# PARTHENOS: UMA ABORDAGEM DE INJEÇÃO DE CÓDIGO-FONTE PARA TRANSFORMAÇÃO DE SOFTWARE

Gabriel Lopes Nunes*

Prof. Dr. Kleinner Silva Farias de Oliveira**

**Resumo:** Realizar manutenção de *software* legado é uma necessidade em empresas. Porém, é ainda um processo caro e demorado, tanto para elas quanto para os clientes, que gastam tempo e dinheiro para conseguirem que pequenas mudanças sejam implementadas. O MITRAS é um modelo de sistema de manutenção automático baseado em transformações de grafos que objetiva reduzir esses custos, provendo a usuários o poder de modificar a interface gráfica e serviços de um sistema completo sem necessidade de interação ou auxílio de um desenvolvedor de software. No entanto, o MITRAS é limitado em seu suporte a sistemas que usem diversas linguagens e ferramentas. Além disso, ele carece de mecanismos que garantam as corretas transformações e injeções feitas para modificar o código-fonte. O presente trabalho apresenta o Parthenos, uma abordagem de implementação do modelo MITRAS que fornece uma maneira de transformar o grafo e injetar código-fonte garantindo a correção das modificações introduzidas em nível de sintaxe e de semântica de tipos. Junto disso, ele propõe uma arquitetura extensível, que possibilita os mais variados tipos de sistemas a realizar essas manutenções. O Parthenos foi avaliado através de testes funcionais, obtendo-se métricas associadas a quatro cenários de transformação e injeção de código-fonte apoiados por extensões desenvolvidas para o propósito da avaliação. O resultado obtido pelo Parthenos foi promissor: além de ficar demonstrada a extensibilidade da solução, foi mostrado que as transformações e injeções foram aplicadas corretamente, mantendo a correção da sintaxe e da semântica de tipos através das várias aplicações.

**Palavras-chave:** Injeção de Código-fonte. Sistemas de Manutenção Automáticos. Transformações Algébricas de Grafos. Engenharia de Software Automática.

## 1 INTRODUÇÃO

O processo de manutenção e operação de sistemas tem custo elevado tanto para desenvolvedores, quanto para as empresas que os mantém. (DE ESPINDOLA; MAJDENBAUM; AUDY, 2004; DE OLIVEIRA, 2016). Mesmo representando a maior parcela do custo de um projeto de *software*, a manutenção recebe muita atenção por parte das empresas, já que, muitas vezes, elas possuem *softwares* legados lucrativos que precisam continuar funcionando corretamente com o passar do tempo. (DE

* Estudante de Ciência da Computação na Universidade do Vale do Rio dos Sinos – UNISINOS, São Leopoldo. E-mail: glnunes@edu.unisinos.br.

** Professor assistente na UNISINOS, Doutor em Informática pela PUC-Rio (2012), Mestre em Ciência da Computação pela PUC-Rio (2008). E-mail: kleinnerfarias@unisinos.br.

OLIVEIRA, 2016; PADUELLI, 2007). Muitas vezes, existem demandas por pequenas alterações que são requisitadas às empresas simultaneamente com outros projetos. (KÜPSSINSKI, 2019). Assim, os clientes podem não ter suas necessidades atendidas em tempo, ou precisam despender muito dinheiro para que as mudanças desejadas sejam implementadas. (DE ESPINDOLA; MAJDENBAUM; AUDY, 2004; DE OLIVEIRA, 2016). Para mitigar esses tipos de problemas, Küpssinski (2019) desenvolveu o MITRAS.

O MITRAS foi um projeto realizado no Programa de Pós-Graduação em Computação Aplicada (PPGCA) da UNISINOS que introduziu um modelo de *software* que usa de formalismos de manipulação e transformação de grafos para conferir a sistemas a capacidade de habilitar usuários finais, que não possuem conhecimento de programação, a realizar manutenções adaptativas e perfectivas neles. (KÜPSSINSKI, 2019). O modelo MITRAS possui três etapas principais para executar sua função: extração do modelo, transformação e injeção do código-fonte. Possui, também, um módulo de comunicação com o usuário final, que usa processamento de linguagem natural (PLN) para compreender a manutenção que o usuário deseja realizar e aplicar as transformações e injeções necessárias. (KÜPSSINSKI, 2019).

Atualmente, o MITRAS possui uma solução para o processo de injeção que é bastante limitada, sendo ela bastante específica para as linguagens Java, JSP e *framework* Hibernate. Além disso, ele desconsidera a tipagem de dados. Assim, todos os tipos são tomados como *String* tanto nas transformações, quanto nas injeções. Outro ponto é a falta de metadados no modelo, que dificulta o rastreamento do código-fonte através do grafo.

Assim, esse artigo propõe o Parthenos, uma abordagem de implementação das etapas principais do modelo MITRAS que busca trazer maior atenção à etapa de injeção no código-fonte, criando mecanismos que permitam garantir a correção sintática e correta semântica de tipos do sistema ao longo das aplicações de transformações, mantendo a sincronia entre os modelos. Ademais, o Parthenos também deseja prover certo nível de extensibilidade em sua arquitetura, permitindo que sistemas usando diversas linguagens, *frameworks*, bibliotecas e ferramentas, possam tirar proveito desse tipo de manutenção.

O restante do artigo está estruturado da seguinte maneira: a seção 2 apresenta os fundamentos teóricos que se julgam necessários para o entendimento das seções subsequentes. A seção 3 apresenta e compara os trabalhos relacionados. A seção 4

descreve o modelo e a arquitetura propostos pelo Parthenos. A seção 5 define os métodos de avaliação, mostra os cenários e apresenta os resultados obtidos. Por fim, a seção 6 retoma aspectos do trabalho e apresenta sugestões de trabalhos futuros.

## 2 FUNDAMENTAÇÃO TEÓRICA

Esta seção aborda os conceitos teóricos utilizados durante a construção e desenvolvimento do Parthenos. Ela contempla as estruturas empregadas e os formalismos de manipulação dessas, bem como o modelo MITRAS, usado como base para o trabalho.

### 2.1 Grafos

O conceito de grafos trata de uma representação matemática para um modelo que é bastante visual. Tal modelo dispõe de pontos e linhas que ligam alguns desses pontos dois a dois. (BONDY; MURTY, 1976). De fato, essa representação é tão genérica, que ela pode ser usada para modelar diversos tipos de situações do mundo real onde se queira exibir relações entre elementos. Assim, fica evidente que os grafos são ótimas ferramentas para modelar relações binárias. (KNAUER, 2011).

Um grafo pode ser definido como uma tripla $G = (V, E, p)$, tal que $V \neq \emptyset$, onde $V$ é um conjunto finito de vértices, $E$ é um conjunto de arestas e $p : E \rightarrow V^2$ é um mapeamento que relaciona, para alguma aresta $e \in E$, um par de vértices $(u, v) \in V^2$ e representa a ligação de $u$ e $v$ através de uma linha. (KNAUER, 2011). É possível denotar $G$ como $(V_G, E_G, p_G)$ ou $(V(G), E(G), p_G)$, onde $p_G(e) = uv$, para identificar a qual grafo cada elemento da tripla pertence. (BONDY; MURTY, 1976).

Para incorporar outros tipos de informações semânticas, é possível modificar a estrutura de um grafo. Pode-se impor, por exemplo, que as arestas de um grafo sejam setas, a fim de representar relações unidirecionais, chama-se esses grafos de dígrafos ou grafos dirigidos. Ou ainda, pode-se associar rótulos ou pesos aos vértices e arestas de um grafo, que são os chamados grafos rotulados. (GERSTING, 2006). O modelo MITRAS utiliza um modelo de grafo dirigido e rotulado para que seja possível identificar hierarquias de classes ou diagramas de banco de dados. (KÜPSSINSKI, 2019).

## 2.2 Gramáticas de grafo

As gramáticas de grafo são uma generalização da teoria das linguagens formais baseadas em *strings* e da teoria de sistemas de reescrita baseados em árvores. Da maneira como tais gramáticas são definidas, se torna fácil representar transformações locais em grafos de forma matematicamente precisa, o que permite inferir-se propriedades e provar-se determinados aspectos delas mais facilmente. (ROZENBERG, 1997).

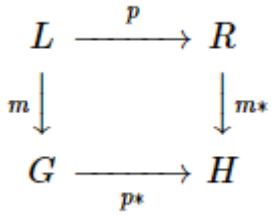
As gramáticas são constituídas, genericamente, de produções, que são regras a se seguir para realizar a transformação de um grafo. Essas produções são vistas como triplas $(M, D, E)$, em que $M$ é o grafo mãe, $D$ o grafo filha e $E$ é algum mecanismo de reinserção de tal forma que a produção pode ser aplicada para algum grafo $G$, sempre que uma ocorrência de $M$ é identificada ou casada em $G$. Da aplicação da produção, remove-se $M$ de $G$, obtendo-se um grafo $G^-$, no qual $D$ é reinserido utilizando o mecanismo $E$.

Existem diversos tipos de mecanismos de reinserção que podem ser considerados na aplicação de uma produção, dentre os quais dois são os mais importantes: o adesivo e o conectivo. No primeiro, partes do grafo $D$ são mapeadas para o $G^-$, enquanto no segundo, novas arestas são criadas para conectar $D$ a $G^-$. Baseado nesses mecanismos, definem-se, entre outras, duas importantes abordagens para reescrita de grafos: a algébrica, baseada na teoria das categorias, e a algorítmica, baseada na teoria dos conjuntos. (ROZENBERG, 1997).

