UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

DEPARTAMENTO ACADÊMICO DE INFORMÁTICA

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RODRIGO DE FRANÇA MISS NAIRNEI

PROPOSTA DE UM MODELO USANDO AGENTES PARA REFATORAÇÃO DE SOFTWARE

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA

2018

RODRIGO DE FRANÇA MISS NAIRNEI

PROPOSTA DE UM MODELO USANDO AGENTES PARA REFATORAÇÃO DE SOFTWARE

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação, do Departamento Acadêmico de Informática, da Universidade Tecnológica Federal do Paraná.

Orientador: Profª. Drª. Simone Nasser Matos

PONTA GROSSA

2018

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Ponta Grossa
Diretoria de Graduação e Educação Profissional
Departamento Acadêmico de Informática
Bacharelado em Ciência da Computação

TERMO DE APROVAÇÃO

PROPOSTA DE UM MODELO USANDO AGENTES PARA REFATORAÇÃO DE SOFTWARE

por

RODRIGO DE FRAÇA MISS NAIRNEI

Este Trabalho de Conclusão de Curso (TCC) foi apresentado em 23 de novembro de 2018 como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Profª. Dra. Simone Nasser Matos
Orientadora

_____
Prof. Dr. Tarcizio Alexandre Bini
Membro titular

_____
Prof. MSc Vinicius Camargo Andrade
Membro titular

_____
Profª. Dra. Helyane Borges
Responsável pelo Trabalho de Conclusão de Curso

_____
Prof. Dr. Saulo Jorge Beltrão de Queiroz
Coordenador do Curso

- O Termo de Aprovação assinado encontra-se na Coordenação do Curso -

Dedico este trabalho à minha família pelo
amor e carinho, aos meus amigos e professores
pela paciência.

AGRADECIMENTOS

À minha família pelo incentivo e apoio incondicional.
A minha orientadora pelas orientação e paciência.
Aos meus colegas de sala e amigos.

RESUMO

NAIRNEI, Rodrigo. **Proposta de um modelo usando agentes para refatoração de software**. 2018. 61 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2018.

A refatoração de software tem como finalidade aumentar a qualidade do projeto em relação aos atributos de reusabilidade, manutenibilidade e legibilidade. Existem métodos de refatoração que são capazes de ler um código-fonte e detectar e inserir padrões de projeto. As ferramentas automatizadas implementam somente um método de refatoração, por isto, o desenvolvedor deve executar seu projeto em várias ferramentas para conseguir detectar e inserir todos os possíveis padrões de projeto em seu código-fonte. Este trabalho propõe a modelagem de um sistema multiagentes para unificar os métodos de refatoração em um único ambiente contemplando um modelo de agente capaz de ler um código-fonte e refatorá-lo usando padrões de projeto. A metodologia *Prometheus* foi utilizada na criação do modelo proposto. Como resultado, é apresentado um cenário de teste no qual o modelo criado é capaz de ler um código e aplicar padrões de projeto.

Palavras-chave: Modelagem. Agentes. Refatoração de software.

ABSTRACT

NAIRNEI, Rodrigo. **Proposal of a model using agents for software refactoring.** 2018. 61 p. Work of Conclusion Course Graduation in Computer Science - Federal Technology University - Paraná. Ponta Grossa, 2018.

Software refactoring aims to increase the quality of the project in relation to reusability, maintainability and legibility attributes. There are refactoring methods that are able to read a source code and detect and insert design patterns. Automated tools implement only one refactoring method, so the develop must run your project on multiple tools to be able to detect and insert all possible design patterns into your source code. This work proposes the modeling of a multi-agent system to unify the refactoring methods in a single environment by contemplating an agent model capable of reading a source code and refactoring it using design patterns. The Prometheus methodology was used in the creation of the proposed model. As a result, a test scenario is presented in which the created model is able to read a code and apply design patterns.

Keywords: Modeling. Agents. Software refactoring.

LISTA DE QUADROS

LISTA DE FIGURAS

LISTA DE SIGLAS

AOP	*Agent-Oriented Programming*
AUML	*Agent Unified Modeling Language*
AST	Árvore de Sintaxe Abstrata
BDI	*Believe Desire Intention*
JADE	*Java Agent Development framework*
JDE	*Jack Development Environment*
JIAD	*Java based Intent Aspects Detector*
MORE	*Multi-objective Refactoring Recommendation*
PDT	*Prometheus Design Tool*

SUMÁRIO

1 INTRODUÇÃO

Os padrões de projetos representam estruturas que permitem atingir os requisitos de qualidade durante o processo de refatoração de software (TSANTALIS *et al*., 2006). Um padrão de projeto nomeia, abstrai e identifica os aspectos-chave de uma estrutura de projeto comum para torná-la útil na criação de um projeto orientado a objetos reutilizável. Cada padrão de projeto foca em um problema ou tópico particular e descreve em que situação pode ser aplicado, quais restrições e suas consequências, custos e benefícios de sua utilização (GAMMA *et al*., 2000). Existem 24 (vinte e quadro) padrões de projetos, divididos em três categorias: criacionais, estruturais e comportamentais (GAMMA *et al*., 2000).

Desenvolver software de qualidade não é uma tarefa trivial e é considerada árdua para projetistas sem experiência, o qual deve decidir qual padrão de projeto usar e para isto considera-se sua natureza, por quem será usado e a sua linguagem. Uma forma de atingir a qualidade é obtida pelo uso de refatoração (GAMMA *et al*., 2000).

A refatoração é uma técnica controlada para aperfeiçoar um projeto de software. Refatorar consiste em aplicar uma série de transformações, que mesmo pequenas, transformam o código-fonte em algo melhor quando comparada à versão original sem alterar seus resultados (FOWLER; BECK, 1999).

Existem diversos métodos de refatoração propostos na literatura, como os métodos de Cinnéide e Nixon (2001); Mens e Tourwé (2001); Jeon, Lee e Bae (2002); Rajesh e Janakim (2004); Liu *et al* (2014); Gaitani *et al* (2015); Ouni *et al.* (2017). Estes métodos foram identificados por meio de uma revisão sistemática realizada por Beluzzo (2018). Os estudos de Gaitani *et al*. (2015), Liu *et al*. (2014), Zafeiris *et al*. (2017), por exemplo, utilizam árvores de sintaxe abstrata (AST) como seu meio de extração de dados do código-fonte. Quando se trata da obtenção de pontos de inserção de padrões, os autores Jeon, Lee e Bae (2002) e Rajesh e Janakim (2004) utilizam a mesma técnica baseada em fatos e regras Prolog. Existem ferramentas que implementam um único método de refatoração e isto dificulta o trabalho do desenvolvedor, pois precisa utilizar de várias ferramentas para aplicar padrões de projeto em seu código-fonte.

Os padrões de projetos e os processos de refatoração estão presentes na área da Engenharia de Software. Esta área é voltada para especificação, desenvolvimento, manutenção e criação de sistema de software. A Engenharia de Software orientada a objetos possui técnicas para a construção de sistemas de computação com comportamentos pré-definidos. Portanto, o

software irá executar aquilo que foi planejado e qualquer situação não contemplada no momento do desenvolvimento pode provocar falhas no sistema. A Engenharia de Software tem bases conceituais e origens no campo de estudo da cognição que também influencia a área de Inteligência Artificial (GIRARDI, 2004).

Na inteligência artificial, um objeto em um software orientado a objetos é modelado como agente. Um agente é uma entidade autônoma que percebe seus ambientes através de sensores e age sobre o este ambiente utilizando os executores. Por exemplo, nos agentes humanos os olhos e ouvidos são sensores, enquanto as mãos e boca são executores (RUSSEL, 1995).

Segundo Girardi (2004) "o desenvolvimento baseado em agentes difere da Engenharia de Software tradicional, pois a Engenharia de software baseada em agentes fornece soluções para abordar a crescente complexidade dos sistemas de computação. Estes sistemas operam geralmente em ambientes não previsíveis, abertos e que mudam rapidamente e têm autonomia para decidir por si mesmos o que fazer em qualquer situação para alcançar seus objetivos".

Para o desenvolvimento baseado em agentes, existem diversas metodologias tais como Tropos, Gaia e Prometheus (BERNY *et al*., 2008; WOOLDRIDGE, 2002). Uma metodologia de desenvolvimento de Sistema Multiagentes é constituída por uma série de passos e procedimentos a serem seguidos durante o processo de concepção do sistema. Ela deve capturar a flexibilidade, autonomia dos agentes, com variados graus de abstração, auxiliando o projetista nas tomadas de decisão relativas à análise, projeto e implementação (BERNY *et al.*, 2005).

Este trabalho propõe a modelagem de um sistema multiagentes para refatoração de software usando como fundamento os métodos de detecção e inserção de padrões de projeto. A modelagem proposta foi construída usando a metodologia *Prometheus*. Como resultado deste trabalho, foi apresentando um caso de teste para avaliação do modelo gerado, apresentando de forma abstrata a relação dos agentes envolvidos com seus arquivos e diretórios, além das percepções e ações que eles realizam no ambiente em que estão inseridos.

1.1 OBJETIVOS

O objetivo geral deste trabalho é propor um modelo de agente capaz de ajudar no processo de refatoração de software utilizando os métodos de refatoração para detecção e inserção de padrões de projeto da literatura a fim oferecer um ambiente unificado ao desenvolvedor para aplicação de padrões de projeto.

Os objetivos específicos para o desenvolvimento deste trabalho são:

- Realizar uma análise comparativa entre os métodos de refatoração.
- Identificar metodologias e ferramenta para modelagem de agentes.
- Aplicar o modelo proposto em um cenário de teste.

1.2 ORGANIZAÇÃO DO TRABALHO

Este trabalho está estruturado em seis capítulos. O capitulo 2 faz um levantamento bibliográfico sobre agentes. O capitulo 3 apresenta conceitos sobre a modelagem de sistemas baseados em agentes e suas metodologias.

O capítulo 4 faz o relato sobre refatoração de software, técnicas e métodos de detecção de pontos de inserção de padrões de projetos.

O Capitulo 5 descreve o desenvolvimento do modelo utilizando a metodologia *Prometheus* e a aplicação do modelo em um cenário de teste. O capitulo 6 apresenta as conclusões e trabalhos futuros.

2 AGENTES

Este capítulo apresenta uma visão geral sobre agentes. A seção 2.1 relata as definições sobre agentes dadas pela literatura. A seção 2.2 apresenta o funcionamento da arquitetura BDI (*Belief-Desire-Intention*). A seção 2.3 descreve o que são sistemas multiagentes. A Seção 2.4 apresenta linguagens e ferramentas para implementar agentes e relata uma comparação entre o desenvolvimento de um código com e outro sem agentes. Por fim, a última seção apresenta as considerações finais do capítulo.

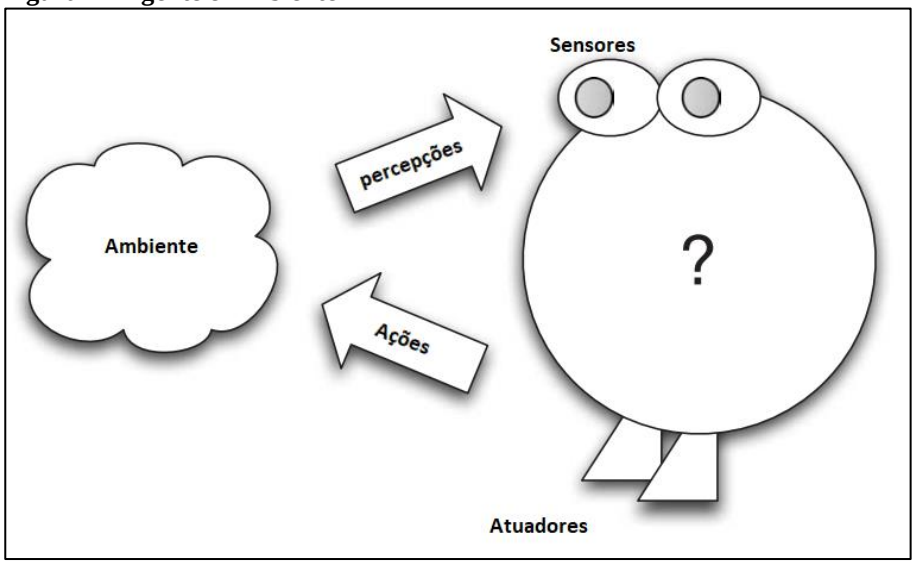
2.1 DEFINIÇÃO DE AGENTE

Shoham (1993) diz que o termo agente se refere a uma entidade que funciona de forma continua e autônoma em um ambiente no qual podem existir outros agentes. Wooldridge (2002) conceituam um agente como um sistema de computador que está situado em um ambiente e que é capaz de realizar ações a fim de atingir seus objetivos.

Bordini (2007) considera que um agente é um sistema que está situado em um ambiente em que o sistema é capaz de receber percepções do ambiente por meio de sensores e possui uma base de conhecimento para possíveis execuções de atuadores para modificar seu ambiente.

A Figura 1 apresenta uma representação de um agente. O ambiente em que o agente está inserido pode ser um espaço físico ou virtual, como por exemplo, um campo de golfe ou um arquivo de texto. Seus sensores são quaisquer meios que o agente possui para observar o ambiente e seus atuadores são as ferramentas que ele disponibiliza para modificar o ambiente (BORDINI, 2007).

Figura 1 - Agente e Ambiente



Fonte: Adaptado de (Bordini, 2007)

Em aplicações mais realistas, os agentes não possuem controle total do ambiente mesmo que eles possam executar ações que alteram o ambiente em que estão inseridos. O ambiente não pode ser controlado completamente, pois podem existir outros agentes inseridos (BORDINI, 2007).

Como diversos agentes podem estar inseridos em um mesmo ambiente, Wooldridge e Jennigs (1995) (apud BORDINI, 2007) definem outras propriedades fundamentais para agentes racionais como: Autonomia; Proatividade; Reatividade e Habilidade Social. A autonomia expressa a capacidade de um agente operar independentemente a fim de atingir as metas delegadas a ele. A proatividade descreve que o agente deve ser capaz de realizar comportamentos direcionados por objetivos. A reatividade descreve que o agente deve ser capaz de reagir às mudanças do ambiente. A capacidade social descreve que os agentes devem ser capazes de trocar informações com outros agentes a fim de realizar seus objetivos.

Para Wooldridge e Jennings (1995), os agentes podem ser classificados em 3 arquiteturas na abordagem clássica: arquitetura deliberativa, abordagem alternativa (arquitetura reativa) e arquitetura híbrida.

Arquitetura deliberativa ou baseada em lógica contém um modelo simbólico representando explicitamente o ambiente e suas decisões são feitas por meio de lógica. As principais dificuldades de utilizar esta arquitetura é a tradução do problema, ou seja, traduzir o problema real para um sistema simbólico significante (WOOLDRIDGE; JENNINGS, 1995).

Nas arquiteturas reativas não está incluso o modelo simbólico do ambiente e não há o uso de raciocínios simbólicos complexos, como os utilizados nas arquiteturas deliberativas (WOOLDRIDGE; JENNINGS, 1995). Esta arquitetura se resume a receber estímulos externos por meio de sensores, processá-los e responder ao ambiente utilizando dos atuadores.

E por fim, a arquitetura híbrida realiza uma combinação da arquitetura deliberativa e reativa. Conforme Wooldridge e Jennings (1995), os agentes dessa arquitetura possuem dois ou mais subsistemas: um sistema deliberativo contendo uma representação simbólica do ambiente e um sistema reativo capaz de responder de forma rápida às mudanças do ambiente. Esta arquitetura pode ser chamada de em camadas, pois os subsistemas são organizados em camadas.

