INSTITUTO FEDERAL DO ESPÍRITO SANTO
CURSO SUPERIOR DE SISTEMAS DE INFORMAÇÃO

KELLEN MOURA FERREIRA

APERFEIÇOAMENTO DE UMA FERRAMENTA PARA DAR SUPORTE AO DESENVOLVIMENTO ORIENTADO A MODELO

Serra
2023

KELLEN MOURA FERREIRA

APERFEIÇOAMENTO DE UMA FERRAMENTA PARA DAR SUPORTE AO DESENVOLVIMENTO ORIENTADO A MODELO

Trabalho de Conclusão de Curso apresentado à Coordenadoria do Curso de Sistemas de Informação do Instituto Federal do Espírito Santo, Campus Serra, como requisito parcial para a obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. Maxwell Eduardo Monteiro

Serra
2023

Dedico este trabalho aos meus antigos professores Eric Xavier e Sarah Signorelli, que me incentivaram a buscar evolução dos meus estudos e me orientaram na escolha desse curso

AGRADECIMENTOS

Agradeço a Deus por me ajudar a cada dia e permitir mais essa realização.

Ao meu orientador, Prof. Dr. Maxwell Eduardo Monteiro, por todo apoio, aprendizado, paciência e dedicação.

A todos aqueles que me auxiliaram de algum modo e estiveram comigo durante esta jornada.

RESUMO

O uso do Desenvolvimento Orientado a Modelos (MDD) em conjunto com uma Linguagem Específica de Domínio (DSL) possibilita que os desenvolvedores criem modelos de alto nível sem precisar se dedicar tanto aos detalhes da implementação. Os modelos são automaticamente transformados em códigos executáveis, o que auxilia em um desenvolvimento mais eficaz e produtivo, eliminando a necessidade de um forte vínculo entre o conhecimento do domínio e a elaboração técnica. O aplicativo Gaphor permite criar diversos plugins e, para solucionar os desafios do desenvolvimento de programas utilizando MDD e DSL, foi criado um plugin que auxilia na criação e manipulação dos modelos, bem como na geração automática dos códigos correspondentes. Essa ferramenta já disponibiliza modelos na linguagem UML. No entanto, para melhorar os recursos oferecidos, é adequado ser desenvolvida a opção de desenvolver modelos também na linguagem OntoUML.

Palavras-chave: MDD, DSL, Gaphor.

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

A utilização de programas de computador está em contínua ampliação. Devido à extensa diversidade de uso, os softwares se tornam extremamente complexos (EISHIMA, 2014).

Os sistemas estão evoluindo de forma constante e a abordagem de desenvolvimento utilizada frequentemente resulta em lacunas que interferem na excelência do produto. Isso é evidenciado pela falta de atualização e pela obsolescência dos documentos de requisitos e dos modelos desenvolvidos no começo do processo de fabricação (EISHIMA, 2014).

Diante dessa situação, o desenvolvimento orientado a modelos (Model Driven Development - MDD) é proposto como uma solução para minimizar essas questões e oferecer algumas vantagens, como a produção a partir de modelos (EISHIMA, 2014).

A abordagem de Desenvolvimento Orientado a Modelos visa utilizar modelos como principais elementos do processo de desenvolvimento de software, por meio de transformações que geram diferentes representações em níveis de abstração variados. Por exemplo, a geração de código fonte. O objetivo do MDD é diminuir a distância significativa entre o domínio do problema e o domínio da solução e/ou implementação, mediante ao uso de modelos menos concretos, nos quais preservam os desenvolvedores das dificuldades próprias das plataformas de execução (NETO, 2012).

Um grande benefício do MDD é a capacidade de representar modelos utilizando conceitos menos relacionados aos detalhes da implementação. Além disso, os modelos estão mais próximos do domínio do problema, tornando-os mais simples de compreender, caracterizar e armazenar. Em determinadas circunstâncias, os especialistas de domínio podem assumir a responsabilidade pela criação dos sistemas, em lugar dos profissionais em programação. (NETO, 2012).

O MDD se distingue de outras técnicas de desenvolvimento de software devido ao fato de não descartar o modelo ao longo do processo, mas sim utilizá-lo como base para gerar o código e outros elementos. Para viabilizar a geração de uma aplicação funcional com base em um modelo de domínio, uma das necessidades é empregar linguagens específicas de domínio (DSL) com geradores de código adaptados a ela (SEVERIEN, 2008).

A utilização da DSL pode variar de acordo com o nível de conhecimento técnico do usuário. Quando a DSL é voltada para pessoas com um bom conhecimento técnico, como desenvolvedores, é mais comum o uso de DSLs textuais por causa da familiaridade com a programação em formato de código. No entanto, se a DSL for utilizada, por exemplo, por um gerente de empresa ou um diretor de televisão, é mais adequado utilizar os recursos gráficos que facilitem a interação do usuário. Isso possibilita uma melhor compreensão e utilização da DSL, mesmo por pessoas sem grande experiência técnica (SEVERIEN, 2008).

Por meio da DSL e diagrama de classe, no MDD, é possível gerar o código de domínio do mesmo modo que os atributos e funções que representam as conexões dos modelos. Portanto, a responsabilidade da pessoa que irá desenvolver nesse caso é apenas finalizar as funções necessárias ou fazer atribuições adequadas. Logo, reduz o tempo gasto pelo desenvolvedor e possibilita que não programadores também consigam desenvolver programas (QUITERIO, 2018).

Uma DSL possui elementos para representar os conceitos de um domínio. Portanto, toda DSL é também uma linguagem de modelagem conceitual de um domínio, por exemplo a UML (LUCRÉDIO, 2009). Logo, por meio dessa justificava pode-se considerar que OntoUML que também representa conceitos de modelagem conceitual de múltiplos domínios é uma DSL (BANASZESKI, 2015).

Gaphor é uma das ferramentas de modelagem que possuem suporte para DSL e tem a vantagem de ser Open Source (MOLENAAR; YEAW, 2020).

A proposta deste trabalho é aproveitar uma ferramenta de modelagem pronta e entender como intervir para estender linguagens de modelagem e fazer produzir códigos, permitindo que não programadores sejam capazes de desenvolver programas. A aplicação Gaphor foi escolhida porque permite a criação de diagramas UML e possibilita gerar e configurar novos plugins, por exemplo, adicionar outras linguagens de modelagem, no qual para este propósito é a OntoUML.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

O objetivo geral deste trabalho é desenvolver uma ferramenta para suporte a linguagens específicas de domínio que permita modelar e gerar o código fonte dos modelos feitos.

1.1.2 Objetivos Específicos

Os objetivos específicos reconhecidos com o intuito de alcançar o objetivo geral são os seguinte plguins criados na ferramenta Gaphor:

- Implementar plugin de suporte à criação de modelos OntoUML
- Implementar plugin de geração automática de código Python para modelos de diagrama de classes UML.
- Implementar plugin de geração automática de código Alloy para diagrama de classes UML e modelos OntoUML.

- Validar modelos por meio da linguagem Alloy.

1.2 ESTRUTURA DO TRABALHO

Este trabalho está dividido em seis capítulos. O capítulo 1 apresenta uma contextualização do tema e a introdução do problema pesquisado, junto com a justificativa e os objetivos almejados. O capítulo 2 aborda a fundamentação teórica, no qual inclui conceitos relacionados ao desenvolvimento orientado a modelo, linguagem específica de domínio e modelagem de sistemas. Em sequência, no capítulo 3 é apresentado o método utilizado. O capítulo 4 descreve como foi realizado o desenvolvimento das novas configurações do sistema. No capítulo 5 é apresentado o resultado dos testes que validam o desenvolvimento. Por fim, no capítulo 6 é apresentado a conclusão e trabalhos futuros.

2 REFERENCIAL TEÓRICO

O Model-Driven Development (MDD) ou Desenvolvimento Orientado a Modelos, é um método que se concentra na geração de modelos como elemento principal de seus artefatos para o desenvolvimento de software (EISHIMA, 2014).

Um modelo é um conjunto de partes que formalmente descreve certos elementos, como a segurança, o banco de dados, a interface e o cenário de uso de um sistema. O processo de criação de um software pode envolver uma ou mais construções diferentes (EISHIMA, 2014).

A sugestão de MDD é que o arquiteto de software dedique-se a modelos de nível elevado em vez de trabalhar de forma direta com o código-fonte, prevenindo-se da dificuldade das implementações de cada sistema. Dessa maneira, os modelos começam a ser integrantes do software e não apenas estimado como um simples objeto de documentação, (HEINECK, 2016).

Logo, o padrão no MDD é um componente primordial para o processo de construção e, desse modo, difere do desenvolvimento de software tradicional, onde os padrões são usados para representar o problema e são utilizados apenas como um auxílio para o desenvolvimento. No MDD a sugestão é que esses padrões sejam possibilitados de produzir códigos e não fiquem restritos para que sejam considerados apenas um suporte para o desenvolvedor (NETO; FROTA; FORTES, 2011).

Segundo Lucrédio (2009), as principais vantagens ao utilizar o MDD são:

- Produtividade: aprimoramento no prazo de construção. Tarefas que são realizadas frequentemente são aplicadas nas modificações, economizando prazo e energia que podem ser utilizados em atividades mais significativas;

- Portabilidade e interoperabilidade: códigos têm a capacidade de serem produzidos para diversas plataformas a partir de um único padrão;

- Manutenção e Documentação: Nos métodos tradicionais, os modelos são usados como um artefato inicial e posteriormente descartados. No entanto, no MDD, os padrões são integrantes do software e, por isso, permanecem atualizados de acordo como o software é desenvolvido.

- Comunicação: Os padrões são mais simples de compreender do que o código-fonte, facilitando assim a comunicação entre os desenvolvedores, as partes interessadas e outros colaboradores envolvidos.

- Reutilização: Os modelos podem ser reaproveitados em distintos projetos, nos quais o código fonte pode ser gerado novamente para se adequar ao novo cenário.

- Verificações e otimizações: Os padrões fornecem mais provisão para que validadores semânticos possam ser empregados para diminuir a incidência de falhas, proporcionando implementações mais eficazes.

- Corretude: Geradores de código podem impossibilitar a ocorrência de erros de digitação ou erros que possam ser produzidos por um desenvolvedor manualmente.

Há também algumas desvantagens em utilizar o MDD, sendo elas destacadas por Lucrédio (2009):

- Rigidez: A abordagem MDD pode resultar em um software mais rígido, pois uma parcela significativa do código é gerada automaticamente e fica fora do controle direto do desenvolvedor.

- Complexidade: Os elementos fundamentais para uma abordagem baseada em modelos, como softwares de modelagem, conversões e criadores de código, aumentam a dificuldade do processo, visto que são componentes naturalmente mais complexos de criar e manter.

- Desempenho: Embora algumas melhorias possam ser inseridas de um modo mais abstrato, em geral, geradores de código tendem a incluir excesso de código, resultando em um desempenho inferior ao comparar com um código escrito manualmente.

- Curva de aprendizado: Para construir artefatos específicos do MDD, é necessário contar com profissionais capacitados em desenvolver linguagens, aplicações de modelagem, conversores e criadores de código. Embora não seja muito complicado, aprender essas técnicas exige um estudo e prática que devem possuir muita dedicação.

