Instituto de Pesquisas Tecnológicas do Estado de São Paulo

Luiz Carlos Barboza Junior

JCFoundation, um *Framework* para desenvolvimento de aplicações Java em *Smart Cards* microprocessados

São Paulo
2008

Lombada

Luiz Barboza Junior

JCFoundation, um *Framework* para desenvolvimento de aplicações Java em Smart Cards microprocessados

Luiz Carlos Barboza Junior

JCFoundation, um *Framework* para Desenvolvimento de Aplicações Java em *Smart Cards* Microprocessados

Dissertação apresentada ao Instituto de Pesquisas Tecnológicas do Estado de São Paulo - IPT, para obtenção do título de Mestre em Engenharia da Computação.
Área de concentração: Engenharia de *Software*

Orientador: Dr. Fernando Antonio de Castro Giorno

São Paulo
2008

Ficha Catalográfica
Elaborada pelo Departamento de Acervo e Informação Tecnológica – DAIT
do Instituto de Pesquisas Tecnológicas do Estado de São Paulo - IPT

B238f Barboza Junior, Luiz Carlos
JCFoundation, um *Framework* para desenvolvimento de aplicações Java em *Smart Cards* microprocessados. / Luiz Carlos Barboza Junior. São Paulo, 2008.
103p.

Dissertação (Mestrado em Engenharia de Computação) - Instituto de Pesquisas Tecnológicas do Estado de São Paulo. Área de concentração: Engenharia de Software.

Orientador: Prof. Dr. Fernando Antonio de Castro Giorno

1. Smart cards 2. Framework Java Card 3. Cartão de memória 4. Tese I. Instituto de Pesquisas Tecnológicas do Estado de São Paulo. Coordenadoria de Ensino Tecnológico II. Título

08-01 CDU 004.087.5(043)

Agradecimentos

Dedico este trabalho a meu avô, Luiz Barboza e a meu tio, Antônio Galvão, *in memoriam*, que, apesar de não estarem presentes para acompanhar a conclusão desta dissertação, contribuíram muito para esse importante passo na minha formação acadêmica, e que, além de tudo, tiveram um papel fundamental na minha formação como pessoa.

Agradeço também a minha esposa, Simone, e a meus pais, Luiz e Mônica, por todo apoio e suporte, não apenas durante o período de estudo para confecção deste trabalho, mas por toda trajetória que me trouxe até aqui.

Ao meu orientador, Prof. Dr. Fernando Giorno, minha gratidão por todos os ensinamentos acadêmicos e pela paciência durante nossas seções de orientação, bem como a todo corpo de funcionários de IPT, que fazem desta instituição a excelência que é.

E por último, mas não menos importantes, aos meus irmãos, Eduardo e Cristiane, além de agradecer a todos os amigos que estão em tão grande número que não me atrevo a relacionar todos eles com receio de esquecer algum em particular.

Resumo

Como qualquer outra aplicação Java, as aplicações *Java Card* podem ter seu desenvolvimento facilitado por meio de um *framework* base. Com esse objetivo é que este trabalho apresenta um *framework Java Card* que endereça as questões mais comuns da família de problemas relacionados à construção de aplicações para *Smart Cards* microprocessados. Mais especificamente, esta dissertação procura estruturar a troca de comandos entre a aplicação Java no *Smart Card* e a leitora em que ele se encontra inserido, isolar a camada de persistência da camada de negócio da aplicação e por fim encapsular a camada de comunicação baseada no protocolo *Java Card* RMI. Essas soluções estão construídas com base em padrões de projetos existentes, similarmente realizados por outros autores, que construíram *frameworks* combinando padrões de projeto existentes. Com essa base inicial para a construção de aplicações *Java Card*, o desenvolvedor pode concentrar seus esforços na codificação da camada de negócio, utilizando-se de técnicas apresentadas por outros autores, como os métodos formais.

Por fim, o *framework* é validado por meio de um estudo de caso que utiliza aplicações exemplo já disponíveis na literatura.

Palavras-chave: *Java Card*, *Frameworks* e *Smart Cards*.

Abstract

As any other Java application, the Java Card applications can have its development improved by a base framework. With this objective, this research presents a Java Card framework which addresses the most common questions related with the application development for microprossed Smart Cards. More specifically, this dissertation aims to structure the command exchange between the Java application within the Smart Card and the reader in which it is inserted, isolate the persistence layer from the application business layer and finally, encapsulate the communication layer based on the Java Card RMI protocol. These solutions are build based on existing design patterns, similarly to other authors, who built frameworks based on existing design patterns. With this initial foundation for the Java Card application development, the programmer can focus his efforts on the business layer coding, supported by techniques presented by other authors, such as formal methods.

Last but not least, the framework is validated by a study case, which deploys the framework into sample applications available in the current literature.

Key words: ***Java Card*, *Frameworks* e *Smart Cards*.**

Lista de ilustrações

Figura 1 — Método de Desenvolvimento de *Frameworks* 18

Figura 2 — Características Físicas de um *Smart Card* 20

Figura 3 — Os Oito Pontos de Contato de um *Smart Card* 21

Figura 4 — Arquitetura *Java Card* 28

Figura 5 — Estrutura da Máquina Virtual de *Java Card* (JCVM) 30

Figura 6 — Estrutura do Instalador *Java Card* 31

Figura 7 — Comunicação por Meio de *Java Card* RMI 38

Figura 8 — Diagrama de Classe do Padrão *Command* Aplicado ao *Framework* ... 53

Figura 9 — Diagrama de Seqüência do Padrão *Command* Aplicado ao Framework 56

Figura 10 — Diagrama de Classe do Padrão *Bridge* Aplicado ao Framework 60

Figura 11 — Operações de Inicialização e de Adição 64

Figura 12 — Operações de Localização e de Remoção 65

Figura 13 — Operação de Atualização 66

Figura 14 — Diagrama de Classe do Padrão *Adapter* Aplicado ao *Framework* 70

Figura 15 — Operação de Inicialização dos Adaptadores 72

Figura 16 — Invocação Remota 73

Listagem 1 — Aplicação Exemplo *Wallet* 77

Listagem 2 — Fragmento de Código: *Wallet.java* 78

Listagem 3 — Fragmento de Código: *WalletApplet.java* 79

Listagem 4 — Fragmento de Código: *WalletAPDUDispatcher.java* 80

Listagem 5 — Fragmento de Código: *CreditCommand.java* 81

Listagem 6 — Aplicação Exemplo *HelloWorldEcho* 82

Listagem 7 — Fragmento de Código: *HelloWorldApplet.java* 83

Listagem 8 — Fragmento de Código: *EchoMessage.java* 84

Listagem 9 — Interface Remota de Negócio da Aplicação Exemplo 85

Listagem 10 — Implementação da Interface de Negócio da Aplicação Exemplo 86

Listagem 11 *Applet* da Aplicação Exemplo 87
Listagem 12 Classe Cliente da Aplicação Exemplo 88
Listagem 13 Fragmento de Código: *PurseApplet.java* 89
Listagem 14 Fragmento de Código: *PurseTargetAdapter.java* 90
Listagem 15 Fragmento de Código: *Purse.java* 90
Listagem 16 Fragmento de Código: *RemotePurse.java* 91
Listagem 17 Fragmento de Código: *PurseImpl.java* 92
Listagem 18 Fragmento de Código: *PurseException.java* 92
Listagem 19 Fragmento de Código: *PurseClient.java* 93
Listagem 20 Fragmento de Código: *PurseSourceAdapter.java* 94

Lista de tabelas

Tabela 1 Requisição APDU 25
Tabela 2 Resposta APDU 25
Tabela 3 Campos do Comando APDU 26
Tabela 4 Funcionalidades de Java Suportadas por *Java Card* 29
Tabela 5 Métodos da Classe *Applet* 34

Lista de abreviaturas e siglas

ABECS	Associação Brasileira das Empresas de Cartões de Crédito e Serviços
AID	*Application Identifier*
APDU	*Application Protocol Data Unit*
API	*Application Programming Interface*
CAD	*Card Acceptance Device*
EEPROM	*Electric Erasable Programmable Read Only Memory*
EJB	*Enterprise Java Beans*
ETSI	*European Telecommunications Standards Institute*
GSM	*Global System for Mobile Communications*
J2ME	Java 2 *Micro Edition*
JCRE	*Java Card Runtime Environment*
JCRMI	*Java Card Remote Method Invocation*
JCVM	*Java Card Virtual Machine*
JML	Java *Modeling Language*
JTA	Java *Transaction* API
JVML	Java *Virtual Machine Language*
JCF	*Java Card* Fórum
OCF	*Open Card Framework*
OCL	*Object Constraint Language*
PIX	*Personal Identifier Extension*
RAM	*Random Access Memory*
RID	*Resource Identifier*
RMI	*Remote Method Invocation*
ROM	*Read Only Memory*
SIMCards	*Subscriber Identity Module Smart Card*
SUS	Sistema Único de Saúde
WLAN	*Wireless Local Area Network*

Sumário

1 INTRODUÇÃO 11

1.1 Motivação.. 13

1.2 Objetivos Gerais e Específicos.. 16

1.3 Resultados Esperados e Contribuições... 16

1.4 Método de Pesquisa... 17

1.5 Organização do Trabalho 19

2 FUNDAMENTOS 20

2.1 *Smart Cards*.. 20

2.1.1 Benefícios do *Smart Cards*.. 21

2.1.2 Visão geral sobre *Smart Cards* 22

2.1.3 Cartões de memória ou cartões microprocessados.................................... 22

2.1.4 Sistema de memória dos *Smart Cards*.. 24

2.1.5 Modelo de comunicação dos *Smart Cards*.. 24

2.1.6 Padrões e especificações.. 26

2.2 *Java Card*... 27

2.2.1 Arquitetura 27

2.2.2 Subconjunto suportado da linguagem Java 29

2.2.3 Máquina virtual de *Java Card* 29

2.2.4 Instalador *Java Card* 30

2.2.5 Ambiente de execução *Java Card* (JCRE) 31

2.2.6 API de *Java Card* 32

2.2.7 Modelo de persistência 36

2.2.8 Comunicação por meio de *Java Card* RMI 37

3 ESTADO DA ARTE DE *JAVA CARD* 39

3.1 Verificação de Código Binário 39

3.2 Métodos Formais 41

3.3 Segurança em *Smart Cards* 44
3.4 Pesquisas Atuais sobre *Java Cards* 46
3.5 Conclusão 48

4 *FRAMEWORK* JCFOUNDATION.......... 50
4.1 Estruturação da Troca de Comandos entre a Aplicação Java no *Smart Card* e o Leitor de Cartões 51
4.1.1 Cenário de uso 51
4.1.2 Estrutura interna 52
4.1.3 Implentação dos pontos de extensão 57
4.2 Isolamento da Camada de Persistência da Camada de Negócio da Aplicação 57
4.2.1 Cenário de uso 58
4.2.2 Estrutura interna 59
4.2.3 Implementação dos pontos de extensão 66
4.3 Encapsulamento da Camada de Comunicação *Java Card* RMI.......... 67
4.3.1 Cenário de uso 68
4.3.2 Estrutura interna 69
4.3.3 Implementação dos pontos de extensão 73
4.4 Conclusão 75

5 ESTUDO DE CASOS 76
5.1 Aplicação Exemplo *Wallet* 76
5.2 Aplicação Exemplo *HelloWorldEcho* 81
5.3 Aplicação Exemplo *Purse* 85
5.4 Conclusão 95

6 CONCLUSÕES 96
6.1 Trabalhos Futuros 97

7 REFERÊNCIAS 100

1 INTRODUÇÃO

Desde sua invenção nos anos 50, o cartão de crédito mudou drasticamente o cenário dos meios de pagamentos na economia mundial, permitindo que consumidores comprem sem dinheiro em espécie em qualquer parte do globo com a respectiva moeda local. Além desse beneficio, os cartões de crédito trouxeram a segurança contra roubo para um novo patamar. Por outro lado, o volume de fraudes tem crescido nos últimos anos em função da evolução tecnológica.

Dessa maneira, os *Smart Cards* têm se apresentado como o substituto para o cartão de tarja magnética, melhorando significativamente o nível de segurança das transações com cartão de crédito. Eles dispõem de dimensões idênticas às do cartão de crédito típico e estão disponíveis em duas categorias de finalidade: os que apresentam apenas capacidade de armazenar dados e os que, além dessa capacidade, podem processar informações por meio de circuitos eletrônicos embutidos. Uma das principais aplicabilidades desse poder de processamento, a segurança eletrônica, é obtida por meio de uso de algoritmos de criptografia, tanto simétrica quanto assimétrica. Até mesmo o gerenciamento das chaves públicas e privadas utilizadas na criptografia é facilitado pela capacidade de armazenamento do cartão.

Outros benefícios proporcionados pelos *Smart Cards* podem ainda ser citados, tais como: a possibilidade de efetuar transações *off-line*, na qual o crédito do portador pode ser aprovado sem a necessidade de comunicação com a central da administradora do cartão, bem como a melhoria do processo de identificação e de autenticação dos usuários por meio de certificados digitais, assim como o aumento da confiabilidade para transações na internet pode ser aumentada substancialmente. Dentre as áreas que atualmente mais utilizam os *Smart Cards*, destacam-se a telefonia celular GSM - *Global System for Mobile Communications* (HEINE, 1999), o transporte público, o *voucher* eletrônico de alimentação, a identificação de pessoas e as transações financeiras.

Frost & Sullivan (2004) estimam que, no Brasil, o número de *Smart Cards* emitidos até 2005 é de quase 80 milhões. De acordo com a Anatel, aproximadamente 55 milhões deles referem-se aos SIMCards (*Subscriber Identity Module*, *Smart Card* utilizado na telefonia GSM), número que corresponde a um pouco mais da metade do total de telefones celulares no país. Na área de transporte público, a iniciativa mais significativa no Brasil pertence à prefeitura de

São Paulo com o desenvolvimento do Bilhete Único que garante o acesso do cidadão aos transportes públicos por meio de ônibus, metrô ou trem. A própria prefeitura municipal estima o número de 3 milhões de usuários desse serviço. No setor de *vouchers* de alimentação, as duas maiores empresas que atuam no país, a nacional VR e a francesa *Sodexho*, que contabilizam um total de 5,5 milhões de usuários. De acordo com as próprias empresas, a tecnologia de *Smart Card* encontra-se implantada em quase 100% dessa base. Para a área de identificação de pessoas, a Receita Federal dispõe de uma iniciativa ainda em fase embrionária, mas com perspectivas bastante interessantes, o e-CPF. Os números ainda não são tão significativos, pois são apenas 240 mil cidadãos que dispõem do cartão, o que corresponde apenas a 1% das declarações de imposto de renda no país. Por outro lado, a Receita já oferece alguns benefícios interessantes para esses usuários como, por exemplo, a prioridade na restituição do imposto. No mercado financeiro, o volume de cartões também é ainda discreto. A ABECS (Associação Brasileira das Empresas de Cartões de Crédito e Serviços) estima que já tenham sido emitidos cerca de 7 milhões de cartões de crédito ou débito com *chip*, o que correspondem apenas a 2% do número de cartões emitidos no país.

No Brasil, pode ser citada outra iniciativa que se destaca das apresentadas anteriormente, o Cartão Nacional de Saúde, que é utilizado na rede do SUS (Sistema Único de Saúde) para identificar os pacientes. O diferencial tecnológico desse sistema é que, além do seu projeto de migrar toda sua base de cartões magnéticos para cartões inteligentes, ele não se restringirá ao uso de cartões de memória como nas iniciativas indicadas anteriormente e sim utilizará o poder de processamento dos *Smart Cards* microprocessados. Mais do que isso, a plataforma de *software* escolhida para operações nesses cartões será Java, mais especificamente *Java Card*, o que corresponde ao cerne deste trabalho. Essa definição é estabelecida pelo governo federal por meio do Datasus, Departamento de Informática do SUS, conforme apontado nos padrões de interoperabilidade eletrônica, *e-PING* (e-ping, 2005).

Algumas das características de negócio que fomentam as diretrizes tecnológicas de uso da arquitetura *Java* e *Smart Cards* no sistema brasileiro de informação de saúde são apontadas por Leão (2004): todos os 180 milhões de cidadãos do país têm direito a tratamento médico integral em todo território nacional, 6.5 mil hospitais, 65 mil postos de saúde, 19 milhões de consultas médicas por mês, 100 milhões de procedimentos médicos por mês.

Por ser originalmente uma iniciativa da cidade de São Paulo a idéia de padronizar um sistema de informação unificado para todo país, os primeiros desenvolvimentos foram realizados nessa cidade que apresenta as seguintes características: 22 milhões de habitantes na região metropolitana, 390 postos de saúde, 160 policlínicas, 105 hospitais, 40 mil consultas por dia, 7 milhões de pacientes, 75 mil profissionais da saúde e 7 mil computadores.

Além do uso da linguagem Java e do emprego de *Java Card* como plataforma de desenvolvimento, a construção de aplicações *Smart Card* pode ser simplificada por meio do *framework* proposto neste trabalho. Dessa maneira, tem-se a principal motivação deste trabalho, conforme apresentado na seção a seguir.

1.1 Motivação

De acordo com Rankl e Effing (2000), os *Smart Cards* foram inventados e patenteados em 1970. Nessa época, existiam na Alemanha pesquisas conduzidas por Juergen Dethloff (RANKL e EFFING, 2000), bem como no Japão, conduzidas por Arimura (RANKL e EFFING, 2000), e na França, por Moreno (RANKL; EFFING, 2000). O primeiro uso em massa dessa tecnologia ocorre em 1983, na modalidade de cartão para pagamento de telefone público. O segundo ocorre com a integração de *microchips* em todos os cartões de débito na França. Sistemas de carteira eletrônica baseados em *smart cards* são experimentados na Europa nos anos 90, mas não atraem o interesse da opinião pública. A taxa de utilização é considerada inexpressiva. O aumento de uso vem com a introdução dos *chips* GSM para os telefones celulares na Europa, na mesma década.

Já o *Java Card* em sua versão 1.0, conforme Chen (2000), é proposto em 1997, inicialmente por engenheiros da Schlumberger e consiste apenas de uma especificação de API. Posteriormente, outras empresas, como Bull e Gemplus, formam com a Schlumberger o *Java Card* Fórum (JCF). Esse consórcio de empresas trabalha para a adoção da tecnologia *Java Card* na indústria dos *Smart Cards*. No ano seguinte, a Sun Microsystems trabalha com o JCF para o desenvolvimento do *Java Card* 2.0. Essa nova versão inclui a especificação da *Java Card Runtime Environment* (JCRE).

Em Hansmann et al. (2002), é apontada uma dificuldade central do desenvolvimento para essa plataforma, decorrente do fato de o desenvolvedor ter que trabalhar em um nível de abstração

baixo. Tal dificuldade se deve basicamente a dois fatores: as restrições rigorosas de *hardware* e a maturidade da plataforma, por se tratar ainda de um ambiente de desenvolvimento recente.

Java (GOSLING; JOY; STEELE, 1996), por sua vez, é considerada, conforme Horstmann e Cornell (2004), uma linguagem de programação que provê produtividade, segurança, robustez e principalmente portabilidade. Por isso ela pode ser considerada uma linguagem adequada para uma plataforma altamente restrita e heterogênea como os *Smart Cards*, conforme descrito por Rankl e Effing (2000). Os conceitos de orientação a objetos de *Java* permitem que o desenvolvedor trabalhe com elementos de abstração altos mesmo em uma plataforma de *hardware* restrita. A questão da heterogenia também é crítica na tecnologia de *Smart Cards*, pois existe um grande número de cartões de fabricantes diversos gerenciados por sistemas operacionais de outros fornecedores e que precisam se comunicar com uma infinidade de leitores de cartões fabricados por empresas variadas.

De acordo com Chen (2000), além dessas questões, outras características de Java são introduzidas no desenvolvimento de aplicações para *Smart Cards* ao se utilizar a plataforma *Java Card* como facilidade de desenvolvimento, pois a linguagem Java aumenta o nível de abstração para a programação em *smart cards*, simplificando a programação com linguagem *assembly* de microcontroladores. Além da própria linguagem, existe um grande número de ambientes integrados de desenvolvimento. Com essa plataforma, os programadores utilizam APIs de alto nível, evitando preocupações com detalhes físicos dos cartões com que estão trabalhando. Outra característica é a segurança, visto que as funcionalidades embutidas no Java são incorporadas nos *Smart Cards*, impossibilitando, por exemplo, forjar ponteiros a fim de permitir que programas maliciosos visualizem indevidamente a memória. Adicionalmente, aplicações na plataforma *Java Card* são separadas por um *firewall* interno, o que restringe o acesso aos dados de outra aplicação ou a outras partes do sistema. Uma das mais importantes características de Java incorporada aos *Smart Cards* é a portabilidade, pois o uso da máquina virtual permite sua utilização em diversos tipos de *hardware* e em sistemas operacionais. Essa variedade de plataformas é ainda maior ao se falar em *Smart Cards*, visto que existem muitos modelos de cartões e vários fabricantes suportados por diversos sistemas operacionais. E, por fim, o suporte a múltiplas aplicações, permite que, uma vez o cartão emitido, outras aplicações possam ser instaladas. A funcionalidade do cartão pode ser continuamente incrementada sem nenhum trabalho para o seu portador, que ainda tem o benefício de utilizar um cartão integrado, sem a necessidade de transportar vários cartões para diferentes

finalidades. Vale ainda lembrar que a tecnologia *Java Card* é baseada no padrão internacional de *Smart Cards* ISO 7816, o que permite que aplicações nessa plataforma sejam compatíveis com aplicações *Smart Card* implementadas em outras linguagens além da Java.

Apesar do desenvolvimento de aplicações para *Smart Cards* microprocessados ser simplificado significativamente com o uso da linguagem Java por meio de sua plataforma *Java Card*, esse tipo de aplicação pode ainda ser agilizado por um *framework*. Esse conceito pode ser definido como "um conjunto de classes que encapsula o desenho de solução para uma família de problemas relacionados" (JOHNSON e FOOTE, 1988). Mais do que sua definição, esse conceito pode aumentar a produtividade, qualidade e flexibilidade no desenvolvimento de aplicações por meio de algumas de suas características, apresentadas por Fayad e Schmidt (1997): modularidade, com o isolamento do impacto das alterações, tanto no projeto como na implementação, e com a redução do esforço de manutenção; reusabilidade, com o reaproveitamento do conhecimento de análise, do projeto e do código do *framework* no desenvolvimento de várias aplicações; extensibilidade, com a utilização dos pontos de extensão definidos pelo *framework*, as aplicações podem ter suas funcionalidades incrementadas; inversão de controle, a qual possibilita que parte da lógica de controle seja centralizada no *framework*, enquanto a aplicação implementa apenas métodos pré-definidos que são invocados, quando necessário, pelo *framework*.

Conforme a própria de definição de *framework*, a proposta deste trabalho procura solucionar as questões mais comuns da família de problemas relacionados à construção de aplicações *Java Card*, como: estruturar a troca de comandos APDU (*Application Protocol Data Unit*) entre a aplicação Java no *Smart Card* e a leitora em que ele se encontra inserido; isolar a camada de persistência da camada de negócio da aplicação; e encapsular a camada de comunicação, em particular o protocolo *Java Card* RMI, disponível nos *Smart Cards* suportados pelas ultima versões de *Java Card*. O *framework* proposto, JCFoundation, procura solucionar questões comuns no desenvolvimento de aplicações *Java Card*, permitindo que o desenvolvedor empregue seus esforços na construção de conceitos de negócio específicos da aplicação com a qual ele trabalha.

Outro aspecto importante deste *framework* é que ele é baseado em padrões de projeto amplamente usados e validados. Essa idéia de compor *frameworks* com base em padrões de projeto é apresentada originalmente por Johnson (1992) e aplicada por diversos outros autores

como Srinivasan (1999). Os padrões empregados na construção do *framework* proposto são construídos com a especialização, para o contexto de *Java Card,* dos padrões do catálogo de Gamma (1994): *Command*, *Bridge* e *Adapter*. O detalhamento das funcionalidades fornecidas pelo *framework* é apresentado no capítulo 4.

1.2 Objetivos Gerais e Específicos

O objetivo específico deste trabalho consiste em propor um *framework* que encapsule a solução para problemas comuns encontrados no desenvolvimento da maioria das aplicações *Java Card*. Dessa maneira, o *framework* JCFoundation está organizado em módulos específicos para estruturar a troca de comandos entre a aplicação Java no *Smart Card* e o receptor, isolar a camada de persistência da camada de negócio da aplicação e encapsular a camada de comunicação via *Java Card* RMI.

