, que existem apenas na mente do modelador. Uma vez que o modelador nota um desvio entre as instâncias válidas e as esperadas (seja pelo excesso ou falta de constrições no modelo), ele retifica o modelo, adicionando ou removendo as constrições necessárias.

v. **Codificação de ontologia:** No processo de engenharia de ontologia defendido em (GUIZZARDI, 2007), (GUIZZARDI; HALPIN, 2008) e (GUIZZARDI, 2010), é advocado pela separação entre abordagens de modelagem que devem ser usadas para a fase de modelagem conceitual em engenharia de ontologias e as abordagens de representação que podem ser usadas para perceber implementações alternativas de uma dada ontologia (como modelo conceitual). Por exemplo, dado um modelo conceitual representando uma ontologia de domínio em OntoUML, nós podemos ter diferentes mapeamentos para diferentes linguagens de código (OWL DL, RDFS, F-Logic, Haskell, etc.). A escolha de cada linguagem deve ser feita a favor de um conjunto específico de requisitos não-funcionais. Além do mais, com o espaço de solução definido por essas linguagens de código, tem-se uma multitude de escolhas nos quesitos decidibilidade, completude, complexidade computacional, expressividade (necessidade de restrições modais ou tipos de alta ordem, por exemplo), verificação de satisfatibilidade, entre outras. Na versão em questão do editor, foram implementadas seis mapeamentos automáticos de OntoUML para OWL, contemplando diferentes estilos para sanar diferentes requisitos não-funcionais (GUIZZARDI; ZAMBORLINI, 2014; ZAMBORLINI; GUIZZARDI, 2010), além de outros mapeamentos para XML (BAUMAN, 2009), Smalltalk (PERGL; SALES; RYBOLA, 2013) e outros.

vi. **Detecção e retificação de anti-padrões:** Dada a difusão da linguagem, foi montado um repositório contendo modelos OntoUML de diferentes domínios (telecomunicações, governo, biodiversidade, etc.), diferentes tamanhos (desde dúzias até

milhares de conceitos), e produzidos para diferentes contextos (desde exercícios acadêmicos até modelos produzidos por times em ambientes governamentais) (SALES; GUIZZARDI, 2015). Usando esse repositório de modelos como *benchmark*, em três diferentes estudos empíricos, (GUIZZARDI; SALES, 2014) e (SALES; GUIZZARDI, 2015) mostram que essa abordagem de validação de modelo visual não só é útil para detectar desvios entre o modelo esperado e o modelo implementado, mas também para detectar padrões recorrentes que causam esses desvios, por exemplo, anti-padrões ontológicos. Uma vez que esses anti-padrões são catalogados, é possível montar padrões de solução, por exemplo, propostas de solução que eliminem os desvios entre o intencional e o implementado (GUIZZARDI; SALES, 2014) (SALES; GUIZZARDI, 2015). O editor implementa tanto um mecanismo para detectar anti-padrões e uma implementação para propor soluções. Sales e Guizzardi, em 2015, apresentaram um estudo de validação desenvolvido com um grande modelo industrial e conseguiram empiricamente demonstrar, para a maioria dos anti-padrões identificados, uma alta correlação entre a presença desses anti-padrões e a adoção das soluções propostas para melhoria do modelo.

2.5.2 OntoUML Schema

Ontouml-schema define a padronização de como modelos OntoUML são serializados para objetos JSON. O propósito de tal especificação é duplo. Primeiramente, de informar desenvolvedores sobre os elementos do metamodelo OntoUML, como *classes*, *relations*, e *generalization sets*, assim como suas propriedades. Por exemplo, um objeto descrevendo uma classe tem um campo booleano chamado *isAbstract*. Também define como serializar informação diagramática de um jeito não relacionado a nenhuma ferramenta. Segundamente, pode ser usado para automaticamente validar se uma modelo serializado está conforme suas regras, acusando quaisquer desvios. (FONSECA et al., 2021)

A especificação desse esquema age como um "metamodelo" da OntoUML, compartilhado entre serviços e ferramentas, sem a necessidade de eles usarem esse mesmo modelo internamente. (FONSECA et al., 2021)

2.5.3 OntoUML JS

O projeto *ontouml-js* existe para atender as necessidades de desenvolvedores implementando soluções baseadas em OntoUML. É uma biblioteca TypeScript que fornece uma coleção de classes, correspondendo aos elementos do metamodelo da OntoUML, e um extensível conjunto de métodos para facilitar a criação, modificação e leitura eficientes de seus elementos. Também inclui suporte para serializar e desserializar modelos que sejam compatíveis com o formato do OntoUML Schema (2.5.2), além de oferecer meios de validar tais modelos (FONSECA et al., 2021).

2.5.4 Tonto

Tonto é um acrônimo com as palavras Textual e Ontology, por ser um jeito textual de descrever modelos ontológicos. Tonto foi desenvolvido como uma sintaxe textual amigável para ontologias. Ele oferece suporte especializado para construções refletindo a ontologia UFO, o que torna possível identificar erros na ontologia que outrora passariam despercebidos. A linguagem foi desenvolvida de forma a permitir transformações para várias linguagens incluindo UML (mais especificamente, OntoUML), OWL (para ontologias baseadas em gUFO), Alloy, Common Logic e a sintáxe TPTP. (COUTINHO, 2023)

A linguagem suporta:

- Anotações baseadas em UFO para facilitar checagem de erros
- Tipos de alta ordem para taxonomias multi níveis
- Comentários estruturados para a geração de documentação
- Especificação de restrições para quando é necessária maior precisão
- Diretivas para testagem e verificação

Por ser uma sintaxe textual, a linguagem pode se beneficiar de ferramentas de versionamento como *git*, e permite que ontologias sejam examinadas e editadas sem ferramentas especiais. A extensão da linguagem para VS Code também providencia suporte para verificação de sintaxe, coloração de sintaxe e representação visual. A extensão é integrada com o servidor da OntoUML, para se beneficiar de serviços desenvolvidas para a linguagem, como transformação para OWL e geração de esquema de banco de dados. (COUTINHO, 2023)

2.6 Considerações Finais do Capítulo

Nesse capítulo foi apresentado o background técnico necessário para entender como o projeto foi feito. OntoUML (2.5.1) é uma ontologia feita em cima dos conceitos da UFO (2.2.1), e é utilizada como entrada. A ontologia é representada textualmente no formato JSON, seguindo o padrão OntoUML Schema (2.5.2), sendo esse padrão processado utilizando a ferramenta OntoUML-JS (2.5.3). O modelo fonte será lido e transformado, principalmente pelas tranformações de *flattening* e *lifting* (2.4.4).

3 Tonto's Code Compiler

3.1 Introdução

O trabalho proposto é uma biblioteca que foi nomeada *Tonto's Code Compiler* (ou TCC) e que tem como objetivo transformar modelos OntoUML em códigos Java, para desenvolvimento de serviços web (OBS) e bancos de dados (OBDR). O código Java gerado é baseado na padrão de projeto Model-View-Control (MVC) (FOWLER, 2012). A Figura 9 apresenta a arquitetura gerada, por meio de um diagrama de pacotes.

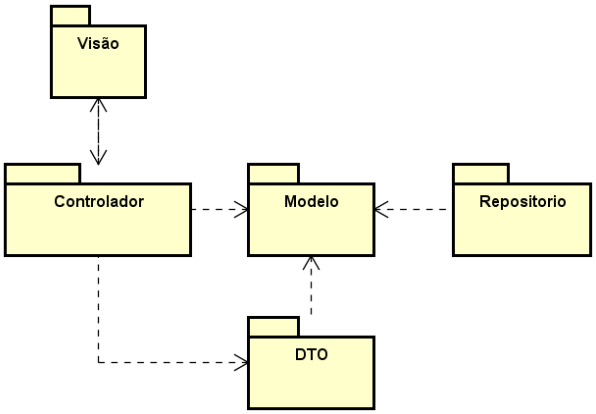

Figura 9 – Padrão MVC basedo em (FOWLER, 2012).

De forma breve, a camada Visão é responsável pela interface com o usuário. No caso deste trabalho é a interface do serviço web, ou OBS, conforme exemplificado na Figura 10. A camada de controle tem como objetivo receber a requisição do usuário, enviada por meio da camada de visão, e enviar para a camada de Modelo. A camada de Modelo possui as regras de negócio enquanto a camada de Repositório é responsável por gerenciar a persistência dos dados, servindo de OBDR. Por fim, a camada de DTO (*Data Transfer Object*) contém objetos que são utilizados para transferência de dados entre a aplicação e os usuários do serviço web.

Em relação às transformações, no escopo deste trabalho foram implementadas transformações que abrangem os *Sortals* - *Kind*, *Subkind*, *Phase* e *Role* - e *Category*, desde que a *Category* não tenha relações. Essa limitação de escopo foi devido à limitação do tempo de desenvolvimento deste trabalho.