- Alto investimento inicial: Estabelecer uma infraestrutura que permita a produção e a reutilização de modelos requer investimento em termos de tempo e esforço. Entretanto, os benefícios posteriores são consideráveis, o que torna esse investimento rentável em um período médio a longo prazo.

No intuito de criar modelos, é preciso empregar uma ferramenta de modelagem. Com esse recurso, o desenvolvedor de sistema pode criar modelos que representem os conceitos do domínio. É fundamental que essa ferramenta seja fácil de utilizar e intuitiva para que o processo de modelagem seja eficiente. Simultaneamente, os modelos produzidos devem ser de forma semântica precisos e não devem ser incompletos, uma vez que precisam ser

interpretados por um computador e não por uma pessoa, que pode corrigir erros menores ou preencher campos por conta própria. O componente responsável por incorporar essas propriedades é geralmente uma Linguagem Específica de Domínio (DSL) (LUCRÉDIO, 2009).

Tornar automático as transformações de modelos para código-fonte ou outros modelos é uma tarefa complexa, logo, é necessário utilizar uma ferramenta que permite criar regras de mapeamento de maneira intuitiva, com objetivo de simplificar o processo de criar transformadores (LUCRÉDIO, 2009).

2.1 LINGUAGEM ESPECÍFICA DE DOMÍNIO

Uma linguagem específica de domínio, ou DSL (Domain-Specific Language), é uma linguagem de programação que oferece por meio de símbolos e abstrações adequadas, uma capacidade de expressão direcionada, e geralmente limitada a um campo de problema específico. As DSLs surgiram de modo natural, como um meio de tornar menos complexo a codificação em situações particulares (JUMES; HONDA, 2007).

A concepção de particularização das línguas específicas de domínio geralmente ocasiona que sejam descritivas e sejam consideradas como linguagens para realizar a especificação, bem como linguagens utilizadas para a programação (JUMES; HONDA, 2007). Especificamente, essa definição de ser específico é geralmente alcançada por meio de registros e conceitos adequados a um domínio de problema particular. Segundo alguns especialistas, as DSLs possibilitam descrever de maneira precisa e concisa como realizar processos computacionais específicos em áreas de aplicação especializadas ou com terminologia própria de determinado domínio (JUMES; HONDA, 2007).

Segundo Jumes e Honda (2007), uma linguagem de programação específica de domínio (DSL) é uma linguagem criada para atender a um domínio de aplicação específico, permitindo o desenvolvimento de aplicações completas dentro desse domínio e buscando a semântica específica do mesmo. Ao contrário de linguagens de programação genéricas, as DSLs abrem mão da generalidade em benefício da aproximação a um conjunto específico de problemas.

Apesar de exigir uma análise e design inicial do domínio, uma vez criada, uma DSL pode ser usada e reaproveitada para resolver diversos problemas no mesmo domínio. De acordo com Bragança (2002), uma DSL pode ser vista como uma forma de registrar, representar e fortalecer o design em um formato que estimule a reutilização (BORELLI, 2016).

Há duas categorias de DSLs: as externas e internas. As DSLs externas requerem uma ferramenta para traduzir, interpretar ou compilar suas diretrizes, enquanto as internas são criadas utilizando a estrutura de uma linguagem de programação já existente, o que

permite aproveitar os recursos da linguagem. Uma boa estratégia para iniciar a criação de uma DSL é desenvolver um metamodelo, que é uma linguagem específica para a construção de modelos (BORELLI, 2016).

2.2 MODELAGEM DE SISTEMAS

O modelo conceitual tem a tarefa de estabelecer as entidades que serão controladas pelo sistema e está vinculado ao domínio do problema, não ao da solução. É crucial distinguir o modelo conceitual da arquitetura do software, que é baseada no modelo conceitual, mas faz parte do domínio da solução e serve a uma finalidade distinta (PAULO, 2008).

Para realizar esta tarefa com sucesso, é fundamental elaborar uma especificação de requisitos clara e acessível para o analista e os usuários. O uso de uma linguagem simples e direta é essencial, porém a descrição pode ser desafiadora devido à diferença na linguagem utilizada por cada parte envolvida. Por isso, é preciso alinhar previamente os termos e conceitos para evitar que isso prejudique a compreensão mútua e gere distanciamento entre as partes (PAULO, 2008).

Referente a modelagem visual, os modelos são empregados para simplificar sistemas complexos e apresentá-los a partir de perspectivas específicas. Por exemplo, um projeto arquitetônico é constituído por diferentes modelos que representam diversos aspectos do projeto, como elétrica, hidráulica e estrutura. Isso nos auxilia a compreender de forma mais clara e abrangente esses sistemas (TACLA, s.d).

Projetar um sistema envolve a criação de diversos modelos com diferentes níveis de abstração, representados por uma notação exata e aprimorados cada vez mais. Em certas ocasiões, o foco deve ser na interação dos componentes do sistema, abstraindo-se os detalhes internos, enquanto em outras situações é necessário detalhar o comportamento dos componentes. O propósito é transformar os modelos em algo próximo da implementação, sempre garantindo que os requisitos sejam satisfeitos (TACLA, s.d).

A utilização de diagramas na modelagem visual pode ajudar a manter a consistência entre os diferentes artefatos relacionados ao desenvolvimento de um sistema, como requisitos, projeto e implementação. Dessa forma, a equipe pode gerenciar melhor a complexidade do software, aumentando a eficiência do processo de desenvolvimento (TACLA, s.d).

2.2.1 Linguagem de modelagem conceitual

Para que um modelo seja claramente compreendido, é preciso usar uma linguagem que tenha símbolos e regras bem definidos e seja capaz de descrever todos os aspectos importantes do cenário modelado. Há diversas linguagens que atendem a essas necessidades de maneiras diferentes, entre elas a UML (PEREIRA, 2011).

Entretanto, a linguagem UML pode não incluir características específicas em seus diagramas para manter a organização e evitar excesso de informações. No entanto, essas informações não estão ausentes e a OntoUML surge para melhorar a compreensão dos modelos (BANASZESKI, 2015).

2.2.1.1 UML

A Unified Modeling Language (UML) é uma linguagem de modelo e não deve ser considerada como um método. Em um método, é necessário ter um modelo de linguagem para descrever o projeto e um processo a ser seguido. O modelo de linguagem é a maneira como o método descreve o projeto, e o processo refere-se às etapas que devem ser seguidas para criar o projeto (JUNIOR, 2010).

A utilização da UML serve para especificar, registrar, mostrar e construir sistemas que são orientados a objetos. É útil para descrição de projetos de programas porque fornece um esquema claro que ajuda as pessoas envolvidas em um projeto a se comunicarem. A UML surgiu da mescla de variadas línguas gráficas que se orientam aos objetos, com o intuito de estabelecer processos padronizados e aperfeiçoar o progresso de programas por meio da uniformização de esquemas e figuras (BANASZESKI, 2015).

Reconhecido como um padrão amplamente utilizado na indústria, a UML é empregada na documentação de software orientado a objetos e é utilizada para criar modelos precisos, concisos e sem ambiguidade que representam tanto os aspectos estáticos (estruturais) quanto os dinâmicos (comportamentais) do sistema. Ela oferece símbolos para representar informações, operações e limitações, permitindo a modelagem de conceitos de negócio e sistemas com uma única linguagem. A UML também é uma linguagem que pode ser compreendida por humanos (que possuem entendimento na linguagem) e por máquinas, além de possuir um mecanismo que mapeia a representação textual em XML e a sua representação gráfica. É extensível e adaptável a necessidades específicas, permitindo que seja usada em domínios diversos. Possibilita ainda a modelagem usando técnicas orientadas a objeto, do conceito ao programa funcional (PEREIRA, 2011).

A UML atende às exigências de criação de modelos que se estendem desde os mais singelos e reduzidos até os amplos e complexos, com o auxílio de conexões e compartimentos que possibilitam a separação dos modelos em agrupamentos menores, tanto no domínio quanto em níveis de generalização. Além disso, a UML é capaz de criar modelos de processos, tanto manuais quanto automatizados, sem depender da tecnologia utilizada. É uma linguagem que possibilita visualizar o modelo e isso facilita o compreendimento das equipes da área de desenvolvimento de sistemas, do setor de análise de negócio e também dos clientes. Apesar de não ser uma linguagem de programação de computadores, a UML pode ser empregada para desenvolver código de computador (PEREIRA, 2011).

Portanto, a UML é uma linguagem muito utilizada em análise e projeto de software orientados a objetos e está em constante evolução, com atualizações regulares. De maneira geral, a UML é uma linguagem que pode ser ampliada e aplicada para modelar qualquer tipo de sistema computacional, sendo entendida e utilizada tanto por pessoas quanto por máquinas. Isso a torna útil para satisfazer as demandas dos projetistas e analistas em todas as etapas do ciclo de vida do sistema (PEREIRA, 2011).

2.2.1.2 OntoUML

A OntoUML surgiu como uma expansão da versão 2.0 da UML. Por meio dessa expansão, OntoUML incorporou diversas diferenciações e teoremas ao modelo básico da UML. Um exemplo é a ampliação do elemento Class da UML para abranger as especificidades ontológicas do UFO (Unified Foundational Ontology) (SANTOS, 2015).

O UFO é empregado como um paradigma de orientação para estabelecer os conceitos que uma linguagem de modelagem conceitual robusta deve adotar, de modo a conferir àquelas construções linguísticas que representam esses conceitos uma semântica condizente com a realidade (BANASZESKI, 2015).

A base ontológica UFO é separada em três partes: UFO-A, UFO-B e UFO-C. A Ontologia de Endurantes (UFO-A) representa uma teoria sólida e bem definida, caracterizada por fazer uso de uma lógica modal altamente expressiva e possuindo forte respaldo empírico comprovado por testes em psicologia cognitiva. A maior parte dos estudos na área de ontologia tem se concentrado nos fragmentos UFO-B (Ontologia de Perdurantes), que inclui conceitos como situações, mudanças e ocorrências, entre outros, e UFO-C (Ontologia de Entidades Sociais e Intencionais), que inclui conceitos como atividades e atores sociais, entre outros (BANASZESKI, 2015).

Ao discutir o uso de linguagens de modelagem ontológica conceitual baseadas na fundamentação ontológica, foi proposto o desenvolvimento de uma linguagem de modelagem conceitual que usa as propriedades ontológicas sugeridas pela ontologia UFO-A como elementos básicos de modelagem. A OntoUML, passou por um processo de construção que envolveu a adaptação do metamodelo da UML 2.0 para assegurar sua correspondência com a estrutura estabelecida pela UFO-A. Em seguida, a ontologia de base é formalizada e incluída no metamodelo de linguagem, mediante a aplicação de limitações formais no mesmo (GUIZZARDI et al., s.d).

Segundo (SUCHÁNEK, 2018), na OntoUML há 12 tipos de estereótipos de classe, sendo eles: Kind; Subkind; Phase; Role; Collective; Quantity; Relator; Category; PhaseMixin; RoleMixin; Mixin; Mode; Quality. Também possui diferentes tipos de estereótipos de relacionamento, são eles: Formal; Material; Mediation; Characterization; Derivation; Structuration; Part-Whole; ComponentOf; Containment; MemberOf; SubCollectionOf;

SubQuantityOf; Dependency; Generalization.