Por meio deste objetivo específico, é que o objetivo mais amplo é alcançado, o qual consiste em fornecer uma estrutura base para o desenvolvimento de aplicações *Java Card*. Dessa forma, o desenvolvedor pode focar seus esforços na camada de aplicação responsável pelos conceitos de negócio empregados na aplicação a ser construída, enquanto algumas das funcionalidades básicas de infra-estrutura são fornecidas de antemão pelo JCFoundation. Conforme apresentado no capítulo Estado da Arte, o desenho da camada de negócio pode ser realizado por meio de técnicas como métodos formais, o que depende da criticidade da aplicação a ser desenvolvida.

1.3 Resultados Esperados e Contribuições

O resultado esperado é que o JCFoundation forneça ao desenvolvedor uma estrutura básica com algumas funcionalidades de suporte, como tratamento de comandos, persistência e comunicação remota, o que o auxilia no desenvolvimento de aplicações *Java Card* para *Smart Cards* microprocessados.

Como contribuição, este trabalho apresenta um novo *framework* para uma plataforma ainda pouco explorada, cobrindo uma família de problemas de desenho de *software* ainda não atendida pelas iniciativas atuais dessa área de pesquisa. Apesar de existir atualmente um

grande número de publicações sobre *frameworks*, não se tem notícia de uma solução específica que atenda à plataforma *Java Card*. Este trabalho está ainda consoante com uma das linhas de pesquisas mais recentes desta área, apresentadas por Grimaud e Vandewalle (2003), a qual procura estruturar a construção de aplicações para *Smart Cards*. Outra linha apresentada por esses autores que dispõem de um número significativo de publicações é a de métodos formais. Essa técnica pode, então, ser empregada na construção das partes críticas das aplicações em questão, enquanto o JCFoundation procura atender às questões comuns ao desenvolvimento de aplicações *Java Card*. Os métodos formais e outras linhas de pesquisa sobre *Java Card* são apresentados na seção de Estado da Arte.

1.4 Método de Pesquisa

O método da pesquisa foi composto pelas atividades descritas a seguir. Inicialmente foi realizada uma pesquisa bibliográfica dos conceitos principais envolvidos: *Java Card*, *Smart Cards*, *Frameworks,* a fim de se determinar o estado da arte dessas áreas de pesquisa. Em seguida, há um período de estudo das ferramentas de desenvolvimento disponíveis para *Java Card*, JCOP e do próprio *toolkit* fornecido pela Sun Microsystems. Como principal atividade, propõe-se o *framework* JCFoundation como ponto de partida para o desenvolvimento de aplicações nessa plataforma. Por fim, é realizado um estudo de caso empregando o JCFoundation a aplicações de exemplo, fornecidas pelo próprio fabricante da tecnologia, o que possibilita validar a proposta.

A principal atividade deste trabalho, a proposição do JCFoundation, segue de acordo com o método proposto por Fontoura (2001) para projeto de *framework*. Segundo esse autor, o desenvolvimento de um *framework* é composto por quatro atividades principais conforme a Figura 1 a seguir.

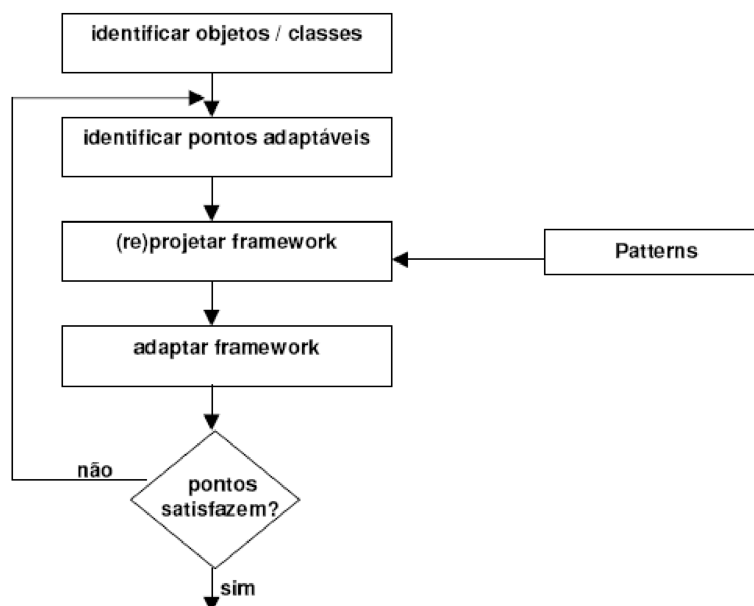

Figura 1 - Método de Desenvolvimento de *Frameworks* (FONTOURA, 2001).

A primeira atividade, identificar objetos/classes, consiste na identificação da estrutura de classes de conceitos comuns no domínio de aplicação no qual o *framework* é aplicado. Na segunda, identificar pontos adaptáveis, deve-se definir os aspectos que podem ser estendidos para cada aplicação nas quais o *framework* é aplicado. Quanto mais aspectos são identificados, mais flexível é o *framework*. A terceira atividade, (re) projetar o *framework*, consiste na codificação propriamente dita do *framework*. Essa etapa pode ser suportada pelos *design patterns*, padrões de projeto, que podem contribuir para a confecção do *framework* ao incorporar soluções para problemas de projeto já validadas, além de servir como base para documentação do próprio *framework*. Conforme mencionado anteriormente, a idéia de aplicar padrão de projetos da confecção de *framework* é utilizada por diversos autores, tais como Johnson (1992) e Srinivasan (1999). A última atividade consiste em um refinamento estrutural e na posterior avaliação dos pontos adaptáveis.

1.5 Organização do Trabalho

Além deste capítulo introdutório, os demais capítulos deste trabalho estão organizados da seguinte forma:

No Capítulo 2, Fundamentos, são apresentados os conceitos centrais necessários para o entendimento do trabalho como um todo: *Smart Cards* e sua plataforma de desenvolvimento *Java Card*.

No Capítulo 3, Estado da Arte, é apresentado o cenário acadêmico atual das publicações sobre *Java Card*.

No Capítulo 4, *Framework* JCFoundation, são apresentados os elementos do *framework* e há a resolução dos seguintes problemas: estruturar a troca de comandos entre a aplicação Java no *Smart Card* e o receptor no qual o cartão encontra-se inserido; isolar a camada de persistência da camada de negócio da aplicação; e encapsular a camada de comunicação, em particular o protocolo *Java Card* RMI.

No Capítulo 5, Estudo de Caso, é validado o *framework* proposto por meio do emprego deste às aplicações de exemplo fornecidas pelo próprio fabricante da tecnologia.

No Capítulo 6, Conclusões, são analisados os resultados obtidos durante o trabalho de pesquisa e esboçam-se propostas de trabalhos futuros.

2 FUNDAMENTOS

2.1 *Smart Cards*

De acordo com Rankl e Effing (2000), “um *smart card* é um cartão que tem embutido um microprocessador e um *chip* de memória ou apenas um *chip* de memória com uma lógica não programável”. Um cartão inteligente (*Smart Card*) conta com diversas funcionalidades e com utilizações proporcionadas pelos circuitos eletrônicos integrados que o compõem.

O *Smart Card* apresenta internamente diversos circuitos eletrônicos, dispostos em um *chip*, capazes de armazenar e de processar dados de maneira segura, ou seja, as informações armazenadas no cartão são protegidas contra acessos não autorizados. Podem ser utilizadas diversas técnicas para proteção desses dados, como a criptografia, entre outros mecanismos que garantem a segurança sem restringir funcionalidades e flexibilidade, isso tudo gerenciado por um sistema, operacional ou não.

Eles são geralmente utilizados em, mas não restrita a, aplicações em que se necessita de segurança. Pode-se encontrar *Smart Cards* sendo utilizados como cartão de crédito, dinheiro eletrônico, utilização em soluções de identificação, armazenamento de certificados digitais entre tantos outros usos, triviais ou não.

A norma ISO 7816 especifica vários aspectos dos *smart card*, entre eles a estrutura física conforme ilustrado na Figura 2 a seguir.

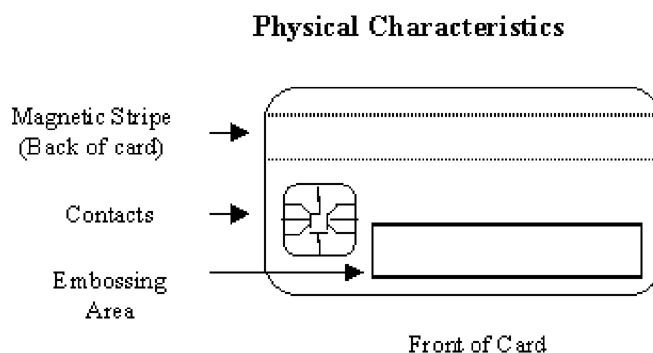

Figura 2 - Características Físicas de um *Smart Card*

A norma ISO 7816 especifica também questões relativas ao contato elétrico entre o cartão e o dispositivo no qual ele é inserido. A norma determina que devem existir oito pontos de contato: energia, *reset*, *check*, terra (*ground*), entrada/saída e o contato opcional, conforme ilustrado a seguir, na Figura 3:

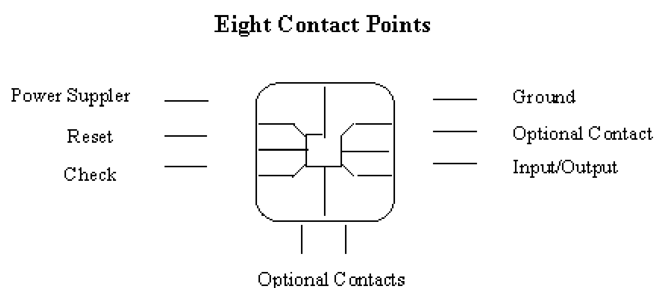

Figura 3 - Os Oito Pontos de Contato de um *Smart Card*

2.1.1 Benefícios dos *Smart Cards*

Ainda conforme Rankl e Effing (2000), o principal benefício do *smart card* é seu poder computacional embutido. Além desse benefício, podem-se ainda mencionar outros como: segurança, portabilidade e facilidade de uso, descritos a seguir:

- Segurança

O processador, a memória e a capacidade de entrada / saída de um *smart card* estão empacotados em um único circuito integrado, embutido em um cartão plástico. O *smart card* está protegido contra ataques, pelo fato que seus dados não são acessados diretamente por um dispositivo potencialmente não confiável. Para acessar os dados do *smart card,* esse dispositivo solicita-os a uma aplicação no cartão, que fornece o controle de segurança dos dados.

Outra forma de ataque para acessar os dados do *smart card* é dispor deste em mãos e utilizar sobre este equipamentos para extração de dados centenas de vezes mais caro do que similares utilizados em cartões magnéticos. Uma vez lidos os dados do cartão, não é possível utilizá-los diretamente, pois esses estão criptografados.

Interceptar a comunicação entre o dispositivo de leitura e o cartão pode ser outra opção de ataque ao cartão. No entanto, esse tipo de ataque pode ser bloqueado com o uso dos mecanismos embutidos no cartão, como criptografia e assinatura digital, sendo que, na assinatura, é utilizada a própria senha do portador com chave privada.

- Portabilidade

A portabilidade é um beneficio intrínseco do *smart card*. É possível carregá-lo na carteira da mesma forma que um cartão de crédito comum.

- Facilidade de uso

Pelo fato de o *smart card* suportar múltiplas aplicações, é possível que um único cartão substitua diversos cartões magnéticos ou até mesmo documentos. Por exemplo, é possível em um único *smart card* armazenar informações de cartão de crédito, cartão bancário, identidade e cartão fidelidade.

2.1.2 Visão geral sobre *Smart Cards*

O circuito integrado incorporado ao substrato de plástico do tamanho de um cartão de crédito contém elementos usados para transmissão, armazenamento e processamento de dados. Um *smart card* também pode ter uma área de *embossing* em uma das faces e uma faixa magnética na outra. A aparência física e propriedades dele estão definidas pelo padrão ISO 7816.

Normalmente, um *smart card* não contém uma provisão de energia elétrica, uma tela ou um teclado. Para se comunicar com o mundo externo, o cartão é inserido ou aproximado de um dispositivo de aceitação de cartão (CAD - *Card Acceptance Device*) que é conectado a um computador.

2.1.3 Cartões de memória ou cartões microprocessados

Os cartões podem ser classificados de acordo com sua capacidade de processamento, conforme descrito a seguir:

- **Cartões de Memória:**

Os primeiros *smart cards* eram cartões de memória. Cartões de memória não contêm um microprocessador. Eles são embutidos com um *chip* de memória ou com um *chip* de lógica não programável e de memória.

Tipicamente, cartões de memória suportam de 1K a 4K *bytes* de dados. Eles são principalmente usados como cartões pré-pagos para telefones públicos ou para outros serviços que são vendidos sob pagamento prévio.

Visto que um cartão de memória não possui um microprocessador, seu processamento de dados é realizado por um circuito simples, capaz de executar algumas instruções pré-programadas. Tal circuito tem funções limitadas e não pode ser reprogramado. Portanto eles não podem ser reutilizados e, quando o valor no cartão já estiver esgotado, ele deve ser descartado.

Dependendo das exigências de segurança dos dados armazenados, o acesso aos dados pode ser protegido por meio de uma memória protegida, ou por uma lógica de segurança. Por exemplo, cartões telefônicos pré-pagos podem conter lógica para impedir que o número de créditos seja aumentado.

A vantagem dos cartões de memória é a tecnologia simples, por isso eles são produzidos a um baixo custo. A desvantagem é que eles podem ser falsificados de maneira relativamente fácil.

- **Cartões Microprocessados:**

Como indica o nome, os cartões microprocessados contêm um processador. Eles oferecem uma maior segurança e a capacidade de executar múltiplas funções. Com um cartão microprocessado, os dados nunca estão diretamente disponíveis para as aplicações externas. O microprocessador controla o manuseio dos dados e o acesso à memória, de acordo com um determinado conjunto de condições (senhas, criptografia e assim por diante) e com instruções das aplicações externas. Muitos modelos atuais trazem embutido o suporte à criptografia. Tais cartões são particularmente úteis para aplicações que precisam solucionar problemas de segurança de dados. Cartões microprocessados são muito flexíveis, pois sua funcionalidade só está restrita pelos recursos de memória e pelo poder computacional disponível. Cartões microprocessados são extensamente usados para controle de acesso, aplicações bancárias,

programas de fidelidade, telefonia celular e assim por diante, em situações em que segurança de dados e privacidade são as principais preocupações.

Como resultado da produção em massa, o custo dos cartões caiu drasticamente desde o início dos anos 90. Atualmente, eles custam entre US$1 e US$20, dependendo dos recursos de memória e do processamento inclusos.

2.1.4 Sistema de memória dos *Smart Cards*

Nos *smart cards* podem ser aplicados três tipos de memória:

- **ROM (*Read Only Memory*)**

Memória gravada pelo próprio fabricante para armazenar os programas permanentes, como, por exemplo, a própria máquina virtual de Java.

- **EEPROM (*Electric Erasable Programmable Read Only Memory*)**

Essa memória pode ser vista de forma análoga à de um disco rígido de computador. Nela são armazenadas as aplicações do usuário, bem como as informações de longa vida manipuladas por essas aplicações.

- ***RAM (Random Access Memory*)**

Memória usada como espaço de armazenamento temporário. Nessa memória são armazenados tipicamente dados perenes como, por exemplo, variáveis locais utilizadas como resultado intermediário de um processamento.

2.1.5 Modelo de Comunicação dos *Smart Cards*

A comunicação entre o cartão e o dispositivo no qual ele é inserido é do tipo *half-duplex*, ou seja, os dados podem ser enviados do CAD ao cartão e vice-versa, mas não para ambos simultaneamente.

O pacote do protocolo de comunicação entre o CAD e o cartão é chamado APDU (*Application Protocol Data Unit*). Esse pacote contém um requisição, ou uma mensagem de resposta.

Nesse modelo, é empregada a técnica cliente/servidor, no qual o cartão desempenha o papel do servidor, e o CAD, o papel do cliente. O CAD envia comandos APDUs para o cartão, que, por sua vez, executa a instrução especificada no comando, devolvendo uma resposta APDU.

O protocolo APDU, como especificado pela norma ISO 7816-4, é um protocolo de nível de aplicação, que define mensagens APDU com duas estruturas: uma, usada pelo dispositivo para enviar comandos ao cartão, chamada Comando APDU, e outra, usada pelo cartão para mandar de volta as respostas ao dispositivo, chamada Resposta APDU. Essas estruturas estão ilustradas nas tabelas a seguir:

Tabela 1 - Requisição APDU

Requisição APDU						
Cabeçalho Obrigatório				**Corpo Opcional**		
CLA	INS	P1	P2	LC	Data field	LE

Tabela 2 - Resposta APDU

Resposta APDU		
Corpo Opcional	**Corpo Obrigatório**	
Data field	SW1	SW2

O cabeçalho do comando APDU consiste de 4 *bytes*: CLA (classe de instrução), INS (código de instrução) e PI e P2 (parâmetros 1 e 2). O *byte* de classe identifica uma categoria de comando / resposta. O *byte* de instrução especifica a instrução do comando. Os dois *bytes* de parâmetro PI e P2 são usados para prover qualificações adicionais à instrução.

A seção depois do cabeçalho é um corpo opcional que varia em comprimento. O campo de Lc especifica o comprimento do campo de dados (em *bytes*). O campo de dados contém dados que são enviados ao cartão. O último *byte* no comando do corpo de APDU é o campo de Le que especifica o número de *bytes* esperado pela aplicação cliente na resposta do cartão.

A resposta APDU consiste de um corpo opcional e em um finalizador obrigatório. O corpo consiste no campo de dados cujo comprimento é determinado pelo campo Le do comando APDU. O finalizador consiste em dois campos SW e SW2 que determinam o estado do

processamento após a execução do comando APDU.

A informação contida em cada campo está resumida na seguinte tabela:

Tabela 3 - Campos do Comando APDU

Campo	Tamanho	Descrição
REQUISIÇÃO		
CLA	1 *byte*	Classe: indica a estrutura e o formato de um comando APDU.
INS	1 *byte*	Código da instrução: especifica a instrução do comando.
P1	1 *byte*	Parâmetros da instrução: fornecem mais informação sobre a instrução.
P2	1 *byte*	
LC	1 *byte*	Número de *byte*s no *Data Field* do APDU.
Data Field	LC *bytes*	Seqüência de *bytes* com a informação.
LE	1 *byte*	Quantidade máxima de *bytes* esperados como resposta.
RESPOSTA		
Data Field	Até LE *byte*s	Seqüência de *bytes* com informação.
SW1	1 *byte*	*Status Word* (palavra do estado). Denota o estado do processamento do comando no cartão.
SW2	1 *byte*	

2.1.6 Padrões e especificações

Os padrões e as especificações internacionais envolvidas com a plataforma *Smart Card* são os seguintes:

- **ISO 7816**

Publicado pela ISO, é o padrão mais importante que define as características de cartões com *chip*. Esse padrão (ISO, 1987) cobre vários aspectos de *smart cards*: características físicas, dimensões e local dos contatos, sinais eletrônicos, protocolos de transmissão, comandos para intercâmbio entre indústrias e identificadores de aplicação.

- **GSM**

O Instituto de Padrões de Telecomunicações Europeu, ETSI (*European Telecommunications Standards Institute*) publicou um conjunto de padrões que cobrem cartões inteligentes para

uso em telefonia celular. O Sistema GSM (HEINE, 1999), definido pela ETSI, é uma especificação para um sistema de telefonia móvel terrestre internacional. Originalmente pretendido para cobrir alguns países na Europa Central, está se tornando um padrão internacional para telefones móveis.

- **Open Platform:**

Open Platform (GLOBAL PLATFORM ASSOCIATION, 2001) define um ambiente integrado para o desenvolvimento e operação de sistemas com suporte a aplicações múltiplas. Essa plataforma consiste em uma especificação de cartão e uma especificação de terminal.

Essas especificações são desenvolvidas inicialmente pela Visa e, posteriormente, transferidas para a *GlobalPlatform*, uma organização criada para promover uma infra-estrutura global para implementação de *smart cards* para várias indústrias.

- **OpenCard Framework:**

O *OpenCard Framework* (OPEN CARD CONSORTIUM, 2000) foi produzido inicialmente pela IBM e atualmente é gerido por um consórcio que inclui as principais empresas da indústria de *smarts*. Esse *framework* provê uma interface padrão para interação entre a aplicação cliente e o cartão. Essa arquitetura é um modelo estruturado que divide funções entre fabricantes de cartões, fabricantes de terminais, provedores de sistema operacional e emissores de cartão. A meta é reduzir a dependência entre cada uma dessas partes.

2.2 *Java Card*

2.2.1 Arquitetura

Smart Cards representam atualmente uma das menores plataformas computacionais. O maior desafio da tecnologia *Java Card* é incorporar o ambiente de execução Java nessa plataforma restrita, conservando espaço suficiente para as aplicações de usuário. A solução é suportar apenas um subconjunto da linguagem Java e utilizar um modelo particionado da máquina virtual: uma parte que é executada no cartão, e outra, fora do cartão. Tarefas como carregamento de classes, verificação de *bytecode* e otimização, são responsabilidades da porção da máquina virtual localizada fora do cartão.

Outra preocupação importante dessa tecnologia é atender a padrões, como o ISO 7816 que define uma série de características dos *smart cards*. Outro padrão é o *OpenCard Framework* que é apresentando em Fodor e Hassler (1999) juntamente com a estratégia de *Java Card,* para atender a esse padrão. Esse *framework* provê uma interface padrão para interação entre a aplicação cliente e o cartão.

A característica mais significativa do ambiente de execução *Java Card* é a separação clara entre o sistema operacional do cartão e a plataforma de execução das aplicações.

Além da máquina virtual (*Java Card Virtual Machine*, JCVM), encontra-se especificada a interface de programação (*Application Programming Interface*, API) e o ambiente de execução (*Java Card Runtime Environment*, JCRE). A JCVM define o subconjunto da linguagem Java válido para *Java Cards*. A API *Java Card* compreende um conjunto de classes Java para o desenvolvimento de aplicações. Por fim, a JCRE detalha o comportamento da plataforma em tempo de execução, englobando aspectos como gerenciamento de memória e aplicações, conforme a Figura 4 a seguir.

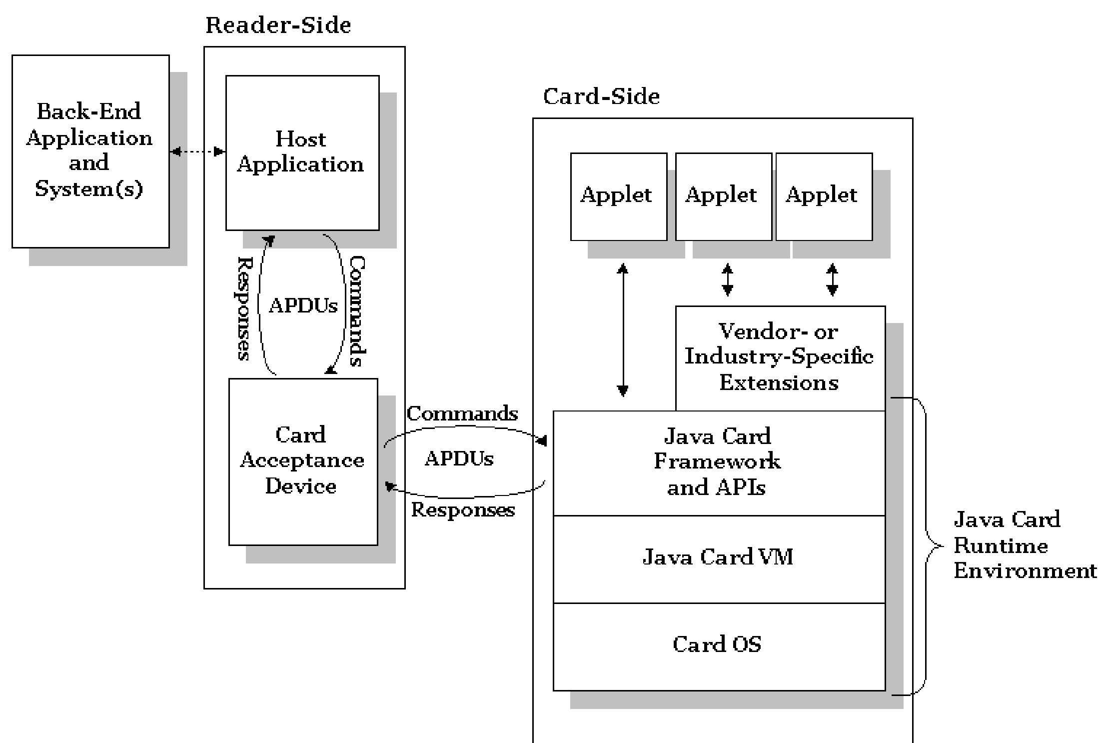

Figura 4 - Arquitetura *Java Card*

2.2.2 Subconjunto suportado da linguagem Java

Em função de sua memória restrita, *Java Card* suporta apenas um subconjunto da linguagem Java. A Tabela 4, a seguir, apresenta, em uma coluna, as funcionalidades que são suportadas por Java *Card,* e em outra coluna, as funcionalidades de Java que *Java Card* ainda não suporta.