Existe uma quarta arquitetura, definida como Arquitetura PRS (*Procedural Reasoning System*), Arquitetura de Sistema de raciocínio procedural ou frequentemente referido como uma arquitetura crença-desejo-intenção (BDI) (*Belief - Desire - Intention*) (WOOLDRIDGE, 2000). As ideias básicas da abordagem BDI baseiam-se na descrição do processamento interno de um agente utilizando um conjunto básico de estados mentais (crença, desejo e intenções) (ZAMBERLAM; GIRAFFA, 2001). Nesta arquitetura os agentes possuem um conjunto de planos pré-compilados e são desenvolvidos manualmente pelo programador do agente e possuem os seguintes componentes: uma meta, um contexto e um "corpo ou receita" que representa como o plano deve ser executado (WOOLDRIDGE, 2000).

Para este trabalho foi utilizada a arquitetura BDI pois ela permite um detalhamento maior das funcionalidades dos agentes, como está detalhada na próxima seção. Os agentes que que foram modelados são classificados como agentes reativos, pois os agentes agem de acordo com sua percepção atual.

2.2 A ARQUITETURA DE AGENTE BDI

A Arquitetura BDI (*Belief - Desire - Intention*) pode ser classificada como uma arquitetura de estados mentais. Esta abordagem descreve o processamento interno do agente utilizando um conjunto de categorias mentais que definem o curso de suas ações.

A Arquitetura BDI foi inspirada em um modelo de comportamento humano e foi originalmente proposta por Bratman (1987) como uma teoria filosófica do raciocínio prático (apud DE NUNES, 2007). Segundo De Nunes (2007), a suposição básica da arquitetura BDI é derivada de um processo chamado raciocínio prático, que é constituído de duas etapas. Na primeira etapa, é selecionado um conjunto de desejos que devem ser alcançados de acordo com

a crença atual do agente. Na segunda, é definido como esses desejos serão atingidos por meio dos meios disponibilizados pelo agente.

As Crenças (*Belief*) são informações que o agente tem sobre o mundo. Essas informações podem estar desatualizadas ou imprecisas. Os Desejos (*Desire*) são todas as possibilidades que o agente gostaria de realizar. Ter um desejo, no entanto, não implica que um agente atue sobre ele. O desejo é apenas um potencial influenciador das ações do agente. As Intenções (*Intention*) são os estados em que o agente decidiu trabalhar e podem ser metas delegadas ao agente ou resultar das opções selecionadas por ele.

A ideia de programar sistemas de computador em termos de noções "metalistas" tais como crença, desejo e intenção é uma componente chave do modelo BDI. A Programação baseada em agentes *Agent-oriented programming* (AOP) oferece uma maneira familiar e não técnica de falar sobre sistemas complexos que não precisa de treinamento formal para entender a conversa metalista[1]. A AOP pode ser considerada como uma espécie de programação "pós-declarativa" que descreve o que um sistema deve fazer e declara precisamente como o processo é até o objetivo ser concluído por meio de um algoritmo detalhado.

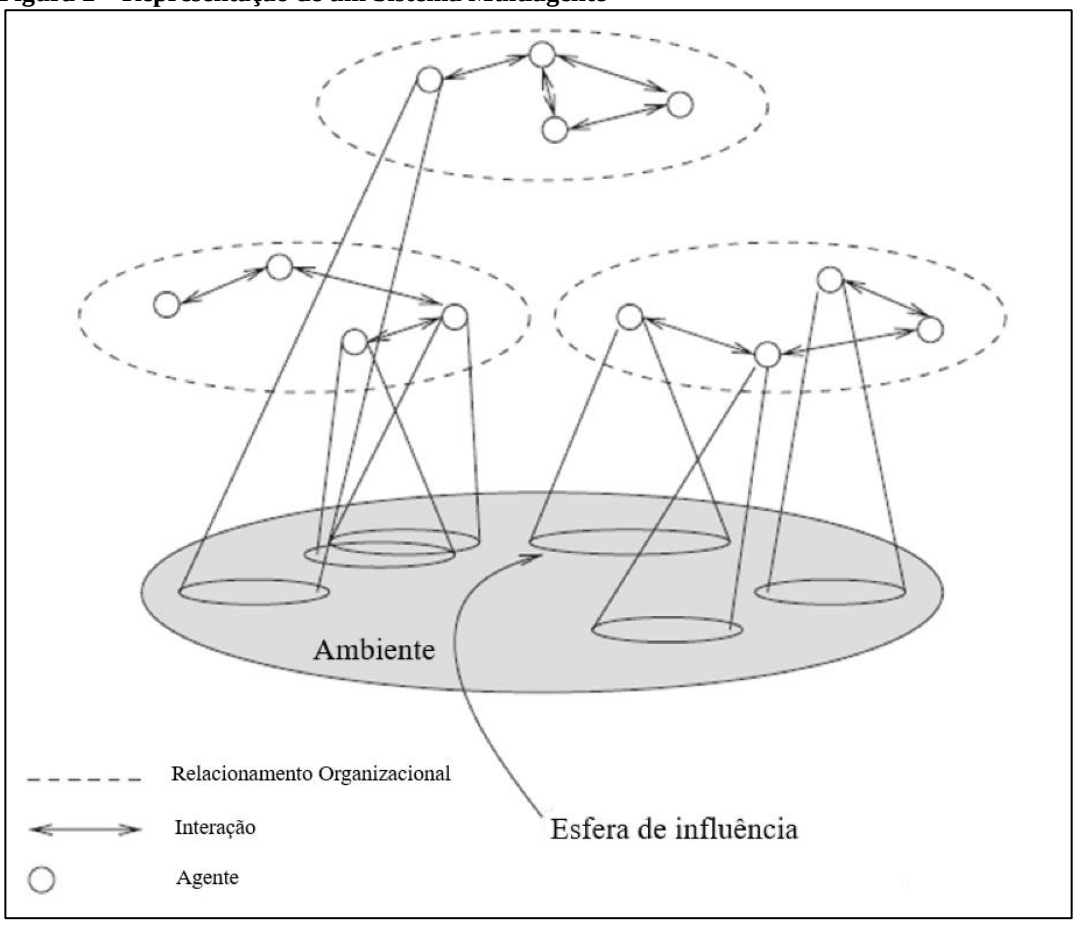
Os agentes podem trabalhar em grupos e recebem o nome de sistemas multiagentes, como será explicado na seção 2.3.

2.3 SISTEMAS MULTIAGENTES

Agentes que trabalham sozinhos em um ambiente são raros; é comum que agentes habitem em um ambiente que contém outros agentes, tornando-se assim um sistema multiagente, como apresentado na Figura 2.

[1] É parte da nossa habilidade linguística cotidiana.

Figura 2 – Representação de um Sistema Multiagente



Fonte: Adaptado de (Bordini, 2007)

A Figura 2 apresenta uma visão geral de um SMA. É possível observar o ambiente compartilhado que os agentes ocupam. Cada um possui uma esfera de influência representada pelas elipses pontilhadas. Nesta representação, os agentes possuem a capacidade de controlar em algum nível o ambiente.

Existe a possibilidade de vários agentes poderem manipular suas esferas de influência e o ambiente. Desse modo, alcançar um resultado no ambiente torna-se complicado devido à interação com outros agentes e à possibilidade de não se conhecer todos os agentes envolvidos.

2.4 LINGUAGENS E FERRAMENTAS PARA AGENTES

Nesta seção são apresentadas algumas ferramentas para o desenvolvimento de modelagem de agentes. O Quadro 1 apresenta alguns *frameworks* para desenvolvimento de agentes.

Quadro 1 - Frameworks para agentes

Framework	Implementada	Licença	Paradigma	Documentação
Jade	Java	LGPL	Reativo	Sim
Jason	Java	LGPL	Cognitivo	Sim
Jadex	Java	GPL V3.0	Cognitivo	Sim
3APL	Java e Haskell	-	Cognitivo	Sim
Jack	Java	Proprietário	Cognitivo	Sim
Jatlit	Java	-	Reativo	Não
Swarm	Objective-C / Java	GLP V3.0	Reativo	Não
Jafmas	Java	-	-	Não
SesAm	Lisp	LGPL	-	Não

Fonte: Autoria própria

Os *frameworks* Jason, Jade, Jadex foram os principais objetos de estudo inicial neste trabalho. Esses são compatíveis com a proposta do trabalho, pois utilizam o conceito que como BDI.

Além dos *frameworks* citados anteriormente, existem outras bases importantes como o Agentspeak (L) (BORDINI; VIEIRA, 2003) e FIPA (FIPA, 2018), embora não sejam *frameworks* de desenvolvimento, são relevantes para comunicação e estruturação dos agentes por serem sua base de desenvolvimento.

2.4.1 AgentSpeak(L)

A linguagem AgentSpeak(L) foi projetada para a programação de agentes BDI na forma de sistemas de planejamentos reativos (*planning reactive system*) (BORDINI; VIEIRA, 2003). Um agente modelado na linguagem AgentSpeak(L) possui um conjunto de crenças, desejos, regras de planos, intenções e eventos que constituem seu estado mental (ZAMBERLAM; GIRAFFA, 2001).

Segundo Zamberlam e Giraffa (2001), o comportamento do agente ao ambiente se dá por comando internos que não são representados explicitamente como fórmulas modais. O estado corrente do agente pode ser observado como sua crença atual. Os objetivos que o agente

quer agir são vistos como desejos, e a adoção de um objetivo a ser alcançado é definido como uma intenção.

Segundo Bordini e Vieira (2003) a AgentSpeak(L) distingui dois tipos de objetivos: Objetivos de testes e de realização, estes são predicados com operadores fixos “?” e “!”, respectivamente. Um objetivo de realização indica que o agente quer alcançar um estado no ambiente em que o predicado associado é verdadeiro e um objetivo de teste retorna uma unificação do predicado de teste com a crença do agente. Também existem os eventos ativadores, os quais estão associados à adição e à remoção de crenças e objetivos e são representados pelos operadores “+” e “-”. A Figura 3 ilustra um exemplo de plano em agentspeak(L).

Figura 3 – Exemplo de Planos AgentSpeak(L)

```
+concerto(A, V) : preferencia(A)
        ← !reservar_ingresso(A,V).
+!reservar_ingresso(A,V) ← ¬ocupado(telefone)
        ← ligar(V)
        ...;
        !escolher_acentos(A,V).
```

Fonte: Adaptado de Bordini e Vieira (2003)

A Figura 3 especifica um plano, adaptado de Bordini e Vieira (2003), que ao ser anunciado um concerto realizado pelo artista *A* no Local *V*, deve ser adicionado uma crença *concerto(A, V)* ao agente como uma consequência da percepção do ambiente. Caso o agente tenha preferência pelo artista *A*, ou seja, gostar(*A*). O agente tem como objetivo na linha 2 de reservar o ingresso para o concerto, sendo este um objetivo de realização. Na linha 3, o agente adota o objetivo de reservar ingresso. Se a linha telefônica não estiver ocupada, executa a ação de *ligar(V)* na linha 4, ou seja, liga para o local do concerto utilizando um protocolo de reserva indicado por “...”. Na linha 5 é finalizada a ligação com um subplano para escolher os acentos *!escolherAcento(A,V)*.

2.4.2 FIPA

O FIPA foi originalmente formado como uma organização suíça em 1996 para produzir especificações de padrões de software para agentes heterogêneos e interativos e

sistemas baseados em agentes. Essas especificações vão desde especificações para iteração entre agentes, ciclo de vida, nomenclatura, serviços de troca de mensagens entre outros (FIPA, 2018).

Nas especificações FIPA, um agente é um processo computacional que implementa funcionalidades de comunicação autônoma, que se comunicam por meio de uma linguagem de comunicação definida por eles, devem suportar pelo menos uma notação de identidade e são fundamentados em afiliação, ou seja, possuir outro agente originador ou pertencer ao usuário humano.

2.4.3 JADE

Java Agent Development Framework (JADE) é uma das plataformas de agente mais populares que estão disponíveis para a comunidade de código aberto. O JADE é compatível com as especificações FIPA. Uma plataforma de agente JADE pode ser distribuída em vários equipamentos que possuam a máquina virtual Java, enquanto várias plataformas podem interoperar por meio de padrões do FIPA (BELLIFEMINE *et al*., 2005).

Os agentes do JADE usam um modelo de execução especializado baseado no planejamento não preemptivo de *plug-ins* JAVA dinamicamente carregáveis, chamados comportamentos.

O modelo de execução de agentes combinado com a interface de programação intuitiva do JADE permite que o programador desenvolva com relativa facilidade agentes de software capazes de ter comportamentos flexíveis reativos e/ou proativos.

A ferramenta JADE permite a abstração de agentes, fornece um modelo simples de execução e composição de tarefas, baseia-se no paradigma de passagens de mensagens assíncronas para realizar comunicação entre agentes e possuí outros recursos que possibilitam o desenvolvimento de sistemas distribuídos (BELLIFEMINE *et al*., 2005).

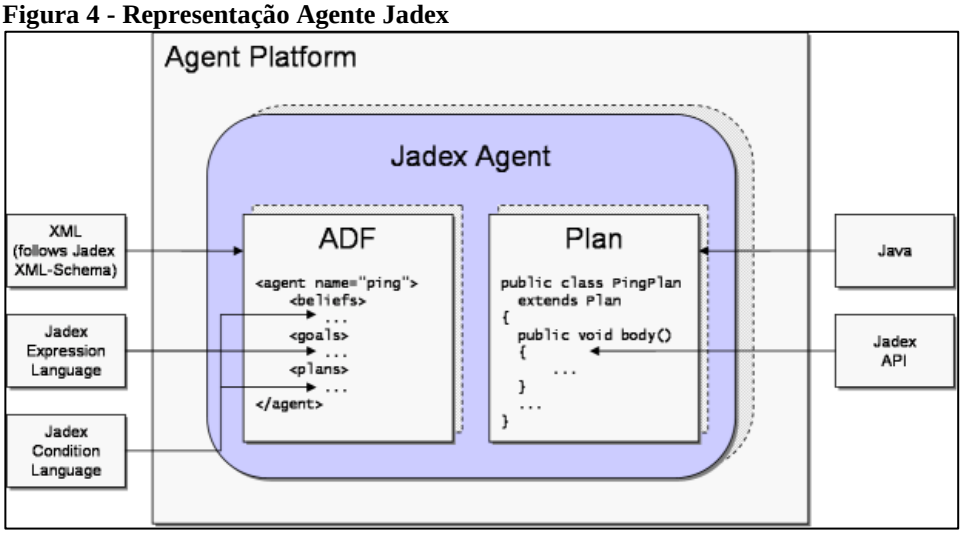
2.4.4 JADEX

Segundo Mohamad, Deris e Ammar (2006), os agentes Jadex seguem o modelo BDI com crenças, desejos e intenções. Entretanto, os agentes Jadex empregam uma representação de crenças orientada a objetos. Além disso, as crenças têm um papel ativo, ou seja, sua atualização pode desencadear a geração de eventos ou adoção e descarte de metas.

A linguagem Jadex combina a especificação declarativa de um agente que contém seu conjunto de crenças, metas e planos usando um arquivo de definição de agente (ADF) e a especificação procedural dos corpos de plano usando a linguagem de programação Java. O corpo do plano acessa os componentes internos de um agente por meio de uma API especializada.

A Figura 4 apresenta a estrutura de um agente JADEX, em que é possível visualizar as suas duas partes básicas. No lado esquerdo, há um arquivo *xml* com as definições do agente (ADF), e no direito há classes Java estendidas que especializam o jadex para especificar planos crenças e objetivos (MOHAMAD; DERIS; AMMAR, 2006).

Figura 4 - Representação Agente Jadex



Fonte: Mohamad, Deris e Ammar (2006)

2.4.5 JASON

Jason é a primeira implementação do *AgentSpeak(L)* usando a linguagem de programação Java e que pertence ao paradigma de agentes híbridos. A sintaxe de Jason exibe algumas semelhanças com o Prolog e a semântica é baseada no *AgentSpeak(L)* (JASON, 2018).