Referente aos estereótipos de relacionamento, alguns possuem particularidades no qual a relação pode ser considerada como parte compartilhada (branco ◊) ou exclusiva (preto ⧫) (SUCHÁNEK, 2018).

As limitações de integridade presentes no metamodelo OntoUML estabelecem formas legítimas de combinar representações conceituais básicas com representações que possuem distinções ontológicas (BANASZESKI, 2015).

2.2.2 Verificação de modelos conceituais

De acordo com Silva (2014), a validação de um modelo conceitual significa verificar se ele representa com precisão o domínio que se pretende modelar. Em outras palavras, é importante garantir que o modelo esteja alinhado com a realidade que ele representa. Para fazer isso, é necessário verificar se as informações que foram incluídas no modelo estão corretas, se o modelo é consistente, se as suposições que foram feitas estão corretas e se o modelo é capaz de explicar e prever os comportamentos e fenômenos do mundo real que ele representa. A validação é uma parte importante do processo de modelagem, pois ajuda a garantir que o modelo possa ser usado com segurança para tomar decisões e realizar previsões no mundo real.

2.2.2.1 Alloy

Em 1997, no MIT, uma equipe liderada por Daniel Jackson desenvolveu a Alloy, uma linguagem textual que tem a capacidade de descrever sistemas mais complexos e seus comportamentos e restrições. Ao contrário de muitas linguagens descritivas, a Alloy foi projetada para ser analisável, possibilitando verificar a exatidão e coerência das características escritas na linguagem, no qual essa validação é feita por meio da aplicação Alloy Analyzer (CAPELO, 2018).

O Alloy Analyzer é uma ferramenta integrada que suporta a linguagem Alloy. Ele é capaz de encontrar estruturas que satisfaçam as restrições de um modelo, funcionando como um solucionador ou "model finder". A ferramenta tenta encontrar uma atribuição de valores a variáveis que torne uma fórmula lógica escrita na linguagem Alloy verdadeira. Além disso, ela pode ser usada para gerar exemplos de estruturas e avaliar as propriedades do modelo, gerando contraexemplos. Dessa forma, o Alloy Analyzer é uma ferramenta valiosa tanto para a exploração quanto para a validação de modelos (GUIMARÃES, 2015).

A linguagem Alloy é fundamentada por lógica de primeira ordem e álgebra relacional, fornecendo uma sintaxe expressiva que permite a modelagem detalhada de softwares reais. Porém, a simplicidade da linguagem também possibilita a realização automática das

análises (CORDEIRO, 2017).

Por meio da Alloy é possível simular, verificar e realizar depuração, pois essas 3 formas de análise para modelo são disponibilizadas pela linguagem. Durante as simulações, o analisador busca por uma instância que esteja em conformidade com os critérios do modelo no qual está sendo analisado. Durante as verificações, o analisador busca um contraexemplo que não satisfaça uma determinada propriedade do sistema. Além disso, o analisador pode ajudar na depuração da especificação, destacando quais contenções do modelo apresentam conflitos (CORDEIRO, 2017).

A composição básica de um modelo em Alloy é formada por várias partes: Primeiro, existem declarações de assinaturas, que são definidas com a palavra sig e representam conjuntos de elementos ou relações. Em seguida, há seções de restrições, reconhecidas pelas palavras fact, fun e pred, que contêm diferentes modos de se expressar e restringir. Além disso, existem seções assertivas, marcadas pela palavra assert, que expressam propriedades desejadas do modelo. Também existem seções de comandos, que são identificados pelas palavras run e check, e contém diretivas para que o Alloy Analyzer realize procedimentos de análise (GUIMARÃES, 2015).

A hierarquia de classificação é definida através das declarações de assinatura e que tem a possibilidade de ser expandida para outros conjuntos distintos utilizando a palavra reservada extend. A inclusão da palavra reservada "abstract"em uma assinatura ampliada denota que todos os elementos do conjunto estão contidos em suas respectivas extensões (GUIMARÃES, 2015).

Dessa forma, um modelo Alloy é composto por um grupo de limitações lógicas que são indicadas por meio de seções sig. Ao instanciar esse modelo no Alloy Analyzer, são criados elementos com base nas assinaturas, seguindo as limitações lógicas daquele modelo. Isto é, uma assinatura no estado do modelo apresenta um grupo de elementos no estado de instância (GUIMARÃES, 2015).

Devido à sua flexibilidade e capacidade de análise automática, a linguagem tem sido aplicada em diversas áreas, incluindo tanto aplicações práticas quanto de exploração (CORDEIRO, 2017).

2.2.3 Ferramenta de modelagem

Para a criação de modelos, é imprescindível ter à disposição uma ferramenta de modelagem que possibilite ao engenheiro de software produzir modelos que representem os conceitos do domínio. Essa ferramenta deve ser fácil de utilizar e ser de forma intuitiva. No entanto, é fundamental que os modelos gerados sejam semanticamente precisos e completos, tendo em vista que precisam ser interpretados por máquinas, que não são capazes de corrigir

equívocos ou preencher automaticamente lacunas (LUCRÉDIO, 2009).

2.2.3.1 Gaphor

O Gaphor é uma aplicação de modelagem que suporta UML, SysML, RAAML e C4. O propósito dessa ferramenta é facilitar a utilização por meio de uma interface simples, sem comprometer sua capacidade de ser um software robusto. Com a implementação do modelo de dados UML 2, o Gaphor se diferencia como uma ferramenta completa para além do simples desenho de diagramas. Utilizando o Gaphor, é possível ter uma rápida visualização de diferentes aspectos de um sistema, bem como criar modelos de alta complexidade (MOLENAAR; YEAW, 2020).

Uma ferramenta multi plataforma, Gaphor é compatível com o sistema operacional Windows, MacOS e Linux. Possui código aberto, no qual foi desenvolvido com a linguagem Python e não possui nenhum bloqueio do fornecedor, além de ser disponível de acordo com uma licença do Apache 2 (MOLENAAR; YEAW, 2020).

Independentemente de ser um modelador ocasional que está documentando um projeto ou um especialista em Desenvolvimento Orientado a Modelos, a Gaphor oferece soluções completas para as necessidades do usuário (MOLENAAR; YEAW, 2020).

A Gaphor tem a capacidade de ser expandida, permitindo integrar um gerador de código ou exportar diagramas para fins de documentação. É possível ainda criar extensões personalizadas e acessá-las tanto pela interface gráfica do usuário quanto pela linha de comando (MOLENAAR; YEAW, 2020).

2.3 EQUIVALÊNCIA ENTRE MODELOS CONCEITUAIS E CONSTRUÇÕES SEMÂNTICAS NAS LINGUAGENS DE PROGRAMAÇÃO

A Orientação a Objetos (OO) é uma metodologia para gerar representações que definem a área de atuação de um sistema. Quando elaborados adequadamente, sistemas baseados em OO são adaptáveis a alterações, têm organizações bem estabelecidas e viabilizam a criação e utilização de partes reaproveitáveis. Modelos orientadas a objetos são executados de maneira prática por meio de uma linguagem de programação orientada a objetos (BRAGA; MASIERO, 2007).

Há várias linguagens baseadas em métodos de OO, como Python, Java, Ruby, etc. Por meio dessas linguagens, é possível adicionar conceitos de OO (classe, herança, objeto, entre outros) compatíveis com o código e permitir também a manutenção (BRAGA; MASIERO, 2007).

Modelos orientados a objetos são executados por meio de uma linguagem de POO (progra-

mação orientada a objetos). A engenharia de software orientada a objetos não se resume apenas à utilização dos recursos de uma linguagem de programação; na realidade, implica em saber empregar de maneira eficiente os diversos métodos de modelagem orientada a objetos (RAMOS, 2019).

2.4 TRABALHO CORRELATO

Há trabalhos semelhantes a este, entre eles o TCC de Eishima (2014) no qual aborda o tema de Desenvolvimento Orientado a Modelos, porém tem como intuito expor um metamodelo para a produção de programas com foco em modelos orientados a software, que aprimore a estruturação, eficiência e excelência do processo de produção de sistemas em organizações (EISHIMA, 2014).

Outro trabalho similar a este é o de Heineck (2016), que apresenta sobre o Desenvolvimento Orientado a Modelos no Domínio de Robótica, no qual por meio de uma Revisão Sistemática da Literatura (RSL) desenvolve uma contribuição para o melhor entendimento aos projetos que utilizam MDD em robótica. Para a realização dessa atividade foram escolhidos 86 artigos para revisar e levantados 4 pontos, no qual para cada questão eram analisadas as respostas considerando todos os artigos selecionados. Obteve-se a conclusão que o uso de MDD em robótica pode contribuir com melhorias na etapa do desenvolvimento de sistema, seja por meio de diagramas UML ou por criação de extensão, padrões ou outras linguagens próprias para esse objetivo (HEINECK, 2016).

A tese feita por Lucrédio (2009) também é semelhante a este trabalho, visto que possui como tema “Uma Abordagem Orientada a Modelos para Reutilização de Software”, no qual apresenta a importância da reutilização de sistemas e como o MDD pode auxiliar para esse propósito. Para a realização da dissertação foi feita uma avaliação de estudos empíricos e uma descrição da pesquisa destinada ao determinado assunto. Concluiu-se que por meio da pesquisa realizada é possível juntar diversas técnicas de reutilização e MDD de um modo original, no qual o suporte oferecido é mais viável do que a utilização de métodos isolados (LUCRÉDIO, 2009).

3 MÉTODO

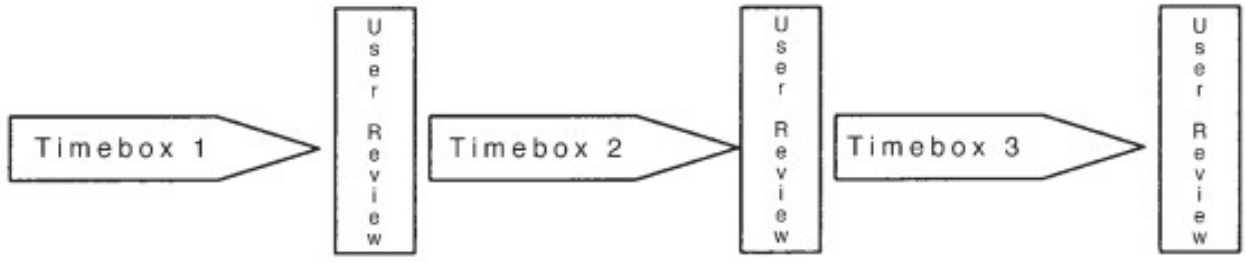
O presente trabalho utiliza o método de prototipação rápida com validação através de estudos de caso, no qual é realizado a análise do aplicativo Gaphor e na sequência, o desenvolvimento de novas funcionalidades, conforme os objetivos deste trabalho.

