UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

TÓRGAN FLORES DE SIQUEIRA

Um Middleware Reflexivo para Apoiar o Desenvolvimento de Aplicações com Requisitos de Segurança

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Profa. Dra. Maria Lúcia Blanck Lisbôa
Orientadora

Porto Alegre, junho de 2004

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Siqueira, Tórgan Flores de

Um Middleware Reflexivo para Apoiar o Desenvolvimento de Aplicações com Requisitos de Segurança / Tórgan Flores de Siqueira. – Porto Alegre: PPGC da UFRGS, 2004.

67 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2004. Orientadora: Maria Lúcia Blanck Lisbôa.

1. Arquiteturas de *software*. 2. Padrões de projeto. 3. Reflexão computacional. 4. Programação orientada a aspectos. 5. *MIDDLEWARE*. 6. Segurança. 7. Java. I. Lisbôa, Maria Lúcia Blanck. II. Título.

“Dedicado à memória de Olida e Ervino Tag”

AGRADECIMENTOS

Agradeço à Maria Lúcia, minha orientadora, não só pelo conhecimento e apoio acadêmico, mas principalmente pela dedicação, paciência e empenho durante estes dois anos. Sua ajuda foi fundamental para que eu concluisse o mestrado, e seu incentivo, para que eu mantivesse vivo o interesse pela carreira acadêmica.

À professora Ingrid Porto, outrora minha orientadora de iniciação científica, por permitir que eu continuasse usando o laboratório-convênio e, além disso, permitir que eu o usasse nos horários fora do expediente.

Ao Luís Otávio, chefe dos laboratórios, sempre bem disposto, pelo auxílio quando precisava de equipamentos, além, é claro, das concessões para uso do laboratório fora de hora.

Aos professores com os quais tive aulas, assim como aqueles com quem não tive, pois formam, junto com os técnicos, a grande equipe, e de excelente nível, que é o Instituto de Informática. E também ao pessoal da portaria dos laboratórios e os guardas, sempre atenciosos e prestativos.

À Universidade Federal, que mesmo passando por tantas dificuldades, consegue manter a qualidade de ensino e pesquisa que a distingue.

À minha família, que entendeu mais este período de ausências e poucas visitas, me apoiando e incentivando a concluir esta etapa. Em especial, ao meu pai, minha irmã e minha vó, que tanto sentem minha falta em casa, mas não medem esforços para que eu siga minha carreira na academia.

Aos meus amigos da CEFAV, a Casa de Estudantes que me possibilitou além da moradia, bons momentos de convivência durante a graduação e o mestrado.

Finalmente, agradeço a CAPES, pela concessão da bolsa de mestrado que me permitiu alçar mais este degrau.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ACL	*Access Control List* - Lista de Controle de Acesso
CORBA	*Common Object Request Broker Architecture*
FC	Filtros de Composição
FTP	*File Transfer Protocol* - Protocolo de Transferência de Arquivos
HTTP	*HyperText Transfer Protocol* - Protocolo de Transferência de Hypertexto
JAAS	*Java Authentication and Authorization Service* - Serviço de Autenticação e Autorização Java
JCE	*Java Cryptography Extension* - Extensão de Criptografia Java
JSSE	*Java Secure Socket Extension* - Extensão de *Socket* Seguro Java
JVM	*Java Virtual Machine* - Máquina Virtual Java
OO	Orientação a Objetos
PAM	*Pluggable Authentication Modules* - Módulos de Autenticação Conectáveis
PMO	Protocolo de Meta-Objetos
POA	Programação Orientada a Aspectos
POO	Programação Orientada a Objetos
PP	Padrões de Projeto
RC	Reflexão Computacional
RMI	*Remote Method Invocation* - Chamada Remota de Métodos
SO	Sistema Operacional
TF	Tolerância a Falhas
URL	*Uniform Resource Locator* - Localizador Uniforme de Recursos
XML	*eXtended Markup Language*

LISTA DE FIGURAS

RESUMO

Muitos aplicativos atuais, envolvendo diversos domínios de conhecimento, são estruturados como arquiteturas de *software* que incorporam, além dos requisitos funcionais, requisitos não funcionais, como segurança, por exemplo. Tais requisitos podem constituir um domínio próprio, e, portanto, serem comuns a várias outras arquiteturas de *software*. Tecnologias como Programação Orientada a Aspectos, Reflexão Computacional e Padrões de Projeto colaboram no desenvolvimento de arquiteturas que provêem a separação de requisitos não funcionais. Porém, sua experimentação e adoção no domínio da segurança computacional ainda é incipiente. O foco deste trabalho é a elaboração de um padrão de projeto voltado à segurança, utilizando como arquitetura conceitual programação orientada a aspectos, e como arquitetura de implementação, reflexão computacional. A composição destas tecnologias resulta em um *middleware* orientado à segurança, voltado a aplicações desenvolvidas em Java. Estuda-se as tecnologias, seus relacionamentos com a área de segurança, seguido da proposta de uma arquitetura de referência, a partir da qual é extraído um protótipo do *middleware* de segurança. Este, por sua vez, provê mecanismos de segurança tão transparentes quanto possível para as aplicações que suporta.

Com o objetivo de realizar a implementação do middleware de segurança, também são estudadas os mecanismos de segurança da plataforma Java, porém limitado ao escopo deste trabalho. Segue-se o estudo da base conceitual das tecnologias de Reflexão Computacional, o modelo de implementação, seguido de Programação Orientada a Aspectos, o modelo conceitual, e, por fim, têm-se os Padrões de Projeto, a arquitetura de referência.

Integrando as três tecnologias apresentadas, propõe-se um modelo, que estabelece a composição de um Padrão *Proxy*, estruturado de acordo com a arquitetura reflexiva. Este modelo de arquitetura objetiva implementar o aspecto de segurança de acesso a componentes Java, de forma não intrusiva,.

Baseado no modelo, descreve-se a implementação dos diversos elementos do *middleware*, estruturados de forma a ilustrar os conceitos propostos.

Ao final, apresenta-se resultados obtidos durante a elaboração deste trabalho, bem como críticas e sugestões de trabalhos futuros.

Palavras-chave: Arquiteturas de *software*, padrões de projeto, reflexão computacional, programação orientada a aspectos, *middleware*, segurança, Java.

A Reflective Middleware to Support the Development of Applications with Security Concerns

ABSTRACT

Nowadays many applications involving diverse knowledge domains are structured according to software architectures that include both functional and non-functional requirements, as security, for example. Such requirements may constitute a separate knowledge domain, and, therefore, being common to many other software architectures. Technologies such as Aspect-Oriented Programming, Computational Reflection and Design Patterns can help in the development of architectures that provide the separation of non-functional requirements. But only recently some practical work and applications involving together such technologies began to appear. This work focuses on the elaboration of a design pattern with security as the target, using aspect-oriented programming as the conceptual architecture and computational reflection as the implementation architecture. The composition of these technologies results in a security-oriented middleware, which aims to support Java applications. First a survey about those technologies and their relationship with the security area is presented. Follows a proposal of a reference architecture that serves to structure a prototype of the security middleware. This middleware provides security mechanisms as transparently as possible to the applications it supports.

In order to implement the prototype of the security middleware, a summary of security mechanisms of the Java platform is also presented, but limited by the scope of our objectives. Following that, we summarize the conceptual bases of the tree core technologies adopted in this work: Computational Reflection, which serves as the implementation model, Aspect-Oriented Programming, which guides us throughout our conceptual model, and Design Patterns, that serves as the reference architecture.

Integrating these three technologies, we propose a model, which composes a Proxy Pattern, structured according to the reflective architecture. This model intends to implement the access security aspect to Java components in a non-intrusive way.

Following this model, we describe the implementation of some of our middleware elements, in order to illustrate the proposed concepts.

Finally, the results are summarized, along with critics and future work suggestions.

Keywords: design patterns, computational reflection, aspect-oriented programming, middleware, security, Java.

1 INTRODUÇÃO

1.1 Motivação

Tal como é desenvolvida hoje, uma aplicação pode envolver diversos domínios de conhecimento. Por vezes, alguns destes domínios não constituem requisitos funcionais da aplicação, mas sim, requisitos não funcionais, como por exemplo, operações atômicas, sincronização, segurança etc.

Destes requisitos não funcionais, a exemplo de segurança, alguns se constituem em um domínio próprio, que atuam sobre aplicações de diferentes domínios. A diversidade entre requisitos funcionais e não funcionais sugere que estes domínios sejam separados durante o desenvolvimento da aplicação, objetivando a reutilização de componentes, o aumento da confiabilidade e agilizar o processo de desenvolvimento de aplicações. A separação explícita de requisitos, ao invés de misturá-los estreitamente no código, aumenta também a eficiência na gerência da complexidade (MOHAMMAD, 2002).

Tecnologias como Programação Orientada a Aspectos (POA), Padrões de Projeto (PP) e Reflexão Computacional (RC) podem ajudar a alcançar os objetivos da separação de requisitos. Estas tecnologias apresentam características que facilitam a adaptabilidade, a reutilização de código e soluções, a redução da complexidade e a extensão de mecanismos típicos de Programação Orientada a Objetos (POO). As características citadas fazem parte de um conjunto maior de necessidades que os sistemas computacionais demandam, aumentadas pela expansão das tecnologias de informação. POA provê os meios de separação de requisitos (DIAZ; CAMPO, 2001), enquanto que RC tem sido explorada como uma elegante opção no desenvolvimento de *software* tolerante a falhas (NICOMETTE, J.-C. et al., 1995), (XU; RANDELL; RUBIRA, 1994) e (LISBôA, 1997). PP, por sua vez, ajudam a diminuir a complexidade da aplicação, através de uma clara separação entre estrutura, dinâmica, comportamento e contexto de uma aplicação (GAMMA, E. et al., 1994).

Concomitante ao crescimento, tanto em abrangência quanto em complexidade, dos sistemas computacionais, observado principalmente ao longo das duas últimas décadas, a área de Tolerância a Falhas (TF) também expandiu-se em termos de alcance e aplicabilidade. A diminuição dos custos, até então proibitivos para sistemas que não fossem críticos ou dedicados, permitiu a inserção dos resultados de estudos na área em sistemas que atingem um público mais amplo.

A informatização e automação de processos outrora manuais foi outro fator que impeliu o crescimento de TF, pois de muitos destes processos se exige uma alta dependabilidade e, de acordo com Laprie (LAPRIE, 1998), é neste cenário que são encontradas características de (alta) disponibilidade, confiabilidade, segurança, confidencialidade, integridade, manutenibilidade, etc.

Segurança computacional, vista como uma das sub-áreas de TF, não é um assunto novo. A segurança, tal como é tratada ao longo do texto, é definida por Laprie como sendo a junção de três outras propriedades de TF: integridade, disponibilidade e confidencialidade. É oportuno notar que Laprie distingue entre *safety*, a não ocorrência de danos ao ambiente, de *security*, a noção comum propriamente dita de segurança computacional. Ambos os termos são comumente traduzidos como segurança na língua portuguesa.

A necessidade de proteger os sistemas de usuários não-autorizados tem sido alvo de estudo de administradores e desenvolvedores de sistemas antes mesmo da popularização da Internet. O controle de acesso aos primeiros sistemas de tempo compartilhado era baseado em políticas de restrição e senhas, a fim de confirmar a identidade do usuário pretendendo acesso. Conforme os serviços prestados via rede foram se tornando lugar-comum na vida dos usuários, segurança se tornou uma característica fundamental para protegê-los das adversidades que acompanharam o crescimento da rede, tais como vírus, cavalos-de-tróia, invasão de privacidade, negação de serviço e outros. Assim, foi necessário que os esquemas de autenticação baseados em senhas evoluissem, assim como os mecanismos de autorização baseados em propriedades de arquivos, a exemplo dos bits de controle encontrados nos sistemas Unix. Exemplos desta evolução são o sistema de autenticação *Kerberos* e as listas de controles de acesso (*ACL - Access Control Lists*) dos sistemas de arquivos recentes.

Neste cenário, encontram-se aplicações legadas e aplicações em desenvolvimento que não têm segurança como um requisito funcional, mas latente como um requisito não funcional. E por ser latente, muitas vezes é relegada a segundo plano, introduzindo problemas de segurança posteriores.

As aplicações tolerantes a falhas não diferem das demais quando vistas à luz de problemas como aumento de complexidade, dificuldade de manutenção, envelhecimento, adaptabilidade etc. Assim, era esperado que as tecnologias de POA, PP e RC viessem em auxílio de TF, situação que apenas recentemente começa a se verificar com mais intensidade. Contribui para isso o fato de que são tecnologias recentes, principalmente POA, e que ainda não estão muito integradas. Em alguns trabalhos é possível encontrar relacionadas as técnicas de tolerância a falhas e reflexão computacional, (LISBôA, 1997) e (BUSATO; RUBIRA; LISBôA, 1997), com arquiteturas de meta-nível, (NICOMETTE, J.-C. et al., 1995), com replicação, (KICZALES, 1991), com persistência, (OKAMURA; ISHIKAWA, 1994), com controle de localização de objetos, (STROUD, 1992), com distribuição. Wu (WU; SCHWIDERSKI, 1997) relaciona RC diretamente com Java, e Welch (WELCH; STROUD, 2001) relaciona RC e Java utilizando a técnica de reescrita de *bytecodes*, assim como (WELCH; STROUD; ROMANOVSKY, 2001) relaciona reflexão e exceções em meta-nível. Por sua vez, RC pode ser relacionada a padrões, e alguns trabalhos, por exemplo (FERREIRA; RUBIRA, 1998), relacionam estas duas metodologias, mas sem associação com a POA. Em (WELCH; STROUD, 2002), há relação de RC com segurança em código compilado, enquanto que em (WELCH; STROUD, 2000a), a relação de RC é com código móvel, em Java. Na área de segurança, há trabalhos recentes que envolvem POA, a exemplo de (VANHAUTE; WIN; DECKER, 2001), que explora uma ferramenta (AspectJ) de POA, (WIN; VANHAUTE; DECKER, 2001), (WIN; VANHAUTE; DECKER, 2002) e (WIN, B. D. et al., 2002), mais voltados à exploração da tecnologia e (WIN; JOOSEN; PIESSENS, 2003), com uma perspectiva de análise de resultados obtidos a partir do uso de POA. Com vistas a modelos de segurança aplicáveis em Java, pode-se referir (WELCH; STROUD, 1999), por exemplo.

.

1.2 Objetivos e Escopo

Assim, surge a proposta deste trabalho, que é incorporar características destas tecnologias na área de Tolerância a Falhas, mais especificamente, na sub-área de segurança computacional. Pretende-se extrair de cada tecnologia as características relevantes, e compor este resultado num *middleware* capaz de prover e adaptar-se aos diferentes requisitos de segurança de aplicações (EMMERICH, 2000a), englobando, primariamente, serviços de autenticação e autorização.

O escopo é definido, em parte, pela plataforma escolhida: Java.

Contribuiram para sua escolha a disponibilidade, a facilidade de programação, relativa independência de plataforma, bibliotecas padrão de reflexão e criptografia, carregadores de classe com verificação de código (*bytecode verifier*), literatura e documentação básica apropriados e também por já estar sendo usada no grupo de pesquisas em Tolerância a Falhas do Instituto de Informática, mas sobretudo, por ser uma plataforma (linguagem e máquina virtual) que tem segurança como item considerado desde a concepção.

1.3 Composição do *Middleware*

1.3.1 Programação Orientada a Aspectos

A Programação Orientada a Aspectos contribui como modelo de referência, pois a visão das propriedades de segurança aqui tratadas é a de requisito não-funcional, ao que POA se adapta muito bem. POA surgiu como uma evolução do modelo de objetos, em resposta à complexidade que aumentou dentro daquele próprio modelo. No contexto deste trabalho, serviços de autenticação e autorização são considerados como aspectos que podem ser separados na implementação, mas que, ao se associarem a uma aplicação, se entrelaçam ao código da aplicação através de um ponto de junção. Ao estruturar o *middleware* é considerado que a aplicação, ou conjunto delas, deva ser varrida transversalmente em busca de requisitos de segurança. Percorrer transversalmente a aplicação significa que todos os requisitos não-funcionais têm de ser encontrados e estruturados com algum grau de ortogonalidade em relação à hierarquia de classes original.

POA não é aplicada diretamente a este trabalho, no sentido de se estar utilizando ferramentas de apoio ao desenvolvimento orientadas a aspectos, tais como o AspectJ (ECLIPSE Project. AspectJ, 2003) ou DJlib (DEMETER Project. DJ Library, 2002), mas serve bem ao propósito de modelo conceitual, pois pretende-se que o protótipo desenvolvido possa, em etapa posterior, ser adaptado às ferramentas que utilizem esta tecnologia. Contudo, uma das técnicas de Aspectos, o modelo de Filtros de Composição, descrito na sub-seção 2.3.1, aproxima o modelo conceitual do *middleware* proposto com o modelo de POA de forma mais concreta.

1.3.2 Padrões de Projeto

Padrões de Projeto são utilizados como base arquitetural do *middleware*, fornecendo os blocos básicos sobre os quais se compõe o novo padrão de segurança pretendido. Padrões oferecem soluções genéricas, adaptáveis a contextos particulares de maneira modular e incremental, tornando-se uma arquitetura de referência bastante adequada para esta proposta. Dentre os padrões de interesse, dois se destacam: *Proxy* e *Bodyguard*, sendo que o primeiro tem especializações, dentre as quais *Remote*, *Protection* e *Firewall*, de maior relevância ao assunto.

1.3.3 Reflexão Computacional

Como forma de adequar a arquitetura de referência, será usada Reflexão Computacional como arquitetura e técnica de implementação. Esta tecnologia, se vista individualmente, também poderia ser considerada como arquitetura de referência, a exemplo de PP.

É importante notar que o Padrão de Projeto *Proxy* é implementado por meio de reflexão computacional, assim como os serviços do *middleware* Java/RMI também utilizam esta técnica para obter maior flexibilidade de instanciação e minimizar as informações a serem supridas pelo desenvolvedor.

1.3.4 Ambiente de Prototipação

Como ambiente de prototipação, escolheu-se a plataforma Java. Esta escolha é fundamentada em dois pontos:

1. Infraestrutura adequada - Java dispõe dos mecanismos necessários para implementar reflexão computacional e, nas versões mais recentes, serviços de segurança além dos intrínsecos à linguagem.

2. O objetivo para compor *middleware* orientado à segurança é atingir aplicações legadas em Java, e ao fazê-lo, prever que tal solução também pode ser adotada por aplicações em desenvolvimento, desonerando os desenvolvedores de entrar em detalhes não funcionais do aplicativo.

A concepção da plataforma Java previu alguns mecanismos de segurança desde os primeiros estágios, resultando em um ambiente e uma linguagem bastante robustos se comparados aos que lhe precederam. O compilador se encarrega de verificar muitas das características em tempo de compilação, praticamente eliminando a possibilidade de código mal-formado. O ambiente de execução, isto é, a máquina virtual (JVM - *Java Virtual Machine*) partilha e suporta a concepção de segurança. A JVM faz a verificação em tempo de carga e execução do código dos aplicativos. Também é responsabilidade dela a coleta de lixo (desalocação de memória), reduzindo problemas como exaustão de memória e recursos. Ainda, a partir das versões recentes (Java2), é disponibilizada uma infraestrutura de segurança maior, abrangendo criptografia, autenticação e serviços de chaves e assinaturas digitais.

1.4 Organização do Texto

O texto está organizado como segue. O capítulo 2 inicia com um breve estudo sobre segurança, limitado ao escopo da plataforma Java e aplicado tão somente ao modelo de objetos. Após, é conduzido o estudo de Programação Orientada a Aspectos, Reflexão Computacional e Padrões de Projeto.

O capítulo 3 propõe um modelo que integra as três metodologias estudadas. Este modelo estabelece a composição de um Padrão *Proxy*, estruturado sobre Reflexão, que implementa o aspecto de segurança de acesso a componentes (objetos) Java.

O capítulo 4 mostra a implementação dos diversos elementos do *middleware*, estruturados de forma a ilustrar os conceitos da proposta.

Finalmente, o capítulo 5 discute os resultados obtidos, melhorias e trabalhos futuros. Atenção é dada quanto à eficácia e eficiência na aplicação de cada metodologia a cada uma das propriedades que compõe a propriedade de segurança.

2 BASE CONCEITUAL E TECNOLÓGICA

Neste capítulo são apresentadas, brevemente, as bases conceituais das tecnologias envolvidas. Inicia-se com os aspectos de segurança considerados, juntamente com o modelo de segurança Java, por estarem estreitamente relacionados neste trabalho. Após, apresenta-se breve consideração sobre Reflexão Computacional, o modelo de implementação, seguido de Programação Orientada a Aspectos, o modelo conceitual. Por fim, têm-se os Padrões de Projeto, a arquitetura de referência.

2.1 Aspectos de Segurança

A segurança de computadores, tal como é vista hoje, evoluiu das diretivas aplicadas pelos administradores de redes às suas máquinas para modelos mais complexos. Tais modelos incluem criação, verificação (restrição ou extensão) e revogação dinâmica de usuários e seus direitos e privilégios sobre um sistema ou parte dele.

Hoje, é possível reconhecer extensões desta área em várias outras, onde, primariamente, ela não é o objetivo principal. Os exemplos incluem desde os grandes bancos de dados corporativos até os programas de criptografia transparentemente embutidos no correio eletrônico de usuários domésticos.

Segundo Laprie (LAPRIE, 1998), a segurança pode ser definida em termos de três outras propriedades de sistemas tolerantes a falhas: integridade, disponibilidade e confidencialidade. Tomados em conjunto, a aplicação destes requisitos sobre um sistema pode demandar recursos além dos mecanismos oferecidos pela maioria dos sistemas operacionais (SO).

Problemas típicos envolvidos são a autenticação de usuários, validação de clientes e verificação da integridade dos dados e dos objetos servidores. Outra categoria de problemas diz respeito ao sistema operacional (SO): quais os serviços de segurança oferecidos? É possível autenticar os clientes? Há meios adequados para que se possa verificar (e garantir) a autenticidade do servidor? Além disso, há o problema de nem sempre ser possível modificar o SO, ou, fazê-lo traz impactos indesejáveis sobre o desempenho e a facilidade de uso.

Muitos aplicativos são desenvolvidos confiando-se no suporte do SO, desta forma, atribuindo importância menor às questões de segurança. A impossibilidade de modificá-los impõe novo problema. A disponibilidade do código-fonte não garante que este possa ser modificado, como no caso dos componentes já estarem em uso e demandarem compatibilidade entre versões. A solução consiste em criar uma camada de *software* intermediária.