O interpretador Jason pode ser usado para construir agentes fornecendo uma API Java para integração com um modelo de ambiente que é desenvolvido com Java e que pode ser integrado a algumas estruturas de agentes existentes incluindo JADE, *AgentScape* e *Agent Factory*.

2.4.6 Desenvolvimento com Agentspeak(L) e Jason: Um Pequeno Exemplo

A Figura 5 apresenta uma comparação entre a execução de um programa "HelloWorld" na linguagem Java e *Agentspeak(L)* implementada em Jason.

No Quadro 2 a representação a direita constitui a definição de um único agente. Esta é salva em um único arquivo e foi denominado de *hello.asl*; a extensão. *asl* é usada para todos os programas AgentSpeak*(L)* que consiste em duas partes (PADGHAM; WINIKOFF, 2005):

- Crenças iniciais do agente (e possivelmente metas iniciais);
- Planos do agente.

Quadro 2 - Comparação de uma Classe .java com Agente .asl

.JAVA	AGENTE (.ASL)

public class HelloWorld
{
public static void main(String args[])
{System.out.println("Hello World!");}
}
	!inicia.
	
	+inicia <- .print("Hello World!").

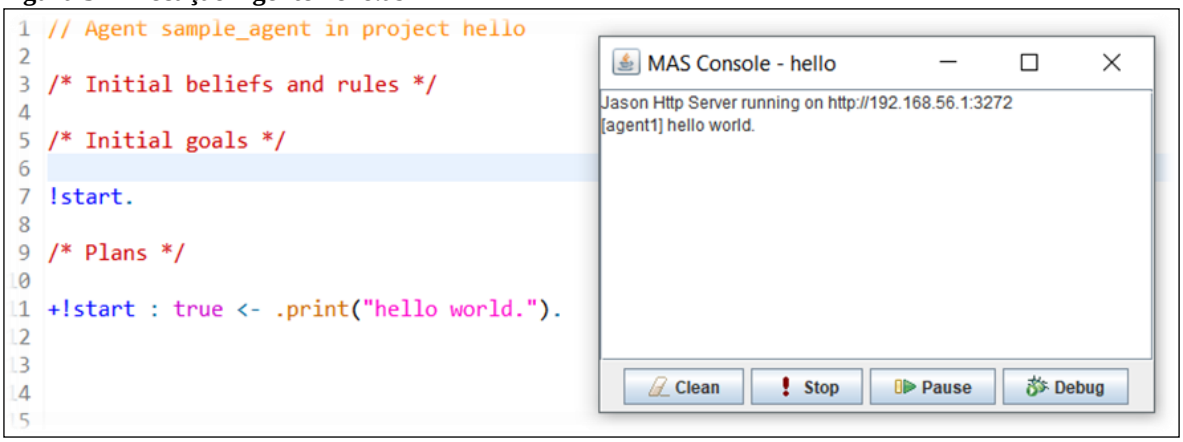
Fonte: Autoria própria

A primeira linha do arquivo utilizando agentes define uma crença inicial para o agente. Embora haja apenas uma crença inicial, poderia ter sido utilizado uma lista de crenças. Na *AgentSpeak(L)* não existem variáveis como em linguagens de programação como Java ou C, a construção do agente é especificada com crenças, metas e planos.

Precisa-se de "crenças" porque eles representam as informações que o agente conseguiu obter no momento sobre o seu ambiente. O ponto final "." é um separador sintático, como um ponto e vírgula em Java ou C. Portanto, quando o agente começa a executar, ele terá a crença única; intuitivamente, acredita que começou a funcionar.

O símbolo “+” neste contexto significa “quando você adquire a crença.” e, assim, em geral, a condição de ativação é “quando você adquire a crença ‘inicia’”. Executando o exemplo em JASON, o resultado é exibido na Figura 5.

Figura 5 - Execução Agente hello.asl



Fonte: Autoria própria

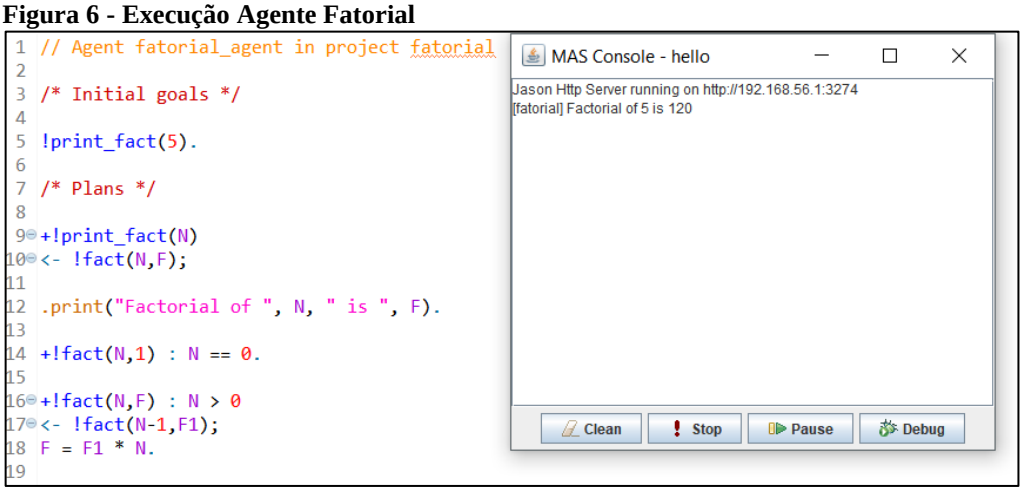
Outro exemplo é a utilização de recursões, como apresentado no Quadro 3. A execução deste exemplo é ilustrada na Figura 6.

Quadro 3 - Exemplo Fatorial.asl

Fatorial.asl
!print_fact(5). +!print_fact(N) <- !fact(N,F); .print("Factorial of ", N, " is ", F). +!fact(N,1) : N == 0. +!fact(N,F) : N > 0 <- !fact(N-1,F1); F = F1 * N.

Fonte: Autoria própria

Figura 6 - Execução Agente Fatorial



Fonte: Autoria própria

No exemplo do Quadro 3, o agente fatorial tem o objetivo inicial e executar o fatorial de 5. Em seguida, são executadas chamadas recursivas até que *N* seja igual a 0. Por fim, é apresentado o resultado no console.

2.5 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Este capítulo apresentou alguns dos trabalhos técnicos, *frameworks* e assuntos relevantes sobre agentes que contribuem para a realização deste trabalho.

Os agentes constituem parte do núcleo desta pesquisa, sendo essencial seu entendimento e conhecimento para posterior aplicação no processo de modelagem que será utilizado como exemplo para modelagem no próximo capítulo.

3 MODELAGEM DE SISTEMAS BASEADOS EM AGENTES

Este capítulo apresenta os conceitos em engenharia de software baseada em agentes e ferramentas para sua modelagem. A seção 3.1 descreve sobre engenharia de software baseada em agentes. A seção 3.2 relata como é realizado a modelagem de agentes e apresenta metodologias usadas para a construção do modelo. Por fim, última seção descreve as considerações deste capítulo.

3.1 ENGENHARIA DE SOFTWARE BASEADA EM AGENTES

Segundo Girardi (2004) a “engenharia de software baseada em agentes fornece soluções para abordar a complexidade de sistemas não predizíveis que podem mudar rapidamente, em que devem ser capazes de tomar decisões afim de atingir seus objetivos”.

Segundo Juchem e Bastos (2001), sistemas complexos são compostos por hierarquias, que tipicamente contém um grande número de interações com subsistemas e sistemas complexos que podem ser gerenciados por três ferramentas: decomposição, abstração e organização. Na decomposição, o problema é dividido em parte menores, potencializando a sua resolução. A abstração consiste em considerar detalhes e as propriedades relevantes com o objetivo de gerar um modelo simplificado da realidade e a organização trata de identificar e gerenciar os interacionamentos entre os componentes do problema.

Segundos Jennings (1999) agentes possuem algumas características, como serem autônomos, capazes de exibir comportamento flexível para a resolução de problemas e realizam um papel específico (apud JUCHEN, 2001). Logo, é perceptível que um único agente é insuficiente para resolver problemas complexos. Para isso é necessário envolver sistemas multiagente, no qual os agentes interagem com outros para atingir seus objetivos individuais.

Os sistemas multiagente são considerados uma excelente metáfora para caracterizar sistemas complexos e o conceito de agente como abstração de software é útil para a compreensão, engenharia e uso desse tipo de sistemas (GIRARDI, 2004).

Um objeto é equivalente a um agente: ambos encapsulam estados e comportamentos e comunicam-se por meio da passagem de mensagens. Contudo, os agentes são objetos racionais e realizam tomadas de decisões. Exige-se que um agente tenha um comportamento reativo e proativo e possua a capacidade de intercalar esses tipos de comportamento conforme suas necessidades, algo que a orientação a objetos não pode proporcionar (DE LIMA *et al*., 2009).

3.2 MODELAGEM DE AGENTES

Existem diversas definições relacionadas ao termo agente, assim como para a modelagem baseada em agentes. A modelagem pode receber nomes tais como: sistemas baseados em agentes (*agent-based systems*) e modelagem baseada em indivíduos (*individual-based modeling*) (DE LIMA, 2009).

Em algumas situações a modelagem baseada em agentes pode oferecer vantagens sobre abordagens tradicionais, portanto, é benéfico pensar em termos de agentes quando é possível ter uma representação natural de seus comportamento e decisões sendo que estes podem ser definidos discretamente. É importante que os agentes se adaptem a situação quando tenham relacionamentos dinâmicos com outros agentes e que suas relações possam ser criadas ou desfeitas.

Segundo Ferreira (2008) as técnicas de engenharia de software convencional não convêm para o projeto de sistemas baseado em agentes devido a suas peculiaridades.

Para o desenvolvimento de sistema baseado em agentes, existem metodologias específicas tais como Tropos, Gaia e *Prometheus*. A metodologia Tropos fornece suporte as atividades de análise e projeto no desenvolvimento do sistema e está dividida em cinco fases: Requisitos Iniciais, Requisitos Finais, Projeto Arquitetural, Projeto Detalhado e Implementação (BERNY *et al.*, 2008).

Segundo Ferreira (2008), a metodologia Gaia é usada na fase de análise e projeto de sistemas multiagentes e possui uma linguagem própria para sua modelagem contendo as fases de Especificação e de *Design*. Nesta metodologia os agentes devem ser estáticos, ou seja, a quantidade de agentes deve ser constante (BERNY *et al.*, 2005).

A Metodologia Prometheus para o desenvolvimento de SMAs abrange desde a modelagem até a implementação e conta com uma ferramenta que permite o acompanhamento da metodologia por meio de diagrama. Esta metodologia e sua ferramenta serão mais detalhadas nas próximas seções.

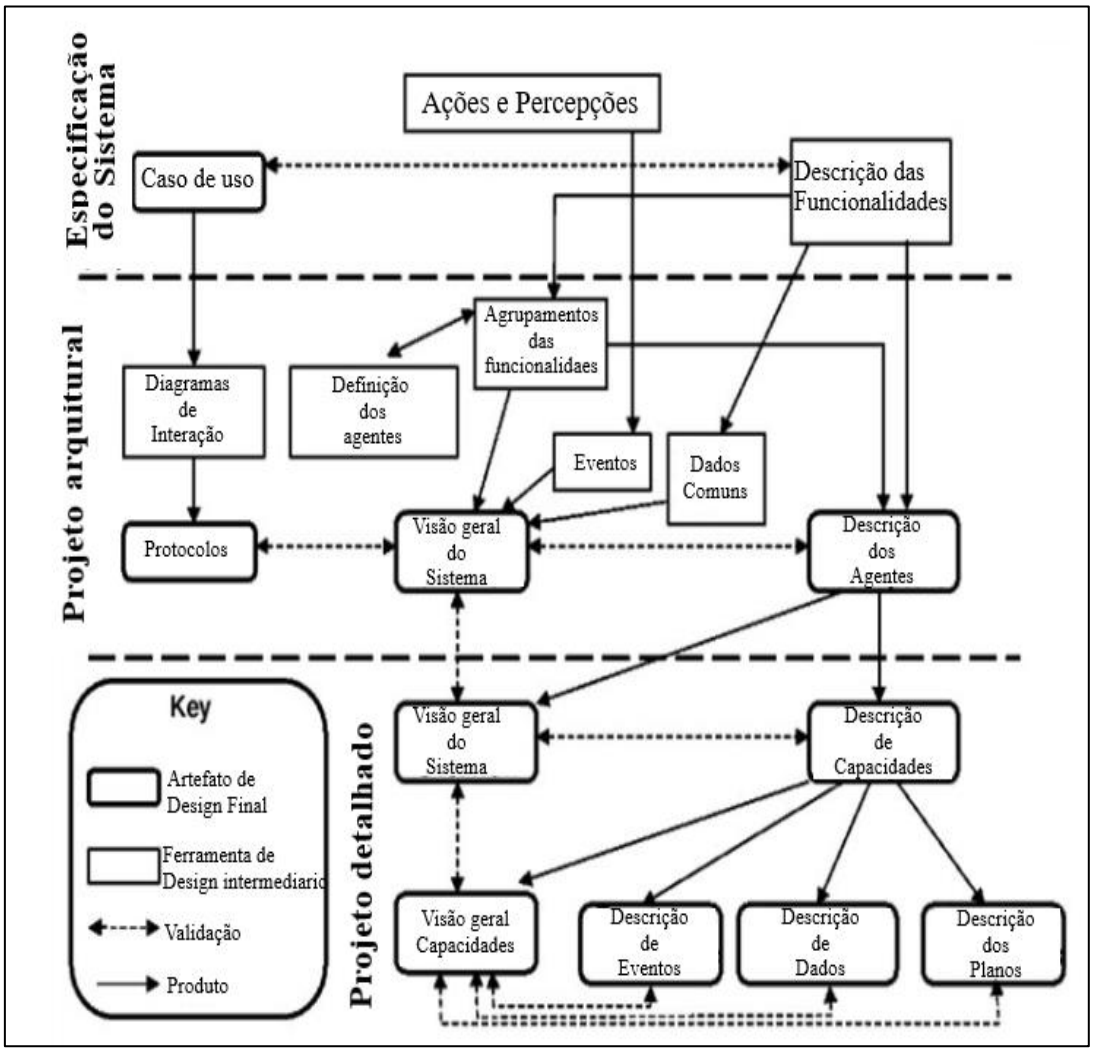
3.2.1 Metodologia Prometheus

Esta metodologia é composta por três etapas em que os componentes produzidos são utilizados tanto na geração do esqueleto do código como para a realização de testes de (PADGHAM; WINIKOFF, 2005).

A primeira etapa corresponde à especificação do sistema que compreende duas atividades: determinar o ambiente do sistema, seus objetivos e funcionalidades. A segunda etapa utiliza as especificações geradas a fim de determinar quais agentes existirão no sistema e como devem interagir. Esta fase é composta por três atividades, sendo elas, definição dos agentes, definição da estrutura do sistema e iteração entre os agentes.

A terceira etapa, chamada de *Design* detalhado, é definida as descrições eventos, planos, capacidades e dados. A Figura 7 apresenta as três etapas da metodologia.

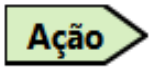
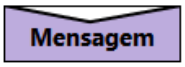
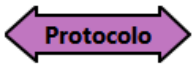
Figura 7 - Etapas Metodologia Prometheus



Fonte: Adaptado de Padgham; Winikoff (2005)

Existem duas ferramentas que utilizam o Prometheus: o *Jack Development Environment* (JDE) e Prometheus. A *Jack* é um software comercial que inclui uma ferramenta de modelagem para construção de diagramas, gerando códigos na linguagem de programação *Jack.* A *Prometheus Design Tool* (PDT) é usada como uma ferramenta *standalone*, *plugin* para o eclipse, criando diagramas e a descrição do projeto baseado nas entidades descritas (PROMETHEUS, 20017). O Quadro 4 apresenta as entidades para modelagem na ferramenta PDT e uma breve explicação do que representa cada elemento.

