Universidade Federal do Rio de Janeiro

Escola Politécnica

Departamento de Eletrônica e de Computação

Emprego de Smartcards em Problemas de Autenticação

Autor:

Luís Renato Azevedo de Araujo Silva

Autor:

Thiago Henrique da Silva

Orientador:

Flávio Luis de Mello, DSc

Examinador:

Prof Edilberto Strauss, PhD

Examinador:

Prof. Felipe Maia Galvão França, PhD

DEL

Março de 2009

DEDICATÓRIA

Este trabalho é dedicado a todos meus familiares pela colaboração com meu crescimento profissional e pessoal, especialmente a minha tia-avó, dona Helena, por todo apoio e carinho que tem me dado nesses 24 anos, meu pai por me auxiliar e orientar nas horas mais difíceis e minha mãe, aonde quer que esteja, por ter me ensinado grandes valores na vida.
Dedico também a minha namorada Isabela, que sempre foi compreensiva e deu apoio na minha longa jornada de estudos e trabalhos durante a faculdade.
Muito obrigado a todos.

Luís Renato Azevedo de Araujo Silva

Esse trabalho é dedicado aos meus familiares e companheiros de faculdade, sem eles eu não conseguiria chegar onde cheguei.
Ao meu pai, pelo auxilio e orientação, mesmo estando em outro país.
Dedico este trabalho também a minha namorada, minha irmã e a minha mãe, pela compreensão e apoio nas horas mais difíceis nesta longa caminhada.
Muito obrigado.

Thiago Henrique da Silva

AGRADECIMENTO

Primeiramente a Deus, pelas glórias que temos conseguido em nossas vidas.

Aos nossos pais que nos deram a vida e nos ensinaram a vivê-la com dignidade e honestidade e que se doaram por inteiro e renunciaram alguns de seus sonhos para que muitas vezes pudéssemos realizar os nossos.

Aos amigos e familiares que compartilharam e alimentaram nossos ideais, compreendendo nossa falta de tempo, finais de semana de estudo e noites em claro, tendo sempre uma palavra e um sorriso a nos incentivar para que chegássemos ao final de mais uma etapa.

Nossa eterna e singela gratidão a todos os professores do DEL pelas lições e conhecimentos passados, em especial ao professor e orientador Flávio Luis de Mello, pelo apoio e incentivo na construção desse trabalho.

Vocês fazem parte dessa vitória e da nossa história.

RESUMO

Smart cards são chips capazes de armazenar e executar aplicações computacionais, trabalhando com segurança robusta devido a criptografia utilizada nestes dispositivos. Melhoram a comodidade e a segurança de qualquer transação pois armazenam informações à prova de falsificações além de fornecer componentes vitais para a troca de informações no meio virtual.

A tecnologia dos *Smart cards* (Cartões Inteligentes) vem crescendo nos últimos anos e é visível como os cartões tradicionais vêm sendo substituídos por eles. Pode-se encontrá-los por toda parte: bancos, transporte coletivo e até mesmo nos celulares, com os chamados SIM Cards. Como será mostrado ao longo do trabalho, *Java Card* é um subconjunto da linguagem Java para ser utilizado em componentes embarcados, especificamente em *Smart cards*.

O presente material pretende mostrar a segurança de uma solução baseada em *Smart cards* através de um sistema de autenticação de usuários nos laboratórios do DEL. Com esse intuito foram desenvolvidos um *applet*, programa em *Java Card* que roda dentro de um *Smart card*, um aplicativo host leitor, para fazer a interface de autenticação dos usuários, e um aplicativo host escritor, que serve como interface para gravação de dados no cartão.

Palavras-Chave: *Smart card*, *Java Card*, *applet*, *APDU*, host, segurança.

ABSTRACT

Smart cards are chip capable of storing and executing computer applications, working with robust security due to the encryption facilities used in these devices. They greatly improve the convenience and security of any transaction to store tamper-proof information in addition to providing vital components to exchange information in the virtual environment.

The technology of *Smart cards* (*Smart card*) has been growing in recent years and is visible as the traditional cards are being replaced by them. You may find them everywhere: banks, public transport and even on mobile devices, with the so-called SIM Cards. As will be shown throughout the work, *Java Card* is a subset of the Java language for use on board components, specifically in *Smart cards*.

This material aims to show the security of a solution based on *Smart cards* through a system of users authentication in DEL laboratories. To this end it was developed an applet, which is a *Java Card* program stored inside a *Smart card*, an reader host application to make the authentication procedures, and a writer host application, which serves as an interface for recording data on the card.

SIGLAS

UFRJ – Universidade Federal do Rio de Janeiro

DEL – Departamento de Eletrônica

CAD – Card Acceptance Device

ISO – International Organization for Standardization

IEC - International Electrotechnical Commission

ROM – Read-Only Memory

EEPROM – Electrical Erasable Programmable Read-Only Memory

RAM – Random Access Memory

ICC – Integrated Circuit Chip

CPU – Central Processor Unit

JCVM – Java Card Virtual Machine

JCRE – Java Card Runtime Environment

API - Application Programming Interface

JCDK – Java Card Development Kit

J2ME – Java 2 Micro Edition

J2SE - Java 2 Standard EditionPC – Personal Computer

APDU – Aplication Protocol Data Unit

JCRMI - Java Card Remote Method Invocation

USB - Universal Serial Bus

RF – Radio Frequency

JVM - Java Virtual Machine

CAP – Converted Applet

SDK - software development kit

AID – Applet Identification

CLDC - Connected Limited Device Configuration

PIN – Personal Identification Number

CRCs - Cyclic Redundancy Checks

MD4 – Message Digest 4

MD5 – Messaged Digest 5

DRE - Divisão de Registro de Estudante

LIG – Laboratório de Informática da Graduação

LIF – Laboratório de Instrumentação e Fotônica

GTA – Grupo de Teleinformática e Automação

LPS – Laboratório de Processamento de Sinais

PADS – Processamento Analógico e Digital de Sinais

ASCII - American Standard Code for Information Interchange

CBC - Cypher Block Chaining

DES - Data Encryption Standard

3DES or TDES – Triple Data Encryption Standard

SIM – Subscriber Identity Module

GSM – Global System for Mobile Communications

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 – Tema

Emprego de *Smart cards* e as tecnologias que materializam sua utilização.

1.2 – Delimitação

Neste trabalho se dará ênfase ao chip de contato e com microprocessador. A tecnologia empregada para programação dos *Smart cards* será o JavaCard.

1.3 – Justificativa

A rápida inovação tecnológica e a complexidade dos ambientes de negócios levam as organizações a alterarem a maneira como fazem seus negócios [2], a maneira como desenvolvem novos produtos e serviços, bem como o modo como disponibilizam isso ao mercado. Uma das tecnologias que está emergindo e modificando a economia é a dos cartões com circuitos integrados (ICCs), também conhecidos como *Smart cards* ou *Chip Cards*.

Os *Smart cards* oferecem o poder da computação portável e individual, possibilitando o surgimento de uma variedade de produtos e serviços baseados nesses cartões. Viabilizam novas oportunidades de negócios e novos mercados voltados, principalmente, para o comércio eletrônico, entre outras aplicações.

Grandes fabricantes estão desenvolvendo novas tecnologias e produtos para área de atuação dos *Smart cards*, incentivando, viabilizando e popularizando a tecnologia. O uso dessa tecnologia é uma tendência futura, para as mais variadas soluções computacionais que demandam soluções flexíveis, de baixo custo, com alta segurança. [14].

Sistemas baseados na tecnologia dos *Smart cards* estão em uso todos os dias, em aplicações como: sistema de saúde, transações bancárias, entretenimento, transporte, telefonia, armazenamento de certificados digitais e muitas outras aplicações. Todas estas aplicações são beneficiadas pelas funcionalidades e segurança [4] que os *Smart cards* provêem.

Existe uma grande carência de aplicações computacionais, principalmente em aplicações nas quais a segurança é um ponto crucial, fazendo crescer o interesse das or-

ganizações, da comunidade acadêmica e dos desenvolvedores em criar soluções baseadas em *Smart cards*, com suas variadas tecnologias de programação, dentre elas o *Java Card* [10], devido as suas facilidades e características únicas até o momento.

Com o crescimento e a popularização dos *Smart cards*, cresce a preocupação com a segurança dos mesmos, demandando mais pesquisas nessa área. A exploração de vulnerabilidades, no que diz respeito tanto a parte física quanto a parte lógica, está sendo um campo de estudo muito interessante e de grande valia para os desenvolvedores de *Smart cards*. Teses de Doutorado, como por exemplo, [3] procuram identificar as vulnerabilidades através de diferentes tipos de ataques sugerindo correções para os mesmos.

Produtos desenvolvidos com tecnologia *Java Card* disponibilizam uma série de funcionalidades inovadoras [8], incluindo:

- Possibilidade para o desenvolvimento de programa para *Smart card* em linguagens de alto nível (um subconjunto de linguagem Java);
- Possibilidade de atualizar os cartões, com novas aplicações, mesmo depois de estarem em circulação;
- Mecanismos fortes de segurança como a rigorosa separação dos *applets* no cartão;

Um aspecto o qual leva a tecnologia *Java Card* ter cada vez mais aceitação, além das suas facilidades e funcionalidades, é que a mesma obedece rigorosamente um padrão sólido para o desenvolvimento dos *Smart cards* regido pelo ISO/IEC. Sendo assim, há uma garantia de coexistência e compatibilidade entre equipamentos. Além desses benefícios, evita-se que um padrão fechado de *Smart cards* seja desenvolvido e domine o mercado.

Quanto a ferramentas de desenvolvimento para a programação dos *Smart cards* existe uma grande variedade de kits de desenvolvimento que vão desde aplicações específicas, *frameworks* de desenvolvimento até *plugins* os quais são incorporados a ambientes de desenvolvimento [1].

Muitas soluções podem ser desenvolvidas, integradas ou até mesmo reformuladas para uma operação de forma mais efetiva utilizando os *Smart cards* com tecnologia *Java Card*. A tecnologia e as ferramentas existem, pode-se notar que a utilização de *Smart cards* agrega um ganho considerável em diversos aspectos como segurança, mobilidade, praticidade entre outros.

1.4 – Objetivo

O presente trabalho tem por objetivo apresentar um estudo das características principais dos cartões inteligentes (*Smart cards*), enfatizando a tecnologia *Java Card* como plataforma de *software*, e mostrar aspectos estruturais e algumas funcionalidades referentes a essa tecnologia.

1.5 – Metodologia

Para alcançar os objetivos propostos para este trabalho foram realizadas as seguintes etapas: O primeiro passo foi estudar os conceitos relacionados a *smart cards* e compreender aspectos da tecnologia *Java Card* através de levantamentos bibliográficos. Em seguida, foi criado um tutorial e instalação de uma ferramenta de desenvolvimento em *Java Card*. Após isto, foram elaborados documentos parciais com idéias mais relevantes para serem apresentadas no documento final, seguido da implementação de uma aplicação com funcionalidades *Java Card*. Na etapa seguinte foram realizados testes e validações (através da utilização e aplicabilidade da solução) dessas funcionalidades. Por fim foi efetuada a redação conclusiva do projeto final.

1.6 – Descrição

Neste capítulo é apresentada uma visão geral do trabalho e o contexto no qual está inserido.

No capítulo 2 é abordada uma visão geral sobre *Smart cards* e suas áreas de aplicação.

O capítulo 3 apresenta o *Java Card*, uma tecnologia empregada na manipulação de *Smart cards*.

O capítulo 4 mostra as ferramentas utilizadas para a elaboração, simulação e testes de aplicações em *Smart cards* usando *Java Card*, mais especificamente o *Java Card Development Kit*.

O capítulos 5 e 6 descrevem aplicações práticas utilizando a tecnologia explicitada no capítulo anterior, os resultados obtidos e as dificuldades encontradas. Com base nas ferramentas do capítulo anterior, demonstraremos algumas aplicações de *Smart cards* com casos de uso, bem como a validação e a análise dos resultados obtidos com a aplicabilidade da solução proposta.

O capítulo 7 apresenta a conclusão do trabalho.

Capítulo 2

Smart cards

Neste capítulo será introduzido o conceito de *Smart cards* que abrange sua arquitetura interna, seu funcionamento e suas características físicas.

2.1 – Conceitos Introdutórios

Um *Smart card* é um cartão de plástico que contém um circuito integrado embutido com capacidade de processar; armazenar e transmitir dados [1]. Assemelha-se em tamanho e forma aos cartões de crédito. São altamente seguros na própria concepção e desenho e ao mesmo tempo manuteníveis à medida que podem ser apagadas as informações contidas nele. Uma ilustração de um *Smart card* típico pode ser vista na Figura 2.1.

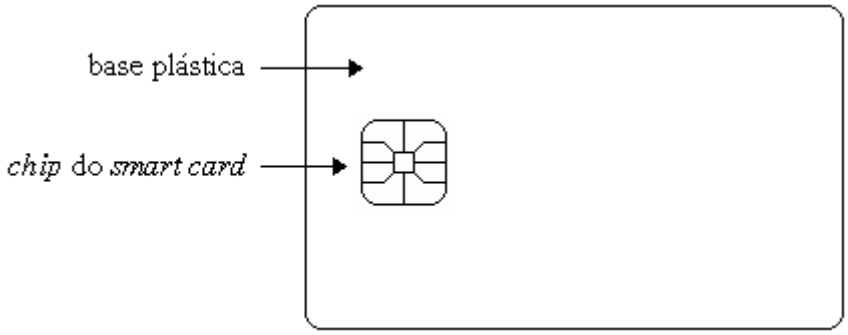


FIGURA 2.1 – *SMART CARD* TÍPICO.
FONTE: CHEN, ZHIQUN [1]

Para que haja uma interoperabilidade entre diferentes fornecedores foi estabelecida uma padronização para a indústria de *Smart cards*. O documento ISO7816 descreve padrões tais como características físicas; dimensão e localização dos contatos; sinais eletrônicos e protocolos de transmissão; comandos não-proprietários; entre outros.

Cartões inteligentes podem ser classificados em algumas categorias, como por exemplo os cartões de memória ou aqueles com microprocessadores, e ainda, de contato ou sem contato (contactless) [1]. Neste trabalho se dará ênfase ao *chip* de contato e com microprocessador.

Os primeiros *smart cards* produzidos em larga escala eram cartões de memória. Esse tipo de cartão não é exatamente inteligente, pois não possuem microprocessador e conseqüentemente não podem processar dados por si só.

Cartões de memória de maneira geral podem realizar poucas instruções previamente programadas, essas funções são limitadas e não podem ser reprogramadas. Por essas características, uma vez que o conteúdo do cartão é usado ele deve ser descartado. Este tipo de tecnologia é amplamente empregada em serviços pré-pagos como, por exemplo, o telefone público.

Por outro lado, existem os cartões com microprocessadores, e tal como o nome sugere, podem processar dados. A característica principal desse tipo de cartão é a capacidade multifuncional e um mecanismo de segurança melhorado, uma vez que um dado nunca é acessado diretamente por aplicações externas. Toda a manipulação de dados e acesso a memória é controlada pelo microprocessador de acordo com certas condições de acesso como o uso de senha e criptografia de dados. Essa tecnologia é usada em casos em que aplicações necessitam segurança de dados. O termo "*Smart card*" é usado tanto para cartões de memória como para cartões com microprocessadores, no entanto, algumas publicações preferem usar essa nomenclatura apenas para cartões com microprocessador, pela sua capacidade de processar dados e realizar funções lógicas. Neste trabalho o termo *Smart card* fará referência aos cartões com microprocessadores.

2.2 – A arquitetura dos cartões

Os *Smart cards* com contato possuem 8 pontos de contato (Figura 2.2), uma unidade de processamento central e vários tipos de memória. Quando inseridos em um CAD (*Card Acceptance Device*) o chip faz contato com os conectores elétricos que viabilizam a leitura e escrita de informação no cartão. A tabela 2.1 descreve cada um dos pontos de contato ilustrados na Figura 2.2.

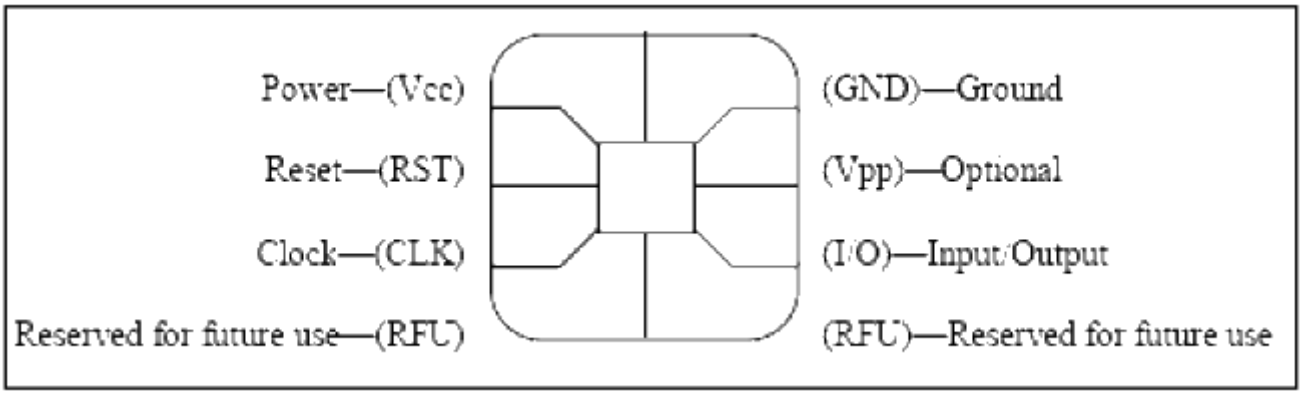


FIGURA 2.2 - PONTOS DE CONTATOS DO CARTÃO INTELIGENTE (*SMART CARD*)
FONTE: CHEN, ZHIQUN [1]

Vcc	Fornece a energia ao *Chip*
RST	Ponto usado para enviar um sinal de *reset* ao microprocessador
CLK	Fornecer o sinal de *Clock* externo, a partir do qual sinal interno de *Clock* e de *Clock* é derivado
GND	Ponto usado como tensão de referência e o seu valor é considerado 0 volts
Vpp	Ponto Opcional usado unicamente em cartões antigos
I/O	Ponto de contato usado para transferir dados e comandos entre o Cartão Inteligente e o mundo exterior, mas em modo *Half- Duplex*
RFU	Ponto reservado para um uso futuro

TABELA 2.1 – DESCRIÇÃO DOS PONTOS DE CONTATOS DE UM CARTÃO INTELIGENTE.
FONTE: ISO/IEC [7]

2.3 – *Smart cards* como dispositivo de segurança

Pela sua prórpia concepção e sua capacidade de processar dados além de armazená-los, os *Smart cards* vem sendo amplamente utilizado como dispositivo de segurança. Esta tecnologia está substituindo os atuais métodos de segurança e os principais exemplos são: a estrutura GSM para celulares e os cartões bancários. Nos capítulos posteriores será abordado um sistema de autenticação baseado em *smart cards*, que mostrará o potencial da tecnologia em sistemas de segurança digital.

A principal dificuldade encontrada em clonar um *smart card* é que para que tal pratica seja realizada, deve-se conhecer o protocolo de autenticação, algoritmos utilizados no processo e como é realizada a derivação de chaves. Atualmente existem

práticas que são utilizadas para tentar quebrar um processo de autenticação, como engenharia reversa (somente pode ser realizada se o algoritmo de autenticação for conhecido) e ataque, que consiste em tratar de descobrir o processo de autenticação através de tentativas sucessivas. Porém ainda que se descubra um meio de quebrar a autenticação, não significa que seja possível clonar um *smart card* uma vez que para tal a chave de autenticação de um usuário deve ser conhecida e isso implicaria obter informação privilegiada a respeito do processo de derivação de chaves, que pode ser realizado de diversas maneiras.

Existem casos conhecidos onde a clonagem de *smart cards* utilizados em processos de autenticação foi realizada com sucesso, como a quebra do algoritmo COMP128v1, utilizado anteriormente para autenticação dos celulares na rede GSM. Uma vez conhecido o processo foi possível realizar tentativas sucessivas de autenticação combinadas com engenharia reversa para obter as chaves de autenticação de um determinado usuário, e com elas clonar um SIM card. Com o desenvolvimento do poder computacional a quebra do algoritmo de autenticação tornou-se trivial, em processo descoberto pela área de pesquisa da IBM [18]. Após episódios de clonagem, o algoritmo COMP128v2 passou a ser utilizado na autenticação e até os dias atuais não há registro de quebra ou clonagem. Com uma maior preocupação em segurança, a rede 3G utiliza o algoritmo Milenage, que é mais complexo que o altual e preve outros casos, como autenticação mutua (a rede e o SIM card são autenticados mutuamente) e prevenção contra quebra por repetição [19].

Além da quebra da autenticação do algoritmo COMP128v1 para autenticação nas redes GSM, já foram reportados pela mídia casos onde a colonagem de cartão de crédito ou débito. Estes processos, em geral, envolvem além de aparelhagem de tecnologia avançada e custosa para extrair a informação dos *smart cards*, o conhecimento da senha pessoal do usuário e o funcionamento do protocolo.

Capítulo 3

Java Card

Nesse capítulo será apresentada a tecnologia *Java Card*, que é uma adaptação da plataforma Java para ser utilizada em *Smart cards* e outros dispositivos cujos ambientes têm memória limitada e restrições de processamento.

3.1 – Contextualização

Anos atrás a *Sun Microsystems* percebeu o potencial dos cartões inteligentes e dos dispositivos semelhantes. Assim, definiu especificações para um subconjunto da tecnologia Java para a criação de aplicativos voltados para este contexto, os *applets Java Card*. Um dispositivo que suporte essas especificações é definido como uma plataforma *Java Card*. Em uma plataforma *Java Card* vários aplicativos de diferentes fornecedores podem coexistir seguramente.

A especificação dessa tecnologia, é constituída de três partes:

JCVM (*Java Card Virtual Machine*): define um subconjunto da linguagem Java e a adapta as aplicações no *smart card*.

JCRE (*Java Card Runtime Environment*): define como se comporta o ambiente de execução, incluindo gerenciamento de memória, de aplicações, reforço de segurança e outras características da execução.

Java Card API specification, que define o *framework* e as extensões dos pacotes (*packages* e classes Java) para aplicações do tipo *smart card.*

A Sun também oferece o *Java Card Development Kit* (JCDK), que inclui uma referência à implementação do *JCRE* e da *JCVM* e outras ferramentas que ajudam a desenvolver *applets* de *Java Card*.

3.2 – Arquitetura de um aplicativo *Java Card*

Um aplicativo *Java Card* completo consiste em aplicativos e sistemas *back-end*, um aplicativo *host* (*off-card*), uma interface (leitora), *applets* do cartão, credenciais de usuário, e suporte a *software*. Todos estes elementos juntos compõem um aplicativo *end-to-end* seguro. Esta arquitetura, ilustrada na Figura 3.1, será descrita em detalhes a seguir.