O Parthenos faz uso da abordagem algébrica, usando, mais especificamente, a técnica de coproduto fibrado único (ou *single pushout* em inglês) para aplicar as transformações. Nessa técnica, uma transformação (casamento, deleção e inserção de vértices e arestas) pode ser feita em um único passo como um construto de dois morfismos $p$ e $m$, em que $p$ é um morfismo parcial e $m$ um morfismo total de grafos. O morfismo parcial $p : L \rightarrow R$ pode ser visto como uma produção e é definido como parcial, pois permite que vértices ou arestas de $L$ não tenham imagem em $R$. Assim, esses elementos serão excluídos pela transformação. De forma similar, elementos de $R$ que não possuem pré-imagem em $L$ serão criados. Já o morfismo total $m : L \rightarrow G$ é o casamento (*match*) entre $L$ e $G$. Exige-se que ele seja total, pois todos os vértices e arestas de $L$ devem estar presentes em $G$ através de $m$. (ROZENBERG, 1997). O

objeto do coproduto fibrado de $\langle p, m \rangle$ é um grafo $H$ como mostrado no diagrama comutativo da Figura 1, que é o grafo resultante da transformação.

Figura 1 – Diagrama comutativo do coproduto fibrado único




Fonte: Rozenberg (1997).

## 2.3 MITRAS

O MITRAS foi um projeto desenvolvido no PPGCA da UNISINOS. Ele é um modelo de manutenção automática de *software* baseado em processamento de linguagem natural e transformações de grafos, que tem como objetivo prover ao usuário final de uma determinada aplicação o poder de realizar manutenções sem conhecimento prévio de programação. (KÜPSSINSKI, 2019). Como visto na Figura 2, ele possui 3 etapas e um módulo de interação com o usuário final.

Na primeira etapa, o código-fonte passa por um processo de extração. Nessa etapa, o MITRAS varre o repositório da aplicação a fim de encontrar arquivos importantes para a representação dela. Arquivos contendo classes, configurações, informações de persistência, interface gráfica etc., além de dependências, associações, entre outros, são todos mapeados em um modelo abstrato de grafo, que retrata a aplicação num nível suficientemente alto de forma que ainda é possível rastrear o código-fonte a partir dele, mas evitando expor complexidades que são desnecessárias para a execução do processo como um todo. (KÜPSSINSKI, 2019).

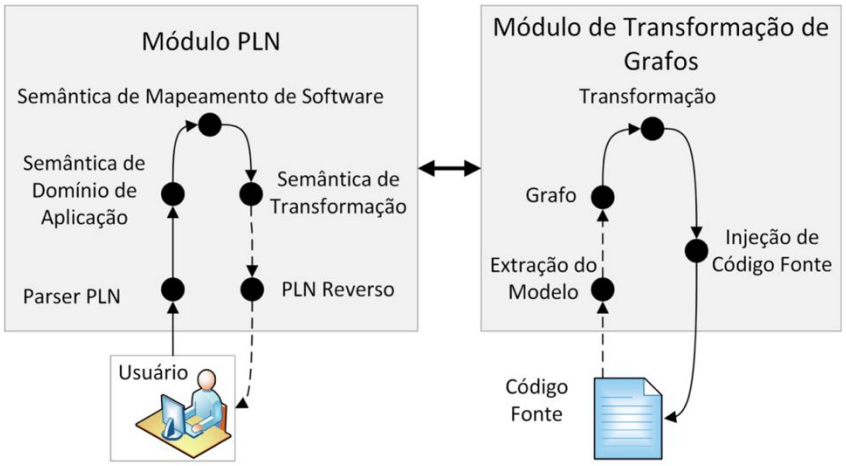
Na segunda etapa, as transformações podem ser aplicadas ao grafo abstrato. No trabalho de Küpssinski (2019) são propostos quatro tipos de transformação: adição de um campo, oclusão de um campo, desfazer a oclusão e reposicionamento de um painel. Seguindo a abordagem algébrica para transformações de grafos, antes de aplicar uma transformação, um subgrafo $D$ deve ser identificado no grafo original através de um morfismo. A transformação é aplicada sobre o subgrafo identificado seguindo a regra definida em uma produção $p$ que leva $D$ para algum grafo $D'$ que

pode ter arestas e vértices inseridos ou removidos. O grafo $D'$ é, então, reconectado ao grafo original seguindo uma política de reinserção.

Na terceira etapa, todas as transformações realizadas no modelo abstrato são traduzidas para código e inseridas nos arquivos-fonte da aplicação. A esse passo dá-se o nome de *injeção*, já que o MITRAS se encarrega de inserir *snippets* de código ou alterar o código da própria aplicação em pontos específicos dos arquivos-fonte. Dessa forma, eles não são todos reescritos e, assim, mantém-se a sincronia entre o modelo abstrato e o código-fonte da aplicação com um bom desempenho.

O MITRAS disponibiliza um módulo de processamento de linguagem natural, que permite que o usuário comunique ao sistema como deseja modificá-lo utilizando linguagem natural. Uma vez que o MITRAS conhece a linguagem de domínio da aplicação, ele consegue identificar qual transformação aplicar a partir do *input* do usuário.

Figura 2 – Visão de alto nível do modelo MITRAS




Fonte: Küpssinski (2019).

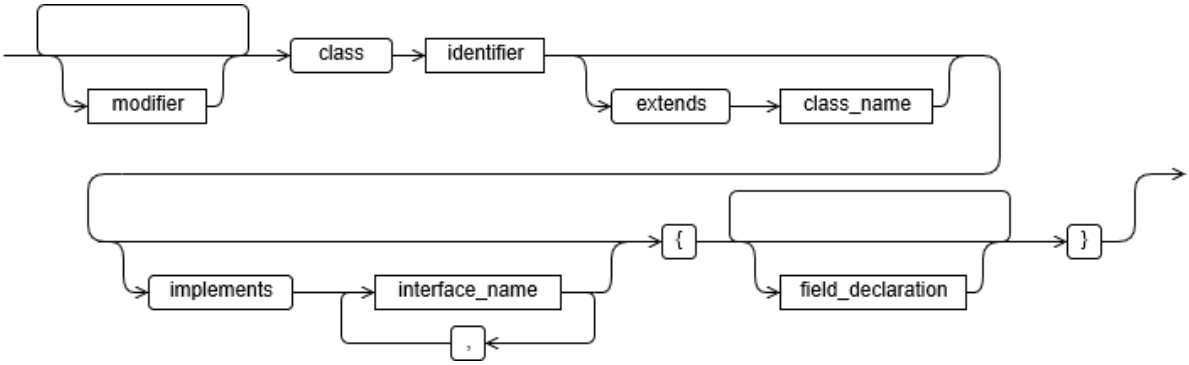
## 2.4 Gramáticas livre de contexto e Análise sintática

Uma gramática livre de contexto é um formalismo que permite definir construções que são naturalmente recursivas. (HOPCROFT; MOTWANI; ULLMAN, 2001). As linguagens geradas por essas gramáticas ganharam bastante atenção, pois conseguem lidar bem com construções típicas das linguagens de programação, como o balanceamento de parênteses, blocos estruturados, etc. (MENEZES, 2000). Uma gramática livre de contexto é formalmente descrita como uma quádrupla $G =$

$(V, T, P, S)$, onde $V$ é um conjunto de variáveis (ou símbolos não-terminais), $T$ é um conjunto de símbolos terminais, $P$ é um conjunto de produções e $S \in V$ é o símbolo inicial. O que distingue $G$ como uma gramática livre de contexto é que suas produções são da forma $A \to \alpha$, em que $A \in V$ e $\alpha \in (V \cup T)^*$, ou seja, o lado esquerdo da produção contém sempre uma única variável e o lado direito é uma palavra que pode conter variáveis e símbolos terminais. (HOPCROFT; MOTWANI; ULLMAN, 2001).

Uma forma bastante intuitiva de se definir as gramáticas de linguagens livres de contexto e, portanto, da grande maioria das linguagens de programação, é o denominado diagrama ferroviário (ou *railroad diagram* em inglês) que define, graficamente, as construções de uma dada linguagem. (JENSEN; WIRTH, 1991). Esses diagramas são bastante úteis, sendo utilizados nos processos de extração e injeção de código-fonte do Parthenos. Como exemplo, a Figura 3 apresenta o diagrama ferroviário para a definição de classes em Java.

Figura 3 – Diagrama ferroviário da construção de classes em Java




Fonte: elaborado pelo autor.

A análise de um código-fonte gira em torno de sua sintaxe. Após um processamento do código-fonte, lexemas contendo um rótulo e um identificador são gerados. A análise sintática, também chamada de *parsing*, é responsável por verificar se os rótulos desses lexemas podem ser gerados pela gramática da respectiva linguagem e reportar erros sintáticos. Quando um programa é bem formado, isto é, não contém erros, o analisador léxico constrói uma árvore abstrata de sintaxe (AST) que representa as derivações, preservando a ordem em que as produções foram aplicadas. (AHO *et al.*, 2007).

# 3 TRABALHOS RELACIONADOS

A pesquisa por trabalhos relacionados se deu através dos repositórios integrados da UNISINOS, bem como a partir do *Google Scholar*. Alguns dos trabalhos apresentados foram encontrados pesquisando-se pelas *strings* de busca "*source code transformation"*, "*automatic code transformation*" e "*automatic code refactoring*", ao passo que outros foram obtidos por conveniência.

