ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de
Sistemas
Computacionais

MAFIDS
Multi-Agent Framework for Intrusion Detection Systems

Trabalho de Conclusão de Curso

Engenharia da Computação

Bruno de Melo Arôxa
Orientador: Prof. Tiago Massoni

Recife, 22 de novembro de 2006

ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de
Sistemas
Computacionais

MAFIDS
Multi-Agent Framework for Intrusion Detection Systems

Trabalho de Conclusão de Curso

Engenharia da Computação

Este Projeto é apresentado como requisito parcial para obtenção do diploma de Bacharel em Engenharia da Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Bruno de Melo Arôxa
Orientador: Prof. Tiago Massoni

Recife, 22 de novembro de 2006

UNIVERSIDADE
DE PERNAMBUCO

Bruno de Melo Arôxa

MAFIDS
Multi-Agent Framework for Intrusion Detection Systems

Resumo

Nos últimos anos os investimentos em segurança de informação têm sido intensificados, em virtude do aumento registrado no número de ataques sofridos por empresas no mundo todo. Segundo a pesquisa realizada pela *PricewaterhouseCoopers* (*PwC*) este ano com profissionais de Tecnologia de Informação (TI) em todo o mundo, os Sistemas de Detecção de Intrusos (SDI) têm se apresentado como uma das ferramentas mais eficazes na descoberta de falhas de segurança. Apesar disso, menos da metade dos entrevistados adotam esse tipo de ferramenta como parte da estratégia de segurança da empresa. Acredita-se que as limitações inerentes da arquitetura monolítica dos SDIs tradicionais em conjunto com o alto custo de desenvolvimento deste tipo de ferramenta tenha colaborado para o seu baixo índice de adoção no mercado corporativo.

Este trabalho propõe o desenvolvimento de um *framework* que possibilite a construção de SDIs com baixo custo e que resolva os problemas presentes na sua arquitetura tradicional – de núcleo monolítico; pretende-se utilizar a tecnologia de SMAs – Sistemas Multi-Agentes para solucionar esses problemas. A utilização do *framework* proposto possibilitará a construção de SDIs com todas as características que os SMAs proporcionam: flexibilidade, facilmente expansível, de núcleo distribuído e facilidade de reconfiguração. Além disso, a qualidade de software obtida com a aplicação de padrões de projeto, resulta no desenvolvimento de um *framework* fundamentado no paradigma de orientação a objetos (OO). Desta forma, pretende-se incentivar a adoção de SDIs como parte de uma infra-estrutura de segurança mais complexa.

Abstract

Recently, investiments on information security have been intensified due to the increase of the number of attacks suffered by organizations. According to a research carried through by *PricewaterhouseCoopers* (*PwC*) in 2006 with Information Technology (IT) professionals in several countries, Intrusion Detection Systems (IDS) have been pointed out as one of the best methods that organizations may employ to detect breaches. Despite these numbers, less than half of the interviewees adopt this kind of tool as part of the overall company security strategy. We believe that limitations inherent to traditional monolithic IDS architecture together with high development costs are collaborating with the low IDS adoption.

This work proposes a framework development that makes IDS building easier, with lower costs, solving issues present in its traditional architerture (with a monolithic kernel). We use Multi-Agent System (MAS) technology to tackle those problems. The proposed framework enables building IDSs with SMA features such as flexibility, extensibility, with distributed kernel and easily reconfigurable. Moreover the software quality reached with design patterns appliance, came out with a fully OO framework developing. This may lead to higher IDS adoption as part of a more powerful security infrastructure.

Sumário

Índice de Figuras

Índice de Tabelas

Tabela de Símbolos e Siglas

AAFID	– *Autonomous Agents for Intrusion Detection*
AAM	– *Agentes Autônomos Móveis*
ACK	– *Acknowledgment*
AES	– *Advanced Encryption Standard*
AMEP	– *Agent Messaging Exchange Protocol*
CERIAS	– *Center for Educational and Research in Information Assurance and Security*
DES	– *Data Encryption Standard*
GUI	– *General User Interface*
IA	– *Inteligência Artificial*
IDE	– *Integrated Development Environment*
IP	– *Internet Protocol*
JVM	– *Java Virtual Machine*
MAFIDS	– *Multi-Agent Framework for Intrusion Detection Systems*
OO	– *Orientação a Objetos*
PGP	– *Pretty Good Privacy*
RFC	– *Request for Comment*
RSA	– *Rivest, Shamir, Adleman*
SDI	– *Sistema de Detecção de Intrusos*
SDIH	– *Sistema de Detecção de Intrusos baseado em Host*
SDIR	– *Sistema de Detecção de Intrusos baseado em Rede*
SI	– *Segurança de Informação*
SMA	– *Sistemas Multi-Agentes*
TCP	– *Transmission Control Protocol*
TI	– *Tecnologia de Informação*
UML	– *Unified Modeling Language*
XML	– *eXpansible Markup Language*

Agradecimentos

Agradeço primeiramente a Deus, porque Ele nos amou primeiro. Senhor Jesus, obrigado por mais essa conquista que tu me concedes; por derramar sobre mim a Tua sabedoria para que este trabalho fosse concluído e nisso, o Teu nome fosse glorificado; por colocar no meu caminho as pessoas certas, nos momentos mais oportunos, que me ajudaram a não perder o foco diante das inúmeras dificuldades que surgiram.

À minha linda esposa Taciana Arôxa, por sua dedicação, paciência, apoio, compreensão, carinho, enfim, por seu amor. Obrigado por estar ao meu lado, ombro a ombro, durante todo esse tempo, me dando forças para superar as dificuldades.

Aos meus pais, Ricardo e Cristiane Arôxa, por todo esforço, carinho e dedicação na formação do meu caráter, que me tornaram a pessoa que sou hoje.

Aos meus pastores Ranilson e Simone, pela cobertura espiritual, paciência e compreensão que tiveram durante essa jornada.

Aos meus professores: Carlos Alexandre, Ricardo Massa, Adriano Lorena, Fernando Buarque, Marcio Lopes, Maria Lencastre; obrigado pelo ótimo trabalho que vocês vem fazendo no curso de engenharia da computação na POLI, trabalho este que resultou na minha formação acadêmica.

Aos meus queridos amigos, companheiros de turma: César, Pedro, Túlio, Adriano, Chapa, pelas 'rodadas de estudos', noites de trabalhos, por tudo o que aprendi com vocês durante esses cinco anos de estudos.

Ao meu professor e orientador Tiago Massoni, por seus valiosos conselhos, sempre colocando as metas de forma clara e objetiva, o que facilitou o cumprimento das atividades no cronograma.

Ao C.E.S.A.R pela experiência adquirida neste último ano, que foi essencial para o desenvolvimento deste projeto.

Aos meus amigos, colegas de trabalho, que me deram apoio sempre que precisei nesta reta final do projeto. Especialmente a Leandro Marques, pelas dicas de reuso, e a Daniel Melo, por sua chata insistência em me manter no foco para finalizar a escrita desta monografia.

Enfim, a todos que de alguma forma contribuíram para a realização deste projeto, que começou com uma vaga idéia de algo que eu nunca tinha trabalhado e que hoje me faz uma pessoa um pouco mais realizada.

Obrigado Senhor, por cada conquista durante esses cinco anos.

Capítulo 1

Introdução

O despreparo dos protocolos de rede e sistemas para o modelo de negócio utilizado atualmente – *e-commerce*, *e-business*, transações bancárias on-line, etc. – tem contribuído para um crescimento expressivo dos *cybercrimes* (crimes de computador). Semanalmente são divulgadas novas vulnerabilidades nos protocolos, sistemas operacionais e serviços de rede [5].

Os ataques aos sistemas são em sua maioria efetuados em cima de vulnerabilidades que passam despercebidas pelo administrador, que em geral são mais fáceis de serem encontradas pelos atacantes. Soares, Lemos e Colcher [27] definem um ataque como "a violação intencional das regras de uma política de segurança, visando à destruição, modificação, roubo ou perda da informação ou recurso, revelação de informação ou interrupção de serviços de um sistema". Existem várias formas de ataque a um sistema, e também existem métodos e ferramentas de defesa, para a prevenção contra os ataques que possam vir a ocorrer, diminuindo as chances de invasão de um sistema. Mecanismos de defesa comuns são: a política de segurança, a criptografia, os *firewalls* e os Sistemas de Detecção de Intrusos (SDI). As próximas seções apresentam, de forma detalhada, a caracterização do problema e os objetivos a serem alcançados por este projeto.

1.1 Caracterização do Problema

Nos últimos anos, as falhas de segurança encontradas nas redes e sistemas em geral, têm chamado à atenção dos especialistas e aumentado o orçamento de Tecnologia da Informação (TI) das empresas, que têm investido maciçamente em ferramentas, capacitação e políticas de conscientização em busca da integridade e segurança de informação. No entanto, apesar de o orçamento para segurança nas empresas ter aumentado, o desperdício de recursos tem se tornado o grande fator que impede a segurança efetiva [8]. Diante deste cenário, a importância da segurança da informação é evidenciada, principalmente nas grandes empresas, onde a complexidade da administração da rede e o tráfego de informações são bastante elevados, fatores que contribuem para o trabalho dos invasores.

Segundo pesquisa realizada pelo *CIO, CSO* e *PricewaterhouseCoopers* [9] com mais de 7000 profissionais de TI em 50 países, revelou que apesar dos investimento em Segurança de Informação (SI) vir aumentado gradativamente nos últimos anos (tendo registrado um aumento de 4% no último ano, quando foi investido em segurança cerca de 17% do investimento total em TI) as implementação de ações de segurança (políticas, planos de recuperação, ferramentas, etc.)

quando não cresceram lentamente, regrediram. A Figura 1-1 apresenta a evolução do investimento em segurança (em termos percentuais do investimento total em TI) nos últimos anos. Dentre as ações em cuja implantação observou-se uma redução no último ano estão os SDIs que, apesar de serem considerados um dos melhores mecanismos de detecção de falhas de segurança, a sua implantação caiu de 49% para 47% segundo a mesma pesquisa [9].

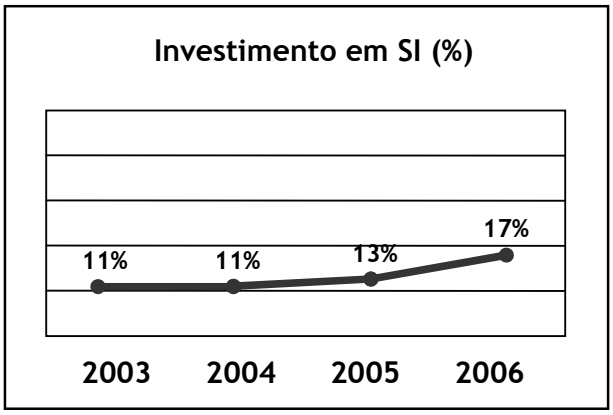

Figura 1-1 Evolução do investimento em segurança em termos percentuais do investimento total em TI.

Dentre os crimes de computador mais citados na pesquisa, destacam-se: os crimes que resultaram em perda financeira, que foi relatado por 19% das companhias em todo o mundo; em seguida aparece o crime de roubo de endereços IP – *Internet Protocol* (para mais informações sobre os protocolos de rede, a exemplo do IP, consulte [17][20][27][31]); o terceiro crime mais sofrido pelas empresas no mundo é o de fraude com um índice de 9%; e ainda o crime de extorsão, que foi observado em 5% das empresas em todo o mundo. A Figura 1-2 apresenta um resumo desses crimes.

Crimes de computador mais sofridos pelas empresas em todo o mundo em 2006

Perdas Financeiras	19%
Roubo de IP	12%
Fraude	9%
Extorsão	5%

Figura 1-2 Resumo dos crimes de computador mais sofridos pelas empresas (no mundo) em 2006.

Dentre os principais mecanismos de segurança apontados pelos entrevistados, destacam-se os *firewalls,* políticas de segurança e os SDIs, tendo este último sido apontado como um dos melhores mecanismos para detecção de falhas de segurança [9]. Os SDIs monitoram aspectos dos *hosts* espalhados na rede e montam um perfil desse *host.* Após montado o perfil, algumas abordagens procuram por anomalias no comportamento desse *host* para detectar a intrusão; uma

outra abordagem procura assinaturas de ataques (ataques com padrões específicos, nos quais se percebe uma seqüência clara de eventos ou dados) na tentativa de inibi-los. *Muhkerjee* [23] apresenta mais detalhes sobre esses tipos de SDI.

Diante do fato de que menos da metade das empresas no mundo adotaram a solução de SDIs (apontados como um dos melhores mecanismos de detecção de falhas de segurança) como parte do conjunto de SI, percebe-se a necessidade de criação de mecanismos que facilitem o desenvolvimento desse tipo de ferramenta de segurança.

Um dos possíveis motivos da resistência à implantação dos SDI no meio corporativo são as limitações conhecidas (a exemplo da escalabilidade limitada, dificuldade de reconfiguração do sistema, utilização de um módulo central de analise, etc.) existentes nos modelos tradicionais de SDI – de núcleo monolítico – dentre os quais se destacam os Sistemas de Detecção de Intrusos baseado em *Host* (SDIH) e os Sistemas de Detecção de Intrusos baseado em Rede (SDIR). Uma das motivações desse projeto é a proposta de uma ferramenta que permita a construção de SDIs livres das limitações inerentes da arquitetura monolítica, o que impulsionaria a adoção de SDIs no meio corporativo.

Uma outra forma de incentivar o desenvolvimento de SDIs é com a redução de custos e do tempo de desenvolvimento. Além da escalabilidade, expansibilidade, facilidade de reconfiguração, etc. – que são características desejáveis a qualquer sistema – um outro aspecto deve ser levado em consideração por impactar diretamente os custos de projeto: a facilidade de manutenção do sistema. A utilização de um framework permite o reuso dos componentes já desenvolvidos e validados, oferecendo uma infra-estrutura básica para a especialização desse framework na construção de um sistema específico. Segundo *Sommerville* [29], um *framework* pode ser definido como um conjunto de classes (concretas e abstratas) ligadas por meio de interfaces que podem ser estendidos para criar uma aplicação mais específica ou um subsistema.

No intuito de fomentar o desenvolvimento de SDI de forma simples, viabilizando a implantação deste tipo de ferramenta de segurança com um baixo custo, este projeto propõe o desenvolvimento de um *framework* que possibilite a construção de SDIs de núcleo *não-monolítico*, ou seja, cujas operações de coleta, manipulação e análise de dados, bem como a tomada de decisões não sejam de responsabilidade de um único módulo do sistema. Para tanto o *framework* utilizará a tecnologia de Sistemas Multi-Agentes (SMA) que permite o desenvolvimento de sistemas utilizando múltiplos agentes com responsabilidades específicas. Para efeito de teste e validação do *framework* proposto, o MAFIDS será instanciado numa aplicação específica para detecção de *backdoors* (*BackIDS*), a qual será testada em ambiente real de trabalho (um conjunto restrito de máquinas de uma empresa de TI do Porto Digital) para posterior avaliação dos resultados obtidos.

Outros projetos semelhantes foram propostos na busca de superar as limitações inerentes da arquitetura monolítica utilizada nos SDIs tradicionais. *Crosbie*. e *Spafford* [6], [7] propõem a construção de um SDI baseado em Agentes Autônomos Móveis (AAM) [26] que resolve as limitações das arquiteturas tradicionais. Bernardes [1] propôs um SDI multicamadas também utilizando AAM. O *framework* proposto por este projeto – *Multi-Agent Framework for Intrusion Detection Systems* (MAFIDS) – permite o desenvolvimento de SDIs bastante parecidos com o proposto por Bernardes. A tecnologia de AAM foi utilizada ainda por *Foukia* [11] na proposta de um SDI que utiliza uma abordagem semelhante ao sistema imunológico humano.

1.2 Objetivos e Metas

Durante o processo de desenvolvimento do projeto alguns objetivos (parciais) podem ser observados. Percebe-se ainda que algumas dessas etapas possuem dependências entre si. Abaixo

estão listados os objetivos a serem cumpridos durante o desenvolvimento do projeto, ordenados por dependência, a fim de alcançar a meta do projeto: o desenvolvimento do MAFIDS.

I. *Determinar o protocolo de interoperabilidade entre os agentes:* Contempla uma das atividades mais importantes no desenvolvimento do projeto, visto que o protocolo de comunicação determinará a interoperabilidade entre os agentes;

II. *Determinar o modelo em camadas, que definirá a arquitetura do MAFIDS:* Esta etapa define os níveis de atuação dos agentes do sistema. A arquitetura proposta é baseada nesse modelo;

III. *Implementar o protocolo de comunicação entre agentes:* Esta atividade contempla a codificação do protocolo já definido anteriormente;

IV. *Codificar as classes do MAFIDS:* Etapa que define a estrutura de classes básica do *framework* de acordo com os padrões de projeto indicados;

V. *Instanciar o MAFIDS:* Nesse momento será construído um SDI para um cenário específico – um SDI para detecção de *backdoors* – *BackIDS* – na tentativa de avaliar a aplicação do *MAFIDS* numa aplicação específica.

1.3 Organização dos Capítulos

O texto desta monografia está organizado da seguinte maneira:

- O *Capítulo 2* apresenta os conceitos básicos de segurança computacional necessários para o entendimento deste trabalho bem como apresenta os conceitos de SMA e AAM – tecnologia que dá suporte ao desenvolvimento deste projeto;

- O *Capítulo 3* apresenta a teoria necessária para o entendimento acerca de *frameworks* – características, benefícios e limitações – e dos principais padrões utilizados neste projeto – tais como *Singleton, Strategy, Factory,* etc.

- O *Capítulo 4* apresenta a proposta para um *framework* baseado em SMA (MAFIDS) que possa ser utilizado para a construção de SDIs de núcleo *não-monolítico,* a um baixo custo e com um tempo de desenvolvimento mínimo. Este capítulo apresentará a arquitetura definida para o *framework,* detalhando o modelo e os componentes dessa arquitetura; além disso, será apresentada a definição do protocolo de comunicação entre agentes, proposto neste projeto;

- O *Capítulo 5* apresenta uma aplicação do MAFIDS para o desenvolvimento de um SDI específico. O cenário utilizado para a instanciação do MAFIDS é o desenvolvimento de um SDI para detecção de *backdoors* – o *BackIDS;*

- O *Capítulo 6* apresenta as conclusões para este trabalho e a proposta de trabalhos futuros.

Capítulo 2

Sistemas de Detecção de Intrusos (SDI) e Sistemas Multi-Agentes (SMA)

Diante das limitações inerentes da arquitetura monolítica dos SDIs tradicionais – caracterizada pela presença de um grande módulo monolítico que, segundo *Crosbie e Spafford* [7], é responsável por realizar a coleta, análise e manipulação dos dados, além de ser responsável pela tomada de decisão – a comunidade científica vem procurando mecanismos alternativos para o desenvolvimento de SDIs. Os SMAs vêm tomando força no desenvolvimento de SDIs de núcleo distribuído, onde as responsabilidades são delegadas para *agentes de software* (as definições sobre SMA e agentes serão discutidas mais adiante neste capítulo) especializados em atividades específicas. As seções seguintes apresentam as definições, conceitos e características acerca dos SDIs – abordando as características desejáveis e suas limitações – e SMAs, abordando os conceitos sobre os agentes de *software,* AAM, etc.

2.1 Sistemas de Detecção de Intrusos

Dentre os diversos mecanismos de segurança disponíveis no mercado (ou como projetos no meio acadêmico), os SDIs vêm se destacando pela sua eficácia contra um tipo de ataque que vem aumentando gradativamente nos últimos anos: os ataques de origem interna – realizados por usuários legítimos que estão fazendo mau uso do sistema [30], contra os quais os *firewalls* – mecanismos de proteção de perímetro largamente adotados – são completamente indefesos [27].

Os SDI aparecem como uma solução complementar aos *firewalls*, visto que eles monitoram o tráfego da rede em busca de intrusos, devendo no mínimo reportar ao administrador da rede a ocorrência de intrusão no sistema [30]. A adoção dos SDI vem aumentando a cada ano, e já se pode perceber uma tendência de integração desse tipo de ferramenta com os já adotados mecanismos de proteção de periferia da rede, como os *firewalls* [8].

Segundo *Heady* [14], uma intrusão pode ser definida como "Qualquer conjunto de ações que tentem comprometer a integridade, confidencialidade ou disponibilidade de um recurso". Existem duas abordagens principais utilizadas para o termo *intrusão*, que *Mukherjee, Herberlein e Levitt* [23] definem da seguinte forma:

- *Intrusão devido ao mau uso do sistema*: A detecção é feita analisando os pontos fracos conhecidos do sistema, os quais podem ser descritos por um padrão específico, ou por uma seqüência de eventos ou dados (a "assinatura" da intrusão);

- *Intrusão devido à mudança de padrão*: A detecção é feita analisando mudanças no padrão de utilização ou no comportamento do sistema. Para tanto *Kumar* e *Spafford* [19] sugerem montar um perfil do sistema para que se possa comparar o comportamento atual com o perfil construído em busca de divergências significativas entre os dois modelos.