Tabela 4 - Funcionalidades de Java Suportadas por *Java Card*

Funcionalidades Suportadas	Funcionalidades Não Suportadas
• Tipos de dados primitivos: *boolean*, *byte* e *short*. • *Arrays* unidimensionais. • Pacotes, classes, interfaces e exceções. • Herança, métodos e classes abstratas, sobrecarga, criação dinâmica de objetos, escopo de acesso e de ligação dinâmica. • Suporte ao inteiro de 32 *bits* opcional.	• Tipos de dados primitivos: *long*, *float* e *double*. • Caracteres e *strings*. • *Arrays* Multidimensionais. • Carregamento dinâmico de classes. • Gerenciamento de segurança. • *Garbage collection* e *finalization*. • *Threads.* • Serialização. • Clones.

2.2.3 Máquina virtual de *Java Card*

A principal diferença entre a JCVM (JCVM, 2002) e a JVM é a divisão em duas partes, conforme ilustrado na Figura 5 a seguir. O principal elemento da porção da máquina virtual localizada no cartão é o interpretador. E o principal elemento da porção da máquina virtual localizada fora do cartão é o conversor. O conversor recebe como entrada os arquivos .class e faz o carregamento de classes, a verificação de *bytecode* e a otimização, gerando o .cap (*converted applet*).

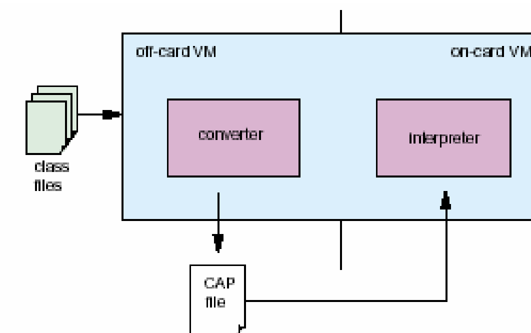

Figura 5 - Estrutura da Máquina Virtual de *Java Card* (JCVM)

2.2.4 Instalador *Java Card*

O interpretador não faz o autocarregamento de arquivos .cap. Os mecanismos para carregar e instalar um arquivo .cap são executados por uma unidade chamada instalador.

O instalador também está dividido em duas partes, uma no cartão e outra fora dele. O instalador localizado fora do cartão comunica-se com o instalador localizado dentro do cartão por meio do receptor do cartão (*Card Acceptance Device*, CAD) para enviar o arquivo .cap.

O instalador escreve o binário na memória de cartão, liga-o com as outras classes que já foram colocadas no cartão, cria e inicializa qualquer estrutura de dados usados pelo JCRE. Esse processo está ilustrado na Figura 6 a seguir.

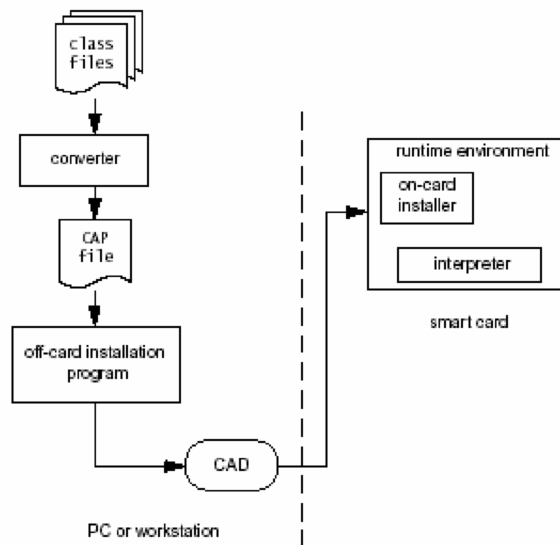

Figura 6 - Estrutura do Instalador *Java Car*

2.2.5 Ambiente de execução *Java Card* (JCRE)

O JCRE (JCRE, 2002) consiste em um conjunto de componentes que são executados dentro de um *smart card*. O JCRE é responsável pela administração dos recursos do cartão, de comunicações de rede e da execução e segurança de aplicativos.

O JCRE é iniciado em conjunto com a própria inicialização do cartão. Esse processo só é executado uma vez durante o ciclo de vida do cartão. Durante esse processo, o JCRE inicializa a máquina virtual e cria objetos que provêm gerência e serviços às aplicações.

Ao se inserir um cartão no CAD, dá-se início a uma sessão. A partir desse momento, o JCRE está pronto para atender a comandos de APDU. A JCRE encapsula esses comandos em objetos Java e os envia à aplicação selecionada.

Além das funções-padrão de ambiente de execução Java apresentadas, a JCRE suporta as seguintes funcionalidades específicas:

- **Objetos persistentes e objetos transientes:**

Conforme apresentado em Oestreicher e Ksheeradbhi (1999), todos os objetos são criados na memória persistente. Esses objetos permanecem acessíveis por várias sessões. Objetos criados

como variáveis locais são instanciados na memória transiente, sendo descartados ao final da sessão. O modelo de persistência de *Java Card* é discutido com um pouco mais de detalhes ainda nesse capítulo.

- **Operações atômicas e transações:**

A máquina virtual assegura que cada operação escrita em um único atributo de um objeto ou de uma classe seja atômica. Além disso, o JCRE provê APIs para controle de transações, permitindo que o desenvolvedor defina um contexto em que todas as escritas são concretizadas ou todas são desfeitas. A implementação detalhada desses mecanismos pode ser vista em Oestreicher (1999).

- ***Firewall* e mecanismos de compartilhamento:**

O *firewall* isola os *applets*. Cada aplicativo é executado dentro de um espaço designado, o que impossibilita a interferência de um aplicativo em outros aplicativos instalados no cartão. Caso seja necessária a interação entre aplicativos de um mesmo cartão, o *Java Card* especifica uma interface segura e bem definida para isso, como apresentada em Montgomery e Krishna (1999).

2.2.6 API de *Java Card*

As APIs de *Java Card* (JCAPI, 2002) definem uma série de classes para o desenvolvimento de aplicações *Smart Card,* de acordo com padrão ISO 7816. Ela compreende três pacotes básicos *java.lang, javacard.framework* e *javacard.security*. Além de um pacote extra *javacardx.crypto*.

- **Pacote java.lang:**

Esse pacote é um subconjunto reduzido do pacote de mesmo nome da plataforma Java. As classes suportadas são *Object*, *Throwable* e algumas exceções da linguagem. Alguns dos métodos definidos em Java não estão disponíveis para *Java Card*.

- **Pacote *javacard.framework:***

Esse pacote provê as classes e interfaces para as funcionalidades principais de uma aplicação *Java Card*, provendo um modelo para criação de aplicativos, cujo ciclo de vida é controlado

pelo JCRE. As classes fornecidas por esse pacote representam conceitos do padrão ISO 7816, por exemplo, a classe APDU. Essa classe encapsula requisições oriundas e respostas enviadas à aplicação cliente. Outra classe importante desse pacote é a *JCSystem*. Por meio dela, a aplicação pode acessar informações sobre o ambiente de execução. Dessa forma, é possível gerenciar transações e interagir com outras aplicações.

Outra importante interface desse pacote é a *Shareable*, que deve ser estendida a fim de declarar os métodos a serem expostos publicamente aos outros *applets*.

- **Pacote javacard.security:**

Esse pacote provê uma série de funcionalidades de criptografia, incluindo algoritmos de chave simétrica e assimétrica, assinatura digital, geradores de números randômicos e mensagens *digest*. Essas funcionalidades são de fundamental importância para a plataforma *Java Card*, pois as aplicações *Java Card* normalmente trabalham com informações confidenciais.

- **Pacote *javacardx.crypt:***

Esse pacote é uma extensão do pacote de segurança. Ele provê funcionalidade de criptografia forte. As classes e as interfaces contidas nesse pacote estão sujeitas a controle pela legislação de importação norte-americana.

- ***Applet Java Card:***

As aplicações no ambiente *Java Card* são denominadas *applets*. A especificação de *Java Card* define um conjunto de convenções, às quais os *applets* devem aderir. Essas convenções permitem que o JCRE controle o ciclo de vida das aplicações instaladas no cartão. Para aderir a essas convenções, os *applets* do usuário devem estender a classe *javacard.framework.Applet.*

- **Identificação da *Applets* e Pacotes:**

Na plataforma *Java Card*, cada *applet* é exclusivamente identificado por um AID (*Application Identifier*). Cada pacote Java é também identificado por um AID. A norma ISO 7816 especifica os AIDs de forma que eles sejam únicos globalmente. O AID é composto por duas partes. A primeira parte é um identificador de recursos chamado RID (*Resource Identifier*). A segunda parte é um identificador pessoal chamado PIX (*Personal Identifier*

Extension). O RID é único por empresa e é especificado pela ISO. O PIX é único por *applet* ou pacote e é especificado pela empresa.

- **Classe *javacard.framework.Applet:***

Toda aplicação *Java Card* precisa estender a classe *javacard.framework.Applet*. Essa classe define uma série de métodos que são invocados pelo JCRE durante o gerenciamento do ciclo de vida da aplicação. Esses métodos, que devem ser redefinidos pelos *applets* do usuário com o comportamento adequado da aplicação em questão, estão detalhados na Tabela 5 a seguir:

Tabela 5 - Métodos da Classe *Applet*

deselect ()	Chamado pelo JCRE para informar ao *applet* atualmente ativado que outro *applet* será selecionado.
getShareableInterfaceObject ()	Chamado pelo JCRE para obter um objeto que implementa uma interface de compartilhamento de um *applet* servidor. A JCRE faz essa solicitação em nome de um *applet* cliente.
install ()	O JCRE chama esse método para criar uma instância de *Applet*.
process (APDU apdu)	Chamado pelo JCRE para processar um comando APDU.
register ()	Esse método é usado pelo *applet* para registrar sua instância junto ao JCRE e para definir o AID.
select ()	Chamado pelo JCRE para informar que o *applet* está selecionado.
selectingApplet ()	Esse método é usado pelo *applet* para distinguir o comando APDU de seleção.

Uma vez que o código do *applet* esteja corretamente carregado em um *Java Card*, o seu ciclo de vida se inicia. O primeiro método a ser chamado pela JCRE é o *install()*. Durante a execução desse método, o método *registry()* deve ser chamado, registrando o *applet* junto ao JCRE. O *applet* permanece inativo até o momento em que ele é selecionado. No momento em que isso acontece, o JCRE invoca o método *select()*. Esse método deve fazer as inicializações necessárias para que o *applet* fique pronto para atender a comandos APDU. Caso algum outro *applet* já esteja selecionado, a seleção é desfeita por meio do método *deselect()*. Esse método deve liberar os recursos alocados pelo *applet*. Ao receber um comando APDU, o JCRE envia-o para ser processado pelo método *process()*. Caso o *applet* em questão interaja com outros

applets no papel de servidor, ele deve retornar uma implementação de sua interface

compartilhada, por meio do método *getShareableInterfaceObject()*.

- **O Método *process()* e a Classe APDU:**

A classe javacard.framework.APDU encapsula um comando APDU. Uma instância dessa classe é passada ao *applet* no método *process()*. Esse método e essa classe são os principais objetos de estudo deste documento, pois o *framework* aqui definido simplifica a codificação do método e cria um tratamento uniforme para os comandos APDU recebidos.

O tratamento desses comandos é normalmente composto dos seguintes passos:

1. **Validar o cabeçalho APDU:**

Ao receber o comando APDU, o *applet* é responsável por verificar se esse comando é suportado por ele e se o comando está formatado corretamente, e por executar as validações necessárias de segurança;

2. **Ler os dados da requisição APDU:**

Caso exista algum dado de entrada do comando APDU, ele deve ser lido por meio de uma chamada inicial ao método *setIncomingAndReceive()* da classe APDU e por chamadas seqüenciais ao método *receiveBytes()* dessa mesma classe;

3. **Escrever o *status* e os dados da resposta APDU:**

Por fim, o *applet* deve retornar o *status* do processamento do comando e os dados de resposta, caso eles existam. Se o *applet* executar o método *process()*, e o executar completamente, o *status* de processamento normal é automaticamente retornado. Caso algum erro ocorra durante o processamento, o *status* de erro deve ser retornado dentro de uma *ISOException*. Para enviar algum dado de resposta, o *applet* deve inicialmente utilizar o método *setOutgoingLength()* da classe APDU e fazer chamadas seqüenciais ao método *sendBytes()* da mesma classe.

2.2.7 Modelo de persistência

Diferentemente do Java tradicional, no qual todos os objetos são transientes, salvo estejam explicitamente armazenados em alguma memória auxiliar como sistema de arquivos por meio de serialização, o *Java Card* apresenta um modelo de persistência que oferece tanto objetos transientes quanto persistentes. As aplicações instaladas no *smart card,* após a sua emissão, são armazenadas em meio persistente, mais especificamente em memória apagável programaticamente.

Por definição, o mesmo acontece com objetos criados por essas aplicações por meio do construtor padrão, *new*. No entanto, os objetos que são criados pelo construtor padrão dentro do corpo de um método e que são referenciados apenas por uma variável local desse método, podem ficar inacessíveis. Isso se deve ao fato que os objetos criados com o construtor padrão são armazenados na memória não-volátil, enquanto a variável local é armazenada na área de memória volátil. Dessa forma, ao fim da execução do método, a variável local é removida da memória. Por outro lado, o objeto criado permanece na memória persistente sem ser referenciado, ficando inacessível e consumindo espaço até que o *garbage collector* o remova.

Já os objetos transientes precisam ser explicitamente criados como tal. Para tanto, deve ser usado um dos métodos da classe *JCSystem* destinado à criação de *arrays* transientes de objetos, ou à criação de um dos três tipos primitivos *boolean, byte* e *short*. A sintaxe para a essa criação é a seguinte: *JCSystem.makeTransientXXXArray*, em que XXX pode ser *Object*, *Boolean*, *Byte* ou *Short*. A transiência desses objetos não se deve ao fato de eles serem completamente descartados, uma vez que o cartão seja desenergizado e sim ao fato que seus atributos são reiniciados para seu valor padrão (*null*, *false*, zero) uma vez finalizada a sessão do cartão. Essas informações ficam na memória volátil do cartão, enquanto a referência para o descritor da classe fica na memória não-volátil. Dessa forma, a referência de objeto persistente para um objeto transiente é mantida válida entre as sessões do cartão, e apenas atributos do objeto transiente não dispõem dos seus valores originais e sim dos seus valores padrão. Além de limitado ao tempo da sessão do cartão, o tempo de vida pode ser limitado ao tempo para o qual a aplicação está selecionada. Para definir qual tempo de vida deve ser usado, deve-se passar, como parâmetro, para método *makeTransienteXXX()* um dos seguintes

valores: *CLEAR_ON_RESET* para o primeiro caso, ou *CLEAR_ON_DESELECT* para o segundo.

Apesar de não ser obrigatório, o conceito de *garbage collector* também está previsto na especificação de *Java Card*. Dessa forma, para os *smart cards* que implementam esse mecanismo, seus objetos (tanto transientes quanto persistentes) são descartados tão logo não sejam mais referenciados.

2.2.8 Comunicação por meio de *Java Card* RMI

A partir da versão 2.2 de *Java Card*, o modelo de distribuição dessa plataforma incorpora a tecnologia RMI, que consiste em um subconjunto do modelo de distribuição de objetos Java RMI. Nesse tipo de comunicação, uma aplicação servidora cria e torna acessíveis objetos remotos. Dessa forma, a aplicação cliente pode obter as referências a esses objetos, e então invocar os seus métodos. No caso de *Java Card* RMI (JCRMI, 2002), o *applet Java Card* atua como servidor, e a aplicação *host,* como cliente.

Para que os seus métodos possam ser acessados remotamente, um *applet Java Card* deve implementar, direta ou indiretamente (por herança), a interface *java.rmi.Remote*. Somente os métodos especificados nessa interface são disponibilizados para acesso remoto. Qualquer comportamento excepcional que ocorra quando da execução de uma chamada remota de um desses métodos é reportado pelo JCRE por meio de um objeto *RemoteException*. Dessa forma, todo método de uma interface remota deve incluir em sua assinatura, na cláusula *throws*, a classe *RemoteException*.

A camada de transporte para as mensagens RMI é fornecida pela classe *RMIService*, que pertence ao pacote javacard.framework.service. A mensagem *Java Card* RMI é encapsulada dentro do objeto APDU passado pelos métodos de *RMIService*. Essa classe implementa o protocolo RMI, ocultando do desenvolvedor os detalhes de baixo nível. A execução de métodos remotos em *Java Card* está sujeita a algumas restrições: uma aplicação *host* cliente não pode passar um objeto remoto como argumento para um método remoto contido no *applet*; métodos remotos da aplicação *host* não podem ser invocados por um *applet* e os dados passados como argumento; e os valores de retorno de métodos devem estar de acordo com o

tamanho do campo de dados da requisição e da resposta APDU, respectivamente. Esse fluxo está ilustrado na Figura 7 a seguir.

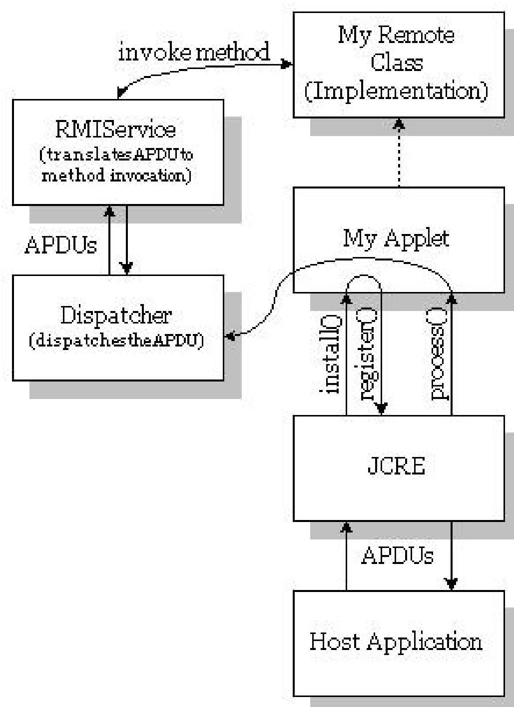

Figura 7 - Comunicação por meio de *Java Card* RMI

3 ESTADO DA ARTE DE *JAVA CARD*

As primeiras linhas de pesquisa publicadas sobre a plataforma *Java Card* remetem ao final dos anos noventa e as mais modernas estão elencadas em Grimaud e Vandewalle (2003): a verificação de *bytecode on-card*, que atualmente é feita *off-card*; a validação formal de programas (*applets*); a segurança, quer seja a aplicabilidade de *Smart Card* nessa área, ou a adulteração de informações no cartão; e a estruturação/projeto das aplicações para *Smart Cards*. Essa última linha corresponde às publicações mais recentes, e dentro dela é que se encontra o tema central desta dissertação.

3.1 Verificação de Código Binário

A verificação de *bytecode* existe para qualquer máquina virtual Java, quer ela seja para *Smart Cards*, JCVM, ou até mesmo para máquinas virtuais Java convencionais para servidores de grande porte. Essa verificação consiste em certificar se o programa contém seqüências de código binário válidas. Dessa forma, a verificação de *bytecode* procura garantir que o programa, uma vez compilado, não seja interpretado de forma incorreta, ou até mesmo adulterado. A questão de adulteração é particularmente relevante no contexto de *Java Card*, pois, conforme mencionado anteriormente, essa plataforma trabalha tipicamente com aplicações críticas. Essa verificação existe para qualquer plataforma e nela são certificados os seguintes aspectos:

- Não existência de nenhuma violação no gerenciamento de memória e nenhum *overflow* ou *underflow* de memória;
- Manutenção das restrições de acesso, por exemplo, que métodos ou atributos privados não possam ser acessados fora do contexto da classe;
- Invocação dos métodos com os argumentos apropriados e seus respectivos tipos;
- Alteração das variáveis com valores do tipo apropriado;
- Tipagem correta dos objetos em sua classe correspondente;
- Não adulteração dos ponteiros;
- Não realização de conversão ilegal de dados, como converter um inteiro em um apontador;
- Garantia da compatibilidade de binário.

Em uma JVM convencional, essa verificação é feita em tempo de execução e, conforme visto anteriormente, até a versão atual de *Java Card*, 2.2, o verificador de *Java Card* fica localizado fora do cartão (OFFCARD, 2002). Isso se deve ao fato que a máquina virtual inserida no cartão é a mais reduzida possível, em função de os *Smart Cards* serem plataformas de *hardware* bastante restritas. Além das verificações feitas por qualquer máquina virtual, o verificador de *Java Card* certifica os seguintes itens no momento em que ele converte o arquivo *.class no .cap*:

- Não utilização de qualquer tipo de dado não suportado: *char, long, float* e *double.*
- Não utilização de qualquer funcionalidade não suportada, como *array* multidimensionais ou *threads*;
- Limitação das operações. Limites esses que diferem de seus correspondentes em Java convencional, como, por exemplo, um pacote não pode apresentar mais de 255 classes, ou um *array* não pode apresentar mais de 32.767 elementos;
- Impedimento de que nenhum potencial *overflow* ou *underflow* de memória incorra em geração de um resultado incorreto, em vez de haver um retorno de erro.

Existe ainda outro conjunto de verificações feitas pelo instalador, que são feitas no momento em que ele carrega o *arquivo .cap* dentro do *Smart Card*, como se segue:

- Os pacotes *applets* precisam ter um AID válido e cada *applet* deve apresentar um AID com o mesmo prefixo do AID do pacote ao qual ele pertence;
- Os *applets* precisam implementar um método *install()* com parâmetros apropriados, para que, assim, o *applet* possa ser instanciado corretamente;
- As interfaces precisam ser carregadas antes das classes e das superclasses de seus descendentes, de forma que o processo de ligação (*linking*) seja feito de forma seqüencial;
- O *flag* de inteiro precisa ser ativado para os *applets* que utilizam o tipo primitivo *"int"*. Dessa forma, as JCVMs que não suportarem esse tipo podem rejeitar a aplicação.

Atualmente existe um grande esforço de pesquisa para introduzir a verificação de *bytecode*

internamente no cartão (*on-card*). Uma das primeiras propostas para realizar a validação de tipos *on-card* foi publicada por Grimaud, Lanet e Vandewalle (1999). Desde então, diversas outras propostas foram apresentadas, destacando-se como um dos maiores publicadores dessa área, Xavier Leroy (2001) que propõe um dos primeiros mecanismos para verificação *on-card*. Ele aponta ainda outras variações otimizadas desse mecanismo (LEROY, 2002), utilizando o conceito de armazenamento de transformações binárias históricas. Em uma de suas mais recentes publicações, Leroy (2003) fez um comparativo das propostas existentes de verificadores de *bytecode on-card*.

Uma linha de verificadores *on-card* que se destaca é a de verificadores leves que se limitam a um conjunto restrito de verificações a serem realizadas. Um desses verificadores é proposto por Klein e Nipkow (2001), que prevê a verificação de tipos em um único passo. Outro verificador dessa linha com mais funcionalidades é o proposto por Rose (2003), que originalmente é proposto para outra máquina virtual de Java restrita, J2ME (Java 2 *Micro Edition*), e posteriormente reformulada para *Java Card*.

3.2 Métodos Formais

Conforme apresentado por Hartel e Moreau (2001), existe um volume grande de publicações de Java *Card* que está relacionado à validação formal de seus programas ou ao processo de verificação da transformação de seus códigos binários, conforme apresentado na seção anterior. Ambos os assuntos são analisados em mais detalhes nas seções a seguir, mas antes dessa análise, segue uma breve descrição sobre as técnicas de métodos formais utilizadas nessas validações e verificações.