## 3.1 Análise dos trabalhos relacionados

***MITRAS: Modelo Inteligente para Transformação de Aplicações de Software.*** **(KÜPSSINSKI, 2019)**. O trabalho apresenta e discute o modelo do MITRAS com bastante profundidade. Mostra a viabilidade do uso do modelo, bem como a desenvoltura dele em cenários realísticos. Apesar de abranger todas as etapas em que o modelo se fundamenta, desde a extração de um repositório até a injeção do código-fonte, o foco do trabalho está em demonstrar a usabilidade dele. A dissertação selecionada é a base para a pesquisa envolvida neste trabalho. Fica claro que o modelo MITRAS é viável e pode ajudar a realizar manutenções adaptativas e perfectivas. No entanto, o trabalho pouco discorre sobre a etapa de injeção. As ferramentas necessárias e os desafios de se injetar as alterações introduzidas a partir das transformações no código-fonte não são abordados. Além disso, o protótipo implementa essa etapa de forma bastante limitada.

***Hermes: A Natural Language Interface Model for Software Transformation.*** **(CHAGAS *et al.*, 2019)**. O artigo apresenta e analisa uma solução que trata do módulo externo ao MITRAS responsável pela comunicação com o usuário final através de PLN. Tal solução utiliza o *CoreNLP* para marcação de *part-of-speech* das palavras das frases requisitadas e usa uma ontologia modelada através da linguagem OWL que representa os conceitos específicos do cenário analisado como entidades, painéis da interface gráfica, etc. para atrelar as palavras e suas inter-relações às intenções do usuário. Após isso, ele invoca o módulo de transformação do MITRAS provendo os parâmetros necessários. Os resultados do trabalho mostram que o Hermes consegue ter desempenho satisfatório em entender as transformações desejadas pelo usuário. Apesar disso, o *software* implementado não modela inter-relações de tipagens como, por exemplo, entender que um campo "idade" possui tipo numérico.

Tais tipagens se mostram essenciais para a etapa de injeção ser semanticamente correta.

***Transforming C++11 Code to C++03 to Support Legacy Compilation Environments.*** **(ANTAL *et al.*, 2016)**. Esse trabalho apresenta um *framework* para transformação de código C++11 para código C++03. A ideia do projeto é prover uma maneira de possibilitar aos desenvolvedores utilizar uma versão mais recente da linguagem a fim de aproveitar as facilidades de que ela dispõe sem que isso afete a interoperabilidade com sistemas legados. A solução, que tem código-aberto, possui uma lista de diversas transformações que podem ser aplicadas a determinados tipos de construções em C++11. O sistema pré-processa os arquivos-fonte a fim de encontrar as transformações necessárias, transforma os arquivos e grava informações de rastreamento entre versões do código-fonte transformado. Ele opera em um nível baixo, fazendo processamento de texto puro para detectar e aplicar transformações.

***Automated refactoring of super-class method invocations to the Template Method design pattern.*** **(ZAFEIRIS *et al.*, 2017)**. Nesse trabalho, se desenvolve uma refatoração automática do *design pattern Call Super* para *Template Method* em classes Java. O primeiro padrão é muito comum na orientação a objetos. Quando se deseja estender o comportamento de um método concreto da classe pai, faz-se uma chamada para ele a partir de um método de mesmo nome contido na classe filha, usando a palavra-chave *super*. O segundo padrão, pertence ao conjunto de padrões de projeto comportamentais. Ele introduz pontos de extensão mais bem definidos, em que se pode obter maior controle sobre o comportamento do método que se deseja estender. Esse trabalho detecta algoritmicamente o uso do padrão *Call Super* e transforma as classes pai e filha a fim de realizar uma refatoração desse padrão para o *Template Method*. A detecção e a transformação acontecem em nível maior do que o trabalho anterior, sendo aplicadas na árvore abstrata de sintaxe das classes.

***BRCode: An interpretive model-driven engineering approach for enterprise applications.*** **(OLIVEIRA *et al.*, 2018)**. Esse artigo propõe o BRCode, uma solução que usa uma abordagem interpretativa em MDE (*model-driven engineering*) para auxiliar no desenvolvimento de aplicações. O BRCode interpreta os modelos e metadados criados pelo desenvolvedor para gerar uma aplicação que já possui diversas funções incorporadas como, por exemplo, um sistema de autorização e autenticação e um sistema de internacionalização. O artigo foi selecionado por se assemelhar ao modelo MITRAS no sentido de interpretar e lidar com modelos para

auxiliar no desenvolvimento automático de aplicações. No entanto, o BRCode trabalha com um nível de abstração médio, já que lida com metadados e modelos que estão bem próximos da aplicação que eles representam.

***The Spartanizer: Massive Automatic Refactoring.*** **(GIL; ORRÙ, 2017)**. Esse trabalho apresenta o Spartanizer, um plugin para o Eclipse que aplica refatorações automáticas no código-fonte, a fim de o adequar à programação espartana. Tal estilo de programação valoriza o minimalismo e a escrita enxuta nos códigos. Assim, dá preferência a nomes de variáveis curtos a longos, operadores ternários a blocos condicionais etc. O Spartanizer realiza o *parsing* do código-fonte e identifica sugestões de refatoração. As sugestões são feitas através do Eclipse e o desenvolvedor pode escolher quais aceitará. O Spartanizer é capaz de refatorar o código-fonte sucessivamente através de transformações. Essas transformações são previamente identificadas e mantém a correção semântica do código na maioria dos casos. Além disso, apesar de ele não possuir um modelo abstrato, onde as transformações são aplicadas, elas são feitas na AST, ajudando a manter a correção sintática do código.

## 3.2 Análise comparativa dos trabalhos

Para a análise comparativa dos trabalhos, elencou-se uma série de aspectos que ajudam a melhor entender a proposta do presente trabalho. Uma descrição para cada aspecto é dada a seguir e a comparação entre os trabalhos pode ser vista no Quadro 1.

1. **Nível de abstração**: informa o nível de abstração que a solução apresenta para aplicar as transformações, sendo o código-fonte o nível mais baixo.
2. **Modelo abstrato**: indica se a solução utiliza algum tipo de modelo abstrato onde as transformações são aplicadas.
3. **Transformações**: enumera os tipos de transformação que a solução apresenta. Cada trabalho pode apresentar tipos dentre *text-to-text* (T2T), *text-to-model* (T2M), *model-to-model* (M2M), *model-to-text* (M2T), de acordo com as classificações das ferramentas de transformação propostas por Kahani *et al.* (2018).
4. **Tratamento de tipos**: indica se a solução se preocupa em tratar tipos em linguagens fortemente tipadas.

5. **Correção semântica**: indica se a solução tenta manter a correção semântica entre transformações.
6. **Tipo de avaliação**: descreve como a solução foi validada.

Quadro 1 – Comparativo dos trabalhos relacionados

| | *Nível de abstração* | *Modelo abstrato* | *Transformações* | *Tratamento de tipos* | *Correção semântica* | *Tipo de avaliação* |
|---|---|---|---|---|---|---|
| (KÜPSSINSKI, 2019) | Alto | Sim | T2M, M2M, M2T | Não | Não | Caso de uso e teste de performance |
| (CHAGAS *et al.*, 2019) | Alto | Sim | M2M | Não | Não | Questionário e teste de performance |
| (ANTAL *et al.*, 2016) | Baixo | Não | T2T | Não | Não | Testes funcional e de performance |
| (ZAFEIRIS *et al.*, 2017) | Médio | Não | T2T | Não | Sim | Teste funcional |
| (OLIVEIRA *et al.*, 2018) | Médio | Sim | M2T, T2T | Sim | Sim | Estudo de caso |
| (GIL; ORRÙ, 2017) | Médio | Não | T2T | Não | Sim | Estudo de caso |
| Parthenos | Alto | Sim | M2T | Sim | Sim | Teste funcional |

Fonte: elaborado pelo autor.

### 3.3 Oportunidade de pesquisa

Observando os trabalhos selecionados, é possível verificar que apenas dois trabalham com o modelo do MITRAS. O primeiro introduz o modelo do MITRAS e apresenta uma solução que tenta cobrir todas as etapas que ele descreve. Já o segundo apresenta uma solução para um módulo externo ao modelo do MITRAS, o módulo de processamento de linguagem natural. Ao passo que ambos estão focados em aplicar as transformações ao modelo abstrato de grafo, nenhum apresenta uma solução que se preocupe com a etapa da injeção do código-fonte. O trabalho de Küpssinski (2019) lida com as injeções de forma simples, assumindo, por exemplo, que os novos campos introduzidos pelas transformações são todos do tipo *String*.

Outro problema é a falta de um mecanismo que persista as alterações introduzidas no modelo após a aplicação de uma transformação. Além disso, a injeção é feita diretamente no código-fonte da aplicação, podendo introduzir erros sintáticos nela. O trabalho de Chagas *et al.* (2019) sofre com o mesmo problema de tipos. Assim, a solução não é capaz de inferir, através de sua ontologia, as tipagens envolvidas nas transformações a serem aplicadas, permitindo que erros semânticos possam ser introduzidos na aplicação. Em ambos trabalhos, não existem mecanismos que permitam o uso de sistemas não escritos em linguagem Java, ou que usem frameworks, bibliotecas ou ferramentas diferentes.