Para cada *Kind* e *Subkind* são gerados os seguintes elementos da arquitetura:

- *Modelo*: representando a estrutura da classe tanto no código quanto no banco de dados;

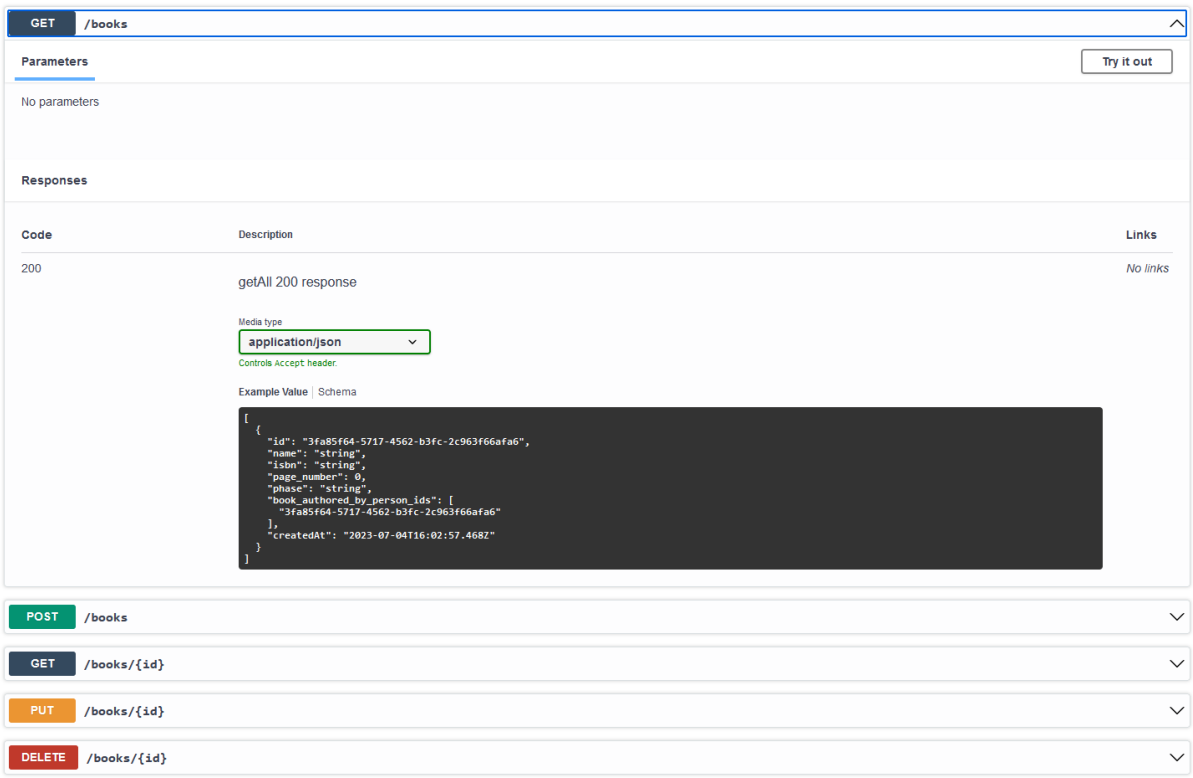

Figura 10 – Exemplo da interface web

- *Repositório*: responsável por servir as operações diretas no banco de dados;
- *Aplicação*: contendo a lógica que precisa ser feita antes de o repositório ser chamado para realizar a operação;
- *Controlador*: controlando as requisições feitas via a interface, lendo as informações da requisição e passando-as para a aplicação;
- *DTOs*: um DTO para entrada e um DTO para a saída, especificando, respectivamente, o formato na qual o controlador recebe ou retorna as informações de uma dada classe;
- *Exceção*: especificando a exceção que ocorre caso uma busca via ID da classe não encontre nada.

O TCC também gera exceções específicas para *Roles* e *Phases*. Essas exceções são levantadas, respectivamente, quando algum valor obrigatório da *Role* não foi passado, e quando é passado um valor inválido para uma *Phase*.

O processo completo de funcionamento de TCC, como ilustrado na Figura 11, é o seguinte:

i. O arquivo passado como entrada é lido. No caso do formato ontouml-schema, é usado o ontouml-js para lê-lo;

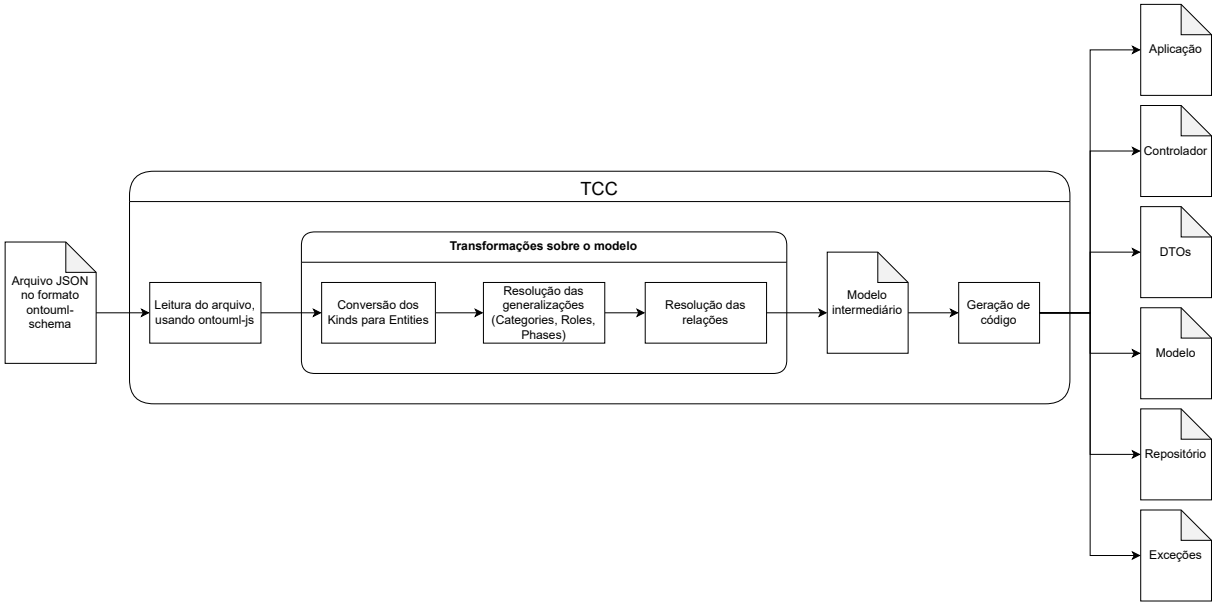

Figura 11 – Diagrama denotando o fluxo de funcionamento de TCC

ii. Com o modelo de entrada lido apropriadamente, são realizadas as transformações:

   a) Primeiramente, os *Kinds* e *Subkinds* são convertidos em Entidades;

   b) São tratadas as generalizações: *Roles*, *Phases*, *Subkinds* e *Categories*. Os *Subkinds* serão linkados a seus supertipos, todas as *Categories* serão *flattenned*, e todas as *Phases* e *Roles* serão *lifted*, restando apenas *Kinds* e *Subkinds* no modelo;

   c) São tratadas as relações. Nessa etapa, todas as relações do tipo *OnetoMany* são invertidas, e tratadas como *ManytoOne* no sentido oposto.

iii. Com o modelo interno finalizado, são gerados os arquivos de código a partir das Entidades do modelo.

As próximas seções descrevem como as transformações propostas por Guidoni, Almeida e Guizzardi (2021) foram utilizadas para criar a arquitetura apresentada na Figura 9.

3.2 Transformações em *Rigids*

Como instâncias de *Kind* sempre serão instâncias do mesmo *Kind* enquanto existirem, elas são transformadas em *Entities*, ou Entidades: uma coleção de atributos e relações que quando for gerado o banco de dados correspondente, cada entidade terá sua própria tabela, registrando suas instanciações, e que terá um conjunto completo de serviços, via métodos HTTP[1], para ser manipulado: POST para criação; GET para leitura; PUT para atualização; e DELETE para remoção.

[1] https://pt.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Métodos_de_solicitação

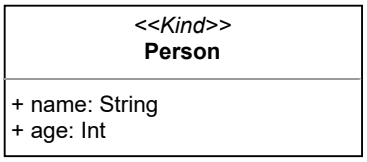

Figura 12 – Exemplo de *Kind*

Listagem 3.1 – Exemplo de modelo gerado para um *Kind*

```
@Table(name = "person")
public class Person implements Serializable {
    @Id
    @GeneratedValue(generator = "uuid")
    @GenericGenerator(name = "uuid", strategy = "uuid2")
    UUID id;

    String name;
    Float age;

    @Builder.Default
    LocalDateTime createdAt = LocalDateTime.now();

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || this.getClass() != o.getClass()) return false;

        Person elem = (Person) o;
        return getId().equals(elem.getId());
    }

    @Override
    public int hashCode() {
        return Objects.hash(getId());
    }
}
```

Como mostrado no código 3.1, referente a Figura 12, para um *Kind* é criado uma `class` Java com os atributos correspondentes (`int` é convertido para `Float` pois a geração de ontouml-js, a partir do Tonto, não distingue entre `int` e `float`, usando apenas `number`) e um atributo de `id`.

Listagem 3.2 – Exemplo de Input DTO gerado para um *Kind*

```
public record PersonInputDto(
    @NotNull String name,
    @NotNull Float age
) {}
```

Listagem 3.3 – Exemplo de Output DTO gerado para um *Kind*

```
public record PersonOutputDto(
    @NotBlank UUID id,
    @NotNull String name,
```

```
4     @NotNull Float age,
5     LocalDateTime createdAt
6 ) {}
```

Os códigos 3.2 e 3.3 mostram como ficam os DTOs gerados do *Kind*: O Output DTO representa o conjunto de todos atributos presentes no modelo do *Kind*, enquanto que o Input DTO representa apenas os atributos necessários para sua criação, não contendo valores gerados pelo sistema, como o `id`.

Listagem 3.4 – Exemplo de controlador gerado para um *Kind*

```
1  public class PersonController {
2      private final PersonApplication personApp;
3
4      @Post(uri = "/", consumes = MediaType.APPLICATION_JSON)
5      @Transactional
6      public HttpResponse<Void> create(@Body @Valid PersonInputDto dto) {
7          var saved = this.personApp.save(dto);
8          return HttpResponse.created(URI.create("/persons/" + saved.getId()));
9      }
10
11     @Get(uri = "/", produces = MediaType.APPLICATION_JSON)
12     @Transactional
13     public HttpResponse<List<PersonOutputDto>> getAll() {
14         var body = this.personApp.findAll()
15             .stream()
16             .map(elem -> this.modelToOutputDto(elem))
17             .toList();
18         return HttpResponse.ok(body);
19     }
20
21     @Get(uri = "/{id}", produces = MediaType.APPLICATION_JSON)
22     @Transactional
23     public HttpResponse<PersonOutputDto> getById(@PathVariable UUID id) {
24         var elem = this.personApp.findById(id);
25         return HttpResponse.ok(this.modelToOutputDto(elem));
26     }
27
28     @Put(uri = "/{id}", consumes = MediaType.APPLICATION_JSON)
29     @Transactional
30     public HttpResponse<?> updateById(@PathVariable UUID id, @Body @Valid
           PersonInputDto dto) {
31         var elem = this.personApp.update(id, dto);
32         return HttpResponse.ok(this.modelToOutputDto(elem));
33     }
34
35     @Delete(uri = "/{id}", produces = MediaType.APPLICATION_JSON)
36     @Transactional
37     public HttpResponse<?> deleteById(@PathVariable UUID id) {
38         this.personApp.delete(id);
39         return HttpResponse.noContent();
40     }
```

Como mostrado no código 3.4, o trabalho do controlador é o de declarar as rotas disponíveis para acessar o *Kind*, quais métodos HTTP estão disponíveis nessas rotas e

de gerar a resposta das requisições. Ele também é responsável por chamar as funções apropriadas da aplicação para cada método.