O método escolhido teve como objetivo um sistema de alta qualidade, baixo custo, desenvolvimento e entregas rápidas. Logo, permitiu entregar aplicações funcionais com menores prazos e investimento (DAVIES et al., 1999). A abordagem selecionada também considerou questões de desenvolvimento padrão e teste de software. As características apresentadas possibilitaram a construção de plugins contendo as funcionalidades pretendidas.

O desenvolvimento das novas funcionalidades do sistema apresentado ocorreram por meio de uma análise dos códigos-fonte disponíveis na aplicação, os quais foram utilizados como base para a criação dos novos scripts e diretórios. Esses elementos foram devidamente organizados, seguindo o padrão já estabelecido pela ferramenta.

A abordagem de prototipação rápida possui como um de seus componentes o "Time Boxing"que é uma prática de controle de projeto, no qual consiste em delimitar o escopo do projeto priorizando o desenvolvimento e definindo prazos de entrega ou "timeboxes". Essa técnica é ilustrada na Figura 1, onde representa a relação entre as caixas de entrada e a revisão de usuários, ou seja, para cada tarefa realizada deverá haver a examinação da mesma antes de passar para a próxima.

Figura 1 – Caixas de entrada e revisão de usuários



Fonte: Davies et al. (1999)

Neste trabalho, serão utilizadas 4 timeboxes, seguindo os critérios estabelecidos, que consistem em:

- Elaboração de suporte para criação de diagramas em OntoUML.
- Criação de plugin para automatização de diagrama UML para código em Python.
- Desenvolvimento de plugin de automatização de diagrama UML para código em Alloy.

- Construção de plugin para conversão automática de diagrama OntoUML para código em Alloy.

A validação da pesquisa realizada foi conduzida por meio de estudos de caso, o que possibilita verificar a eficácia da ferramenta e dos processos executados. Logo, para constatar se a aplicação atende aos aspectos desejados e se os procedimentos estão em conformidade com as requisições, serão feitos testes de validação das novas funcionalidades efetuadas.

Os testes de validação foram realizados de acordo com os critérios estabelecidos pelas fontes primárias das pesquisas, por exemplo, para os exames de verificação de diagramas OntoUML foi utilizado como base a documentação disponibilizada no site de documentação *Read the Docs* que pode ser acessado por meio do link https://ontouml.readthedocs.io/en/latest/.

4 DESENVOLVIMENTO

Neste capítulo é apresentado como foram desenvolvidos os plugins na ferramenta Gaphor, para que fosse possível concretizar os objetivos descritos no Capítulo 1.

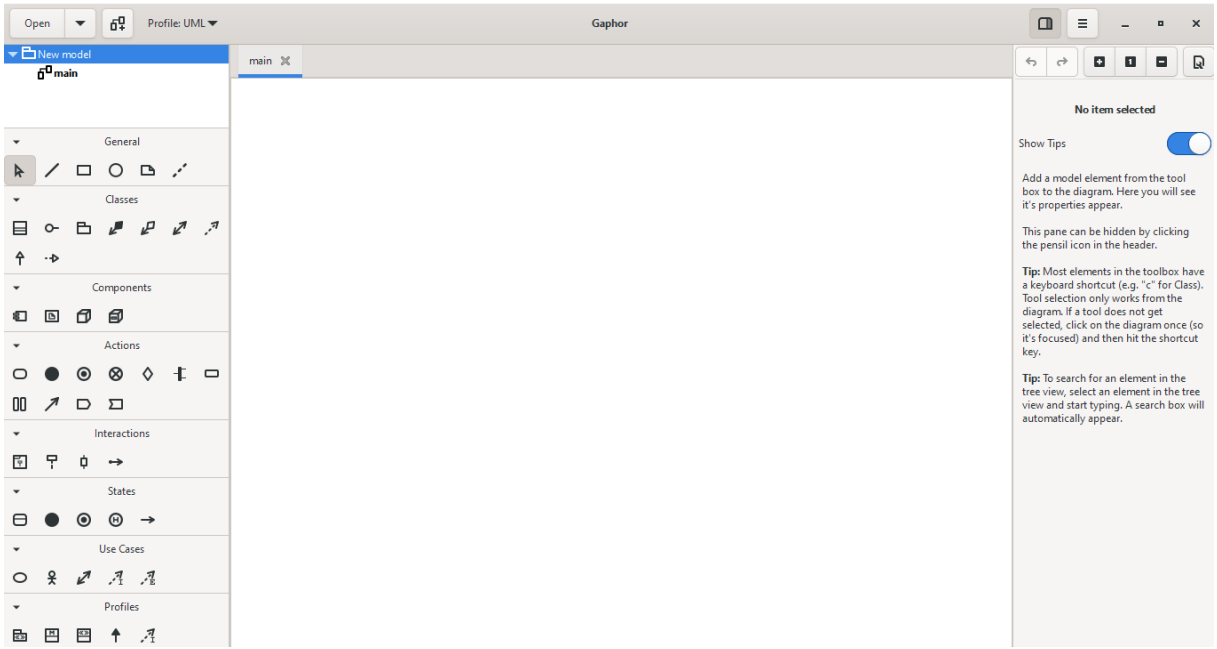
Vale ressaltar que os códigos utilizados para o desenvolvimento estão disponíveis no Github e podem ser acessados por meio do link: https://github.com/kellenmoura/tcc_bsi_kellen.

4.1 VISÃO GERAL

As funcionalidades da ferramenta Gaphor descritas nesta seção são referentes a sua versão 2.2.2, na qual o aplicativo disponibiliza a criação de modelos nas linguagens: UML, SysML e C4 model.

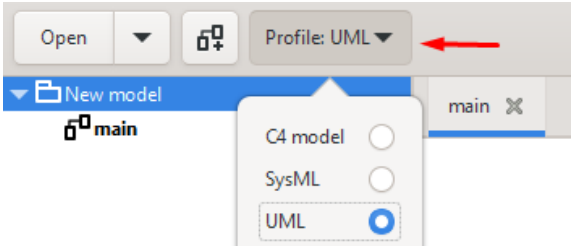
Na Figura 2 é apresentada a tela inicial do sistema, visto que a linguagem selecionada para desenvolvimento de diagramas é a UML.

Figura 2 – Página Principal do Aplicativo Gaphor



Fonte: Print da tela do aplicativo Gaphor.

Figura 3 – Campo para seleção de linguagem

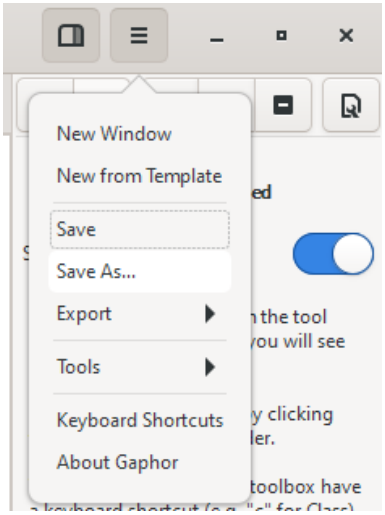


Fonte: Print da tela do aplicativo Gaphor.

Entretanto é oferecido ao usuário a opção de selecionar outra linguagem fornecida na aplicação, dado que para executar essa ação é preciso se direcionar para o canto superior esquerdo no campo escrito “Profile”, onde ao efetuar um clique serão exibidas às demais opções disponíveis, conforme mostra na Figura 3. Ao escolher as linguagens podem haver diferentes seções e elementos fornecidos, visto que cada uma possui suas próprias particularidades.

Gaphor oferece também outras possibilidades, por exemplo, exportar o modelo para pdf, png, svg e xmi. Além de salvar o arquivo com a extensão “.gaphor” no qual pode ser aberto diretamente na ferramenta apresentando o diagrama ou abrir o arquivo com um editor de texto visto que é também um xml. No intuito de salvar a representação, o usuário deve se direcionar para o canto superior direito da aplicação e selecionar a opção “save” ou “save as”, assim como mostra a Figura 4. Entretanto, se o objetivo do usuário for exportar a reprodução, é necessário que clique em “export” e logo em seguida escolha a exportação requerida, conforme Figura 5.

Figura 4 – Opções de Salvar o Documento



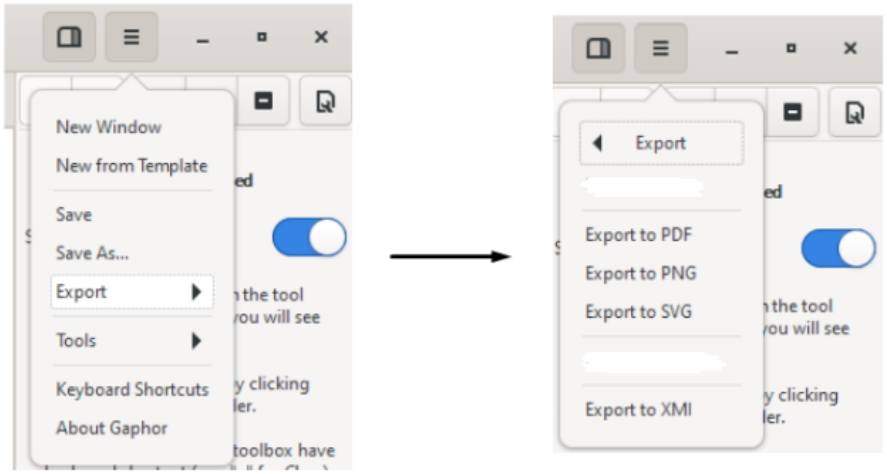
Fonte: Print da tela do aplicativo Gaphor.

A instalação da ferramenta Gaphor pode ser feita de forma comum para usuários finais, no qual é necessário apenas fazer o download do arquivo executável e seguir os procedimentos. Por ser uma aplicação open-source, para os desenvolvedores há a alternativa de baixar o código-fonte da aplicação por meio do repositório do github e é disponibilizado um tutorial de instalação para diferentes sistemas operacionais, onde possibilita que o programador faça um correto uso da transferência do código aberto e execute o aplicativo sem causar possíveis problemas futuros.

4.2 Plugin para criar modelos OntoUML

Para adicionar uma nova linguagem de modelagem ao aplicativo, foi necessário criar uma nova pasta, nomeada por “UML3”, ao diretório do código-fonte do gaphor. Em seguida

Figura 5 – Opções de Exportação do Documento



Fonte: Print da tela do aplicativo Gaphor.

realizar nos devidos arquivos as importações referenciando a pasta criada e adicionar a UML3 os códigos com suas particularidades.

O arquivo “modelinglanguage.py” é responsável por identificar o nome da nova linguagem configurada no sistema, neste caso relacionado a pasta “UML3” foi escolhido o nome “OntoUML” como mostra na linha 12 do Quadro 1.