De acordo com Morgan (1994), métodos formais são técnicas matemáticas para especificação, desenvolvimento e verificação de *software* e *hardware*. A principal motivação para o uso de métodos formais é que, tal como em outras engenharias, mesmo que sejam realizadas as análises matemáticas adequadas, a confiabilidade e a robustez dos programas podem ser incrementadas. A proposta dos métodos formais é que os sistemas não sejam "programados", mas sim informados do "como" eles devem proceder para um determinado processamento. Deve-se especificar o "comportamento" esperado do sistema, apenas indicando "o que" é esperado como comportamento. O refinamento da especificação para programa é feito por

meio de transformações matemáticas, o que impossibilita a introdução de um erro entre o programa a ser executado e o comportamento especificado. No entanto, o alto custo de aplicação de métodos formais implica que eles sejam utilizados apenas no desenvolvimento de sistemas de alta integridade, nos quais a segurança seja fundamental. Esse é justamente o caso das aplicações *Java Card* que têm grande uso em sistemas de pagamento eletrônico, ou até mesmo na identificação digital. Dessa forma, ocorre a motivação para a aplicação de métodos formais nessa plataforma.

O primeiro método formal apresentado nesta seção, utilizado por alguns autores para realizar provas matemáticas de algumas propriedades da plataforma *Java Card*, é a própria linguagem de código binário, a JVML – *Java Virtual Machine Language* (LINDHOLM; YELLIN, 1999) que provê os operadores de baixo nível disponíveis na máquina virtual Java como, por exemplo, “empilhar” e “desempilhar”.

Uma das técnicas formais mais usadas nas publicações relacionadas com *Java Card* é o Método-B. Segundo Abrial (1996), o Método-B é um método formal baseado na notação de máquina abstrata. O exemplo mais conhecido desse tipo de máquina é a máquina de Turing (TURING, 1950). Essas máquinas são modelos teóricos de sistema de *hardware* ou *software* nos quais são definidas as entradas, as saídas e as operações. Com isso, é possível fazer validações matemáticas nos programas analisados. O Método-B, como todo método formal, possibilita o refinamento de especificação para código fonte e, para tal, conta com vasto ferramental. Esse método é bem similar a outro método também bastante aplicado em publicações sobre *Java Card*, a notação Z. Essa notação é considerada de mais alto nível que o Método-B, mas não apresenta a mesma facilidade que B para especificação de programas corretos.

Em Meyer, Abrial e Schuman (1980), a notação Z é baseada na teoria axiomática dos conjuntos, cálculo *lambda* (MCCARTHY,1963) e cálculo de predicados de primeira ordem (VAN DALEN, 1997). Nela todos os elementos são tipados e já dispõem de um catálogo de funções matemáticas e de predicados comumente utilizados.

Uma notação mais moderna utilizada também na validação de programas *Java Card* é a JML - Java Modeling Language (LEAVENS; BAKER; RUBY, 1999). Essa notação é uma linguagem de especificação para programas Java e utiliza a lógica de Hoare (HOARE, 1969)

para declarar pré/pós-condições e invariantes. As especificações são anotadas como comentários de programas, o que permite que eles sejam compilados em qualquer ferramenta Java. Existem ainda ferramentas de verificação de JML que trabalham sobre as especificações inseridas nos comentários. Por ser uma linguagem de especificação de comportamento de interface, a JML provê semântica para descrever formalmente o comportamento de um módulo Java, removendo, assim, qualquer ambigüidade sobre a intenção do projetista do módulo.

A última linguagem de especificação também utilizada nos trabalhos sobre *Java Card* é a OCL - *Object Constraint Language* (WARMER; KLEPPE, 2003). A OCL é uma linguagem declarativa para descrever as regras que se aplicam a modelos UML, provendo o modelo como expressões livres de ambigüidades da linguagem natural e sem a dificuldade intrínseca do uso de expressões matemáticas complexas. As expressões em OCL são compostas de quatro partes: um contexto, delimitando a situação em que a expressão é válida; uma propriedade, que representa alguma característica do contexto; uma operação, que manipula ou qualifica a propriedade; e palavras chaves, tais como *if, then, else, and, or, not* e *implies*.

Alguns trabalhos referentes ao uso de métodos formais em *Java Card* estão relacionados com a questão de verificação de *bytecode*. O trabalho publicado por Casset, Burdy e Requet (2002) apresenta uma validação formal do verificador *off-card* de *bytecode*, usando o Método-B.

Outro trabalho (LANET; REQUET, 1998) também utiliza esse mesmo método que, além da verificação de código binário, faz uma validação da otimização de código dos *applets*.

Outros trabalhos que estão relacionados a métodos formais focam na validação formal de programas *Java Card*. Uma dessas publicações (FREUND; MITCHELL, 1999) provê uma especificação formal de toda a API da plataforma, utilizando JVML (LINDHOLM; YELLIN, 1999). Outro artigo (BECKERT, 2000) apresenta uma forma de representação de programas *Java Card*, como cláusulas da lógica de Hoare (HOARE, 1969). Em Barthe et al. (2001), o verificador de *bytecode* de *Java Card* é especificado por meio de expressões matemáticas e, dessa forma, sua correção é validada. Esse processo é realizado com o suporte da ferramenta *Coq Proof Assistant*, que é também utilizada por Andronick et al. (2003), para validar a propriedade de confidencialidade da API de segurança fornecida pelo *Java Card* (outros aspectos de segurança são discutidos na seção a seguir).

Dentro das publicações que se preocupam com a validação formal de programas *Java Card*, um grande número delas utiliza a notação JML. Uma das primeiras iniciativas em especificar a API de *Java Card* utilizando JML foi proposta por Poll, Van Denberg e Jacobs (2001). Nela, o comportamento da interface de todas as classes da API é especificado com análise detalhada na classe APDU. Outra publicação sobre JML em *Java Card* é feita por um grupo de pesquisa do fabricante de *Smart Cards*, Gemplus. Nesse artigo (BURDY; REQUET; LANET, 2003), a correção de alguns *applets* desse fabricante é validada por meio de JML. Essa validação é suportada ainda por uma ferramenta utilitária produzida pelo próprio grupo. Uma proposta de ferramenta especializada em validações com JML é realizada por Marche, Paulin-Mohring e Urbain (2004). Essa ferramenta automatiza a geração de obrigações de prova e auxilia no refinamento de pré e pós-condições. A API de *Java Card* é especificada por outras notações formais além de JML. Uma dessas especificações é realizada por Larsson e Mostowski (2003), utilizando OCL, que ainda apresenta um comparativo da especificação com JML em contraposição a sua versão proposta em OCL.

3.3 Segurança em *Smart Cards*

Um vasto campo de pesquisa para *Smart Cards* é a segurança, pois suas aplicações são tipicamente críticas como pagamento eletrônico, o que propicia uma preocupação para que as informações armazenadas no cartão não sejam adulteradas. Ou ainda a própria aplicação dessa tecnologia em sistemas diversos, de forma a aumentar a segurança deles.

Mais especificamente, os principais riscos de segurança inerentes à não-adulteração de informações na plataforma *Java Card* são analisados por Ghosh (1998) e revisados por Renaudin et al. (2004), os quais são:

- Protocolos de transação eletrônica;
- Interação por protocolo;
- Cartões com suporte a multiaplicação;
- Implantação de tecnologias imaturas em aplicações críticas;
- Questões sobre segurança crítica.

Alguns mecanismos que se preocupam com a garantia de não-adulteração de dados em cartões estão implementados na própria máquina virtual de *Java Card*, conforme McGraw e Felten (1999), que são: verificador de *bytecode*, instalador de *applets*, compartilhamento de objetos (*object sharing*), atomicidade de transações e a API de segurança.

Conforme JCPS (2001), a JCVM faz algumas verificações durante a instalação do arquivo .cap, descritas a seguir:

- De instalação de dados corrompidos;
- De instalação de dados adulterados;
- De incompatibilidade entre o arquivo .cap e os recursos disponíveis no cartão;
- De acesso ilegal a partir do exterior do arquivo .cap;
- De recursos insuficientes e outros erros durante a inicialização do aplicativo;
- De estado inconsistente em virtude de falta de energia no cartão.

Além dessas verificações, o instalador pode se utilizar assinaturas digitais (MENEZES, 1997) para garantia da origem do *applet* a ser carregado.
Durante a execução de um programa Java convencional, é aplicado o modelo caixa de areia (*sandbox*), no qual as diretivas de segurança configuradas para essa máquina virtual são garantidas pela caixa. No entanto, ainda por se tratar de um *hardware* restrito, *Java Card* não dispõe desse modelo de *sandbox*. A fim de garantir algumas diretivas de segurança em tempo de execução, o ambiente de execução de *Java Card* fornece um contexto de execução para cada *applet*. Separando cada contexto, existe um *firewall*, que impossibilita que um *applet* consulte informações em outros *applets*. A única forma que os objetos têm de trocarem informações é o modelo de compartilhamento de objeto por meio de declaração explícita desse compartilhamento, por meio da implementação de uma interface (*Shareable*).

A integridade desse mecanismo de *object sharing* padrão da JCRE é verificada por meio de experimentos realizados por Bieber et al. (2000) e testada por meio de uma ferramenta proposta por ele (BIEBER et al., 2001). Em artigos mais recentes (SCHWAN, 2007), é proposta uma extensão para o modelo de segurança dos *Smart Cards*, na qual é estruturada uma política de segurança sobre a comunicação entre o cartão e o aparelho no qual ele é inserido.

Em Hubbers e Poll (2003), são apresentadas as características de *Java Card* que se preocupam com a atomicidade das transações, são elas: a JCVM garante que qualquer atualização em um único atributo é feita de forma atômica; garante também que qualquer cópia atômica feita por meio da classe Util é feita de forma íntegra; e, por fim, o modelo de transação de *Java Card* garante que qualquer conjunto de operações feitas dentro de um escopo transacional seja feito de forma atômica. Uma ferramenta foi proposta por Chaumette et al. (2003) para tentar quebrar a segurança desse controle transacional de *Java Card* e também de seu modelo de *object sharing*.

Em um artigo publicado por Jurjens (2006), as funcionalidades de segurança fornecidas pela API de *Java Card*, tais como criptografia simétrica e assimétrica (MENEZES, 1997), são analisadas por meio da validação das propriedades de segurança, autenticidade, confidencialidade e integridade de uma aplicação. Essa validação é suportada por uma ferramenta *opensource*, chamada *JavaSec*.

Com relação ao ramo de pesquisa de *Smart Cards* que procura aumentar o nível de segurança de outras plataformas com a aplicação da própria tecnologia de cartões, pode-se citar a publicação de Chan (2004). Em seu artigo, ele propõe o uso de *Smart Cards* no armazenamento de *cookies* HTTP (KRISTOL, 2001), o que aumenta a segurança e a facilidade de uso de *cookies* na Internet. Outra aplicabilidade da tecnologia de *Smart Card* é proposta por Loutrell e Pujolle (2003), utilizando-se de cartões para autenticação em redes locais sem fio (WLANs).

3.4 Pesquisas Atuais sobre *Java Cards*

Uma série de temas de pesquisa para *Java Card*s é proposta por Grimaud e Vandewalle (2003), em que os principais assuntos a serem estudados nessa plataforma estão descritos a seguir. Assim, são apresentados os principais problemas encontrados atualmente e os desafios para superá-los.

- Java Completo para *Smart Cards*

Um dos desafios do *Java Card* é expandir seu suporte às facilidades de Java. Até a versão 2.2 dessa plataforma, não existe, por exemplo, suporte a tipos primitivos como *double* e *float* ou

ainda operações aritméticas de ponto flutuante. Dessa maneira, os principais desafios são:

- Disponibilizar JCVMs mais poderosas suportadas por *Smart Cards* com maior poder de processamento, possibilitando, assim, o fornecimento de funcionalidades como *garbage collector* ou *multi-threading*;

- A comunicação com cartão está limitada ao protocolo APDU ou RMI, mas mesmo o suporte que existe hoje para esse protocolo nada mais é do que uma conversão da invocação de um método remoto RMI para um comando APDU;

- As APIs de *Java Card* fornecem um conjunto muito restrito de Java, visto que nem ao menos APIs básicas, como a de tratamento de coleções ou de internacionalização, são suportadas.

- Integração de Cartões em um Mundo Conectado

O Cenário atual prevê sistemas cada vez mais interligados por meio de redes. Dessa maneira, é possível fornecer conteúdos mais ricos e serviços com maior valor agregado sem comprometer a segurança e preferencialmente de forma ubíqua para os clientes. Dessa forma, os principais desafios são:

- Apesar de os *Smart Cards* potencialmente contribuírem para a garantia da segurança dos sistemas, sua interligação com outros sistemas é limitada, devido ao fato que seu protocolo de comunicação se restringe ao APDU.

- Emprego de um modelo de aplicações mais complexo com, por exemplo, suporte a integração com sistemas de mensageria (LINTHICUM et al., 2003) ou a paradigmas de programação mais atuais, como orientação a aspectos (KICZALES et al., 1997).

- Uso extensivo de componentes comuns de Java de forma estruturada com a aplicação de padrões e de técnicas de projetos orientados a objetos.

- Cartões Flexíveis e Adaptáveis

O principal desafio desse tópico está relacionado às questões discutidas nas seções anteriores desta dissertação. Por se trabalhar em um *hardware* restrito, a JCVM não apresenta uma série de características disponíveis nas máquinas virtuais convencionais:

  - A JCVM não apresenta um mecanismo de carregamento de classe dinâmico em tempo de execução. Essa operação é feita de forma estática em um módulo da máquina virtual externo ao cartão;

  - O gerenciamento de aplicações ainda é deficiente. Conforme apresentado em seções anteriores, o processo de instalação prevê algumas verificações durante a instalação da aplicação e, até mesmo, um processo de validação de assinatura digital, mas ainda existem pontos de melhoria nesse gerenciamento, entre eles alguns propostos por Global Platform Association (2001).

Algumas dessas questões estão sendo atendidas pela especificação de *Java Card* a ser lançada, na versão 3.0.

O cerne desta dissertação concentra-se em dois dos pontos apresentados no item Integração de Cartões em um Mundo Conectado, pontos indicados por Grimaud e Vandewalle (2003): suportar aplicações *Java Card* cada vez mais complexas e estruturá-las para a integração com outras aplicações.

3.5 Conclusão

Neste capítulo, é apresentado o cenário atual das áreas de pesquisa sobre *Java Card*, podendo-se constatar que as principais vertentes de pesquisa da área: verificação de *bytecode on-card,* validação de programas por meio de métodos formais e segurança, tanto para o caso contra adulteração de informações no cartão, quanto para o caso de aumentar a segurança de outras plataformas com o uso dos *Smart Cards*. Além dessas, conforme Grimaud e Vandewalle (2003), têm surgido novas linhas de pesquisa que procuram estruturar as aplicações a serem

desenvolvidas para *Java Card*, para as quais justamente este trabalho se orienta.

4 *FRAMEWORK* JCFOUNDATION

Conforme mencionado na introdução, uma definição usualmente aceita para o conceito de *framework* é apresentada por Johnson e Foote (1988) como sendo "um conjunto de classes que encapsula o desenho de solução para uma família de problemas relacionados". Outro conceito importante de *frameworks* é apresentado posteriormente por Pree (1994), que define os termos de *frozen spots* e *hot spots*. O primeiro refere-se à estrutura fixa do sistema já fornecida previamente pelo próprio *framework*, enquanto os *hot spots* são os pontos de extensão do *framework* que devem construídos para cada aplicação. Esse mesmo autor apresenta outra contribuição importante, descrevendo como documentar *frameworks* (PREE, 1995). Nessa publicação, ele indica que a documentação de um *framework* deve apresentar ao menos uma descrição de cenário de uso, a estrutura interna e as orientações para implementação dos pontos de extensão. Um último conceito a ser observado sobre *frameworks* é que eles podem ser baseados em padrões de projeto (JOHNSON, 1992 e SRINIVASAN, 1999). Essa mesma idéia é aplicada ao *framework* proposto, JCFoundation, em cada um de seus módulos: de estruturação da troca de comandos entre a aplicação Java no *Smart Card* e o receptor; de isolamento da camada de persistência da camada de negócio da aplicação;e de encapsulamento da camada de comunicação via *Java Card* RMI, são utilizados respectivamente os padrões de projeto *Command*, *Bridge* e *Adapter* (GAMMA et al., 1995).

Conforme Fayad, Schmidt e Johnson (1999), os *frameworks* podem ser classificados de acordo com seu escopo de uso e pelo seu modelo de extensão. A classificação segundo o escopo de uso define três tipos de *frameworks*: de infra-estrutura, de integração e de aplicação. Os de infra-estrutura simplificam o desenvolvimento da estrutura base das aplicações, como interação com sistema operacional, como a construção da interface gráfica entre outras utilidades. Os de integração são usados, em geral, para integrar aplicações e componentes, encapsulando protocolos e outras complexidades da camada de comunicação. E os de aplicação estão voltados para domínios de aplicação mais amplos e são a base para atividades de negócios das empresas como, por exemplo, nos sistemas de telecomunicações, na aviação, na manufatura e na engenharia financeira. Em outras palavras, esse tipo de *framework* encapsula conceitos do domínio de negócio no qual ele encontra-se inserido.

A classificação segundo seu modelo de extensão define dois tipos: os caixa branca e os caixa preta. No *framework* caixa branca, o reuso é provido por herança, ou seja, o usuário deve criar

subclasses das classes abstratas contidas no *framework,* para criar aplicações específicas. Para tal, o projetista deve entender detalhes de como o *framework* funciona para poder usá-lo.

Já o *framework* caixa preta é instanciado por meio de *scripts* de configuração e por uma ferramenta de compilação que produz os código fonte e o executável. Nesse tipo de *framework,* o projetista não precisa conhecer a sua estrutura interna, o que simplifica a sua aplicação, mas tipicamente limita sua flexibilidade. O JCFoundation pode ser classificado como caixa branca de infra-estrutura.

4.1 Estruturação da Troca de Comandos Entre a Aplicação *Java* no *Smart Card* e no Leitor de Cartões

Para o módulo do JCFoundation que procura estruturar a troca de comandos entre a aplicação Java no *Smart Card* e o receptor, é aplicado o padrão de projeto *Command* (GAMMA et al., 1995), no qual é introduzido adicionalmente um despachante de comandos.

4.1.1 Cenário de uso

A plataforma *Java Card* fornece uma interface de desenvolvimento em um nível de abstração baixo, o que permite a construção de *applets* de difícil manutenção e extensão.

Assim, esse módulo do JCFoundation procura solucionar tais problemas por meio do encapsulamento dos comandos *Application Protocol Data Unit* (APDU) como um objeto, permitindo que haja o tratamento desses comandos de forma uniforme, e que ele seja utilizado nos seguintes cenários:

- O processamento de comandos APDU deve ser executado de forma encapsulada, simplificando o código do *applet*;
- A introdução de novos comandos deve ser simplificada;
- O tratamento do fluxo de execução de comandos deve ser simplificado e centralizado;
- A necessidade de suportar reversão (*undo*). A execução dos comandos de aplicações críticas (por exemplo: carteira eletrônica), dentro de um contexto

transacional, é de fundamental importância.

4.1.2 Estrutura Interna

Para cada comando APDU que encapsula a operação a ser executada. Além desses comandos, é apresentado um despachante que seleciona o comando adequado a ser executado. O uso dos comandos permite uma implementação padronizada e isolada da operação a ser executada. Já o despachante permite centralizar o controle de fluxo de execução dos comandos.

- **Visão Estática**

O diagrama de classe na Figura 8 a seguir representa a visão estática desse módulo do JCFoundation, acompanhado da descrição de cada um dos seus elementos.

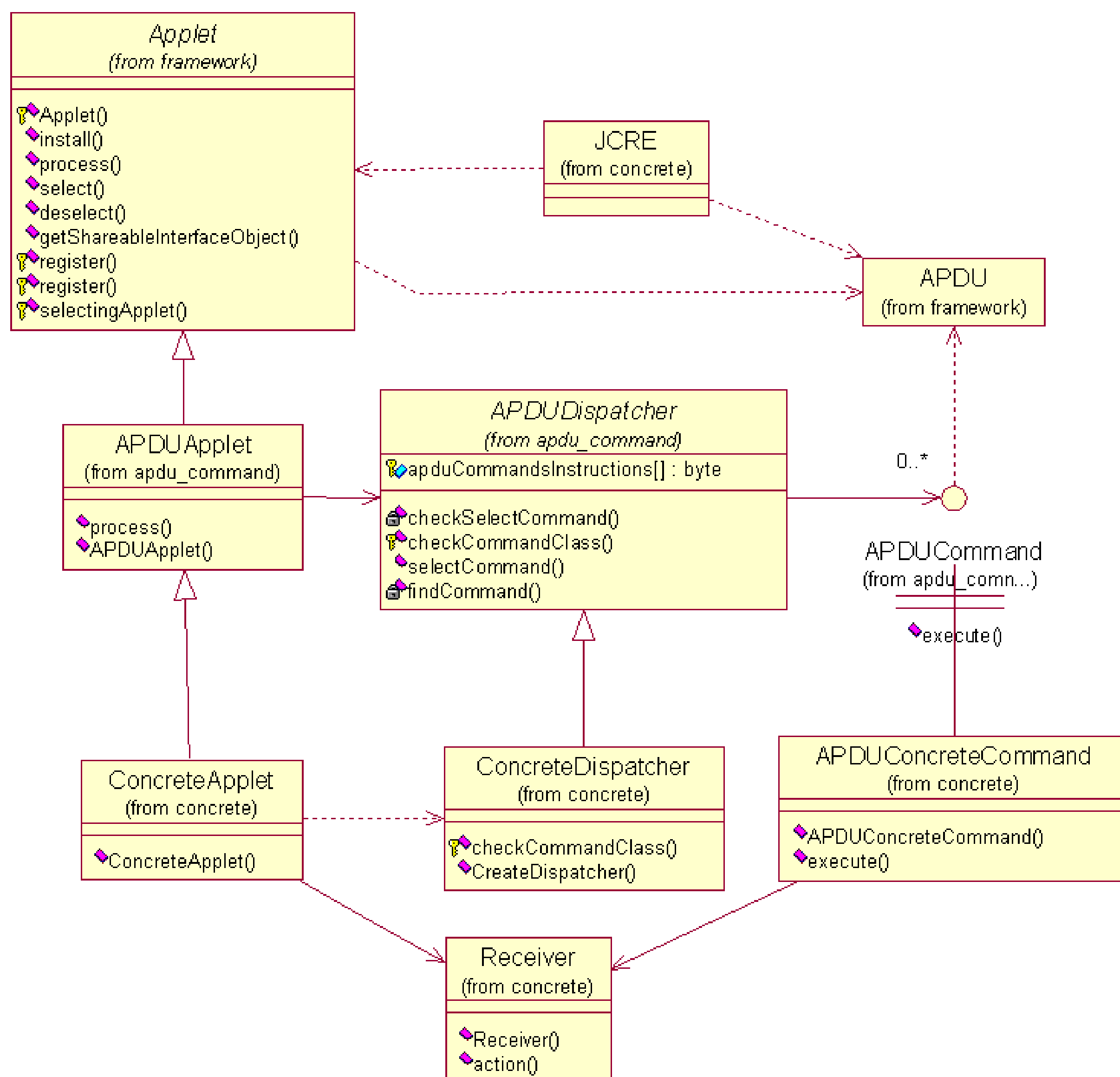

Figura 8 - Diagrama de Classe do padrão *Command* aplicado ao *Framework*

- ***JCRE* (*Java Card Runtime Environment*)**

A classe *JCRE* representa o ambiente de execução dos *applets Java Card* e é responsável pelo ciclo de vida dos *applets*, incluindo o envio aos *applets* de objetos APDU que representam comandos APDU.

- ***Applet***

A classe abstrata *javacard.framework.Applet*, pertencente à API de *Java Card*, deve ser estendida por todo *applet* que precise ser carregado, instalado e executado, em uma plataforma *Smart Card* compatível com *Java Card*.

- ***APDU***

A classe *javacard.framework.APDU* pertence à API de *Java Card*. Essa classe representa a APDU, que é a unidade de dados utilizada no protocolo de comunicação entre a aplicação residente no *Java Card* e a aplicação cliente. O formato do APDU está definido na especificação ISO 7816-4.

- ***APDUApplet***

A classe *APDUApplet* recebe objetos APDU, interage com o *APDUDispatcher* para selecionar o *APDUCommand* correto e executá-lo.