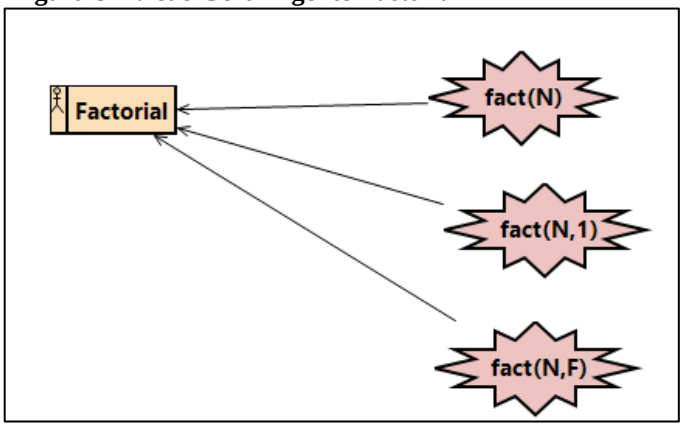
Quadro 4 - Entidades para modelagem *Prometheus Design Tool*

Elemento	Descrição
	O Agente
	Representação de uma ação que indica o que o agente faz que afeta o ambiente.
	Uma percepção é a entrada proveniente do ambiente para o agente.
	Um armazenamento de dados é usado para guardar as crenças de um agente.
	Uma mensagem que vai de um agente para outro.
	Um protocolo que é uma especificação de sequências de interação de agente permitidas dentro de uma determinada conversa.

Fonte: Autoria Própria

A Figura 8 apresentada o diagrama de visão geral segundo a metodologia Prometheus, utilizando a ferramenta PDT para um agente fatorial que foi apresentado no Quadro 3.

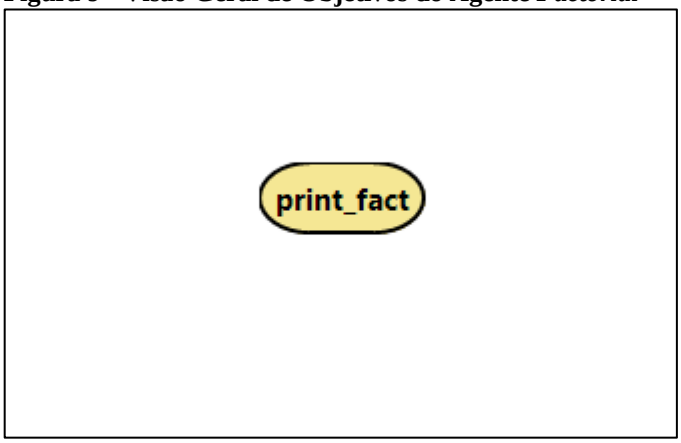
Figura 8 - Visão Geral Agente *Factorial*



Fonte: Autoria própria

A Figura 9 mostra a representação da visão geral específica do agente fatorial em que é possível visualizar que sua meta é imprimir o fatorial (print_fact).

Figura 9 - Visão Geral de Objetivos do Agente *Factorial*



Fonte: Autoria própria

No exemplo da figura 9 tem-se apenas uma meta (poderia ser uma lista de metas) a ser alcançada, no qual se pode adicionar como objetivo antecessor ao “print_fact” a verificação de entrada para garantir que seja um número natural. Em caso falso, o agente não pode realizar este objetivo.

3.3 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Este capítulo apresentou o conceito da engenheira de software baseada em agentes e as metodologias de desenvolvimentos mais utilizada para modelagem de agentes BDI.

No desenvolvimento deste trabalho, a metodologia *Prometheus* foi adotada para modelagem dos agentes, pois os modelos encontrados na literatura, documentação do Jason utilizam dos modelos criados com a ferramenta *Prometheus Design Tool*, por serem mais didáticas e completas.

4 REFATORAÇÃO DE SOFTWARE

Este capítulo apresenta uma visão geral sobre refatoração de software. A seção 4.1 relata a importância e vantagens da refatoração. A seção 4.2 descreve as técnicas de refatoração publicadas na literatura. A seção 4.3 aborda conceitos sobre refatoração baseada em padrões. A seção 4.4 apresenta métodos para detecção e inserção de padrões de projeto. Por fim, a última seção relata as considerações finais do capítulo.

4.1 A IMPORTÂNCIA DA REFATORAÇÃO

Para Fowler *et al*. (1999) refatorar é o processo que altera um sistema de software de maneira que não modifique seu comportamento externo (resultado), mas melhora sua estrutura interna o que torna o código-fonte mais fácil de ser compreendido e menos custoso para modificações.

O processo de refatoração pode ser realizado de forma manual ou automatizado. Refatorações manuais são frequentemente realizadas, porém podem introduzir mais erros ao código-fonte. Isso ocorre porque desenvolvedores não estão tecnicamente preparados o suficiente para realizarem as refatorações (GE; DUBOSE; MURPHY-HILL, 2012).

O processo de refatoração pode ser realizado por meio de técnicas ou baseadas em padrões de projeto, relatadas nas seções 4.2, 4.3 e 4.4.

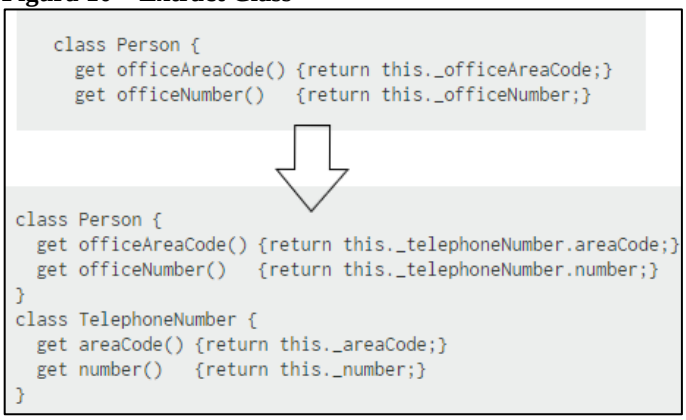
4.2 TÉCNICAS DE REFATORAÇÃO

Fowler *et al*. (1999) apresenta um conjunto de técnicas de refatoração de software. Ao todo, em seu catálogo são apresentadas 91 técnicas de refatoração divididas em 17 categorias. Algumas categorias definidas são: *Composing Methods, Moving Features Between Objects e Composing Methods*.

Um exemplo de uma técnica de refatoração presente na categoria *Composing Methods* e a *Extract Class* (Extrair Classe), que consiste em retirar um código existente dentro de uma classe e criar uma nova classe com o código extraído (FOWLER *et al*.; 1999). Este método de refatoração ajuda a manter a adesão ao Princípio de Responsabilidade Única. O código-fonte gerado após a refatoração será mais compreensível para reutilização futura e

classes com responsabilidade única são mais confiáveis e tolerantes a mudanças. Na Figura 10 é apresentada a técnica *Extract Class*, em que as informações sobre telefone (*TelephoneNumber*) são delegadas a uma nova classe, deixando a classe Pessoa (*Person*) apenas com ações relacionadas a ela.

Figura 10 – Extract Class



Fonte: Adaptado Fowler (2006)

Distribuir métodos e atributos entre um conjunto de classes é uma tarefa difícil. É provável que a modelagem inicialmente feita para este conjunto seja alterada com o passar do tempo, assim, uma prática comum é ajustar esses comportamentos e características. Esta é a responsabilidade do grupo *Moving Features Between Objects.* Se uma classe *C* precisa de novas funcionalidades e esta não pode ser alterada, uma técnica que pode ser utilizada é a de *Introduce Local Extension* (FOWLER, 2006).

O *Composing Methods* avalia como um método é construído, especialmente quando se tratam de métodos grandes os quais contém uma lógica complexa. O *Extract Method* é um exemplo deste grupo e tem a finalidade de dividir o método em outros menores (FOWLER, 2006).

4.3 REFATORAÇÃO BASEADA EM PADRÕES DE PROJETOS

De acordo com Gamma *et al*. (2000), os padrões de projetos são definidos com base em entidades primitivas como objetos e classe. Por isso, um padrão deve abstrair, nomear e identificar pontos importantes de uma estrutura de projeto para torná-la útil na criação de um projeto reutilizável.

Padrões de projetos afetam a maneira como o software é projetado, criando um vocabulário comum e eleva o nível de projeto para melhorar a comunicação e a sua documentação.

Os padrões de projeto variam de acordo com sua granularidade e seu nível de abstração. A classificação deles é feita baseada em dois critérios: finidade e escopo. A finalidade representa o que o padrão faz, podendo ser criacional, estrutural e comportamental. Em relação ao escopo se refere aonde os padrões podem ser aplicados, como por exemplo, classes ou objetos.

A Figura 11 apresenta a classificação dos padrões de projetos de acordo com estes dois critérios.

Figura 11 - Classificação dos Padrões de Projetos

		Finalidade		
		De criação	**Estrutural**	**Comportamental**
Escopo	**Classe**	*Factory Method*	*Adapter*	*Interpreter* *Template Method*
	Objeto	*Abstract Factory* *Builder* *Prototype* *Singleton*	*Adapter* *Bridge* *Composite* *Decorator* *Façade* *Flyweight* *Proxy*	*Chain of Responsability* *Command* *Iterator* *Mediator* *Memento* *Observer* *State* *Strategy* *Visitor*

Fonte: Gamma *et al*. (2000)

Os padrões de criação se preocupam com o processo de criação de objetos. Os padrões estruturais lidam com a composição de classes ou de objetos. Os padrões comportamentais referem-se como as classes e objetos interagem e distribuem suas responsabilidades (GAMMA *et al*., 2000).

Em relação ao critério chamado de escopo, é especificado se o padrão se aplica primariamente as classes ou a objetos. Os padrões de classes tratam dos relacionamentos entre classe e subclasse. E os padrões de objetos com os relacionamentos entre objetos podem ser alterados em tempo de execução (GAMMA *et al*., 2000).

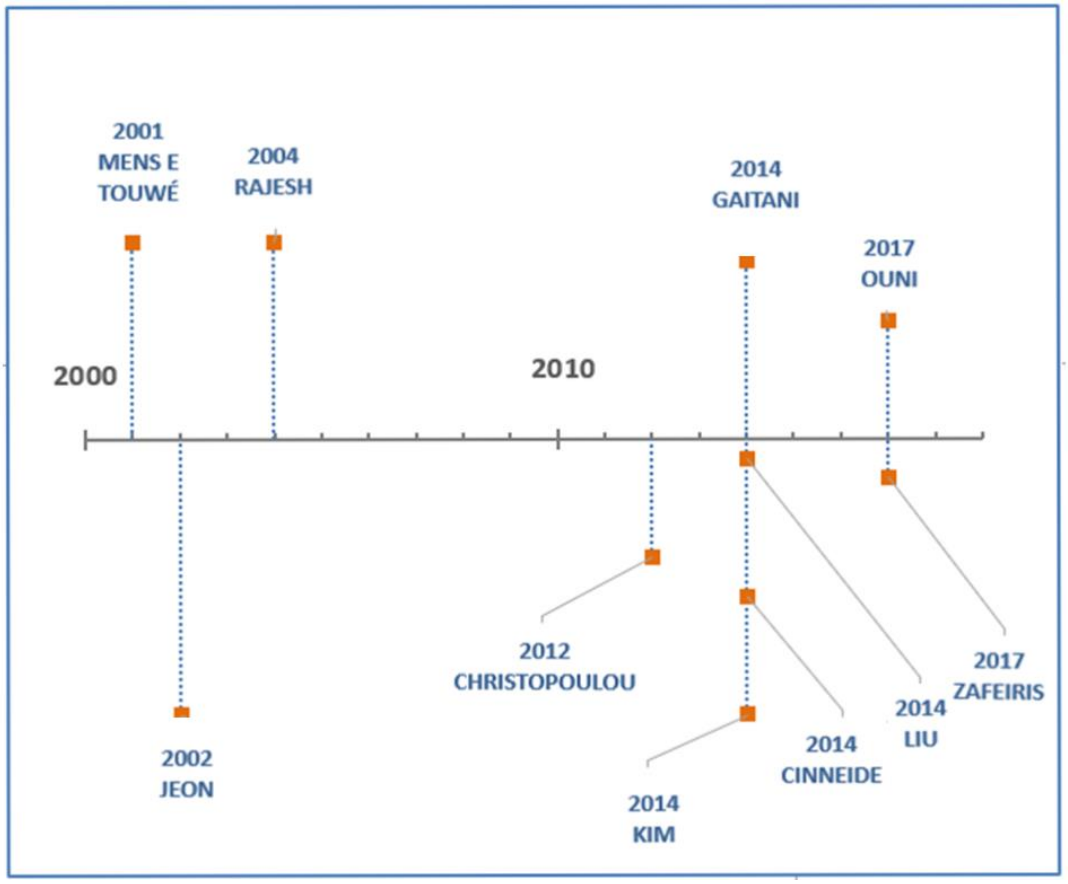
4.4 MÉTODOS DE DETECÇÃO DE PONTOS DE INSERÇÃO

Nesta seção são apresentados os métodos de detecção de pontos de inserção. Os métodos de detecção têm a finalidade de analisar o código-fonte e de aplicar refatorações a fim de se adequar a algum padrão de projeto. Um método de detecção de pontos de inserção tem como objetivo utilizar algum meio para analisar um código-fonte e detectar possíveis trechos a serem refatorados.

As descrições dos métodos apresentadas nesta seção usaram como referência os resultados obtidos pelo mapeamento sistemático realizado por Belluzo (2018). Durante sua revisão foram identificados 13 trabalhos que abordam o assunto de detecção de pontos de inserção de padrões de projetos e metapadrões. Em relação a esses trabalhos, observou-se que nenhum deles aborda o uso de metapadrões.

A figura 12 apresenta uma linha do tempo com a data de publicação de cada método, sendo possível notar que existe um crescimento no interesse na área a partir do ano de 2010.

Figura 12 - Linha do tempo de métodos de detecção de padrões de projeto



Fonte: Autoria Própria

A Tabela 1 apresenta de forma numérica quantos padrões (comportamentais, criacionais, estruturais) cada método contempla. Nesta tabela é possível observar que os trabalhos de Cinnèide (2000, 2001) abrangem 23 dos 24 padrões de projetos definidos por Gamma *et al*. (2000), sendo este um dos autores mais relevantes na área.

Tabela 1 - Quantidade de padrões por classificações contemplados

Método	**Comportamentais**	**Criacionais**	**Estruturais**
MENS E TOUWÉ (2001)	2	1	1
JEON, LEE E BAE (2002)	-	1	-
RAJESH E JANAKIM (2004)	-	-	1
CHRISTOPOULOU (2012)	1	-	-
KIM (2014)	3	1	-
LIU *et al*. (2014)	1	1	-
CINNEIDE (2000, 2001)	11	5	7
GAITANI *et al*. (2015)	1	-	-
OUNI *et al*. (2017)	2	2	-
ZAFEIRIS *et al*. (2017)	-	-	1

Fonte: Autoria Própria

O Quadro 5 apresenta os métodos que possuem ferramenta para detecção e inserção de padrões de projeto em código-fonte e se utilizam como base outros métodos.

Quadro 5 - Bases, Ferramentas e data de publicação

Método	**Ferramentas**	**Base**
MENS E TOUWÉ (2001)	Protótipo	-
JEON, LEE E BAE (2002)	Protótipo	CINNEIDE
RAJESH E JANAKIM (2004)	JIAD	-
CHRISTOPOULOU (2012)	*Jdeodorant*	-
CINNEIDE (2000, 2001)	Design Patern Tool	CINNEIDE *
GAITANI *et al*. (2015)	*Plugin* do Eclipse	-
KIM (2014)	Plugin do Eclipse	-
LIU (2014)	-	-
OUNI *et al*. (2017)	MORE	CINNEIDE
ZAFEIRIS *et al*. (2017)	JDeodorant	-

Fonte: Autoria própria

Cerca de 33% dos 10 trabalhos utilizam os trabalhos publicados por Cinnéide e Nixon (1999, 2001) como base de suas pesquisas.

