UNIVERSIDADE DO ESTADO DO RIO GRANDE DO NORTE
FACULDADE DE CIÊNCIAS EXATAS E NATURAIS
DEPARTAMENTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

PEDRO VITOR LIMA RODRIGUES

PIABA
UM FRAMEWORK PARA DESENVOLVIMENTO DE APLICAÇÕES MULTIAGENTES EM AMBIENTES MÓVEIS NA PLATAFORMA ANDROID

NATAL
2013

PEDRO VITOR LIMA RODRIGUES

PIABA
UM FRAMEWORK PARA DESENVOLVIMENTO DE APLICAÇÕES MULTIAGENTES EM AMBIENTES MÓVEIS NA PLATAFORMA ANDROID

Monografia apresentada à Universidade do Estado do Rio Grande do Norte como um dos pré-requisito para obtenção do grau de bacharel em Ciência da Computação.

ORIENTADOR: Alberto Signoretti

NATAL
2013

Catalogação da Publicação na Fonte.
Universidade do Estado do Rio Grande do Norte.

Rodrigues, Pedro Vitor Lima

Piaba: um framework para desenvolvimento de aplicações multiagentes em ambientes móveis na plataforma android. / Pedro Vitor Lima Rodrigues. – Mossoró, RN, 2013.

53f.
Orientador: Prof. Alberto Signoretti

Monografia (Graduação). Universidade do Estado do Rio Grande do Norte. Departamento de Ciência da Computação.

1. Inteligência artificial distribuída. 2. Sistemas multiagentes – Android. 3. Framework. I. Signoretti, Alberto II.Universidade do Estado do Rio Grande do Norte. III.Título.

UERN/BC CDD 003.3

Bibliotecária: Jocelania Marinho Maia de Oliveira CRB 15 / 319

PEDRO VITOR LIMA RODRIGUES

PIABA
UM FRAMEWORK PARA DESENVOLVIMENTO DE APLICAÇÕES MULTIAGENTES EM AMBIENTES MÓVEIS NA PLATAFORMA ANDROID

Monografia apresentada à Universidade do Estado do Rio Grande do Norte como um dos pré-requisito para obtenção do grau de bacharel em Ciência da Computação.
Orientador: Dr. Alberto Signoretti

Aprovado em **_____/_____/_____.**

Banca Examinadora

_____
Orientador: Dr. Alberto Signoretti
UERN

_____
Dr. Carlos André Guerra Fonseca
UERN

_____
MSc. Camila de Araújo
UERN

_____
Dr. André M. C. Campos
UFRN

DEDICATÓRIA

À todas as pessoas que me ajudaram e me deram forças de forma direta ou indireta.

AGRADECIMENTOS

Ao meu pai, que infelizmente partiu antes da minha graduação chegar ao fim. Era meu sonho antigo fazê-lo ver pessoalmente. Infelizmente não foi possível.

Ao meu orientador, Alberto Signoretti, pelo apoio não só na esfera acadêmica também na pessoal. Características de um verdadeiro amigo, e assim o considero.

À minha família, Elizabete, João Carlos e José Arnaldo, pelo apoio insistente durante toda a jornada.

Aos meus amigos Marcos Vinícius, Werllon Melo, Luiz Henrique, Luciana Josiedson, Israel Sousa e Hélio Júnior, pela amizade, conselhos e força.

À minha namorada Marcela Veras, pelo amor e carinho onde encontrei tantas vezes.

O que você sabe não tem valor. O valor está no que você faz com o que sabe.

Bruce Lee

RESUMO

O objetivo desse trabalho é explicar o *framework* Piaba, voltado para desenvolvimento de simulações multiagentes em dispositivos móveis utilizando o sistema operacional Android. O *framework* foi desenvolvido em linguagem nativa desse ambiente e é descrito através da explicação de seu funcionamento e da exemplificação de seu uso em um jogo que utiliza simulação multiagente e interações com usuário para o funcionamento. Neste trabalho, além da descrição da estrutura do *framework* também são abordados aspectos da modelagem da aplicação, do seu ciclo de vida e o modelo de atualização das percepções dos agentes. A ferramenta possibilita o desenvolvimento de aplicações de forma rápida, simples e consumindo poucos recursos do hardware.

O *framework* Piaba oferece suporte nativo a agentes com foco afetivo e atualização de emoções que podem ser utilizadas de forma opcional. Essa atualização é realizada através da execução de operadores que devem ser implementados pelo desenvolvedor utilizando as classes auxiliares da ferramenta. O desenvolvedor é livre para implementar o modelo afetivo que desejar, utilizando os operadores como atuadores de incremento e decremento emocional.

O desenvolvimento de uma aplicação multiagente utilizando o *framework* Piaba permite a utilização de interfaces gráficas independentes, criadas pelo próprio desenvolvedor. A atualização dessas interfaces gráficas é realizada automaticamente através da execução de instâncias de classes específicas criadas pelo próprio desenvolvedor a partir de classes do *framework*.

As configurações da aplicação são realizadas por meio de arquivos *XML*, separando o código de configuração do código de aplicação. Nesses arquivos devem ser colocados informações e parâmetros referentes às configurações necessárias para definir o comportamento da aplicação desenvolvida.

PALAVRAS-CHAVE**:** agentes, inteligência artificial distribuída; sistemas multiagentes; android; *framework*; agentes afetivos;

ABSTRACT

The aim of this work is to explain the *framework* Piaba, facing development of multi-agent simulations on mobile devices using the Android operating system. The *framework* was developed in native language and that environment is described through its use in a game that uses multi-agent simulation and user interactions for the operations. In this work, besides the description of the structure of the *framework* are also explained aspects of the modeling of the application, its life cycle and update model perceptions of agents. The tool enables the development of applications quickly, easily and consumes few hardware resources.

The *framework* supports native Piaba agents focusing affective and updating of emotions that can be used optionally. This update is performed through the execution of operators that must be implemented by the developer using the tool's classes. The developer is free to implement the affective model you want, using the operators as actuators increment and decrement emotions.

The development of an multi-agent application using the *framework* Piaba permits the use of independent graphic interfaces, created by the developer. The update of these GUIs is performed automatically by running instances of specific classes created by the developer from *framework* classes.

Application settings are performed via XML files, separating the configuration code of application code. In these files some informations should be placed, like parameters pertaining to the settings needed to define behaviours of the application.

KEYWORDS**:** *agents, distributed artificial inteligence; multi-agent systems; android; framework; affective agents*;

LISTA DE SIGLAS E ABREVIAÇÕES

ACL – Agent Communication Language

API – Application Programming Interface

BSD – Berkeley *software* Distribution

FIPA – Foundation for Intelligent Physical Agents

IA – Inteligência Artificial

JADE – Java Agent Development *framework*

JME3 – jMonkey 3D

MIO – Meta-Info Operators

OCC - Ortony, Clore, & Collins, 1988

PC – Personal Computer

SDK – Standard Development Kit

XML – Extensible Markup Language

3D – Três dimensões

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Atualmente existem algumas ferramentas que podem ser utilizadas para criação de agentes inteligentes. O Jade e o Jason estão entre essas, sendo o primeiro uma plataforma que fornece funcionalidades básicas de um middleware multicamadas independentes do restante da aplicação e que simplifica a realização de aplicações distribuídas que exploram a abstração de agente (Jennings, Wooldridge, 1988), enquanto o segundo é um *framework* para desenvolvimento de agentes BDI *(Belief-Desire-Intention)* através da implementação dos agentes em linguagem Java e da especificação das suas crenças, desejos e intenções em uma linguagem chamada *AgentSpeak* (Bordini, Hübner, 2004). Apesar da popularidade, essas ferramentas, assim como várias outras disponíveis, não são voltadas ao desenvolvimento específico de agentes móveis para a plataforma móvel Android. Com o surgimento e evolução da plataforma, houveram esforços para migrar ou adaptar uma versão estável dessas ferramentas de forma à possibilitar sua utilização nesse sistema operacional (Santi et al., 2010; Caire et al., 2012). O Jade originou o Jade-Android enquanto o Jason, integrado com outro *framework* chamado CartAgO voltado para desenvolvimento de ambientes para execução de agentes, originou o JaCa-Android. Apesar dessas versões tornarem os *framework*s originais compatíveis com o sistema operacional Android, restrições ainda impedem que essas versões se tornem tão uteis e populares quanto às originais quando o objetivo é desenvolver simulações multiagentes móveis. Esse trabalho foi desenvolvido com o principal objetivo de disponibilizar um novo *framework* como ferramenta para desenvolvimento de aplicações desse tipo, em meio às poucas opções existentes no momento atual.

Sendo este um assunto bastante abrangente, envolvendo desde simulações até a aplicação em jogos, a falta de opções de padrões para *design* de aplicação nesse sentido acabam levando os desenvolvedores a codificar aplicações seguindo padrões próprios, específicos da aplicação e que na maioria das vezes não podem ser reutilizados por outros desenvolvedores. Esse código criado pelos desenvolvedores é desenvolvidos com o objetivo de resolver um determinado problema específico da aplicação em desenvolvimento. A consequência disso é que esse código, por exigir um esforço maior na sua elaboração, faz com que o desenvolvedor codifique recursos que muitas vezes já foram implementados por

outros desenvolvedores. A reutilização de código nesses casos, possibilitado pela utilização de um *framework*, diminuiria o custo final dessas aplicações tornando o foco do desenvolvedor voltado a criar componentes fortes, pequenos e específicos do domínio de sua aplicação (Rierson, 2000).

É nesse cenário que foi idealizado o *framework Piaba*. Trata-se de uma ferramenta livre para o desenvolvimento de aplicações de forma simples, que utiliza poucos recursos de hardware de forma otimizada em uma plataforma móvel com sistema operacional Android. A ferramenta é um *software* livre distribuída sob licença BSD e sua utilização por outros desenvolvedores permitirá um aumento na taxa de aprimoramento em versões futuras, uma vez que os usuários poderão reportar novas necessidades além de problemas encontrados. O *framework* foi desenvolvido utilizando linguagem Java, mesma linguagem utilizada para desenvolvimento da maioria das aplicações do sistema operacional onde é executada, além de XML para realização de configurações necessárias para o funcionamento. Utilizando a ferramenta para desenvolver aplicações, o programador focará principalmente no desenvolvimento de um ambiente de execução (mundo) e dos agentes que serão executados nesse mundo.

O objetivo da criação desse framework é definir uma ferramenta para o desenvolvimento de sistemas multiagentes que possa ser utilizado para criação de aplicações móveis isoladas da interface gráfica. O trabalho aborda as funcionalidades disponibilizadas pelo framework, além dos estudos e modelos utilizados para sua concepção. Foi realizado um comparativo entre Piaba e outras ferramentas com o mesmo intuito, o Jade-Android e o JaCa-Jason já citados anteriormente. Na parte final há a explicação de um caso de uso que foi desenvolvido com o intuito de mostrar as funcionalidades em execução. Esse caso de uso é um pequeno jogo onde os agentes, representados por robôs, executam diversas ações durante a execução da aplicação. O jogador pode interagir com os agentes, interrompendo a execução das ações deles para que eles realizem outras ações. Uma interface gráfica simples foi desenvolvida, de forma isolada do framework, afim de demonstrar a integração entre as duas camadas.

Inicialmente, o *framework* estava sendo desenvolvido com intuito de utilização em plataformas fixas (PC). O destaque e popularidade da plataforma Android nos últimos anos (Koetsier, 2013) aliado ao desenvolvimento da ferramenta ser realizada em Java, principal linguagem utilizada no desenvolvimento das aplicações para Android, motivaram a ideia de tentar migrar a ferramenta para a plataforma

móvel. A implantação foi feita com relativa facilidade e sem necessidade de muitas alterações, indicando que é viável a possibilidade do desenvolvimento de duas versões do *framework* (uma para plataforma móvel e outra para plataforma fixa).

Esse documento está dividido basicamente em quatro partes. A primeira (capítulo 2) parte aborda os conceitos básicos necessários para entender o funcionamento do *framework*, como agentes inteligentes e sistemas multiagentes. A segunda parte (capítulos 3 e 4) explica o funcionamento dos componentes da ferramenta, sua inicialização e seu ciclo de vida, além de realizar um comparativo entre *framework* e outras ferramentas disponíveis. A terceira parte (capítulo 5) foca na descrição de uma aplicação de exemplo, que utiliza as principais funcionalidades do *framework* Piaba. A quarta e última parte (capítulos 6 e 7) refere-se à conclusão e aos trabalhos futuros relacionados a este.

2 CONCEITOS BÁSICOS

2.1 AGENTES INTELIGENTES E SISTEMAS MULTIAGENTES

Um agente inteligente pode ser definido como um componente de *software* que possui como características básicas estar situado e um ambiente e ser autônomo (Padgham, Winikoff, 2004). Possuindo autonomia, eles tornam-se independentes e podem tomar suas próprias decisões no ambiente onde estão localizados. Dessa forma, um sistema multiagente é um programa de computador que utiliza um ou mais agentes inteligentes localizados em um ambiente. Os agentes podem possuir objetivos a serem alcançados, e sua localização no ambientem por exemplo, pode ser uma informação que o agente possa perceber e usar para tomar decisões. Por exemplo, um agente *aluno* com o objetivo *encontrar um local para sentar,* pode perceber um ambiente *sala de aula* que uma cadeira está vazia. Em seguida, o aluno pode *decidir* sentar nessa cadeira, alterando o estado da cadeira de vazia para ocupada. Esse estado poderá ser percebido por agentes que venham depois desse primeiro que decidirão por não sentar nessa mesma cadeira, pois a mesma encontra-se ocupada.