- ***APDUDispatcher***

A classe *APDUDispatcher* é responsável por gerenciar o fluxo de execução dos comandos. Ele seleciona o *APDUCommand* de acordo com o APDU requisitado.

- ***APDUCommand***

A interface *APDUCommand* define um único método *execute()*. Cada comando APDU deve implementar esse método, contendo a operação referente ao comando requisitado.

- ***ConcreteApplet***

A classe *ConcreteApplet* representa a aplicação, executando no JCRE, por exemplo, uma carteira eletrônica. Essa classe define o *ConcreteDispatcher* correto para a aplicação. O estado da aplicação pode ser armazenado no *Receiver* ao qual o *applet* concreto faz referência.

- ***ConcreteDispatcher***

A classe *ConcreteDispatcher* representa o despachante. Ela contém o mapeamento de cada comando APDU para o comando concreto correspondente. Além disso, implementa o método abstrato *checkCommandClass(),* que verifica se o comando APDU pertence à categoria da aplicação.

- ***APDUConcreteCommand***

A classe *APDUConcreteCommand* representa cada comando ADPU a ser executado pelo *applet*. Para cada comando ao qual o *applet* responda, deve ser criado um comando concreto. A implementação do método *execute()* encapsula toda operação a ser executada requisitando o respectivo comando APDU. O comando concreto pode também atuar sobre um *Receiver* para alterar o estado da aplicação.

- ***Receiver***

A classe *Receiver* representa o estado da aplicação, que normalmente é um componente da camada de negócio. Essa classe é referenciada pelo *applet* concreto e sobre ele os comandos concretos atuam.

- **Visão Dinâmica**

Na fase inicial do ciclo de vida do *applet*, o JCRE invoca o método *install()* para instalá-lo. Nesse método, a maioria dos objetos é instanciada. O primeiro objeto a ser instanciado é o *ConcreteApplet*. Este, por sua vez, instancia o *Receiver* e o *ConcreteDispatcher,* enquanto o despachante instancia os *APDUConcreteCommands*. Por fim o *applet* registra-se no JCRE.

Outra fase do ciclo de vida é a seleção, quando o JCRE seleciona o *applet*, por meio do método *select()*. A partir de então, ele fica disponível para atender às requisições.

Ao receber um comando APDU externo, o JCRE cria um objeto *APDU* e envia-o ao *ConcreteApplet* selecionado, por meio do método *process()*. O *applet,* por sua vez, invoca o método *selectCommand()* do *ConcreteDispatcher* para selecionar o *APDUCommand* correto. Nesse momento, o despachante verifica se o comando enviado é um comando de seleção, e caso seja, ignora-o. Ele também verifica se o comando enviado pertence à categoria dos comandos da aplicação. Por fim, o despachante localiza o *APDUConcreteCommand* adequado e, uma vez de posse do comando concreto, o *applet* executa-o, por meio do método *execute()*.

Na Figura 9, é representada a visão dinâmica desse módulo do JCFoundation por meio de um diagrama de seqüência.

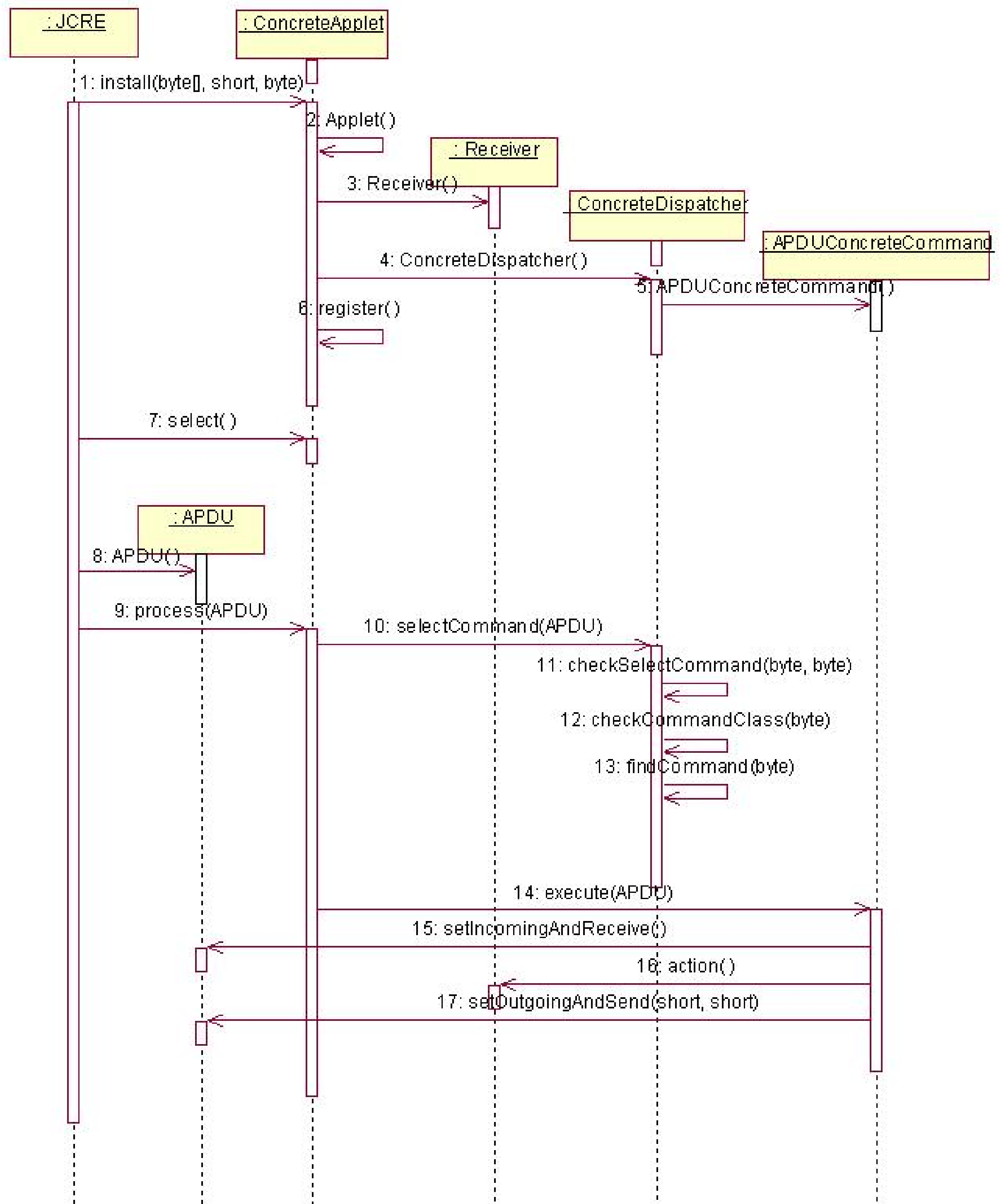

Figura 9 - Diagrama de Seqüência do padrão *Command* aplicado ao *Framework*

4.1.3 Implementação dos pontos de extensão

- **Implementação Padrão**

Para cada comando APDU, deve ser criado um comando concreto que se estende à classe abstrata *APDUCommand.* Deve-se, ainda, criar um despachante concreto que estenda o *APDUDispatcher* e contenha o mapeamento de cada comando APDU para seu respectivo comando concreto.

- **Implementação da Reversão (*undo*)**

A plataforma *Java Card* oferece um mecanismo de controle de transação por meio dos métodos *abortTransaction(), beginTransaction()* e *commitTransaction()* da classe *javacard.framework.JCSystem.* Esse controle permite criar uma transação que englobe apenas um único comando. Para suportar uma transação englobando diversos comandos, a implementação descrita nesta seção pode ser útil.

Para tal, os comandos devem implementar uma operação *undo(),* contendo uma operação inversa à operação *execute().* Além disso, é necessário criar comandos APDUs específicos para início, confirmação e reversão da transação. O *applet,* por sua vez, precisa armazenar os últimos comandos executados a partir do recebimento do comando de início da transação e, ao receber o comando de reversão, deve invocar o método *undo()* dos comandos armazenados. Caso receba um comando de confirmação de transação, ele pode descartar os comandos armazenados.

4.2 Isolamento da Camada de Persistência da Camada de Negócio da Aplicação

Para o módulo do JCFoundation que procura isolar a camada de persistência da camada de negócio da aplicação, é aplicado o padrão de projeto *Bridge* (GAMMA et al., 1994), no qual a abstração em questão refere-se a repositório de dados do objeto de negócio e sua implementação à estrutura de dados utilizada por ele. Dessa maneira, ambos os conceitos podem ser estendidos independentemente.

4.2.1 Cenário de uso

O modelo de memória da plataforma *Java Card* fornece dois tipos de armazenamento de objetos, persistente e transiente. Os objetos persistentes existem a partir do momento em que são criados, até o final da vida útil do *smart card*. No entanto, eles podem ser descartados pelo *garbage collector*, uma vez que nenhum outro objeto os referencie. Os objetos transientes, por sua vez, têm o tempo de vida bem mais curto. Eles só existem enquanto o cartão está energizado pelo seu leitor e podem, ainda, ter um tempo de vida ainda mais curto, restrito ao tempo da aplicação em o objeto está selecionado.

Dependendo do requisito da aplicação, é preciso armazenar tanto objetos persistentes quanto transientes. A atividade de manutenção pode ser simplificada se o gerenciamento dos dois tipos de armazenamento for feito de forma transparente, ou seja, operações realizadas em repositório transiente são idênticas, apenas variando o meio em que os dados são armazenados. Além desse requisito, a aplicação pode demandar que o mecanismo utilizado para armazenar informações seja cambiável, sem impactar os objetos de negócio. Estruturas de dados diferentes podem atender mais adequadamente a diferentes aplicações.

Dessa maneira, esse módulo do JCFoundation pode ser aplicado nos seguintes cenários:

- A aplicação necessita trabalhar com objetos persistentes ou transientes de forma transparente;
- O repositório de dados da aplicação deve utilizar a estrutura de dados mais adequada para atender aos requerimentos em questão, possibilitando a troca dessas estruturas, ou de suas implementações, sem impactar os objetos de negócio;
- O tempo de vida de objetos transientes precisa ser alterado em tempo de execução. Conforme mencionado anteriormente, os objetos transientes podem ter seus ciclos de vida associados à inativação do cartão, *CLEAR_ON_RESET*, ou à inativação da aplicação à qual pertence, *CLEAR_ON_DESELECT*.

4.2.2 Estrutura Interna

Para solucionar o problema apresentado, o JCFoundation fornece um repositório de dados que, por sua vez, tem sua abstração, *Datastore*, segregada das suas diversas implementações. Estas últimas são divididas em dois tipos, persistentes e transientes. As transientes são ainda subdivididas de acordo com o tempo de vida de seus objetos, *CLEAR_ON_RESET* e *CLEAR_ON_DESELECT*. A persistente ainda dispõe de uma implementação básica que utiliza como estrutura de dados *array* de objetos.

O repositório define as operações básicas de inclusão, remoção, localização e atualização. Para as operações que envolvem busca, é preciso definir uma chave de identificação para os objetos a serem armazenados. Dessa forma, os objetos a serem persistidos precisam implementar uma interface que define uma unidade de dados, *IDataUnit*. Essa interface especifica ainda uma operação de atualização de dados que é acionada pelo método de atualização do repositório.

- **Visão Estática**

Na Figura 10, está representada a visão estática desse módulo do JCFoundation por meio de um diagrama de classes, seguido da descrição de cada um dos seus elementos.

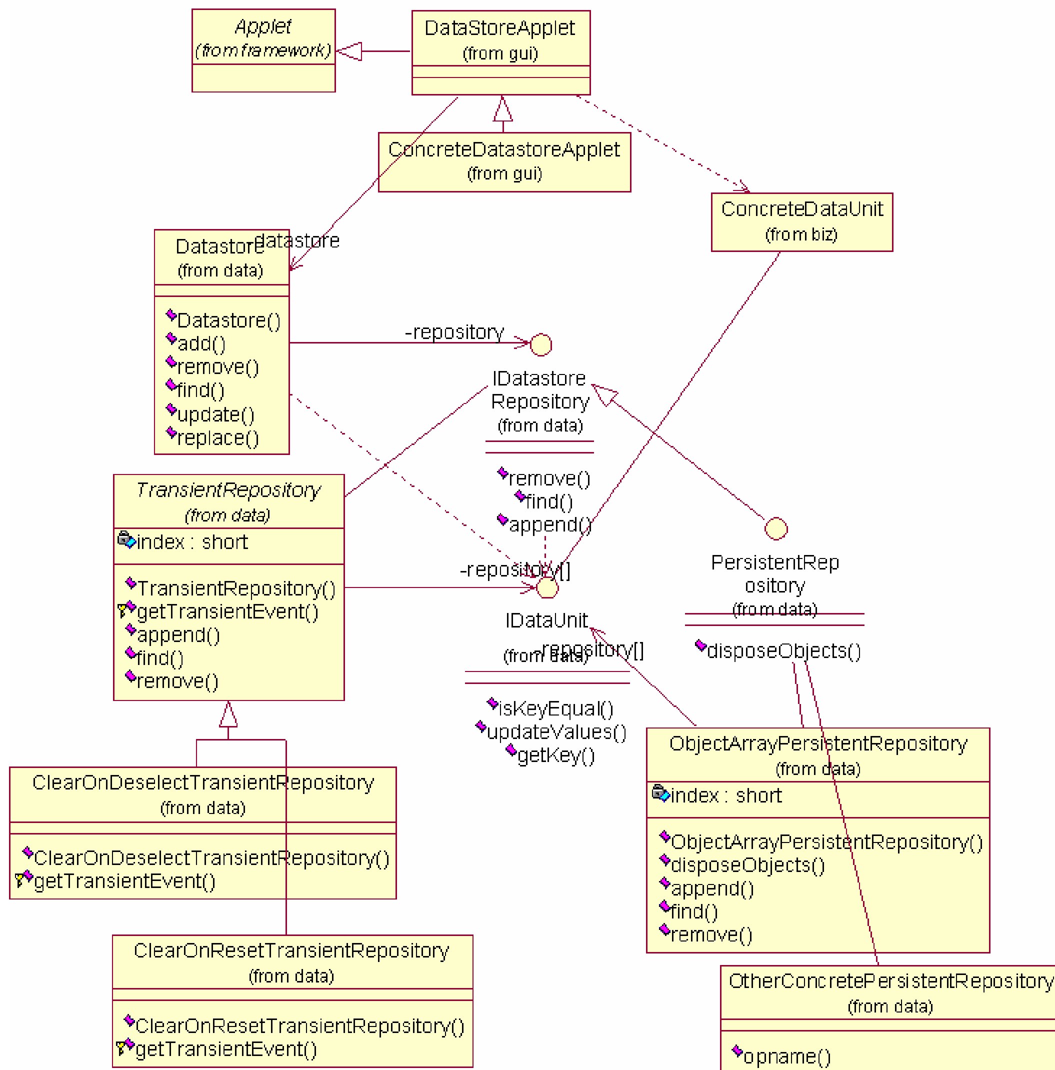

Figura 10 - Diagrama de Classe do padrão *Bridge* aplicado ao *Framework*

- ***Applet***

A classe abstrata *javacard.framework.Applet*, pertencente à API de *Java Card*, deve ser estendida por todo *applet* que precise ser carregado, instalado e executado em uma plataforma *Smart Card* compatível com *Java Card*.

- ***DatastoreApplet***

A classe *DatastoreApplet* recebe objetos APDU e mantém uma referência para o repositório de dados, *Datastore*. A implementação concreta do *applet* não precisa obrigatoriamente derivar dessa superclasse, pois ela fornece apenas uma referência para o *Datastore* com o

objetivo de simplificar a construção de aplicações *Java Card* que necessitem de um repositório de dados que tenham uma camada de negócio simplificada. Caso a aplicação em questão apresente uma camada de negócio elaborada, o *applet* dessa aplicação deve derivar diretamente de *javacard.framework.Applet*, e a referência ao *Datastore* deve ficar na classe de negócio adequada para fazer a integração com a camada de dados.

- ***ConcreteDatastoreApplet***

A classe *ConcreteDatastoreApplet* representa a aplicação no JCRE e define a implementação de repositório correto para a aplicação. Essa classe também manipula uma unidade de dados concreta, *ConcreteDataUnit*, referente ao domínio de negócio da aplicação.

- ***IDataUnit***

A interface *IDataUnit* define o tipo de dados que pode ser armazenado no repositório, especificando que os objetos a serem armazenados precisam apresentar uma chave identificadora e devem ter a capacidade de atualizar seu estado interno.

- ***ConcreteDataUnit***

A classe *ConcreteDataUnit* representa um objeto de negócio concreto que implementa a interface de unidade de dados, de forma que ele possa ser armazenado no repositório.

- ***Datastore***

A classe *Datastore* é uma abstração do repositório. Ela define as operações de inserção, deleção, localização e atualização, juntamente com suas respectivas regras de negócio. As regras de negócio definidas são referentes a objetos não localizados e estouro de espaço no repositório. Essa classe interage com a implementação do repositório que, por sua vez, encapsula efetivamente o meio no qual os dados são armazenados.

- ***IDatastoreRepository***

A interface *IDatastoreRepository* especifica as operações básicas de armazenamento de dados, adição, localização e remoção.

- ***TransientRepository***

A classe abstrata *TransientRepository* implementa as operações referentes ao armazenamento de objetos transientes. Para tal, ela utiliza como estrutura de dados um *array* de objetos que é

inicializado pela operação definida pela API de *Java Card*, *JCSystem.makeTransientObjectArray(),* a qual cria, justamente, um *array* com essa propriedade de transiência. A única propriedade que essa classe delega para suas subclasses é a definição do tempo de vida do objeto, que pode ser limitado ao tempo que o *smart card* permanecer alimentado ou ao período de atividade da aplicação.

- ***ClearOnDeselectTransientRepository***

A classe *ClearOnDeselectTransientRepository* é um subtipo de repositório transiente que define como tempo de vida o período de atividade da aplicação, *CLEAR_ON_DESELECT*. Dessa forma, no momento em que outra aplicação do cartão é selecionada, ou em que o *smart card* é removido do dispositivo no qual ele está inserido, os objetos do repositório são descartados.

- ***ClearOnResetTransientRepository***

A classe *ClearOnResetTransientRepository* é subtipo de repositório transiente que define como tempo de vida o período no qual o *smart card* permanece energizado, *CLEAR_ON_RESET*. Dessa forma, no momento que o *smart card* é removido do dispositivo no qual ele está inserido, os objetos do repositório são descartados.

- ***PersistentRepository***

A interface *PersistentRepository* define a hierarquia de repositórios persistentes. A única operação que essa interface especifica é a *disposeObjects(),* que permite que os objetos do repositório persistentes sejam descartados. Visto que a memória nos *smart cards* é um recurso escasso, objetos persistentes devem ser utilizados com cautela, porque essa operação é importante para que a aplicação possa se desfazer dos objetos armazenados uma vez que eles não sejam mais necessários.

- ***ObjectArrayPersistentRepository***

A classe *ObjectArrayPersistentRepository* implementa a interface de repositório persistente. Esse tipo de repositório utiliza como estrutura de dados a coleção de dados mais usual no *Java Card*, o *array* de objetos. Diferentemente do *TransientRepository,* essa classe não utiliza o método disponibilizado pela API da plataforma, *makeTransientObjectArray().* Para inicializar sua estrutura de dados interna, esse repositório utiliza simplesmente o construtor

new, no qual (por definição da especificação *Java Card*) todo objeto criado com esse construtor é um objeto persistente.

- ***OtherConcretePersistentRepository***

A classe *OtherConcretePersistentRepository* representa outras implementações do repositório persistente que tipicamente utiliza uma estrutura de dados distinta do *array* de objetos.

- ***DataUnitNotFoundException***

A classe *DataUnitNotFoundException* é uma exceção de usuário que é levantada quando uma operação tenta localizar um objeto que não está armazenado. Nela ainda é armazenada a chave do objeto que não foi localizado.

- ***RepositoryOverflowException***

A classe *RepositoryOverflowException* é uma exceção de usuário que é levantada quando se tenta adicionar um objeto quando o repositório já está na sua capacidade máxima. Nela ainda é armazenada a capacidade máxima do repositório.

- **Visão Dinâmica**

- **Inicialização e Operação de Adição**

Na fase inicial do ciclo de vida do *applet*, o JCRE invoca o método *install()* para instalar o *applet*. Nesse método, a maioria dos objetos é instanciada. O primeiro objeto a ser instanciado é o *ConcreteDatastoreApplet* que, por sua vez, instancia o *ConcreteRepository* adequado para a aplicação e, logo em seguida, o *Datastore,* passando ao construtor o repositório concreto definido. Por fim, o *applet* registra-se no JCRE.

Uma vez instalado e registrado, o *applet* deve ser selecionado por meio do método *select()* para começar a receber comandos. Ao receber um comando APDU externo, o JCRE cria um objeto *APDU* e envia-o ao *ConcreteDatastoreApplet* selecionado, por meio do método *process()*. A partir de então, o *applet* pode operar sobre o repositório de dados. Uma das operações disponíveis é a de adição. O primeiro passo para efetuar a adição é criar a unidade de dados a ser adicionada, *ConcreteDataUnit*. Para realizar a operação efetivamente, basta chamar o método *add()* do *Datastore*. Este, por sua vez, delega a adição para o repositório concreto, por meio do método *append()*. Caso a capacidade do repositório seja excedida,

levanta a exceção *RepositoryOverflowException.* No diagrama da Figura 11, são apresentadas a inicialização e a operação de adição.

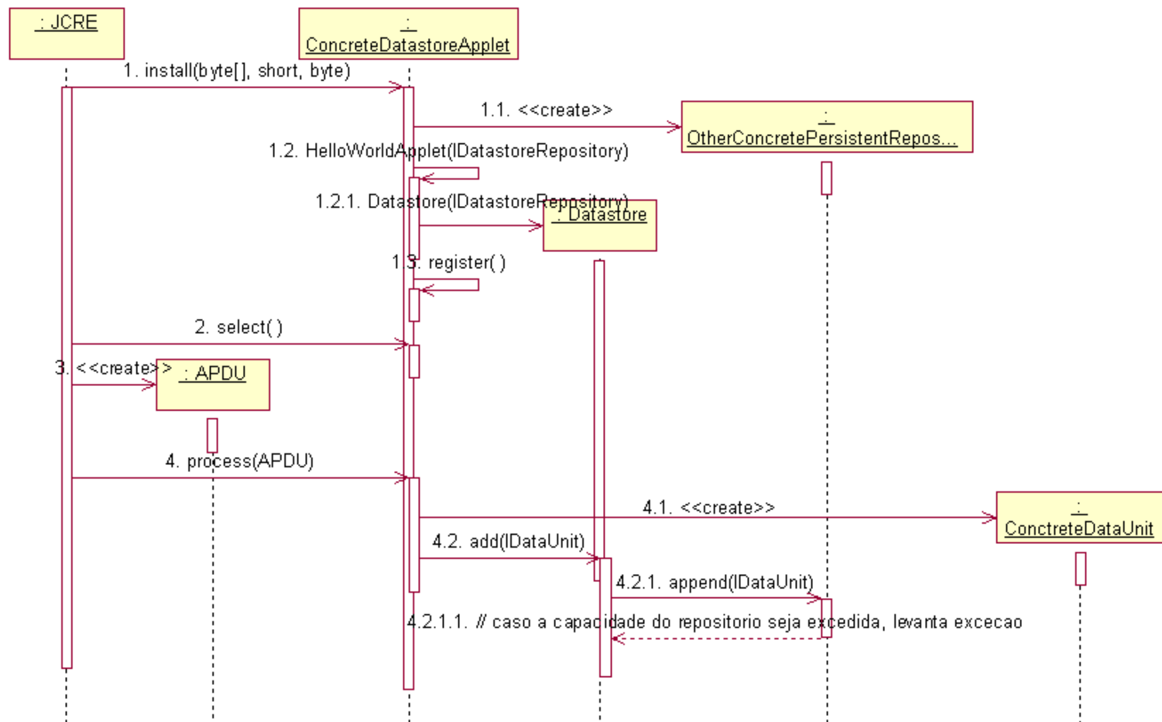

Figura 11 - Operações de Inicialização e de Adição

- **Operação de Localização e Remoção**

Além da operação de adição, existem também as operações de localização e de remoção. Ambas funcionam de forma análoga, a única diferença entre elas é que cada uma tem seu respectivo método de delegação no repositório concreto. Como toda operação no *applet*, o processo se inicia quando a JCRE instancia o *APDU* e passa-o na chamada do método *process()* do *applet*. A partir desse momento, o *ConcreteDatastoreApplet* pode executar as operações de localização e remoção no repositório. Para qualquer uma dessas operações, o primeiro passo é criar uma chave de identificação, chamada *key*. Com essa chave, o *applet* aciona o método *find()* ou *remove()* do *Datastore*, que, por sua vez, delega a execução ao *ConcreteRepository* por meio dos métodos *find()* ou *remove().* Ambos os métodos precisam comparar a chave passada por parâmetro às chaves de cada objeto dentro do repositório para realizar suas respectivas operações. Para tanto, ele aciona o método *isKeyEqual()* de cada um

dos seus *ConcreteDataUnit* armazenados. Caso o objeto não seja localizado, é levantada a exceção *DataUnitNotFoundException*. Na Figura 12, por diagrama, são apresentadas as operações de localização e de remoção.