2.1.1 Características Desejáveis dos SDIs

Sendo uma das ferramentas de segurança mais importante do conjunto que compõe a infra-estrutura de SI, algumas características mínimas são esperadas dos SDIs. Abaixo seguem algumas das características definidas por *Crosbie* e *Spafford* [6] como desejáveis para um SDI:

- O sistema deve *executar continuamente* com o mínimo de supervisão humana;
- O sistema deve ser *tolerante a falhas,* ou seja, ser capaz de recuperar-se após uma interrupção, seja ela acidental ou resultado de um ataque, devendo retornar ao estado anterior à interrupção;
- O sistema não deve sobrecarregar o *host* a ponto de interferir nas atividades normais do mesmo;
- O sistema deve ser configurável para se adaptar a novas realidades de atividade do *host* monitorado;

Para este projeto serão consideradas também as características apontadas por *Zamboni* [33], que abordam a realidade dos SDIs de núcleo não-monolítico:

- O sistema deve ser *escalável* para monitorar um grande número de *hosts* retornando os resultados em um tempo satisfatório;
- O sistema deve lidar adequadamente com a degradação de seus componentes, no sentido de que se um dos componentes do sistema (um agente, por exemplo) estiver impossibilitado de realizar suas atividades, o resto do sistema deve ser afetado o mínimo possível;
- O sistema deve permitir *reconfiguração dinâmica.* Quando o sistema estiver monitorando um grande numero de *hosts,* fica impraticável reiniciar todos os elementos para realizar a atualização do mesmo.

2.1.2 Limitações dos SDIs Existentes

As implementações de SDIs – SDIH e SDIR (para mais informações sobre esses tipos de SDIs consulte [23]) – tradicionais apresentam algumas limitações que reduzem a sua capacidade de configuração, escalabilidade ou eficiência [7]. *Mukherjee*, *Herberlein* e *Levitt* apresentam um resumo dos problemas que podem ser observados nessas abordagens [23], dentre os quais estão:

- *O analisador central é também um ponto central de falhas*: Caso um intruso ataque esse bloco do sistema, impedindo o processamento dos dados coletados, toda a rede que está sendo coberta pelo SDI ficará vulnerável;
- *A escalabilidade é limitada*: Como todo o processamento é realizado num único servidor, o consumo de recursos dessa máquina trabalha como um limitador e determina o número máximo de agentes que podem ser processados ao mesmo tempo;
- *Dificuldade de reconfiguração*: Alterações para adicionar novas funcionalidades, ampliar o poder de detecção, corrigir problemas, etc., envolvem retrabalho no SDI como um todo;

- *A análise dos dados pode ser forjada*: O fato de os dados coletados serem analisados em um outro *host* (diferente daquele onde os dados foram coletados) torna o sistema vulnerável a ataques durante o envio dos dados [32].

Uma das motivações desse projeto é solucionar os problemas inerentes de arquiteturas monolíticas (descritos acima) pois acredita-se que esses problemas podem estar contribuindo para a resistência à adoção de SDIs em ambientes corporativos [9]. Uma outra abordagem que vem sendo estudada atualmente propõe a construção de SDIs utilizando Agentes Autônomos Móveis – AAM (os conceitos de SMAs e AAM será explicados com mais detalhes na próxima seção), na tentativa de resolver os problemas apresentados por *Mukherjee*, *Herberlein* e *Levitt* [23].

2.2 Sistemas Multi-Agentes e Agentes Móveis

Ainda não existe uma definição exata para o termo agente. *Russel e Norving* [26] definem, de forma mais abrangente, um agente como "tudo o que pode ser considerado capaz de perceber seu ambiente por meio de sensores e de agir sobre esse ambiente por intermédio de atuadores". *Nikitas* e *Kin* [24] definem um agente de software como uma entidade que age de acordo com os interesses dos outros de forma autônoma; executa suas atividades com algum nível de pró-atividade e reatividade; e apresenta algum nível de aprendizado, cooperação e mobilidade.

As características como execução contínua, tolerância a falhas, facilidade de expansão, facilidade de reconfiguração, entre outras apresentadas pelos agentes, vêm sendo bastante exploradas na comunidade científica para a solução de problemas conhecidos no domínio do desenvolvimento de SDI. Após analisar várias definições sobre o que é um agente, *Franklin* e *Graesser* [12] definiram um conjunto de características que um agente pode possuir. A Tabela 2-1 resume essas características de forma a facilitar a classificação dos diversos tipos de agentes.

Diante das classificações apresentadas e das definições dadas por outros autores, nesse trabalho o termo *agente* será utilizado para denotar um componente de *software* com capacidade de locomoção e com características pró-ativas.

Tabela 2-1 Características possíveis de ser encontradas nos agentes [12].

Propriedade	**Significado**
Reativo	Responde conforme as mudanças no ambiente
Autônomo	Exerce controle sobre suas ações
Orientado a Metas	Não age simplesmente em função do ambiente
Contínuo	É um processo executado continuamente
Comunicativo	Comunica-se com outros agentes
Inteligente	Muda seu comportamento com base na experiencia anterior
Móvel	Capaz de se mover de uma máquina para a outra
Flexível	Ações não são definidas através de scripts
Caráter	Personalidade e estado emocionais críveis

Os SMAs podem ser caracterizados pela interação de vários agentes que trabalham em prol de um objetivo único – agentes cooperativos, ou pela interação entre agentes cujos objetivos são divergentes, ou seja, para um agente alcançar seu objetivo outro deverá falhar – agentes

competitivos [26]. A primeira abordagem, por apresentar um comportamento colaborativo, será utilizada neste trabalho, na proposta de um SDI de núcleo distribuído.

2.2.1 Estrutura dos Agentes

Segundo *Russel e Norving* [26], a estrutura básica de um agente é definida pela presença de uma *arquitetura* – que denota o conjunto de sensores e atuadores que o agente terá para interagir com o ambiente – e pela presença de *programa de agente* – uma implementação do algoritmo (*função agente*) que mapeia as percepções que o agente tem do ambiente à respectivas reações do agente sobre esse mesmo ambiente. É interessante que a arquitetura e o programa definidos sejam compatíveis, para evitar que o programa exija algum recurso que a arquitetura não foi projetada para oferecer (por exemplo, um programa exigir mobilidade de uma arquitetura desenvolvida para um agente sem essa característica). A Figura 2-1 apresenta um modelo conceitual de um agente e o seu relacionamento com o ambiente externo.

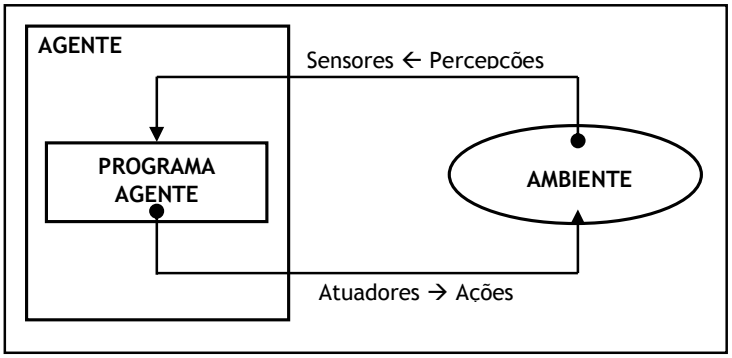

Figura 2-1 Modelo conceitual de um agente e seu relacionamento com o ambiente.

2.2.2 Características dos SMAs que podem beneficiar os SDIs

As características inerentes dos SMAs possibilitam a solução dos problemas levantados na seção 2.1.3:

- A dificuldade de configuração do sistema é resolvida visto que os agentes podem ser iniciados e finalizados sem trazer alterações para o restante do sistema. Desta forma, é possível adicionar novos agentes ao sistema, aumentando o poder de detecção do mesmo, sem ser necessário que todo o sistema seja reiniciado;
- Uma outra vantagem que os agentes trazem para os SDIs é devido ao fato de cada agente ter a sua própria linha de execução. Caso algum problema ocorra durante a execução das atividades de um determinado agente, apenas esse agente será finalizado;
- Como todo o processamento do SDI é distribuído entre os diversos agentes do sistema espalhados pela a rede (ou sub-rede) que está sendo coberta, o monitor não fica sobrecarregado com todo o processamento, o que permite uma maior escalabilidade do sistema;
- Outro benefício trazido pelos SMAs é que não existe um analisador central de dados, que caracteriza um ponto de vulnerabilidade do sistema;
- O fato de os dados serem analisados no mesmo *host* onde são coletados elimina a possibilidade de que os dados coletados sejam forjados antes da análise, por não haver necessidade de envio desses dados para o monitor.

Capítulo *3*

Frameworks e Padrões de Projetos

Ao longo dos anos a engenharia de software vem ganhando o foco da comunidade científica, com propostas para o desenvolvimento de software de maior qualidade a um custo mais baixo. Para tanto, diversos trabalhos têm sido propostos na área de reuso de software, onde são identificadas partes do código que podem ser reutilizadas, evitando duplicação de código.

O desenvolvimento de *frameworks* se apresenta como uma solução para promover a redução de custos e tempo de desenvolvimento de *software.* A utilização de padrões de projetos, largamente adotada em projetos de *frameworks*, garante um mínimo de qualidade necessário para promover o reuso de código, a expansibilidade, estensibilidade e manutenibilidade. As seções seguintes apresentam os conceitos básicos sobre *frameworks* e padrões de projetos, necessários para o entendimento deste trabalho.

3.1 Frameworks

Diversas definições para *framework* foram propostas ao longo do tempo na comunidade científica. *Sommerville* [29] define que frameworks são subsistemas desenvolvidos com base numa coleção de classes concretas e abstratas ligadas através de uma interface. Em um outro momento, *Johnson* [18] define: "Um *framework* é um conjunto de objetos que colaboram para realizar um conjunto de responsabilidades para uma aplicação ou um subsistema de domínio específico". Uma definição mais abrangente foi proposta por *Mattsson* [22] será adotada neste trabalho; segundo *Mattsson*, um *framework* pode ser definido como: "Uma arquitetura desenvolvida para maximizar o reuso, representada por uma coleção de classes abstratas e concretas, encapsulando um comportamento potencial para as subclasses (das classes abstratas)".

Com base na definição assumida para este trabalho, pode-se perceber o grande benefício que o *framework* traz para o desenvolvimento de *softwares* – o reuso de componentes que resulta na redução de custos de projeto (uma das motivações deste trabalho). A Figura 3-1 apresenta um diagrama comparativo entre os modelos de desenvolvimento (de *software)* tradicional e baseado em *frameworks.* No desenvolvimento tradicional o esforço de desenvolvimento (análise e projeto) não são reusados para novas aplicação, o que resulta no mesmo custo de desenvolvimento para a segunda aplicação; Já no desenvolvimento baseado em *frameworks,* pode-se perceber que os esforços de análise e projeto só são feitos na primeira aplicação (quando o *framework* é desenvolvido); por reusarem a estrutura do *framework* as aplicações seguintes podem ser

desenvolvidas com um custo mais baixo. As seções seguintes apresentam as características e componentes relacionados ao desenvolvimento de aplicações baseadas em *frameworks.*

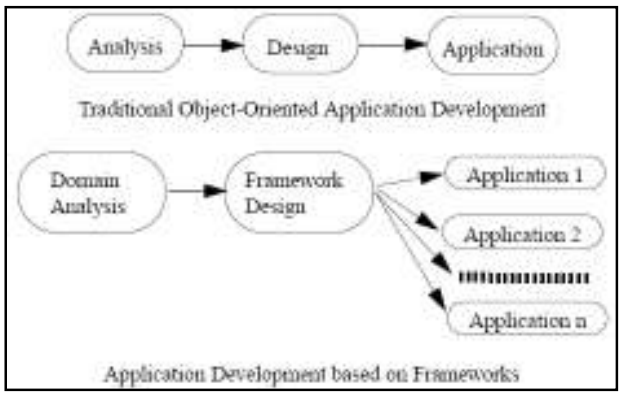

Figura 3-1 Desenvolvimento (de aplicações) tradicional *versus* desenvolvimento baseado em *frameworks*. [22]

De acordo com suas características, os *frameworks* podem ser categorizados de forma diferente. *Fayad* e *Schimidt* [11] propõem as seguintes classificações:

- *Frameworks horizontais:* constituem os *frameworks* de propósito geral, ou seja, que apresentam uma infra-estrutura comum que pode ser utilizada para diversos domínios de problema (GUI, persistência, comunicação, etc.);
- *Frameworks verticais:* constituem os *frameworks* destinados a aplicações em um domínio de problema específico (*frameworks* para SDIs, por exemplo). Como esse tipo de *framework* precisa implementar as classes que modelam o domínio em questão, eles se caracterizam por serem complexos e difíceis de usar. Para superar essas dificuldades, pode-se fazer uso de padrões de projeto no intuito de simplificar e componentizar os subsistemas do *framework* (os padrões de projeto serão explicados mais adiante, na seção 3.2);

3.1.1 Características

Algumas características podem ser observadas na utilização dos *frameworks* para o desenvolvimento de aplicações; dentre elas destacam-se:

- *Modularidade:* O *framework* encapsula os detalhes de implementação disponibilizando interfaces estáveis que dão acesso a suas funcionalidades. Desta forma, as alterações de projeto ou implementação tornam-se pontuais, o que aumenta a manutenibilidade do *framework*, favorecendo a qualidade de software;
- *Reusabilidade:* Com a utilização do *framework*, é possível compartilhar o conhecimento de outras pessoas através da sua utilização. Com o reuso de código, a *expertise* utilizada para o desenvolvimento do *framework* pode ser inteiramente aproveitada pelas aplicações que o utilizam – motivo pelo qual existe a redução de custos no projeto;
- *Extensibilidade:* O *framework* oferece mecanismos que fornece à aplicação que o utiliza a facilidade de implementação de novas funcionalidades; Esses mecanismos, chamados *hot-spots*, serão abordados na próxima seção;
- *Inversão de Controle:* É uma característica presente na arquitetura do *framework* em tempo de execução que permite que o próprio *framework* (ao invés da aplicação) decida que método será chamado na ocorrência de um determinado evento.

3.1.2 Estrutura

O desenvolvimento de um *framework* é similar ao desenvolvimento tradicional de aplicações Orientadas a Objeto (OO) com algumas características adicionais que define a sua estrutura. *Mattsson* [22] define essas características como elementos de projeto e lista os seguintes elementos:

- *Classes Abstratas*: São classes caracterizadas por não poderem ser instanciadas e normalmente possuírem métodos não implementados; A utilização de classes abstratas é essencial para possibilitar o desenvolvimento dos *hot-spots* – aspectos variáveis de um domínio de aplicação [25];
- *Padrões de Projetos*: Representam descrições abstratas de soluções para os problemas conhecidos de projeto. Os padrões de projetos serão explicados com mais detalhes na próxima seção;
- *Ligação Dinâmica*: Os *frameworks* diferem das bibliotecas de funções tradicionais pelo fato de que numa biblioteca tradicional as chamadas são feitas apenas no sentido aplicação-biblioteca. Nos *frameworks* essas chamadas podem ser feitas nos dois sentidos (inversão de controle). Isso só é possível graças á ligação dinâmica;

3.1.3 Documentação e Uso do *Framework*

Uma das partes mais importantes de um *framework* é a sua documentação. Visto que o *framework* realiza diversas atividades (muitas delas de certo grau de complexidade) é extremamente necessário que ele seja bem documentado, de forma a apresentar para seus possíveis usuários uma descrição detalhada do *framework,* sua utilização e em alguns casos, detalhes de funcionamento. Em alguns casos também é interessante prover a API – *Application Programmer Interface* do *framework,* que apresenta os detalhes de implementação em nível de classes e métodos, necessários para o entendimento mais aprofundado do *framework. Mattsson* [22] lista alguns possíveis usuários do *framework* apresentando as suas necessidades na documentação desse *framework*:

- *Engenheiros de software que precisam decidir que framework usar*: Estão interessados numa breve descrição das capacidades do *framework.* A documentação deve explicar as características mais importantes através de exemplos;
- *Engenheiros de software que decidiram usar o framework*: Estão interessados em como esse *framework* pode ser utilizado. Uma boa forma de apresentar esse tipo de informação é através de *cookbooks* – tipo de documentação que descreve de forma objetiva como usar o *framework* através de exemplos [22];
- *Engenheiros de software que querem adicionar funcionalidades ao framework*: Estão interessados numa descrição mais aprofundada do *framework.* A documentação deve explicar os algoritmos abstratos usados e os modelos de colaboração do *framework.*

A maioria dos usuários de um *framework* não está interessada em como ele funciona, e sim procuram por uma documentação que mostre como utilizá-lo. A abordagem mais interessante para esses casos é apresentar um *cookbook* que descreva os detalhes necessários para o entendimento do *framework.*

Segundo *Sommerville* [29] a aplicação do *framework* envolve duas tarefas: adicionar classes concretas que herdam o comportamento definido nas classes abstratas do *framework*; implementar os métodos que são chamados pelo *framework* em resposta a determinados eventos que são reconhecidos por ele. De forma mais generalizada, a utilização do *framework* pode ser

definida da seguinte forma: Entende-se por uso do *framework* a sua instanciação em uma nova aplicação que faz reuso de suas funcionalidades (do *framework*), implementando os *hot-spots* por ele definido para um domínio de aplicação. O MAFIDS apresenta um conjunto bem definido de *hot-spots* que devem ser implementados com o intuito de adicionar ao sistema uma característica específica de um domínio de aplicação. A utilização do MAFIDS será discutida em mais detalhes no Capítulo 4.

3.2 Padrões de Projetos

Buschmann [3] definiu padrões de projeto como um mapeamento de um problema de desenvolvimento recorrente que surge numa situação específica e propõe uma solução para esse problema. Segundo *Gamma* [13], os padrões identificam e especificam abstrações que vão além de simples classes e objetos ou componentes. Em termos gerais, pode-se definir os padrões de projeto como mecanismos que auxiliam na solução dos problemas conhecidos de implementação. O uso de padrões de projetos facilita o desenvolvimento de projetos OO que possam ser reusados posteriormente. No desenvolvimento de *frameworks* essa facilidade é de fundamental importância, visto que eles são desenvolvidos essencialmente para serem reusados. As seções a seguir apresentam as definições propostas por *Gamma* [13] acerca dos padrões de projetos, separados por suas categorias. Apenas serão apresentadas as categorias que contém os padrões utilizados no desenvolvimento do MAFIDS.

3.2.1 Padrões de Criação

Os padrões de projeto desta categoria se propõem a abstrair o processo de criação de objetos, tornando o sistema independente desse processo. A seguir estão listados os padrões desta categoria que foram aplicados no desenvolvimento do MAFIDS:

3.2.1.1 Abstract Factory

É caracterizado por definir o comportamento específico para uma ou mais classes correlacionadas através de uma *interface*. Essa *interface* deve definir apenas *o quê* uma determinada classe deve fazer; o *como* fica a cargo das classes concretas que implementam essa interface. A Figura 3-2 apresenta um diagrama UML – *Universal Markup Language* – de classes, típico de uma implementação do *Abstract Factory*.

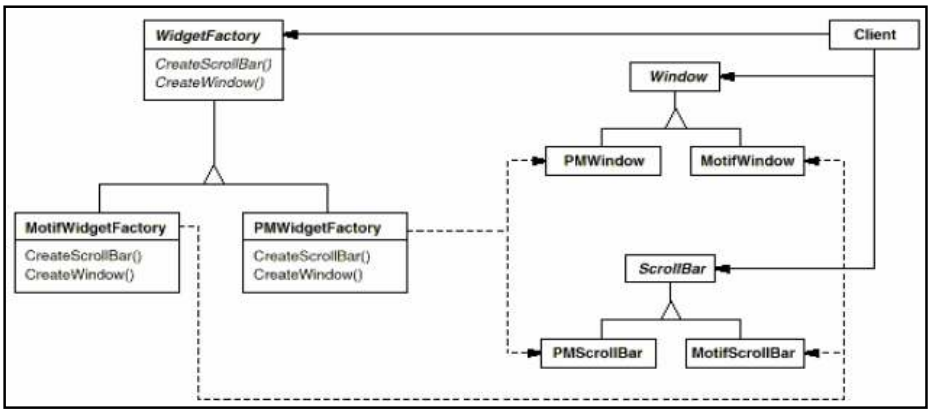

Figura 3-2 Diagrama de classes de uma implementação típica do *Abstract Factory*. [13]

Segundo o diagrama, a classe abstrata `WidgetFactory` define métodos para a criação de elementos da GUI. Existe uma classe concreta para cada padrão de *look and feel* (aparência das telas) existente. Para o cliente não importa que classe concreta foi instanciada, o que importa é que ela é capaz de realizar as operações definidas por `WidgetFactory`. De acordo com o *look and feel* selecionado, a operação terá um comportamento diferente.

Algumas situações sugerem claramente o uso do *Abstract Factory* para promover o desacoplamento e a consistência entre os objetos que implementam a *interface* definida. Dentre elas *Gamma* [13] cita:

- Quando se quer tornar o sistema independente de como os objetos são criados, compostos e representados;
- Quando se quer configurar o sistema de acordo com um ou mais objetos da mesma família, ou seja, que implementam essa mesma *interface*;
- Quando se quer desenvolver uma biblioteca de componentes deseja-se revelar apenas a *interface* desses componentes e não suas implementações.

3.2.1.2 Factory Method

O padrão de projeto *Factory Method* é caracterizado por definir uma interface para a criação de objetos, no entanto, a decisão de que classe será instanciada fica a cargo da subclasse. Segundo *Gamma* [13], esse padrão também é conhecido por *Construtor Virtual*. Este padrão é bastante utilizado em *frameworks* – a exemplo do MAFIDS – que não podem antecipar (por só conhecer as classes abstratas que definem os *hot-spots*) que classe será instanciada em tempo de execução. A Figura 3-3 apresenta um diagrama de classes típico de uma implementação do *Factory Method.*

Em termos gerais, o *Factory Method* pode ser definido por um único método parametrizado – `createInstance(param)`, por exemplo – que utiliza o parâmetro informado para determinar que subclasse será instanciada; ou pode ser definido pela definição de vários métodos para a criação de objetos específicos, normalmente sem parâmetros.