Apesar de existirem ferramentas para detectar e inserir padrões de projeto em código-fonte, o desenvolvedor deve usá-las de forma individual, ou seja, não se tem um ambiente que contemple vários métodos em um único ambiente. Por isto, o uso de agentes pode ajudar na criação deste ambiente integrado e que foi o foco deste trabalho.

4.5 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Este capítulo apresentou conteúdos sobre a refatoração de software, sua importância e trabalhos que criaram métodos para detecção e inserção padrões de projetos a fim de fornecer uma organização e reestruturação de um sistema o tornando mais manutenível.

A refatoração pode ser aplicada de uma forma independente, identificando onde é necessário ser aplicada e qual técnica deve ser usada. Para isso, o desenvolvedor deve ter conhecimento e experiência das técnicas de refatoração e padrões de projeto.

O capítulo seguinte apresenta um modelo de um agente que incorpora métodos de detecção e inserção de padrões de projetos, modelado utilizando a metodologia Prometheus e utilizando da ferramenta *Prometheus Design Tool*, a fim de mostrar que é possível desenvolver um agente para auxiliar no processo de refatoração.

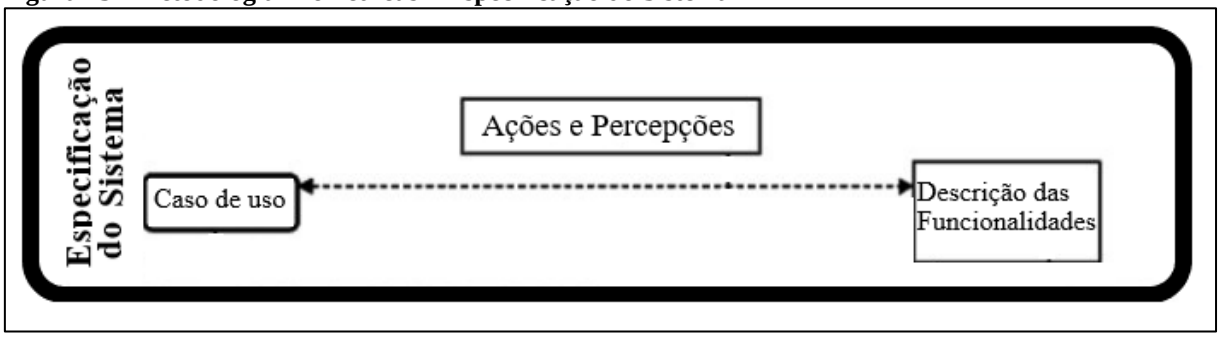
5 MODELAGEM DE AGENTES PARA REFATORAÇAO

Este capítulo apresenta a modelagem de um sistema multiagente para refatoração de software, seguindo as etapas propostas pela metodologia *Prometheus*. O problema exige que um sistema multiagente forneça opções de refatoração de software aos usuários com base na leitura de um código-fonte, identificando pontos de inserção e apresentado soluções de refatoração baseados na literatura. A seção 5.1 relata a aplicação da metodologia *Prometheus* para criar um modelo de agentes voltado a refatoração de software. A seção 5.2 apresenta um estudo de caso para ilustrar o funcionamento dos agentes. Por fim, na última seção são abordadas as considerações finais do capítulo.

5.1 APLICAÇÃO DA METODOLOGIA PROMETHEUS NA MODELAGEM DO AGENTE PROPOSTO

Segundo a metodologia Prometheus, a fase de especificação do sistema constitui de quatro etapas: especificação dos objetivos, desenvolvimento de cenários, definição de funcionalidades e descrição das iterações do sistema. Na Figura 13, é apresentada a etapa de especificação do sistema da metodologia *Prometheus*, conforme explicado na Seção 3.2.1.

Figura 13 - Metodologia Prometheus – Especificação do Sistema



Fonte: Adaptado de Padgham; Winikoff (2005)

Segundo Padgham e Winikoff (2005) é necessário realizar uma breve descrição inicial do sistema, que geralmente contém algumas indicações implícitas sobre seus objetivos. Desse modo, é fornecido um ponto de partida para ser iniciada a primeira etapa. Esta etapa inicia-se por meio do levantamento dos objetivos e demarcação dos pontos de interesse. No Quadro 7 é apresentada a problemática que foi modelada neste trabalho.

Quadro 6 - Problemática

O problema exige que um sistema multiagente forneça opções de refatoração de software aos usuários com base na leitura de um código-fonte, identificando pontos de inserção e apresentado soluções de refatoração baseados na literatura.

Fonte: Autoria própria

Com as informações do Quadro 6, é realizado as demarcações dos pontos de interesse segundo Padgham e Winikoff (2005). As demarcações para o problema proposto são apresentadas no Quadro 7, em que se usou o sublinhado para identificar que tipo é o sistema e o que deve fazer.

Quadro 7 – Demarcação da Problemática

O problema exige que um sistema multiagente forneça opções de refatoração de software aos usuários com base na leitura de um código-fonte, identificando pontos de inserção e apresentado soluções de refatoração baseados na literatura.

Fonte: Autoria própria

A partir das demarcações, no Quadro 8 são apresentadas a lista de objetivos iniciais levantadas.

Quadro 8 - Lista de Objetivos iniciais

- Leitura código-fonte
- Detecção de pontos de inserção
- Apresentação das opções de refatoração
- Refatoração do código
- Análise da refatoração

Fonte: Autoria própria

Durante esta etapa na metodologia, os objetivos são refinados. Para cada objetivo são atribuídos subobjetivos. Os subobjetivos são métodos de como os objetivos podem ser alcançados, reunindo pequenos fragmentos descritíveis em poucas frases que poderão integrar o sistema. O Quadro 9 ilustra o conjunto de objetivos refinados para o problema proposto neste trabalho.

Quadro 9 – Lista dos Sub Objetivos

- Leitura do código-fonte

- Identificar código-fonte
- Identificar arquivos do projeto

- Detecção de pontos de inserção

- Encontrar possíveis pontos de inserção
- Verificar capacidade de refatoração

- Apresentação das opções de refatoração

- Fornece aos usuários possíveis refatorações
- Informações sobre as mudanças que a refatoração selecionada implica

- Refatoração do código

- Providenciar *backup* do arquivo original
- Atualizar código-fonte

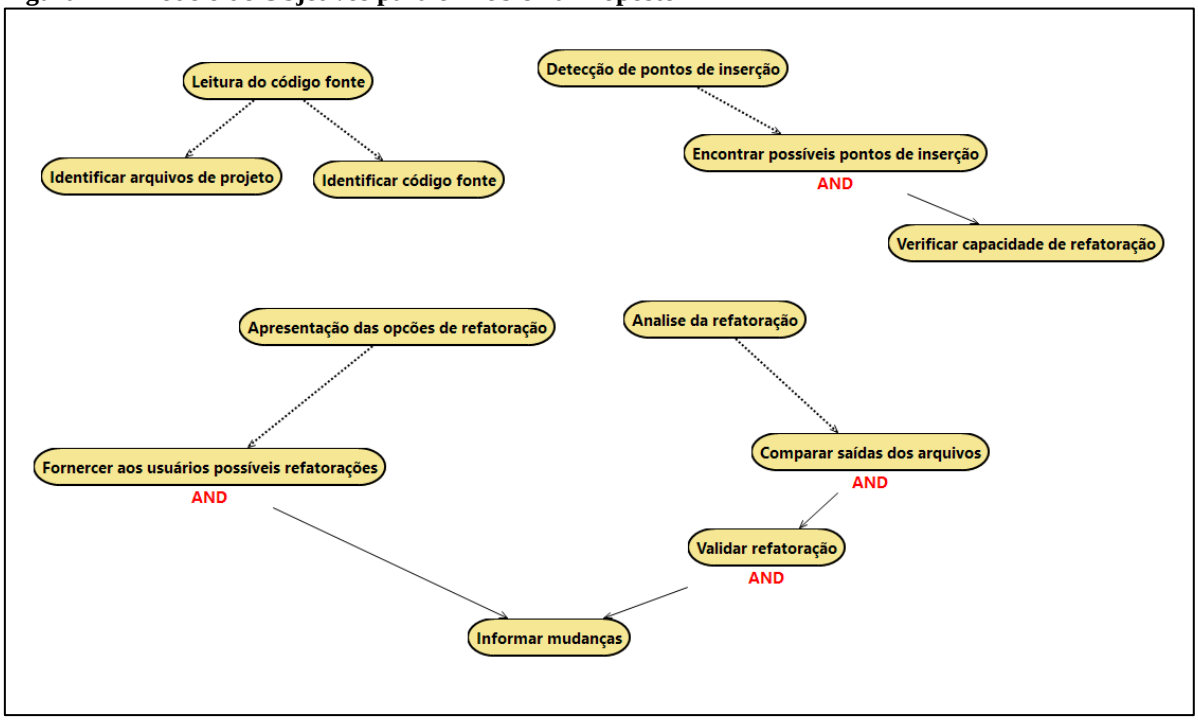
- Análise da refatoração

- Comparar saída do arquivo *backup* com a saída do arquivo atualizado
- Informar a validade da refatoração
- Informar mudanças no código-fonte

Fonte: Autoria própria

A partir dos objetivos identificados e refinados, pode-se representá-los graficamente usando a ferramenta *Prometheus design tool*, conforme exibe a Figura 13. O diagrama é chamado de visão de objetivos (*goal overview*) em que é possível visualizar os objetivos e seus subobjetivos do sistema e que foram apresentados no Quadro 9.

Figura 14 - Modelo de Objetivos para o Problema Proposto



Fonte: Autoria Própria

Depois da etapa de refinamento e diagramação dos objetivos, eles devem ser reorganizados e agrupados em objetivos semelhantes, definindo-se um nome significante para cada agrupamento. Este nome pode estar relacionado a uma das metas originais de nível superior ou pode ser um novo nome significante para o objetivo. Estes agrupamentos são realizados a fim de serem definidas as possíveis funcionalidades do sistema. Os agrupamentos realizados nesta etapa estão apresentados no Quadro 10.

Quadro 10 - Funcionalidades do Sistema

- **Analise de arquivos:**
  - Leitura do código-fonte.
  - Identificar código-fonte.
  - Identificar arquivos do projeto.

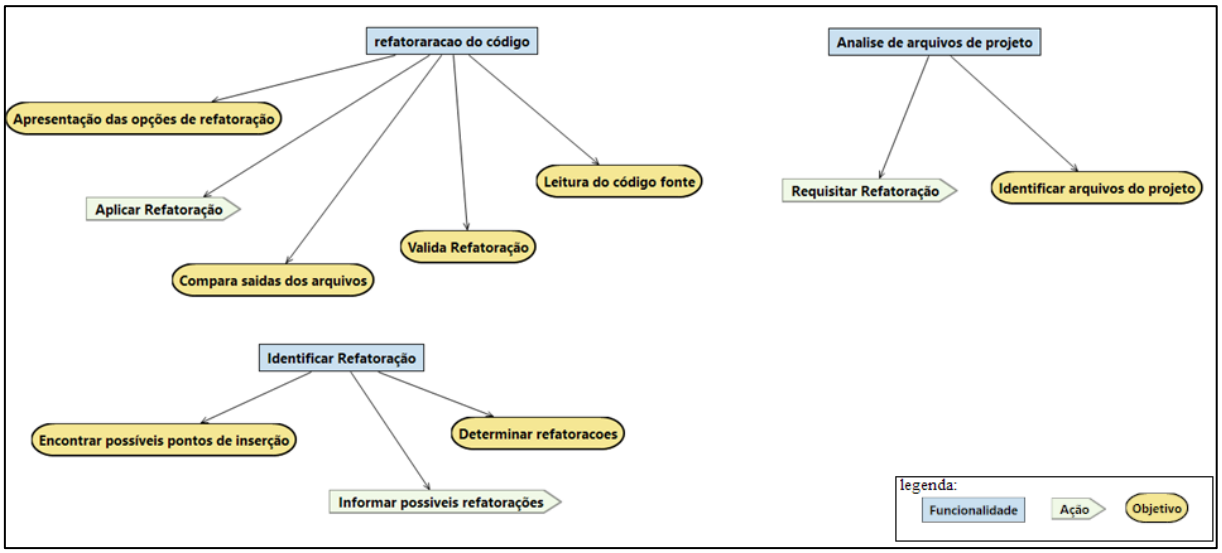
- **Apresentação das refatorações:**
  - Apresentação das opções de refatoração.
  - Fornece aos usuários possíveis refatorações.

- **Refatoração de arquivo:**
  - Refatoração do código.
  - Providenciar *backup* do arquivo original.
  - Atualizar código-fonte.
  - Analisar da refatoração
  - Comparar saída do arquivo *backup* com a saída do arquivo atualizado.
  - Informar a validade da refatoração.
  - Informar mudanças no código-fonte.
  - Informações sobre as mudanças que a refatoração selecionada implica

Fonte: Autoria Própria

A Figura 15 ilustra como os agrupamentos do Quadro foram modelados na ferramenta. Neste modelo definem-se as funcionalidades, objetivos e ações do sistema.

Figura 15 - Funcionalidades, Objetivos e Ações



Fonte: Autoria Própria

Ainda nesta etapa de especificação do sistema, é criada uma lista de cenários para o sistema a fim de encontrar objetivos e ações não encontradas anteriormente. Esta lista é chamada de etapa de refinamento e são criados os cenários, percepções e ações.

O Quadro 11 representa uma lista de cenários possíveis para que o agente atue. Quanto maior a quantidade de cenários possíveis, menor é a possibilidade de falha, ou seja, do agente encontrar uma situação que o mesmo não tenha como lidar.

Quadro 11 - Lista de Cenários

- Novo arquivo
- Código de projeto encontrado
- Arquivo de projeto modificado
- Pedido de refatoração de arquivo
- Informar refatorações
- Validação da refatoração

Fonte: Autoria Própria

Os Quadros 12 e 13 representam a lista de percepções e de ações, respectivamente. Esta lista baseia-se nos possíveis cenários gerados. Esta etapa pode ser revisada a fim de encontrar novas ações e percepções para o sistema.

Quadro 12 - Lista de Percepções

- Novo arquivo de projeto
- Mudança em arquivo
- Pedido de refatoração

Fonte: Autoria Própria

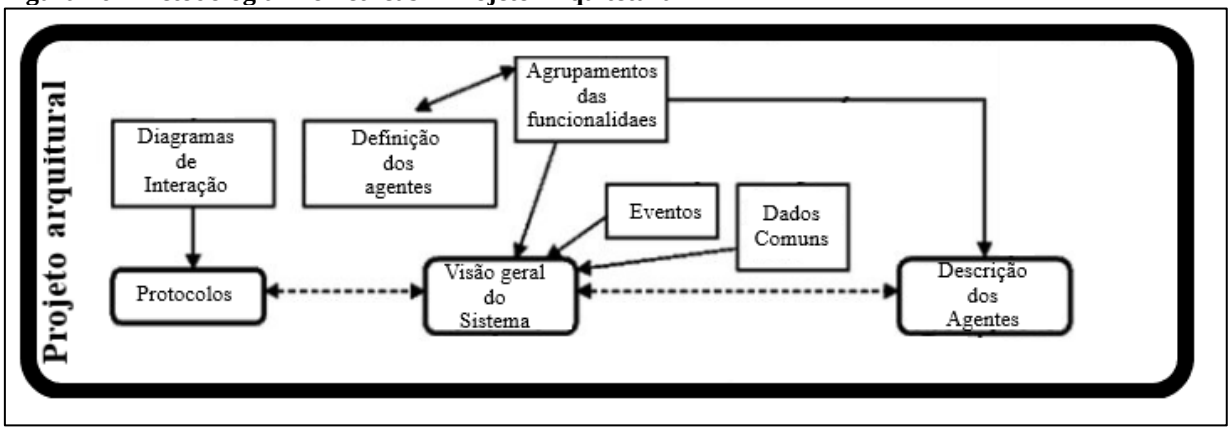
Quadro 13 - Lista de Ações

- Requisitar refatoração
- Apresentar refatoração ao usuário (opções e impacto)
- Comparar saídas

Fonte: Autoria Própria

A segunda fase da metodologia, o *Design* arquitetural, tem como finalidade agrupar objetivos, subobjetivos e funcionalidades a fim de definir quais serão os agentes presentes e sua cardinalidade e como se comunicam por meio de protocolos. A Figura 16 apresenta o projeto arquitetural proposto na metodologia *Prometheus*.