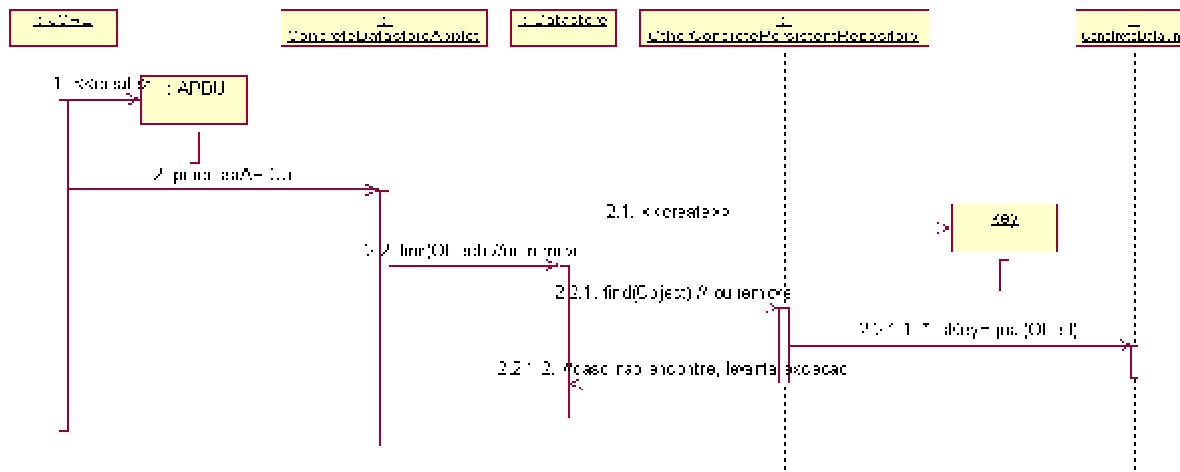

Figura 12 – Operações de Localização e de Remoção

- **Operação de Atualização**

Por fim, a última operação disponível no repositório é a de atualização. Da mesma forma que as outras operações, o processo inicia quando a JCRE instancia o *ADPU* e executa o método *process()* do *ConcreteDatastoreApplet.* O primeiro passo para fazer a atualização é a instanciação da chave do objeto a ser atualizado, *key*, e uma unidade de dados concreta, *ConcreteDataUnit*, contendo os novos valores a serem atualizados. De posse deles, o *applet* executa o método *update()* do *Datastore*, que tenta localizar a unidade de dados desejada acionando o método *find()* do *ConcreteRepository*. Uma vez localizado o objeto desejado, os dados dele são atualizados por meio do método *updateValues()* do *ConcreteDataUnit*. No diagrama da Figura 13, é apresentada a operação de atualização.

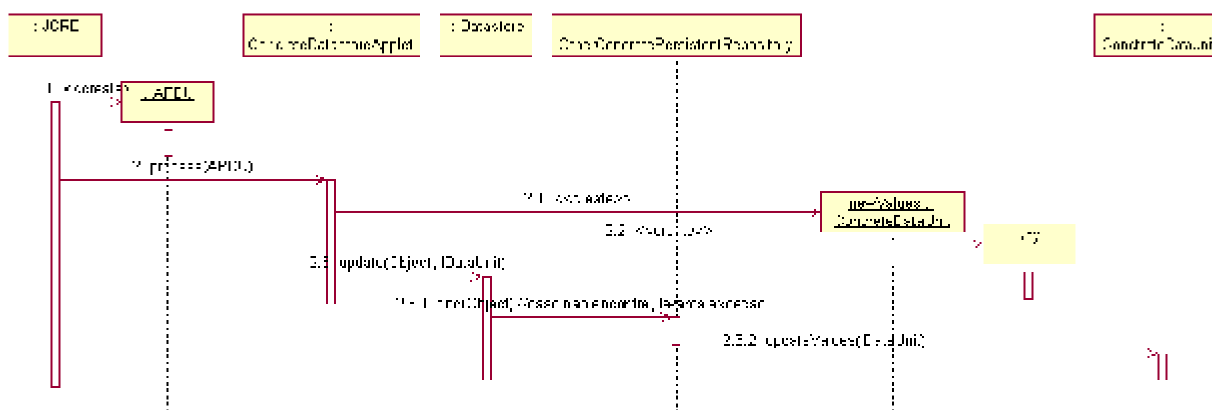

Figura 13 – Operação de Atualização

4.2.3 Implementação dos pontos de extensão

- **Implementação Padrão**

De acordo com os requisitos da aplicação, deve-se definir o uso de armazenamento transiente ou persistente. Sendo o caso de armazenamento transiente, deve-se instanciar uma classe específica, de acordo com o tempo de vida dos objetos armazenados. Caso esse tempo de vida esteja restrito à seleção da aplicação dentro do cartão, deve-se usar o *ClearOnDeselectTransientRepository* e, caso esse tempo esteja limitado ao tempo em que o cartão se encontra inserido no receptor, deve-se usar *ClearOnResetTransientRepository.* Para o armazenamento persistente, pode-se usar uma implementação padrão baseada em *array*, *ObjectArrayPersistentRepository.* Pode-se, ainda, criar uma implementação utilizando-se outra estrutura de dados, apenas tendo o cuidado de estender o *PersistentRepository***.**

- **Extensão do Datastore**

Um dos benefícios da separação de abstração da implementação do repositório de dados é o fato de as regras de negócio serem centralizadas na abstração, *Datastore*, enquanto a implementação, *ConcreteDatastore*, manipula efetivamente o armazenamento dos dados. No *Datastore* fornecido, estão definidas apenas duas regras de negócio, uma referente à falha na localização da unidade de dados, *DataUnitNotFoundException*, e outra referente ao estouro de espaço no repositório, *RepositoryOverflowException*. No entanto, o *Datastore* pode ser estendido de forma a apresentar regras de negócio mais especializadas, como, por exemplo,

não aceitar a adição de unidades de dado duplicados.

- **Implementar o Controle Transacional**

A solução proposta no JCFoundation não prevê um controle transacional. No entanto, esse controle pode ser facilmente introduzido. Para tal, deve-se inserir uma interface de controle transacional, *ITransactionControl.* Essa interface define os métodos básicos para se gerenciar transações: *begin(), commit()* e *rollback().*

Deve-se, ainda, implementar essa interface de acordo com o mecanismo de controle de transação existente hoje em *Java Card*, *JCSystem*. Caso nas versões futuras dessa plataforma sejam disponibilizados novos mecanismos de controle de transação, como, por exemplo, JTA - *Java Transaction API* (SUSAN; VLADA, 2002), novas implementações dessa interface devem ser introduzidas.

O controle de transação deve ser, então, utilizado no repositório de dados persistente concreto. Cabe observar que, por definição de *Java Card*, repositórios transientes não podem utilizar controle transacional. No início e no fim de cada método do repositório, devem ser introduzidas invocações ao controle de transação *begin()* e *commit()* respectivamente. Deve-se, ainda, adicionar uma invocação ao *rollback*, caso ocorra alguma exceção no método do repositório.

A fim de evitar alterações freqüentes no código do repositório concreto, podem-se manter as invocações ao controle transacional. E, caso isso não seja necessário, pode-se trabalhar com uma implementação da *ITransactionControl* que siga o modelo proposto pelo *refactoring* (FOWLER et al., 1999), o *Introduce Null Object*, no qual cada método tem a implementação do método vazia. Dessa forma, as invocações ao controle transacional são mantidas, mas não realizam qualquer comportamento.

4.3 Encapsulamento da Camada de Comunicação via *Java Card* RMI

Para o módulo do JCFoundation que procura encapsular a camada de comunicação *Java Card* RMI entre o *Smart Card* e o receptor, é aplicado o padrão de projeto *Adapter* (GAMMA et al., 1994), no qual as interfaces incompatíveis em questão são a interface remota e a interface

local de negócio. Dessa maneira, é possível que a aplicação cliente invoque a interface remota de forma transparente, ou seja, como se ela fosse local. A causa dessa incompatibilidade são requisitos exigidos pela tecnologia de distribuição, neste caso, RMI (RMI, 2002). Assim, tais requisitos podem ser isolados em uma camada específica sem impactarem a camada de negócio.

4.3.1 Cenário de uso

A partir da versão 2.2 de *Java Card*, o modelo de distribuição dessa plataforma incorporou a tecnologia RMI. Dessa forma, é possível que *applets* do *smart card* sejam invocados pelas suas aplicações clientes, por meio de uma chamada de método remoto. A especificação de RMI para *Java Card* é análoga à versão para J2SE, mas, como toda tecnologia de camada de distribuição, demanda certas adaptações nas interfaces e objetos de negócio em troca de comunicação remota de forma interoperável e transparente de localização. Para *Java Card*, as principais demandas são: que a interface de negócio estenda a interface *Remote* e que seus métodos levantem a exceção *RemoteException*, que o objeto de negócio estenda a classe *CardRemoteObject*, que a implementação da interface remota se registre no serviço de localização e que a obtenção da referência remota seja feita também por meio desse serviço.

Dessa maneira, esse módulo do JCFoundation pode ser aplicado nos seguintes cenários:

- Resguardo dos objetos de negócio de detalhes específicos da camada de distribuição, mais especificamente, estender a classe *CardRemoteObject*. Essa extensão, no contexto da linguagem Java, significa ainda impossibilitar a definição de uma hierarquia de tipos específica do domínio da aplicação, pelo fato de essa linguagem dispor apenas do conceito de herança simples;

- Resguardo também da interface de negócio quanto a incorporação indevida de detalhes específicos da camada de distribuição, o que, no contexto de *Java Card* RMI, significa estender a interface *Remote* e todos os seus métodos declararem *throws RemoteException*;

- Isolamento do processo de registro do objeto remoto no serviço de localização;
- Simplificação da localização do objeto remoto pela aplicação cliente.

4.3.2 Estrutura Interna

Para solucionar o problema apresentado, recomenda-se que a interface de negócio não incorpore detalhes específicos da camada de distribuição (no caso de *Java Card* RMI, que os métodos levantem *RemoteException*). Deve ser criada, adicionalmente, uma interface remota análoga à interface de negócio, incorporando as especificidades referentes à comunicação remota. Essa estratégia implica a manipulação de duas interfaces incompatíveis. Para solucionar tal questão, deve ser introduzido um adaptador que apresenta ainda uma característica específica: adaptar interfaces de uma comunicação remota. Por isso, ele é segmentado em duas partes: a origem (*Source*), que se localiza do lado cliente, e o destino (*Target*), que se apresenta no lado remoto.

O adaptador de origem encapsula a complexidade da procura do objeto remoto no serviço de localização, enquanto o adaptador remoto faz o papel do objeto remoto, implementando a interface remota, além de encapsular a complexidade de se registrar no serviço de localização.

- **Visão Estática**

A Figura 14, representa a visão estática desse módulo do JCFoundation por meio de um diagrama de classe, seguido da descrição de cada um dos seus elementos.

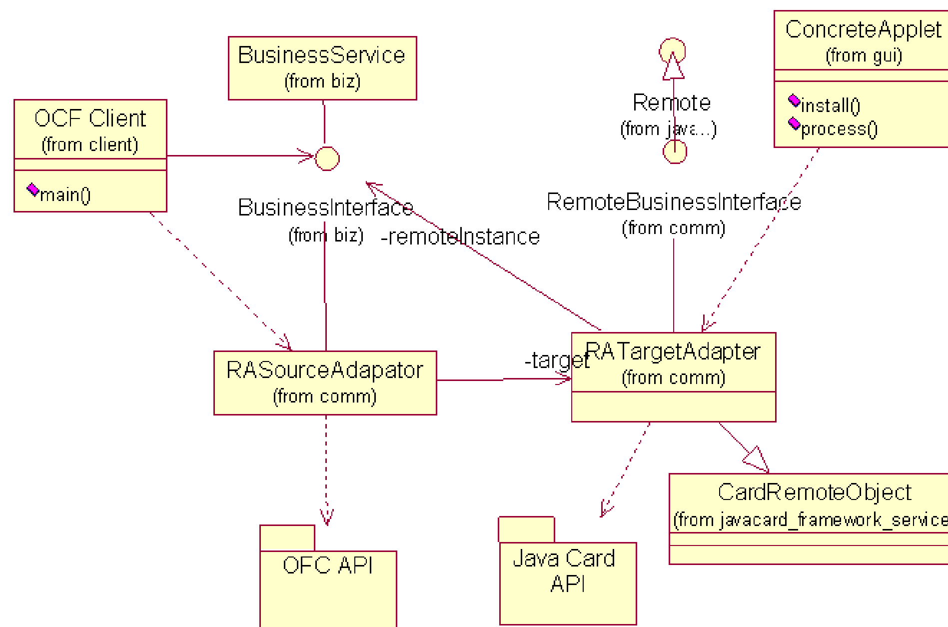

Figura 14 - Diagrama de Classe do padrão *Adapter* aplicado ao *Framework*

- ***ConcreteApplet***

A classe *ConcreteApplet* representa a aplicação executada no JCRE, que tem apenas duas responsabilidades. A primeira é criar o adaptador destino, *RATargetAdapter*, que automaticamente se registra no serviço de localização, chamado em *Java Card* de *Dispatcher*. A segunda é delegar o processamento do método *process()* ao método homônimo do *Dispatcher*.

- ***RATargetAdapter***

O adaptador destino é um subtipo de um objeto remoto, *CardRemoteObject*, e é responsável por se registrar no serviço de localização. Para tal, ele utiliza objetos específicos da API de *Java Card*, criando um *RMIService*, encapsulando uma referência para si mesmo e adicionando esse serviço ao *Dispatcher*. Além de se registrar, o *RATargetAdapter* implementa a interface remota, *RemoteBusinessInterface*, delegando as chamadas a uma referência mantida por ele do objeto de negócio, *BusinessService*. Referência que não é feita de forma direta e sim por meio da interface de negócio local, *BusinessInterface*.

- ***CardRemoteObject***

CardRemoteObject é a superclasse definida por *Java Card* RMI, da qual devem ser derivados os objetos remotos criados pela aplicação, *RATargetAdapter*.

- ***BusinessInterface***

A interface local dispõe apenas dos métodos de negócio, contendo apenas conceitos específicos do domínio da aplicação.

- ***RemoteBusinessInterface***

A interface remota estende a interface *Remote* e dispõe dos mesmos métodos de negócio, apenas incorporando as exigências da camada de distribuição. O que, no contexto de *Java Card* RMI, significa declarar *throws RemoteException* para cada método de negócio.

- ***Remote***

A interface *Remote* pertence ao pacote padrão *java.rmi* e deve ser estendida por qualquer interface de negócio que se deseje publicar remotamente.

- ***BusinessService***

O objeto de negócio concreto basicamente implementa a interface de negócio, *BusinessInterface*.

- ***OCFClient***

As aplicações clientes de *Java Card* RMI devem comunicar-se com o *applet* no *smart card,* por meio da OCF (*OpenCard Framework*) API. O *OCFClient* faz apenas chamadas ao *RASourceAdapter*, que, por sua vez, encapsula toda a complexidade específica da OCF API.

- ***RASourceAdapter***

O adaptador de origem encapsula toda complexidade para a localização do objeto remoto, *RATargetAdapter*. Essa localização envolve objetos específicos da OCF API, tais como: *SmartCard, CardRequest, CardAccessor* e *JavaCardRMIConnect*. Além disso, ele implementa a interface de negócio, *BusinessInterface*, delegando a chamada aos métodos de negócio e uma chamada ao método remoto equivalente.

- **Visão Dinâmica**

- **Inicialização**

Na fase inicial do ciclo de vida do *applet*, o JCRE invoca o método *install()* para instalar o *applet.* A principal responsabilidade desse método é criar uma instância do adaptador destino.

Para tanto, ele cria uma instância do *Dispatcher* de *Java Card* RMI e, em seguida, cria uma instância do *RATargetAdapter* passando, como parâmetro, o despachante. No construtor do adaptador destino, ele cria uma instância do objeto de negócio, *BusinessService* e, em seguida, um *RMIService*, passando a si próprio como referência e, por fim, adiciona esse serviço ao despachante. Como último passo do método *install(),* o *applet* se registra na JCRE. No diagrama da Figura 15, é apresentada a seqüência de inicialização dos adaptadores.

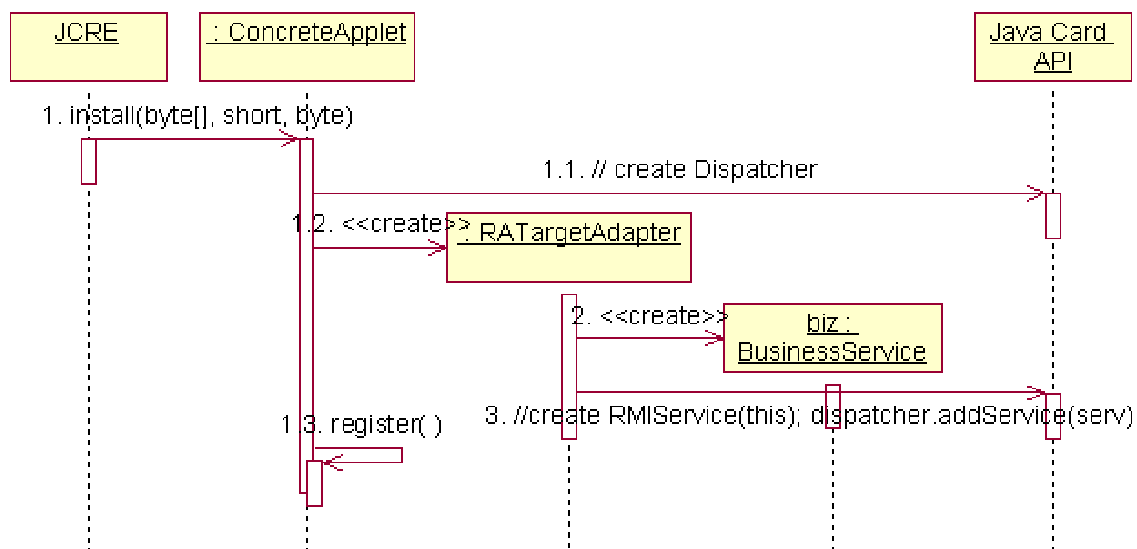

Figura 15 – Operação de Inicialização dos Adaptadores

- **Invocação Remota**

Para o *OCFClient*, uma invocação de um método remoto é análoga a uma invocação local. Para tal, ele deve criar uma instância do adaptador de origem e invocar o método de negócio desejado.

O primeiro passo do construtor do *RASourceAdapter* é invocar o método *connect*, que é responsável por obter a referência remota do adaptador destino. Para tanto, ele interage com os seguintes objetos da OCF API: *SmartCard, CardRequest, CardAccessor* e *JavaCardRMIConnect*. Este último é o mais importante, pois ele interage com JCRE para fazer a seleção do *applet* e retorna uma referência ao objeto remoto.

De posse da referência remota ao *RATargetAdapter*, o *RASourceAdapter* pode invocar métodos de negócio do adaptador destino para atender ao *OCFClient*. Ao receber uma chamada remota, o *RATargetAdapter* delega essa chamada para o método equivalente do objeto de negócio, *BusinessService*, com o qual o adaptador destino mantém uma referência.

No diagrama da Figura 16, é apresentada a comunicação distribuída entre o cliente e a aplicação remota:

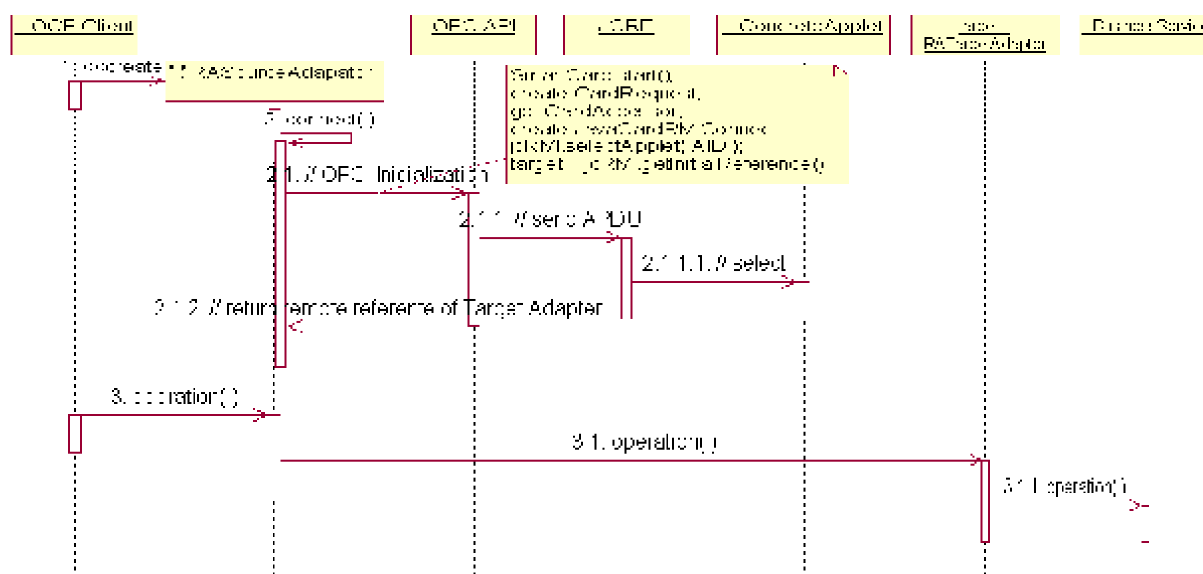

Figura 16 - Invocação Remota

4.3.3 Implementação dos pontos de extensão

- **Implementação Padrão**

Os serviços de negócio devem ser especificados por meio de uma interface, *BusinessInterface*, que não contenha qualquer conceito referente à camada de comunicação.

Deve-se, também, fornecer uma implementação padrão local para essa interface de negócio, *BusinessService*. Em seguida, deve-se criar uma interface remota, *RemoteBusinessInterface*, com as mesmas operações da interface de negócio, acrescida dos pré-requisitos impostos pela camada de comunicação. No caso de *Java Card* RMI, todo método deve levantar

RemoteException, e a interface deve estender a interface *Remote.* Por fim, deve-se criar um par de adaptadores, um de origem e um de destino, respectivamente *RASourceAdapter* e *RATargetAdapter.* O adaptador de origem deve implementar a interface de negócio e manter uma referência remota para o adaptador de destino. Enquanto o adaptador de destino deve implementar a interface remota e manter uma referência local para a implementação padrão da interface de negócio.

- **Implementação da Comunicação via APDU**

A própria comunicação nativa de *Java Card*, APDU, pode ser encapsulada em um par de adaptadores, nos quais o adaptador de origem faz a criação e o envio de comandos APDU, a partir de cada chamada de um método de negócio, enquanto o adaptador de destino converte cada comando APDU em uma chamada ao respectivo método de negócio local. O adaptador destino pode, ainda, ser auxiliado pelo despachante de comandos para realizar essa distribuição de comandos APDU.

- **Implementação da Comunicação entre *Applets***

O mecanismo de *Java Card* RMI permite que cada *applet* seja tratado como uma aplicação servidora que pode ser invocada por uma aplicação cliente disponível no dispositivo no qual o *smart card* está introduzido. No entanto, a plataforma *Java Card* fornece outro mecanismo que permite a comunicação entre *applets* no mesmo *smart card* de forma segura, chamada de compartilhamento de objetos (*Object Sharing*). Nesse mecanismo, o *applet* deve implementar um método chamado *getShareableInterfaceObject(),* que deve retornar um objeto que implemente a interface *Shareable*.