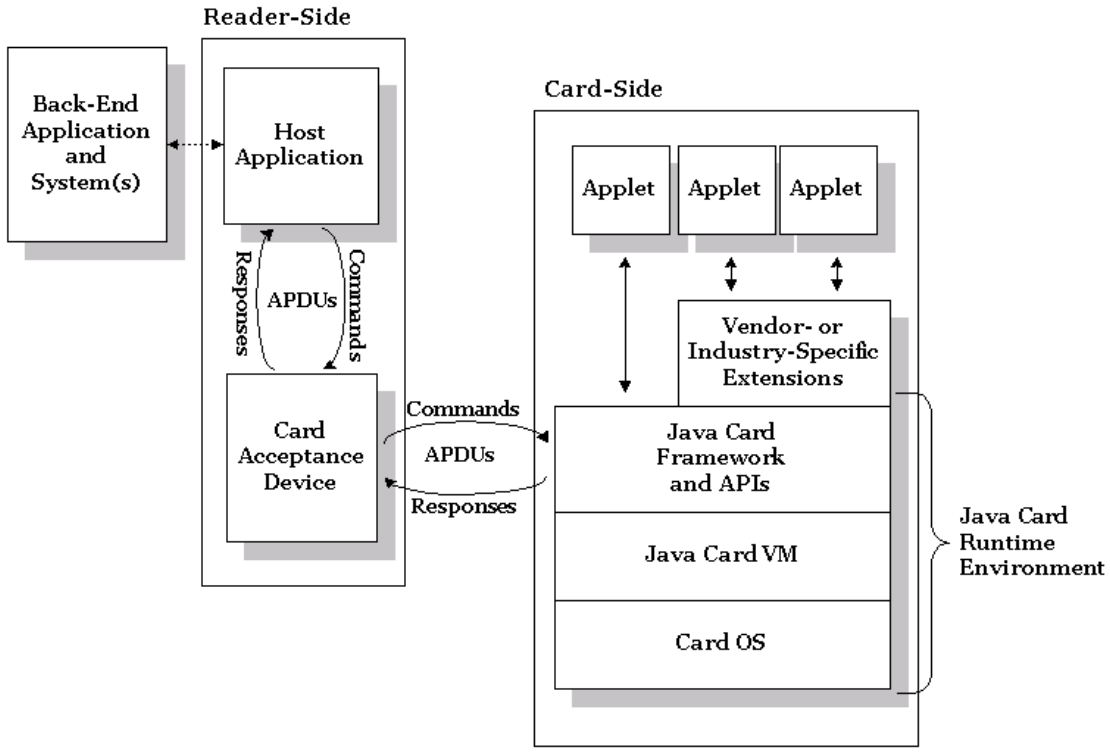


FIGURA 3.1 - ARQUITETURA DE UM APLICATIVO *JAVA CARD*.
FONTE: SUN MICROSYSTEMS [11]

Aplicativos *Back-End* fornecem serviços que suportam *applets* internos ao cartão. Por exemplo, um aplicativo *Back-End* poderia fornecer conectividade a sistemas de segurança que, em conjunto com credenciais internas ao cartão, contribuam para o aumento do nível de segurança. Em um sistema de débito eletrônico, o aplicativo *Back-End* poderia fornecer acesso a cartão de crédito ou outro tipo de informação de pagamento.

O aplicativo cliente reside em um *desktop* ou um terminal de pagamento eletrônico, um celular ou até um subsistema de segurança. O aplicativo cliente é responsável pela comunicação entre o usuário, o *applet* e o provedor do aplicativo *back-end*. Tradicionalmente, os aplicativos de leitura têm sido escritos em linguagem C. Contudo, a recente adoção da tecnologia J2ME permite que esse aplicativo seja também codificado em linguagem Java.

Os vendedores de *Smart cards* geralmente fornecem não só o *kit* de desenvolvimento mas também API's que suportam aplicativos que fazem a leitura assim como *applets* de *Java Card*. Como exemplo para ilustrar podemos citar o *OpenCard Framework*, um conjunto de API's em Java que escondem de diferentes vendedores alguns detalhes da interação do cartão com a leitora.

O *Card Acceptance Device* (CAD - Dispositivo de aceitação de cartão) é a interface entre o dispositivo que fica entre o aplicativo cliente de leitura e o cartão propriamente dito. O *CAD* fornece energia, seja através de comunicação elétrica seja de rádio freqüência ao cartão. Ele pode atuar como um leitor de cartão quando conectado a um *PC* usando uma porta serial ou

integrado a um terminal eletrônico de pagamento. O dispositivo envia comandos *APDU* (*Application Protocol Data Units*) do cliente para o cartão e envia respostas do cartão de volta para o cliente. Alguns *CAD*'s têm teclas para digitação de senhas e podem conter um *display*.

A plataforma *Java Card* é um ambiente de múltiplos aplicativos. Um ou mais *applets* podem estar contidos em um cartão em conjunto com *softwares* suportados: o sistema operacional do cartão é o *Java Card Runtime Enviroment* (*JCRE*). O *JCRE* consiste na máquina virtual Java, *Java Card Framework*, *API*'s e algumas extensões de *API*'s.

Todos os *applets Java Card* fazem parte da classe base e devem implementar os métodos *Install()* e *Process()*; O *JCRE* faz a chamada do método *Install()* ao instalar o *applet*, e faz a chamada do *Process()* toda vez que há uma *APDU* sendo encaminhada ao *applet*.

Os *applets* são instanciados quando carregados e ficam presentes mesmo quando não há mais fornecimento de energia. Um *applet* se comporta como um servidor e é passivo. Depois que um cartão é ativado (*powered up*) cada *applet* permanece inativo até que seja selecionado por uma *APDU* especial interpretada pela *JCRE*, a partir desse momento todos os comandos serão direcionadas a aplicação selecionada. Mais oportunamente descreveremos como uma aplicação em um *smart card* se torna ativa.

3.3 – Comunicação com *Applet*

Podem ser utilizados dois tipos de comunicações entre cliente e uma aplicação. O primeiro modelo é um modelo de troca de mensagem e o segundo é baseado em *JCRMI* (*Java Card Remote Method Invocation*) ou método de chamada remota de *Java Card*, um subconjunto de modelos-objetos *RMI* do *J2SE*. Estaremos utilizando o modelo padrão de troca de mensagens e por isso não abordaremos o *JCRMI*.

O modelo de troca de mensagem é o modelo básico de todas as comunicações efetuadas no contexto do *Java Card*. No seu centro está o *Application Protocol Data Unit* (*APDU*), um pacote lógico de dados que realiza trocas entre o *CAD* e o *framework Java Card*. Este último recebe e encaminha para ao *applet* apropriado qualquer comando *APDU* enviado pelo *CAD*. O *applet* processa o comando *APDU*, e retorna uma resposta *APDU*. Os comandos *APDU*'s respeitam normas internacionais ISO / IEC 7816-3 e 7816-4. A Figura 3.2 apresenta este procedimento de comunicação usando o modelo de troca de mensagem

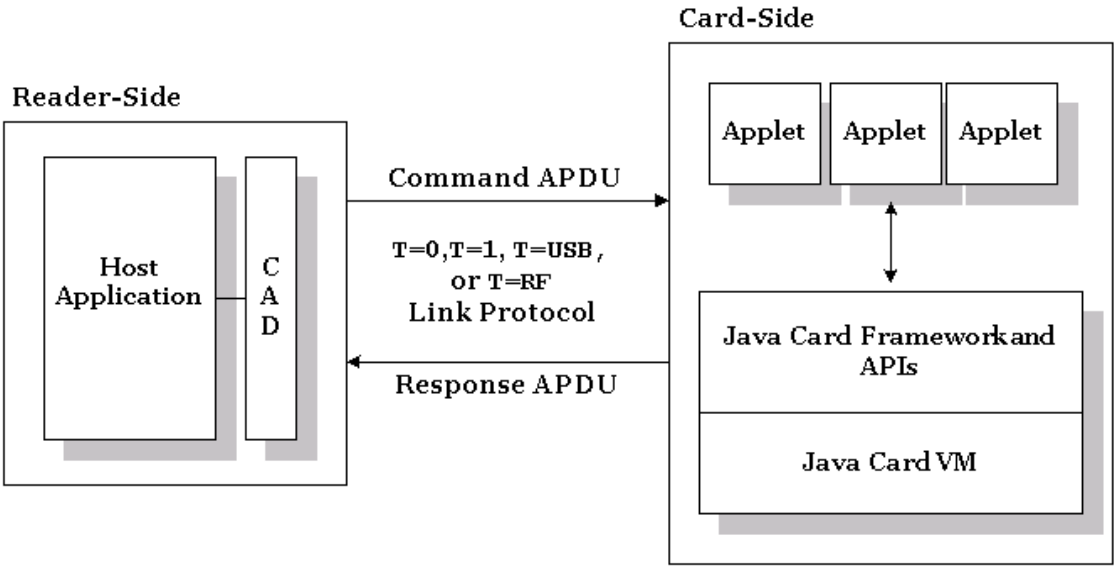


FIGURA 3.2 - COMUNICAÇÃO USANDO O MODELO DE TROCA DE MENSAGEM.
FONTE: SUN MICROSYSTEMS [11]

A comunicação entre a leitora e o cartão normalmente é baseada em um dos dois protocolos *link*, o *byte-oriented* T = 0 ou o *block-oriented* T = 1. Outros protocolos como T = USB e T= RF podem ser utilizados. A classe *APDU* do *JCRE* oculta alguns dos detalhes de protocolo do pedido, mas não de todos, porque o protocolo T = 0 é bastante complexo.

A estrutura de um comando *APDU* (Command *APDU*) é controlada pelo valor do seu primeiro *byte* e na maioria dos casos é descrita tal como a Figura 3.3:

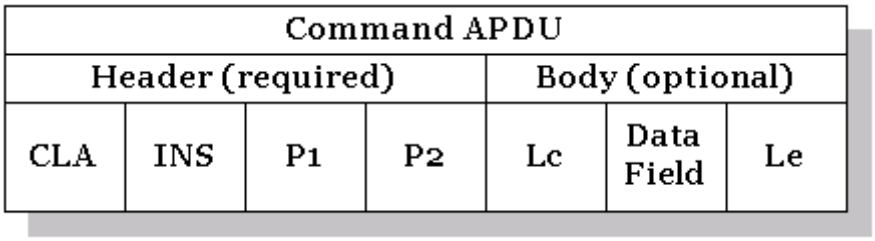


FIGURA 3.3. COMMAND *APDU*
FONTE: SUN MICROSYSTEMS [11]

Um comando *APDU* necessita de um cabeçalho e pode possuir ou não um corpo, contendo um CLA (1 *byte*). Este campo obrigatório identifica uma classe de instruções especifica para os aplicativos. Valores CLA válidos são definidos na especificação da norma ISO 7816-4, conforme apresentado na Tabela 3.1.

CLA	Classe de Instrução
0x0n, 0x1n	**Instruções de cartão ISO 7816-4, bem como acesso de arquivos e operações de segurança**
20 to 0x7F	Reservado
0x8n or 0x9n	**Formato ISO/IEC 7816-4 que pode ser usado para instruções específicas do aplicativo e interpretado de acordo com o padrão.**
0xAn	Instruções específicas do vendedor ou do aplicativo
B0 to CF	**Formato ISO/IEC 7816-4 que pode ser usado para instruções específicas do aplicativo**
D0 to FE	Instruções específicas do vendedor ou do aplicativo
FF	Reservado para seleção do tipo de protocolo

TABELA 3.1 - VALORES CLA DE ACORDO COM O PADRÃO ISO 7816

FONTE: SUN MICROSYSTEMS [11]

Em teoria, podem ser usados todos os valores CLA de 0x80 em diante para instruções específicas do aplicativo, mas em muitas implementações atuais de *Java Card* apenas as que estão em negrito são realmente reconhecidas.

O INS (1 *byte*) corresponde a um campo obrigatório que indica uma instrução específica interna à classe de instrução identificada pelo campo CLA. A norma ISO 7816-4 especifica as instruções básicas de uso para o acesso aos dados de um cartão quando este está estruturado de acordo com um sistema de arquivos "*on-card*", tal como definido na norma. Outras funções foram especificadas em outros pontos do padrão, alguns dos quais são funções de segurança. Ver Tabela 3.2, para uma lista de algumas das instruções ISO 7816. Podemos ainda, de acordo com a norma, definir outros valores específicos de aplicações INS apenas quando utilizamos um valor adequado para o *byte* CLA.

INS	Command Description
0E	*Erase Binary*
20	*Verify*
70	*Manage Channel*
82	*External Authenticate*
84	*Get Challenge*
88	*Internal Authenticate*
A4	*Select File*
B0	*Read Binary*
B2	*Read Record(s)*
C0	*Get Response*
C2	*Envelope*
CA	*Get Data*
D0	*Write Binary*
D2	*Write Record*

D6	*Update Binary*
DA	*Put Data*
DC	*Update Record*
E2	*Append Record*

TABELA 3.2 - ISO 7816-4 VALORES DE INS QUANDO CLA = 0X

FONTE: SUN MICROSYSTEMS [11]

O P1 (1 *byte*) é um campo obrigatório que define instruções para o parâmetro 1. Pode ser utilizado para qualificar o campo INS ou para entrada de dados. O P2 (1 *byte*), por sua vez, corresponde a um campo obrigatório que define instruções para o parâmetro 2. Pode ser utilizado para qualificar o campo INS ou para entrada de dados. O Lc (1 *byte*) é um campo opcional que define o número de *bytes* do comando. *Data field* (variável, número de *bytes* Lc) também é um campo opcional, mas que no entanto detém os dados do comando (*Command Data*). Por fim o Le (1 *byte*) trata-se de um campo opcional que especifica o número máximo de *bytes* de dados da resposta esperada.

Dependendo da presença de dados de comando (*Command Data*), e se é necessária uma resposta a ser enviada, existem quatro variações possíveis para o comando *APDU* (ver Figura 3.3). Precisamos apenas nos preocupar com essas variações se estivermos utilizando o protocolo T = 0:

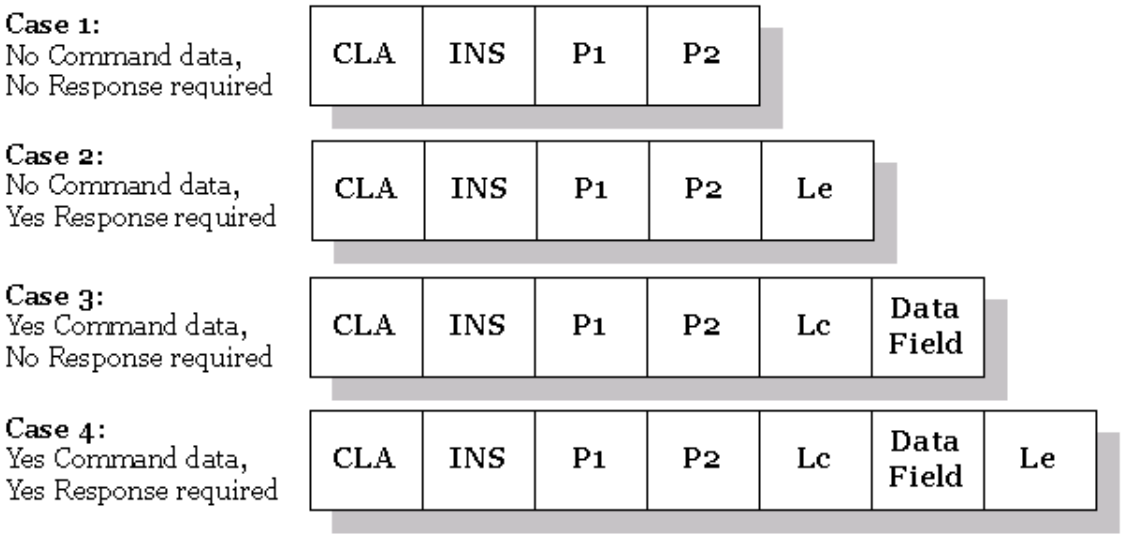


FIGURA 3.4 - QUATRO POSSÍVEIS ESTRUTURAS DE UM COMANDO *APDU*.
FONTE: ISO/IEC[5]

Uma aplicação típica irá utilizar vários comandos *APDU* com diferentes estruturas.

O formato de uma resposta *APDU* (*Response APDU*) é muito mais simples. Assim como um comando *APDU*, uma resposta *APDU* possui campos obrigatórios e opcionais (ver Figura 3.5).

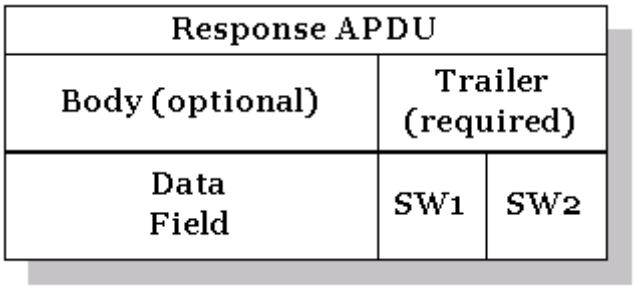


FIGURA 3.5 - RESPONSE *APDU* – *APDU* DE RESPOSTA
FONTE: SUN MICROSYSTEMS [11]

O *Data Field* (tamanho variável, determinada pelo Le no comando *APDU*) é um campo opcional que contém os dados retornados pelo *applet*. O SW1 (1 *byte*) e o SW2 (1 *byte*) são campos obrigatórios referentes aos *Status Word* 1 e *Status Word* 2, respectivamente. Os valores dos *Status Words* são definidos na especificação ISO 7816-4 e apresentados na Figura 3.6.

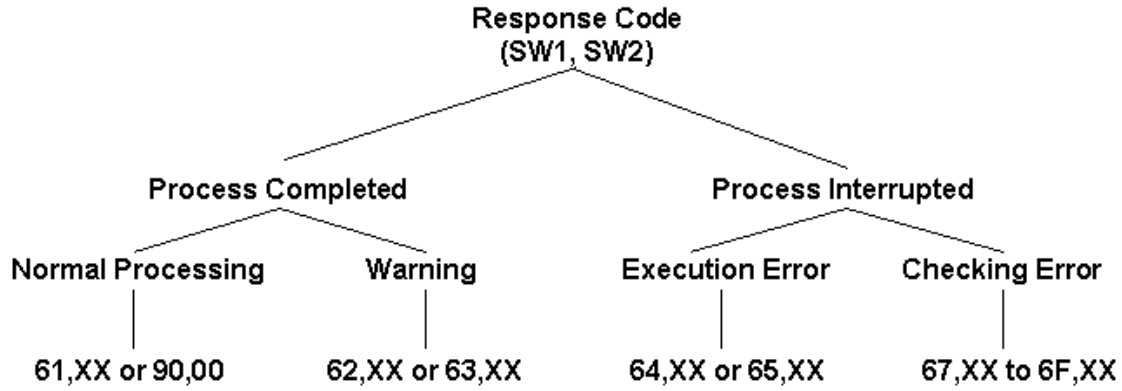


FIGURA 3.6 - RESPONSE STATUS CODES
FONTE: ISO/IEC [5]

A interface Java da norma ISO7816 na API do *Framework* de *Java Card* define um conjunto de constantes para facilitar a especificação de códigos de retorno de erros. A Resposta *Status Word*, códigos de erro e códigos de exceção, são apresentados na Tabelas 3.3 a 3.9

Insufficient Memory	**0x6400**

Incomplete Packet	0x6A87
Invalid Signature	0x6982
Invalid Packet	0x6A87
Class Length Overrun	0x6A84
Invalid Loader Command	0x6A86
Invalid AID Length	0x6901
Invalid API Version	0x6902
Invalid Password	0x6903
Invalid Signature Length	0x6904
Hash Corruption	0x6905
Hash Failure	0x6906
Success Packet	0x6301
Success Load	0x9000

TABELA 3.3 - LOADER STATUS WORDS

FONTE: SUN MICROSYSTEMS [11]

Unable to Select Applet	**0x8453**
Unable to Find Applet	0x8450
Unable to Select Applet	0x6A82

TABELA 3.4 - AID STATUS WORDS

FONTE: SUN MICROSYSTEMS [11]

Bad Master PIN	**0x6681**

TABELA 3.5 - COMMAND PROCESSOR STATUS WORDS

FONTE: SUN MICROSYSTEMS [11]

Uncaught Exception	**0x6F00**

TABELA 3.6 - VIRTUAL MACHINE STATUS WORDS

FONTE: SUN MICROSYSTEMS [11]

StackOverflowError	**0x61**
OutOfMemoryError	**0x62**
UnknownError	**0x63**
InternalError	**0x64**
IllegalAccessError	**0x65**

TABELA 3.7 - ERROR *BYTES*

FONTE: SUN MICROSYSTEMS [11]

SW_BYTES_REMAINING_00	**0x6100**
SW_CLA_NOT_SUPPORTED	0x6E00
SW_CONDITIONS_NOT_SATISIFED	0x6985
SW_CORRECT_LENGTH_00	0x6C00
SW_DATA_INVALID	0x6984
SW_FUNC_NOT_SUPPORTED	0x6A81
SW_INCORRECT_P1P2	0x6A86
SW_INS_NOT_SUPPORTED	0x6D00
SW_NO_ERROR	0x9000
SW_PIN_REQUIRED	0x6982
SW_RECORD_NOT_FOUND	0x6A83
SW_SECURITY_STATUS_NOT_SATIS-FIED	0x6982
SW_UNKNOWN	0x6F00
SW_WRONG_DATA	0x6A80

SW_WRONG_LENGTH	0x6700
SW_WRONG_P1P2	0x6B00

TABELA 3.8 – ISOEXCEPTIONS

FONTE: SUN MICROSYSTEMS [11]

Exception *bytes*

Exception Name	JiB32 (ver. 0.0)	JiB33 (ver. 1.0)
ArithmeticException	0x01,0x81	0x04,0x84
ArrayIndexOutOfBoundsException	0x02,0x82	0x05,0x85
ArrayStoreException	0x03,0x83	0x06,0x86
ClassCastException	0x04,0x84	0x07,0x87
Exception	0x05,0x85	0x08,0x88
IndexOutOfBoundsException	0x06,0x86	0x09,0x89
NegativeArraySizeException	0x07,0x87	0x0A,0x8A
NullPointerException	0x08,0x88	0x0B,0x8B
RuntimeException	0x0A,0x8A	0x0D,0x8D
SecurityException	0x0B,0x8B	0x0E,0x8E
Throwable	0x0C,0x8C	0x0F,0x8F
APDUException	0x0F,0x8F	0x12,0x92
ISOException	0x12,0x92	0x15,0x95
PINException	0x14,0x94	0x18,0x98
SystemException	0x17,0x97	0x1B,0x9B
TransactionException	0x18,0x98	0x1C,0x9C
UserException	0x19,0x99	0x1D,0x9D
CoprocessorException	N/A	0x02,0x82

TABELA 3.9 - EXCEPTION *BYTES*

FONTE: SUN MICROSYSTEMS [11]

Cada vez que há uma *APDU* chegando a um *applet* selecionado, o *JCRE* invoca o método process() do *applet* passando a *APDU* como um argumento. O *applet* deve analisar o comando *APDU*, processar os dados, gerar uma resposta *APDU*, e retornar o controle para o *JCRE*.

3.4 – A máquina virtual de *Java Card* – *Java Card* VM

A especificação da *Java Card Virtual Machine* a define como um subconjunto da linguagem de programação Java e como uma máquina virtual compatível com Java para *Smart cards*, incluindo a representação de dados binários, formatos de arquivos e conjunto de instruções de *JCVM*.

A máquina virtual para a plataforma de *Java Card* é implementada em duas partes, com uma parte externa à placa e a outra em execução no próprio cartão. A *virtual machine* da parte do cartão interpreta os *bytecodes* de *Java Card*, gerencia classes e objetos, e assim por diante. A parte externa da Java VM é uma ferramenta de desenvolvimento, normalmente referida como a ferramenta de conversão de *Java Card*, que carrega, verifica, e ainda prepara as classes Java em um *applet* de cartão para execução interna. A saída da ferramenta de conversão é um arquivo de

applet convertido (*Converted Applet, CAP*), um arquivo que contém todas as classes em um pacote Java numa representação carregável e de executável binária. O conversor também verifica se as classes estão em conformidade com a especificação de *Java Card*.

O *JCVM* suporta apenas um subconjunto restrito da linguagem de programação Java, porém preserva muitas das características familiares, incluindo objetos, herança, pacotes, criação dinâmica de objetos, métodos virtuais, interfaces e exceções. A especificação *JCVM* não suporta elementos de linguagem que utilizam em demasia o limite de memória de um *Smart card*. Tais limitações são apresentadas na Tabela 3.10.