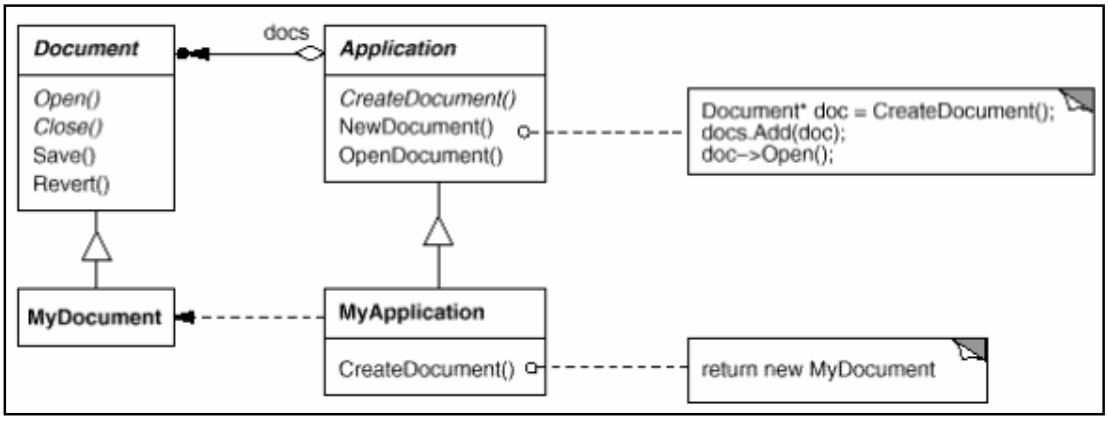

Figura 3-3 Diagrama de classes de uma implementação típica do *Factory Method*. [13]

Segundo o diagrama, a subclasse `MyApplication` redefine o método abstrato `CreateDocument()` definido em `Application`, eliminando a necessidade de `Application` conhecer o tipo de documento a ser criado. Este padrão é muito útil para generalizar o conhecimento do *framework* tornando-o mais abrangente.

3.2.1.3 Singleton

Este padrão de projeto garante que só existirá uma única instancia de uma determinada classe em todo o sistema, e que essa classe deverá provê um ponto global de acesso a instancia do *Singleton*. A Figura 3-4 apresenta o diagrama de classes típico do *Singleton* e a Figura 3-5 apresenta uma estrutura de código (JAVA) típica de um *Singleton*.

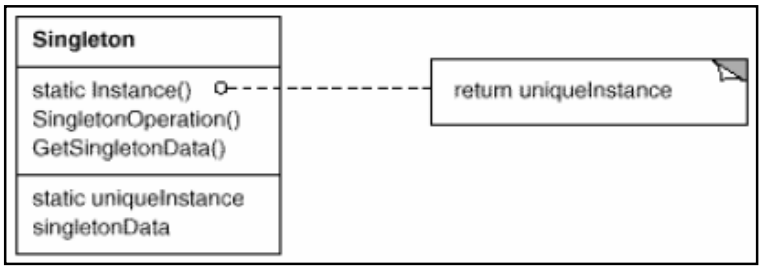

Figura 3-4 Diagrama de classes de uma implementação típica do *Singleton*. [13]

Na primeira vez que o método `Instance()` é chamado, uma nova instância desse *Singleton* é criada; a partir de então, as chamadas subseqüentes retornarão essa mesma (única) instância.

```
public class MySingleton{

    // MySingleton instance
    private static MySingleton INSTANCE = new MySingleton();

    // Others attributes…

    /* Private constructor to avoid multiple
       class creation. */
    private MySingleton(){
      // some initialization …
    }

    /* The global access point */
    public static MySingleton getInstance(){
        return INSTANCE;
    }
}
```

Figura 3-5 Estrutura de código (JAVA) típica de um *Singleton*.

3.2.2 Padrões Estruturais

Os padrões de projeto desta categoria definem como classes e objetos se relacionam para formar estruturas mais complexas, e são particularmente úteis quando se deseja que classes completamente independente trabalhem juntas.

3.2.2.1 Adapter

É caracterizado por definir uma *interface* que permita que duas classes, que antes não podiam trabalhar juntas por conta de incompatibilidades entre as suas *interfaces*, possam trabalhar juntas. A Figura 3-6 apresenta o diagrama de classes típico da implementação de um *Adapter*.

O padrão de projeto *Adapter* se apresenta bastante útil quando se quer criar um relacionamento entre classes onde apenas as classes de controle acessam as classes de GUI – *General User Interface* – e que o contrário não aconteça. Em algum momento, as classes de GUI precisarão acessar alguma classe de controle; nesse momento entra em ação o *Adapter*, que define uma interface com o comportamento (da classe de controle) esperado pela GUI. Essa interface é implementada pela classe de controle e, desta forma, as classes de GUI não precisam acessar as classes do pacote de controle diretamente.

Segundo o diagrama, através do adaptador `TextShape` as operações `BoundingBox()` e `CreateManipulator()` – definidas em `Shape` – puderam ser adaptadas para retornar `GetExtent()` e `TextManipulador`, respectivamente. Apesar de serem operações incompatíveis, o adaptador consegue relacionar essas duas classes.

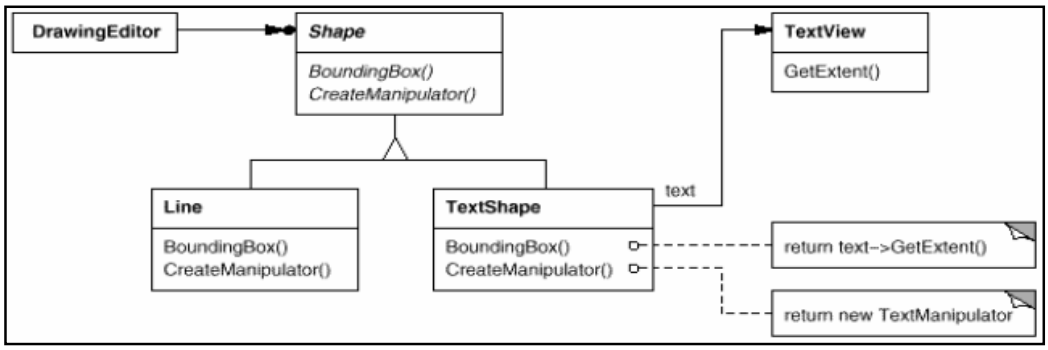

Figura 3-6 Diagrama de classes de uma implementação típica do *Adapter*. [13]

3.2.2.2 Facade

Este padrão é caracterizado pela presença de uma *interface* única para o sistema, facilitando o acesso aos subsistemas presentes. Desta forma é possível ocultar detalhes desnecessários para o cliente do sistema, tornando-o mais simples e com maior usabilidade. A Figura 3-7 apresenta o diagrama de classes típico da implementação de um *Facade.*

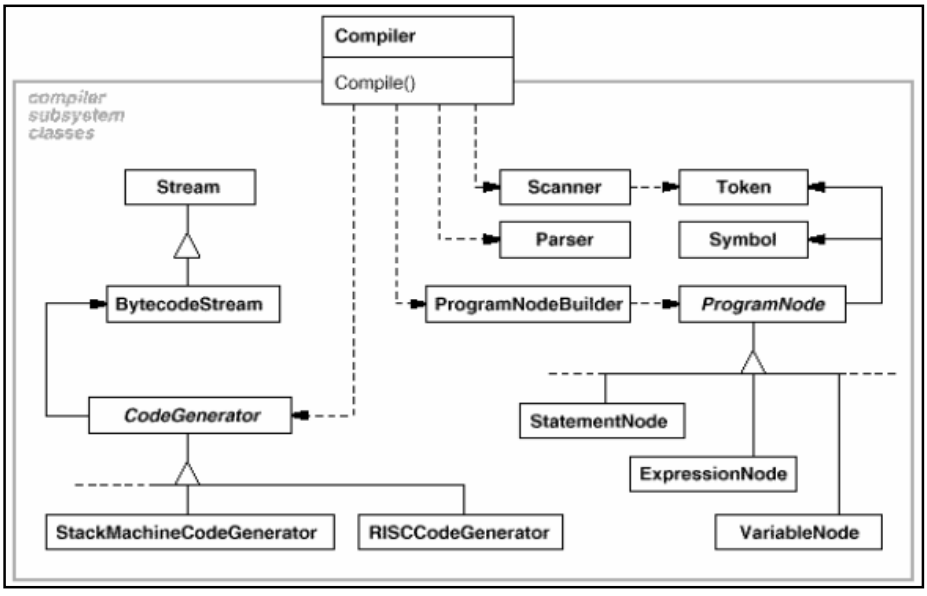

Figura 3-7 Diagrama de classes de uma implementação típica do *Facade*. [13]

No diagrama apresentado, a classe `Compiler` faz o papel de *interface* (ou fachada) para o subsistema de compilação. Desta forma é possível proteger as classes componentes desse subsistema ocultando os detalhes de implementação para os usuários do subsistema de compilação.

3.2.3 Padrões Comportamentais

Compreendem os padrões de projeto relacionados com os algoritmos (como os objetos realizam suas respectivas atividades) e com a delegação de responsabilidades entre os objetos. Os *Behavioral Patterns* não descrevem apenas padrões de classes e objeto, mas definem também padrões de comportamento entre eles.

3.2.3.1 Strategy

O padrão *Strategy* é caracterizado por encapsular um determinado algoritmo em um objeto, de forma que uma futura mudança desse algoritmo se resuma à utilização de um novo objeto (que deve encapsular o novo algoritmo). A Figura 3-8 apresenta o diagrama de classes típico da implementação de um *Strategy.*

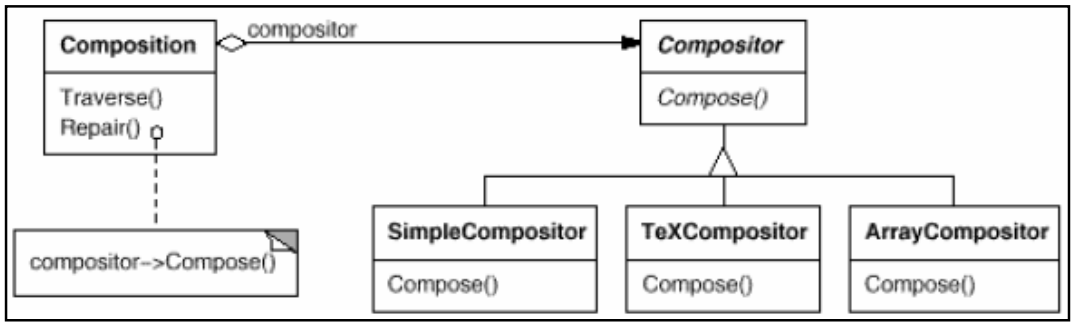

Figura 3-8 Diagrama de classes de uma implementação típica do *Strategy*. [13]

Segundo o diagrama, a classe `Composition` tem várias instancias de `Compositor`, que define o método abstrato `Compose()`. Imagine que existam várias formas de realizar esse compose. Se a classe `Compositor` fornecesse uma implementação concreta para o método `Compose()`, seria necessário que a classe `Composition` conhecesse todas as implementações possíveis para `Compositor`. No entanto, a aplicação do padrão de projeto *Strategy* permitiu que a classe `Composition` conhecesse apenas a classe abstrata `Compositor`. Cada subclasse de `Compositor` implementa o método `Compose()` de uma forma específica e essa implementação é chamada por `Composition` através de `Compositor`.

3.2.3.2 Template Method

Este padrão de projeto se caracteriza pela definição de algoritmos onde um ou mais passos são delegados para as subclasses da classe que implementa esses métodos. Desta forma é possível manter um padrão de execução de um determinado método modificando o seu comportamento apenas nos pontos específicos que são delegados às subclasses, o que permite um reuso considerável de código. A Figura 3-9 apresenta o diagrama de classes típico da implementação de um *Template Method.*

Considere que a classe abstrata Application define três métodos abstratos: `CanOpenDocument()`, `DoCreateDocument()` e `AboutToOpenDocument()` que são

chamados dentro do corpo do método `OpenDocument()`. A subclasse `MyApplication` implementa os três métodos abstrados definindo o comportamento desejado para esses pontos específicos do método `OpenDocument()`. Cada subclasse de `Application` pode implementar esses métodos de forma diferente, fazendo com que o mesmo método `OpenDocument()` tenha comportamentos diferentes em alguns pontos.

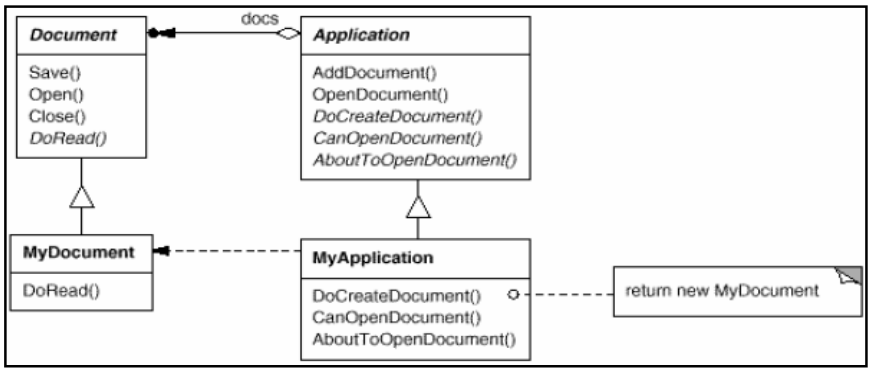

Figura 3-9 Diagrama de classes de uma implementação típica do *Template Method*. [13]

Capítulo 4

MAFIDS – Um *Framework* para SDIs baseados em SMAs

Diante dos problemas no cenário mundial de TI, apontados pela pesquisa realizada pela *PwC* em parceria com a *CIO* e *CSO Magazine* [9], abordados no Capítulo 1, e tendo em vista que os SDI se apresentam como uma das ferramentas mais eficazes na detecção de falhas de segurança (segundo a mesma pesquisa [9]), decidiu-se desenvolver neste projeto um mecanismo que facilitasse a adoção desse tipo de ferramenta no mercado de TI.

As seções seguintes apresentam a proposta do MAFIDS – *Multi-Agent Framework for Intrusion Detection Systems* – um *framework* que vem superar dois grandes desafios no desenvolvimento de SDIs (seu domínio de aplicação): reduzir os custos de desenvolvimento deste tipo de ferramenta, através do reuso e do desenvolvimento baseado em componentes; e superar as limitações inerentes da arquitetura dos SDIs tradicionais, de núcleo monolítico, tais como: dificuldade de reconfiguração, escalabilidade, expansibilidade, etc. (os Capítulos 2 e 3 abordam com mais detalhes as características dos *frameworks* e SDIs).

Devido às facilidades oferecidas pela linguagem JAVA – desalocação automática de memória, portabilidade, facilidade para a criação de sistemas distribuídos, etc. – decidiu-se implementar este projeto utilizando esta linguagem, inclusive explorando as novas características da versão 5.0 (tais como *generics, unboxing, auto-boxing* e *enums* [28]), que trouxe muitas facilidades na manipulação de coleções de agentes, *hosts* e programas (para mais informações sobre a linguagem JAVA consulte [15][16] [28]).

4.1 Considerações Iniciais

Em busca de soluções para as limitações conhecidas na arquitetura tradicional de SDIs, outras abordagens vêm sendo desenvolvidas ao longo do tempo, tais como os SDIs distribuídos [2]. As vantagens do desenvolvimento de SDIs utilizando a tecnologia de agentes inteligentes, também chamados "SDIs não-monolítico" – devido à divisão do grande bloco responsável por toda a operação de detecção de intrusão em vários agentes especializados – têm sido alvo de muitos estudos na comunidade científica.

O CERIAS – *Center for Educational and Research in Information Assurance and Security* [4] – apresenta uma implementação para um ambiente de detecção de intrusão utilizando a abordagem de agentes autônomos, que foi denominado AAFID – *Autonomous Agents For*

Intrusion Detection. A implementação foi baseada no ambiente proposto por *Crosbie* e *Spafford* [6][7], que definem algumas entidades (*transceivers,* agentes e monitores) estruturadas hierarquicamente, pretendendo solucionar as limitações conhecidas dos SDIs monolíticos. O projeto foi abandonado em 1999 após a liberação de seu segundo protótipo, o AAFID2, para o domínio público – por ser considerado inviável [4].

Um outro trabalho apresentado por Bernardes [1] também propõe o desenvolvimento de um SDI baseado em agentes autônomos utilizando um modelo multicamadas. Para o desenvolvimento desse trabalho, Bernardes utilizou um *framework* para desenvolvimentos de agentes móveis proposto pelo *IBM Tokyo Research Group* [21]. Nesse modelo, os agentes são especializados numa determinada tarefa e cada agente das camadas inferiores oferecem serviços aos agentes das camadas imediatamente superiores.

As seções seguintes apresentam de forma mais detalhada as etapas de desenvolvimento do MAFIDS, abordando a definição do protocolo de comunicação utilizado e detalhes da arquitetura, explicando os padrões de projeto utilizados.

4.2 Arquitetura Proposta

A arquitetura do MAFIDS foi projetada no intuito de promover o desenvolvimento de sistemas facilmente escaláveis – capaz de lidar com uma grande quantidade de *hosts* – e expansíveis – capazes de aumentar a sua área de atuação (poder de detecção) num domínio de aplicação. Além dessas características, também foram levados em consideração outros fatores que permitam superar as limitações dos SDIs tradicionais (uma das motivações deste trabalho); dentre elas pode-se citar: promover a facilidade de reconfiguração dos sistemas desenvolvidos sobre o *framework* e eliminar a tarefa de análise dos dados de um único módulo, dividindo-as entre os diversos agentes do sistema.

Para promover um maior reuso de códigos, foram aplicados alguns padrões de projeto que garantem a qualidade da arquitetura proposta. Além disso, a aplicação desses padrões possibilitou a implementação de características essenciais para um *framework* como, por exemplo, a inversão de controle (para mais informações sobre as características dos *frameworks,* consulte o Capítulo 3). A Figura 4-1 apresenta de modo conciso o relacionamento entre o MAFIDS e a aplicação construída sobre ele.

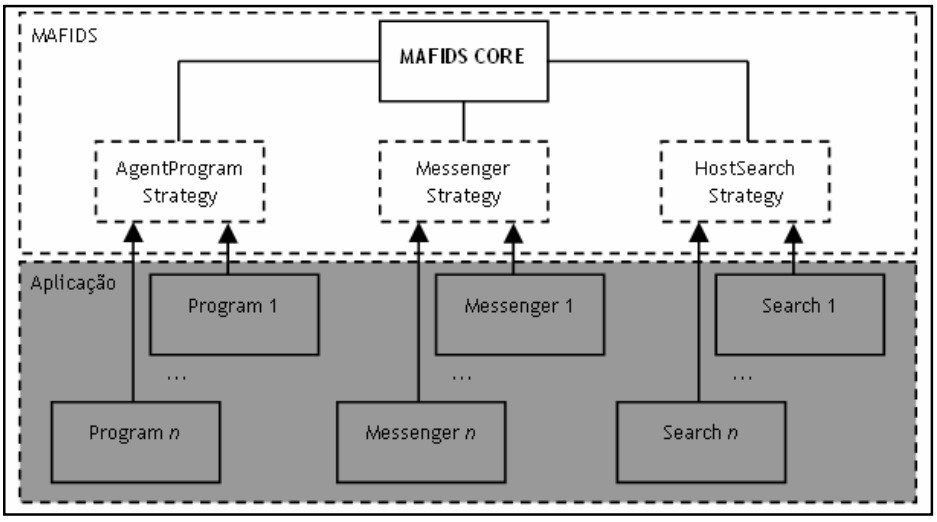

Figura 4-1 Relação entre o MAFIDS e a aplicação construída sobre ele.

Segundo o relacionamento apresentado, o elo entre o *framework* e a aplicação se concentra nas classes abstratas que definem os *hot-spots*. Desta forma, o usuário não precisa conhecer detalhes do *core* do MAFIDS, tendo apenas que implementar subclasses para os *hot-spots* definidos.
As seções seguintes focam no projeto relativo ao MAFIDS, detalhando o funcionamento da arquitetura proposta, apresentando o modelo multicamadas, os componentes da arquitetura, os mecanismos de comunicação utilizados. A Figura 4-2 apresenta o diagrama de classes, que será explicado nas seções seguintes, modelado para o MAFIDS.