Listagem 3.5 – Exemplo de aplicação gerada para um *Kind*

```
1  public class PersonApplication {
2      private final PersonRepository repo;
3
4      public Person save(PersonInputDto dto) {
5          var builder = Person.builder()
6
7              .name(dto.name())
8              .age(dto.age());
9
10         var data = builder.build();
11         return this.repo.save(data);
12     }
13
14     public List<Person> findAll() {
15         return this.repo.findAll();
16     }
17
18     public Person findById(UUID id) {
19         return this.repo.findById(id).orElseThrow(() -> new
               PersonNotFoundException(id));
20     }
21
22     public Person update(UUID id, PersonInputDto dto) {
23         var elem = this.repo.findById(id).orElseThrow(() -> new
               PersonNotFoundException(id));
24
25         elem.setName(dto.name());
26         elem.setAge(dto.age());
27
28         return this.repo.update(elem);
29     }
30
31     public void delete(UUID id) {
32         var elem = this.repo.findById(id).orElseThrow(() -> new
               PersonNotFoundException(id));
33         this.repo.delete(elem);
34     }
35 }
```

No código 3.5, vemos a lógica básica das operações. As operações de criação e atualização ambas leem os atributos do DTO e registrados no modelo; As operações de busca, atualização e deleção, que puxam entidades do banco, usam o `id` da instância para fazer essa busca, e levantam exceções caso o `id` não seja encontrado. A aplicação usa funções do repositório para ler e escrever as mudanças no banco de dados.

Listagem 3.6 – Exemplo de repositório gerado para um *Kind*

```
1  import io.micronaut.data.annotation.Repository;
2  import io.micronaut.data.jpa.repository.JpaRepository;
3
```

```
4 @Repository
5 public interface PersonRepository extends JpaRepository<Person, UUID>{
6
7 }
```

O repositório, como mostrado no código 3.6, é o código mais simples gerado: ele apenas gera uma interface que especializa `JpaRepository`, parametrizada pelo modelo gerado, e o framework *Micronaut*[2] faz o resto do trabalho por debaixo dos panos. O código do repositório sempre será o mesmo para qualquer tipo de *Kind*, mudando apenas o nome da interface e o modelo que é passado de parâmetro.

Listagem 3.7 – Exemplo de exceção gerada para o caso de *Kind* não encontrado

```
1 public class PersonNotFoundException extends BaseException {
2     public PersonNotFoundException(UUID id) {
3         super("Person [id = "+id+"] was not found");
4     }
5 }
```

Por fim, o código 3.7, para a exceção que é levantada em caso de entidade não encontrada via `id`, também é um código tão simples quanto o do repositório. O código será virtualmente o mesmo para qualquer *Kind*, mudando apenas o nome da exceção e o nome do modelo na mensagem de erro.

Subkinds são transformados de maneira análoga aos *Kinds*, mas com algumas adições. Eles também são transformados em Entidades, mas como eles são especializações de um *Kind* ou *Subkind*, a tabela resultante no banco de dados terá apenas as informações específicas do *Subkind* e uma referência para a tabela do super-tipo, cada instância do subtipo apontando para a instância do super-tipo que ela especializa.

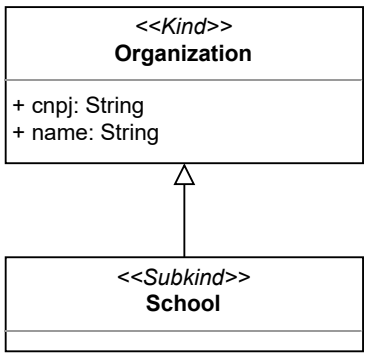

Figura 13 – Exemplo de modelo com *Subkind*

Listagem 3.8 – Exemplo de modelo gerado para um *Subkind*

```
1 @Table(name = "school")
2 public class School extends Organization {
3     @Id
```

[2] https://micronaut.io/

```
4       @GeneratedValue(generator = "uuid")
5       @GenericGenerator(name = "uuid", strategy = "uuid2")
6       UUID id;
7
8
9       @Builder.Default
10      LocalDateTime createdAt = LocalDateTime.now();
11
12      @Override
13      public boolean equals(Object o) {
14          if (this == o) return true;
15          if (o == null || this.getClass() != o.getClass()) return false;
16
17          School elem = (School) o;
18          return getId().equals(elem.getId());
19      }
20
21      @Override
22      public int hashCode() {
23          return Objects.hash(getId());
24      }
25  }
```

O código 3.8, gerado a partir da Figura 13, mostra o arquivo de modelo gerado. Diferente do modelo que é gerado para um *Kind* comum, ele estende o seu supertipo, e apenas os atributos específicos do *Subkind* são declarados.

Enquanto que no modelo são declarados, apenas, os atributos específicos do *Subkind*, na aplicação (Código 3.9) e nos DTOs (Códigos 3.10 e 3.11) é preciso tratar todos os atributos que compõem o *Subkind*, ou seja, os atributos de toda a sua linha hereditária.

Listagem 3.9 – Exemplo da aplicação gerada para um *Subkind*

```
1  public class SchoolApplication {
2      private final SchoolRepository repo;
3
4      public School save(SchoolInputDto dto) {
5          var builder = School.builder()
6
7              .cnpj(dto.cnpj())
8              .name(dto.name());
9
10         var data = builder.build();
11         return this.repo.save(data);
12     }
13
14     public List<School> findAll() {
15         return this.repo.findAll();
16     }
17
18     public School findById(UUID id) {
19         return this.repo.findById(id).orElseThrow(() -> new
               SchoolNotFoundException(id));
20     }
21
```

```
22      public School update(UUID id, SchoolInputDto dto) {
23          var elem = this.repo.findById(id).orElseThrow(() -> new
                SchoolNotFoundException(id));
24
25          elem.setCnpj(dto.cnpj());
26          elem.setName(dto.name());
27
28          return this.repo.update(elem);
29      }
30
31      public void delete(UUID id) {
32          var elem = this.repo.findById(id).orElseThrow(() -> new
                SchoolNotFoundException(id));
33          this.repo.delete(elem);
34      }
35  }
```

Listagem 3.10 – Exemplo do Input DTO gerado para um *Subkind*

```
1  public record SchoolInputDto(
2      @NotNull String cnpj,
3      @NotNull String name
4  ) {}
```

Listagem 3.11 – Exemplo do Output DTO gerado para um *Subkind*

```
1  public record SchoolOutputDto(
2      @NotBlank UUID id,
3      @NotNull String cnpj,
4      @NotNull String name,
5      LocalDateTime createdAt
6  ) {}
```

Uma *Category*, como mencionado na seção 2.2.1, representa um conjunto de atributos e relações que são comuns em múltiplos *Rigid Sortal* – *Kinds* e *Subkinds* – e não tem noção própria de identidade. Por conta disso, não há necessidade de transformar uma *Category* em Entidade. Ao invés disso, para todas as *Categories* do modelo, é feito o *flattening* dela para os *Kinds* e *Subkinds* que a especializam, e consequentemente para as entidades geradas a partir desses *Kinds* e *Subkinds*.

Listagem 3.12 – Exemplo de modelo gerado para um *Kind* com *Category*

```
1  @Table(name = "book")
2  public class Book implements Serializable {
3      @Id
4      @GeneratedValue(generator = "uuid")
5      @GenericGenerator(name = "uuid", strategy = "uuid2")
6      UUID id;
7
8      String name;
9      Float page_number;
10     String publisher;
11
```

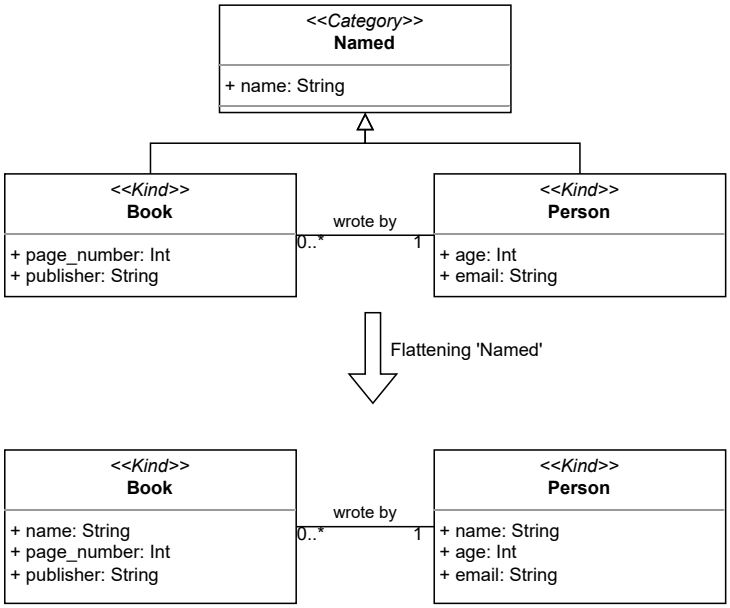

Figura 14 – Exemplo de transformação envolvendo *Category*

```
12      @ManyToOne
13      @JoinColumn(name="book_wrote_by_person_id")
14      private Person book_wrote_by_person;
15
16      @Builder.Default
17      LocalDateTime createdAt = LocalDateTime.now();
18
19      @Override
20      public boolean equals(Object o) {
21          if (this == o) return true;
22          if (o == null || this.getClass() != o.getClass()) return false;
23
24          Book elem = (Book) o;
25          return getId().equals(elem.getId());
26      }
```

Como mostrado na Figura 14, o atributo `name` era parte da *Category* Named. Após o *flattening*, esse atributo é passado para as classes específicas (Book e Person) e a *Category* é removida do modelo. A geração de código então, como demonstrado no código 3.12, é feita normalmente para *Kinds* e *Subkinds*.