A própria interface remota, *RemoteInterface,* pode estender a interface *Shareable*, o que não implica qualquer introdução de método extra, mas apenas especifica um supertipo de objetos que podem ser compartilhados entre *applets*. O adaptador remoto, *RATargetAdapter*, se mantém como implementador da interface remota, apenas com a característica adicional de poder ser compartilhado com outros *applets*. Dessa forma, o *applet* concreto pode retornar o *RATargetAdapter* na chamada do método *getShareableInterfaceObject()*. O *applet* que faz essa chamada tem, então, uma referência ao adaptador remoto que, por sua vez, dispõe de uma referência ao objeto de negócio. Assim, o *applet* requisitante pode se comunicar como objeto de negócio de outro *applet,* em um contexto de execução diferente do seu, de forma análoga à que uma aplicação cliente o faz.

4.4 Conclusão

Inicialmente neste capítulo é apresentada uma conceituação básica de *frameworks*, incluindo sua definição, bem com conceitos como *frozen spots* e *hot spots*. Além disso, são apresentadas duas classificações para os *frameworks:* uma que diz respeito ao escopo de uso, e outra, ao seu modelo de extensão, as quais possibilitam classificar o *framework* apresentado neste capítulo, JCFoundation, como caixa branca de infra-estrutura. Com base nesses fundamentos, torna-se possível trabalhar o *framework* proposto neste trabalho que apresenta três grandes módulos. A construção do *framework* se fundamenta na sua composição, a partir de padrões de projeto, conforme técnica apresentada por Johnson (1992). Dessa maneira, os módulos do JCFoundation: de estruturação da troca de comandos entre a aplicação Java no *Smart Card* e o receptor, de isolamento da camada de persistência da camada de negócio da aplicação e de encapsulamento da camada de comunicação via *Java Card* RMI, foram definidos estendendo-se três dos padrões de projeto de Gamma (1995).

5 Estudo de Casos

5.1 Aplicação Exemplo *Wallet*

O código apresentado nesta seção refatora (FOWLER et al., 1999) a aplicação de exemplo apresentado em Chen (2000). Trata-se de uma aplicação simples de carteira eletrônica com operações básicas de crédito e débito. A alteração se refere apenas à aplicação do JCFoundation, mantendo o máximo do código original. O módulo do JCFoundation apresentado nesta seção procura encapsular os tratamento de comandos APDU e estruturar as operações de negócio em objetos específicos.

Conforme apresentado na listagem 1, a versão original da aplicação apresenta uma única classe que trata tanto de conceitos da plataforma *Java Card* quanto de regras de negócio. Seu principal método *process()* apresenta basicamente um tratamento do tipo de comando recebido pelo *Applet Java Card* e o direcionamento dele para o respectivo método de tratamento. Em cada um desses métodos, por exemplo *credit()*, é feita toda a leitura dos parâmetros dos *buffers* do *Smart Card*, em seguida, o tratamento da regra de negócio, por exemplo, creditar valor no saldo da carteira e, por fim, a formatação do *status* de retorno para o *Smart Card*, por meio dos seus *buffers* de saída.

```
public class WalletApplet extends Applet {
         final static byte Wallet_CLA =(byte)0xB0;
         ...
         OwnerPIN pin;
         short balance;
         public void process(APDU apdu) {
            switch (buffer[ISO7816.OFFSET_INS]) {
                  case GET_BALANCE: getBalance(apdu); return;
                  case DEBIT: debit(apdu); return;
                  case CREDIT: credit(apdu); return;
                  case VERIFY: verify(apdu); return;
                  default:ISOException.throwIt(SW_INS_NOT_SUPPORT);
            }
         }
         private void credit(APDU apdu) {
            ...
            balance = (short)(balance + creditAmount);
         }
         private void debit(APDU apdu) {
            ...
            balance = (short) (balance - debitAmount);
         }
         private void getBalance(APDU apdu) {
            ...
         }
         private void verify(APDU apdu) {
            ...
         }
}
```

Listagem 1 - Aplicação Exemplo *Wallet*

A primeira modificação para adequar a aplicação exemplo ao JCFoundation consiste na criação de um objeto de negócio, *Wallet*, isolando as regras de negócio em um único componente. Esse objeto faz o papel do *Receiver* no módulo de estruturação de comandos entre a aplicação e o leitor de cartões.

```
public class Wallet {
         ...
         public void credit(short amount){
            balance = (short)(balance + amount);
         }
         public void debit(short amount){
            balance = (short)(balance - amount);
         }
         public short getBalance() {
            return balance;
         }
         ...
}
```

Listagem 2 - Fragmento de Código: *Wallet.java*

O JCFoundation fornece uma subclasse de *Applet Java Card* que armazena uma referência para o despachante abstrato (também fornecido pelo JCFoundation) e define o método *process(),* delegando o processamento deste para o despachante. O *Applet* concreto da aplicação deve estendê-la e manter uma referência para o objeto de negócio, *Wallet*. Além disso, ele define o despachante concreto adequado, *WalletAPDUDispatcher*.

```
public class WalletApplet extends APDUApplet  {
          ...
          private Wallet wallet;
          private WalletApplet (byte[] bAr,short bOff,byte bL){
             super(new WalletAPDUDispatcher(new Wallet(bAr,bOff,bL)));
             this.wallet = wallet;
          }
          public static void install(byte[] bAr, short bOff, byte bL){
             WalletApplet app = new WalletApplet(bAr, bOff, bL);
             walletApplet.register();
          }
          public boolean select() {...}
          public void deselect() {...}
          ...
}
```

Listagem 3 - Fragmento de Código: ***WalletApplet.java***

O despachante concreto deve estender o despachante abstrato, o que tem como função principal localizar o comando associado à instrução APDU e fornecer algumas facilidades como o tratamento do comando de seleção e validação da categoria do comando. O despachante concreto deve complementar o comportamento da sua superclasse abstrata, definindo o mapeamento entre os comandos concretos e as instruções APDU. Além disso, ele implementa a validação da categoria dos comandos da aplicação, por meio do método *checkCommandClass().*

```
public class WalletAPDUDispatcher extends APDUDispatcher {
          ...
          public WalletAPDUDispatcher(Wallet wallet)   {
             apduCommandsInstructions = new byte[]{
                    (byte) 0x20,
                    (byte) 0x30,
                    (byte) 0x40,
                    (byte) 0x50,;
          apduCommands = new APDUCommand[]{
             new VerifyCommand(wallet),
             new CreditCommand(wallet),
             new DebitCommand(wallet),
             new GetBalanceCommand(wallet)};
          }
          ...
}
```

Listagem 4 - Fragmento de Código: *WalletAPDUDispatcher.java*

O JCFoundation especifica uma interface dos comandos que define apenas um método, *execute()*. Os comandos concretos da aplicação devem implementar essa interface, com, por exemplo, *CreditCommand*, que faz o processamento de leitura do *buffer* de *Java Card*, o qual delega o método operação de negócio que credita o valor na carteira eletrônica para a classe *Wallet* e, por fim, realiza a escrita do *status* de retorno no *buffer*. Além desse, foram criados mais três comandos concretos *DebitCommand, GetBalanceCommand*, e *VerifyCommand* para as operações de débito, consulta de saldo e validação de PIN, respectivamente.

```
public class CreditCommand implements APDUCommand {
          ...
          private Wallet wallet;
          public void execute(APDU apdu) {
             byte[] buffer = apdu.getBuffer();
             byte numBytes = buffer[ISO7816.OFFSET_LC];
             byte byteRead = (byte)(apdu.setIncomingAndReceive());
             if ( !getWallet().isTransactionAmountOk(creditAmount)){
                    ISOException.throwIt(SW_INVALID_TRANS_AMOUNT);
             }
             getWallet().credit(creditAmount);
             ...
          }
          ...
}
```

Listagem 5 - Fragmento de Código: ***CreditCommand.java***

5.2 Aplicação Exemplo *HelloWorldEcho*

Tal qual na seção anterior, o código apresentado nesta seção refatora (FOWLER et al., 1999) se dedica à aplicação de exemplo apresentado em JCAPI (2002). A aplicação de exemplo é bastante simples, visto que consiste apenas em receber uma mensagem com parâmetro e retornar essa mesma mensagem para o cliente. A alteração introduzida nesse exemplo possibilita o armazenamento da mensagem recebida na memória transiente do *Smart Card.* Dessa maneira, é aplicado o módulo do JCFoundation que isola a camada de persistência da camada de negócio da aplicação. A primeira modificação na lógica da aplicação é a criação de um objeto de negócio, *EchoMessage*, isolando as regras de negócio em um único componente, que faz o papel da unidade de dados concreto no módulo do JCFoundation observado nesta seção.

```
public class HelloWorld extends Applet{
         private byte[] echoBytes;
         private static final short LENGTH_ECHO_BYTES = 256;
         protected HelloWorld(){
            echoBytes = new byte[LENGTH_ECHO_BYTES];
            register();
         }
         public void process(APDU apdu) {
            byte buffer[] = apdu.getBuffer();
            short bytesRead = apdu.setIncomingAndReceive();
            short echoOffset = (short)0;
            while ( bytesRead > 0 ) {
                  Util.arrayCopyNonAtomic(buffer,OFFSET_CDATA,
                              echoBytes, echoOffset, bytesRead);
                  echoOffset += bytesRead;
                  bytesRead = apdu.receiveBytes(ISO7816.OFF_CDATA);
            }
            apdu.setOutgoing();
            apdu.setOutgoingLength( (short) (echoOffset + 5) );
            apdu.sendBytes( (short)0, (short) 5);
            apdu.sendBytesLong( echoBytes, (short) 0, echoOffset );
         }
}
```

Listagem 6 - Aplicação Exemplo *HelloWorldEcho*

O JCFoundation fornece uma superclasse, chamada *DatastoreApplet*, para os *Applets* de aplicação que têm uma camada de negócio simplificada e seu armazenamento persistente restrito a um único repositório. Esse é justamente o caso do exemplo apresentado nesta seção. Dessa maneira, o *Applet* concreto da aplicação, *HelloWorldApplet*, estende essa superclasse e mantém uma referência para a chave do único objeto que se faz necessário armazenar no repositório. Além disso, o *applet* concreto define o repositório concreto adequado, que, no caso desta aplicação, é um objeto transiente com tempo de vida restrito ao período de atividade da aplicação.

```
public class HelloWorldApplet extends DatastoreApplet{
         public static void install(byte[] bA, short bOf, byte bL){
            IDatastoreRepository rep = null;
            rep = new ClearOnDeselectTransientRepository(1);
            new HelloWorldApplet(rep);
         }
         public void readMessage(APDU apdu)    {
         ...
            EchoMessage hellomsg = new EchoMessage(echoBytes);
            datastore.add(hellomsg);
            key=hellomsg.getKey();
         }
         public void echoMessage(APDU apdu)    {
            EchoMessage msg=(EchoMessage)datastore.find(key);
         ...
         }
}
```

Listagem 7 - Fragmento de Código: *HelloWorldApplet.java*

O JCFoundation especifica uma interface de unidade de dados que define as operações sobre a chave de identificação do objeto, *isKeyEqual()* e *getKey(),* e um método para atualizar os seus dados, *updateValues().* A classe que encapsula a informação a ser persistida na aplicação deve implementar essa interface, definindo uma estrutura de dados adequada e uma chave de identificação. No exemplo analisado nesta seção, essa unidade de dados concreta utiliza, como estrutura de dados para armazenar a mensagem, um *array* de *bytes* e, como chave de identificação, uma classe interna *HelloMessageKey*, que considera o identificador do objeto o seu *hash*.

```
public class EchoMessage implements IDataUnit {
         private byte[] echoBytes;
         private HelloMessageKey key;
         public boolean isKeyEqual(Object key) {
            return this.key.equals(key);
         }
         public void updateValues(IDataUnit newValues) {
            this.setEchoBytes(newValues.getEchoBytes());
            int newKey=(newValues.getKey()).getHash();
            key.setHash(newKey);
         }
         public Object getKey() {
            return key;
         }
         public class HelloMessageKey{
            private int hash;
            public HelloMessageKey(int hash){
                  this.hash=hash;
            }
         }
}
```

Listagem 8 - Fragmento de Código: *EchoMessage.java*

O JCFoundation fornece uma classe chamada *Datastore* que define as operações de armazenamento dos dados (adição, remoção, localização e atualização), além de manter uma referência para a interface do repositório de dados. Essa interface, por sua vez, define apenas as operações básicas a serem implementadas pela estrutura de dados: *append(), remove()* e *find()*. A aplicação precisa, então, implementar sua própria estrutura de dados, ou utilizar alguma fornecida pelo JCFoundation dentro de uma das hierarquias de repositórios transientes, *TransientRepository*, ou persistentes, *PersistentRepository*. Na aplicação analisada nesta seção, é utilizado o *ClearOnDeselectTransientRepository,* que trabalha com o tempo de vida dos dados do repositório limitando o período de atividade da aplicação, e que usa como estrutura de dados um *array* de objetos transientes fornecidos por *Java Card*, *JCSystem.makeTransientObjectArray(len, CLEAR_ON_DESELECT*). Opcionalmente pode ser utilizado outro repositório transiente com tempo de vida diferente, *ClearOnResetTransientRepository*, o que pode restringir o armazenamento ao período em que

o cartão esteja inserido no leitor. Caso se opte por armazenamento persistente, deve ser usada a implementação base da interface *PersistentRepository* fornecida pelo JCFoundation, *ObjectArrayPersistentRepository*, que persiste os dados utilizando uma estrutura de dados composta por um *array* de objetos. Como alternativa a essa implementação padrão, pode ser criada uma classe que utilize uma estrutura de dados diferente de *array* de objetos.

5.3 Aplicação Exemplo *Purse*

Seguindo os mesmos preceitos das seções anteriores, o código apresentado nesta seção refatora (FOWLER et al., 1999) se refere à aplicação RMI de exemplo apresentado em JCAPI (2002). A aplicação exemplo desta seção é análoga à aplicação apresentada na primeira seção deste capítulo, apenas com a diferença de esta versão expor a funcionalidade de carteira eletrônica do protocolo JCRMI e não mais por meio de APDU diretamente. O módulo do JCFoundation aplicado nesta seção procura encapsular essa camada de comunicação de JCRMI em objetos específicos.

O exemplo original é composto por três classes e uma interface. A interface define os métodos de negócio a serem expostos pela aplicação no *Smart Card*. Por exigência do protocolo JCRMI, essa interface deve estender da interface *Remote* e todos seus métodos devem levantar exceções *RemoteException.*

```
public interface Purse extends Remote{
         ...
         public short getBalance() throws RemoteException;
         public void debit(short m)
                             throws RemoteException, UserException;
         public void credit(short m)
                             throws RemoteException, UserException;
         public void setAccountNumber(byte[] number)
                             throws RemoteException, UserException;
         public byte[] getAccountNumber() throws RemoteException;
}
```

Listagem 9 - Interface Remota de Negócio da Aplicação Exemplo

Essa interface é implementada, então, por uma classe que, além de incorporar todas as operações de negócio, precisa estender de *CardRemoteObject* também por exigência de JCRMI.

```
Public class PurseImpl extends CardRemoteObject implements Purse {
         private short balance = 0;
         private byte[] number;
         public void setAccountNumber(byte[] number)
                        throws RemoteException, UserException {... }
         public void credit(short m)
                        throws RemoteException, UserException {
            if(m<=0) UserException.throwIt(BAD_ARGUMENT);
            if((short)(balance+m) > MAX_AMOUNT)
                  UserException.throwIt(OVERFLOW);
            balance +=m;
         }
         public void debit(short m)
                        throws RemoteException, UserException {... }
         public byte[] getAccountNumber() throws RemoteException {
            return number;
         }
         public short getBalance() throws RemoteException {
            return balance;
         }
}
```

Listagem 10 - Implementação da Interface de Negócio da Aplicação Exemplo

Nesse exemplo, o *Applet Java Card* é bastante simplificado. Ele consiste basicamente do registro da implementação da interface no serviço de despacho, juntamente com o serviço de comunicação de JCRMI da própria plataforma *Java Card*.

```
public class PurseApplet extends Applet {
         private Dispatcher disp;
         public PurseApplet() {
            Remote purse = new PurseImpl();
            disp = new Dispatcher( (short) 1);
            RemoteService serv = new RMIService(purse);
            disp.addService(serv, Dispatcher.PROCESS_COMMAND);
            register();
         }
         public static void install(byte[] aid, short s, byte b) {
            new PurseApplet();
         }
         public void process(APDU apdu) throws ISOException {
            disp.process(apdu);
         }
}
```

Listagem 11 - *Applet* da Aplicação Exemplo

Uma vez exposta a interface de negócio por meio do protocolo JCRMI, a aplicação cliente no leitor do cartão pode acessá-la de maneira análoga à que é feita para qualquer objeto Java remoto por meio de RMI. A única especificidade consiste no método utilizado para obter a referência remota da interface de negócio. No exemplo apresentado nesta seção, é utilizada a API OCF (*Open Card Framework)*. Uma vez obtida esta referência remota, *Purse*, é possível operá-la de forma transparente.

```
public class PurseClient {
         public static void main(String[] a)throws RemoteException{
            try {
                  SmartCard.start();
                  CardRequest cr = new CardRequest(
                    CardRequest.NEWCARD,null,OCFCardAccessor.class);
                  SmartCard myCard = SmartCard.waitForCard( cr );
                  CardAccessor cs=(CardAccessor)
                    myCard.getCardService(OCFCardAccessor.class);
                  JavaCardRMIConnect con=new JavaCardRMIConnect(cs);
                  Con.selectApplet( RMI_DEMO_AID );
                  Purse myPurse = (Purse) con.getInitialReference();
                  short balance = myPurse.getBalance();
            }catch (Exception e){
                  System.out.println(e);
            }finally {
                  SmartCard.shutdown();
            }
         }
}
```

Listagem 12 - Classe Cliente da Aplicação Exemplo

Ao utilizar o JCFoundation no exemplo para *Java Card* RMI, no *Applet,* é preciso apenas criar o adaptador destino. Ele, por sua vez, é responsável por se registrar no serviço de despacho de JCRMI.

```
public class PurseApplet extends Applet {
         private Dispatcher disp;
         public PurseApplet() {
            Dispatcher disp = new Dispatcher( (short) 1);
            new PurseTargetAdapter(disp);
            register();
         }
         public static void install(byte[] aid, short s, byte b) {
            new PurseApplet();
         }
         public void process(APDU apdu) throws ISOException {
            disp.process(apdu);
         }
}
```

Listagem 13 - Fragmento de Código: *PurseApplet.java*.

O adaptador destino implementa a interface de negócio remota, *RemotePurse*, e estende a classe *CardRemoteObject*, conforme exigido pela *Java Card* RMI. Uma das suas principais responsabilidades é a de ser registrado no serviço de nomes. Para tal, o *PurseTargetAdapter* no seu construtor cria um *RMIService* e o adiciona ao *Dispatcher*. Pelo fato de essa classe implementar a interface de negócio remota, ela disponibiliza os métodos de negócio, *credit()*, *debit(), set/getAccountNumber() e getBalance()*. A implementação desses métodos é realizada por meio da delegação do processamento deles para seus respectivos métodos do objeto de negócio, Purse, cuja referência é mantida pelo adaptador. Nota-se, ainda, que as exceções de negócio, *PurseException*, são convertidas para exceções trafegáveis na camada de distribuição, o que, no caso de *Java Card* RMI, corresponde à *RemoteException* ou à *UserException*. A primeira está relacionada a problemas de comunicação, enquanto a última representa um erro genérico proveniente de uma exceção de negócio.

```
public class PurseTargetAdapter extends CardRemoteObject
                                          implements RemotePurse {
         private Purse purse;
         public PurseTargetAdapter(Dispatcher disp){
            purse = new PurseImpl();
            RemoteService serv = new RMIService(this);
            disp.addService(serv, Dispatcher.PROCESS_COMMAND);
         }
         public void credit(short m)... {
            purse.credit(m);
         }
         public void debit(short m)...{
            purse.debit(m);
         }
}
```

Listagem 14 - Fragmento de Código: *PurseTargetAdapter.java*

A interface de negócio define algumas constantes referentes ao domínio da aplicação e aos métodos de negócio com exceções restritas ao contexto da aplicação, ou seja, nessa interface não é inserido qualquer conceito referente à camada de distribuição. Essa interface é implementada pelo objeto de negócio, *PurseImpl,* e pelo adaptador de origem, *PurseSourceAdapter.* Além disto, o adaptador de destino, *PurseTargetAdapter*, mantém uma referência ao objeto de negócio por meio dessa interface.

```
public interface Purse {
         ...
         public short getBalance()  ;
         public void debit(short m) throws PurseException;
         public void credit(short m) throws PurseException;
         public void setAccountNumber(byte[] number)
                                          throws PurseException;
         public byte[] getAccountNumber() ;
}
```

Listagem 15 - Fragmento de Código: *Purse.java*

A interface de negócio remota, *RemotePurse*, define os mesmos métodos que sua interface de negócio equivalente, *Purse*, com a diferença de ela atender aos requisitos exigidos pela camada de distribuição: estender a interface *Remote* e seus métodos levantarem a exceção *RemoteException.*

```
public interface RemotePurse extends Remote{
         public short getBalance()  throws RemoteException;
         public void debit(short m)
                        throws UserException,RemoteException;
         public void credit(short m)
                        throws UserException,RemoteException;
         public void setAccountNumber(byte[] number)
                        throws UserException,RemoteException;
         public byte[] getAccountNumber() throws RemoteException;
}
```

Listagem 16 - Fragmento de Código: *RemotePurse.java*

A classe de negócio basicamente implementa a interface de negócio. Essa classe não incorpora qualquer conceito da camada de distribuição, por exemplo, seus métodos levantam exceções específicas de negócio, *PurseException*, e não exceções remotas definidas por *Java Card* RMI. Essa classe utiliza, ainda, como estrutura de dados interna para suportar os métodos de negócio, um tipo *short* para armazenar o saldo da carteira e um *array* de *bytes* para registrar o número da conta.

```
public class PurseImpl implements Purse {
         private short balance = 0;
         private byte[] number;
         public void setAccountNumber(byte[] number) throws ... {
            this.number=number;
         }
         public void credit(short m) throws PurseException {
            if(balance+m>MAX_AMOUNT) PurseException.throwIt(OVERFLOW);
            balance +=m;
         }
         public void debit(short m) throws PurseException {
            if((short)(balance-m)< 0)
                         PurseException.throwIt(UNDERFLOW);
            balance -=m;
         }
}
```

Listagem 17 - Fragmento de Código: *PurseImpl.java*

A *checked exception*, *PurseException,* define uma exceção de negócio e é utilizada pela interface de negócio, *Purse*, e pelo objeto de negócio que a implementa, *PurseImpl*.

```
public class PurseException extends Exception{
         private short reason;
         private PurseException exception;
         public static void throwIt(Object key)
                                          throws PurseException{
            if(exception!=null){
                  exception = new DataUnitNotFoundException(key);
            }
            throw (DataUnitNotFoundException)exception;
         }
}
```

Listagem 18 - Fragmento de Código: *PurseException.java*

A classe cliente, *PurseClient,* comunica-se com o objeto de negócio remoto como se ele estivesse disponível localmente. Para tal, ele se comunica, na verdade, com o adaptador de origem, mantido por meio de uma referência para a interface de negócio, *Purse*. Como pode ser visto, a classe cliente basicamente instancia uma implementação da interface de negócio. Nesse caso, o adaptador de origem executa métodos de negócio sobre ele (no exemplo a seguir, *getBalance()* e *debit()* são executados remotamente). Inclusive as exceções manipuladas por esse cliente pertencem exclusivamente ao domínio da aplicação, *PurseException* e não às exceções remotas da camada de distribuição.

```
public class PurseClient {
         public static void main(String[] argv)
                                   throws RemoteException{
            Purse myPurse = new PurseSourceAdapter();
            try {
                  short balance = myPurse.getBalance();
                  myPurse.debit(balance);
                  balance = myPurse.getBalance();
            }catch (PurseException pe) {
                  // tratamento de negócio
                  pe.printStackTrace();
            }
         }
}
```

Listagem 19 - Fragmento de Código: *PurseClient.java*

O adaptador de origem, *PurseSourceAdapter*, encapsula toda a complexidade de acesso ao objeto remoto por meio da OCF API e implementa a interface de negócio, *Purse*. Essa implementação é realizada delegando as chamadas aos métodos de negócio para seus métodos equivalentes no adaptador destino, *PurseTargetAdapter*, cuja referência é mantida pelo adaptador de origem. Como o adaptador destino é um objeto remoto, a principal responsabilidade do construtor do adaptador de origem é se conectar ao de destino. Para tal, o *PurseSourceAdapter* utiliza os recursos da OCF API: *SmartCard, CardRequest*, *CardAccessor* e *JavaCardRMIConnect*. Este último é responsável por selecionar o *applet* e retornar uma referência para o objeto remoto, no presente caso, o adaptador destino. O adaptador de origem é responsável, ainda, por converter exceções remotas em exceções de

negócio. No exemplo a seguir, é ele quem converte a exceção de negócio genérica suportada por *Java Card*, *UserException*, em uma exceção específica do domínio da aplicação, *PurseException*. Além disso, a exceção de comunicação, *RemoteException*, é convertida em uma exceção não checada, *RuntimeException*, por se tratar de um erro de infra-estrutura não esperado.

```
public class PurseSourceAdapter implements Purse {
         private PurseTargetAdapter purse;
         public PurseSourceAdapter(){connect();}
         private void connect(){
            SmartCard.start();
            CardRequest cr = new CardRequest(OCFCardAccessor.class);
            SmartCard myCard = SmartCard.waitForCard( cr );
            CardAccessor cs = (CardAccessor)
            myCard.getCardService(OCFCardAccessor.class, true );
            JavaCardRMIConnect con=new JavaCardRMIConnect(cs);
            con.selectApplet( RMI_DEMO_AID );
            purse=(PurseTargetAdapter)con.getInitialReference();
         }
         public void credit(short m) throws PurseException{
            try { purse.credit(m);
            }catch (UserException ue) {
                  PurseException.throwIt(ue.getReason());
            }catch(RemoteException e){
                  throw new RuntimeException(e);}
         }
         public void debit(short m) throws PurseException {...}
         public byte[] getAccountNumber() {...}
         public short getBalance() {...}
         public void setAccountNumber(byte[] num)... {...}
}
```

Listagem 20 - Fragmento de Código: *PurseSourceAdapter.java*

5.4 Conclusão

Neste capítulo, são apresentadas três aplicações de exemplo em sua versão original disponível na literatura, então, para cada um destes é aplicado um módulo do JCFoundation. A primeira aplicação de exemplo consiste de uma carteira eletrônica que centraliza as operações de crédito e de débito em uma única classe. Após o refatoramento desta aplicação, com base no *framework,* cada uma das instruções APDU é encapsulada em uma classe específica, o que torna a aplicação mais modular e extensível. Dessa forma, é introduzir novas instruções APDU facilmente. A segunda aplicação exemplo trata de uma aplicação simples que ecoa uma mensagem de boas vindas. Uma vez refatorada a aplicação, essa mensagem é armazenada em repositório transiente, isolando a camada de persistência da camada de negócio. Dessa forma, é possível que estrutura de dados seja alterada independentemente da evolução da camada de negócio. A última aplicação de exemplo é uma carteira eletrônica similar à primeira aplicação, mas com a característica adicional de suportar comunicação remota por meio do protocolo *Java Card* RMI. Ao se empregar o JCFoundation sobre essa aplicação, todas as questões específicas exigidas pelo protocolo de comunicação, como, por exemplo, estender a interface *Remote*, são encapsuladas em adaptadores próprios para tal. Até mesmo o processo de localização do objeto remoto por parte do cliente é encapsulado em adaptador de origem. Com isso, é possível trocar facilmente todo a camada de comunicação sem qualquer impacto na camada de negócio.

