CENTRO UNIVERSITÁRIO “EURÍPIDES DE MARÍLIA” - UNIVEM -
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

CARLOS SEIKI FUJII

MODULARIZAÇÃO DE UMA BIBLIOTECA DE CRIPTOGRAFIA NO CONTEXTO DA PROGRAMAÇÃO ORIENTADA A ASPECTOS

MARÍLIA
2007

CARLOS SEIKI FUJII

Modularização de uma Biblioteca de Criptografia no Contexto da Programação Orientada a Aspectos

Monografia apresentada à "Fundação de Ensino Eurípides Soares da Rocha", mantenedora do Centro Universitário "Eurípides de Marília", para obtenção do Título de Bacharel em Ciência da Computação.

Orientador:
Prof. Dr. Valter Vieira de Camargo
Co-Orientador:
Prof. Ms. Rodolfo Barros Chiaramonte

MARÍLIA
2007

FUJII, Carlos Seiki
Modularização de uma Biblioteca de Criptografia no Contexto da Programação Orientada a Aspectos / Carlos Seiki Fujii; orientador: Prof. Dr. Valter Vieira de Carmargo; co-orientador: Prof. Ms. Rodolfo Barros Chiaramonte. Marília, SP: [s.n.], 2007.

65 f.

Monografia (Bacharel em Ciência da Computação) – Centro Universitário Eurípides de Marília – Fundação de Ensino Eurípides Soares da Rocha.

CDD: 005.1

CARLOS SEIKI FUJII

MODULARIZAÇÃO DE UMA BIBLIOTECA DE CRIPTOGRAFIA NO CONTEXTO DA PROGRAMAÇÃO ORIENTADA A ASPECTOS

Banca examinadora da monografia apresentada ao Curso de Bacharel em Ciência da Computação da UNIVEM, /F.E.E.S.R, para obtenção do Título de Bacharel em Ciência da Computação.

Resultado: ____________________________

ORIENTADOR: Prof. Dr. Valter Vieira de Camargo

1º EXAMINADOR:

2º EXAMINADOR:

Marília, 09 de novembro de 2007.

Aos meus pais, Shiromitsu e Silvia.

Ao meu irmão, Celso.

Aos meus amigos.

AGRADECIMENTOS

Agradeço a Deus por me conceder uma vida cheia de saúde, alegrias e amizades.

Agradeço aos meus amados pais, Shiromitsu e Silvia, pelas suas preocupações, conselhos, sacrifícios e principalmente pelo amor e dedicação que sempre tem me dado.

Agradeço ao meu também amado irmão, Celso, que me acompanhou e acompanhará em inúmeras boas lembranças, sempre me incentivando e ajudando a progredir.

Agradeço aos meus amigos, que apesar de não terem participação direta neste trabalho, certamente me ajudaram indiretamente.

Agradeço ao meu orientador Valter, pois sua participação neste trabalho foi essencial aos resultados obtidos. Gostaria de destacar seu empenho e dedicação, que reanimaram minha motivação pelos estudos.

Agradeço ao meu co-orientador Rodolfo, sua biblioteca de criptografia e sua ajuda nas correções dos erros no programa tornaram possível a conclusão deste trabalho.

"O que me preocupa não é o grito dos maus. É o silêncio dos bons."

(Martin Luther King)

FUJII, Carlos Seiki. Modularização de uma Biblioteca de Criptografia no Contexto da Programação Orientada a Aspectos. 2007. 65 f. Monografia (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2007.

RESUMO

A segurança é um interesse primordial para determinadas aplicações computacionais que compartilham informações ou realizam operações confidenciais por meio de uma rede e o uso de criptografia é uma forma de garantir essa segurança. Apesar de existirem inúmeras aplicações com criptografia, o estudo de criptografia com o uso da programação orientada a aspecto ainda é pouco aprofundado. Neste projeto, o objetivo foi estudar a possibilidade de modularização de uma biblioteca de criptografia existente com Programação Orientada a Aspectos. Por intermédio desse estudo foi possível definir uma arquitetura genérica para a implementação de criptografia com aspectos a partir de um padrão proposto em um trabalho existente. Além disso, foi possível formular diretrizes para a construção de uma camada de aspectos genéricos que faz parte da arquitetura proposta. Assim, os resultados principais do estudo são: a apresentação de uma arquitetura genérica e a elaboração de diretrizes para a construção da camada de aspectos genéricos, que pode ser utilizados por desenvolvedores de software para separar o interesse de criptografia dos interesses base do sistema, resultando assim em sistemas mais bem modularizados.

Palavras-chave: Programação orientada a aspecto, POA, AspectJ, criptografia, separação de interesses, modularização.

FUJII, Carlos Seiki. Modularização de uma Biblioteca de Criptografia no Contexto da Programação Orientada a Aspectos. 2007. 65 f. Monografia (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2007.

ABSTRACT

Security is an important concern for some applications which share information or perform confidential operations by means of a network. So, cryptography is an alternative for ensuring this security. Although a lot of research has been conducted on this issue, the study of cryptography with aspect-oriented programming is still a recent area. In this project, the aim was to study the possibility to modularize a cryptography library with aspect-oriented programming. By means of this study, it was possible to define a generic architecture for implementing cryptography with aspects making use of a pattern proposed in a previous work. Besides, it was possible prescribe guidelines for building a layer of generic aspects which make part of the proposed architecture. Thus, the main results of this work are the proposed architecture and guidelines for building a similar architecture which can be used for software developers to separate the cryptography concern from the base concerns of the system, generating well modularized systems.

Keywords: Aspect-oriented programming, AOP, AspectJ, cryptography, separation of concerns, modularization.

LISTA DE ILUSTRAÇÕES

Figura 1 – Exemplo de criptografia simétrica 22
Figura 2 – Exemplo de criptografia assimétrica..24
Figura 3 – Exemplo de sistema híbrido 26
Figura 4 – Diagrama de classe do pacote Cripto de Chiaramonte (2006) 28
Figura 5 – Exemplo 1 de entrelaçamento de código. Composição do sistema com múltiplos interesses transversais (Laddad, 2002)...29
Figura 6 – Exemplo 2 de entrelaçamento de código (Resende e Silva, 2005)...........30
Figura 7 – Código para *Logging* no Servidor Tomcat (Winck e Goetten Junior, 2006) 30
Figura 8 – Diagrama de Classes de um Editor Gráfico (Elrad et al., 2001) 31
Figura 9 – Implementação Orientada a Objetos do Interesse de Atualização do *Display* 32
Figura 10 – Implementação do exemplo da Figura 9 com AspecJ............................34
Figura 11 – Sintaxe de um conjunto de junção (Gradecki e Lesiecki, 2003) 35
Figura 12 – Sintaxe de um aspecto (Gradecki e Lesiecki, 2003) 37
Figura 13 – Exemplo de aplicação Java...38
Figura 14 – Exemplo de aspecto em AspectJ...38
Figura 15 – Resultado da saída da classe `OlaMundo` 38
Figura 16 – Exemplo da Camada de Aspectos Genérica com uso do padrão Capturador de Dados 40
Figura 17 – Exemplo do efeito da criptografia com aspectos (Boström, 2004).........44
Figura 18 – Exemplo de aspecto abstrato para criptografia em DES 46
Figura 19 – Exemplo de aspecto concreto sobre o exemplo da Figura 4 47
Figura 20 – Arquitetura de três camadas para isolamento do interesse de criptografia da aplicação base (Fujii et. al., 2007b)..48
Figura 21 – Exemplo de sistema desenvolvido utilizando a arquitetura proposta 49
Figura 22 – Diagrama de classe do sistema de cadastros 52
Figura 23 – Diagrama de classe da parte de criptografia da camada de aspectos genéricos 53

Figura 24 – Aspecto abstrato: `AbstractEncryption`..54
Figura 25 – Aspecto abstrato: `IntermediaryEncryption`.................................54
Figura 26 – Aspecto Abstrato: `EncryptionReturn`...55
Figura 27 – Aspecto abstrato: `EncryptionThis`..55
Figura 28 – Aspecto abstrato: `EncryptionTarget`..56
Figura 29 – Aspecto abstrato: `EncryptionArgs`..56
Figura 30 – Aspecto concreto: `Encryption`...57
Figura 31 – Diagrama de classe da parte de decriptografia da camada de aspectos genéricos 57
Figura 32 – Aspecto concreto: `Decryption`...58
Figura 33 – Diagrama de classe da aplicação da arquitetura proposta no sistema de estudo de caso...59

LISTA DE TABELAS

Tabela 1 – Designadores de pontos de junção (Gradecki e Lesiecki, 2003)..............36
Tabela 2 – Alternativas de composição do padrão Capturador de Dados (Camargo e Masiero, 2008)..40

LISTA DE QUADROS

Quadro 1 - Exemplo de Cifra de César..18

LISTA DE ABREVIATURAS E SIGLAS

3DES: *Triple* DES

AES: *Advanced Encryption Standard*

DES: *Data Encryption Standard*

IBM: *International Business Machines*

IDEA: *International Data Encryption Algorithm*

JAAS: *Java Authentication and Authorization Service*

JCE: *Java Cryptography Extension*

JDBC: *Java Database Connectivity*

JSAL: *Java Security Aspect Library*

NBS: *National Bureau of Standards*

OO: Orientado a Objeto

POA: Programação Orientada a Aspecto

POO: Programação Orientada a Objeto

RSA: *Rivest, Shamir e Adleman*

UML: *Unified Modeling Language*

SUMÁRIO

1. INTRODUÇÃO 15

1.1. Motivação 15

1.2. Objetivos 16

1.3. Organização do Texto 16

2. CONCEITOS FUNDAMENTAIS 17

2.1. Criptografia 17

2.1.1. Requisitos de segurança 18

2.1.2. Tipos de ataque e segurança de algoritmos de criptografia 19

2.1.3. Criptografia Simétrica 21

2.1.4. Criptografia Assimétrica 23

2.1.5. Sistema Híbrido 25

2.1.6. Pacote Cripto 27

2.2. Programação Orientada a Aspectos (POA) 28

2.2.1. Linguagem AspectJ 33

2.2.2. Ponto de Junção 34

2.2.3. Conjunto de Junção 35

2.2.4. Adendo 36

2.2.5. Aspectos 37

2.2.6. Declaração Inter-tipos 38

2.3. Padrão Capturador de Dados 39

2.4. Considerações Finais 41

3. TRABALHOS RELACIONADOS 42

3.1. Criptografia com aspecto 42

3.2. Considerações Finais 47

4. ARQUITETURA PROPOSTA E ESTUDO DE CASO 48

4.1. Descrição Genérica da Arquitetura 48

4.2. Estudo de Caso..51
4.2.1. Aplicação Base..52
4.2.2. Camada de Aspectos..53
4.3. Aplicação da Arquitetura Proposta 58
4.4. Considerações Finais..61
5. CONCLUSÕES..62
REFERÊNCIAS BIBLIOGRÁFICAS..63

1. INTRODUÇÃO

Atualmente, em conseqüência da necessidade por agilidade, economia e comodidade o compartilhamento de informações e as transações de operações por uma rede vêm crescendo. Infelizmente, quando essas informações trafegam por uma rede podem ser observadas por terceiro, que nem sempre são bem intencionados. Assim surge a necessidade de proteger as informações por meio de algum método de segurança.

A criptografia é utilizada para garantir uma parcela de segurança em aplicações computacionais. Existem várias maneiras de modelar aplicações com criptografia. A mais utilizada é por meio da programação orientada a objetos (POO) que proporciona a decomposição de sistemas complexos em componentes modulares e funcionais (Kiczales, 2001a). No entanto, a POO possui limitações na modularização de interesses transversais, e a criptografia é um exemplo de interesse transversal. Assim, quando POO é usada na implementação de criptografia, o código fonte desse interesse transversal encontra-se espalhado e entrelaçado com o código fonte da aplicação, causando problemas como dificuldade de entendimento, manutenção, reúso e adição de outros interesses (Kiczales, 2001a).

Visando a amenizar problemas como esses, surgiu a Programação Orientada a Aspectos (POA) que oferece ferramentas mais eficientes para a modularização da aplicação, e assim possibilitar o desenvolvimento dos interesses em módulos isolados.

1.1. Motivação

A motivação principal para a condução desse estudo são as limitações existentes quando se utiliza o paradigma orientado a objetos para a implementação de criptografia. Quando isso ocorre, o código do interesse de criptografia fica espalhado e entrelaçado pelos módulos funcionais da aplicação. Isso faz com que o entendimento do sistema e conseqüentemente, seus níveis de manutenção e reúso sejam prejudicados.

Além disso, outras motivações para o estudo aqui descrito são: o pouco aprofundamento encontrado na literatura com relação à aplicação de POA com criptografia

(Boström, 2004; Huang *et al.*, 2001) e lacunas presentes nesses trabalhos relacionados em relação à tentativa de generalizar a parte de aspecto com criptografia.

1.2. Objetivos

O objetivo principal deste trabalho é melhorar a qualidade de uma aplicação em termos de "modularização". Quando uma aplicação é mais bem modularizada, sua manutenibilidade e reusabilbidade também são melhoradas.

Para que o objetivo principal possa ser alcançado, têm-se como objetivos secundários o desenvolvimento de uma arquitetura genérica para a implementação de criptografia com POA. Para que essa arquitetura pudesse ser elaborada, adaptou-se uma biblioteca de criptografia existente (Chiaramonte, 2006) para o contexto da POA utilizando um padrão de projeto proposto em trabalho existente (Camargo e Masiero, 2008). A utilização desse padrão permite que essa arquitetura torne-se genérica para que possa ser reutilizada em outros contextos.

1.3. Organização do Texto

O Capítulo 2 faz uma introdução os conceitos fundamentais para o entendimento desta monografia. Para isso, descreve-se sobre os conceitos de segurança, de criptografia, de POA e de um padrão de projeto existente.

O Capítulo 3 apresenta trabalhos de segurança com ênfase em criptografia com o uso de POA que serviram de base para o este trabalho.

O Capítulo 4 apresenta a arquitetura proposta neste trabalho e o estudo de caso que foi realizado para a obtenção dessa arquitetura.

As conclusões obtidas por meio do desenvolvimento deste trabalho são apresentadas no Capítulo 5.

2. CONCEITOS FUNDAMENTAIS

Nesta seção são descritos os conceitos fundamentais para o entendimento desta monografia. Na Seção 2.1 descreve-se sobre os conceitos básicos de segurança, de criptografia e sobre o pacote Cripto, que foi desenvolvido por Chiaramonte (2006) e é responsável pela realização de criptografia deste trabalho. Na Seção 2.2 são descritos princípios e técnicas da programação orientada a aspectos. Na Seção 2.3 é descrito o padrão Capturador de Dados que é utilizado na arquitetura proposta neste trabalho.

2.1. Criptografia

Criptografia é uma extensão da criptologia (estudo sobre segurança em comunicações) que projeta algoritmos para criptografar e decriptar informações com o intuito de assegurar o segredo e/ou autenticidade dessas informações (Stallings, 2005).

Criptografar é a descrição dada ao processo de converter uma mensagem original em uma mensagem cifrada de forma que a mensagem cifrada possa posteriormente ser revertida na mensagem original e decriptar é o termo usado para descrever o processo oposto a criptografar, ou seja, restaurar a mensagem original por meio da tradução da mensagem cifrada (Stallings, 2005).

A preocupação do Homem com a segurança de uma informação originou a criptografia, apesar de não ser possível datar a primeira forma de criptografia, que foi conhecida que por volta de 1.900 a.C. em uma vila egípcia em que aconteceu um dos primeiros exemplos documentados de escrita cifrada (Tkotz, 2007).

Na Roma Antiga originou-se um algoritmo muito famoso, a Cifra de César, que consiste em substituir uma letra do alfabeto por outra de acordo com um deslocamento fixo e circular, assim a chave determina quanto a letra será deslocada. Dessa forma, como o alfabeto possui 26 letras, podem ser formadas 25 chaves no total, o que torna a Cifra de César um algoritmo de fraca confiabilidade.