Recursos de linguagem	Carregar classes dinâmicas, gerenciamento de segurança (*java.lang.SecurityManager*), ameaças, clonagem de objetos, e certos aspectos de controle de acesso a pacotes não são suportados.
Palavras-Chave	native, *synchronized, transient, volatile, strictfp* não são suportadas.
Tipos	Não há suporte para *char, double, float, e long*, ou para vetores multidimensionais. Suporte a int é opcional.
Classes e Interfaces	As classes *API* do núcleo Java e as interfaces (*java.io, java.lang, java.util*) não são suportadas com exceção de *Object* e *Throwable*, e a maioria dos métodos de *Object* e *Throwable* não estão disponíveis.
Exceções	Algumas subclasses *Exceptions* e *Errors* são omitidas porque as exceções e erros que encapsulam não estão disponíveis na plataforma *Java Card*.

TABELA 3.10 - RESUMO DAS LIMITAÇÕES DA LINGUAGEM *JAVA CARD*
FONTE: SUN MICROSYSTEMS [11]

Existem ainda limitações no modelo de programação. Por exemplo, uma classe de biblioteca carregada não pode mais ser estendida no cartão. Ela é programada como definitiva implicitamente.

De acordo com as restrições de memória, a especificação *JCVM* também define restrições (*constraints*) em muitos atributos de programa. A Tabela 3.11 resume as limitações de recursos da *JCVM*. Note-se que muitas destas restrições são tipicamente transparentes para os desenvolvedores *Java Card*.

Packages	Um pacote pode se referir a até 128 outros pacotes.
	Um nome altamente qualificado de pacote é limitado a até 225 *bytes*. O tamanho de um caracter depende do tipo de codificação utilizada.
	Um pacote pode conter até 255 classes.
Classes	Uma classe pode diretamente ou indiretamente implementar até 15 inter-

	faces.
	Uma interface pode herdar até 14 interfaces.
	Um *package* pode conter até 256 métodos estáticos caso contenham *applets* (um *applet package*), ou 255 caso não contenha (uma *library package*).
	Uma classe pode implementar 128 métodos instanciados públicos ou protegidos, e até 128 com visibilidade a pacotes.

TABELA 3.11 - RESUMO DAS RESTRIÇÕES (CONSTRAINTS) DE *JAVA CARD* VM.

FONTE: SUN MICROSYSTEMS [11]

Na *Java Card Virtual Machine*, como na *VM* do *J2SE* (*Java 2 Platform, Standard Edition*), os arquivos de classes são centrais, mas a especificação *JCVM* define dois outros formatos de arquivos para uma maior independência da plataforma, o *CAP* e os formatos de exportação (*Export formats*).

3.5 – API de *Java Card*

A especificação de *API* de *Java Card* define um pequeno subconjunto da tradicional linguagem de programação de *API* para Java - ainda menor do que a *CLDC* (*Connected Limited Device Configuration*) do *J2ME* (plataforma da *Sun Microsystems* que usa a linguagem Java para o desenvolvimento de aplicativos para dispositivos móveis). Não há suporte para *Strings*, ou para *threads* múltiplos. Não há nenhuma classe *wrapper* como *Boolean* ou *Integer*, e nenhuma classe ou classes de sistema.

Somado ao seu pequeno conjunto de classes familiares ao núcleo Java, o *framework* de *Java Card* define seu próprio conjunto de classes fundamentais especificamente para suportar as aplicações de *Java Card*. Estas estão incluídas nos seguintes pacotes:

- *Java.io* é definida como uma classe de exceção, a classe base *IOException*, para completar a exceção hierárquica de RM. Nenhuma das outras classes java.io tradicionais estão incluídas.
- *Java.lang* define objetos e classes *Throwable* em que faltam muitos dos seus métodos de *J2SE* homólogos. Além disso, define um conjunto de classes de exceção: a classe base *Exception*, várias exceções de *runtime*, e *CardException*. Nenhuma das outras classes *java.lang* tradicionais estão incluídas.
- *Java.rmi* define a interface remota e a classe *RemoteException*. Nenhuma das classes tradicionais *java.rmi* estão incluídas. Suporte para a Invoca-

ção de Método Remoto (*Remote Method Invocation, RMI*) é incluído para simplificar a migração e integração com dispositivos que utilizam tecnologia *Java Card*.

– *Javacard.framework* define as interfaces, classes, e as exceções que compõem o núcleo *Java Card Framework*. Ele define conceitos importantes, como o Número de Identificação Pessoal (PIN), o *Application Protocol Data Unit* (*APDU*), o *applet Java Card* (*Applet*), o *Java Card System* (*JCSystem*), e uma classe utilitária. Também define várias constantes ISO7816 e várias exceções específicas de *Java Card*. A Tabela 3.12 resume o conteúdo deste pacote.

Interfaces	ISO7816 define constantes relacionadas a ISO 7816-3 e a ISO 7816-4.
	MultiSelectable identifica *applets* que suportam seleções concorrentes.
	PIN representa um número de identificação pessoal usado para segurança (autenticação).
	Shareable identifica um objeto compartilhado. Objetos que devem estar disponíveis através do *firewall* do *applet* devem ser implementardos nessa interface.
Classes	*AID* define um identificador da aplicação associado a um *Content Provider* de acordo com a norma ISO7816-5, É um atributo obrigatório de um *applet*.
	APDU ou *Application Protocol Data Unit*, é o formato de comunicação entre o *Applet* (interno ao cartão) e o aplicativo cliente (externo ao cartão) em conformidade com a norma ISO7816-4.
	Applet define uma aplicação *Java Card*. Todas as aplicações devem estender essa classe abstrata.
	JCSystem fornece métodos para controlar ciclos de vida, recursos, gerenciamento de transações e compartilhamento e exclusão de objetos internos de um *applet*.
	OwnerPIN é uma implementação da interface PIN.
	Util fornece métodos úteis para manipular *arrays* and *shorts*. Estão incluídos: *arrayCompare(), arrayCopy(), arrayCopyNonAtomic(), arrayFillNonAtomic(), getShort(), makeShort(), setShort()*.
Exceptions	Várias classes de exceções da *Java Card* VM são definidas: *APDUException, CardException, CardRuntimeException, ISOException, PINException, SystemException, TransactionException, UserException*.

TABELA 3.12 - JAVACARD.FRAMEWORK V2.2

FONTE: SUN MICROSYSTEMS [11]

– O *javacard.framework.service* define as interfaces, classes, e as exceções para Services. Um serviço processa comandos recebidos sob a forma de um *APDU*. A Tabela 3.13 resume o *framework* de serviço *API*.

Interfaces	Service, o serviço base de interface define os métodos *processCommand(), processDataIn(),* e *processDataOut()*.
	RemoteService é um *Service* genérico que fornece acesso a processos remotos a serviços no cartão.
	SecurityService estende o serviço base de interface e fornece métodos de consulta ao status de segurança atual, incluindo *isAuthenticated(), isChannelSecure(),* e *isCommandSecure()*.
Classes	BasicService é uma implementação padrão de um *Service*; Fornece métodos que ajudam a lidar com *APDU*s e colaborações de serviços.
	Dispatcher mantém um registro de serviços. Um *dispatcher* é usado quando se quer delegar o processamento de uma *APDU* a vários serviços. Pode processar uma *APDU* completamente com o método *process()*, ou despachá-la para vários serviços com o método *dispatch()*.
Exceptions	*ServiceException* é um serviço relacionado a exceção.

TABELA 3.13 - JAVACARD.FRAMEWORK.SERVICE

FONTE: SUN MICROSYSTEMS [11]

– O *javacard.security* define as classes e interfaces para o framework de segurança *Java Card*. A especificação *Java Card* define uma *API* de segurança robusta que inclui vários tipos de chaves privadas e públicas e de algoritmos, métodos para calcular os controles de redundância cíclica (*Cyclic Redundancy Checks*, *CRCs*), *Message Digest* (*MD4* e *MD5* por exemplo), e assinaturas, apresentadas na Tabela 3.14.

Interfaces	Chaves de interface base genéricas, chaves privadas, chaves públicas, chaves secretas e subinterfaces que representam vários tipos de algoritmos de chaves de segurança: *AESKey, DESKey, DSAKey, DSAPrivateKey, DSAPublicKey, ECKey, ECPrivateKey, ECPublicKey, RSAPri-*

	vateCrtKey, RSAPrivateKey, RSAPublicKey
Classes	*Checksum*: classe base para algoritmos *CRC*
	KeyAgreement: classe base para algoritmos "chaves de acordo
	KeyBuilder: fábrica de objetos chave
	KeyPair: um recipiente para armazenar um par de chaves, uma privada e uma pública
	MessageDigest: classe base para algoritmos *hash*
	RandomData: classe base para geradores de números aleatórios
	Signature: classe abstrata base para algoritmos de assinaturas
Excepti-ons	*CryptoException*: exceções relacionadas a encriptação como por exemplo um algoritmo não suportado ou uma chave não inicializada .

TABELA 3.14 - JAVACARD.SECURITY

FONTE: SUN MICROSYSTEMS [11]

- O *javacardx.crypto* é um pacote de extensão que define a interface de encripatação de chave (*KeyEncription*) e a classe *Cypher*, cada uma em seu pacote para um controle de exportação mais fácil. *KeyEncryption* é usada para descriptografar uma chave inserida por algoritmos de encriptação. *Cypher* é a classe base abstrata em que todos as cifras devem ser implementadas.

- O j*avacardx.rmi* é um pacote de extensão, que define as classes *RMI* de *Java Card*. Ele define duas classes, *CardRemoteObject* e *RMIService*. *CardRemoteObject* define dois métodos, *Export* () e *Unexport* (), para habilitar e desabilitar o acesso remoto a um objeto exterino ao cartão. *RMIService* estende *BasicService* e implementa o *RemoteService* para processar os pedidos *RMI*.

3.6 – *Java Card* Runtime

A especificação *JCRE* define o ciclo de vida da *Java Card VM*, o ciclo de vida do *applet*, a forma através da qual os *applets* são selecionados e isolados uns dos outros, as transações, e a persistência e o compartilhamento de objeto. O *JCRE* fornece uma interface independente de plataforma para os serviços prestados pelo sistema operacional do cartão. Consiste na *Java Card Virtual Machine*, a *API Java Card*, e qualquer extensão específica de vendedores. Esta arquitetura é apresentada na Figura 3.7.

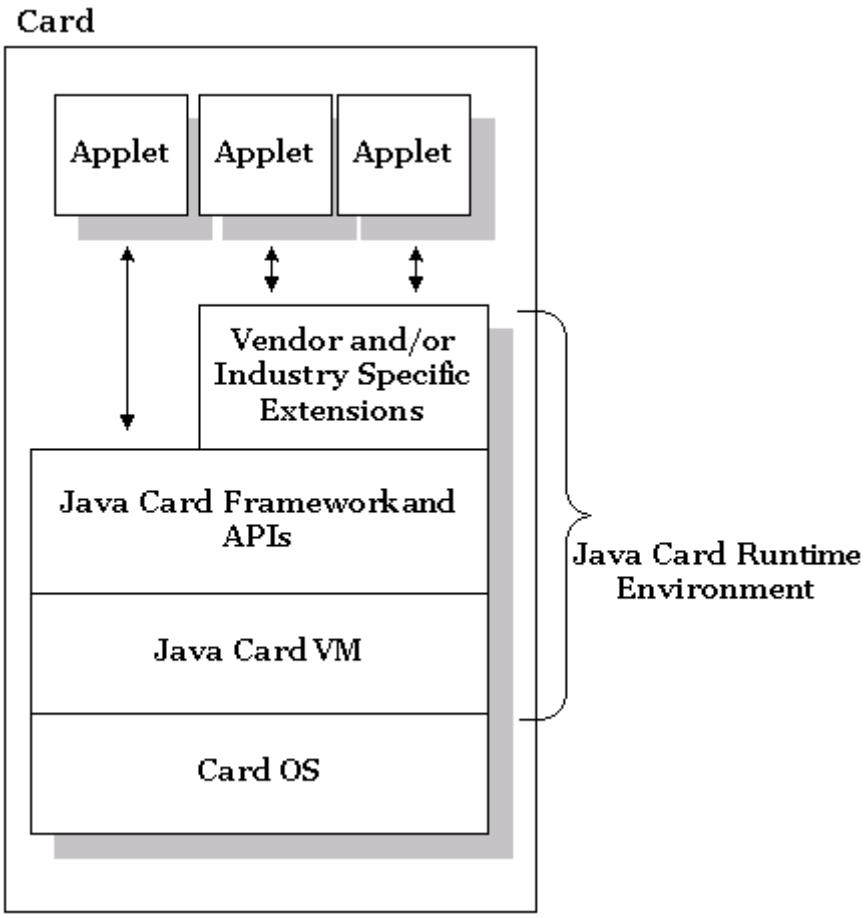


FIGURA 3.7 - ARQUITETURA *JAVA CARD* RUNTIME ENVIRONMENT
FONTE: SUN MICROSYSTEMS, INC. [11]

A vida útil da *JCVM* coincide com a do próprio cartão: Inicia-se em algum momento após o cartão ser fabricado e testado e antes que seja emitido para o titular do cartão, e termina quando o cartão é descartado ou destruído. A *JCVM* não pára quando a energia do cartão é cortada, já que o seu estado é mantido na memória não-volátil do cartão. Ao iniciarmos a *JCVM* inicializamos o *JCRE* e criamos todos os objetos do framework *JCRE*, que sobrevive durante toda a vida útil da *JCVM*. Depois de ter começado a *JCVM*, todas as interações com o cartão são, em princípio, controlada por um dos *applets* no cartão. Quando a energia é cortada, todos os dados contidos na *RAM* são perdidos, mas qualquer estado armazenado em memória não-volátil é mantido. Quando a energia é reaplicada, a *VM* se torna ativa novamente, momento em que os estados da *VM* e dos objetos são restaurados, e a execução e recomeça à espera de mais contributos.

Cada *applet* em um cartão é identificado exclusivamente por um *Application ID* (*AID*). Uma *AID*, tal como definido na ISO 7816-5, é uma seqüência entre 5 e 16 *bytes*.

Todos os *applets* devem estender a classe base abstrata dos *Applet*, que define os métodos utilizados pela *JCRE* para controlar o ciclo de vida do *applet*, como resumidos na Figura 3.8.

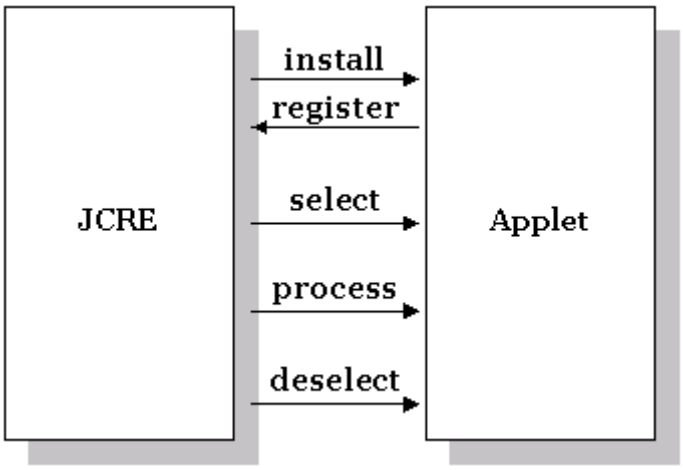


FIGURA 3.8 - MÉTODO DE CICLO DE VIDA DE UM *APPLET* DE *JAVA CARD*
FONTE: SUN MICROSYSTEMS, INC. [11]

O ciclo de vida do *applet* começa quando o *applet* é transferido para o cartão e o *JCRE* invoca o método estático *Applet.install* () do *Applet*, e o *applet* se registra com o *JCRE* invocando o método *Applet.register* (). Uma vez que o aplicativo é instalado e registrado, ele está no estado não-selecionado, disponível para seleção e para o processamento de *APDU*. A Figura 3.9 resume o funcionamento dos métodos de *Applets*.

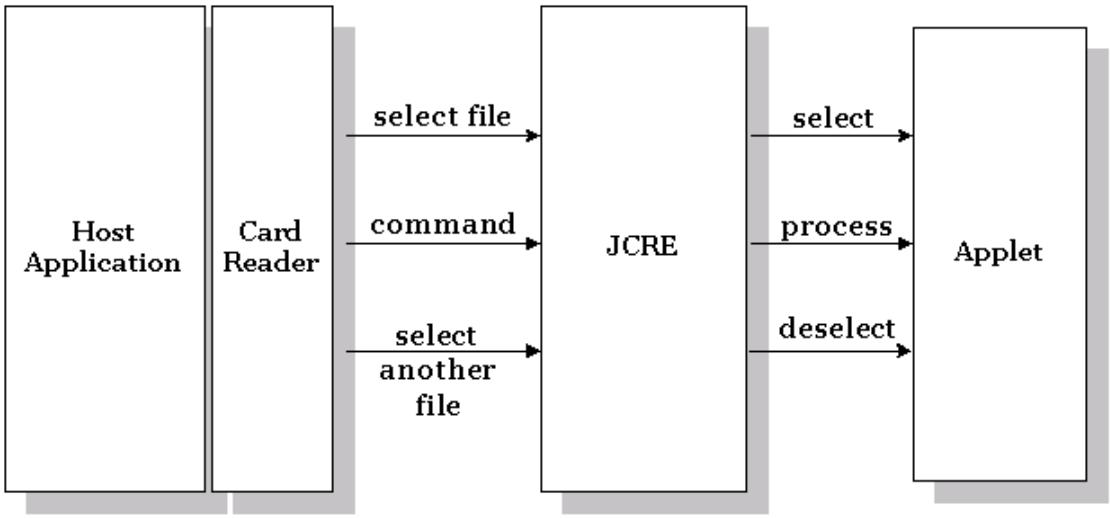


FIGURA 3.9 - USANDO MÉTODOS DE *APPLETS* DE *JAVA CARD*
FONTE: SUN MICROSYSTEMS, INC. [11]

Enquanto no estado não-selecionado, o *applet* está inativo. Um *applet* fica selecionado para processamento de *APDU* quando o aplicativo cliente pede ao *JCRE* para selecionar um *applet* específico no cartão (instruindo o leitor de cartão a enviar um *SELECT APDU* ou *MANAGE CHANNEL APDU*). Para notificar o *applet* que um aplicativo cliente o selecionou, o *JCRE* faz a chamada do método *select();* o *applet* normalmente executa uma inicialização adequada na preparação do processamento de *APDU*.

Depois que a seleção é feita, o *JCRE* passa os comandos *APDU* recebidos para o *applet* para processamento invocando o seu método *process*(). O *JCRE* captura quaisquer exceções que um *applet* não consegue pegar.

O descarte do *applet* ocorre quando o aplicativo cliente avisa ao *JCRE* para selecionar outro *applet*. O *JCRE* notifica o *applet* ativo que ele foi desselecionado pela chamada do método *deselect* (), que normalmente realiza qualquer tipo de lógica de limpeza (*clean-up logic*) e retorna o *applet* para o estado inativo e não selecionado.

Uma sessão de cartão (*card session*) é o período de tempo em que o cartão é ativado e troca *APDU*s com o leitor de cartão. O *Java Card* 2.2 suporta o conceito de canais lógicos que permitem a abertura simultânea de até 16 sessões em um *smart card*, com uma sessão por canal lógico. Como o processamento de uma *APDU* não pode ser interrompido, e cada *APDU* contém uma referência a um canal lógico (no *byte* CLA), com a alternância de *APDU*s é possível acessar pseudo-simultaneamente uma série de *applets* no cartão. Podemos projetar um *applet* para ser multiselecionável, isto é, podemos fazer com que ele se comunique com mais de um canal lógico em uma única vez. *Applets* multiselecionáveis devem implementar a interface *javacard.framework.MultiSelectable* e seus métodos correspondentes.

Em algumas implementações de cartão, um *applet* padrão pode ser instruído a ser selecionado automaticamente depois que o cartão for reiniciado, para a comunicação no canal base lógico (canal 0). *Java Card* 2.2 permite definir *applets* padrões, mas não especifica como, o mecanismo é o vendedor quem específica.

A plataforma *Java Card* é uma aplicação segura multi-ambiente. Muitas aplicações diferentes de diferentes fornecedores podem coexistir com segurança no mesmo cartão. Cada *applet* é atribuído a um contexto de execução que controla o acesso aos objetos que lhe estão atribuídos. O limite entre um contexto de execução e outro é frequentemente chamado de *firewall* do *applet*. É uma melhoria do *Java Card Runtime* sobre o conceito de uma "*sandbox*" de segurança Java (*Sandbox* é o conceito de que uma aplica-

ção, seja qual for o ambiente, tem definições pré-estabelecidas do que pode ou não ser feito. Cada ambiente possui a sua "caixa de areia" específica, com suas permissões e proibições), e combina as funcionalidades da classe que faz o load, *java.ClassLoader*, e do controle de acesso*, java.AccessController*.

O *firewall Java Card* cria um *heap* que faz com que um objeto possa acessar (publicamente) os métodos e os dados dos objetos que estão dentro de um mesmo *firewall*. Um *firewall* pode conter um número de *applets* e outros objetos, tais como chaves secretas comuns. Um contexto de execução *Java Card* atualmente tem escopo de pacote. Quando cada objeto é criado, ele é atribuído ao contexto de execução do chamador.

A plataforma *Java Card* suporta compartilhamento seguro de objetos através de *firewalls*. A figura 3.10 representa o isolamento do *applet* e o compartilhamento de objetos.

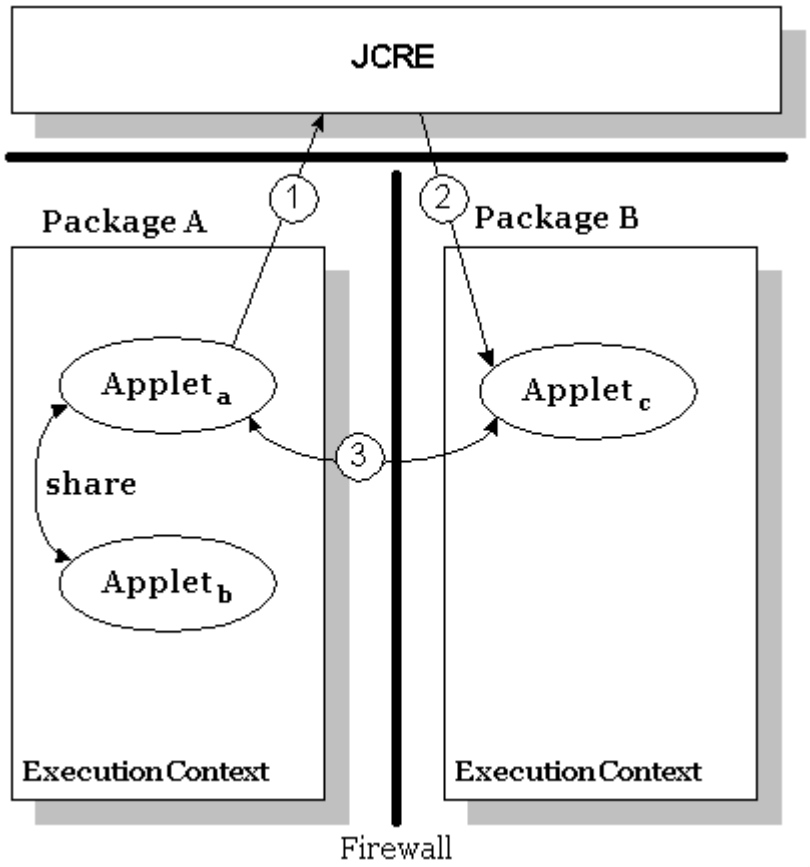


FIGURA 3.10 - ISOLAMENTO DO *APPLET* E COMPARTILHAMENTO DE OBJETOS
FONTE: SUN MICROSYSTEMS, INC. [11]

O fluxo típico, conforme representado na figura 3.10, pode ser descrito como se segue.