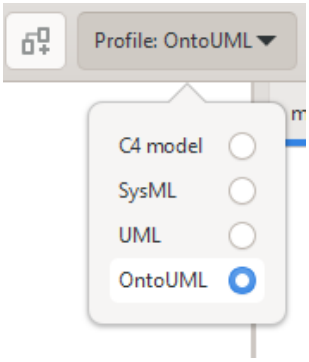
Quadro 1 – Código do arquivo: modelinglinguage.py

```
1  import gaphor.UML3.propertypages  # noqa
2  from gaphor.abc import ModelingLanguage
3  from gaphor.core import gettext
4  from gaphor.diagram.diagramtoolbox import ToolboxDefinition
5  from gaphor.UML3 import diagramitems, uml3
6  from gaphor.UML3.toolbox import uml3_toolbox_actions
7
8
9  class UML3ModelingLanguage(ModelingLanguage):
10     @property
11     def name(self) -> str:
12         return gettext("OntoUML")
13
14     @property
15     def toolbox_definition(self) -> ToolboxDefinition:
16         return uml3_toolbox_actions
17
18     def lookup_element(self, name):
19         return getattr(uml3, name, None)
20
21     def lookup_diagram_item(self, name):
22         return getattr(diagramitems, name, None)
```

Fonte: Elaborado pelo autor (2023).

Por meio desse script e de outras importações devidamente realizadas, foi possível encontrar na aplicação a nova opção de linguagem “OntoUML”, conforme apresentado na Figura 6.

Figura 6 – Linguagem OntoUML Disponibilizada



Fonte: Elaborado pelo próprio autor (2023).

No intuito de possuir um correto desempenho do plugin, foi preciso configurar suas propriedades, logo foram adicionados as classes de estereótipos, as relações e declarados as possíveis conexões entre as relações e os estereótipos existentes. Para realizar estas atividades, foram criadas dentro de UML3 novas pastas, entre elas: “classesonto” e “profile”. Em “classesonto”, estão contidos os script relacionados aos estereótipos e como é organizado o menu lateral do OntoUML. Por exemplo, o Quadro 2 exibe o respectivo código da seção “General”, no qual foi utilizado como base o código disponibilizado para o padrão da UML, que pode ser visualizada na aplicação como na Figura 7. As mesmas informações podem ser consideradas para os demais campos.

Além disso, possui também o arquivo “classesontoconnect.py” referente às conexões das relações, no qual apresenta-se um demonstrativo do objeto de relação “Member Of” no modo exclusivo e por isso denominado na aplicação como “Member Of Black” (por causa do símbolo que é pintado de preto, entretanto no modo compartilhado o símbolo não é pintado e é nomeado apenas como “Member Of”), dado que essa relação pode conectar qualquer um dos estereótipos seguindo a restrição de que a extremidade seja um “Stereotype Collective”, conforme configurado no código mostrado no Quadro 3 que é respectivo ao exemplo exibido na Figura 8, onde o conector cumpre o que foi restrito.

Já em “profile” que contém os scripts referentes às relações é configurado o modo em que determinado objeto deverá se comportar, como por exemplo, o nome que deverá aparecer e o símbolo que será utilizado para representar as extremidades da conexão. A Figura 9 apresenta a relação “Characterization” que possui o nome destacado e uma das pontas como forma de navegação.

Para o desenvolvimento do menu de opções do OntoUML foi necessário criar novos ícones associados a cada um dos novos objetos disponibilizados, no qual para realizar essa configuração foram adicionados a um determinado diretório presente na pasta “ui”, juntamente com as outras representações já existentes das demais linguagens, arquivos com extensão “svg” que foram gerados em um programa externo para a criação de cada

Quadro 2 – Parte do código do arquivo: modelinglinguage.py

```
ontogeneral = ToolSection (
        gettext("General"),
        (
            ToolDef(
                "toolbox-pointer",
                gettext("Pointer"),
                "gaphor-pointer-symbolic",
                "Escape",
                item_factory=None,
            ),
            ToolDef(
                "toolbox-package",
                gettext("Package"),
                "gaphor-package-symbolic",
                "p",
                new_item_factory(
                    diagramitems.PackageItem, UML.Package, config_func=
                        namespace_config,
                ),
                handle_index=SE,
            ),
            ToolDef(
                "toolbox-comment",
                gettext("Comment"),
                "gaphor-comment-symbolic",
                "k",
                new_item_factory(general.CommentItem, UML.Comment),
                handle_index=SE,
            ),

        ),
    )
```

Fonte: Elaborado pelo autor (2023).

um dos símbolos. Apresenta-se na Figura 10 a seção do aplicativo que estão presentes as reproduções citadas.

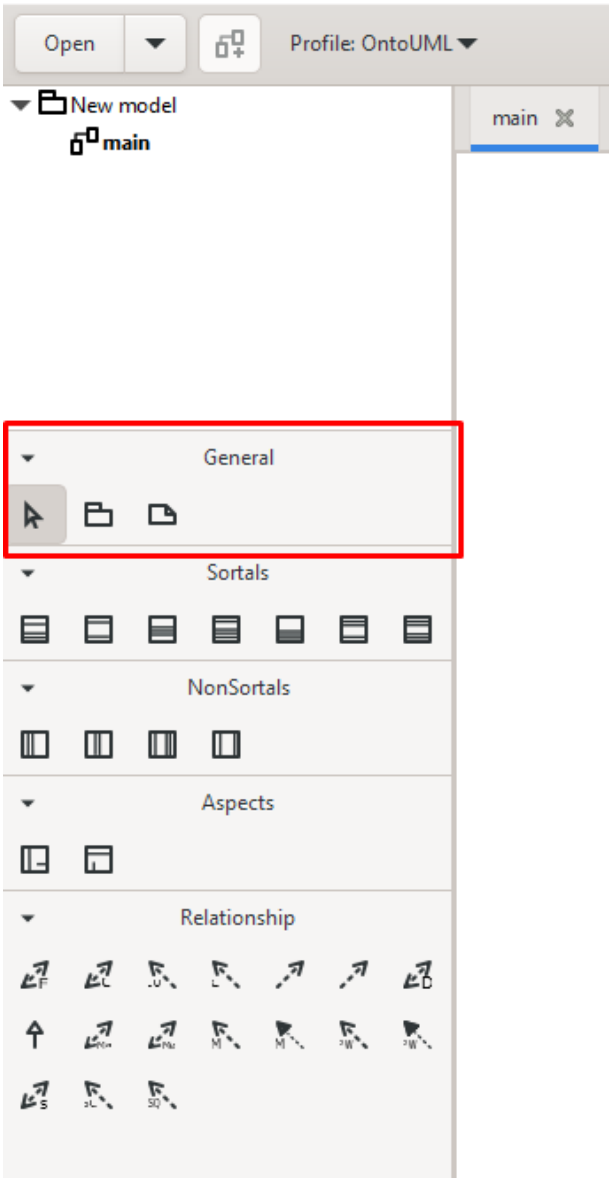
Cada uma das novas configurações feitas se complementam e para o desenvolvimento dos ajustes citados foram seguidos exemplos de scripts já prontos e disponibilizados no código-fonte da ferramenta.

4.3 Plugin para exportar modelos UML para Python

No intuito de desenvolver o plugin de exportação de modelos UML para Python, foi criado uma nova pasta, nomeada por “python_export”, no armazenamento de “plugins”. Também foi gerado o arquivo “___init___.py”, no qual está o script do plugin.

Para realização do código dessa configuração, foi utilizado o arquivo XMI gerado no Gaphor por meio de um serviço já disponibilizado na aplicação, logo, todas as informações necessárias do XMI foram extraídas e armazenadas em dicionários no “___init___.py”,

Figura 7 – Menu de Opções da linguagem OntoUML



Fonte: Elaborado pelo próprio autor (2023).

possibilitando o tratamento desses dados. O Quadro 4 mostra um trecho do script onde foi feita a condição para que se achasse uma classe no arquivo base elaborasse o dicionário.

Após o processo de organização dos dados, como a pequena demonstração apresentada no Quadro 4, sucedeu-se que fosse possível fazer a criação do novo arquivo conforme solicitado pelo usuário (código python que representa o modelo UML), conforme mostra uma parte do script de configuração no Quadro 5.

Por meio do desenvolvimento especificado anteriormente foi gerado o plugin, no qual já pode ser encontrado no aplicativo, assim como a Figura 11, e ao ser solicitada essa opção será gerada a exportação desejada.

Após todo o procedimento que envolve a configuração dos códigos, o arquivo XMI gerado

Quadro 3 – Parte do código do arquivo: classesontoconnect

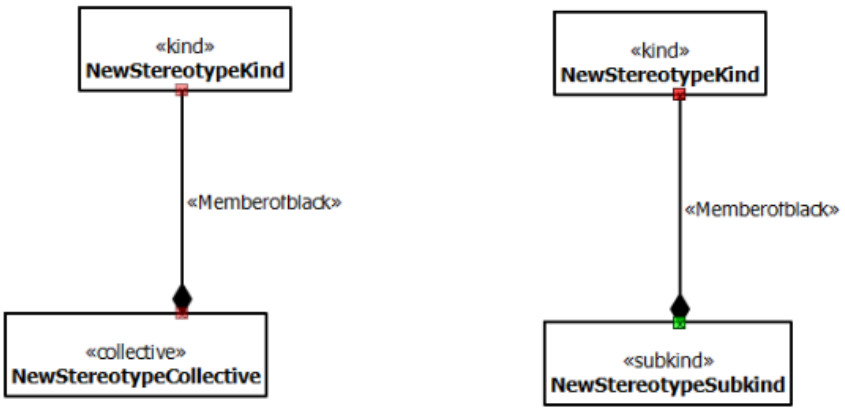
```
@Connector.register(Classified, MemberofblackItem)
class MemberofblackConnect(UnaryRelationshipConnect):

    line: MemberofblackItem

    def allow(self, handle, port):
        line = self.line
        subject = self.element.subject
        tipo = str(type(subject))
        result = False

        #condicoes para saber qual é o stereotype
        if tipo == "<class 'gaphor.UML3.uml3.StereotypeKind'>":
            nome = (UML3.StereotypeKind)
        elif tipo == "<class 'gaphor.UML3.uml3.StereotypeSubkind'>":
            nome = (UML3.StereotypeSubkind)
        elif tipo == "<class 'gaphor.UML3.uml3.StereotypePhase'>":
            nome = (UML3.StereotypePhase)
        elif tipo == "<class 'gaphor.UML3.uml3.StereotypeRole'>":
            nome = (UML3.StereotypeRole)
        elif tipo == "<class 'gaphor.UML3.uml3.StereotypeCollective'>":
            nome = (UML3.StereotypeCollective)
        elif tipo == "<class 'gaphor.UML3.uml3.StereotypeQuantity'>":
            nome = (UML3.StereotypeQuantity)
        elif tipo == "<class 'gaphor.UML3.uml3.StereotypeRelator'>":
            nome = (UML3.StereotypeRelator)
        elif tipo == "<class 'gaphor.UML3.uml3.StereotypeCategory'>":
            nome = (UML3.StereotypeCategory)
        elif tipo == "<class 'gaphor.UML3.uml3.StereotypePhasemixin'>":
            nome = (UML3.StereotypePhasemixin)
        elif tipo == "<class 'gaphor.UML3.uml3.StereotypeMixin'>":
            nome = (UML3.StereotypeMixin)
        elif tipo == "<class 'gaphor.UML3.uml3.StereotypeRolemixin'>":
            nome = (UML3.StereotypeRolemixin)
        elif tipo == "<class 'gaphor.UML3.uml3.StereotypeMode'>":
            nome = (UML3.StereotypeMode)
        elif tipo == "<class 'gaphor.UML3.uml3.StereotypeQuality'>":
            nome = (UML3.StereotypeQuality)


        if handle is line.head:
            result = isinstance(subject, UML3.StereotypeCollective)


        if handle is line.tail:
            result = isinstance(subject, nome)
```

Fonte: Elaborado pelo autor (2023).

como base é removido da memória do dispositivo, logo, permanece apenas o de extensão ".py" que foi requisitado.