Assim, o Parthenos propõe uma solução para a etapa de injeção do código-fonte que cubra essas lacunas, visando melhorar o uso do modelo MITRAS:

1) Provendo um mecanismo que persista as alterações feitas nos modelos após uma transformação e garantindo sua sincronia;
2) Realizando a injeção e garantindo a semântica de tipos e correção sintática do código-fonte;
3) Provendo mecanismos que permita diversos tipos de sistemas a usarem o modelo, mediante o desenvolvimento de extensões que os suportem.

# 4 MODELO PROPOSTO

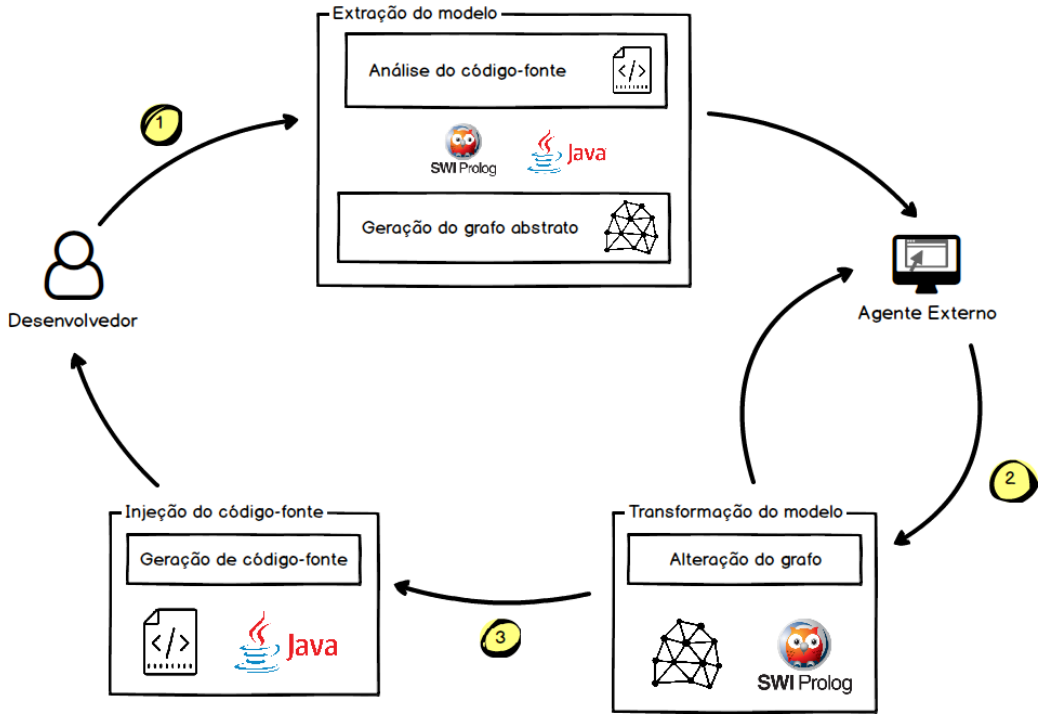
Esta seção apresenta o modelo proposto pelo Parthenos, constituído de 3 etapas, bem como a arquitetura proposta para realizar a implementação dele. Além disso, apresenta-se, também, aspectos da implementação do modelo, onde são introduzidas extensões desenvolvidas para auxiliar na posterior avaliação do trabalho.

## 4.1 Visão geral

O Parthenos objetiva solucionar os problemas existentes na etapa de injeção de código-fonte do modelo MITRAS. Mas, para poder realizar a injeção no código-fonte, mantendo-se a semântica de tipos e a sintaxe corretas, é necessário realizar diversas melhorias na forma como a solução de Küpssinski (2019) implementa as etapas de extração, transformação e injeção. Isso porque, nelas, não há preocupação com tipos: o modelo abstrato de grafo não gera tipagens de campos ou métodos, o transformador não recebe o tipo dos novos campos ou métodos para adicionar ou

alterar e, como consequência, o injetor assume todos esses como sendo de tipo *String*. Além disso, é impossível que sistemas escritos em linguagens, que não Java, sejam mantidos pelo MITRAS. Dessa forma, optou-se por reimplementar as etapas principais que ele apresenta: extração do modelo, transformação e injeção do código-fonte. (KÜPSSINSKI, 2019). Sendo assim, o Parthenos faz uso do modelo descrito pelo MITRAS, mas provendo uma nova implementação. Na Figura 4 é possível observar, em alto nível, as etapas do modelo proposto pelo Parthenos. Uma descrição de cada uma é dada a seguir.

Figura 4 – Modelo proposto pelo Parthenos




Fonte: elaborado pelo autor.

**Etapa 1: Extração do modelo**. Nessa etapa, um desenvolvedor, ou usuário com conhecimento de desenvolvimento, realiza a extração do modelo abstrato de grafo a partir do repositório de um sistema. O repositório desse sistema é varrido pelo Parthenos e todos os arquivos-fonte importantes para a representação dele, como, por exemplo, arquivos de classes, interface gráfica e configurações, são coletados, analisados e um grafo contendo as informações necessárias para representação desse sistema é gerado. O grafo fica disponível para que outras aplicações possam utilizá-lo para aplicar transformações ou obter informações e metadados do sistema representado por ele.

**Etapa 2: Transformação do modelo**. Essa etapa é acionada, quando algum agente externo necessita transformar o grafo produzido na etapa anterior. O agente comunica ao Parthenos a transformação que deseja realizar e aponta para o grafo, onde a transformação será aplicada. O Parthenos altera o grafo e o disponibiliza para futuras transformações. Além disso, a etapa de injeção do código-fonte é iniciada pelo Parthenos para que o código-fonte seja sincronizado com o modelo abstrato de grafo. Essa etapa será executada diversas vezes, sempre que for necessário dar manutenção ao sistema.

**Etapa 3: Injeção do código-fonte**. Nessa etapa as alterações que o grafo sofreu serão sincronizadas com o repositório de arquivos-fonte da aplicação. Ela garante que ambos modelos estejam corretos e em sincronia, para que seja possível evoluir a aplicação através de sucessivas transformações. Essa etapa é invocada pelo Parthenos de forma transparente na Etapa 2. Logo, sempre que a Etapa 2 for executada, a Etapa 3 será também e, por isso, tanto o grafo quanto o código-fonte serão alterados. Aqui, o Parthenos informa qual a transformação precisa ser injetada e aponta para o repositório onde se encontram os arquivos-fonte. O respectivo código é gerado e salvo no repositório.

O Parthenos foca na apresentação de uma maneira de injetar no código-fonte as alterações que são introduzidas quando alguma transformação é aplicada. Para que isso seja possível, ele executa uma série de passos, ao longo de todas as etapas, que são necessários para possibilitar a injeção a ser feita. Os passos são como segue:

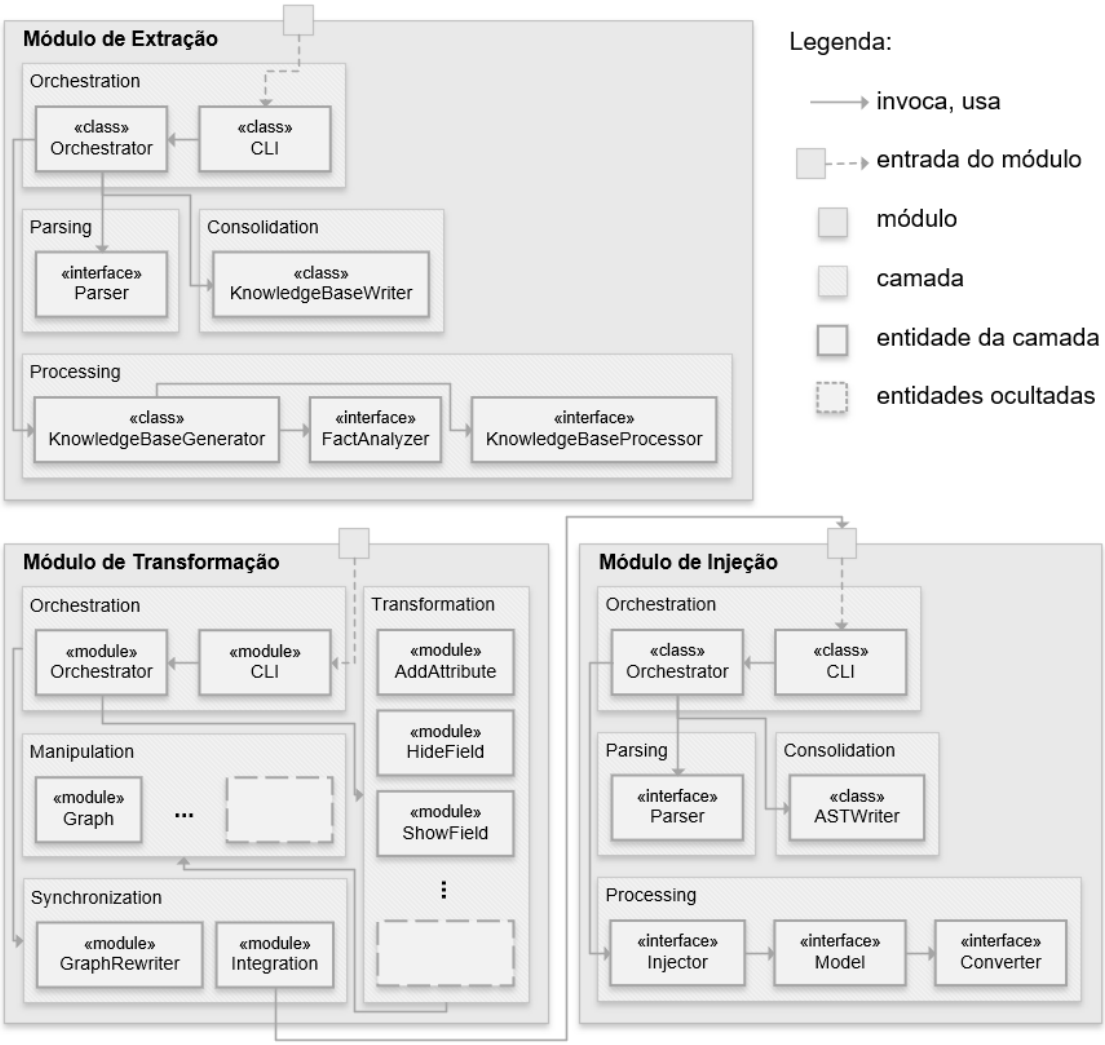
1. Encontrar as modificações introduzidas por uma transformação;
2. Identificar os arquivos que precisam ser injetados;
3. Identificar o tipo de injeção a ser feito;
4. Obter, a partir do grafo, os parâmetros necessários para realizar a injeção;
5. Encontrar o ponto de injeção dentro de cada arquivo, considerando a gramática dele;
6. Realizar a injeção;
7. Persistir o modelo abstrato de grafo com a transformação aplicada.