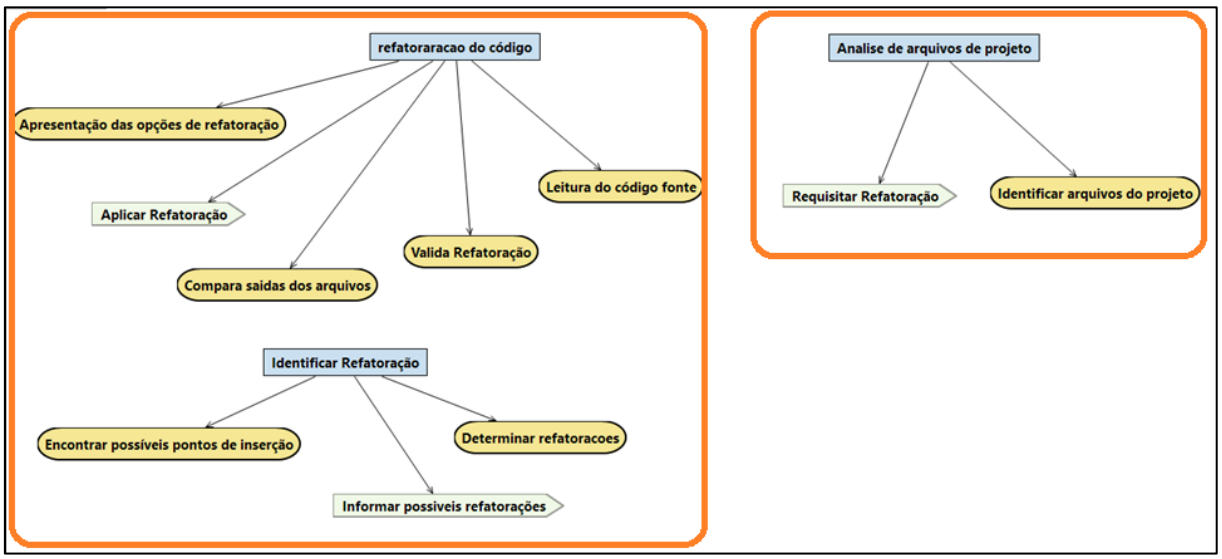
Figura 16 - Metodologia Prometheus – Projeto Arquitetural



Fonte: Adaptado de Padgham; Winikoff (2005)

A Figura 16 apresenta os dois agrupamentos viáveis para se tornarem agentes. Estes agrupamentos são baseados em funcionalidades em comum utilizadas como base no diagrama de funcionalidades apresentado na Figura 15. As funcionalidades agrupadas para o problema abordado neste trabalho abstraem um grupo responsável por refatoração e o outro grupo para a análise de arquivos.

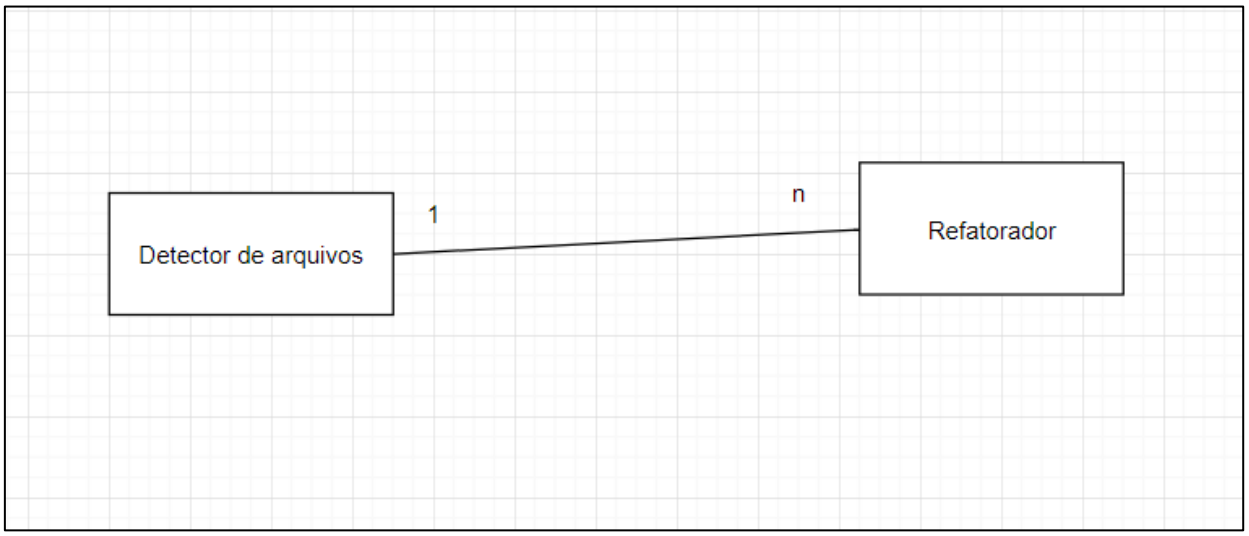
Figura 17 - Agrupamento de Agentes Proposto



Fonte: Autoria Própria

Considerando a Figura 17, os agrupamentos selecionados anteriormente são nomeados de forma significativa, representando sua funcionalidade dentro do ambiente e sua possível cardinalidade. O primeiro agrupamento foi denominado de Refatorador (contém refatoração do código e identificar refatoração) e o segundo de Detector de Arquivo (análise de arquivos de projeto), conforme exibe a Figura 18.

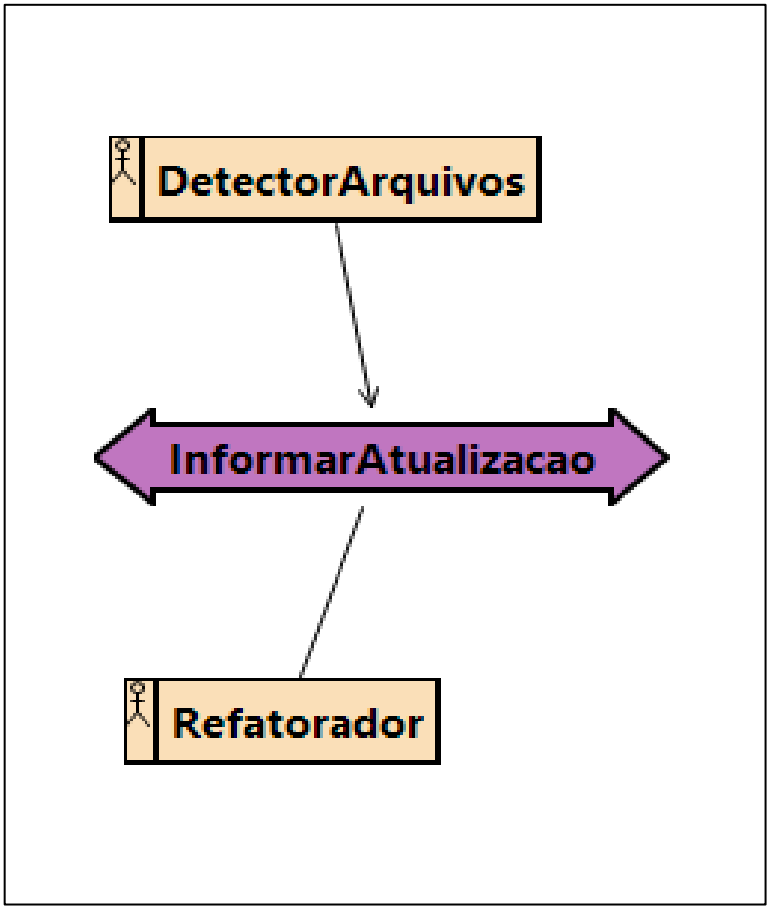
Figura 18 - Cardinalidade a Partir dos Agrupamentos



Fonte: Autoria Própria

Nesta etapa ainda são definidos os protocolos de comunicação entre os agentes. No modelo proposto existe apenas um protocolo de comunicação que informa aos agentes refatorados que seu arquivo foi modificado ou que um novo arquivo foi gerado, conforme apresenta a Figura 19. O nome do protocolo criado é *InformarAtualizacao*.

Figura 19 - Protocolo InformarAtualização



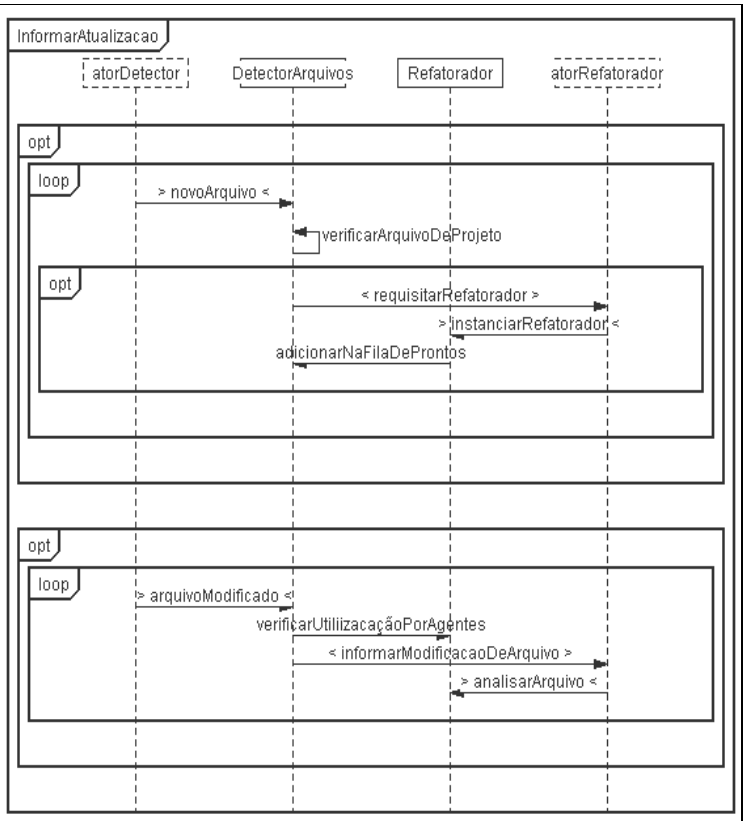
Fonte: Autoria Própria

O protocolo dentro da ferramenta PDT é modelado utilizando a linguagem AUML que gera um diagrama de sequência correspondente a ele. A AUML é uma notação textual que descreve o diagrama de iteração e protocolos de um agente. Nesta notação podem ser representados atores (*actor*) que se comportam como sensores dos agentes (pois são eles que interagem com o ambiente e obtém as percepções), os agentes (*agent*) (objetos agentes em si que podem encaminhar mensagens (*message*) para outros agentes), Percepções (*percept*) (a comunicação entre os sensores) e o próprio agente.

O Quadro 14 apresenta o diagrama de sequência usando AUML do protocolo *InformarAtualização*, em que o *atorDetector* e *atorRefatorador* são representações de sensores dos agentes *DetectorArquivos* e *Refatorador*, respectivamente.

Quadro 14 - AUML - Diagrama Sequencia Protocolo InformarAtualizacao

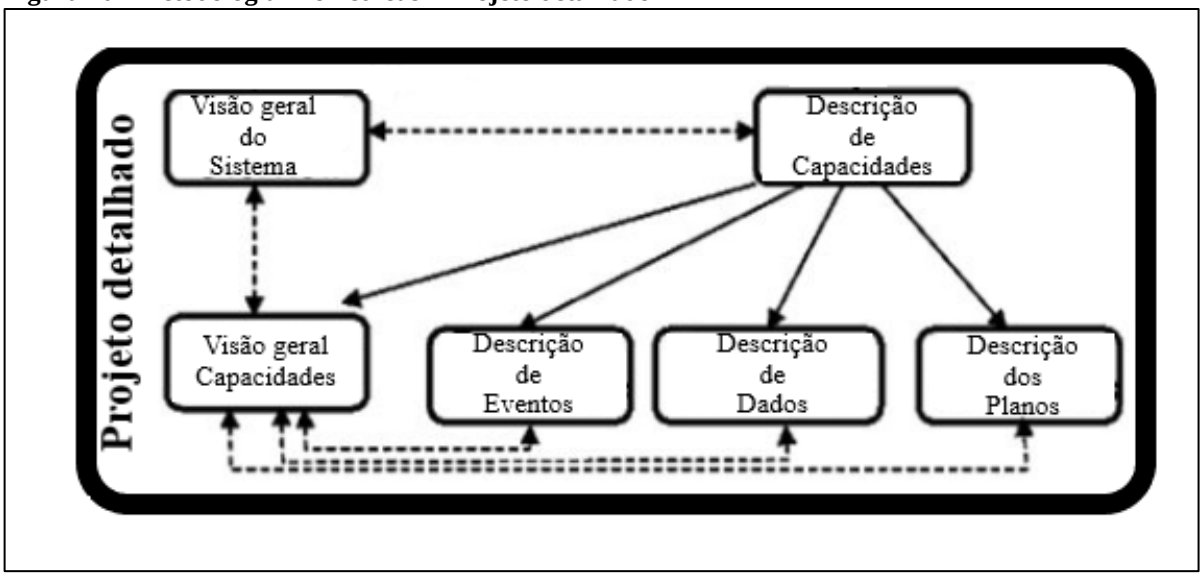
```
start InformarAtualizacao

actor A sensorDetector
agent B DetectorArquivos

agent C Refatorador
actor D sensorRefatorador

box opt
box loop
   percept  A B novoArquivo
   message B B verificarArquivoDeProjeto
box opt
   action B D requisitarRefatorador
   percept D C instanciarRefatorador
   message C B adicionarNaFilaDeProntos
end opt

end loop
end opt

box opt
box loop
   percept  A B arquivoModificado
   message B C verificarUtiliizacaçãoPorAgentes
   action B D informarModificacaoDeArquivo
   percept D C analisarArquivo
end loop
end opt
finish
```



Fonte: Autoria Própria

A última etapa da metodologia *Prometheus* é utilizada para criar a visão geral dos agentes, em que é possível visualizá-los com os protocolos que são substituídos pelas mensagens para facilitar a leitura. Nesta etapa são visíveis as ações e percepções de cada um dos agentes. A Figura 20 apresenta o projeto detalhado da metodologia *Prometheus*.