1. *Applet* A pede acesso a interface compartilhável do *Applet* C chamando o método do sistema *JCSystem.getAppletshareableInterfaceObject* ().
2. Por parte do *Applet* A, o *JCRE* faz o pedido da interface compartilhável do *Applet* C, chamando o método *getShareableInterfaceObject* () do *applet*.
3. Se o *Applet* C permitir o compartilhamento, *Applet*A irá obter uma referência aos objetos compartilhados do *Applet*C. Agora o *applet* A tem acesso ao *Applet* C. O *applet*A será dono de qualquer objeto que criar, inclusive os criados no *applet*C.

Applets no mesmo contexto de execução têm acesso uns aos outros por padrão, por isso *Applet* A e *Applet* B não precisam seguir este procedimento para compartilhar objetos.

Em um dispositivo *Java Card* a memória é o recurso mais valioso. Em algumas implementações *Java Card* um *garbage collector* pode não estar disponível. Quando um objeto é criado, o objeto e seu conteúdo são preservados em memória não-volátil, tornando-o disponível entre as sessões. Em alguns casos, os dados não precisam ser persistentes: são temporários ou transitórios. Para reduzir o desgaste da memória persistente de um *smart card* e assim maximizar o seu tempo de vida, é aconselhável tratar tanto quanto for possível os dados que são atualizados com freqüência como transitórios.

A tecnologia *Java Card* não suporta a palavra-chave *transient*. Em vez disso, a *API* de *Java Card* (*javacard.framework.JCSystem*) define três métodos que permite que sejam criados dados transitórios em tempo de execução, e um quarto que lhe permite verificar se um objeto é transitório:

- *static byte[] makeTransientByteArray(short length, byte event)*
- *static Object makeTransientObjectArray(short length, byte event)*
- *static short[] makeTransientShortArray(short length, byte event)*
- *static byte isTransient(java.lang.Object theObj)*

O *JCRE* suporta transações atômicas para atualizar um ou mais objetos persistentes seguramente. Transações garantem a integridade dos dados em caso de perda de energia ou erros no programa. Transações são suportadas em nível de sistema, pelos seguintes meios:

- *JCSystem.beginTransaction()*

- *JCSystem.commitTransaction()*
- *JCSystem.abortTransaction()*

Em um padrão comum a muitos modelos de transação, uma transação *Java Card* começa com uma chamada a *beginTransaction (),* e termina com uma chamada a *commitTransaction ()* ou *abortTransaction ().*

A atualização do balanço das variáveis instanciadas é garantida por uma operação atômica. Se um erro no programa ou uma reinicialização ocorrer, a transação garante que o último valor de balance seja restaurado.

Capítulo 4

Desenvolvendo uma aplicação *Java Card*

Os passos típicos para a criação de uma aplicação *Java Card* são:

1. Escrever o código Java.
2. Compilar o código fonte.
3. Converter os arquivos das classes em arquivo(s) de *Applets* Convertidos (*CAP*).
4. Verificar se o *CAP* é válido; este passo é opcional.
5. Instalar o arquivo *CAP*.

Os primeiros dois passos são os mesmos que fazemos ao desenvolver programas tradicionais na linguagem de programação Java: escrever arquivos "java" e compilá-los em arquivos "*class*". Uma vez que são criados arquivos de classe *Java Card*, no entanto, o processo muda.

A Java Card Virtual Machine é dividida em uma *JVM* interna ao cartão e outra *JVM* externa ao cartão. Essa divisão faz com que operações custosas fiquem externas ao cartão e permite um pequeno vestígio de memória dentro do cartão, porém são necessários alguns passos a mais ao desenvolver aplicações de *Java Card.*

Antes das classes *Java Card* serem carregadas em um dispositivo *Java Card*, elas devem ser convertidas para o formato padrão de arquivo *CAP* e, que em seguida, podem ser opcionalmente verificadas.

A conversão implica transformar cada pacote Java em um arquivo *CAP*, que contém a representação da combinação binária de classes e interfaces em um pacote. Esta conversão é uma operação externa ao cartão.

A verificação é um processo facultativo para validar o arquivo *CAP* em termos de estrutura, subconjunto de *byte*codes válidos, e interdependências de pacote. Podem ser feitas verificações sobre pacotes ou ferramentas de conversões de terceiros. A verificação é tipicamente uma operação externa ao cartão, mas alguns produtos do cartão podem incluir um verificador "*on-board*". Uma vez verificada, o arquivo *CAP* está pronto para ser instalado no dispositivo *Java Card*.

Escolhemos para o desenvolvimento do ambiente e dos aplicativos *Java Card* as seguintes ferramentas:

- Eclipse
- *JavaTM 2 SDK, Standard Edition Version 1.4.2*
- *Java Card Development Kit 2.2.2*
- EclipseJCDE

A escolha dessas ferramentas se deve ao fato de serem gratuitas e facilmente encontradas na internet, ou seja, de fácil acesso a qualquer pessoa.

Eclipse é uma comunidade "*open source*", cujos projetos são direcionados em construir uma plataforma de desenvolvimento aberta constituída de *frameworks*, ferramentas e "*runtimes*" para construir, desenvolver e gerenciar *softwares*.

JavaTM 2 SDK é uma versão da biblioteca do *kit* de desenvolvimento de *software* Java (*Java Software Development Kit*) na versão 2.2

Java Card Development Kit proporciona um ambiente de desenvolvimento, onde aplicações em *Java Card* podem ser desenvolvidas e testadas. O pacote inclui:

- Ferramentas de programação essenciais para o desenvolvimento de aplicações em *Java Card*.
- Componentes de simulação e emulação completamente compatíveis com a última especificação em *Java Card*, incluindo as mais novas funcionalidades criptográficas.

EclipseJCDE é um conjunto de *plug-ins* envolvendo o *Java Card Development Kit* da *Sun Microsystems* para proporcionar uma interface visual no ambiente de desenvolvimento, automatizando muitos dos passos necessários para desenvolver uma aplicação em *Java Card*.

4.1 – Instalação do Eclipse

Para obter a ferramenta acessar o seguinte site: http://www.eclipse.org/downloads. Escolher a opção "*Eclipse Classic*" e posteriormente um dos espelhos para realizar o *download*.

FIGURA 4.1 – ECLIPSE DOWNLOAD
FONTE: ECLIPSE FOUNDATION, THE [15]

FIGURA 4.2 – ECLIPSE MIRROR
FONTE: ECLIPSE FOUNDATION, THE [15]

Em seguida salve a aplicação na pasta que preferir.

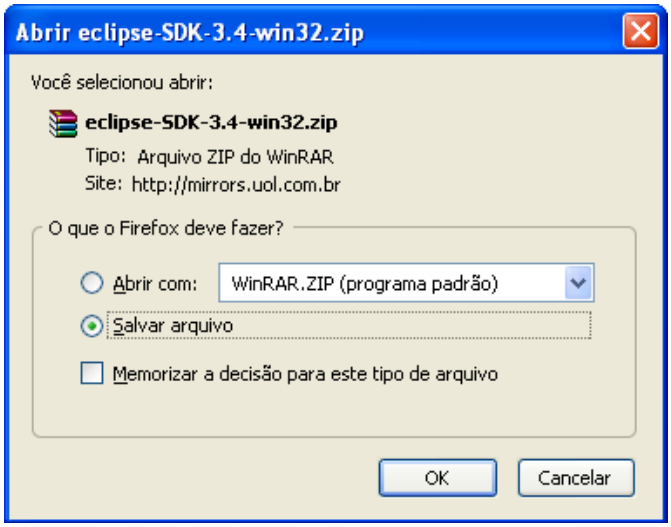
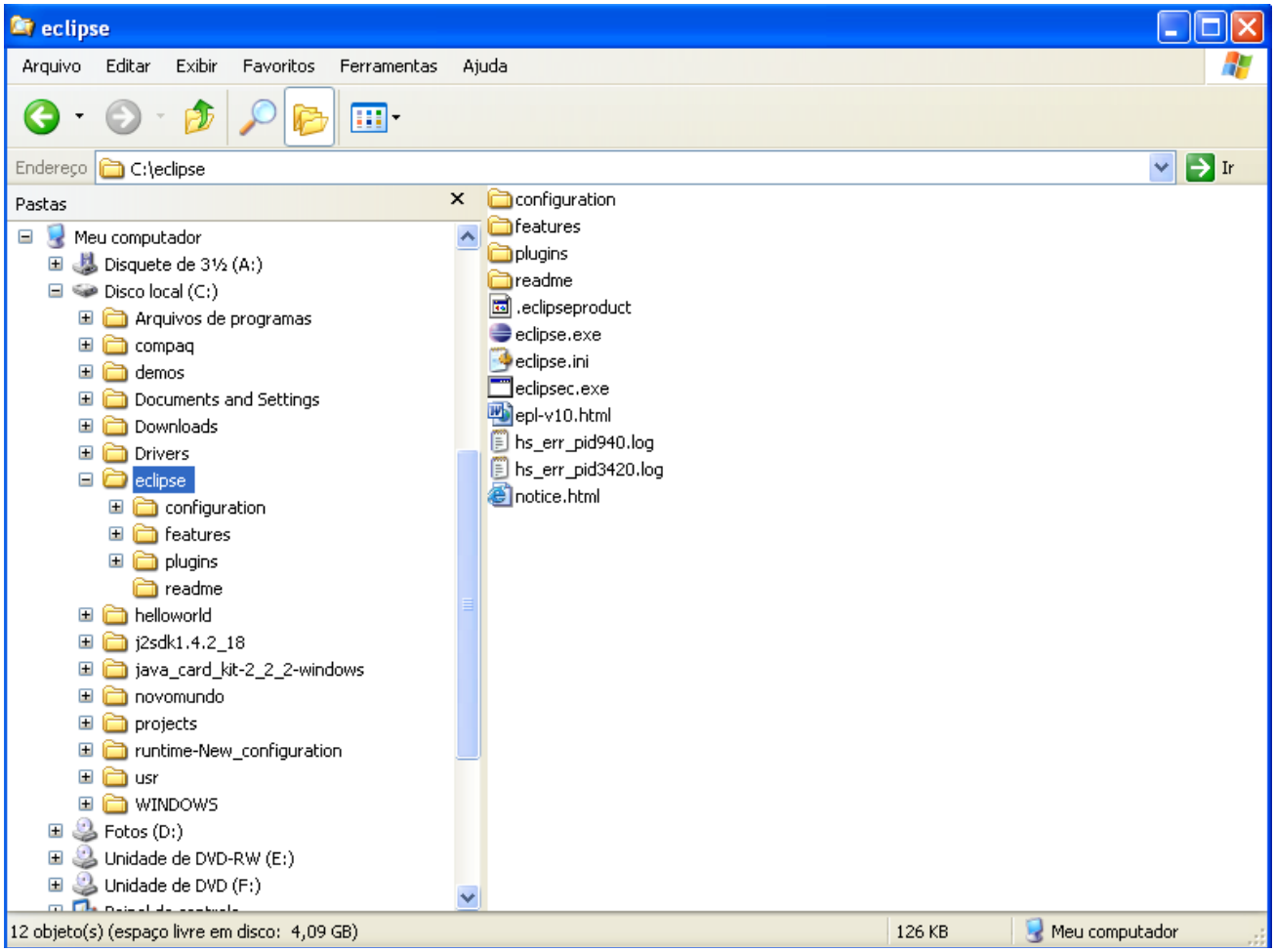


FIGURA 4.3 – PASTA DE DESTINO

Eclipse irá ser executado a partir da pasta em que o arquivo foi descompactado. No exemplo, extraído para o disco local.



ECLIPSE

4.2 – Instalação do *Java Card* Development Kit

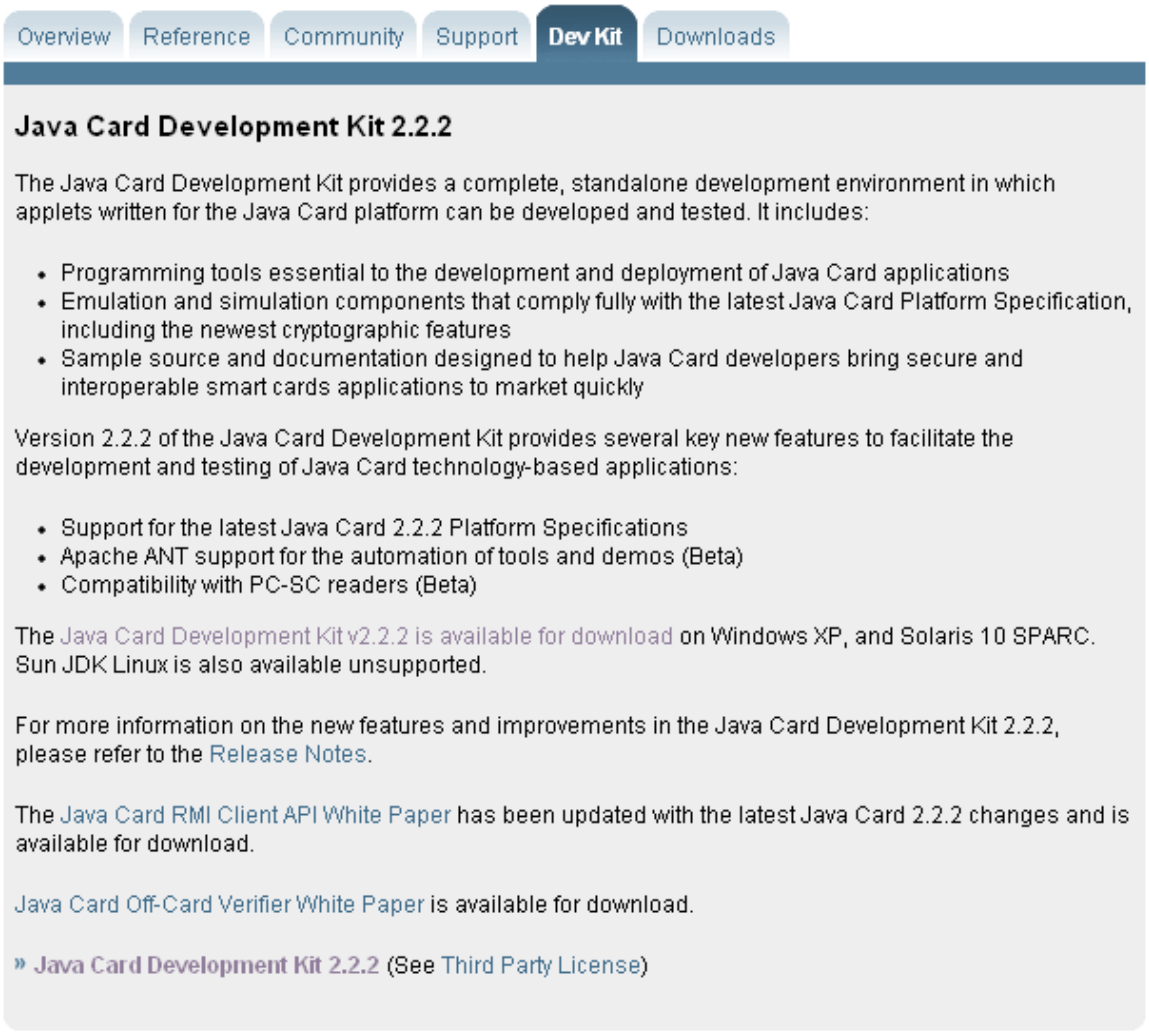
Primeiro, deve ser realizado o *download* desta ferramenta no endereço http://java.sun.com/j2se/1.4.2/download.html, clicando em "*Download J2SE SDK*".

FIGURA 4.5 – DOWNLOAD J2SE SDK
FONTE: SUN MICROSYSTEMS [16]

A seguir, a seguinte página será exibida, escolher a plataforma onde a ferramenta será executada Deve-se aceitar os termos da licença e clicar em "*Continue*" para realizar o *download*.

FIGURA 4.6 – ESCOLHA DA PLATAFORMA
FONTE: SUN MICROSYSTEMS [16]

Após este *download*, deve-se realizar um novo *download*, desta vez do kit de desenvolvimento disponível em http://java.sun.com/javacard/devkit/index.jsp. Para isto, deve-se clicar em "*Java Card Development Kit 2.2.2*", conforme figura abaixo.

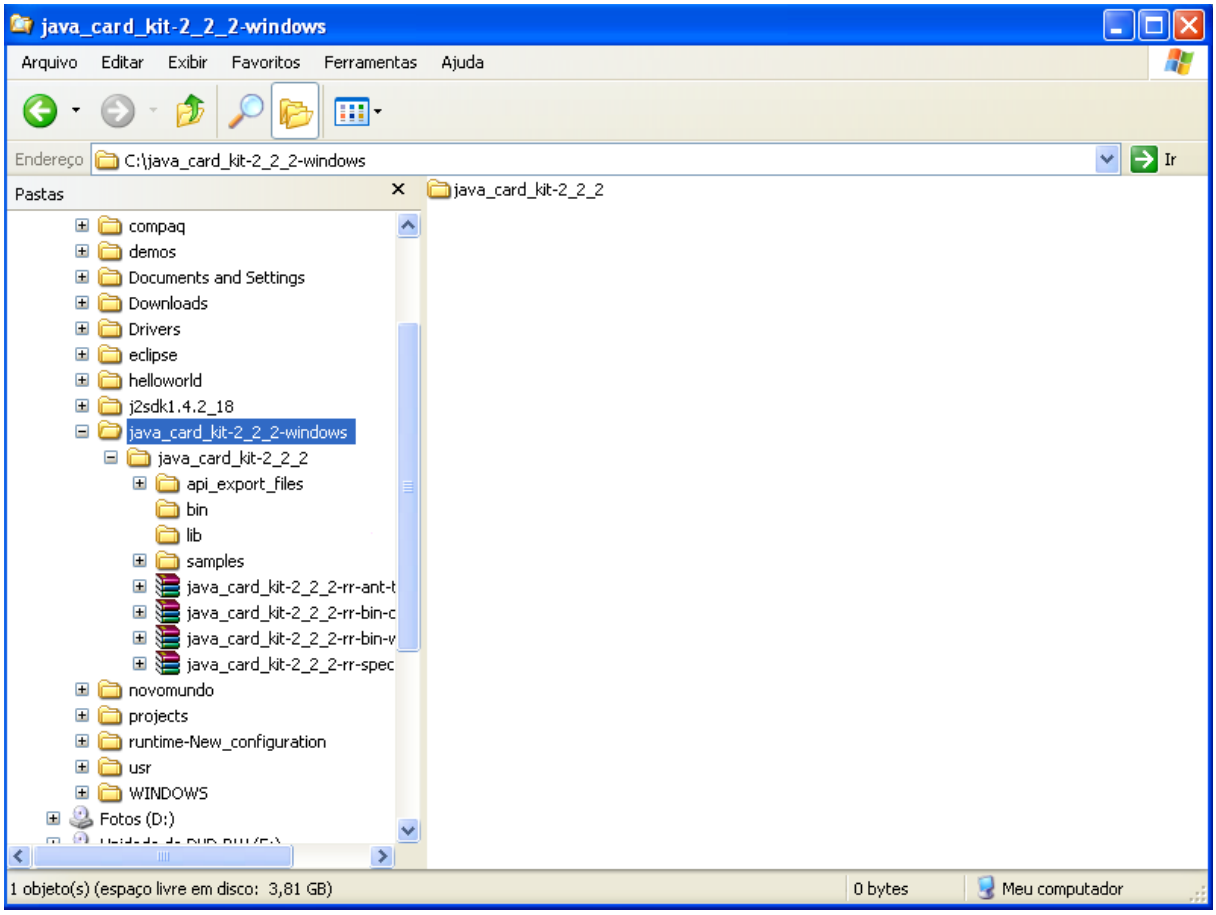


JAVA CARD DEV KIT 2.2.2
FONTE: SUN MICROSYSTEMS [16]

Escolha a plataforma onde o aplicativo será executado, aceite a licença e clique em "*Continue*".

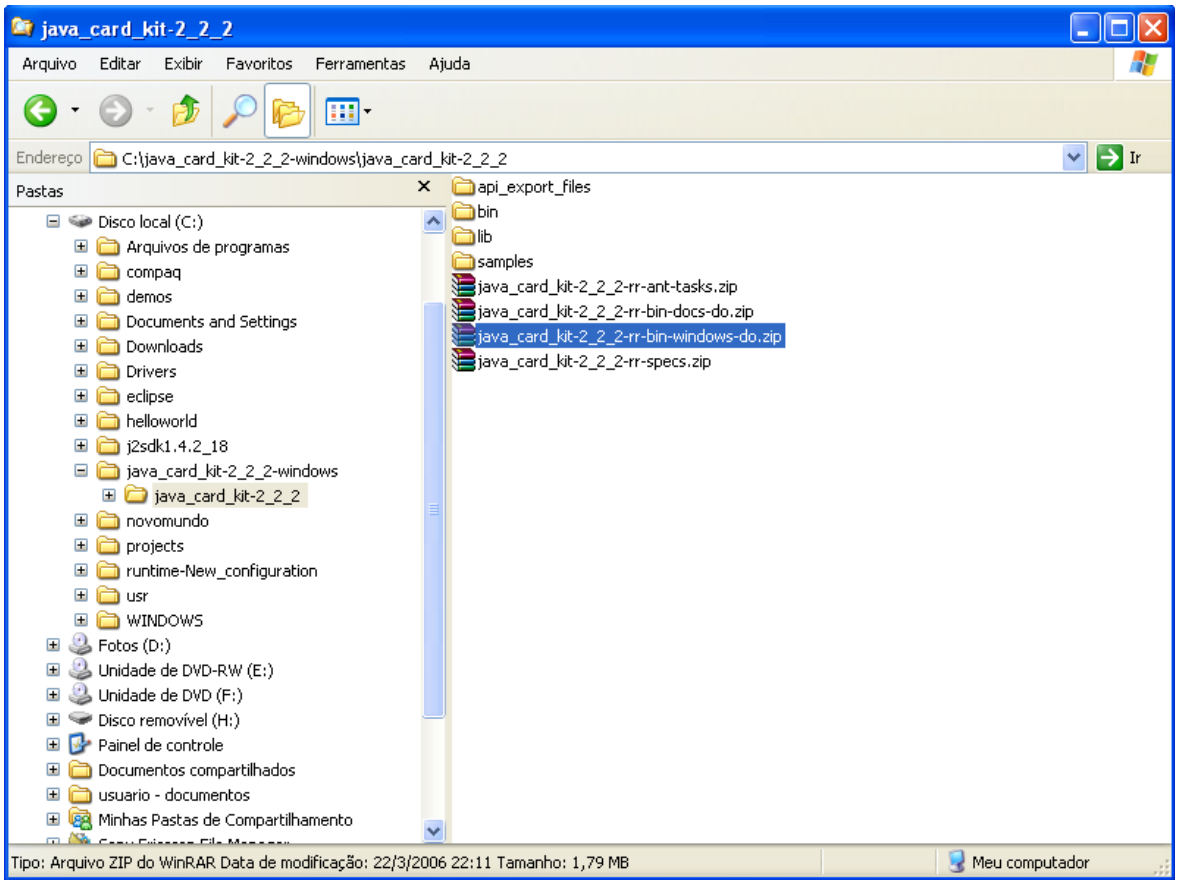
FIGURA 4.8 – *JAVA CARD* DEVELOPMENT KIT 2.2.2
FONTE: SUN MICROSYSTEMS [16]

Uma vez que o download tenha sido realizado, a pasta “java_card_kit-2_2_2”, deve ser descompactada no caminho escolhido. No exemplo abaixo foi descompactada em “C:\java_card_kit-2_2_2-windows”.



JCDK

Também deve ser descompactado o conteúdo do arquivo “java_card_kit-2_2_2-rr-bin-windows-do.zip” na pasta “java_card_kit-2_2_2”.