Na Quadro 1 é mostrado um exemplo da Cifra de César. Neste exemplo a primeira linha da tabela representa o alfabeto legível e a segunda linha da tabela representa o alfabeto ilegível com a chave correspondendo ao número 3, ou seja, é preciso avançar

três letras no alfabeto legível para obter o texto ilegível pela Cifra de César, assim a letra B do texto legível torna-se a letra E no texto ilegível. Exemplo: a palavra “criptografia” no texto legível torna-se a nova palavra “fulswrjudild” no texto ilegível.

legível →	A	B	C	D	E	F	G	H	I	J	K	L	M
ilegível →	D	E	F	G	H	I	J	K	L	M	N	O	P

legível →	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
ilegível →	Q	R	S	T	U	V	W	X	Y	Z	A	B	C

Quadro 1 - Exemplo de Cifra de César

Por volta de 1920 surgiu um aparelho, denominado Enigma, muito utilizado pelos alemães até o final da Segunda Guerra Mundial. Desde essa época até o final dos anos 70, os principais algoritmos de criptografia eram secretos, principalmente aqueles utilizados pela diplomacia e forças armadas de cada país, chegando ao final desse período a constatação que os sistemas criptográficos deveriam ser baseados no segredo da chave ao invés dos algoritmos.

A nova idéia deu origem ao DES (*Data Encryption Standard*), que foi o primeiro algoritmo simétrico de domínio público. Criado pela IBM (*International Business Machines*), o DES foi publicado no NBS (*National Bureau of Standards*) em 1977 e a partir dessa data foi adotado como padrão nos EUA.

2.1.1. Requisitos de segurança

Segundo Chiaramonte (Stalling, 1998 apud (Chiaramonte, 2006)[1], Menezes, e*t al*,1996), com a ajuda da criptografia, é possível atingir os seguintes requisitos necessário a fim de estabelecer uma comunicação de forma segura:

[1] Chiaramonte, R. Sico: Um Sistema Inteligente de Comunicação de Dados com Suporte Dinâmico a Segurança. 2006. Dissertação (Mestrado em Ciência da Computação) - Centro Universitário Eurípides de Marília, Marília, 2006.

- Confidencialidade: garantia de que as informações armazenadas ou transmitidas por meio de computadores sejam acessadas ou manipuladas somente pelos usuários devidamente autorizados;
- Autenticidade: garantia de que os participantes de uma comunicação sejam devidamente identificados, tendo assim certeza absoluta das identidades dos mesmos;
- Integridade: garantia de que a informação processada ou transmitida chegue ao seu destino exatamente da mesma forma em que partiu da origem;
- Não Repudio (Autoria Garantida): garantia de que nem a origem nem o destino das informações possam negar, posteriormente, sua transmissão, recepção ou posse.
- Controle de Acesso: garantia de que informações armazenadas ou transmitidas por computadores somente sejam manipuladas por usuários devidamente autorizados;
- Disponibilidade: garantia de que recursos necessários aos usuários estejam sempre disponíveis.

2.1.2. Tipos de ataque e segurança de algoritmos de criptografia

Ataque é todo tipo de tentativa deliberada a fim de violar as regras de segurança de um sistema por meio da exploração de alguma vulnerabilidade (Stalling, 2005).

Os ataques violam os requisitos de segurança e podem ser classificados em ataques passivos ou ativos.

Os ataques passivos provem de monitoramento ou espionagem de transmissões. Nos ataques passivos o intruso procura somente ter acesso às informações que estão sendo transmitidas, sem alterá-las. Dessa forma nem o remetente, nem o receptor tem ciência que um terceiro leu as informações. Em conseqüência dessa natureza, é muito difícil detectar os ataques passivos, portanto é preferível prevenir que o intruso leia as informações, com a utilização da criptografia, ao invés de detectar a invasão (Stalling, 2005).

Os ataques ativos envolvem modificação de informações ou criação de informações falsas e podem ser subdivididas em quatro tipos: mascaração, reprodução, modificação e negação de serviço.

O ataque mascarado ocorre quando uma entidade finge ser uma entidade diferente. Exemplo: um terceiro (intruso) envia uma mensagem a alguém (receptor) fingindo ser outra pessoa (remetente). A reprodução ocorre quando um serviço já autorizado e concluído é capturado e forjado por outro pedido na tentativa de retransmitir comandos autorizados. Na modificação alguma parcela de uma informação legítima é alterada. A negação de serviço impede ou inibe o uso ou a gerência normal de uma comunicação, podendo romper uma rede inteira ou degradar o seu desempenho.

A criptografia possui características capazes de bloquear determinados ataques. Exemplo disso é o uso da assinatura digital que pode desqualificar os ataques do tipo mascarado, por reprodução e por modificação. Assinatura digital é um mecanismo presente na criptografia assimétrica[2] e que garante a autenticidade das informações.

Assim como a criptografia possui características especificas para bloquear determinados ataques, existem também ataques específicos para tentar quebrar a criptografia. A segurança de um algoritmo de criptografia é proporcional à dificuldade em obter o texto original a partir de seu texto criptografado ou em descobrir a sua chave.

Segundo Benits (Terada, 2003 apud (Benits, 2003))[3], existem vários tipos de ataque a fim de tentar quebrar um algoritmo de criptografia :

- Ataque só por texto ilegível: nesse ataque o criptanalista tenta adquirir o texto legível analisando apenas o texto ilegível. Caso esse tipo de ataque tenha sucesso, o algoritmo em questão é considerado totalmente inseguro.
- Ataque por texto legível conhecido: nesse ataque o criptanalista tenta conseguir a chave analisando partes correspondentes do texto legível e ilegível.
- Ataque por texto legível escolhido: nesse ataque, além do conhecido no ataque por texto legível conhecido, o criptanalista tem a possibilidade de escolher trechos pares legíveis e ilegíveis, os quais apresentem características estruturais que lhe permita melhor análise e, posteriormente, maior conhecimento do algoritmo e chave empregados e assim poder deduzir o texto legível a partir do ilegível.

---

[2] Criptografia assimétrica é uma forma de criptografia que utiliza dois tipos de chaves diferentes. Esse assunte é abordado na Seção 2.1.4.

[3] Benits Júnior, W. Sistemas criptográficos baseados em identidades pessoais. 2003. Dissertação (Mestrado em Ciência da Computação) – Instituto de Matemática e Estatística da Universidade de São Paulo, São Paulo, 2006.

- Ataque adaptativo por texto escolhido: nesse ataque, além do conhecido no ataque por texto legível escolhido, o criptanalista escolhe os próximos trechos de texto a serem analisados baseando-se em conhecimento adquirido pela análise de trechos já analisados anteriormente.
- Ataque por texto ilegível escolhido: nesse ataque o criptanalista escolhe inicialmente o texto ilegível e então obtém o texto legível correspondente. Supõe-se que o criptanalista tenha acesso apenas ao algoritmo de decriptografia (sem ter acesso à chave) e o seu objetivo é, mais tarde, sem ter mais acesso à decriptografia, ser capaz de deduzir o legível correspondente a um ilegível novo.
- Ataque adaptativo por texto ilegível escolhido: nesse ataque, além do conhecido no ataque por texto ilegível escolhido, a escolha de um novo texto ilegível pelo criptanalista pode depender do conhecimento adquirido por meio dos textos legíveis analisados.
- Ataque por chave conhecida: nesse ataque o criptanalista deduz novas chaves por meio do conhecimento de chaves já usadas.
- Ataque por dicionário: nesse ataque o criptanalista compara senhas mais comuns (nome de pessoas, data especiais, números de documentos ou telefones) criptografadas com senhas ilegíveis, dessa forma, quando uma senha criptografadas suposta coincidir com uma senha criptografada do usuário, terá encontrado a senha válida.

2.1.3. Criptografia Simétrica

Esse tipo de criptografia de chave simétrica, também é conhecido como criptografia de chave privada e possui uma única chave utilizada nas duas extremidades da comunicação, tanto pra quem irá criptografar quanto para quem irá decriptar a mensagem (emissor e receptor).

Nesse sistema, um usuário cria a chave e a utiliza para criptografar alguma mensagem. A mensagem é enviada normalmente por um meio de comunicação, contudo a chave precisa ser enviada em um canal seguro, ou ser previamente combinada entre os usuários da comunicação.

Na Figura 1 é apresentado um exemplo de um sistema que utiliza a criptografia simétrica.

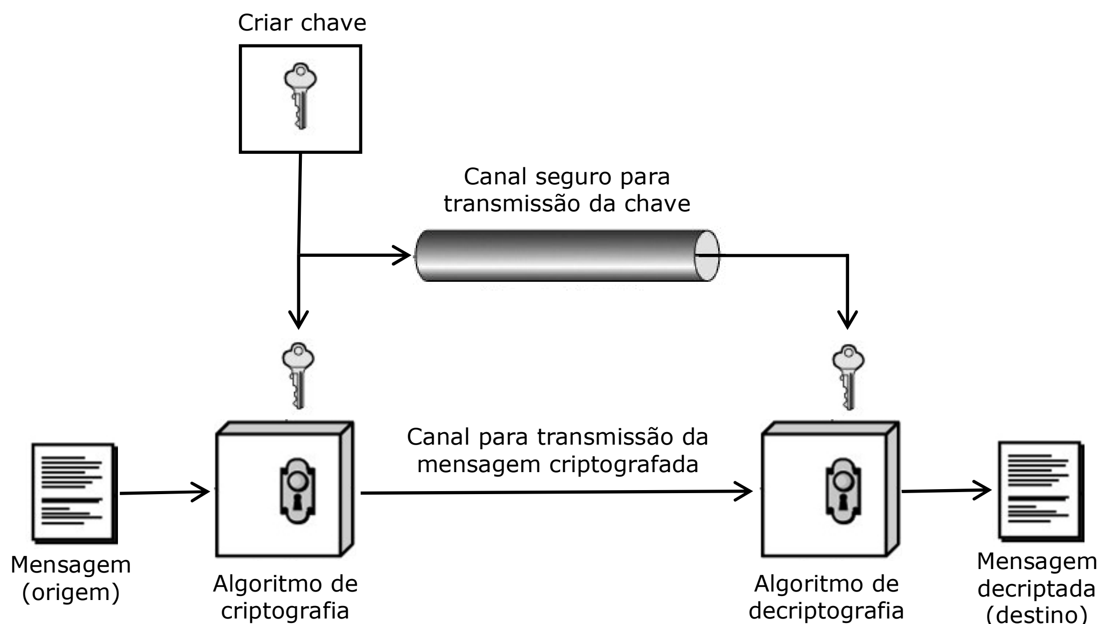

Figura 1 – Exemplo de criptografia simétrica[4]

A criptografia simétrica possui vários algoritmos. A seguir será feita uma breve descrição de alguns.

O DES é o exemplo mais conhecido de um algoritmo de chave simétrica. O algoritmo DES realiza a criptografia por meio de uma transposição inicial sobre blocos com tamanho de 64 bits, seguidos de 16 iterações e concluída com a inversa da transposição inicial. Nas 16 iterações utilizam-se 16 sub-chaves, todas derivadas da chave original de 56 bits por meio de deslocamentos e transposições. As operações utilizadas nas iterações são permutações, transposições, substituições, expansões, operações lógicas do tipo ou-exclusivo e deslocamentos (Stalling, 2005).

A partir do algoritmo DES surgiu o Triple DES, ou 3DES, que é basicamente o DES aplicado três vezes, podendo utilizar sempre a mesma chave ou senão duas ou três chaves diferentes.

O *International Data Encryption* Algorithm, ou IDEA como é mais conhecido, foi desenvolvido por X. Lai e J. Massey em 1991 e utiliza o mesmo algoritmo tanto pra a criptografar quanto para a decriptar . Para isso, o algoritmo utiliza uma chave de 128 bits e

[4] Exemplo baseado no modelo de convencional de sistemas criptográficos de Stalling (2005).

opera em oito iterações sobre blocos com tamanho de 64 bits seguidos de uma transformação final. Cada iteração utiliza uma sub-chave distinta (Menezes, 1996).

Tendo em vista que o mesmo algoritmo é utilizado tanto para criptografar quanto para decriptar, a definição de qual o tipo de operação a ser executada é feita na geração das sub-chaves.

Outro algoritmo bastante conhecido é o AES (*Advanced Encryption Standard*). O AES utiliza o algoritmo denominado Rijndael, que foi desenvolvido por Joan Daemen e Vincente Rijmen. Esse algoritmo utiliza tanto blocos quanto chaves variáveis com tamanhos entre 128, 192 e 256 bits. Isto significa que se pode ter tamanho de blocos com tamanhos de chaves diferentes. Em função do tamanho de bloco e chaves, é determinada a quantidade de rodadas necessárias para criptografar ou decriptar (Stalling, 2005).

O AES, diferentemente do IDEA, trabalha em ciclos completos e é composto de quatro funções distintas: substituição de bytes, permutações de operações aritméticas sobre um campo finito e operações XOR com uma chave (Stalling, 2005).

2.1.4. Criptografia Assimétrica

A criptografia de chave assimétrica, também conhecida como criptografia de chave pública, em que, diferentemente da criptografia simétrica, possui duas chaves utilizadas individualmente em cada extremidade da comunicação, uma utilizada para criptografar (chave pública) e outra utilizada para decriptar (chave privada). Uma mensagem criptografada com uma chave pública só pode ser decriptada com o uso da chave privada correspondente ao seu par de chaves.

Nesse sistema, um usuário X cria um par de chaves assimétricas, chave pública e chave privada, e repassa somente a chave pública a um usuário Y. O usuário Y utiliza a chave pública para criptografar alguma mensagem e então envia esta mensagem criptografada ao usuário X. O usuário X de posse de sua chave privada correspondente decripta a mensagem criptografada e assim obtém a mensagem legível.

Na Figura 2 é apresentado um exemplo da criptografia assimétrica.

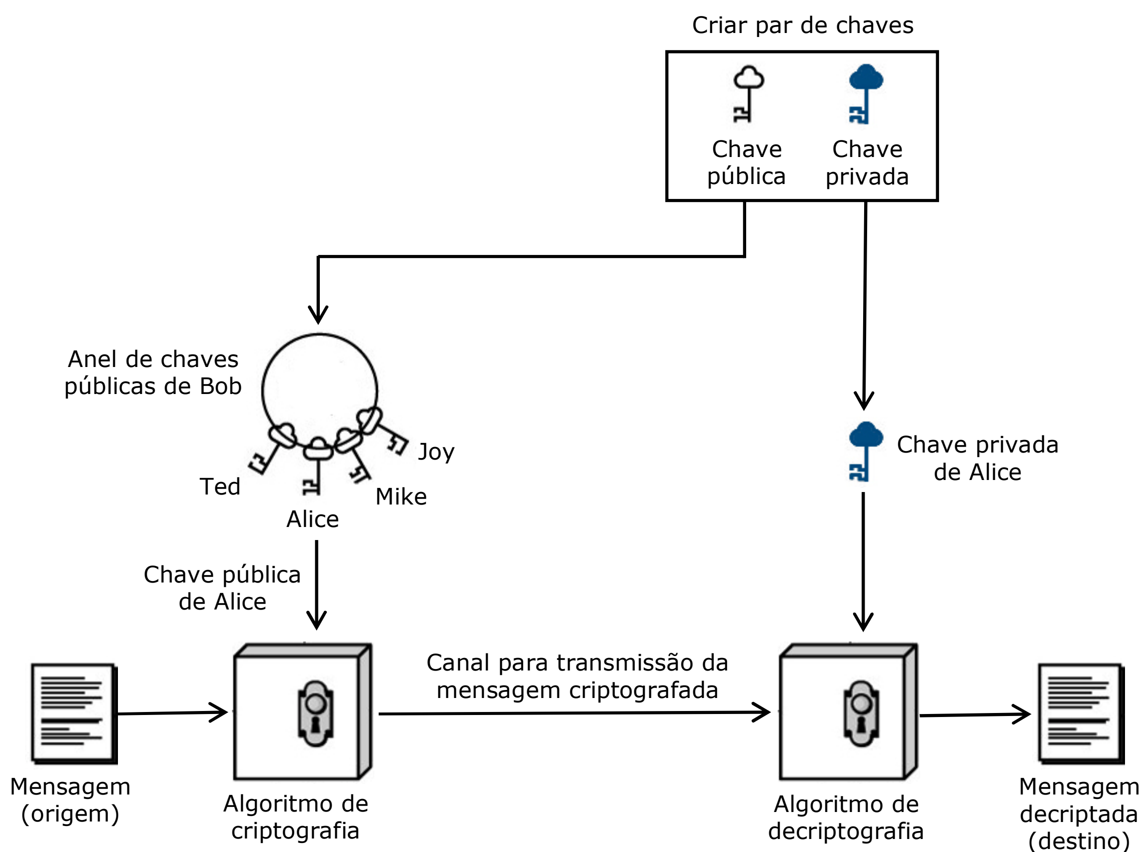

Figura 2 – Exemplo de criptografia assimétrica[5]

Por meio do uso da criptografia assimétrica pode-se gerar a assinatura digital, essa que é utilizada para garantir a autenticidade de uma mensagem. Resumidamente, a assinatura digital é um anexo da mensagem gerada criptografando-se, com a chave privada, um valor Y gerado por uma função cujo propósito é criar um valor Y diferente para cada mensagem (função *Hash*), com isso cada valor Y fica associado univocamente a uma mensagem. Dessa forma quando o receptor receber a mensagem ele decripta a assinatura com a chave pública correspondente e confirma o valor Y.