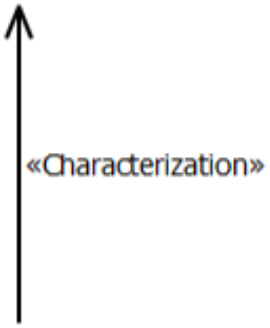
A Figura 12 apresenta um simples exemplo de um diagrama em UML desenvolvido no Gaphor e o código-fonte do modelo gerado em python por meio do plugin.

Figura 8 – Conexão aceita apenas onde a extremidade era o Stereotype Collective



Fonte: Elaborado pelo próprio autor (2023).

Figura 9 – Relação de Caracterização



Fonte: Elaborado pelo próprio autor (2023).

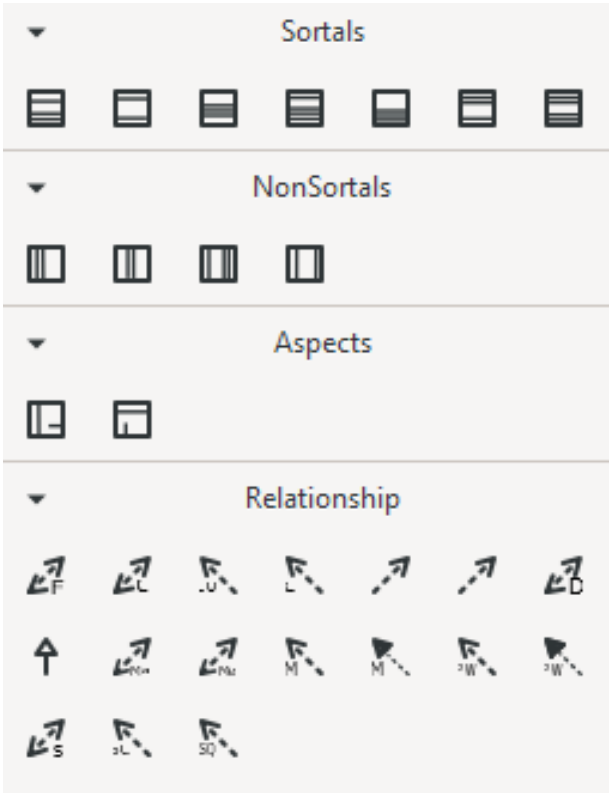
4.4 Plugin para exportar modelos UML/OntoUML para Alloy

Com o objetivo de desenvolver um plugin que permita a exportação de modelos UML/OntoUML para Alloy, uma nova pasta intitulada "alloy_export"foi criada dentro do armazenamento de "plugins". Além disso, um arquivo chamado "___init___.py"foi gerado contendo o script do plugin.

Para implementar essa configuração, foi utilizado um arquivo XML gerado no Gaphor por meio de um serviço que já estava disponível na aplicação, no qual ao salvar o diagrama é criado um arquivo com extensão “.gaphor” que pode ser reaberto novamente no aplicativo para algum ajuste ou visualização, entretanto, ao abrir o documento em um editor de texto é retornado o texto XML do mesmo contendo todas as informações necessárias extraídas do diagrama. Logo, os dados obtidos foram armazenados em dicionários no "___init___.py", permitindo que os dados fossem manipulados.

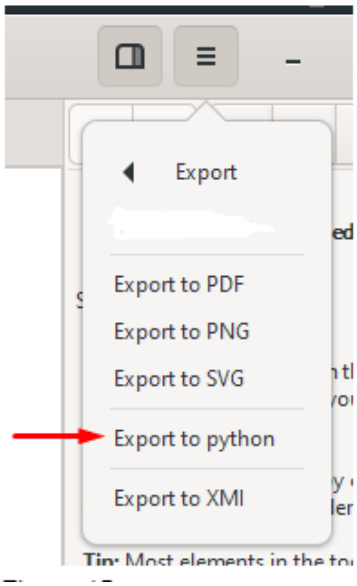
No intuito de manipular e tratar o arquivo XML foi importado a biblioteca “ElementTree” e utilizado o método “getroot”, que facilitou o modo de separar os elementos requeridos,

Figura 10 – Ícones representados por meio dos arquivos em svg



Fonte: Elaborado pelo próprio autor (2023).

Figura 11 – Plugin de exportação do Python disponibilizado



Fonte: Elaborado pelo próprio autor (2023).

assim como mostra o Quadro 6, no qual um trecho do script apresenta a forma que foi feita a captura das classes existentes no documento gerado por meio do diagrama.

Depois de realizar a organização dos dados, assim como evidenciado na breve demonstração apresentada anteriormente, foi possível efetuar a elaboração de um novo arquivo de acordo com a solicitação do usuário (código Alloy que representa o modelo UML ou OntoUML),

Quadro 4 – Parte do código do plugin de exportação para Python

```
if (line.find("<UML:Class XMI:id=") != -1):
    class_id = ''
    class_name = ''
    class_abstract = ''

    i = line.find("\"") + 1
    while (line[i] != '\"'):
        class_id += line[i]
        i += 1

    i = line.find("name=\"") + 6
    while (line[i] != '\"'):
        class_name += line[i]
        i += 1

    i = line.find("isAbstract=\"") + 12
    while (line[i] != '\"'):
        class_abstract += line[i]
        i += 1

    classes = {}
    classes['id'] = class_id
    classes['name'] = class_name
    classes['abstract'] = class_abstract
    classes['property'] = []
    dic['class'].append(classes)
```

Fonte: Elaborado pelo autor (2023).

Quadro 5 – Pequena parte do código do plugin de exportação para Python

```
arquivo_py.writelines("class " + dic_class['name'] + ":\n")

arquivo_py.writelines("\tdef __init__(self")

for dic_property in dic_class['property']:
    if dic_property['name'] != '':
        arquivo_py.writelines(", " + dic_property['name'])

arquivo_py.writelines("):\n")
```

Fonte: Elaborado pelo autor (2023).

conforme ilustrado em uma parte do script de configuração presente no Quadro 7, no qual inicia a escrita do script e avalia a condição de ser OntoUML. Caso seja UML, o desenvolvimento do texto começa em uma verificação posterior.

A partir da execução do processo descrito anteriormente, o plugin foi criado e está disponível na ferramenta, conforme ilustrado na Figura 13. Ao suceder a solicitação, a exportação desejada é gerada.

Após todo o processo que engloba a configuração dos códigos, o arquivo XML (elaborado por meio do documento com formato “.gaphor”), utilizado como base, é eliminado da

Figura 12 – Modelo UML e código-fonte gerado em Python por meio do plugin

Fonte: Elaborado pelo próprio autor (2023).

Quadro 6 – Parte do código que separa os dados em dicionários

```
    for property_id in root.findall("./{http://gaphor.sourceforge.net/model
        }Property"):
        propertylst = []
        dic2['property' + str(i)] = []
        properties = {}
        properties['property_id'] = property_id.attrib
        prop2id = property_id.attrib['id']

        for class_id in property_id.findall("./{http://gaphor.sourceforge.
            net/model}type/{http://gaphor.sourceforge.net/model}ref"):
            properties['class_id'] = class_id.attrib
```

Fonte: Elaborado pelo autor (2023).

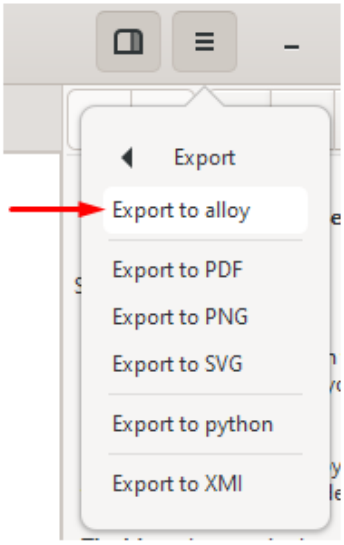
Quadro 7 – Parte do código que inicia o arquivo em Alloy

```
with open(caminhoc, "w") as arquivo:
    arquivo.writelines("module " + dic['module'].replace(" ", ""))

    if auxverificaonto == True:
        arquivo.writelines("\nopen util/ordering[State] as state")
        arquivo.writelines("\nopen util/relation\n")

        #criando assinatura kind
        if len(lstkind) > 0:
            for i in lstkind:
                arquivo.writelines("\nsig " + i + "{}")
```

Fonte: Elaborado pelo autor (2023).

memória do dispositivo, deixando apenas o arquivo de extensão ".txt"que contém o código Alloy requisitado pelo usuário.

Figura 13 – Plugin de exportação para Alloy disponibilizado



Fonte: Elaborado pelo próprio autor (2023).

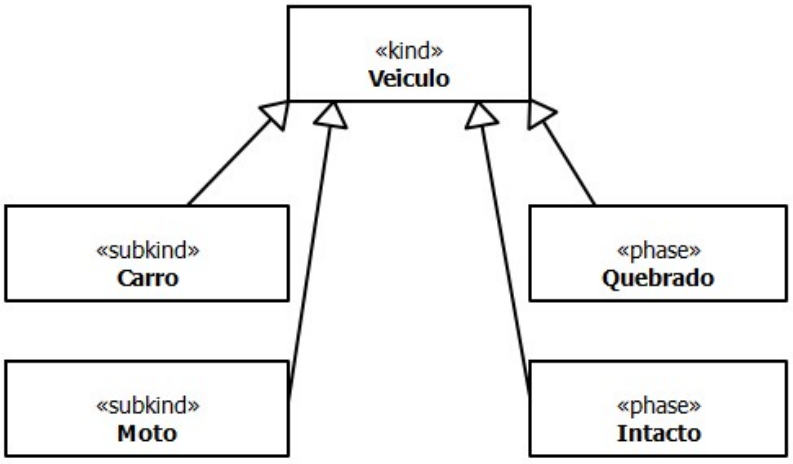
5 TESTES DE VALIDAÇÃO

Neste capítulo são apresentados os testes realizados para a validação dos plugins indicados no capítulo 3.

5.1 MODELO ONTOUML

Por meio de plugins gerados, a ferramenta permite a criação de diagramas OntoUML assim como mostra o exemplo simples da Figura 14.

Figura 14 – Diagrama OntoUML simples



Fonte: Elaborado pelo autor (2023)

Há uma representação mais complexa assim como mostra a Figura 15 que também foi elaborada na aplicação por meio das novas funcionalidades criadas.