A utilização de sistemas multiagentes permite delegar responsabilidades diferentes e mais exatas a cada agente, tornando maior o nível de coesão das partes de uma aplicação e diminuindo seu acoplamento, tornando o sistema mais modular, descentralizado e mutável (Padgham, Winikoff, 2010). Nesses casos, a redução do acoplamento é ocasionada pela robustez, reatividade e proatividade dos agentes. Dessa forma, quando um objetivo é definido por um agente, cabe

exclusivamente a ele tomar as decisões e realizar as ações necessárias para alcançar esse objetivo.

A abordagem multiagente constitui uma maneira inovadora e promissora de desenvolver e gerenciar sistemas distribuídos (Jennings 2001). Muitas são as áreas onde aplicações desse tipo podem ser utilizadas para atingir algum objetivo. Um exemplo de utilização é a análise de dados para detecção de inconsistências e falhas durante a execução de ações. Atualmente, a utilização de sensores e dados captados por eles são cada vez mais utilizados para analisar e avaliar a qualidade e a eficácia da execução de máquinas na indústria. Com o avanço tecnológico, a quantidade de sensores e a interação entre eles é cada vez maior, tornando a análise e processamento desses dados para realização de inferências significantes mais difíceis e custosas para um ser humano. Dessa forma, sistemas computacionais com abordagem multiagente podem ser utilizados para analisar esses dados, tomar conhecimento de situações de falha e até tomar decisões para melhorar a produção (Queiroz, Guilherme, 2012).

Os agentes inteligentes podem ser classificados em:

- Cognitivos, que são baseados em organizações sociais humanas. Eles representam explicitamente o ambiente e os agentes da sociedade e possuem seu sistema de percepção e de comunicação distintos, além de poderem armazenar as ações realizadas na memória (Faber, 1991);
- reativos, onde não há representação explícita do ambiente ou representação de conhecimento nem memória das ações realizadas pelos agentes (Costa, Feijó, 1995).

Enquanto agentes cognitivos são executados independente dos acontecimentos do mundo, os agentes reativos respondem apenas à determinadas condições preestabelecidas. Um agente reativo pode ser utilizado para detecção e reação a sinais de perigo. Nesse caso, o agente é executado (reage) apenas quando o perigo ocorre. Um agente cognitivo pode procurar sinais de perigo e, ao encontrá-los, tomar decisões e realizar ações. A ferramenta desenvolvida nesse trabalho pode ser utilizada para desenvolvimento de ambos os tipos.

2.2 SIMULAÇÃO MULTIAGENTE

Uma simulação multiagente é uma simulação que utiliza agentes inteligentes como entidades ativas nela, ao contrário do que ocorre em simulações a eventos

discretos onde os objetos são elementos passivos controlados por um fluxo algorítmico (Signoretti et al., 2008).

A utilização de simulações multiagentes nas diversas áreas do conhecimento humano tem trazido inúmeros benefícios, como a redução de riscos na tomada de decisão e a identificação de possíveis problemas antes mesmo de suas ocorrências (Signoretti et al., 2008). Através de uma simulação de comportamento humano é possível prever também situações e ações de uma população que se deseja evitar. Configurando agentes inteligentes com atributos humanos, como emoções e sentimentos, em um mundo (ambiente) que representa uma sala com saídas e sinais, é possível simular como seria o comportamento de uma pequena massa de pessoas em caso de incêndio (Signoretti, 2012). Através da alteração de sinais sonoros de incêndio e da posição das saídas, as percepções dos agentes podem ser melhoradas e a rota traçada até a saída mais próxima pode ser otimizada. As alterações no ambiente, como a propagação do fogo e o aumento da fumaça, podem ser percebidas pelos agentes e ocasionar na alteração de suas emoções, que influenciam nas suas ações. Dessa forma, um sistema multiagente pode ajudar na elaboração arquitetônica de um ambiente que auxilie melhor na segurança de seus ocupantes.

Em um jogo de estratégia que envolve extração de recursos, os trabalhadores visualizam uma parte do estado real do mundo (ambiente) em um determinado momento e, caso percebam a presença de recursos na sua área de visão, podem intuitivamente iniciar a extração sem necessidade da ordem do jogador. A mesma inteligência pode ser utilizada nos agentes guerreiros, que ao detectar a presença de inimigos em sua área de visão inicia um ataque. Sistemas multiagentes podem ser utilizados para a implementação da inteligência artificial de jogos de estratégia desse tipo (Sandberg, 2011), tornando cada componente de ação independente do jogo como um agente com atributos relacionados, como força, defesa e agilidade, que são utilizados para calcular ações.

2.4 ANDROID

O Android é um sistema operacional *open source* voltado para dispositivos móveis, mantido por um consórcio de várias empresas denominado Open Handset Alliance, criado com o intuito de acelerar a inovação em ambiente móvel e oferecer a consumidores uma experiência mais rica e de menor custo (Gargenta, 2011). A plataforma oferece um ambiente padrão para o desenvolvimento de aplicativos de

telefonia móvel e com isso criou um mercado para aplicações do tipo (Rogers, 2009). O idealizador e principal mantenedor do sistema é a empresa Google.

Após ser anunciado pela empresa Google em 2007, em poucos meses mais de um milhão de pessoas já haviam baixado o seu SDK para desenvolvimento de aplicativos. Com a evolução e consequente barateamento do hardware móvel, ocasionando a popularização dos smartfones e tablets, o sistema operacional tornou-se cada vez mais acessível e comum nesses dispositivos. Hoje, o Android é o sistema operacional mais utilizado no mundo (Koetsier, 2013).

A plataforma foi criada não apenas como um sistema operacional multimídia para dispositivos móveis, mas também como uma plataforma para desenvolvimento de aplicações móveis. Através da sua popularização, o Android minimizou vários problemas que até então inviabilizavam o desenvolvimento de aplicações móveis. Um desses problemas é a fragmentação que havia na época, com a existência de uma quantidade muito grande de hardwares disponíveis tornando o desenvolvimento de aplicações um desafio complicado, tendo em vista que detalhes como tamanho de tela e processadores de modelos diferentes, por exemplo, precisavam ser tratados de maneiras diferentes (Rogers 2009).

O ambiente Android possui uma loja virtual denominada Google Play, onde qualquer desenvolvedor de aplicativos registrado pode disponibilizar seu aplicativo, desde que o mesmo esteja de acordo com algumas normas da loja, e cobrar por isso. A simplicidade do hardware e *software* necessários para desenvolvimento dessas aplicações levaram à um grande crescimento de aplicações de terceiros disponibilizadas na comunidade. Em 2013 eram registrados mais de 50 milhões de downloads na loja online, como mostra a Figura 2, e mais de 1 milhão de aplicativos disponíveis para compra e download.

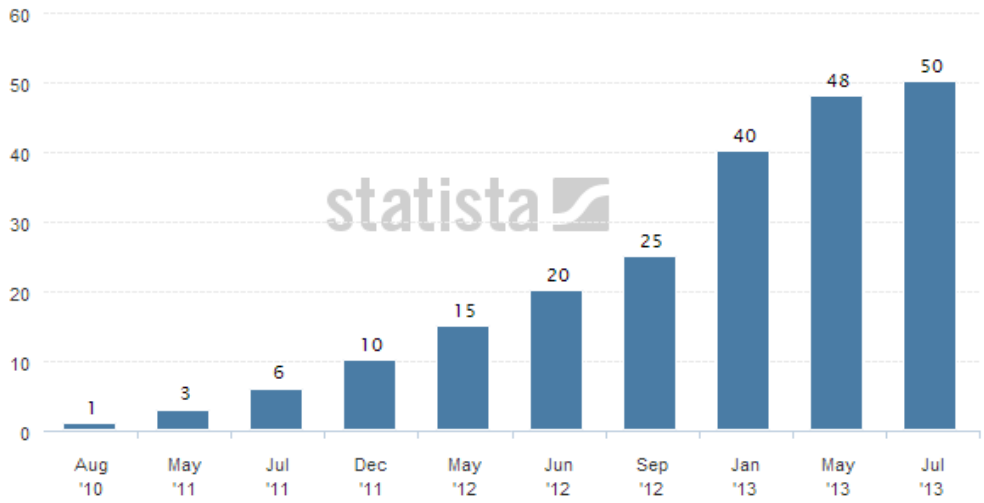

Figura 1: Comparação da quantidade de downloads de aplicações do Google Play em 2012 e 2013. Fonte: Statista, Google, Android

As aplicações do Android, bem como a maior parte do sistema operacional, são desenvolvidas em Java. O sistema operacional possui alguns componentes desenvolvidos em C e C++ e um kernel baseado em linux (Ableson et al., 2012). Há ainda uma grande quantidade de elementos gráficos que podem ser utilizados na elaboração de interfaces gráficas ricas e funcionais, além de um suporte multimídia, *framework*s de aplicações, entre outros.

2.3 FRAMEWORK

Segundo (Deutsch, 1989), o *design* de interface e a construção de software de maneira funcional é a chave da propriedade intelectual de um *software* e é bem mais difícil de ser criada que o próprio código escrito. Essa afirmação é corroborada por (Freeman et al., 2004), que afirma ainda que um problema qualquer de *design* de interface de aplicação, seja ele qual for, provavelmente já foi resolvido por alguém. É comum alguém procurar soluções de um problema do tipo por terceiros, afim de poder utilizá-lo para sanar a situação e economizar tempo de desenvolvimento. Dessa forma, padronizar códigos escritos e disponibilizá-los para outros programadores podem fazer com que eles não precisem pensar no *design* de interface. Um *framework* pode ser definido como *design*s de *software* reutilizáveis em toda uma aplicação ou parte dela. Um bom *framework* pode reduzir o custo do desenvolvimento de uma aplicação referente à quantidade de código escrito, pois permite reutilizar um código já existente (Johnson, 1997).

Atualmente, existem uma grande variedade de *framework*s para

desenvolvimentos de *software*, dependendo da tecnologia utilizada e do propósito. Para desenvolvimento de sistemas multiagentes, o Jade e o Jason são exemplos de ferramentas que podem ser utilizadas como *framework*, conforme citado anteriormente. O Jade (Java Agent DEvelopment *framework*) é um *framework* desenvolvido em Java que possibilita a criação de aplicações multiagentes em ambiente fixo ou móvel baseados em uma abordagem autônoma inteligente de agentes peer-to-peers (Bellifemine, et al., 2005). Ele disponibiliza um conjunto de classes auxiliares que podem ser utilizadas para implementar agentes inteligentes de forma independente do restante da aplicação. Os agentes desenvolvidos utilizando Jade são baseados em comportamentos,_que devem ser especificados pelo desenvolvedor. A execução dos comportamentos e o ciclo de vida da aplicação é gerenciado pelo resto do código existente no *framework*. O Jason é um *framework* utilizado para desenvolvimento de aplicações multiagentes BDI (*belief* - *desire* – *intention*). O modelo BDI é um dos modelos de raciocínio de agentes mais estudados e utilizados no mundo (Georgeff et. al., 1999). Esse modelo foi inspirado e baseado por filósofos no comportamento humano, descrevendo esse comportamento composto por um conjunto de crenças, desejos e intenções (Bordini, 2004). O *framework* utiliza uma linguagem denominada agent speak para determinar as crenças, desejos e intenções que compõem o comportamento dos agentes da aplicação. O trabalho realizado pelo usuário para desenvolver uma aplicação utilizando esses *framework*s é essencialmente herdar algumas classes dele e implementar interfaces e métodos.

3 O FRAMEWORK PIABA

O batismo desse *framework* provém provém de um peixe de mesmo nome que possui como características principais a simplicidade e eficiência: um peixe de pequeno porte e boca também pequena, com dentes fortes, de hábitos simples e de movimentação ágil, que precisa de poucos recursos alimentares em relação a outros peixes para sua perfeita sobrevivência.

O Piaba é um *framework* que possibilita a criação de sistemas que utilizam agentes inteligentes no Android, através da modelagem de agentes e do mundo via arquivos XML e código escrito em linguagem Java. Sua utilização permite criar um programa simples e robusto com um ciclo de vida bem definido, deixando o desenvolvedor responsável pela modelagem dos componentes da aplicação e a

implementação de como os agentes devem se comportar no ambiente em que estão localizados. Os agentes possuem a capacidade de perceber o ambiente e de interagir com ele, modificando seu estado.

As classes do *framework* foram desenvolvidas visando um código final limpo e coeso. Um código limpo pode ser definido como um código simples e direto, repleto de abstrações claras e linhas de controle objetivas (Booch, 2007). Essas características podem ser percebidas no *framework* Piaba. Algumas partes de um sistema multiagente, por exemplo, como o conhecimento bem definido, as estruturas dos agentes e do ambiente onde eles executam e a comunicação estão bem expressos em abstrações da ferramenta. Cada parte citada corresponde a uma classe ou uma composição de classes que foi elaborada para aquele fim. A Figura 3 mostra as relações entre os agentes e o ambiente da aplicação.