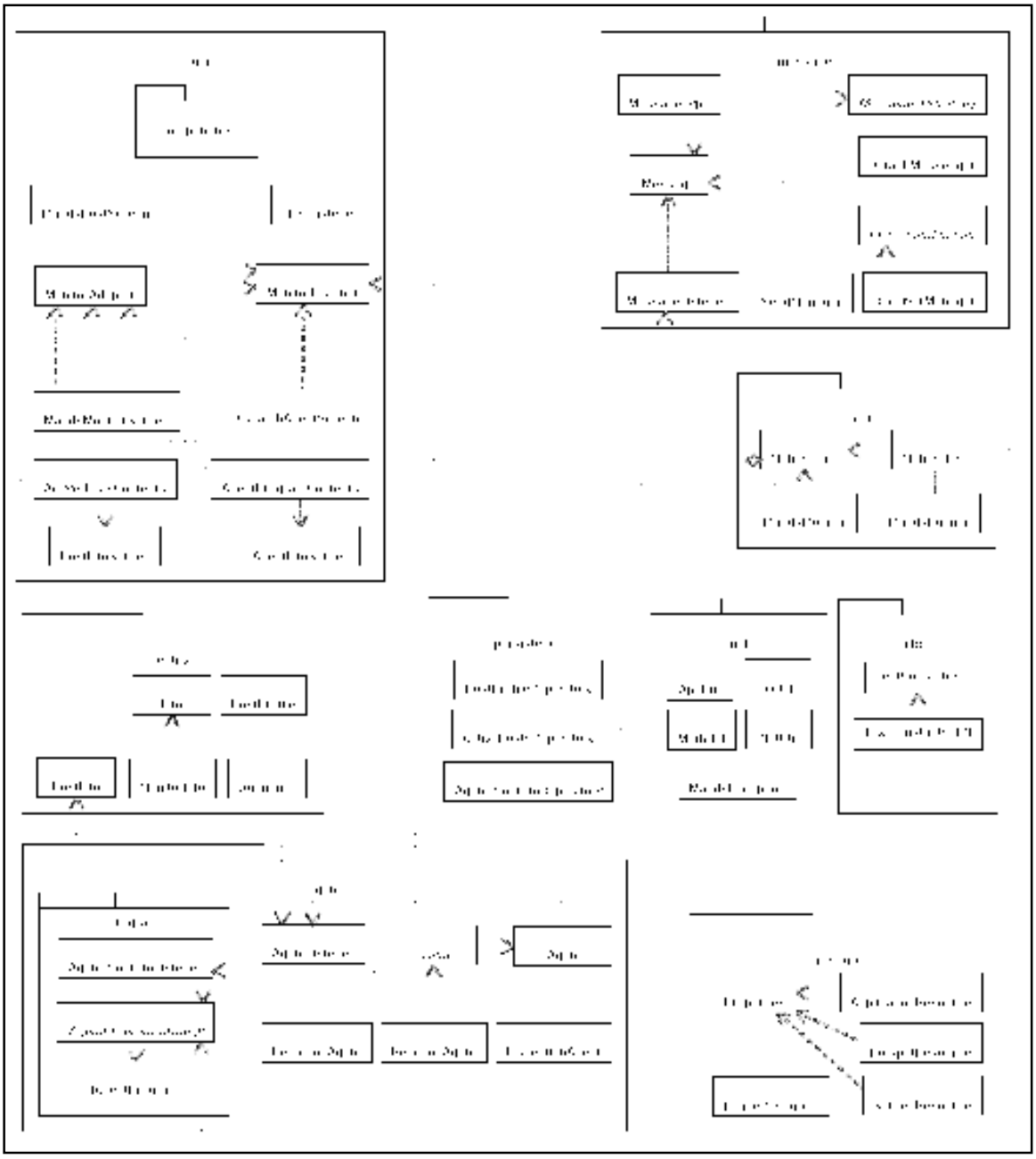

Figura 4-2 Diagrama de classes do MAFIDS.

4.2.1 Modelo multicamadas

Este projeto apresenta uma arquitetura multicamadas, análoga ao modelo de segurança utilizado por policiais militares. Um dos pontos mais fortes do modelo utilizado pelos militares é o policiamento hierárquico; neste modelo encontramos desde os simples policiais de patrulha de ruas, que constituem o nível mais simples de policiamento, até um nível de policiamento mais ostensivo, como a ação da tropa de choque, esquadrão anti-bombas, etc. Enquanto os patrulheiros observam o movimento nas ruas em busca de situações anormais ao dia-a-dia como, por exemplo: correria, indivíduos suspeitos observando carros, gritos, etc., as tropas de elite, que possuem objetivos específicos bem definidos, são especializadas em medidas reativas. A interação entre os patrulheiros e as tropas de elite passa por níveis intermediários de policiamento – rádio patrulha, viaturas, etc. – e é coordenada por um mecanismo de monitoramento que está no topo da cadeia, a central de polícia.

O modelo apresentado define um sistema multicamadas, conforme apresentado na Figura 4-3. Cada camada define um grupo de agentes de comportamentos e características específicas para lidar com uma determinada etapa do problema. A cobertura completa de um ataque é realizada pela implementação dos agentes dos três níveis: *Detecção, Inspeção e Reação.*

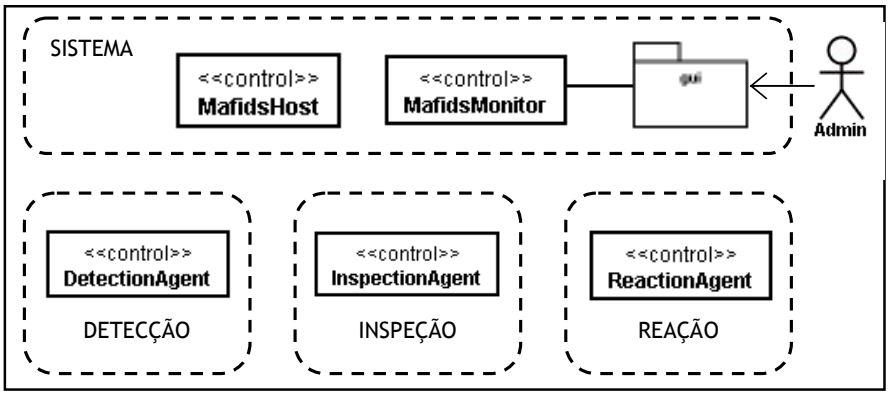

Figura 4-3 Modelo multicamadas proposto no projeto.

As classes `DetectionAgent`, `InspectionAgent` e `ReactionAgent` definem as camadas de detecção, inspeção e reação, respectivamente. A camada de sistema abrange os componentes do *core* do MAFIDS, dentre os quais os seus principais controladores são evidenciados: `MafidsMonitor` e `MafidsHost`. O pacote GUI foi colocado no modelo para enfatizar a interação do administrador com o *framework.*

A exemplo dos modelos em camada conhecidos na área de redes – OSI, TCP/IP, etc. (para saber mais sobre os modelos e protocolos de rede, consulte [20][27][31]) – os agentes das camadas inferiores oferecem serviços aos agentes da camada imediatamente superior, e assim por diante. O comportamento colaborativo que se observa entre os agentes proporciona um maior poder de detecção ao sistema, reduzindo as ocorrências dos falso-positivos (alarmes falsos) e falso-negativos (deixa de detectar uma intrusão real).

O modelo proposto considera que todo tráfego é suspeito até que se prove o contrário. Essa abordagem proporciona maior segurança ao sistema, visto que minimiza a ocorrência de falso-negativos. A Figura 4-4 apresenta a interação entre os componentes de cada camada. As setas indicam o sentido da comunicação entre os componentes relacionados. Como pode ser observado na figura, apesar de os agentes serem enviados para os *hosts,* eles não se reportam ao *host;* qualquer notificação ou solicitação é feita diretamente ao monitor. As seções seguintes

descrevem com mais detalhes as características, comportamentos e estrutura dos agentes de cada camada.

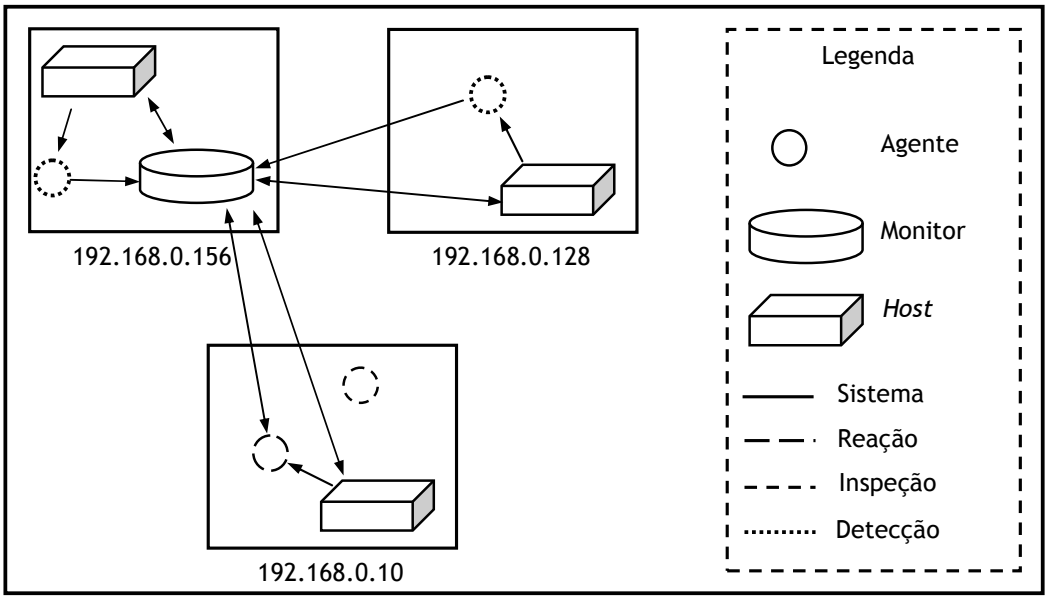

Figura 4-4 Interação entre os componentes de cada camada na rede.

4.2.1.1 Camada de Detecção

Os agentes de detecção são caracterizados por serem agentes autônomos dotados de pouco ou nenhum mecanismo de aprendizagem, responsáveis por efetuar a ronda no sistema em busca de dados específicos que denunciem alguma atividade suspeita. O desenvolvimento de agentes especializados em atividades específicas proporciona a codificação de agentes mais simples, permitindo uma boa manutenibilidade.

Esta camada determina o primeiro nível de monitoramento do sistema. Fazendo a analogia ao sistema de policiamento militar, os agentes de detecção são comparados aos policiais que trabalham nas ruas, normalmente em duplas. As duplas são posicionadas em pontos estratégicos das ruas e avenidas da cidade e são responsáveis por patrulhar uma determinada região. Normalmente esses policiais procuram por anomalias no comportamento das pessoas, tais como correria, assalto, batidas no trânsito, badernas, etc.; ao detectar alguma anomalia, a dupla pode solicitar o reforço de uma viatura, a qual seguirá para o local do incidente. De forma análoga, os agentes de detecção são espalhados na rede e se responsabilizam por monitorar o comportamento de aspectos específicos do *host.* Ao detectar alguma anomalia, esses solicitam o reforço de agentes mais preparados para lidar com a situação.

4.2.1.2 Camada de Inspeção

Os agentes da camada de inspeção realizam o trabalho mais importante do processo de detecção da intrusão – a análise dos dados coletados pelos agentes de detecção. Além da característica de análise de dados, os agentes de inspeção possuem a habilidade de acionar agentes de características reativas, responsáveis por inibir o ataque em andamento. Os agentes de inspeção, ou simplesmente inspetores, coordenam o trabalho de uma equipe de agentes de detecção, conduzindo a investigação do caso.

As atividades dos agentes inspetores são análogas às atividades desempenhadas pelos inspetores de polícia, visto que coordenam as investigações de um caso, analisando as evidências encontradas em busca de uma solução. Devido à sua característica de análise de dados, esse é o agente mais indicado para se aplicar alguma técnica de Inteligência Artificial (IA) – Sistemas Especialistas, *Fuzzy Logic,* Sistemas baseados em Regras [26].

4.2.1.3 Camada de Reação

Os agentes da camada de reação têm como característica principal o comportamento reativo, ou seja, são especializados na realização de uma medida reativa específica como, por exemplo: notificar o administrador, finalizar um processo, finalizar um serviço, cancelar uma conexão, etc. Outra característica dos agentes dessa camada é que eles só são acionados quando é comprovada a existência de um ataque na rede.

A analogia com o modelo de policiamento militar pode ser feita com as tropas de elite, o esquadrão anti-bombas (para ameaças de bombas), tropas de choque (para tumultos generalizados), esquadrão anti-seqüestro, etc., que são equipes especializadas em atividades específicas.

4.2.1.4 Camada de Sistema

Camada de serviços que está localizada no topo da hierarquia do modelo, onde rodam o monitor e os *hosts (*servidores contextos) do sistema. Essa camada constitui a central de monitoramento das atividades dos agentes. No modelo de policiamento militar, o monitor pode ser comparado às centrais de operação da polícia. Ao receber uma chamada (devido a algum problema identificado pelos policiais de patrulha) a central de polícia registra a nova ocorrência e envia uma viatura para averiguar a chamada. Enquanto a ocorrência não é finalizada, a central mantém contato com as equipes enviadas – patrulheiros e viaturas – para acompanhar o andamento das atividades e, caso seja necessário, estar pronta para acionar as equipes especializadas para resolver problemas mais sérios (tropa de choque, esquadrão anti-bombas, etc.).

De forma análoga, ao receber uma indicação de intrusão o monitor envia o agente de inspeção mais indicado para conduzir as investigações do possível ataque e instancia um novo quadro de intrusão. Enquanto esse quadro de intrusão não é finalizado (é configurado o ataque ou um falso-positivo), o monitor mantém contato com o agente de inspeção para acompanhar o andamento das investigações.

4.2.2 Componentes do Framework

Para facilitar o entendimento dos conceitos apresentados nesta seção algumas simplificações foram criadas. Como cada nome de pacote é único em todo o projeto, os pacotes serão referenciados apenas pelos seus nomes (sem levar em consideração a hierarquia dos pacotes); desta forma, o pacote `com.mafids.control.agent.program`, por exemplo, será referenciado apenas por `program`.

4.2.2.1 mafids.control.agent.program

O pacote `program` define um dos *hot-spots* do MAFIDS, permitindo que o usuário do *framework* defina seus próprios *programas agentes.* A Figura 4-5 apresenta o diagrama de classes desse pacote.

Os programas agentes implementam a *função agente* que define o comportamento que um agente deverá apresentar. A Figura 2-1 apresentou o modelo conceitual de um agente, incluindo o

programa agente. Essa definição da estrutura do agente, apresentada por *Russel* e *Norving* [26] segue o mesmo principio definido pelo *Strategy Pattern,* permitindo que a estrutura do agente seja completamente reutilizada para a criação de novos agentes. No MAFIDS os programas agentes são representados pela classe `AgentProgramStrategy` – o nome faz alusão ao padrão de projeto aplicado.

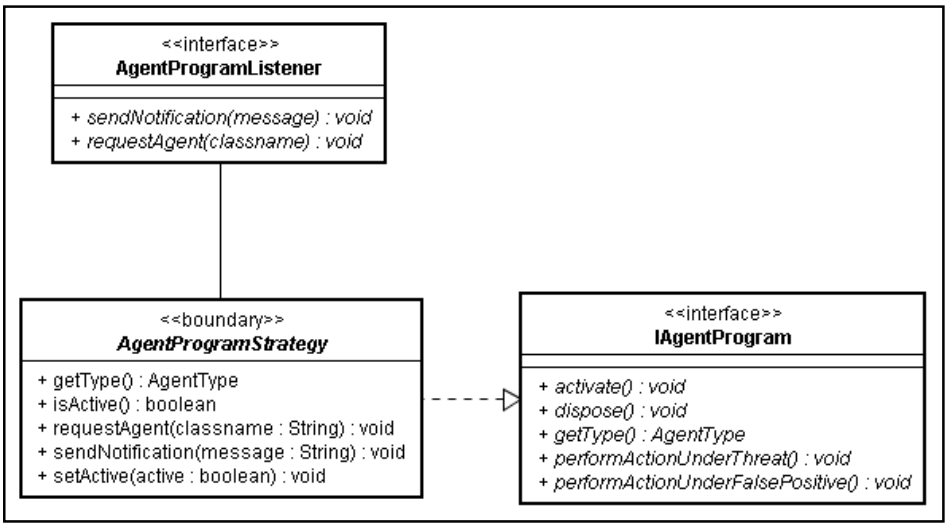

Figura 4-5 Diagrama de classes do pacote `program`.

Devido à grande flexibilidade que esse padrão proporciona para a definição de novos programas, resolveu-se definir esse componente como um *hot-spot* do *framework.* Alguns detalhes da classe e interfaces foram suprimidos para facilitar o entendimento dos relacionamentos entre elas. Algumas considerações sobre os componentes do pacote `program`:

IAgentProgram

É a *interface* que define o comportamento necessário para o programa agente. Esse componente foi criado como resultado direto da aplicação do padrão de projeto *Abstract Factory* e *Adapter* (relativos à criação dos objetos e à sua estrutura, respectivamente). Para aumentar a coesão do pacote, essa *interface* está visível apenas para os componentes desse mesmo pacote.

AgentProgramStrategy

É o principal componente deste pacote, pois modela o programa agente propriamente dito. O `AgentProgramStrategy` é uma classe abstrata que implementa alguns métodos da interface IAgentProgram. Devido às estratégias do projeto de definir o programa agente como um *hot-spot,* permitindo que a implementação da *função agente* fosse delegada aos usuários do MAFIDS, alguns métodos permaneceram abstratos – sem implementação; são eles:

- `activate()`: chamado quando o agente é ativado pelo *host.* Esse método deve conter a implementação das atividades realizadas pelo programa, ou seja, a implementação da *função agente*;

- `dispose()`: chamado quando o agente é finalizado pelo *host*, normalmente devido a algum problema durante sua execução. Utilizado para liberação dos recursos alocados pelo programa para a sua execução tais como: *arrays, streams, threads,* etc.;

- `performActionUnderThreat()`: chamado quando é identificado que o *host* está sob ataque. Nesse método devem ser implementadas as medidas reativas que o agente deve tomar (quando for aplicável). Nas situações em que se quer definir um programa para um agente que não tem características reativas, esse método deve ser implementado com o corpo vazio;

- `performActionUnderFalsePositive()`: chamado quando é verificado que o ataque em andamento na verdade foi um falso-positivo – um alarme falso. Nesse método são implementadas (quando necessárias) operações de *rollback*, de forma a alterar o estado do perfil do *host* definindo que o fato ocorrido não é, de fato, uma ameaça.

Uma outra característica interessante sobre os programas agentes é que eles carregam a informação sobre o tipo de agente para o qual esse programa foi desenvolvido. No momento da instanciação do agente, essa informação é delegada para a classe `Agent`, que utilizando um *Factory Method,* instancia a estrutura de agente correspondente ao tipo informado pelo programa. Essa abordagem foi utilizada para garantir a conformidade entre *arquitetura* e *programa,* proposta por *Russel* e *Norving* [26] (para mais informações a respeito da estrutura dos agentes, consulte o Capitulo 2).

AgentProgramListener

A *interface* `AgentProgramListener` é o resultado da aplicação do padrão de projeto *Adapter* para disponibilizar na classe `AgentProgramStrategy` serviços que de outra forma não seriam possíveis sem que a coesão e o baixo acoplamento do MAFIDS fossem comprometidos. Ela define dois métodos, que são implementados pela classe `Agent`, e oferecidos como métodos utilitários (que de alguma forma facilitam o desenvolvimento de métodos mais complexos) na classe `AgentProgramStrategy`. São eles:

- `sendNotification()`: Método que envia uma notificação de ataque para o monitor. A notificação é apresentada ao administrador e retorna a opinião do administrador acerca do referido ataque (se é realmente um ataque, ou se é um falso-positivo). Essa informação é retornada para o agente que enviou a notificação para que ele possa considerar a opinião do administrador na análise do ataque;

- `requestAgent()`: Método criado para encapsular o processo de requisição de agentes, oferecendo ao usuário do MAFIDS um mecanismo simplificado de requisição de agentes, bastando que o usuário informe o nome do agente que se deseja invocar.

4.2.2.2 mafids.control.agent

O pacote `agent` encapsula os componentes necessários para a modelagem da *arquitetura* dos agentes. A seção 2.2.1 apresentou a estrutura básica dos agentes, segundo *Russel* e *Norving* [26]. Conforme o modelo apresentado, a estrutura do agente é composta por uma *arquitetura* e um *programa* – o qual é modelado pelo `AgentProgramStrategy`, e foi explicado na seção anterior. Esta seção dará enfoque na *arquitetura* do agente, que fornece os recursos necessários para que o *programa* atue num dado ambiente, no caso, nos *hosts.* A Figura 4-6 apresenta o diagrama de classes desse pacote.

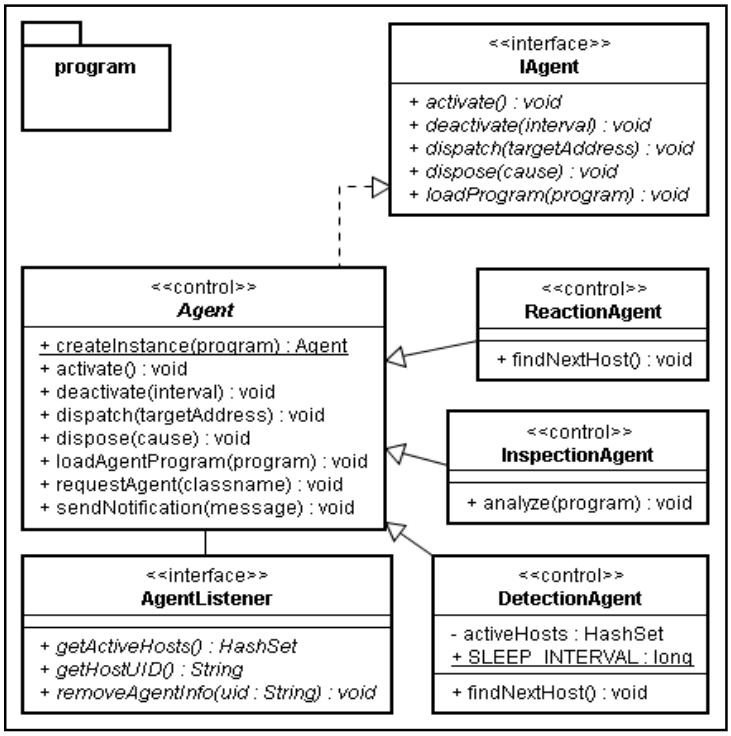

Figura 4-6 Diagrama de classes do pacote `agent`.