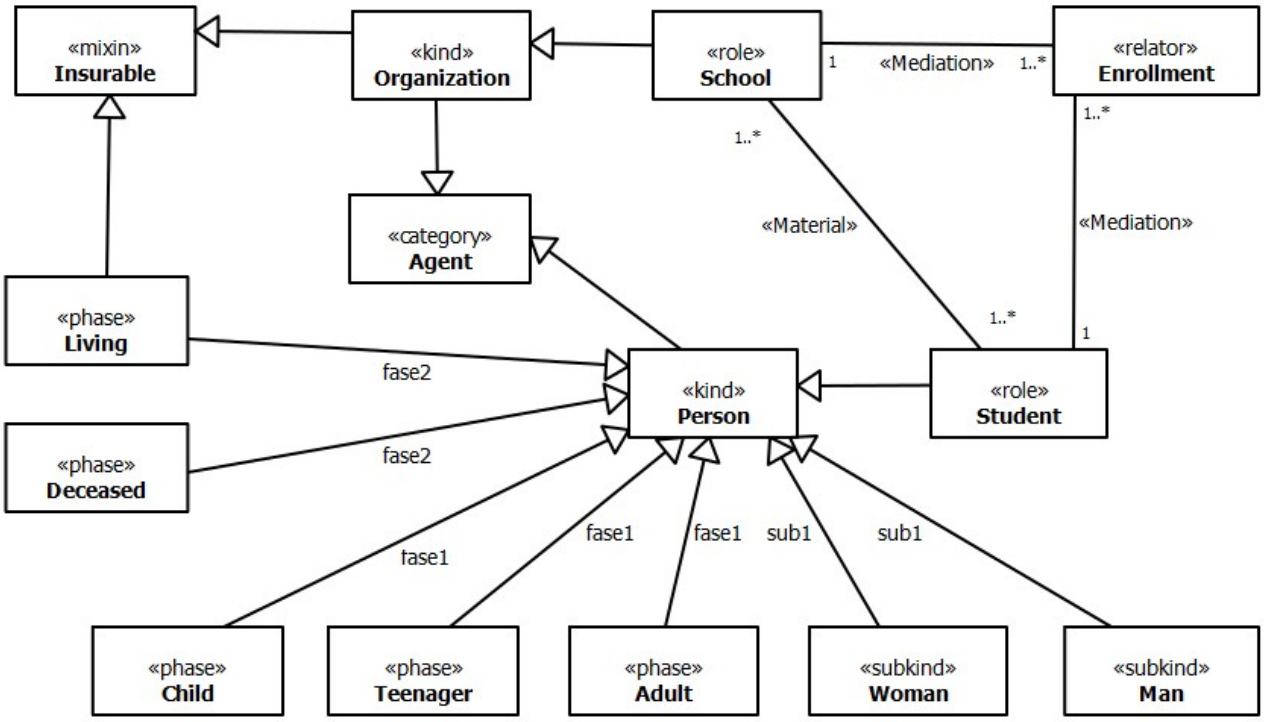
A Figura 15 mostra um diagrama mais complexo, apresentado em um artigo de Braga et al. (2010) em que é abordado um diagrama OntoUML e a conversão em Alloy, no qual essa semelhante amostra é utilizada para validar e comparar com o resultado gerado desta seção.

5.1.1 Código em Alloy gerado por meio de modelo OntoUML

Os diagramas de modelo OntoUML podem ser exportados para a linguagem de validação Alloy. O Quadro 8 representa o código em Alloy da Figura 14 gerado de forma automática na aplicação.

A Figura 15 possui sua representação de código em Alloy no Quadro 9, no qual contém o script gerado após selecionar a opção de exportação para a linguagem de validação.

Figura 15 – Diagrama OntoUML mais complexo



Fonte: Elaborado pelo autor (2023)

Quadro 8 – Código gerado em Alloy do diagrama da Figura 14

```
1  module Newmodel
2  open util/ordering[State] as state
3  open util/relation
4
5  sig Veiculo{}
6  sig Carro, Moto in Veiculo{}
7  fact generalization_set{
8          disj[Carro,Moto]
9          Veiculo = Carro+Moto
10 }
11 sig State{
12         exists: set (Veiculo),
13         Quebrado: set Veiculo:>exists,
14         Intacto: set Veiculo:>exists,
15 } {
16         all x:exists|x not in this.next.@exists implies x not in this.^next
               .@exists
17 }
18
19 pred show[] {}
20 run show
```

Fonte: Elaborado pelo autor (2023).

Os exemplos abordados anteriormente validam as funcionalidades já citadas na seção de desenvolvimento.

Quadro 9 – Código gerado em Alloy do diagrama da Figura 15

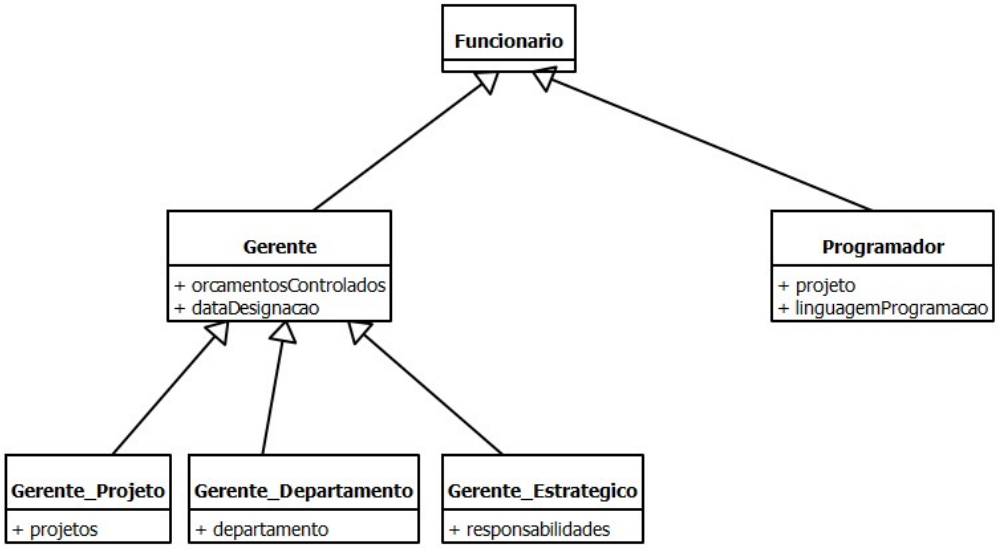
```
module Newmodel
open util/ordering[State] as state
open util/relation

sig Organization{}
sig Person{}
sig Enrollment{
        School: some Organization,
        Student: some Person,
        derived_material_relation0: Student some -> School,
}
sig Woman, Man in Person{}
fact generalization_set{
        disj[Woman,Man]
        Person = Woman+Man
}
fun Agent:(Person+Organization){
        Person+Organization
}
sig State{
        exists: set (Organization+Person+Enrollment),
        disj Adult, Teenager, Child: set Person:>exists,
        disj Deceased, Living: set Person:>exists,
        Student: set Person:>exists,
        School: set Organization:>exists,
        Insurable: set Organization:>exists+Living,
        study: set Student -> School,
} {
        all x:exists|x not in this.next.@exists implies x not in this.^next
            .@exists
        Person:>exists = Adult+Teenager+Child
        Person:>exists = Deceased+Living
        all x:Enrollment:>exists | x.School in Organization:>exists and x.
            Student in Person:>exists
        Student = (Enrollment:>exists).Student
        (Enrollment:>exists).School in School
        all x: School | some Enrollment:>exists:>School.x
        Insurable = Organization:>exists+Living
        study in exists.derived_material_relation0
}

pred show[] {}
run show
```

Fonte: Elaborado pelo autor (2023).

5.2 MODELO UML

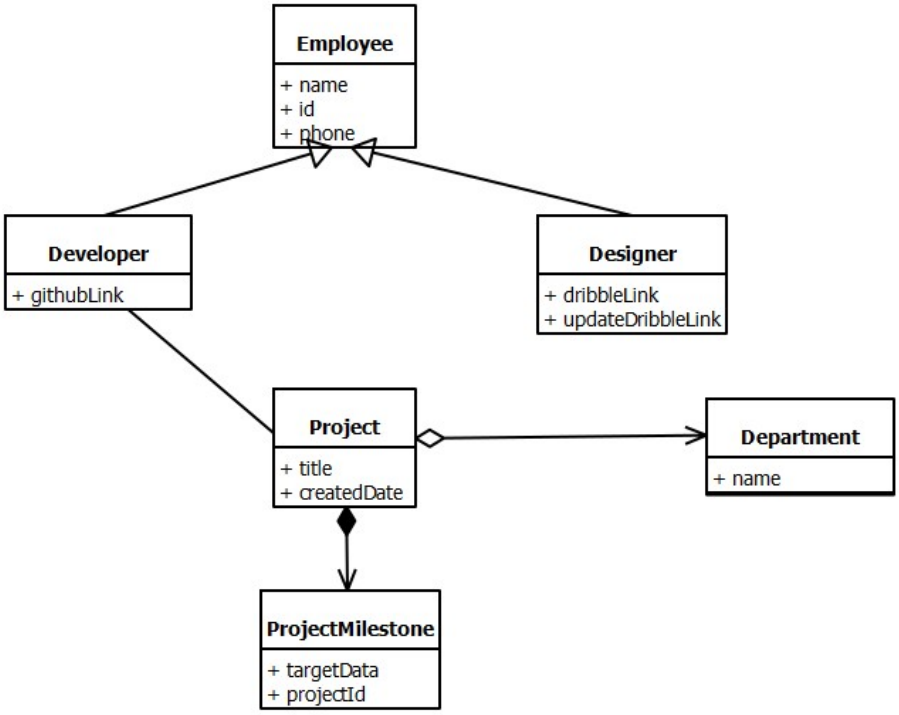
Serão apresentados dois modelos UML, mostrados na Figura 16 e Figura 17, no qual serão reproduzidos em seguida o script do código em Python e em Alloy

Figura 16 – Diagrama UML simples



Fonte: Elaborado pelo autor (2023)

Figura 17 – Diagrama UML mais complexo



Fonte: Elaborado pelo autor (2023)

5.2.1 Código em Python gerado por meio de modelo UML

Neste subtópico são abordados os scripts na linguagem Python gerado de forma automática na aplicação, conforme exibidos no Quadro 10 que está relacionado a Figura 16 e Quadro 11 que foi exportado por meio do modelo da Figura 17.

5.2.2 Código em Alloy gerado por meio dos modelos UML

Neste subtópico são abordados os scripts na linguagem Alloy gerado de forma automática na aplicação, conforme exibidos no Quadro 12 e Quadro 13.

Os códigos em Alloy foram executados em uma ferramenta externa, o Alloy Analyzer, que possibilitou verificar a sintaxe e compilação dos scripts, validando assim que os códigos gerados são funcionais.