Um algoritmo famoso de criptografia assimétrica é o RSA (*Rivest, Shamir e Adleman*) e tem esse nome em conseqüência dos nomes de seus autores Ronald Rivest, Adi Shamir e Leonard Adleman. O RSA é baseado no algoritmo de exponenciação modular e é um algoritmo geralmente aceito como prático e seguro para a criptografia assimétrica (Stalling, 2005). A segurança do RSA é baseada na dificuldade de fatorar grandes números

[5] Exemplo baseado no modelo de criptografia assimétrica de Stalling (2005).

e suas chaves são geradas a partir de grandes números primos. Quanto maior forem os números primos utilizados para a criação das chaves, maior é a segurança proporcionada por esse algoritmo. Hoje em dia os números primos que são utilizados têm geralmente 512 bits de comprimento e combinados formam chaves de 1024 bits.

2.1.5. Sistema Híbrido

A criptografia simétrica possui o benefício de ser muito mais rápida quando comparada com a criptografia assimétrica, contudo possui um ponto negativo, pois a chave utilizada para criptografar a mensagem precisará ser previamente combinada entre o emissor e o receptor ou então essa chave precisará ser transmitida por um canal seguro. Nos dois casos um intruso pode interceptar a conversa sobre a combinação da chave ou interceptar a própria chave na sua transmissão. Além desses comparativos, os algoritmos assimétricos permitem a autenticação via assinatura digital, enquanto os algoritmos simétricos não oferecem esse benefício (Menezes, 1996).

A fim de melhorar a criptografia, foi criado o sistema híbrido, que consiste na união do sistema simétrico e assimétrico. Nesse tipo de sistema, o algoritmo simétrico, que é o mais veloz, é utilizado para criptografar a mensagem a ser transmitida (serviço de maior proporção) e o algoritmo assimétrico, que é mais lento, mas também mais seguro, é utilizado para criptografar a chave simétrica utilizada na criptografia da mensagem (serviço de menor proporção).

Na Figura 3 é apresentado um exemplo de como é composto o sistema híbrido.

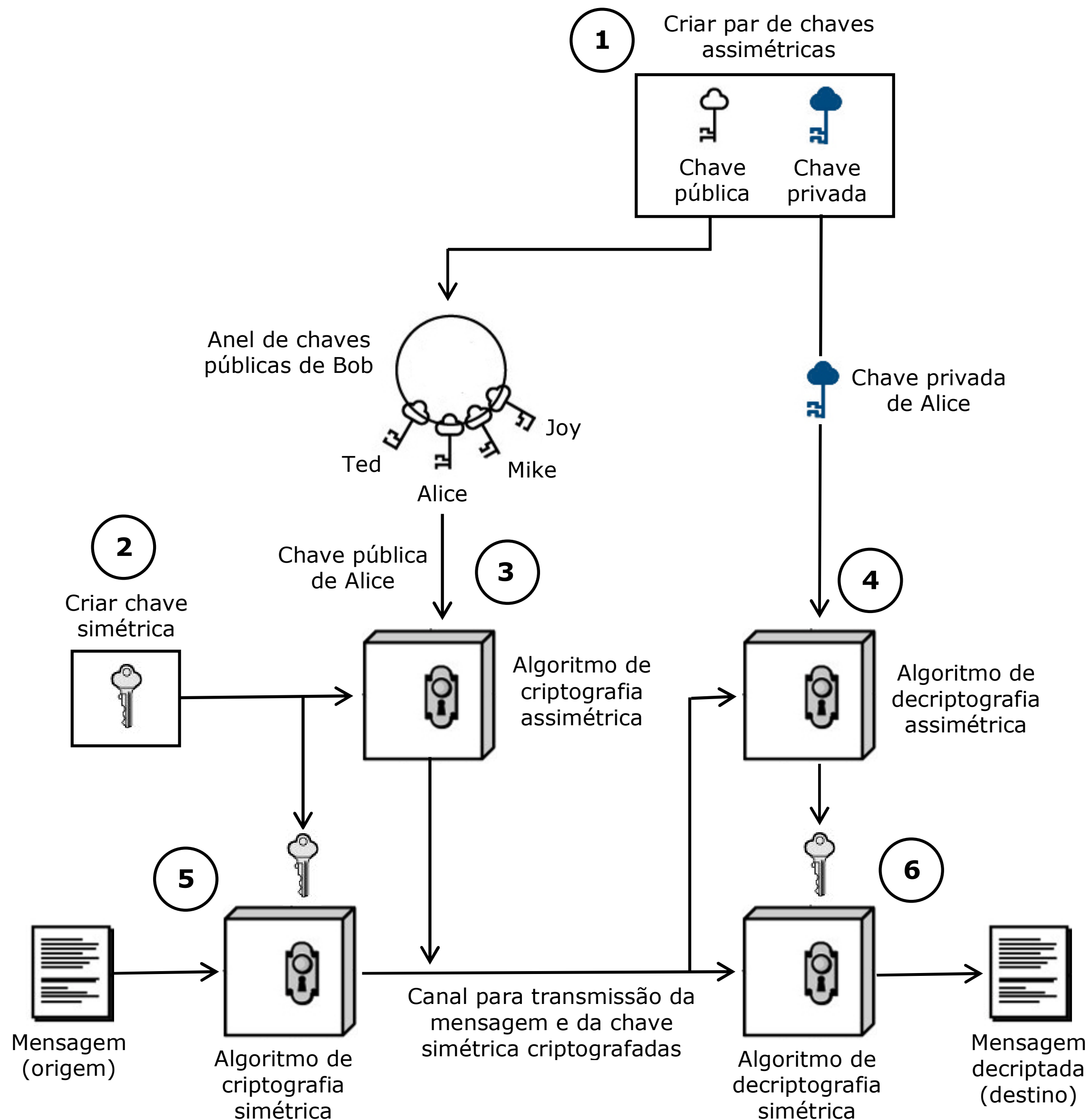

Figura 3 – Exemplo de sistema híbrido[6]

Passos para realizar a criptografia com o uso de um sistema híbrido:

1. Alice cria o par de chaves assimétricas e enviar a chave pública para Bob;
2. Bob cria uma chave simétrica;
3. Bob criptografa a chave simétrica utilizando a chave pública de Alice e envia a chave simétrica criptografada através do canal de transmissão;

[6] Modelo baseado no *software* Pretty Good Privacy (PGP) de Zimmermann (2007)

4. Utilizando a sua chave privada correspondente, Alice decripta a chave simétrica de Bob, essa que foi criptografada com a chave pública de Alice;
5. Bob criptografa a mensagem com a chave simétrica e envia a mensagem criptografada através do canal de transmissão;
6. Alice decripta a mensagem critptografada simetricamente utilizando a chave simétrica de Bob e assim obtém a mensagem.

2.1.6. Pacote Cripto

O pacote Cripto, escrito em Linguagem Java, pertence ao sistema SISCO, ambos desenvolvidos por Chiaramonte (2006). Esse pacote possui três *interfaces*[7] básicas (Chiaramonte, 2006):

- A *Interface* "Algoritmo" possui os métodos mais genéricos que são comuns entre os algoritmos criptográficos utilizados, sendo eles: `doFinal()` responsável por finalizar uma operação envolvendo criptografia; `update`() responsável por realizar uma operação parcial de criptografia; e `getOutputSize`() que retorna qual o tamanho necessário para armazenar o resultado após uma chamada ao método `doFinal()`.
- A *Interface* "Simétrico" foi criada para agrupar os algoritmos simétricos de criptografia (Exemplo: AES, DES, IDEA). Ela é composta dos métodos: `defineChave()` que é responsável por definir a chave que será utilizada na operação; `criaChave()` responsável por criar uma chave aleatória; `obtemChave()` que retorna qual a chave atualmente utilizada; `defineModo()` para definir se será realizada a criptografia ou decriptografia; `limpaChave()` responsável por apagar a chave atualmente em uso; e `temChave()` para verificar se existe alguma chave definida para algoritmo.
- A *Interface* "Assimétrico", criada para os algoritmos assimétricos é semelhante à utilizada para os algoritmos simétricos, no entanto, existe uma grande

[7] Segundo Sun (2007b) uma *Interface* é um conjunto de definições de métodos e valores constantes que devem ser implementados por classes que definem esta *Interface*. Ela pode mais tarde ser implementadas pelas classes que definem esta interface com o uso da palavra reservada "*implements*".

diferença no método de definição de chaves uma vez que devem existir métodos específicos para manipulação da chave.

As *Interfaces* definem os principais métodos referentes a cada tipo de algoritmo. Assim, cada algoritmo implementado definir qual *Interface* ele implementa, de acordo com seu tipo. Isto separa todos os algoritmos do mesmo tipo aos mesmos métodos de acesso.

Na Figura 4 é mostrado o diagrama de classe do pacote Cripto.

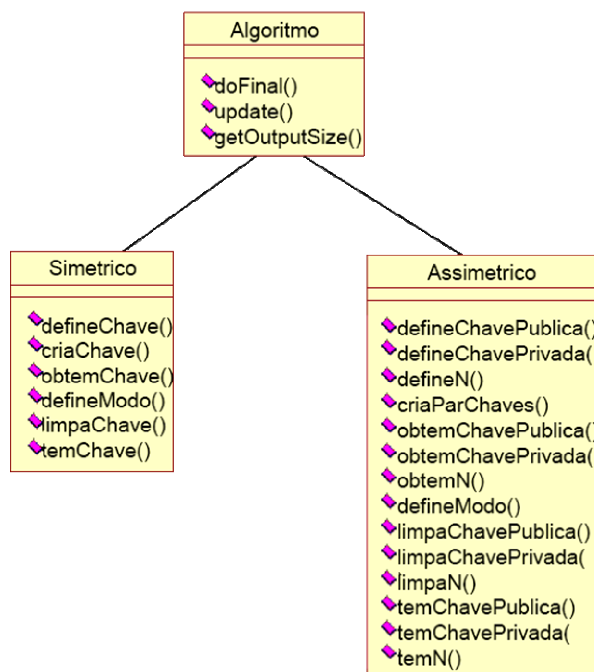

Figura 4 – Diagrama de classe do pacote Cripto de Chiaramonte (2006)

A criptografia utilizada por meio do pacote Cripto faz parte do sistema desenvolvido neste trabalho. Outra parte importante nesse sistema é a Programação Orientada a Aspectos, que é descrita na seção seguinte.

2.2. Programação Orientada a Aspectos (POA)

As linguagens de programação, sejam elas do paradigma estruturado ou orientado a objeto, realizam por meio de funções, procedimentos e classes a decomposição de suas funcionalidades com o objetivo de separar seus interesses. Entretanto, existem interesses transversais (*crosscuting concern*) que não são adequadamente modularizados com as

construções fornecidas pelos paradigmas convencionais (Kiczales, 1997). Exemplos desses interesses transversais são os não-funcionais ou sistêmicos: como gerenciamento de transação, tratamento de exceção, distribuição, sincronização, controle de concorrência, persistência e segurança. Geralmente, a implementação desses interesses transversais fica espalhada de maneira intrusiva dentro dos interesses-base, esse que corresponde à parte funcional da aplicação.

A Programação Orientada a Objetos (POO) contribuiu bastante para o desenvolvimento de programas computacionais, porém possui limitações. Uma das limitações é a dificuldade em modularizar determinados interesses transversais. Quando existe a necessidade de implementarem os interesses transversais, devem-se incluir na parte base da aplicação, chamadas aos métodos que realizam o interesse transversal desejado, gerando entrelaçamento (*Tangled Code*) e espalhamento (*Spread Code*) entre os códigos dos interesses distintos (Gradecki e Lesiecki, 2003).

Na Figura 5 ilustra-se um sistema com um conjunto de preocupações implementadas por diversos módulos. Nesse sistema os interesses transversais (persistência, segurança e *log*) estão entrelaçados dentro dos três módulos que o compõem. Cada interesse é representado por uma tonalidade diferente.

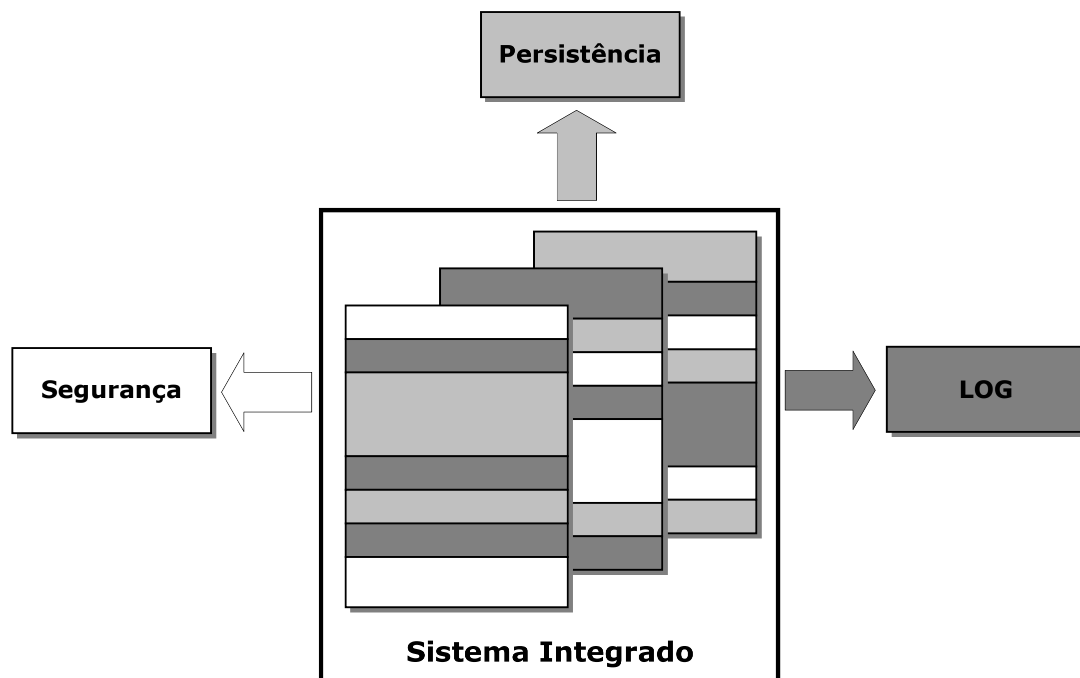

Figura 5 – Exemplo 1 de entrelaçamento de código. Composição do sistema com múltiplos interesses transversais (Laddad, 2002)

Na Figura 6 é mostrado um exemplo com duas classes, denominadas de Ponto e Reta. O entrelaçamento ocorre na chamada feita pelo construtor da classe Reta ao método da classe Ponto. As linhas na classe Reta, sublinhadas, que chamam métodos de Ponto, são chamadas de códigos intrusivos.

```
class Ponto {
      int x;
      int y;
      public Ponto(int x, int y) {
             this.x = x;
             this.y = y;
      }
      public int getX() { return this.x; }
      public int gety() { return this.y; }
}
class Reta {
      int x1;
      int y1;
      public Reta(Ponto x, Ponto y) {
             this.x1 = x.getS();
             this.y1 = y.getY();
      }
}
```

Figura 6 – Exemplo 2 de entrelaçamento de código (Resende e Silva, 2005)

Em relação a espalhamento de código, um caso típico é o uso de auditoria (*log*), que por meio da Figura 7, pode-se notar a falta de modularização através do espalhamento do código por diversas classes em um sistema. As colunas representam as 42 classes desse sistema e as barras horizontais presentes dentro das classes representam à ocorrência do código de *logging*.