Entre os dois níveis, a camada de *software* intermediário - *middleware*, funciona como elo de ligação. Dentre as suas atribuições, inclui-se suprir as deficiências do SO e dos

objetos, aplicando aspectos de segurança dos níveis mais baixos aos mais altos de forma gradual, na tentativa de impor o mínimo impacto ao desempenho e à facilidade de uso. O que se pretende é que o SO e os aplicativos não percebam este nível extra de *software*.

Particularmente, dois aspectos de segurança são abordados neste trabalho: autenticação e autorização, que constituem a base do controle de acesso. São aspectos muito próximos e que por vezes requerem colaboração um do outro.

Autenticação é o mecanismo de segurança que visa a reconhecer um usuário perante um sistema. Tipicamente, o sistema apresenta um desafio ao usuário para que este prove sua identidade, ou a identidade que alega ter. Os desafios, em sua grande maioria, compreendem a requisição de uma senha, palavra ou fase, e que é verificada contra os registros que o sistema têm deste usuário. Reconhecido o usuário, o sistema prossegue e permite o acesso. Caso contrário, bloqueia e/ou apresenta novo desafio ao usuário.

Enquanto a autenticação permite ou nega o uso do sistema por parte do usuário, o nível de autorização controla quais os serviços ou recursos este usuário pode utilizar. De modo geral, uma vez autenticado, o sistema atribui credenciais (identificadores únicos) aos usuários, e toda vez que o usuário solicita um serviço ou tenta acessar um recurso protegido, suas credenciais são verificadas e seu acesso é permitido ou negado conforme possa ou não acessá-lo. O controle de acesso no mecanismo de autorização pode alcançar vários níveis de complexidade, tal como o controle de acesso a bancos de dados, que estende o conceito de simples acesso a arquivos, ou os controles impostos pela máquina virtual Java aos programas que executa, que estendem não só controle de acesso a arquivos, mas o de acesso a conexões de rede, recursos de memória e processador, por exemplo.

2.1.1 O Modelo de Segurança Java

Em Java, o modelo de segurança compreende várias interpretações, isto é, vários contextos em que pode a segurança pode ser interpretada (OAKS, 1998), (SUN, 2003), (GONG; ELLISON; DAGEFORDE, 1998):

- Segurança contra programas malévolos: neste contexto, segurança refere-se à impossibilidade de programas mal-intencionados interferirem no ambiente de computação do usuário. Inclui os cavalos-de-tróia.
- Não-intrusão: aqui, pretende-se que programas não sejam capazes de descobrir informações confidenciais do usuário.
- Autenticação: partes de programas envolvidos num sistema devem ser autenticadas.
- Cifragem: dados sensíveis devem ser transportados de forma codificada.
- Auditoria: operações sensíveis devem ser sempre auditadas, isto é, registradas para posterior análise, se necessário.
- Bem-definidos: descrições, ou políticas de segurança corretamente especificadas devem ser seguidas.
- Verificação: regras de operação devem ser sempre estabelecidas e seguidas.
- Bem-comportados: programas devem ser impedidos de consumir recursos do sistema em demasia, ou mesmo tomar conta deles.

- Certificações: programas podem ter certificações de órgãos competentes por incluirem certos procedimentos de segurança.

Até a plataforma Java atual (versão 1.4.2), somente os quatro primeiros itens estão disponíveis como recursos de utilização direta do ambiente. É encargo do desenvolvedor prover auditoria, mecanismos de aplicação de políticas de segurança, verificação e mecanismos de controle de recursos do sistema. Derivado da própria licença de uso da plataforma Java, implementações de programas que possam ser certificados ainda não é uma realidade nesta plataforma.

Segundo Oaks (OAKS, 1998), as noções de segurança da plataforma são derivadas do modelo distribuído de Java, e das implicações decorrentes, tais como a possibilidade de carga dinâmica de classes via rede, por exemplo A aceitação de Java multiplicou as possibilidades de disseminação de vírus e cavalos-de-tróia, principalmente no momento em que se passaram a utilizar as facilidade de carregar e descarregar códigos via rede, sob demanda, referido genéricamente como distribuição de *software*.

McGraw (MCGRAW; FELTEN, 1996) aborda especificamente a questão dos *applets* (programas que são carregados e executados em navegadores), talvez a parte mais suscetível a problemas de segurança na plataforma Java. O modelo Java de computação ubíqua e mobilidade de código direcionou as estruturas do modelo de segurança, de modo que toda a plataforma reflete esta preocupação.

Entretanto, é necessário observar que a assertiva de que Java é seguro por concepção nem sempre é verdadeira. É bem verdade que os *applets* receberam atenção especial, e que os navegadores implementam políticas de segurança rígidas quanto a código distribuído (recebido via rede). Mas no outro extremo residem as aplicações locais, que usam a política menos restrita da plataforma, e para as quais se deixou ao encargo dos desenvolvedores a adoção de políticas específicas. Do ponto de vista arquitetural, um navegador e uma aplicação com vistas à segurança nada mais são do que extensões de classes-base da plataforma: carregadores de classe, gerenciadores de segurança, controladores de acesso, etc. A diferença está no papel atribuído a cada classe nas duas situações, de onde se infere que a natureza segura de Java, desconsiderado o aspecto da linguagem, depende da utilização feita dos recursos da plataforma.

O modelo Java de segurança, no contexto aqui apresentado, deve ser entendido como segurança de aplicações do usuário contra programas (aplicações e *applets*) malévolos, e não contra usuários mal-intencionados.

A segurança contra usuários mal-intencionados é um problema maior, e depende de outros fatores, como a segurança da máquina, sistema operacional, rede e seus protocolos. Java não tenta resolver problemas nestes níveis, mas apenas no nível de aplicativos.

A figura 2.1 ilustra o modelo de segurança da plataforma Java com respeito às aplicações, sejam elas programas ou *applets*.

Os retângulos com cantos arrendondados representam a infra-estrutura geral da aplicação, e que não necessariamente abordam aspectos de segurança. Os retângulos normais são os componentes que podem ser encontrados nos ambientes protegidos (*sandboxes*). Estes são os elementos aos quais se dedica maior estudo no contexto de segurança, pois é a partir deles que se dá o desenvolvimento de componentes (e ambientes) seguros.

Os componentes que um ambiente seguro pode apresentar são:

- Verificador de *bytecode*: garante que as classes estão em conformidade com a especificação da linguagem Java e com as restrições de segurança impostas a *applets*.

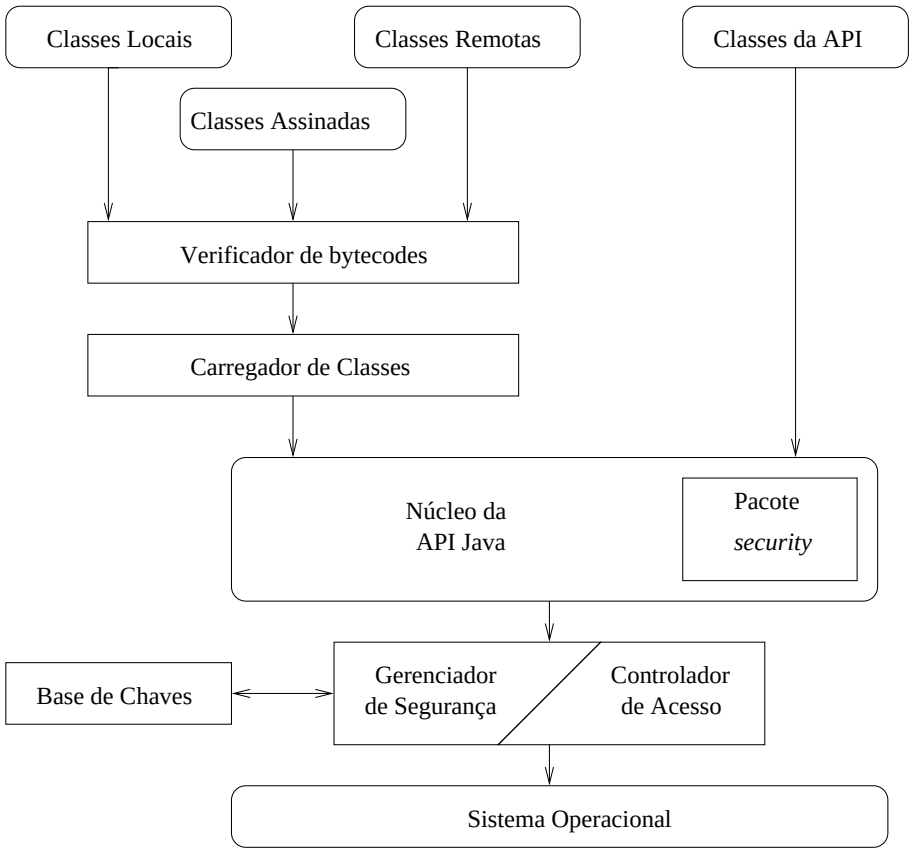

Figura 2.1: Modelo de segurança para uma aplicação Java.

- Carregador de Classes: o carregador de classes, que pode não ser único. Carrega as classes que estão[1] e as que não estão no caminho **CLASSPATH**.

- Controlador de Acesso: a partir da versão 1.2, o Controlador de Acesso controla os acessos da API de Java ao sistema operacional.

- Gerenciador de Segurança: é a interface primária entre a API Java e o SO.

- Pacote *security*: mais especificamente, *java.security*. Este pacote é a base para autenticação de classes assinadas. Particularmente interessantes são as facilidades para a interface de segurança, *message digests*, chaves e certificados, assinaturas digitais e encriptação [2].

- Base de chaves: banco de dados que contém as chaves usadas pelo gerenciador de segurança e controlador de acesso para verificar as assinaturas das classes assinadas. Pode ser um arquivo externo.

Com respeito ao sistema operacional, uma aplicação Java não pode aumentar seus privilégios, mas apenas restringí-los. A política de segurança vigente, durante a execução, pode ser vista como a intersecção entre as políticas de segurança da aplicação e do sistema operacional.

[1] Somente a partir da versão 1.2
[2] Pacote opcional

2.1.2 Aspectos de Segurança em uma Aplicação Java

Uma aplicação Java, ou mesmo um componente de um sistema maior, pode ter aspectos de segurança nos seguintes níveis:

- Construções da linguagem, cuja função principal é proteger os recursos da máquina, principalmente memória. A mais simples forma de proteção são os atributos de nível de acesso aos objetos, *private*, *default*, *protected* e *public*. Os métodos de acesso são obrigados a obedecer estritamente aos atributos. A serialização de objetos é uma exceção, mas pode ser controlada pelo desenvolvedor. Seguindo, acessos a posições arbitrárias de memória são proibidos, sendo estendido às promoções e conversões de objetos. Entidades *final* não podem ser modificadas, são imutáveis após a inicialização. Variáveis nunca podem ser usadas antes de serem inicializadas, e se não o forem explicitamente, uma inicialização padrão é oferecida. Limites de *arrays* são obedecidos estritamente, não podendo ser ultrapassados.

- Compilador, que verifica todas as regras acima, exceto verificação de limites de *arrays* e promoções, embora possa detectar as de tipos não relacionados.

- Verificador de *bytecodes*, parte integrante da máquina virtual Java (JVM). Não há maneira de interagir com ele, quer no nível de usuário, quer no nível de programador. Sua função é carregar o código intermediário (*bytecodes*) e construir os objetos de classes. O verificador é um mini-provador de teoremas que pode provar:

  1. Correto formato do arquivo *.class*.
  2. Não derivação de classes finais e não sobrescrita de métodos finais.
  3. Cada classe, exceto *java.lang.Object*, tem somente uma superclasse.
  4. Não ocorrência de conversões ilegais de dados de tipos primitivos.
  5. Não ocorrência de conversões ilegais de tipos de objetos. Antes de cada tentativa de conversões, a legalidade é testada.
  6. Não ocorrência de estouro de pilha.

- Máquina Virtual Java (JVM - *Java Virtual Machine*), que assegura verificações em tempo de execução. Estas verificações são as de limites de *arrays* e conversões de objetos.

- Carregadores de Classes (*Class Loaders*), responsáveis por ler código intermediário para a JVM e convertê-lo em definições de classes. Este componente é importante no modelo de segurança, pois é o único que conhece determinadas informações sobre as classes, tais como origem e assinatura. Normalmente, trabalha junto com o gerenciador de segurança e o controlador de acesso, nos ambientes protegidos. Essencialmente, este componente coordena atividades entre o gerenciador de segurança e o controlador de acesso e, principalmente, visa garantir obediência estrita a certas regras de ambientes de nomes (*namespaces*), que, se não forem seguidas, abrem sérios problemas de segurança nas aplicações.

- Gerenciador de Segurança, responsável por determinar parâmetros para o ambiente seguro Java. Ele é quem determina quais operações são ou não permitidas. Geralmente aplicações não têm um gerenciador padrão, enquanto que *applets* têm um

muito restrito. Este componente integra a API Java e os componentes seguros do desenvolvedor, através da classe *java.lang.SecurityManager*. A classe gerenciadora de segurança tem métodos voltados à segurança em si, isto é, que através de reflexão podem prover segurança de acesso à própria classe.

- Controlador de Acesso, que de fato executa as regras de segurança determinadas pelo Gerenciador. Sua inserção no modelo de segurança, a partir da versão 1.2, tem razões históricas, e sua presença é redundante, embora tenha flexibilizado em muito a aplicação de políticas. Deve ser observado que a API Java não chama o Controlador diretamente a não ser que haja um Gerenciador ativo. Um Controlador nunca é ativado enquanto não for chamado. Este componente é fundamental na medida em que protege os recursos do sistema, e permite a customização de políticas de segurança. É construído sobre os blocos fundamentais (classes): *CodeSource*, *Permissions*, *Policy* e *Protection Domains*.

- *Guarded Objects*, que encapsula em um objeto a noção de permissões e Controladores de Acesso. Tipicamente, o desenvolvedor encapsula um objeto qualquer dentro de um objeto *GuardedObject*, de modo que todos os acessos àquele objeto passam pelo seu *guard*, geralmente o próprio Controlador de Acesso.

Adicionalmente, a plataforma Java provê criptografia, que pode ser usada para esquemas de assinaturas de classes, autenticação de autoria e de dados. Também oferece mecanismos de criptografia como *message digests*, chaves, assinaturas digitais e criptografia prática, esta última através da JCE - *Java Cryptographic Extension* e JSSE - *Java Secure Sockets Extensios*, e as demais, através do pacote *security*.

2.2 Reflexão Computacional

Reflexão computacional (RC) é a atividade que um sistema faz sobre si mesmo, de forma independente das atividades realizadas para resolver o problema para o qual foi concebido (LISBôA, 1995). Esta atividade sobre si tem por objetivo obter informações sobre o seu estado (computações), com vistas a melhorar o desempenho, adicionar capacidades (funcionalidades) ou resolver problemas com possibilidade de escolher a melhor forma. É também uma técnica de implementação que diz respeito às arquiteturas de *software* nas quais é possível que componentes possam obter informações sobre propriedades internas de outros componentes (LISBôA, 1997). Uma arquitetura reflexiva permite recuperar informações sobre seu estado, sua estrutura e seu comportamento, assim como permite modificar e adaptá-las.

Esta seção apresenta, na forma de um breve resumo, as principais características de Reflexão Computacional. Primeiramente, pretende-se mostrar a RC como um mecanismo de computação em meta-nível, com respeito às aplicações. A seguir, RC é apresentada no domínio de Tolerância a Falhas, enfatizando-a como um meta-nível que implementa requisitos não funcionais das aplicações. RC é utilizada, neste trabalho, como técnica de implementação.

2.2.1 Arquiteturas de Meta-nível

Reflexão também é definida como sendo a habilidade de observar e manipular o comportamento computacional de um sistema através de um processo conhecido como reificação (ou materialização) (BUSATO; RUBIRA; LISBôA, 1997).

Arquiteturas de meta-nível são arquiteturas de *software* baseadas e construídas com reflexão computacional. Um dos conceitos-chave das arquiteturas de meta-nível é a separação do projeto de *software* em dois níveis: um nível base e um meta-nível. No primeiro, é encontrada a funcionalidade do aplicativo, e, no segundo, a sua auto-representação.

Um Protocolo de Meta-Objeto (PMO) estabelece relacionamentos entre o nível base e o meta-nível. Trata-se de uma interface de alto nível para a linguagem de programação. As informações que tendem a não mudar durante a execução do programa, isto é, aquelas obtidas em tempo de compilação, constituem a meta-informação estrutural do protocolo de meta-objetos. Já as informações não permanentes, isto é, que dependem da execução do programa, tratam da parte comportamental do programa, e fazem parte das informações dinâmicas do PMO.

A interface de meta-nível define os métodos que implementam os serviços de reflexão. É possível identificar quatro grupos representando os aspectos básicos dos protocolos de meta-objetos:

- **Conexão entre nível base e meta**: cujo propósito é fazer a vinculação entre componentes do nível base e entidades do meta-nível, tanto estática quanto dinamicamente.

- **Reificação**: tendo como propósito materializar a informação que é obtida a respeito do programa. A materialização se dá sobre métodos, parâmetros, dados e outras informações estruturais e comportamentais.

- **Execução**: que aborda a computação de meta-nível, e a habilidade de invocação de métodos da classe base. Meta-objetos são objetos ordinários, e seus métodos são oferecidos através de uma interface pública, da mesma forma como invocam outros objetos. Entretanto, os meta-objetos necessitam da reificação para serem transparentes ao servidor. Quando um objeto base recebe uma mensagem, o correspondente meta-objeto intercepta e materializa o identificador de método e os parâmetros. A mensagem pode ser enviada ao objeto receptor antes ou depois de algum processamento, o que torna o meta-objeto um cliente indireto do objeto base.

- **Modificação**: já que aspectos dos objetos base podem ser modificados, pois o meta-objeto detém informação sobre sua estrutura e seu comportamento.

Não há um PMO universalmente aceito, que sirva para implementar suporte à reflexão nas linguagens. Em primeiro lugar, é necessário estabelecer qual tipo de meta-informação deve ser revelada ao programa, para, só então, estabelecer o protocolo adequado.

A plataforma Java desde a sua primeira versão, disponibilizou aos desenvolvedores um PMO, através da biblioteca *java.lang.reflect*. As sucessivas versões aprimoraram o PMO, adicionando mais funcionalidades e melhorando o desempenho no que se refere à obtenção de informações dinâmicas. A partir de Java2, *proxies* reflexivos foram incorporados, visando à implementação de um mecanismo de interceptação de mensagens. Este assunto é tratado na seção 2.4.1 com maiores detalhes.

2.2.2 Reflexão e Tolerância a Falhas

No domínio das aplicações tolerantes a falhas, a necessidade de considerar requisitos não funcionais é uma característica inerente, porém, considerá-los acrescenta complexidade ao sistema. Em geral, pretende-se que o desenvolvedor da aplicação não precise

tratar o acréscimo de complexidade, sendo desejável, por vezes, que ele nem mesmo precise estar ciente dos requisitos extras. Um exemplo é a redundância, técnica fundamental das aplicações tolerantes a falhas, que é suportada por mecanismos como manipulação de exceções, pontos de verificação e persistência (LISBôA, 1997).

Ocultar a complexidade adicional, bem como prover requisitos não funcionais transparentemente, pode ser atingido pelo uso de arquiteturas de meta-nível. A idéia é fazer com que os requisitos de TF sejam tratados no meta-nível, de forma a não interferirem com os requisitos da aplicação.

De modo geral, uma arquitetura específica a um domínio compreende: a) uma arquitetura de referência; b) uma biblioteca de componentes, com conhecimento reusável, e; c) um método de configuração de aplicação, para selecionar e configurar os componentes da arquitetura que atendem a um requisito particular da aplicação (LISBôA, 1997).

Assim, no domínio das arquiteturas tolerantes a falhas, o meta-nível compreende:

- Uma arquitetura de meta-nível de referência, dividida em dois níveis: o nível de tolerância a falhas e o nível de lógica da aplicação.
- Uma coleção de padrões de projeto, que descrevem soluções para problemas relacionados à tolerância a falhas.
- Uma biblioteca que contém meta-classes reusáveis, as quais implementam mecanismos de suporte ao desenvolvimento de aplicações de TF.
- Um protocolo de meta-objetos, que provê uma interface entre os componentes da aplicação e o nível de tolerância a falhas.

Nas arquiteturas de meta-nível, uma das principais características é a separação entre requisitos da aplicação e os de tolerância a falhas. A separação é altamente recomendável, pois tem por objetivo reduzir os problemas de interação entre componentes. Porém, ainda que as responsabilidades de cada componente tenham sido declaradas independentemente, espera-se e confia-se que estes interajam estreitamente. Esta interação pode ser entendida como a não interferência do meta-nível na lógica da aplicação, enquanto que a aplicação desconhece os mecanismos de TF que lhe dão suporte.

Estas idéias são adaptáveis aos conceitos de segurança, que são aplicáveis como um requisito global (isto é, atingindo a aplicação completa), ou como requisitos parciais (isto é, atingindo certos componentes da aplicação). Neste último caso, a arquitetura reflexiva aproveita o seu maior potencial, ou seja, permite que meta-objetos controlem cada componente e cada iteração entre componentes.

2.3 Programação Orientada a Aspectos

No que diz respeito ao projeto de sistemas complexos, o modelo de orientação a objetos foi um primeiro passo, importante, na tentativa de diminuir a complexidade e facilitar o projeto de grandes sistemas (ELRAD; FILMAN; BADER, 2001). O conceito aplicado, segundo este paradigma, é decompor o problema de modo que ele possa ser representado por objetos caracterizados por propriedades (atributos), comportamentos (métodos) e uma identidade (nome do objeto) (ECKEL, 2000).

Neste modelo, cada objeto é visto como um provedor de serviços a outros objetos, ditos clientes. O processo de decomposição procura separar do todo cada um de seus

conceitos, de forma que cada classe de objetos represente um conceito através de seus atributos e operações que a estes se apliquem.

Entretanto, a complexidade aumentou dentro do próprio modelo de OO, dado que nem todos os problemas no domínio físico se adaptam, ou se decompõem, no modelo de objetos. Assim, houve uma evolução, resultando no paradigma denominado de Programação Pós-Orientação a Objetos (POO), que envolve técnicas mais apuradas e que não são do domínio de OO (ELRAD; FILMAN; BADER, 2001).