Kinds com relações, como esse, além de afetarem o arquivo do modelo, afetam também os DTOs (códigos 3.14 e 3.15), que precisam registrar o atributo que representa a relação, e afetam principalmente o arquivo da aplicação (código 3.13), que precisa, nas operações de criação e atualização, usar o `id` passado como alvo da relação para procurar no banco de dados a instanciação correspondente.

Listagem 3.13 – Exemplo de código de aplicação gerado por um *Kind* com relação

```
1 public class BookApplication {
2     private final BookRepository repo;
3     private final PersonApplication personApp;
```

```
4
5       public Book save(BookInputDto dto) {
6           var book_wrote_by_person = personApp.findById(dto.book_wrote_by_person_id
                ());
7
8           var builder = Book.builder()
9               .book_wrote_by_person(book_wrote_by_person)
10              .name(dto.name())
11              .page_number(dto.page_number())
12              .publisher(dto.publisher());
13
14          var data = builder.build();
15          return this.repo.save(data);
16      }
17
18      public List<Book> findAll() {
19          return this.repo.findAll();
20      }
21
22      public Book findById(UUID id) {
23          return this.repo.findById(id).orElseThrow(() -> new BookNotFoundException
                (id));
24      }
25
26      public Book update(UUID id, BookInputDto dto) {
27          var book_wrote_by_person = personApp.findById(dto.book_wrote_by_person_id
                ());
28
29          var elem = this.repo.findById(id).orElseThrow(() -> new
                BookNotFoundException(id));
30          elem.setBook_wrote_by_person(book_wrote_by_person);
31          elem.setName(dto.name());
32          elem.setPage_number(dto.page_number());
33          elem.setPublisher(dto.publisher());
34
35          return this.repo.update(elem);
36      }
37
38      public void delete(UUID id) {
39          var elem = this.repo.findById(id).orElseThrow(() -> new
                BookNotFoundException(id));
40          this.repo.delete(elem);
41      }
42  }
```

Listagem 3.14 – Exemplo de código de Input DTO gerado por um *Kind* com relação

```
1 public record BookInputDto(
2     @NotNull String name,
3     @NotNull Float page_number,
4     @NotNull String publisher,
5     UUID book_wrote_by_person_id
6 ) {}
```

Listagem 3.15 – Exemplo de código de Output DTO gerado por um *Kind* com relação

```
public record BookOutputDto(
    @NotBlank UUID id,
    @NotNull String name,
    @NotNull Float page_number,
    @NotNull String publisher,
    UUID book_wrote_by_person_id,
    LocalDateTime createdAt
) {}
```

3.3 Transformações em *AntiRigids*

Phases representam as fases que um *Rigid Sortal* pode ser. Portanto, elas passarão por *lifting*, sendo convertidas em um Enum na entidade gerada pelo *Rigid Sortal* que especializam. Isso é feito com alguma limitações: todas as *Phases* de um *Rigid Sortal* específico são consideradas parte de um mesmo conjunto de generalização, disjunto e completo – cada instanciação do *Rigid Sortal* não pode ter múltiplas *Phases* (disjunto) e obrigatoriamente terá alguma *Phase* (completo).

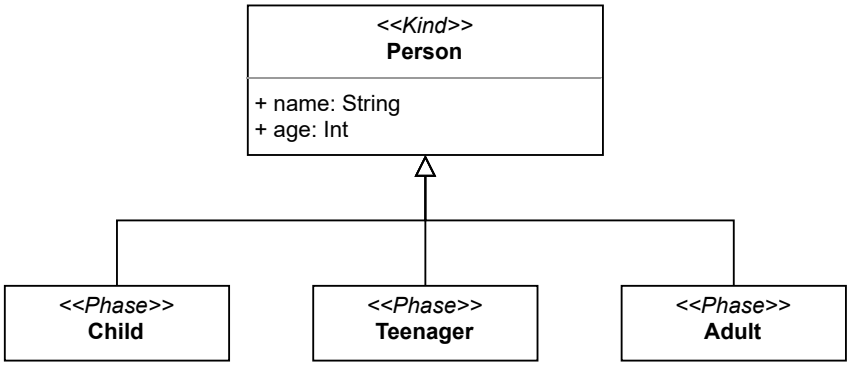

Figura 15 – Exemplo de modelo com *Phase*

Listagem 3.16 – Exemplo de modelo gerado para um *Kind* com *Phases*

```
@Table(name = "person")
public class Person implements Serializable {
    public enum Phase {
        CHILD, TEENAGER, ADULT;

        public static Optional<Phase> get(String s) {
            try {
                return Optional.of(Phase.valueOf(s.toUpperCase()));
            } catch (Exception e) {
                return Optional.empty();
            }
        }
    }

    @Id
    @GeneratedValue(generator = "uuid")
    @GenericGenerator(name = "uuid", strategy = "uuid2")
    UUID id;
```

```
19
20      String name;
21      Float age;
22
23      Phase phase;
24
25      @Builder.Default
26      LocalDateTime createdAt = LocalDateTime.now();
27
28      @Override
29      public boolean equals(Object o) {
30          if (this == o) return true;
31          if (o == null || this.getClass() != o.getClass()) return false;
32
33          Person elem = (Person) o;
34          return getId().equals(elem.getId());
35      }
36
37      @Override
38      public int hashCode() {
39          return Objects.hash(getId());
40      }
```

No código 3.16, gerado para o arquivo de modelo referente à figura 15, é criado um Enum interno à classe, chamado Phase, que tem apenas os valores especificados, e é adicionado ao modelo um atributo do tipo *Phase*.

Como mostrado nos códigos 3.17 e 3.18, dos DTOs, e no código 3.19, da aplicação, a *Phase* é tratada como se fosse um atributo do tipo String com apenas alguns valores válidos. Tentativas de colocar qualquer valor que não represente uma *Phase* válida fazem a aplicação levantar uma exceção, especificada no código 3.20.

Listagem 3.17 – Exemplo de Input DTO gerado para um *Kind* com *Phases*

```
1 public record PersonInputDto(
2     @NotNull String name,
3     @NotNull Float age,
4     @NotNull String phase
5 ) {}
```

Listagem 3.18 – Exemplo de Output DTO gerado para um *Kind* com *Phases*

```
1 public record PersonOutputDto(
2     @NotBlank UUID id,
3     @NotNull String name,
4     @NotNull Float age,
5     @NotNull String phase,
6     LocalDateTime createdAt
7 ) {}
```

Listagem 3.19 – Exemplo de aplicação gerada para um *Kind* com *Phases*

```
1 public class PersonApplication {
2     private final PersonRepository repo;
```

```
3
4       public Person save(PersonInputDto dto) {
5           var phase = Person.Phase.get(dto.phase()).orElseThrow(() -> new
                PersonPhaseNotFoundException(dto.phase()));
6
7           var builder = Person.builder()
8
9               .phase(phase)
10              .name(dto.name())
11              .age(dto.age());
12
13          var data = builder.build();
14          return this.repo.save(data);
15      }
16
17      public List<Person> findAll() {
18          return this.repo.findAll();
19      }
20
21      public Person findById(UUID id) {
22          return this.repo.findById(id).orElseThrow(() -> new
                PersonNotFoundException(id));
23      }
24
25      public Person update(UUID id, PersonInputDto dto) {
26          var phase = Person.Phase.get(dto.phase()).orElseThrow(() -> new
                PersonPhaseNotFoundException(dto.phase()));
27
28          var elem = this.repo.findById(id).orElseThrow(() -> new
                PersonNotFoundException(id));
29
30          elem.setName(dto.name());
31          elem.setAge(dto.age());
32          elem.setPhase(phase);
33
34          return this.repo.update(elem);
35      }
36
37      public void delete(UUID id) {
38          var elem = this.repo.findById(id).orElseThrow(() -> new
                PersonNotFoundException(id));
39          this.repo.delete(elem);
40      }
41  }
```

Listagem 3.20 – Exemplo de exceção gerada para um *Kind* com *Phases*

```
1  public class PersonPhaseNotFoundException extends BaseException {
2      public PersonPhaseNotFoundException(String phase) {
3          super("Phase of Person '"+phase+"' was not found. Valid options are:
                Child | Teenager | Adult");
4      }
5  }
```

Roles são um conjunto de informações sobre um *Rigid Sortal* condicionados pelo fato de ele estar exercendo uma função ou não. A transformação reflete isso fazendo o

lifting da *Role* para o *Rigid Sortal* que ela especializa, criando um atributo booleano cuja função é especificar se o *Rigid Sortal* está exercendo a função, e tratando os atributos e relações da *Role* como opcionais. Se o booleano tiver valor verdadeiro, os atributos e relações da *Role* devem estar preenchidos, do contrário, eles terão valores nulos.