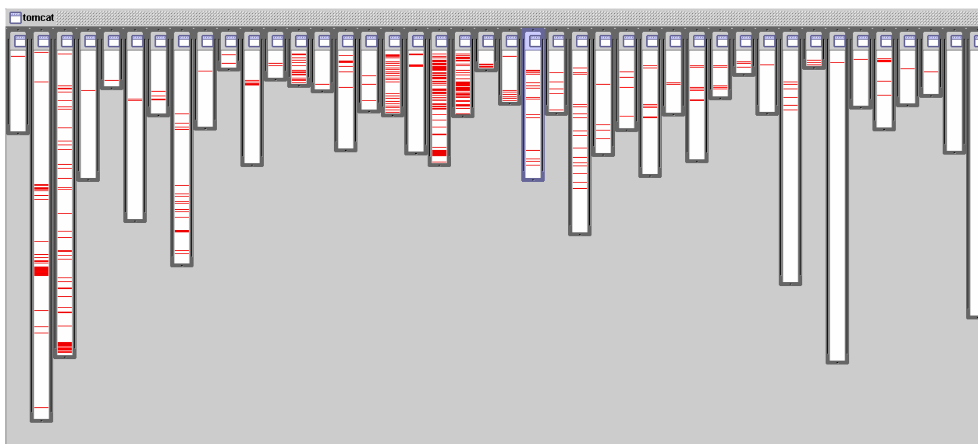

Figura 7 – Código para Logging no Servidor Tomcat (Winck e Goetten Junior, 2006)

Segundo Kiczales *et al.* (2001a) e Laddad (2002), a programação orientada a aspectos (POA) é uma tecnologia que tem como principal objetivo melhorar a separação de interesses de um sistema, em unidades modulares. Essas unidades modulares são posteriormente recompostas (*weaving*) num sistema completo, gerando um código mais simples, fácil de manter e alterar e, assim possibilitando um aumento da reusabilidade do código.

O desenvolvimento de sistemas utilizando a POA abrange três etapas (Laddad, 2002):

- Decomposição dos aspectos: Nessa etapa, são identificados e separados os interesses transversais dos interesses base;
- Implementação dos interesses: Nessa etapa, cada interesse tranversal é implementado separadamente;
- Recomposição dos aspectos: Nessa etapa, os interesses transversais implementados são combinados em um processo denominado *weaving* para a geração do sistema.

Elrad *et al.* (2001b) ilustram por meio de um modelo simplificado de UML (*Unified Modeling Language*), mostrado na Figura 8, uma boa forma de compreender um interesse transversal. O modelo deve incorporar um novo interesse, que consiste em notificar o dispositivo de exibição (classe `Display`) toda vez que um elemento do tipo `FigureElement` for movimentado. Isso requer que cada método que movimente `FigureElement` faça a notificação.

O retângulo pontilhado presente na Figura 8, engloba os métodos que devem implementar o interesse transversal da atualização do `Display`.

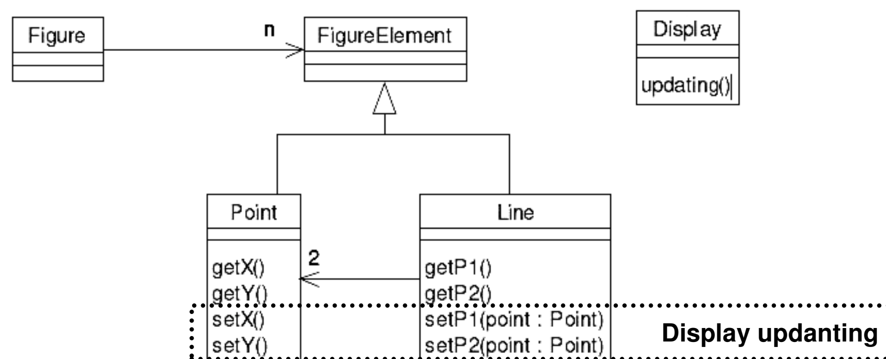

Figura 8 – Diagrama de Classes de um Editor Gráfico (Elrad *et al.*, 2001)

Executar esse interesse transversal utilizando exclusivamente a POO tende a espalhar o método `updating()` em todos os métodos que atualizam o *display* (métodos `set*()` das classes `Point` e `Line`). Para representar essa situação, é apresentada a Figura 9. Em destaque, o método `updating()` que está entrelaçado e espalhado a outros métodos do sistema.

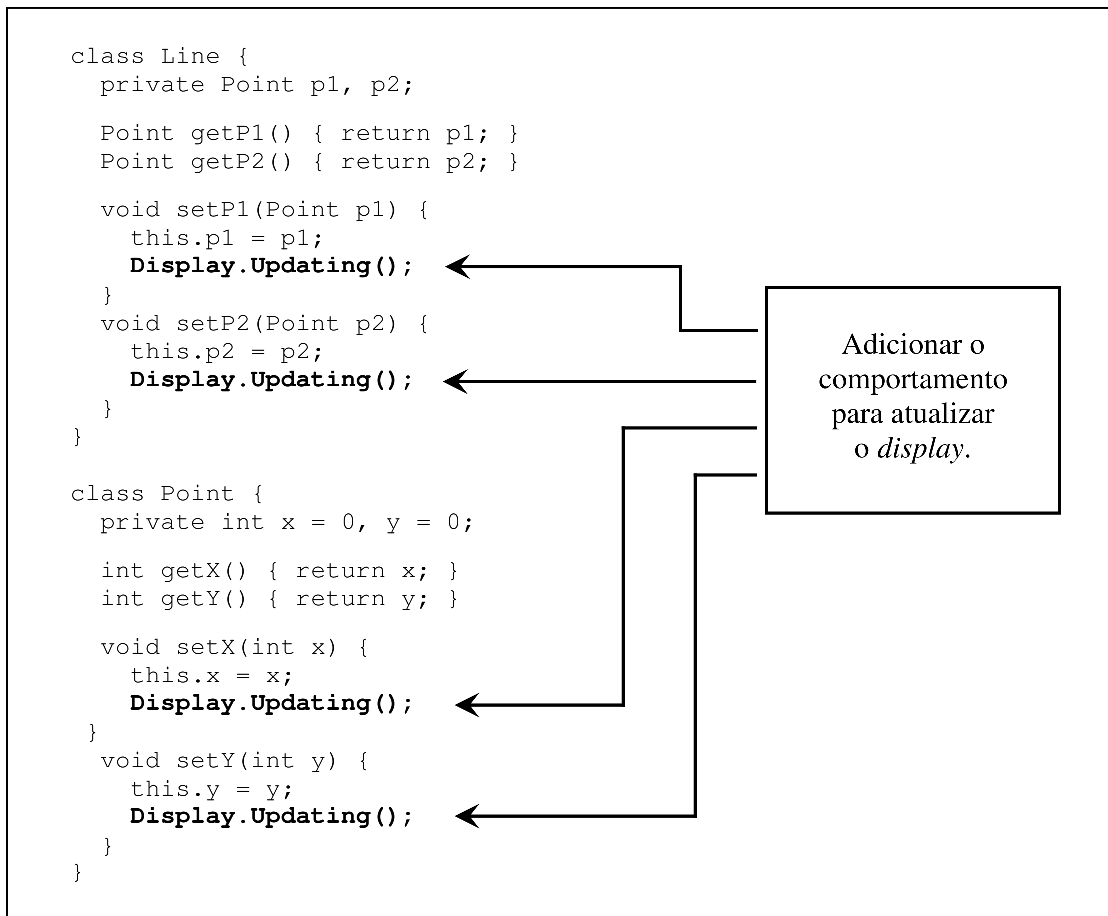

Figura 9 – Implementação Orientada a Objetos do Interesse de Atualização do *Display*

A POA introduz novas abstrações de modularização e mecanismos de composição para melhorar a separação de interesses transversais. As novas abstrações propostas pela POA são: aspectos, pontos de junção (*join points*), conjuntos de junção (*pointcuts*), adendos (*advices*) e declarações intertipos (*inter-type declarations*), as quais são explicadas, neste trabalho, utilizando exemplos na Linguagem AspectJ.

2.2.1. Linguagem AspectJ

AspectJ é uma extensão da Linguagem Java (Kiczales *et al.*, 2001ª; Gradecki e Lesiecki, 2003). Nela, a separação de interesse é realizada por meio da inserção de uma nova abstração denominada "aspecto" à programação orientada a objetos. A composição é realizada através de um combinador denominado *weaver*, que é responsável por pré-processar o código orientado a objetos, inserindo ou modificando os objetos com o comportamento dos aspectos.

Além dos métodos e atributos, um aspecto é composto de conjuntos de junção, adendos, e declarações inter-tipos. Eles podem alterar a estrutura estática ou dinâmica de um programa. A estrutura estática é alterada adicionando, por meio das declarações inter-tipos, membros (atributos, métodos ou construtores) a uma classe, modificando, assim, a hierarquia do sistema. A alteração na estrutura dinâmica de um programa ocorre por meio dos pontos de junção, que são selecionados por conjuntos de junção, e pela adição de comportamentos (adendos) antes ou depois dos pontos de junção.

Na Figura 10 é mostrado um exemplo de programa com o uso do AspectJ. Esse exemplo é a implementação do exemplo da Figura 9 com o auxílio do POA por meio do AspectJ. Note a melhor modularização em comparação ao exemplo da Figura 9.

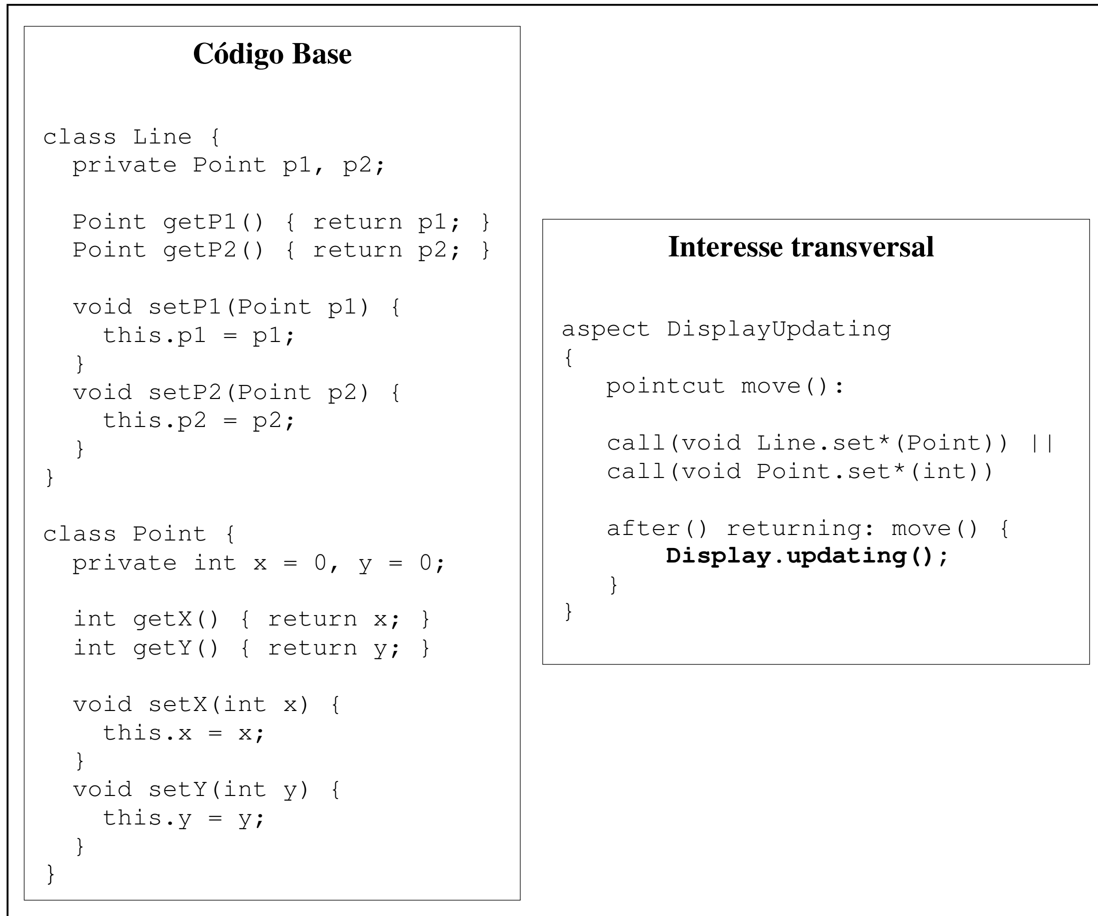

Figura 10 – Implementação do exemplo da Figura 9 com AspecJ

2.2.2. Ponto de Junção

Um ponto de junção é um ponto bem definido no fluxo de execução de um programa (Elrad *et al.*, 2001b; Kiczales *et al.*, 2001ª; Kiczales *et al.*, 2001b). Os pontos de junção são os locais onde ocorrem as combinações dos interesses transversais, ou seja, são os pontos na execução do sistema onde ocorre o tratamento dos interesses transversais.

A AspectJ pode detectar e operar sobre os seguintes tipos de pontos de junção (Kiczales *et al.*, 2001a):

- Chamada ou execução de métodos;
- Chamada ou execução de construtores;
- Execução de inicialização de classe e objetos;
- Acesso a campos;

- Execução de tratamento de exceções;
- Execução de adendo.

2.2.3. Conjunto de Junção

Os conjuntos de junção são responsáveis por selecionar pontos de junção, ou seja, eles detectam em que ponto do programa os aspectos deverão interceptar.

Na Figura 11 é mostrada uma sintaxe de um conjunto de junção:

```
<pointcut> ::= <access_type> <pointcut_name> ( { <parameters> } )
: { designator [ && | || ] };
<access_type> ::= public | private [abstract]
<pointcut_name> ::= { <identifier> }
<parameters> ::= { <identifier> <type> }
<designator> ::= [!]Call | execution | target | args | cflow |
cflowbelow | staticinitialization | within | if | adviceexecution |
preinitialization
<identifier> ::= letter { letter | digit }
<type> ::= defined valid Java type
```

Figura 11 – Sintaxe de um conjunto de junção (Gradecki e Lesiecki, 2003)

Um conjunto de junção pode ser aninhado em outro conjunto de junção por meio do uso dos operadores lógicos “e”, “ou” e “não” (&&, ||, !).

Além do uso de operadores lógicos, a Linguagem AspectJ utiliza elementos denominados coringas (*Wildcards*). Esses elementos permitem especificar assinaturas (*signature*) e são representados por “*”, “..” e “+” (“*”: qualquer seqüência de caracteres não contendo pontos; “..”: qualquer seqüência de caracteres, inclusive pontos; “+”: qualquer subclasse de uma classe).

Na Linguagem AspectJ, os conjuntos de junção são definidos por meio de construtores denominados de designadores. Esses designadores podem ser vistos na Tabela 1:

Tabela 1 – Designadores de pontos de junção (Gradecki e Lesiecki, 2003)

Designadores	Descrição
Execution	Corresponde á execução de um método ou de um construtor.
Call	Corresponde à chamada para um método ou para um construtor.
Inicialization	Corresponde à inicialização de um objeto, representada pela execução do primeiro construtor para uma classe.
Staticinicialization	Corresponde à inicialização dos elementos estáticos de um objeto.
Preinicialization	Corresponde à pré-inicialização de um ponto de junção.
Handler	Corresponde à manipulação das exceções.
Get	Corresponde à referência para um atributo de uma classe.
Set	Corresponde à definição de um atributo de uma classe.
This	Retorna o objeto associado com o ponto de junção em particular ou limita o escopo de um ponto de junção utilizando um tipo de classe.
Target	Retorna o objeto alvo de um ponto de junção ou limita o escopo do mesmo.
Args	Expõe os argumentos para ponto de junção ou limita o escopo de um conjunto de junção.
Cflow	Retorna os pontos de junção na execução fluxo de outro ponto de junção.
Cflowbelow	Retorna os pontos de junção na execução do fluxo de outro ponto de junção, exceto o ponto de junção corrente.
Withincode	Corresponde aos pontos de junção contidos em um método ou construtor.
Within	Corresponde aos pontos de junção contidos em um tipo específico.
If	Permite que uma condição dinâmica faça parte de um conjunto de junção.
Adviceexecution	Corresponde ao adendo do ponto de junção.