Uma destas técnicas é a Programação Orientada a Aspectos (POA), cuja idéia central é extrair da aplicação aspectos, ou seja, características que refletem a visão que o desenvolvedor e o usuário têm do sistema (ELRAD; FILMAN; BADER, 2001).

Este novo paradigma não exclui as técnicas de OO, e seu uso em aplicações desenvolvidas no modelo de objetos deixa claro que as técnicas até então aplicadas nem sempre são adequadas quando se trata de sistemas mais complexos, principalmente aqueles que envolvem diversos requisitos não funcionais.

Programação Orientada a Aspectos (POA) é uma técnica de desenvolvimento de *software* baseada no princípio de que um sistema pode ser dividido em componentes que refletem suas responsabilidades (*concerns*) e descrições de relacionamentos entre as mesmas (ELRAD; FILMAN; BADER, 2001). A POA, uma das tecnologias de Pós-Orientação a Objetos, é uma nova evolução na linha tecnológica para separação de requisitos, que permite projeto e código serem estruturados do modo como os desenvolvedores querem compreender o sistema (ELRAD, T. et al., 2001).

Assim como a tecnologia de Orientação a Objetos (OO) sucedeu a procedural, a tecnologia POA sucede e complementa a de OO. A diferença básica entre as duas é que em OO os elementos básicos são classes, que visam abstrair elementos do mundo real, enquanto que em POA, os elementos de primeira ordem são aspectos (*concerns*), que nada mais são do que responsabilidades do sistema. Estas responsabilidades, ao contrário da hierarquia de classes encontrada em OO, podem estar dispersas pelo sistema, possivelmente em múltiplas hierarquias de objetos.

E este é o papel de POA: localizar e expressar estas propriedades entrelaçadas com o código funcional do sistema. Em sua essência, POA é uma técnica de programação cuja finalidade precípua é isolar aspectos das demais funcionalidades de uma aplicação, fazer a composição destes aspectos por demanda e integrá-los a pontos específicos da aplicação.

Aspectos variam de requisitos de alto nível, como qualidade de serviço (QoS - *Quality of Service*) a propriedades de baixo nível, como esquemas de *cache* e *buffers*. Exemplos de requisitos que nem sempre se encaixam perfeitamente em uma única estrutura de classes são os mecanismos de sincronização, que geralmente requerem vários objetos concorrendo para a execução de um protocolo; mecanismos de registro (*logs*), nos quais a notificação pode compreender métodos e propriedades de muitas classes, em reação a um evento global do sistema; mecanismos de tolerância a falhas, cujo processo de criação de cópias redundantes precisa ser coerente, etc. Os aspectos que interessam ao trabalho são os sistêmicos, que dizem respeito aos requisitos não-funcionais de uma aplicação. Adicionalmente, aspectos podem ser homogêneos, como um comportamento de registro, ou heterogêneos, como os dois lados de um protocolo implementado por duas classes distintas (ELRAD, T. et al., 2001).

Entre os mecanismos adotados por POA, encontra-se a invocação implícita de métodos, similar aos mecanismos de tratamento de exceções. Contrária à invocação explícita, que exige código contendo os elementos sintáticos da invocação, a implícita desonera o desenvolvedor deste conhecimento. Assim, o código e o fluxo normal de execução do

programa é distinto do código e do fluxo excepcional - que neste caso representa o código não-funcional. A invocação implícita pode ser usada para a implementação de pontos de combinação (*joint points*), ou seja, lugares onde o código do aspecto interage com o restante do sistema - outros aspectos ou com a aplicação alvo. Muitas vezes, esta invocação opera por meio de Reflexão Computacional, e a composição dos mecanismos em um programa coerente é tarefa de níveis mais baixos, segundo Elrad (ELRAD; FILMAN; BADER, 2001).

A principal contribuição de POA é oferecer um modelo conceitual. Ao longo do trabalho, não será diretamente utilizada, embora existam ferramentas cientes desta tecnologia e que já estão disponíveis, por exemplo, o Hyper/J (IBM, 2002), DJlib (DEMETER Project. DJ Library, 2002) e AspectJ (ECLIPSE Project. AspectJ, 2003). Embora se tenha testado o AspectJ, ele não foi adotado, pois diferente do que se propõe, opera por meio de alteração (pré-compilação) de *bytecodes* Java, o que não é desejado.

2.3.1 Filtros de Composição

Embora a tecnologia de Aspectos não seja usada diretamente através das ferramentas disponíveis, uma de suas derivações, a técnica de Filtros de Composição (FC) (*Composition Filters*), tem aplicação neste trabalho.

A abordagem de Filtros tem origem na dificuldade de expressar a coordenação de mensagens no modelo de objetos convencional. Para atingir a sincronização neste modelo é necessário que o código de sincronização esteja disperso entre o código funcional.

O modelo de FC adiciona ao modelo de objetos filtros (*wrappers*) que determinam quando uma mensagem deve ser aceita/rejeitada. Um objeto pode estar associado a um ou mais filtros distintos.

Essencialmente, um filtro é composto por três partes:

1. Condição: o critério a ser satisfeito para que o filtro seja avaliado.

2. Combinação: (*matching part*), corresponde à mensagem que é avaliada e combinada com um padrão definido.

3. Substituição: determina onde e quando as partes de uma mensagem podem ser substituídas.

O modelo de FC compreende duas partes: um objeto interno e uma camada de interface. O primeiro é um objeto convencional, definido por uma linguagem OO. A camada de interface contém os filtros de mensagens de entrada e saída, por onde passam as mensagens. Os filtros podem operar sobre as mensagens modificando-as ou redirecionando-as, tanto para objetos internos ou externos.

As ações que um filtro realiza dependem do seu tipo. Existem filtros pré-definidos, como os de delegação, de espera (*buffers*), de erros e exceções, e novos filtros podem ser adicionados aos já existentes.

A técnica de FC é extensível e adaptável, pois os entrelaçamentos entre código funcional e não-funcional podem ser encapsulados pela adição ou remoção de filtros.

2.3.2 Filtros de Composição como Elementos de Meta-nível

Uma possível maneira de implementar Filtros de Composição é através de *proxies*. Neste caso, o objeto interno é o objeto do nível base e os filtros são rotinas que interceptam este objeto e atuam em meta-nível.

Desta forma, a implementação de um aspecto torna-se possível pela composição de um ou mais filtros sobre determinada classe de mensagens. As classes de mensagens nada mais são do que um conjunto de mensagens que atendem a um requisito específico, por exemplo, segurança, sincronização, persistência, etc.

Para ilustrar, considere-se numa aplicação todas as mensagens - invocações de métodos - que acessem um arquivo para escrita. Um filtro associado a esta mensagem pode verificar os direitos de acesso do requisitante, isto é, se ele tem direito de escrita. A composição de filtros ocorre quando, no mesmo exemplo, se associa à operação de leitura/escrita um filtro criptográfico, de modo que o conteúdo persistente do arquivo nunca esteja disponível em texto plano.

Operar apenas sobre invocações de métodos é bastante adequado, pois o meta-nível só opera ou modifica as mensagens para as quais tenha sido programaticamente desenvolvido, consistente com o modelo de filtros. Em outras palavras, os *proxies* nunca estarão associados a propriedades, nem a construções da linguagem (laços, testes e atribuições).

As operações de meta-nível definidas para este trabalho, de acordo com os padrões adotados, são uma forma de Filtros de Composição. Elas operam sobre mensagens, redirecionando-as para os objetos internos ou externos adequados, e quando uma operação não é suficiente para atingir um objetivo, a composição de um filtro adicional pode ser usada.

Um exemplo de filtro é a substituição dos *sockets* RMI de Java. Quando se deseja uma comunicação segura, é possível substituir a classe base de *sockets* por uma customizada, que implementa criptografia. Ou, nos casos em que a compactação de dados é vantajosa, substituir pela classe que implementa um algoritmo de compactação. A composição, dado que as funcionalidades são ortogonais, também é possível. Embora somente uma classe customizada possa substituir a classe básica, por razões tecnológicas, uma vez que Java suporta apenas herança simples, a implementação das duas funcionalidades em uma única classe customizada não é problemática.

2.4 Padrões de Projeto

De acordo com Gamma (GAMMA, E. et al., 1994), Padrões de Projeto (PP) são "descrições de objetos e classes que se comunicam e que são customizados para resolver um problema de projeto genérico em um contexto particular".

Frameworks também descrevem soluções para problemas. Entretanto, enquanto os *frameworks* resolvem problemas específicos a um domínio, apresentando uma coleção de classes e dependente de linguagens de programação, PP descrevem soluções genéricas, usando uma linguagem independente de linguagem de programação específica. O objetivo para padrões de projeto é o desenvolvimento de soluções apropriadas ao contexto do problema, mas que possa ser reutilizado em ambientes e ou contextos semelhantes sem remodelamentos desnecessários e onerosos. Essencialmente, segue-se o princípio de reutilização estendido a um nível mais alto de abstração.

Um padrão descreve um problema e sua solução, de modo a se poder reutilizá-la, e compreende quatro elementos essenciais:

- **Nome**: que nada mais é do que um identificador, a maneira pela qual se pode referenciá-lo. O nome de um padrão, se bem escolhido e adequado, pode servir de referência aos projetistas e desenvolvedores.

- **Problema**: que é a descrição de quando se aplica o padrão. Descreve o contexto de

aplicação, bem como as condições nas quais se pode aplicá-lo.

- **Solução**: que é a descrição dos elementos que compõem o projeto, suas interrelações, responsabilidades e colaborações. A solução não descreve implementação particular, pois um padrão é neste contexto, um gabarito genérico.

- **Conseqüências**: que são os resultados da aplicação de um padrão. Junto com os resultados, podem ser inclusos custos e benefícios da aplicação, geralmente em termos de tempo e espaço, assim questões de impacto sobre flexibilidade, extensibilidade e portabilidade do sistema.

A interpretação exata de um padrão é difícil de ser obtida, pois é dependente do contexto e de quem observa o sistema. Assim, a decisão entre identificar uma característica recorrente como sendo um padrão ou apenas um bloco constituinte, é deixada a encargo do nível de abstração desejado.

É importante observar que PP estão ligados à linguagem de programação no que diz respeito à expressividade, isto é, o quanto uma linguagem é adequada para implementar um determinado padrão em particular. Isto é verdadeiro tanto na diferenciação entre linguagens procedurais e orientadas a objetos, quanto na diferenciação entre linguagens de um mesmo paradigma, C++ e Java por exemplo.

Padrões são variáveis quanto à granularidade e nível de abstração, e, dada a diversidade de padrões existentes, é adequado classificá-los em famílias, segundo suas relações. Uma classificação, sugerida por Gamma (GAMMA, E. et al., 1994), separa os padrões bidimensionalmente.

A primeira dimensão em que padrões são separados divide-os em três categorias:

- Criacionais, na qual padrões abordam o processo de criação de objetos.

- Estruturais, na qual os padrões tratam de composição de classes ou objetos.

- Comportamentais, na qual os padrões caracterizam a interação entre classes e objetos.

A segunda dimensão, **escopo**, diz respeito a aplicação primária do padrão, se em classes ou objetos. Em classes, os padrões abordam relacionamentos de subclasses, através de herança, cuja natureza é estática (tempo de compilação). Em objetos, os padrões, abordam relacionamentos entre objetos, de natureza mais dinâmica (tempo de execução).

Os problemas de projeto que podem se beneficiar de padrões envolvem: encontrar os objetos adequados a uma abstração de decomposição do sistema; determinar a granularidade dos objetos; especificar interfaces; especificar implementações, através de classes; operacionalizar o mecanismo de reusabilidade, quer por herança, composição, delegação ou tipos parametrizados; relacionar estruturas em tempo de compilação e execução, e, projetar visando alterações futuras.

Padrões são utilizados para desenvolver, além de aplicações, ferramentas e *frameworks*. Ferramentas, neste contexto, significa um conjunto de classes relacionadas, que provêem funcionalidades de propósito geral, tendo nas bibliotecas de linguagens o seu exemplo mais conhecido.

Por sua vez, os *frameworks* são um conjunto de classes que se destinam a implementar um projeto reusável para um domínio específico de aplicações. *Frameworks* são customizados para atender uma determinada aplicação, e, por sua natureza, determinam a arquitetura da mesma.

No caso de Tolerância a Falhas, PP podem apresentar soluções relativamente independentes para propriedades não-funcionais de problemas como ações atômicas, pontos de recuperação, políticas de replicação, exceções, distribuição, registros, segurança etc.

As próximas sub-seções apresentam os padrões relevantes para este trabalho. Trata-se dos padrões que fundamentaram a concepção da arquitetura de referência.

2.4.1 Padrão *Proxy*

Seguindo a classificação de Gamma, o *Proxy* é um padrão estrutural, com escopo de objeto. O objetivo deste padrão é prover um nível de indireção em relação ao objeto real. Este padrão também pode ser denominado *Surrogate*.

A figura 2.2 ilustra o contexto de utilização do *Proxy* (ROHNERT, 1995). Há três participantes, no mínimo: o cliente, que faz requisições ao objeto servidor; o *Proxy*, que intercepta as mensagens e provê correto e eficiente acesso ao objeto real e o objeto real, que provê serviços a clientes. A classe abstrata ilustra o conceito de que tanto o *Proxy* quanto o objeto servidor real compartilham exatamente a mesma interface (embora isto possa ser levemente modificado nas especializações *Remote* e *Firewall*).

O ponto-chave a ser considerado é que o cliente não precisa, necessariamente, saber da existência deste nível de indireção, e para ele, o acesso se dá como se fosse o objeto real. Neste ponto, o objeto servidor pode processar a mensagem e fazer com que a resposta seja entregue via *Proxy*, ou diretamente, de acordo com a necessidade.

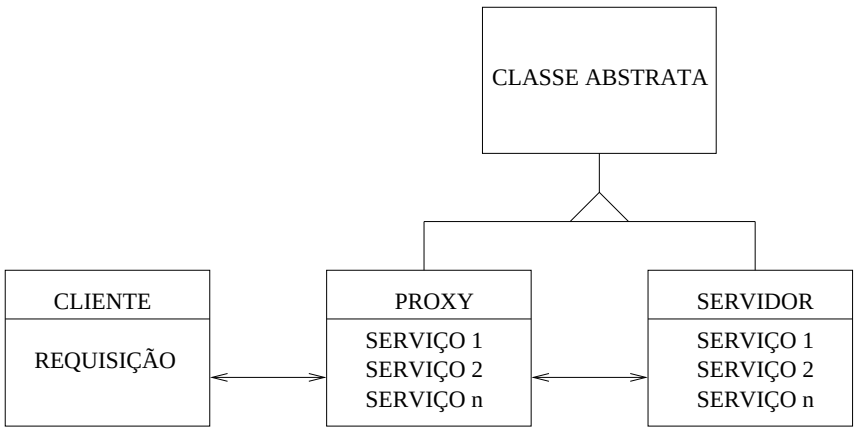

Figura 2.2: O padrão *Proxy*.

A dinâmica de utilização pode ser vista na figura 2.3. A seqüência é como segue: (ROHNERT, 1995)

1. O cliente requisita um serviço, ciente ou não do *Proxy*.
2. O *Proxy* recebe a mensagem e faz eventuais pré-processamentos.
3. Se o pré-processamento atender os objetivos, o *Proxy* entrega a mensagem ao objeto real.
4. O objeto real processa a mensagem, executando o serviço requisitado, e devolve a resposta ao *Proxy*.
5. O *Proxy* pós-processa a mensagem, executando quaisquer tarefas que eventualmente possam ser necessárias, e entrega a resposta ao cliente.

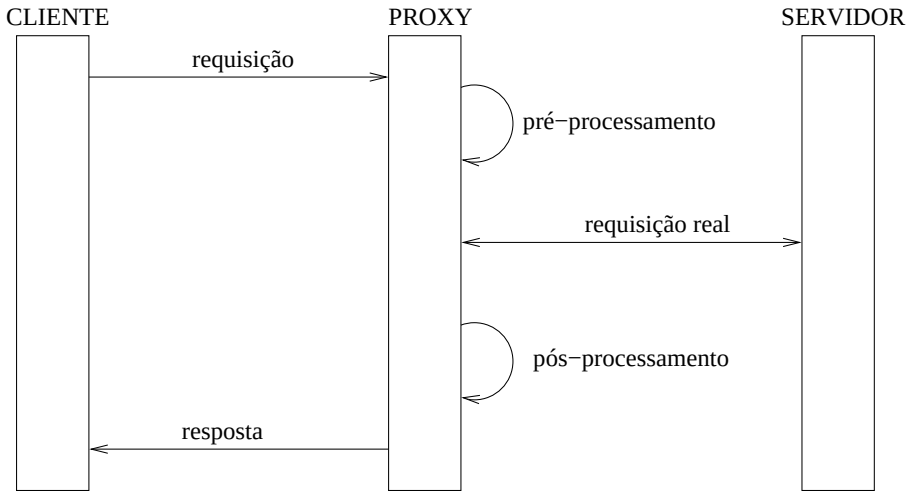

Figura 2.3: Dinâmica de utilização do padrão *Proxy*.

Uma modificação no esquema acima, visando geralmente otimização de desempenho, é permitir ao objeto real devolver a resposta ao cliente. Conceitualmente, é o equivalente a ter pós-processamento nulo.

Do ponto de vista do modelo OO, o *Proxy* aborda a questão de projeto visando mudanças futuras, no que tange à dependência de representações ou implementações. Em outras palavras, o padrão *Proxy* tenta esconder do cliente detalhes de representação, armazenamento, localização e implementação, de modo que uma alteração no objeto servidor não provoque o efeito "avalanche"nos clientes que dele requisitam serviços. Somente a interface é conhecida e não deve, a princípio, ser modificada. É assim, por exemplo, que funciona o mecanismo de invocação remota de métodos (RMI - *Remote Method Invocation*) de Java.

Uma das motivações deste padrão foi permitir que se postergasse a criação de objetos reais muito custosos, em termos de *software*. O padrão, neste caso, serviria como um marcador, apenas assinalando a existência de um objeto, sem, entretanto, carregar ou inicializar o objeto real. Inclusive, existe a possibilidade de esconder do cliente esquemas de cópia-sob-demanda, (*copy-on-write*). Nestes esquemas, um objeto real só é copiado se for alterado, nunca quando for somente lido. Um exemplo conhecido desta aplicação é a ativação de objetos Java via RMI em que um *daemon* no servidor somente ativa o objeto servidor sob requisição de um cliente.

Entretanto, o padrão tem especializações que o tornam mais flexível do que simplesmente atuar como marcador. Três destas especializações são particularmente interessantes em aplicações que envolvam segurança: *Remote*, *Firewall* e *Protection*.

2.4.1.1 Padrão Proxy Remote

Esta especialização do padrão *Proxy* provê um representante local para um objeto que está em outro espaço de endereçamento. Através deste padrão, é possível isolar um componente, ou objeto, dos demais, no que diz respeito ao endereçamento.

Em termos de aplicações tolerantes a falhas seguras, a contribuição deste padrão reside, principalmente, no projeto de *sandboxes* - os ambientes isolados. Nestes ambientes, é possível executar uma aplicação e estudar seu comportamento em casos extremos, sem comprometer o restante do sistema.

Outra característica deste padrão é a possibilidade de codificar requisições e argumen-

tos antes de enviá-los ao objeto real, desta forma, permitindo a comunicação segura sobre canais inseguros.

2.4.1.2 Padrão Proxy Firewall

De acordo com Rohnert, em (ROHNERT, 1995), este *Proxy* tem por objetivo reunir código de rede e proteção necessário à comunicação segura em ambientes hostis. O padrão trata dos detalhes complicados envolvidos na elaboração de protocolos de rede. O modelo sugerido para o padrão é codificá-lo como um processo residente *daemon*. Os clientes se registram junto ao processo e a partir daí, a comunicação é transparente entre cliente e servidor.

2.4.1.3 Padrão Proxy Protection

Esta especialização do *Proxy* controla o acesso aos objetos originais. É particularmente adequado quando se quer que os clientes acessem os servidores segundo políticas de acesso baseadas em direitos. Exemplos destas políticas de acesso são as Listas de Controle de Acesso (ACL - *Access Control Lists*).

As políticas de acesso podem ser derivadas das que o ambiente nativo oferece, a exemplo dos sistemas operacionais e máquinas virtuais, ou podem ser criadas explicitamente, para atender a uma finalidade específica, por exemplo, através de descrições em XML (*eXtended Markup Language*).

O *Proxy* e seus derivados podem ter atribuições extras, como criar e apagar objetos sob demanda. Esta característica o torna flexível, em termos de segurança, pois pode substituir objetos comprometidos por objetos confiáveis, ao detectá-los.

Um *Proxy* sempre oferece ao cliente a mesma interface do objeto real. Se for desejado modificá-la, pode-se recorrer ao uso do padrão *Adapter*, adequado a esta tarefa (GAMMA, E. et al., 1994).

2.4.2 Padrão *BodyGuard*

O padrão *Bodyguard* é descrito por Neves (NEVES; GARRIDO, 1997), e tem como objetivo resolver o problema de compartilhamento e controle de acesso a objetos distribuídos em ambientes que não têm suporte à distribuição.

O padrão pertence à categoria Comportamental, com escopo em objetos. Ele provê envio de mensagens com validação e atribuição de direitos de acesso a objetos não locais, isto é, que não pertencem ao mesmo ambiente colaborativo.

Conceitualmente, existem três subcomponentes dentro do padrão: um objeto a ser compartilhado, um *Transporter* e um *Bodyguard*. O primeiro é o próprio objeto. O segundo, trata das mensagens, especificamente da serialização e reconstituição de objetos. Há um *Transporter* por cada nodo da rede que esteja envolvido. Por fim, o *Bodyguard* trata do acesso ao objeto compartilhado.

O *Bodyguard*, por seus objetivos, é adequado quando há chamadas de sistema dependentes de plataforma, no que tange à comunicação. Pela mesma razão, é adequado quando a separação entre o sistema de mensagens e o controle de acesso for indistigüível. Também pode ser utilizado para prover sincronização, controle de acesso mais especializado que um *Proxy* possa prover, mudança dinâmica nos direitos de acesso e verificação de identidade.

O padrão não é adequado quando se deseja unicamente autenticação, e neste caso o *proxy* pode ser uma opção mais simples, assim como não suporta serviços distribuídos

complexos e ambientes muito heterogêneos de rede.

A definição completa do padrão *Bodyguard* pode ser vista na figura 2.4 (NEVES; GARRIDO, 1997). Por simplicidade, foram omitidos os métodos e responsabilidades da figura.