Quadro 10 – Código gerado em Python do diagrama da Figura 16

```
class Funcionario:
        def __init__(self):
                pass
class Programador(Funcionario):
        def __init__(self, projeto, linguagemProgramacao):
                super().__init__(projeto, linguagemProgramacao)
                self.projeto = ''
                self.linguagemProgramacao = ''

        def setprojeto(self, projeto):
                self.projeto = ''

        def setlinguagemProgramacao(self, linguagemProgramacao):
                self.linguagemProgramacao = ''

class Gerente_Projeto(Gerente):
        def __init__(self, projetos, orcamentosControlados, dataDesignacao)
            :
                super().__init__(projetos, orcamentosControlados,
                    dataDesignacao)
                self.projetos = ''

        def setprojetos(self, projetos):
                self.projetos = ''

class Gerente_Departamento(Gerente):
        def __init__(self, departamento, orcamentosControlados,
            dataDesignacao):
                super().__init__(departamento, orcamentosControlados,
                    dataDesignacao)
                self.departamento = ''

        def setdepartamento(self, departamento):
                self.departamento = ''

class Gerente_Estrategico(Gerente):
        def __init__(self, responsabilidades, orcamentosControlados,
            dataDesignacao):
                super().__init__(responsabilidades, orcamentosControlados,
                    dataDesignacao)
                self.responsabilidades = ''

        def setresponsabilidades(self, responsabilidades):
                self.responsabilidades = ''

class Gerente(Funcionario):
        def __init__(self, orcamentosControlados, dataDesignacao):
                super().__init__(orcamentosControlados, dataDesignacao)
                self.orcamentosControlados = ''
                self.dataDesignacao = ''

        def setorcamentosControlados(self, orcamentosControlados):
                self.orcamentosControlados = ''

        def setdataDesignacao(self, dataDesignacao):
                self.dataDesignacao = ''
```

Fonte: Elaborado pelo autor (2023).

Quadro 11 – Código gerado em Python do diagrama da Figura 17

```
1  class Designer(Employee):
2          def __init__(self, dribbleLink, updateDribbleLink, name, id, phone)
               :
3                  super().__init__(dribbleLink, updateDribbleLink, name, id,
                       phone)
4                  self.dribbleLink = ''
5                  self.updateDribbleLink = ''
6
7          def setdribbleLink(self, dribbleLink):
8                  self.dribbleLink = ''
9
10         def setupdateDribbleLink(self, updateDribbleLink):
11                 self.updateDribbleLink = ''
12
13 class Project:
14         def __init__(self, title, createdDate):
15                 self.title = ''
16                 self.createdDate = ''
17
18         def settitle(self, title):
19                 self.title = ''
20
21         def setcreatedDate(self, createdDate):
22                 self.createdDate = ''
23
24 class Developer(Employee):
25         def __init__(self, githubLink, name, id, phone):
26                 super().__init__(githubLink, name, id, phone)
27                 self.githubLink = ''
28
29         def setgithubLink(self, githubLink):
30                 self.githubLink = ''
31
32 class Department:
33         def __init__(self, name):
34                 self.name = ''
35         def setname(self, name):
36                 self.name = ''
37
38 class ProjectMilestone:
39         def __init__(self, targetData, projectId):
40                 self.targetData = ''
41                 self.projectId = ''
42         def settargetData(self, targetData):
43                 self.targetData = ''
44         def setprojectId(self, projectId):
45                 self.projectId = ''
46
47 class Employee:
48         def __init__(self, name, id, phone):
49                 self.name = ''
50                 self.id = ''
51                 self.phone = ''
52         def setname(self, name):
53                 self.name = ''
54         def setid(self, id):
55                 self.id = ''
56         def setphone(self, phone):
57                 self.phone = ''
```

Fonte: Elaborado pelo autor (2023).

Quadro 12 – Código gerado em Alloy do diagrama da Figura 16

```
module Newmodel

sig Funcionario {}

sig Gerente extends Funcionario {}

sig Programador extends Funcionario {}

sig Gerente_projeto extends Gerente {}

sig Gerente_departamento extends Gerente {}

sig Gerente_estrategico extends Gerente {}

pred show[] {}
run show
```

Fonte: Elaborado pelo autor (2023).

Quadro 13 – Código gerado em Alloy do diagrama da Figura 17

```
module Newmodel

sig Employee {}

sig Developer extends Employee {
Project: one Project}

sig Designer extends Employee {}

sig Project {
Developer: one Developer,
Department: one Department,
ProjectMilestone: one Projectmilestone}

sig Projectmilestone {
Project: one Project}

sig Department {
Project: one Project}

pred show[] {}
run show
```

Fonte: Elaborado pelo autor (2023).

6 CONSIDERAÇÕES FINAIS

Neste trabalho foram implementados diferentes plugins na aplicação Gaphor. Essa tarefa foi realizada com o intuito de oferecer suporte a linguagens específicas de domínio que permitiu a modelagem de modelos.

Dessa forma possibilitou a criação de modelos em uma nova linguagem (OntoUML) que antes não era disponível, assim como a elaboração de conversão automática de diagramas para scripts de códigos (Alloy e Python).

Os plugins foram desenvolvidos por meio da linguagem Python utilizando como base o código-fonte da ferramenta, no qual é totalmente gratuito e de livre acesso.

Os objetivos deste trabalhado foram alcançados de modo que facilitou que não programadores possam desenvolver programas por meio da extração de códigos de modelos feitos.

A análise da ferramenta foi realizada pelo autor por meio de testes de validação, conforme descrito no capítulo 5, seguindo os critérios estabelecidos pelas fontes primárias das pesquisas, que são detalhadas no capítulo 2. Para efetuar a validação o autor também utilizou uma ferramenta externa para verificar se os códigos gerados em Alloy estavam corretos, a aplicação utilizada é conhecida como Alloy Analyzer e está disponível para download no site: https://alloytools.org/download.html.

Constatou-se que a aplicação atende aos aspectos desejados e os procedimentos estão em conformidade com os objetivos, com base em um estudo abrangente que analisou a ferramenta e documentos relacionados aos processos.

6.1 TRABALHOS FUTUROS

Apesar de que os objetivos propostos no trabalho tenham sido atingidos, são possíveis algumas melhorias visando trabalho futuro:

- Ajustar plugins criados para a versão atual do Gaphor, visto que ao instalar qualquer versão diferente, é necessário realizar configurações para que os plugins sejam compatíveis com a aplicação;

- Alterar a estética das relações dos modelos, visto que as representações dos relacionamentos não atualizam de acordo com a mudança de espaço local dos objetos;

- Possibilitar novas linguagens de códigos a serem extraídos, por exemplo, criar um plugin para que os modelos sejam extraídos para código-fonte na linguagem C ou Java;

- Ajustar plugin em Python para que gere também código-fonte de modelo em OntoUML;

- Corrigir a relação "Material"da OntoUML para que seja possível interligar os relacionamentos;

- Ajustar plugin de criação de diagramas OntoUML para que não perca informações visuais ao importar os modelos, visto que ao reabrir um arquivo há dados dos objetos que não são mostrados diretamente.

- Possibilitar que o plugin em Python reconheça todos os tipos de relações presentes no diagrama de classes da UML, visto que está descrevendo apenas relações de herança.

REFERÊNCIAS

BANASZESKI, F. Comparativo entre modelagem conceitual uml e modelagem ontouml baseada em ontologias. Universidade Tecnológica Federal do Paraná, p. 66, 2015.

BORELLI, H. Uma linguagem de modelagem de domínio específico para linhas de produto de software dinâmicas. Universidade Federal de Goiás, p. 88, 2016.

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, p. 10, 2010.

BRAGA, R. T. V.; MASIERO, P. C. Um método completo para desenvolvimento orientado a objetos com uml: da análise à implementação em java. Universidade de São Paulo, p. 181, 2007.

CAPELO, M. A. B. Especificação e validação de processos etl em alloy. Universidade do Minho, p. 152, 2018.

CORDEIRO, J. P. M. Síntese de programas utilizando a linguagem alloy. Universidade Federal de Pernambuco, p. 48, 2017.

DAVIES, P. B. et al. Rapid application development (rad): an empirical review. School of Computing, University of Glamorgan, Wales, p. 13, 1999.

EISHIMA, T. E. Proposta de metamodelo para desenvolvimento orientado a modelo para empresas do apl de londrina. Universidade Estadual de Londrina, 2014.

GUIMARÃES, D. d. S. Inconsistências em regras de negócio: um método para identificação automatizada usando alloy. Universidade Federal do Rio de Janeiro, p. 92, 2015.

GUIZZARDI, G. et al. Ontologias de fundamentação, modelagem conceitual e interoperabilidade semântica. *Núcleo de Estudos em Modelagem Conceitual e Ontologias (NEMO)*, Universidade Federal do Espírito Santo, p. 10, s.d.

HEINECK, T. Desenvolvimento orientado a modelos no domínio de robótica: uma revisão sistemática da literatura. Universidade Federal de Pernambuco, p. 131, 2016.

JUMES, F.; HONDA, L. C. R. Linguagem específica de domínio para programação de robôs. Universidade Federal de Santa Catarina, p. 57, 2007.

JUNIOR, W. M. P. Apostila engenharia de software. Universidade do Estado de Minas Gerais, p. 109, 2010.

LUCRÉDIO, D. Uma abordagem orientada a modelos para reutilização de software. Universidade de São Paulo, p. 277, 2009.

MOLENAAR, A.; YEAW, D. *Modelagem para Todos*. 2020. Disponível em: <https://gaphor.org/#about>. Acesso em: 30 março de 2023.

NETO, D. F. Comdd: uma abordagem colaborativa para auxiliar o desenvolvimento orientado a modelos. Universidade de São Paulo, p. 87, 2012.

NETO, D. F.; FROTA, P. C. da; FORTES, R. P. d. M. Uma abordagem distribuída para o desenvolvimento orientado a modelos. *Workshop de Desenvolvimento Distribuído de Software*, Universidade de São Paulo, 2011.

PAULO. *Introdução a modelagem conceitual*. 2008. Disponível em: <https://www.devmedia.com.br/introducao-a-modelagem-conceitual/10793>. Acesso em: 22 março de 2023.

PEREIRA, L. A. d. M. Análise e modelagem de sistemas com a uml: com dicas e exercícios resolvidos. SINDICATO NACIONAL DOS EDITORES DE LIVROS, RJ, 2011.

QUITERIO, I. d. C. Proposta de uma abordagem orientada a modelos para linha de produtos de software. Universidade Estadual de Londrina, p. 64, 2018.

RAMOS, D. C. A. Conceitos de orientação a objetos. Unicamp, 2019.

SANTOS, V. A. D. Uma abordagem baseada em padrões para construção de modelos conceituais em ontouml. Universidade Federal do Espírito Santo, p. 107, 2015.

SEVERIEN, A. L. Web application language engine (wale): Geração de código para aplicações j2ee baseado em técnicas de desenvolvimento orientado a modelos. Universidade Federal de Pernambuco, p. 52, 2008.

SILVA, S. d. M. Avaliação da modelagem conceitual de sistemas de informação a partir de ontologias de fundamentação: verificação de relações parte-todo. Universidade Federal de Minas Gerais, Escola de Ciência da Informação, p. 94, 2014.

SUCHÁNEK, M. *OntoUML Especificação*. 2018. Disponível em: <https://ontouml.readthedocs.io/en/latest/index.html>. Acesso em: 29 março de 2023.

TACLA, C. A. Análise e projeto oo uml 2.0. Universidade Tecnológica Federal do Paraná, p. 97, s.d. Disponível em: <https://pessoal.dainf.ct.utfpr.edu.br/tacla/UML/Apostila.pdf>. Acesso em: 22 março 2023.