2.2.4. Adendo

Os adendos são trechos de código semelhantes a métodos de uma classe que executam quando algum ponto de junção definido em um conjunto de junção for capturado (Kiczales, 2001a). Existem três tipos de adendos, são eles: *before*, *after* e *around*.

- *before*: é executado no momento em que o pontos de junção associados são alcançados, mas imediatamente antes de seu processamento.
- *after*: é executado após o pontos de junção associados serem alcançados e antes do retorno do controle ao chamador, tendo ocorrido exceção ou não. Especialmente, o adendo do tipo *after* pode ser subdividido em dois tipos: *after returning* e *after throwing*. *After returning* é executado após o processamento

executar com sucesso o ponto de junção, sem ocorrência de exceção. *After throwing* é executado após o processamento executar sem sucesso o ponto de junção, com ocorrência de exceção.

- *around*: é executado no momento em que o pontos de junção associados são alcançados e tem total controle sobre o fluxo de execução do programa, podendo devolver ou não o controle ao programa.

2.2.5. Aspectos

Aspectos são unidades modulares para a execução de interesses transversais (Kiczales, 2001a). Em outras palavras, aspectos são unidades que permitem encapsular os interesses transversais em módulos isolados com a finalidade de entrecortar classes ou objetos de um sistema e assim melhorar a separação desses interesses transversais.

Semelhante a classes do paradigma de orientação a objeto, os aspectos podem conter elementos como: variáveis, atributos, métodos e tudo que uma classe possa ter. Em AspectJ, além desses elementos, os aspectos podem conter outros elementos como: os conjuntos de junção, adendos e declaração intertipos.

Na Figura 11 é mostrada uma sintaxe de um aspecto:

```
aspect ::= <access> [privilege] [static] aspect identifier>
<class identifier><instantiation>
<access> ::= public | private [abstract]
<identifier> ::= letter { letter | digit }
<class identifier> ::= [dominates] [extends]
<instantiation> ::= [issingleton | perthis | pertarget |
percflow | perflowbelow]
// pontos de junção
// adendos
// métodos/atributos
}
```

Figura 12 – Sintaxe de um aspecto (Gradecki e Lesiecki, 2003)

Em seguida, nas Figuras 12 e 13 serão mostrados, respectivamente, um simples exemplo da construção de uma aplicação em Java e um aspecto em AspectJ que entrecorta a aplicação e exibe duas mensagens, uma antes e outra depois da mensagem da aplicação.

```java
public class OlaMundo {
    public void exibeMensagem{
        System.out.println("Ola Mundo!");
    }
    public static void main(String args[]){
        OlaMundo ola = new OlaMundo();
        ola.exibeMensagem();
    }
}
```

Figura 13 – Exemplo de aplicação Java

```java
public aspect TesteOlaMundo {
    pointcut callExibeMensagem(): call(public void OlaMundo.*(..));

    before(): callExibeMensagem(){
        System.out.println("Antes do exibeMensagem");
    }

    after(): callExibeMensagem(){
        System.out.println("Depois do exibeMensagem");
    }
}
```

Figura 14 – Exemplo de aspecto em AspectJ

O resultado dessa aplicação é mostrado na Figura 15:

```
Antes do exibeMensagem
Ola Mundo!
Depois do exibeMensagem
```

Figura 15 – Resultado da saída da classe `OlaMundo`

2.2.6. Declaração Inter-tipos

Gradecki *et al.* (2003) descrevem a declaração inter-tipos como interesse estático (*static crosscutting*), o qual é utilizado para alterar a estrutura estática de uma aplicação. Com o uso de declarações inter-tipos é possível realizar alterações como:

- Adicionar membros (métodos, construtores, campos) para tipos (incluindo outros aspectos);
- Adicionar implementação concreta para interfaces;

- Declarar qual tipo estende um novo tipo ou implementar uma nova interface;
- Declarar a precedência do aspecto;
- Declarar erros de compilação ou avisos;
- Converter exceções checadas para exceções não checadas.

2.3. Padrão Capturador de Dados

Gamma *et al.* (1995) definem padrões de projeto como “... a descrição planejada de objetos e classes interconectadados para a resolução de um problema genérico de projeto em um contexto em particular”. Um padrão de projeto nomeia, identifica e abstrai os aspectos chave de uma estrutura de projeto identificando uma estrutura para criação de objetos reutilizáveis. Nesse contexto eles são independentes da tecnologia e da arquitetura dependendo somente do domínio do problema de projeto e do contexto em que o mesmo acontece.

Gamma *et al.* (1995) propõem o uso de padrões de projeto de software como um novo mecanismo para expressar soluções na elaboração de projetos orientados a objetos. Esses padrões fornecem uma linguagem comum que irá facilitar a comunicação entre desenvolvedores e projetistas, melhoram o aprendizado de jovens desenvolvedores incrementando a padronização do desenvolvimento, permitem a construção de softwares reutilizáveis que se comportam como blocos de construção para sistemas mais complexos.

O padrão Capturador de Dados proposto por Camargo e Maseiro (2008), auxilia na generalização dos adendos, diferentemente da maioria dos trabalhos que procuram somente generalizar os conjuntos de junção. Com o uso desse padrão de projeto, o engenheiro de aplicação tem maior flexibilidade para identificar um ponto de junção no código-base para realizar o entrecorte por parte dos aspectos.

Na Tabela 2 são mostrados e descritos quatro alternativas de composição do padrão Capturador de Dados.

Tabela 2 – Alternativas de composição do padrão Capturador de Dados (Camargo e Masiero, 2008)

Alternativa de composição	Descrição
WithReturn	Esta alternativa consiste em estender o aspecto `WithReturn` e fornecer um ponto de junção cujo retorno é o valor a ser modificado. Esse ponto de junção pode ser tanto uma chamada quanto a execução de um método.
This	Esta alternativa consiste em estender o aspecto `This` e fornecer um ponto de junção que a partir do seu objeto *this* pode-se obter o valor a ser modificado. Também deve ser possível atribuir o valor modificado a esse mesmo objeto. Esse ponto de junção pode ser uma chamada ou execução de um método, ou o acesso a um atributo.
Target	Semelhante à alternativa anterior, porém o aspecto a ser estendido é `Target` e o objeto *`target`* do ponto de junção fornecido é que será utilizado para obtenção e atribuição do valor modificado.
Arguments	Semelhante à alternativa anterior, porém o aspecto a ser estendido é o `Arguments` e o parâmetro do ponto de junção fornecido será determinado pelo índice fornecido pelo engenheiro de aplicação por meio da concretização de um método abstrato.

Na Figura 16 é mostrado um diagrama de classe que apresentada como é utilizado o padrão de projeto.

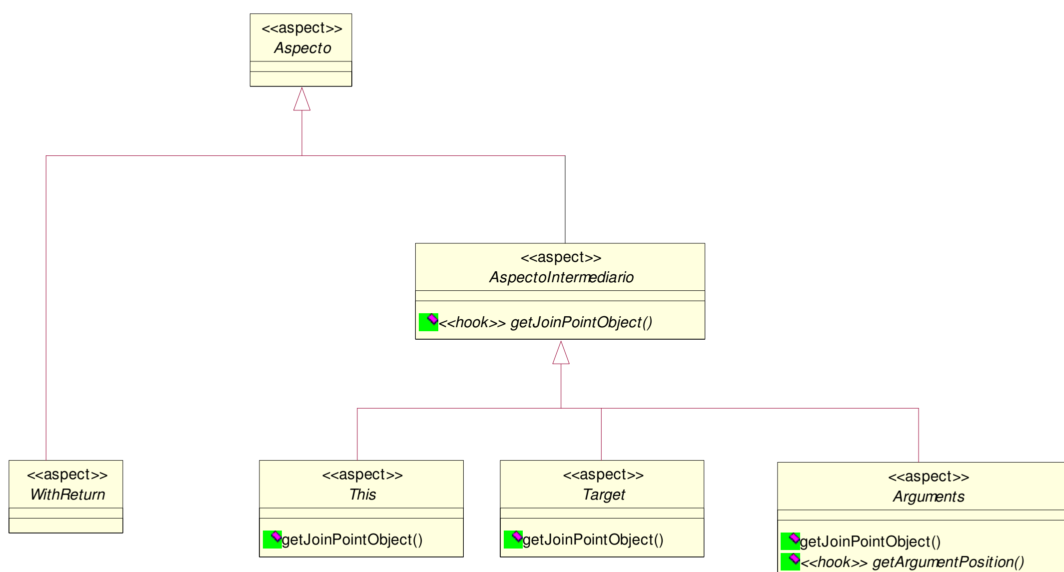

Figura 16 – Exemplo da Camada de Aspectos Genérica com uso do padrão Capturador de Dados

O estereótipo `<<hook>>`, utilizado no exemplo da Figura 16, representa métodos-gancho que devem ser sobrepostos pelo engenheiro de aplicação. Sua finalidade é facilitar a documentação e a identificação dos pontos de extensão dos aspectos (Fontoura *et al.*, 2002).

Os aspectos abstratos `Aspecto` e `AspectoIntermediario` são responsáveis pelos conjuntos de junção abstratos e adendos. O método abstrato `getPointObject()` é utilizado para obter um objeto do contexto de execução que possa ser utilizado para capturar o dado desejado. Esse método é redefinido em cada um dos aspectos especializados (`This`, `Target` e `Arguments`) com o objetivo de retornar diferentes objetos do contexto de execução, fornecendo assim três alternativas de composição (Camargo, 2006).

O aspecto abstrato `WithReturn` é uma alternativa que deve ser estendida se o objeto de interesse for um retorno de um método ou uma chamada.

Com base nessas quatro alternativas de composição, o engenheiro de aplicação escolhe uma alternativa para ser concretizada de acordo com o dado requerido no ponto de junção identificado na aplicação base.

2.4. Considerações Finais

Com o uso da criptografia podem-se atender os requisitos de segurança e a combinação da criptografia simétrica e assimétrica em um sistema híbrido, soma os benefícios e cobre os pontos negativos das duas formas de criptografia.

A Programação Orientada a Aspectos permite que um sistema seja mais adequadamente modularizado separando os "interesses transversais", como exemplo a criptografia, dos "interesses-base" de um sistema.

Os padrões de projeto são definidos como uma solução para determinado problema recorrente com a finalidade de auxiliarem desenvolvedores e projetistas de software a usá-los toda vez que situações similares ocorrerem.

3. TRABALHOS RELACIONADOS

Nesta seção são apresentados trabalhos de segurança com ênfase em criptografia com o uso de POA.

3.1. Criptografia com aspecto

O uso da criptografia com POA ainda é pouco explorado, visto que poucos pesquisadores têm trabalhado nessa linha (Boström, 2004; Huang *et al.*, 2001). Boström (2004) descreve os passos realizados para a inclusão do interesse transversal de Criptografia em uma aplicação que não foi desenvolvida com esse interesse em mente nas primeiras fases do desenvolvimento. Uma aplicação, denominada Lumbago, foi utilizada por esse autor como estudo de caso.

Lumbago é uma pequena aplicação feita em Linguagem Java e usada pelos naprapaths[8]. Essa aplicação tem a finalidade de realizar o registro dos dados dos pacientes, como: nome, endereço, número de segurança social[9] e doenças, e anotações ocorridas durante os dias, como registros das atividades diárias e diagnósticos dos pacientes.

Os passos estabelecidos por Boström (2004) para a inclusão de criptografia em uma aplicação são mostrados a seguir com uma breve descrição:

1. Construir uma classe em Java para criptografar e decriptar. No estudo de caso descrito pelo autor, esse passo deve ser conduzido de forma normal por meio da implementação OO da classe. Para essa primeira etapa foi utilizada a biblioteca de classes Java Versão 1.4 que contem JCE[10] (*Java Cryptography Extension*).
2. Identificar os pontos de junção. Para a identificação dos pontos de junção o autor sugere que o código seja analisado e que os pontos de junção sejam divididos em duas categorias. Na primeira encontram-se os pontos de junção

---

[8] Naprapaths são médicos especializados na medicina alternativa chamada Naprapathy. Naprapathy é uma forma de tratamento baseado na manipulação manual do tecido conexivo inelástico que pode irritar o sistema nervoso e assim causar dor.

[9] Número da Segurança social é um número individual que actua como número de referência para todo sistema de segurança social do país, nesse caso a Suécia. Esse sistema assemalha-se ao INSS (Instituto Nacional do Seguro Social) no Brasil.

[10] Biblioteca que provê infra-estrutura de acesso e desenvolvimento de criptografia.

relacionados à escrita, ou seja, os que precisam ser criptografados e na segunda encontram-se os pontos de junção relacionados à leitura, ou seja, os que precisam ser decriptados. Eles também comentam que esses pontos de junção devam ser genéricos, pois dessa forma os aspectos podem ser reusados em outros contextos. Por exemplo, como Lumbago utiliza JDBC (*Java Database Connectivity*), as chamadas aos procedimentos das classes dessa biblioteca eram pontos muito prováveis de serem considerados como os pontos de junção, pois essas chamadas são independentes da lógica da aplicação.

O autor decidiu criptografar somente as *strings*. Então na realização desse passo, iniciou-se com a separação de todas as chamadas de escrita e leitura e aplicou-se a criptografia. Em relação à escrita, as chamadas ao método `setString()` da classe `PreparedStatement` e ao método `setObject()` da classe `ResultSet` é que foram selecionadas. Em relação à leitura considerou-se as chamadas ao método a `getString()` da classe `ResultSet` e ao método `getObject()` da classe `ResultSet`. Note-se que essas classes pertencem à biblioteca JDBC, o que faria os aspectos independentes da lógica da aplicação.

3. Construir conjuntos de junção para os pontos de junção identificados no passo 2. Nesse terceiro passo o objetivo foi a construção dos conjuntos de junção com base nos pontos de junção identificados anteriormente. Esse passo foi facilitado, pois a única tarefa realizada foi o agrupamento dos pontos de junção, encontrados anteriormente, em conjuntos de junção. Boström (2004) relata que há a necessidade de indicar não somente as chamadas que devam ser interceptadas pelos conjuntos de junção, mas também se devem verificar quais dados dessas chamadas é que precisam ser ligados aos adendos, isto é, quais dados precisam ser criptografados ou decriptados. Isso pode ser feito por meio da passagem de parâmetros entre conjuntos de junção e adendos.
4. Adicionar adendos que atuem sobre os conjuntos de junção definidos no passo 3. Nesse quarto passo o objetivo foi a criação de adendos para atuar sobre os conjuntos de junção definidos no passo anterior. Assim, o adendo do tipo `around` foi considerado a melhor opção, pois em muitos momentos, era necessário esperar o retorno de um método (conjunto de junção) e modificar

esse retorno para o código-cliente que havia chamado esse método. Essa estratégia não é possível de ser feita apenas com o adendo do tipo `before` e `after`.

5. Testar a aplicação. Na quinta etapa o objetivo era testar o novo módulo de criptografia introduzido e o funcionamento do resultado da integração com a aplicação. Para isso utilizou-se um conjunto de testes, o qual facilitou e agilizou na descoberta de problemas.

Após a realização do estudo de caso, os autores concluíram que a utilização da POA melhora a modularidade do sistema, a independência com a base de dados e resulta em menos código. Entretanto, sua utilização também traz como desvantagem a necessidade de treinamento com um novo tipo de programação.

Na Figura 17 é mostrado um exemplo dos efeitos da modularidade conseguida no uso da criptografia com aspectos. Neste exemplo, Boström (2004), por meio da ferramenta *Aspect Visualizer* do Eclipse, apresenta uma imagem mostrando onde os pontos de junção estão localizados dentro das 31 classes. As colunas brancas representam as seis classes afetadas pelo aspecto de criptografia. Essa figura fornece um panorama de como a criptografia estaria entrelaçada e espalhada caso não fosse implementada com POA.