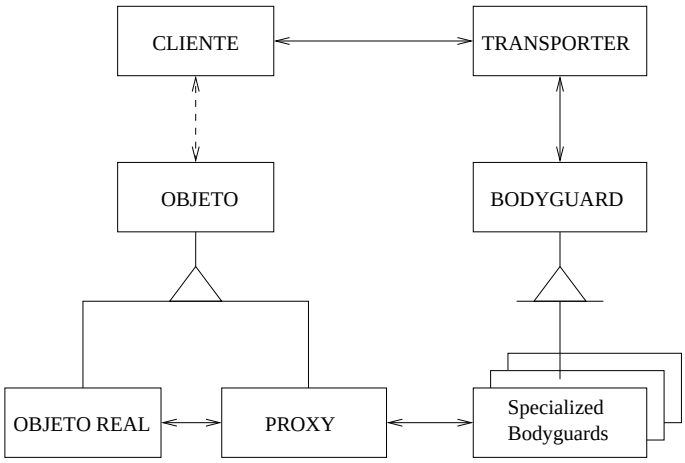

Figura 2.4: Descrição do Padrão *Bodyguard*.

Os participantes são:

- *Objeto Real*: é o objeto a ser compartilhado.
- *Transporter*: cuida dos mecanismos de comunicação e mensagens. Adicionalmente, este componente registra as associações entre *proxies* remotos e objetos compartilhados, bem como, pode implementar ações atômicas e criar *proxies*.
- *Bodyguard*: é um mediador entre o *Proxy* e o *Transporter*. Manipula requisições, respostas, notificações e manipulação de erros, considerando controle de acesso, sempre que necessário, e protocolos de comunicação. *Bodyguard* é a classe abstrata, enquanto que *SpecializedBodyguard* é a implementação concreta.
- *SpecializedBodyguard*: implementa o *Bodyguard* para um determinado objeto compartilhado. É uma subclasse daquela.
- *Proxy*: atua como um objeto *proxy* do objeto real.

A figura 2.5 ilustra a dinâmica de operação do padrão, no seu modo de funcionamento generalizado. É possível elaborar esquemas variados sobre este modo básico, geralmente na tentativa de obter melhora de desempenho ou adequar um protocolo muito particular.

O cliente inicia a requisição de um serviço, que é interceptada pelo *Proxy*. Este, por sua vez, repassa a mensagem ao *Bodyguard*, que, sob validação, comunica-se com o *Transporter* local. A mensagem é enviada ao *Transporter* remoto que finalmente acessa o objeto real.

Não mostrado na figura 2.5, há uma fase prévia de busca do objeto servidor. A omissão visa a simplicidade de representação, e seu funcionamento é simples: o cliente, no primeiro acesso, faz com que os padrões *Transporter* e *Proxy* estabeleçam a localização de um servidor e a ele associem um *SpecializedBodyguard*, que representa uma possível implementação do *BodyGuard*, cuja referência ficará armazenada no *Proxy*.

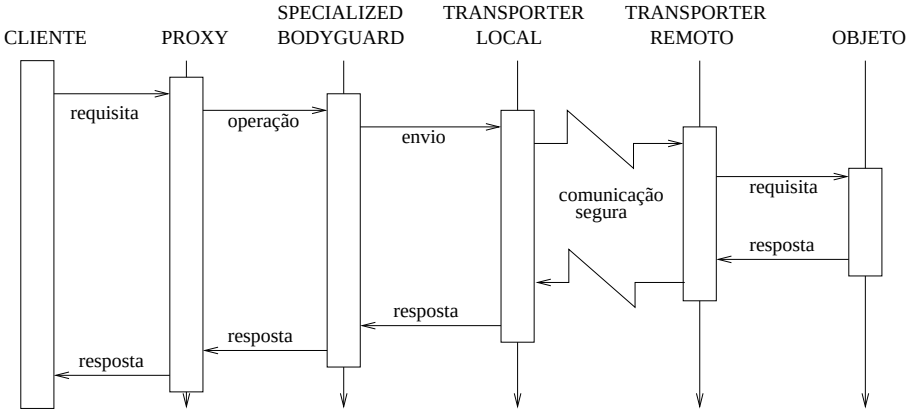

Figura 2.5: Dinâmica de utilização do padrão *Bodyguard*.

O padrão apresentado introduz níveis de indireção no sistema. Portanto, se o desempenho na comunicação for dominante em relação ao controle de acesso, não é aconselhável utilizá-lo.

Além de auxiliar na implementação de objetos distribuídos, um *Bodyguard* também pode implementar mecanismos de depuração, registro (*log*) e filas de mensagens, podendo estas implementações particulares corresponder a diferentes *SpecializedBodyGuards*.

Neves (NEVES; GARRIDO, 1997) sugere, nos aspectos de implementação, que o componente *Transporter* gerencie os objetos compartilhados em cada nó da rede. Além dos aspectos de comunicação, ele deveria também ser o responsável pela criação dinâmica dos *Proxies* de cada objeto real. Outra sugestão é a de que, se o *Transporter* conhecer os meta-dados do objeto servidor real, ele poderia clonar servidores sob demanda.

Quanto a direitos de acesso que podem mudar dinamicamente, Neves sugere um protocolo no qual um objeto cliente conheça a natureza do servidor, isto é, se é local ou remoto, compartilhado ou privado e os direitos de acesso atuais, antes de invocar qualquer operação sobre ele. Neste caso, o objeto real pode implementar métodos padrão do protocolo e que são redefinidos no *Proxy*.

Por fim, Neves atribui ao *Transporter* algum controle sobre a contagem de referências ao objeto real, objetivando gerenciá-lo quanto à coleta de lixo. Em linguagens que implementem coleta de lixo, como Java, esta operação pode ser simplificada, desassociando o *Proxy* do objeto real, por exemplo. Neste caso, o *Transporter* apenas anula suas referências ao *Proxy* correspondente quando o cliente finalizar a sessão.

2.4.3 Visão Geral da Composição dos Padrões

Reunindo-se as características dos padrões apresentados, chega-se ao diagrama ilustrado na figura 2.6.

Neste diagrama, percebe-se que o padrão *Bodyguard* é uma construção que visa ampliar o potencial do *Proxy*. Resumidamente, o *Transporter* se encarrega de criar dinamicamente, no espaço do cliente, *Proxies*, e estes, por sua vez, de criar objetos reais. O controle de direitos de acesso é melhorado com o *SpecializedBodyguard*, que atua antes dos *proxies Remote*, *Protection* e *Firewall*, abrindo a possibilidade de níveis diversos de direitos de acesso dentro das próprias políticas de acesso, ou seja, controle da arquitetura sobre si mesma.

A composição exata, assim como os relacionamentos entre os componentes do *Bodyguard* e o(s) *Proxy(ies)*, é determinada pelo desenvolvedor, segundo as necessidades da

aplicação, dentro da arquitetura acima.

A figura 2.7 ilustra o diagrama genérico de um *Bodyguard*, de acordo com os padrões apresentados. Esta é a situação a partir da qual se pode customizar grande parte das aplicações que se enquadram neste tipo de arquitetura.

Na figura 2.7, as entidades foram agrupadas em três níveis:

1. Cliente-Servidor: nível do modelo de comunicação da aplicação. A rigor, desconhece quaisquer níveis que estejam abaixo.

2. *Proxy*: nível que representa a implementação das políticas de acesso definidas pelo usuário-desenvolvedor. Conceitualmente, desconhece o nível inferior, mas, visando à melhora de desempenho, geralmente é ciente e interage com o *Bodyguard*.

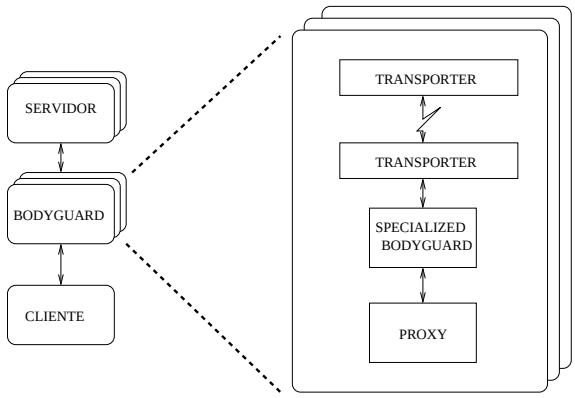

Figura 2.6: Visão geral da composição dos padrões apresentados.

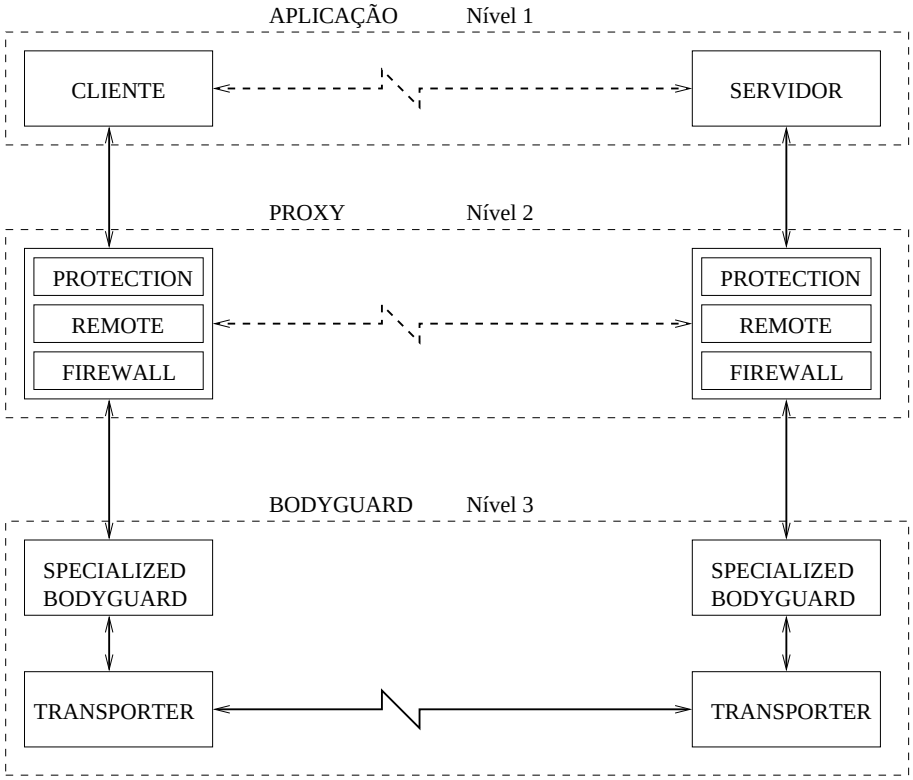

Figura 2.7: Composição genérica de todos os padrões apresentados.

3. *Bodyguard*: nível que trata de aspectos de controle fino do sistema, assim como aspectos de distribuição e comunicação. De acordo com o que foi apresentado, deve gerenciar os *proxies* do nível superior, mas, por questões de desempenho, pode-se fazer com que haja colaboração mais estreita, com todos os componentes cientes da existência dos demais.

A linha tracejada entre o cliente e o servidor, assim como a linha entre os componentes *Remote* indicam abstrações dos níveis de comunicação. As linhas em traçado sólido representam o caminho efetivo da comunicação.

A simetria da aplicação de padrões visa à aplicações que requeiram aspectos semelhantes tanto no cliente quanto no servidor. Entretanto, não invalida a aplicação em sistemas que requeiram implementação de novos padrões somente no servidor, que é caso, por exemplo, de clientes legados e servidores em fase de atualização.

2.5 Conclusão

Este capítulo apresentou as tecnologias e bases conceituais que facilitam o desenvolvimento de aplicações seguras através de um conjunto de *SpecializedBodyGuards*, implementados como componentes que interagem e estruturados em uma arquitetura de *middleware*, descrita no capítulo 3.

Cada *SpecializedBodyGuard* contempla um aspecto específico de segurança, e estes podem ser combinados para atuar nos pontos de junção da aplicação, definidos pelo desenvolvedor. O capítulo 4 descreve estas relações.

3 ARQUITETURA PROPOSTA

3.1 Introdução

Este capítulo descreve a arquitetura proposta, com base nas tecnologias e conceitos do capítulo anterior.

A arquitetura que se está propondo tem por objetivo fornecer serviços de segurança para aplicações desenvolvidas na plataforma Java. Os serviços de segurança abordados primariamente são autenticação e autorização. A plataforma Java permite ainda serviços de auditoria, confidencialidade e confinamento. A confidencialidade é abordada secundariamente, em dois momentos: quando se utiliza métodos de comunicação segura e quando se provê persistência confidencial. Destes dois, apenas se considera comunicação segura. O confinamento, por sua vez, é visto como propriedade inerente ao *middleware*, pois, construtivamente, nada executa fora dele.

As aplicações-alvo são, primariamente, aquelas desenvolvidas no modelo cliente-servidor, utilizando invocação remota de métodos (*Remote Method Invocation* - RMI). Este modelo de aplicação foi escolhido por ser simples o suficiente para compreender seus vários aspectos e ainda ser um modelo utilizado de fato.

Como componente estrutural da arquitetura, um *middleware* executa sob a aplicação-alvo, intermediando seu acesso aos recursos do sistema e acessos entre cliente e servidor. Esta opção de desenvolvimento permite atingir dois objetivos: contemplar aplicações existentes e servir de base para aplicações em desenvolvimento que visem o uso de *middleware* com características de segurança.

A primeira seção introduz a noção de *middleware*, tal como é relevante aqui. As características são apresentadas para que se possa, ao longo das seções seguintes, compreender algumas das particularidades da arquitetura proposta.

Após, são apresentados os diagramas de caso de uso e de colaboração que norteiam a arquitetura geral. Nestes diagramas estão representados a visão geral da arquitetura segundo as interações do usuário, e não dos componentes, isto é, eles não representam todas as interações entre classes, mas as que ocorrem segundo as principais ações de alto nível.

Finalmente, a seção 3.4 apresenta a arquitetura do *middleware* proposta, com as inter-relações entre os padrões e o mapeamento para as principais classes Java envolvidas.

3.2 *Middleware*

Middleware é um programa, ou um conjunto deles, que atua entre os sistemas operacionais de rede (SO) e os aplicativos de uma máquina, objetivando resolver problemas de heterogeneidade e facilitar a comunicação e coordenação de componentes distribuídos

(EMMERICH, 2000a). Entretanto cabe observar que existem *middlewares* residentes em níveis acima das próprias máquinas virtuais, como é o caso do RMI.

O conceito e as idéias básicas de *middleware* já contam com algum tempo de pesquisa, inclusive com adoção no meio comercial (CHARLES, 1999). Entretanto, a tecnologia ainda enfrenta desafios quanto à aplicação em larga escala, adaptatibilidade e reconfiguração de soluções. Usualmente associado à integração de sistemas comerciais, escalabilidade, tolerância a falhas e migração de componentes legados, recentemente também é possível encontrar esta tecnologia associada à solução de problemas de segurança (WELCH; STROUD, 2000a).

Ao se adotar uma solução baseada em *software* desta natureza, deve-se levar em conta a disponibilidade de soluções, a adequação de cada uma e como elas podem ser usadas na arquitetura, projeto e implementação geral do sistema distribuído. Nos casos em que a solução precisa ser desenvolvida, além das ponderações anteriores, também se fazem algumas considerações tais como:

- O custo e o tempo de desenvolvimento do sistema distribuído em termos de primitivas do sistema operacional.
- A que extensão o *middleware* deve implementar interfaces para o SO e a que grau de abstração estas são oferecidas à camada superior de *software*.
- Detalhes sobre os níveis de sessão e apresentação que devem ser implementadas, notadamente encapsulamento e desencapsulamento de dados.
- Semânticas de sincronização entre componentes distribuídos, bem como comunicação de grupo, politicas de ativação e persistência e políticas de atendimento, no caso de múltiplos servidores.
- Grau de confiabilidade do sistema, tanto em relação à entrega de mensagens quanto nos casos de transações e replicação.
- Escalabilidade e questões relativas às transparências oferecidas, incluso as necessárias se o objetivo for prover balanceamento de carga entre os servidores.
- Heterogeneidade entre as plataformas atingidas, envolvendo o *hardware*, SO e, eventualmente, as linguagens de programação.

Emmerich (EMMERICH, 2000a) analisa algumas soluções comerciais de *middleware*, classificando-as entre transacionais, orientadas a mensagens, procedurais e orientadas a objetos e componentes. De acordo com esta separação, o RMI da plataforma Java se enquadra como orientado a objetos. As abordagens são diversas, e não há uma solução única a todos os problemas, tendo cada solução aplicabilidade em domínios que empregam técnicas semelhantes à sua natureza.

Nem sempre o uso de *middleware* em um projeto é totalmente transparente aos projetistas (EMMERICH, 2000b). Isto significa que se pode esperar alguma interação entre o sistema distribuído e a camada inferior de *software*. Esta efeito também pode se fazer sentir pelo usuário do sistema, por exemplo no caso em que parâmetros devem ser configurados e ou ajustados sob demanda.

A figura 3.1 ilustra o contexto de aplicação do *middleware* orientado à segurança. Como pode ser visto, o *middleware* reside entre a aplicação e a máquina virtual, usando o RMI como camada de comunicação.

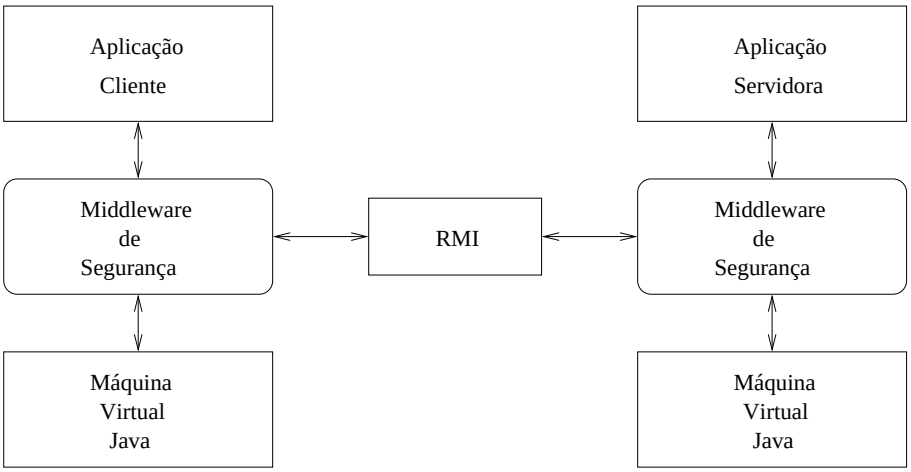

Figura 3.1: Contexto de aplicação de *middleware* orientado à segurança.

No que tange a este trabalho, as implicações do *middleware* num sistema compreendem as implicações diretas do uso de RMI e as introduzidas pelo código criado. As principais implicações de se usar RMI são a pouca flexibilidade, características reflexivas e escalabilidade. Pouca flexibilidade significa que ainda é necessário a vinculação de um cliente com um servidor por nomeação, isto é, não há transparência de localização (quanto a recursos - no caso, invocação de métodos). As características de reflexão dizem respeito a uma invocação sobre um objeto retornar outro, alcançado através do mecanismo de serialização; descoberta em tempo de execução de tipos e classes, também suportado pela linguagem; protocolos de meta-objetos, estes com vistas a inspeção e adaptação do próprio *middleware*. Este último deve ser feito manualmente, isto é, o código que utiliza RMI, se precisar ser modificado em tempo de execução, deve conter o código que (re)carrega objetos. Ainda, há o problema inerente da escalabilidade, pois RMI funciona de acordo com o modelo cliente-servidor, tendo no último um limitante. Pela mesma razão, RMI não suporta naturalmente replicação, e as interações entre componentes é de total responsabilidade dos desenvolvedores das aplicações, assim como a sincronização entre os objetos.

Por outro lado, RMI pode interoperar com CORBA (*Common Object Request Broker Architecture*, beneficiando-se das características deste outro *middleware*, quando se trata de comunicação entre objetos distribuídos heterogêneos. O RMI também trata do nível de sessão e apresentação, e juntamente com a plataforma cuida dos detalhes de encapsulamento (serialização). Aplicações que usam RMI naturalmente defrontam-se com o problema da latência de comunicação entre objetos locais e remotos, de modo que este não é um problema a ser resolvido pelo código que o estende. RMI fornece serviços de ativação e desativação de servidores, para poupar recursos nas plataformas servidoras. Persistência é atingida com ajuda da linguagem e da JVM. Por outro lado, as interações entre componentes são de total responsabilidade dos desenvolvedores das aplicações, assim como a sincronização entre os objetos.

Por fim, é pertinente uma observação sobre a importância a distinção entre os requisitos funcionais e não-funcionais de uma aplicação já na etapa de projeto concomitante à determinação das atribuições do *middleware*. Tanto no caso de *middleware* de prateleira quanto no desenvolvido especificamente, a preocupação centra-se em elicitar os requisitos de modo que se adaptem à situação atual e à futura. Isto porque torna-se difícil e custoso migrar aplicações desenvolvidas sobre uma solução para outra, assim como torna difícil

a modificação das bases sem impactar as aplicações. Em outras palavras, a distinção e atribuição de funcionalidades deve ser bem planejada.

Visto o que se pode esperar de um *middleware*, a próxima seção apresenta os diagramas de casos de uso e colaboração dos principais componentes do *middleware* de segurança proposto neste trabalho.

3.3 Diagramas de Caso de Uso e Colaboração

Para que se possa compreender a composição geral da arquitetura, esta seção apresenta três dos mais importantes diagramas, um de caso de uso e dois de colaboração dos componentes.

O diagrama de caso de uso representado na figura 3.2 ilustra a interação entre o usuário e a interface de configuração do sistema.

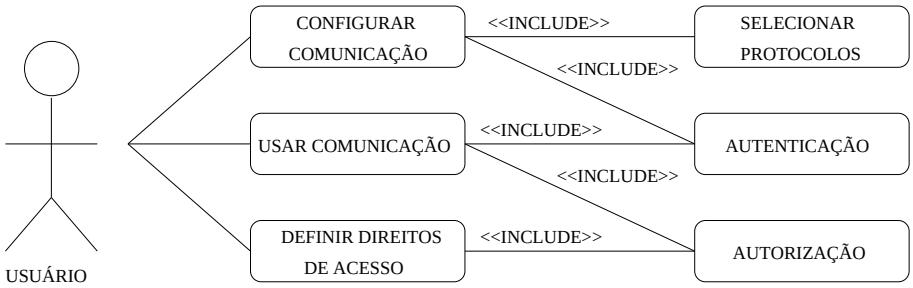

Figura 3.2: Diagrama de caso de uso do sistema.

