Universidade Federal do Rio Grande
Programa de Pós-Graduação em Modelagem Computacional

Ambiente de desenvolvimento de jogos com reuso de software e inteligência artificial

Carlos Alberto Barros Cruz Westhead Madsen

Rio Grande, 17 de Setembro de 2012

Universidade Federal do Rio Grande
Programa de Pós-Graduação em Modelagem Computacional

Ambiente de desenvolvimento de jogos com reuso de software e inteligência artificial

Carlos Alberto Barros Cruz Westhead Madsen

Dissertação apresentada ao Programa de Pós-Graduação em Modelagem Computacional, como requisito parcial para obtenção do título de Mestre em Modelagem Computacional.

Orientador(a): Profa. Dra. Diana Francisca Adamatti

Rio Grande, 17 de Setembro de 2012

Este trabalho foi analisado e julgado adequado para a obtenção do título de Mestre em Modelagem Computacional e aprovado em sua forma final pelo orientador.

Profa. Dra. Diana Francisca Adamatti

Banca Examinadora:

Prof. Dr. Adriano Velasque Werhli
Centro de Ciências Computacionais – FURG
Prof. Dr. Ricardo Matsumura Araujo
Centro de Desenvolvimento Tecnológico – UFPEL
Profa. Dra. Sílvia Silva da Costa Botelho
Centro de Ciências Computacionais – FURG

Dedico esta dissertação a minha família que me deu muito
apoio nos momentos mais difíceis da minha vida,
especialmente a minha esposa Denise e a minha filha Érica.

AGRADECIMENTOS

Dentre todas as pessoas e instituições que me apoiaram para que eu alcançasse esse objetivo, agradeço primeiramente a Deus, por ter me dado força, saúde e determinação. A minha esposa Denise, pelo carinho, paciência e dedicação nestes dois anos. Aos meus colegas de mestrado, por terem se mostrado tão abertos e generosos durante as disciplinas que cursamos juntos. A Universidade Federal do Rio Grande, por ter me liberado de minhas atividades profissionais, possibilitando que este trabalho viesse a ser realizado. E por fim, a Profa. Dra. Diana F. Adamatti, pela generosidade em me aceitar como orientando e por todo o apoio durante esta trajetória.

Se você quiser alguém em quem confiar, confie em si mesmo.

Quem acredita sempre alcança.

Renato Russo

RESUMO

Este trabalho apresenta um ambiente de desenvolvimento, da camada de tomada de decisão de NPCs (*Non-Player Characters*), chamado FURG Smart Games. Seu principal objetivo é agilizar, através de reuso de software, a inserção de técnicas de inteligência artificial no desenvolvimento de jogos eletrônicos. Para tanto, este ambiente é composto de um framework e de um conjunto de ferramentas RAD (*Rapid Application Development*). O framework é responsável por implementar, utilizando herança, a FSM (*Finite-State Machine*), que neste contexto é o cerne da tomada de decisão, e por intermédio de composição, as seguintes técnicas de inteligência artificial: ANN (*Artificial Neural Network*), FIS (*Fuzzy Inference Systems*) e GA (*Genetic Algorithm*). Para cada uma das quatro técnicas apresentadas (FSM, ANN, FIS e GA) foi proposto uma ferramenta RAD, com o objetivo de agilizar a sua configuração e proporcionar a geração automatizada de código fonte, no padrão do framework.

Por fim, é apresentado um estudo de caso para a validação do ambiente desenvolvido, o qual demonstra a utilização de cada uma das ferramentas RAD, bem como a aplicação do framework em um jogo eletrônico.

Palavras-chaves: Frameworks, Reuso de Software, Jogos Eletrônicos, Inteligência Artificial, Rapid Application Development, Finite-State Machine, Artificial Neural Network, Fuzzy Inference Systems, Genetic Algorithm.

ABSTRACT

This work presents a development environment of the decision-making layer of NPCs (Non-Player Characters), called FURG Smart Games. Its main purpose is to speed up through the reuse of software, the insertion of artificial intelligence techniques in the development of electronic games. In order to do so, this environment is composed of a framework and a number of RAD (Rapid Application Development) tools. The framework is responsible for implementing, using heritage, the FSM (Finite-State Machine), which in this context is core for the decision-making process, and by composition, the following artificial intelligence techniques used are: ANN (Artificial Neural Network), FIS (Fuzzy Inference Systems) and GA (Genetic Algorithm). For each of the four techniques presented (FSM, ANN, FIS e GA) a RAD tool was proposed to speed up its configuration and provide the automatic generation of the source code, in the framework standard.

Finally, a case study is presented for the validation of the developed environment, which demonstrates the use of each of the RAD tools as well as the application of the framework in an electronic game.

Keywords: Frameworks, Software Reuse, Games, Artificial Intelligence, Rapid Application Development, Finite-State Machine, Artificial Neural Network, Fuzzy Inference Systems, Genetic Algorithm.

Lista de Figuras

2.1 Software desenvolvido a partir do zero (adaptado de [Salva, 2011]). 9

2.2 Software desenvolvido reutilizando classes de bibliotecas (adaptado de [Salva, 2011]). 9

2.3 Software desenvolvido utilizando framework (adaptado de [Salva, 2011]). . 9

3.1 Modelo de Game AI (adaptada de [Millington, 2006]). 15

3.2 Representação gráfica de uma FSM (adaptada de [Bourg and Seemann, 2004]). 17

3.3 FSM de um soldado (NPC) (adaptado de [Smed and Hakonen, 2006]). . . . 19

3.4 Transição entre o estado Pattrolling e Attacking (adaptado de [Madsen et al., 2012]). 20

3.5 Exemplo de modelo de classes do padrão de projeto State. 21

3.6 Simplificação do neurônio biológico (adaptado de [Packer, 2010]). 23

3.7 Modelo de neurônio artificial de McCulloch e Pitts (adaptado de [Haykin, 2001]). 24

3.8 Exemplo de função de pertinência triangular que define o conjunto Fuzzy dos adultos (adaptado de [Ganga, 2010]). 27

3.9 Variável linguística “Poder de Ataque” (adaptado de [Ganga, 2010]). 28

3.10 Módulos de um sistema fuzzy (adaptado de [Ganga, 2010]). 28

4.1 Diagrama de classes simplificado do FSG 39

4.2 Diagrama de classes de ANN no FSG 40

4.3 Diagrama de classes de GA no FSG 41

4.4 Diagrama de classes do FSG referente a FIS 42

4.5 Processo de geração de código fonte a partir das ferramentas RAD. 44
4.6 Edição de uma FSM na ferramenta RAD 46
4.7 Relacionando uma transição com uma Técnica de IA. 47
4.8 Gerando o código fonte do NPC. 47
4.9 Diagrama de classes no padrão FSG do NPC "Soldier". 48
4.10 Interface inicial do RAD de redes neurais. 48
4.11 Área de configuração do treinamento da rede. 49
4.12 Treinamento das redes. 50
4.13 Interface inicial do RAD de algoritmos genéticos. 51
4.14 Execução do GA configurado na figura anterior. 53
4.15 Catalogação das variáveis linguísticas 54
4.16 Configuração de um termo linguístico 54
4.17 Base de regras de produção do FIS 55
4.18 Configuração de uma nova regra de produção 56
4.19 Teste de uma regra de produção 56
4.20 Teste da base de regras 57

5.1 Jogo proposto para validação da ferramenta. 58
5.2 FSM responsável pelo comportamento do NPC "Guarda". 61
5.3 Diagrama de classes do jogo implementado. 62
5.4 Código de utilização de ANN no jogo. 68
5.5 Termos linguísticos das variáveis Energia, Poder de Ataque e Poder de Defesa. 69
5.6 Termos linguísticos da variável Experiência. 69
5.7 Termos linguísticos da variável Raio de Ação. 69
5.8 Termos linguísticos da variável Velocidade. 69
5.9 Termos linguísticos da variável Atacar. 70
5.10 Código de utilização de FIS no jogo. 77
5.11 Código fonte do cromossomo. 78
5.12 Código de utilização de GA. 79
5.13 Comparação em micro segundos dos tempos de execução das três técnicas de IA. 80

5.14 Comparação em micro segundos dos tempos de execução do ANN e do FIS. 80

Lista de Tabelas

3.1 Exemplo de regra de produção (adaptado de [Simões and Shaw, 2007]) . . . 29

4.1 Um laço de repetição FOR na linguagem XML. 45

4.2 Tradução para PHP. 45

5.1 Atributos do Guarda na primeira simulação. 65

5.2 Atributos do Jogador na primeira simulação. 65

5.3 Atributos do Guarda na segunda simulação. 65

5.4 Atributos do Jogador na segunda simulação. 65

5.5 Conjunto de regras referentes as energias dos oponentes. 70

5.6 Conjunto de regras referentes as experiência em combate dos oponentes. . . 70

5.7 Conjunto de regras referentes as capacidade de defesa do Jogador e ao poder de ataque do Guarda. 71

5.8 Conjunto de regras referentes as capacidade de ataque do Jogador e ao poder de defesa do Guarda. 71

5.9 Conjunto de regras referentes ao raio de ação dos oponentes. 71

5.10 Conjunto de regras referentes a velocidade dos oponentes. 71

5.11 Exemplo de regra de produção. 71

5.12 Definição do cromossomo. 73

5.13 Tempo de execução das técnicas de IA. 75

LISTA DE SIGLAS

ANN *Artificial Neural Network*

COTS *Commercial off-the-shelf*

CASE *Computer-Aided Software Engineering*

FIS *Fuzzy Inference Systems*

FSM *Finite-State Machine*

FuSM *Fuzzy State Machines*

ERP *Enterprise Resource Planning*

GA *Genetic Algorithm*

Game AI *Game Artificial Intelligence*

IA *Inteligência Artificial*

MCP Modelo de Neurônio Artificial de McCulloch e Pitts

NPC *Non-Player Character*

RAD *Rapid Application Development*

RPG *Role-Playing Game*

SUS *Stochastic Universal Sampling*

UML *Unified Modeling Language*

Conteúdo

Resumo

Abstract

Lista de figuras

Lista de tabelas

Lista de Siglas

1 Introdução **1**

1.1 Motivação 2

1.2 Objetivos 2

1.3 Organização 3

2 Reuso de Software **4**

2.1 Padrões de Projetos (*Design Patterns*) 7

2.2 Frameworks 8

2.2.1 Características Básicas 10

2.2.2 Classificação 11

2.3 RAD (*Rapid Application Development*) 12

2.4 Trabalhos Relacionados 13

3 Game AI **14**

3.1 Máquinas de Estados Finitos 17

3.2 Redes Neurais Artificiais 21

3.3 Lógica Fuzzy 26
3.4 Algoritmos Genéticos 30
3.5 Trabalhos Relacionados 36

4 Ambiente Proposto **37**

4.1 Framework FSG 38
4.1.1 Artificial Neural Network 40
4.1.2 Algoritmos Genéticos 40
4.1.3 Lógica Fuzzy 42
4.2 RAD (*Rapid Application Development*) 44
4.2.1 FSG - Finite-Sate Machine 44
4.2.2 FSG - ANNBack 47
4.2.3 FSG - GA 51
4.2.4 FSG - Fuzzy 53

5 Experimentos e Resultados **58**

5.1 Jogo Implementado 58
5.2 Configuração das Técnicas de IA 64
5.3 Artificial Neural Network 66
5.4 Fuzzy Inference System (FIS) 68
5.5 Genetic Algorithm 72
5.6 Avaliação de Desempenho dos Testes 75
5.7 Análise dos resultados 75

6 Conclusão **81**

Bibliografia **84**

Capítulo 1

Introdução

Há mais de trinta anos os jogos eletrônicos têm marcado presença na sociedade contemporânea, principalmente pelo viés do entretenimento [Alves, 2008]. Todavia, a partir da década de oitenta eles se distanciam completamente do modelo inicial, o qual era apenas um processo de ação e reação, testando os reflexos do jogador [Aranha, 2007], evoluindo para ambientes altamente interativos, nos quais os usuários podem resolver problemas, estabelecer estratégias, construir hipóteses, aprender regras e simular o mundo real [de Falco, 2007]. Dando prosseguimento a sua maturação, eles vieram a transpor a barreira do entretenimento, tendo aplicações na educação, segurança industrial, capacitação de funcionários e propaganda (*advergames*). Estas aplicações distintas de seu objetivo original são conhecidas hoje como jogos sérios (*serious games*) [Perucia et al., 2011] [Softex, 2005] [de Falco, 2007].

A indústria dos jogos eletrônicos, no contexto do software, é a que mais se expande no mundo, sendo um dos setores mais promissores e dinâmicos do mercado internacional [Softex, 2005], além de ser um dos segmentos culturais mais rentáveis e estratégicos, dado seu alto valor agregado [Abragames, 2008]. Nos países onde o segmento se encontra mais consolidado, EUA, Japão e Reino Unido, seu faturamento é maior que a bilheteria do cinema e começa a tirar espectadores do horário nobre da televisão [Abragames, 2008]. No contexto mundial, movimenta cerca de U$ 41 bilhões de dólares, superando a indústria do cinema, tendo uma previsão crescimento de 5% ao ano até 2014 [PWC, 2010].

Grande parte deste valor agregado vem da qualidade da interação entre o software e o jogador, em última instância das técnicas de Inteligência Artificial (IA) que o jogo

implementa [Millington, 2006]. Os jogos, por sua vez, proporcionam à IA uma grande gama de problemas a serem solucionados e com um grande desafio de, normalmente, requererem respostas em tempo real [Nareyek, 2001]. Todavia, a utilização de IA nos jogos requer uma mão de obra muito qualificada e um tempo considerável de produção. Neste contexto, modelos de desenvolvimento e técnicas de reuso de software ganham importância substancial.

1.1 Motivação

Ainda hoje a utilização da IA é um desafio para a indústria, principalmente pelo curto período de desenvolvimento dos jogos atuais, o que dificulta o aprendizado dos desenvolvedores para estas tecnologias e suas aplicações [Bourg and Seemann, 2004]. Em contraposição a isso, grande parte da qualidade presente em jogos advem da utilização adequada destas técnicas.

Neste contexto, o emprego de técnicas de reuso de software, como frameworks e padrões de projetos, além da utilização de ferramentas RAD (*Rapid Application Development*) é providencial, no intuito de diminuir o tempo e custo de desenvolvimento, bem como aumentar a qualidade do produto final.

1.2 Objetivos

Como objetivo geral, deseja-se um ambiente de desenvolvimento da camada de tomada de decisão dos agentes, presentes nos jogos, conhecidos como NPCs (*Non-Player Characters*), sendo a mesma regida por uma Máquinas de Estados Finitos (*Finite-Sate Machine* - FSM), agregada a técnicas de IA: Algoritmos Genéticos (*Genetic Algorithm* - GA), Redes Neurais Artificiais (*Artificial Neural Network* - ANN) e Sistemas de Inferência Fuzzy (*Fuzzy Inference Systems* - FIS). O presente trabalho tem o intuito de prover um comportamento menos previsível e mais realista aos NPCs, contribuindo assim para a qualidade do jogo que venha a ser implementado através deste ambiente. Esta qualidade não se refere apenas a experiência do jogador, mas também ao código que é gerado. Para tanto, o ambiente FURG Smart Games (FSG) se apresenta de duas formas bem distintas, a

primeira é um framework que engloba todas as técnicas citadas acima e a segunda é um conjunto de ferramentas RAD que visam agilizar e automatizar a geração de código fonte no padrão deste framework.

Objetivos Específicos

- Proposição e criação de um framework que englobe e abstraia as seguintes técnicas de IA: ANN, FIS, FSM e GA;
- Implementação de uma ferramenta RAD para modelagem da FSM, utilizando UML (*Unified Modeling Language*). Esta é responsável pela camada de tomada de decisão dos NPCs. E ainda geração do código fonte de acordo com o padrão de projeto State;
- Criação de ferramentas RAD que possibilitem a configuração e validação de cada uma das técnicas de IA presentes no framework e posterior geração de código fonte;
- Utilização do framework e das ferramentas RAD desenvolvidas para implementação de um jogo, onde os personagens duelam nos moldes dos RPGs (*Role-Playing Game*), através de uma interface gráfica gerenciada pelo framework JGame [van Schooten, 2012].

1.3 Organização

Esta dissertação está organizada em 6 capítulos. O capítulo 2 discorre sobre as principais técnicas de reuso de software, principalmente frameworks e padrões de projetos. No capítulo 3 é apresentado o conceito e um modelo de Game AI (*Game Artificial Intelligence*), além de apresentar as técnicas de IA selecionadas para este trabalho. Já no capítulo 4 é descrita a proposta de ambiente de desenvolvimento para a camada de tomada de decisão. No capítulo 5 são apresentados os testes e resultados da utilização da ferramenta. Finalmente, no capítulo 6 estão as conclusões, bem como os trabalhos futuros.

Capítulo 2

Reuso de Software

O reuso é intrínseco ao processo de solução de problemas utilizado pela raça humana, desde seus primórdios, na medida que as soluções para determinados problemas eram encontradas, estas eram registradas e, caso possível, adaptadas a outros problemas similares. Isso foi possível graças a capacidade de abstração e adaptação presentes nos seres humanos [Sommerville, 2007] [Gimenes and Huzita, 2005].

No que tange ao software, atualmente seu processo de desenvolvimento é cada vez mais intrincado e consumidor de recursos humanos, principalmente pelo fato de que o escopo e complexidade dos sistemas tem se tornado cada vez maior. Sendo assim, há necessidade de reuso no projeto de software. Este consiste em uma prática sistematizada de desenvolvimento em blocos, visando que possíveis semelhanças e requisitos de diferente projetos possam ser explorados. No intuito de tornar o software desenvolvido mais confiável, flexível, facilitar a manutenção e evolução, aumentar sua qualidade, diminuir o tempo desenvolvimento, custos, riscos do projeto e retrabalho [Sommerville, 2007] [da Silva Oliveira and de Mattos, 2001].

Seu surgimento ocorreu devido a necessidade de se economizar recursos de hardware, em vista de que há algumas décadas atrás não havia memória suficiente para se armazenar muitas rotinas. Então, a solução encontrada foi executar tarefas similares por meio de um mesmo procedimento parametrizável. As primeiras idéias de reuso foram apresentadas em meados da década de sessenta, com o surgimento de mecanismos do tipo sub-rotinas, funções e macros. Já na década de setenta surgem as primeiras bibliotecas de rotinas comerciais, nas quais o seu código objeto é utilizando por diversos programas ao mesmo

tempo. Mas o conceito de reuso realmente se consolidou nos anos oitenta com a maior difusão das linguagens orientadas a objeto, principalmente pelo advento dos conceito de classes e objetos reusáveis, que se comunicam através de interfaces abstraindo por inteiro seu funcionamento interno, permitindo assim uma maior desacoplação entre os componentes do software. E finalmente a partir dos noventa, com a disseminação de computação distribuída e da internet, o reuso tornou-se fundamental para qualquer desenvolvimento de software profissional. Todavia, somente nesses últimos dez anos que essa técnica vem sendo utilizado em grande escala [Sommerville, 2007].

O reuso de software ocorre em diversas escalas. Pode-se reutilizar uma aplicação inteira, incorporando a mesma dentro de outros sistemas, sem que hajam mudanças, pela configuração para novos clientes ou pelo desenvolvimento de família de aplicações com uma arquitetura em comum. Em contraposição, pode-se reutilizar um simples componente, podendo ser uma única função (um exemplo rotineiro são as funções matemáticas), ou ainda uma determinada classe.

No decorrer do amadurecimento desta técnica de desenvolvimento, diferentes abordagens foram propostas, todas explorando o fato de que sistemas em um mesmo domínio de aplicação são similares e tem potencial para utilização de reuso. Dentre elas destacam-se as seguintes [Sommerville, 2007] [Gimenes and Huzita, 2005] [Gamma et al., 2000] [Franzosi et al., 2009]:

- **Bibliotecas de rotinas**: uma das primeiras formas de reuso, originalmente implementada através de linguagens procedurais, nas quais uma biblioteca era constituída por um conjunto de procedimentos ou funções dentro de um dado domínio.

- **Componentes**: neste caso, o desenvolvimento do sistema é decomposto em um conjunto de componentes que devem interagir através de suas interfaces.

- **Empacotamento de sistemas legados**: sistemas legados podem ser reusados através de seu empacotamento pela definição de um conjunto de interfaces pela quais são fornecidos acesso a outros softwares que venham a ser desenvolvidos.

- **Frameworks**: neste contexto o reuso ocorre através de um conjunto de classes abstratas e concretas implementadas no intuíto de fornecer um arcabouço de soluções genéricas, dado um domínio de problema em específico.

- **Geradores de software**: neste caso, o conhecimento reusável é capturado em um sistema gerador de software que pode ser programado por especialistas em uma dada área de conhecimento, usado uma linguagem orientada a domínio ou uma ferramenta CASE (*Computer-Aided Software Engineering*) interativa, que suporte geração de sistemas.

- **Integração de COTS**: um COTS (*Commercial off-the-shelf*) é um software completo que pode ser reusado através da integração ao sistema que está sendo desenvolvido. Os exemplos mais comuns são sistemas gerenciadores de banco de dados e ferramentas de relatórios.

- **Linhas de produtos de aplicações**: nesta abordagem é criada uma aplicação generalizada com base em uma arquitetura comum de tal forma que possa ser adaptada para atender a necessidade de diferentes clientes. Um exemplo desta forma de reuso são os sistemas ERP (*Enterprise Resource Planning*).

- **Padrões de Projeto (*Design patterns*)**: nessa abordagem se tem a descrição de um dado problema e a essência de sua solução de forma abstrata, ou seja, sem se ater a detalhes de implementação. Assim, ela pode ser adaptada para diversas aplicações.

- **Sistemas orientados a serviço**: o reuso ocorre através da interoperabilidade entre os sistemas já desenvolvidos, onde esses softwares disponibilizam seus serviços de forma compartilhada. Dentro desta abordagem se destacam os serviços web (*web services*) devido a facilidade de integração de sistemas construídos em ambientes heterogêneos.

Como apresentado acima, muitas técnicas foram propostas ou desenvolvidas visando maximizar o reuso de software. No entanto, dentre estas, duas são o foco deste trabalho: padrões de projeto e frameworks.

2.1 Padrões de Projetos (*Design Patterns*)

Essa técnica de reuso descreve um dado problema e a essência de sua solução. Dessa forma, ela pode ser adaptada para diversas aplicações, pois tem natureza abstrata e não enfatiza detalhes de implementação. Pode ser encarada como uma descrição de conhecimento e experiências acumuladas de uma dada solução comprovada para um problema comum.

Em vista disso, um padrão de projeto nomeia, abstrai e identifica aspectos chaves de uma estrutura de projeto comum, no intuito de torná-la útil para a criação de softwares orientados a objeto. Essa relação estreita com a programação orientada a objetos se deve ao fato dos padrões contarem com características de objetos, tais como a herança e polimorfismo. Assim, são definidos por [Gamma et al., 2000] como: "*descrições de objetos e classes comunicantes que precisam ser personalizadas para resolver um problema geral de projeto num contexto particular*".

Por fim, de acordo com [Gamma et al., 2000] e [Sommerville, 2007] existem quatro elementos essências de um padrão de projeto:

- **Nome do padrão**: referência que pode ser usada para descrever um problema de projeto, suas soluções e consequências em uma ou duas palavras.

- **Problema**: descreve em que situação o padrão deve ser utilizado, explicando o problema e seu contexto. Pode incluir uma lista de condições que deve ser satisfeita, para que haja sentido a utilização deste padrão.

- **Solução**: descreve os elementos que compõem o padrão de projeto, seus relacionamentos, suas responsabilidades e colaborações. Nesse item, não são descritas implementações específicas, pois o padrão age mais como um gabarito. Assim, ele fornece uma descrição abstrata de um problema e de como um conjunto de classes e objetos o resolve.

- **Consequências**: são resultados de análises das vantagens e desvantagens da aplicação do padrão. Elas incluem seu impacto sobre a flexibilidade, extensibilidade ou portabilidade de um sistema.

Os padrões de projetos podem ser subdivido em três famílias distintas: padrões de criação, estruturais e comportamentais. Os padrões de criação estão voltados a forma

de instanciação de objetos, dentre os quais pode-se destacar: Abstract Factory, Builder, Factory Method, Prototype e Singleton. Já os padrões estruturais tratam da associação entre classes ou objetos, sendo eles: Adapter, Bridge, Composite, Decorator, Facade, Flyweight e Proxy. Por fim, os padrões comportamentais focam na interação ou em divisões de responsabilidades entre as classes ou objetos, dos quais pode-se citar: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, **State**, Strategy, Template Method e Visitor.

Neste trabalho, utiliza-se o padrão de projeto State, dado a possibilidade de seu emprego na implementação de FSMs.

Padrão de Projeto State

O State é um padrão comportamental voltado a implementação de FSM, ou seja, ele tem por objetivo controlar a maneira com que os objetos, neste contexto os NPCs, interagem entre si, encapsulando comportamentos intercambiáveis e se valendo de delegação para decidir qual comportamento deve ser usado [Freeman et al., 2004]. Assim, permitindo que um dado objeto mude de comportamento de acordo com o seu estado interno, dando a impressão de que este objeto mudou de classe [Gamma et al., 2000] [Larman, 2004]. Aplicações em tempo real, como os jogos, tendem a se beneficiar especialmente desta arquitetura, pois frequentemente trabalham com uma perspectiva de evento e troca de estado.

2.2 Frameworks

Um framework é uma aplicação reusável e semi-completa que pode ser especializada para produzir aplicações personalizadas, se caracterizando por um conjunto de classes abstratas e concretas com o intuito de minimizar o esforço de desenvolvimento e manutenção de um dado software, visando o reuso não somente na codificação, mas também na análise e projeto [Bittencourt and Osório, 2001]. Este conjunto deve ser estendido e customizado pelo desenvolvedor de acordo com as regras de negócio que pretende implementar [Soares, 2009]. Além das classes propriamente ditas, esta técnica também especifica como as instâncias das mesmas devem interagir entre si [de Medeiros et al., 2006]

[Weschter, 2008] [Ferreira, 2005].