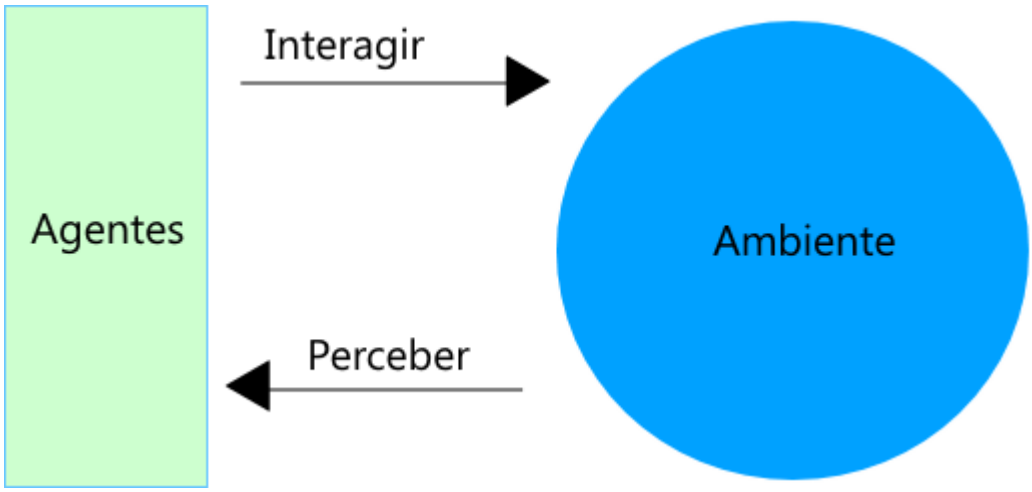

Figura 2: Elementos que compõem uma aplicação Piaba.

O ambiente é denominado mundo e é representado como um conjunto de características que representam seu estado atual. A compreensão dessas características são denominadas percepções, atualizadas automaticamente nos agentes a cada ciclo de execução, fazendo com que eles possam entender cada mudança do mundo. O conjunto de percepções dos agentes é chamado de base de crenças, e é composta pelas percepções que os agentes obtém do mundo somado às suas percepções próprias, que são modeladas nos arquivos XML. As crenças do agente referentes às percepções do mundo são atualizadas sempre que elas mudam no ambiente, mas as percepções próprias são atualizadas apenas manualmente pelos próprios agentes.

A interface gráfica é implementada de forma independente ao resto da aplicação. Ela é executada em paralelo e é atualizada a cada ciclo de execução da aplicação. O *framework* suporta também interações entre a interface gráfica da aplicação com o usuário, de forma a transmitir essas interações ao resto da aplicação. Por exemplo, um usuário pode dar ordens para que um agente faça alguma tarefa através de toques na tela do aparelho e essa informação pode ser transmitida a ele.

A separação das classes do *framework* de acordo com o seu objetivo ocasiona em uma maior coesão do código escrito pelo desenvolvedor. Isso se deve ao fato de o código relacionado ao que o agente faz ser restritivamente escrito em uma classe específica para ele, permitindo o desenvolvedor modularizar ainda mais o programa escrito em classes auxiliares e realizar uma composição final onde a classe principal constitui o agente. O mesmo acontece como o ambiente de execução dos agentes, as ações que os agentes executam, os operadores de alteração de estado afetivo, entre outros.

O desenvolvimento do *framework* foi realizado através da utilização de metodologias ágeis baseadas em entregas. Essas metodologias focam a produtividade e qualidade das funcionalidades entregues em cada período de desenvolvimento, alem de facilitarem no planejamento e controle do projeto (Amaral et al., 2011; Cohn 2011). Uma quantidade de dias foi determinada para ser utilizada como período de desenvolvimento, chamado timebox. As timeboxes variaram de 15 a 30 dias, onde funcionalidades planejadas anteriormente eram desenvolvidas e entregues como prontas. Um período de 3 à 5 dias anteriores às timeboxes foi utilizado como período de planejamento para definição de quais funcionalidades deveriam ser entregues. Nesses períodos, houveram reuniões com o orientador para discutir o que tinha sido entregue e as funcionalidades ainda pendentes para a próxima timebox. Nas timeboxes deveriam ocorrer o desenvolvimento e testes das funcionalidades previstas. Cada funcionalidade a ser desenvolvida nas timeboxes deu origem a um item denominado backlog item.

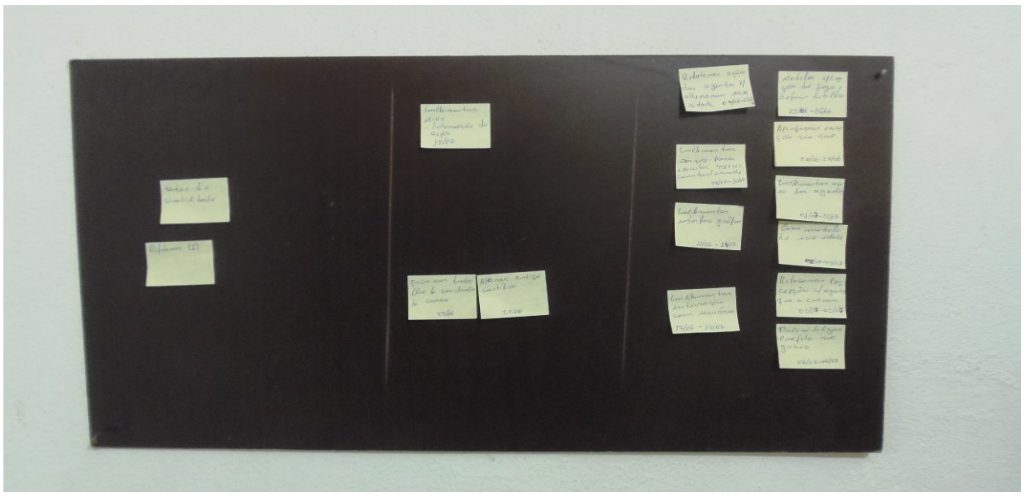

Figura 3: Quadro kanban utilizado no desenvolvimento

Os backlog items de uma timebox foram distribuídos em um quadro do tipo *kanbam* de três baias, a fazer, fazendo e pronto, respectivamente. A utilização do quadro kanbam permitiu a otimização do acompanhamento do desenvolvimento de cada timebox através da visualização dos backlog items distribuídos entre as baias do quadro, sendo possível observar o andamento do desenvolvimento das funcionalidades e permitindo a tomada de decisões em casos de previsão de possíveis atrasos e adiantamentos na entrega. A Figura 1 mostra o quadro de três baias utilizada. A baia esquerda foi utilizada para colocar as tarefas pendentes da timebox, enquanto a baia do meio continha as tarefas em andamento. As tarefas eram representadas por pequenos papéis de cor amarela que continham uma breve descrição da tarefa e a data inicial de sua execução. Ao finalizar uma tarefa, a data final era inserida no papel e o mesmo era transicionado para a baia direita.

3.1 INICIALIZAÇÃO

Uma aplicação que utiliza o *framework* Piaba é composta por dois arquivos de configurações XML. Um deles (MASWorld.xml) descreve o mundo a ser percebido como um conjunto de informações. Nele, além de especificada uma classe que será instanciada para representar o mundo que os agentes perceberão e executarão ações, serão especificadas também as percepções que representam ele. Essas percepções poderão atualizar a base de crenças dos agentes e utilizadas em processamentos posteriores. O outro arquivo (MASAgents.xml) descreve todos os agentes que serão utilizados na execução. Esses agentes são representados por classes que devem ser escritas pelo desenvolvedor. A cada agente definido no arquivo XML deve-se associar também um nome além do conjunto de percepções

iniciais. A Figura 4 mostra a relação dos agentes e do ambiente com seus respectivos arquivos de configuração.

Figura 4: Arquivos XML relacionados aos componentes principais de uma aplicação Piaba

A aplicação é iniciada através da execução de uma classe controladora denominada SystemController. Quando executada, o *framework* lerá os arquivo XML das configurações referentes ao mundo e agentes e carregará as propriedades necessárias para criar as instâncias dos objetos. Entre as propriedades do mundo está a que especifica a forma como os agentes serão executados. Essa execução pode ser síncrona, onde cada agente por vez percebe o mundo e realiza o processamento necessário, ou assíncrona, onde todos os agentes percebem o mundo e realizam suas ações de forma independente. Após a inicialização da aplicação, uma instância de SystemController estará disponível para o desenvolvedor.

3.2 CICLO DE VIDA

O ciclo de vida de uma aplicação Piaba compreende a execução das ações do mundo e dos agentes. A execução da aplicação é dividida em passos de execução, que por sua vez é composta por ciclos de execução. Cada ciclo é iniciado com a execução do mundo, que por sua vez ativa o módulo executor dos agentes. O módulo executor é responsável por executar a lista de agentes obtida na inicialização da aplicação. Em seguida, o agente pode tentar realizar ações no ambiente e alterar seu estado. A execução ou não dessas ações é avaliada pelo mundo, e caso as condições para execução sejam satisfeitas elas são executadas e o estado do mundo é alterado. Finalmente cada agente perceberá o mundo

novamente e atualizará sua base de crenças. Ao fim de cada ciclo de execução, as informações de execução ou não das ações de cada agente são atualizadas para que, no próximo ciclo, o agente saiba quais ações conseguiu executar. A Figura 5 mostra os passos essenciais de um ciclo de execução.

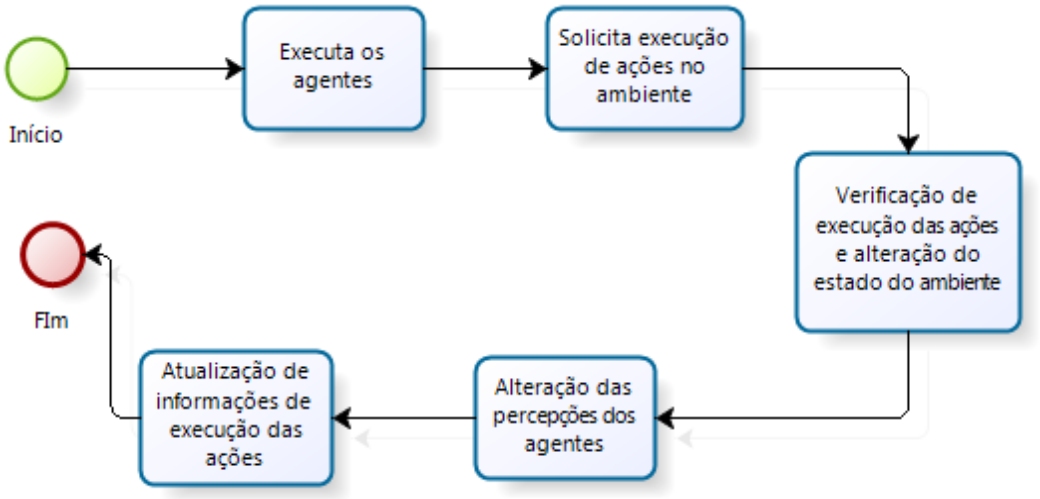

Figura 5: Um ciclo de execução

Existirão casos em que a execução de alguma ação não poderá ser realizada. Essa verificação é realizada por meio da análise do estado do mundo no momento da tentativa de execução. Por exemplo, se um agente *motorista* percebe o ambiente *estacionamento* e identifica uma vaga disponível, ele tentará executar a ação *estacionar* naquela vaga. Antes da ação ser executada, outro agente motorista poderia estacionar na mesma vaga, tornando-a ocupada. Dessa forma, a ação do primeiro agente *motorista* não poderá ser executada, pois no momento da sua execução o estado do ambiente define a vaga como ocupada. As informações de execução bem sucedida das ações de cada agente podem ser acessadas por eles no próximo ciclo e ser utilizadas, por exemplo, para aumentar o nível de uma percepção própria do agente referente ao seu nível de *estresse*. Cada agente tem acesso somente às informações de execução de ações solicitadas por ele no ciclo de execução anterior.

Um passo de execução pode ser realizado manualmente pelo desenvolvedor através da chamada ao método step() da instância de SystemController, disponível após a inicialização. Isso permite que o desenvolvedor possa controlar a realização dos passos de execução da aplicação. O desenvolvedor pode, por exemplo,

acompanhar a execução passo a passo da aplicação para fins de *debug*.

3.3 IMPLEMENTAÇÃO

Para implementar uma aplicação Piaba, o programador deverá codificar essencialmente os agentes e o mundo. Para cada tipo de agente, deve ser codificada uma classe que herdará de outra classe do *framework*, denominada GenericAgent. O método executeAgent da superclasse precisa ser sobrescrito para que seja criado o código responsável pela análise das crenças do agente, que podem ser acessadas através de uma instância da classe PerceptUtil (discutida adiante) e realizar ações baseadas nelas. O modelo de classes da abstração do agente é representado através da Figura 6.