O usuário, neste diagrama, é tipicamente a figura do administrador do sistema, ou quem esteja no encargo de configurar o *software*, tanto do servidor quanto dos clientes. A primeira interação ocorre com o processo de configurar a comunicação, momento em que são selecionados os protocolos e o método de autenticação. A segunda interação ocorre com o processo de definir os direitos de acesso, quando são estabelecidos os direitos de execução de ações por usuário (ou grupo de usuários). Nestas duas etapas iniciais definem-se os diversos parâmetros de protocolos de comunicação, se usam criptografia, compressão (ou eventualmente algum outro mecanismo); os esquemas de autenticação, se utilizam a base de senhas do sistema (tradicionalmente, arquivos *passwd*), base própria ou base distribuída (por exemplo *Kerberos*) e o controle fino de ações executáveis por usuário, controláveis ao nível de métodos de classes.

Numa terceira etapa, o usuário, que agora pode ser visto também como o usuário final do sistema, utiliza a comunicação entre cliente e servidor, na forma de solicitação de serviços. A invocação de serviços está sujeita à sistemática de autenticação e a compleção de uma requisição está sujeita aos direitos de acesso. O trânsito das mensagens segue o protocolo e os parâmetros configurados, imutavelmente.

O segundo diagrama, visto na figura 3.3, ilustra o processo de configuração da comunicação e controle de acesso, ou seja, um perfil. Nele, pode ser visto que o configurador, ou seja, a classe que procede à configuração geral do *middleware*, age baseada em perfis de usuário. Tipicamente, o administrador, ao registrar um novo usuário à base, cria para ele um perfil, onde constam seus direitos de acesso e as opções de protocolo a serem usadas. O configurador, por sua vez, valida o perfil contra um filtro, garantindo a consistência da entrada.

Após a validação do perfil, é apresentado ao usuário uma lista de protocolos válidos, de acordo com o perfil apresentado. Escolhe-se o protocolo e uma segunda verificação

de consistência pode ser feita, tal como redundância (empilhar protocolos criptográficos sobre os canais de comunicação é um exemplo, pois embora válido, é geralmente ineficiente e não adiciona segurança extra ao canal). Uma vez configurado o protocolo, toda vez que um usuário da aplicação apresentar seu perfil, o nível do *Transporter* irá usar o protocolo pré-definido.

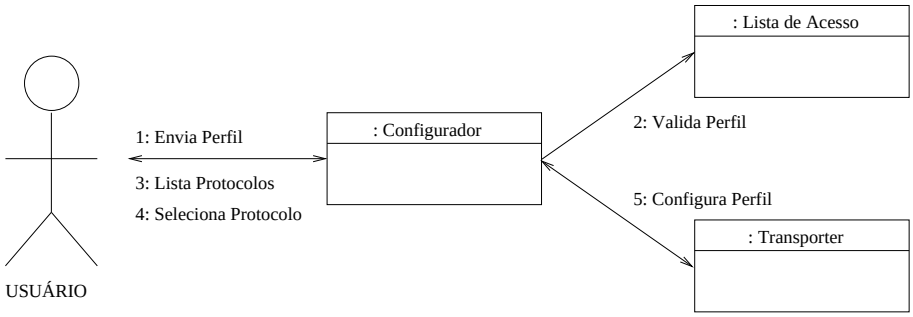

Figura 3.3: Diagrama de colaboração para configurar o perfil de usuário.

Por fim, o diagrama da figura 3.4 ilustra o uso do sistema pelo usuário final da aplicação. O usuário inicia ativando o processo cliente, que eventualmente requisita algum serviço do servidor. Esta requisição passa pelo *Proxy* do servidor, que valida a requisição, isto é, determina a legitimidade do usuário contra o perfil registrado pelo administrador. Sendo legítima, o *Proxy* repassa a mensagem para o *Transporter*, que usando o protocolo configurado, repassa a mensagem ao servidor pelo canal de comunicação.

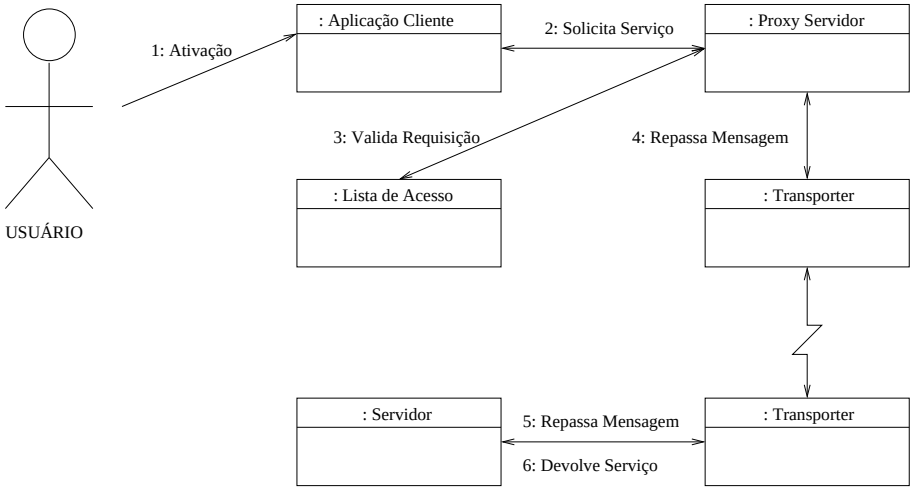

Figura 3.4: Diagrama de uso do sistema por parte dos usuários finais.

Num cenário onde os requisitos de segurança não são tão rigorosos, o sistema do lado servidor simplesmente processa a requisição e devolve a resposta ao cliente. Em cenários onde se requer nível mais estrito de segurança, o sistema beneficia-se da simetria da camada de *middleware*, podendo fazer as mesmas verificações que o *Proxy* fez e outras adicionais, como validar a origem da requisição, a autenticidade do usuário contra a base registrada no servidor e a legitimidade da requisição em si, isto é, se a ação pretendida é autorizada pela lista de controle de acesso.

Na elaboração do sistema, deverá ser previsto uma forma de configuração, ainda que não implementada neste estágio. Isto significa, principalmente, evitar a codificação de

propriedades e comportamentos diretamente no código principal. A pré-configuração do sistema tem por objetivo reduzir o processamento extra que seria necessário caso todas as funcionalidades devessem ser estabelecidas em tempo de execução, bem como o tempo de inicialização do sistema no caso de interação com o usuário. Além disso, gerenciadores de segurança e controladores de acesso precisam ser definidos o mais cedo possível, preferencialmente antes de executar a aplicação-alvo.

Dado que o objetivo deste trabalho é oferecer uma infraestrutura da natureza que se apresentou nesta seção, e que a aplicação original pode não estar ciente ou mesmo adequada a este tipo de serviço, a próxima seção descreve a camada de *software* responsável pela introdução destas capacidades. Após, apresenta-se a arquitetura do sistema, cujos detalhes de implementação são descritos no capítulo 4.

3.4 Arquitetura Proposta

A arquitetura do *middleware* de segurança baseia-se na arquitetura em camadas do RMI, representada na figura 3.5 pelos elementos com traçado sólido. O elemento pontilhado representa o ponto em que parte da arquitetura proposta se insere na original. No topo da figura, sobre os *stubs* e *skeletons* está representada a camada de aplicação (cliente-servidor), a título de contextualização (SUN, 2004).

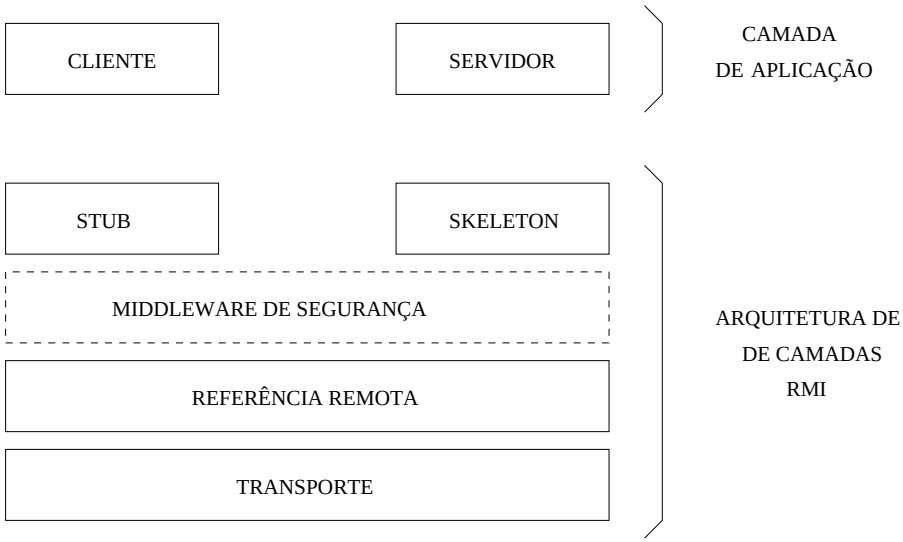

Figura 3.5: A arquitetura em camadas do RMI Java.

A camada de *stubs* e *skeletons* implementa o padrão *Proxy* internamente ao RMI, sendo que o *Stub* implementa o *proxy* e o *Skeleton*, o objeto original. O *proxy* cuida de propagar chamadas de métodos entre os objetos participantes. Até a versão 1.1 de Java, o *skeleton* é uma classe auxiliar, gerada via compilador *rmic* a partir da definição da interface (classe) do objeto servidor. A partir da versão 2, o RMI de Java utiliza reflexão para se conectar ao servidor remoto, tornando obsoleto o componente *skeleton*. Entretanto, para manter compatibilidade com sistemas legados, este ainda é necessário.

A camada de referência remota define e suporta a semântica de comunicação de uma conexão RMI, que é ponto-a-ponto e pode utilizar objetos *Activatable*, isto é, carregados no servidor sob demanda. Em termos funcionais, este nível fornece um objeto ao cliente que é representativo da conexão com o servidor remoto. Sobre este objeto, nomeado *RemoteRef*, o *stub* chama o método *invoke()*, que por sua vez propaga a chamada ao

servidor remoto. Até a versão 1.1 de Java, somente a semântica de conexão ponto-a-ponto é suportada. Além disso, os objetos servidores obrigatoriamente precisam ser instanciados no servidor e exportados para o subsistema RMI, sendo também obrigatório nomear e registrá-lo (via *RMIregistry*) caso seja o serviço primário. A partir da versão 2, o RMI oferece uma nova semântica, a de objetos ativáveis. Neste caso, quando um cliente invoca um método de um objeto remoto, o RMI verifica se o servidor está ativo, isto é, se está instanciado. Se não estiver, o RMI instancia um servidor em memória e restaura seu estado.

A nomeação de um serviço RMI é da forma:

rmi://<nome-do-servidor>[<:porta>]/<nome-do-serviço>.

O protocolo suportado originalmente se restringia ao RMI (*rmi:*), mas foi estendido para suportar HTTP (*HyperText Transfer Protocol*) e o FTP (*File Transfer Protocol*), notados como *http:* e *ftp:*, respectivamente. Para os fins deste trabalho, apenas se considera o protocolo RMI, pois o *middleware* deve prever simetria de operações entre cliente e servidor, o que pode não ser possível com servidores HTTP e FTP. O nome do servidor pode ser um nome de máquina da rede local ou um nome reconhecido pelo DNS (*Domain Name Service*). O componente *<:porta>* é opcional e pode ser omitido se for utilizada a porta padrão do RMI (1099). Finalmente, *<nome-do-serviço>* é o nome através do qual o servidor se registrou no registro RMI.

Por fim, a camada de transporte faz a conexão entre máquinas virtuais, utilizando os protocolos TCP/IP do sistema. Esta camada também é responsável por multiplexar várias conexões virtuais feitas pela camada de referência remota em um uma única conexão TCP/IP, para o caso dos sistemas que permitem apenas uma conexão entre cliente e servidor.

A arquitetura modificada passa a incluir mais uma camada, entre a de *stubs* e *skeletons* e a de referência remota. Esta última responde pelos serviços de segurança. Os serviços de segurança oferecidos são autenticação e autorização.

Por autenticação, deve ser prevista uma ou mais formas de os usuários das aplicações se identificarem perante o sistema. Usualmente, no lado servidor é o próprio administrador do sistema que inicia o serviço, de modo que um esquema baseado em senhas é suficiente para autenticar o usuário e permitir o disparo da aplicação. Entretanto, no lado do cliente este mecanismo pode não ser suficiente, pois os usuários ainda poderiam executar a aplicação original e proceder com o uso normalmente. Neste caso, o uso se daria à extensão em que o servidor não fosse requerer alguma característica introduzida pelo *middleware*. Por exemplo, se a aplicação faz uso do *middleware* apenas para autenticar usuários válidos, o cliente teria pleno acesso ao servidor. Uma das formas de minimizar este problema é através do uso de *sockets* customizados, pois o cliente não conseguirá estabelecer o protocolo correto com o servidor sem utilizar o mesmo tipo, tarefa que cabe ao *middleware*.

O serviço de autorização terá por responsabilidade, uma vez credenciado o usuário, delimitar suas ações sobre o sistema. Localmente, isto se traduz primariamente pelo controle de acesso a arquivos, processador e memória e, distribuidamente, pelo controle de acesso dos clientes aos servidores. Neste nível de serviço, começa-se a minimizar a possibilidade de contorno às medidas de segurança, pois a simetria do *middleware* cliente em relação ao servidor elimina a possibilidade de o servidor responder a um cliente que não passa pelas etapas adequadas. Como visto anteriormente, esta etapa pode ser tão simples quanto estabelecer um *socket* customizado, ou pode ser tão robusta quanto utilizar certificados digitais para assinar mensagens, ou ainda, cifrá-las totalmente.

Por fim, o controle de acesso via autorização proverá controle fino de acesso a métodos. O diferencial está no fato de este controle ser dedicado aos objetos servidores, isto é, não há preocupação com recursos tais como comunicação e arquivos. Pode ser visto como complementar à autorização em nível global, agindo em um nível mais interno e especializado. Por exemplo, não é utilizado no nível que cuida de direitos de acesso à rede, mas é capaz de identificar, internamente à aplicação, os direitos de um usuário com relação a um serviço. Tipicamente, este controle é exercido no *Proxy* do servidor, antes da chamada de método no nível base.

Deve ser entendido, na representação da figura 3.5, que o *middleware* não opera exclusivamente no nível destacado, tendo outros componentes que interagem em outros locais da plataforma, como por exemplo, os componentes que procedem à carga da aplicação, e que são melhor representados em outros contextos, apresentados na seção 3.4.1. O posicionamento do destaque objetiva uma representação conceitual da interação entre o *middleware* de segurança e o *middleware* RMI. Igualmente, deve-se notar que há outro ponto de interação entre os dois, de origem mais tecnológica do que conceitual, propriamente, e que diz respeito à forma de comunicação entre cliente e servidor, no nível de *sockets*, na qual se acresce os serviços de segurança na forma de canais criptografados, por exemplo.

Existem metodologias (VETTERLING; WIMMEL; WISSPEINTNER, 2002) que consideram critérios como o *CommonCriteria* para o desenvolvimento deste tipo de aplicação, porém, não serão utilizadas aqui. Isto se deve ao fato de estas metodologias requererem recursos humanos e temporais dispendiosos, bom conhecimento dos critérios e geralmente são voltadas à certificação do produto final perante algum órgão de reconhecimento. Além disso, um dos objetivos secundários é explorar a plataforma/linguagem quanto à sua adequação a um *middleware* desta natureza, o que se conseguirá em etapas, não esgotando-se neste trabalho.

Outro dos preceitos na elaboração desta arquitetura é o uso exclusivo de elementos da linguagem e nativos da plataforma, isto é, não se pretende utilizar recursos externos como classe proprietárias, ferramentas de desenvolvimento específicas ou plataformas especiais. Nesta mesma linha de exploração das capacidades da linguagem/plataforma, uma restrição adicional é a de que não se precise modificar o código intermediário (*bytecodes*) ou da plataforma. Sobre este último, cabe notar que há soluções que se valem da modificação de código intermediário (WELCH; STROUD, 2000b) e (WELCH; STROUD, 2001), também no sentido de compor *middleware* de segurança.

3.4.1 Elaboração de um Padrão de Projeto Seguro

A proposta geral é refinada, nesta seção, pelo detalhamento do Padrão de Projeto voltado à segurança.

Este padrão é composto a partir de dois outros: o *Proxy* e o *Bodyguard*. A figura 3.6 ilustra esta composição, com todos os elementos que compõem o cenário. Entre parênteses estão as classes ou interfaces para as quais se mapeiam os serviços.

Os primeiros elementos desta arquitetura são o cliente e o servidor. O fato de estarem representados conjuntamente no diagrama ilustra um ponto característico deste tipo de arquitetura: a simetria em relação ao *middleware*. Esta simetria é necessária, pois a rigor, um novo protocolo está sendo agregado ao já existente, isto é, não se trata mais unicamente de uma comunicação cliente-servidor, mas de uma comunicação encapsulada pelo *middleware*.

O próximo elemento da arquitetura é o *Proxy*, esquematizado na figura 3.7.

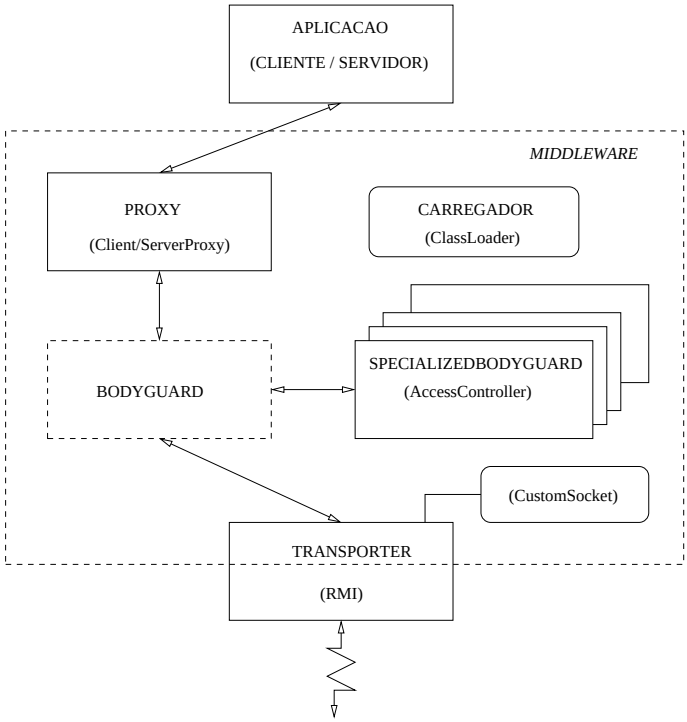

Figura 3.6: Padrão proposto para a arquitetura.

```
1  import java.lang.reflect.InvocationHandler;
2
3  public interface ProxyInterface {
4          public Object Method();
5  }
6
7  public class ProxyClass implements InvocationHandler {
8          Object obj;
9          public ProxyClass(Object obj) { this.obj = obj; }
10         public Object invoke(Object proxy, Method m, Object[] args) throws Throwable {
11                 try {
12                         /* Código do proxy ... */
13                 } catch( InvocationHandlerException ihe) {
14                         throw e.getTargetException();
15                 } catch(Exception e) { throw e; }
16                 return algumValor;      /* Retorno da função */
17         }
18         static public Object newInstance(Object obj, Class[] interfaces) {
19                return java.lang.reflect.Proxy.newProxyInstance
20                       (obj.getClass().getClassLoader(), interfaces,
21                        new ProxyClass(obj));
22         }
23 }
24
25 import java.lang.reflect.Proxy;
26
27 public class Middleware {
28         /* ... */
29         ProxyInterface pi = (ProxyInterface)ProxyClass.newInstance(obj,
30                                                 new Class[]{ProxyInterface});
31         /* ... */
32 }
```

Figura 3.7: Esquema de um objeto *proxy* (java.)

Entre as linhas 3 e 5 têm-se a interface do *proxy*, enquanto que entre 7 e 23 estão as linhas correspondentes ao esqueleto do mesmo. O comentário na linha 12 marca onde deve

ser implementado o código referente ao *proxy*, ou seja, sua funcionalidade. Finalmente, as linhas de 25 a 32 ilustram como se faz a chamada ao *proxy*, e normalmente residem em arquivo separado, estando agrupadas a título de simplificação. *ProxyClass* implementa a *InvocationHandler*, de *java.lang.reflect.InvocationHandler*, que *java.lang.reflect.Proxy* utiliza para construir o objeto *ProxyClass*.

A construção é feita pelo método *Proxy.newProxyInstance()*, linha 22.

Este objeto é carregado pelo Carregador de Classes *ClassLoader* juntamente com o objeto cliente ou servidor original. É o *Proxy* que, interceptando as chamadas, verifica se uma determinada ação é válida. Isto é feito dentro do método *invoke()* do *Proxy*, linhas 12 a 26, quando ele consulta o gerenciador de segurança para saber se pode executar uma chamada de método ou não. No lado do cliente, o *Proxy* simplesmente consulta o gerenciador de segurança a fim de saber se pode ou não executar uma invocação de método. No lado do servidor, além desta atribuição ele também implementa a lógica de atribuição de acesso a clientes baseados na sua identificação, isto é, de acordo com a identidade do cliente ele procede ou não com quaisquer verificações posteriores.

A seguir, têm-se o *BodyGuard*, que representa uma abstração de serviços, efetivamente executados pelos *SpecializedBodyGuards*. Estes últimos são representados pelos diversos Controladores de Acesso e Gerenciadores de Segurança que podem ser atribuídos a um aplicativo. Não representados na figura 3.6, existem associados ao *middleware* uma classe derivada de *Permission*, que representa um conjunto de permissões do sistema, assim como um encapsulamento deste conjunto de permissões dentro de um *ProtectionDomain* - um domínio de proteção. Isto é feito pelo carregador de classes antes da carga dos objetos cliente e servidor e seus *proxies*.

Os *SpecializedBodyGuards* têm código disperso entre os vários sub-componentes do *middleware*, como por exemplo *ClassLoader*, *SecurityManager*, *AccessControler*, *Permission*, *ProtectionDomain*, etc. O agrupamento lógico é apenas conveniente para o entendimento de sua função, assim como o ocultamento das classes que não são primordialmente referenciadas na figura. Estas classes são detalhadas no capítulo 4.