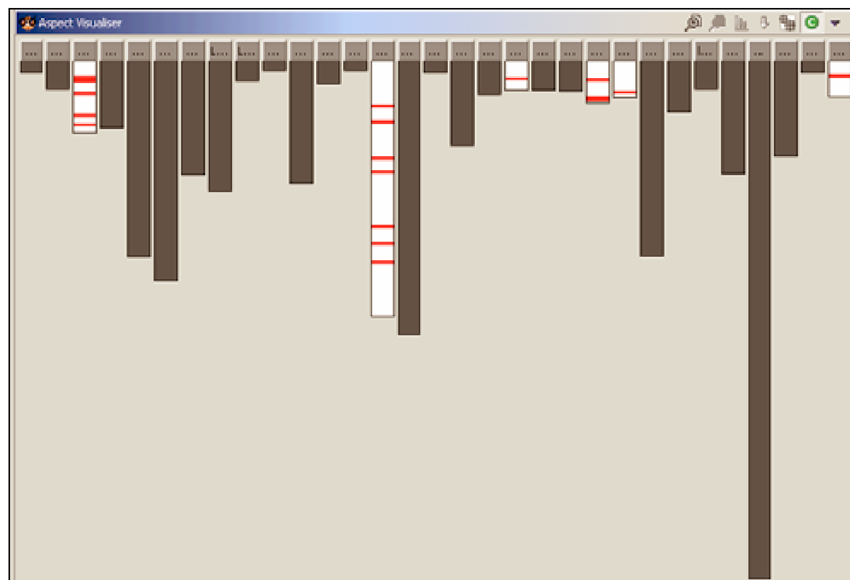

Figura 17 – Exemplo do efeito da criptografia com aspectos (Boström, 2004)

Huang *et al.*(2001) descrevem em AspectJ, uma biblioteca reusável e genérica com funções de segurança denominada JSAL (*Java Security Aspect Library*). A construção da biblioteca começou com a extração de funções de segurança de aplicações já existentes na literatura e posteriormente realizou-se a implementação dessas funções com POA.

Durante a construção da biblioteca utilizaram-se pacotes de segurança JCE e JAAS[11] (*Java Authentication and Authorization Service*), que são popularmente difundidos. Também foram utilizados aspectos abstratos e conjuntos de junção abstratos para obter bons níveis de generalidade e com isso contribuir para a reusabilidade dos aspectos.

JSAL é composta por quatro partes independentes: criptografia, autenticação, autorização e auditoria de segurança. Algumas dessas partes podem subdividir-se em outras subpartes, como o caso da criptografia que pode ser dividida em seus tipos, tais como o DES e o AES.

Um exemplo mostrado no artigo é o aspecto abstrato para criptografia com o uso do algoritmo DES, que pode ser visto na Figura 18. Esse exemplo possui dois conjuntos de junção abstratos: `encryptOperations()` e `decryptOperations(),` e adendos do tipo `around` que atuam sobre ambos os conjuntos de junção. Um passo importante a ser visto é que os conjuntos de junção foram construídos com um parâmetro do tipo *string*. Isso fez com que os pontos de junção que forem fornecidos ao aspecto concreto, devam obrigatoriamente possuir também somente um parâmetro desse tipo.

[11] API (*Application Programming Interface*) que provê autenticação de usuários para autorização de tarefas.

```
public abstract aspect AbstractDESAspect {
    public abstract pointcut encryptOperations(String msg);
    public abstract pointcut decryptOperations(String msg);

    public void around(String msg): encryptOperations(msg) {
       //Create encrypter class
       DesCipher enc = new DesCipher();
       enc.savekey("Deskey");
       //Encrypt
       String encryptedMsg = enc.encrypt(msg);
       proceed(encryptedMsg);
    }

    public void around(String msg): decryptOperations(msg) {
       //Create encrypter class
       DesCipher enc2=new DesCipher("Deskey");
       //Decrypt
       String decryptedMsg = enc2.decrypt(msg);
       proceed(decryptedMsg);
    }
}
```

Figura 18 – Exemplo de aspecto abstrato para criptografia em DES

O uso de aspectos abstratos possibilita a extensão desses aspectos, conseqüentemente, gera praticidade no momento da modelagem da aplicação. Assim o desenvolvedor da aplicação pode escolher usar ou não determinados conjuntos de junção conforme sua necessidade. Por exemplo, para o aspecto mostrado na Figura 18, o engenheiro de aplicação poderia querer apenas criptografar algum dado. Para isso, somente o conjunto de junção `encryptOperations()` deve ser concretizado no aspecto concreto.

O próximo exemplo, mostrado na Figura 19, é uma concretização do aspecto abstrato da Figura 18. Nesse exemplo existe a preocupação em proteger a mensagem e para isso é criptografado o envio e decriptada a recepção. Observando o exemplo, pode-se notar três importantes informações:

- a primeira, é a intenção dos autores em entrecortar quaisquer chamadas (`call`) aos métodos `sendMsg()` e `recvMsg()` e que contenham uma *string* como único parâmetro;
- a segunda, são os conjuntos de junções concretos. O conjunto de junção da Figura 19: `Public pointcut encryptOperations(String msg)` é a concretização do conjunto de junção abstrato que criptografa na Figura 18: `public abstract pointcut encryptOperations (String msg)`, da mesma forma, o outro conjunto de junção da Figura 19: `pointcut`

`decryptOperations(String msg)` é a concretização do conjunto de junção abstrato que decripta na Figura 18: `public abstract pointcut decryptOperations(String msg);`

- a terceira, é a preocupação em utilizar um designador de conjunto de junção do tipo `args` que permite capturar a *string* empregada nas chamadas dos métodos e utilizá-la dentro do adendo.

```
public aspect MyDESAspect extends AbstractDESAspect {
   public pointcut encryptOperations(String msg):
      call(String sendMsg(String))
      && args(msg);
   pointcut decryptOperations(String msg):
      call(String recvMsg(String))
      && args(msg);
}
```

Figura 19 – Exemplo de aspecto concreto sobre o exemplo da Figura 4

3.2. Considerações Finais

O uso da POA para o tratamento do interesse transversal da criptografia ainda é poucos explorado. Acrescentada a pouca exploração dessa área de estudo, o contexto de determinados trabalhos relacionados ao assunto (Boström, 2004; Huang et al., 2001), não fornecem diretrizes claras de como desenvolver uma camada de aspectos genérica para tratar esse interesse. Além disso, também não foi encontrada uma proposta de arquitetura que considera os aspectos apenas como “conectores” entre o interesse transversal e o base.

4. ARQUITETURA PROPOSTA E ESTUDO DE CASO

Nesta seção é apresentada tanto a arquitetura de criptografia com POA proposta quanto o estudo de caso que foi realizado para a obtenção dessa arquitetura.

4.1. Descrição Genérica da Arquitetura

Neste trabalho, é proposta uma arquitetura evoluída a partir de diretrizes formuladas por Fujii *et. al*.(2007a). Essa arquitetura é composta de três níveis horizontais com a finalidade de isolar o interesse transversal de criptografia de um lado e a aplicação base do outro. A arquitetura proposta, representada pela Figura 20, é composta de três camadas bem definidas: 1) parte base (funcional) da aplicação implementada com o paradigma OO, 2) aspectos genéricos e 3) interesse transversal de criptografia implementada com o paradigma OO.

A vantagem em possuir uma camada de aspectos genéricos que agem como conectores entre as duas partes OO é a melhoria na modularização do interesse de criptografia. Conseqüentemente, a melhoria na modularização facilita o acoplamento da biblioteca de criptografia a outros códigos-base sem que esses tenham consciência de sua presença, melhorando também seu nível de reúso.

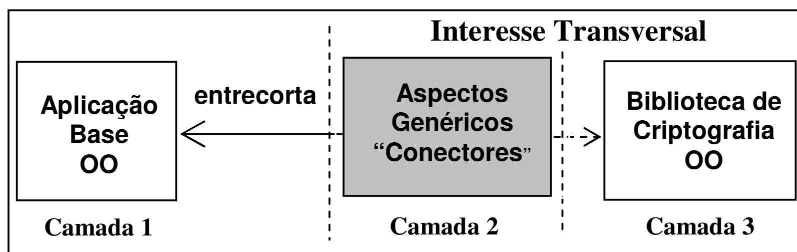

Figura 20 – Arquitetura de três camadas para isolamento do interesse de criptografia da aplicação base (Fujii *et. al.*, 2007b)

Na arquitetura proposta, a camada de aspectos genéricos foi desenvolvida com a finalidade de agir como intermediária entrecortando pontos da aplicação base e utilizando a biblioteca de criptografia. Essa camada utiliza o padrão de projeto proposto por Camargo e

Masiero (2008), denominado padrão Capturador de Dados. Diferentemente de outros trabalhos relacionados à POA, o objetivo dessa arquitetura é não incluir o código do interesse transversal de criptografia dentro dos aspectos, como ocorre com a maior parte dos trabalhos (Boström, 2004; Huang et al., 2001). Ao contrário disso, o objetivo aqui é apenas a utilização dos aspectos como conectores entre as duas partes OO. Dessa forma, a biblioteca de criptografia permanece intacta de um lado e a aplicação base da mesma forma do outro.

Para exemplificar essa arquitetura, na Figura 21, por meio de um diagrama de classe, é mostrado um exemplo de sistema bancário com uso de criptografia utilizando a arquitetura proposta.

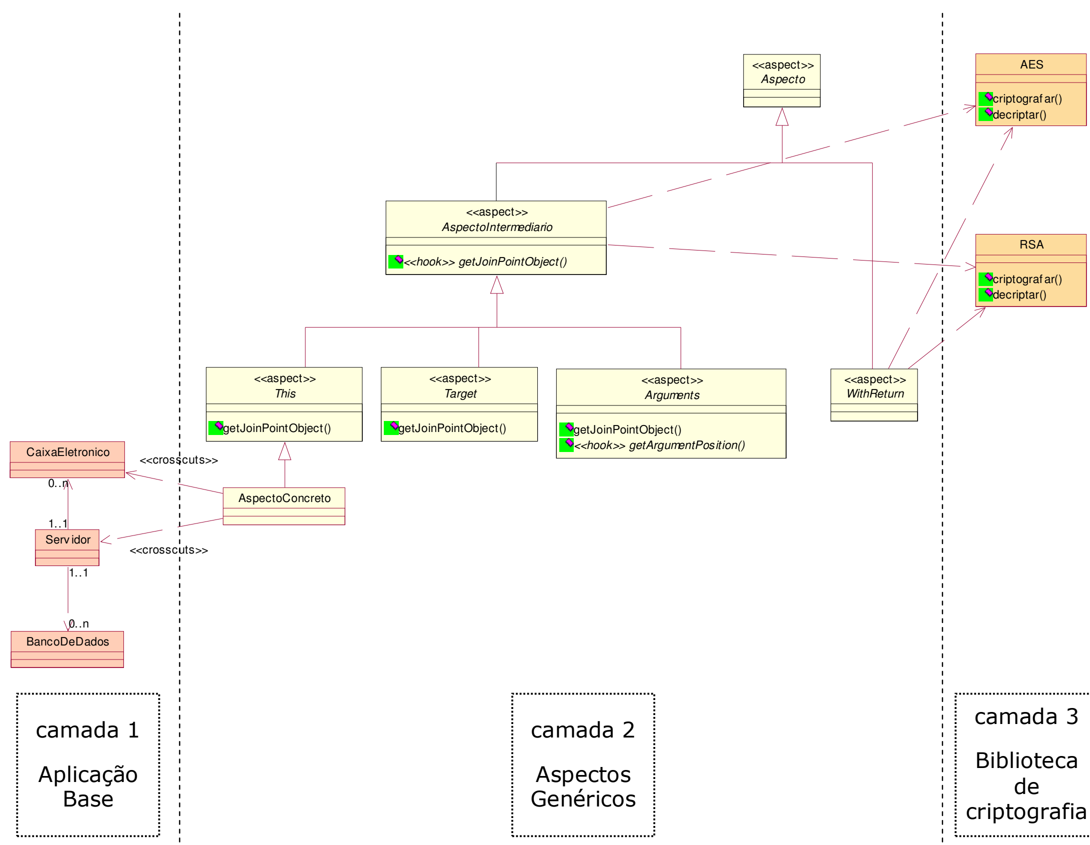

Figura 21 – Exemplo de sistema desenvolvido utilizando a arquitetura proposta

Na Figura 21 pode-se observar a separação do sistema bancário em três camadas distintas. A camada 1 corresponde à aplicação base e é formada pelas classes `CaixaEletronico`, `Servidor` e `BancoDeDados`. Sua função primária é realizar

operações bancárias por meio de um caixa eletrônico que está interligado a um servidor que retorna as solicitações pedidas pelo caixa eletrônico consultando um banco de dados.

Na camada 2 é realizada a conexão ente as camadas 1 e 3 e é formada por aspectos abstratos e um aspecto concreto. Os aspectos abstratos são implementados de acordo com o padrão Caputador de Dados. Os aspectos abstratos `WithReturn` e `AspectoIntermediario`, contidos na camada de aspecto, possuem chamadas a métodos de classes que realizam a criptografia, ou seja, fazem a conexão com a camada 3. O aspecto abstrato `Aspecto` possui conjuntos de junção abstratos que selecionam os pontos de junção E por fim, os aspectos `abstratos This`, `Target` e `Arguments` que, junto com `WithReturn`, são alternativas para a concretização por meio do `AspectoConcreto`.

A fim de ilustração, o aspecto abstrato `This` foi o escolhido dentre as quatro alternativas como a melhor opção para concretização, não esquecendo que o aspecto deve ser concretizado de acordo com o ponto de junção identificado na aplicação base. Esse aspecto concreto entrecorta pontos específicos das classes `CaixaEletronico` e `Servidor` onde se requer o uso da criptografia, ou seja, ele realiza a conexão com a camada 1.

Na camada 3, também como ilustração, são apresentadas duas classes que são responsáveis pela realização da criptografia.

Por meio do exemplo do sistema bancário, evidencia-se uma situação real para melhor esclarecer a influência da estrutura em um sistema. A situação em questão é uma solicitação de saldo por um usuário. Quando o usuário solicita seu saldo em um caixa eletrônico as seguintes ações são geradas: 1) para validar a solicitação de saldo, o cliente precisa informar sua senha correta; 2) a senha precisa ser conferida no servidor para ser validada, contudo não é seguro o tráfego de uma senha bancária por uma rede; 3) antes do envio da senha ao servidor, o aspecto `AspectoConcreto` entrecorta a classe `CaixaEletronico` para criptografar a senha; 4) ainda na camada de aspectos é feita uma chamada ao método `criptografar()`das classes `AES` ou `RSA` e finalmente, é aplicada a criptografia na senha; 5) a camada de aspecto devolve a senha, agora criptografada, e o fluxo de execução à classe `CaixaEletronico`; 6) a classe `CaixaEletronico` envia a senha criptografada à classe `Servidor`; 7) antes da classe `Servidor` validar a senha, a camada de aspecto a entrecorta e realiza o processo inverso da criptografia, ou seja, pega a senha criptografada, faz chamadas ao método `decriptar()`, decripta a senha e a devolve

à classe `Servidor`. 8) a classe `Servidor` confere a senha no banco de dados e caso seja válida, ele devolve o valor do saldo à classe `CaixaEletronico`; 9) e por fim, para o envio do valor do saldo, acontece um processo semelhante ao ocorrido no envio da senha. Com a diferença que agora quem envia um dado é a classe `Servidor` para a classe `CaixaEletronico`.

4.2. Estudo de Caso

O estudo de caso é um sistema composto de uma aplicação base, de uma camada de aspectos genéricos e uma biblioteca de criptografia. A aplicação base é um sistema cliente/servidor, desenvolvido em Linguagem Java, para a realização de cadastro de usuários da aplicação base, a camada de aspectos genéricos utiliza o Padrão Capturador de Dados e foi desenvolvido em Linguagem AspectJ e para a biblioteca de criptografia utilizou-se o pacote Cripto.