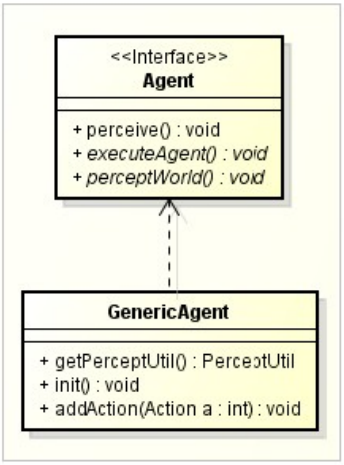

Figura 6: Principais métodos da classe GenericAgent e de sua interface

Os agentes podem tentar executar ações para alterarem o estado do mundo em situações específicas. Essas ações são representadas por instâncias de classes específicas para esse fim. Para a criação de classes desse tipo, o desenvolvedor precisará implementar uma classe que herde de outra contida no *framework*, chamada InternalAction.

A memória dos agentes da ferramenta é formada pelas informações das ações que executaram ou não e suas percepções, que são instâncias de uma classe do *framework* denominada *Percept.* A base de crenças dos agentes representam o entendimento das suas próprias características e do ambiente em que estão sendo executados.

InternalAction é uma classe genérica que implementa uma interface Action. Essa, por sua vez, descreve dois métodos que devem ser implementados pelo programador ao desenvolver uma ação interna em uma aplicação Piaba:

- *boolean evaluate(PerceptionUtil bb)* – Utilizado para verificar se uma ação

deve ser executada. Percepções do agente contida em sua base de crenças podem ser utilizadas para a decisão de execução ou não de uma ação. A superclasse InternalAction possui uma implementação padrão que retorna sempre o valor verdadeiro;

- *List<Percept> execute(PerceptionUtil bb)* – esse método realiza a ação do agente e retorna uma lista de percepções que representam a parte do mundo que a ação alterou. Por exemplo, uma ação *sentar* pode alterar o estado de uma cadeira do ambiente para *ocupada*.

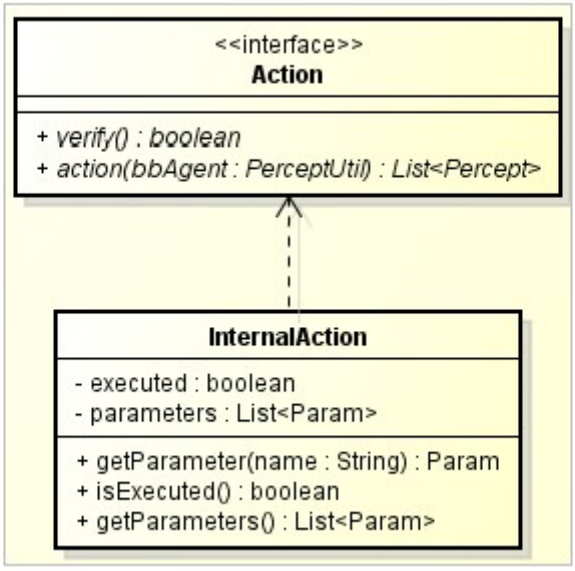

Figura 7: Principais atributos e métodos de uma InternalAction e sua interface

As ações que os agentes tentam executar no ambiente devem ser criadas e adicionadas na própria implementação do agente, no método executeAgent(). Cada agente possui uma lista de ações que pode ser manipulada na lógica do agente para adicionar ações específicas para cada ciclo de execução. As ações que um agente tenta executar em um ciclo *x* estarão disponíveis no ciclo *x+1*, bem como a informação que indica se a ação foi ou não executada (retorno do método evaluate()). O desenvolvedor pode optar por utilizar essa informação para influenciar as decisões que o agente toma em suas execuções.

A parte da aplicação referente ao mundo, deve ser composta por uma classe que herde de outra classe interna do *framework*, denominada GenericWorld. O principal método é o action, que deve ser sobrescrito pela subclasse para realizar o processamento das percepções do mundo e verificar a possibilidade de executar as ações de cada agente da aplicação. O método recebe uma instância de Agent, uma

interface implementada por GenericAgent, como parâmetro. Ele representa o agente que tentará executar as ações no mundo. O outro parâmetro do método é action, que especifica a ação que tentará ser executada. O código referente ao método executeAgent deve conter a lógica de processamento do mundo para verificar se é possível a execução das ações e, em caso positivo, realizar as alterações no estado do ambiente se necessário. A figura 8 mostra o diagrama de classes para a abstração do ambiente no framework Piaba.

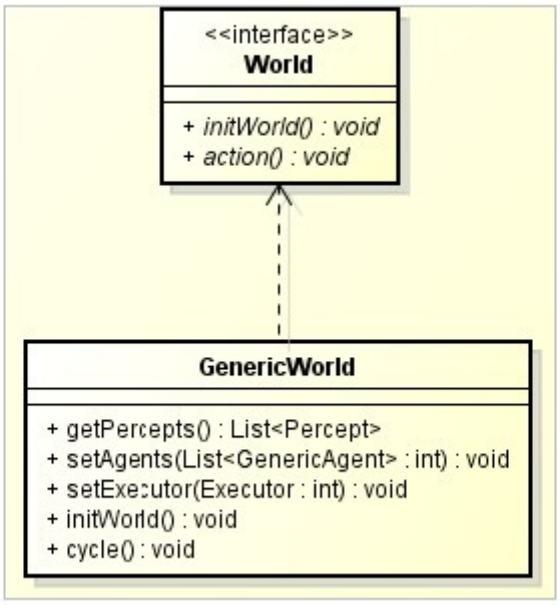

Figura 8: Modelo de classes genéricas de um ambiente

3.3.1 Executores

O *framework* fornece algumas implementações de executores que realizam a execução da aplicação de modo diferente. Dois tipos de execução estão disponíveis: A síncrona e a assíncrona. A configuração de qual tipo de execução de agente a aplicação deve utilizar deve ser realizada no arquivo MASWorld.xml, através da definição do valor da tag agent-executor. Cada abordagem de execução dos agentes influenciam na forma como ocorre o processamento de cada agente. A execução dos agentes é acionada sempre após o mundo realizar a execução das ações dos agentes, conforme descrito no item *ciclo de vida* deste trabalho. Essa característica pode alterar o resultado final das ações e consequentemente o comportamento da aplicação. Na execução síncrona, os agentes percebem o mundo e realizam seus processamentos um após o outro. Um novo agente só executa sua ação após o anterior finalizar suas ações. Dessa forma, o executor

executa a lista de agentes até que todos tenham executado e por fim envia ao mundo a lista de ações resultantes da execução de cada agente.

A execução de uma aplicação multiagente onde os agentes realizam consultas e ações que alteram informações em um banco de dados pode resultar em saídas diferentes após a execução de agentes síncronos em relação aos agentes assíncronos. No primeiro caso, como cada execução do agente altera o estado do banco de dados, a execução do agente seguinte pode ser influenciada pela execução do agente anterior. No segundo caso, a execução de um agente só afetará a execução de outros agentes se o momento da alteração do banco de dados de um agente A for anterior ao momento da leitura dos dados de um agente B.

A execução síncrona pode ser configurada definindo o valor *SYNCHRONOUS* na tag de configuração do arquivo XML. Há ainda um suporte a execução dos agentes em ordem, sendo essa configuração realizada através da tag execution-order no XML de definição de cada agentes. Caso haja a necessidade dessa execução, todos os agentes devem ter suas ordens de execução configuradas e diferentes. Não há garantias de qual agente será executado primeiro se um ou mais agentes estiverem com o mesmo valor de ordem de execução.

Em uma simulação onde necessita-se que cada agente seja executado em paralelo, é preferível que a execução seja assíncrona. O desenvolvedor pode desejar criar uma aplicação onde os agentes fazem buscas sem alterar o estado do local onde a busca é realizada. Dessa forma, não há influência da execução de um agente na execução de outros agentes pois o estado percebido por cada um deles não é influenciado por agentes executados anteriormente. Nesses casos, a execução dos agentes de forma assíncrona pode ser uma boa alternativa para diminuir o tempo de execução de cada ciclo. Nesse tipo de execução, a aplicação utilizará uma *thread* para execução de cada agente. O executor assíncrono não possui controle das *thread*s dos agentes após o início da execução delas, mas é possível identificar quando todos os agentes finalizarem suas execuções do ciclo. Nesse momento, a aplicação prossegue a execução do restante da aplicação conforme seu ciclo de vida. Esse tipo de execução pode ser configurada definindo o valor *ASSYNCHRONOUS* na tag de configuração do arquivo XML.

3.3.2 Comunicação

Uma das principais características dos agentes inteligentes é a capacidade

de comunicação entre eles (Bordini, 2004). Através do *framework* Piaba, os agentes podem realizar troca de mensagens entre si para comunicação de informações importantes. Cada agente possui uma caixa de entrada e uma caixa de saída, por onde ocorre o envio e o recebimento de mensagens respectivamente. Apesar desse recurso, o *framework* não implementa um protocolo específico.

A leitura dessas mensagens pode ser realizada no código do próprio agente, através de métodos auxiliares que o *framework* fornece. A leitura pode ser feita no momento em quê o agente é executado, uma vez a cada ciclo de execução. Uma vez aberta, a mensagem é marcada como lida e será descartada, impossibilitando sua leitura em outra ocasião. Além da leitura, o agente pode criar uma mensagem caso deseje se comunicar com outro agente. Essa mensagem deve conter o conteúdo e o identificador do agente de destino. O próprio *framework* realiza a manipulação das mensagens, retirando da caixa de saída do emissor e colocando na caixa de entrada do destinatário.

Há ainda suporte à envio de mensagens em broadcast, sem necessidade de especificar o agente destinatário. Essa funcionalidade pode ser útil em simulações onde os agentes possuem o mesmo objetivo. Por exemplo, um agente minerador pode avisar a outros agentes mineradores que encontrou minério ao passo que os demais, ao receberem a mensagem, dirigem-se para a localização do emissor afim de realizar a extração. Essa abordagem também é bastante utilizada em jogos de estratégia que envolvem extração de recursos do ambiente e ataque a bases inimigas (Sandberg et al., 2011). Ao detectar uma ameaça em ataque, um agente camponês pode enviar uma mensagem de socorro a um agente como pedido de ajuda.

3.3.3 Integração com Interface Gráfica

As interfaces gráficas no Android são implementadas em atividades ou Activities. Atividades são trechos de código executável que vão e voltam no tempo, instanciadas pelo usuário ou pelo sistema operacional e executadas enquanto forem necessárias. Elas podem interagir com usuários e com outras atividades (Rogers, 2009).

O *framework* Piaba suporta interações com interfaces gráficas independentes, criadas pelo próprio desenvolvedor. Para tal, a Activity que a representa deve herdar uma classe abstrata do próprio *framework*, denominada PiabaActivity. Através dessa classe, o programador tem acesso à instância do

SystemController, que pode ser utilizado para realizar execuções da aplicação.

A atualização automática da interface gráfica pode ser realizada através de uma classe do *framework* chamada GenericCycleUpdateGUI. Ao herdar essa classe, o desenvolvedor deve implementar o método *run(),* escrevendo o código que realiza a atualização da interface gráfica em um ciclo de execução. É possível, por exemplo, obter a instância da *Activity* que está em execução e alterar seus elementos gráficos ou obter seus valores. A classe GenericCycleUpdateGUI implementa a interface Runnable, permitindo sua utilização em uma *thread*, tornando sua execução assíncrona com a interface gráfica. No exemplo descrito nesse trabalho, essa classe é utilizada através de um *service* que, após cada ciclo de execução, cria uma *thread* com uma instância da classe de atualização da interface gráfica para que ela possa ser executada e alterar a *Activity* principal.

3.4 ATUALIZAÇÃO DE PERCEPÇÕES

As percepções, que originam crenças nos agentes, são compostas por um identificador e um valor. O agente possui em sua base de crenças dois tipos básicos de crenças:

- crenças próprias: configuradas em cada agente no arquivo de configuração XML;
- crenças de percepção: adquiridas pelo processo de percepção do mundo;

A cada ciclo de execução, os agentes devem perceber novamente o mundo e atualizar suas crenças. Esse processo é necessário para que os agentes possam identificar as mudanças do mundo e reagir a elas. Antes da execução de cada agente, este obtém uma lista de percepções que constitui o estado atual do mundo. Esse conjunto de percepções é comparado com o conjunto de percepções interno do agente para que seja feita a atualização.

A atualização de uma crença percebida do mundo ocorre de três formas diferente. Se a percepção vinda do mundo já existir na base de crença do agente, ela é preservada mas seu valor é atualizado. Caso a crença vinda do mundo ainda não exista na base de crenças do agente, ela é adicionada. Por ultimo, caso alguma crença da base de crenças do agente não exista na lista de percepções obtidas do mundo, ela é removida.

Esse processo de atualização incide somente nas percepções da base de

crenças do agente provenientes do mundo. As percepções próprias dos agentes são restritas, e sua atualização depende somente do próprio agente, podendo ele atualizar seus valores, removê-las ou até mesmo criá-las apenas no seu próprio processamento. Essa manipulação deve ficar sob a responsabilidade da implementação dos agentes da aplicação.