DO.ZIP

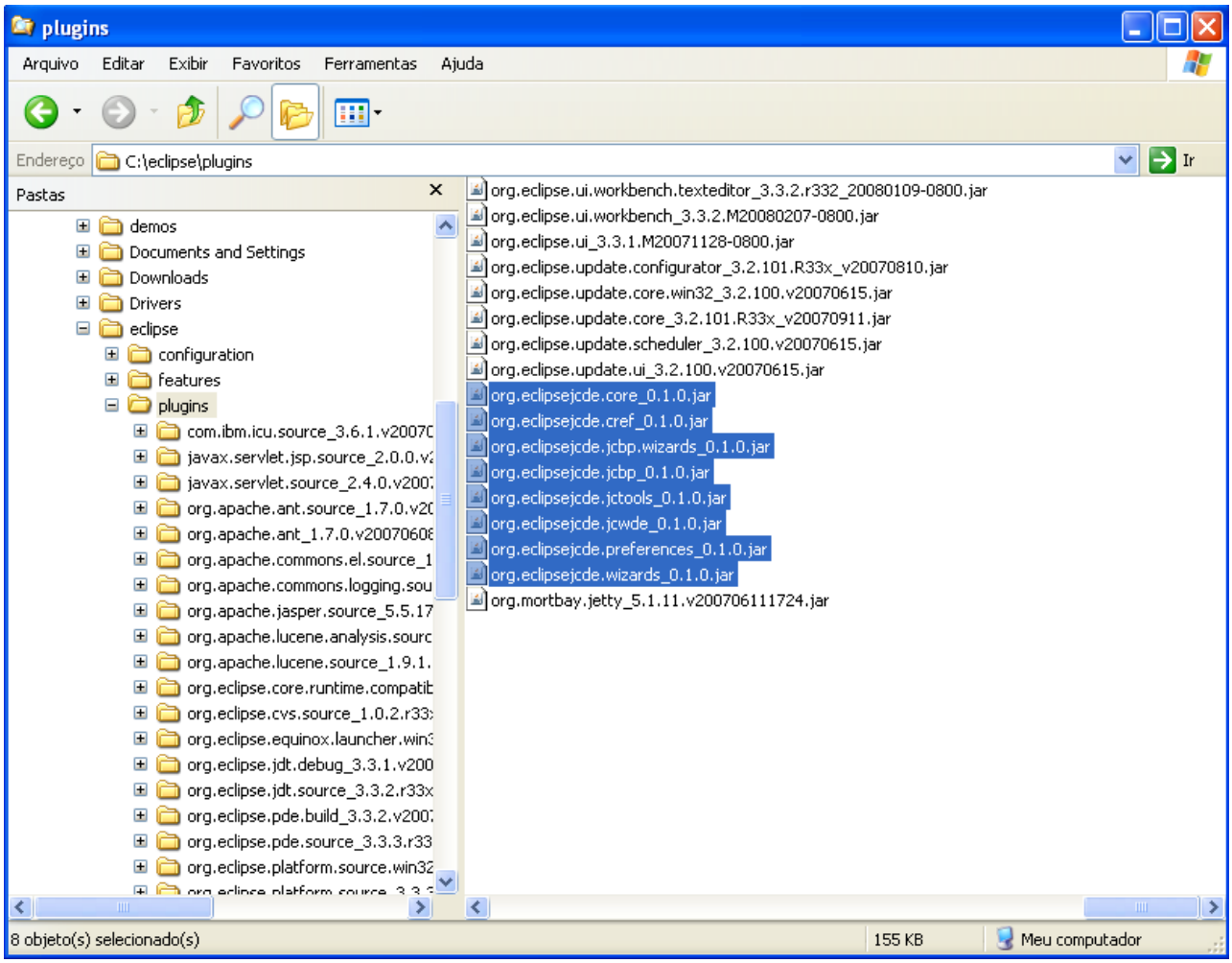
4.3 – *Java Card* Development Environment

Realize o *download* da ferramenta no endereço http://sourceforge.net/projects/eclipse-jcde/ clicando em em *download*, para baixar o arquivo.

Package	Release	Date	Notes / Monitor	Downloads
eclipse-jcde	eclipse-jcde-0.1	September 22, 2006		Download

FIGURA 4.11 – DOWNLOAD JCDE
FONTE: SOURCEFORGE, INC [17]

Uma vez que o arquivo esteja baixado, descompactar o conteúdo na pasta ".../eclipse/plugins". No exemplo "C:\eclipse\plugins".



"../ECLIPSE/PLUGINS"

4.4 – Projeto *Java Card* Development Environment

Uma vez que as ferramentas estejam instaladas o ambiente de desenvolvimento está pronto para um projeto em *Java Card*. Ao iniciar o Eclipse pela primeira vez, a seguinte tela será exibida:

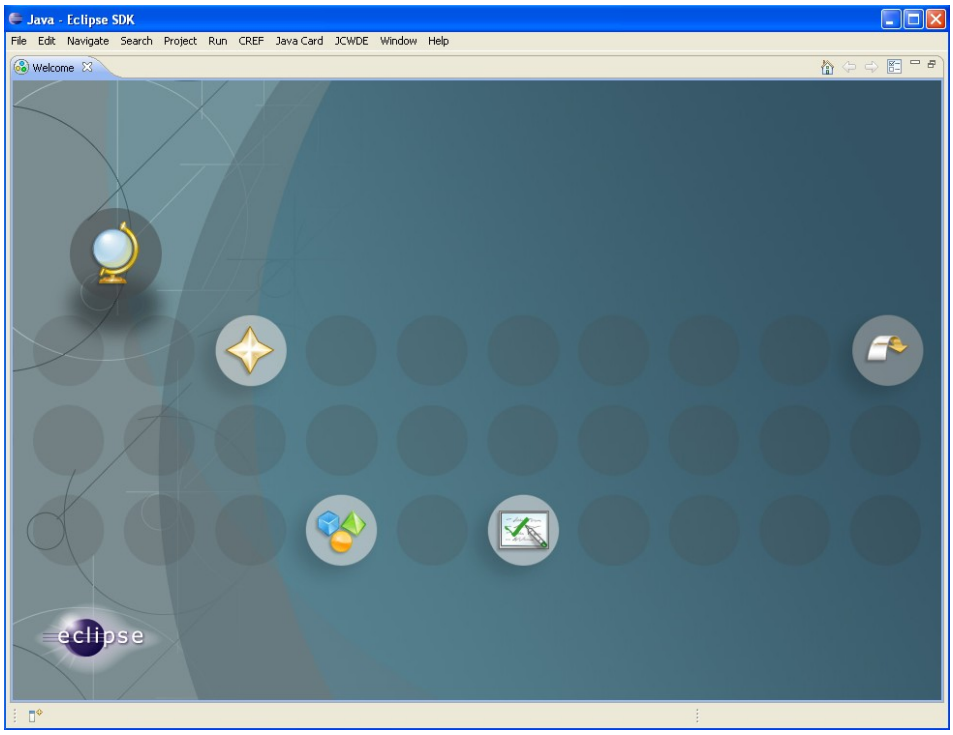


FIGURA 4.13 – JANELA INICIAL DO ECLIPSE

Se o *plugin* foi instalado com sucesso novas opções deverão aparecer no menu principal, como "*Java Card*" e "*JCWE*". Clicar em "*Go to Workbench*".

Para criar um projeto em *Java Card* deve-se clicar na seta ao lado da figura de projeto e selecionar "*Other...*". Escolher a pasta "*Java Card*" e depois em "*Java Card Project*".

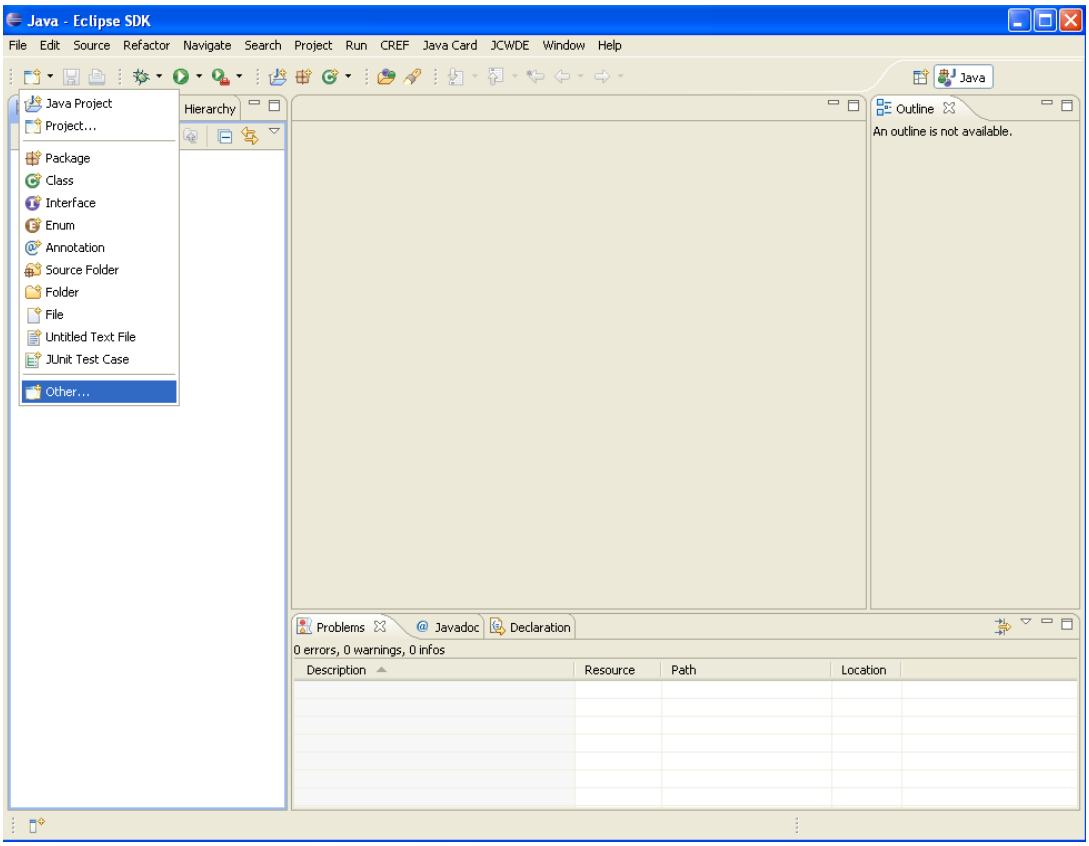


FIGURA 4.14 – NOVO PROJETO NO ECLIPSE

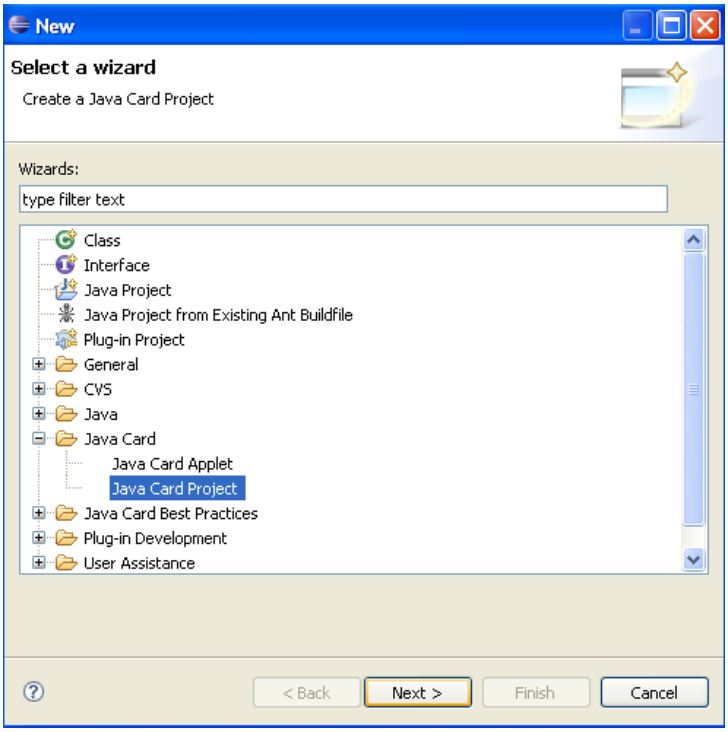


FIGURA 4.15 – NOVO PROJETO *JAVA CARD*

Ao pressionar em "*next*" uma nova janela deverá aparecer para que seja colocado o nome para o projeto.

Após colocar um nome para o projeto clicar em "*Next*", um alerta irá aparecer para que seja especificado o diretório para o *JCDK*. Uma nova janela será exibida e deve-se escolher o diretório onde o *JCDK* foi instalado. No caso "C:\java_card_kit-2_2_2-windows\ java_card_kit-2_2_2".

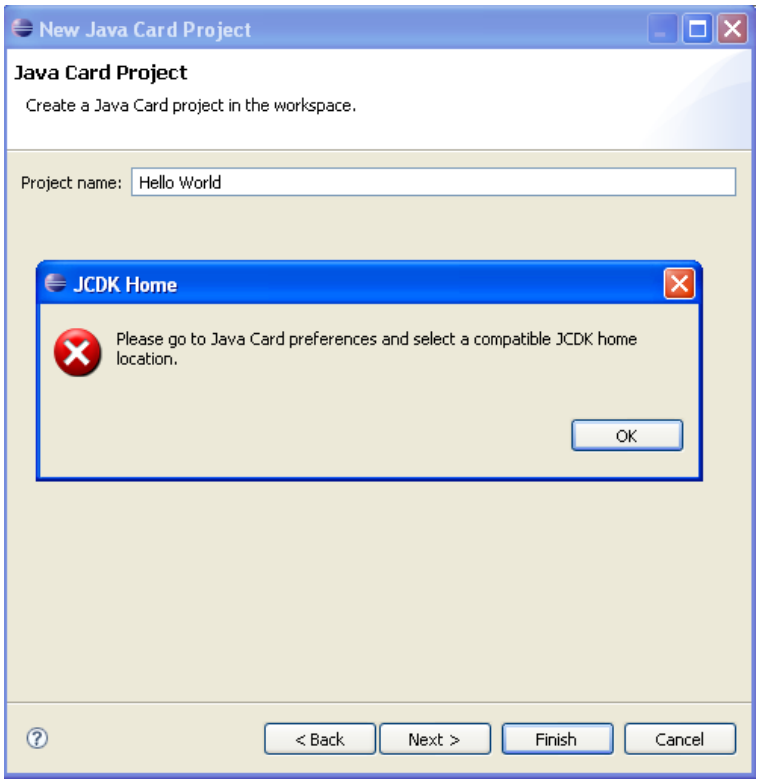


FIGURA 4.16 – ALERTA PARA DEFINIR CAMINHO PARA DIRETÓRIO JCDK

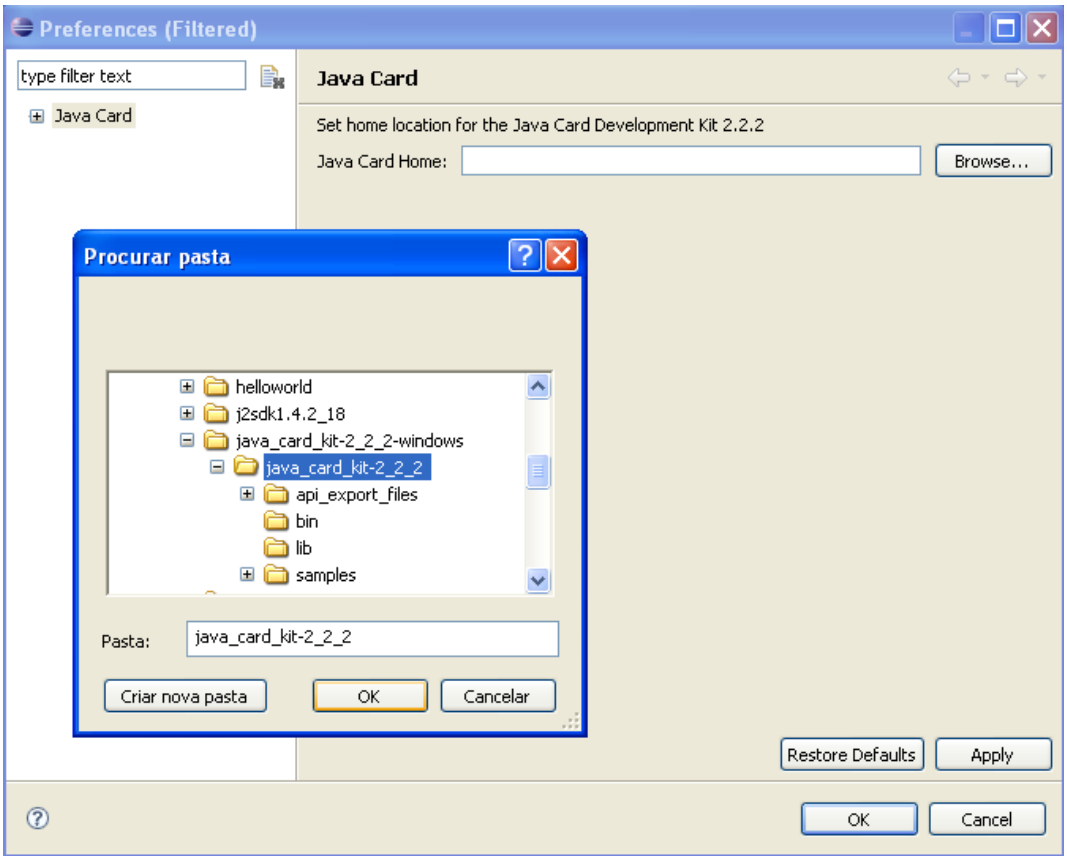


FIGURA 4.17 – SELECIONANDO CAMINHO PARA O JCDK

Basta clicar em "*Finish*" e um novo projeto será criado no "*workspace*" escolhido.

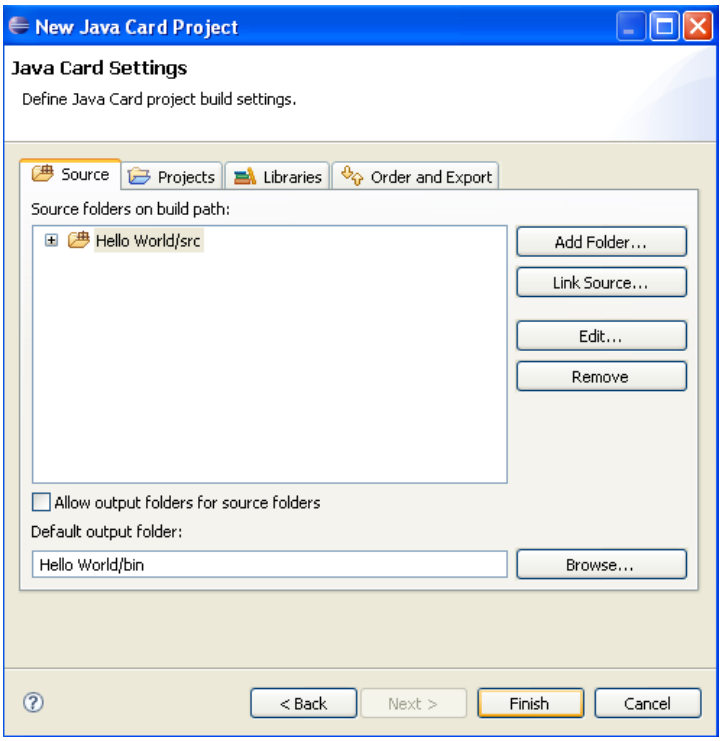
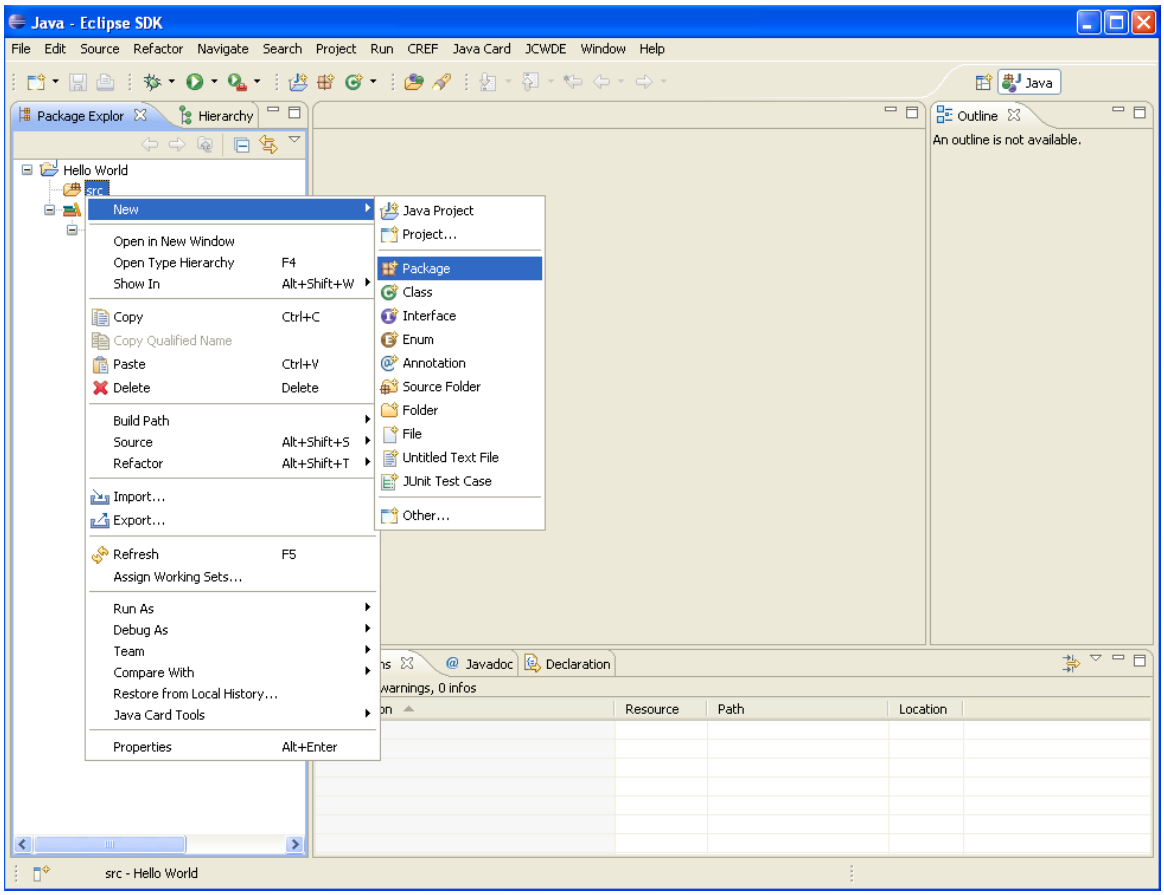


FIGURA 4.18 – CRIAÇÃO DO WORKSPACE

O próximo passo é a criação de um pacote para o projeto, que irá conter a aplicação desenvolvida. Para criar um pacote deve-se clicar com o botão direito do mouse na pasta "src", na área de desenvolvimento. Um menu será exibido onde deverá ser escolhida a opção "*new*" e posteriormente "*package*", como ilustrado na figura abaixo.



PACOTE

Assim como na criação do projeto um nome para o pacote deve ser especificado e ao clicar em "*Finish*", o pacote será criado.