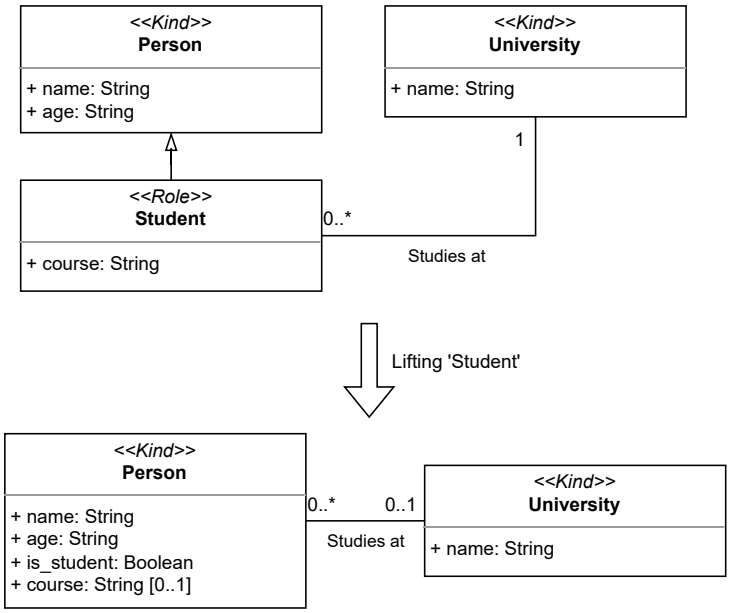

Figura 16 – Exemplo de transformação envolvendo *Role*

A lógica por trás do tratamento da *Role* fica no arquivo de aplicação. A aplicação de Person da Figura 16 é mostrada no código 3.21. Atributos e relações não relacionados à *Role* são tratados normalmente, enquanto que os atributos da *Role* só terão valores não-nulos se o booleano `is_student` for verdadeiro. Se o booleano for verdadeiro, mas algum dos atributos necessários para a *Role* for nulo, é levantada a exceção descrita no código 3.22.

Listagem 3.21 – Exemplo de aplicação de um *Kind* com *Role*

```
1  public class PersonApplication {
2      private final PersonRepository repo;
3      private final UniversityApplication universityApp;
4
5      public Person save(PersonInputDto dto) {
6          var builder = Person.builder()
7
8              .name(dto.name())
9              .age(dto.age());
10
11         boolean is_student = Boolean.TRUE.equals(dto.is_student());
12         builder = builder.is_student(is_student);
13         if(is_student) {
14             if(dto.student_course() == null || dto.
                   student_studies_at_university_id() == null) {
15                 throw new PersonStudentMissingValueException();
16             }
17
```

```
18              var student_studies_at_university = universityApp.findById(dto.
                    student_studies_at_university_id());
19
20              builder = builder
21                  .student_course(dto.student_course())
22                  .student_studies_at_university(student_studies_at_university);
23          }
24
25          var data = builder.build();
26          return this.repo.save(data);
27      }
```

Listagem 3.22 – Exemplo de exceção de um *Kind* com *Role*

```
1 public class PersonStudentMissingValueException extends BaseException {
2     public PersonStudentMissingValueException() {
3         super("Value required for the Role Student missing, but is_student is
              true. Values required for this Role are: is_student, student_course,
              student_studies_at_university.");
4     }
5 }
```

3.4 Demonstração de Uso

Para analisar o programa desenvolvido, foram realizados alguns casos de testes, gerando código a partir de modelos arbitrários, a fim de demonstrar a geração padronizada do código e o quão rápido essa geração é feita. Todos os modelos de teste foram escritos na linguagem do Tonto e convertidos para JSON, para então serem usados no TCC (como o Tonto está em desenvolvimento ativo, foi utilizado um *fork* do Tonto criado pelo autor[3], para garantir estabilidade).

Primeiramente, foi feito um exemplo mínimo, ilustrado na figura 17 e escrito em Tonto como no código 3.23, com o intuito apenas de verificar a padronização do código gerado.

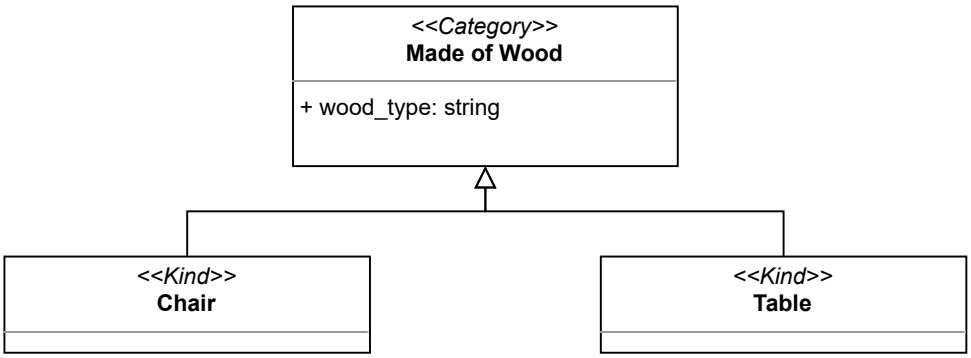

Figura 17 – Exemplo de modelo mínimo

[3] https://github.com/Danfs64/Tonto/tree/json

Listagem 3.23 – Representação em Tonto do modelo mínimo

```
module main {
    category MadeOfWood {
        wood_type: string
    }

    kind Chair specializes MadeOfWood
    kind Table specializes MadeOfWood
}
```

A geração desse exemplo mínimo é bastante rápida, demorando entre 7 e 9 milissegundos, e gera código bastante padronizado entre os dois *Kinds* presentes. De fato, como eles têm os mesmos atributos (apenas `wood_type`) e as mesmas relações (nenhuma), o código gerado para um *Kind* é virtualmente idêntico ao gerado para o outro. As únicas diferenças sendo nos nomes das classes (como exemplificado pelas aplicações em 3.24 e 3.25) e nas rotas declaradas no controlador (como mostrado nos código 3.26 e 3.27).

Listagem 3.24 – Código gerado para a aplicação de Chair

```
public class ChairApplication {
    private final ChairRepository repo;

    public Chair save(ChairInputDto dto) {
        var builder = Chair.builder()

            .wood_type(dto.wood_type());

        var data = builder.build();
        return this.repo.save(data);
    }

    public List<Chair> findAll() {
        return this.repo.findAll();
    }

    public Chair findById(UUID id) {
        return this.repo.findById(id).orElseThrow(() -> new
            ChairNotFoundException(id));
    }

    public Chair update(UUID id, ChairInputDto dto) {
        var elem = this.repo.findById(id).orElseThrow(() -> new
            ChairNotFoundException(id));

        elem.setWood_type(dto.wood_type());

        return this.repo.update(elem);
    }

    public void delete(UUID id) {
        var elem = this.repo.findById(id).orElseThrow(() -> new
            ChairNotFoundException(id));
        this.repo.delete(elem);
```

```
32     }
33 }
```

Listagem 3.25 – Código gerado para a aplicação de Table

```
1  public class TableApplication {
2      private final TableRepository repo;
3
4      public Table save(TableInputDto dto) {
5          var builder = Table.builder()
6
7              .wood_type(dto.wood_type());
8
9          var data = builder.build();
10         return this.repo.save(data);
11     }
12
13     public List<Table> findAll() {
14         return this.repo.findAll();
15     }
16
17     public Table findById(UUID id) {
18         return this.repo.findById(id).orElseThrow(() -> new
               TableNotFoundException(id));
19     }
20
21     public Table update(UUID id, TableInputDto dto) {
22         var elem = this.repo.findById(id).orElseThrow(() -> new
               TableNotFoundException(id));
23
24         elem.setWood_type(dto.wood_type());
25
26         return this.repo.update(elem);
27     }
28
29     public void delete(UUID id) {
30         var elem = this.repo.findById(id).orElseThrow(() -> new
               TableNotFoundException(id));
31         this.repo.delete(elem);
32     }
33 }
```

Listagem 3.26 – Código gerado para o controlador de Chair

```
1  @Controller("/chairs")
2  public class ChairController {
3      private final ChairApplication chairApp;
4
5      @Post(uri = "/", consumes = MediaType.APPLICATION_JSON)
6      @Transactional
7      public HttpResponse<Void> create(@Body @Valid ChairInputDto dto) {
8          var saved = this.chairApp.save(dto);
9          return HttpResponse.created(URI.create("/chairs/" + saved.getId()));
10     }
11
12     @Get(uri = "/", produces = MediaType.APPLICATION_JSON)
```

```
13      @Transactional
14      public HttpResponse<List<ChairOutputDto>> getAll() {
15          var body = this.chairApp.findAll()
16              .stream()
17              .map(elem -> this.modelToOutputDto(elem))
18              .toList();
19          return HttpResponse.ok(body);
20      }
21
22      @Get(uri = "/{id}", produces = MediaType.APPLICATION_JSON)
23      @Transactional
24      public HttpResponse<ChairOutputDto> getById(@PathVariable UUID id) {
25          var elem = this.chairApp.findById(id);
26          return HttpResponse.ok(this.modelToOutputDto(elem));
27      }
28
29      @Put(uri = "/{id}", consumes = MediaType.APPLICATION_JSON)
30      @Transactional
31      public HttpResponse<?> updateById(@PathVariable UUID id, @Body @Valid
            ChairInputDto dto) {
32          var elem = this.chairApp.update(id, dto);
33          return HttpResponse.ok(this.modelToOutputDto(elem));
34      }
35
36      @Delete(uri = "/{id}", produces = MediaType.APPLICATION_JSON)
37      @Transactional
38      public HttpResponse<?> deleteById(@PathVariable UUID id) {
39          this.chairApp.delete(id);
40          return HttpResponse.noContent();
41      }
42
43      private ChairOutputDto modelToOutputDto(Chair elem) {
44          return new ChairOutputDto(
45              elem.getId(),
46              elem.getWood_type(),
47
48              elem.getCreatedAt()
49          );
50      }
51  }
```