3.5 AGENTES COM FOCO AFETIVO

Recentemente, cientistas chegaram a conclusão de que emoção e conhecimento são processos humanos complementares intimamente relacionados (Clore, Palmer, 2009). É fácil percebermos isso ao observar o comportamento humano diariamente. A capacidade de um ser humano de adquirir e processar o conhecimento adquirido em um determinado momento é geralmente afetada pelo estado emocional em quê o mesmo se encontra. Níveis diferentes de *estresse*, ansiedade e felicidade, entre outras emoções, podem afetar as decisões que um indivíduo toma no seu dia a dia diante de determinadas situações. Por exemplo, um funcionário de uma empresa pode, através de altos níveis de *estresse*, ter problemas no desempenho de suas atividades. O oposto também é verdadeiro: o desempenho das atividades do funcionário somados à fatores externos podem afetar no nível de *estresse* do indivíduo (Pellegrini et al., 2010).

Segundo (Signoretti 2012), a aplicação de um modelo de atenção de agentes baseada na própria afetividade humana pode ajudá-los a tomarem decisões melhores em determinadas situações. Dessa forma, a utilização de um modelo afetivo como parte do perfil de identificação dos agentes influencia diretamente em seus comportamentos.

O Piaba implementa um modelo genérico de execução que permite ao desenvolvedor criar aplicações usando um foco de atenção afetivo nos agentes. Esse modelo é opcional, sendo portanto uma decisão do desenvolvedor utilizá-las e, independente dessa decisão, todos os outros recursos continuam disponíveis. Caso o desenvolvedor não deseje utilizar essa funcionalidade, basta apenas não implementar ou configurar os operadores emocionais na aplicação.

A forma como as emoções afetam as decisões dos agentes bem como o estado afetivo que deve influenciar nelas devem ser definidos pelo desenvolvedor da aplicação. Os parâmetros de afetividade de cada agente podem ser configurados na própria definição do arquivo *XML* que configura o agente, como percepções próprias. Cada agente terá então um conjunto de emoções, compostas por um valor

referente ao nível de intensidade de cada uma delas e a denominação das mesmas.

O modelo OCC (Steunebrink et al., 2009) mostrado de forma hierárquica na Figura 7 é um exemplo de modelo que poderia ser implementado pelo desenvolvedor para utilização da afetividade nos agentes de uma aplicação que utilize o *framework* Piaba. Todas as emoções descritas nesse modelo estão implementadas na ferramenta para utilização.

Esse modelo descreve uma hierarquia que classifica 22 tipos de emoções. Essa hierarquia é divida em três ramos, classificando as emoções de acordo com as consequências de eventos (*joy* e *pity,* ou alegria e pena), de acordo com as ações dos agentes (*pride* e *reproach*, ou orgulho e vergonha) e aspectos do objeto (*love* e *hate*, ou amor e ódio). Os ramos das emoções podem ser combinados para criar um grupo de emoções compostas relativas à consequência de eventos causados por ações dos agentes (*gratitude* e *anger*, ou gratidão e raiva). Essas noções são comumente utilizados em modelos de agentes, tornando o modelo OCC uma opção vantajosa para utilização em agentes inteligentes.

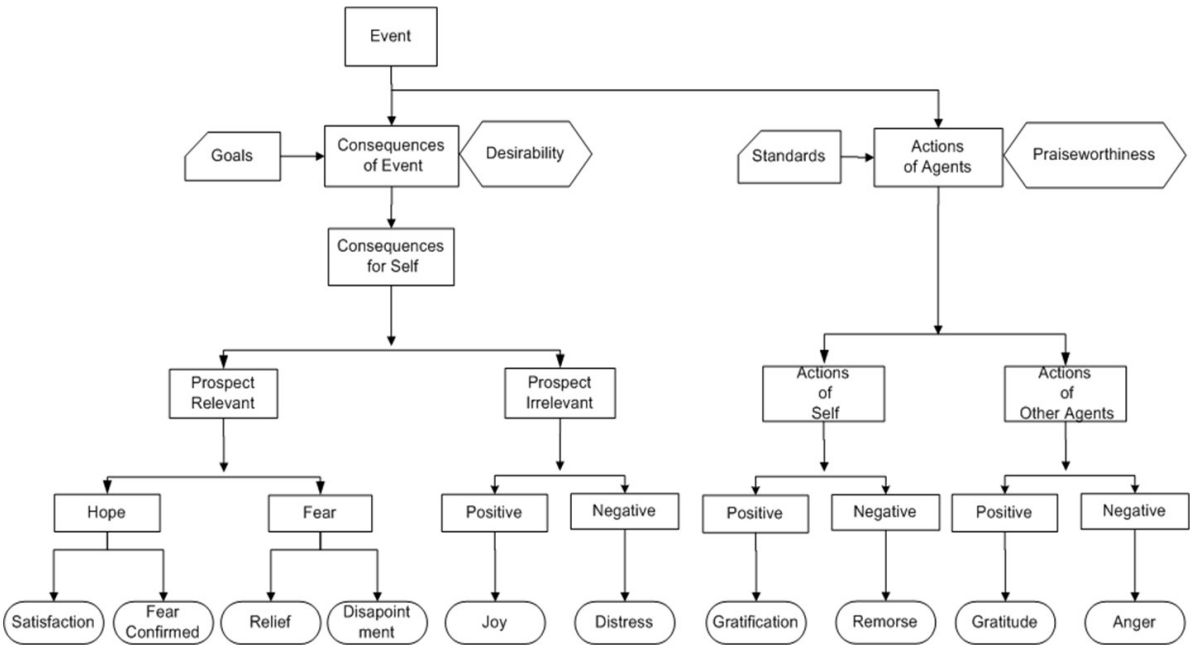

Figura 9: Modelo OCC, baseado em (Signoretti, 2013)

As emoções estão disponíveis na base de crenças do agente e podem ser obtidas pelo desenvolvedor para utilizá-las como parâmetro adicional que pode influenciar nas decisões dos agentes em uma aplicação Piaba. A forma como as emoções serão utilizadas para esse fim deve ser definido pelo desenvolvedor, no

código que executa o agente ou na execução de suas ações no ambiente. Isso deixa o desenvolvedor livre para definir o modelo afetivo que deseja trabalhar em seus agentes.

Durante a execução da aplicação, as emoções dos agentes podem ser alteradas ciclo a ciclo. Essa alteração ocorre por meio de MIOs (sigla para Meta-Info Operator). O conceito de MIO também é definido no trabalho de Signoretti (Signoretti, 2012) como um *gatilho de informação* responsável por tratar os eventos relevantes como uma descrição de um conjunto de informações percebidas do ambiente. Ele compõe um submódulo da aplicação que analisa as crenças do agente para entender o que acontece ao redor dele a fim de redefinir as configurações do foco de atenção do agente e seu estado afetivo.

Há um arquivo XML denominado MASMios.xml para definição das MIOs. Nesse arquivo, devem ser definidas as classes que representam cada MIO e as condições de afetividade do agente para execução. As condições são definidas como níveis de emoção do agente. Por exemplo, a MIO aumentarFelicidade pode ser executada caso ocorra algum acontecimento positivo no mundo.

As classes que representam as MIOs devem herdar de uma classe genérica do *framework* denominada MioAction, e dois métodos devem ser sobrescritos:

- *boolean evaluate(PerceptionUtil bb)* – utilizado para verificar se a MIO deve ser executada. Como comentado anteriormente, a MIO verifica através desse método o conjunto de crenças do agente para analisar o estado afetivo do agente. A superclasse MioAction possui uma implementação padrão desse método que retorna sempre valor verdadeiro, sendo a sobrescrita dele *opcional*;
- *void execute(PerceptionUtil bb)* – contém o código fonte da execução da MIO, bem como o cálculo de decaimento ou acréscimo de emoções desse agente. Não há implementação padrão na superclasse MioAction e a sobrescrita dele é obrigatória;

O processo de execução das MIOs é gerenciado pelo *framework*, permitindo que o desenvolvedor foque seus esforços apenas em implementar *quando* uma MIO deve ser executada e o quê cada MIO faz. A Figura 10 mostra o modelo de classes de uma MIO no *framework* Piaba.

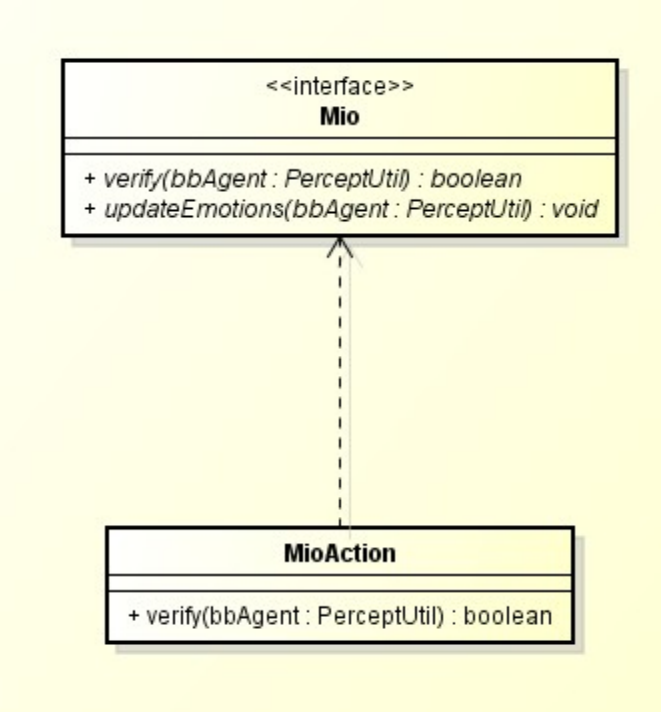

Figura 10: Estrutura de uma Mio do Piaba

Em seu trabalho, Signoretti (Signoretti, 2012) desenvolveu um programa de teste contendo uma sala com saídas e sinais de incêndio e um foco de incêndio incremental. A cada ciclo de execução, os agentes analisavam as percepções do mundo e atualizavam a sua base de crenças, ao passo que MIOs quando executadas, alteravam o estado afetivo dos agentes. Esses níveis influenciavam diretamente na decisão dos agentes de se moverem para outro lugar ou permanecerem no mesmo, afim de manterem-se vivos e dirigirem-se o quanto antes à saída de incêndio mais próxima.

A execução e a implementação das MIOs são parecidos com as WorldActions da aplicação: ambas utilizam um estado específico do agente para serem executadas. As principais diferenças entre elas estão no objetivo de cada uma: enquanto a primeira é utilizada para alterar o estado afetivo do agente, a segunda é utilizada para alterar as percepções do mesmo. Além disso, as MIOs são configuradas em um arquivo XML de configuração e sua execução é analisada pelo próprio *framework* automaticamente no ciclo de vida da aplicação, enquanto a execução das WorldActions são solicitadas pelos próprios agentes na sua lógica de execução.

4 COMPARAÇÃO COM OUTROS FRAMEWORKS MULTIA GENTES

Alguns conceitos abordados e alguns modelos implementados para execução das funcionalidades do Piaba, como a comunicação dos agentes e a execução desses, são semelhantes à outros existentes em outras ferramentas. As diferenças abordadas nesse trabalho entre o Piaba e dois outros *framework*s, o Jade-Android e o JaCa-Android, constituem vantagens e desvantagens dependendo da aplicação que deseja-se desenvolver, e devem ser analisados pelo desenvolvedor antes de iniciar o projeto.

4.1 PIABA X JADE-ANDROID

Um aplicação Jade-Android é dividida em duas partes partes. A saber:

- O container, onde os agentes são executados. Containers podem estar em locais físicos diferentes que se comunicam através de uma rede de computadores;
- Os agentes propriamente ditos;

Os agentes do *framework* Jade-Android baseiam-se em um comportamento explícito no código que pode ser executado através do método behaviour(), herdado da superclasse do *framework* Agent. Cada agente possui um identificador, atribuído automaticamente pela própria aplicação, e um nome. O desenvolvedor deve criar seus agentes herdando a classe Agent, implementando o comportamento deles e em seguida registrando no container (Caire et al., 2012). O container, entretanto, deve estar localizado em uma plataforma fixa (PC) para execução correta dos agentes. Na versão original do *framework*, ambos (agente e container) podem compartilhar o mesmo hardware.

A comunicação entre os agentes no Jade-Android, assim como no próprio Jade, é uma das principais funcionalidades da ferramenta (Bellifemine, et al., 2005). A troca de mensagem é baseada no envio de mensagem assíncrono, com cada agente possuindo uma caixa de correio para envio e recebimento de mensagem. Quando um agente deseja enviar a mensagem a outro agente, ele coloca uma mensagem endereçada ao destinatário em sua caixa de correio e a aplicação, por meio da comunicação entre os containers, entregam a mensagem na caixa de

correio do destinatário. Essa funcionalidade permite, graças à capacidade dos containers conseguirem se comunicar mesmo estando localizados em locais físicos diferentes, que um agente envie e receba mensagens de outro agente executando em um hardware diferente e distante. Há um suporte no *framework* para utilização de um protocolo de comunicação chamado ACL (agent communication language) definido pela FIPA, uma organização de padrões voltados para agentes inteligentes.