## 4.2 Arquitetura proposta

A arquitetura do Parthenos engloba as três etapas elencadas na Seção 4.1. Como visto na Figura 5, ela possui três módulos distintos, que se encarregam de

realizar, cada um, uma etapa específica. Os módulos são constituídos de camadas, que visam facilitar não somente a implementação, já que separam bem os diferentes fluxos de cada etapa, mas também a extensão dos módulos para suportar outras linguagens e ferramentas.

Figura 5 – Arquitetura proposta para o Parthenos

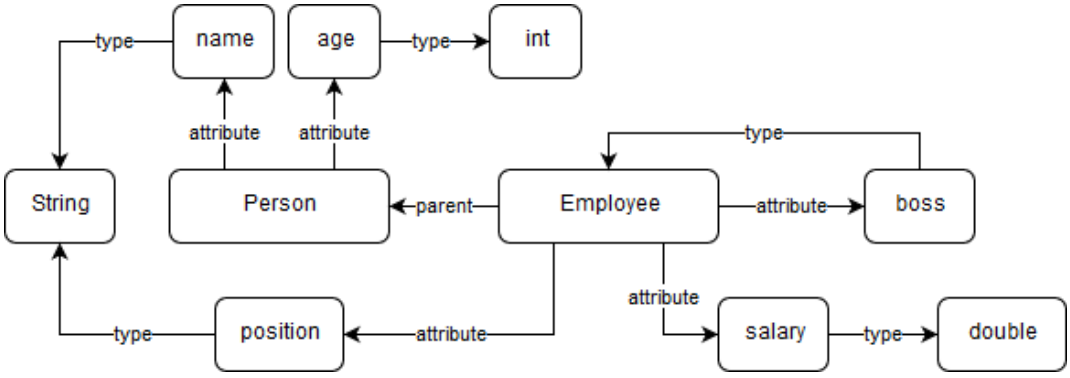



Fonte: elaborado pelo autor.

**Módulo de extração**. Esse módulo é responsável por gerar o modelo abstrato de grafo. Ele é escrito em linguagem Java e é invocado através da linha de comando. A entrada do módulo de extração é um repositório de arquivos-fonte, as linguagens que devem ser consideradas para geração do grafo e um caminho de arquivo destino onde o grafo será persistido. O módulo de extração possui quatro camadas: *orchestration*, *parsing*, *processing* e *consolidation*. A primeira camada é responsável

por comandar a aplicação, interpretando a linha de comando e invocando as outras camadas. A segunda camada lida com o *parsing* do arquivo-fonte para uma árvore abstrata de sintaxe (AST). Ela faz isso através da interface *Parser*. A terceira camada trabalha com o processamento da AST, a fim de gerar o modelo abstrato de grafo. Para isso, ela usa uma classe *KnowledgeBaseGenerator*, que gera, com o auxílio de duas interfaces, a base de conhecimento que contém o grafo, cujo exemplo pode ser observado na Figura 6. Essas interfaces são: 1) *FactAnalyzer*: analisa a AST e gera fatos em forma de grafo, constituindo, assim, a base de conhecimento (*knowledge base*); 2) *KnowledgeBaseProcessor*: processa a base de conhecimento, adicionando metadados e outras informações. A quarta camada tem o propósito de traduzir os fatos em linguagem Prolog e escrever o arquivo de saída, que é feito através da interface *KnowledgeBaseWriter*. Através dessas interfaces providas, se torna possível estender o extrator para suportar diversos tipos de linguagem que possam existir no repositório. A saída do módulo de extração é um arquivo escrito em Prolog que possui a base de conhecimento.

Figura 6 – Exemplo de grafo obtido a partir do Módulo de Extração

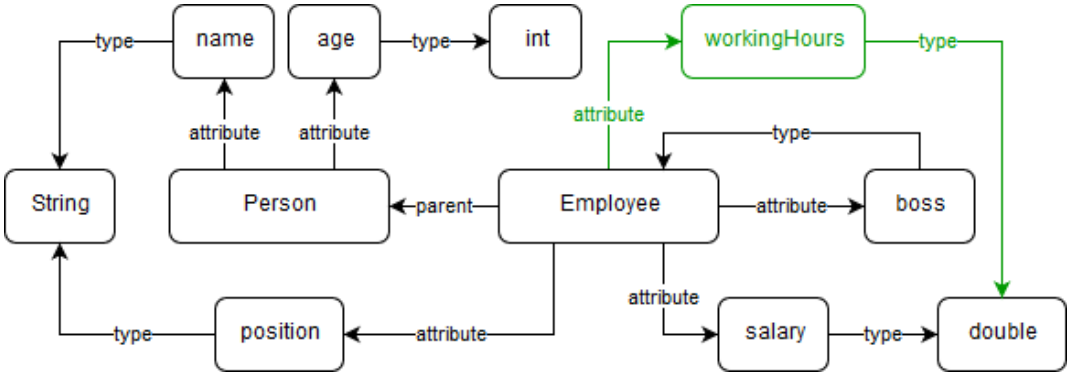



Fonte: elaborado pelo autor.

**Módulo de transformação**. O módulo de transformação é responsável por transformar o grafo gerado pelo módulo anterior. Ele é escrito em Prolog e também é acionado através da linha de comando. A entrada desse módulo é o arquivo que representa o modelo abstrato de grafo, o tipo de transformação a ser feito e os parâmetros para essa transformação. O módulo é dividido em quatro camadas: *orchestration*, *transformation*, *manipulation* e *synchronization*. A primeira camada, assim como no módulo anterior, interpreta a linha de comando e orquestra o fluxo da aplicação da transformação. A segunda camada comporta as transformações

implementadas. Cada transformação é responsável por realizar o *matching* no grafo e, subsequentemente, as alterações necessárias. A terceira camada oferece à primeira os mecanismos de manipulação do modelo abstrato de grafo, além dos algoritmos que permitem a extração de informações dele. A quarta camada é responsável pela sincronização do grafo e do código-fonte. Ela provê, além dos mecanismos de sincronização do grafo, a parte de integração com o módulo de injeção, identificando os parâmetros para realizar a injeção e o invocando com os argumentos necessários. A saída desse módulo é o modelo abstrato de grafo modificado como visto no exemplo da Figura 7.

Figura 7 – Exemplo de grafo modificado pela transformação




Fonte: elaborado pelo autor.

**Módulo de injeção**. O módulo de injeção trata de injetar as transformações no código-fonte, sincronizando assim a parte concreta do sistema. Ele é escrito em Java e, apesar de poder ser invocado através da linha de comando, ele é acionado de forma transparente pelo módulo de transformação. A entrada desse módulo são os arquivos-fonte a ser injetados, as linguagens em que são escritos, o tipo de injeção e um modelo JSON que contém os parâmetros para injeção. O módulo de injeção é dividido em quatro camadas tal como o módulo de extração: *orchestration*, *parsing*, *processing* e *consolidation*. Dessas camadas apenas as duas últimas fazem funções diferentes nesse módulo. A camada *processing* é responsável por interpretar o modelo JSON e realizar a injeção na AST fornecida pela camada *parsing*, sendo três interfaces usadas para isso: 1) *Model*: interpreta e fornece os parâmetros que constam no modelo JSON; 2) *Converter*: preocupa-se em converter os parâmetros contidos no *Model* para valores entendidos pela interface *Injector*; 3) *Injector*: faz uso do *Model* para injetar a transformação na AST. A camada *consolidation* tem o papel de escrever as mudanças

realizadas na AST para o arquivo-fonte original, usando, para isso, a interface *ASTWriter*. A saída desse módulo são os arquivos-fonte alterados.