Essa técnica difere de uma simples biblioteca de classes, onde estas são independentes entre si, isoladas, cabendo ao desenvolvedor a tarefa de estabelecer suas interligações, pois em um framework as classes devem ser interligas e colaborativas [Silva, 2000] [Salva, 2011] [Ferreira, 2005]. No intuito de elucidar essa diferença, a Figura 2.1 apresenta um exemplo do arranjo das classes em um software desenvolvido a partir do zero. Já na Figura 2.2, a utilização de bibliotecas de classes, onde as áreas em cinza mostram as classes reutilizadas. Finalmente a Figura 2.3 apresenta os frameworks. Neste caso a área em cinza representa todas as classes que pertencem ao framework.

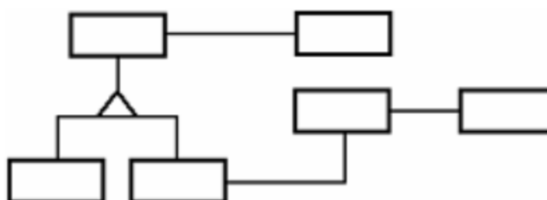

Figura 2.1: Software desenvolvido a partir do zero (adaptado de [Salva, 2011]).

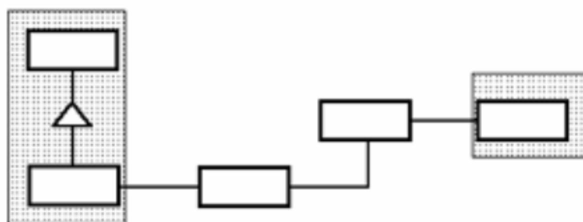

Figura 2.2: Software desenvolvido reutilizando classes de bibliotecas (adaptado de [Salva, 2011]).

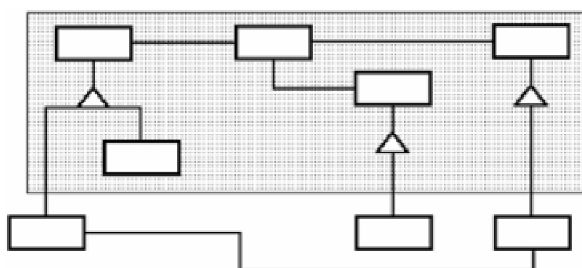

Figura 2.3: Software desenvolvido utilizando framework (adaptado de [Salva, 2011]).

Em consonância com os objetivos do reuso de software, pode-se destacar as seguintes vantagens na utilização de frameworks [Soares, 2009]:

- Redução no custo e tempo total de desenvolvimento;
- O desenvolvedor fica mais focado nas regras de negócio;
- Menor tempo para colocar um produto no mercado (*time-to-market*);
- Consistência;
- Compatibilidade entre sistemas;
- Independência de especialista;
- Padronização;
- Maior rapidez na detecção e correção de erros;
- Redução da manutenção dos sistemas;
- Maior facilidade para implementar ferramentas de teste;
- Otimização dos recursos do desenvolvimento.

2.2.1 Características Básicas

As principais características que um framework deve apresentar, para ser bem sucedido são [de Medeiros et al., 2006]:

- **Modularidade**: o uso de frameworks possibilita uma maior modularidade dos sistemas implementados por abstrair a sua forma de funcionamento, dando acesso, ao desenvolvedor, somente de suas interfaces. Essa forte modularidade facilita a localização de possíveis problemas, além simplificar as mudanças no projeto, reduzindo assim o esforço de manutenção.
- **Reusabilidade**: no que tange a reusabilidade, pode-se destacar o enorme ganho que estas ferramentas proporcionam. A programação através de componentes genéricos e pré-formatados é notoriamente mais produtiva, além de aumentar a confiabilidade e a qualidade dos softwares produzidos.

- **Extensibilidade**: com a utilização desta técnica, obtem-se uma maior capacidade de extensão de componentes, possibilitando, através do mecanismo de herança, a customização de acordo com o domínio do problema em questão.

2.2.2 Classificação

Normalmente os frameworks são classificados em dois grupos distintos: os "caixa branca" (*whitebox*) e os "caixa preta" (*blackbox*) [de Medeiros et al., 2006] [Sommerville, 2007].

Nos de caixa branca, também conhecido como focados na herança, o reuso é realizado exclusivamente por herança das classes abstratas disponíveis no framework. Sendo assim, o desenvolvedor deve, necessariamente, criar subclasses para a customizar e implementar os recursos necessários a sua aplicação. Em contrapartida, é preciso ter um conhecimento aprofundado do funcionamento interno do conjunto de classes, prejudicando a modularidade. Nesse caso, o projeto do framework fica facilitado, pois não há a necessidade de se prever todas as alternativas de implementação possíveis, por se valer apenas de classes abstratas. No entanto, dificulta seu uso, pois fica cargo do desenvolvedor realizar toda a programação.

Em contraposição, os classificados como caixa preta, também conhecido como focados na composição, atuam predominantemente através de composição. Sendo assim, o desenvolvedor só se preocupa em combinar os objetos instanciados, das classes concretas existentes, da melhor forma possível, tendo em vista a regra de negócio que pretende implementar. Como o próprio nome sugere, o funcionamento interno das classes fica totalmente abstraído do usuário, beneficiando a modularidade. No entanto, frameworks deste formato são de mais difícil implementação, por serem compostos em sua maioria de classes concretas.

Apesar dessa classificação ser amplamente empregada, poucos frameworks atualmente podem ser classificados como puramente caixa branca ou caixa preta. O mais comum é estar em uma escala intermediária [Soares, 2009].

Não distante dessa regra, o FSG, framework proposto neste trabalho, quando empregado para a implementação da FSM se vale fundamentalmente de herança, caracterizando-se como caixa branca. Por outro lado, no que diz respeitos as demais técnicas de IA, o mesmo é empregado predominantemente através de composição, agindo como um caixa

preta.

2.3 RAD (*Rapid Application Development*)

As ferramentas RAD tem o objetivo de agilizar o desenvolvimento de software, através de ambientes gráficos de programação, que possibilitam a geração automatizada de código fonte mediante as diretrizes fornecidas pelo desenvolvedor [Piske and Seidel, 2006].

A integração destes sistemas com frameworks é providencial, permitindo mais do que a criação de um software em menor tempo, mas ainda um produto final de maior qualidade e de manutenção mais fácil [Seára and Benitti, 2006].

Dentre os principais benefícios da utilização de ferramentas RAD no desenvolvimento de software pode-se destacar os seguintes [Ritu et al., 2000]:

- A ferramenta RAD incrementa a produtividade no desenvolvimento da aplicação;
- Um software desenvolvido usando uma ferramenta RAD usa menos recursos que um desenvolvido utilizando outras ferramentas de desenvolvimento;
- O uso de ferramentas RAD aumenta o planejamento / gerenciamento de um projeto de desenvolvimento de software;
- Com uma ferramenta RAD pode-se desenvolver aplicações de forma mais rápida do que usando outros tipos de ferramentas;
- Uma ferramenta RAD provê um alto nível de flexibilidade no desenvolvimento de uma aplicação;
- O uso de uma ferramenta RAD reduz a quantia de codificação necessária;
- Uma ferramenta RAD ajuda a categorizar e classificar objetos;
- Com uma ferramenta RAD o desenvolvimento de aplicação requer menos trabalho;
- Uma ferramenta RAD aumenta o reuso de código;
- Uma ferramenta RAD ajuda a desenvolver aplicações mais sustentáveis.

2.4 Trabalhos Relacionados

No que tange o reuso de software, a maioria dos trabalhos relacionados tem como foco o desenvolvimento de frameworks, que englobam os mais distintos ramos do desenvolvimento de jogos eletrônicos, dentre os quais pode-se destacar os seguintes:

- **Dispositivos móveis**: nesta área pode-se citar o FMMG proposto por [Kubo, 2006], com o objetivo de estabelecer uma arquitetura para jogos multiplayer móveis. Na mesma área, porém com um objetivo bem distinto, tem-se a ferramenta proposta por [Xu, 2008] que visa abstrair a camada de rede deste dispositivos, focando em pacotes UDP. Por fim, pode-se citar o framework "gRmobile" [Joselli and Clua, 2009] que tem por objetivo facilitar a utilização de acelerômetro e telas touch screen de smart phones;

- **Educacionais**: neste contexto o framework proposto por [Abdullah et al., 2008], que tem por objetivo ajudar a implementação de jogos educacionais voltando-se para a aplicação de conteúdos interativos. Em outra esfera, [Kochanski, 2009] propõe uma ferramenta específica para avaliar o impacto dos jogos educacionais na área de engenharia de software. Esta apoia a construção de experimentos, no intuito de verificar a eficácia do ensino através dos jogos;

- **Jogos Sérios**: nesta área existe o interesse especial na implementação do tipo de interação dos NPC, [Jepp et al., 2010] propõe uma ferramenta específica para a implementação de agentes modulares. Como outro exemplo tem-se o SmartSim [Silva et al., 2006], que visa a implementação de agentes inteligentes com capacidades estendidas, como personalidade e emoções.

- **TV digital**: no Brasil, o exemplo mais proeminente deste tipo de ferramenta é o GingaGame framework [Barboza and Clua, 2009], que tem por objetivo criar um arcabouço de classes que proporcionem a implementação de jogos eletrônicos no padrão brasileiro de TV digital.

Como pode-se observar os frameworks são um tema de interesse da comunidade científica, sendo alvo de diversos estudos.

Capítulo 3

Game AI

A Inteligência Artificial (IA) é uma ciência relativamente nova, advinda em sua essência da ciência da computação, tendo seu termo cunhado em 1956. Entretanto, se relaciona com as mais distintas áreas do conhecimento humano como filosofia, matemática, economia, neurociência, psicologia, engenharia, cibernética, linguística, dentre outras [Russel, 2004]. Dada esta pluralidade, uma definição única de IA se torna muito difícil, haja visto que este tema é abordado por diferente enfoques. No entanto, pode-se destacar duas linhas principais, a baseada no raciocínio que tem a seguinte definição: "*IA é o estudo das faculdades mentais através dos modelos computacionais.*" [Bittencourt, 2006]; e a focada no comportamento inteligente onde: "*A IA é parte da ciência da computação que compreende o projeto de sistemas computacionais que exibam características associadas, quando presentes no comportamento humano, à inteligência.*" [Bittencourt, 2006].

A IA aplicada em jogos eletrônicos, conhecida como *Games AI* (*Game Artificial Intelligence*), se enquadra na segunda linha, onde o algoritmo que controla os personagens, ou NPCs, tem o objetivo de simular um comportamento inteligente dado cenários com com múltiplas escolhas [de Santana, 2006] [Funge, 2004]. Nesta simulação, o NPC deve transparecer um nível de inteligência similar a humana, ter personalidade, cometer erros e ser capaz de fornecer diferentes níveis de dificuldade ao jogador, no intuito de aumentar a experiência e imersão do jogo e melhorar sua jogabilidade [Schwab, 2004].

Estas técnicas começaram a ser aplicadas nos jogos desde seus primórdios. No início da década de setenta havia a "programação de jogabilidade", que consistia no controle autônomo de NPCs focadas principalmente nos movimentos, que podiam ser aleatórios

ou seguir padrões. Essa limitação no comportamento era causada pelas restrições de memória e velocidade de processamento da época [Schwab, 2004].

Os problemas encontrados na indústria de jogos eletrônicos são considerados "*Killer Aplication*" para o campo da IA, tendo vista a necessidade de respostas em tempo real e de comportamento muito similar ao dos humanos (*Humam-Level AI*), acrescido da grande variedade de gêneros de jogos existentes, assim como comportamentos dos personagens, oferecendo um ambiente rico para pesquisa em IA [Funge, 2004].

A IA ainda hoje é um desafio para a indústria de jogos, principalmente pelos seguintes fatores: período de desenvolvimento, teste dos algoritmos de aprendizagem e poder de processamento. O curto período de desenvolvimento dos jogos dificulta o aprendizado dos desenvolvedores para tecnologias que envolvem a IA e suas aplicações [Bourg and Seemann, 2004]. Os resultados apresentados pelos algoritmos de aprendizado são difíceis de serem testados e depurados, tendo em vista a imprevisibilidade dos mesmos. Em vista disso, os desenvolvedores têm preferência por técnicas mais simples e determinísticas. Dado a necessidade de execução em tempo real, algumas técnicas de IA, em vista de seu alto custo de computacional, ainda não podem ser implementadas em sua plenitude.

Existem diversos modelos de Game AI. Todavia, neste trabalho, será utilizada o proposto por [Millington, 2006], ilustrado na Figura 3.1 onde a utilização de IA é subdividida em três níveis: movimento, tomada de decisão e estratégia.

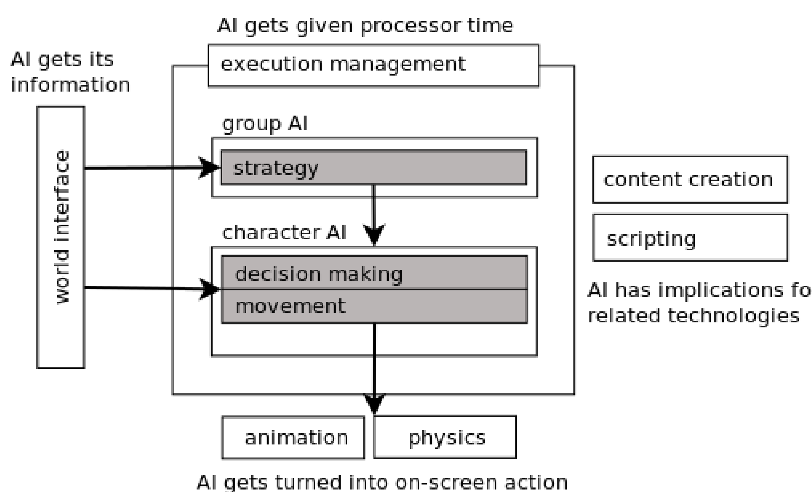

Figura 3.1: Modelo de Game AI (adaptada de [Millington, 2006]).

- **Movimento**: neste grupo encontram-se as técnicas que atuam sobre o conjunto de decisões que fazem com que um personagem se desloque de um ponto a outro do ambiente, evitando eventuais obstáculos. Estas são conhecidas como *pathfinding*, tendo o algoritmo A* como seu representante mais proeminente, dado sua eficiência e facilidade de implementação;

- **Tomada de decisão**: neste caso, pretende-se prover o personagem da capacidade de decidir que ação realizar mediante informações que tem a disposição, tanto do ambiente quanto de si próprio. Normalmente, cada NPC tem um conjunto de comportamentos, dos quais deve decidir qual é o mais apropriado a ser empregado dado a situação atual do jogo. Diversas técnicas podem ser utilizadas para tal objetivo: Máquinas de estados finitos, *Goal-Oriented Action Planning*, Árvores de decisão, Sistemas de inferência fuzzy, processos de decisão de Markov, *Goal-Oriented Behavior* e Sistemas baseados em regras. Todavia as FSMs são as mais empregadas atualmente pela indústria [Millington, 2006];

- **Estratégia**: utilizada para coordenar grupos de NPCs a realizarem determinada tarefa. Esse tipo de algoritmos não é voltado para controlar um personagem, mas para influenciar o comportamento do grupo como um todo. Cada personagem vai utilizar sua camada de tomada de decisão e movimento no intuito de contribuir para que a estratégia seja bem sucedida. Dentre as técnicas que podem ser implementadas nesta camada, destacam-se: *Waypoint tactics*, *Tatical Analyses*, *Tactical Pathfinding* e *Coordinated Action*.

Este trabalho propõe um ambiente de desenvolvimento da camada de tomada de decisão dos NPCs, onde esta é regida por uma FSM, agregando as técnicas de IA de Algoritmos Genéticos, Lógica Fuzzy e Redes Neurais Artificiais. Assim, nas próximas seções cada uma destas técnicas será apresentada.

Diversas outras técnicas inteligentes podem ser utilizadas na camada de tomada de decisão, e foram preteridas neste trabalho. Contudo, uma das características principais de framework é a sua estensibilidade, havendo a possibilidade de que as mesmas venham a ser agregadas ao FSG em trabalho posteriores.

3.1 Máquinas de Estados Finitos

As Máquinas de Estados Finitos (*Finite State Machines - FSMs*) se originaram na matemática, mais especificamente na teoria da computabilidade e da complexibilidade [Smed and Hakonen, 2006]. O seu modelo computacional teve grande impacto na industria dos jogos eletrônicos, sendo amplamente utilizado desde seus primórdios até os dias atuais. Normalmente é peça chave na implementação da camada de tomada de decisão dos NPCs presentes em um jogo, sendo parte importante do que é conhecido como *Game AI* [Bourg and Seemann, 2004].

Neste contexto, as FSMs são um modelo de comportamento composto por três elementos fundamentais: um conjunto finito de estados *S*, de eventos de entrada *I*, e funções de transição *T(s,i)*. Onde a cada estado pertencente a *S* representa um determinado comportamento do personagem, tendo em vista que o estado atual da máquina é único. O conjunto *I* dos possíveis eventos representam os estímulos que o NPC pode vir a receber do ambiente. Por fim, as funções transição são responsáveis por definir quais condições devem ser satisfeitas para que a máquina venha a transitar do seu estado atual para um outro [Rabin, 2002] [Alberto, 2009].

Em seu modelo clássico, a FSM é representada por um grafo onde os vértices representam os estados e as arestas as transições, como ilustrado na Figura 3.2.

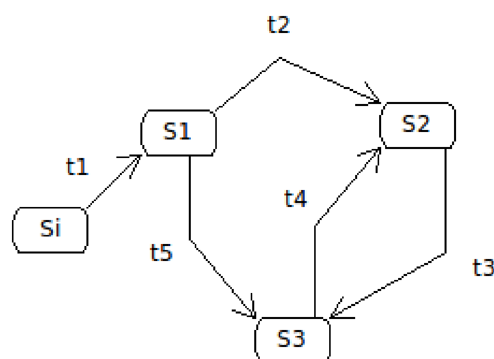

Figura 3.2: Representação gráfica de uma FSM (adaptada de [Bourg and Seemann, 2004]).

Na FSM apresentada na Figura 3.2, tem-se o conjunto dos possíveis estados *S = Si, S1,S2,S3*, e seu respectivo conjunto de transições *T = t1,t2,t3,t4,t5*. Neste exemplo, a máquina inicia no estado *Si* e permanece nele até que receba um evento de entrada

associado a transição *t1*, caso esta tenha suas condições atendidas, ocorrerá a transição para o Estado *S1*, e assim sucessivamente.

Existem duas formas principais de classificação desta técnica, sendo conhecidas como Máquina de Moore e a Máquina de Mearly. Na Máquina de Mearly as saídas da FSM são decorrentes das transições entre os estados, enquanto que na Máquina de Moore as saídas são geradas como produto do estado atual. As FSMs empregadas em jogos se enquadram no conjunto das Máquinas de Moore, tendo em vista que a saída do sistema, neste caso, são as ações que o personagem vai realizar e normalmente, codificadas dentro dos próprios estados [Dinízio and Simões, 2003] [Bourg and Seemann, 2004].

A longevidade e sucesso das FSMs, no desenvolvimento de jogos, se deve a diversos fatores, mas podem-se destacar os seguintes [Rabin, 2002] [Alberto, 2009] [Malfatti and Fraga, 2006]:

- **Menor tempo de desenvolvimento**: tendo em vista que são conceitualmente simples, o seu projeto e implementação são relativamente rápidos, além de facilmente extensíveis. Inclusive tendo um padrão de projeto, chamado State, específico para sua implementação.

- **Baixa curva de aprendizagem**: sua simplicidade faz com que o tempo que um desenvolvedor precisa para dominar a técnica seja, em média, curto.

- **Previsibilidade**: dado um conjunto de entradas e um estado atual conhecido, a transição de estados pode ser facilmente prevista, facilitando a tarefa de testes e manutenção do software.

- **Baixa manutenção**: a sua implementação ocorre através de subdivisão de seu código fonte. Normalmente tem-se uma classe para cada estado e nesta métodos referentes as possíveis transições. Assim, caso o NPC implementado por esta máquina apresente um comportamento considerado inesperado, é relativamente fácil a localização do problema.

- **Baixo custo computacional**: somente o código do estado atual é executado a cada momento, além de um conjunto pequeno de lógica para determinar o estado atual e as possíveis transições.

- **Ferramenta de comunicação**: elas são suficientemente simples para que pessoas sem conhecimento técnico de programação, tais como designers ou artistas, as compreendam.

No intuito de demostrar o emprego das FSMs em jogos, apresenta-se na Figura 3.3, um exemplo completo de um modelo de NPC através desta técnica. Neste caso o personagem é um soldado que deve vigiar e defender um território. Inicialmente ele se encontra em sua base (estado "Homing") e uma vez que sua energia vital esteja no auge, ele inicia sua patrulha (estado "Patrolling"). Na sequência, dado o seu nível de energia e a possibilidade de surgirem inimigos, ele vem a transitar para os outros estados, moldando assim o seu comportamento.

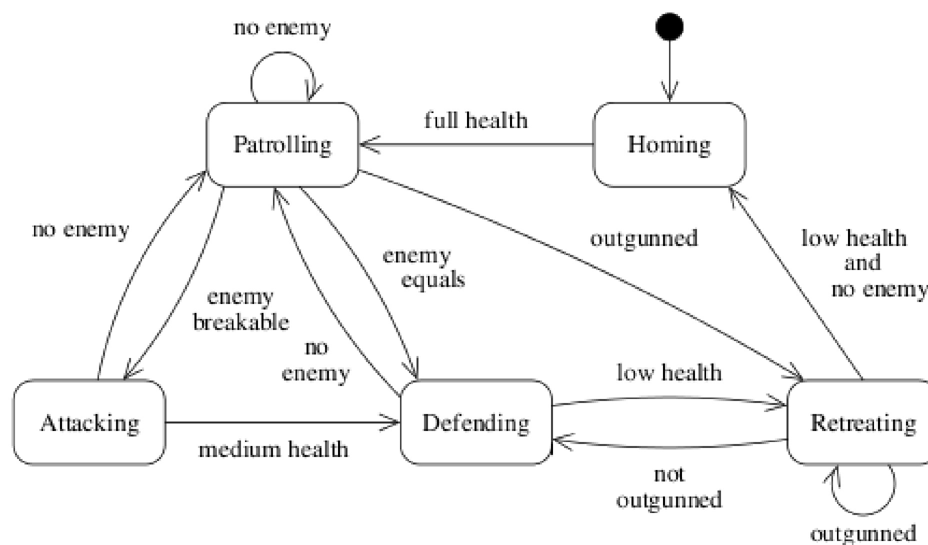

Figura 3.3: FSM de um soldado (NPC) (adaptado de [Smed and Hakonen, 2006]).

O modelo computacional de FSM possui diversas definições. Todavia, a amplamente empregada pela indústria é o diagrama de transição de estados da UML (Figura 3.3), que é basicamente constituído dos seguinte componentes [Larman, 2004] [Booch et al., 2000]:

- **State**: descreve o estado interno do NPC em determinado instante de tempo, além de estar relacionado a uma atividade e diversas ações.

- **Activity**: é uma execução contínua e não atômica associada a um estado, representada pelo label "do". Nela que tem-se implementado o comportamento do NPC. Esta somente é interrompida quando da transição para um próximo estado.

- **Action**: é uma execução atômica que é invocada quando de uma mudança de estado. Normalmente associado a um estado tem-se a ação "entry" que é executada quando se entra no estado e a "exit" que, por sua vez, é invocada quando de sai do mesmo. Ainda para cada transição, opcionalmente, pode-se ter uma ação relacionada.

- **Transition**: é uma relação entre dois estados, indicando que um NPC pode sair do "Estado 1" para um "Estado 2" dado a ocorrência de um evento. Opcionalmente, cada transição pode ter uma condição de guarda além de uma ação.

- **Event**: estímulo do ambiente que pode desencadear uma transição do estado atual.

- **Guard condition**: condição que deve ser verdadeira para que uma transição disparada por um evento venha a acontecer.

- **Start state**: estado, representado por um círculo preto preenchido, que indica por onde a FSM inicia sua execução.

- **Final state**: estado, normalmente representado por um círculo vazado, que representa o fim da execução da FSM.

Com o objetivo de elucidar o funcionamento dos componentes apresentados acima, na Figura 3.4, tem-se um detalhamento de como ocorre a transição do estado "Patrolling" para o estado "Attacking" da FSM apresentada na Figura 3.3.

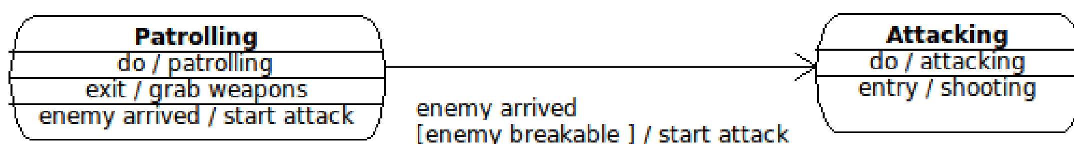

Figura 3.4: Transição entre o estado Pattrolling e Attacking (adaptado de [Madsen et al., 2012]).

A transição apresentada na Figura 3.4 ocorrerá de acordo com os seguintes passos:

1. O estado "Patrolling" percebe a ocorrência de um evento "enemy arrived" ao qual tem-se associado a transição para o estado "Attacking". Esta, por sua vez, possui a condição de guarda "enemy breakable", que caso seja atendida possibilita a ocorrência da transição.

2. A atividade “patrolling” que estava sendo realizada pelo estado “Patrolling” é interrompida.

3. É executada a ação de saída “grab weapons” do estado “Patrolling”.

4. É executada a ação “start attack” associada a transição.

5. É executada a ação de entrada “shooting” do estado “Attacking”.

6. Por fim inicia-se a execução da atividade “attacking” do estado “Attacking”.

A FSM modelada na Figura 3.3 pode vir a ser implementada utilizando-se o padrão de projeto State, como ilustra a Figura 3.5. Nesta, observa-se que o ambiente pode (representado pela classe Game) lançar eventos a serem atendidos pelo NPC, que delega a tarefa ao seu estado atual.