O objetivo do estudo de caso é verificar a possibilidade e os problemas encontrados na tentativa do acoplamento de uma biblioteca de criptografia já finalizada em uma aplicação sem fazer alterações tanto na aplicação quanto na biblioteca de criptografia.

Em virtude do estudo de caso foi possível elaborar três diretrizes para a criação da camada de aspectos genéricos. Essas diretrizes são: construir duas hierarquias de aspectos de acordo com o padrão Capturador de Dados, desenvolver adendos e testar a aplicação.

A primeira diretriz (construir duas hierarquias de aspectos de acordo com o padrão Capturador de Dados) consiste em construir duas hierarquias de aspectos, uma responsável por criptografar e outra por decriptar as informações, independentemente das extremidades – cliente ou servidor. Sugere-se que essas hierarquias sejam implementadas de acordo com o padrão proposto por Camargo e Masiero (2008), pois este padrão garante que os aspectos tornem-se genéricos.

A segunda diretriz (desenvolver adendos) consiste em criar adendos e adicionar a esses as chamadas aos métodos da biblioteca que, de outra forma, estariam espalhados nos módulos da aplicação.

A terceira diretriz (testar a aplicação) envolve a realização de testes a fim de verificar a correta realização da criptografia.

4.2.1. Aplicação Base

A aplicação base utilizada nesse estudo de caso é um sistema de cadastros utilizado em compras *on-line*, onde o usuário informa seus dados por meio de um *applet* (representado no sistema pela classe `ClienteGUI`), os quais são enviados ao servidor (classe `Server`) para que sejam armazenados em um banco de dados e posteriormente consultados se necessário.

Na Figura 22 é mostrada o diagrama de classe do sistema de cadastros.

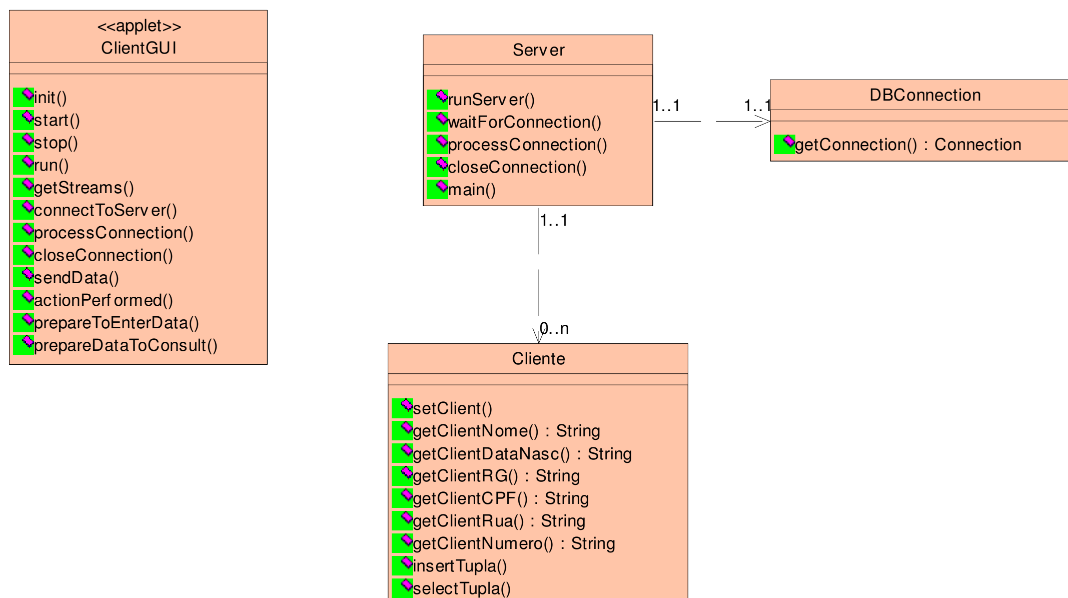

Figura 22 – Diagrama de classe do sistema de cadastros

A classe `ClienteGUI` faz a interface com o usuário. O usuário faz o *download* desse *applet* de uma página *web*. Assim que `ClienteGUI` é iniciado ele faz a conexão, conforme sua restrição de segurança[12], com a classe `Server`. A classe `Server` recebe o pedido de conexão e após aceitá-lo, fica aguardando algum pedido de `ClienteGUI`. Esse pedido aguardado pelo `Server` pode ser um cadastro enviado pela interface para concretizar uma compra ou apenas um consulta ou alteração do cadastro já existente.

[12] Os navegadores atuais impõem as seguintes restrições em qualquer *applet* que é carregado por meio de uma rede (Java Sun, 2007a): 1) Um *applet* não pode carregar bibliotecas ou definir métodos nativos; 2) Não pode ordinariamente ler ou escrever arquivos na máquina que o está executando; 3) Não pode fazer conexões de rede, exceto com o servidor do qual ele está hospedado; 4) Não pode iniciar qualquer programa na máquina que o está executando; 5) Ela não pode ler certas propriedades sistema; 6) A aparência de uma janela a qual um *applet* é iniciado é diferente da aparência da janela de uma aplicação.

Além das classes `ClienteGUI` e `Server`, a aplicação base também contém as classes `Client` e `DBConnection`. A classe `Client` possui métodos para o cadastramento, consulta ou alteração dos dados do usuário no banco de dados e a classe `DBConnection` realiza a conexão do servidor com o banco de dados.

4.2.2. Camada de Aspectos

A camada de aspectos é formada por uma hierarquia de aspectos de acordo com o padrão Capturador de Dados, e é composta de 14 aspectos e uma classe. Dentro da camada de aspectos, existe uma divisão em duas partes: uma relacionada à criptografia e outra relacionada à decriptografia.

Cada uma dessas duas partes implementa o padrão de projeto. Na Figura 23 é mostrada a parte relacionada à criptografia.

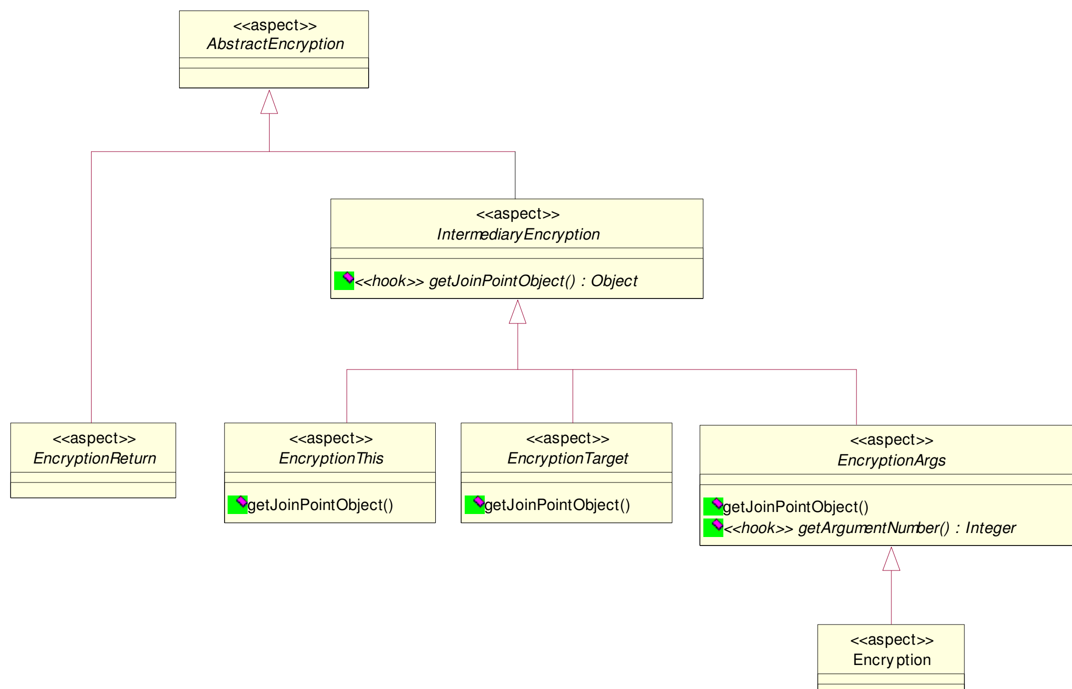

Figura 23 – Diagrama de classe da parte de criptografia da camada de aspectos genéricos

Na parte da criptografia, a hierarquia começa com o aspecto abstrato `AbstractEncryption` que contém um conjunto de junção abstrato. Seu código fonte é apresentado na Figura 24.

```
public abstract aspect AbstractEncryption {
    public abstract pointcut callCriptografaMsg();
}
```

Figura 24 – Aspecto abstrato: `AbstractEncryption`

Estendendo do aspecto `AbstractEncryption`, vêm os aspectos, também abstratos, `IntermediaryEncryption` e `EncryptionReturn`. Ambos possuem adendos do tipo *around* e chamadas a métodos das classes `AES` e `RSA` da biblioteca Cripto de Chiaramonte (2006) para a realização da criptografia por meio do sistema híbrido. Esses aspectos fazem a conexão entre camada de aspectos e a biblioteca de criptografia.

Além da criptografia, o aspecto `IntermediaryEncryption` possui um método abstrato denominado `getJoinPointObject()` que possui um retorno do tipo `Object` e um parâmetro do tipo `JoinPoint`. O código fonte de `IntermediaryEncryption` é apresentado nas Figuras 25.

```
public abstract aspect IntermediaryEncryption extends
    AbstractEncryption {

    before(): callCriptografaMsg(){

                        .
        Métodos para a realização da criptografia
                        .

    }

    public abstract Object getJoinPointObject(JoinPoint joinPoint);
}
```

Figura 25 – Aspecto abstrato: `IntermediaryEncryption`

Diferente de `IntermediaryEncryption` que retorna `void`, `EncryptionReturns` tem retorno do tipo `Vector`. Outra característica desse aspecto é que ele faz parte das alternativas de composição do padrão de projeto. Particularmente, essa alternativa consiste em fornecer a possibilidade de estender a um aspecto cujo ponto de junção possui um retorno o qual deseja ser modificado. Podendo, esse ponto de junção ser uma chamada ou uma execução de um método na aplicação base. Na Figura 26 é apresentado o aspecto `EncryptionReturn`.

```
public abstract aspect EncryptionReturn extends
    AbstractEncryption{

    Vector around(): callCriptografaMsg(){

                            .
        Métodos para a realização da criptografia
                            .

    return(vCriptografado);
}
```

Figura 26 – Aspecto Abstrato: `EncryptionReturn`

Os últimos aspectos abstratos da hierarquia são `EncryptionThis`, `EncryptionTarget` e `EncryptionArgs`. Todos eles concretizam o método `getJoinPointObject()` da classe `IntermediaryEncryption`() de acordo com sua finalidade.

O `EncryptionThis` é uma alternativa que consistem em estender o aspecto `IntermediryEncryption` e fornecer um ponto de junção que a partir de seu objeto *this*, pode-se obter o valor a ser modificado. Esse ponto de junção pode ser uma chamada ou uma execução de um método, ou ainda pode ser um acesso a um atributo. Seu código fonte é apresentado na Figura 27.

```
public abstract aspect EncryptionThis extends IntermediaryEncryption{
    public Object getJoinPointObject(JoinPoint jp){
        return jp.getThis();
    }
}
```

Figura 27 – Aspecto abstrato: `EncryptionThis`

O `EncryptionTarget` é uma alternativa semelhante à alternativa anterior, com a diferença de que, nesse caso, será utilizado o objeto *target*. Na Figura 28 é apresentado seu código fonte.

```
public abstract aspect EncryptionTarget extends
    IntermediaryEncryption{

    public Object getJoinPointObject(JoinPoint jp){
        return jp.getTarget();
    }
}
```

Figura 28 – Aspecto abstrato: `EncryptionTarget`

O `EncryptionArgs` é também uma alternativa semelhante à `EncryptionThis`, contudo o aspecto a ser estendido é o `EncryptionArgs` que utiliza parâmetros. O parâmetro fornecido pelo ponto de junção é determinado pelo índice fornecido pelo engenheiro de aplicação, quando este concretizar o método abstrato `getArgumentNumber()` presente nesse aspecto. Na Figura 29 é apresentada o código fonte desse aspecto.

```
public abstract aspect EncryptionArgs extends IntermediaryEncryption{

    public Object getJoinPointObject(JoinPoint jp){
        Object[] objects = jp.getArgs();
        return objects[getArgumentNumber()];
    }

    public abstract int getArgumentNumber();
}
```

Figura 29 – Aspecto abstrato: `EncryptionArgs`

No final da hierarquia está o aspecto `Encryption` que faz uma parte do acoplamento com a aplicação base. Neste estudo de caso, o aspecto abstrato `EncryptionArgs` foi o escolhido para ser concretizado pelo aspecto `Encryption`.

A escolha foi feita devido ao ponto de junção indicado para ser entrecortado. O ponto de junção é o método `writeObject()` que possui um argumento do tipo `Vector` (ponto de junção da criptografia: "`output.writeObject(v);`"). Como esse método possui apenas um argumento o índice do método `getArgumentNumber()` é zero.

Na Figura 30 é apresentada o aspecto `Encryption`.

```
public aspect Encryption extends EncryptionArgs{

     public pointcut callCriptografaMsg():
         call(* *.writeObject(..)) &&
         !within(FileCryptography);

     public int getArgumentNumber(){
         int argumentNumber = 0;
         return argumentNumber;
     }
}
```

Figura 30 – Aspecto concreto: `Encryption`

O uso do designador `within` é devido à necessidade de discriminar as classes que serão entrecortadas pela camada de aspectos genéricos. Visto que a classe `FileCryptography` da parte de decriptografia também possui o mesmo método indicado como ponto de junção, foi preciso excluí-la da relação de classes afetadas pelo conjunto de junção do aspecto `Encryption`.

A outra parte da camada de aspectos genéricos está relacionada à decriptografia, mostrada na Figura 31.

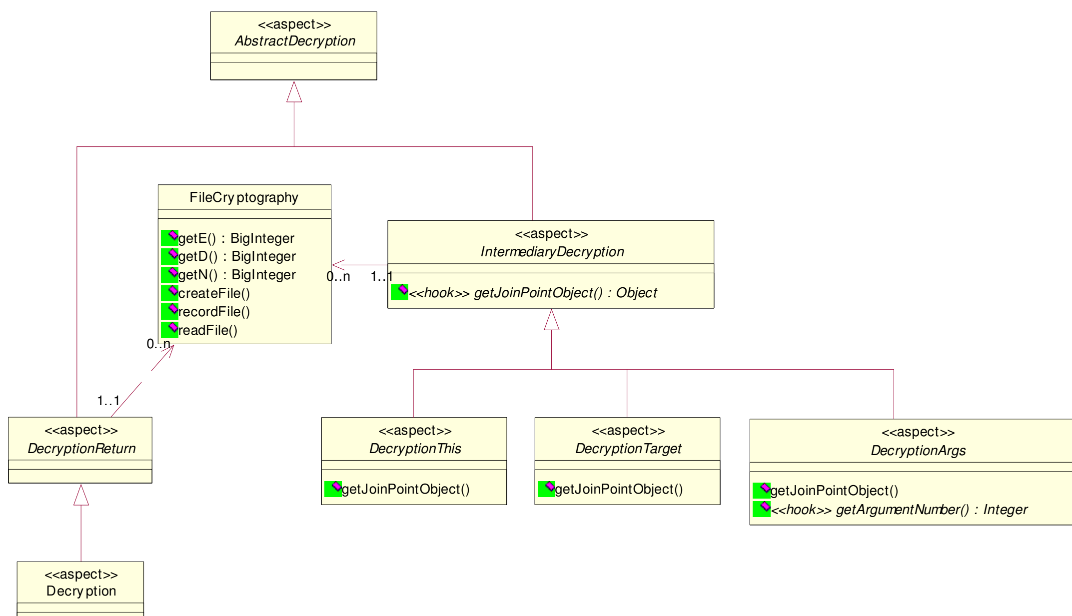

Figura 31 – Diagrama de classe da parte de decriptografia da camada de aspectos genéricos

A parte de decriptografia é muito semelhante à parte de criptografia, porém possui algumas diferenças. A primeira das diferenças são as chamadas a métodos relacionados a decriptografia e não a criptografia, a segunda diferença é a presença de uma classe chamada `FileCryptograhy` que realiza ações pertinentes a criptografia assimétrica. E a terceira diferença é a escolha pela concretização do aspecto `DecryptionReturn` dentre as quatro alternativa possíveis.