### 4.3 Aspectos de implementação

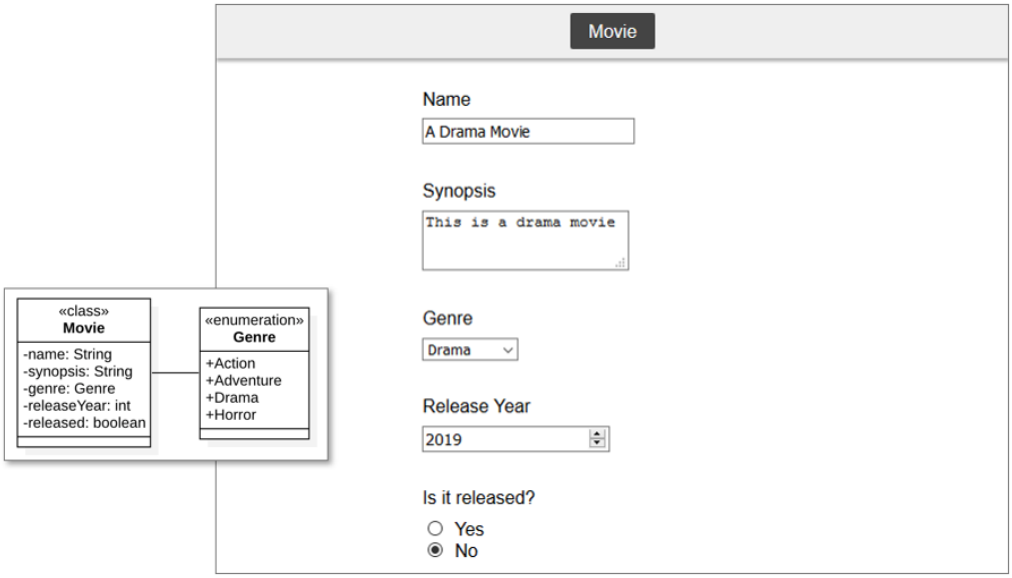
A arquitetura foi implementada como o *core* do Parthenos, provendo mecanismos e APIs que facilitam o desenvolvimento de extensões para ele. As extensões adicionam comportamento funcional ao Parthenos, já que, sozinho, ele não consegue gerar um grafo, transformar ou injetar o código-fonte de uma linguagem ou ferramenta específica. Dessa forma, desenvolveu-se dois conjuntos de extensões para o Parthenos, sendo que cada um provê extensões para os módulos de extração, transformação e injeção do modelo separadamente.

O primeiro conjunto de extensões se preocupa em prover suporte à linguagem Java. Ele possibilita o modelo abstrato de grafo a conter as representações de classes e suas relações, atributos, métodos, além de seus metadados. Ele adiciona, ainda, duas transformações: *criar* classe e *adicionar* atributo (e seus métodos de acesso *get* e *set*). Nesse conjunto de extensões utilizou-se linguagem Java para estender os módulos de extração e injeção e linguagem Prolog para o módulo de transformação.

O segundo conjunto de extensões trata de prover suporte à ferramenta Pojo UI. Essa ferramenta foi desenvolvida junto desse trabalho para demonstrar a capacidade, que a arquitetura proposta tem, de receber novas extensões. Ela cria, através de anotações de classes Java e seus atributos, uma interface gráfica básica, que exibe painéis que representam as classes, onde cada painel apresenta um formulário sem funcionalidade que possui campos refletindo os atributos da classe e suas respectivas tipagens. Os painéis e campos podem ter seus rótulos, posições e visibilidades modificados também através das anotações. Um exemplo de interface gerada pela ferramenta pode ser visto na Figura 8. A extensão para a Pojo UI confere ao grafo vértices e arestas que representam os painéis e campos, além de suas propriedades, ligadas às classes e atributos extraídos com a extensão para linguagem Java. Ao módulo de transformação são adicionadas transformações que permitem criar e remover os painéis e campos e, também, modificar os atributos de rótulo, posição e visibilidade de cada. A Pojo UI é escrita em linguagem Java e gera arquivos nas linguagens JavaScript, HTML e CSS. A respectiva extensão é desenvolvida usando linguagem Java para os módulos de extração e injeção e linguagem Prolog para o

módulo de transformação. As implementações da arquitetura e extensões podem ser encontradas no GitHub[1].

Figura 8 – Exemplo de interface gerada pela Pojo UI




Fonte: elaborado pelo autor.

## 5 AVALIAÇÃO

Para medir a eficácia do Parthenos ao realizar a etapa de injeção do código-fonte proposto pelo modelo MITRAS, elencou-se uma série de métricas, que são observadas ao se aplicar transformações em um modelo abstrato de grafo. Essas métricas são aplicadas através de quatro cenários que cobrem a funcionalidade do Parthenos e, unidas aos cenários, são avaliadas através das medidas de *precision*, *recall* e *f-measure*. (VAN RIJSBERGEN, 1979).

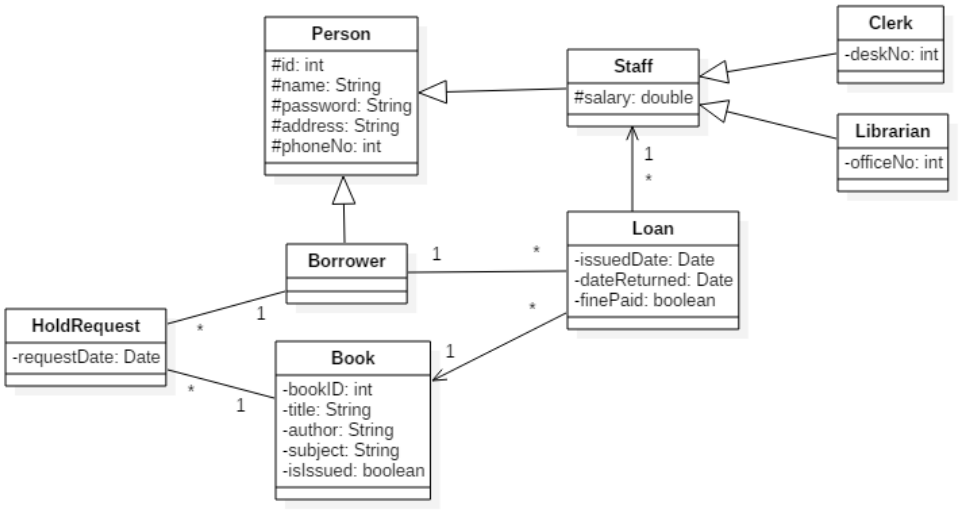
### 5.1 Método de avaliação

Como forma de avaliar o Parthenos, aplicou-se quatro cenários de transformação a um sistema de gerenciamento de biblioteca disponível em um repositório do GitHub[2]. O sistema possui um diagrama de classes como visto na Figura 9 e foi ajustado para satisfazer a necessidade dos cenários. Dentre os ajustes

[1] Endereço dos repositórios: https://github.com/gabelopes.
[2] Endereço do repositório: https://github.com/harismuneer/Library-Management-System-JAVA.

feitos, estruturou-se o sistema para suportar o *framework* Maven e adicionou-se como dependência a biblioteca da Pojo UI.

Figura 9 – Diagrama de classes do sistema de gerenciamento de biblioteca




Fonte: adaptado do GitHub[2].

Em cada cenário executado, comparam-se as métricas obtidas ao se aplicar uma transformação (e, por consequência, se injetar o código-fonte) com as métricas obtidas a partir da extração do modelo transformado manualmente. Isso, porque o modelo obtido a partir do extrator é sempre o desejado. Para isso, são estabelecidos modelos que indicam os estados do sistema durante a execução de um cenário. Os modelos são: 1) modelo inicial $M_A$, que representa o estado do sistema antes de se aplicar o cenário; 2) um modelo intermediário $M_B$ contendo os elementos, como classes, atributos, fatos, etc., gerados pela aplicação das transformações e injeções feitas pelo cenário; 3) um modelo esperado $M_{AB}$, que representa o produto desejado obtido a partir da execução desse cenário; 4) um modelo $M_C$, que é o resultado obtido da aplicação do cenário. Esses modelos trazem métricas numéricas que servem como entrada para as fórmulas das medidas *precision*, *recall* e *f-measure*. Essas medidas serão usadas como forma de mensurar a acurácia do Parthenos ao executar cada cenário.

A medida *precision* indica qual a exatidão do resultado obtido. Como pode ser visto na fórmula $P = \frac{|M_C \cap M_{AB}|}{|M_C|}$, ela calcula a porcentagem das adições feitas aos modelos que são relevantes para o cenário executado. (CHINCHOR, 1992). A medida *recall*, por sua vez, indica se todos os resultados esperados foram obtidos, isto é, qual

o percentual das adições esperadas que foram, de fato, feitas aos modelos. (CHINCHOR, 1992). A fórmula da medida é dada por $\mathrm{R} = \frac{|M_C \cap M_{AB}|}{|M_{AB}|}$. Por fim, a medida *f-measure* calcula a média harmônica entra as medidas *precision* e *recall*, a fim de combiná-las em uma medida unificada que pontue a efetividade do Parthenos no cenário executado. Essa pontuação varia de 0 a 1, sendo 1 o melhor resultado e 0 o pior. A *f-measure* é importante, pois medir a eficácia de um cenário observando-se somente a *precision* ou a *recall* é incompleto e impreciso, já que elas têm papel complementar nesse sentido. (CHINCHOR, 1992). A fórmula da *f-measure* é dada por $\mathrm{F} = 2 \cdot \frac{P \cdot R}{P+R}$.

## 5.2 Métricas

As métricas selecionadas para avaliação do Parthenos têm como objetivo mensurar a acurácia da base de conhecimento e do código-fonte modificados quando se aplicam uma ou várias transformações sucessivas ao modelo abstrato de grafo. Como pode ser observado no Quadro 2, elas abrangem desde a quantidade de elementos gerados ou modificados até a correção desses elementos.