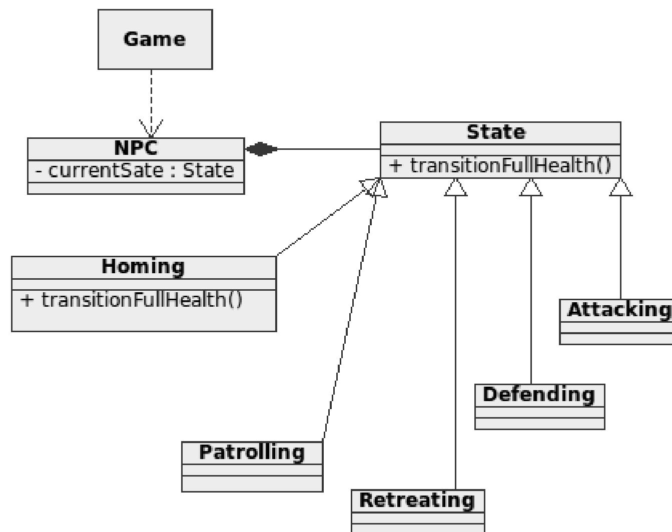

Figura 3.5: Exemplo de modelo de classes do padrão de projeto State.

3.2 Redes Neurais Artificiais

As Redes Neurais Artificiais (*Artificial Neural Networks - ANNs*) são a principal técnica do conexionismo, linha da IA que estuda a possibilidade de simulação de comportamento inteligente através de modelos matemáticos que buscam se assemelhar as estruturas neurais

biológicas. Estas redes se caracterizam por serem processadores maciçamente distribuídos, constituídos de um grande número de unidades processadoras simples, conhecidas como neurônios artificiais, que tem a capacidade de armazenar conhecimento experimental e torná-lo disponível para o uso. Adquirindo assim capacidade computacional através de aprendizado e generalização, dado um processo interativo com o meio externo [Rezende, 2005] [Haykin, 2001].

Esta técnica de IA vem sendo aplicada em diversas áreas com um grande sucesso, principalmente em problemas de aproximação, predição, classificação, categorização e otimização, como reconhecimento de caracteres, reconhecimento de voz, predição de séries temporais, modelagem de processos, visão computacional e processamento de sinais [Russel, 2004]. Todavia, as ANNs ainda têm um impacto pequeno na indústria de jogos eletrônicos [Darryl and Stephen, 2004].

Dentre suas principais características, pode-se destacar as seguintes [Rezende, 2005] [Haykin, 2001]:

- **Generalização**: capacidade de aprender através de exemplos e por sua vez generalizar de forma a reconhecer instâncias similares das quais nunca havia sido apresentadas;

- **Adaptabilidade**: possibilidade de adaptar seus pesos sinápticos de forma a absorver modificações no ambiente. Assim, uma rede que foi treinada para atuar em determinadas condições pode ser retreinada para lidar com modificações;

- **Tolerância a falhas**: capacidade de realizar seu objetivo mediante sinais com ruídos, ou mesmo perda de comunicação em parte da rede.

- **Auto-aprendizado**: não existe a necessidade do conhecimento de um especialista para tomar decisões. A ANN se baseia unicamente nos exemplos históricos que lhes são fornecidos;

- **Modelagem não linear**: o processo de mapeamento da rede neural envolve funções não lineares que podem cobrir um limite maior da complexidade do problema.

Modelo de Neurônio Artificial

Os modelos matemáticos de neurônios são descritos a partir de uma simplificação do neurônio biológico. Esta é formada por um corpo celular que contém o núcleo da célula, também conhecido como soma, um conjunto de dendritos, por onde os estímulos são recebidos e um axônio por onde os impulsos elétricos deste neurônios são propagados para o resto da rede [Bittencourt, 2006] [Russel, 2004] [Haykin, 2001], como apresenta a Figura 3.6.

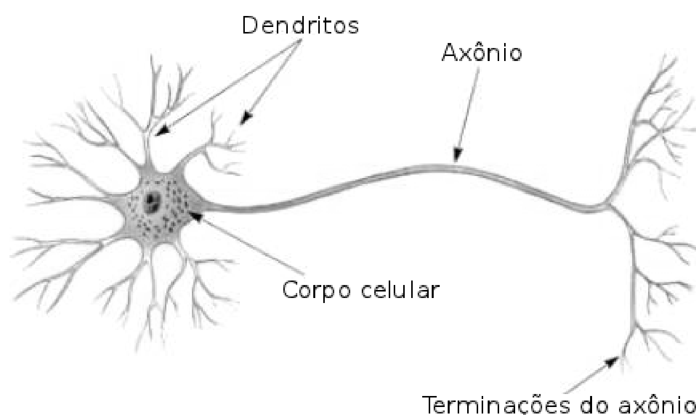

Figura 3.6: Simplificação do neurônio biológico (adaptado de [Packer, 2010]).

Baseado nesta simplificação, McCulloch e Pitts (MCP) propuseram um modelo de neurônio artificial, apresentado na Figura 3.7, que serve como base para todas as implementações contemporâneas de ANN.

O neurônio artificial ilustrado na Figura 3.7 é composto das seguintes estruturas:

- Conjunto de entradas ou elos de conexão, normalmente representado por um vetor de números reais $X_k = [x_1, x_2, ...x_m]$, por onde são recebidos os sinais dos neurônios interconectados. Cada elo tem associado a si um peso sináptico (w_{ki}), o qual define o grau de conexão entre os dois elementos. Caso seja positivo o sinal é amplificado, caso contrário é inibido e quando zerado não existe a conexão. Correspondendo no modelo biológico aos dendritos.

- Um integrador (net_k) para somar os sinais de entrada, ponderados pelos seus respectivos pesos, esta operação é conhecida como combinador linear (Equação 3.1). Fazendo as vezes de corpo celular do modelo biológico.

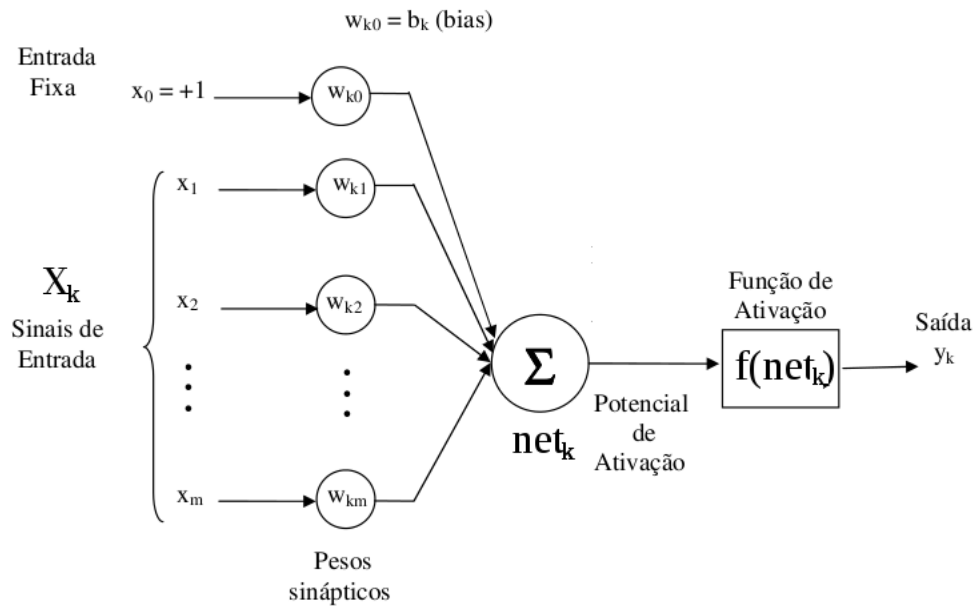

Figura 3.7: Modelo de neurônio artificial de McCulloch e Pitts (adaptado de [Haykin, 2001]).

$$net_k = \sum_{i=1}^{m} w_{ki}.x_{ki} \tag{3.1}$$

- Uma função de ativação ($f(net_k)$) para restringir a amplitude da saída do neurônio, normalmente definida pleo intervalo unitário fechado $[0, 1]$ ou alternativamente $[-1, 1]$. Correspondendo ao axônio no modelo biológico.
- Por fim, o peso *bias* aplicado externamente (b_k), tem como objetivo aumentar ou diminuir a entrada da função de ativação, dependendo se ele é positivo ou negativo, respectivamente.

Apesar do MCP não ser mais empregado, a imensa maioria dos modelos de neurônios artificiais da atualidade são fortemente inspirado nele, destacando-se o *Perceptron* e o *Adaline* [Haykin, 2001].

Aprendizado

O aprendizado de uma ANN é realizado através da modificação de seus pesos sinápticos de uma forma ordenada, dado um algoritmo de treinamento. Estes por sua vez, podem

ser classificados em dois paradigmas distintos, o supervisionado e o não supervisionado [Haykin, 2001].

- **Supervisionado**: neste caso, as saídas desejadas são conhecidas durante o treinamento, e um programa chamado indutor monitora o erro, diferença entre o valor desejado e saída propagada, e realimenta a rede com o erro no intuito de minimizá-lo até um limiar aceitável. Um exemplo desta técnica é o algoritmo *backpropagation*, que pode ser aplicado sobre redes *Perceptron* de múltiplas camadas.

- **Não Supervisionado**: neste paradigma não existe o papel do indutor nem da realimentação, a ANN tão somente estabelece correlações entre os padrões de entrada para categorizá-los de acordo com as classes autodescobertas. Redes que se valem desse tipo de treinamento são conhecidas como mapa auto organizáveis (*Self-Organizing Maps* - SOM).

Multilayer Perceptron (MLP)

Este trabalho irá se ater as redes *multilayer perceptrons feedforward* totalmente conectadas, dado sua utilização em jogos [Sweetser and Wiles, 2002].

Estas redes neurais se caracterizam por disporem seus neurônios em múltiplas camadas, normalmente uma camada de entrada, diversas camadas ocultas, e por fim uma camada de saída. O neurônio de cada camada se conecta com todos os neurônios das camadas imediatamente posterior e anterior. Assim, dado um sinal de entrada, o mesmo se propaga, de acordo com o integrador e função de ativação de cada neurônio, camada por camada até que um conjunto de saída é produzido como resposta final da rede [Haykin, 2001]. Redes MLP com três camadas tem a capacidade de aproximar qualquer função não-linear. Ao encontro deste fato, normalmente as informações que um NPC captura do ambiente de jogo são não-lineares [Chellapilla and Fogel, 1999], justificando a sua utilização em jogos.

Finalmente, o aprendizado deste tipo de rede ocorre de forma supervisionada através do algoritmo *backpropagation*. Este é constituído de dois passos fundamentais, a propagação da rede e a retropropagação do erro. No primeiro passo, um dado sinal de entrada é propagado por toda a rede e a saída resultante é armazenada. Nessa etapa, os pesos

sinápticos permanecem inalterados. Na segunda etapa, a saída da rede é comparada com valores que o indutor conhece como verdadeiro para a entrada apresentada, através da diferença entre esse dois conjuntos é gerado um valor de erro. De posse deste erro, o indutor retropropaga o mesmo através da rede, ajustando seus pesos sinápticos com o objetivo de movê-los para mais perto da solução desejada. E assim, sucessivamente, até que a rede venha a convergir, tendo o erro de saída dentro de um limite aceitável [Haykin, 2001].

Uma questão que surge é a possibilidade de durante o treinamento a MLP ficar presa em mínimos locais, não apresentado assim o resultado ideal. Porém, no contexto dos jogos eletrônicos, isso não é necessariamente um problema, tendo em vista que não se espera que o NPC tenha um comportamento ótimo ou perfeito. Eventuais erros, desde que menos previsíveis, também são interessantes.

3.3 Lógica Fuzzy

Os conjuntos ou números fuzzy são os modelos mais conhecidos para tomada de decisão a partir de informações imprecisas ou vagas, tendo sido proposto originalmente por [Zadeh, 1965]. Seu processo de decisão é baseado em elementos conhecidos como variáveis linguísticas. Estas por sua vez, constituídas por um conjunto de termos linguísticos, que buscam simular representações utilizadas no pensamento humano [de Barros and Bassanezi, 2006].

Frente a dados incertos, não é mais interessante que a informação seja representada por números reais ou "crisp", mas sim por conjuntos fuzzy ou nebulosos. Para manipulação destes novos conjuntos, utiliza-se a lógica fuzzy que permite que termos linguísticos, uma vez dada sua conversão em funções de pertinência (*membership function*), possam ser processados numericamente [de Barros and Bassanezi, 2006].

A diferença fundamental entre a lógica convencional e a lógica fuzzy, é que enquanto no primeiro caso as funções de pertinência são bivalentes, ou seja, ao se verificar se um elemento pertence ou não a um conjunto, tem-se como resultado um valor verdadeiro ou falso. Em contrapartida, no segundo caso as funções de pertinência podem retornar qualquer valor dentro do intervalo fechado [0,1]. Em vista disso, um elemento pode ser parcialmente membro de um conjunto [Simões and Shaw, 2007].

Assim, as funções de pertinência atribuem graus de pertinência fuzzy (*degree of truth*) para valores de uma dada variável crisp, pertencente a um universo de discurso. Este universo representa o intervalo de todos os possíveis valores reais, que a variável pode vir a assumir. Estas funções tomam as mais diversas formas. Todavia, as funções triangular e trapezoidal são as mais empregadas, dado sua simplicidade e forma intuitiva [Ganga, 2010] [Simões and Shaw, 2007].

Um exemplo deste tipo de função é ilustrado na Figura 3.8. Esta define o conjunto fuzzy referente ao termo linguístico "pessoas adultas". Desta forma, alguém com 35 anos de idade (variável crisp) pertenceria a este conjunto com grau de pertinência 0.8, já alguém com 50 anos de idade pertenceria a este conjunto com grau de pertinência 0.2.

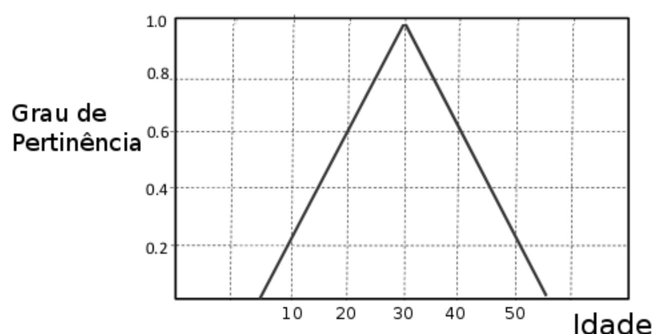

Figura 3.8: Exemplo de função de pertinência triangular que define o conjunto Fuzzy dos adultos (adaptado de [Ganga, 2010]).

Por fim, como mencionado anteriormente, as variáveis linguísticas são expressas por conjuntos de termos linguísticos inter-relacionados. Um exemplo deste tipo de variável, já no contexto dos jogos, é apresentado na Figura 3.9, denominada "Poder de Ataque", representando com que força um dado NPC pode desferir um ataque num oponente a partir de sua energia vital. Assim, um personagem com 55 pontos de energia poderia desferir um ataque que seria considerado "forte" ou "muito forte".

Sistemas de Inferência Fuzzy

Os Sistemas de Inferência Fuzzy (*Fuzzy Inference Systems - FIS*) são constituídos fundamentalmente através de quatro módulos: um processador de entrada responsável pela fuzzificação dos dados, uma base de regras, uma máquina inferência e por fim um defuzzificador. Estes módulos ficam dispostos no sistema de acordo com a Figura 3.10

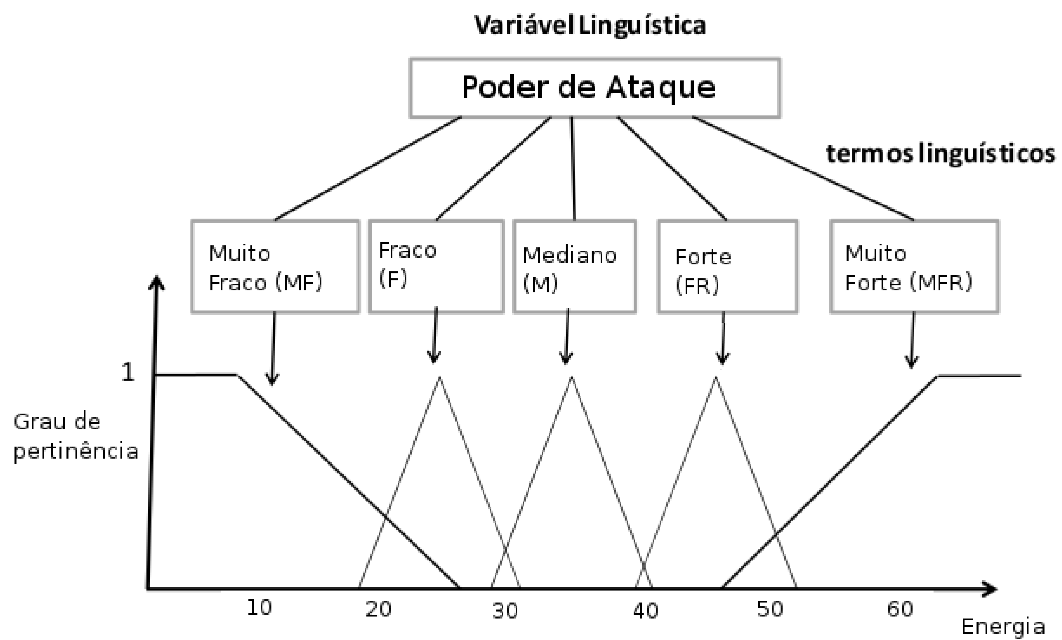

Figura 3.9: Variável linguística "Poder de Ataque" (adaptado de [Ganga, 2010]).

[de Barros and Bassanezi, 2006] [Ganga, 2010] [Simões and Shaw, 2007].

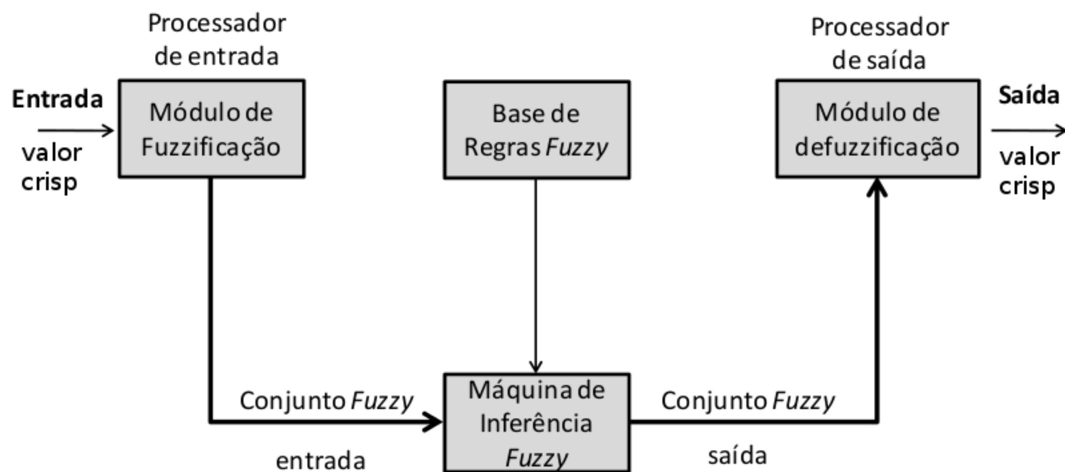

Figura 3.10: Módulos de um sistema fuzzy (adaptado de [Ganga, 2010]).

- **Processador de entrada (Fuzzificação)**: este componente realiza o mapeamento entre os valores das variáveis crisp de entrada, para graus de pertinência nos termos linguísticos.

- **Base de Regras**: é neste módulo que o conhecimento do especialista é catalogado

SE Poder de Ataque = Forte E Poder de Ataque = Muito Forte ENTÃO Dano = Grande

Tabela 3.1: Exemplo de regra de produção (adaptado de [Simões and Shaw, 2007]) .

através de regras de produção, desempenhando o papel de relacionar logicamente as informações que constituem base de conhecimentos do sistema fuzzy.

Cada regra de produção possui o seguinte formato "SE <antecedentes> ENTÃO <consequente>", onde cada antecedente e cada consequente assume valores definidos por variáveis linguísticas. Os antecedentes são um grupo de condições que quando satisfeitas, mesmo que parcialmente, determinam o processamento do consequente por meio de um mecanismo de inferência difusa. Já os consequentes são um conjunto de ações ou diagnósticos que são gerados pela regra ativada.

Um exemplo de regra de produção é apresentado na Tabela 3.1,baseado na variável linguística ilustrada na Figura 3.9, onde "Dano" é uma variável linguística que define o quanto um inimigo irá sofrer com o ataque desferido.

- **Máquina de Inferência**: este componente é o responsável pelo processamento fuzzy, onde cada regra de produção é traduzida matematicamente por meio de operações dos conjuntos fuzzy. Os dois principais métodos de inferência que podem ser utilizados nestas máquinas são o método de Mamdani e o método de Kang-Takagi-Sugen, tendo como diferença básica a forma de defuzzificação [de Barros and Bassanezi, 2006] [Vermaas, 2010].

- **Processador de Saída (Defuzzificação)**: tem por objetivo transformar o termo linguístico, obtido pela agregação das saídas da máquina de inferência, em um número real ("crisp"), o qual é utilizado como a saída efetiva do sistema fuzzy. Os métodos mais conhecidos para a defuzzificação de funções de pertinência são o Centro-da-Área (C-o-A) (*Center of Area*), Centro-do-Máximo (C-o-M) (*Center of Maxima*) e por fim a Média-do-Máximo (M-o-M) (*Mean of Maxima*).

A grande utilização dos FIS na tomada de decisão se deve a diversos fatores, todavia pode-se destacar os seguintes [Vermaas, 2010] [Simões and Shaw, 2007]:

- Essa técnica é comparativamente simples, além de possuir um grande espectro de aplicabilidade.
- Suas regras de produção são de fácil compreensão. O efeito e o resultado de cada regra pode ser facilmente interpretado.
- Todas as funções de pertinência podem testadas individualmente, simplificando assim a manutenção.
- Tanto funções lineares quanto não lineares podem ser implementadas por um sistema baseado em regras, usando o conhecimento de um especialista formulado em termos linguístico.
- Eles são inerentemente confiáveis e robustos. Uma falha parcial no sistema não necessariamente deteriora significativamente a performance.

3.4 Algoritmos Genéticos

Os Algoritmos Genéticos (*Genetic Algorithms - GAs*) são modelos computacionais inspirados na teoria da seleção natural e na transmissão das características genéticas dos genitores aos descendentes, tendo como objetivo resolver problemas de otimização [Linden, 2008].

Baseado na ideia de que a partir de uma população, os indivíduos mais bem adaptados possuem maior probabilidade de se reproduzir, concebendo uma próxima geração mais apta que a anterior. Onde cada indivíduo da população representa uma possível solução para o problema estudado. Assim, são caracterizados como uma busca direcionada em um espaço de soluções, onde a direção é dada por informações pertinentes ao problema, diferindo de técnicas conhecidas como caminhadas aleatórias (*Random Walk*). Todavia, são probabilísticas e não determinísticas, dado sua grande dependência de fatores, como o sorteio da população inicial. Em vista disso, com um mesmo conjunto de parâmetros iniciais a cada execução se chega a resultados finais distintos.

Neste método, dentro de cada geração, existe a aplicação dos princípios de reprodução e seleção. Através da seleção, são determinados quais indivíduos poderão se reproduzir, tendo em vista a quantificação de sua aptidão, concebendo um determinado número de descendentes para a próxima geração.

Os GAs se caracterizam por serem algoritmos de otimização global, que empregam estratégias de busca paralela e estruturada, apesar de aleatória, direcionada ao ponto de "alta aptidão" ou mais próximo da solução ótima do problema.

Ao longo do tempo se mostraram muito eficientes para a busca de soluções ótimas, ou próximas das ótimas, em uma grande variedade de problemas, tendo em vista que não apresentam as limitações dos métodos de busca e otimização tradicionais. Podendo-se destacar as seguintes vantagens [Rezende, 2005]:

- Trabalham sobre uma codificação do conjunto de parâmetros que devem ser otimizados e não com os valores originais destes;
- São altamente paralelizáveis, pois atuam sobre uma população inteira de soluções e não sobre conjuntos específicos;
- Se valem de informações custo ou recompensa e não derivadas ou conhecimentos auxiliares;
- Utilizam regras de transição probabilísticas e não determinísticas;
- Durante o processo de otimização não se valem somente de informações locais, não correndo o risco de ficarem presos a máximos locais;
- Apesar de terem componentes aleatórios, usam a informação da população corrente para determinar o próximo estado de busca;
- Não são afetados por descontinuidades bruscas na função a ser otimizada ou em suas derivadas, tendo em vista que não usam informações de derivadas na sua evolução nem de informações sobre seu entorno;
- Tem a capacidade de lidar com funções discretas e contínuas.

Até mesmo os GAs mais simples tem a capacidade de resolver problemas complexos de uma maneira muito elegante.

Representação

A representação cromossômica é a uma forma de traduzir as informações do problema que deve ser otimizado para um formato que possa ser tratado pelo GA.

Neste contexto o cromossomo é uma estrutura de dados, composta por n unidades indivisíveis conhecidas como genes, que por sua vez, são definidos pelos alelos (tipo de dado e conjunto de seus possíveis valores) e locus (posição de um dado gene dentro de um cromossomo). Ao encontro do modelo biológico, cada gene representa uma característica específica. Por outro lado, diferindo deste modelo, nos GAs, um único cromossomo representa um indivíduo da população e por sua vez uma possível solução do problema [Rezende, 2005].

A definição da representação é uma etapa muito importante da construção de um GA, sendo necessários alguns cuidados: esta deve ser a mais simples possível; na existência de soluções proibidas, estas não podem ter uma representação; finalmente, caso o problema necessite de algum tipo de condição, ela devem ser incluídas na representação [Linden, 2008].

Apesar de existirem as mais diversas formas, a representação mais simples e amplamente utilizada é a binária. Neste caso, o cromossomo é definido como uma cadeia de bits, onde cada gene pode ter o valor 0 ou 1 (alelo) e seu locus é dado por uma posição nesta cadeia [Linden, 2008].

Por fim, em teoria, a representação é independente do problema a ser otimizado, pois uma vez tendo o cromossomo representado, por exemplo em binário, as operações padrões podem ser realizadas sobre o mesmo.

Seleção

O objetivo de um GA é o de, a partir de uma população inicial normalmente criada de forma aleatória, depois de várias gerações, alcançar uma população composta dos indivíduos mais aptos. Para tanto, ele deve se inspirar no modelo biológico, onde os indivíduos com maior aptidão geram mais descendentes. Todavia, os de menor aptidão também devem conseguir se reproduzir, apesar de ser em número reduzido, mantendo assim uma população geneticamente saudável, evitando uma convergência prematura.