4.1.1 Diferenças

Enquanto o Jade-Android necessita da utilização de um container para execução dos agentes, o Piaba implementa todos os recursos necessários para a execução da aplicação em um único programa, sendo executável em sua totalidade na plataforma Android. O Piaba "liberta" o desenvolvedor da necessidade de um outro programa ser executado para o funcionamento e execução dos agentes. Entretanto, o Piaba não possui um suporte nativo à comunicação de agentes distribuídos em hardwares diferentes. Essa funcionalidade, caso necessária, deve ser implementada pelo próprio desenvolvedor com o uso de classes auxiliares para esse fim.

Ambas ferramentas suportam a comunicação entre agentes via troca de mensagens por caixa de correio, entretanto o Piaba não implementa um protocolo robusto como o ACL, implementado no Jade-Android. Nele, a troca de mensagens é mais simples: a cada ciclo, o *framework* verifica quais caixas de correio possuem mensagens a serem enviadas e transfere a mensagem de um agente para outro. Entretanto, não há casos de indisponibilidade para troca de mensagens entre os agentes, pois nativamente os agentes executam em um único local, enquanto o Jade-Android permite a execução em locais distintos.

O Jade-Android não implementa ambientes onde os agentes possam perceber ou realizar interações. Os agentes são apenas componentes de código que executam funções, sejam elas quais forem. O Piaba suporta nativamente a utilização de ambientes de execução, além de implementar um modelo de conhecimento que define o estado do mundo e dos agentes.

Cada agente do Jade possui sua própria *thread* de controle (Bellifemine, et al., 2005). Ao ser executado, o agente executa seu comportamento até que sua execução chegue ao fim, sem sincronia com outros agentes. O Piaba suporta diferentes tipos de execução dos agentes, síncrona e assíncrona, que podem ser

escolhidos pelo desenvolvedor através de configuração na aplicação. Entretanto cada agente executa um ciclo de execução por vez. Os agentes só executam um novo ciclo quando todos concluírem suas execuções.

Na Tabela 4.1.1.1 estão listadas algumas das diferenças entre os *framework*s.

	Jade-Android	**Piaba**
Ambiente	Sem ambiente nativo explicito.	Com ambiente nativo explícito.
Comunicação	Entre caixas de correios dos agentes, utilizando protocolo ACL.	Entre caixas de correios dos agentes sem um protocolo definido. Os agentes apenas solicitam e as mensagens são entregues ao destino.
Distribuição	Suporta distribuição entre agentes em locais físicos diferentes, através da comunicação de containers.	Não suporta nativamente a distribuição de agentes em locais físicos diferentes.
Execução	Restrita. Cada agente possui sua *thread* de controle.	Configurável. O desenvolvedor pode configurar as execuções entre as opções síncronas e assíncronas.
Tecnologia Utilizada	Java.	Java e XML
Agentes Afetivos	Sem nenhum suporte nativo.	Suporte nativo à utilização e ateração de níveis afetivos dos agentes

Tabela 1: Diferenças entre o Jade-Android e o Piaba

4.2 PIABA X JaCa-JASON

O JaCa-Jason é o resultado da integração entre dois outros *framework*s: O Jason e o CartAgO, que é um *framework* para desenvolvimento de ambientes para execução dos agentes. O Jason implementa agentes bdi através da descrição dos planos e objetivos (goals) dos agentes da aplicação (Bordini, 2004). Nele, planos são definidos como ações que são realizadas pelos agentes para alcançar determinados objetivos (goals). Por exemplo, para que um agente possa dirigir um carro (objetivo) ele precisa da chave do carro. Caso a chave não seja encontrada, ele não poderá alcançar seu objetivo.

A utilização do CArtAgO foi escolhida como alternativa para desenvolvimento dos ambientes onde os agentes da ferramenta executam. O ambiente é constituído por um conjunto de artefatos utilizados como abstração básica para modelar os

objetos que os agentes podem perceber e interagir no ambiente (Ricci et al. 2006).

O *framework* Jason analisa, em seu ciclo de vida, quais objetivos os agentes possuem e quais ações precisam ser executadas para alcançá-los. Essa verificação é realizada a cada ciclo de raciocínio, conceito utilizado no *framework* para especificar a unidade de execução do agente (o agente executa uma vez a cada ciclo de raciocínio). Os agentes possuem acesso às informações de quais planos foram executados ou não. Objetivos alternativos podem ser traçados caso um objetivo não possa ser alcançado. Votando ao exemplo anterior, se um agente não conseguir ir a um destino dirigindo um carro (objetivo I) ele pode tentar fazê-lo de bicicleta (objetivo II). Todos os planos e objetivos dos agentes são configurados e implementados pelo desenvolvedor em um arquivo utilizando linguagem denominada AgentSpeak(L) (Rao 1996).

Cada agente possui uma base de crenças, compostas pelo conjunto de percepções que o mesmo obtém do mundo. O *framework* suporta nativamente um filtro de percepções para ser aplicado sempre que um agente observa um mundo. Dessa forma, um agente pode perceber somente o que lhe convém ou o que é possível. Agentes localizados em uma parte do ambiente podem ser configurados para não perceberem o quê está atrás dessa parede.

O Jason implementa vários tipos de ambientes, como ambiente em grid. Esse tipo de ambiente torna o mundo uma matriz de elementos. Essa matriz pode ser utilizada para localizar agentes e fazê-los se mover durante a execução.

4.2.1 Diferenças

Enquanto o JaCa-Android utiliza Java e AgentSpeak(L) para implementação, o Piaba utiliza Java e XML. O Piaba não utiliza a abordagem de objetivos e planos configuráveis, e nele as ações que o agente deseja realizar é implementada manualmente pelo desenvolvedor e a verificação de condições para execução é realizada automaticamente pelo sistema. No Piaba, os agentes tentam executar ações no mundo enquanto no Jason essas ações são configuradas previamente através de AgentSpeal(L) como planos para um objetivo específico. Em ambos os casos, as ações só são realizadas se determinados pré-requisitos anteriormente estabelecidos forem satisfeitos. Ambas ferramentas possuem execução dos agentes baseados em ciclos e seus agentes possuem acesso às ações que foram executadas ou não no ciclo anterior.

No JaCa-Android, cada agente executa em uma *thread* de execução diferente e a execução de um agente pode alterar imediatamente o estado do mundo que será percebido pelo agente que executará após ele. No Piaba, a execução além de poder ser configurada como síncrona ou assíncrona, a percepção do estado do ambiente de um agente é o mesmo em um ciclo de vida para todos os agentes. Isso significa que, mesmo após a execução de um agente, as alterações do estado do ambiente que ele realiza através das suas ações só serão aplicados após todos os agentes serem executados.

O Piaba utiliza um ambiente bem definido, cujo estado é composto por informações que os agentes podem perceber. O JaCa-Android utiliza o conceito de artefatos como composição do ambiente. Os agentes utilizam esses artefatos para perceberem o ambiente e interagirem entre eles, enquanto no Piaba as interações são realizadas através de ações que os próprios agentes decidem tentar realizar ou não.

A noção de workspace é utilizada no JaCa-Android para definir a topologia dos ambientes, que podem ser organizados como um conjunto de sub-workspaces. Dessa forma, é possível distribuir essas pequenas partes em hardwares localizados em lugares diferentes e executar agentes em cada um deles (Santi et al. 2010). A Tabela 2 compara o JaCa-Android com o *framework* Piaba.

	JaCa-Android	Piaba
Ambiente	Com ambiente nativo explicito.	Com ambiente nativo explícito.
Comunicação	Realizada através de planos (o desenvolvedor deve especificar planos para que as ações de comunicação sejam realizadas)	Os agentes apenas solicitam e as mensagens são entregues ao destino.
Distribuição	Realizada através da divisão do ambiente em pequenas partes executadas em cada nó da rede	Não suporta nativamente a distribuição de agentes em locais físicos diferentes.
Execução	Restrita. Cada agente possui sua *thread* de controle.	Configurável. O desenvolvedor pode configurar as execuções entre as opções síncronas e assíncronas.
Tecnologia Utilizada	Java e AgentSpeak(L)	Java e XML
Agentes Afetivos	Sem nenhum suporte nativo.	Suporte nativo à utilização e ateração de níveis afetivos dos agentes

Tabela 2: Diferenças entre o Piaba e o JaCa-Android

5 CASO DE USO: ROBÔ ENTREGADOR DE CERVEJA

Um caso de uso foi implementado para utilizar e demonstrar o funcionamento das principais funcionalidades do *framework*. Trata-se de um jogo, que utiliza três agentes: dois robores e um agente humano. Durante a simulação de execução dos agentes, ações e MIOs são realizadas nos agentes e no ambiente onde estão localizados.

Os dois agentes robôs realizam diversas ações naturalmente enquanto estão em execução. São elas: varrer, descansar, colocar o lixo para fora e ler um livro. Cada ação demora cinco ciclos para completar sua execução, e após cada ação ser executada o agente torna-se ocioso por 3 ciclos. O agente humano apenas bebe cerveja. Conforme a cerveja é consumida, seu nível diminui até o nível zero. Quando isso acontece, o mesmo torna-se ocioso e esse estado ativa a execução de uma MIO que aumenta o *estresse* do agente humano. Cada ciclo sem beber significa a execução da MIO.

Após o nível de cerveja se esgotar, o usuário (jogador) pode *design*ar um

agente para entregar a cerveja através do toque na tela. Caso o usuário deseje que algum robô que esteja executando uma ação particular entregue a cerveja, o agente interrompe a execução da ação atual e executa a ação de entregar a cerveja. Em seguida, após a execução da ação que entrega a cerveja, a execução da ação interrompida é retomada. Caso um agente ocioso seja selecionado para entregar a ação, ele simplesmente o faz e em seguida retoma o tempo de ociosidade até executar uma ação particular.

A interrupção de uma ação do agente em execução bem como a continuidade da execução de uma ação constituem estados que servem de gatilho para a execução de uma MIO. Ao serem executadas, essas MIOs realizam um cálculo que considera o nível de *estresse* atual e o estado do agente para em seguida atualizar a emoção *estresse* do agente. A Figura 11 mostra a interface gráfica principal do jogo.

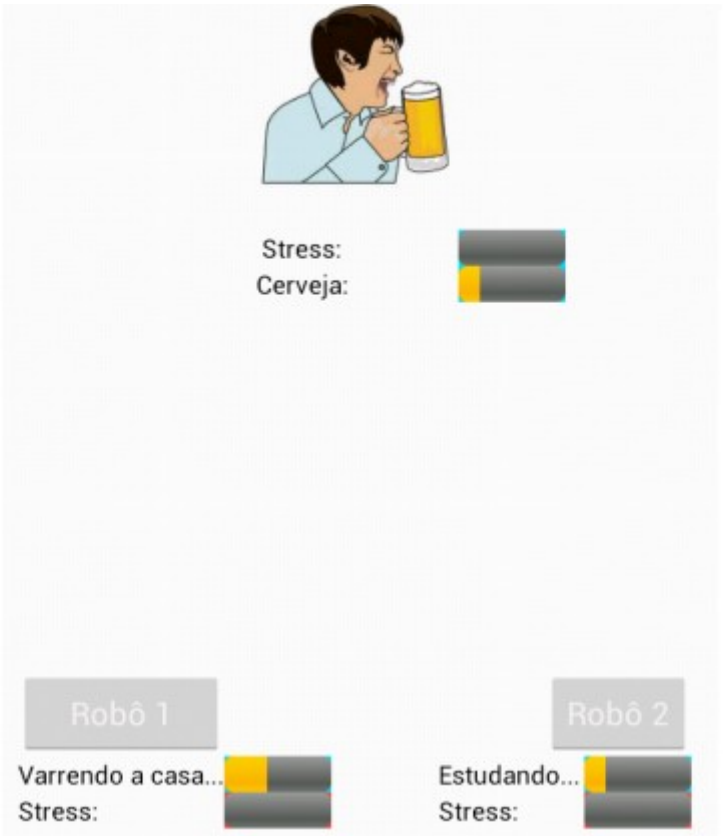

Figura 11: Interface Gráfica do Jogo. Os botões dos agentes podem ser acionados para ordenar qual robô deve servir o agente humano

O objetivo do jogo é manter o sistema em sincronia e execução com o mínimo de *estresse* possível em cada um dos agentes. O jogador deve tomar a decisão estratégica de qual agente deve entregar a cerveja, dando preferência ao

que estiver ocioso no momento ou mais próximo de finalizar a execução da ação atual, podendo ainda decidir que o agente humano espere durante algum tempo necessário para que um agente esteja disponível para servi-lo.