Listagem 3.27 – Código gerado para o controlador de Table

```
1  @Controller("/tables")
2  public class TableController {
3      private final TableApplication tableApp;
4
5      @Post(uri = "/", consumes = MediaType.APPLICATION_JSON)
6      @Transactional
7      public HttpResponse<Void> create(@Body @Valid TableInputDto dto) {
8          var saved = this.tableApp.save(dto);
9          return HttpResponse.created(URI.create("/tables/" + saved.getId()));
10     }
11
12     @Get(uri = "/", produces = MediaType.APPLICATION_JSON)
13     @Transactional
```

```
14      public HttpResponse<List<TableOutputDto>> getAll() {
15          var body = this.tableApp.findAll()
16              .stream()
17              .map(elem -> this.modelToOutputDto(elem))
18              .toList();
19          return HttpResponse.ok(body);
20      }
21
22      @Get(uri = "/{id}", produces = MediaType.APPLICATION_JSON)
23      @Transactional
24      public HttpResponse<TableOutputDto> getById(@PathVariable UUID id) {
25          var elem = this.tableApp.findById(id);
26          return HttpResponse.ok(this.modelToOutputDto(elem));
27      }
28
29      @Put(uri = "/{id}", consumes = MediaType.APPLICATION_JSON)
30      @Transactional
31      public HttpResponse<?> updateById(@PathVariable UUID id, @Body @Valid
            TableInputDto dto) {
32          var elem = this.tableApp.update(id, dto);
33          return HttpResponse.ok(this.modelToOutputDto(elem));
34      }
35
36      @Delete(uri = "/{id}", produces = MediaType.APPLICATION_JSON)
37      @Transactional
38      public HttpResponse<?> deleteById(@PathVariable UUID id) {
39          this.tableApp.delete(id);
40          return HttpResponse.noContent();
41      }
42
43      private TableOutputDto modelToOutputDto(Table elem) {
44          return new TableOutputDto(
45              elem.getId(),
46              elem.getWood_type(),
47
48              elem.getCreatedAt()
49          );
50      }
51  }
```

Em seguida, para uma demonstração mais completa da capacidade do gerador, foi gerado o código a partir de um modelo (Figura 18, código 3.28) que acoberta todas as transformações tratadas nesse trabalho. A geração do código demora entre 10 e 50 milissegundos

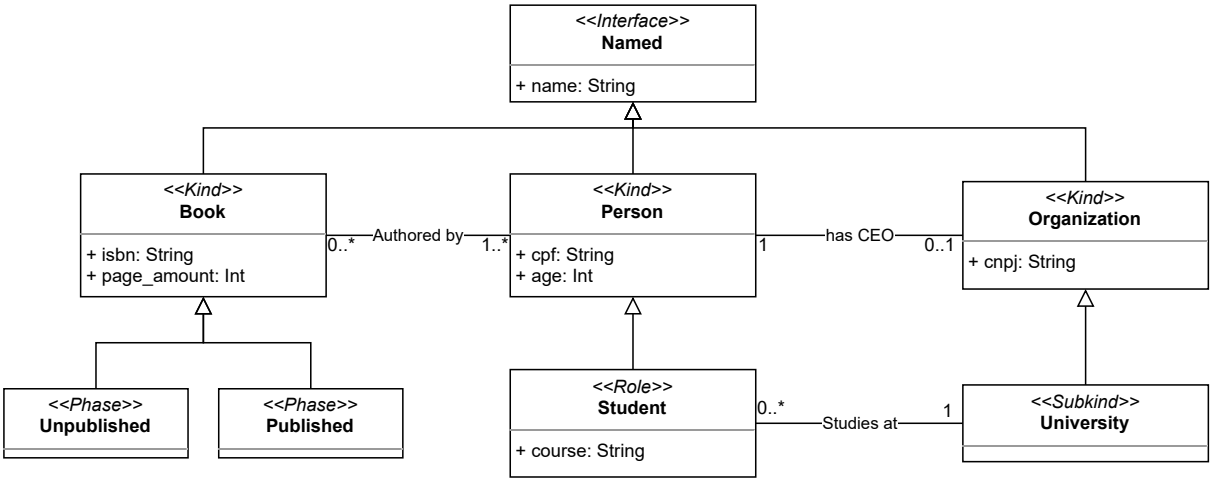

Figura 18 – Modelo usado para gerar o estudo de caso

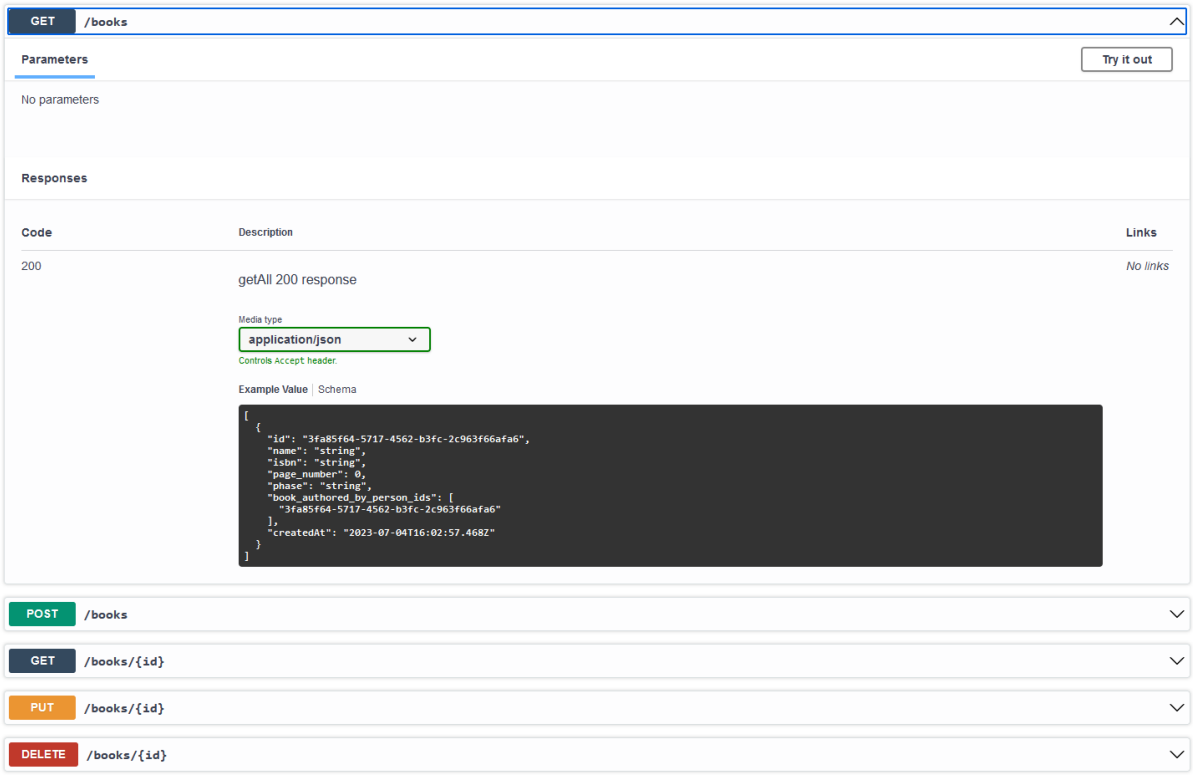

Figura 19 – Interface gerada para o modelo da Figura 18

Listagem 3.28 – Representação em Tonto do modela da Figura 18

```
1  module main {
2      category Named {
3          name: string
4      }
5
6      kind Person specializes Named {
7          cpf: string
8          age: number
9      }
10
```

```
11      kind Organization specializes Named {
12          cnpj: string
13      }
14
15      relation Organization [0..1] — has_ceo — [1] Person
16
17      subkind University specializes Organization
18
19      role Student specializes Person {
20          course: string
21      }
22
23      relation Student [0..*] — studies_at — [1] University
24
25      kind Book specializes Named {
26          isbn: string
27          page_number: number
28      }
29
30      phase Unpublished specializes Book
31      phase Published specializes Book
32
33      relation Book [0..*] — authored_by — [1..*] Person
34  }
```

Por fim, foi usado um modelo abstrato (Figura 20), com várias entidades, a fim de testar a eficiência computacional do gerador. A geração do código desse modelo, que resulta em mais de 150 arquivos de código gerados, não demora mais do que 1 segundo. Mais especificamente, seu tempo de execução varia dentre 35 e 105 milissegundos.

Nota-se, então, que os códigos são gerados de forma rápida e padronizada, sendo que mais tempo é gasto na escrita dos modelos do que na geração do código correspondente. Como mencionado em 2.4, a utilização desses modelos já é uma prática comum no âmbito de desenvolvimento de software, então fazer o intervalo de tempo entre a criação do modelo e a codificação desse modelo estar na casa dos milissegundos, um processo que normalmente gastaria múltiplos homem-hora, é de grande significância.

3.5 Considerações Finais

Neste capítulo foi apresentada um biblioteca chamada TCC (*Tonto's Code Compiler*) que permite transformar modelos OntoUML em códigos em Java para o desenvolvimento de serviços Web. Além disso, foi apresentado um conjunto de estudos de casos no qual TCC auxiliou no desenvolvimento de serviços web usando como entrada um modelo OntoUML. TCC foi desenvolvido no contexto de *Immigrant* (JÚNIOR; BARCELLOS; ALMEIDA, 2021), uma abordagem que visa à integração de dados para apoiar desenvolvimento de software orientado a dados no contexto de ESC. Mais especificamente, TCC está relacionado a *The Band* (SANTOS JÚNIOR, 2023), o componente de *Immigrant* que é

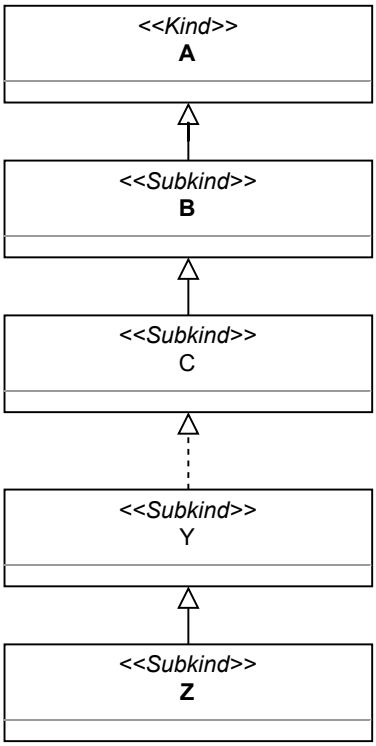

Figura 20 – Exemplo de modelo com 1 *Kinds* e 25 *Subkinds* encadeados. Alguns *Subkinds* foram omitido por brevidade

responsável pela integração de dados. Em *The Band*, ontologias em rede são utilizadas como base para o desenvolvimento de serviços web e bancos de dados, chamados respectivamente de *Ontology-based Services* (OBS) e *Ontology-based Data Repositories* (OBDR). Com o uso de TCC, será possível transformar os modelos OntoUML das ontologias em código Java para o desenvolvimento de OBSs e OBDRs.