Para que sejam distinguidos os espécimes mais aptos, cada cromossomo da população

recebe um valor de aptidão, também chamado valor de avaliação ou *fitness*. A função de aptidão (também conhecida como função objetivo ou *fitness*) é específica do domínio do problema que está sendo otimizado e gera o valor de aptidão. Em vista disso e da generalidade dos GA, esta função é a única ligação entre o algoritmo e o problema real. Os GAs são técnicas de otimização, assim esta função deve ser de tal forma que um dado cromossomo que seja uma solução melhor que um outro, tenha seu valor de avaliação necessariamente ser maior que este segundo [Linden, 2008].

Através do valor de avaliação, o método de seleção escolhe um subconjunto de indivíduos da população atual, gerando uma população de pais, que irão conceber a próxima geração. Tendo em vista que indivíduos com um bom valor de aptidão tendem a gerar descendentes tão aptos quanto eles ou ainda melhores.

Dentre os diversos métodos empregados, três se destacam: o Método da Roleta, o Método do Torneio e o Método da Amostra Universal Estocástica [Rezende, 2005].

- **Método da Roleta**: método mais simples e também o mais utilizado, no qual a seleção ocorre através de uma roleta onde cada indivíduo é representado por uma fatia proporcional ao seu valor de avaliação. Sendo assim, espécimes com maiores índices ocupam mais espaço na roleta, tendo maior possibilidade de serem selecionados. A roleta é girada o número de vezes igual ao número de indivíduos que se deseja selecionar. A cada giro da roleta, a seta aponta para um dado indivíduo;

- **Método do Torneio**: neste caso um subgrupo de indivíduos da população é selecionado aleatoriamente. O cromossomo com maior aptidão deste conjunto é selecionado para a população de pais. Assim, sucessivamente, até que a população de pais seja completada;

- **Método da Amostra Universal Estocástica**: este método, também conhecido como SUS (*Stochastic Universal Sampling*) é uma variação do método da roleta, onde ao invés de uma única agulha, n agulhas igualmente espaçadas são utilizadas, sendo n é o número de indivíduos a serem selecionados para a população intermediária. Assim, a roleta é girada uma única vez, exibindo menos variância que as repetidas chamadas do método da roleta.

Por fim, ainda existe a possibilidade de se separar de uma população um grupo de seus indivíduos mais aptos, e transportar os mesmos para a próxima geração, no intuito de beneficiar a convergência genética. Tal técnica é conhecida como elitismo.

Operadores Genéticos

Para que haja a reprodução, entre dois indivíduos em um GA, estão disponíveis dois operadores genéticos, o cruzamento (*crossover*) e a mutação. Estes operadores tem como objetivo assegurar que a nova população seja inédita, mas ainda assim herde características de seus genitores.

O cruzamento, operador genético predominante, é responsável pela recombinação dos genes dos pais, durante a reprodução, possibilitando assim que as próximas gerações herdem suas características. Existem diversas implementações deste operador, todavia três se destacam: o cruzamento de um ponto, o cruzamento de dois pontos e o uniforme [Linden, 2008].

- **De um ponto**: é o método mais simples e o menos eficiente. Nele, a posição de um gene é selecionada, os cromossomos dos genitores são divididos neste ponto. Em seguida, os descendentes são gerados através da concatenação cruzada destas duas partes dos pais;

- **De dois pontos**: muito similar ao de um ponto, diferenciando-se por utilizar dois pontos de corte. Sua operação é ligeiramente mais complexa que a anterior, no entanto a melhora no desempenho, normalmente faz com que o custo extra seja aceitável;

- **Uniforme**: neste caso, para cada gene é sorteado um valor o 0 ou 1, caso seja 0 o primeiro filho recebe o gene da mãe e o segundo o do pai, e vice versa. Esse método é o mais poderoso, gerando combinações de bits que os anteriores não teriam capacidade.

Já o operador mutação, tem associado a si uma probabilidade extremamente baixa de ocorrer, normalmente por volta de 0,5% a cada gene do cromossomo. Neste caso, todos os genes dos cromossomos filhos são percorridos, e havendo a mutação, seu valor original

é alterado. Todavia, esse operador é fundamental ao GA, pois garante a existência de diversidade genética na população, tendo em vista que o cruzamento contribui para a convergência genética. Caso o valor de mutação seja alto, o GA fica similar as técnicas conhecidas como *Random Walk*, na qual a solução é determinada de forma aleatória [Linden, 2008].

Convergência Genética

O critério de parada em um GA pode ser estabelecido de duas formas distintas: através de um número máximo de gerações ou ainda pela convergência genética.

A convergência genética ocorre quando a população tem uma baixa diversidade. Neste caso, muitos indivíduos possuem genes similares e apesar de continuarem se reproduzindo não há evolução significativa. A verificação desta convergência deve ser realizada através do genótipo (estrutura do cromossomo) do indivíduo e não através de sua função de aptidão, tendo em vista que indivíduos muito diferente podem ter valores de aptidão similares.

Existem diversos métodos para se determinar a convergência genética. Neste trabalho utiliza-se o algoritmo *K-Means* [Hartigan and Wong, 1979]. Este atua dividindo a população em K grupos, onde inicialmente são escolhidos K indivíduos aleatoriamente, definidos como centróides. Em seguida, os demais cromossomos são agrupados em torno destes centroides de acordo com sua proximidade, estabelecida pela distância euclidiana. Salientando-se que esta distância é calculada a partir dos valores das variáveis que estão representadas pelo cromossomo e não pelo valor de aptidão. Em uma próxima etapa, uma vez que todos os elementos da população estejam alocados em um grupo, para cada grupo é recalculado um novo centróide, havendo o reagrupamento dos indivíduos de acordo com esta nova posição. Esse processo se repete até que nenhum indivíduo troque de grupo. Uma vez esse processo tenha sido terminado, determina-se se houve convergência genética, primeiramente verificando-se se algum grupo excedeu um número máximo de indivíduos pré-determinado e ainda se a distância entre dois centróides é menor que um dado limite.

3.5 Trabalhos Relacionados

No que diz respeito as técnicas de IA apresentadas neste capítulo, existem diversos trabalhos que as empregam na construção de jogos, dentre as quais destacam-se os seguintes:

- **Sistemas de Inferência Fuzzy**: dentre os diversos empregos dentro dos jogos eletrônicos, [Rieder and Brancher, 2004] propõe a sua utilização jogos educacionais matemáticos, no intuito de criar tutores inteligente mais capazes de atuar no processo de ensino aprendizagem. Já [Sanches, 2012], a emprega em jogos de entretenimento, objetivando a tomada de decisão quanto utilização de armas e estratégia de deslocamento em ambientes. Por fim, [de Oliveira Cruz, 2011] estuda o emprego de lógica fuzzy para tomada de decisão de NPCs que levem em conta a simulação de emoções;

- **Algoritmos Genéticos**: neste contexto, tem-se o trabalho de [Tang and Wan, 2002] que aplica o GA no processo de autoaprendizado inteligente NPCs em ambientes virtuais, tendo em vista tarefas bem definidas. Já [Demasi and de Oliveira Cruz, 2003], propõe a sua utilização na evolução dos personagens dos jogos, de acordo com o desempenho do jogador, com o intuito de criar um acréscimo gradual de dificuldade. Ainda [Plant et al., 2008], explana sobre sobre a utilização de GA em jogos de futebol e posterior emprego em competições de robôs como a RoboCup. Finalmente, [Esparcia-Alcázar et al., 2010] demostra a utilização desta técnica na tomada de decisão de NPCs do jogo comercial *Unreal Tournament*;

- **Redes Neurais Artificiais**: [Castro and de Souza, 2010] aplicam esta técnica para reconhecimento de padrões das estratégias do jogador. Já [Victor et al., 2010] empregam ANN na tomada de ação em agentes cognitivos deliberativos. E por fim, o trabalhos de [Stanley et al., 2005], [Reeder et al., 2008] e [Charoenkwan et al., 2010] utilizam as ANNs em conjunto com GAs em métodos neuroevolutivos aplicados aos jogos.

Capítulo 4

Ambiente Proposto

O ambiente de desenvolvimento da camada de tomada de decisão de NPC proposto neste trabalho, se divide em um framework, que abrange as técnicas: FSM, GA, ANN e FIS. Também compõem o ambiente um grupo de ferramentas RAD para geração automatizada do código fonte.

Como foi apresentado na seção 2.4, existem diversos frameworks que englobam os mais distintos domínios dentro do desenvolvimento de jogos. Já em relação as ferramentas RAD não há um espectro tão abrangente, normalmente estes softwares se concentram nas questões de interface. Dentre estes, pode-se destacar o Gameka [de Faria Costa et al., 2011] voltando para o desenvolvimento de jogos 2D por não programadores, tendo como resultado um *bytecode* que necessita de um programa *runtimer* específico. Já o Visual Game [Barboza and da Silva, 2009] permite definir toda a lógica do jogo apenas através de componentes gráficos, sem necessidade de codificação, com saída gera um executável do sistema operacional Windows. Por fim, o SDK Gameplay [Motta et al., 2008] trabalha em conjunto com o framework GameplayLib, sendo focado na mecânica de jogo, gerando como resultado um arquivo XML com as regras do mesmo.

Contudo, um ambiente composto por um framework e RAD voltado para a tomada de decisão de NPCs não foi encontrado na literatura.

4.1 Framework FSG

Em consonância com a afirmação de Bittencourt et al. [Bittencourt, 2006] "*A melhor metodologia é a construção de ferramentas: a IA é suficientemente complexa para que seja contraproducente inicar cada novo programa a partir do zero. Ferramentas são fundamentais.*", o framework FSG (FURG Smart Games) tem o objetivo de agilizar a incorporação de técnicas de IA no desenvolvimento de jogos eletrônicos, assumindo que a camada de tomada de decisão de um NPC é de responsabilidade de uma FSM, técnica esta amplamente utilizada pela indústria [Bourg and Seemann, 2004].

Para tanto, sempre que a FSM recebe do ambiente algum evento que faça com que ela, em resposta, transite de seu estado atual para algum outro, a tomada de decisão que faz com essa transição se realize ou não, é regida por uma condição de guarda, podendo estar relacionada a uma técnica de IA. Tendo por objetivo final fazer com que o personagem tenha um comportamento menos determinístico, característica da FSM, e venha a responder de forma mais realista [Madsen et al., 2012].

Ao utilizar o framework, o desenvolvedor irá se deparar com duas formas de trabalho bem distintas. Primeiramente, no tocante a FSM implementada, dado o emprego do padrão de projeto State [Gamma et al., 2000], é caracterizado o uso intensivo de herança, havendo a necessidade de se implementar os métodos presentes nas super classes abstratas, bem como conhecer em que ordem e momento o FSG se valerá dos mesmos. Neste caso, sua utilização é evidenciada como framework do tipo caixa branca. Por outro lado, as técnicas de IA são providas por classes concretas, abstraindo assim a complexidade de sua implementação, limitando o trabalho de desenvolvimento a associar e agregar suas instâncias, caracterizando um framework caixa preta.

Entende-se que com esta forma de trabalho, o emprego do FSG é facilitado. Tendo em vista que é necessário ter um conhecimento mais aprofundado de FSM, que como mencionado na seção 3.1, são conceitualmente simples e com uma baixa curva de aprendizagem. Em contraposição, no tocante as técnicas de IA, que em sua maioria exigem uma maior curva de aprendizagem para implementação, o framework já realiza esse trabalho, restando ao desenvolvedor somente conhecer em quais momentos uma dada técnica deve ser utilizada, em detrimento de outra.

Dado que os jogos são implementados nas mais diversas linguagens e executados em plataformas distintas, optou-se pela proposta e implementação do FSG em uma *XML based language*, tendo suas estruturas baseadas na da linguagem de programação Java. Ficando a cargo de uma ferramenta RAD a geração deste XML e posterior tradução para uma linguagem de programação.

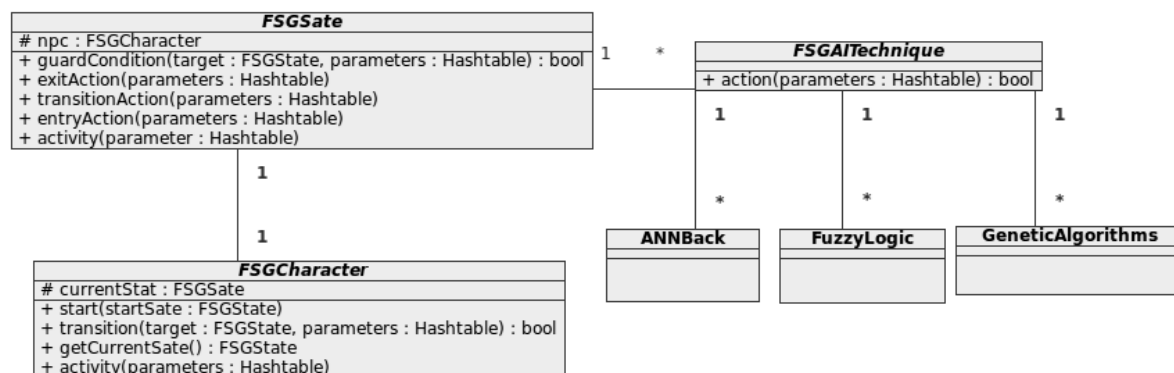

Figura 4.1: Diagrama de classes simplificado do FSG

Na Figura 4.1 é apresentado um diagrama de classes com o núcleo do FSG. Neste destacam-se três classes abstratas: FSGCharacter, FSGState e FSGAITechnique. A partir da classe FSGCharacter ocorre a implementação do NPC. Nela observa-se a presença do atributo "currentState" que denota em que estado a máquina se encontra, o método "transition" responsável pela a transição entre os estados e por fim no método "activity", o qual delega a responsabilidade do comportamento do personagem ao método de mesmo nome do objeto "currentSate". Já, a partir da classe FSGState, todos os estados presentes na FSM devem ser implementados, ressaltando-se o fato de que todas as ações, atividades e condições de guarda propostas no diagrama de estados da UML, estão previstas em seus métodos. Finalmente a classe FSGAITechnique, serve de interface entre as técnicas de IA e as transições de estado da FSM. Normalmente, seus objetos são instanciados dentro do método "guardCondition" de uma dada classe que herde FSGState.

De acordo com o que foi exposto acima e apresentado na seção 3.1, fica claro que a ferramenta, além estar de acordo com o modelo de estados da UML, apresenta uma implementação consistente do padrão de projetos State.

Por fim, observa-se na Figura 4.1 que as classes que dizem respeito as técnicas de IA foram suprimidas. Assim, as próximas três subseções serão dedicadas a discorrer sobre o

assunto.

4.1.1 Artificial Neural Network

A princípio, o conjunto de classes responsável por implementar ANN no FSG se restringe as apresentadas na Figura 4.2, da qual pode-se destacar as seguintes:

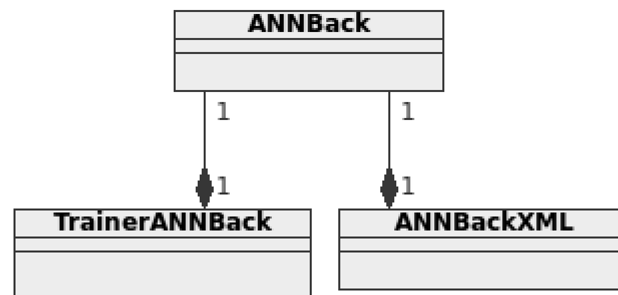

Figura 4.2: Diagrama de classes de ANN no FSG

- **ANNBack**: classe responsável pela implementação da rede neural, caracterizando-se por uma MLP com uma camada oculta, onde são definidos suas três camadas, seus neurônios, pesos sinápticos, métodos de propagação e retropropagação, além de função de ativação sigmoidal;
- **TrainerANNBack**: classe que tem como objetivo treinar uma dada rede através do algoritmo *backpropagation*, na qual é definido o erro aceitável, taxa de aprendizagem e o número máximo de épocas;
- **ANNBackXML**: dado uma rede previamente treinada, essa classe possibilita a geração e leitura de um arquivo XML com todas as características da rede, principalmente todos os seus pesos sinápticos. Possibilitando que mediante um arquivo XML se obtenha uma instância de ANNBack pronta para a utilização.

4.1.2 Algoritmos Genéticos

Fundamentalmente, o grupo de classes responsável pela implementação do GA é apresentada no diagrama de classes da Figura 4.3, no qual pode-se destacar as seguintes:

- **GA**: classe principal desta técnica da IA onde se encontram a população de cromossomos, além de ser responsável pela execução do algoritmo genético;

Figura 4.3: Diagrama de classes de GA no FSG

- **GAChromosome**: esta classe contém a representação cromossômica de cada indivíduo, que neste trabalho é binária. É composta de um conjunto de parâmetros do problema a ser otimizado, que são alocados sequencialmente em cadeias de bits;

- **GAParameter**: classe que define os parâmetros da função a ser otimizada, atuando sobre um dado intervalo e possuindo um determinado número de bits para sua representação junto ao cromossomo;

- **GARoulette**: esta classe implementa o método da roleta viciada com uma única agulha. A partir da população, esta classe gera uma roleta viciada de acordo com a função de aptidão de cada indivíduo e seleciona um a cada vez que é solicitada;

- **GACrossover**: classe abstrata com um único método "crossover", que tem como parâmetro dois indivíduos da população (instâncias de GAChromosome). A partir desta, tem-se as subclasses, GACrossoverOnePoint, GACrossoverTwoPoints e GACrossoverUniform, uma para cada tipo de reprodução;

- **GAMutation**: classe responsável pela mutação de algum gene dos descendentes oriundos do crossover. Na implementação realizada, existe uma taxa de mutação fixa. Todavia, a partir desta tem-se a subclasse GAMutationLinear, que aumenta linearmente sua taxa de mutação no decorrer das gerações;

- **GAGeneticConvergence**: classe abstrata com um único método "test" que re-

torna true/false caso a convergência genética tenha sido alcançada. A partir desta tem-se a subclasse GAGeneticConvergenceKmean, que implementa o algoritmo K-Mean;

- **GAXML**: classe que, ao receber uma instância de GA como parâmetro, gera um arquivo XML com toda sua especificação. Bem como, ao receber um arquivo XML como parâmetro, devolve uma instância de GA.

4.1.3 Lógica Fuzzy

A parte do framework FSG que implementa FIS é ilustrada no diagrama de classes da Figura 4.4, no qual pode-se destacar as seguintes classes:

Figura 4.4: Diagrama de classes do FSG referente a FIS

- **FIS**: classe abstrata responsável pela definição do sistema de inferência fuzzy, além de possuir a base de regras de produção. Neste trabalho o FIS é concretizado através da classe Mamdani;

- **Rule**: esta classe define uma regra de produção, composta por um conjunto de variáveis linguísticas, no intuito de implementar seus antecedentes e consequentes;

- **LinguisticVariable**: classe responsável pela implementação das variáveis linguísticas do sistema, sendo composta por um universo de discurso e vários termos linguísticos;
- **Universe**: classe que estabelece o universo de discurso de uma dada variável linguística;
- **LinguisticTerm**: esta classe implementa um dado termo linguístico, tendo em sua composição uma função de pertinência;
- **PertinenceFunction**: classe abstrata que define o formato das funções de pertinência, sendo implementada pelas classes PertinenceFunctionTriangular e PertinenceFunctionTrapezoidal;
- **FISDefuzzification**: classe abstrata responsável pela agregação dos graus de pertinência resultante das regras de produção e posterior defuzzificação. Sendo concretizada pela classe FITA, que implementa a técnica "First Infer, Then Aggregate" (primeiro infere depois agrega) do FIS Mamdani;
- **Defuzzification**: classe abstrata responsável pela técnica de defuzzificação dado um grau e uma função de pertinência. Sendo implementada pelas classes DefuzzificationCoA (*Center of Area*), DefuzzificationCoM (*Center of Maxima*) e DefuzzificationMoM (*Mean of Maxima*);
- **FISXML**: classe que, ao receber uma instância de FIS como parâmetro, gera um arquivo XML com toda sua especificação. Bem como, ao receber um arquivo XML como parâmetro, devolve uma instância de FIS.

Por fim, uma característica importante da ferramenta é a possibilidade, em todas as técnicas de IA implementadas, de geração de arquivos XML com a instância atual do objeto. Assim, por exemplo, todo o treinamento de uma rede neural pode ser realizado através de uma linguagem compilada, como C++, uma vez a rede tendo convergido, gerar o XML com todos os seus pesos sinápticos, para posterior utilização em outra linguagem, como um jogo Web implementado em PHP.

4.2 RAD (*Rapid Application Development*)

O ambiente FSG é composto de quatro ferramentas RAD, uma para cada técnica de IA, tendo como ponto de convergência a geração de código fonte, no padrão do framework FSG, em um *XML based language* específica e posterior tradução para uma linguagem de programação.

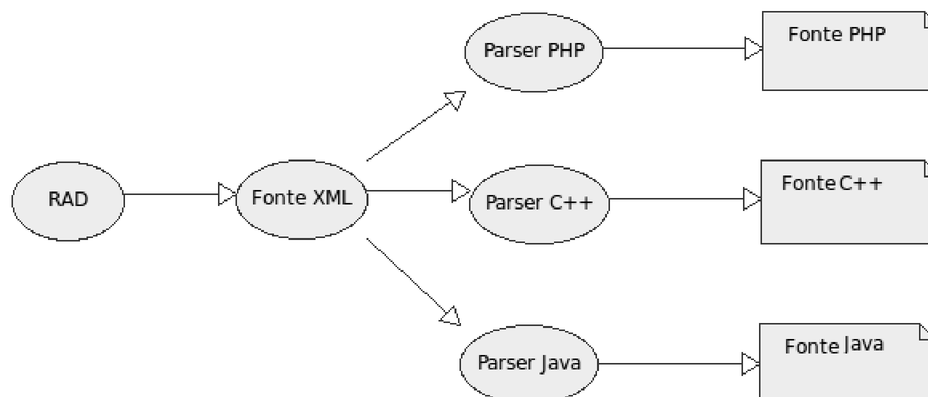

Figura 4.5: Processo de geração de código fonte a partir das ferramentas RAD.

Na Figura 4.5 é ilustrado esse processo de geração de código fonte, uma vez que a técnica de IA tenha sido configurada e validada, a ferramente RAD gera o seu respectivo algoritmo em XML. Por fim, tradutores (*parsers*) geram o código fonte na linguagem de destino.

Um exemplo desta linguagem XML pode ser visto na Tabela 4.2, bem como sua posterior tradução na Tabela 4.2.

Nas subseções a seguir serão apresentadas as ferramentas RAD para FSM, ANN, GA e FIS.

4.2.1 FSG - Finite-Sate Machine

No que diz respeito ao FSM, o RAD é responsável por sua edição gráfica e geração de código fonte no padrão do FSG. A edição ocorre através de um diagrama de estados UML simplificado, onde questões como ações, atividades e condições de guarda não são apresentadas, tendo em vista que já estão disponíveis na classe FSGState do framework, como ilustra a Figura 4.6. Ainda nesta figura pode-se observar a esquerda que o software permite, no caso de um estado, a especificação do nome da classe que irá herdar FSGSate,

```
<for variable=“i” >
   <forinit value=“0” />
     <expression>
       <left><variable>i</variable></left>
       <operator><![CDATA[ < ]]></operator>
       <right>
         <length>
           <variable vector=“true” >vetor</variable>
         </length>
       </right>
     </expression>
  <forincrement increment=“true” value=“1” />
   <code>
     <assignment name=“vetor” >
       <row><variable this=“false” >i</variable></row>
       <value>0</value>
     </assignment>
   </code>
</for>
```

Tabela 4.1: Um laço de repetição FOR na linguagem XML.

```
for ($i=0;$i<count($vetor);$i=$i+1) {
      $vetor[$i]=0;
}
```

Tabela 4.2: Tradução para PHP.

e no caso de uma transição, a definição do método que dispara a mesma, bem como o nome método responsável por implementar sua condição de guarda.

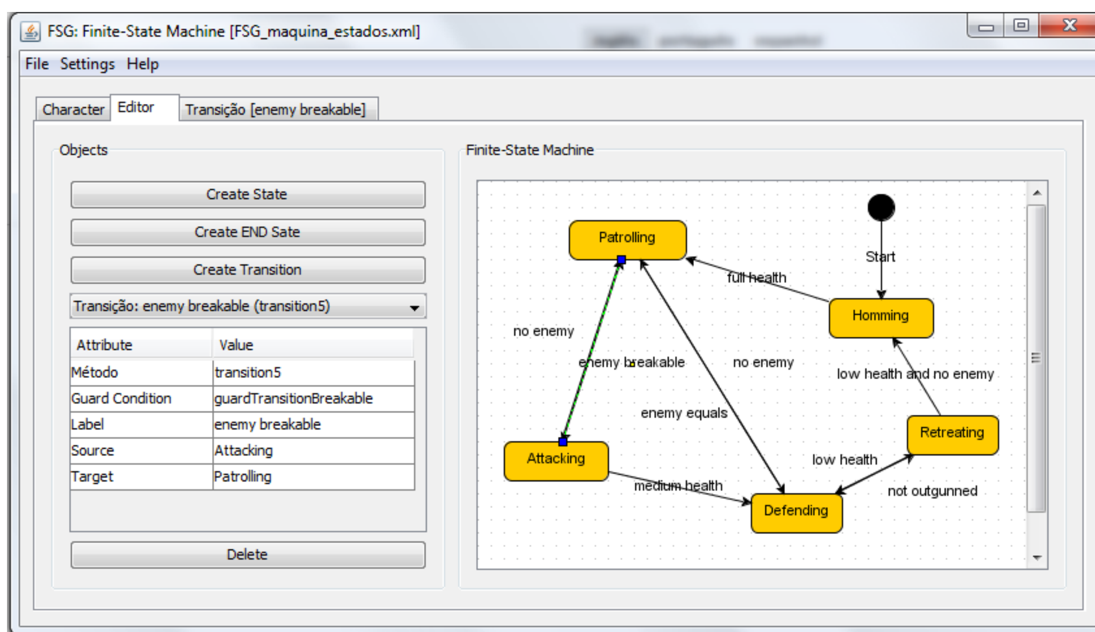

Figura 4.6: Edição de uma FSM na ferramenta RAD

Como mencionado na seção 4.1, existe a possibilidade de se relacionar a condição de guarda a uma técnica de IA específica, como observa-se na Figura 4.7. Para tanto, o desenvolvedor define no nome da classe que irá herdar FSGAITechnique, seleciona uma técnica e por fim carrega um arquivo XML com sua configuração. Neste exemplo, o arquivo deve conter a definição da rede neural artificial bem como todos os valores de seus pesos previamente treinados. É importante salientar que o RAD permite que outros algoritmos, além dos apresentados, venham a ser catalogados.