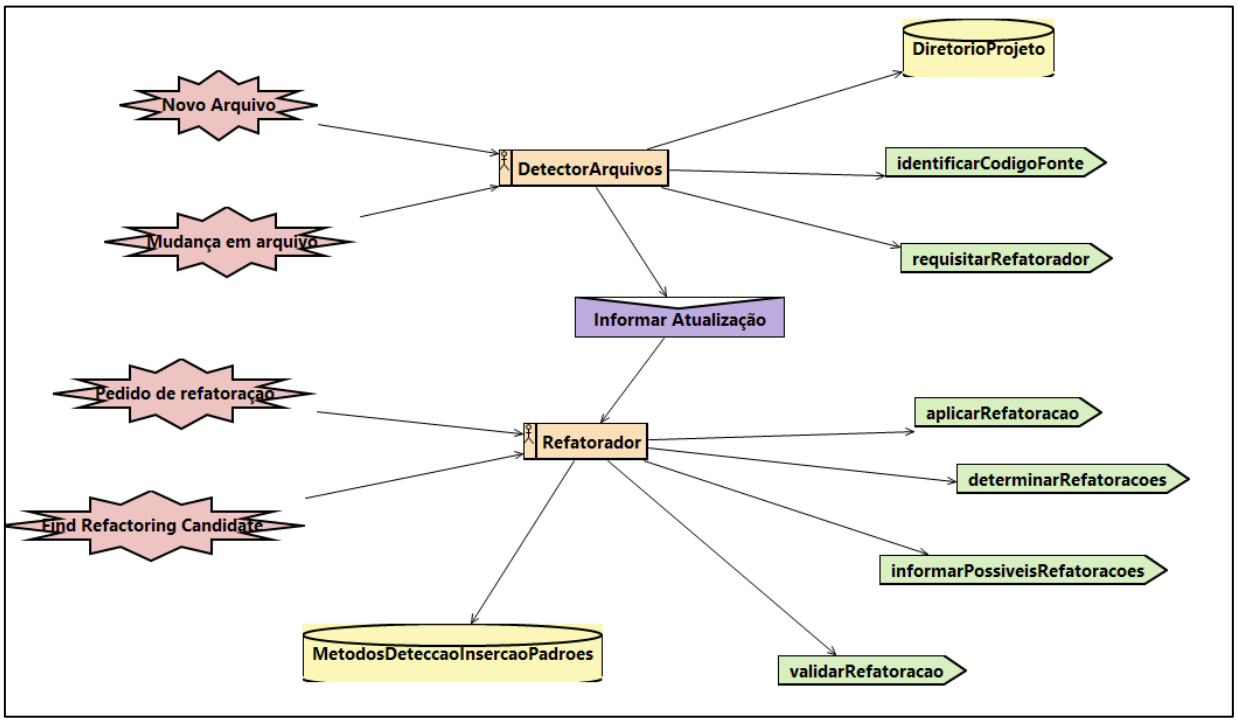
Figura 20 - Metodologia Prometheus – Projeto detalhado



Fonte: Adaptado de Padgham; Winikoff (2005)

A visão geral do agente proposto para refatoração é apresentado na Figura 21, que futuramente pode ser refinada a fim de ser implementada. O agente modelado consiste de duas partes: *DetectorArquivos* que tem como finalidade selecionar um diretório de interesse que contenha arquivos *.java*. Para cada arquivo *.java* encontrado, ele tem a responsabilidade de instanciar um novo Agente *Refatorador*. O agente Refatorador possui a finalidade de analisar o código-fonte de seus respectivos arquivos e aplicar possíveis refatorações utilizando os métodos de refatoração que foram apresentados no capítulo 4.

Figura 21 - Visão Geral dos Agentes



Fonte: Autoria Própria

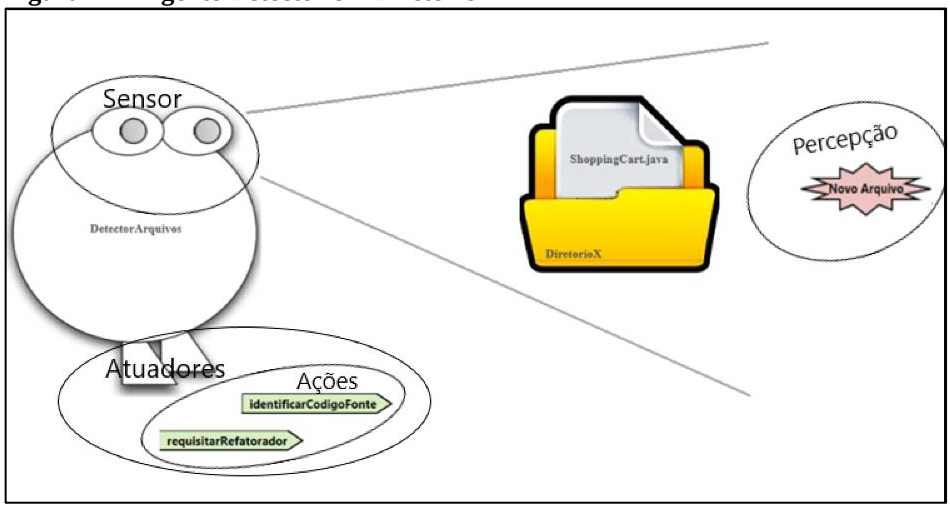
Este diagrama reúne diversos elementos que anteriormente estavam separados nos demais descritivos e diagramas (agentes, eventos e recursos compartilhados).

5.2 CENÁRIO DE TESTE PARA APLICAÇÃO DO AGENTE PROPOSTO

Considerando que o desenvolvedor necessite aplicar padrões de projeto em seu código-fonte ele pode utilizar o modelo proposto neste trabalho. Neste caso, o agente *DetectorArquivos* é instanciado em um diretório *X* e tem como finalidade encontrar arquivos de projeto a serem avaliados. Para o agente, um arquivo de projeto deve ser um *.java* e conter um código-fonte-válido.

A Figura 22 exibe como o agente *DetectorArquivos* está observando um diretório e recebendo a percepção de que um novo arquivo, denominado *ShoppingCart.java*, foi criado. O exemplo utilizado de código-fonte como cenário de teste foi retirado de Gaitani *et al*. (2015). O agente *DetectorArquivos* possui a função de *indentificarCodigoFonte* e caso o código-fonte seja válido, ele também possui a função de acionar o *requisitarRefatorador*.

Figura 22 – Agente Detector em Diretório X



Fonte: Autoria própria

O código-fonte de entrada do agente *DetectorArquivos* é ilustrado no Quadro 15. Neste exemplo, o agente utiliza uma base de conhecimento prévia para verificar se o arquivo *.java* é válido.

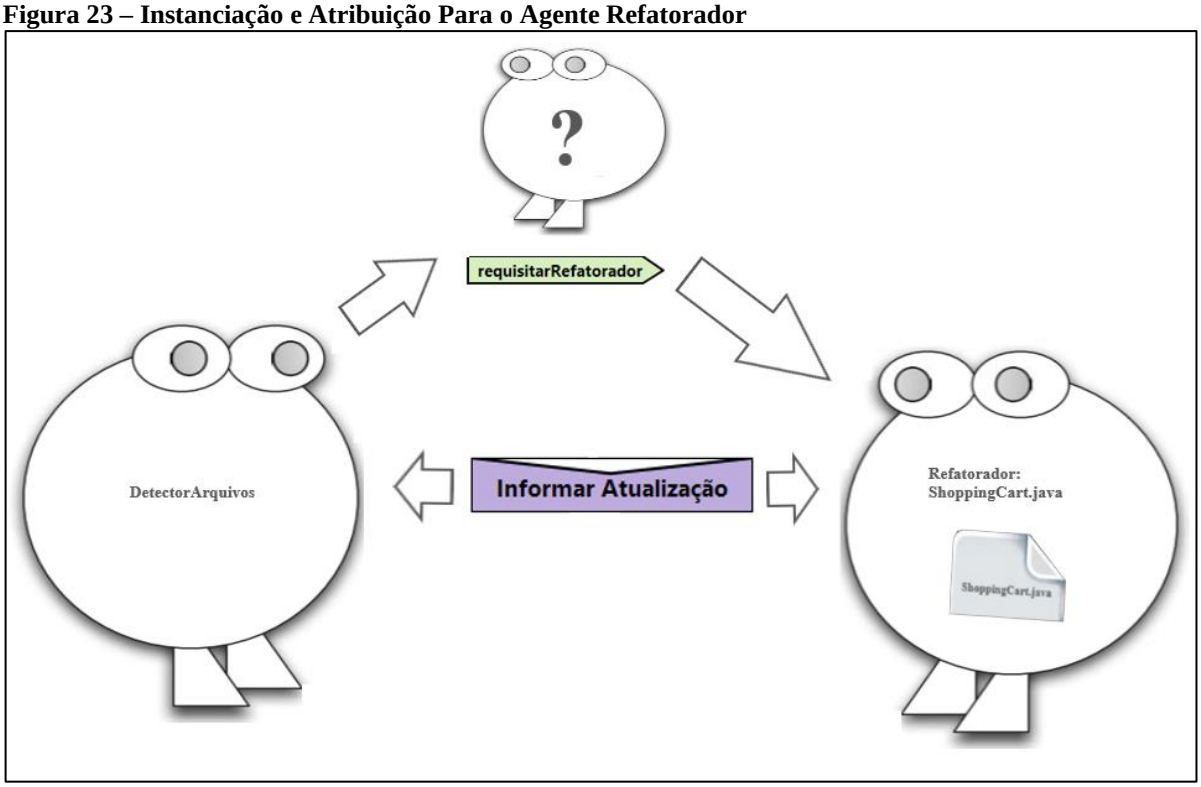
Quadro 15 – Código-Fonte de Entrada - ShoppingCart.java

```
[1]  private Customer buyer = null
[2]  //....
[3]  public shoppingCart(Customer Customer){    buyer = custumer;}
[4]  public void setBuyer(Customer buyer){  this.buyer = buyer;}
[5]  public Costumer getBuyer(){    return buyer;}
[6]  public void updateShipmentInfo( String street,String city, String country) {
[7]     Adress shipmentAdress = new Adress(street, city, country);
[8]     buyer.shipmentAdress(shipmentAdress)}
[9]  public double getDiscount(){
[10]    double discout = 0;
[11]    if(buyer != null){discout = buyer.getDiscount(); }
[12]    return discout;}
[13] public void updatePaymentInfo(String creditCard, Date issueDate) throws
[14] CustomerNotFoundException{
[15]    if(buyer != null){
[16]       buyer.setCreditCardNumber(creditCard);
[17]       buyer.setcCreditIssueDate(issueDate);
[18]    }else{   throws new CustomerNotFoundException();}}
[19] public void placeOrder() throws CustomerNotFoundException
[20] {
[21]    if(buyer == null)    {
[22]       throws new CustomerNotFoundException();}
[23]    order = new Order();
[24]    order.setCustomer(buyer);
[25]    //..}
     //..
```

Fonte: Gaitani *et al.* (2015)

Caso o arquivo *.java* seja válido, o *DetectorArquivo* possui a função de instanciar um novo agente. O agente *DetectorArquivo* informa ao novo agente que o arquivo encontrado deve ser refatorado. Como todo agente deve possuir um nome, este será relacionado ao nome do arquivo que ele está manipulando. Neste exemplo, ele receberá um nome como *refatoradorShoppingCart* ou *agenteShoppingCart*, mas para este exemplo foi denominado de *refatorador*. Esta ação de instanciação e criação de um novo agente para o arquivo específico é ilustrado na Figura 23.

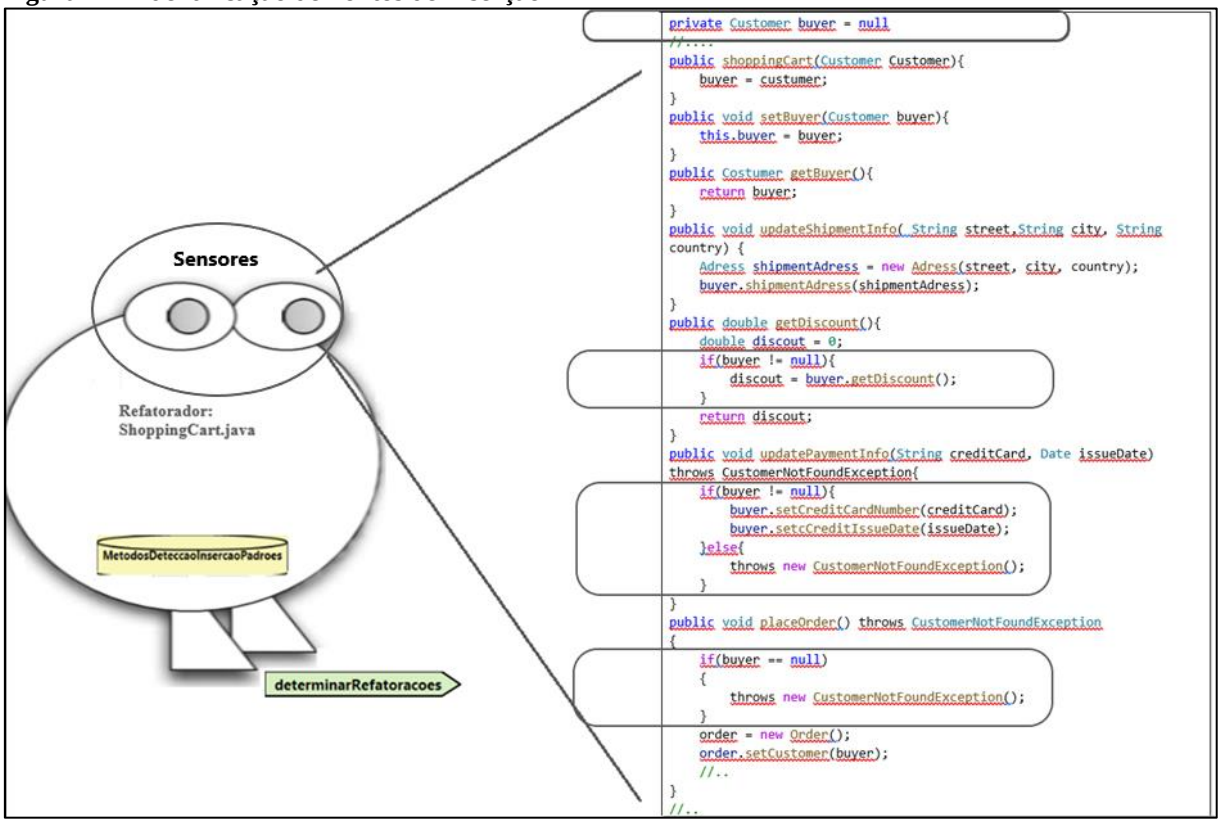
Figura 23 – Instanciação e Atribuição Para o Agente Refatorador



Fonte: Autoria própria

Após a instanciação do novo agente refatorador, ele utilizará sua base de conhecimento, explicada na seção 4.4 e analisará por meio dos métodos de detecção e inserção de padrões se algum trecho do código *.java* pode ser refatorado. Caso seja possível, o agente *refatorador* determinará as possíveis refatorações. Esta análise é representada na Figura 24.