4 Conclusão e Trabalhos Futuros

4.1 Conclusão

Nesse trabalho foi desenvolvido um programa baseado nos princípios de desenvolvimento orientado a modelos para gerar código a partir de modelos OntoUML, a fim de diminuir o tempo de desenvolvimento de software.

Ao longo do trabalho, foram apresentados os conceitos relevantes no capítulo 2, falando sobre ontologias, UFO e abordagens de desenvolvimento como a orientada a modelos, e as principais tecnologias utilizadas, como OntoUML e seu ecossistema de ferramentas. Esses conceitos e ferramentas tornaram possível o desenvolvimento deste trabalho, cujo principal resultado foi descrito no Capítulo 3.

No capítulo 1 foram listados os objetivos principais do trabalho. A Tabela 1 lista novamente esses objetivos, dessa vez acompanhados da sua indicação de atendimento e dos resultados alcançados evidenciando o atendimento descrito.

Tabela 1 – Objetivos e sua situação na conclusão da monografia

Objetivos do trabalho	**Status**	**Resultado**
Desenvolver uma biblioteca que recebe como entrada um modelo textual OntoUML e realize transformações nesse modelo	Atendido	Demonstração das transformações, listadas no Capítulo 3
A partir do modelo gerado no objetivo anterior, gerar código Java correspondente a OBDR e a OBS do modelo de entrada	Atendido	Demonstração dos arquivos gerados, listados no Capítulo 3
Disponibilizar essa funcionalidade como um pacote público, para que possa ser utilizada em outros projetos	Atendido	Pacote disponibilizado no npm

Ao longo desse trabalho, vários conhecimentos adquiridos durante o graduação foram colocados em prática. Os conhecimentos mais significativos vieram das matérias de:

- **Programação 1, 2, 3 e Estruturas de Dados 1 e 2:** que deram toda a base de conhecimento de programação; práticas de boa organização de código; conhecimento de estruturas de dados apropriadas para cada situação; e análise de complexidade assintótica de código.
- **Engenharia de Software:** que deu toda a base de conhecimento sobre Ontologias,

e sobre padrões de desenvolvimento de software.

- **Compiladores:** que deu a base de como projetar um software cujo objetivo é processar arquivos em linguagens fonte para converter em arquivos em linguagem alvo, com fácil extensibilidade para adicionar linguagens em ambos os lados.
- **Banco de Dados:** que apesar de os conhecimentos passados não terem sido usados diretamente no desenvolvimento do trabalho, ter esses conhecimentos facilitaram o entendimento de ferramentas utilizadas.

A principal dificuldade do trabalho foi o aprendizado da sintaxe e do ambiente de desenvolvimento da linguagem TypeScript, que o autor nunca tinha utilizado anteriormente, mas que foi escolhida para poder usar a ferramenta ontouml-js (2.5.3) e permitir que esse trabalho possa ser integrado ao Tonto (2.5.4).

Outras dificuldades incluem o aprendizado do framework *Micronaut*, que é um framework bastante extenso, mas com uma documentação que deixa a desejar; e a definição de como estrutura o modelo intermediário utilizado pelo programa, para garantir que ele seja o mais genérico e completo possível para que outras linguagens, tanto fonte quanto alvo, sejam adicionadas facilmente.

Por fim, apesar de desenvolvido bem sucedido, o trabalho apresenta limitações. As principais limitações foram: a falta de tempo para efetuar estudos de caso com usuários não relacionados ao desenvolvimento do trabalho; e o fato de que nem todos os esteriótipos do OntoUML foram integrados nas transformações.

Como o trabalho se trata de uma biblioteca JavaScript, ele foi disponibilizado publicamente[1] no npm[2] para que possa ser utilizado em outros projetos.

4.2 Trabalhos Futuros

Como trabalhos futuros, os seguintes pontos foram identificados como melhorias para a ferramenta:

- Integrar mais estereótipos de OntoUML nas transformações e geração de código, especialmente *NonSortals* com relações, e *Relators*;
- Realizar estudos de casos aonde usuários finais utilizem a ferramenta, para avaliar sua facilidade de uso;
- Integrar outras linguagens como linguagens fonte ou linguagens alvo, aumentando a abrangência da ferramenta;

[1] <https://www.npmjs.com/package/@danfs/tcc>
[2] Node Package Manager. <https://www.npmjs.com/>

- Integrar a ferramenta com a versão mais recente do Tonto, permitindo que usuários do Tonto possam usá-lo diretamente para geração de código;
- Com a adição de outras linguagens na ferramenta, revisar a representação intermediária, para ver se ela necessita ser expandida;
- Integrar diretamente a ferramenta no fluxo do *Immigrant*.

Referências

ALMEIDA, J. *Model-Driven Design of Distributed Applications*. 1. ed. [S.l.]: Telematica Instituut / CTIT, 2006. CTIT Ph.D.-Thesis Series, ISSN 1381-3617, No. 06-85 ; Telematica Instituut Fundamental Research Series, ISSN 1388-1795, No. 018

. ISBN 90-75176-422. Citado 2 vezes nas páginas 22 e 23.

ALMEIDA, J. et al. Model Driven Design, Refinement and Transformation of Abstract Interactions. *International Journal of Cooperative Information Systems (Singapore)*, v. 15, p. 599–632, 2006. ISSN 02188430. Citado 2 vezes nas páginas 22 e 23.

ALMEIDA, J. P.; ECK, P. van; IACOB, M.-E. Requirements traceability and transformation conformance in model-driven development. In: IEEE. *2006 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06)*. [S.l.], 2006. p. 355–366. Citado 4 vezes nas páginas 22, 24, 25 e 26.

ALMEIDA, J. P. A.; FALBO, R. A.; GUIZZARDI, G. Events as Entities in Ontology-Driven Conceptual Modeling. In: *Proc. of the 38th International Conference on Conceptual Modeling (ER 2019)*. [S.l.]: Springer-Verlag, 2019. Citado na página 16.

BARCELLOS, M. P. Towards a framework for continuous software engineering. In: *Proceedings of the 34th Brazilian Symposium on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020. (SBES '20), p. 626–631. ISBN 9781450387538. Disponível em: <https://doi.org/10.1145/3422392.3422469>. Citado 2 vezes nas páginas 10 e 14.

BAUMAN, B. T. Prying apart semantics and implementation: Generating xml schemata directly from ontologically sound conceptual models. In: *Balisage: The Markup Conference*. [S.l.: s.n.], 2009. p. 11–14. Citado na página 29.

BENEVIDES, A. B.; GUIZZARDI, G. A Model-Based Tool for Conceptual Modeling and Domain Ontology Engineering in OntoUML. In: *Lecture Notes in Business Information Processing (LNBIP)*. [S.l.]: Springer-Verlag, 2009. p. 528–537. Citado na página 28.

BENEVIDES, A. B. et al. Validating modal aspects of OntoUML conceptual models using automatically generated visual world structures. *Journal of Universal Computer Science (Print)*, v. 16, p. 2904–2933, 2011. ISSN 0948695X. Citado na página 29.

BRAGA, B. F. B. et al. Transforming OntoUML into Alloy: towards conceptual model validation using a lightweight formal method. *Innovations in Systems and Software Engineering (Print)*, p. 17–24, 2010. ISSN 16145046. Citado na página 29.

BUSSE, S. et al. Federated information systems: Concepts, terminology and architectures. 06 1999. Citado na página 20.

CALHAU, R. F.; FALBO, R. d. A. An Ontology-based Approach for Semantic Integration. In: *Proceedings 14th IEEE International Enterprise Distributed Object Computing Conference*. [S.l.]: IEEE Computer Society, 2010. p. 111–120. Citado na página 11.

CARRARETTO, R. *A Modeling Infrastructure for OntoUML*. [S.l.], 2010. Citado na página 28.

CARVALHO, V. A. et al. Multi-level ontology-based conceptual modeling. *DATA & KNOWLEDGE ENGINEERING*, v. 109, p. 3–24, 2017. ISSN 0169023X. Citado na página 19.

COUTINHO, M. L. *Tonto - README*. 2023. <https://github.com/matheuslenke/Tonto/blob/18c6cde34547b85c60a24641a3b1d66341c1d9a9/README.md>. Acessed: 12-Apr-2023. Citado na página 31.

FALBO, R. SABiO: Systematic approach for building ontologies. *CEUR Workshop Proceedings*, v. 1301, 01 2014. Citado na página 16.

FITZGERALD, B.; STOL, K.-J. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, Elsevier BV, v. 123, p. 176–189, jan 2017. Disponível em: <https://doi.org/10.1016%2Fj.jss.2015.06.063>. Citado 2 vezes nas páginas 10 e 14.

FONSECA, C. M. et al. Ontology-driven conceptual modelling as a service. In: *Proceedings of the Joint Ontology Workshops 2021. FOMI 2021: 11th International Workshop on Formal Ontologies meet Industry*. CEUR-WS.org, 2021. (CEUR Workshop Proceedings). Disponível em: <http://ceur-ws.org/Vol-2969/>. Citado na página 30.

FONSECA, V. S.; BARCELLOS, M. P.; de Almeida Falbo, R. An ontology-based approach for integrating tools supporting the software measurement process. *Science of Computer Programming*, v. 135, p. 20–44, 2017. ISSN 0167-6423. Special Issue on Advances in Software Measurement. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167642316301599>. Citado na página 11.

FOWLER, M. *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch*. [S.l.]: Addison-Wesley, 2012. Citado na página 32.

GRUBER, T. R. A translation approach to portable ontology specifications. *Knowledge Acquisition*, v. 5, n. 2, p. 199–220, 1993. ISSN 1042-8143. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1042814383710083>. Citado 2 vezes nas páginas 11 e 15.