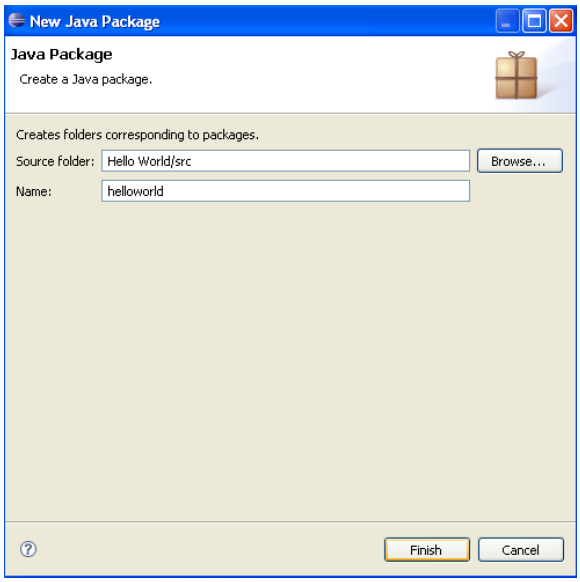
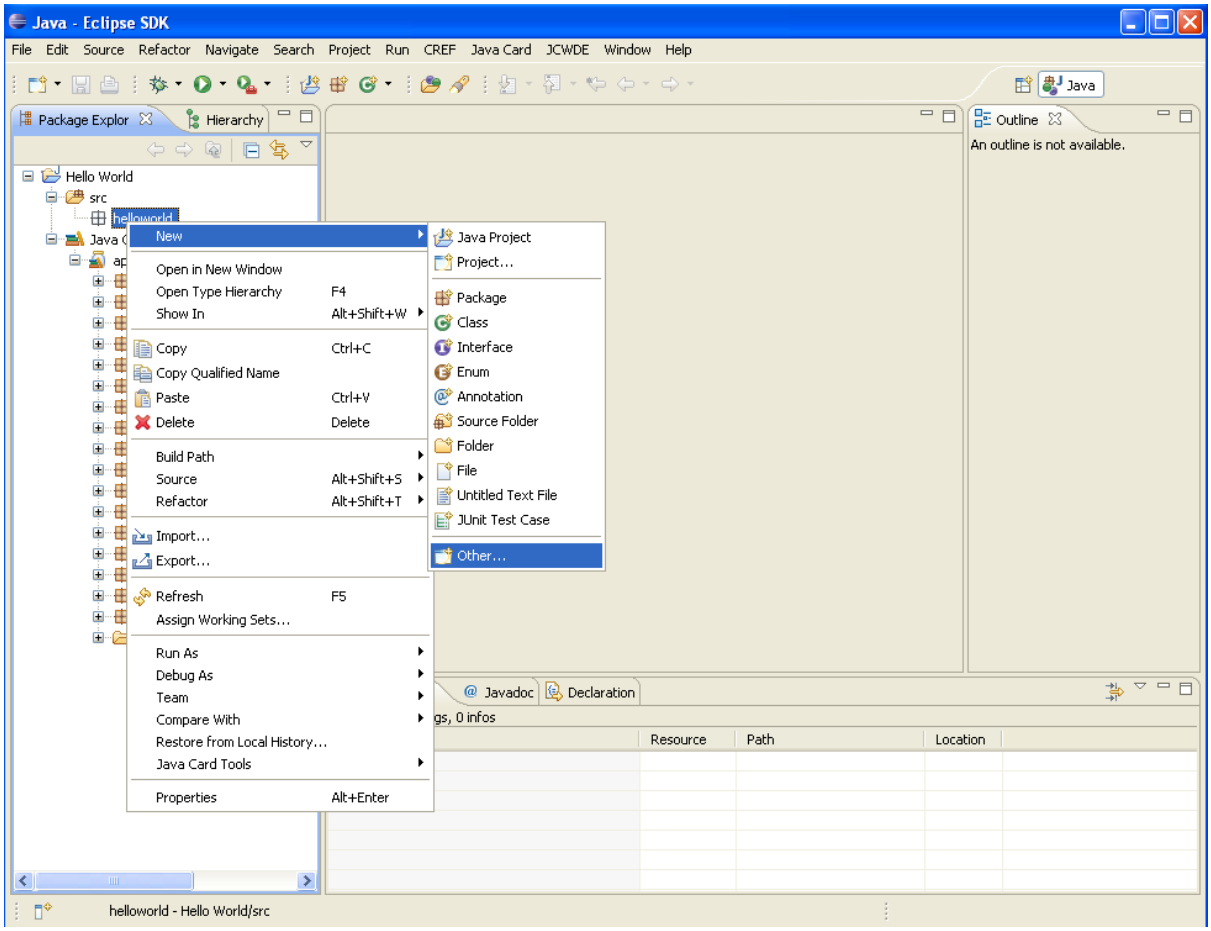


FIGURA 4.20 – DEFINIÇÃO DO NOME DO PACOTE

Após a criação do projeto e o pacote o ambiente está preparado para a criação do *applet* propriamente dito, que é onde se desenvolverá as classes necessárias para a aplicação.

Para que o *applet* seja criado deve-se clicar com o botão direito do *mouse* no pacote e será exibido um menu onde deverá ser escolhida a opção "*new*" e posteriormente "*other*...". Uma nova janela será mostrada e nela selecionar a pasta "*Java Card*" e posteriormente "*Java Card Applet*".



APPLET

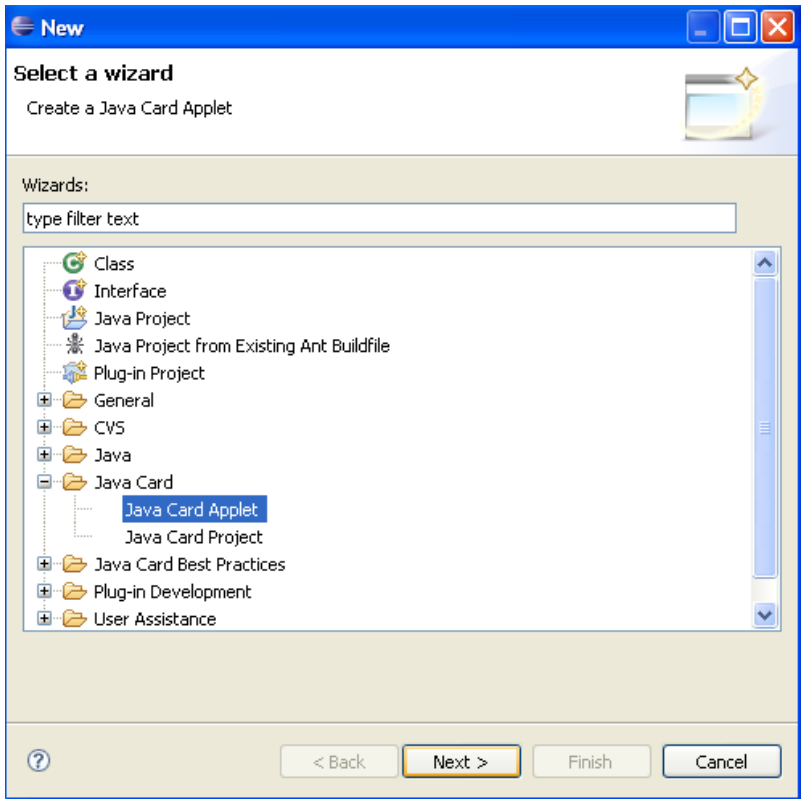


FIGURA 4.22 – CRIANDO UM NOVO *APPLET*

Ao clicar em "*next*", um nome para o *applet* deve ser especificado assim como seu identificador, o *AID*. Uma vez que os parâmetros estejam definidos clicar em "*Finish*" e a estrutura básica para um *applet* em *Java Card* será criada.

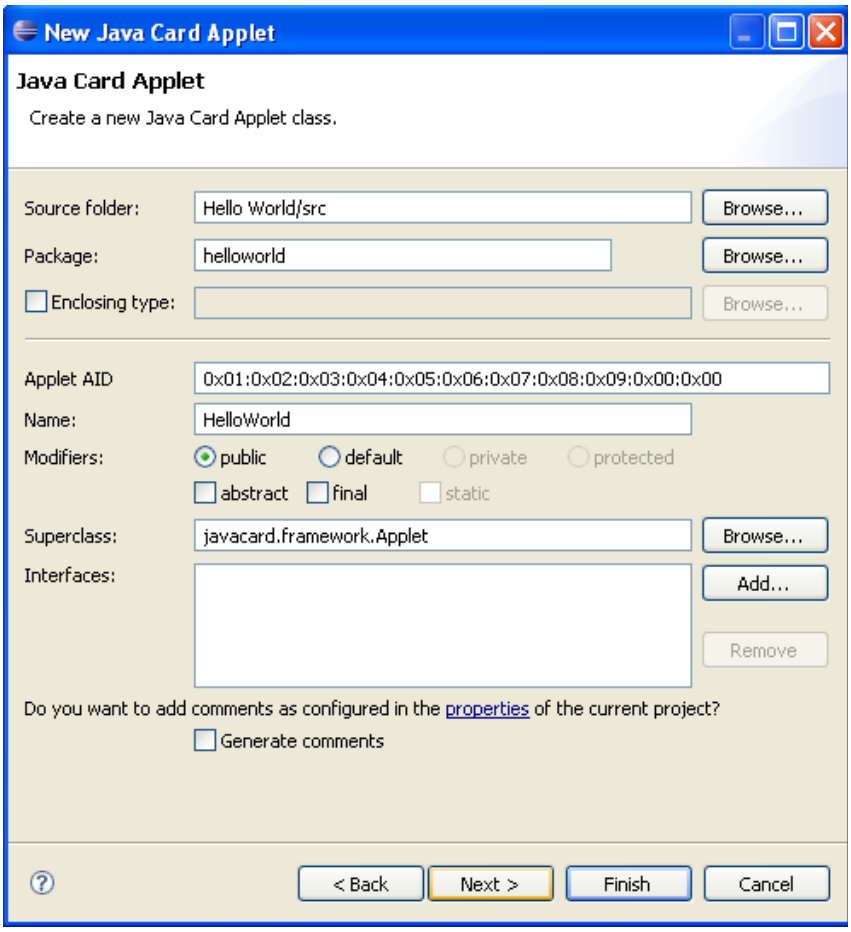


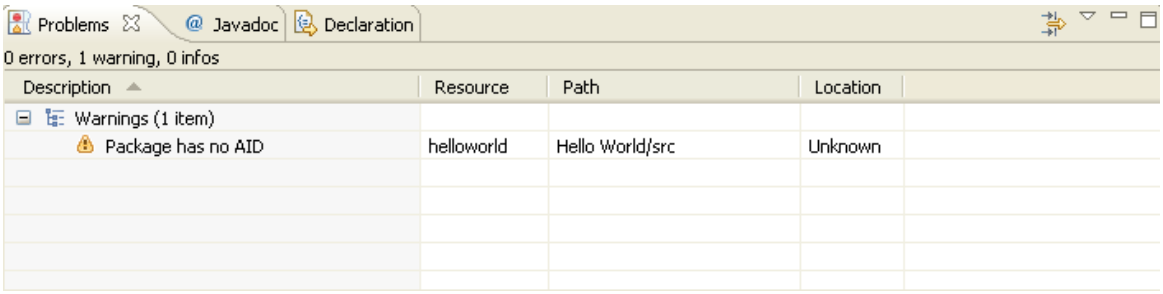
FIGURA 4.23 – DEFINIÇÕES DO *APPLET*

Agora que o ambiente de desenvolvimento está pronto a aplicação será desenvolvida. Neste documento para efeitos de teste será mostrado o desenvolvimento da versão "*Hello World*" para *Java Card*.

4.5 – Hello World

Uma vez que o *applet* tenha sido criado para um determinado pacote, o primeiro passo é definir um *AID* para o pacote. O *AID* do pacote em geral é igual ao do *applet* com exceção do último *byte*. Pelo padrão os 5 primeiros *bytes* devem ser iguais.

Na aba "*Problems*" do Eclipse podemos verificar que é acusada a falta do *AID* do pacote.



PACOTE

Para definir um *AID* para o pacote deve-se selecionar o pacote e ir em "*Java Card*" na barra de ferramentas e posteriormente em "*Set Package AID*". Uma janela será exibida onde o *AID* pode ser especificado, um identificador já é sugerido com base no *AID* especificado na criação do *applet*.

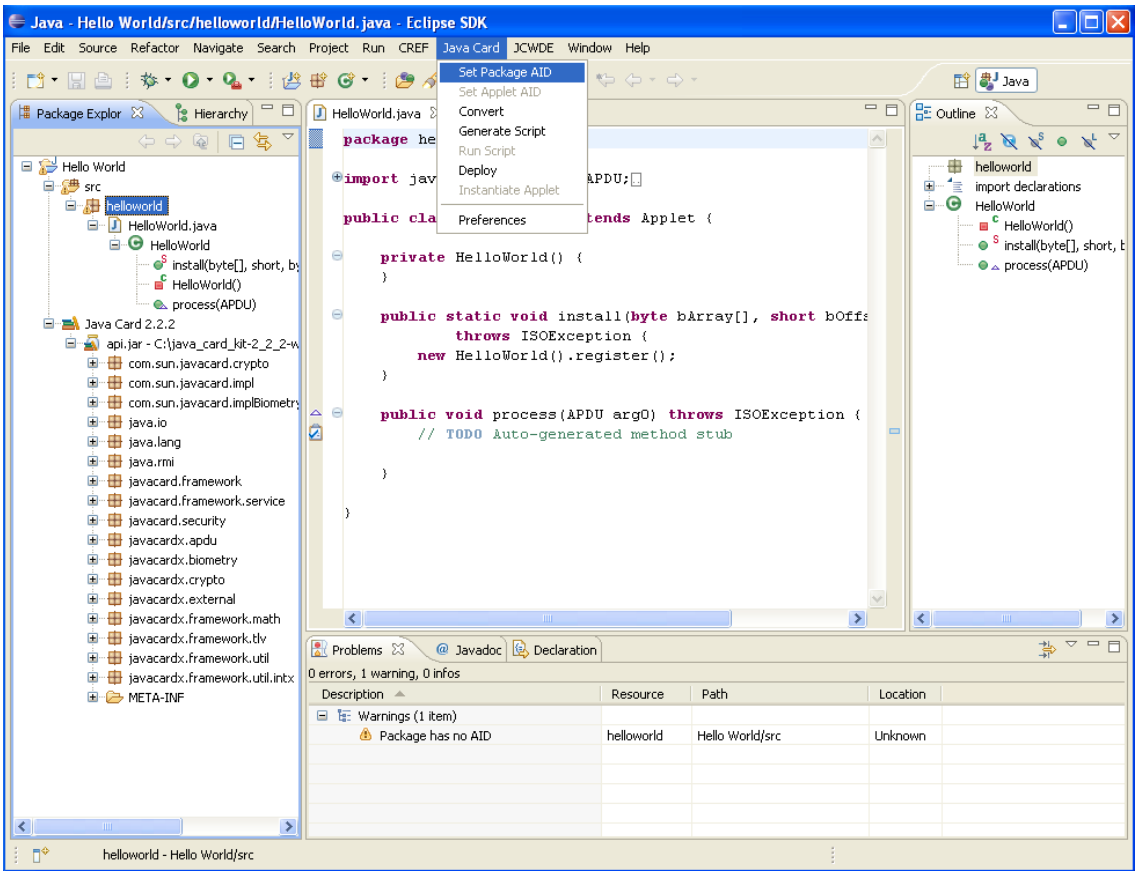


FIGURA 4.25 – DEFININDO O AID DO PACOTE

Uma vez que o *AID* do pacote esteja definido, o *applet* pode ser desenvolvido. O código do *Hello World* para *Java Card* é:

```
package helloworld;
```

```
import javacard.framework.*;

public class HelloWorld extends Applet {
// CLA Byte
final static byte HELLO_CLA = (byte) 0xB0;
// Verify PIN
final static byte INS_HELLO = (byte) 0x20;
        public static void install(byte[] bArray, short bOffset,
byte bLength) {

                new HelloWorld().register();

        }

        // processa o comando APDU
        public void process(APDU APDU) {

                byte[] buffer = APDU.getBuffer();
                if ((buffer[ISO7816.OFFSET_CLA] == 0) &&
                        (buffer[ISO7816.OFFSET_INS] == (byte) (0xA4)))
                        return;
                // Validate the CLA byte
                if (buffer[ISO7816.OFFSET_CLA] != HELLO_CLA)
                 ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);

                // Select the apropriate instruction (Byte INS)
                switch (buffer[ISO7816.OFFSET_INS]) {
                        case INS_HELLO :
                                getHello(APDU);
                                return;

                default:
ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
                }
```

```
        }

        private void getHello(APDU APDU) {
        // cadeia de bytes com a mensagem: "hello world Java Card"
                byte[] hello = {'h', 'e', 'l', 'l', 'o', ' ', 'w',      'o' ,'r' , 'l', 'd', ' ', 'J',
'a', 'v', 'a', ' ', 'C', 'a',   'r', 'd'};
        // informa ao JCRE que será enviado uma resposta
                short le = APDU.setOutgoing();
                short totalBytes = (short) hello.length;

        // informa ao JCRE o tamanho da mensagem em bytes
                APDU.setOutgoingLength(totalBytes);
        // envia a mensgem para o host
                APDU.sendBytesLong(hello, (short) 0,      (short)hello.length);
        }
}
```

Uma vez que o código tenha sido desenvolvido é necessário compilar e converter o código. Para que essas etapas sejam realizadas com sucesso o nível do compilador deve ser alterado.

Para alterar o nível do compilador deve-se ir em "*Window*" e posteriormente em "*Preferences*". Uma nova janela será exibida clicar em "Java" e posteriormente em "*Compiler*", alterar o "*Compiler compilance level*" para 1.3.

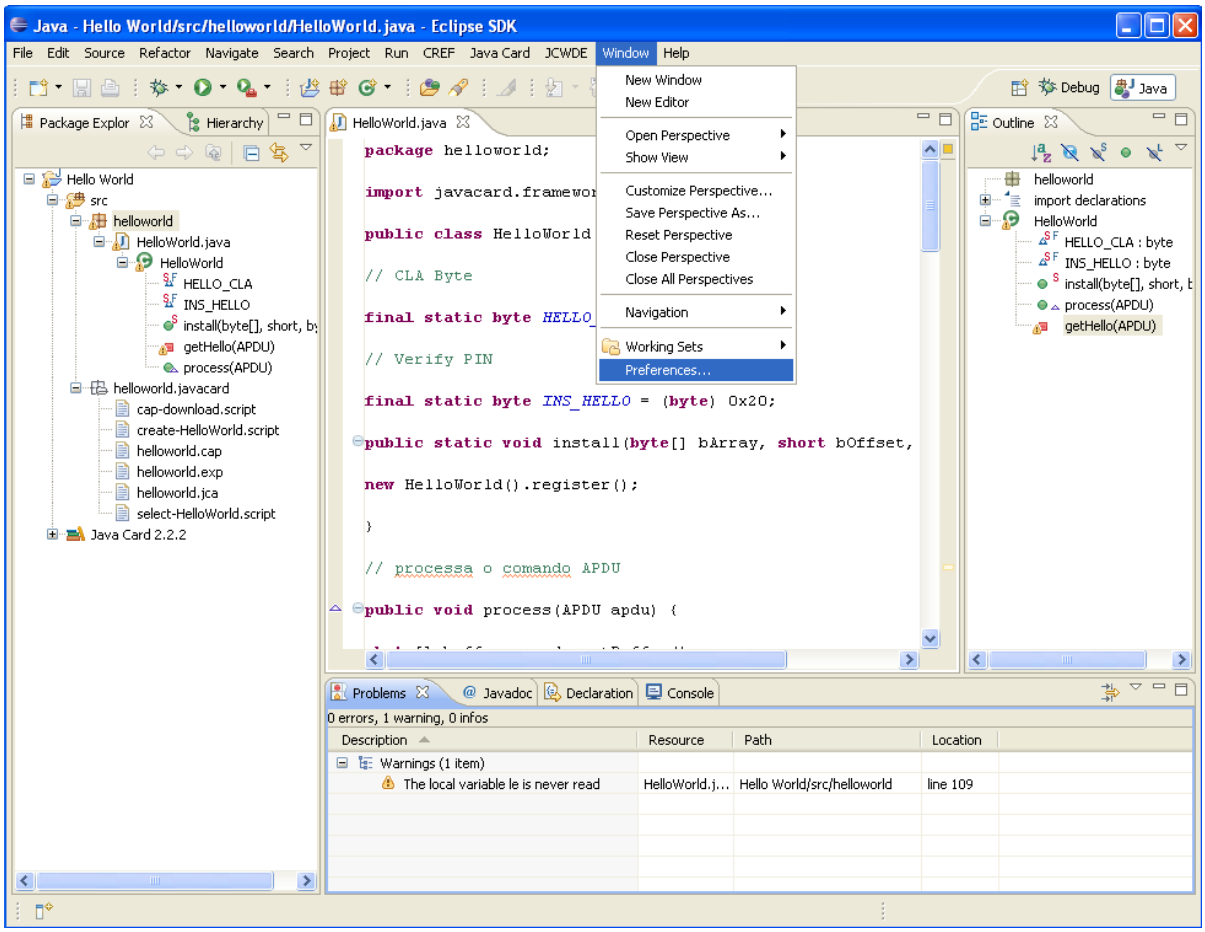


FIGURA 4.26 – ALTERANDO O NÍVEL DO COMPILADOR

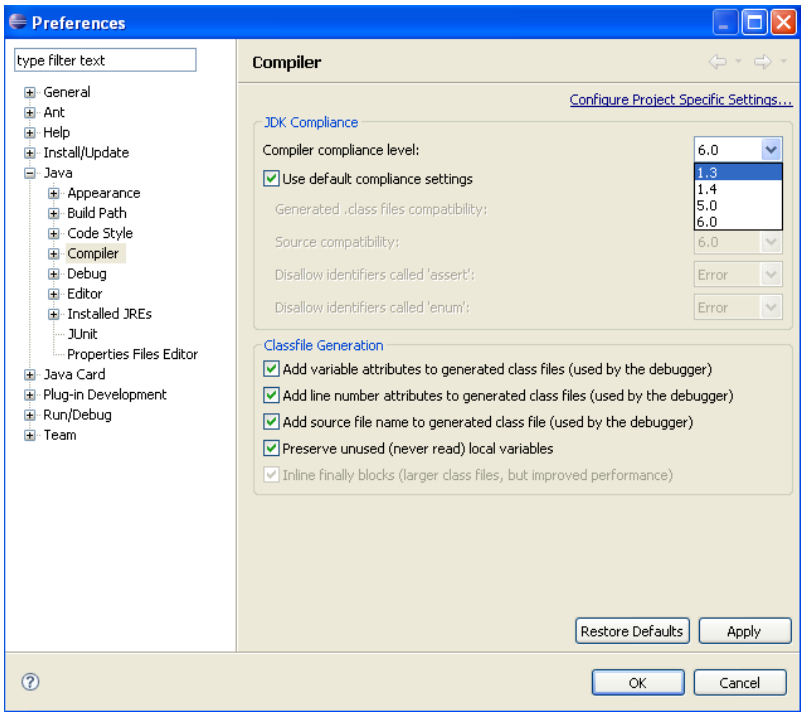


FIGURA 4.27 – COLOCANDO O NÍVEL DO COMPILADOR PARA 1.3

Com o nível do compilador alterado pode-se proceder com a compilação e a conversão em apenas um passo. Selecionar o pacote e ir em "*Java Card*" e posteriormente em "*Convert*".