Em seguida, o desenvolvedor pode solicitar a geração código fonte do NPC na linguagem de programação de sua preferência, como é apresentado na Figura 4.8. Observa-se que a ferramenta identifica a utilização de uma técnica de IA relacionada a transição "enemy breakable". Assim, ela informa que os códigos referentes a implementação desta técnica também serão gerados.

A título de exemplo, o NPC "Soldier" apresentado na Figura 3.3 da seção 3.1, utilizando-se o o ambiente FSG teria um código fonte gerado de acordo com o diagrama de classes da Figura 4.9. É importante ressaltar que o RAD não gera o diagrama de classes.

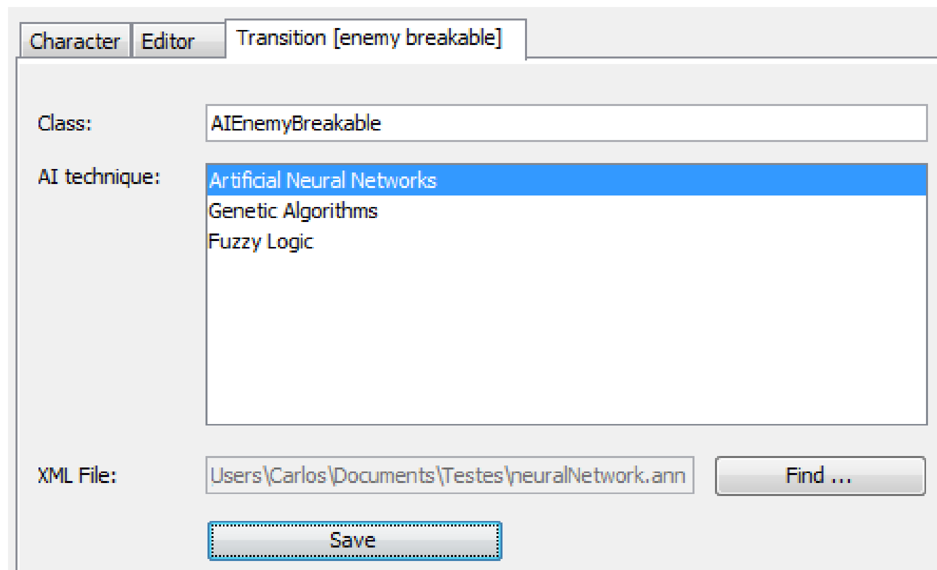

Figura 4.7: Relacionando uma transição com uma Técnica de IA.

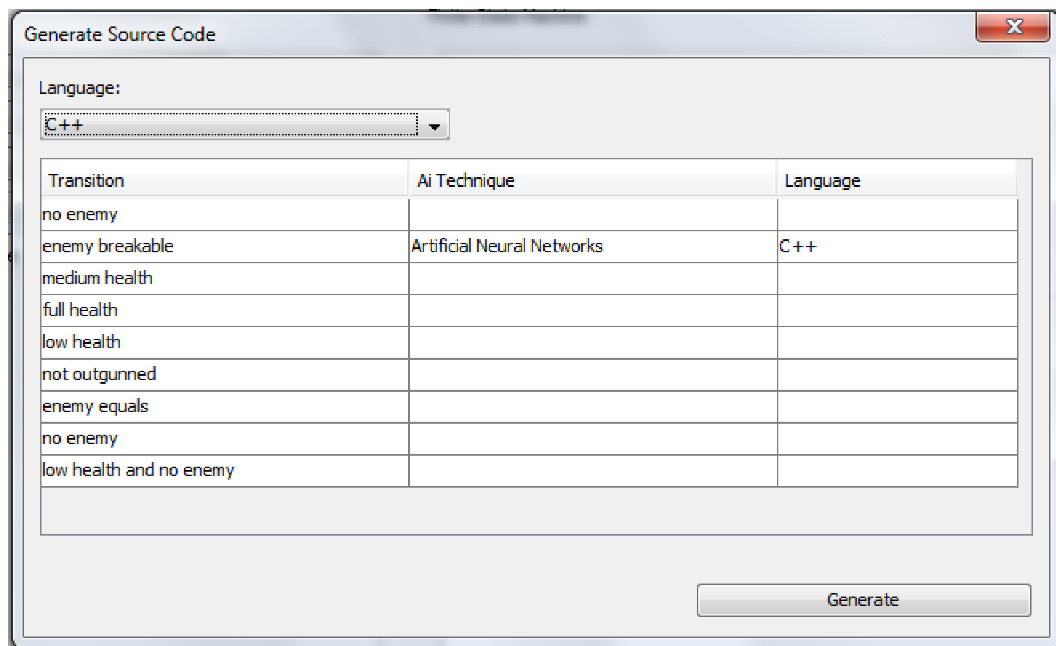

Figura 4.8: Gerando o código fonte do NPC.

4.2.2 FSG - ANNBack

No que diz respeito as ANNs, a ferramenta é responsável pela configuração da rede neural, realização do treinamento, e por fim geração de seu código fonte em uma dada linguagem de programação.

A primeira interface da ferramenta é ilustrada na Figura 4.10, sendo responsável principalmente pela configuração da estrutura da rede neural, onde destacam-se os seguintes

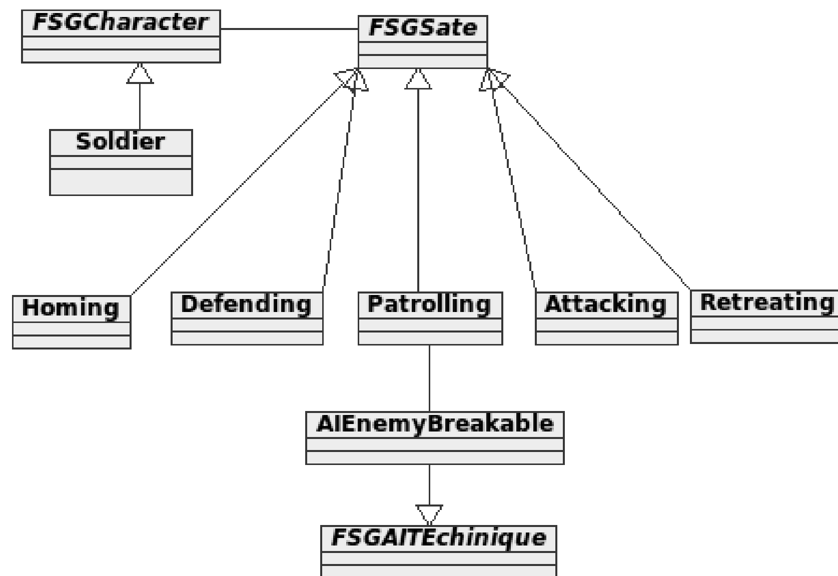

Figura 4.9: Diagrama de classes no padrão FSG do NPC "Soldier".

itens:

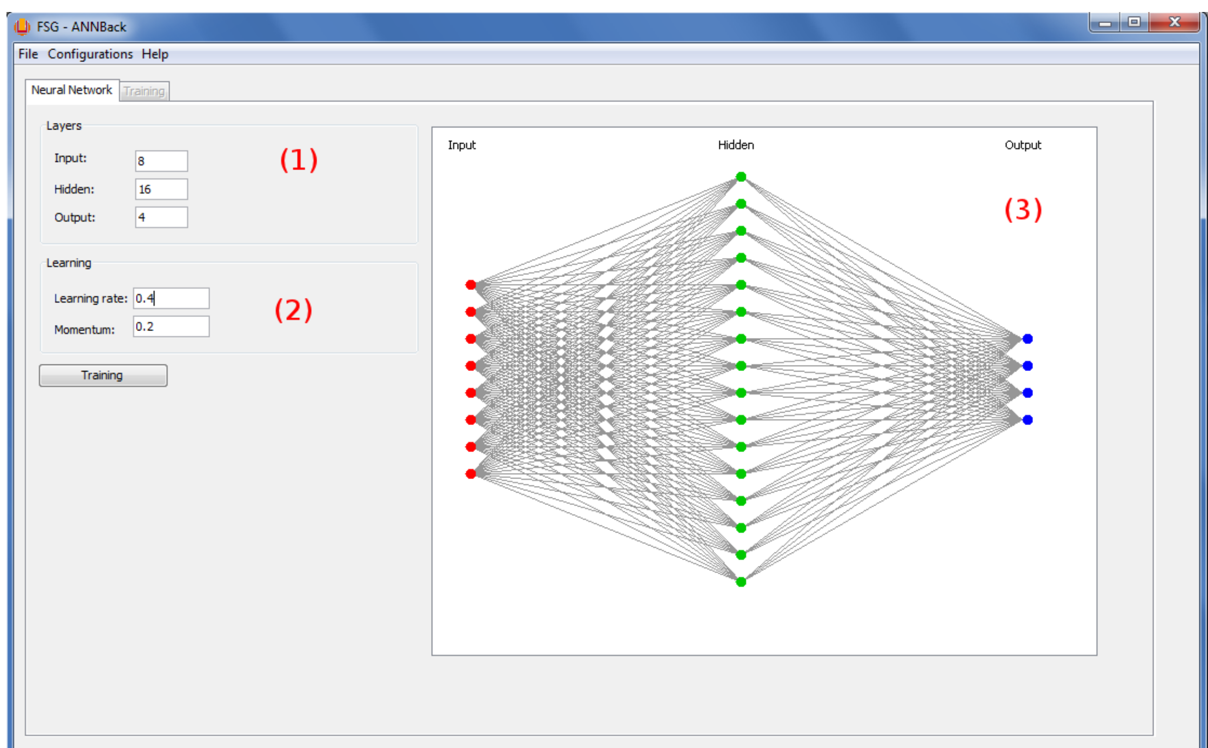

Figura 4.10: Interface inicial do RAD de redes neurais.

1. Definição do número de neurônios de cada camada da rede;

2. Especificação da taxa de aprendizagem e *momentum*;

3. Representação gráfica da rede.

Já na Figura 4.11 são apresentados, referente a configuração do treinamento da rede, os seguintes itens:

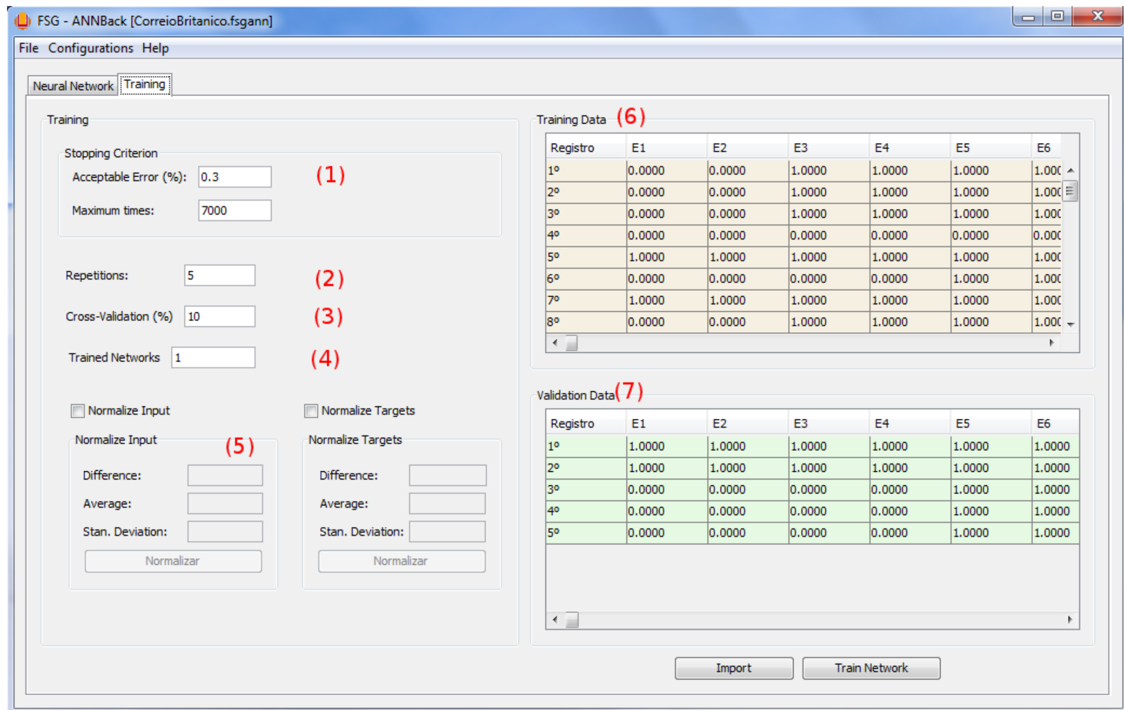

Figura 4.11: Área de configuração do treinamento da rede.

1. Critério de parada, definido primeiramente pelo erro aceitável máximo, onde caso o erro mensurado na saída da rede seja inferior ao mesmo se determina que houve a convergência e o padrão foi aprendido. Em segundo lugar, tem-se o número máximo de épocas, que estabelece quantas vezes o processo de treinamento será executada, caso não haja convergência;

2. Campo onde se tem quantas vezes cada um dos casos de treinamento serão apresentados a ANN;

3. Validação cruzada, onde se determina o quanto dos dados de treinamento será separado para ser utilizado na validação da rede, posteriormente ao treinamento;

4. Redes treinadas, neste campo se define quantas redes serão treinadas, e destas será retirada a que tiver menor erro na saída;

5. Normalização, nesta área pode-se normalizar os dados de treinamento e validação, normalmente para valores no intervalo fechado [0,1];

6. Dados de treinamento, nesta área são apresentados todos os dados que serão utilizados no treinamento da rede;

7. Dados de validação, neste espaço são apresentados todos os dados de validação, tendo eles sido removidos, de forma aleatória, dos dados de treinamento.

Por fim o desenvolvedor tem a possibilidade de executar os treinamentos, como ilustra a Figura 4.12, destacando-se os seguintes itens:

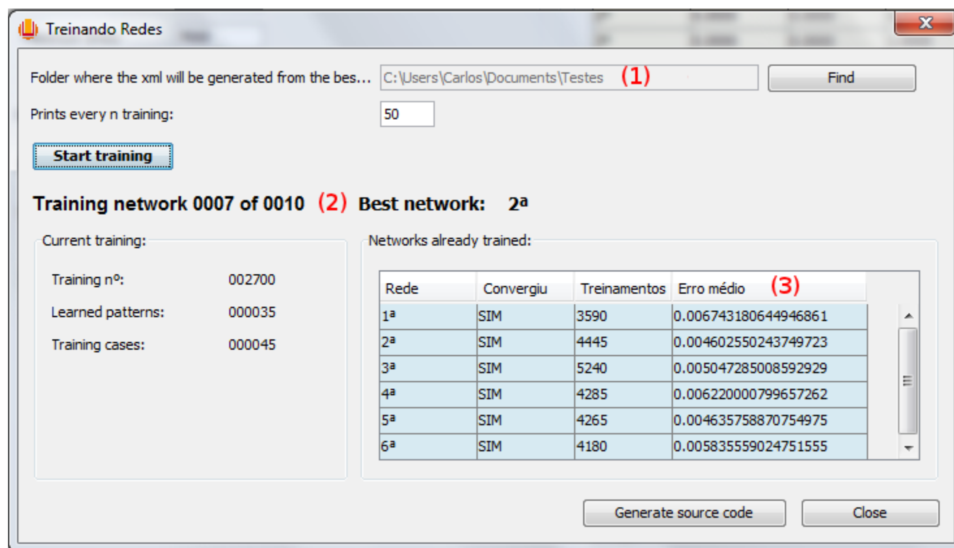

Figura 4.12: Treinamento das redes.

1. Ao término do treinamento o sistema irá gerar um XML com todos os pesos sinápticos da melhor rede. Nesta área define-se onde esse arquivo será gravado;

2. Nesta área são apresentadas informações sobre a rede que está sendo treinada, bem como qual foi a melhor rede até o momento;

3. Nesta listagem, são mostradas todas as redes já treinadas, se houve convergência ou não, quantas épocas foram necessárias e por fim o erro na camada de saída.

4.2.3 FSG - GA

Relativo ao GA, a ferramenta é responsável por sua configuração, realização de teste e validação. Ao final, há a possibilidade de geração do código fonte uma linguagem de programação.

Referente a configuração do GA, a ferramente possui a interface ilustrada na Figura 4.13, da qual pode-se destacar os seguintes itens:

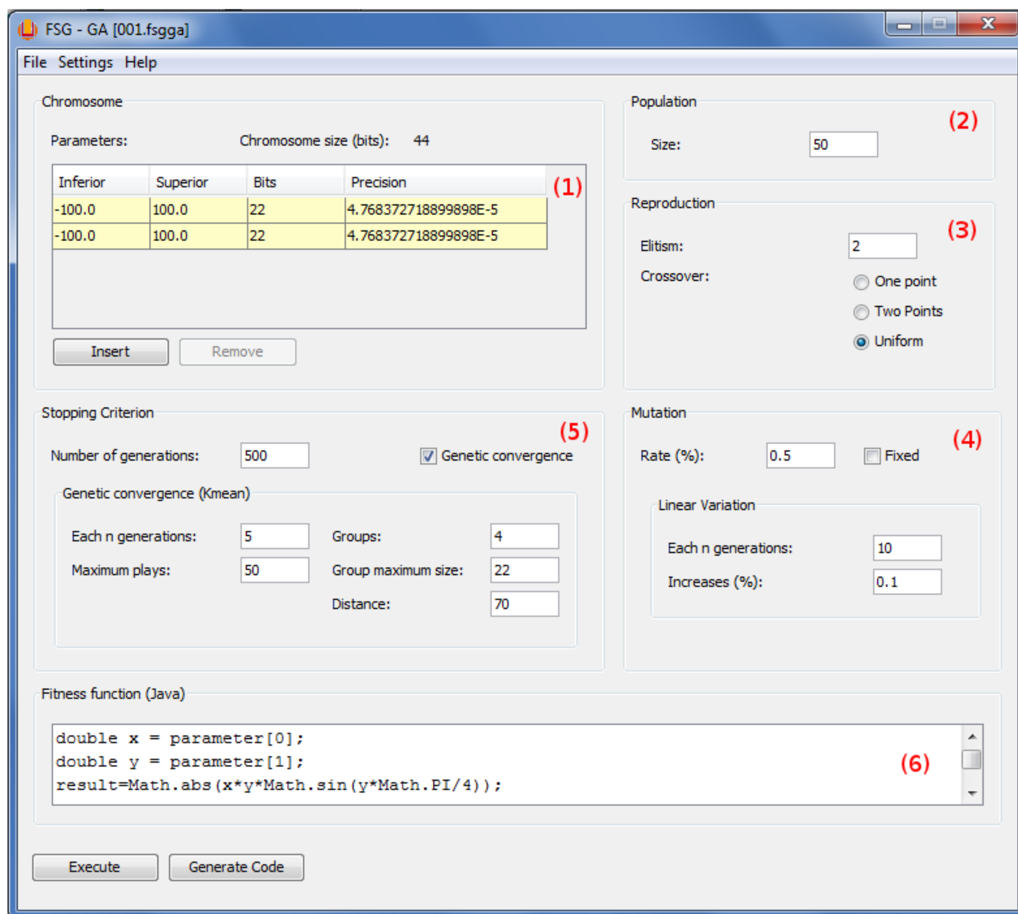

Figura 4.13: Interface inicial do RAD de algoritmos genéticos.

1. Nesta área pode-se definir os diversos parâmetros da função a ser otimizada que irão compor os cromossomos da população. Neste exemplo, tem-se dois parâmetros que possuem valores entre -100 e +100 e são representados por 22 bits cada. Assim, o cromossomo resultante fica com 44 bits;

2. Neste item se especifica o tamanho da população, que neste exemplo é de 50 indivíduos;

3. Nesta parte pode-se estabelecer qual critério de reprodução presente no framework será utilizado (one point / two points / uniform) e ainda se o GA se valerá de elitismo. No exemplo, apresentado os dois melhores indivíduos da população atual serão transmitidos para a nova geração;

4. Nesta área é definida a taxa de mutação e se for o caso o quanto ela vai variar a cada n gerações;

5. Aqui é definido o critério de parada do GA, que pode ser por número máximo de gerações ou utilizando a convergência genética com K-mean. Caso se opte pela utilização do K-means, é necessário definir de quantas em quantas gerações o algoritmo será executado, quantas vezes vai executar, em quantos grupos ele irá dividir a população, qual o tamanho máximo que um grupo pode ter antes que seja identificada a convergência, bem como a distância entre os centróides;

6. Função de aptidão (*fitness function*), que deve ser implementada de acordo com a função que se deseja otimizar. Caso se queira executar o GA dentro do próprio RAD é necessário escrever essa função em código Java. Por outro lado, caso a ideia seja somente gerar o código fonte no padrão do framework, a função pode ser escrita na linguagem de destino.

Ainda, caso se deseje executar o GA dentro do próprio RAD tem-se a interface apresentada na Figura 4.14, onde são apresentados os seguintes itens:

1. Geração atual;

2. Taxa de mutação na geração atual;

3. Indivíduos selecionados pelo elitismo, junto com o seu valor de aptidão;

4. Convergência genética com o algoritmo K-means;

5. Ao término da execução, informações sobre o indivíduo com melhor função de aptidão.

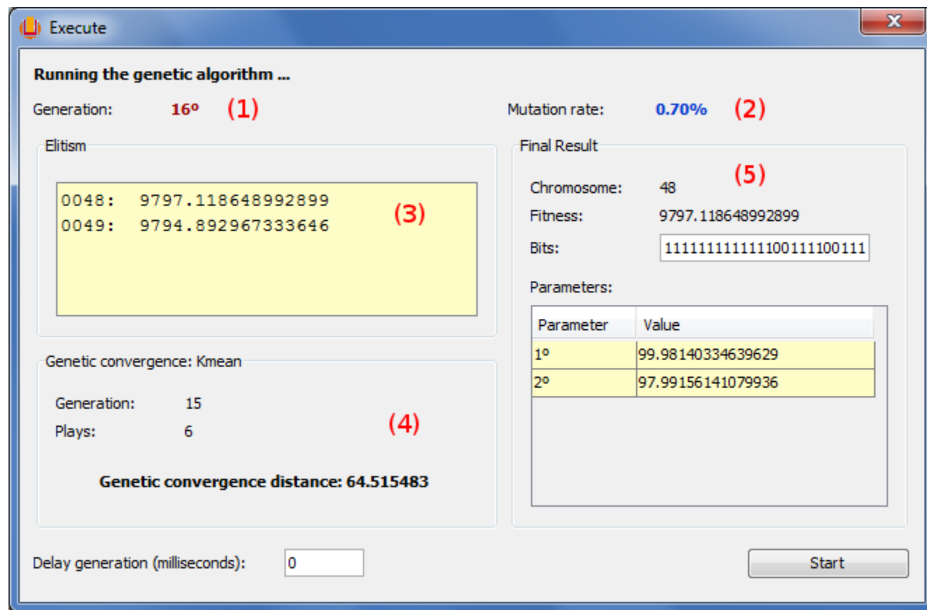

Figura 4.14: Execução do GA configurado na figura anterior.

4.2.4 FSG - Fuzzy

Por fim, no que tange ao FIS, o RAD é responsável pela configuração do sistema de inferência fuzzy (FIS), realização de testes e validação do mesmo, além da geração do código do fonte em uma dada linguagem de programação.

Inicialmente, no que se refere a catalogação das variáveis linguísticas, pode-se destacar os seguintes itens destacados na Figura 4.15:

1. Nesta área se tem uma listagem de todas as variáveis linguísticas previamente registradas, clicando-se sobre um item da lista a respectiva variável é habilitada para edição;
2. Neste item encontram-se o nome da variável, bem como o nome do objeto que será instanciado quando da geração do código fonte;
3. Nesta região são definidos os limites do universo de discurso da variável;
4. Nesta área se tem o registro de todos os termos linguísticos que constituem a variável;
5. Neste gráfico encontram-se representados as funções de pertinência de cada termo.

Como ilustrado na Figura 4.15, a ferramenta apresenta a possibilidade de que sejam

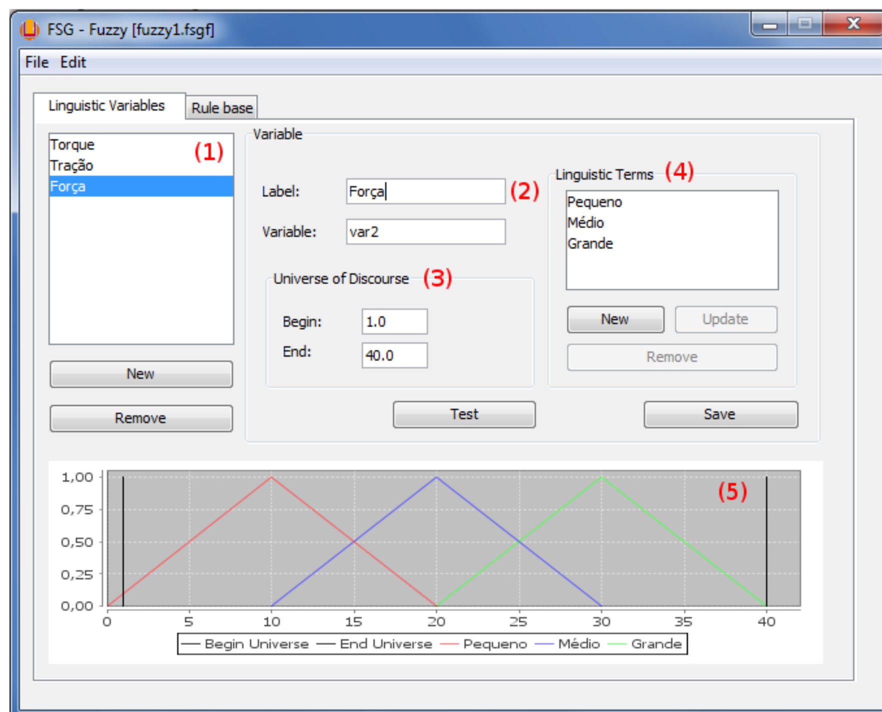

Figura 4.15: Catalogação das variáveis linguísticas

registrados os termos linguísticos de cada variável. Estes são configurados de acordo com a Figura 4.16, destacando-se os seguintes itens:

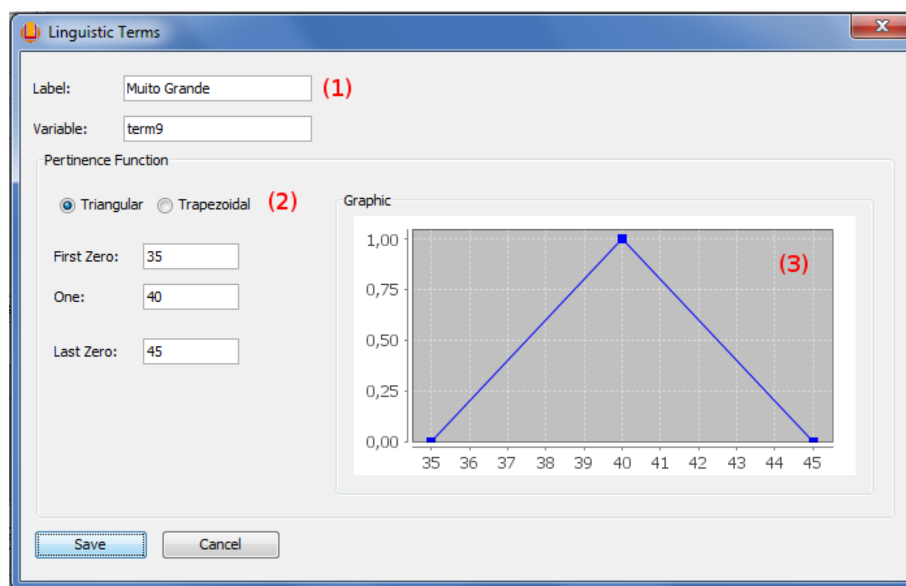

Figura 4.16: Configuração de um termo linguístico

1. Definição do nome do termo linguístico, além do objeto que será instanciado quando da geração do código fonte;

2. Nesta área é estabelecido o tipo de função de pertinência para este termo;

3. Neste gráfico encontram-se representada a função referente ao termo em questão.

Um vez que tenham sido registradas todas as variáveis linguísticas, através da interface ilustrada na Figura 4.15, deve-se criar a base de regras de produção, valendo-se da ferramenta apresentada na Figura 4.17.