Agent e IAgent

Considerada a classe principal desse pacote, a classe abstrata `Agent` modela a arquitetura básica de um agente. No intuito de se encapsular o comportamento básico do agente, foi aplicado o padrão de projeto *Abstract Factory,* para a definição da *interface* `IAgent`. Com a aplicação do padrão de projeto *Factory Method*, foi possível manter a compatibilidade entre a *arquitetura* e o *programa* do agente utilizando o tipo de agente definido no programa e delegado à classe `Agent` através de sua chamada. De acordo com o tipo especificado, o *factory method* determina que tipo de agente deve ser instanciado.

Uma outra característica interessante é que os agentes possuem os mecanismos necessários para se comunicar com outras entidades do sistema (dotadas da mesma habilidade), tais como os *hosts* e os monitores (os componentes de comunicação serão explicados com detalhes mais adiante neste mesmo capítulo).

AgentListener

A *interface* `AgentListener` surgiu da necessidade de comunicação entre o agente e os *hosts* (que serão explicados na próxima seção), sem ferir o desacoplamento entre os pacotes do sistema, como resultado da aplicação do padrão de projeto *Adapter*.

DetectionAgent

Define a estrutura básica para os agentes autônomos do sistema, ou seja, capazes de decidir que ações devem tomar. No MAFIDS, essa autonomia foi implementada na determinação do próximo *host* a ser visitado pelo agente de detecção. O padrão de projeto *Template Method* foi aplicado na classe `Agent`, definindo o método `findNextHost()` que deve ser sobrescrito pelas classes que forem implementar o mecanismo de busca do próximo *host*, que no MAFIDS é representado pelo `DetectionAgent`. A Figura 4-7 apresenta o diagrama de estados típicos de um agente de detecção.

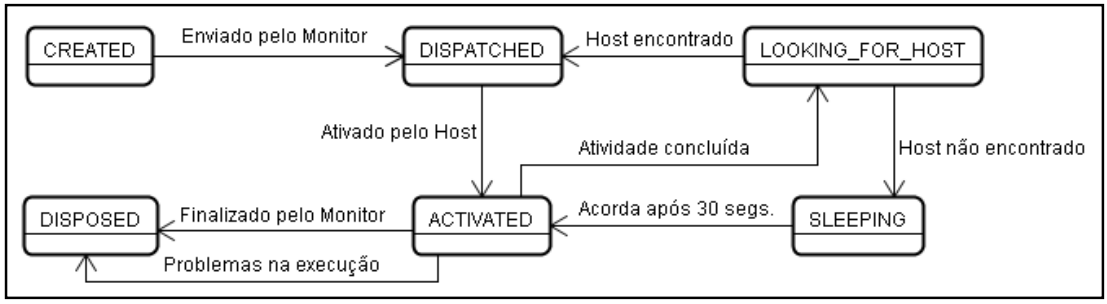

Figura 4-7 Diagrama de estados do `DetectionAgent`.

Após serem criados e despachados pelo monitor (os monitores serão abordados mais adiante neste capítulo), o agente encontra-se no estado DISPATCHED. Ao ser ativado pelo *host* para o qual ele foi despachado, através da chamada ao método `activate()`, o agente passará ao estado ACTIVATED. Ao concluir suas atividades, o agente de detecção vai tentar localizar o próximo *host* a ser visitado passando para o estado LOOKING_FOR_HOST. Ao terminar a busca, caso nenhum outro *host* tenha sido encontrado, o agente de detecção passará ao estado SLEEPING, onde ficará desativado por 30 segundos. Passado esse período, o agente se auto-ativará e realizará suas atividades novamente, voltando ao estado ACTIVATED. Mais uma vez, ao terminar as atividades ele passará ao estado LOOKING_FOR_HOST. Se ao término da busca, um novo *host* for encontrado, o agente será despachado para esse *host* passando ao estado. Caso algum erro ocorra com o agente durante a execução de suas atividades, ou o agente seja finalizado no monitor, ele entrará no estado DISPOSED.

InspectionAgent

Define a estrutura básica para os agentes inteligentes do sistema. Mais uma vez o padrão de projeto *Template Method* foi aplicado na classe `Agent` definindo o método `analyze()`, que deve ser sobrescrito pelas classes que forem implementar algoritmos de IA para efetuar a analise dos dados coletados. No MAFIDS, essa categoria de agentes é representada pelo `InspectionAgent`, que é acionado quando o `DetectionAgent` verifica que existe alguma possível intrusão. A Figura 4-8 apresenta o diagrama de estados típicos de um agente de inspeção.

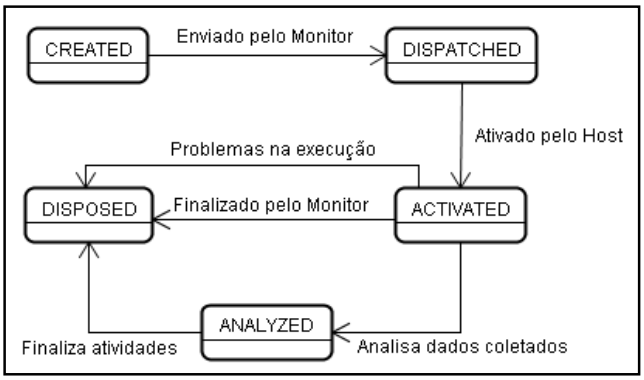

Figura 4-8 Diagrama de estados do `InspectionAgent`.

Após ser criado e despachado pelo monitor para o *host* de destino, o agente de inspeção passará ao estado DISPATCHED. Ao chegar o *host*, o mesmo ativará o agente através do método `activate()` passando-o ao estado ACTIVATED. Após as operações de carregamento dos

dados a serem analisados, o agente efetuará a análise propriamente dita, passando para o estado ANALYZED. Ao terminar suas atividades, o agente de inspeção é finalizado. Este agente possui os mesmo dois fluxos de execução alternativo, já explicados no item anterior, que o leva para o estado DISPOSED.

ReactionAgent

Define a estrutura básica para agentes de reação orientados a metas, ou seja, possuem atividades (reativas) especificas. A Figura 4-9 apresenta o diagrama de estados típicos de um agente de reação.

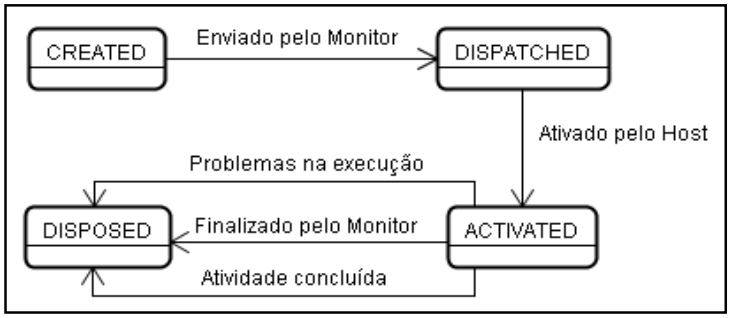

Figura 4-9 Diagrama de estados do `ReactionAgent`.

O agente de reação apresenta um diagrama de estados bastante semelhante aos outros dois já explicados, inclusive com os mesmo fluxos de execução alternativos. A diferença desse tipo de agente para os outros é que após o término de suas atividades, ele sairá do estado ACTIVATED direto para o estado DISPOSED, finalizando assim a sua execução.

4.2.2.3 mafids.control.core

O pacote `core` representa o núcleo de controle do MAFIDS. Neste pacote estão localizados os controladores do monitor e dos *hosts,* além de conter a classe de fachada do *framework* – `MafidsRunner`. A Figura 4-10 apresenta o diagrama de classes desse pacote. A seguir será detalhado cada um dos componentes do pacote `core`.

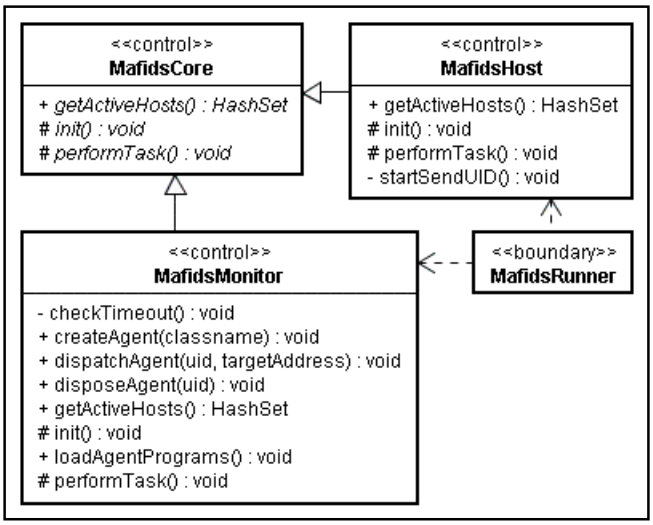

Figura 4-10 Diagrama de classes do pacote `core`.

MafidsCore

Define a estrutura básica dos controladores do MAFIDS, responsável por efetuar a comunicação entre os componentes do sistema. Dentre as responsabilidades deste componente, estão: o controle (criação, envio e finalização) dos agentes em atividade no sistema; processamento das mensagens enviadas e recebidas implementando a *interface* `MessageListener`.

A aplicação do padrão de projeto *Abstract Factory* permitiu o completo reuso da estrutura básica de `MafidsCore`. Um outro padrão de projeto foi aplicado, no intuito de permitir que determinadas partes do código pudesse mudar de comportamento de acordo com algum critério. A aplicação desse padrão foi realizada na definição de alguns métodos abstratos que são chamados no meio de alguns algoritmos da própria classe `MafidsCore`, dentre os quais estão os métodos `init()` e `performTask()`.

MafidsHost

Caracteriza uma especialização do `MafidsCore` que fornece um ambiente de atuação seguro para os agente que chegam num determinado *host.* Cada *host* a ser monitorado pelo SDI deve estar executando uma instância do `MafidsHost`. Devido a essa relação de unicidade entre os *hosts* da rede e o `MafidsHost`, ambos são referenciados como sendo uma única entidade.

Os *hosts* são de vital importância para o funcionamento do MAFIDS, visto que eles são responsáveis por manterem uma porta de comunicação em modo de escuta, ou seja, aguardando pedidos de conexão de outras entidades (monitor e agentes). Para tanto o `MafidsHost` possui uma instância de `ServerSocket`, que possibilita a realização desta tarefa. A porta de escuta utilizada pelos *hosts* pode ser configurada no arquivo de propriedades do MAFIDS, o qual é carregado para o MAFIDS através do subsistema de recursos que será apresentado mais adiante neste capítulo. A Figura 4-11 apresenta o diagrama de estados típicos do `MafidsHost`.

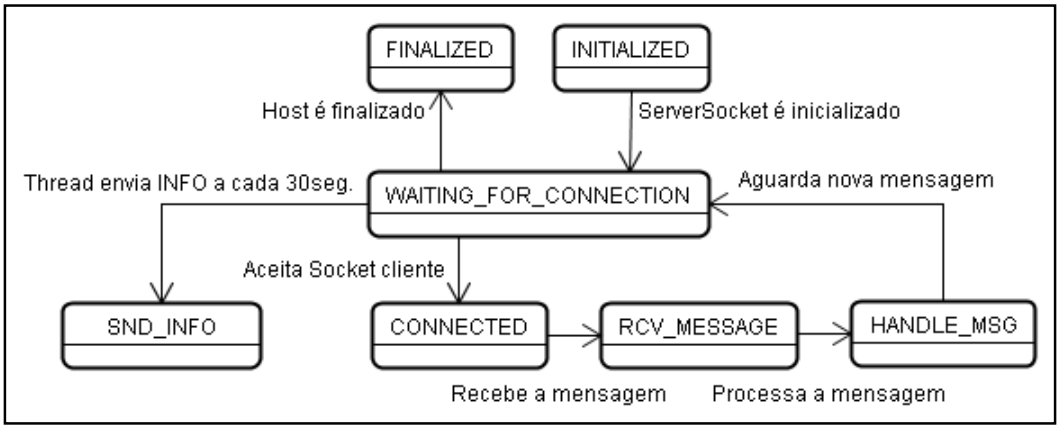

Figura 4-11 Diagrama estados do `MafidsHost`.

Após efetuar a inicialização dos recursos necessários para sua execução, o `MafidsHost` passa para o estado INITIALIZED. Neste estado, a `Thread` do servidor é iniciada e fica executando em *background.* Essa *thread* é responsável por manter uma determinada porta no *host,* configurada no arquivo de propriedades, em modo de escuta; após a inicialização do `ServerSocket`, o *host* passa para o estado WAITING_FOR_CONNECTION, onde aguarda por um pedido de conexão feito por parte dos clientes (monitor ou agentes) e a *thread* de envio de informações para o monitor é iniciada. Essa *thread* ficará ativa enquanto o *host* estiver em atividade, mantendo essa linha de execução no estado SND_INFO. Após o estabelecimento da

conexão com o cliente, o *host* passará ao estado CONNECTED e aguardará o recebimento da mensagem. Ao receber a mensagem, seu estado mudará para RCV_MESSAGE e o *host* deverá processar a mensagem recebida, o que o fará mudar para o estado HANDLE_MSG. Após finalizar o processamento das mensagens (e enviar o ACK), o *host* retornará ao estado WAITING_FOR_CONNECTION onde aguardará por um novo ciclo de recebimento. Caso o *host* seja finalizado, passará ao estado FINALIZED.

MafidsMonitor

Descreve o controlador do MAFIDS utilizado para a administração do *framework.* Alguns padrões de projeto foram aplicados para obter um componente final com baixo acoplamento com o resto do sistema. O padrão *Abstract Factory* foi aplicado para que fosse possível que a estrutura do `MafidsCore` fosse reutilizável. O padrão de projeto *Adapter* foi aplicado no relacionamento entre as classes de GUI, o que resultou na criação de algumas *interfaces* que permitisse a comunicação o `MafidsMonitor` (controlador) e `MafidsMonitorScreen` (GUI) sem que o encapsulamento dos pacotes fosse ferido. A Figura 4-12 apresenta o diagrama de estados típicos do `MafidsMonitor`.

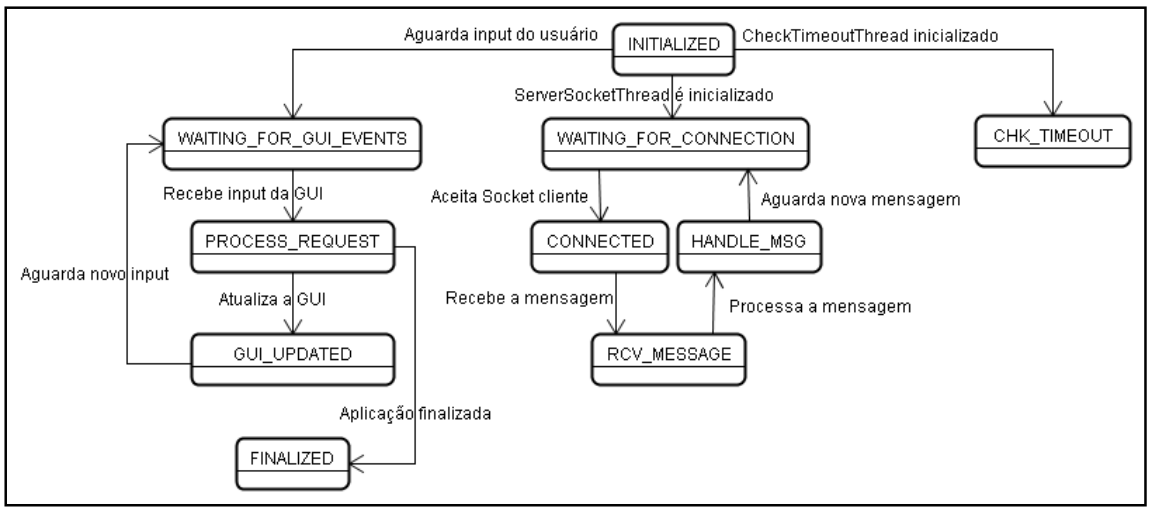

Figura 4-12 Diagrama estados do `MafidsMonitor`.

No diagrama apresentado na Figura 4-12 pode-se perceber três fluxos de execução bastante distintos. No primeiro, está caracterizado o relacionamento entre o controlador `MafidsMonitor` e a `MafidsMonitorScreen`. Enquanto aguarda pelo input do usuário, o monitor estará no estado WAITING_FOR_GUI_EVENTS. Ao receber algum *input* da GUI, o `MafidsMonitor` tratará de processá-lo e entrará no estado PROCESS_REQUEST. Processadas as entradas do usuário, o monitor deverá atualizar a GUI passando para o estado GUI_UPDATED e logo em seguida aguardará por um novo ciclo de entradas na GUI. Caso a usuário finaliza o MAFIDS através da GUI, o sistema passará ao estado FINALIZED. O segundo fluxo diz respeito ao recebimento de mensagens, que é idêntico ao que já foi explicado no `MafidsHost`. O terceiro e último fluxo de execução – CHK_TIMEOUT – diz respeito à *thread* de execução da checagem de *timeout* dos *hosts* ativos.

Através da GUI o administrador pode gerenciar a base de programas agentes carregados no *framework,* bem como instanciar esses programa em novos agentes para serem disparados na rede a ser monitorada. Além disso, é possível ainda acessar informações (endereço IP, agentes ativos, etc.) sobre os *hosts* que estão sendo monitorados, o que facilita a identificação e solução

de problemas durante a execução desses *hosts*. A Figura 4-13 apresenta o módulo da GUI responsável pelo gerenciamento dos programas agentes carregados no MAFIDS.

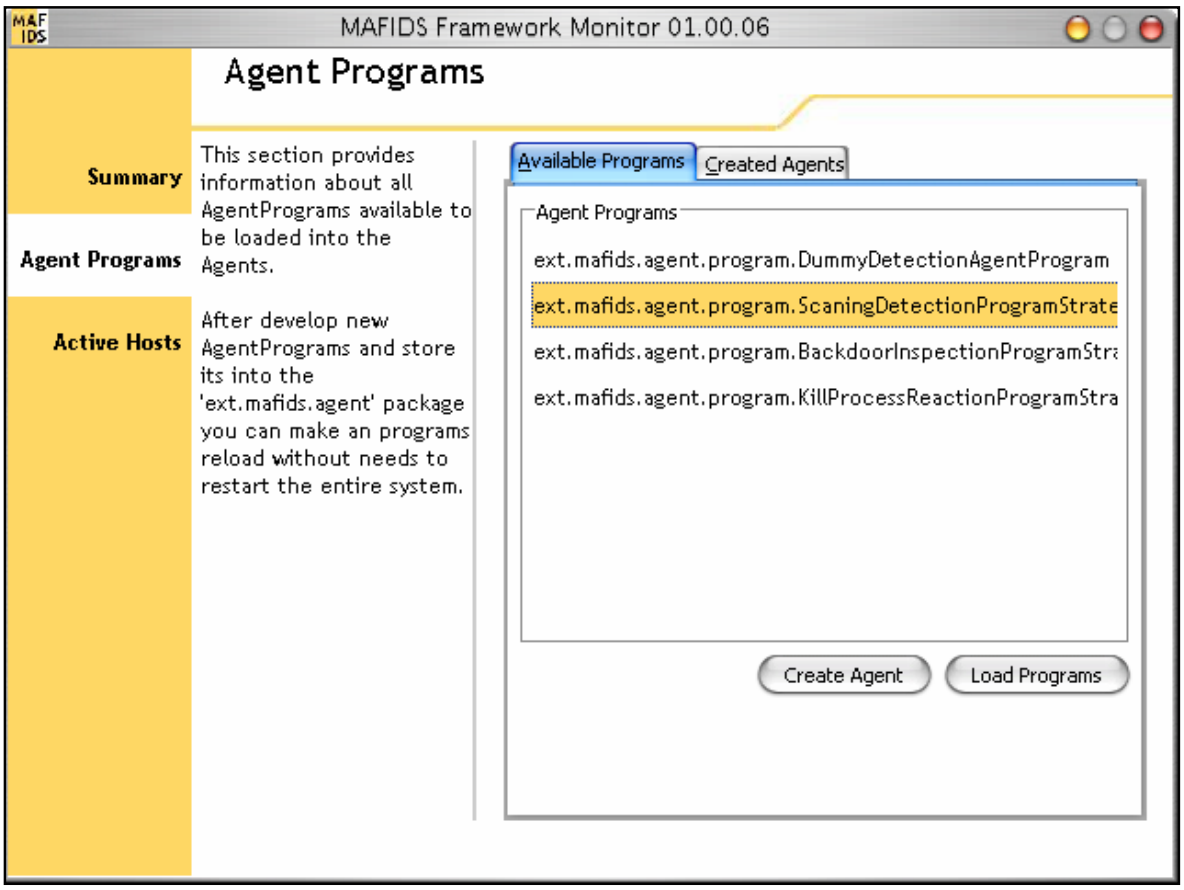

Figura 4-13 MAFIDS Monitor GUI – Agent Programs.

MafidsRunner

Esta classe representa a fachada de acesso ao *framework,* possibilitando a sua execução. O uso desta classe será detalhado mais adiante, na seção 5.3.5.

4.3 Protocolo de Comunicação

Segundo *Kurose* e *Ross* [20] , "um protocolo define o formato e a ordem das mensagens trocadas entre duas ou mais entidades comunicantes, bem como as ações realizadas na transmissão e no recebimento de uma mensagem ou outro evento." Em outras palavras, *Tanenbaum* [31] define um protocolo como "... um contrato entre as partes que querem se comunicar, onde está determinado como a comunicação deve proceder".