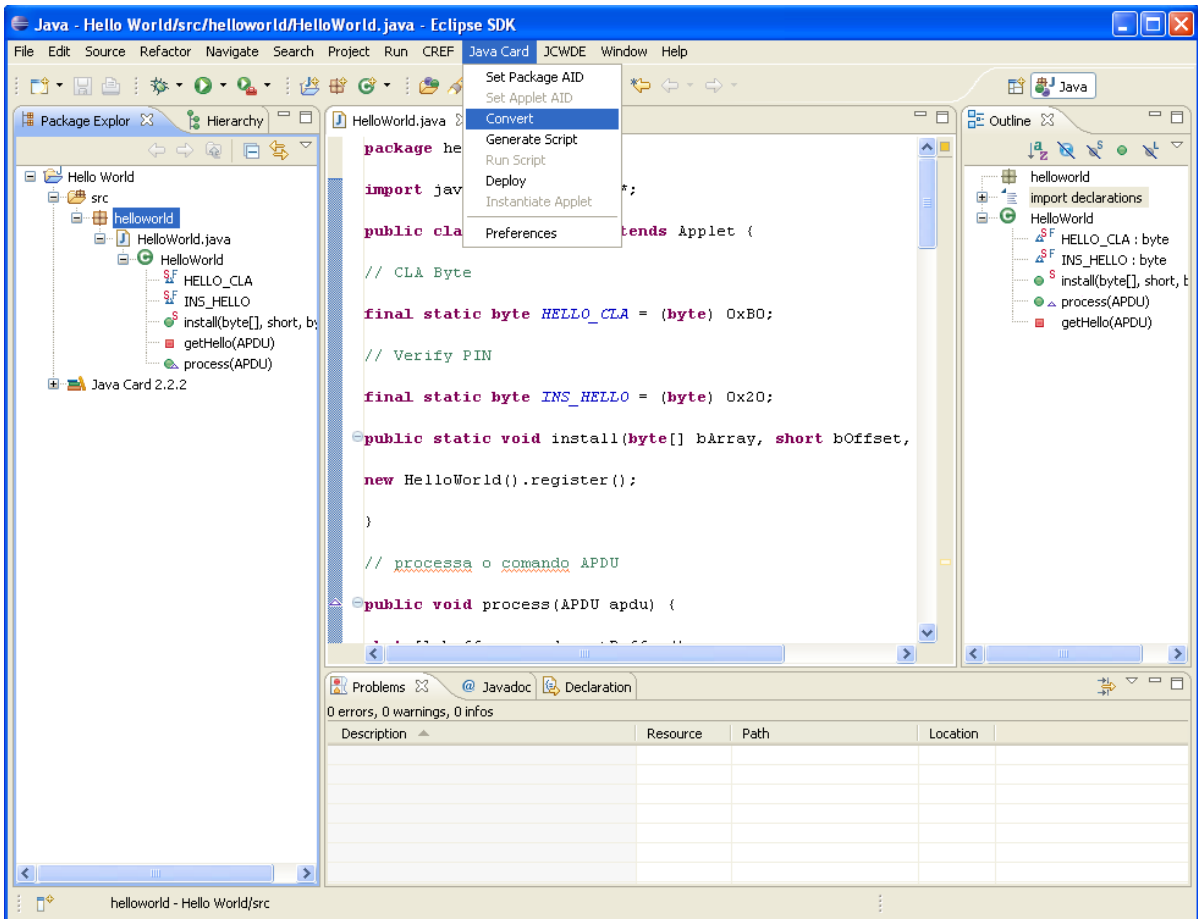


FIGURA 4.28 – COMPILANDO E CONVERTENDO A APLICAÇÃO

Nota-se o aparecimento de três arquivos no ambiente de desenvolvimento após a conversão/compilação o .cap, .exp e .jca.

Não é necessário gravar o arquivo gerado em um cartão para testá-lo, pode-se usar o próprio Eclipse para realizar uma simulação e visualizar o resultado.

Para que ocorra a simulação devem ser criados scripts que simulam o "*host*", que é o programa que envia os comandos para o cartão e para o qual o cartão responde conforme a instrução dada. Essa comunicação é realizada através de *APDU*s. No *applet* de exemplo nesta documentação a instrução para o "*Hello World*" é: B0 20 00 00.

Para gerar scripts de simulação deve-se selecionar o pacote e ir em "*Java Card*" e posteriormente em "*Generate Script*".

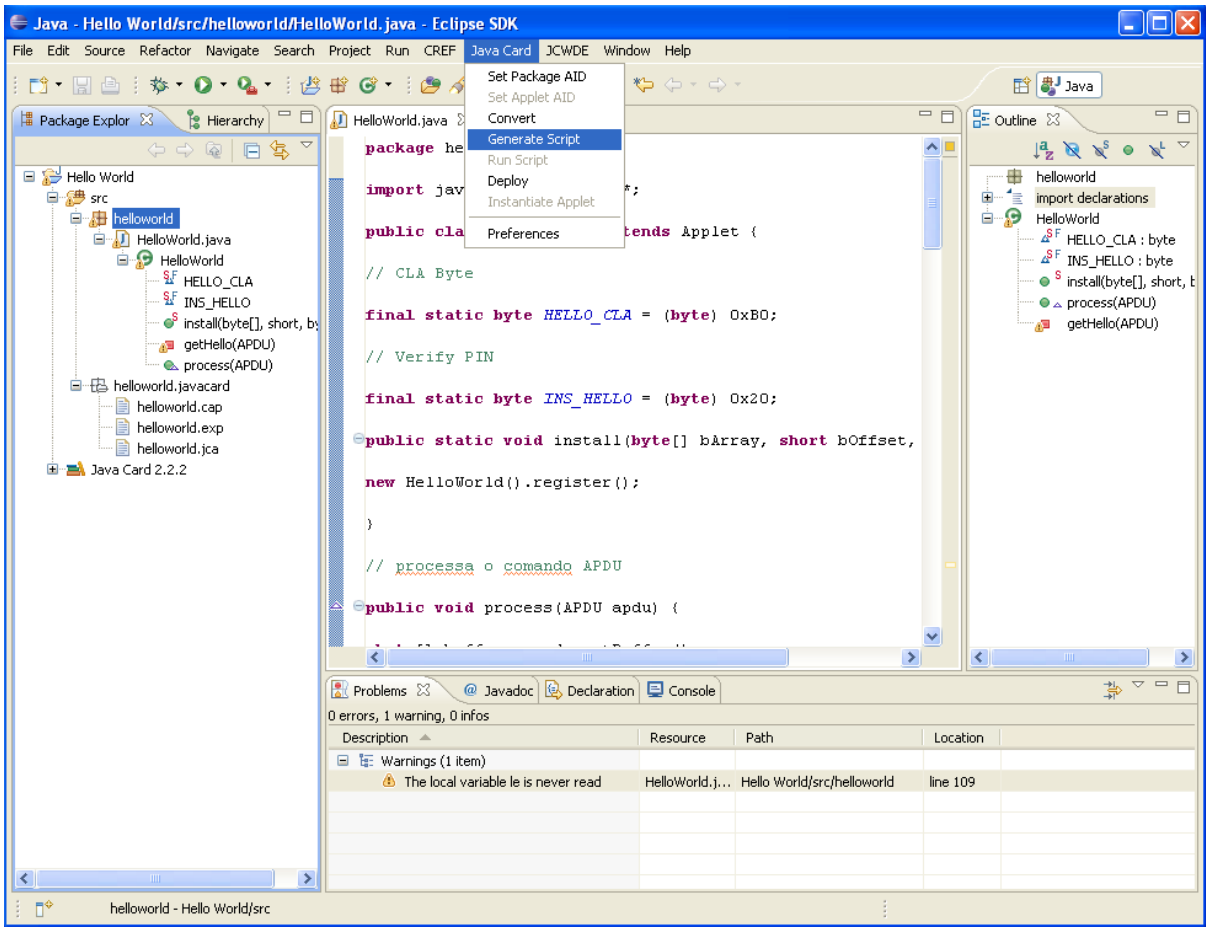


FIGURA 4.29 – CRIANDO SCRIPTS PARA TESTE

Três *scripts* serão gerados e para efeitos de teste dois deles devem ser unidos e salvos na mesma pasta que os demais.

Os scripts "*create-xxx.script*" e "*select-xxx.script*" devem ser unidos e no final deve ser adicionada a *APDU* correspondente ao comando escolhido para o "*Hello World*". O *script* de teste resultante para o exemplo deste documento é:

powerup;

// Select the installer applet

0x00 0xA4 0x04 0x00 0x09 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x08 0x01 0x7F;

// create HelloWorld applet

0x80 0xB8 0x00 0x00 0xd 0xb 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x00 0x00 0x00 0x7F;

// select HelloWorld applet

0x00 0xA4 0x04 0x00 0xb 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x00 0x00 0x7F;

// hello APDU

0xB0 0x20 0x00 0x00 0x00 0x7F;

powerdown;

Para facilitar a simulação mudar a visualização do ambiente de desenvolvimento para "*Debug*". Isso é feito clicando em "*Open Perspective*" no canto superior direito do Eclipse e escolhendo "*Debug*".

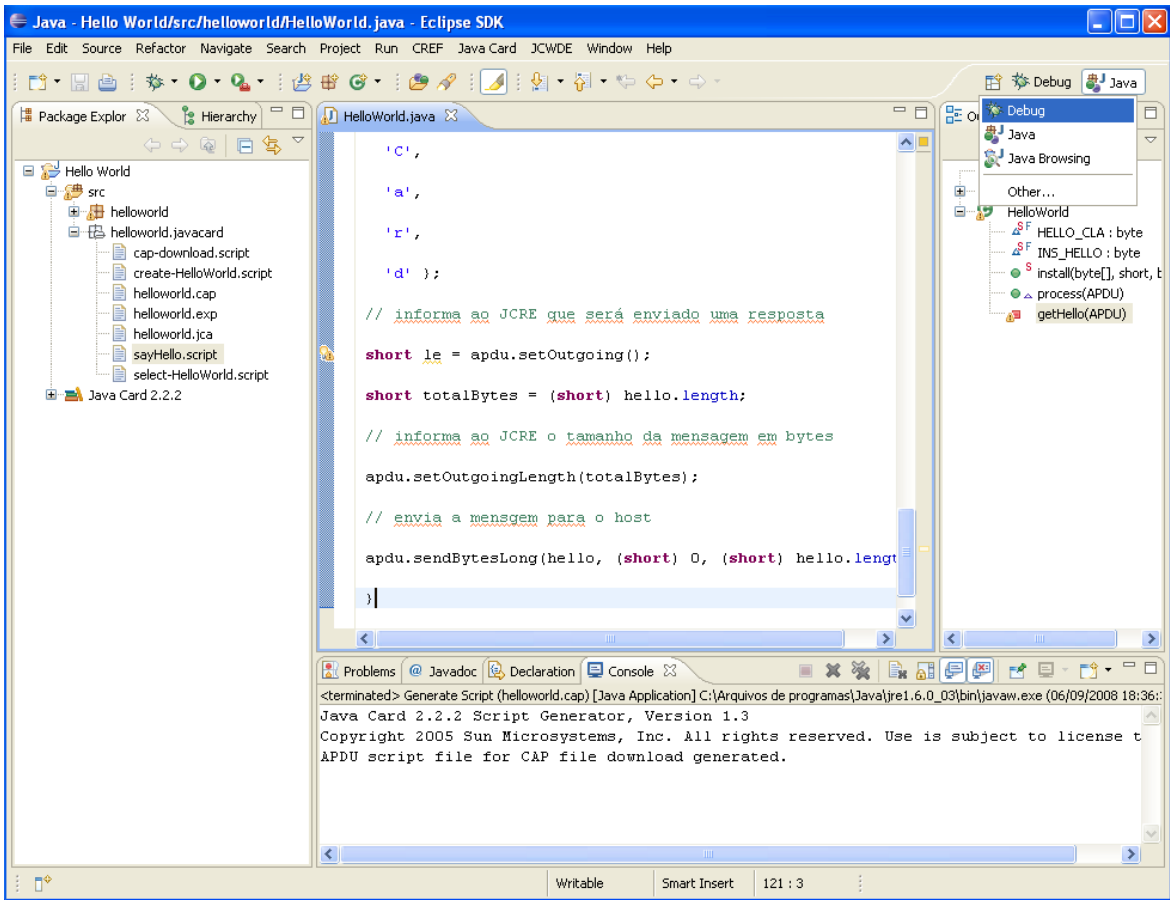


FIGURA 4.30 – COLOCANDO EM MODO DEBUG

Uma vez em modo "*debug*" para visualizar os scripts e classes deve-se abrir o "*Packager Explorer*" para isso ir em "*Window > Show View > Other*". Na janela exibida, selecionar "Java" e posteriormente "*Package Explorer*".

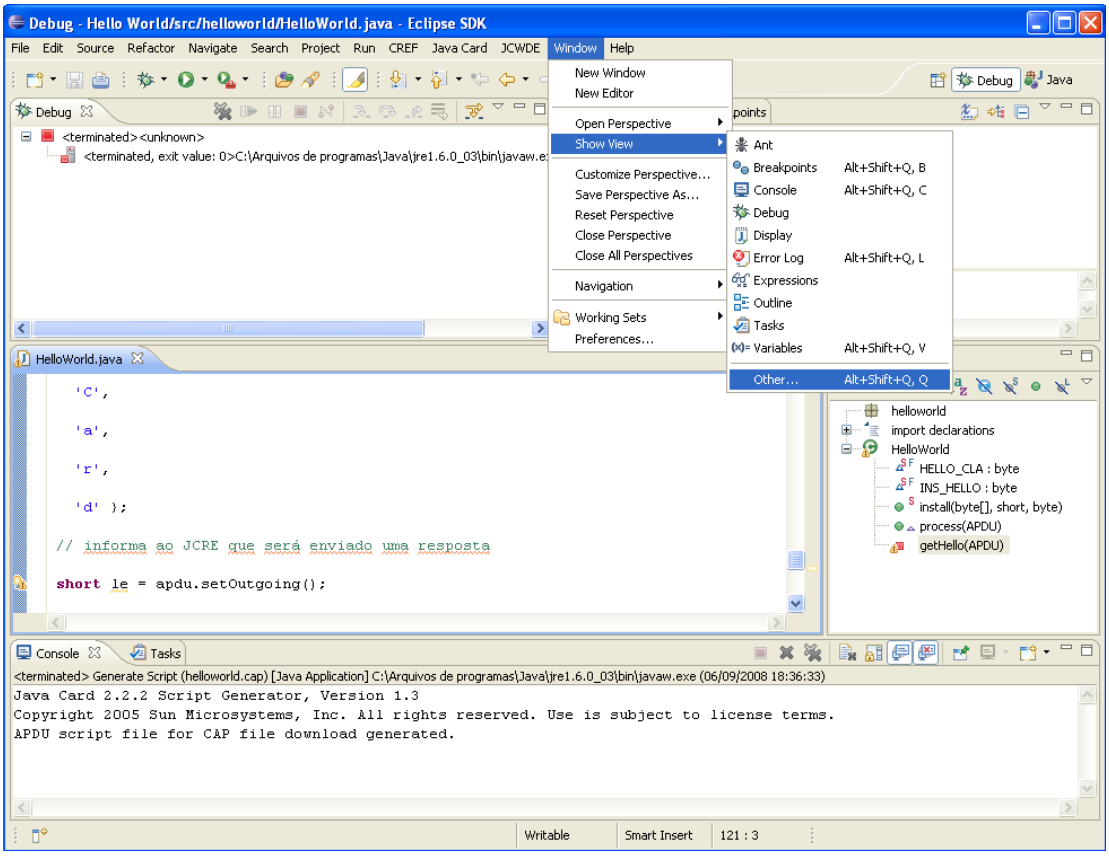


FIGURA 4.31 – MOSTRANDO UMA NOVA VIEW

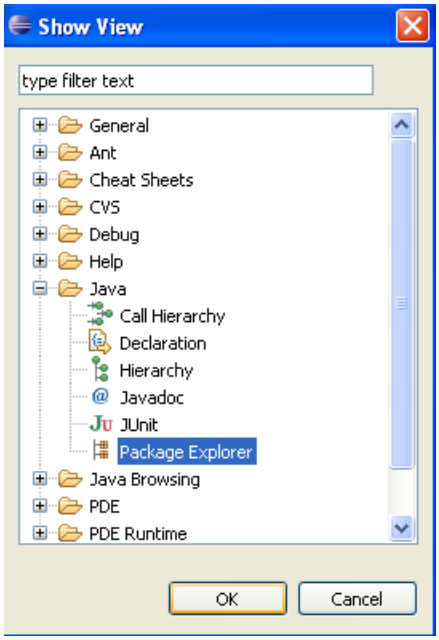


FIGURA 4.32 – SELECIONANDO A VIEW

Uma vez que a nova perspectiva esteja aberta pode-se rodar a simulação. Para isso selecionar "*JCWDE* > *Debug*" e escolher o pacote do *applet*.

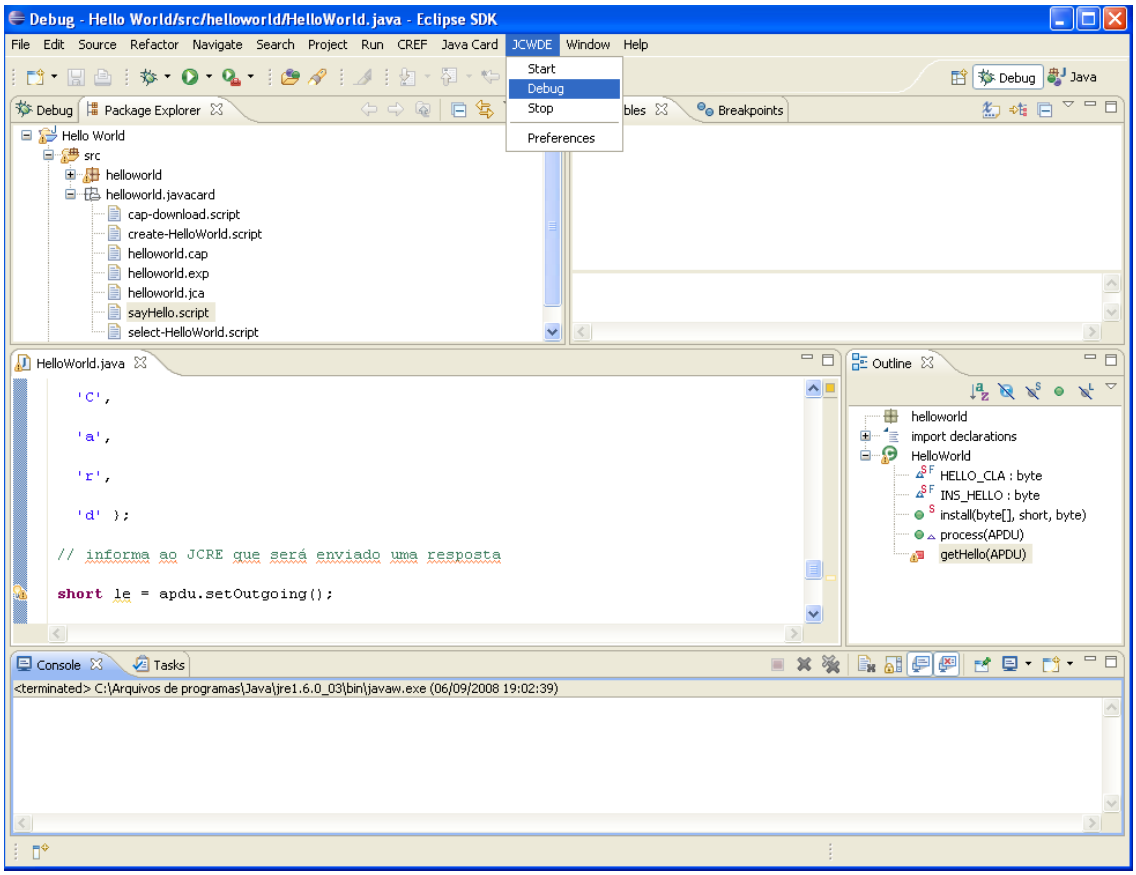


FIGURA 4.33 – EXECUTANDO A SIMULAÇÃO

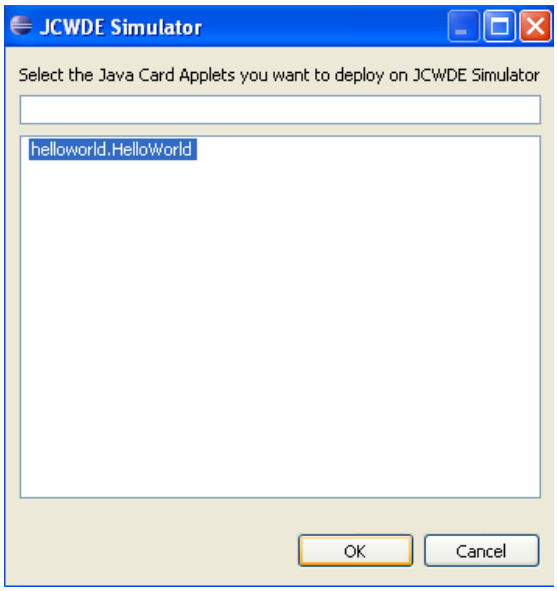


FIGURA 4.34 – SELECIONANDO O *APPLET* PARA SIMULAÇÃO

Quando o *debug* está ativo o Eclipse escuta a porta TCP/IP 9.025 e neste ponto deve-se rodar o *script* contendo as instruções para o *applet*. Para rodar o *script* selecionar o script no "*Package Explorer*" e ir em "*Java Card > Run Script*".

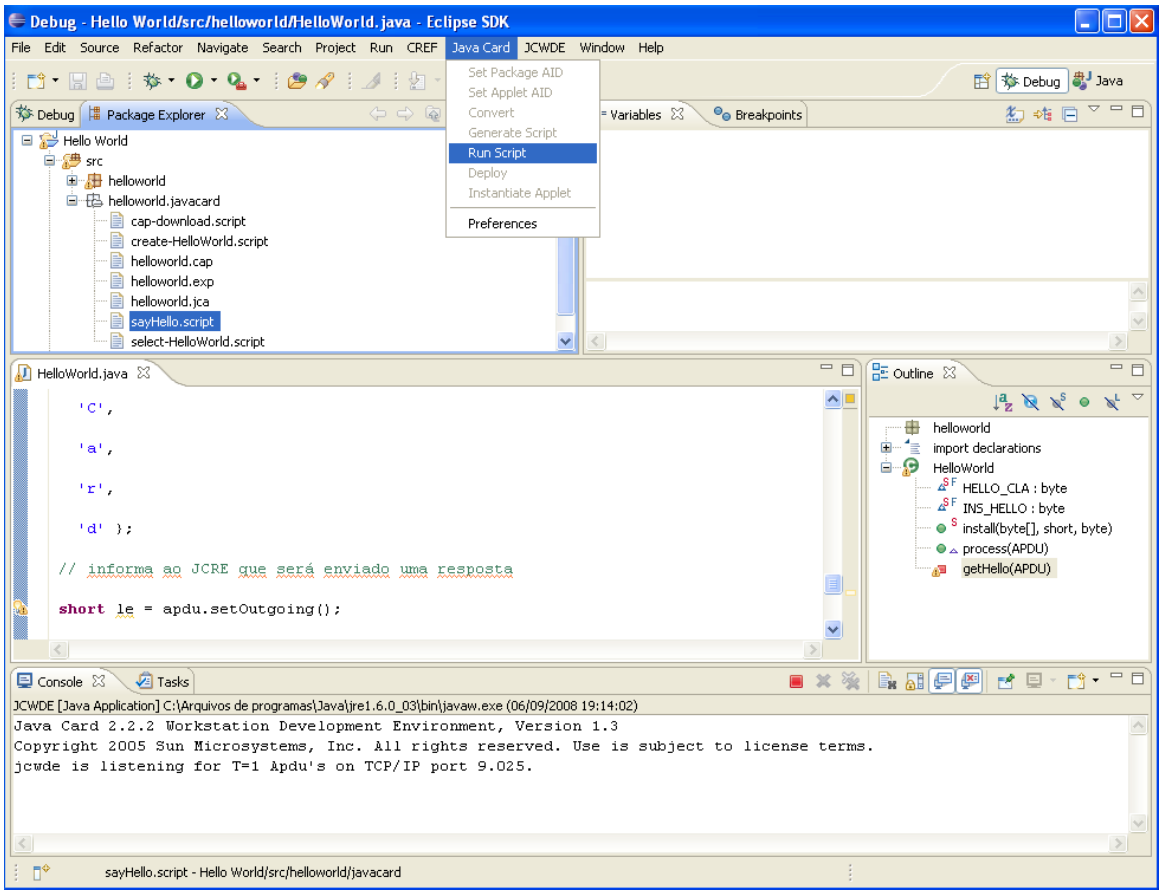


FIGURA 4.35 – EXECUTANDO SCRIPT PARA SIMULAÇÃO

Para visualizar o resultado clicar na aba "*debug*" e clicar na última linha. O resultado será exibido na aba "*Console*" abaixo.