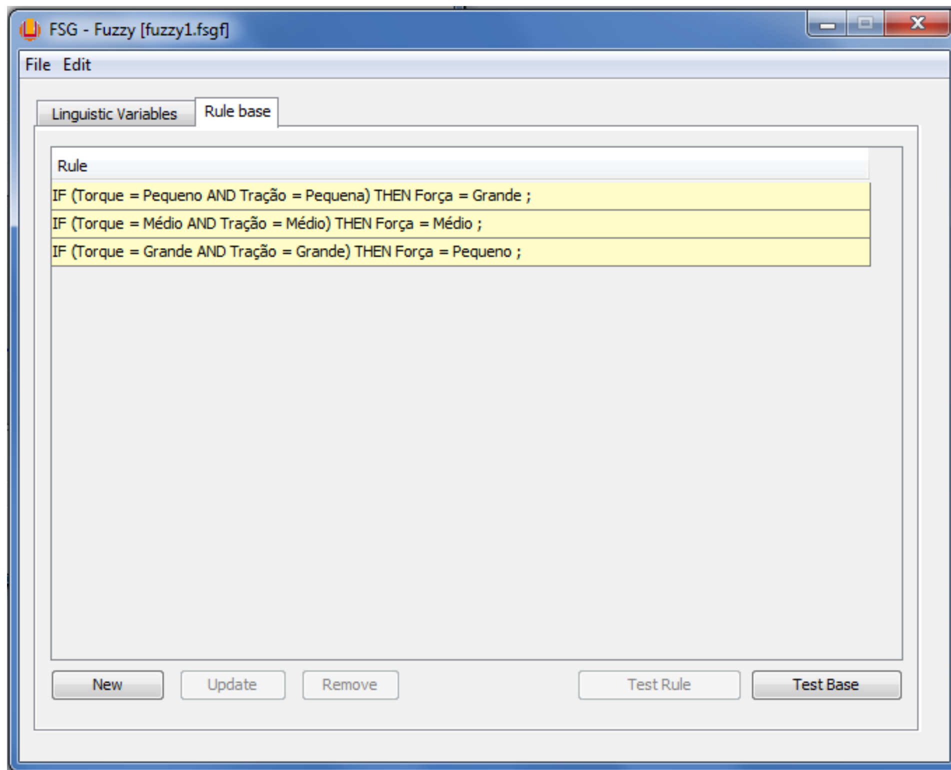

Figura 4.17: Base de regras de produção do FIS

Acessando a interface apresentada na Figura 4.17, tem-se a possibilidade de registrar novas regras de produção, através da tela ilustrada na Figura 4.18. Nesta última, destaca-se o fato de se poder registrar quantos antecedentes da regra se ache necessário e somente um consequente.

Uma vez tendo a base de regras estabelecida, pode-se testar cada regra individualmente, como é observado na Figura 4.19, podendo-se destacar os seguintes itens:

1. Descrição da regra sobre a qual se está trabalhando;

2. Valores crisp de entrada para cada uma das variáveis linguísticas, presentes nos antecedentes da regra;

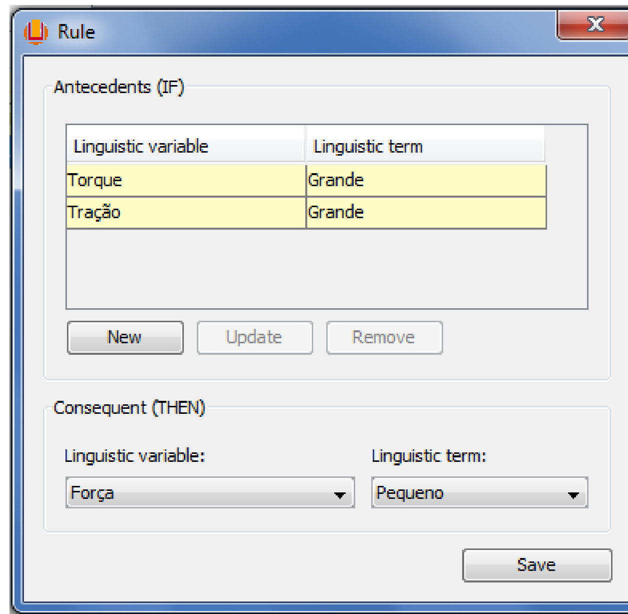

Figura 4.18: Configuração de uma nova regra de produção

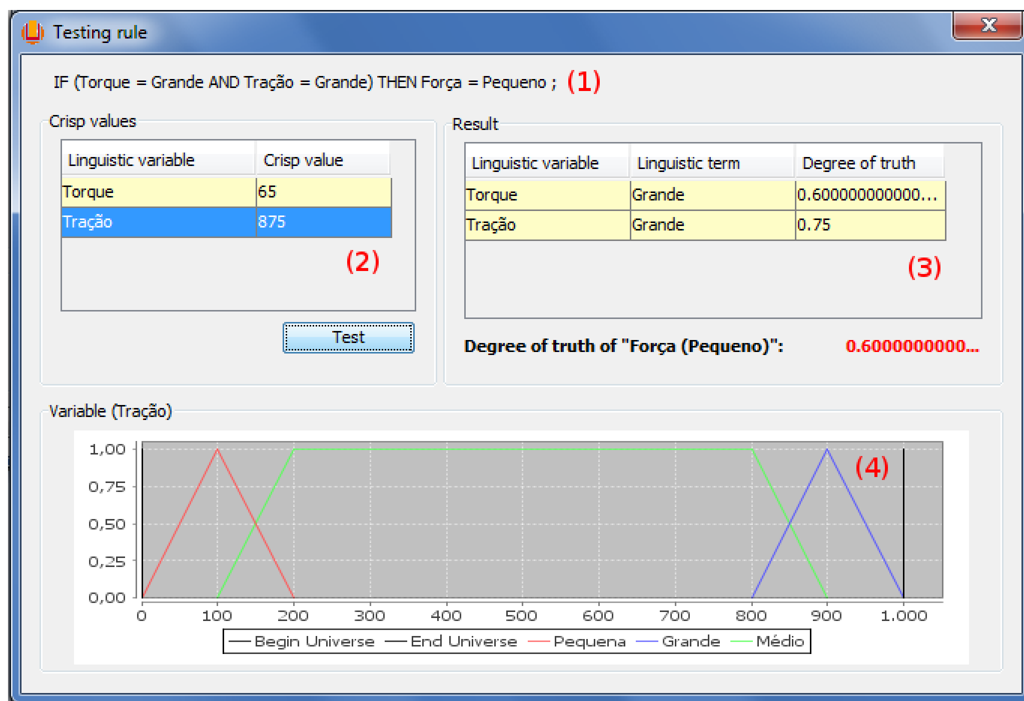

Figura 4.19: Teste de uma regra de produção

3. Grau de pertinência resultante em cada uma das variáveis e seus respectivos termos linguísticos definidos pelos antecedentes da regra, além do grau de pertinência final que será aplicado ao consequente.

Da mesma forma com que se testa uma regra, a ferramenta permite que se teste toda

a base de regras, como ilustrado na Figura 4.20, destacando-se os seguintes itens:

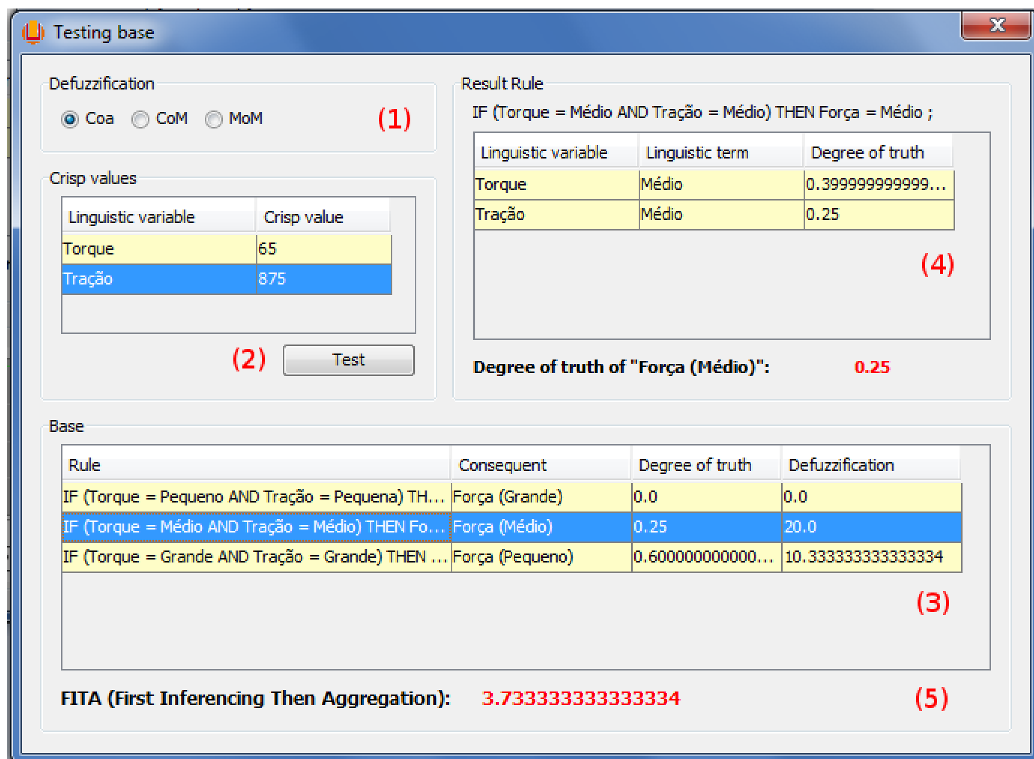

Figura 4.20: Teste da base de regras

1. Técnica de defuzzificação que será utilizada nas funções de pertinência dos termos linguísticos presentes nos consequentes.

2. Valores crisp de entrada para cada uma das variáveis linguísticas, presentes nos antecedentes das regras;

3. Grau de pertinência e respectivo valor defuzzificado, resultante de cada regra de produção;

4. Inspeção do grau de pertinência resultante de uma regra selecionada e de seus respectivos antecedentes;

5. Resultado final do FIS Mamdani utilizando-se a técnica de agregação FITA.

Capítulo 5

Experimentos e Resultados

5.1 Jogo Implementado

Para a realização dos testes com as ferramentas propostas, foi desenvolvido um jogo onde dois personagens duelam nos moldes dos jogos RPGs (*Role-Playing Game*). O mesmo foi implementado em Java empregado o framework JGame [van Schooten, 2012], para a implementação das animações em 2D e o framework FSG para a implementação da camada de tomada de decisão do personagem "Guarda". A interface do jogo é ilustrada na Figura 5.1, na qual destacam-se os seguintes itens:

Figura 5.1: Jogo proposto para validação da ferramenta.

1. Personagem conhecido como “Jogador”, quando o jogo não está no modo automático ele pode ser controlado pelo usuário através das seta direcionais do teclado e da tecla “A” para desferir seus ataques;

2. Personagem conhecido como “Guarda”, é o NPC do jogo e tem por objetivo montar guarda sobre determinada área, além de defendê-la frente a qualquer outro personagem que venha a entrar em seu raio de ação;

3. Área onde, antes de se iniciar uma partida, o jogador pode configurar os atributos dos dois personagens. Inclusive pode-se definir qual a técnica de IA será utilizada pelo “Guarda” em sua tomada de decisão (atacar ou não);

4. Por fim, nesta interface existe a possibilidade de simular automaticamente um determinado número de embates, nos quais os dois personagens são autônomos e seus atributos são definidos de forma aleatória. Ao fim de cada enfrentamento, as informações dos atributos iniciais dos personagens, bem como quem foi o vencedor, é armazenado em um arquivo para posterior utilização na configuração das técnicas de IA.

Cada um dos personagens presentes no jogo apresenta os seguintes atributos:

- **Energia**: representa a sua energia vital, tendo valores restritos ao intervalo [0,100]. Quando um personagem tem energia igual a zero, este é dado como derrotado e o jogo é encerrado;

- **Experiência**: este atributo define a sua experiência em batalha, tendo valores no intervalo [0,100]. Este valor estabelece a probabilidade de um ataque desferido ser bem sucedido ou não. Assim, caso um personagem tenha 20 pontos de experiência, ele terá uma probabilidade de 20% de causar algum dano ao seu oponente;

- **Poder de ataque**: este atributo estabelece, no caso de um ataque bem sucedido, o valor máximo de dano que o oponente pode vir a sofrer, tendo seus valores no intervalo [0,100];

- **Poder de defesa**: define quanto dano o personagem pode suportar antes que a sua energia vital venha a ser afetada, podendo receber os valores no intervalo [0,100];

- **Raio de Ação**: este atributo estabelece o raio em torno do qual o personagem consegue perceber o ambiente, possuindo valores no intervalo [150,250];
- **Velocidade**: este atributo define a velocidade com que o personagem se desloca pelo ambiente, possuindo valores no intervalo [0,10].

O dano que um determinado ataque bem sucedido causa ao adversário é definido pela Equação 5.1. Nesta, observa-se a dependência direta em relação a experiência, energia vital e ao poder de ataque. Em vista disto, o dano somente será igual ao poder de ataque quando o personagem este estiver com sua energia e experiência nos valores máximos.

$$dano = (\frac{experiencia}{100}).(\frac{energia_vital}{100}).(poder_de_ataque) \tag{5.1}$$

De forma sucinta, um confronto pode ser definido através dos seguintes passos:

1. Os personagens se deslocam pelo ambiente até que um oponente esteja em seu raio de ação;
2. Uma vez que um oponente está dentro do raio de ação, um ataque é desferido;
3. Se, de acordo com o nível de experiência do personagem, o ataque for bem sucedido, então o dano é calculado de acordo com a Equação 5.1;
4. Se o dano infligido for menor que o poder de defesa do oponente, este atributo é decrescido;
5. Se o dano infligido for maior ou igual ao poder de defesa do oponente, seu item de defesa (por exemplo um escudo) é destruído e sua energia vital fica vulnerável para um próximo ataque;
6. Se o poder de defesa do oponente for zero, então o dado é descontado da energia vital do oponente;
7. Ao fim do ataque bem sucedido, a experiência do agressor é acrescida em um ponto;
8. O agressor passa a vez ao oponente, caso esteja em seu raio de ação, para que o mesmo desfira o próximo ataque;

9. Este ciclo se repete até que um dos dois seja derrotado ou saia do raio de ação.

O comportamento do NPC "Guarda" é regido pela FSM ilustrada na Figura 5.2, que por sua vez foi concebida utilizando-se o RAD "FSG - Finite-State Machine", conforme a interface apresentada na Figura 4.6.

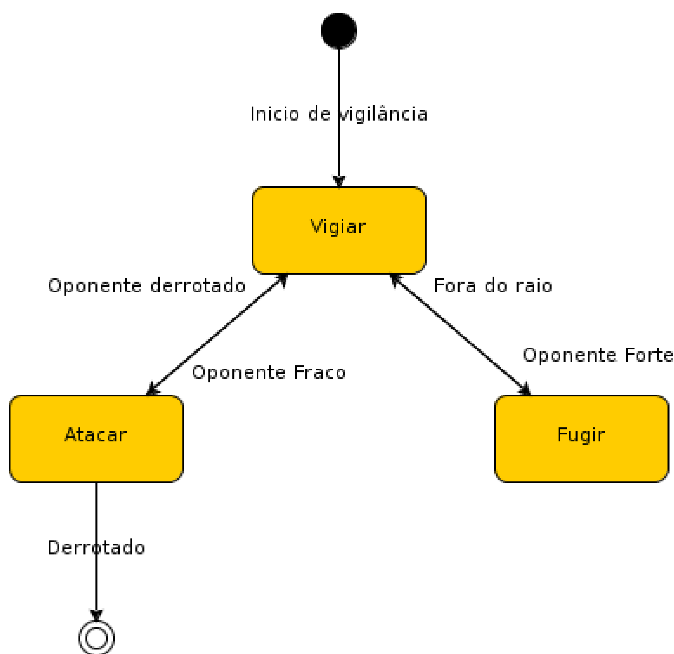

Figura 5.2: FSM responsável pelo comportamento do NPC "Guarda".

De acordo com a FSM apresentada na Figura 5.2, o Guarda inicialmente assume o estado "Vigiar", onde o NPC fica percorrendo a área sobre a qual está montando guarda e caso perceba um oponente, de acordo com os seus atributos internos e com os do inimigo, deve decidir se ataca ou foge. Neste trabalho, esta tomada de decisão será realizada associando-se uma mesma técnica de IA a função de guarda da transição "Oponente fraco", que leva ao estado "Atacar", e ainda a função de guarda da transição "Oponente Forte" que leva ao estado "Fugir". No estado "Atacar" ocorre o duelo entre os dois personagens, e caso saia vitorioso, o Guarda volta ao estado "Vigiar". Caso contrário, o processo é encerrado. Uma vez assumindo o estado "Fugir", o NPC tenta sair do raio de ação do inimigo. Caso tenha sucesso, volta ao estado "Vigiar".

O diagrama de classes do jogo implementado é apresentado na Figura 5.3, o qual engloba as questões de tela, a cargo do JGame, a lógica do jogo, com a definição dos

personagens, e ainda a tomada de decisão do NPC Guarda através do FSG. Pode-se destacar as seguintes classes:

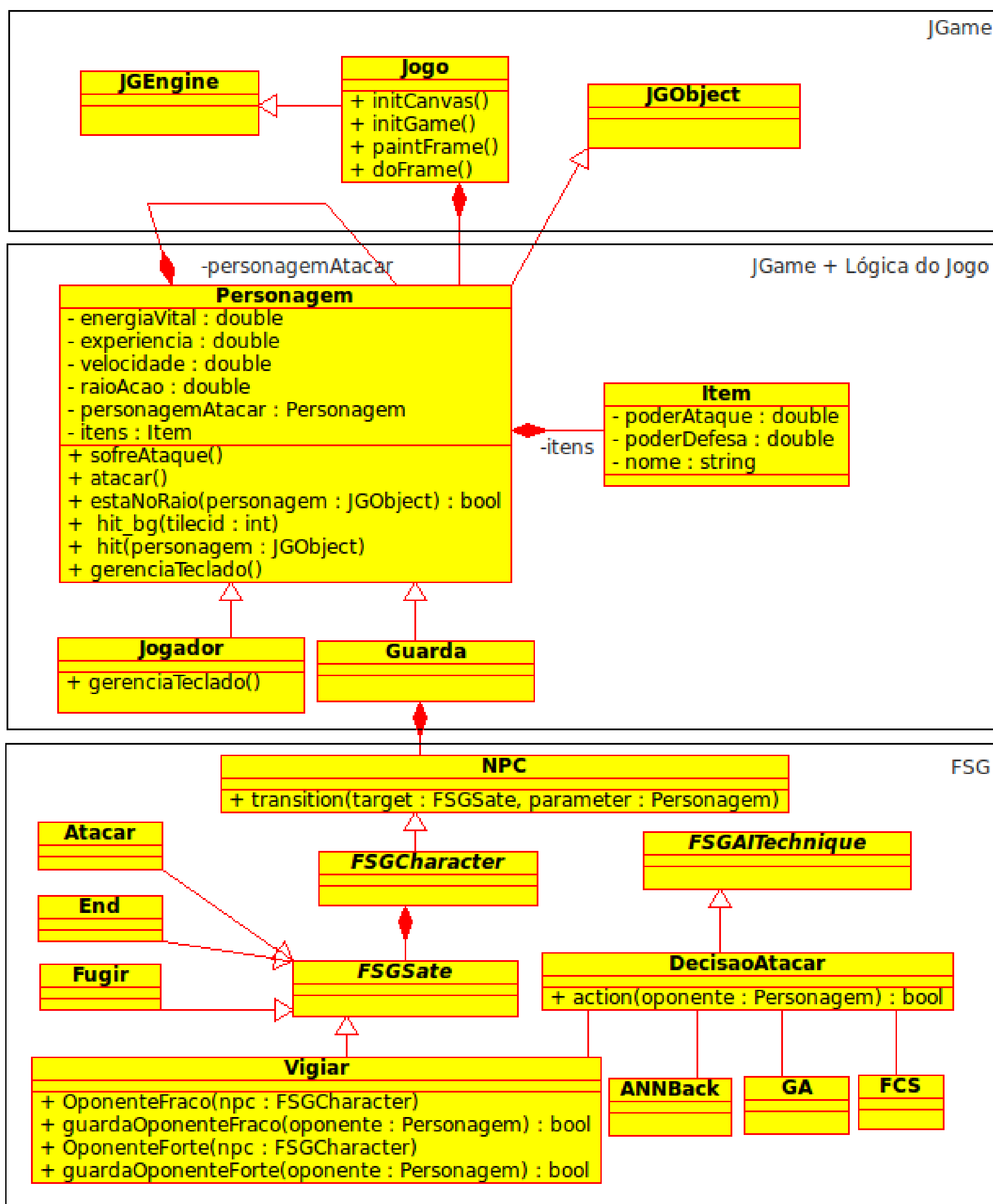

Figura 5.3: Diagrama de classes do jogo implementado.

- **Jogo**: classe principal do jogo, responsável pego gerenciamento da tela e controle de eventos de mouse e teclado, sendo uma extensão da classe JGEngine do JGame;

- **Personagem**: nesta classe estão implementados os atributos dos personagens, bem como um conjunto de itens de defesa e ataque que dão origem ao poder de defesa e ataque, além de uma referência ao oponente. Herda a classe JGObject do JGame, da qual reescreve os métodos "hit_bg", que controla a colisão com estruturas do cenário, e "hit" que controla a colisão entre os personagens. Referente a lógica do jogo implementada, destacam-se os métodos "estaNoRaio", que verifica se existe algum adversário em seu raio de ação; "atacar", responsável por desferir um ataque, calculando sua possibilidade de acerto e dano resultante; "sofreAtaque", responsável por estabelecer se algum item de defesa vai ser destruído ou se a energia vital será decrescida, frente a um ataque;

- **Jogador**: herda a classe Personagem e unicamente implementa o método "gerenciaTeclado" para responder a eventos do teclado repassados pela classe Jogo, quando não estiver operando no modo automático;

- **Guarda**: herda a classe Personagem e, devido a impossibilidade de herança múltipla em Java, é composta por uma instância da classe NPC, que provê a implementação da FSM e tomada de decisão através do framework FSG;

- **NPC**: essa classe representa a FSM que rege o comportamento do personagem Guarda. Como se vale do padrão de projeto State, a mesma possui uma instância da classe FSGState, que representa o seu estado interno, bem como um método para cada transição da FSM, sendo eles: "InicioVigilancia", "OponenteFraco", "OponenteForte", "Derrotado", "OponenteDerrotado" e "ForaRaio". Todos são invocados pelo FSG por intermédio do método "transition". Por sua vez, estes métodos repassam a solicitação a seus homônimos presentes no objeto que representa o estado interno, no qual está implementada a decisão de transitar de estado ou não;

- **Vigiar**: é a classe mais importante, referente ao estado interno do NPC, tendo em vista que nela que estão definidas as técnicas de IA. Esta classe é responsável por implementar os métodos de transição "OponenteFraco", que leva ao estado "Atacar", e o método "OponenteForte", que leva ao estado "Fugir". Como apresentado na seção 3.1, para cada transição há uma condição de guarda, que caso seja satisfeita

possibilita que a transição ocorra. Assim, ao se invocar o método "OponenteFraco", a condição de guarda "guardaOponenteFraco" é acionada e caso retorne verdadeiro, permite que o estado mude. O mesmo ocorre para a transição "OponenteForte". Para estas condições de guarda, uma instância da classe "DecisaoAtacar" está relacionada, e esta implementa uma das três técnicas de IA presentes do FSG.

Observa-se que todas as classes presentes no agrupador "FSG" do diagrama ilustrado na Figura 5.3 tiveram seus códigos, no padrão State, gerados através da ferramenta RAD "FSG - Finite-State Machine" utilizando-se a interface ilustrada na Figura 4.7 e de acordo com a FSM descrita na Figura 5.2. Por fim, destaca-se o fato de que a técnica de IA que será utilizada pela instância da classe ' "DecisaoAtacar" depende da seleção realizada na interface presente no item 3, ilustrado na Figura 5.1. Na próxima seção será descrito como cada uma das técnicas presentes no FSG foram configuradas para utilização no jogo proposto.