Por fim, têm-se o componente *Transporter*, representado pela camada do RMI. O *middleware* interage com esta camada de duas formas: registrando o servidor no serviço de nomes *RMIRegistry* e associando, se necessário, um *socket* particular em substituição ao nativo. Para o protótipo desenvolvido, dois *sockets* foram desenvolvidos, um com capacidade de compressão de dados e outro com capacidade mínima de criptografia, a fim de validar a idéia de trasmissão segura de dados via rede.

3.4.2 Segurança no Nível de Métodos

No nível de métodos, o aspecto de segurança abordado é o controle baseado em direitos de acesso.

A implementação deste aspecto pode ser feita de duas formas: usando um *proxy* sobre o servidor ou usando interfaces distintas.

Na primeira solução, o *proxy* implementa a interface do servidor para interceptar todas as invocações de métodos dos clientes. O *proxy* é registrado no serviço de nomes, e o *stub* correspondente é instalado no cliente, sendo capaz de obter informações locais (isto é, propriedades do ambiente de execução e do programa que invocou os métodos). Quando um cliente invoca um método no servidor, o *proxy* intercepta a chamada, identifica o cliente e procede com o controle de acesso.

A vantagem desta solução é que o controle de acesso é totalmente transparente ao servidor. A título de ilustração, o exemplo da figura 3.8 mostra que é possível informações

tanto a respeito da aplicação cliente (com uso de introspecção), quanto do ambiente de execução, da máquina virtual (pilha de execução) e do sistema operacional (*sysP*).

```
1  class ProxyAgenda implements InvocationHandler,Serializable {
2          private Object obj;             /* objeto servidor                */
3          static  Class objClass;         /* classe do objeto servidor      */
4          static  ProtectionDomain objPD; /* ProtectionDomain associado     */
5          static  ClassLoader objCL;      /* Carregador de classe associado */
6          static  Properties sysP;        /* Propriedades do sistema        */
7
8          public static Object newInstance(Object obj) {
9                  objClass=obj.getClass();
10                 objCL=objClass.getClassLoader();
11                 objPD=objClass.getProtectionDomain();
12                     return Proxy.newProxyInstance(objCL, obj.getClass().getInterfaces(),
13                                                   new ProxyAgenda(obj));
14         }
15         private ProxyAgenda(Object obj) {
16                 this.obj = obj;
17                 sysP     = System.getProperties();
18                 System.out.println("\n Criado proxy sobre o objeto: " +
19                                    "\n Classe: " + objClass.getName().toString() +
20                                    "\n ClassLoader: " + objCL.toString() +
21                                    "\n ProtectionDomain: " + objPD.toString() +
22                                    "\n Propriedades do sistema: "  + sysP );
23         }
24         public Object invoke(Object proxy, Method m, Object[] args) throws Throwable {
25                 Object result;
26                 try {
27                         System.out.println("Tentando criar/capturar exceção " );
28                         throw new Minha();
29                 } catch (Exception e){
30                         StackTraceElement[] pilha = e.getStackTrace();
31                         System.out.println("Identifica chamada do metodo: " +
32                                            m.getName()+ "\t");
33                         for (int i = 2; i<pilha.length; i++) {
34                                 String linha= pilha[i].toString();
35                                 System.out.println("Classe: " + pilha[i].getClassName() +
36                                 "\t" + "Método: " + pilha[i].getMethodName() + "\n" +
37                                 "     Origem: " + linha +
38                                 System.getProperties() );
39                         }
40         }
41         try {
42                 System.out.println("Antes do método " + m.getName());
43                 result = m.invoke(obj, args);
44         } catch (InvocationTargetException e) { throw e.getTargetException();
45         } catch (Exception e) {
46                 throw new RuntimeException("unexpected invocation exception: " +
47                                            e.getMessage());
48         } finally { System.out.println("Depois do metodo " + m.getName()); }
49         System.out.println("Retornou o resultado... " );
50         return result;
51 }
52 class Minha extends Exception {
53         Minha() { super(); System.out.println("Criada exceção " ); }
54 }
```

Figura 3.8: Código de um *proxy* sobre um servidor.

Entre as linhas 8 e 14, no momento da criação do *proxy*, algumas propriedades já são recuperadas, como a classe, o carregador de classes e o *ProtectionDomain* associados à classe servidora.

As linhas de 24 a 50 irão apresentar as informações relativas ao cliente, já que o *invoke* estará interceptando suas chamadas RMI via *stub*.

A desvantagem é a introdução de uma sobrecarga na invocação de métodos, representada pela indireção do *proxy*, já que a interceptação é feita a cada invocação.

A segunda solução utiliza interfaces distintas, isto é, o serviço de nomes oferece aos clientes interfaces distintas dos serviços do servidor, segundo a identidade dos clientes. Nesta solução, o servidor implementa interfaces diferentes, cada qual correspondendo a diferentes direitos de acesso de clientes. Cada cliente faz requisições sobre uma determinada interface, que pode restringir chamadas a métodos do servidor.

A vantagem desta solução é transferir parte do controle de acesso do servidor para o cliente, evitando que este faça requisições inválidas, já que somente conhecerá as interfaces de métodos aos quais tem direito.

Um exemplo desta abordagem é mostrado na figura 3.9, que adota três interfaces distintas para acesso ao servidor.

```
1  public interface IAgendaGet extends Remote
2  { public String get() throws RemoteException; }
3
4  public interface IAgendaPost extends IAgendaGet
5  { public void post(String ag_entry) throws RemoteException; }
6
7  public interface IAgendaDelete extends IAgendaPost
8  { public void delete(String ag_entry) throws RemoteException; }
9
10 class AgendaClientS {
11         IAgendaGet       iag = null;
12         IAgendaPost      iap = null;
13         IAgendaDelete    iad = null;
14         /* ... */
15
16         public AgendaClientS() {
17                 try {
18                         /* ... */
19                         String agendaP = "rmi://192.168.1.2/ProtectionProxy";
20                         System.setSecurityManager(new RMISecurityManager());
21                         ipp = (IProtectionProxy)Naming.lookup(agendaP);
22                         /* ... */
23                 } catch (Exception e) {
24                         System.out.println("Exceção no lookup " + e.getMessage()); }
25                 try {
26                         remoteInter = ipp.clientID("CLIENT-ID");
27                         String look = "rmi://192.168.1.2/" +
28                                       remoteInter.toString().substring(10);
29                         iag = (IAgendaGet)Naming.lookup(look);
30                         String r = iag.get();
31                         iap = (IAgendaPost)iag;
32                         iap.post(r);
33                  } catch(Exception e) {
34                         System.out.println("Exceção no retorno " + e.getMessage());
35                 }
36         }
37         /* ... */
38         public static void main(String args[]) {
39                 /* ... */
40                 AgendaClientS client = new AgendaClientS();
41                 /* ... */
42         }
43 }
```

Figura 3.9: Código implementando três interfaces distintas para o servidor.

Entre as linhas 1 e 8 estão as três interfaces que o servidor exporta para os seus clientes, correspondendo a três níveis de acesso (*delete* tem mais privilégios do que *post*, que por sua vez é mais privilegiado do que *get*). Entre as linhas 10 e 43 está o código do cliente. Neste exemplo, o cliente conhece as três interfaces (linhas 11 a 13). A seguir, identifica-se perante o *proxy* do servidor (linhas 26 a 28). Tendo se identificado, ele solicita a interface cujo nome o *proxy* lhe devolveu. Por simplicidade, omitiu-se o

código que testa o nome, e a título de exemplo, suponha-se que lhe tenha sido dado o menor dos privilégios, isto é, lhe tenha sido retornado a interface correspondente a *get*. O cliente conecta-se ao servidor através desta interface (linha 29), e uma eventual chamada remota a este método, exemplificado na linha 30, sucederá. Entretanto, uma tentativa de escalação de privilégios, como exemplificado nas linhas 31 e 32, falhará e gerará uma exceção.

A desvantagem é que o código do cliente precisa ser modificado, pois ele não mais poderá invocar métodos sobre a interface original do servidor, apenas das subpartes desta. Outra desvantagem é que esta solução é inerentemente hierárquica, significando que ela é melhor aproveitada nos cenários em que os privilégios dos métodos seguem alguma forma de hierarquia. Caso isto não ocorra, o servidor novamente é onerado em cuidar dos privilégios de cada cliente, perdendo-se a vantagem inicial.

3.5 Conclusão

Este capítulo mostrou a adaptação do padrão *BodyGuard* para um padrão de projeto seguro, utilizando o padrão de projeto *Proxy*, base da arquitetura do *middleware* RMI.

Ao estender a arquitetura RMI, a solução proposta abre alternativas para a composição de uma variedade de serviços especializados de segurança, com distintas granularidades. Estas variam desde a segurança de aplicações até a segurança de métodos.

4 IMPLEMENTAÇÃO

Para validar a idéia do padrão proposto, foi construído um protótipo com as funcionalidades mínimas necessárias. Neste capítulo, é descrito de maneira sucinta a implementação dos componentes do protótipo, iniciando pelas principais classes, seguido da interrelação entre elas. Inicialmente, uma visão geral da implementação e seu funcionamento são apresentados, ao que seguem as seções apresentando uma breve descrição dos papéis que as classes têm no contexto geral seguido dos principais trechos de código da implementação.

4.1 Visão Geral da Implementação

O diagrama da figura 4.1 ilustra a colaboração entre os principais componentes (classes) do *middleware*. Tanto do lado cliente quanto do lado servidor, a seqüência de ações é semelhante, mudando apenas o teor e as conseqüências.

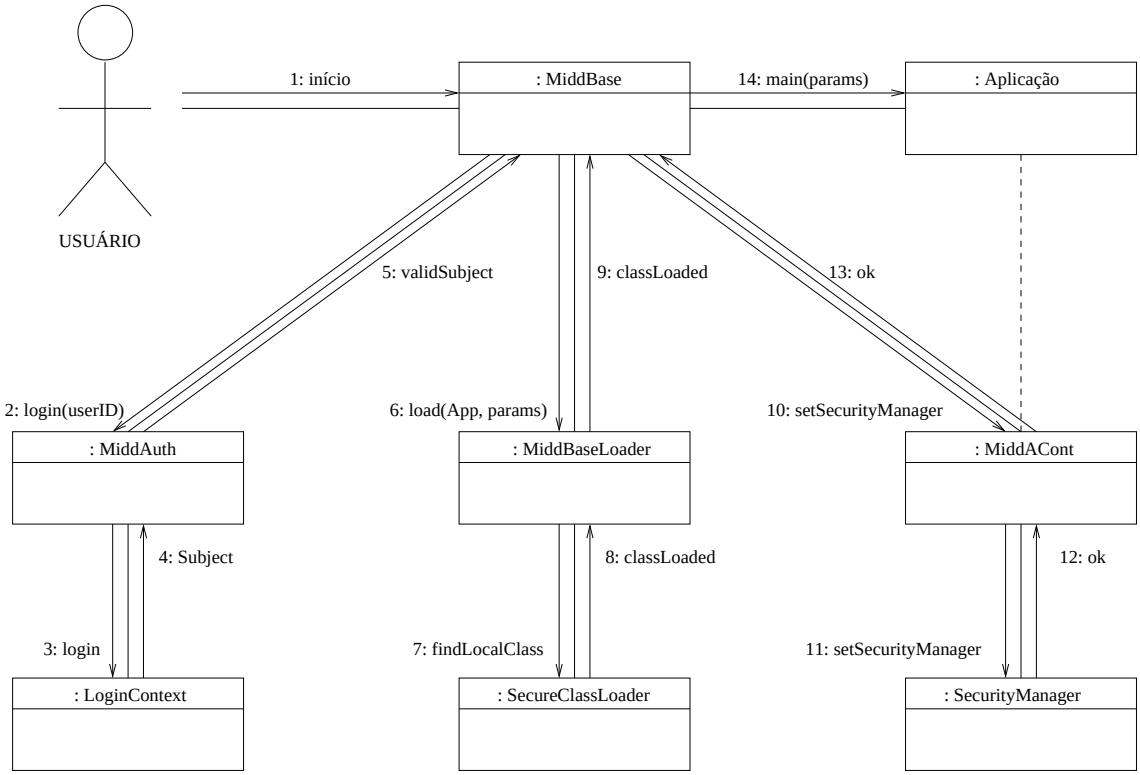

Figura 4.1: Diagrama de colaboração entre as classes principais.

Inicialmente, o *middleware* é disparado pelo usuário, e não mais a classe base (cliente ou servidor). A classe que agora serve de base a todo sistema, e a que deve ser invocada pelo usuário, é a *MiddBase*, responsável pela inicialização do *middleware* e classes da aplicação. Esta inicialização, vista na figura, procede em etapas, a começar pela autenticação do usuário.

A autenticação é feita pela classe *MiddAuth*, que por sua vez pode invocar classes subjacentes para concluir sua tarefa. A autenticação pode ser tão simples quanto obter a identificação do usuário a partir das propriedades do sistema (via *SystemProperties*), quanto complexa como obter via *loginContext* ou JAAS.

Após autenticado, *MiddBase* instancia o carregador de classes *MiddBaseLoader*, que procede à carga de todas as classes do cliente (ou servidor). Como ilustrado, esta classe deriva de *SecureClassLoader*, o que significa que pode tomar decisões a partir da origem e signatários de código remoto.

O próximo passo é o estabelecimento das políticas de segurança. Caso tenha sido configurado para tal, *MiddBase* pode associar *sockets* seguros ao RMI ou verificar a assinatura da classe cliente carregada. Mas o principal desta etapa é associar o *SecurityManager* do *middleware*, pois será ele que fará as verificações de segurança solicitadas pelo *proxy* da aplicação.

Só após a conclusão desta etapa é que a aplicação finalmente é disparada, isto é, invocado seu método principal.

4.2 Funcionamento do Padrão de Segurança

O padrão de segurança, em sua dinâmica, inicia antes da aplicação, como visto na seção anterior. Para atingir seu objetivo, o *middleware* segue uma rotina padronizada de operações.

4.2.1 Carregamento de Classes

O primeiro passo é derivar o carregador de classes para a aplicação. Usualmente, aplicações Java utilizam ou o carregador nativo *ClassLoader*, também referenciado como carregador interno ou padrão, ou a versão segura deste, *SecureClassLoader*, ambos oferecidos pela plataforma. *Applets* se beneficiam de um carregador próprio, implementado pelos navegadores, ou então, de carregadores derivados de outro carregador nativo da plataforma, o *RMIClassLoader*. O carregador seguro não é utilizado por si só, mas serve de base para a construção de carregadores especializados. Este carregador forma a base de praticamente todos os outros, pois além de não ser possível derivar um carregador a partir do padrão, ele também associa domínios de proteção (*ProtectionDomains*) à cada uma das classes que carrega. Estes domínios incorporam código e permissões, que se interpreta como sendo as permissões atribuídas a cada código (classe). Código neste contexto refere-se à classes que tenham uma origem (*CodeSource*) comum, o que em última análise é uma forma de crédito que se dá ao autor do código, isto é, à quem o disponibiliza.

Há outros dois carregadores de interesse, o *RMIClassLoader* e o *URLClassLoader*, que buscam código remoto para execução. O primeiro não pode ser instanciado, devendo-se usar seus métodos estáticos, e o segundo é uma implementação plenamente funcional de carregador, sendo bastante utilizado nas aplicações a partir da versão 1.2 de Java.

Para os fins do *middleware*, o que se quer é prover um carregador adaptável, e a escolha recai em uma derivação do *SecureClassLoader* ou na utilização do *URLClassLoader*. A escolha foi derivar a partir do primeiro e utilizar, dentro deste, o segundo como carre-

gador de classes remotas. Isto permite submeter à verificação tanto classes locais quanto remotas, de forma simples e uniforme. Há também um pequeno ganho em desempenho quando se carrega classes locais, pois não será necessário carregar um *URLClassLoader*.

Apesar de não ser mandatório, é nesta etapa que ocorre o processo de autenticação, que pode ser feito tanto por mecanismos simples da plataforma, como propriedades, quanto por mecanismos mais elaborados, como os *LoginContext()* (e posteriormente utilizando *Subject.doAs()*).

O carregador de classes é responsável não só pelo carregamento individual de todas as classes, mas pela forma em que o fizer, determinará a política global de segurança. Isto se dá pela escolha das classes que compõe a visão particular do *middleware* para a aplicação em questão.

A figura 4.2 mostra as partes significativas de *MiddBase.java*.

```
1  public class MiddBase implements Runnable {
2      final static int argc = 2;
3      ClassLoader cl;              /* ClassLoader associado. */
4      private Object argv[];       /* lista de argumentos da classe carregada. */
5      private String className;    /* nome da classe a ser carregada. */
6
7      MiddBase(ClassLoader cl, String className, Object argv[]) {
8          this.cl = cl; this.className = className; this.argv = argv;
9      }
10     void invokeMain(Class classe) {
11         Class argList[] = new Class[] { String[].class };
12         Method mainMethod = null;
13         try { mainMethod = classe.getMethod("main" , argList);
14         } catch (NoSuchMethodException nsme) {
15             /* ... */ ; System.exit(-1);
16         }
17         try { mainMethod.invoke(null, argv);
18         } catch (Exception e) {
19             /* ... */
20         }
21     }
22     public void run() {
23         Class target = null;
24         try {
25             target = cl.loadClass(className);
26             invokeMain(target);
27         } catch (ClassNotFoundException cnfe) {
28             System.err.println("Não foi possível carregar: " + className); }
29     }
30     static Object[] getArgs(String argv[]) {
31         /* obtém lista de argumentos para a aplicação */
32     }
33     public static void main(String argv[]) throws ClassNotFoundException {
34         MiddAuth aut = new MiddAuth();
35         /* inicializações */
36         if (!aut.login()) System.exit(-1);
37         Class self = Class.forName("MiddBase");
38         System.setSecurityManager(new MiddACont());
39         MiddPerm mp = new MiddPerm("*", "post, delete");
40         MiddBaseLoader parent = new MiddBaseLoader(args[0],
41         MiddBaseLoader jrl = new MiddBaseLoader(args[0], parent);
42         ThreadGroup tg = new ThreadGroup("MiddBase ThreadGroup");
43         Thread t = new Thread(tg, new MiddBase(jrl, args[1], getArgs(args)));
44         t.start();
45         /* ... */
46     }}
```

Figura 4.2: Principais trechos de código de *MiddBase.java*

A linha 36 indica o ponto onde se procede à autenticação, enquanto que na linha 38 se estabelece o gerenciador de segurança apropriado. O código ilustrado é o mínimo neces-

sário ao funcionamento do *middleware*. A execução da classe da aplicação ocorre entre as linhas 22 e 29. Como pode ser visto pelas linhas 40 a 44, a aplicação está confinada a um grupo de *threads* com raiz em *MiddBase*. Se for necessário comunicação segura, nesta etapa pode-se configurar os *sockets* do serviço RMI, escolhendo-se soluções alternativas, via *RMISocketFactory*, pelo que se pode associar transparentemente à aplicação algoritmos de criptografia nativos ou customizados. No caso de se querer instalar *sockets* seguros, a chamada ao construtore deste *socket* deveria ser feita após a linha 38.

4.2.2 Autenticador

A classe *MiddAuth*, vista na figura 4.3, é responsável por autenticar usuários. Foram testadas duas maneiras de autenticação, uma que valida unicamente contra as credenciais do usuário no sistema, e outra que permite ao desenvolvedor estabelecer um critério próprio. O primeiro é previsto para aplicações que não demandem segurança exigente, e é baseado na verificação de propriedades do sistema, semelhante ao que se pode obter via (*user.name*) através de *getProperty()*. O segundo autenticador foi implementado com base na arquitetura JAAS (*Java Authentication and Authorization Service*) (LAI, C. et al., 1999). Dentro desta chamada pode-se implementar qualquer esquema próprio de autenticação, em caso de necessidade, como por exemplo, uma chamada do tipo *callback*, suportado pelo JAAS. O código a que se refere a figura 4.3 é o de autenticação via JAAS.

Essencialmente, JAAS associa um *Principal* (usuário) a um *Subject* (grupo de nomes representativos deste usuário). A autenticação, neste caso, consiste em provar que o um *Subject* demonstre sua identidade, via uma senha ou uma assinatura digital, por exemplo. Esta arquitetura apresenta uma série de detalhes complexos, que fogem ao escopo deste trabalho, mas é interessante notar uma característica, a de que é baseada em PAM (*Pluggable Authentication Modules*), o que a faz suportar uma arquitetura onde é possível ao administrador do sistema fornecer os módulos de autenticação adequados, além de suportar o empilhamento de módulos. A arquitetura também provê meios para que as aplicações não se tornem dependentes da arquitetura de autenticação subjacente.

No caso da classe *MiddAuth*, JAAS é utilizado para autenticar um usuário contra um sistema *Unix*, mas há suporte nativo da plataforma para servidores *Windows NT* e *Kerberos*. Isto é feito pela seleção do módulo de *login* apropriado, no caso *UnixLoginModule* (os outros são *NTLoginModule* e *Krb5LoginModule*, respectivamente). O arquivo *jaas.config*, visto na figura 4.4, contém esta configuração, enquanto que o *jaas.policy*, na figura 4.5, contém as políticas de segurança associadas (no caso, conexão de rede e escrita em arquivo temporário). *MiddAuth* utiliza controle baseado em usuários, mas não há obrigatoriedade do ponto de vista do JAAS. Outros controles possíveis podem baser-se em papéis (*roles*) ou grupos.

Os métodos importantes de *MiddAuth* são *login()*, que procede à autenticação do usuário, *getSubject()*, que retorna sua credencial e *logout()*, que remove as credenciais associadas.

A classe *MiddBase* invoca o *login()* antes de qualquer outra coisa, pois é preciso certificar-se que o usuário é válido. A seguir, obtém o *Subject*, pois este será necessário para outras operações de autorização, e, ao finalizar a aplicação, deve chamar *logout()* para liberar as credenciais. Este último método também poderia ser chamado antes do término da aplicação, no momento em que esta não mais precisasse de recursos protegidos. Isto aumentaria a proteção do sistema, pois o *middleware* não mais permitiria qualquer operação baseada em autorização. Entretanto, a aplicação desta política é dependente de aplicação.

```
1  public class MiddAuth {
2      LoginContext loginContext = null;        /* contexto de login               */
3      Subject subject = null;                /* subject - usuário associado       */
4
5      public MiddAuth() {
6          try {
7              LoginContext loginContext = new LoginContext("MiddAuth");
8          } catch (LoginException loginException) {
9              loginException.printStackTrace(); System.exit (-1);
10         }
11     }
12     public void login() {
13         try {
14             loginContext.login();
15             subject = loginContext.getSubject();
16         } catch (LoginException loginException) {
17             loginException.printStackTrace(); System.exit (-1);
18         }
19     }
20     public void logout() {
21         try {
22             loginContext.logout();
23         } catch (LoginException loginException) {
24             loginException.printStackTrace(); System.exit (-1);
25         }
26     }
27     public Subject getSubject() { return subject; }
28 }
```

Figura 4.3: Principais trechos de código da classe autenticadora.

```
1  MiddAuth {
2     com.sun.security.auth.module.UnixLoginModule required debug=false;
3  };
```

Figura 4.4: Exemplo de configuração utilizada pelo JAAS.

```
1  grant codeBase "file://home/java/testes/-",
2     Principal com.sun.security.auth.UnixUserPrincipal "java" {
3     permission java.io.FilePermission "/tmp/privilegedFile.txt", "write";
4     permission java.io.FilePermission "/tmp/privilegedFile.txt", "read";
5     permission java.net.SocketPermission "192.168.1.2", "accept, listen";
6  };
```

Figura 4.5: Exemplo de política de segurança utilizada pelo JAAS.