Este projeto define um protocolo próprio de interoperabilidade entre os componentes do sistema, chamado *Agent Messaging Exchange Protocol* (AMEP). A definição de um protocolo de comunicação próprio visa aumentar a segurança na interoperabilidade entre esses componentes, visto que sem conhecer o protocolo definido, elementos intrusivos não saberão como se comunicar com outros componentes, o que limitaria, assim, a ação desse elemento. O AMEP é um protocolo baseado em troca de mensagens sobre TCP – *Transmission Control Protocol* (para

mais informações sobre os protocolos TCP/IP consulte [17][20][27][31]), e como tal define um conjunto restrito de mensagens válidas para permitir a comunicação entre as partes envolvidas. Através da troca de mensagens é possível estabelecer uma máquina de estados do protocolo, que será apresentada com detalhes na seção 4.3.2.

4.3.1 Tipos de Mensagem

O protocolo AMEP apresenta um conjunto bem definido de mensagens que garantem a comunicação entre os componentes do sistema (Monitores, *Hosts* e Agentes). Essas mensagens são classificadas em dois grupos:

- *Mensagens de Protocolo*: Compreendem as mensagens estritamente de protocolo, ou seja, as mensagens utilizadas para iniciar e finalizar a comunicação entre as partes envolvidas. Uma outra característica das mensagens de protocolo é que elas não são utilizadas diretamente pela aplicação;

- *Mensagens de Aplicação*: Compreendem o conjunto maior de mensagens utilizadas pela aplicação para enviar e receber informações. Esse tipo de mensagem define que serviços um determinado componente oferece como, por exemplo, disponibilizar a lista de *hosts* ativos no sistema.

Além das mensagens, o protocolo define também o formato básico das mensagens. A Figura 4-14 apresenta o formato básico de mensagem proposto pelo AMEP. A seguir estão listados os diversos tipos e formatos de mensagens definidos pelo protocolo:

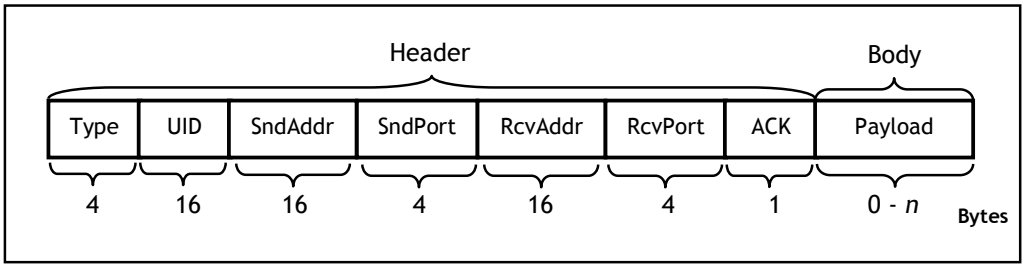

Figura 4-14 Formato básico de mensagem proposto pelo AMEP.

- Mensagens de Protocolo:
  - *CHECK_ALGORITHM:* Utilizada para checar a compatibilidade dos algoritmos de cifragem/decifragem presentes dos componentes comunicantes;
  - *HELLO:* Utilizada para dar inicio ao processo de comunicação entre as partes comunicantes;
  - *FINALIZE:* Mensagem responsável por sinalizar a finalização da comunicação;
- Mensagens de Aplicação:
  - *AGENT:* Mensagem utilizada para sinalizar o envio de um agente para o *host* de destino;
  - *AGENT_CHECKING:* Mensagem utilizada para verificar se um determinado agente foi finalizado pelo administrador no Monitor;
  - *AGENT_DISPOSE:* Mensagem utilizada para reportar ao Monitor que um determinado agente foi finalizado por encontrar problemas na execução;

- *AGENT_INFO:* Mensagem utilizada pelo agente para reportar ao monitor seu estado atual de execução;
- *AGENT_REQUEST:* utilizada pelo agente para solicitar o envio de um determinado agente para o *host* que ele está executando suas atividades;
- *ERROR:* Mensagem utilizada reportar um erro ocorrido em algum ponto da comunicação;
- *HOST_INFO:* Mensagem utilizada pelos *hosts* para reportar o seu estado atual de execução;
- *HOSTS_LIST:* Mensagem utilizada para solicitar a lista de *hosts* ativos ao Monitor.

4.3.2 Máquina de Estados

O processo de envio e recebimento de mensagens, segundo o protocolo definido, pode ser representado numa máquina de estados bem definida. A Figura 4-15 apresenta uma seqüência típica de estados para o envio de mensagens segundo o AMEP.

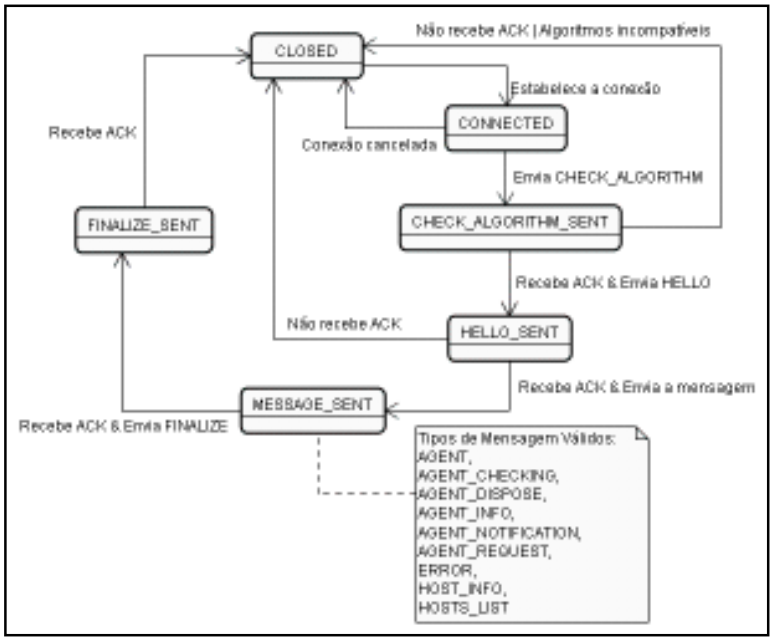

Figura 4-15 Uma seqüência típica de estados para o envio de mensagens segundo o AMEP.

Para facilitar o entendimento das máquinas de estado, duas abstrações serão criadas: entenda-se por *remetente* o lado que inicia o processo de comunicação – normalmente um agente, um *host* ou o próprio Monitor; e por *destinatário* a outra ponta da conexão, que pode ser um *host* ou o Monitor.

Inicialmente o remetente encontra-se no estado *CLOSED*. Após ser estabelecida a conexão com o destinatário, o remetente passa para o estado *CONNECTED*. Caso algum erro ocorra nesse processo de estabelecimento da conexão, o remetente deverá voltar ao estado inicial, *CLOSED*. Após o estabelecimento do canal de comunicação, o remetente envia um *CHECK_ALGORITHM* para checar a compatibilidade entre os algoritmos de cifragem utilizados nas duas pontas da conexão; caso não sejam compatíveis, a troca de mensagens não será possível. Após enviar a mensagem, o remetente passará para o estado *CHECK_ALGORITHM_SENT* e aguardará o recebimento do *acknowledgment (ACK)* correspondente. Recebido o *ACK,* o

remetente deverá enviar um *HELLO* para sinalizar o inicio da troca de mensagens de aplicação. Ao enviar essa mensagem, o remetente passará para o estado *HELLO_SENT* e aguardará o recebimento do *ACK* correspondente. Caso o *ACK* referente ao *HELLO* não seja recebido, o remetente deverá voltar ao estado *CLOSED*. Recebido o *ACK,* o remetente estará apto a enviar a mensagem de aplicação desejada. Após enviar a mensagem desejada, o remetente passará ao estado *MESSAGE_SENT* e aguardará o recebimento do *ACK* correspondente. Recebido o *ACK* indicando que a mensagem foi recebida corretamente, o remetente deverá iniciar o processo de finalização da comunicação enviando um *FINALIZE* que o levará ao estado *FINALIZE_SENT*. Ao receber o *ACK* do *FINALIZE,* o remetente finaliza o canal de comunicação, faz as devidas liberações de recursos e passa ao estado *CLOSED,* encerrando assim o ciclo de envio.

O processo de recebimento de mensagens funciona de modo semelhante. A máquina de estados definida pode ser observada na Figura 4-16.

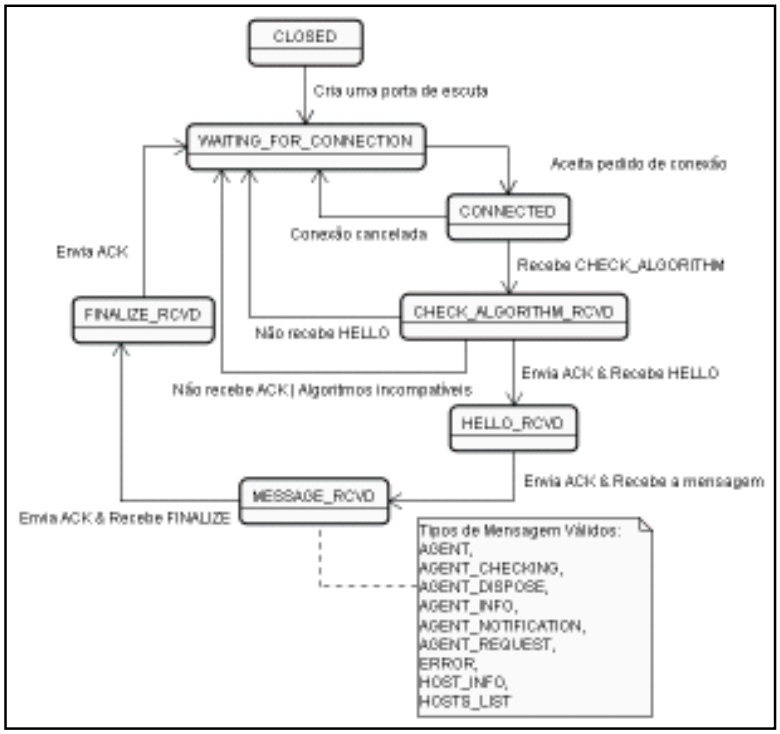

Figura 4-16 Uma seqüência típica de estados para o recebimento de mensagens segundo o AMEP.

Inicialmente o destinatário encontra-se no estado *CLOSED*. Ao iniciar uma porta para escuta da conexão, o destinatário passará ao estado *WAITING_FOR_CONNECTION*. Quando o remetente solicitar uma conexão e esta for aceita, o destinatário passará ao estado *CONNECTED*. Caso algum erro ocorra com a conexão neste momento, o destinatário deverá voltar ao estado *WAITING_FOR_CONNECTION,* onde aguardará um novo pedido de conexão. Após receber o pedido de checagem de compatibilidade dos algoritmos de criptografia, o destinatário passará ao estado *CHECK_ALGORITHM_RCVD*. O destinatário deverá enviar o *ACK* confirmando a compatibilidade dos algoritmos e aguardar pelo recebimento do *HELLO*, que dará inicio ao envio da mensagem de aplicação. Ao receber o *HELLO,* o destinatário passará ao estado *HELLO_RCVD*; caso não receba o *HELLO,* o destinatário deverá voltar ao estado *WAITING_FOR_CONNECTION*. Após enviar o *ACK* o destinatário deverá aguardar o recebimento da mensagem de aplicação, quanto então passará para o estado *MESSAGE_RCVD*.

Após enviar o *ACK* indicando que a mensagem foi recebida corretamente, o destinatário deverá aguardar o recebimento do pedido de finalização da conexão, que o levará ao estado *FINALIZE_RCVD.* Finalizando o ciclo de recebimento de mensagem, o destinatário deverá enviar o *ACK* de finalização e liberar os recursos alocados para esta conexão, passando então para o estado *WAITING_FOR_CONNECTION,* onde aguardará o próximo recebimento de mensagem.

4.3.3 Componentes de Comunicação do AMEP

O protocolo AMEP foi proposto e implementado com base no protocolo TCP utilizando *Sockets* de comunicação. A linguagem JAVA [15][16] permite a criação de *sockets* através das classes `Socket` e `ServerSocket` que se encontram no pacote `java.net` (para mais informações sobre a criação de *sockets,* consulte [16]).

No MAFIDS, a implementação do AMEP encontra-se no pacote `message`. Este pacote compreende as classes básicas que modelam o formato das mensagens trocadas, o componente de envio/recebimento de dados utilizando *sockets*, os componentes de envio e de recebimento de mensagens compatível com as especificações do AMEP e a interface que define a manipulação das mensagens. A Figura 4-17 apresenta o diagrama de classes do pacote `message`, mostrando o relacionamento entre esses componentes. As seções seguintes detalham cada classe do pacote apresentando suas características mais relevantes.

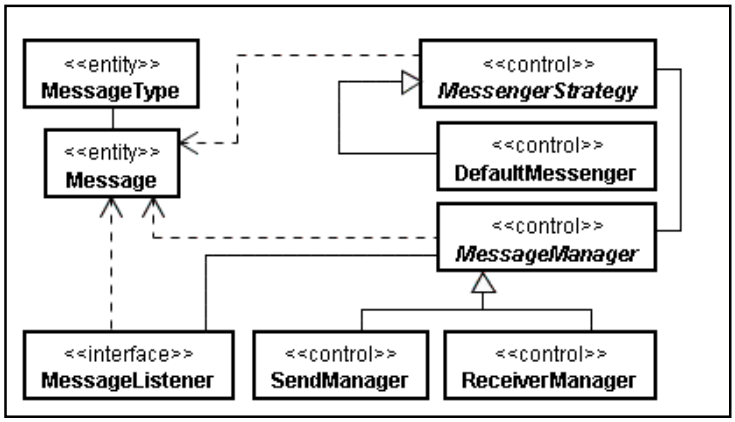

Figura 4-17 Diagrama de classes do pacote `message`.

4.3.3.1 Message

A classe `Message` encapsula as informações necessárias para permitir o envio e recebimento de dados através da rede, simplificando consideravelmente a implementação dos componentes responsáveis de comunicação.

O elemento mais importante da classe `Message` é o `enum` – ferramenta incorporada à linguagem JAVA na versão 5.0 [28] – `MessageType`, que define os tipos de mensagens suportados pelo AMEP. A utilização do `enum`, em lugar da declaração de constantes de tipo (implementação tradicional), apresenta benefícios significativos, dentre os quais um foi decisivo para a sua adoção no projeto: a facilidade de manutenção que a declaração do `enum` proporciona. A Figura 4-18 apresenta o trecho de código onde o `MessageType` é definido.

```
/* Declaração do MessageType */
public static enum MessageType {
      CHECK_ALGORITHM(0), HELLO(1),  FINALIZE(2),
      ERROR(3), AGENT(4), AGENT_NOTIFICATION(5),
      AGENT_REQUEST(6), AGENT_INFO(7), AGENT_CHECKING(8),
      AGENT_DISPOSE(9), HOST_INFO(10),    HOSTS_LIST(11);

      // alguns códigos ...

      // Método que retorna o valor do enum.
      public int getValue(){...}
}
```

Figura 4-18 Definição do `MessageType` na classe `Message`.

4.3.3.2 Message Listener

A *interface* `MessageListener` é resultado da aplicação dos padrões de projeto *Adapter* e *Template Method* na classe `MessageManager`, que será explicada mais adiante. Ela define os métodos de manipulação de mensagens e de erros. Cada classe que deseja efetuar o envio ou recebimento de mensagens deve implementar essa *interface* definindo as ações a serem tomadas para as mensagens de seu interesse. O MAFIDS define três atores para a realização do envio e/ou recebimento de mensagem, são eles: `MafidsMonitor`, `MafidsHost` e `Agent`.

A aplicação desses padrões permitiu a simplificação dos algoritmos de envio e recebimento, localizados na classe `MessageManager`, delegando a manipulação das mensagens para as classes que possuem o conhecimento necessário para manipulá-las. Além disso, os padrões permitiram isolar as classes do pacote `message` mantendo-o um componente para o resto do sistema.

4.3.3.3 Messenger Strategy

A definição do AMEP abrange a utilização de diferentes mecanismos de criptografia para a troca de mensagens seguras. Essa característica foi obtida com a aplicação do padrão de projeto *Strategy* (como o nome da classe sugere), que permitiu isolar o algoritmo de envio e recebimento de dados (através da serialização de objetos [16]) tornando a mudança de algoritmo uma tarefa simples e que não trás maiores impactos para o resto do sistema.

O fato de o `MessengerStrategy` ser definido como o *hot-spot* do MAFIDS permite que o usuário do *framework* implemente seu próprio mensageiro utilizando o algoritmo de criptografia de sua preferência. Para facilitar o trabalho de implementação de novos mensageiros, a classe `MessengerStrategy` oferece implementação para o envio e recebimento de objetos – `Object`; desta forma, o implementação do novo componente mensageiro só precisara se deter ao algoritmo de criptografia das mensagens. A seção 5.3.3 apresenta o uso desse componente para a construção de novos mensageiros.

4.3.3.4 Default Messenger

Implementação concreta do MessengerStrategy que realiza a comunicação através de texto plano, ou seja, sem nenhum mecanismo de criptografia. Esse mensageiro também é utilizado para realizar o envio das mensagens de checagem de compatibilidade de algoritmos. Caso o usuário do MAFIDS não defina nenhum mensageiro extra, o `DefaultMessenger` será utilizado para realizar a comunicação.

4.3.3.5 Message Manager

O `MessageManager` constitui o componente base da implementação do AMEP, que foi divida em três etapas eliminatória, ou seja, a etapa anterior é pré-requisito para a próxima etapa. A primeira etapa do protocolo é a verificação da compatibilidade entre os algoritmos de criptografia utilizados, que é feita através dos métodos `sendAlgorithm()` e `receiveAlgorithm()`. Caso a verificação falhe, a mensagem não será enviada; caso contrário o algoritmo passará para a segunda etapa, o envio do *HELLO* denotando o início do envio das mensagens de aplicação. O envio do *HELLO* foi encapsulado no método `sendHello`

Além de oferecer a implementação para os primeiros passos do protocolo, o `MessageManager` é responsável por guardar as instancias do `MessengerStrategy`, e do `MessageListener`, que são utilizados durante todo o processo de comunicação. Devido à elevada complexidade das operações de envio e recebimento de mensagens, estas foram delegadas para as respectivas subclasses de `MessageManager`: `SendManager` e `ReceiveManager`, que serão explicadas nas próximas seções.

4.3.3.6 Send Manager

Responsável pela implementação do mecanismo de envio de mensagens em conformidade com as especificações do AMEP. O `SendManager` utiliza dois padrões de projeto: O *Adapter* foi utilizado para permitir que o `SendManager` tivesse acesso a recursos de controladores de outros pacotes sem acessá-los diretamente. O *Template Method*, que permitiu flexibilizar (em pontos específicos) o comportamento do algoritmo do `SendManager` que é responsável pela manipulação de mensagens. A Figura 4-19 apresenta o trecho de código onde o *Template Method* está sendo aplicado na classe `SendManager`. A chamada ao método `handleMessage` (a) utilizando o MessageListener permite que o comportamento do método `handleMessage` (b) varie de acordo com as necessidades de quem implementa esse método no *listener*.

```
public class SendManager extends MessageManager {
      // inicialização de variáveis ...
      MessageListener listener;          // herdado da superclasse

      // Outros métodos da classe

      // Algoritmo de manipulação das mensagens - refactoring.
(b)   private void handleMessage (Socket socket, Message msg){
(a)         this.listener.handleMessage(msg);
            // mais código aqui ...
      }
}
```

Figura 4-19 Aplicação do *Template Method* na classe `SendManager`.

4.3.3.7 Receive Manager

O `ReceiveManager` segue os mesmos padrões aplicados para o `SendManager` e as mesma justificativas, pois as duas classes são complementares. Enquanto o `SendManager` implementa o método `receive()` com o corpo vazio, o `ReceiveManager` implementa o método `send()` com o corpo vazio. A Figura 4-20 apresenta o trecho de código onde o *Template Method* está sendo aplicado na classe `ReceiveManager`. Da mesma forma que o

`SendManager`, a chamada ao método `handleMessage` (a) utilizando o MessageListener caracteriza a aplicação do padrão de projeto *Template Method.*

```
public class ReceiveManager extends MessageManager {
      // inicialização de variáveis ...
      MessageListener listener;          // herdado da superclasse

      // Algoritmo de envio das mensagens
      private void receiveMessage(Socket socket){
            Message msg;
            // alguns códigos ...

            // refactoring para diminuir a complexidade do método.
            this.handleMessage(socket, msg);
      }

      // Algoritmo de manipulação das mensagens - refactoring.
      private void handleMessage (Socket socket, Message msg){
(a)         Obj o = this.listener.handleMessage(msg);
            // mais código aqui ...
      }
}
```

Figura 4-20 Aplicação do *Template Method* na classe `ReceiveManager`.

Capítulo 5

Aplicação do *MAFIDS* em um SDI para a detecção de *Backdoors*

Os *frameworks* verticais, a exemplo do MAFIDS, são desenvolvidos para um domínio de aplicação específico. Como forma de avaliar o seu comportamento diante de um problema real (dentro do seu domínio de aplicação), o *framework* deve ser instanciado numa aplicação que reflita problemas reais. As seções seguintes apresentam o cenário proposto para a aplicação do *framework* em ambiente real.