5.2 Configuração das Técnicas de IA

Para realização da configuração das técnicas de IA utilizadas no jogo foram gerados dois arquivos distintos, cada um resultante de 500 embates realizados de forma automática, dada a possibilidade apresentada no item 4 da Figura 5.1. Neste modo automático, o NPC Guarda sempre toma a decisão de atacar quando tem um inimigo em seu raio de visão. Em cada linha destes arquivos, foram registrados os atributos iniciais de cada personagem, bem como quem foi o vencedor do duelo em questão.

A primeira simulação automática resultou em 243 (48,6%) vitórias do Guarda e 257 (51,4%) vitórias do Jogador, sendo que uma descrição dos valores dos seus atributos são apresentados na Tabela 5.1 e 5.2 respectivamente.

A segunda simulação automática resultou em 253 (50,6%) vitórias do Guarda e 247 (49,4%) vitórias do Jogador, sendo que uma descrição dos valores dos seus atributos são apresentados na Tabela 5.3 e 5.4 respectivamente.

Atributo	**Média**	**Desvio Padrão**
Energia	50,89	29,39
Experiência	52,76	27,49
Poder de Ataque	49,25	28,43
Poder de Defesa	50,04	28,39
Raio de ação	202,27	29,91
Velocidade	5,13	2,82

Tabela 5.1: Atributos do Guarda na primeira simulação.

Atributo	**Média**	**Desvio Padrão**
Energia	51,56	27,88
Experiência	53,01	27,92
Poder de Ataque	49,07	27,77
Poder de Defesa	48,62	29,29
Raio de ação	200,91	29,94
Velocidade	4,61	2,91

Tabela 5.2: Atributos do Jogador na primeira simulação.

Atributo	**Média**	**Desvio Padrão**
Energia	51,61	29,1
Experiência	49,62	28,02
Poder de Ataque	51,61	28,65
Poder de Defesa	48,57	28,4
Raio de ação	200,25	28,69
Velocidade	5,08	2,83

Tabela 5.3: Atributos do Guarda na segunda simulação.

Atributo	**Média**	**Desvio Padrão**
Energia	51,43	29,43
Experiência	52,98	28,74
Poder de Ataque	49,48	29,26
Poder de Defesa	48,38	29,8
Raio de ação	202,06	29,2
Velocidade	4,77	2,79

Tabela 5.4: Atributos do Jogador na segunda simulação.

5.3 Artificial Neural Network

Utilizando-se a ferramenta RAD "FSG - ANNBack", apresentada pela interface ilustrada na Figura 4.10, foi definida uma rede neural com as seguintes características:

- Doze neurônios na camada de entrada, onde os seis primeiros são destinados a receber os atributos do Jogador e o seis últimos do Guarda;
- Vinte e quatro neurônios na camada oculta;
- Dois neurônios na camada de saída. Caso o Jogador sai vitorioso, o primeiro neurônio deve apresentar o valor um e o segundo zero, e quando o Guarda tiver êxito esses valores serão invertidos;
- Taxa de aprendizagem como valor 0,4;
- Momentum com valor 0,2;
- Função de ativação sigmoide, padrão do framework.

Ainda utilizando-se o RAD, por intermédio da interface apresentada na Figura 4.11, o treinamento foi concebido com os seguintes valores:

- Erro aceitável na camada de saída: 0,3%;
- Número máximo de épocas: 500.000;
- Apresentação de cada padrão a rede por cinco vezes;
- Dos dados de treinamento, 10% foi separado para validação cruzada;
- Foram treinadas 100 redes, e destas extraída a com menor erro na saída.

Os resultados da primeira simulação foram utilizados neste processo, onde 450 registros foram empregados no treinamento e 50 na validação das redes. Houve a necessidade de se normalizar os dados, apesar da ferramenta dar suporte a esse processo, o mesmo não pode ser utilizando, tendo em vista que ela parte do pressuposto que todas as entradas estão no mesmo intervalo. Todavia os atributos energia, experiência, poder de ataque e

poder e defesa estão no intervalo [0,100], já raio de ação está no intervalo [150,250] e a velocidade no intervalo [0,10]. Assim, foi necessário a implementação de um código que normalizasse estes atributos entre -0.5 e +0.5, para utilização nos treinamentos.

Uma vez que a melhor rede foi escolhida pelo RAD, um arquivo XML com todos os seus pesos e atributos foi salvo para utilização no Jogo, dado a possibilidade apresentada na seção 4.1.1 através da classe "ANNBackXML".

Em seguida, de posse dessa rede, implementou-se um programa, utilizando-se somente as classes de ANN do FSG, que permite a leitura dos arquivos oriundos das simulações e verifica se a tomada de decisão seria acertada. Essa tomada de decisão ocorreu da seguinte forma: caso o primeiro neurônio da camada de saída tenha valor maior que o segundo, o Guarda deve fugir, caso contrário, deve atacar.

Processando o arquivo referente a primeira simulação, o mesmo que serviu de treinamento para a rede, obteve-se os seguintes resultados:

- Decisões corretas, englobando ataques e fugas: 495 de 500, perfazendo 99.0%;
- Decisão de atacar em embates que o Guarda saiu vitorioso: 240 de 243 vitórias, perfazendo 98.71%;
- Decisão de fugir em embates que o Guarda foi derrotado: 255 de 257 vitórias, perfazendo 99.22%;

Ao se processar o arquivo resultante da segunda simulação, para o qual a rede nunca foi apresentada, obteve-se os seguintes resultados:

- Decisões corretas, englobando ataques e fugas: 424 de 500, perfazendo 84.8%;
- Decisão de atacar em embates que o Guarda saiu vitorioso: 210 de 253 vitórias, perfazendo 83.01%;
- Decisão de fugir em embates que o Guarda foi derrotado: 214 de 247 vitórias, perfazendo 86.63%;

Por fim, o código no padrão FSG da rede treinada, gerado pelo RAD, foi acrescido a classe "DecisaoAtacar" em seu método "action", apresentados no diagrama ilustrado na Figura 5.3, havendo apenas a necessidade de um código "cola" que normalizasse as entradas, como pode ser visto na Figura 5.4.

```
1 public class DecisaoAtacar extends FSGAITechnique {
2   private Personagem npc;
3
4   public boolean action (Personagem oponente) {
5     try {
6         ANNBackXML xml = new ANNBackXML();
7         ANNBack rede = xml.read("rede.xml");
8
9         double[] entrada = this.normalizaEntrada(this.npc,oponente);
10        double[] saida = new double[2];
11
12        rede.propagation(entrada);
13        saida = rede.getOutput();
14
15        if (saida[0]>saida[1]) {
16            return (false);
17        }
18        else {
19            return (true);
20        }
21     } catch (ANNBackLayerException e) {
22        return (false);
23     } catch (ANNBackIOException e) {
24        return (false);
25     }
26   }
27 }
```

Figura 5.4: Código de utilização de ANN no jogo.

5.4 Fuzzy Inference System (FIS)

Para o sistema de inferência fuzzy, utilizando-se a ferramenta RAD “FSG - Fuzzy”, através da interface ilustrada na Figura 4.15, foram definidas 13 variáveis linguísticas, 6 referentes ao atributos do Jogador, 6 referentes aos atributos do Guarda, e 1 para a decisão de atacar ao não, sendo elas:

- As variáveis Energia, Poder de Ataque e Poder de Defesa, são constituídas dos termos linguísticos Fraca, Média e Forte, tendo função de pertinência de acordo com a Figura 5.5;
- A variável Experiência é composta dos termos linguísticos Pouca, Média e Muita, com funções de pertinência de acordo com a Figura 5.6;
- A variável Raio de Ação é composta dos termos linguísticos Pequeno, Médio e Grande, com funções de pertinência de acordo com a Figura 5.7;

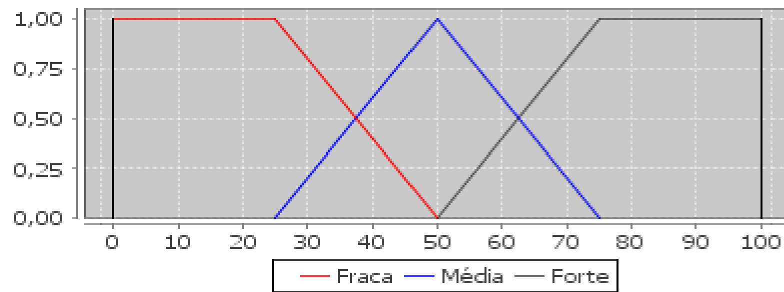

Figura 5.5: Termos linguísticos das variáveis Energia, Poder de Ataque e Poder de Defesa.

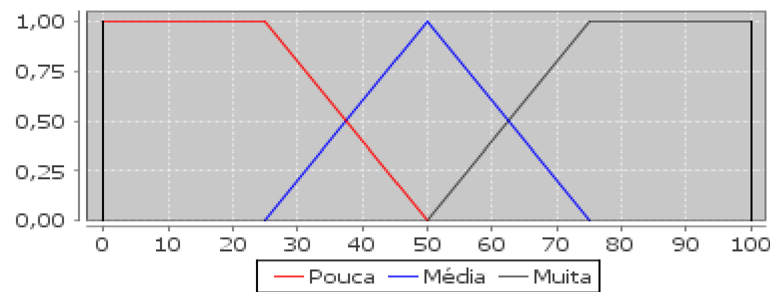

Figura 5.6: Termos linguísticos da variável Experiência.

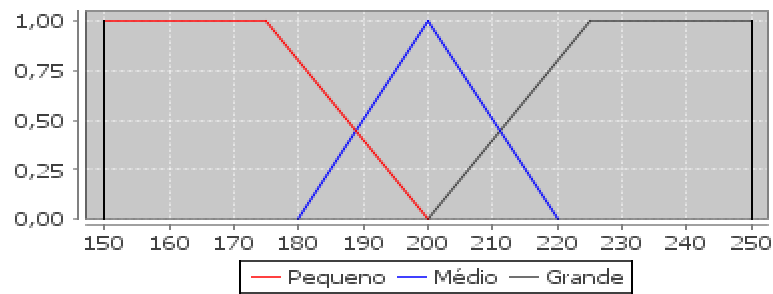

Figura 5.7: Termos linguísticos da variável Raio de Ação.

- A variável Velocidade é composta dos termos linguísticos Lento, Médio e Rápido, com funções de pertinência de acordo com a Figura 5.8;

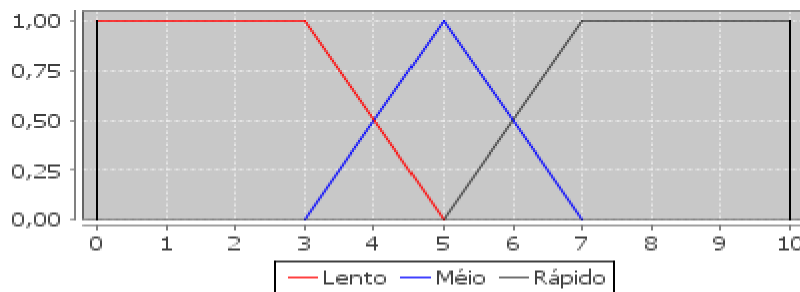

Figura 5.8: Termos linguísticos da variável Velocidade.

		Energina do Jogador		
		Fraca	Média	Forte
Energia do Guarda	Fraca	Sim	Não	Não
	Média	Sim	Sim	Não
	Forte	Sim	Sim	Sim

Tabela 5.5: Conjunto de regras referentes as energias dos oponentes.

		Experiência do Jogador		
		Pouca	Média	Muita
Experiência do Guarda	Pouca	Sim	Não	Não
	Média	Sim	Sim	Não
	Muita	Sim	Sim	Sim

Tabela 5.6: Conjunto de regras referentes as experiência em combate dos oponentes.

- A variável Atacar é composta dos termos linguísticos Não e Sim, com funções de pertinência de acordo com a Figura 5.9.

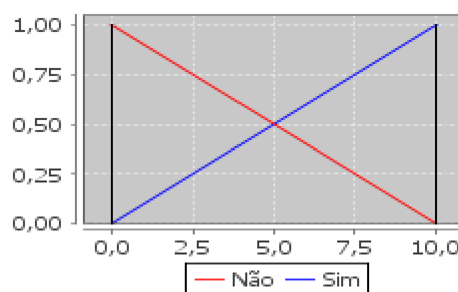

Figura 5.9: Termos linguísticos da variável Atacar.

De posse destas variáveis linguísticas, foi concebida uma base constituída de 54 regras, onde cada uma apresenta como resultado um termo linguístico da variável “Atacar”, que definirá se o Guarda deve atacar ou fugir. Toda essa base é apresentada nas tabelas 5.5, 5.6, 5.7, 5.8, 5.9 e 5.10.

Cada célula das tabelas descritas representa uma regra. Para elucidar, pode-se reescrever a regra hachurada da Tabela 5.10, de acordo com a representação apresentada na seção 3.3, conforme a Tabela 5.11.

A defuzzificação FIS foi configurada com o método FITA e cada função de pertinência através do método CoM.

Em seguida, de posse dessa FIS, implementou-se um programa, utilizando-se somente as classes de Fuzzy do FSG, que leem os arquivos oriundos das simulações e verificam se

Defesa do Jogador

Ataque do Guarda	Fraca	Média	Forte
Fraca	Sim	Não	Não
Média	Sim	Sim	Não
Forte	Sim	Sim	Sim

Tabela 5.7: Conjunto de regras referentes as capacidade de defesa do Jogador e ao poder de ataque do Guarda.

Ataque do Jogador

Defesa do Guarda	Fraca	Média	Forte
Fraca	Sim	Não	Não
Média	Sim	Sim	Não
Forte	Sim	Sim	Sim

Tabela 5.8: Conjunto de regras referentes as capacidade de ataque do Jogador e ao poder de defesa do Guarda.

Raio de Ação do Jogador

Raio de ação do Guarda	Pequeno	Médio	Grande
Pequeno	Sim	Não	Não
Médio	Sim	Sim	Não
Grande	Sim	Sim	Sim

Tabela 5.9: Conjunto de regras referentes ao raio de ação dos oponentes.

Velocidade do Jogador

Velocidade do Guarda	Lento	Médio	Rápido
Lento	Sim	Não	Não
Médio	Sim	Sim	Não
Rápido	Sim	Sim	Sim

Tabela 5.10: Conjunto de regras referentes a velocidade dos oponentes.

```
SE Velocidade do Jogador = Rápido AND Velocidade do Guarda =
Rápido ENTÃO Atacar = Sim
```

Tabela 5.11: Exemplo de regra de produção.

a tomada de decisão seria acertada. Essa tomada de decisão ocorre da seguinte forma: caso o valor crisp da variável linguística “Atacar” tenha valor menor que 5, o Guarda deve fugir, caso contrário, deve atacar.

Processando o arquivo referente a primeira simulação, obteve-se os seguintes resultados:

- Decisões corretas, englobando ataques e fugas: 243 de 500, perfazendo 48.6%;
- Decisão de atacar em embates que o Guarda saiu vitorioso: 102 de 243 vitórias, perfazendo 41.97%;
- Decisão de fugir em embates que o Guarda foi derrotado: 141 de 257 vitórias, perfazendo 54.86%;

Já ao se processar o arquivo resultante da segunda simulação, obteve-se os seguintes resultados:

- Decisões corretas, englobando ataques e fugas: 254 de 500, perfazendo 50.8%;
- Decisão de atacar em embates que o Guarda saiu vitorioso: 125 de 253 vitórias, perfazendo 49.41%;
- Decisão de fugir em embates que o Guarda foi derrotado: 129 de 247 vitórias, perfazendo 52.22%;

Por fim, o código no padrão FSG do FIS configurado, gerado pelo RAD, foi acrescido a classe “DecisaoAtacar” em seu método “action”, apresentados no diagrama ilustrado na Figura 5.3, como pode ser visto na Figura 5.10

5.5 Genetic Algorithm

Utilizando-se a ferramenta RAD “FSG - GA”, através da interface ilustrada na Figura 4.13, foi definido um algoritmo genético com as seguintes características:

- Foi utilizado um cromossomo de 45 bits que comporta todos os atributos do personagem Jogador, conforme é apresentado na tabela 5.5;

Atributo	**Nº de Bits**	**Valor Inicial**	**Valor Final**	**Precisão**
Energia	8	0	100	0.39
Experiência	8	0	100	0.39
Poder de Ataque	8	0	100	0.39
Poder de Defesa	8	0	100	0.39
Raio de Ação	8	150	250	0.39
Velocidade	5	0	10	0.32

Tabela 5.12: Definição do cromossomo.

- População com 100 indivíduos;
- Número máximo de gerações: 1000;
- Crossover de dois pontos;
- Seleção utilizando-se roleta viciada de uma agulha;
- Taxa de mutação: 0,5%;
- Elitismo de 2 indivíduos.
- Convergência genética utilizando o algoritmo K-Mean:
  - Número de grupos que a população foi dividida: 5;
  - É permitido no máximo 35 indivíduos em cada grupo, caso algum grupo exceda esse número, fica caracterizada a convergência genética;
  - Número máximo de execuções do algoritmo: 20;
  - É permitido uma distância máxima de 60 entre dois grupos. Caso haja uma distância menor que essa, fica caracterizada a convergência genética;
  - A convergência é verificada a cada 10 gerações.

Quando se implementa GA no FSG, há a necessidade de se estender a classe "GAChromosome" para se implementar a função de *fitness*. Neste jogo, esta foi definida conforme o código apresentado na Figura 5.11, com o intuito de selecionar o oponente "ideal" (mais fraco) mediante os atributos do Guarda.

No código da Figura 5.11, observa-se que cada cromossomo recebe uma referência ao NPC. No caso o personagem Guarda, o método "evaluation" recebe como parâmetro as

informações de um cromossomo da população. A função de *fitness* utiliza uma diferença entre os atributos do Guarda e cromossomo atual. Quanto maior essa diferença, mais fraco é o oponente. Caso o resultante seja negativo, significa que o oponente é muito mais forte que o Guarda. Assim, é atribuído um valor padrão de 0.0001.

Uma vez definido o GA, o mesmo foi executado sobre cada registro do arquivo resultante da primeira simulação. Ao fim de cada processamento, foi calculada a distância euclidiana entre os atributos do personagem Jogador e o cromossomo resultante. Observou-se que a média destas distâncias, nos casos em que o Guarda foi vitorioso, foi de 107.2, e de 92.8, caso contrário. Assim, foi proposta uma tomada de decisão onde, caso esta distância seja maior ou igual a 100, o Guarda deve atacar, caso contrário, deve fugir.

Processando o arquivo referente a primeira simulação, o mesmo que serviu de base para a definição da tomada de decisão, obteve-se os seguintes resultados:

- Decisões corretas, englobando ataques e fugas: 309 de 500, perfazendo 61.67%;
- Decisão de atacar em embates que o Guarda saiu vitorioso: 154 de 243 vitórias, perfazendo 63.37%;
- Decisão de fugir em embates que o Guarda foi derrotado: 155 de 257 vitórias, perfazendo 60.31%;

Já ao se processar o arquivo resultante da segunda simulação, obteve-se os seguintes resultados:

- Decisões corretas, englobando ataques e fugas: 326 de 500, perfazendo 65.06%;
- Decisão de atacar em embates que o Guarda saiu vitorioso: 166 de 253 vitórias, perfazendo 65.61%;
- Decisão de fugir em embates que o Guarda foi derrotado: 160 de 247 vitórias, perfazendo 64.77%;

Por fim, o código no padrão FSG do GA, gerado pelo RAD, foi acrescido a classe “DecisaoAtacar” em seu método “action”, apresentados no diagrama ilustrado na Figura 5.3, como pode ser visto na Figura 5.12.

	Média (μ_s)	Desvio Padrão (μ_s)
ANN	14,44	17,28
FIS	116,79	135,94
GA	84863,13	67921,63

Tabela 5.13: Tempo de execução das técnicas de IA.

5.6 Avaliação de Desempenho dos Testes

Durante o processamento do segundo arquivo de simulação foram armazenados os tempos de execução das técnicas de IA, especificamente o tempo de propagação da ANN, tempo de inferência do FIS e tempo de execução do GA. Os resultados são apresentados na Tabela 5.13 e ilustrados nas Figura 5.13 e 5.14.

Estes testes foram realizados em um computador com processador Intel Core i5, com dois núcleos de clock 3.2 GHz e 4GB de memória RAM. Os códigos fontes foram compilados na ferramenta NetBens 7.0.1 através do JDK 1.6.

De acordo com os tempos médios apresentados na Tabela 5.13, , constata-se que mesmo o GA, que foi aproximadamente 700 vezes mais lento que o FIS e 5000 vezes mais lento que o ANN, precisou em média de apenas 0,084 segundos para gerar uma tomada de decisão. Mostrando assim que estas técnicas podem ser empregadas no desenvolvimento de jogos, inclusive de tempo real.

5.7 Análise dos resultados

Referente a implementação jogo, tendo em vista o que foi apresentado no diagrama de classes da Figura 5.3, observa-se que a integração o FSG com outros frameworks ocorre de forma facilitada, com a ressalva da impossibilidade de herança múltipla em Java. Em outras linguagens suportadas pelo FSG, como C++, existe essa possibilidade.

Além disso, a utilização das ferramentas RAD agilizaram o processo de desenvolvimento, tanto na edição das FSM, ilustrada na Figura 5.2, como nas demais técnicas de IA. Em relação ao código fonte gerado, para a ANN houve apenas a necessidade de se criar um código que normalizasse as entradas; para o GA foi necessário se reescrever a função de *fitness*, passando-se os atributos do personagem Guarda; para o FIS não foi necessária nenhuma modificação.

Por fim, pode-se observar que os resultados apresentados pelas ANN foram superiores aos demais, chegando a ter uma aproveitamento de 84%, para os registros da segunda simulação. A FIS foi a técnica que apresentou menor desempenho na tomada de decisão, com apenas 50%. Todavia, acredita-se que dedicando-se mais tempo a sintonia do sistema de inferência, pode-se obter melhores resultados. O GA, com 65% de acertos, por ser uma técnica de otimização, não foi beneficiado pela aplicação implementada. Mas em um jogo mais complexo, onde o NPC tenha de escolher, dentre vários inimigos, qual o mais fácil de derrotar, essa técnica deverá ter melhor desempenho.

```
class DecisaoAtacar extends FSGAITechnique {
 private Personagem npc;

 public boolean action (Personagem oponente) {
   try {
     //FIS do tipo Mandani
     Mandani man = new Mandani(54);

     //Definição das Variáveis Linguísticas
     Universe uEnergia = new Universe (0,100);

     PertinenceFunctionTrapezoidal funcao1;
     funcao1 = new PertinenceFunctionTrapezoidal(-25,0,25,50,uEnergia);
     LinguisticTerm energiaFraca = new LinguisticTerm ("Fraca",funcao1);

     PertinenceFunctionTriangular funcao2;
     funcao2 = new PertinenceFunctionTriangular(25,50,75,uEnergia);
     LinguisticTerm energiaMedia = new LinguisticTerm ("Média",funcao2);

     PertinenceFunctionTrapezoidal funcao3;
     funcao3 = new PertinenceFunctionTrapezoidal(50,75,100,125,uEnergia)
        ;
     LinguisticTerm energiaForte = new LinguisticTerm ("Forte",funcao3);

     LinguisticVariable energiaJogador;
     energiaJogador = new LinguisticVariable("Energia Jogador",3,
        uEnergia);
     energiaJogador.addTerm(energiaFraca);
     energiaJogador.addTerm(energiaMedia);
     energiaJogador.addTerm(energiaForte);
     .
     .
     .
     //Definição das regras de produção
     Rule regraEnergia1 = new Rule (2);
     regraEnergia1.addAntecedent(energiaJogador,energiaFraca);
     regraEnergia1.addAntecedent(energiaGuarda,energiaForte);
     regraEnergia1.setConsequent(atacar, atacarSim);
     man.addRule(regraEnergia1);
     .
     .
     .
     //Definindo os valores das variáveis na regra
     regraEnergia1.addAntecedentValue(oponente.getEnergia(),0);
     regraEnergia1.addAntecedentValue(this.npc.getEnergia(),1);
     .
     .
     .
     //Defuzzificação FITA
     FITA fita = new FITA (man.getNumRules(),new DefuzzificationCoM());
     man.setDefuzzificationMethod(fita);

     double crisp = man.defuzzification();

     if (crisp>=5) {
        return (true);
     }
     else {
        return (false);
     }
 }
 catch (UniverseException e) {
     return (false);
 } catch (RuleException e) {
     return (false);
 } catch (PertinenceFunctionException e) {
     return (false);
 } catch (LinguisticVariableException e) {
     return (false);
 } catch (FCSException e) {
     return (false);
 }
}
}
```

Figura 5.10: Código de utilização de FIS no jogo.

```
1  class MeuCromossomo extends  GAChromosome {
2    private Personagem npc;
3
4    public MeuCromossomo (int t,Personagem n) throws GAChromosomeException
         {
5        super(t);
6        this.npc = n;
7    }
8
9    @Override
10   public double evaluation (double[] parameter)  {
11      double result=0;
12      double energia = parameter[0];
13      double experiencia = parameter[1];
14      double defesa = parameter[2];
15      double ataque = parameter[3];
16      double raio = parameter[4];
17      double velocidade = parameter[5];
18
19      result= (this.npc.getEnergia()      - energia) +
20              (this.npc.getExperiencia() - experiencia) +
21              (this.npc.getPoderDefesa() - defesa) +
22              (this.npc.getPoderAtaque() - ataque) +
23              (this.npc.getRaioAcao()    - raio) +
24              (this.npc.getVelocidade()  - velocidade);
25
26      if (result<=0) {
27           result = 0.0001;
28      }
29      return (result);
30   }
31 }
```

Figura 5.11: Código fonte do cromossomo.

```java
class DecisaoAtacar extends FSGAITechnique {
 private Personagem npc;

 public boolean action (Personagem oponente) {
    try {
      GAParameter energia     = new GAParameter (8,0,100);
      GAParameter experiencia = new GAParameter (8,0,100);
      GAParameter defesa      = new GAParameter (8,0,100);
      GAParameter ataque      = new GAParameter (8,0,100);
      GAParameter raio        = new GAParameter (8,150,250);
      GAParameter velocidade  = new GAParameter (5,0,10);

      GA populacao = new GA (100,6);
      populacao.setElitism(2);

      populacao.addParameter(energia);
      populacao.addParameter(experiencia);
      populacao.addParameter(defesa);
      populacao.addParameter(ataque);
      populacao.addParameter(raio);
      populacao.addParameter(velocidade);

      populacao.setGenerationsNumber(1000);

      populacao.setRoulette(new GARoulette(populacao));
      populacao.setCrossover(new GACrossoverTwoPoints());

      GAMutation mutacao = new GAMutation(0.005);
      populacao.setMutation(mutacao);

      GAGeneticConvergenceKmean conv = new GAGeneticConvergenceKmean
         (5,35,20);
      conv.setPrint(false);
      conv.setDistance(60);
      populacao.testConvergenceEachGeneration (10,conv);

      //inserindo indivíduos
      for (int i=0;i<populacao.getPopulationSize();i++) {
        MeuCromossomo cromo;
        cromo = new MeuCromossomo(populacao.getChromosomeSize(),this.npc
           );
        populacao.add(cromo);
      }

      populacao.execute();

      GAChromosome cromo = populacao.getBetterChromosome();

      double[] valores = new double[populacao.getParameters().length];
      valores=cromo.getDecimalValue(populacao.getParameters());

      double dist = this.distancia(valores,oponente);

      if (dist<100) {
        return (false);
      }
      else {
        return (true);
      }
    }
    catch (GAException ex) {
       return (false);
    } catch (GACrossoverException ex) {
       return (false);
    } catch (GAMutationException ex) {
       return (false);
    } catch (GAGeneticConvergenceException ex) {
       return (false);
    } catch (GARouletteException ex) {
       return (false);
    } catch (GAParameterException ex) {
       return (false);
    } catch (GAChromosomeException ex) {
      return (false);
    }
 }
}
```

Figura 5.12: Código de utilização de GA.