6 CONCLUSÕES

No segundo capítulo, após um capítulo introdutório, são apresentados alguns fundamentos essenciais ao entendimento da principal contribuição deste trabalho, o JCFoundation, um *framework* para desenvolvimento de aplicações Java que utiliza a API *Java Card* em *Smart Cards* microprocessados. Esses fundamentos compreendem uma introdução sobre tecnologia de *Smart Cards,* bem como sua aplicabilidade, e os conceitos da plataforma *Java Card*. O capítulo seguinte corresponde à maior contribuição deste trabalho, no qual foi proposto o JCFoundation. Com base na extensão de padrões de projeto, já amplamente validados, é que são construídos os módulos do *framework*, de estruturação da troca de comandos entre a aplicação Java no *Smart Card* e o Receptor, de isolamento da camada de persistência da camada de negócio da aplicação e de encapsulamento da camada de comunicação via *Java Card* RMI. No penúltimo capítulo, é realizado um estudo de caso do JCFoundation em aplicações de exemplo já disponíveis na literatura.

O primeiro módulo permite estruturar a troca de comandos entre a aplicação Java no *Smart Card* e o Receptor. Dessa forma, o código da aplicação se torna mais modular e extensível. Isso se deve ao fato de cada instrução APDU ficar encapsulada em um comando concreto do *framework*, possibilitando ainda que novas instruções sejam introduzidas facilmente.

Já o segundo módulo, de isolamento da camada de persistência da camada de negócio da aplicação, permite que a lógica de negócio da aplicação seja isolada da tecnologia de persistência utilizada pela camada de acesso a dados. O JCFoundation disponibiliza ainda um repositório nos dois modelos de armazenamento disponíveis em *Java Card*: persistente e transiente. Adicionalmente, ele possibilita que os repositórios sejam implementados com as mais diversas estruturas de dados. Por fim, ele suporta opcionalmente o controle de transação, o que permite que mudanças na tecnologia de acesso aos dados sejam centralizadas nessa camada, minimizando ou eliminando o impacto nas demais classes da aplicação.

O último módulo de encapsulamento da camada de comunicação via *Java Card* RMI possibilita que as camadas de interface gráfica e de negócio se comuniquem de forma transparente da tecnologia de distribuição/comunicação, o que ocorre por meio de dois adaptadores, o de origem e o de destino. O adaptador de origem isola o processo de

localização do objeto remoto, nesse caso o adaptador destino. Este, por sua vez, encapsula todo o processo de registro do objeto remoto no serviço de nomes. Dessa forma, as alterações especificadas pela tecnologia de comunicação não são incorporadas nos objetos de negócio, mantendo-os na sua forma original, o que possibilita que a camada de negócio da aplicação seja mais reusável, pois os elementos desta camada podem ser reaproveitado em outras aplicações, inclusive fora do contexto de *smart cards*.

Como conseqüência negativa, o JCFoundation exige um esforço adicional de codificação e demanda um consumo extra de memória do *smart card*, mesmo que reduzido visto que o JCFoundation exige apenas 12K *bytes*. A conseqüência do esforço adicional pode ser minimizada com uso de ferramentas de desenvolvimento que possibilitem a aplicação automática do JCFoundation. E o consumo extra de memória tende a ficar cada vez menos significativo, visto que a tecnologia de *chips* dos cartões tem evoluído de forma a produzir *smart cards* com capacidade cada vez maior. Atualmente já existem *smart cards* de 1Mb fabricados pela Sharp.

O estudo de caso para validar o JCFoundation toma como base aplicações de exemplo fornecidas pela literatura ou pelo próprio fabricante da tecnologia, Sun Microsystems: *Wallet*, *HelloWorldEcho* e *Purse*. Sobre essas aplicações, são aplicados os módulos do *framework*. O resultado obtido, para todas as três comparações entre o código original e sua respectiva versão alterada com aplicação do JCFoundation, é similar. Os códigos originais apresentam toda complexidade e instabilidade da aplicação na camada de negócio, enquanto que os códigos nos quais foi aplicado o JCFoundation apresentam instabilidade e nível de abstração decrescente das camadas mais altas da aplicação para as camadas mais baixas. Isso deve em função das camadas superiores centralizarem as classes referentes a conceitos específicos do protocolo de comunicação ou da interface gráfica, enquanto as inferiores enfocam em classes e interfaces de negócio. Dessa forma, a hierarquia de camadas pode ser utilizada de acordo com o seu propósito, na qual as camadas inferiores podem ter um maior reuso e as superiores tendem a ser alteradas com freqüência, podendo ser simplesmente descartadas.

6.1 Trabalhos Futuros

Em Grimaud e Vandewalle (2003), são apresentadas várias questões a serem desenvolvidas

no universo de *Java Card*. Entre elas, consta a estruturação de aplicações, que cada vez se tornam mais complexas com suporte da plataforma a um número crescente de componentes Java. O cerne da proposta de trabalho está alinhado justamente com essa linha de pesquisa de estruturação de aplicações *Java Card*. Apesar de contribuir para essa estruturação, o *framework* apresentado aqui não cobre toda a demanda de projetos de aplicações para essa plataforma. Dessa maneira, existe um espaço vasto para publicações sobre essa questão. O lançamento da nova versão de *Java Card*, a 3.0, expande a capacidade da plataforma em várias das suas funcionalidades. Mais especificamente, conforme (JAVA CARD FORUM, 2006), apresenta os seguintes benefícios: aumento de velocidade (SQUAWK, 2002); melhoria da segurança; suporte a *strings*, *multi-threading* e XML; maior flexibilidade entre armazenamento persistente e transiente; e comunicação por meio de outros protocolos, além do APDU, como TCP/IP.

A facilitação do uso do armazenamento persistente ou transiente deve implicar a alteração ou a criação de repositórios, estendendo o módulo de isolamento da camada de persistência da camada de negócio da aplicação. Já o suporte a protocolos, como TCP/IP, demanda a construção de novos pares de adaptadores que podem ser construídos para o módulo de estruturação da troca de comandos entre a aplicação e o receptor.

Outros protocolos são ainda apresentados por outros autores como Chan et al. (2002) que propõem uma camada de distribuição de *Smart Card* compatível com EJB – Enterprise Java Beans (DEMICHIEL, 2003). Existem, ainda, outras formas de comunicação com o *Smart Card,* como, por exemplo, a utilização de um computador pessoal como cliente. Para tal, existe uma especificação denominada PC/SC (BULL et al., 1996), implementada por Java, por meio da API denominada JPC/SC. Dessa forma, é possível implementar um adaptador de origem encapsulando o código JPC/SC. Recentemente foi introduzida, no lançamento de J2SE, versão 6, uma comunicação com *Smart Card* por meio de operações de I/O Java, especificado pela API descrita na JSR-268 (ANDREAS, 2006). É possível, então, implementar um adaptador de origem, encapsulando os códigos de I/O de Java 6 para acesso à aplicação no *Smart Card*.

Além dessas evoluções previstas, é possível pressupor que adaptações ou extensões, como uma nova implementação do controlador de transação do módulo de persistência que deve ser introduzida, caso o JTA (SUSAN; VLADA, 2002) substitua o mecanismo atual de controle de

transação de *Java Card*. O número de possibilidades é imenso, mas pode-se citar, como último exemplo, a introdução do paradigma orientado a aspectos (KICZALES et al., 1997) na plataforma *Java Card*. Caso essa introdução ocorra, o JCFoundation pode ser reimplementado na forma de aspectos (KICZALES; HANNEMANN, 2002).

Além das próprias evoluções da plataforma, existem áreas da versão atual que não são contempladas pelo *framework,* como, por exemplo, criptografia e biometria. Dessa forma, novos módulos específicos dessas áreas podem ser propostos. As APIs que estendem a plataforma *Java Card*, como a *SIM Card* API (EUROPEAN TELECOMUNICATIONS STANDARDS INSTITUTE, 2004), podem ser contempladas também com módulos específicos. *SIM Card* é um *Smart Card* utilizado na telefonia celular, em que sua API estende a plataforma *Java Card* para suportar funções específicas para interagir com o aparelho telefônico ou com a rede de telefonia para, entre outras funções, fazer ligações ou enviar mensagens de texto.

Por fim, o JCFoundation pode ser incorporado a alguma ferramenta de desenvolvimento existente auxiliando na geração automática de código. Dessa forma, a ferramenta pode auxiliar na criação dos adaptadores, comandos concretos e repositórios, facilitando a adoção do *framework* no desenvolvimento de aplicações Java *Card* para *Smart Cards* microprocessados.

7 REFERÊNCIAS

ABRIAL, J. **The B-Book: assigning programs to meanings**. Cambridge: University Press, 1996.

ANDREAS, S. **JSR 268 - Java Smart Card I/O API**. [s.L.]: JCP, 2006.

ANDRONICK, J; CHETALI, B; LY, O. **Using coq to verify Java Card applet isolation properties**. Berlin: Springer Berlin, 2003.

BARTHE, G. et al. A formal executable semantics of the JavaCard platform. In: EUROPEAN SYMPOSIUM ON PROGRAMMING, 2001 **Proceedings…** v.1

BECKERT, B**. A dynamic logic for the formal verification of Java Card programs**. Java on Smart Cards: Programming and Security, 2000.

BIEBER, P. et al. Checking secure interactions of smart card applets. In: EUROPEAN SYMPOSIUM ON RESEARCH IN COMPUTER SECURITY, 6., 2000, Toulouse. **Proceedings…**

BIEBER, P. et al. The PACAP prototype: a tool for detecting Java Card illegal flow. Java on Smart Cards: programming and security. In: INTERNATIONAL WORKSHOP ON JAVA CARD, 1., 2000, Cannes. **Proceedings**… 2001

BULL, C. et al. Interoperability specification for ICCs and personal computer Systems. **PC/SC Workgroup**, 1996.

BURDY, L; REQUET, A; LANET, J. **Java Applet Correctness: a developer-oriented approach**. Berlin: Springer Berlin / Heidelberg, 2003.

CASSET, L; BURDY, L; REQUET, A**.** Formal development of an embedded verifier for Java Card byte code. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, 2002. **Proceedings…**

CHAN, A. et al. Enabling distributed Corba access to smart card applications. **IEEE Internet Computing**, v.6, n.3, 2002.

CHAN, A. Cookies on-the-move: managing cookies on a smart card. In: ACM SYMPOSIUM ON APPLIED COMPUTING, 2004. **Proceedings…**

CHAUMETTE, S. et al. An environment for attack and test on Java Card. In: INTERNATIONAL CONFERENCE ON COMPUTER, COMMUNICATION AND CONTROL, 2003. **Proceedings…** V.3, 2003.

CHEN, Z. **Java Card technology for smart cards: architecture and programmer's**. Boston: Addison-Wesley Professional, 2000.

DEMICHIEL L. **Enterprise JavaBeans specification.** Version 2.1. Palo Alto: Sun Microsystems, 2003.

e-PING: padrões de interoperabilidade de governo eletrônico. Versão 1.5. [s.L.]: Comitê Executivo de Governo Eletrônico, 2005.

EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE. **ETSI TS 143 019: Subscriber identity module application programming interface (SIM API) for Java card**. Version 6.0.0. 2004.

FAYAD, M.; SCHMIDT, D.C. Object-oriented application frameworks. **Communications of the ACM**, v.40, n.10, p.32-38, 1997.

FAYAD, M.E.; SCHMIDT, D.C.; JOHNSON, R.E. **Implementing application frameworks: object-oriented frameworks at work**. New York: John Wiley & Sons, 1999.

FODOR, 0.; HASSLER, V. Javacard and opencard framework: a tutorial. In: IEEE INTERNATIONAL CONFERENCE, 7., 1999. **Proceedings…**

FONTOURA, M.; PREE, W.; RUMPE, B. **The UML profile for framework architectures**. [s.L.]: Addison-Wesley, 2001.

FOWLER, M. et al. **Refactoring: improving the design of existing code**. Boston: Addison-Wesley, 1999.

FOWLER, M. **Analysis patterns: reusable object models**. Boston: Addison-Wesley, 1996.

FREUND, S.; MITCHELL, J**.** A formal framework for the Java bytecode language and verifier. ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, 14., 1999. **Proceedings…**

FROST & SULIVAN. **World Smart Card Markets. Market Engineering Research**. 2004. Disponível em: <http://www.frost.com/prod/servlet/report-homepage.pag?repid=F266-01-00-00-00&ctxst=FcmCtx1&ctxht=FcmCtx2&ctxhl=FcmCtx3&ctxixpLink=FcmCtx3&ctxixpLabel=FcmCtx4>

GAMMA, E. et al. **Design patterns: elements of reusable object oriented software**. Boston: Addison Wesley, 1994.

GHOSH, A. Security risks of Java cards. **CardTech/SecurTech**. v.1, 1998.

GOSLING, J.; JOY, B.; STEELE, G. **The Java language specification**. Boston: Longman Publishing, 1996.

GLOBAL PLATFORM ASSOCIATION. **Global platform card specification**. Versão 2.1, 2001.

GRIMAUD, G; LANET J; VANDEWALLE J**. FACADE: a typed intermediate language dedicated to smart cards**. Springer Berlin / Heidelberg, 1999.

GRIMAUD, G; VANDEWALLE, J. Introducing research issues for next Generation Java-based smart card platforms. SMART OBJECTS CONFERENCE, 2003, Grenoble. **Proceedings…**

HANSMANN, U. et al. **Smart card application development using Java**. 2. ed. New York: Springer, 2002.

HARTEL, P; MOREAU, L. Formalizing the safety of Java, the Java virtual machine, and Java card. **Source ACM Computing Surveys,** v.33, n.4, 2001.

HEINE, G. **GSM Networks: protocols, terminology, and implementation.** Boston: Artech House, 1999.

HOARE, C. An axiomatic basis for computer programming. **Communications of the ACM**, v.12, n.10, 1969.

HORSTMANN, C.; CORNELL, G. **Core Java 2.** 7. ed. [s.L.]: Prentice Hall PTR, 2004. v.1

HUBBERS, E.; POLL, E. **Reasoning about card tears and transactions in Java card**. Springer Berlin / Heidelberg, 2003.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. **ISO/IEC 7816:** Integrated circuit(s) cards with contacts. Versão 1.0, 1987.

JAVA CARD FORUM. **Java Card overview and 2.x comparison**. 2006. Disponível em: < http://www.javacardforum.org/03_documents/00_documents/JC3%20overview.pdf, Acesso em: 10 nov. 2006.

JCAPI. **Java Card 2.2 application programming interface**. Versão 2.2. Sun Microsystems, 2002.

JCPS. **Java Card platform security**. White Paper, Sun Microsystems, 2001

JCRE. **Java Card 2.2 runtime environment specification**. Versão 2.2. Sun Microsystems, 2002.

JCRMI. **RMI Client application programming interface**. Versão 2.2. Sun Microsystems, 2002.

JCVM. **Java Card 2.2 virtual machine specification**. Versão 2.2. Sun Microsystems, 2002.

JOHNSON, R.E. Documenting frameworks using patterns. In: OBJECT-ORIENTED PROGRAMMINGSYSTEMS, LANGUAGES, AND APPLICATIONS, 1992, Vancouver. **Proccedings…**

JOHNSON, R.E.; FOOTE, B. Designing reusable classes. **Journal of Object-Oriented Programming**, v,1, n.2, p.22-35, 1988.

JURJENS, J. Security analysis of crypto-based Java programs using automated theorem provers. In: IEEE INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, 21., 2006. **Proceedings…**

KICZALES, G. et al. Aspect-Oriented programming. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 1997. **Proceedings…**

KICZALES, G.; HANNEMANN J. Design pattern implementation in Java and aspectJ. In: ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 2002. **Proceedings…**

KLEIN, G.; NIPKOW, T. Verified lightweight bytecode verification. **Concurrency and Computation: Practice and Experience**, v.13, n.13, 2001.

KRISTOL, D**.** HTTP cookies: standards, privacy, and politics. **ACM Transactions on Internet Technology**, v.1, n.2, 2001.

LANET, J.; REQUET, A. Formal proof of smart card applets correctness. In: INTERNATIONAL CONFERENCE ON SMART CARD RESEARCH AND APPLICATIONS, 1998. **Proceedings…** v.1820

LARSSON, D.; MOSTOWSKI, W**. Specifying Java card API in OCL**. [s.L.]: OLC, 2003.

LEÃO, B.F. et al. The brazilian health informatics and information policy: building the consensus. **Medinfo**. Amsterdam: IOS Press, 2004.

LEAVENS, G; BAKER, A; RUBY, C. **JML: a notation for detailed design**. [s.L.]: Kluwer Academic Publishers, 1999.

LEROY, X. **On-Card Bytecode Verification for Java Card**. Springer Berlin / Heidelberg, 2001.

LEROY, X. Bytecode verification on Java smart cards. **Software: Practice and Experience**, v. 32, n. 4, 2002.

LEROY, X. Java bytecode verification: algorithms and formalizations. **Journal of Automated Reasoning**, v. 30, n. 3-4, 2003.

LINDHOLM, T.; YELLIN, F. **Java Virtual Machine Specification**. [s.L.]: Addison-Wesley, 1999

LINTHICUM, D. et al. **Enterprise Application Integration**. [s.L.]: Addison-Wesley, 1999.

LOUTREL, M.; PUJOLLE, G. A smart card for authentication in WLANs. In: IFIP/ACM LATIN AMERICA CONFERENCE ON TOWARDS A LATIN AMERICAN AGENDA FOR NETWORK RESEARCH, 2003. **Proceedings…**

MARCHE, C.; PAULIN-MOHRING, C.; URBAIN, X. **The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML**. Journal of Logic and Algebraic Programming, v.58, n.1-2, 2004.

MCCARTHY, J. Towards a mathematical theory of computation. In: INTERNATIONAL CONGRESS ON INFORMATION PROCESSING, 1963. **Proceedings...**, p. 21-28

MCGRAW, G; FELTEN, E. **Securing Java: getting down to buiness with mobile code**. [s.L.]: Wiley Computer, 1999.

MENEZES, A. **Handbook of applied cryptography**. [s.L.]: CRC Press, 1997.

MEYER, B; ABRIAL, J; SCHUMAN, S. **A specification language, in on the construction of programs**. Cambridge University Press, 1980.

MONTGOMERY, M.; KRISHNA, K. **Secure object sharing in Java card.** In: USENIX WORKSHOP ON SMART CARD TECHNOLOGY, 1999. **Proceedings…**

MORGAN, C. **Programming from specifications**. 2.ed. [s.L.]: Prentice-Hall, 1994.

OESTREICHER, M; KSHEERADBHI, K. Object Lifetimes in JavaCard. In: USENIX WORKSHOP ON SMART CARD TECHNOLOGY, 1999. **Proceedings…**

OESTREICHER, M. Transactions in JavaCard. In: IEEE ANNUAL COMPUTER SECURITY APPLICATIONS CONFERENCE, 1999. **Proceedings…**

OPEN CARD CONSORTIUM. **Open Card Framework Specification**. Versão 1.2, 2000.

OFFCARD. **Java Card 2.2 Off-Card Verifier**. Version 2.2. [s.L.}: Sun Microsystems, 2002.

POLL, E.; VAN DENBERG, J.; JACOBS, B. Formal specification of the JavaCard API in JML: the APDU class. **Computer Networks**, v.36, n. 4, 2001.

PREE, W. Meta patterns: a means for capturing the essencial of reusable object-oriented design. EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 1994, Bologna. **Proceedings…**

PREE, W. **Design patterns of object-oriented software development**. [s.L.]: Addison-Wesley, 1995.

RANKL, W.; EFFING, W. **Smart card handbook**. 2.ed. New York: John Wiley & Sons, 2000.

RENAUDIN, M. et al. High security smartcards, design, automation and test. In: EUROPE CONFERENCE AND EXHIBITION, 2004. **Proceedings…**

RMI. **Java remote method invocation**. Version 1.1. [s.L.]: Sun Microsystems, 2002.

ROSE, E. Lightweight bytecode verification. **Journal Journal of Automated Reasoning**, v.31, n. 3-4, 2003.

SCHWAN, M. An extended model of security policy for multi-applicative smart cards. ACM SYMPOSIUM ON INFORMATION, COMPUTER AND COMMUNICATIONS SECURITY, 2., 2007. **Proceedings…**

SQUAWK. **The Squawk System, Preliminary Draft Specification**. Version 2.1. [s.L.]: Sun Microsystems, 2002.

SRINIVASAN, S. Design patterns in object-oriented frameworks. **Computer**. [s.L.]: IEEE, 1999.

SUSAN C.; VLADA M. **JSR-000907 Java transaction API (JTA) specification.** Version 1.1. [s.L.]: Sun Microsystems, 2002.

TURING, A. Computing machinery and intelligence. **Mind New Series,** v. 59, n. 236, 1950.

VAN DALEN, D. **Logic and structure**. 2.ed. [s.L.]: Springer, 2004

WARMER, J.; KLEPPE, A. **The object constraint language: precise modeling with UML**. [s.L.]: Addison-Wesley, 1998.