5.1 Considerações Iniciais

Backdoors constituem uma parcela significativa dos ataques sofridos pelas empresas no mundo todo. Segundo a *CIO e PwC* [8], em 2005, 59% das empresas citaram o código malicioso – *software* ou porção de *software* com fins destrutivos que visam prejudicar a vítima apagando ou alterando informação confidenciais, prejudicando o funcionamento de sistemas, roubando informações, etc. – o meio de ataque mais utilizado, seguido pelo acesso não autorizado, que vem crescendo cerca de 25% a cada ano. O *backdoor* é essencialmente composto por código malicioso que, em muitos casos, permitem o acesso não autorizado à *hosts* ou sistemas em geral; esse tipo de ameaça produz efeitos colaterais (comportamento anômalo) perceptíveis no funcionamento do sistema e possuem um certo padrão de atuação. Devido ao volume de ataques dessa categoria ocorridos nos últimos anos, acredita-se que esse seja um cenário de aplicação relevante para o MAFIDS.

O MAFIDS foi desenvolvido para facilitar a construção de aplicações de um domínio específico (no caso, para a construção de SDIs). Desta forma, o MAFIDS propõe alguns *hot-spots* que devem ser implementados pelo usuário do *framework,* permitindo que ele defina o comportamento desses *hot-spots* conforme a sua aplicação. A maioria dos *hot-spots* definidos possui uma implementação padrão definida pelo *framework,* que será usada caso o usuário não apresente sua implementação específica. No entanto, buscando a maior generalização possível para o MAFIDS (dentro de seu domínio de aplicação), o *hot-spot* responsável por modelar os programas agentes não oferece nenhuma implementação padrão (visto que esse conceito não se aplica aos programas agentes). De uma forma geral, a aplicação do MAFIDS para um cenário específico consiste na implementação de seus *hot-spots* e na devida configuração dos arquivos de

propriedades distribuídos com ele. A Figura 5-1 apresenta o relacionamento entre uma instância do MAFIDS para um problema específico e o próprio *framework*; essa figura é semelhante à Figura 4-1 apresentada no início do Capítulo 4.

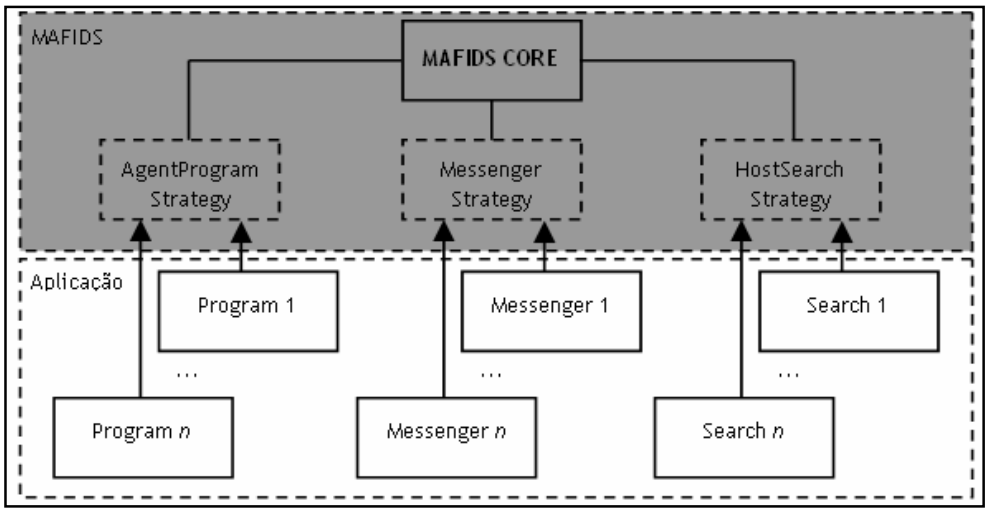

Figura 5-1 Relacionamento entre a aplicação e o MAFIDS.

Como pode ser observado na figura, existe uma clara separação entre o código do *core framework* e o código da aplicação. Desta forma, qualquer mudança poderá ser feita na aplicação sem impactar no código do *framework.*

5.2 Cenário de Aplicação

Para facilitar o entendimento da aplicação do MAFIDS, um cenário foi proposto de forma que as explicações pudessem ser apresentadas com o auxílio de exemplos. Segundo *Mattson* [22], a utilização de exemplos para as explicações de como se utilizar um *framework* são mais importantes que as próprias explicações, visto que o usuário pode saber se ele realmente entendeu as explicações do *framework*. Sendo assim, este capítulo propõe a construção de um SDI para detecção de *backdoors* – o *BackIDS.* A Figura 5-2 apresenta o diagrama de estados completo da execução do *BackIDS,* apresentando também a divisão das camadas.

O BackIDS foi projetado para detectar programas que estejam mantendo as portas de comunicação em modo de escuta, ou seja, aguardando alguma conexão externa. Inicialmente, alguns agentes de detecção são distribuídos pelos *hosts* que estão sendo cobertos pelo SDI (SCANNING DETECTION AGENT SENT). Esses agentes são responsáveis por montar um perfil de cada *host* com a lista das portas que estão em modo de escuta (HOST PROFILE BUILT). Além disso, são registradas também informações a respeito do processo que está controlando aquelas portas. Quando esses agentes de detecção encontram alguma porta desconfiável, os agentes de inspeção são acionados (INSPECTION AGENT REQUEST SENT). Quando acionados, os agentes de inspeção analisam a confiança das portas coletadas pelos agentes de detecção (PROFILE ANALIZED) e, caso o índice de confiança da porta seja menor que o limite de confiabilidade configurado pelo usuário, esse agente de inspeção enviará uma notificação ao administrador (NOTIFICATION SENT).

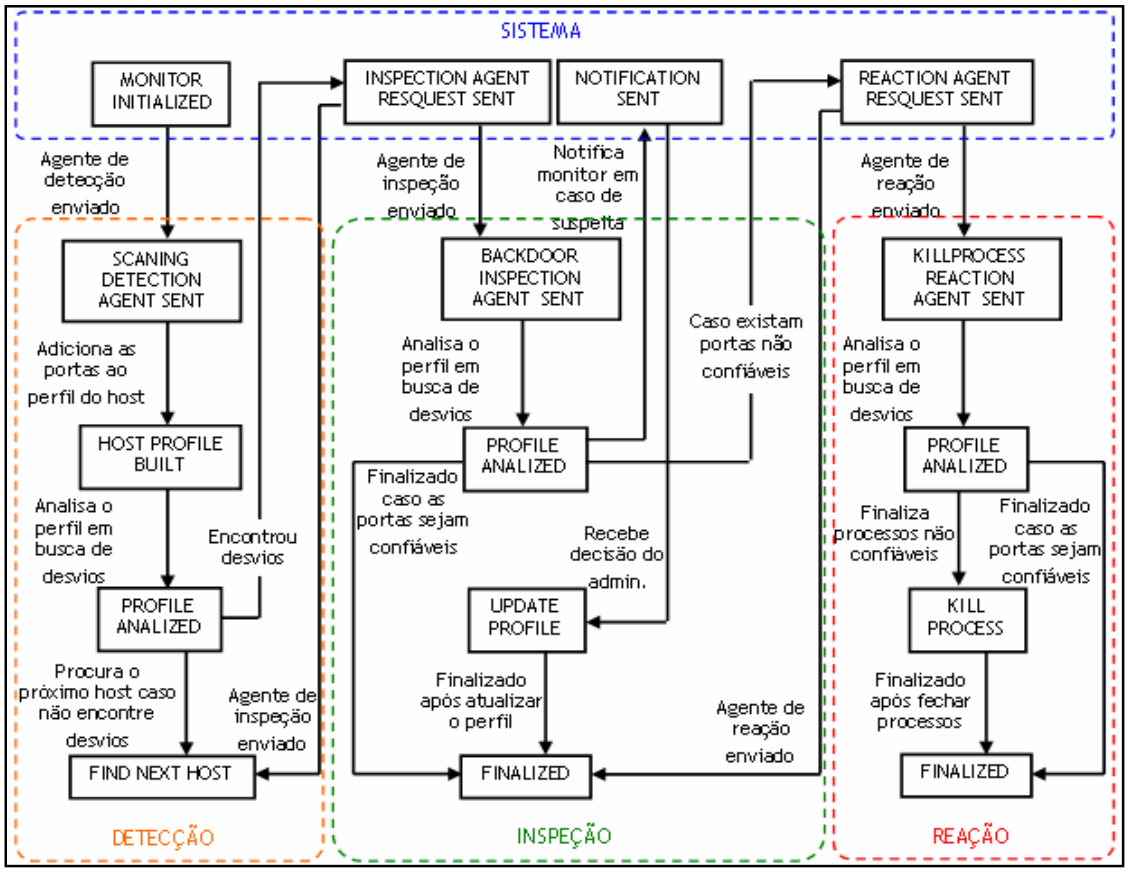

Figura 5-2 Diagrama de estados do *BackIDS*.

A opinião do administrador acerca dessa intrusão é utilizada para que o agente atualize o índice de confiança dessa porta (UPDATE PROFILE). Uma vez que o administrador não considere esse fato uma ameaça, a confiança nessa porta é incrementada; caso contrário, a função de ajuste decrementará a confiança. Quando uma porta for considerada não confiável, o agente de inspeção fará uma requisição para acionar o agente de reação (REACTION AGENT REQUEST SENT). Este último é responsável por tomar uma medida reativa para inibir a intrusão em andamento. No caso, esse agente será responsável por finalizar os processos que estejam mantendo alguma porta, considerada não confiável, em modo de escuta (KILL PROCESS).

O *BackIDS* foi elaborado com caráter acadêmico, no intuito de apresentar de forma clara os passos necessários para a instanciação do MAFIDS em um domínio de aplicação específico – SDI. Portanto, não será dada ênfase nos algoritmos relativos às regras de negócio da aplicação (tais como algoritmos de busca, algoritmos de IA, etc.). Tendo em vista que o cenário de aplicação foi escolhido com base em informações reais sobre os tipos de ataques mais recorrentes, relatados pela *CIO e PwC* [8] e baseado no fato de o *Windows® XP Professional* ser um dos Sistemas Operacionais (SO) mais difundidos no ambiente corporativo, o *BackIDS* foi projetado para esta plataforma. Por ter sido desenvolvido baseado na linguagem de programação JAVA 5.0 (para mais informações sobre a linguagem Java 5.0, consulte [28]), o BackIDS requer a Máquina Virtual Java (JVM – *Java Virtual Machine*) na versão 5.0_06 ou superior. Além disso, como o SO não provê acesso direto à informações sobre portas, processos, e finalização de processos (informações estas necessárias para o BackIDS), faz-se necessário o uso de ferramentas de apoio para esta tarefa. No intuito de diminuir o número de dependências com sistemas

externos, procurou-se utilizar ferramentas já disponíveis no próprio SO, são elas: *netstat.exe, tasklist.exe* e *taskkill.exe*;

5.3 Implementação do *BackIDS*

As seções seguintes apresentarão de forma prática os passos para a instanciação do MAFIDS no domínio de aplicação de SDIs. Para este trabalho será utilizada a IDE – *Integrated Development Environment – Eclipse* [10], por ser uma IDE bastante validada, largamente adotada tanto no meio acadêmico como na indústria, e por ser gratuita (lembrando que uma das motivações deste trabalho é a redução de custo de desenvolvimento de SDIs).

5.3.1 Importando e Configurando o *Core* do MAFIDS

O primeiro passo para o desenvolvimento do *BackIDS,* é importar o *core* do MAFIDS e configurar apropriadamente os arquivos de propriedades. Uma vez importado para aplicação, no caso o *BackIDS*, o *core* provê acesso aos *hot-spots* do *framework,* incluindo a classe `AgentProgramStrategy` – *hot-spot* para os programas agentes. A distribuição do MAFIDS é composta por quatro arquivos: o *core (mafids.jar),* o arquivo de propriedades do *core (mafids.config),* o arquivo de propriedades dos *hot-spots (hotspot.config)* e o arquivo de propriedades das telas *(screens.config).* É importante lembrar que durante a configuração das propriedades, os valores das chaves não devem ser alterados, pois o MAFIDS define chaves específicas deixando os respectivos valores configuráveis pelo usuário. A Figura 5-3 apresenta a estrutura básica de um arquivo de propriedades.

```
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">

<properties>

      <!-- Configurações do monitor -->
      <entry key="monitor.port">12345</entry>
      ... outras propriedades

</properties>
```

Figura 5-3 Estrutura dos arquivos de propriedades do MAFIDS.

Apesar de os arquivos serem configurados manualmente, o desenvolvimento de uma GUI para realizar esta atividade é uma tarefa simples e de fácil integração com o resto do *framework;* tendo sido, inclusive, proposta na seção de trabalhos futuros.

5.3.1.1 mafids.config

Utilizado para armazenar configurações gerais do MAFIDS, tais como o endereço IP do monitor, as portas de comunicação utilizadas pelo monitor e pelos *hosts*, os *paths* utilizados pelo subsistema de persistência, etc. A Figura 5-4 apresenta trechos do arquivo *mafids.config* utilizado para o *BackIDS.*

```
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">

<properties>
      <!-- Configure the IDS Monitor -->
(a)   <entry key="monitor.address">192.168.10.254</entry>
(b)   <entry key="monitor.port">5005</entry>

      <!-- Configure the IDS Host -->
(c)   <entry key="host.port">5006</entry>
      ...
      <!-- Configure Persistence Path -->
(d)   <entry key="programs.db.path">data/programs.db</entry>
      ...
</properties>
```

Figura 5-4 Código do arquivo de propriedades *mafids.config*.

Conforme o arquivo de propriedades, o monitor estará executando na máquina de endereço IP 192.168.10.254 (a) utilizando a porta 5005 (b). O *host* estará utilizando a porta 5006 (c) e a base de programas será salva no caminho `data/programs.db`.

5.3.1.2 hotspot.config

Arquivo de configuraçção dos *hot-spots* do MAFIDS. Dentre as propriedades armazenadas estão o *classpath* da aplicação e o nome do pacote onde estão localizadas as subclasses de `AgentProgramStrategy`, `MessengerStrategy` e `HostSearchStrategy`. A Figura 5-5 apresenta trechos do arquivo *hotspot.config* utilizado para o *BackIDS*.

```
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">

<properties>
      <comment>MAFIDS Framework HotSpot Configuration File</comment>
      <entry key="classpath">bin/</entry>
      <entry key="programs.path">bin/ext/mafids/agent/program</entry>
      <entry key="programs.package">ext.mafids.agent.program</entry>

      <!-- BackIDS does not implements the messenger strategy
            <entry key="messenger.package">ext.mafids.message</entry>
      -->
</properties>
```

Figura 5-5 Código do arquivo de propriedades *hotspot.config*.

No arquivo *hotspot.config* estão sendo definidos os pacotes onde as implementações dos *hot-spots* estão armazenadas. Como se pode observar, o BackIDS só implementa o *hot-spot* dos programas agentes.

5.3.1.3 screens.config

Utilizado para configurar as propriedades de telas do MAFIDS, tais como: posicionamento da janela principal, posicionamento dos *frames* internos, dimensões dos componentes utilizados, etc. Usualmente esse arquivo só é alterado quando o MAFIDS for sofrer alguma alteração na GUI.

5.3.2 Implementando Programas Agentes (`ProgramStrategy`)

5.3.2.1 Programas de Detecção

O `ScanningDetectionProgramStrategy` é o ator inicial do processo de detecção, que se inicia com o escaneamento dos *hosts* cobertos pelo *BackIDS* em busca de portas de comunicação que estejam em modo de escuta. A estrutura da classe dos programas de detecção, inspeção e intrusão são muito parecidas, visto que todo programa agente, herda as características definidas pelo `AgentProgramStratey`. A Figura 5-6 apresenta o diagrama UML da classe `ScanningDetectionProgramStrategy`.

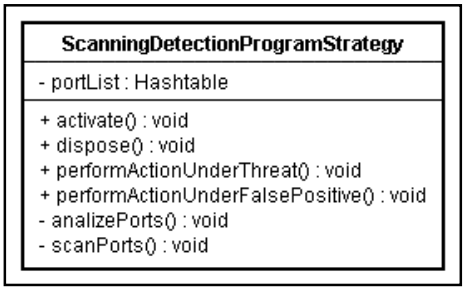

Figura 5-6 Diagrama UML da classe `ScanningDetectionProgramStrategy`.

A classe `AgentProgramStrategy` não fornece o construtor sem argumentos. O único construtor oferecido recebe o tipo do agente como parâmetro, logo, o construtor precisa ser redefinido e na chamada ao `super()` deve ser passado o tipo de agente para o qual o programa está sendo desenvolvido que, no caso dos agentes de detecção é `AgentType.DETECTION_AGENT`. A Figura 5-7 apresenta um trecho de código onde o construtor é redefinido.

```
public class ScaningDetectionProgramStrategy extends AgentProgramStrategy {
      /* Atributos de classe */

      /* Chama o construtor da superclasse */
      public ScaningDetectionProgramStrategy(){
            super(AgentType.DETECTION_AGENT);
      }

      /* Outros métodos */
}
```

Figura 5-7 Construtor do `ScanningDetectionProgramStrategy`.

Conforme já foi apresentado no Capítulo 4, a classe `AgentProgramStrategy` define quatro métodos abstratos que devem ser implementados no momento da instanciação do *framework;* no entanto o `ScanningDetectionProgramStrategy` só implementou três deles: `performActionUnderThreat()`, `activate()` e `dispose()`. O método `dispose()` é utilizado para liberar os recursos alocados pelo programa. O método `activate()` é chamado pelo agente quando o mesmo é ativado; no *BackIDS* este método executa alguns passos bem definidos. A Figura 5-8 apresenta trechos da implementação do método `activate()`.

```
/* Inicia as atividades do agente no host. */
public void activate(){
      /* Carrega o perfil do host (ou cria se não existir) */
(a)   this.portList = HostProfileRepository...getProfile();
      if (portList == null) portList = new Hashtable();

      /* execução do comando para listar as portas */
(b)   String cmd = "netstat -ano -p tcp";
      Process proc = Runtime...exec(cmd);
      BufferedReader input = proc.getInputStream();
      String line;
      while((line = input.readline) != null){
(c)         scanPorts(line);
      }

      /* Analisa as portas encontradas */
(d)   analizePorts();
}
```

Figura 5-8 Implementação do método `activate()` do agente de detecção.

Algumas considerações sobre a implementação do método `activate()`:

- A primeira atividade do programa é carregar o perfil do *host* ou criar um caso ele ainda não tenha sido criado. Esse procedimento pode ser observado na Figura 5-8 (a);
- Para efetuar a busca por portas no estado de escuta, o agente de *scanning* utiliza uma ferramenta de apoio distribuída com o próprio SO – o *Netstat.* A linha de comando executada para a obtenção do resultado desejado é apresentada na Figura 5-8 (b);
- A montagem da lista de portas é feita através do método auxiliar `scanPorts()`, que é chamado a cada linha de execução do *netstat.* Na medida em que as portas em modo de escuta vão sendo encontradas, a `Hashtable portList` vai sendo preenchido com as portas suspeitas. A chamada ao método `scanPorts()` pode ser observada na Figura 5-8 (c);
- Após a análise, um método auxiliar é chamado: o `analizePorts()`, que pode ser observado na Figura 5-8 (d). Cada porta é identificada com um nível de confiança, que é atualizado por funções que a incrementam ou decrementam de acordo com a situação; Caso o nível de confiança seja desconfiável, o agente considerará a porta em questão uma ameaça e chamará o método responsável por tomar as medidas de reação – `performActionUnderThreat()`. Nesse método é feita a requisição ao agente de inspeção que será enviado para auxiliar nas investigações.

5.3.2.2 Programas de Inspeção

Caracteriza o agente inteligente responsável por fazer a análise dos dados obtidos e tomar as decisões necessárias para inibir a ação intrusiva. Com base no perfil montado pelo agente de *scanning,* o `BackdoorInspectionProgramStrategy` atualiza o nível de confiança das portas de acordo com a opinião do administrador a respeito dessa porta.
Para instanciar o programa de inspeção, o usuário deverá redefinir o construtor informando o seu tipo, no caso `AgentType.INSPECTION_AGENT`.

Após carregar o perfil do *host,* o programa de inspeção analisa cada porta com base no seu índice de confiança. Ao se deparar com uma porta cuja confiança não pode ser atestada, o agente de inspeção notifica o administrador. Ao receber o *feedback,* essa informação é usada como entrada para a função de atualização do índice de confiança das portas. Esse algoritmo é implementado no método `updateProfile()`, que incrementa ou decrementa a confiança da porta de acordo com a decisão do administrador. Caso a porta analisada seja confiável, o programa permite que o processo envolvido continue executando. Caso contrário – a porta não é confiável – o agente de reação será acionado para inibir a intrusão derrubando o processo responsável. A Figura 5-2 apresentada no começo deste capítulo mostra o fluxo de estados do agente de inspeção, incluindo a análise e atualização do perfil do *host.* A Figura 5-9 apresenta a implementação da função de atualização da confiança da porta.

```
// Atualiza a confiança da porta com base no feedback do admin
private void updateProfile(boolean isThreat){
(a)   String entry = this.portList.get(this.notifiedPort);
(b)   ArrayList<String> tokens = WXPUtil.parseInfo(entry);

      // Retrieve port info
      ...
(c)   int level   = Integer.parseInt(tokens.get(3));

      // Update reliance level
(d)   level += isThreat ? -1 : 1;
      this.portList.put(... + level);
      ...
}
```

Figura 5-9 Implementação do método `updateProfile()` do agente de inspeção.