A vantagem de se utilizar autenticação com módulos que validam usuários do sistema subjacente, é que esta classe pode ser utilizada sem modificação tanto do lado cliente quanto do servidor, independente do conjunto de usuários de um e de outro. Bases próprias de autenticação podem requerer a implementação de um módulo *LoginModule* específico, tarefa que foge ao escopo deste trabalho. Além disso, bases de usuários próprias poderiam implicar o estabelecimento de sessões, onde o cliente estabelece uma pré-sessão de autenticação com o servidor, o que é indesejável, pois aplicações RMI usualmente não comportam este tipo de mecanismo.

Observe-se que *MiddAuth* obtém, logo após o *login*, o *subject* associado. Este *subject* será requerido pelo gerenciador de segurança e/ou controlador de acesso, posteriormente.

Uma vez autenticado e de posse das credenciais, o *middleware* procede ao estabelecimento das políticas de segurança.

4.2.3 Estabelecimento das Políticas de Segurança

A classe *MiddACont* estabelece as políticas de segurança do sistema, isto é, o que a aplicação pode ou não executar. Para este fim, é necessário estabelecer permissões, que por sua vez são dependentes da aplicação que se queira suportar, assim como o gerenciador de segurança e/ou controlador de acesso.

4.2.3.1 Permissões

A figura 4.6 ilustra um exemplo de permissão, que pode ser atribuído a uma aplicação.

```
1  public class MiddPerm extends Permission {
2      protected int grade;
3      static private int POST = 0x01;
4      static private int DELE = 0x02;
5
6      public MiddPerm(String nome) {
7          this(nome, "post");
8      }
9      public MiddPerm(String nome, String oque) {
10         super(nome); parse(oque);
11     }
12     private void parse(String oque) {
13         StringTokenizer st = new StringTokenizer(oque, ",\t ");
14         grade = 0;
15         while (st.hasMoreTokens()) {
16             String token = st.nextToken();
17             if (token.equals("post")) grade = grade | POST;
18             else if (token.equals("delete")) grade = grade | DELE;
19             else throw new IllegalArgumentException("Ação inválida: " + token);
20         }
21     }
22     public boolean implies(Permission perm) {
23         if (!(perm instanceof MiddPerm)) return false;
24         MiddPerm mp = (MiddPerm)perm;
25         String nome = mp.getName();
26         if (!nome.equals("*") && !name.equals(mp.getName())) return false;
27         if ((grade & mp.grade) != (mp.grade)) return false;
28         return true;
29     }
30     public boolean equals(Object obj) {
31         if (!(obj instanceof MiddPerm)) return false;
32         MiddPerm mp = (MiddPerm)obj;
33         return ((mp.getName().equals(getName())) && (mp.grade == grade));
34     }
35     public String getActions() {
36         if (grade == 0) return "";
37         else if (grade == POST) return "post";
38         else if (grade == DELE) return "delete";
39         else if (grade == (POST | DELE)) return "post, delete";
40         else throw new IllegalArgumentException("Máscara de permissão desconhecida");
41     }
42         /* ... */
43     public int hashCode() { return getName().hashCode() ^ grade; }
44 }
```

Figura 4.6: Exemplo de permissões concedidas a uma aplicação.

Uma Permissão sempre implementa, no mínimo, os métodos ilustrados (além do agrupamento em *Collection*, não mostrado por simplificação). Este exemplo corresponde à lógica dos exemplos das figuras 3.8 e 3.9, no que diz respeito à hierarquia de privilégios. Observe-se que *get* não está representado, ilustrando o que seria uma permissão inerente (e mínima) de uma aplicação. A atribuição desta permissão a uma aplicação pode ser vista na linha 39 da figura 4.2.

4.2.3.2 Gerenciador de Segurança

O Gerenciador de Segurança, *SecurityManager*, implementa uma das funções primordiais no mecanismo de segurança da plataforma Java. É ele quem controla se uma determinada classe pode ou não executar uma operação. Operação, neste contexto, pode ser o estabelecimento de uma conexão de rede, alteração de estado de uma ou um grupo de *threads*, por exemplo.

Usualmente, aplicações Java não possuem gerenciador associado, enquanto que *applets* possuem os mais restritos possíveis. Outra característica deste componente é que, uma vez instalado, não pode ser substituído, inviabilizando a escalação de privilégios via código.

Quando uma aplicação é desenvolvida tendo em vista o uso do gerenciador, ela deve consultá-lo antes de tentar uma operação potencialmente bloqueável. Entretanto, quando a aplicação não foi assim projetada, que é o foco do *middleware* de segurança, este é quem deve interceptar o método via reflexão e proceder à verificação. Embora simples, este método de operação introduz um problema sutil, e sobre o qual ainda há discussão entre as soluções: as exceções geradas pelo *middleware*.

As verificações que um gerenciador de segurança pode efetuar são as mais variadas, e na literatura enquadram-se nos seguintes grupos:

- Operações sobre arquivos, ou seja, acesso a arquivos em disco e em redes locais.
- Operações sobre conexões à rede, através da abertura de *sockets* a outras máquinas. Conceitualmente, o controle é feito sobre a comunicação, ou estado da conexão, dependendo do ponto de vista, mas o efeito prático recai sobre os *sockets* em algum momento.
- Proteção da máquina virtual e do próprio gerenciador de segurança, visando principalmente evitar que os métodos de proteção sejam sobrescritos com código remoto.
- Proteção de *threads*, pelo que se espera que *threads* não-relacionadas não possam interferir umas com as outras. *Threads* relacionadas, neste contexto, significam as que estão dentro de um mesmo grupo ou que sejam filhas na hierarquia.
- Proteção de recursos do sistema, incluindo propriedades globais de ambiente.
- Proteção a aspectos de segurança, ou seja, proteção aos próprios mecanismos de segurança da plataforma, tais como acessos a métodos, propriedades e classes, além de informações sobre certificados.

Para fins do protótipo, apenas alguns exemplos foram desenvolvidos, já que este componente da arquitetura é o mais flexível de todos, e deve ser customizado a cada aplicação que o *middleware* servir.

4.2.3.3 Controlador de Acesso

Até aqui, apenas o gerenciador de segurança foi descrito, mas dentro da arquitetura de segurança Java, o controlador de acesso *AccessController* é quem realmente aplica as políticas de segurança do primeiro.

O papel do controlador é o mesmo do gerenciador: determinar se uma operação pode ou não ser executada, sendo às vezes considerado redundante (OAKS, 1998). Antes da

versão 1.2 de Java, o gerenciador continha a lógica necessária para decidir pelas permissões. Posteriormente, foi permitido que lógica passasse ao controlador, eliminando a necessidade programática de estabelecer políticas de segurança, já que agora estas podem ser definidas em arquivos textuais. Uma das razões pelas quais se mantém ambos é a compatibilidade com código legado, e a outra é que agora uma nova divisão no grão de controle é possível: características de controle mais gerais podem ser feitas via gerenciador, que em geral é mais difícil de ser programado, e características de controle fino podem ser feitas pelo controlador, facilmente estensível, do ponto de vista programático. Do ponto de vista de estabelecimento de políticas de segurança, cabe lembrar que um controlador não estará ativo enquanto o correspondente gerenciador não estiver, e aquele não é inicializado enquanto não for chamado.

O controlador opera através da chamada aos seus métodos estáticos e, em particular, interessam diretamente ao *middleware*: *checkPermission()*, *beginPrivileged()* e *endPrivileged()*.

O primeiro método é quem efetivamente verifica se uma determinada permissão foi concedida ou não à aplicação. No meta-nível, esta operação é realizada para cada método interceptado e que precise realizar operações sensíveis. Nos níveis mais baixos da arquitetura de segurança Java, este método nada mais faz do que avaliar os *ProtectionDomains* que estejam ativos no momento. Um efeito deste comportamento é a impossibilidade de modificar automaticamente os privilégios de uma *thread* de execução, no sentido de aumentá-los. Tal modificação só pode ser feita se os demais componentes do *middleware* assim o permitirem e contiverem o código necessário para mudar a classe *Policy* vigente.

begin e *endPrivileged()* marcam o início e o fim, respectivamente, de um bloco de código o qual executa com os privilégios de quem o chamou.

Não explorado pelo *middleware*, existe a possibilidade de encapsular permissões e seus controladores de acesso em um único objeto, juntamente com os objetos a serem protegidos, os *GuardedObjects*. Esta é uma forma transparente de prover controle de acesso a métodos de um objeto, porém pode se tornar complexa e onerosa dependendo do tipo de controle desejado. Por exemplo, interceptar todos os métodos de uma classe e proceder ao controle de acesso individualmente via *GuardedObjects* é mais oneroso do que prover um único método verificador, e invocá-lo a cada interceptação.

4.2.3.4 Conceitos Envolvidos no Controlador de Acesso

Estabelecer um controlador de acesso envolve conhecer quatro elementos associados, que formam a estrutura de controle efetiva do sistema: *CodeSources*, *Permissions*, *Policies* e *ProtectionDomains*.

O elemento *CodeSource* nada mais representa do que a origem de uma determinada classe, ou seja, sua localização em forma de URL (*Uniform Resource Locator*). O *CodeSource* de uma classe é estabelecido pelo carregador de classes, antes de ser instanciada.

Os *Permissions* são os elementos de nível mais baixo na hierarquia de permissões de um objeto, onde de fato o controlador opera. Em si, *Permission* é uma classe abstrata representativa de uma operação, o que se dá em dois momentos: quando associada a uma classe e um *CodeSource*, representando uma permissão concedida, ou na forma de um objeto ao qual se pode consultar sobre ter ou não uma permissão.

Embora o *middleware* possa utilizar as duas formas de atribuição de permissões, a programática e a estabelecida em arquivos textuais, somente a segunda foi necessária até a elaboração do protótipo. A programática, isto é, derivar classes de *Permission*, tem utilidade quando se quer dinamicamente revogar algum direito da aplicação, por exemplo,

suspender acesso a arquivos ou rede após algum evento, e foi ilustrada na seção 4.2.3.1.

Policies são os componentes que permitem associar permissões a *CodeSources*. Este componente requer um pouco mais de cuidado, pois embora opere de forma semelhante ao gerenciador, pode ser substituído dinamicamente, desde que concedido pelo gerenciador via propriedades. Os *Policies* são utilizados como recurso dinâmico de alteração de políticas de segurança e, contrário aos *Permissions*, podem ser usados para aumentar privilégios até um máximo pré-definido. Por exemplo, pode-se permitir que uma aplicação tenha acesso a uma hierarquia maior de diretórios substituindo-se o arquivo *policy* associado à aplicação. O máximo pré-definido é a intersecção dos privilégios da JVM no sistema computacional com a política estabelecida pelo *SecurityManager* associado à aplicação. No exemplo anterior, isto significa que um usuário não pode acessar arquivos de outro sem que o sistema de arquivos assim o permita (privilégio da JVM), e que não poderia ascender na hierarquia de diretórios, se o *SecurityManager* não permitir a carga do arquivo *policy* apropriado (política do gerenciador).

O *middleware* concentra-se no estabelecimento de políticas estaticamente, pois a maioria dos desenvolvedores e administradores está mais familiarizado com este tipo de configuração. Porém, nas classes internas há previsão para mudança de políticas, traduzida em código pelas chamadas à *getPolicy()* e *setPolicy()*.

Por fim, há os *ProtectionDomains*, que representam agrupamentos de permissões concedidas a *CodeSources*. Uma classe só pode pertencer a um domínio, que lhe é designado pelo carregador na definição da classe. Os domínios não são explorados em profundidade pelo *middleware*.

4.2.3.5 Políticas de Segurança no Middleware

Por fim, é neste passo que são configuradas as políticas de autorização da aplicação, via *SecurityManager* e *AccessController*. A rigor, o estabelecimento de uma política de segurança baseada em *SecurityManager* não precisa ser feita aqui, diminuindo o tempo de carga da aplicação. Poderia ser feita mesmo com a aplicação já carregada e iniciada, em algum momento durante uma interceptação de método. Entretanto, como a característica do gerenciador é não poder ser substituído uma vez operacional, o benefício de se ter ações controladas desde antes da execução do aplicativo quase sempre compensa o pouco ganho que se obtém no tempo de carga. O fato de não poder ser substituído somado ao de estabelecê-lo antes da carga da aplicação também evita que a aplicação estabeleça um gerenciador próprio.

O gerenciador de segurança/controlador de acesso é instalado na linha 38 da figura 4.2, e seu uso, a partir de então, é feito pelo *proxy* da aplicação como ilustra a figura 4.7. Entre as linhas 2 e 8 o controlador é usado diretamente pelo *proxy*, enquanto que o código entre as linhas 11 e 19 usa indiretamente via *SecurityManager*. Neste último caso, o código da linha 25 deve substituir o código da linha 38 em 4.2.

4.2.4 Carregamento da Classe Cliente

Na etapa seguinte, o arquivo da aplicação cliente é carregado e verificado. Esta verificação pode ou não ser feita, e seu método pode ser variado pelo desenvolvedor. A forma mais simples de verificação consiste em calcular a sua soma de verificação (*checksum*) e compará-la com uma base pré-calculada, reduzindo a possibilidade de se carregar classes modificadas. Após a verificação, o arquivo é instanciado como uma classe e invoca-se o seu método principal.

A partir de então, o processamento ocorre normalmente, do ponto de vista da aplica-

```
1  /* uso do controlador de acesso */
2  MiddPerm mp = new MiddPerm("*", "delete");
3  try {
4          AccessControler.checkPermission(mp);
5          app.delete();
6  } catch (AccessControllerException ace) {
7          System.out.println(ace);
8  }
9
10 /* uso do gerenciador de segurança */
11 public class MiddSMan extends SecurityManager {
12         public void checkDelete() {
13                 try {
14                         MiddPerm mp = new MiddPerm("*", "delete");
15                         AccessControler.checkPermission(mp);
16                 } catch (AccessControllerException ace) {
17                         System.out.println(ace);
18                 }
19 }
20
21 /* classe MiddBase */
22 /* ... */
23         System.setSecurityManager(new MiddSMan());
24 /* ... */
```

Figura 4.7: Usando o gerenciador de segurança/controlador de acesso.

ção, até o momento em que ela tenta executar uma operação para a qual o *middleware* tenha sido configurado a interceptar. Neste instante, o processamento é desviado para o meta-nível, via reflexão, e o *middleware* assume o controle. Este controle nada mais é do que o nível de autorização e controle de acesso, e pode ser tão simples como verificar uma permissão de leitura de arquivo quanto complexo quanto o estabelecimento de uma conexão segura com um servidor de recursos autenticado.

4.2.5 Carregamento da Classe Servidora

Apesar de se ter seguido primeiramente as etapas do lado cliente, no lado do servidor, que naturalmente deve ser disparado antes, os passos são semelhantes, exceto pelo fato de que verificações de autenticidade de clientes podem acontecer quando de uma solicitação remota. Neste caso, o servidor terá, além do controle aos seus próprios recursos, a tarefa de verificar a legitimidade do cliente solicitante. O método de verificação pode ser adaptado segundo a necessidade da aplicação, mas existem algumas maneiras que são suportadas nativamente na plataforma Java, e que podem auxiliar o desenvolvedor. Uma delas é a verificação através das propriedades do sistema, obtidas programaticamente através de classes representativas. Outra é através da troca de certificados digitais entre cliente e servidor. A primeira forma pode ser utilizada em ambientes nos quais a segurança não precisa atingir níveis extremos, e é relativamente simples de ser implementada. A segunda é bastante complexa, mesmo com o suporte oferecido pela plataforma, mas também resiste melhor a ambientes de rede hostis onde se requeira nível maior de segurança. Ambas as formas permitem uma granularidade mais fina do que clientes, ou seja, permitem usuários distintos em uma mesma máquina cliente.

Em retrospecto à arquitetura vista na figura 3.6, os *proxies* do cliente e do servidor estão localizados imediatamente abaixo das aplicações, e são classes que implementam o nível de indireção reflexivo. Os métodos dos *proxies* são responsáveis por decidir pela interceptação ou não de um método da classe base. A decisão pela interceptação é uma característica da aplicação, tomada pelo desenvolvedor quando do uso do *middleware*.

Esta estrutura permite que clientes e servidores se autentiquem mutuamente, ou seja, tanto o cliente pode ter certeza de que está se conectando a um servidor autêntico, quanto o servidor pode ter certeza de que o cliente que está solicitando serviços é legítimo. A flexibilidade se estende a usuários distintos que solicitam serviços a partir de uma mesma máquina cliente.

Os *SpecializedBodyGuards* são materializados pelas demais classes, controladores de acesso, gerenciadores de segurança, *policies*, *permissions* e pelas classes próprias do *middleware* que as compõem num conjunto coeso.

4.3 Carregador de Classes

O carregador de classes é um dos principais componentes da plataforma Java envolvidos na aplicação de políticas de segurança, juntamente com o gerenciador de segurança e o controlador de acesso. Entre as atribuições do carregador estão a colaboração com o gerenciado de segurança, no sentido de fornecer dados como a origem e a assinatura do código, se houverem, e a separação do espaço de nomes das classes, sem a qual, o modelo de segurança Java seria inviabilizado em vários aspectos.

A plataforma dispõe de um carregador nativo, ou primordial, implementado internamente à JVM, responsável por carregar as classes nativas. Enquanto o carregador primordial não pode ser derivado, Java oferece outros que podem: o *ClassLoader*, *AppletClassLoader*, *RMIClassLoader*, *SecureClassLoader* e *URLClassLoader*.

O primeiro representa é uma classe abstrata, que precisa ser implementada e apenas serve de base para se estender carregadores customizados. A segunda aplica-se a *applets*, sendo usualmente implementada por navegadores e aplicativos do tipo *appletviewer*. *RMIClassLoader* só pode ser utilizada através da invocação de seus métodos estáticos, isto é, não poder ser derivada, de forma que cedeu espaço à *URLClassLoader* nas versões recentes de Java. As duas últimas formam a base de quase todos os carregadores implementados desde a versão 1.2 de Java. *URLClassLoader* é derivada de *SecureClassLoader*, e geralmente utilizada para carregar código via rede e arquivos *.jar*.

O carregador de classes do *middleware* deriva de *SecureClassLoader*, pois além de código de rede, classes locais são carregadas, e neste caso, a carga direta representa menor custo. Trata-se da classe *MiddBaseLoader*, no arquivo *carregador/MiddBaseLoader.java*.

O método importante desta classe é *findLocalClass()*, que contém a lógica de carregamento das classes. Esta lógica simplesmente abre uma *URL* seguida do nome da classe desejada e a carrega. A *URL* pode ser qualquer uma que possa ser lida através de um *InputStream*, o que inclui classes locais. Na mesma função é definida a classe, juntamente com a sua origem (*CodeSource*), importante para o *SecurityManager* posteriormente decidir sobre a cessão de direitos ao código (invocação de métodos e acesso a recursos).

O método *checkPackageAccess()* verifica se o carregador tem os privilégios necessários para acessar a classe.

Para aplicações legadas da versão 1.1 de Java, a classe *MiddLoader* em *carregador/MiddLoader.java* contém o esqueleto de um carregador completo, que pode utilizar o mesmo código de *findLocalClass()* de *MiddBaseLoader*.

Delegação é usada para resolver classes que estão no CLASSPATH, assim como classes nativas. Usar delegação também tem impacto positivo sobre o desempenho, já que desonera o *middleware* de fazer a *cache* e o controle de classes já carregadas, deixando esta tarefa para o carregador nativo (classe *ClassLoader*) ou a máquina virtual.

```
1  public class MiddBaseLoader extends SecureClassLoader {
2      protected URL urlBase;        /* Localização da classe da aplicação.
3
4      public MiddBaseLoader(String base, ClassLoader parent) {
5          super(parent);
6          // Formata a String 'base'.
7      }
8
9      protected Class findLocalClass(String nome) {
10         byte buff[];
11         Class cl;
12         String urlNome = nome.replace('.', '/');
13         SecurityManager sm = System.getSecurityManager();
14
15         if (sm != null) {
16              // Verifica se pode definir 'packages'.
17         }
18         try {
19             URL url = new URL(urlBase, urlNome + ".class");
20             // Modifica url, se necessário, de acordo com conf.
21             InputStream is = url.openConnection().getInputStream();
22             // ...
23             CodeSource cs = getCodeSource(urlBase, (Certificate[])null);
24             cl = defineClass(nome, buff, 0, buff.length, cs);
25             return cl;
26         } catch (Exception e) {
27             System.err.println("Não foi possível carregar " + nome + ": "  + e);
28             return null;
29         }
30     }
31
32     byte[] getClassBytes(InputStream is) {
33         // Retorna array de bytes representativo da classe.
34     }
35
36     public void checkPackageAccess(String nome) {
37         // Consulta o Gerenciador de Segurança sobre criar 'packages'.
38     }
39 }
```

Figura 4.8: Esqueleto do código da classe carregadora.

4.4 Java *Plug-in*

Embora este trabalho não considere *plug-ins* Java, uma pequena explicação acerca deste mecanismo é pertinente, já que seu uso também pode se beneficiar de um *middleware*. A SUN, por volta de 2001, resolveu alguns dos problemas inerentes aos *applets* Java, tais como desempenho e independência de plataforma, principalmente entre os navegadores Internet Explorer e Netscape Navigator. Um dos pontos de divergência era a infraestrutura de segurança de cada navegador.

Como parte das soluções para os problemas dos *applets*, a SUN desenvolveu um componente que independe da plataforma de execução. Trata-se de um *plug-in*, a exemplo de outros conhecidos, como QuickTime e RealPlayer, que é associado a uma classe de objetos carregados via rede, e neste caso, por um navegador.