Figura 24 – Identificação de Pontos de Inserção



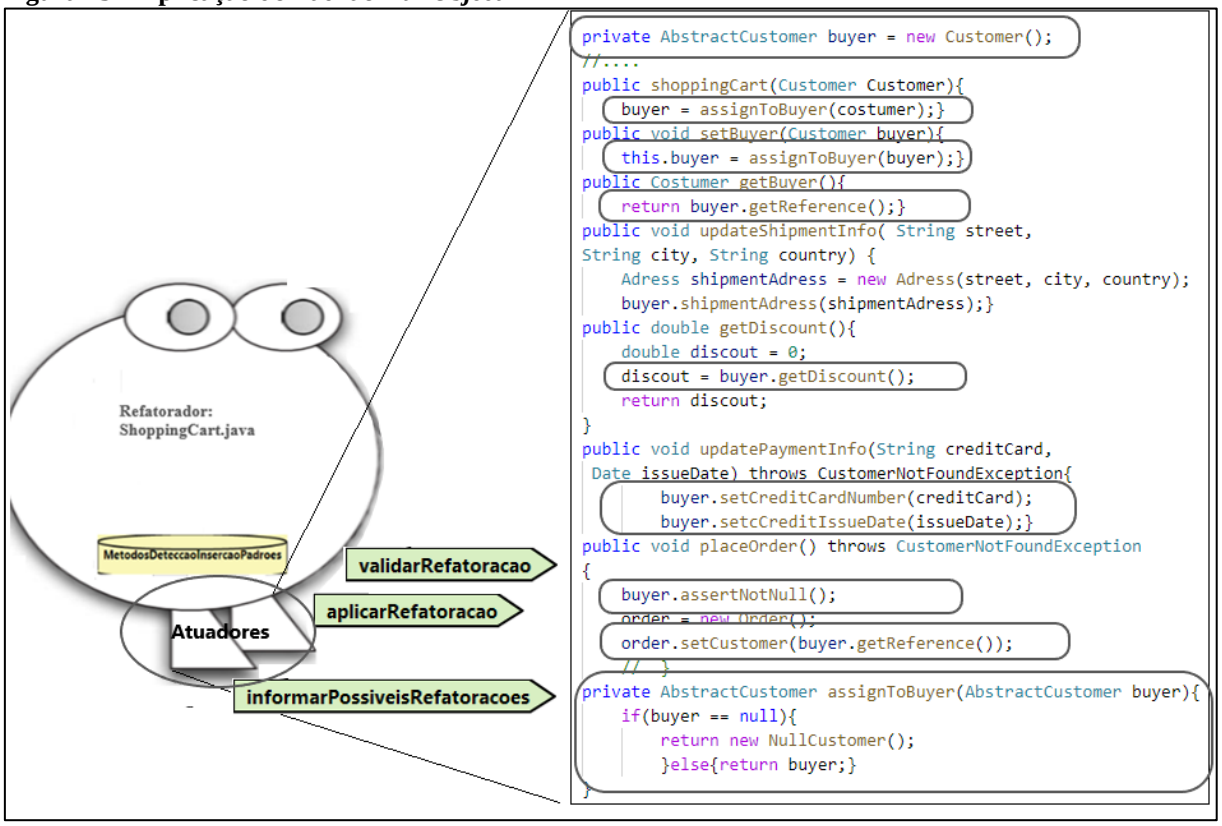
Fonte: Autoria própria

Neste exemplo, um dos padrões de projetos que o agente *Refatorador* identificou é o padrão conhecido como *NullObject*, identificado por meio das etapas definidas pelo método de Gaitani *et al*. (2015).

Segundo Gaitani *et al*. (2015), as etapas a serem seguidas para aplicar a refatoração *NullObject* são: identificar campos que são nulos e que são utilizados apenas dentro do escopo e identificar corpos *if-else* que analisam estes campos. Em seguida, no processo de refatoração, o campo definido anteriormente é definido como uma classe e são atribuídas a esta as responsabilidades de verificação sobre o objeto ser nulo ou não.

A Figura 25 exibe como o agente informa as possíveis refatorações e apresenta as mudanças a serem realizadas. Caso o usuário as aceite, o agente aplica a refatoração e faz a validação da saída.

Figura 25 – Aplicação do Padrão *NullObject*



Fonte: Autoria própria

Por fim, caso a validação seja válida, o agente aplicará a refatoração e devolverá a saída apresentada no Quadro 16.

Quadro 16 – Código-Fonte de Saida - ShoppingCart.java

```
[1]  private AbstractCustomer buyer = new Customer();
[2]  //....
[3]  public shoppingCart(Customer Customer){
[4]      buyer = assignToBuyer(costumer);}
[5]  public void setBuyer(Customer buyer){
[6]      this.buyer = assignToBuyer(buyer);}
[7]  public Costumer getBuyer(){
[8]      return buyer.getReference();}
[9]  public void updateShipmentInfo( String street,
[10] String city, String country) {
[11]     Adress shipmentAdress = new Adress(street, city, country);
[12]     buyer.shipmentAdress(shipmentAdress);}
[13] public double getDiscount(){
[14]     double discout = 0;
[15]     discout = buyer.getDiscount();
[16]     return discout;
[17] }
[18] public void updatePaymentInfo(String creditCard,
[19]  Date issueDate) throws CustomerNotFoundException{
[20]         buyer.setCreditCardNumber(creditCard);
[21]         buyer.setcCreditIssueDate(issueDate);}
[22] public void placeOrder() throws CustomerNotFoundException
[23] {
[24]     buyer.assertNotNull();
[25]     order = new Order();
[26]     order.setCustomer(buyer.getReference());
[27]     //..}
[28] private AbstractCustomer assignToBuyer(AbstractCustomer buyer){
[29]     if(buyer == null){
[30]         return new NullCustomer();
[31]         }else{return buyer;}
[32] }
```

Fonte: Gaitani *et al.* (2015)

No Quadro 16 é apresentado o resultado obtido pelo agente *Refatorador* a partir do arquivo de entrada, mostrando que a proposta do modelo apresentado é válida e passível de aperfeiçoamentos.

5.3 CONSIDERAÇÕES FINAIS

Este capítulo apresentou a proposta de modelagem para um agente de refatoração de software baseada em padrões de projeto utilizando a metodologia *Prometheus*.

O modelo de agente proposto deve englobar os métodos de refatoração presentes na literatura. O agente tem como finalidade analisar todas as possíveis refatorações e informar ao usuário quais são as possíveis mudanças e impactos no código. Neste capítulo foi apresentado um cenário de teste baseado no trabalho de Gaitani *et al*. (2015) a fim de ilustrar o funcionamento do do modelo proposto.

6. CONCLUSÃO

A identificação das metodologias de desenvolvimento de agentes foi essencial para o desenvolvimento deste trabalho, pois serviu como guia para a construção do modelo proposto. A escolha da metodologia *Prometheus Design tool* baseou-se na disponibilidade da documentação e de uma ferramenta que suporta a criação do modelo baseado em agentes (PROMETHEUS, 20017).

O modelo do agente desenvolvido não possui crenças iniciais sobre o ambiente onde está inserido e é ele atua na análise e refatoração de código-fonte Java, obedecendo as regras e princípios dos métodos de refatoração presentes na literatura.

Os métodos de refatoração propostos que devem integrar o agente foram identificados por meio do estudo do trabalho desenvolvido por Belluzo (2018). Estes métodos são capazes de detectar e inserir padrões de projeto em código-fonte, porém as ferramentas que os implementam contemplam apenas um método. Isto torna o trabalho do desenvolvedor difícil, pois necessita utilizar várias ferramentas para aplicar os padrões de projeto.

Com o modelo proposto neste trabalho pretende-se diminuir esta dificuldade porque o desenvolvedor terá um ambiente monitorado por agentes contendo vários métodos para realizar a refatoração de seu código, assim como foi exemplificado por meio do cenário de teste usando o método de refatoração o Gaitani *et al.* (2015).

Para tomar decisões, o agente proposto recebe percepções do ambiente por meio da análise dos arquivos no diretório em que ele está inserido e atua informando aos outros agentes que devem criar e modificar arquivos de código-fonte.

Conclui-se com o desenvolvimento deste trabalho que é possível modelar um agente para refatoração de software, sendo possível implementar nele os métodos de inserção e detecção de padrões de projetos propostos na literatura.

6.1 TRABALHOS FUTUROS

Os trabalhos futuros que podem ser desenvolvidos a partir desta pesquisa são:

- Refinamento do modelo proposto encontrando um nível adequado de abstração para os agentes.
- Implementação do modelo proposto para avaliar o modelo proposto e detectar possíveis falhas.

REFERÊNCIAS

BELLIFEMINE, F. *et al*. JADE, A java agent development framework. In: MULTI-AGENT PROGRAMMING. Springer US, **Proceedings...** Multi-Agent Programming, Springer, 2005, p. 125-147.

BELUZZO, L. B**. Abordagem para avaliar e detectar pontos de inserção e aplicação de padrões de projeto em código-fonte.** 2018. 101f. Dissertação (Mestrado em Ciência da Computação) - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2018.

BERNY, V. M. *et al*. Prometheus: **Metodologia de Modelagem utilizada para a Simulação de Agentes da Construção Naval**. Anais SULCOMP, v. 4, 2008

BERNY, V. M. *et al*. Utilização de metodologias para desenvolvimento de agentes: um estudo de caso na microeconomia. 2005.

BORDINI, R. H. *et al*. **Programming Multi-Agent Systems in AgentSpeak Using Jason**. John Wiley & Sons, 2007.

BORDINI, R. H.; VIEIRA, R. Linguagens de Programação Orientadas a Agentes: uma introdução baseada em AgentSpeak (L). **Revista de informática teórica e aplicada**. Porto Alegre. Vol. 10, n. 1 (2003), p. 7-38, 2003.

CHRISTOPOULOU, A. *et al*. Automated refactoring to the strategy design pattern. **Information and Software Technology**, Elsevier, v. 54, n. 11, p. 1202–1214, 2012.

CINNÈIDE, M. Ó. **Automated application of design patterns:** a refactoring approach. 2001. 242 f. Thesis (Doutorado) — Programa de Pós-Graduação University of Dublin, Trinity College Dublin, 2001.

CINNÈIDE, M. Ó. Automated refactoring to introduce design patterns. In: ACM. **Proceedings of the 22nd international conference on Software engineering**. [S.l.], 2000. p. 722–724.

CINNÈIDE, M. Ó.; NIXON, P. A methodology for the automated introduction of design patterns. In: IEEE. **Software Maintenance, 1999. (ICSM'99) Proceedings. IEEE International Conference on**. [S.l.], 1999. p. 463–472.

CINNÉIDE, M. Ó; NIXON, P. Automated software evolution towards design patterns. In: THE 4th INTERNATIONAL WORKSHOP ON PRINCIPLES OF SOFTWARE EVOLUTION, Viena Autria. **Proceeding...** Viena Áustria, 2001, p. 162–165.

CINNÈIDE, M. Ó; NIXON, P. Automated software evolution towards design patterns. In: ACM. **Proceedings of the 4th international workshop on Principles of software evolution**. [S.l.], 2001. p. 162–165.

DE LIMA, T. F. M. *et al*. Modelagem de sistemas baseada em agentes: Alguns conceitos e ferramentas. In: XIV SIMPÓSIO BRASILEIRO DE SENSORIAMENTO REMOTO, Natal Brasil, **Anais...** Natal, 2009, p. 25-30.

DE NUNES, I. O. Implementação do Modelo e da Arquitetura BDI. **Monografias em Ciência da Computação**, v. 1, 2007.

FERREIRA, A. A. C. **Metodologia para análise e projeto de sistemas multi-agentes**. 2008.

FIPA, A. C. L. Fipa acl message structure specification. Foundation for Intelligent Physical Agents, **Disponivel em:** <http://www. fipa. org/specs/fipa00061/SC00061G.html >. Acesso em 20 out 2018.

FOWLER, *et al*. **Refactoring: improving the design of existing code**. [S.l.]: Addison-Wesley Professional, 1999.

FOWLER, M. **Padrões de Arquitetura de Aplicações Corporativas**, Porto Alegre, 2006.

FOWLER, M. Refactoring in alphabetical order. **Catalog of Refactorings. 10 dec 2013. Disponivel em:**<http://www.refactoring.com/catalog/index.html>. Acesso em: 23 set, 2017.

FOWLER, M.; BECK, K. **Refactoring:** improving the design of existing code. Addison-Wesley Professional, 1999.

GAITANI, M. A. G. *et al*. Automated refactoring to the null object design pattern. **Information and Software Technology**, v. 59, p. 33-52, 2015.

GAMMA, E. *et al*. **Padrões de Projetos:** Soluções Reutilizáveis. Bookman editora, 2000.

GE, X.; DUBOSE, Q. L.; MURPHY-HILL, E. Reconciling manual and automatic refactoring. In: **Proceedings of the 34th International Conference on Software Engineering**. IEEE Press, 2012. p. 211-221.

GIRARDI, R. Engenharia de Software baseada em Agentes. In: IV CONGRESSO BRASILEIRO DE CIÊNCIA DA COMPUTAÇÃO (CBCOMP 2004). Santa Catarina Brasil. **Anais...** Santa Catarina Sul, 2004.

JASON. **About Jason**. Disponível em: <http://jason.sourceforge.net/wp/>. Acesso em: 17 nov. 2018.

JEON, S.; LEE, J.; BAE, D. An automated refactoring approach to design pattern-based program transformations in Java programs. In: SOFTWARE ENGINEERING CONFERENCE, 2002. NINTH ASIA-PACIFIC. **Proceedings...** Ninth Asia-Pacific, 2002, p. 337-345.

JUCHEM, M.; BASTOS, R. M. Arquitetura de Agentes. **Thecnical Report Series. Porto Alegre**, n. 013, 2001.

KIM, J.; BATORY, D.; DIG, D. **Scripting Refactorings in Java to Introduce Design Patterns**. UTexas-Austin, Tech. Rep. TR-14-14 (2014). p.24

LIU, W. *et al*. Automated pattern-directed refactoring for complex conditional statements. **Journal of Central South University**, v. 21, n. 5, p. 1935-1945, 2014.

LIU, W.; *et al*. Automated pattern-directed refactoring for complex conditional statements. **Journal of Central South University**, Springer, v. 21, n. 5, 2014.

MENS, T.; TOURWÉ, T. A Declarative Evolution Framework for Object-Oriented Design Patterns. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, 2001**. Proceedings...** Florence Italy, 2001, p. 570-579.

MOHAMAD, R.; DERIS, S.; AMMAR, H. H. **MaSE2Jadex:** A Roadmap to Engineer Jadex Agents from MaSE Methodology. International Journal of Intelligent Technology, v. 1, n. 3, p. 245-251, 2006.

OUNI, A. *et al*. MORE: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. **Journal of Software: Evolution and Process**, v. 29, n. 5, 2017.

PADGHAM, L.; WINIKOFF, M. **Developing intelligent agent systems:** A practical guide. John Wiley & Sons, 2005.

PROMETHEUS. Prometheus Design Tool (PDT)**. Disponivel em:**< http://researchbank.rmit.edu.au/view/rmit:2462/n2006009739.pdf >. Acesso em: 23 set, 2017.

RAJESH, J.; JANAKIRAM, D. JIAD: A tool to infer design patterns in refactoring. **Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming**, 2004. p.227-237.

RUSSEL, S.; NORVIG, P**. A modern approach:** Artificial Intelligence. Prentice-Hall: Egnlewood Cliffs, 1995, v. 25, p. 27

RUSSEL, S.; NORVIG, P. **Inteligência artificial**. Campus: São Paulo, 2004, p. 26.

SHOHAM, Y. Agent-oriented programming. **Artificial intelligence**, v. 60, n. 1, p. 51-92, 1993.

TOKUDA, L.; BATORY, D. Evolving object-oriented designs with refactorings. **Automated Software Engineering**, v. 8, n. 1, p. 89-120, 2001.

TSANTALIS, N. *et al*. Design pattern detection using similarity scoring. **IEEE transactions on software engineering**, v. 32, n. 11, 2006.

WOOLDRIDGE, M. **An introduction to Multiagents Systems**. 1. ed. Londres: Jhon Wiley & Sons, 2002

WOOLDRIDGE, M.; JENNINGS, N. R. Intelligent agents: Theory and practice. **The knowledge engineering review**, v. 10, n. 2, p. 115-152, 1995.

YRLEYJÂNDER SALMITO, L**. Desenvolvimento Orientado A Modelos Em Sistemas Multi-Agentes Com Diferentes Arquiteturas Internas De Agente.** Dissertação (Mestrado em Ciência da Computação) -Universidade Estadual do Ceará - UECE Centro De Ciências e Tecnologia. Ceará, p. 116. 2015.

ZAFEIRIS, V. E. *et al*. Automated refactoring of super-class method invocations to the template method design patterns. Information and Software Technology. 2017. p.19-35.

ZAFEIRIS, V. E. *et al.* Automated refactoring of super-class method invocations to the template method design patterns. **Information and Software Technology**. 2017. p.19-35.

ZAMBERLAM, A. de O.; GIRAFFA, L. M. M. Modelagem de agentes utilizando a arquitetura BDI. **Porto Alegre: Pontifícia Universidade Católica do Rio Grande do Sul, Faculdade de Informática**, 2001.