A primeira parte do algoritmo, apresentada na Figura 5-9 (a), recupera as informações da porta que está sendo testada. Como as informações são gravadas em um formato próprio, a saber: <número da porta>#<id do processo>#<nome do processo>#<confiança>, faz-se necessário que essa informação seja extraída através da chamada ao método utilitário `parseInfo()` que pode ser observado na Figura 5-9 (b). Com as informações em mãos, o agente recupera o índice de confiança e faz a atualização, como está descrito na Figura 5-9 (c) e (d), respectivamente.

5.3.2.3 Programas de Reação

Para este tipo de agente, o *BackIDS* implementa o `KillProcessProgramStrategy`, que é responsável por finalizar os processos relacionados às portas não confiáveis. Ao identificar que existe alguma porta não confiável em atividade, o agente de inspeção solicita ao monitor o envio deste agente. A estrutura de classe do `KillProcessProgramStrategy` é muito semelhante (por serem subclasses da mesma classe – o *hot-spot* `AgentProgramStrategy`) às outras duas já apresentadas. Sua atividade se inicia na chamada ao método `activate()`, que é feita pela própria arquitetura do agente de reação. Apesar de a execução ser iniciada no método `activate()`, todo o trabalho de análise das portas e finalização do processo devidos é feito pelo método `analizePorts()`. A Figura 5-10 apresenta a implementação desse método.

```
// Finaliza os processos das portas não-confiáveis
private void analizePorts(){
      // Declarações de variáveis...

      // Analisa as portas
      for (String entry : this.portList.values()){
(a)         level = ...

            // Kill untrusted port processes
(b)         if (level < WXPUtil.UNTRUSTED_LEVEL){
                  tokens = WXPUtil.parseInfo(entry);

                  // Kill process.
(c)               WXPUtil.killProcess(pid, pname);
            }
      }
}
```

Figura 5-10 Implementação do método `analizePorts()` do agente de reação.

Algumas considerações a respeito da implementação do método `analizePorts()`:

- Para cada porta registrada no perfil do *host,* o índice de confiança é recuperada para posterior análise, como pode ser observado na Figura 5-10 (a);

- Caso a porta seja não confiável, o agente irá recuperar as informações a respeito da porta para finalizar o processo – Figura 5-10 (b);

- O algoritmo de finalização do processo foi encapsulado numa biblioteca de métodos utilitários para *Windows XP*. Desta forma é possível reusar esse código sem grandes problemas; além disso, a manutenção do código ficou muito mais simples de ser realizada.

Para o completo entendimento desse cenário é necessário entender o funcionamento do método `killProcess()` que está localizado na classe utilitária `WXPUtil`. Para ter acesso aos processos responsáveis pela atividade das portas em questão, foi necessário utilizar mais duas ferramentas de apoio: *tasklist* e *taskkill.* A primeira é responsável por listar os processos ativos no SO e a segunda por finalizar processos. Mais uma vez, tendo em vista a menor dependência possível com ferramentas externas, foram escolhidas ferramentas que são distribuídas com o *Windows® XP Professional* – SO alvo deste trabalho. A Figura 5-11 apresenta a implementação do algoritmo de finalização de processos utilizado no *BackIDS*.

A ferramenta *taskkill* possui uma variada lista de argumentos, no entanto, para a finalização do processo é suficiente a aplicação do filtro que identifica o processo que se quer finalizar. A linha de comando utilizada no BackIDS para a finalização de processos pode ser observada na Figura 5-11 (a). A linguagem Java fornece um mecanismo para execução de aplicações externas – necessário para utilizar as ferramentas de apoio. Esse mecanismo está encapsulado na classe `Process`, que pode ser obtida através da chamada `Runtime.getRuntime().exec()`. Esse procedimento é apresentado na Figura 5-11 (b). Após a finalização das atividades, os recursos alocados são liberados – Figura 5-11 (c).

```
public static void killProcess(String pid, String pname){
  if (isProcessRunning(pid)){
    String cmd =
(a)   "tools\\taskkill -F -fi \"PID eq "+pid+"\"";
    String strLine;
      try{
(b)   Process process = Runtime.getRuntime().exec(cmd);
      BufferedReader inputProcess =
          new BufferedReader(new InputStreamReader(
                                  process.getInputStream()));

(c)   // Finalize resources
      process.destroy();
      inputProcess.close();
      }catch (IOException ioe){ ... }
  }
}
```

Figura 5-11 Implementação do algoritmo de finalização de processos.

5.3.3 Implementando Mensageiros (`MessengerStrategy`)

A implementação padrão do `MessengerStrategy` não utiliza nenhum mecanismo de criptografia. Tendo em vista que, para determinados sistemas, a segurança nas troca de mensagens é de fundamental importância, o MAFIDS oferece ao usuário a possibilidade de implementar seus próprios componentes mensageiros, de forma a atender às suas necessidades. Esta seção mostra como implementar um mensageiro para ser utilizado no MAFIDS. A Figura 5-12 apresenta trechos de uma possível implementação de um mensageiro que utiliza o algoritmo de criptografia AES – *Advanced Encryption Standard* [31], cuja chave simétrica está configurada no arquivo de propriedades do *hot-spot.*

A seguir são apresentados algumas considerações a respeito do código apresentado:

- A instanciação do mensageiro deve herdar da classe `MessengerStrategy` e implementar os métodos abstratos nela definidos: `setup()`, `sendMessage()` e `receiveMessage()` – `Figura 5-13` (a);

- O construtor do mensageiro deve chamar o construtor do `MessengerStrategy` passando um nome que identifique o algoritmo – `Figura 5-13` (b);

- O método `setup()` foi definido no intuito de realizar as operações de inicialização dos recursos necessários pelo algoritmo, tais como: realizar a troca das chaves públicas ou carregar a chave simétrica do arquivo de propriedades, por exemplo – `Figura 5-13` (c);

- O método `sendMessage()` realiza a operação de cifragem da mensagem – `Figura 5-13` (d) – (através de métodos auxiliares, de preferência) e chama o método utilitário `send()` para realizar o envio do objeto cifrado – `Figura 5-13` (e);

- O método `receiveMessage()` recebe o objeto cifrado utilizando o método auxiliar `receive()` – `Figura 5-13` (f), e executa a decifragem desse objeto recuperando a mensagem original – `Figura 5-13` (g).

```
public class AESMessenger extends MessengerStrategy {       // (a)
    // declarações
    int key;

    // Define o algoritmo no construtor.
    public AESMessenger(){
        super("AES");                                        // (b)
    }

    // Faz as configurações necessárias.
    public void setup(){
        this.key = HotspotResource.getInstance().            // (c)
                   getInt("aes.key");
    }

    // Envia mensagens cifradas pelo AES.
    public void sendMessage(Message msg){
        // Chama o método auxiliary encryptAES.
        byte[] cipher = this.encryptAES(msg);                // (d)

        // Envia o objeto cifrado
        this.send(cipher);                                   // (e)
    }

    // Recebe mensagens cifradas pelo AES.
    public Message receiveMessage(Socket socket){
        byte[] cipher = (byte[])this.receive(socket);        // (f)
        Message msg = this.decrypt(cipher);                  // (g)
        return msg;
    }

    // Implementação dos métodos encryptAES e decryptAES …
}
```

Figura 5-12 Implementação de um mensageiro baseado no algoritmo AES

5.3.4 Executando o MAFIDS

A instanciação do MAFIDS se dá através da implementação dos *hot-spots* definidos, provendo assim funcionalidades específicas de um domínio de aplicação. No entanto, após as devidas implementações, o *framework* deve ser executado unindo as implementações da aplicação ao seu *core,* formando um único sistema. A classe `MafidsRunner`, presente no pacote `com.mafids.control.core`, possibilita a realização dessa tarefa. Para tanto, o usuário deve conhecer os parâmetros necessários para a execução do MAFIDS. Ela recebe dois argumentos como parâmetro: o primeiro, *-cp,* é utilizado para informar ao MAFIDS o *classpath* da aplicação construída sobre ele; o segundo parâmetro, [-h|-m] é opcional e determina o modo de operação do MAFIDS; se este parâmetro for omitido, o valor default será *–h*. A Figura 5-13 apresenta o uso do `MafidsRunner` para a execução do MAFIDS, descrevendo os parâmetros utilizados.

```
Java –cp <mafids.jar path><;|:><app classpath>  com.mafids.control.core.MafidsRunner –cp
        <app classpath> [-h|-m]

Onde:
        <mafids.jar path>          Caminho onde o mafids.jar está localizado.

        <;|:>                      Separador Windows e Unix respectivamente.

        -cp <app classpath>        Define o classpath da aplicação.

        <-h|-m>                    Modos de operação: host (default) e monitor
                                   respectivamente.
```

Figura 5-13 Modo de uso do `MafidsRunner` e lista de parâmetros.

Alguma considerações a respeito da execução do `MafidsRunner`:

- Os arquivos de propriedades distribuídos juntamente com o *mafids.jar* devem estar na mesma pasta do arquivo '.jar'. O *path* o *mafids.jar* deve ser colocado no *classpath* para a execução do *framework.* O comando *–cp* do interpretador de *bytecodes* do Java (java.exe) define o *classpath* a ser utilizado para aquela execução (para mais informações a respeito dos modos de execução de aplicações JAVA, consulte [15]). Além do *classpath* do MAFIDS é necessário também colocar o *classpath* da aplicação (o mesmo que será passado ao MAFIDS mais adiante);

- O separador é utilizado para definir mais de um *path* onde as classes necessárias para a execução serão procuradas. Esse separador pode mudar de plataforma para plataforma. No Windows o símbolo ';' é utilizado para esta tarefa; já no Unix esse trabalho é feito com o ':';

- O mesmo *classpath* da aplicação informado para o interpretador java.exe deve ser informado também ao MAFIDS, para que o *framework* saiba onde encontrar as classes que implementam os *hot-spots* definidos por ele. Esse *classpath* é passado com o parâmetro *–cp* (a exemplo da nomenclatura utilizada pelo interpretador);

Capítulo 6

Conclusões e Trabalhos Futuros

Este trabalho procurou desenvolver um *framework* para o desenvolvimento de SDIs baseado em SMA que viabilizasse a implantação deste tipo de ferramenta para o mercado corporativo. Segundo a *PwC e CSO* [9] menos da metade das empresas entrevistadas utilizam, SDI como parte da infra-estrutura de SI. Acredita-se que as limitações inerentes das arquiteturas tradicionais de SDI tenham contribuído para a abstenção deste tipo de ferramenta, visto que a mesma pesquisa revelou que os SDIs constituem um dos melhores mecanismos de detecção de falhas de segurança [9]. A utilização de SMA para o desenvolvimento do MAFIDS permitiu que os objetivos deste trabalho fossem alcançados, dentro do escopo pretendido.

A validação do MAFIDS foi feita através de sua instanciação numa aplicação específica para a detecção de *backdoors,* a qual foi testada (em condições normais de trabalho) em um conjunto restrito de máquinas de uma empresa de TI do Porto Digital – demonstrando resultados satisfatórios para aquilo que a aplicação se propunha a fazer. No entanto, é sugerido como trabalhos futuros, a realização de testes de carga e *stress* para avaliação real do sistema em condições extremas de trabalho.

Apesar de o MAFIDS ter alcançado seu objetivo dentro do escopo definido, alguns elementos podem ser re-projetados para promover um maior reuso do *framework* e facilitar a sua própria atualização e/ou manutenção no futuro. Atualmente, o subsistema de comunicação não está definido como um *hot-spot*, o que significa que o usuário do MAFIDS não terá a liberdade de modificar esse componente. Além de não ser possível substituir o subsistema de comunicação, as alterações de forma a contemplar novos tipos de mensagens demandam certo esforço, visto que será necessário alterar várias classes como, por exemplo: a classe `Message` para adicionar o novo tipo desejado, a classe `MessageManager` para contemplar a manipulação desse novo tipo de mensagem definido, etc. Desta forma, pode-se dizer que o subsistema de comunicação, por ser um dos mais críticos, é o ponto de melhoria de maior prioridade na lista de trabalhos futuros.

6.1 Contribuições

A principal contribuição deste trabalho é a proposta de um *framework* reusável – MAFIDS – que possibilita o desenvolvimento de SDIs utilizando a tecnologia de SMAs. O reuso do MAFIDS permite a construção desse tipo de aplicação em um tempo reduzido, além de propagar a confiabilidade e robustez do código – advindas da aplicação de padrões de projeto – para os sistemas finais construídos sobre este *framework.*

Uma contribuição expressiva do modelo proposto é a facilidade de expansão do potencial de detecção do SDI desenvolvido devido à abordagem de se definir os programas agentes como *hot-spots* do *framework*, o que possibilita que mecanismos de defesa sejam incorporados ao sistema sem mesmo ser necessário que o MAFIDS seja reiniciado. Para tanto é necessário apenas que os programas disponíveis (implementados pelo SDI) sejam recarregados no *framework.*

O MAFIDS foi todo projetado de forma parametrizada utilizando arquivos de propriedades (XML) para armazenar as configurações desejadas. Foram definidos arquivos separados para as configurações do *core* do *framework* e as configurações dos *hot-spots.* Apesar de ser uma boa abordagem para promover o desacoplamento entre os componentes, melhorias podem ser feitas nesse aspecto como, por exemplo, implementar mecanismos na GUI que permitam a configuração dos arquivos de propriedades de forma mais amigável.

6.2 Trabalhos Futuros

A abordagem pela qual o *framework* foi desenvolvido permite que novas funcionalidades sejam adicionadas sem trazer impactos para o restante do *core.* A seguir estão listadas algumas características desejáveis ao *framework* (novas funcionalidades e melhorias) que, apesar de estarem fora do escopo deste projeto, podem servir como base para trabalhos futuros:

- Generalizar o protocolo de comunicação AMEP, construindo mecanismos que facilitem uma posterior atualização ou mudança do protocolo de comunicações do MAFIDS, propondo-o como mais um *hot-spot* do *framework*;

- Implementar componentes para permitir que a configuração dos arquivos de propriedades do MAFIDS seja realizada através da GUI;

- Expansão dos mecanismos de criptografia, implementando mais algumas classes que encapsulem o processo de cifragem/decifragem de mensagens utilizando os algoritmos mais conhecidos e utilizados no mercado, tais como o RSA – *Rivest, Shamir, Adleman*, PGP – *Pretty Good Privacy*, AES, DES – *Data Encryption Standard*, entre outros. Para mais informações acerca de algoritmos de criptografia, consulte [20] [27][31];

- Aplicar mais uma vez o padrão de projeto *Strategy* para delegar a analise de dados dos agentes inteligente para um motor de IA. Desta forma é possível modificar o algoritmo de análise sem trazer impactos para o resto do sistema. Essa estratégia se apresenta como mais um *hot-spot* do MAFIDS, permitindo que o usuário do *framework* define seus próprios algoritmos de IA;

- Realização de testes de carga e *stress* para avaliação real do comportamento do sistema em condições extremas de trabalho;

- Permitir que o modo de operação do SDI construído sobre o MAFIDS seja configurável. O modo de operação definiria o nível de notificações que o administrador iria receber. Uma sugestão para essa atividade seria a criação de um valor configurável (entre 0 e 1, por exemplo) através dos arquivos de propriedades e, possivelmente, através de controles na GUI. Detecções com a probabilidade de ser uma intrusão abaixo desse valor seriam desconsideradas;

- Uma outra atividade interessante de ser desenvolvida seria a criação de *scripts* de configuração de agentes, que seriam executados durante o *startup* da aplicação definindo quais agentes seriam criados e para quais *hosts* seriam enviados;

- Apesar de o MAFIDS ter sido desenvolvido utilizando mecanismos de internacionalização, apenas o idioma inglês foi implementado. Um ponto de melhoria seria a implementação de outros idiomas; uma sugestão interessante para promover o reuso de código seria utilizar um gerenciador de internacionalização, que definiria o idioma a ser utilizado (dentre os implementados) com base nas informações de localização (*Locale)*;

- Desenvolver a GUI dos *hosts*, tornando o *feedback* dos fatos mais intuitivo para o usuário como, por exemplo, quando o *host* não conseguir se comunicar com o monitor (provavelmente pelo segundo estar *offline*) a GUI do *host* apresentar algum ícone diferenciado indicando que aquele *host* pode estar em risco por não estar sendo coberto pelo monitor;

Bibliografia

[1] BERNARDES, M. C. *Avaliação do Uso de Agentes Autônomos Móveis em Segurança Computacional.* 1999. 119f. Dissertação (Mestrado em Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 1999.

[2] BURROUGHS, D.J. et al. *Analysis of Distributed Intrusion Detection Systems Using Bayesian Methods.* IPCCC, 2002.

[3] BUSCHMANN, F. et al. *A System of Patterns.*John Wiley and Sons, New York, 1996.

[4] CERIAS. AAFID – *Autonomous Agents For Intrusion Detection Research Group*. Department of Computer Sciences, Purdue University, 2001.

[5] CERT Advisory CA-96.21. *TCP SYN Flooding and IP Spoofing Attacks*.
Disponível em: <http://www.cert.org/advisories/CA-1996-21.html>.
Acesso em: 10 dez. 2005.

[6] CROSBIE, M. E SPAFFORD, E. H. *Active Defense of a Computer System Using Autonomous Agents*. Department of Computer Sciences, Purdue University, 1995.

[7] CROSBIE, M. E SPAFFORD, E. H. *Defending a Computer System Using Autonomous Agents*. Department of Computer Sciences, Purdue University, 1995.

[8] CSO magazine. "*The State of Information Security 2005*".
Disponível em: <http://www.csoonline.com/csoresearch/report93.html>.
Acesso em: 16 nov. 2005.

[9] CSO magazine. "*The Global State of Information Security 2006*".
Disponível em: < http://www.cio.com/archive/091506/security_survey.html?page=1 >.
Acesso em: 23 out. 2006.

[10] Eclipse Project.
Disponível em: <http://www.eclipse.org>.
Acesso em: 14 set. 2006.

[11] FAYAD, M. e SCHIMIDT, D. *Object-Oriented Application Framework*. Communications of the ACM, v.40 n.10, p.32-38, out. 1997.

[12] FRANKLIN, S. e GRAESSER, A. *Is It an Agent or Just a Program? A Taxonomy for Autonomous Agents.*In Proceeding of the 3rd Internation Workshop on Agent Theories, Achitectures and Languages. Springer-Verlag, 1996.

[13] GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1998.

[14] HEADY, R. E. A. *The Architecture of a Network Level Intrusion Detection System.* Department of Computer Science, University of New Mexico, 1990. Technical Report CS90-20.

[15] HORSTMANN, C. S. e CORNELL, G. Core Java 2: Volume 1, Fundamentos. São Paulo: Makron Books, 2001. 654 p.

[16] HORSTMANN, C. S. e CORNELL, G. Core Java 2: Volume 2, Recursos Avançados. São Paulo: Makron Books, 2002. 823 p.

[17] HUNT, C. *TCP/IP Network Administration.* 2ª Edição, O'REILLY, 1997. 630p.

[18] JOHNSON, R.E. *Reusing Object-Oriented Design.* University of Illinois, 1991. Technical Report UIUCDCS 91-1696.

[19] KUMAR, S.; SPAFFORD, E. H. *A Pattern Matching Model for Misuse Intrusion Detection.* In Proceedings of the 17th National Computer Security Conference, pages 11-21, 1994.

[20] KUROSE, J.F.; ROSS, K.W. Rede de Computadores e a Internet: uma nova abordagem. São Paulo: Addison Wesley, 2003. 548 p.

[21] LANGE, D.B. e OSHIMA, M. *Programming and Depolying Java Mobile Agents with Aglets.* Longman: Addison Wesley, 1998.

[22] MATTSSON, M. *Object-Oriented Framework: A survey of methodological issues.* In Proceeding of the Triennal IFAC World Congress, 1996.

[23] MUKHERJEE, B., HERBERLEIN, L.., E LEVITT, K. *Network Intrusion Detection.* IEEE Network; 8(3):26-41, Jun 1994.

[24] NIKITAS J. D., KIN F. L.. *High Performance Computing Systems and Applications,* Canada: Springer, 2001. 526p.

[25] PREE, W. e SIKORA, H. *Design Patterns for Object-Oriented Software Development.* In Proceedings of the 19th International Conference on Software Engineer, p.663-664, 1997.

[26] RUSSEL, S. J. e NORVIG, P. Inteligência Artificial. 2ª Edição, Rio de Janeiro: Campus, 2004. 1056 p.

[27] SOARES, L. F. G., LEMOS, G. e COLCHER, S. Redes de Computadores: Das LANs, MANs e WANs às Redes ATM. 2ª Edição, Rio de Janeiro: Campus, 1998. 740 p.

[28] SIERRA, K. e BATES, B. *Sun Certified Programmer for Java 5 Study Guide.* McGraw-Hill Osborne Media, 2005. 825 p.

[29] SOMMERVILLE, IAN. *Software Engineer*. 7ª Edição, Boston: Addison-Wesley, 2004. 667 p.

[30] STREBE, M. e Perkins, C. Firewalls. São Paulo: Makron Books, 2002. 442 p.

[31] TANENBAUM, A. S. *Computer Networks*. 4ª Edição, New Jersey: Prentice Hall, 2003. 891 p.

[32] PTACEK, T. H. e NEWSHAM, T. N. *Insertion, evasion and denial of service: Eluding network intrusion detection.* Technical Report, Secure Networks, Inc., 1998.

[33] ZAMBONI, D. et al. *An Architecture for Intrusion Detection using Autonomous Agents.* Department of Computer Sciences, Purdue University, 1998.