`DecryptionReturn` foi escolhido para ser concretizado devido ao ponto de junção da decriptografia retornar o dado que se deseja modificar. Esse ponto de junção é o método `readObject()` (Ponto de junção da decriptografia: "`v = (Vector) input.readObject();`").

A concretização do `DecryptionReturn` é feita no aspecto `Decryption`. Esse aspecto faz a outra parte do acoplamento com a aplicação base e é apresentado na Figura 32.

```
public aspect Decryption extends DecryptionReturn{
    public pointcut callDecriptaMsg():
        call(* *.readObject(..)) && !within(FileCryptography);
}
```

Figura 32 – Aspecto concreto: `Decryption`

4.3. Aplicação da Arquitetura Proposta

A aplicação da arquitetura proposta neste estudo, por intermédio da camada de aspectos genéricos, faz o acoplamento da biblioteca de Chiaramonte (2006) com o sistema de cadastro.

Na Figura 33 é mostrado o diagrama de classe da aplicação da arquitetura proposta no sistema de estudo de caso. O diagrama de classe é formado de três camadas que são representadas por três cores diferentes. As camadas estão separadas e numeradas na seguinte ordem: a primeira camada é o sistema de cadastro, a segunda camada são os aspectos genéricos e a terceira camada é uma ilustração[13] da biblioteca de criptografia.

[13] O uso de uma ilustração da biblioteca de criptografia ao invés da própria biblioteca em si acontece pelo fato da biblioteca possuir muitas classes.

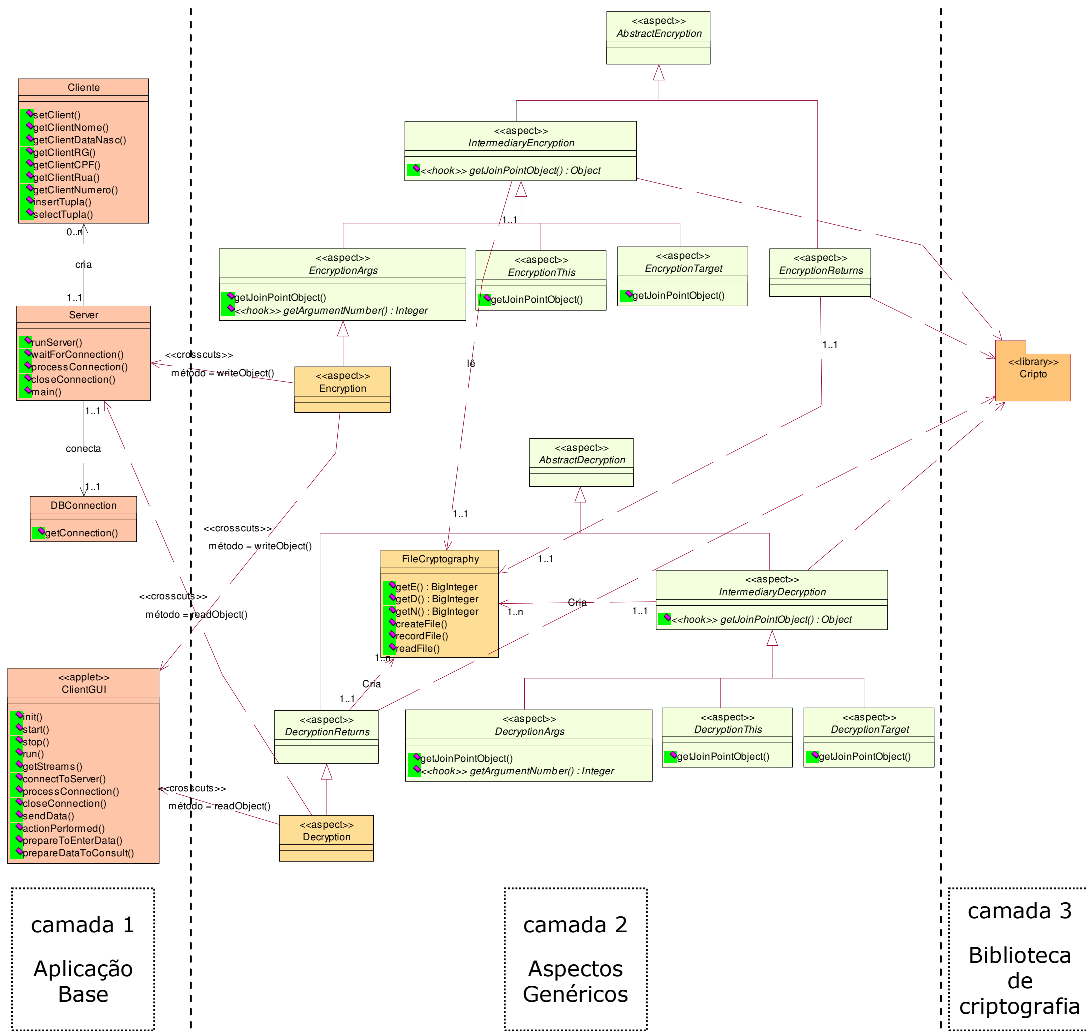

Figura 33 – Diagrama de classe da aplicação da arquitetura proposta no sistema de estudo de caso

Na camada de aspectos genéricos da Figura 33 pode-se observar, em coloração mais clara, o uso do padrão Capturador de Dados.

Um fato importante de ser observado é que tanto a classe `ClientGUI`, quanto a classe `Server` possuem idênticos pontos de junção que são entrecortados pelos aspectos concretos `Encryption` e `Decryption`.

Ambas as classes ClienteGUI e Server possuem os métodos `writeObject()` e `readObject()`. O método `writeObject()` é utilizado para enviar os dados de uma extremidade a outra extremidade da relação cliente/servidor, independente dos lados, e o método `readObject()` é utilizado para fazer a recepção dos dados.

Em conseqüência dessa particularidade, o aspecto `Encryption` utiliza o mesmo conjunto de junção para entrecortar dois pontos de junção, um no `ClienteGUI` e outro no `Server`.

O mesmo acontece com o aspecto `Decryption`, esse aspecto tem um conjunto de junção (`public pointcut callDecriptaMsg(): call(* *.readObject(..)) && !within(FileCryptography);`) que entrecorta o ponto de junção: "`v = (Vector) input.readObject();`", na classe `ClientGUI` e que também entrecorta o outro ponto de junção idêntico: "`v = (Vector) input.readObject();`", na classe `Server`.

Assim, um exemplo para melhor esclarecer o funcionamento do entrecorte em um processo hipotético de cadastro de um usuário, pode ser descrito nos seguintes passos:

1. Um usuário já de posse de um *applet* cadastra seus dados e aperta o botão de enviar os dados ao servidor;
2. A classe `ClienteGUI` grava todos os dados do usuário em um vetor e por meio do método `writeObject()` envia esse vetor a classe `Server`;
3. Quando o método `writeObject()` do `ClienteGUI` é executado, o aspecto `Encryption` entrecorta a classe `ClienteGUI`, toma posse do fluxo de execução e pega o argumento do ponto de junção ("`writeObject(vetor);`");
4. De posse do vetor de dados, o adendo executa a criptografia por meio de chamada a da biblioteca de criptografia. A criptografia usada é o sistema híbrido formado pelos algoritmos `AES` e `RSA`;
5. Após a criptografia dos dados dentro do vetor, o fluxo de execução é devolvido ao *applet* e então o vetor é enviado ao `Server`;
6. Antes do `Server` utilizar o vetor, o ponto de junção da classe `Server` é entrecortado pelo aspecto `Decryption` e seu retorno é capturado (ponto de junção: "`v = (Vector) input.readObject();`");
7. Dentro do adendo de `Decryption`, são feitas chamadas a métodos de decriptografia da biblioteca de criptografia para decriptar o vetor criptografado;
8. Após decriptar os dados dentro do vetor, o fluxo de execução é devolvido ao `Server` e então ele grava os dados do usuário no banco de dados.

4.4. Considerações Finais

O desenvolvimento do estudo de caso possibilitou a descoberta de vários problemas e pontos a serem observados na introdução do interesse transversal de criptografia em uma aplicação base.

Com isso podem-se estabelecer diretrizes para a construção de uma camada de aspectos genéricos para conexão de criptografia a uma aplicação. Para a descrição dessas diretrizes foram avaliadas lacunas existentes em trabalhos relacionados ao assunto e constatações feitas durante o desenvolvimento do estudo de caso.

A primeira diretriz que é a construção de duas hierarquias de aspectos de acordo com o padrão Capturador de Dados foi estabelecida tendo em vista as diferenças entre os pontos de junção relacionados à criptografia e à decriptografia. A escolha pelo padrão Capturador de Dados visa à generalização dos adendos.

A escolha pelo melhor ponto de junção é uma tarefa que deve ser muito bem avaliada, visto que os adendos e os aspectos que serão concretizados dependem do tipo do ponto de junção que será utilizado.

A segunda diretriz (desenvolver adendos) foi estabelecida pelo fato dos adendos serem a estrutura dentro do aspecto que possuirá as chamadas aos métodos para a realização da criptografia, ou seja, nos adendos, deve-se verificar quando o entrecorte deverá proceder: antes, durante ou depois do ponto de junção, como as informações são capturadas no momento do entrecorte pelos conjuntos de junção e como serão devolvidas posteriormente a aplicação base.

A terceira diretriz (testar a aplicação) foi estabelecida, pois é necessário testar se tudo o que foi anteriormente construído funcionará conforme o esperado.

Outro benefício obtido com o estudo de caso foi a proposta de um estrutura de três camadas bem definidas para a separação da aplicação base do interesse transversal de criptografia.

5. CONCLUSÕES

O estudo de caso conduzido neste trabalho possibilitou a elaboração de diretrizes para a criação de uma camada de aspectos genéricos que agem como conectores entre a aplicação base e uma biblioteca de criptografia, o que resultou no desenvolvimento de uma arquitetura de três níveis bem definidos.

A elaboração da camada de aspectos genéricos melhora a modularização do interesse transversal de criptografia, elimina o entrelaçamento e o espalhamento de código quando comparado com sua versão OO (Gradecki e Lesiecki, 2003). Conseqüentemente, tende facilitar o acoplamento da biblioteca de criptografia a outros códigos-base sem que esses tenham consciência da sua presença, podendo elevar os níveis de reúso e manutenibilidade da aplicação final.

Um ponto importante é que o estudo realizado também apresenta uma possível arquitetura para isolar bibliotecas OO existentes. Como trabalhos futuros, pretende-se elaborar experimentos e averiguar se a arquitetura proposta pode ser aplicada a quaisquer outras bibliotecas OO existentes.

REFERÊNCIAS BIBLIOGRÁFICAS

AspectJ Team. The AspectJ programming guide.Online. Disponível em <http://www.eclipse.org/aspectj/doc/released/progguide/index.html>. Acesso em 25 de junho de 2007.

Benits Júnior, W. D. Sistemas criptográficos baseados em identidades pessoais. 2003. Dissertação (Mestrado em Ciência da Computação) – Instituto de Matemática e Estatística da Universidade de São Paulo, São Paulo, 2006.

Boström, G. Database Encryption as an Aspect. In Belgian AOSD Workshop, 1st, 2004, Vrije Universiteit Brussel, Bélgica.

Camargo, V. V. Frameworks transversais: definições, classificações, arquitetura e utilização em um processo de desenvolvimento de software. 2006. Dissertação (Doutorado em Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação ICMC-USP, São Carlos, 2006.

Camargo, V. V.; Masiero, P. C. A Pattern to Design Crosscutting Frameworks. In: Proceedings of the Annual ACM Symposium on Applied Computing (ACM-SAC´08), 23, 2008, Fortaleza, Brasil. (*Aceito para publicação*)

Chiaramonte, R. B. Sico: Um Sistema Inteligente de Comunicação de Dados com Suporte Dinâmico a Segurança. 2006. Dissertação (Mestrado em Ciência da Computação) - Centro Universitário Eurípides de Marília, Marília, 2006.

Elrad, T.; Filman R. E.; Bader A. Aspect-Oriented Programming. Communications of the ACM, vol 44, p 29-32, 2001a.

Elrad, T.; Aksit, M.; Kiczales, G.; Lieberherr, K.; Ossher, H. Aspect-Oriented Programming. Communications of the ACM, vol 44, p 33-38, 2001b.

Fontoura, M.; Pree, W.; Rumpe, B. The UML Profile for Framework Architectures. Addison Wesley, 2002.

Fujii, C. S; Camargo, V. V.; Chiaramonte, R. B. Diretrizes para a Criação de Aspectos Genéricos para Criptografia. In: Congresso de Iniciação Cientifica, 15, 2007a, São Carlos. Anais de Eventos da UFSCar, v.3, p. 58.

Fujii, C. S; Chiaramonte, R. B.; Camargo, V. V. Camada de Aspecto para Modularização de Criptografia. In: Simpósio Internacional de Iniciação Cientifica da Universidade de São Paulo, 15, 2007b, São Carlos. (*Aceito para apresentação*)

Gamma, E., Helm, R., Johnson, R., Vlissides, J. – Design Patterns – Elements of Reusable of Object Oriented Software, 11. Addison-Wesley. 1995.

Gradecki, J. D.; Lesiecki, N, Mastering AspectJ: aspect-oriented programming in Java. Wiley Publishing, Inc. 2003

Horstamann, C. S.; Cornell, G. Core Java, volume I – fundamentals. Prentice Hall. 1999.

Huang, M.; Wang, C.; Zhang, Z. Toward a Reusable and Generic Security Aspect Library. AOSD 2004 Workshop - AOSD Technology for Application-level Security (AOSDSEC), 2004, Lancaster, UK

Java Sun. Disponível em <http://java.sun.com/docs/books/tutorial/deployment/applet/security.html>. Acesso em: 21 de maio de 2007a.

Java Sun. Disponível em http://java.sun.com/docs/books/tutorial/information/glossary.html#I. Acesso em: 20 de julho de 2007b.

Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C. V.; Loingtier, J.; Irwin, J. Aspect-Oriented Programming. In: Published in proceedings of the European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag LNCS 1241, junho de 1997, Xerox Palo Alto Research Center, Technical Report, Finland, p. 25

Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W. G. An Overview of Aspct, 2001a.

Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W. G. Getting Starded with AspectJ. Communications of the ACM, vol 44, p 59-65, 2001b.

Laboratório de Criptografia Aplicada (LCA) - Unicamp. Disponível em <http://www.lca.ic.unicamp.br/modules/pagewrap/>. Acessado em: 30 de maio de 2007.

Laddad, R. Java World. Disponível em <http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html> e <http://www.javaworld.com/javaworld/jw-03-2002/jw-0301-aspect2.html?page=5>. Acessado em: 13 de maio de 2007.

Meneze, A; van Oorschot, P; Vanstone, S. Handbook of Applied Cryptography. CRC Press, 1996. 816p. Livro. Disponível em <http://www.cacr.math.uwaterloo.ca/hac/>. Acesso em: 16 de maio de 2007.

Resende, A. M. P.; Silva, C. C. Programação Orientada a Aspectos em Java. Rio de Janeiro: Brasport, 2005.

Stallings, W. Cryptography and Network Security - Principles and Practice. 2. ed. Prentice Hall, 1998

______. Cryptography and Network Security - Principles and Practice. 4. ed. Prentice Hall, 2005. 592p.

Terada, R. Segurança de Dados- Criptografia em Redes de Computador. 1. ed. São Paulo: Edgard Blücher, 2000.

TKOTZ, Viktoria. Criptografia Numaboa. Disponível em <http://www.numaboa.com/content/view/157/57/>. Acesso em: 28 de abril de 2007.

Winck, D. V.; Goetten Junior, V. AspectJ: Programação Orientada a Aspectos com Java. São Paulo: Novatec Editora, 2006.

Zimmermann, P. Disponível em: <http://www.pgpi.org/docs/pgp26dc-brazilian-vol2.htm>. Acesso em 21 de novembro de 2007.