Quadro 2 – Métricas estabelecidas para avaliação

| *Métrica* | *Descrição* |
|---|---|
| *Classes* | Quantidade de classes presentes no sistema. |
| *Attributes* | Quantidade de atributos presentes no sistema. |
| *Panels* | Quantidade de painéis processados pela Pojo UI. |
| *Fields* | Quantidade de campos processados pela Pojo UI. |
| *Syntax* | Quantidade de classes com sintaxe correta. |
| *Semantics* | Quantidade de classes com semântica correta. |
| *KB* | Quantidade de fatos na base de conhecimento, ou seja, quantidade de elementos presentes no grafo. |

Fonte: elaborado pelo autor.

## 5.3 Cenários de avaliação

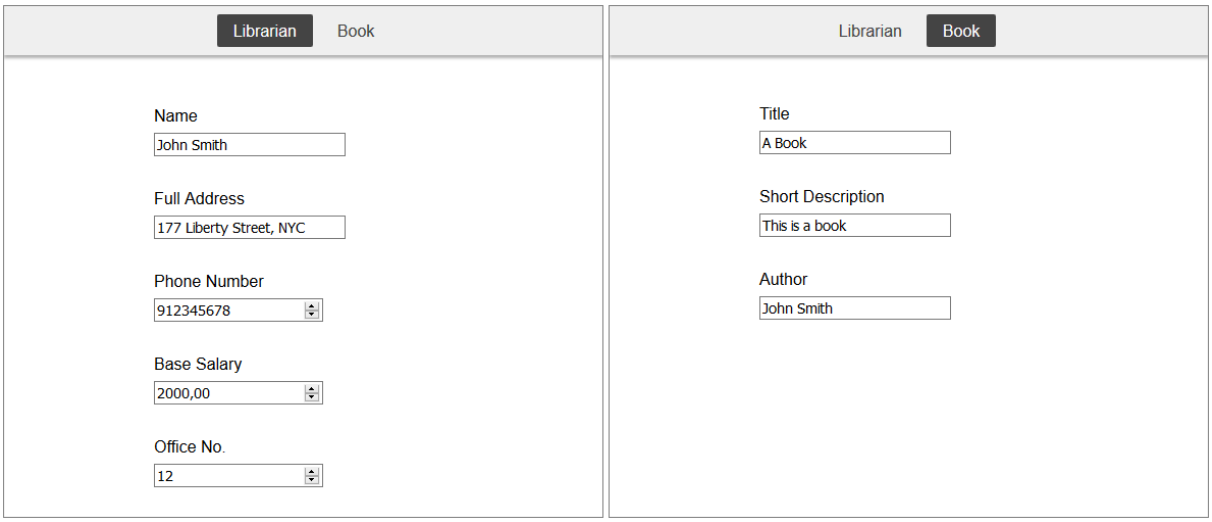
Antes de iniciar os cenários, o modelo abstrato de grafo do sistema de gerenciamento de biblioteca foi extraído, utilizando-se o módulo de extração. Os

cenários mostram a evolução adaptativa e perfectiva desse sistema através de aplicações de transformações ao modelo extraído. Os cenários são descritos nas seções a seguir e, para cada um, são exibidas as métricas obtidas.

### 5.3.1 Cenário 1

Nesse cenário inicia-se o *site* do sistema através da Pojo UI. Na Figura 10, observa-se a adição painéis para as classes *Librarian* e *Book*. Também, são adicionados campos para os atributos *name*, *address*, *phoneNo*, *salary* e *officeNo* na classe Librarian. Ao *Book* são adicionados campos para *title*, *subject* e *author*.

Figura 10 – Site inicial gerado pela Pojo UI




Fonte: elaborado pelo autor.

Tabela 1 – Métricas obtidas no cenário 1

| *Métrica* | $M_A$ | $M_B$ | $M_{AB}$ | $M_C$ | Precision | Recall | F-measure |
|---|---|---|---|---|---|---|---|
| *Classes* | 8 | 8 | 8 | 8 | 1 | 1 | 1 |
| *Attributes* | 17 | 17 | 17 | 17 | 1 | 1 | 1 |
| *Panels* | 0 | 4 | 4 | 4 | 1 | 1 | 1 |
| *Fields* | 0 | 8 | 8 | 8 | 1 | 1 | 1 |
| *Syntax* | 8 | 8 | 8 | 8 | 1 | 1 | 1 |
| *Semantics* | 8 | 8 | 8 | 8 | 1 | 1 | 1 |
| *KB* | 1301 | 68 | 1369 | 1369 | 1 | 1 | 1 |
| | | | | | | Média: | 1 |

Fonte: elaborado pelo autor.

### 5.3.2 Cenário 2

Nesse cenário adiciona-se a classe *Magazine* ao sistema. À classe *Magazine* são adicionados os atributos *title*, *subject*, *publisher* e *issueNo*. Um painel para a classe e campos para os atributos são criados também como visto na Figura 11.

Figura 11 – Painel da classe *Magazine* refletindo a classe gerada


Fonte: elaborado pelo autor.

Tabela 2 – Métricas obtidas no cenário 2

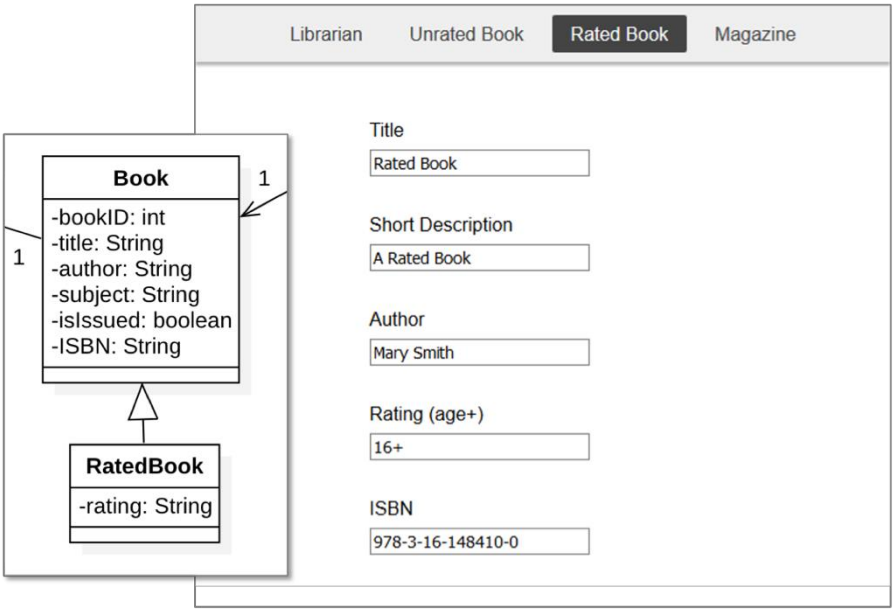
| *Métrica* | $M_A$ | $M_B$ | $M_{AB}$ | $M_C$ | Precision | Recall | F-measure |
|---|---|---|---|---|---|---|---|
| *Classes* | 8 | 1 | 9 | 9 | 1 | 1 | 1 |
| *Attributes* | 17 | 4 | 21 | 21 | 1 | 1 | 1 |
| *Panels* | 4 | 1 | 5 | 5 | 1 | 1 | 1 |
| *Fields* | 8 | 4 | 12 | 12 | 1 | 1 | 1 |
| *Syntax* | 8 | 1 | 9 | 9 | 1 | 1 | 1 |
| *Semantics* | 8 | 1 | 9 | 9 | 1 | 1 | 1 |
| *KB* | 1369 | 142 | 1511 | 1496 | 1 | 0,99 | 0,99 |
| | | | | | | Média: | 0,99 |

Fonte: elaborado pelo autor.

### 5.3.3 Cenário 3

Este cenário cria uma classe *RatedBook* que estende a classe *Book* adicionando um atributo *rating*. Além disso, é adicionado à classe *Book* um atributo *ISBN*. Observando a Figura 12, é possível ver que os respectivos painel e campos são adicionados e modificações cosméticas são feitas: o painel *Book* é renomeado para “Unrated Book” e o painel *RatedBook* é movido para antes do painel *Magazine*.

Figura 12 – Alterações realizadas pelo Cenário 3




Fonte: elaborado pelo autor.

Tabela 3 – Métricas obtidas no cenário 3

| *Métrica* | $M_A$ | $M_B$ | $M_{AB}$ | $M_C$ | *Precision* | *Recall* | *F-measure* |
|---|---|---|---|---|---|---|---|
| *Classes* | 9 | 1 | 10 | 10 | 1 | 1 | 1 |
| *Attributes* | 21 | 2 | 23 | 23 | 1 | 1 | 1 |
| *Panels* | 5 | 1 | 6 | 6 | 1 | 1 | 1 |
| *Fields* | 12 | 2 | 14 | 14 | 1 | 1 | 1 |
| *Syntax* | 9 | 2 | 10 | 10 | 1 | 1 | 1 |
| *Semantics* | 9 | 2 | 10 | 9 | 1 | 0,9 | 0,94 |
| *KB* | 1496 | 92 | 1588 | 1577 | 1 | 0,99 | 0,99 |
| | | | | | | Média: | 0,99 |

Fonte: elaborado pelo autor.