5.1 ESTRUTURA DA APLICAÇÂO

Cada tipo de agente é representado por uma classe que herda da classe do *framework* GenericAgent. O método principal dessa classe do *framework*, executeAgent(), é implementado com o quê o agente deve fazer. Métodos auxiliares, como verificar se uma ação está em execução ou não, foram implementados. Cada agente do jogo está mapeado no arquivo de configuração MASAgents.xml. O mesmo acontece com as ações que os agentes podem solicitar. Cada ação está implementada em uma classe, que herda WorldAction do *framework*. Elas possuem um método que verifica se a ação deve ser executada e um método que realiza a execução da ação, alterando as percepções dos agentes. A classe EntregarCerveja, por exemplo, só pode ser executada se o nível de cerveja estiver no fim (valor zero), e sua execução altera o nível de cerveja de zero para cem. A Figura 12 contém o modelo de classe dos agentes utilizados no jogo, enquanto a Figura 13 contém o modelo de classes das ações implementadas.

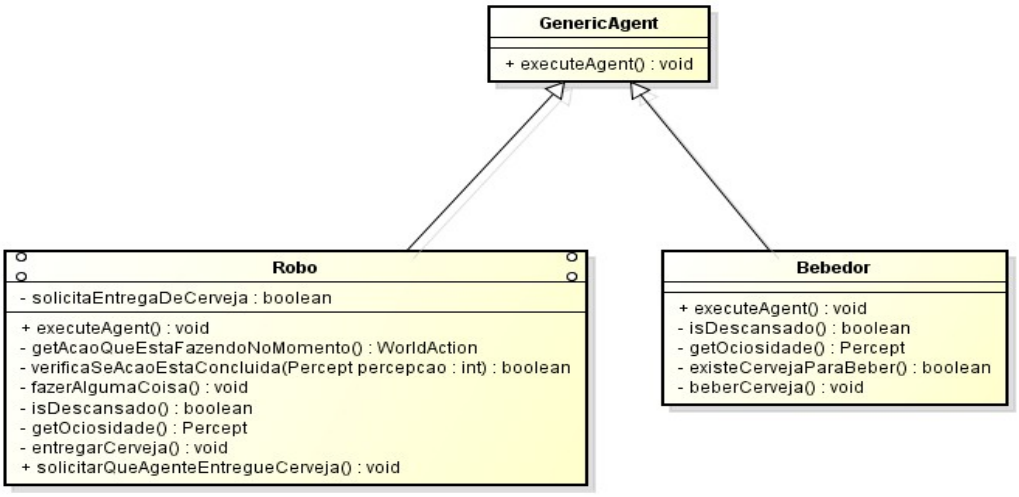

Figura 12: Modelo dos agentes, com seus principais métodos implementados

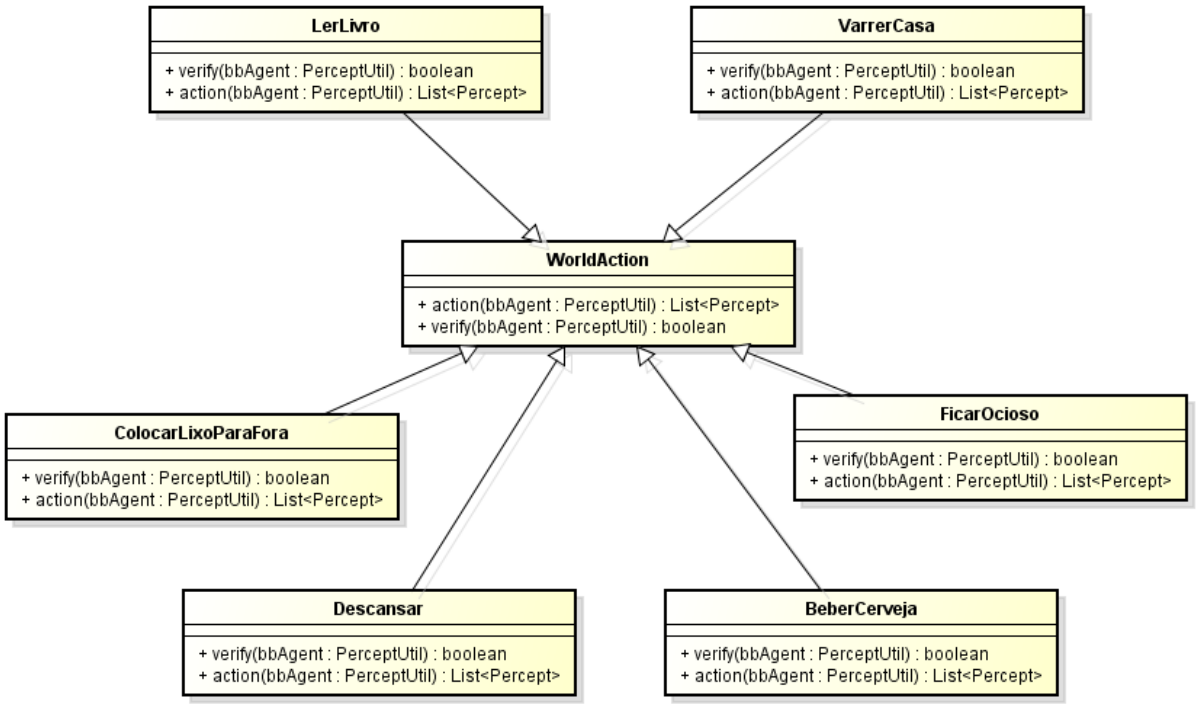

Figura 13: Modelo das ações dos agentes. Cada ação implementa o método descrito na classe abstrata WorldAction

Outros métodos foram implementados em cada classe dos agentes. A maioria desses métodos estão marcados com o modificador de acesso private, por serem utilizados apenas internamente na classe. A execução principal dos agentes é realizada através da chamada ao método executeAgent(). Dentro desse método, em cada classe do agente, chamadas a outros métodos auxiliares da própria classe é realizado de acordo com a necessidade.

Há duas MIOs no jogo: uma que executa quando o agente tem uma ação interrompida e outra que executa quando não há interrupções. A primeira aumenta o nível de estresse (emoção *distress*) em dez pontos quando executada, enquanto a segunda diminui o nível de estresse em um ponto. O nível de estresse pode ser aumentado até o máximo de cem pontos e diminuído a um mínimo de zero pontos. As MIOs foram definidas no arquivo XML de configuração MASMios.xml. A Figura 14 mostra as classes da abstração das MIOs utilizadas no jogo.

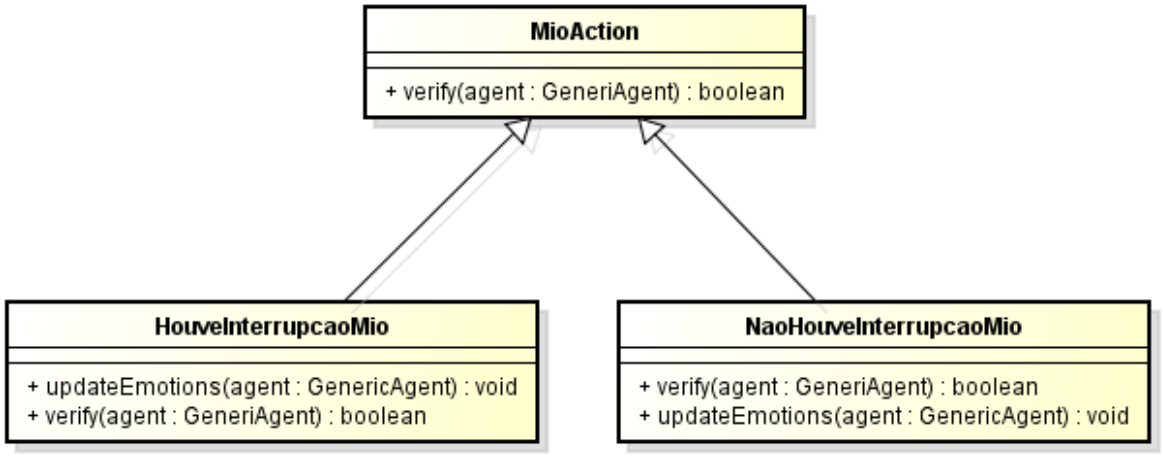

Figura 14: Modelo das MIOs da aplicação

Há um ambiente definido mas não há nenhuma customização para o jogo. O ambiente CervejaWorld é implementado como uma classe que herda GenericWorld do *framework* e mapeado no arquivo XML do mundo como ambiente de execução dos agentes. No mesmo arquivo XML estão definidas percepções iniciais do ambiente, como a quantidade de cerveja no início do jogo e a duração de execução de uma ação (cinco ciclos por ação). Nesse arquivo também foi configurada a execução dos agentes de forma síncrona, onde há execução de um agente por vez durante um ciclo de execução. Nele há também a configuração da classe que atualiza a interface gráfica a cada ciclo. A classe CervejaCicleUpdateGUI, quando executada, atualiza os elementos da interface gráfica de acordo com as percepções dos agentes e do ambiente.

Uma IntentService foi implementada para executar o jogo de forma assíncrona com a interface gráfica. A IntentService é um serviço no Android que permite executar parte de uma aplicação em background, de forma assíncrona com a interface gráfica. Isso é possível porque uma IntentService executa em sua própria *thread*, e não na mesma *thread* que a interface gráfica do Android (Gargenta, 2011) Isso permite que o jogador esteja livre para realizar interações com o jogo enquanto os agentes executam. A IntentService GameService foi implementada para disparar uma *thread* que executa um passo de execução a cada segundo, através da chamada ao método step() da instância de SystemController.

O agente robô deve dar prioridade à ação de entregar a cerveja ao agente humano. No início da sua execução, o agente verifica se houve a interação com o usuário de acionar o agente para que ele entregue a cerveja. Caso positivo o

mesmo adiciona a ação de entregar cerveja na sua lista de ações. Caso negativo, o agente deve verificar se alguma ação sua está em andamento. Caso a ação esteja em execução, ela continuará a ser executada até cinco ciclos sejam executados. Se o agente concluir a execução de uma ação, ele se torna ocioso por três ciclos e em seguida volta a executar uma ação aleatória.

O *framework* Piaba permitiu o desenvolvimento do jogo através da implementação de pouco código escrito, tornando o trabalho de codificação simples e pouco custoso se comparássemos com o trabalho de desenvolvimento sem uma ferramenta específica. Não foi necessário a utilização de nenhuma outra ferramenta para criar as funcionalidades do jogo. A interface gráfica isolada da camada de controle da aplicação, somada à classe de atualização dela, possibilitou uma melhor organização no código escrito e facilitou a manutenção do código durante o período de desenvolvimento. Ao fim da implementação, as classes criadas para o jogo possuíam um alto grau de coesão, ou seja, elas faziam apenas uma coisa e faziam bem.

Apesar do exemplo simples, as MIOs se mostraram eficazes na sua execução e na atualização das emoções de cada agente do jogo. O ciclo de vida bem definido da aplicação permitiu uma certa facilidade em momentos onde foi necessário depurar o código.

6 TRABALHOS FUTUROS

Baseado no trabalho desenvolvido, algumas vertentes de trabalhos futuros podem ser identificados. Como abordado no início desse trabalho, a ideia inicial do *framework* era criar uma versão estável para plataforma fixa (PC), porém no decorrer do desenvolvimento essa ideia foi modificada. O *framework* foi migrado para ambiente móvel e a versão para plataforma fixa foi descontinuada. Tendo em vista a independência da execução dos agentes em relação à interface gráfica, bem como a linguagem utilizada para seu desenvolvimento ser multiplataforma, a versão final deste *framework* pode ser, através de poucas alterações realizadas, migrada novamente para plataforma fixa.

Para aplicações multiagentes de jogos, uma interface gráfica fluída e atrativa torna-se um recurso essencial para o sucesso do mesmo. De acordo com (Filho, 2000), sabe-se que um sistema de leitura visual , a percepção é realizada através

de dois tipos de estímulos:

- O estímulo visual, através da retina;
- O estímulo cerebral, que não ocorre em pontos isolados mas sim em toda sua extensão.

Dessa forma, a importância de uma interface gráfica atrativa, que agrade o usuário quanto à percepção visual, além da jogabilidade e usabilidade do jogo, fornecem um atrativo extra para que o usuário se prenda ao jogo.

Algumas *engines* focadas em jogos estão disponíveis para esse fim. Entre elas, devido à facilidade de implementação, ao suporte de desenvolvimento e à extensa comunidade envolvida, além de tratar-se de uma ferramenta livre sob licença BSD, destacamos o jMonkey. Também conhecido como JME3, Consiste em uma engine gráfica especialmente para desenvolvimento de gráficos 3D que suporta ainda OpenGL 2 e OpenGL 4. Atualmente já é possível realizar a integração dessa engine com Android, apesar dessa integração ainda estar sendo aperfeiçoada. A integração entre o *framework* Piaba e a engine gráfica jMonkey pode compor um ambiente de desenvolvimento poderoso para desenvolvimento de jogos e simulações 3D.

A utilização de agentes em locais diferentes pode ser utilizado em aplicações com intuito de aproveitar informações distintas geograficamente através de hardwares inteligados por uma rede de comunicação. Uma aplicação composta por um agente que interage com um usuário através de uma interface gráfica (cliente) e se comunica com outros agentes em um servidor de aplicações pode ser útil para que um hardware específico mais potente possa ser utilizado para processar informações, enquanto outro menos potente seja responsável por pequenas ações que não necessite muito poder de processamento. Para tal, um protocolo de comunicação segura deve ser implementado, afim de garantir a confidencialidade e confiabilidade dos dados transmitidos entre os agentes de locais diferentes.