Este componente é responsável pelo tratamento de código associado a marcadores que tenham sido convertidos de <*applet*> em marcadores do tipo <*object*> e <*embed*>. Os navegadores, ao encontrarem este segundo tipo de marcadores, carregam o *plug-in* apropriado e lhe passam o controle sobre o objeto trazido da rede. Para programas Java, isto representa uma melhoria no sentido de se poder usar sempre a versão mais recente da plataforma, além de fazer pleno uso de código assinado, já que o novo componente suporta assinaturas/certificados RSA, o que não acontecia com o antigo DSA. Além disto,

agora é possível configurar dinamicamente o gerenciamento de entidades confiáveis, isto é, as autoridades de certificação (CA - *Certification Authority*).

Embora o componente substitua parte da máquina virtual de cada navegador, ela não é totalmente descartada. No caso de marcadores do tipo <*applet*>, os que ainda não foram convertidos, é ela que procede ao tratamento dos objetos.

Uma das vantagens de se usar este componente é a possibilidade de confinar código Java em objetos <*object*> dentro de páginas e permitir que navegadores processem como se fossem uma JVM, passando sobre algumas das limitações de *applets* no que diz respeito à segurança.

Foi desenvolvido um pequeno exemplo ilustrativo, que pode ser carregado via *middleware*. Neste exemplo, Java é utilizado para testar se um determinado servidor está respondendo a um determinado serviço (porta). É uma implementação tradicional de um *port scanner*, que apenas testa se uma determinada porta, e portanto, um determinado servidor/serviço está ou não ativo. Este código é disponibilizado na forma de código móvel assinado, e no modelo de *applets* não poderia ser executado em um navegador, já que ele se conecta a servidores que não o de origem. Entretanto, através do *plug-in*, o usuário pode optar entre permitir a execução ou não deste programa, dinamicamente.

Originalmente, o programa foi desenvolvido para testar a disponibilidade do serviço RMI de um servidor.

4.4.1 Exceções

Em aplicações Java, se uma operação não é permitida ou encontra uma condição excepcional ou errônea, tal indicativo deve ser propagado ao método que a invocou na forma de uma exceção. Caso a aplicação não resolva esta condição, em última instância, o usuário deve ser avisado, o que é feito abortando-se o aplicativo e indicando a pilha de execução.

Existem dois tipos de exceções em Java: as verificadas (*checked*) e as não-verificadas (*unchecked*). O primeiro tipo, também chamado de *runtime exception*, deve ser tratado no código, através de blocos *try*/*catch* ou repassadas para o nível seguinte na hierarquia de chamadas de métodos, via cláusula *throw*. Deste modo, o programador está ciente da sua existência, e pode optar entre tratá-la ou repassá-la até a classe que contém o método *main()*, onde, se não for tratada, aborta a aplicação. O segundo tipo não precisa ser obrigatoriamente tratada em código, e se não for, acaba por abortar a aplicação como no primeiro caso.

Muitas das violações de segurança geram exceções que correspondem ao primeiro tipo, e portanto precisam ser tratadas no código. Duas abordagens podem ser tomadas quanto ao tratamento: repassar a exceção ao usuário, no caso, o código que chamou o método relacionado à segurança, ou tratar localmente a exceção. Repassar a exceção ao método chamador torna o *middleware* simples, poupando a escrita de código. Entretanto, cria um problema para a aplicação, pois ela agora recebe uma exceção para a qual não foi programada, e a rigor não possui o código para manipular a condição. Ainda que ela tenha um bloco *catch* responsável por tratar exceções inesperadas, a inserção do *middleware*, que devia ser transparente ao usuário, acaba não sendo, e ainda pode deixá-lo sem saber como corrigir o problema.

Por outro lado, tratar localmente a exceção, embora desejável, nem sempre pode ser plenamente atingido. Por exemplo, quando o *middleware* insere uma etapa de autenticação e o usuário não a cumpre, uma exceção será gerada e tratada no próprio código que solicita as credenciais. Entretanto, o código não pode prosseguir executando a aplica-

ção, e esta precisa, de alguma forma, ser notificada. Uma possível maneira de notificar a aplicação é gerar, no código do *middleware*, a exceção correspondente ao que ela espera. Assim, se foi requerido autenticação para abrir um arquivo, por exemplo, o *middleware* pode gerar uma exceção relacionada a arquivos, mascarando o erro na autenticação. Claramente esta solução é indesejável, pois o usuário tentará corrigir o problema relacionado ao arquivo, quando de fato deveria solucionar o problema de autenticação.

Na arquitetura proposta, optou-se por tratar as exceções geradas internamente tanto quanto possível, e disparar a exceção de violação de segurança para a aplicação. A razão da escolha baseia-se em tornar visível ao usuário a real causa da falha ao tentar uma operação não permitida, ainda que custe o término da execução do programa, esperando-se que ele tenha condições de tratar o problema. Esta pressuposição considera que o usuário final conta com o suporte do administrador do sistema, o qual é alvo primário, juntamente com os desenvolvedores, desta arquitetura.

5 CONCLUSÕES E TRABALHOS FUTUROS

Tecnologias como programação orientada a aspectos (POA), reflexão computacional (RC) e padrões de projeto (PP) podem ser empregadas no domínio de tolerância a falhas (TF) como ferramentas de auxílio ao desenvolvimento de *software* seguro. Dentro desta categoria de programas, encontram-se os que são implementados como *middleware*, atuando entre o sistema operacional, ou máquina virtual, e os aplicativos que suporta.

Segurança não é um assunto novo em TF, assim como não o é em outros domínios de conhecimento. Porém, muitas vezes por não representar a funcionalidade da aplicação, é relegada a segundo plano, deixando-se sua implementação para etapas posteriores do desenvolvimento, quando não esquecida. A segurança neste caso faz parte dos requisitos não-funcionais da aplicação, a exemplo do que ocorre com persistência, replicação e outros que não são imediatamente visíveis ao usuário. A separação do requisito não-funcional segurança como um aspecto da aplicação permite que o desenvolvedor concentre esforços em resolver problemas comuns a vários domínios.

Para esta visão, a de requisito não-funcional, POA é uma tecnologia que pode ser empregada como modelo conceitual, pois modela justamente estes aspectos. Existem ferramentas que suportam o desenvolvimento de aplicações no modelo de POA, tais como AspectJ e DJLib. Estas ferramentas trabalham com modificação de *bytecodes*, agregando novas funcionalidades e gerando código intermediário compatível com as máquinas virtuais Java (JVM) mais difundidas.

Ao longo deste trabalho não foram utilizadas ferramentas de auxílio à POA, mas somente construções da linguagem e suporte da JVM. POA serviu como modelo conceitual, sendo a tecnologia de filtros de composição o modelo referencial mais próximo. A modelagem através de POA permitiu que o desenvolvimento aproveitasse o conhecimento de projetos afins, pelo que se concluiu a adequação desta tecnologia.

Padrões foram utilizados como elementos arquiteturais, visando reaproveitamento de soluções. Esta tecnologia contribuiu de forma a permitir uma separação entre os componentes sem que houvesse necessidade de se projetar toda a arquitetura da solução. Utilizou-se o conhecimento dos padrões *Proxy* e *BodyGuard*, combinando-os de forma a resultar num padrão que, embora ainda possa ser aperfeiçoado, já se mostrou bastante adequado no tangente à separação arquitetural das classes. A arquitetura proposta, uma vez transportada para o *middleware*, mostrou que uma melhor divisão ainda pode ser alcançada, pois alguns dos componentes de *software* estão dispersos entre *Proxy* e *BodyGuard*. Por outro lado, este entrelaçamento resultou de simplificações notadas ao longo do desenvolvimento, algumas com vistas puramente à simplicidade, outras com vistas a algum ganho em desempenho (minimizando o número de indireções), o que pode ser aperfeiçoado também no padrão. Trabalhar o requisito segurança na forma de um padrão de projeto concorre não só para a reutilização da solução, mas abre caminho para

modelagem semelhante de outros requisitos, como por exemplo replicação e persistência.

Reflexão foi utilizada para adequar a arquitetura de referência à arquitetura proposta. O *middleware*, para que não se precise alterar a implementação dos clientes e/ou servidores, tem característica reflexiva, interceptando métodos conforme seja adequado.

A plataforma escolhida, Java, oferece um bom suporte à implementação dos conceitos apresentados. A linguagem apresenta as construções e a robustez necessárias, tendo também um bom suporte na JVM. Java teve segurança como um de seus preceitos desde a concepção, o que concorre para desenvolvimento de aplicações seguras. Entretanto, levar a termo uma aplicação segura em Java é uma tarefa complexa, e exige conhecimento de vários aspectos da plataforma, além dos próprios de segurança. Existem cerca de 105 classes (e interfaces) que tratam diretamente de alguma propriedade ou característica de segurança. Além destas, há inumeras que tratam indiretamente (são utilizadas ou são implícitas), e outras que não são contadas como tal, a exemplo de classes internas. Ao longo do trabalho estudou-se várias destas classes, mas um número reduzido delas é utilizado de fato. Isto porque muitas dela tratam de segurança em níveis mais baixos, como manipulação de certificados digitais, por exemplo, extensões de classes criptográficas, substituição de autoridades certificadoras, etc. Enquanto pode ser desejável tratar neste nível, não foi necessário para ilustrar o propósito de compor as tecnologias num *middleware* orientado à segurança.

Com respeito ao protótipo, este ainda não opera como uma entidade coesa e plenamente funcional. Os sub-componentes principais foram implementados, para que se pudesse validar conceitos e aperfeiçoar a solução, faltando uma nível de *software*, ou uma classe agregadora que contemple todas as partes. Em parte, isto se deve à ausência de um sistema de configuração, que desde o início fora previsto ser desenvolvido em trabalhos futuros.

Contudo, as partes testadas demonstram que a direção principal do trabalho, a integração das tecnologias, é promissora. Foi observado que é possível, em casos de aplicações não muito complexas, preservar alto grau de transparência, atingido pela reflexão. Observou-se também, e de certo modo previsto, que conforme cresce o grau de complexidade da aplicação, cresce a integração que a interface *middleware*/aplicação precisa mostrar. Isto vem ao encontro da expectativa de se poder utilizar o *middleware* também como ferramenta nas etapas primárias de desenvolvimento de aplicações seguras.

Com respeito ao desempenho do *middleware* em si, cabe observar que somente uma versão plenamente funcional poderia oferecer substrato de avaliação, inclusive no que diz respeito a quais métricas avaliar. Esta indeterminação não é exclusiva deste trabalho, sendo apontada em trabalhos correlatos de grupos que também iniciaram pesquisas nesta área. Algumas observações, no entanto, podem ser feitas. A primeira diz respeito às classes que implementam autenticação, para as quais a interação com o usuário é praticamente inevitável. Para um sistema que antes não dispunha de tal mecanismo, o passo adicional pode parecer custoso, mas em última análise, não causa impacto no desempenho geral. Por outro lado, cifrar dados antes de transmití-los, como é o caso quando se usa *sockets* customizados, pode causar impacto dependendo do volume de informações. Nos testes realizados, o impacto sobre o desempenho foi ínfimo, imperceptível se tomado em conjunto com atrasos outros, como a própria rede, por exemplo. Entretanto, instrumentar uma aplicação não-interativa de transferência de dados seria um melhor parâmetro, neste caso.

Algumas decisões de desenvolvimento, como por exemplo a dispersão de algumas funcionalidades entre classes, visam minimizar um pouco o impacto causado pelos vários

níveis de indireção nas chamadas de métodos. Isto é feito em algumas das interações do carregador de classes e do gerenciador de segurança, nas verificações por exemplo, e a rigor não invalida o padrão de projeto.

Nem sempre enfocado na literatura, o tratamento dado às exceções é ainda passível de melhoramento. A decisão de se tratar no *middleware* todas as exceções nele geradas, e não propagar nenhuma para a aplicação pode ser objeto de estudo futuro, pois existem aplicações que podem requerer este tipo de informação. A experiência com componentes do *middleware* mostra que reduzir o número de pontos de geração de exceções favorece dois aspectos: desempenho, pois o tratamento de casos extraordinários não passa pela interceptação da máquina virtual, e simplicidade, pois não requer do desenvolvedor um tratamento da pilha de exceções, que pode se tornar bem complexo dependendo do ponto em que ocorre e da precisão que se queira atingir no tratamento.

Como trabalhos futuros, sugere-se uma ferramenta de configuração e o desenvolvimento de uma classe integradora, possivelmente em paralelo. Isto possibilitará que as partes desenvolvidas seja composta de maneira mais flexível em relação à programática, ora utilizada.

REFERÊNCIAS

BUSATO, L. E.; RUBIRA, C. M. F.; LISBôA, M. L. B. A Reflective Object-Oriented Architecture for Developing Fault-Tolerant Software. **Journal of the Brazilian Computer Society**, [S.l.], v.4, n.2, p.39–47, Nov. 1997.

CHARLES, J. Middleware Moves to the Forefront. **IEEE Computer**, [S.l.], v.32, n.05, p.17–19, May 1999.

DEMETER Project. DJ Library. Disponível em: <http://www.ccs.neu.edu/research/demeter/DJ>. Acesso em jun. 2004.

DIAZ, P. A.; CAMPO, M. Analyzing the Role of Aspects in Software Design. **Communications of the ACM**, [S.l.], v.44, n.10, p.66–74, Oct. 2001.

ECKEL, B. **Thinking in Java**. 2.ed. [S.l.]: Prentice-Hall, 2000.

ECLIPSE Project. AspectJ. Disponível em: <http://eclipse.org/aspectj>. Acesso em jun. 2004.

ELRAD, T. et al. Discussing Aspects of AOP. **Communications of the ACM**, [S.l.], v.44, n.10, p.33–38, Oct. 2001.

ELRAD, T.; FILMAN, R. E.; BADER, A. Aspect-Oriented Programming. **Communications of the ACM**, [S.l.], v.44, n.10, p.29–32, Oct. 2001.

EMMERICH, W. **Engineering Distributed Objects**. [S.l.]: John Wiley and Sons, 2000.

EMMERICH, W. Software engineering and middleware: a roadmap. In: THE FUTURE OF SOFTWARE ENGINEERING, 2000. **Proceedings...** New York: ACM Press, 2000. p.117–129.

FERREIRA, L. L.; RUBIRA, C. M. F. Reflective Design Patterns to Implement Fault Tolerance. In: WORKSHOP ON REFLECTIVE PROGRAMMING IN C++ AND JAVA, OOPSLA, 1998, Vancouver, BC, Canadá. **Proceedings...** [S.l.: s.n.], 1998. p.81–85.

GAMMA, E. et al. **Design Patterns**: elements of reusable object-oriented software. [S.l.]: Addison-Wesley, 1994. (Addison-Wesley Professional Computing Series).

GONG, L.; ELLISON, G.; DAGEFORDE, M. **Inside Java 2 Platform Security**: architecture, api design, and implementation. [S.l.]: Addison-Wesley, 1998. (The Java Series).

IBM. **Hyper/J**: multi-dimensional separation of concerns for java. Disponível em: <http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.html>. Acesso em jun. 2004.

KICZALES, G. **The Art of the Metaobject Protocol**. [S.l.]: MIT Press, 1991.

LAI, C. et al. User Authentication and Authorization in the Java Platform. In: ANNUAL COMPUTER SECURITY APPLICATIONS CONFERENCE, 15., 1999, Phoenix, Arizona, USA. **Proceedings...** [S.l.: s.n.], 1999.

LAPRIE, J.-C. Dependability of Computer Systems: from concepts to limits. In: INTERNATIONAL WORKSHOP ON DEPENDABLE COMPUTING AND ITS APPLICATIONS, DCIA, 1998, Johannesburg, South Africa. **Proceedings...** [S.l.: s.n.], 1998.

LISBôA, M. L. B. **Reflexão Computacional no Modelo de Objetos**. 1995. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

LISBôA, M. L. B. A New Trend on the Development of Fault-Tolerant Application: software meta-level architectures. **Journal of the Brazilian Computer Society**, [S.l.], v.4, n.2, p.31–37, Nov. 1997.

MCGRAW, G.; FELTEN, E. W. **Java Security**: hostile applets, holes, and antidotes. [S.l.]: Wiley Computer Publishing, 1996.

MOHAMMAD, M. Aspects + GAMMA = AspectGAMMA: a formal framework for aspect-oriented specification. In: ASPECT-ORIENTED REQUIREMENTS ENGINNERING AND ARCHITECTURE DESING, 2002. **Proceedings...** [S.l.: s.n.], 2002.

NEVES, F. D.; GARRIDO, A. Bodyguard. In: MARTIN, R. C.; RIEHLE, D.; BUSCHMANN, F. (Ed.). **Pattern Languages of Programming Design**. [S.l.]: Addison-Wesley, 1997. v.3, p.231–244.

NICOMETTE, J.-C. et al. Implementing Fault-Tolerant Applications Using Reflective Object-Oriented Programming. In: IEEE INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, FTCS, 25., 1995, Pasadena, CA. **Proceedings...** [S.l.: s.n.], 1995.

OAKS, S. **Java Security**. [S.l.]: O'Reilly & Associates, 1998. (The Java Series).

OKAMURA, H.; ISHIKAWA, Y. Object Location Control Using Meta-level Programming. In: **ECOOP, 1994. Object-Oriented Programming**. Berlim: Springer-Verlag, 1994. p.299–319. (Lecture Notes in Computer Science, v.821).

ROHNERT, H. The Proxy Design Pattern Revisited. In: VLISSIDES, J. M.; COPLIEN, J. O.; KERTH, N. L. (Ed.). **Pattern Languages of Program Design**. [S.l.]: Addison-Wesley, 1995. v.2, p.105–118.

RYUTOV, T.; NEUMAN, C. **Access Control Framework for Distributed Applications**. Internet-draft [S.l.]: IETF, Nov. 200. Disponível em <www.ietf.org/proceedings/01aug/ I-D/draft-ietf-cat-acc-cntrl-frmw-05.txt>. Acesso em: jun. 2004.

STROUD, R. J. Transparency and Reflection in Distributed Systems. In: ACM SIGOPS EUROPEAN WORKSHOP ON DISTRIBUTED SYSTEMS, 5., 1992. **Proceedings...** [S.l.: s.n.], 1992.

SUN. **The Java Tutorial**. Disponível em: <http://java.sun.com/docs/boks/index.html>. Acesso em jun. 2004.

SUN. **Remote Method Invocation (RMI)**. Disponível em: <http://java.sun.com/developer/onlineTraining/rmi/RMI.html>. Acesso em junho de 2004.

VANHAUTE, B.; WIN, B. D.; DECKER, B. D. Building Frameworks in AspectJ. In: WORKSHOP ON ADVANCED SEPARATION OF CONCERNS, ECOOP, 2001. **Proceedings...** [S.l.: s.n.], 2001. p.1–6.

VETTERLING, M.; WIMMEL, G.; WISSPEINTNER, A. Secure systems development based on the common criteria: the palme project. **SIGSOFT Software Engeneering Notes**, New York, v.27, n.6, p.129–138, 2002.

WELCH, I.; STROUD, R. Using Reflection as a Mechanism for Enforcing Security Policies in Mobile Code. In: EUROPEAN SYMPOSIUM ON RESEARCH IN COMPUTER SECURITY, ESORICS, 6., 2000, Toulouse, França. **Proceedings...** Berlin: Springer-Verlag, 2000. (Lecture Notes in Computer Science, v.1895).

WELCH, I.; STROUD, R. J. Supporting real world security models in Java. In: IEEE INTERNATIONAL WORKSHOP ON FUTURE TRENDS OF DISTRIBUTED COMPUTING SYSTEMS, 7., 1999, Cape Town, South Africa. **Proceedings...** [S.l.: s.n.], 1999. p.155–159.

WELCH, I.; STROUD, R. J. Kava - a reflective Java based on bytecode rewriting. In: REFLECTION AND SOFTWARE ENGINEERING, 2000, Heidelberg, Germany. **Proceedings...** Berlin: Springer-Verlag, 2000. p.157–169. (Lecture Notes in Computer Science, v.1826).

WELCH, I.; STROUD, R. J. Kava - using byte-code rewriting to add behavioural reflection to Java. In: USENIX CONFERENCE ON OBJECT-ORIENTED TECHNOLOGIES AND SYSTEMS, 2001, San Antonio, Texas. **Proceedings...** [S.l.: s.n.], 2001. p.119–130.

WELCH, I.; STROUD, R. J. Using reflection as a mechanism for enforcing security policies on compiled code. **Journal of Computer Security**, [S.l.], v.10, n.4, p.399–432, 2002.

WELCH, I.; STROUD, R. J.; ROMANOVSKY, A. Aspects of exceptions at the meta-level. In: METALEVEL ARCHITECTURES AND SEPARATION OF CROSS-CUTTING CONCERNS INTERNATIONAL CONFERENCE, REFLECTION, 3., 2001, Kyoto, Japan. **Proceedings...** Berlin: Springer-Verlag, 2001. p.280–282. (Lecture Notes in Compuwr Science, v.2192).

WIN, B. D. et al. On the importance of the separation-of-concerns principle in secure software engineering. In: WORKSHOP ON THE APPLICATION OF ENGINEERING PRINCIPLES TO SYSTEM SECURITY DESIGN, 2002. **Proceedings...** [S.l.: s.n.], 2002.

WIN, B. D.; VANHAUTE, B.; DECKER, B. D. Security through Aspect-Oriented Programming. In: **Advances in Network and Distributed Systems Security**. [S.l.]: Kluwer Academic Publishers, 2001. p.125–138.

WIN, B. D.; VANHAUTE, B.; DECKER, B. D. How aspect-oriented programming can help to build secure software. **Informatica**, [S.l.], v.26, n.2, p.141–149, 2002.

WIN, B. de; JOOSEN, W.; PIESSENS, F. AOSD & Security: a practical assessment. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, AOSD, 2., 2003. **Proceedings...** [S.l.: s.n.], 2003. p.1–6.

WU, Z.; SCHWIDERSKI, S. **Reflective Java - making Java even more flexible**. United Kingdom: Architecture Projects Management Limited (ANSA), 1997. Technical Report, Disponível em: <http://www.ansa.co.uk>. Acesso em jun. 2004.

XU, J.; RANDELL, B.; RUBIRA, C. M. F. Towards and Object-Oriented Approach to Software Fault-Tolerance. In: FAULT-TOLERANT PARALLEL AND DISTRIBUTED SYSTEMS, 1994, Pasadena, USA. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1994. p.226–233.