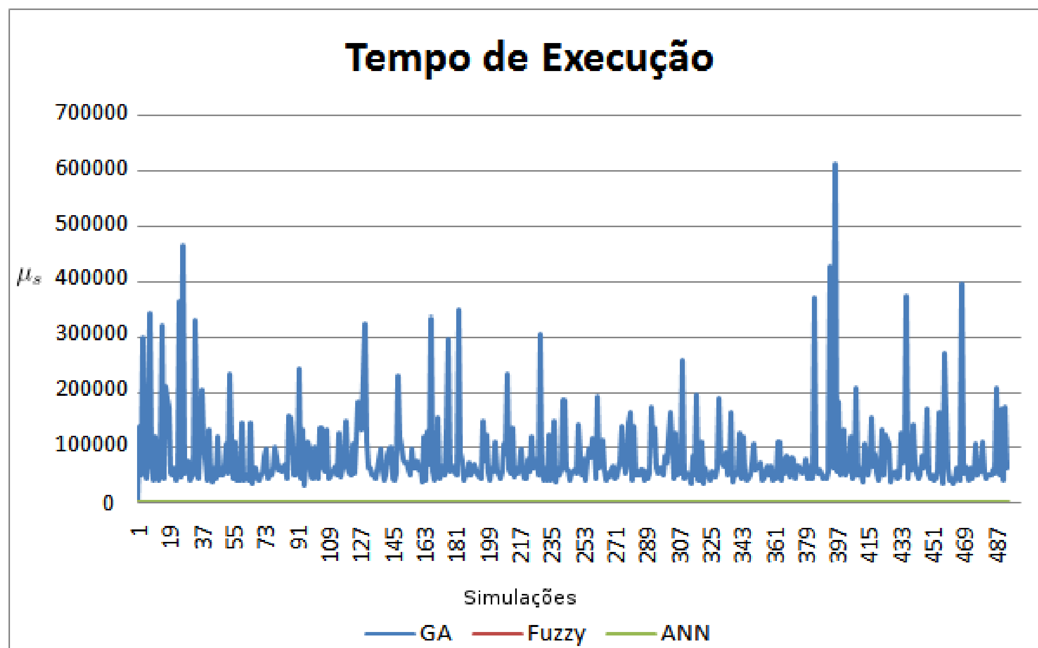

Figura 5.13: Comparação em micro segundos dos tempos de execução das três técnicas de IA.

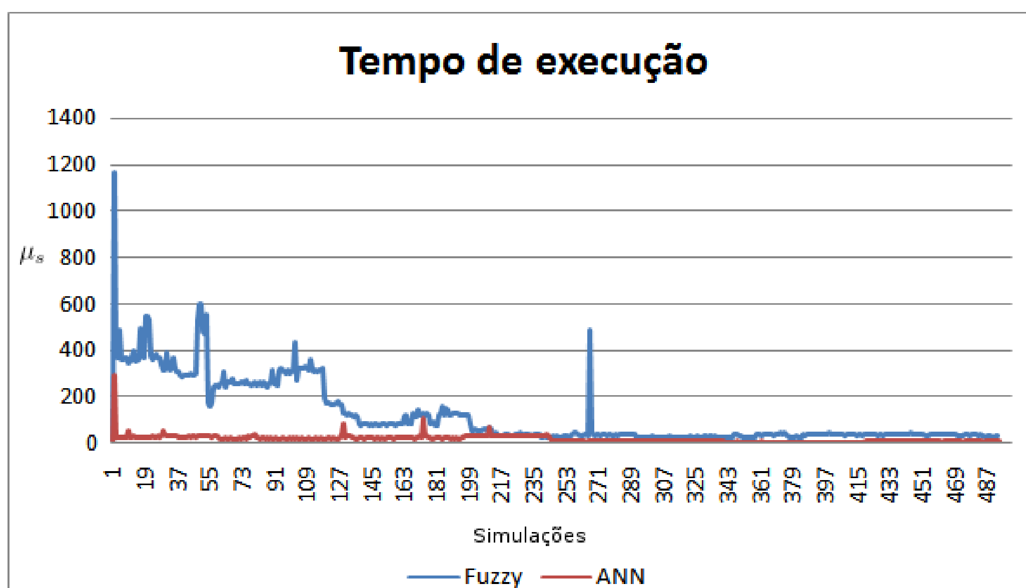

Figura 5.14: Comparação em micro segundos dos tempos de execução do ANN e do FIS.

Capítulo 6

Conclusão

De acordo com tudo que foi apresentado, a maior parte do valor agregado em um jogo advêm da qualidade de interação do software com o jogador, ou seja, da implementação adequada de técnicas de IA. No entanto, a utilização destas técnicas ainda é um desafio para a indústria [Bourg and Seemann, 2004], dado o nível de qualificação da mão de obra necessária e o curto tempo de desenvolvimento do jogos atuais. Neste contexto, o reuso de software adquire grande importância.

Em vista disto este trabalho visou contribuir para o uso intensivo de reuso de software no que diz respeitos ao emprego de técnicas de IA, focando-se a camada de tomada de decisão presentes nos NPCs. Para tanto, foi proposto o ambiente FURG Smart Games, composto por um framework e um conjunto de ferramentas RAD. Este ambiente engloba as Máquinas de Estados Finitos (FSM), sendo esta o cerne da tomada de decisão, os Algoritmos Genéticos (GA), Sistemas de Inferência Fuzzy (FIS) e Redes Neurais Artificiais (ANN). Tendo por objetivo agilizar o processo de desenvolvimento, além de associar as transições presentes nas FSMs as técnicas da GA, FIS e ANN, fazendo com que o comportamento do personagem seja menos previsível.

No que diz respeito ao GA, FIS e ANN, o framework se comporta como uma caixa preta, abstraindo toda a implementação, não exigindo um conhecimento aprofundado por parte do desenvolvedor para sua utilização, mas sim quais são as situações mais propícias do emprego de um técnica em detrimento das demais.

Para validar essa ferramenta foi desenvolvido um jogo utilizando-se o FSG para a camada de tomada de decisão e o framework JGame para a interface 2D. Todos os códigos da

camada de tomada de decisão foram gerados através das ferramentas RAD propostas, proporcionado grande agilidade na implementação. Houve a necessidade de eventuais códigos "cola" para a interligação os componente do FSG com as demais classes do jogo. Tendo em vista o tipo de jogo implementado e a estratégia adotada para a tomada de decisão apresentada no capítulo 5, a ANN teve melhor desempenho tendo um aproveitamento de 84%, enquanto que o FIS chegou a 50% e o GA a 65%.

Destacam-se os benefícios da utilização do ambiente FSG: primeiramente, tem-se a abstração da implementação das técnicas de IA; segundo, a agilidade da configuração dos testes e posterior geração no código fonte através dos RADs; por fim, o comportamento final dos NPCs torna-se menos previsível, se comparado com a FSM pura.

Relacionado aos desafios existentes na indústria de jogos eletrônicos, dos três elencados por [Bourg and Seemann, 2004], este trabalho foca no período de desenvolvimento do jogo, visto que utiliza técnicas de reuso de software. No que se refere aos outros dois, testes dos algoritmos de aprendizagem e poder de processamento, existe a limitação relacionada a o quê e onde a aplicação será executada, ou seja, dependendo da quantidade de personagens e técnicas de IA utilizadas, bem como sua plataforma.

Ainda, como fruto deste trabalho obteve-se algumas publicações científicas, inicialmente o artigo intitulado "*Using Artificial Intelligence in Computational Games*" publicado no periódico "*Journal of Information & System Management*" em Junho de 2011. E posteriormente o artigo "*FURG Smart Games: A Proposal for an Environment to Game Development with Software Reuse and Artificial Intelligence*" publicado no evento "*The Fourth International Conference on Networked Digital Technologies*" em 2012, onde foi destacado entre os melhores trabalhos do evento. Por fim, foi publicado o trabalho "*FURG Smart Games: a development enviroment to NPC decisions in games*", no periódico *Journal of Information & Systems Management*, em Junho de 2012.

No que diz respeito a trabalhos futuros, espera-se que este trabalho venha a ser um percursor de diversos outros. A princípio, no que tange as FSMs, pode-se estudar recente artigo de Hu et al. [Hu et al., 2011] que propõe uma nova técnica chamada "*Component-Based Hierarchical State Machine*" (CBHSM), que introduz técnicas de componente de software em FSM hierárquicas. Outra linha que pode ser atacada é o acréscimo de novas técnicas de IA ao FSG, como Processos de Decisão de Markov. Outro tema de interesse é

a investigação ou proposição de padrões de projetos para técnicas de IA, tais como ANN.

Um aspecto não tratado neste trabalho, refere-se a validação do framework desenvolvido. Essa validação é qualitativa e envolve testes com grupos de desenvolvedores de software, com o objetivo de verificar o ganho quantitativo (tempo) e qualitativo (previsibilidade) na produção do jogo.

Bibliografia

[Abdullah et al., 2008] Abdullah, Kamaruddin, Razak, e Mohd, B. (2008). A toolkit design framework for authoring multimedia game-oriented educational content. *Advanced Learning Technologies, Eighth IEEE International Conference*, páginas 144–145.

[Abragames, 2004] Abragames (2004). Plano diretor da promoção da indústria de desenvolvimento de jogos eletrônicos no brasil. Technical report, Abragames. Disponível em: http://www.abragames.org/page.php?id=downloads (acessado em Maio de 2011).

[Abragames, 2008] Abragames (2008). A indústria brasileira de jogos eletrônicos. um mapeamento do crescimento do setor nos últimos 4 anos. Technical report, Abragames. Disponível em: http://abragames.org/docs/Abragames-Pesquisa2008.pdf (acessado em Maio de 2011).

[Alberto, 2009] Alberto, A. D. B. (2009). Uma estratégia para a minimização de máquinas de estados finitos parciais. Mestrado em ciências de computação e matemática computacional, USP, São Paulo.

[Alves, 2008] Alves, L. R. G. (2008). Estado da arte dos games no brasil: trilhando caminhos. *Actas da Conferência ZON | Digital Games*, páginas 9–18.

[Aranha, 2007] Aranha, G. (2007). O processo de consolidação dos jogos eletrônicos como instrumento de comunicação e de construção de conhecimentos. *Ciências & Cognição*, 3:21–62.

[Barboza e Clua, 2009] Barboza, D. C. e Clua, E. W. G. (2009). Ginga game: A framework for game development for the interactive digital television. *VIII Brazilian Symposium on Games e Digital Entertainment*, páginas 105–111.

[Barboza e da Silva, 2009] Barboza, D. C. e da Silva, J. C. (2009). Ambiente visual para o desenvolvimento de jogos eletrônicos. *Revista TECCEN*, 2(1).

[Bittencourt, 2006] Bittencourt, G. (2006). *Inteligência Artificial - Ferramentas e Teorias*. UFSC, Florianópolis SC, 3ª ed.

[Bittencourt e Osório, 2001] Bittencourt, J. R. e Osório, F. (2001). Annef - artificial neural networks framework: Uma solução software livre para o desenvolvimento, ensino e pesquisa de aplicações de inteligência artificial multiplataforma. *WorkShop de Software Livre*, (II):13–16.

[Booch et al., 2000] Booch, Rumbaugh, e Jacobson (2000). *The Unified Modeling Language User Guide*. Addison-Wesley Professional.

[Bourg e Seemann, 2004] Bourg, D. e Seemann, G. (2004). *AI for Game Developers*. OReilly, 2ª ed.

[Castro e de Souza, 2010] Castro, L. e de Souza, T. S. (2010). Redes neurais artificiais aplicadas a reconhecimento de padrões em jogos eletrônicos. *Escola Regional de Informática do Rio de Janeiro*, (64-67).

[Charoenkwan et al., 2010] Charoenkwan, Fang, S.-W., e Wong, S.-K. (2010). A study on genetic algorithm e neural network for implementing mini-games. *Technologies e Applications of Artificial Intelligence*, pages 158–165.

[Chellapilla e Fogel, 1999] Chellapilla e Fogel (1999). Evolution, neural networks, games, e intelligence. *Proceedings of the IEEE*, 87(9):1471–1496.

[da Silva Oliveira e de Mattos, 2001] da Silva Oliveira, K. e de Mattos, H. D. (2001). Abordagens de reuso de software no desenvolvimento de aplicacoes orientadas a objetos. Disponível em: http://www.munif.com.br/munif/arquivos/reuso-1.pdf (acessado em Julho de 2012).

[Darryl e Stephen, 2004] Darryl e Stephen (2004). The past, present e future of artificial neural networks in digital games. *International Conference on Computer Games: Artificial Intelligence, Design e Education*. s.p.

[de Barros e Bassanezi, 2006] de Barros, L. C. e Bassanezi, R. C. (2006). *Tópicos de Lógica Fuzzy e Biomatemática.* IMECC/Campinas, 1ª ed.

[de Falco, 2007] de Falco, A. (2007). Jogos eletrônicos: do real ao virtual e vice-versa. *Inovação*, 3(3):52–55.

[de Faria Costa et al., 2011] de Faria Costa, I. A., de Souza, A. S., e Castanho, C. D. (2011). Gameka: Uma ferramenta de desenvolvimento de jogos para não programadores. *Simpósio Brasileiro de Games e Entreterimento Digital.* s.n. s.p.

[de Medeiros et al., 2006] de Medeiros, F. N., Medeiros, F. M., e Domínguez, A. H. (2006). Fa port: Um framework para sistemas portfólio-tutor utilizando agentes. *Simpósio Brasileiro de Informática na Educação*, (XVII):08–10.

[de Oliveira Cruz, 2011] de Oliveira Cruz, A. J. (2011). Um simulador de emoções utilizando lógica nebulosa. Monografia, Universidade Federal do Rio de Janeiro.

[de Santana, 2006] de Santana, R. T. (2006). Ia em jogos a busca competitiva entre homem e a máquina. Graduação em tecnologia em informática, Faculdade de Tecnologia de Praia Grande, Praia Grande, Brasil.

[Demasi e de Oliveira Cruz, 2003] Demasi, P. e de Oliveira Cruz, A. J. (2003). Evolução de agentes em tempo real para jogos eletrônicos de ação. *XI Simpósio Brasileiro de Sistemas Multimídia e Web.* s.p.

[Dinízio e Simões, 2003] Dinízio, C. S. e Simões, M. A. C. (2003). Inteligência artificial em jogos de tiro em primeira pessoa. *Revista Eletrônica de Iniciação Científica*, 3(1).

[Esparcia-Alcázar et al., 2010] Esparcia-Alcázar, A., Martínez-García, A., Mora, A., e García-Sánchez (2010). Controlling bots in a first person shooter game using genetic algorithms. *Evolutionary Computation (CEC)*, páginas 1–8.

[Ferreira, 2005] Ferreira, F. M. G. (2005). Desenvolvimento e aplicações de um framework orientado a objetos para análise dinâmica de linhas de ancoragem de risers. Mestrado em engenharia civil, Universidade Federal de Alagoas, Maceió, Alagoas.

[Franzosi et al., 2009] Franzosi, E. M., Garcia, A., e AL., E. (2009). Uma proposta de arquitetura referencial soa para desenvolvimento de sistemas para o governo. *Workshop de Computação Aplicada em Governo Eletrônico*, pages 1417–1430.

[Freeman et al., 2004] Freeman, Robson, Bates, e Sierra (2004). *Head First Design Patterns*. OReilly Media.

[Funge, 2004] Funge, J. D. (2004). *Artificial Intelligence for Computer Games: An Introduction*. AK Peters/CRC Press, 1ª ed.

[Gamma et al., 2000] Gamma, E., Helm, R., Johnson, R., e et al. (2000). *Padrões de Projeto - Soluções Reutilizaveis de Software Orientado a Objetos*. Bookman.

[Ganga, 2010] Ganga, G. M. D. (2010). Proposta de um modelo de simulação baseado em lógica fuzzy e scor para predizer o desempenho da empresa-foco em cadeis de suprimento. Doutorado, Universidade de São Paulo, São Paulo, SP, Brasil.

[Gimenes e Huzita, 2005] Gimenes, I. M. S. e Huzita, E. H. M. (2005). *Desenvolvimento Baseado em Componentes Conceitos e Técnicas*. Ciencia Moderna, 1ª ed.

[Hartigan e Wong, 1979] Hartigan e Wong (1979). A k-means clustering algorithm. *Journal of the Royal Statistical Society*, 28(1):100–108.

[Haykin, 2001] Haykin, S. (2001). *Redes Neurais Princípios e Práticas*. Bookman, Higienópulis - SP, 2ª ed.

[Hu et al., 2011] Hu, W., Zhang, Q., e Mao, Y. (2011). Component-based hierarchical state machine - a reusable e flexible game ai technology. *Information Technology e Artificial Intelligence Conference*, 2:319–324.

[Jepp et al., 2010] Jepp, P., Fradinho, M., e Pereira, J. M. (2010). An agent framework for a modular serious game. *Second International Conference on Games e Virtual Worlds for Serious Applications*, páginas 19–26.

[Joselli e Clua, 2009] Joselli, M. e Clua, E. (2009). grmobile: A framework for touch e accelerometer gesture recognition for mobile games. *VIII Brazilian Symposium on Games e Digital Entertainment*, páginas 136–142.

[Kochanski, 2009] Kochanski, D. (2009). Um framework para apoiar a construção de experimetos na avaliação empírica de jogos educacionais. Mestrado, Universidade do Vale do Itajaí.

[Kubo, 2006] Kubo, M. M. (2006). Fmmg: Um framework para jogos multiplayer móveis. Doutorado, Universidade de São Paulo.

[Larman, 2004] Larman (2004). *Applying UML e Patterns: An Introduction to Object Oriented Analysis e Design e Iterative Development*. Bookman.

[Linden, 2008] Linden, R. (2008). *Algoritmos Genéticos - Uma Importante Ferramenta da Inteligência Computacional*. Brasport, Rio de Janeiro / RJ, 2ª ed.

[Madsen et al., 2012] Madsen, C. B. C. W., Lucca, G., Daniel, G., e Adamatti, D. F. (2012). Furg smart games: a proposal for an environment to game development with software reuse e artificial intelligence. *The Fourth International Conference on Networked Digital technologies*, páginas 369–381.

[Malfatti e Fraga, 2006] Malfatti, S. M. e Fraga, L. M. (2006). Utilizando behaviors para o gerenciamento da máquina de estados em jogos desenvolvidos com java 3d. *Simpósio Brasileiro de Games e Entreterimento Digital*.

[Millington, 2006] Millington, I. (2006). *Artificial Intelligence for Games*. MK Morgan Kaufmann, 1ª ed.

[Motta et al., 2008] Motta, L., Contreras, J., e Osório, F. (2008). Sdk gameplay - ferramenta voltada para ediÇÃo de gameplay. *Simpósio Brasileiro de Games e Entreterimento Digital*. s. p.

[Nareyek, 2001] Nareyek (2001). Constraint-based agents - an architecture for constraint-based modeling e local-search-based reasoning for planning e scheduling in open e dynamic worlds. *LNAI*, páginas 369–381.

[Packer, 2010] Packer (2010). *Neurociência - Elementos de Neuro-fisiologia Farmacologia Psiquiatria*. Disponível em: http://www.filosofiaadistancia.com.br/ (acessado em Novembro de 2010).

[Perucia et al., 2011] Perucia, A., Balestrin, A., e Verschoore, J. (2011). Coordenação das atividades produtivas na indústria brasileira de jogos eletrônicos: hierarquia, mercado ou aliança? *Produção*, 21(1):64–75.

[Piske e Seidel, 2006] Piske, O. e Seidel, F. (2006). Rapid application development. Especialização em software livre, Centro Universitário Positivo - UnicenP.

[Plant et al., 2008] Plant, W., Schaefer, G., e Nakashima, T. (2008). An overview of genetic algorithms in simulation soccer. *IEEE World Congress on Computational Intelligence*, pages 3897–3904.

[PWC, 2010] PWC (2010). Global entertainment e media outlook: 2010-2014. Technical report, PricewaterhouseCoopers LLP, USA.

[Rabin, 2002] Rabin, S. (2002). *AI Game Programming Wisdom*. Charles River Media, 1ª ed.

[Reeder et al., 2008] Reeder, Miguez, Sparks, Georgiopoulos, e Anagnostopoulos (2008). Interactively evolved modular neural networks for game agent control. *Computational Intelligence e Games*, páginas 167–174.

[Rezende, 2005] Rezende, S. O. (2005). *Sistemas Inteligentes Fundamentos e Aplicações*. MAN - MANOLE, 1ª ed.

[Rieder e Brancher, 2004] Rieder, R. e Brancher, J. (2004). Aplicação lógica fuzzy a jogos didáticos de computador - a experiência do mercadão gl. *Actas do VII Congresso Iberoamericano de Informática Educativa*, páginas 127–136.

[Ritu et al., 2000] Ritu, P., Tanniru, J. M., e Lynch, J. (2000). Risks of rapid application development. *Communications of ACM*, 43(11):177–188.

[Russel, 2004] Russel, S. (2004). *Inteligência Artificial*. Campus, Rio de Janeiro RJ Brasil, 2ª ed.

[Salva, 2011] Salva, T. (2011). Sisbolsas: Informatização do processo de gestão das bolsas de permanência na universidade federal do rio grande utilizando o framework casca.

Graduação em tecnologia em análise e desenvolvimento de sistemas, IFRS, Rio Grande, Brasil.

[Sanches, 2012] Sanches, B. C. (2012). Utilizando lógica fuzzy para controle de jogos. Link: http://www.pontov.com.br/ (acessado em Agosto de 2012).

[Schwab, 2004] Schwab, B. (2004). *AI Game Engine Programming*. Hingham: Charles River Media, 2ª ed.

[Seára e Benitti, 2006] Seára, E. e Benitti, F. (2006). Incrementando a produtividade de software através de ferramenta rad e framework. *Revista Hífen*, 30(58).

[Silva et al., 2006] Silva, D., Tedesco, P., e Ramalho, G. (2006). Usando atores sintéticos em jogos sérios: O case smartsim. *Brazilian Symposium on Games e Digital Entertainment*. s.p.

[Silva, 2000] Silva, R. (2000). Suporte ao desenvolvimento e uso de frameworks e componentes. Doutorado em ciência da computação, UFRGS, Porto Alegre.

[Simões e Shaw, 2007] Simões, M. G. e Shaw, I. (2007). *Controle e Modelagem Fuzzy*. Blucher, 2ª ed.

[Smed e Hakonen, 2006] Smed e Hakonen (2006). *Algorithms e Networking for Computer Games*. John Wiley & Sons Ltd, 1ª ed. University of Turku, Finland.

[Soares, 2009] Soares, W. (2009). *Crie um Framework para Sistemas Web com PHP 5 e AJAX*. Érica, 1ª ed.

[Softex, 2005] Softex (2005). Tecnologias de visualização na indústria de jogos digitais. potencial econômico e tecnológico para a industria brasileira de software. Technical report, Observatório Digital Softex.

[Sommerville, 2007] Sommerville, I. (2007). *Engenharia de Software*. Addison-Wesley, 8ª ed.

[Stanley et al., 2005] Stanley, K., Bryant, B., e Miikkulainen, R. (2005). Evolving neural network agents in the nero video game. *Symposium on Computational Intelligence e Games*. s.n. s.l.

[Sweetser e Wiles, 2002] Sweetser e Wiles (2002). Current ai in games: a review. *Australian Journal of Intelligent Information Processing Systems*, 8(1):24–42.

[Tang e Wan, 2002] Tang, W. e Wan, T. R. (2002). Intelligent self-learning characters for computer games. *Eurographics UK Conference*, páginas 51–58.

[van Schooten, 2012] van Schooten, B. (2012). *JGame - a Java/Flash game engine for 2D games*. Disponível em: http://www.13thmonkey.org/~boris/jgame/ (acessado em Julho de 2012).

[Vermaas, 2010] Vermaas, L. L. G. (2010). Aprendizado supervisionado de sistemas de inferência fuzzy aplicados em veículos inteligentes. Doutorado em ciências em engenharia elétrica na área de automação e sistemas elétricos industriais, Universidade Federal de Itajubá.

[Victor et al., 2010] Victor, Oliveira, Pablo, e Soares (2010). Brian - um agente inteligente cognitivo e deliberativo para aplicações em jogos de computadores. *III WORKSHOP DE INFORMÁTICA APLICADA*. s.p.

[Weschter, 2008] Weschter, E. O. (2008). Arquitetura do gerador de aplicação web baseado no framework titan. Mestrado em ciência da computação, Fundação Universidade Federal de Mato Grosso do Sul.

[Xu, 2008] Xu, C.-W. (2008). A software framework for online mobile games. *Computer Science e Software Engineering*, 2:558–561.

[Zadeh, 1965] Zadeh, L. A. (1965). Fuzzy sets. *Information e Control*, (8):338–353.