Diferentemente dos testes de *softwar*es convencionais, onde uma entrada de dados proporciona uma mesma saída de dados tantas vezes quanto forem processados, o teste de aplicações com agentes inteligentes torna-se um desafio por não podermos garantir que a execução da aplicação com uma determinada entrada proporcione sempre a mesma saída. A simples mudança de clock do processador do hardware utilizado em relação a uma execução anterior, pode fazer

com que uma *thread* de um agente execute antes de outro agente, ocasionando em uma possível mudança na tomada de decisão desses agentes. Os testes aplicados no *framework* Piaba foram realizados através da abordagem de teste funcional, onde uma funcionalidade é implementada e em seguida executada manualmente para analisar a sua execução. Por muitas vezes, *breakpoints* no código da aplicação foi utilizada e a mesma executada em modo *debug,* para analisar o ciclo de vida de uma aplicação. O desenvolvimento de uma suíte de testes específica para o desenvolvimento de aplicações multiagentes no Piaba pode ajudar nesse problema.

Considerando a definição do termo turístico “viver uma localidade” como sendo o processo de conhecer uma localidade através das suas características mais íntimas, com o intuito de entender e compreender a sua cultura, sua história, suas linguagens e dialetos, sua religiosidade, seus costumes, seu povo e sua herança, podemos dizer que “viver uma localidade” é um grande desafio. Um desafio para os moradores nativos e um muito maior para os turistas. O uso de modernas tecnologias de informação e comunicação poderiam ser utilizados com o intuito de criar uma ferramenta de imersão pessoal que possa ser classificada como o “viver uma localidade”. Uma ferramenta para ser usada tanto na educação como para prover mais informações aos turistas que visitam a localidade. Nesse contexto, este é um trabalho inovador por definir uma interface configurável pelo contexto de uso. Esse contexto aborda não só as peculiaridades do usuário em si, como também os objetivos de uso da plataforma: turístico ou educativo. Esse projeto foi submetido à aprovação e nele há a possibilidade de utilizar o Piaba como *framework* principal para desenvolvimento da aplicação.

7 CONCLUSÃO

O *framework* Piaba foi concebido através de metodologias ágeis de desenvolvimento de *software*s e de pesquisas no campo de agentes inteligentes e ambientes móveis. O código final será em breve disponibilizado para a comunidade através de um projeto *github*, uma plataforma de distribuição de projetos de *software* gratuita onde o usuário pode criar um projeto e ter acesso à uma área especifica para ele, podendo disponibilizar versões e informações sobre alterações da aplicação. Dessa forma, outras pessoas podem baixar o código disponibilizado, realizar alterações e disponibilizar melhorias para a ferramenta. Espera-se que essa interação permita aumentar as funcionalidades do *framework*, bem como facilite o

acesso da comunidade de desenvolvedores ao mesmo. O github pode ser acessado através da url https://github.com/.

Esse trabalho propõe uma ferramenta pronta, específica para o desenvolvimento de sistemas multiagentes em plataforma de desenvolvimento Android. Sua idealização ocorreu em meio às poucas ferramentas existentes para esse fim e de necessidades que não eram especificamente contempladas por elas, como a afetividade dos agentes implementados. As tecnologias escolhidas para sua concepção (linguagem Java e XML) são populares na comunidade de desenvolvedores, tornando mais fácil o desenvolvimento de aplicações com esse *framework*. As abstrações dos elementos de um sistema multiagente inteligente, como estrutura dos agentes, de conhecimento e do ambiente de execução, torna a aplicação final mais coesa e mais fácil de ser alterada. Os métodos nativos da ferramenta são intuitivos, permitindo uma identificação rápida de seus objetivos.

O ciclo de vida bem definido permite uma maior facilidade na depuração da execução da aplicação. A ferramenta é independente de interfaces gráficas e permite o desenvolvedor realizar a integração com sua interface através de uma classe específica para esse fim, que é executada após cada ciclo de execução. Os agentes possuem memória de execução das ações no ambiente e alteram o estado do mundo através da execução destas. A cada ciclo de execução a base de crenças do agente é atualizada para perceber novamente o ambiente.

A ferramenta não implementa um protocolo específico de comunicação entre os agentes, mas suporta funcionalidades relacionadas como o envio e leitura de mensagens. Cada agente possui uma caixa de entrada e uma caixa de saída para manipulação do envio e recebimento de mensagens. O envio da caixa de saída de um agente para a caixa de entrada do destinatário é realizado pelo próprio *framework*.

Apesar de não implementar um modelo afetivo específico, o suporte a agentes afetivos do *framework* provê meios de recalcular intensidades de emoções durante a execução dos agentes. Essa funcionalidade deixa o desenvolvedor livre para implementar seu modelo de afetividade e a forma como essa afetividade influencia nas decisões e realizações dos agentes. Isso facilita o desenvolvimento de agentes que simulem o comportamento humano.

REFERÊNCIAS

ABLESON, W. Frank, SEN, Robi, KING, Chris, ORTIZ, C. Enrique. **Android in Action. 3 edição**. Editora Manning, New York, NY. 2012.

AMARAL, Daniel Capaldo, CONFORTO, Edivandro Carlos, BENASSI, João Luís Guilherme, ARAUJO, Camila de. **Gerenciamento Ágil de Projetos: Aplicação em Produtos Inovadores**. Editora Saraiva. Edição 1. São Paulo, 2011. 240 p.

BELLIFEMINE, Fabio Luigi. CAIRE, Giovanni. GRENWOOD, Dominic. **Developing Multi-Agent Systems with JADE**. Series Editor: Michael Wooldridge, Liverpool University, UK : Wiley; 1 edition. April 2, 2007.

BOOCH, Grandy. **Object-Oriented Analysis and *design* with Applications**. Editora Addison-Wesley Professional. 3ª edição. 720 páginas. Abril de 2007.

BORDINI, Rafael H., JOMI, Fred Hübner, WOOLDRODGE, Michael. **Programming Multi-Agent Systems in AgentSpeak using Jason.** John Wiley Sons Ltd, West Sussex, UK: Wiley; 1 edition. November 12, 2007.

CAIRE, Giovanni, IAVARONE, Giovanni, IZZO, Michele, HEFFNER, Kelvin. **Jade Tutorial: Jade Programming for Android**. Disponível em: http://jade.tilab.com/doc/tutorials/JadeAndroid-Programming-Tutorial.pdf. Acesso em 06 de Junho de 2013. Telecom Italia. Junho de 2012.

CHOUCHANE, Khamsa, ALOUI, Ahmed, et al. **Agent-Based Approach for Mobile Learning using Jade-LEAP.** Computer Science Department, Faculty of Sciences. Universiy Hadj Lakhdar. Batna, Argeria. Conference Name: Proceedings of the 4th International conference on Web and Information Technologies, ICWIT 2012, Sidi Bel Abbes, Algeria, April 29-30, 2012.

CLORE, Gerald L., PALMER, Janet E. **Affective Guidance of Intelligent Agents: How Emotion Controls Cognition**. Journal Cognitive Systems Research. Março de 2009. P. 21-30. Volume 10.

COHN, Mike. **Desenvolvimento de *software* com Scrum: Aplicando Métodos Ágeis com Sucesso**. Editora Bookman. Edição 1. Porto Alegre, 2011. 496 p.

COSTA, Mônica, FEIJÓ, Bruno. **Reactive Agents in Behavioral Animation**. 8ª Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens. p. 159-

166. Campinas, São Paulo, 1995.

DEUTSCH, L. Peter. ***design* Reuse and *framework*s in the Smalltalk-80 System**. In ted J. Biggerstaff and Alan T. Perlis, editors, *software* Reusability. Applications and Experience. Addison-Wesley, Reading, MA, 1989. Volume II. P. 55-71.

FERBER, J.; GASSER, L. Intelligence artificielle distribuée. In: INTERNATIONAL WORKSHOP ON EXPERT SYSTEMS & THEIR APPLICATIONS, 10., 1991, Avignon. Cours n. 9. France: (s.n), 1991

FILHO, João Gomes. **Gestalt do objeto: sistema de leitura visual do objeto**. 8ª edição. Editora Escrituras. São Paulo, SP. 136p. 2000.

FREEMAN, Eric, FREEMAN, Elizabeth, SIERRA, Kathy, BATES, Bert. **Head First: *design* Patterns**. Edição 1. O'Reilly. ISBN 0596007124. Novembro de 2004.

GARGENTA, Marko. **Learning Android**. O'Reilly. 1ª edição. Março de 2011. USA.

JENNINGS, N. R. An agent-based approach for building complex *software* systems. Communications of the ACM. **Communications of the ACM**. New York, 44, n. 4, 35-41, Abril, 2001.

JENNINGS, N., WOOLDRIDGE, M. **Applications of Intelligent Agents**. In *Agent Technology: Foundations, Applications, and Markets*, , Secaucus, NJ, Springer-Verlag, Berlin, 1998. p. 3–28.

JOHNSON, E. Ralph. **Framworks = Components + Patterns**. How *framework*s compare to other object-oriented reuse techniques. Communications Of The ACM. 1997, vol 40, n. 10.

KOETSIER, John. **Android captured almost 70% global smartphone market share in 2012, Apple just under 20%. Venture Beat**. 2013. Disponível em <http://venturebeat.com/2013/01/28/android-captured-almost-70-global-smartphone-market-share-in-2012-apple-just-under-20/>. Acesso em: 21 jul. 2013.

GEORGEFF, M., PELL, B., POLLACK, M. **Tambe, and M. Wooldridge. The Belief-Desire-Intention Model of Agency**. In J. P. Muller, M. Singh, e A. Rao, editors *Intelligent Agents V* Springer-Verlag Lecture Notes in AI. March 1999. Volume 1365.

PADGHAM, Lin, WINIKOFF, Muchael. **Developing Intelligent Agent Systems. A**

Practical Guide. RMIT University. John Wiley & Sons, LTD. Melbourne. Australia. 2004.

PELLEGRINI, Ana Lúcia, FERNANDES, Sônia Regina, GOMES, Almiralva. ***estresse* e Fatores Psicossociais**. Psicologia, Ciência e Profissão. Conselho Federal de Psicologia. ISSN 1414-9893. Edição 30. Maio de 2010. Brasília, DF. P 712-725.

QUEIROZ, Jonas Felipe, GUILHERME, Ivan Rizzo. **Arquitetura de Um Sistema Multiagente para Supervisão e Controle. Revista Interciência e Sociedade**. Faculdade Municipal Professor Franco Montoro., Ano 1, p. 51 a 61. Volume 2.

RAO, Anand S. **AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language.** MAAMAW '96 Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world : agents breaking away: agents breaking away. p. 45 – 55. Australian Artificicial Intelligence Institute. Melbourne, Australia.

RICCI, Alessandro, VIROLI, Mirko, OMICINI, Andrea. **CArtAgO: a *framework* for prototyping artifact-based environments in MAS**. Proceeding E4MAS'06 Proceedings of the 3rd international conference on Environments for multi-agent systems III. p. 67-86. Berlin, 2007.

RIERSON, Leanna. ***software* Reuse in Safety-Critical Systems**.. Tese (Mestrado em Engenharia de *software*) – Rochester Institute of Technology. Rochester, Nova York. Maio de 2000. 44p.

ROGERS, Rick. LOMBARDO, John. MEDNIEKS, Zigurd, MEIKE, Blake. **Android. Desenvolvimento de Aplicações Android**. 1ª edição. O'Reilly. Novatec Editora. 2009.

SANDBERG, Thomas Willer. **Evolutionary Multi-Agent Potential Field based AI approach for SSC scenarios in RTS games**. Tese (Mestrado) - IT University of Copenhagen. Copenhagen, Fevereiro de 2011. 55p.

SANTI, Andrea, GUIDI, Marco, RICCI, Alessandro. **JaCa-Android: An Agent-based Plataform for Building Smart Mobile Applications**. Languages, Methodologies, and Development Tools for Multi-Agent Systems. Third International Workshop, LADS 2010, Lyon, France, August 30 – September 1, 2010, Revised Selected Papers, P. 95-114.

SIGNORETTI, Alberto. **Agentes Inteligentes com Foco de Atenção Afetivo em Simulações Baseadas em Agentes**. Tese (Doutorado em Ciência da Computação) - Departamento de Informática e Matemática Aplicada, Universidade Federal do Rio

Grande do Norte. Natal. Agosto de 2012. 228p.

SIGNORETTI, Alberto, CAMPOS, André M., CANUTO, Laura E. A. Santana & Sergio V. Fialho (2008), Escola Potiguar de Computação e suas Aplicações, Vol. 1 de EPOCA, EDUFRN, Natal - RN, capítulo Simulação de Organizações Através do Paradigma de Agentes Inteligentes, p. 67–95.

STEUNEBRINK, B.R., DASTANI, M.M. & Meyer, J.-J.Ch. **The OCC Model Revisited**. In D. Reichardt (Ed.), Proceedings of the 4th Workshop on Emotion and Computing - Current Research and Future Impact. Paderborn, Germany. 2009.