### 5.3.4 Cenário 4

O cenário 4 tenta realizar a adição de uma classe *PuzzleBook* que estenda uma classe *UnratedBook* (inexistente). Após isso tenta adicionar um atributo *puzzleType* à classe *PuzzleBook*. O resultado esperado nesse cenário, diferente dos outros, é que ambas transformações falhem e os modelos continuem iguais.

Tabela 4 – Métricas obtidas no cenário 4

| *Métrica* | $M_A$ | $M_B$ | $M_{AB}$ | $M_C$ | Precision | Recall | F-measure |
|---|---|---|---|---|---|---|---|
| *Classes* | 10 | 0 | 10 | 10 | 1 | 1 | 1 |
| *Attributes* | 23 | 0 | 23 | 23 | 1 | 1 | 1 |
| *Panels* | 6 | 0 | 6 | 6 | 1 | 1 | 1 |
| *Fields* | 14 | 0 | 14 | 14 | 1 | 1 | 1 |
| *Syntax* | 10 | 0 | 10 | 10 | 1 | 1 | 1 |
| *Semantics* | 9 | 0 | 9 | 9 | 1 | 1 | 1 |
| *KB* | 1577 | 0 | 1577 | 1577 | 1 | 1 | 1 |
| | | | | | | Média: | 1 |

Fonte: elaborado pelo autor.

## 5.3 Discussão e limitações

As *f-measures* obtidas tiveram valor máximo ou muito próximo disso em todos os cenários apresentados na seção 5.2. Assim, verifica-se que Parthenos obteve excelente resultado e, portanto, foi eficaz ao extrair, transformar e injetar os modelos. Além disso, é possível observar que o Parthenos permite cobrir diversos cenários com necessidades diferentes em cada um, evitando erros sintáticos e semânticos no que tange as tipagens. A arquitetura extensível se mostrou muito importante, já que permite que sistemas escritos nas mais variadas linguagens e que usem frameworks, bibliotecas e ferramentas de diferentes tipos possam ser representados e transformados com facilidade através de um modelo abstrato de grafo, permitindo, assim, a usuários finais de diversos sistemas realizarem manutenções sem possuir conhecimento de programação.

Apesar de ter se mostrado bastante eficaz, o Parthenos falha em alguns aspectos. Como é possível ver na Tabela 2, nem todos os elementos desejados do

modelo abstrato de grafo foram gerados. Isso acontece porque o módulo de extração possui uma etapa de processamento da base de conhecimento que adiciona metadados e outras informações ao grafo que não são conhecidas pelo módulo de transformação quando elementos completamente novos são adicionados, como é o caso de classes. Isso pode acarretar problemas para a semântica de tipos do grafo em cenários específicos, mas não atrapalhou na execução do restante dos cenários. Na Tabela 3, observa-se que uma das classes não está semanticamente correta. Esse problema ocorreu, pois, devido a uma especificidade da linguagem Java, a classe *RatedBook*, que estende a *Book*, necessitava possuir um construtor padrão. Como o Parthenos lida com as linguagens no nível sintático (nas ASTs), ele não consegue lidar com as semânticas inerentes a elas, podendo ocasionar erros semânticos mais graves. Nesse caso, o sistema deixou de compilar e, para poder continuar gerando a interface gráfica, usando a Pojo UI, foi necessário contornar esse problema manualmente.

## 6 CONCLUSÃO

Este trabalho apresentou o Parthenos, uma abordagem de implementação do modelo MITRAS que tem foco em realizar a etapa de injeção do modelo garantindo a correção sintática e a correta semântica de tipos, além de garantir a sincronia entre o modelo abstrato de grafo e o código-fonte do sistema. A arquitetura proposta pelo Parthenos é robusta e extensível, permitindo que sistemas escritos em variadas linguagens, que usem frameworks, bibliotecas e ferramentas diversas possam usufruir do modelo proposto pelo MITRAS.

Os resultados obtidos se mostraram bastante promissores nos cenários avaliados. Eles foram possíveis devido à extensibilidade oferecida pela arquitetura e, também, devido à eficácia do módulo de extração em gerar as informações e metadados necessários. Dessa forma, o grafo pôde ser devidamente operado na etapa de transformação, consequentemente, permitindo a realização da injeção no código-fonte de forma acurada.

Em trabalhos futuros sugere-se explorar uma maneira de se injetar código-fonte observando-se as semânticas das linguagens, como, por exemplo, implementar um módulo que analise semanticamente os arquivos injetados e proveja a capacidade de se recuperar, automaticamente, de erros semânticos não-ambíguos. Outra

possibilidade é prover a capacidade de gerar código que adicione funcionalidades a sistemas de forma automática através de transformações. Em ambos casos, a arquitetura e a implementação feitas pelo Parthenos poderão servir como base para o desenvolvimento.

## REFERÊNCIAS


AHO, Alfred V.; LAM, Monica S.; SETHI, Ravi.; ULLMAN, Jeffrey D. **Compilers**: principles, techniques, and tools. 2. ed. Boston: Pearson Education, 2007.

ANTAL, Gábor; HAVAS, Dávid; SIKET, István; BESZÉDES, Árpád; FERENC, Rudolf; MIHALICZA, József. **Transforming C++11 Code to C++03 to Support Legacy Compilation Environments**. *In*: 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM), Raleigh, 2016. *[S. l.]*: IEEE, p. 177-186.

BONDY, John Adrian; MURTY, U. S. R. **Graph Theory with Applications**. Amsterdã: North-Holland, 1976.

CHAGAS, Michael William; FARIAS, Kleinner; GLUZ, João Carlos; KÜPSSINSKI, Lucas Silveira; GONÇALVES, Lucian. **Hermes**: a natural language interface model for software transformation. *In*: XV BRAZILIAN SYMPOSIUM ON INFORMATION SYSTEMS, 15., 2019, Aracajú. **Anais** […] Porto Alegre: SBC, 2019.

CHINCHOR, Nancy. **MUC-4 Evaluation Metrics**. *In*: MUC4 '92 Proceedings of the 4th conference on Message understanding, 1992, McLean. **Anais** [...] Stroudsburg: Association for Computational Linguistics, 1992. p. 22-29.

DE ESPINDOLA, Rodrigo Santos; MAJDENBAUM, Azriel; AUDY, Jorge Luis Nicolas. **Uma Análise Crítica dos Desafios para Engenharia de Requisitos em Manutenção de Software**. *In*: WER, Tandil, 2004. Gávea: PUC-Rio, p. 226-238.

DE OLIVEIRA, Paulo Henrique Ribeiro. **Engenharia de requisitos aplicada em sistema legado de gestão e custeio de propostas comerciais**: pesquisa-ação em empresa do setor de estamparia. 2016. Dissertação (Programa de Mestrado em Engenharia de Produção) – Universidade Nove de Julho, São Paulo, 2016.

GERSTING, Judith L. **Mathematical Structures for Computer Science**. Nova Iorque: W. H. Freeman and Company, 2006.

GIL, Yossi; ORRÙ, Matteo. **The Spartanizer**: massive automatic refactoring. *In*: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Klagenfurt, 2017. *[S. l.]*: IEEE, p. 477-481.

HOPCROFT, John E.; MOTWANI, Rajeev; ULLMAN, Jeffrey D. **Introduction to Automata Theory, Languages, and Computation**. 2. ed. *[S. l.]*: Pearson Education, 2001.

JENSEN, Kathleen; WIRTH, Niklaus. **Pascal User Manual and Report**: ISO Pascal Standard. 4. ed. Nova Iorque: Springer-Verlag New York, 1991.

KAHANI, Nafiseh; BAGHERZADEH, Mojtaba; CORDY, James R.; DINGEL, Juergen; VARRÓ, Daniel. Survey and classification of model transformation. **Software & Systems Modeling**, *[s. l.]*, p. 1-37, Mar. 2018.

KNAUER, Ulrich. **Algebraic Graph Theory**: morphisms, monoids and matrices. Berlim: De Gruyter, 2011.

KÜPSSINSKI, Lucas Silveira. **MITRAS**: modelo inteligente para transformação de aplicações de software. 2019. Dissertação (Mestrado em Computação Aplicada) – Programa de Pós-Graduação em Computação Aplicada, Universidade do Vale do Rio dos Sinos, São Leopoldo, 2019.

MENEZES, Paulo Blauth. **Linguagens Formais e Autômatos**. 3. ed. Porto Alegre: Sagra Luzzatto, 2000.

OLIVEIRA, Anderson; BISCHOFF, Vinicius; GONÇALVES, Lucian José; FARIAS, Kleinner; SEGALOTTO, Matheus. BRCode: an interpretive model-driven engineering approach for enterprise applications. **Computers in Industry**, *[s. l.]*, p. 86-97, Apr. 2018.

PADUELLI, Mateus Maida. **Manutenção de Software**: problemas típicos e diretrizes para uma disciplina específica. 2007. Dissertação (Mestrado em Ciências de Computação e Matemática Computacional) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 2007.

ROZENBERG, Grzegorz. **Handbook of Graph Grammars and Computing by Graph Transformation**. River Edge: 1997. v. 1.

VAN RIJSBERGEN, C. J. **Information Retrieval**. 2. ed. Newton: Butterworth-Heinemann, 1979.

ZAFEIRIS, Vassilis E.; POULIAS, Sotiris H.; DIAMANTIDIS, N. A.; GIAKOUMAKIS, E. A. Automated refactoring of super-class method invocations to the Template Method design pattern. **Information and Software Technology**, v. 82, p. 19–35, Fev. 2017.