GUERSON, J.; ALMEIDA, J. P. A. Representing Dynamic Invariants in Ontologically Well-Founded Conceptual Models. In: *17th International Conference, BPMDS 2016, 21st International Conference, EMMSAD 2016, Held at CAiSE 2016, Ljubljana, Slovenia, June 13-14,2016 , Proceedings*. [S.l.]: Springer, 2016. p. 303–317. Citado na página 29.

GUERSON, J.; ALMEIDA, J. P. A.; GUIZZARDI, G. Support for Domain Constraints in the Validation of Ontologically Well-Founded Conceptual Models. In: *Proceedings 15th International Conference, BPMDS 2014, 19th International Conference, EMMSAD 2014, Held at CAiSE 2014*. [S.l.]: Springer, 2014. p. 302–316. Citado na página 29.

GUERSON, J. et al. OntoUML Lightweight Editor: A Model-Based Environment to Build, Evaluate and Implement Reference Ontologies. In: *Proceedings of the 19th IEEE Enterprise Computing Conference (EDOC 2015)(DEMO TRACK)*. [S.l.]: San Francisco, 2015. Citado na página 28.

GUIDONI, G. L.; ALMEIDA, J. P. A.; GUIZZARDI, G. Forward engineering relational schemas and high-level data access from conceptual models. In: SPRINGER. *Conceptual Modeling: 40th International Conference, ER 2021, Virtual Event, October 18–21, 2021, Proceedings 40*. [S.l.], 2021. p. 133–148. Citado 4 vezes nas páginas 12, 26, 27 e 34.

GUIZZARDI, G. *Ontological foundations for structural conceptual models*. Tese (Doutorado) — University of Twente, out. 2005. Citado 5 vezes nas páginas 16, 17, 18, 19 e 28.

GUIZZARDI, G. On Ontology, ontologies, Conceptualizations, Modeling Languages, and (Meta)Models. In: *Frontiers in Artificial Intelligence and Applications, Databases and Information Systems IV*. [S.l.]: IOS Press, 2007. Citado 2 vezes nas páginas 16 e 29.

GUIZZARDI, G. The Problem of Transitivity of Part-Whole Relations in Conceptual Modeling Revisited. In: *Lecture Notes in Computer Science (LNCS)*. [S.l.]: Springer-Verlag, 2009. p. 94–109. Citado na página 28.

GUIZZARDI, G. Theoretical foundations and engineering tools for building ontologies as reference conceptual models. *Semantic Web: interoperability, usability, applicability*, v. 1, p. 3–10, 2010. ISSN 15700844. Citado na página 29.

GUIZZARDI, G. Ontological Patterns, Anti-Patterns and Pattern Languages for Next-Generation Conceptual Modeling. In: *Proceedings of the 33rd International Conference on Conceptual Modeling (ER 2014)*. [S.l.]: Springer-Verlag, 2014. Citado na página 28.

GUIZZARDI, G. et al. Ufo: Unified foundational ontology. *Applied Ontology*, IOS Press, v. 17, n. 1, p. 167–210, 2022. Citado 2 vezes nas páginas 16 e 17.

GUIZZARDI, G.; FALBO, R. D. A.; GUIZZARDI, R. S. S. Grounding Software Domain Ontologies in the Unified Foundational Ontology (UFO): The case of the ODE Software Process Ontology. In: *Proceedings of the XI Iberoamerican Workshop on Requirements Engineering and Software Environments (IDEAS?2008)*. [S.l.: s.n.], 2008. Citado na página 16.

GUIZZARDI, G. et al. Endurant Types in Ontology-Driven Conceptual Modeling: Towards OntoUML 2.0. In: *Lecture Notes in Computer Science*. [S.l.]: Springer-Verlag, 2018. Citado 3 vezes nas páginas 17, 18 e 19.

GUIZZARDI, G.; HALPIN, T. Ontological Foundations for Conceptual Modeling. *Applied Ontology*, v. 3, p. 91–110, 2008. ISSN 15705838. Citado na página 29.

GUIZZARDI, G.; SALES, T. P. Detection, Simulation and Elimination of Semantic Anti-patterns in Ontology-Driven Conceptual Models. In: *Proceedings of the 33rd International Conference on Conceptual Modeling (ER 2014)*. [S.l.]: Springer-Verlag, 2014. Citado na página 30.

GUIZZARDI, G. et al. Towards ontological foundations for conceptual modeling: The unified foundational ontology (UFO) story. *Applied Ontology*, IOS Press, v. 10, n. 3-4, p. 259–271, Dec 2015. Disponível em: <http://dx.doi.org/10.3233/ao-150157>. Citado 2 vezes nas páginas 16 e 28.

GUIZZARDI, G. et al. Towards Ontological Foundations for the Conceptual Modeling of

Events. In: *Proceedings of the 32nd International Conference on Conceptual Modeling (ER 2013)*. [S.l.]: Springer-Verlag, 2013. Citado na página 16.

GUIZZARDI, G.; ZAMBORLINI, V. Using a trope-based foundational ontology for bridging different areas of concern in ontology-driven conceptual modeling. *Science of Computer Programming (Print)*, v. 1, p. 1–27, 2014. ISSN 01676423. Citado na página 29.

GUIZZARDI, R. S. S.; GUIZZARDI, G. Ontology-Based Transformation Framework from Tropos to AORML. In: *Social Modeling for Requirements Engineering, Cooperative Information Systems Series*. [S.l.]: MIT Press, 2010. Citado na página 16.

JOHANSSEN, J. O. et al. Continuous software engineering and its support by usage and decision knowledge: An interview study with practitioners. *Journal of Software: Evolution and Process*, Wiley, v. 31, n. 5, p. e2169, may 2019. Disponível em: <https://doi.org/10.1002%2Fsmr.2169>. Citado na página 14.

JÚNIOR, P. S. dos S.; BARCELLOS, M. P.; ALMEIDA, J. P. A. An ontology-based approach to enable data-driven decision-making in agile software organizations. In: *ONTOBRAS*. [S.l.: s.n.], 2021. p. 279–284. Citado na página 53.

JUNIOR, P. S. S. *Uma Abordagem de Desenvolvimento Baseada em Modelos de Arquitetura Organizacional de TI: da Semântica ao Desenvolvimento de Sistemas*. Dissertação (Mestrado), 2009. Citado na página 22.

KURAPATI, N.; MANYAM, V.; PETERSEN, K. Agile software development practice adoption survey. In: . [S.l.: s.n.], 2012. v. 111, p. 16–30. ISBN 978-3-642-30349-4. Citado na página 10.

NARDI, J. C.; FALBO, R. D. A.; ALMEIDA, J. P. A. Foundational Ontologies for Semantic Integration in EAI: A Systematic Literature Review. In: *Proceedings 12th IFIP WG 6.11 Conference on e-Business, e-Services, and e-Society, I3E 2013*. [S.l.]: Springer, 2013. p. 238–249. Citado na página 11.

OLSSON, H. H.; ALAHYARI, H.; BOSCH, J. Climbing the "stairway to heaven-- a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In: *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*. [S.l.: s.n.], 2012. p. 392–399. Citado 2 vezes nas páginas 10 e 14.

OMG, O. M. G. *MDA Guide rev. 2.0*. 2014. <https://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf>. Acessed: 07-Jun-2023. Citado 3 vezes nas páginas 11, 22 e 24.

PERGL, R.; SALES, T. P.; RYBOLA, Z. Towards ontouml for software engineering: from domain ontology to implementation model. In: SPRINGER. *Model and Data Engineering: Third International Conference, MEDI 2013, Amantea, Italy, September 25-27, 2013. Proceedings 3*. [S.l.], 2013. p. 249–263. Citado na página 29.

POKRAEV, S. Model-driven semantic integration of service-oriented applications. In: . [S.l.: s.n.], 2009. Citado na página 11.

SALES, T. P.; GUIZZARDI, G. Ontological anti-patterns: empirically uncovered error-prone structures in ontology-driven conceptual models. *Data & Knowledge Engineering*, v. 1, p. 1–50, 2015. ISSN 0169023X. Citado na página 30.

SANTOS JÚNIOR, P. S. *From Continuous Software Engineering Reference Ontologies to the Integration of Data for Data-Driven Software Development*. Tese (Doutorado), 2023. Citado 5 vezes nas páginas 11, 19, 20, 22 e 53.

SANTOS JÚNIOR, P. S. et al. From a scrum reference ontology to the integration of applications for data-driven software development. *Information and Software Technology*, v. 136, p. 106570, 2021. ISSN 0950-5849. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584921000537>. Citado na página 19.

SANTOS, P. S. d.; BARCELLOS, M. P.; CALHAU, R. F. Am i going to heaven? first step climbing the stairway to heaven model results from a case study in industry. In: *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020. (SBES '20), p. 309–318. ISBN 9781450387538. Disponível em: <https://doi.org/10.1145/3422392.3422406>. Citado na página 19.

SCHERP, A. et al. Designing core ontologies. *Applied Ontology*, v. 6, 08 2011. Citado na página 15.

STUDER, R.; BENJAMINS, V. R.; FENSEL, D. Knowledge engineering: principles and methods. *Data & knowledge engineering*, Elsevier, v. 25, n. 1-2, p. 161–197, 1998. Citado na página 15.

SUÁREZ-FIGUEROA, M. C. et al. *Ontology Engineering in a Networked World*. Springer Berlin Heidelberg, 2012. 2 p. Disponível em: <https://doi.org/10.1007%2F978-3-642-24794-1>. Citado na página 12.

WACHE, H. et al. Ontology-based information integration: A survey. *Bremen, The BUSTER Project, Intelligent Systems Group*, 2001. Citado na página 11.

ZAMBORLINI, V.; GUIZZARDI, G. On the representation of temporally changing information in OWL. In: *Workshop Proceedings of the 15th International Enterprise Computing Conference (EDOC 2010)*. [S.l.: s.n.], 2010. Citado na página 29.