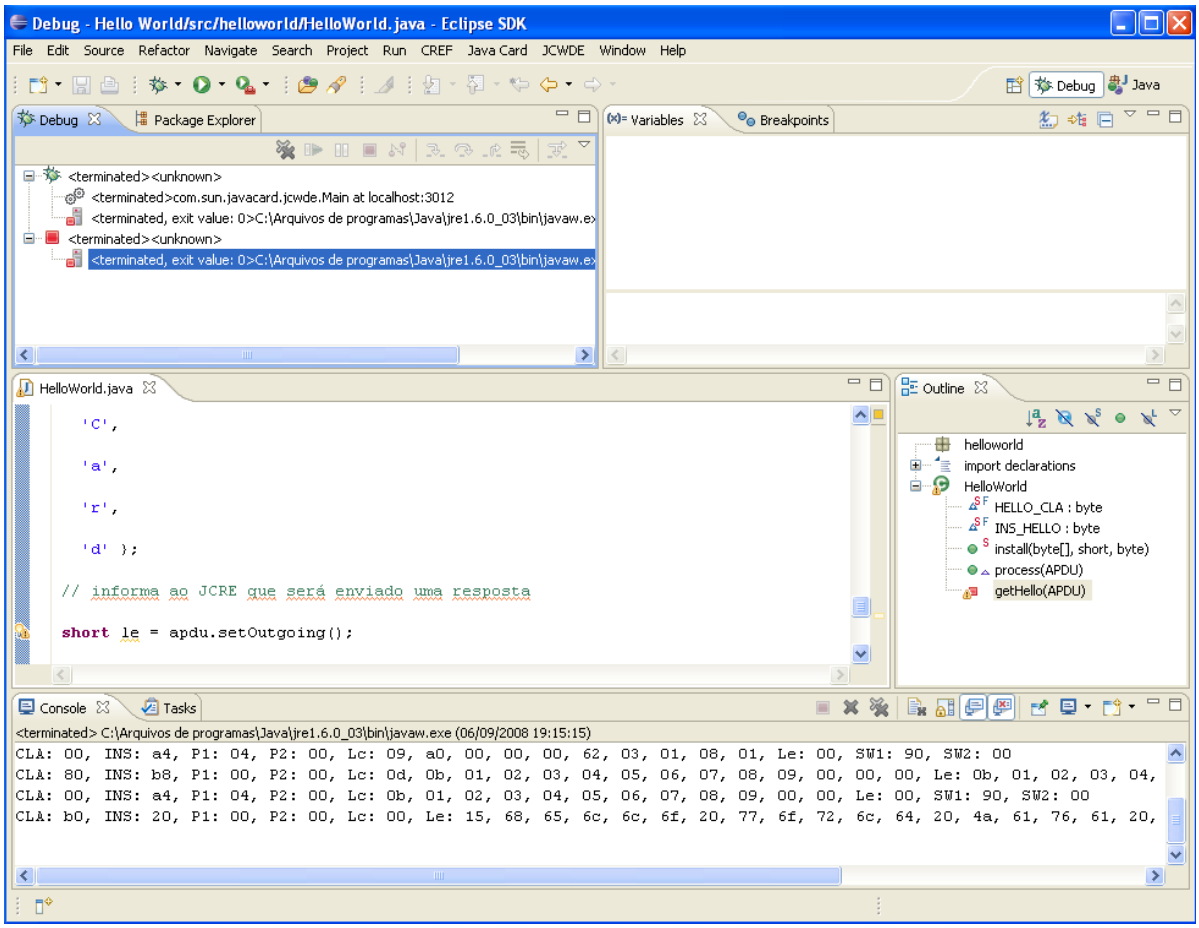


FIGURA 4.36 – VERIFICANDO O RESULTADO

O resultado obtido foi:

CLA: 00, INS: a4, P1: 04, P2: 00, Lc: 09, a0, 00, 00, 00, 62, 03, 01, 08, 01, Le: 00, SW1: 90, SW2: 00

CLA: 80, INS: b8, P1: 00, P2: 00, Lc: 0d, 0b, 01, 02, 03, 04,05, 06, 07, 08, 09, 00, 00, 00, Le: 0b, 01, 02, 03, 04, 05, 06, 07, 08, 09, 00, 00, SW1: 90, SW2: 00

CLA: 00, INS: a4, P1: 04, P2: 00, Lc: 0b, 01, 02, 03, 04, 05, 06, 07, 08, 09, 00, 00, Le: 00, SW1: 90, SW2: 00

CLA: b0, INS: 20, P1: 00, P2: 00, Lc: 00, Le: 15, 68, 65, 6c, 6c, 6f, 20, 77, 6f, 72, 6c, 64, 20, 4a, 61, 76, 61, 20, 43, 61, 72, 64, SW1: 90, SW2: 00

Para cada comando do script temos uma resposta por parte do "cartão". Para o último comando que é a *APDU* para "*Hello World*" a resposta foi: 68, 65, 6c, 6c, 6f, 20, 77, 6f, 72, 6c, 64, 20, 4a, 61, 76, 61, 20, 43, 61, 72, 64.

Com o auxílio de uma tabela *ASCII*, nota-se que a resposta corresponde a "*hello world Java Card*" em hexadecimal, que é a reposta esperada.

Capítulo 5

Programa host escritor

Neste capítulo será apresentado um programa *host* que é responsável por escrever os dados necessários no *applet*, que reside no *smart card*, para autenticação de usuários nos laboratórios do DEL.

5.1 – Contextualização

Para demonstração da linguagem *Java Card* e as vantagens que a segurança que a uma solução envolvendo *smart card* proporciona foi desenvolvido um sistema de identificação e autenticação de alunos em laboratórios do DEL.

Como mencionado no capítulo 3 para o funcionamento de uma solução em *smart card* é necessário uma aplicação residente no cartão – o *applet* – e uma aplicação no *PC* – aplicação *host* – que será a responsável por enviar comandos ao *applet* através do *CAD*.

A aplicação residente no cartão será responsável por armazenar as informações referentes a cada aluno como DRE e as chaves de cada laboratório no qual o aluno tem acesso (utilizadas no processo de autenticação) além dos comandos *APDU* responsáveis pelas troca de informações com o *host* leitor e o *host* escritor e atualização de dados.

A aplicação *host* escritor será responsável pela comunicação entre o usuário, o *applet Java Card* e o aplicativo *back-end* e visa oferecer um mecanismo simples e ágil para a realização de atividades de manipulação de um cartão, no caso em âmbito de administrador, podendo realizar operações como:

(1) Atualização de DRE

(2) Inserção de chave única, para um laboratório

(3) Inserção de chaves múltiplas, para vários laboratórios

(4) Atualização de PIN

5.2 – O *Applet* SmartDEL

A aplicação residente no cartão é a responsável por armazenar as informações necessárias para identificação, atualização e autenticação do aluno nos laboratórios do DEL, assim como realizar os cálculos necessários em todo o processo de autenticação

utilizando métodos de criptografia de dados. Na tabela 5.1 são mostrados os métodos implementados pela aplicação e os respectivos comandos *APDU* que devem ser enviados pelo *host*.

Método	CLA	INS	P1	P2	Lc/Le (Hex)	Dado
Verify	B0	20	00	00	08	PIN
sendID	B0	30	00	00	09	
sendResponse	B0	30	01	LabID	10	RAND
updatePIN	B0	40	00	00	08	PIN
keysInitialization	B0	50	00	00	A0	KEYS
updateSpecificKey	B0	50	01	LabID	10	KEY
writeID	B0	50	02	00	09	DRE

TABELA 5.1 – MÉTODOS EXECUTADOS PELO *APPLET*

O método "*verify*" é utilizado para verificar o PIN, que é uma chave de autenticação com o *smart card*, armazenado na aplicação. Assim que a aplicação *host* for iniciada o usuário deverá inserir um código de *PIN* válido durante a autenticação, se a informação inserida pelo usuário for correta ele estará autenticado com o *smart card* provando que é o legítimo possuidor do cartão. Atualmente existem outras formas de autenticar o usuário com o cartão como a biometria, amplamente utilizado em soluções de passaporte digital.

Ao receber o comando *sendID* o cartão deverá retornar o DRE armazenado para a aplicação *host*, e uma vez na posse dessa informação a aplicação *host* realizará uma busca interna em seu arquivo de dados de modo a identificar o usuário que está tentando acessar o laboratório. Este procedimento é um passo importante no processo de autenticação uma vez que cada usuário possui uma chave única por laboratório.

O método *sendResponse* é o responsável pela autenticação do usuário com o laboratório, que libera ou não o acesso de acordo com o resultado. Neste comando o laboratório (aplicação *host*) envia seu identificador através do "P2" e um número randômico criado para cada processo de autenticação, a tabela 5.2 mostra a lista de laboratórios possíveis e seu respectivo identificador. O processo de autenticação completo é exemplificado no apêndice A.

Identificador	Laboratório
00	LIG
01	LIF

02	PADS
03	LPS
04	GTA
05	Reservado para uso futuro
06	Reservado para uso futuro
07	Reservado para uso futuro
08	Reservado para uso futuro
09	Reservado para uso futuro

TABELA 5.2 – LISTA DOS LABORATÓRIOS E SEU IDENTIFICADOR

Uma vez que o cartão recebe os dados enviados é calculada a resposta "res" encriptando o dado randômico com a chave do laboratório armazenada no *smart card* utilizando o algoritmo *triple DES*. Os 8 primeiros *bytes* da resposta "res" são enviadas à aplicação *host* que realiza o mesmo procedimento de codificação considerando a chave do usuário em questão e o número randômico previamente enviado. O comando *sendResponse* somente pode ser enviado pelo laboratório (aplicação *host*) quando o usuário estiver autenticado com o *smart card* e identificado pelo laboratório utilizando os métodos "*verify*" e "*sendID*" respectivamente.

O método *updatePIN* atualiza o *PIN* atual por um novo código escolhido pelo usuário. Para utilizar este método deve-se realizar a verificação do *PIN* previamente através do comando "*verify*".

A gravação de chaves para os laboratórios e DREs no cartão também é realizada pelo *applet* através dos comandos de escrita. O comando "*keysInitialization*", que somente pode ser executado uma vez por cartão é responsável por armazenar em um array de *bytes* as chaves de todos os laboratórios, onde cada grupo de 16 *bytes* corresponde à chave de um laboratório para o portador do cartão. As chaves são ordenadas no *array* de acordo com o índice do laboratório, como ilustrado na figura 5.1.

Lab00	Lab01	Lab02	Lab03	Lab04	Lab05	Lab06	Lab07	Lab08	Lab09

FIGURA 5.1 – ARRAY DE CHAVES

O método "*updateSpecificKey*" atualiza uma chave de laboratório específica, que é indicada no parâmetro "P2" da *APDU*, e pode ser executado quantas vezes forem necessárias.

Já o método "*writeID*" é responsável por escrever o DRE em um arquivo do *smart card* que contem o DRE do aluno portador do *smart card*.

5.3 – Aplicação *Host* Escritora

A aplicação *host* escritora é responsável por gerar e armazenar os dados necessários para cada usuário no cartão e armazená-los em um banco de dados seguro para que possam ser consultados posteriormente pela aplicação *host* 'leitora', que será responsável por realizar todo o processo de identificação e autenticação do usuário no laboratório.

Para o desenvolvimento da aplicação *host* foi utilizada a ferramenta *Visual Studio 2005*, registrado em nome de Luís Renato Azevedo de Araujo Silva (usuário cadastrado no programa *Academic Alliance*), sendo a linguagem de programação vb.net. Para a conexão com o *CAD* foram utilizados métodos da biblioteca *Winscard.dll*, biblioteca nativa do Windows. Mais detalhes da biblioteca, seus métodos e utilização podem ser encontrados no anexo A.

Na figura 5.2 pode ser vista a interface gráfica da aplicação *host* escritora. No apêndice A é ilustrado e exemplificado o processo completo de autenticação com o laboratório.

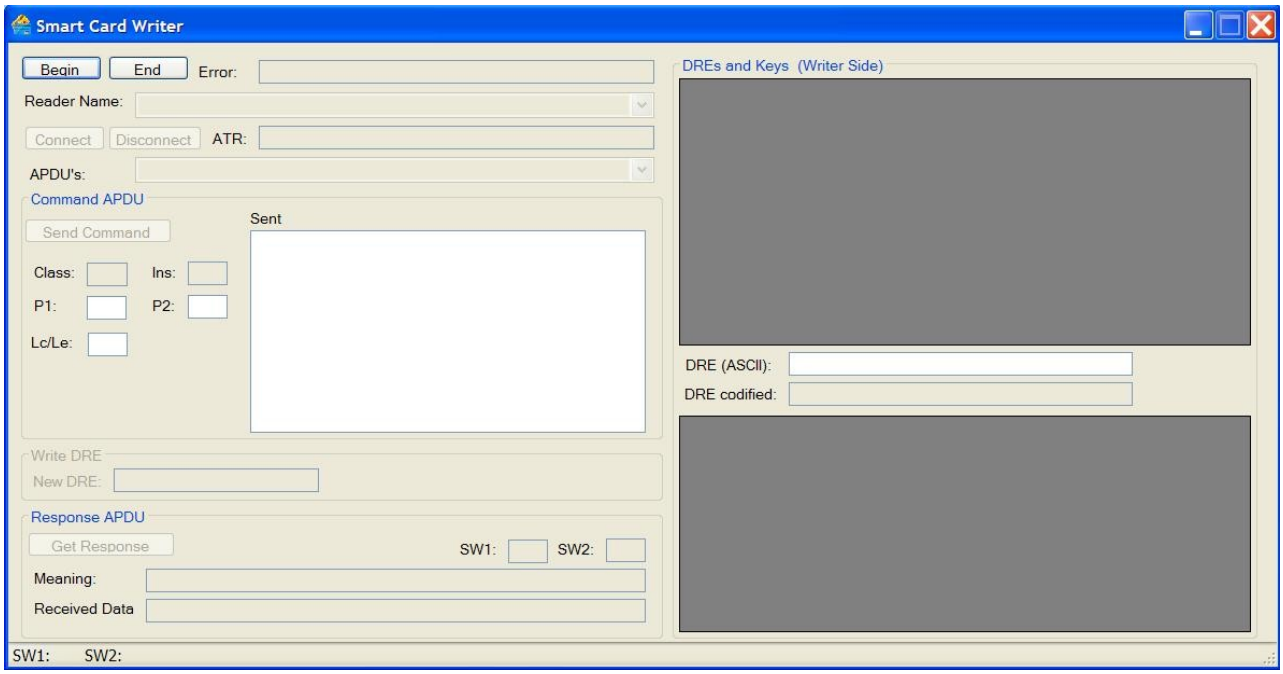


FIGURA 5.2 – INTERFACE GRÁFICA DO PROGRAMA *HOST*

A interface gráfica, figura 5.2, ilustra todo o processo de comunicação entre a aplicação *host* e o *smart card*: as *APDU*s que são enviadas, as chaves, os laboratórios, a geração das chaves e as repostas das *APDU*s enviadas com seus respectivos

significados. Para popular os campos "*APDU*s", "DREs and *Keys*" e armazenar os dados referentes a cada aluno foram utilizados arquivos especiais com a estrutura mostrada no apêndice B. A lista de todos os arquivos e a informação armazenada está descrita na tabela 5.3.

Arquivo	Conteúdo	Aplicativo Utilizador
SmartCardWriter.config	Nome da *APDU* e Código da *APDU*	*Host* Escritor
MasterKeyFile.txt	Código do Laboratório, Nome do Laboratório e as Chaves Mãe.	*Host* Escritor
DREContentDataLabXX.txt	DRE dos alunos permitidos no laboratório xx e suas respectivas chaves filhas.	*Host* Leitor e *Host* Escritor

TABELA 5.3 – LISTA DE ARQUIVOS

O arquivo "*SmartCardWriter.config*" relaciona o nome do comando *APDU* ao seu respectivo código binário, com base neste arquivo a programa *host* preenche todos os comandos *APDU*s enviados ao cartão.

No arquivo "*MasterKeyFile.txt*" estão armazenadas as informações relacionando cada laboratório à sua "Chave Mãe", uma cadeia de 16 *bytes* gerada randomicamente. Esta chave combinada com o DRE para geração da "Chave Filha", igualmente de 16 *bytes*, será armazenada tanto no *smart card* como no banco de dados do laboratório e será utilizada no processo de autenticação do aluno no laboratório. O processo de geração da "Chave Filha" a partir da "Chave Mãe" é exemplificado no apêndice B. Na interface gráfica, figura 5.2, da aplicação *host* de escrita são mostradas, para fins didáticos, todas as chaves e todos os dados gerados, porém em um ambiente de produção as chaves devem ser geradas automaticamente e armazenadas em um ambiente seguro, com acesso restrito, a fim de evitar falhas de segurança.

Para que o processo de autenticação seja realizado com sucesso, apêndice A, o *smart card* deve relacionar todas as chaves armazenadas em seu buffer ao laboratório correspondente. O laboratório, por sua vez, deve relacionar o aluno à sua respectiva

chave, relação obtida no arquivo "*DREContentDataLabXX*.txt", onde XX indica o código do laboratório. Deste modo cada laboratório possui seu arquivo onde possui armazenada a relação aluno-chave que será utilizada posteriormente no processo de autenticação. Na figura 5.3 pode ser vista a modelagem dos arquivos.

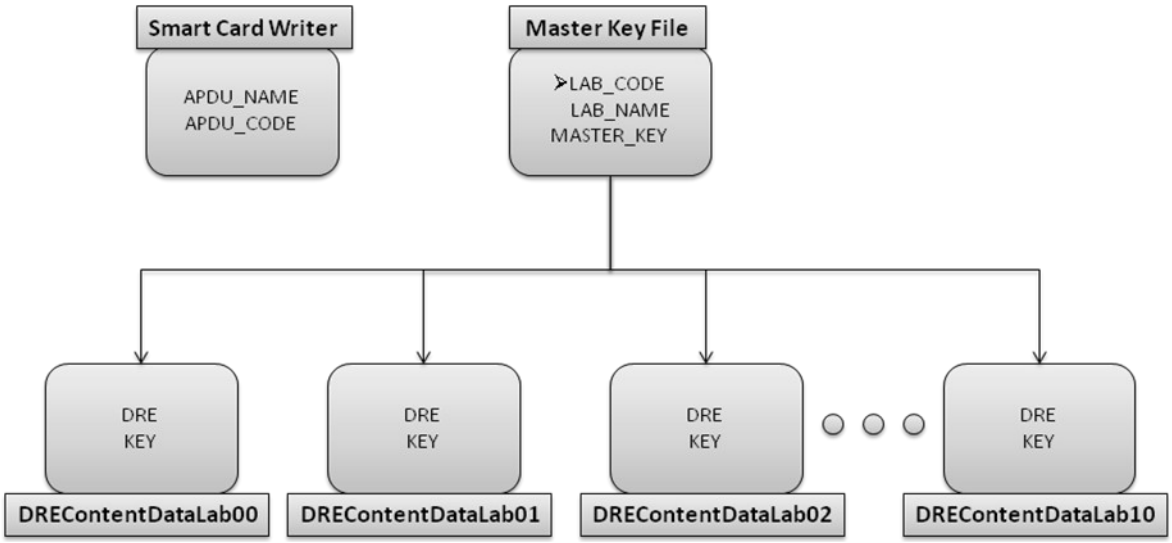


FIGURA 5.3 – MODELAGEM DOS ARQUIVOS.

Para iniciar o processo de comunicação do cartão com a aplicação *host* deve-se inserir um *smart card* a um *CAD* conectado ao *PC* em seguida clicar no botão "*Begin*" e escolher a leitora na qual foi inserida o cartão. Na figura 5.4 é ilustrado o resultado do procedimento no caso em que foi escolhida a leitora "*Gemplus USB SmartCard Reader* 0", gentilmente cedida para fins de pesquisa pela empresa Gemalto Cartões e Terminais Ltda.

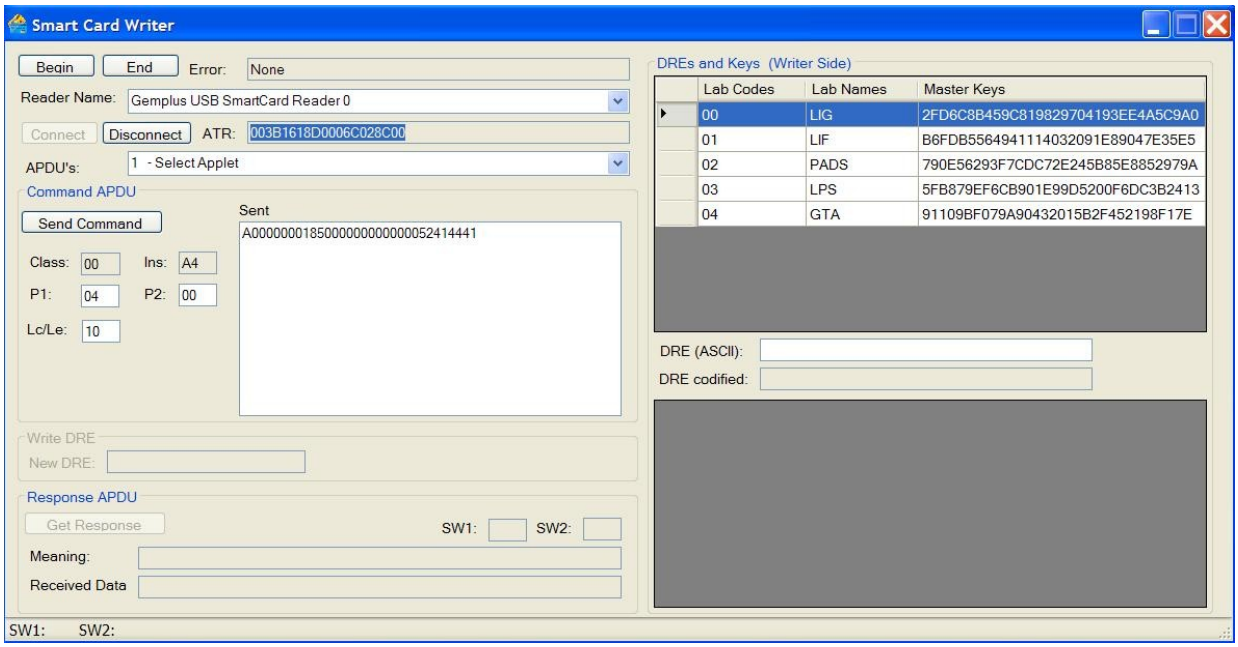


FIGURA 5.4 – ESCOLHA DA LEITORA

Ao ser iniciada, a aplicação realiza a leitura dos arquivos "*SmartCardWriter.config*" e "*MasterKeyFile*.txt" populando os campos "*APDU*s" e a tabela contendo as chaves Mãe relacionadas a cada laboratório, que são geradas randomicamente e ficam armazenadas no arquivo.

Cada item da combo "*APDU*s" representa uma chamada ao método do *Applet* SmartDEL, com exceção do "*Select Applet*", que representa a própria seleção do *Applet* e obrigatoriamente deve ser feita.

A tabela 5.4 a seguir relaciona os nomes na combo "*APDU*s" com os métodos do SmartDEL.

Método	**Nome de *APDU* relacionado**
verify	Verify PIN
sendID	Request DRE
sendResponse	Não utilizado
updatePIN	Update PIN
keysInitialization	Write Default Keys
updateSpecificKey	Update Specific Keys
writeID	Update DRE

TABELA 5.4 – COMANDOS *APDU*

Antes que qualquer outro comando seja enviado ao *applet* deve-se selecioná-lo previamente enviado o comando "*Select Applet*" para que o *JCRE* encaminhe os comandos *APDU*s enviados ao cartão para o *applet* selecionado. Este procedimento é

realizado ao clicar no botão “Send Command” após selecionar o comando desejado. O resultado do comando “Select *Applet*” ao selecionar SmartDEL é ilustrado na figura 5.5.

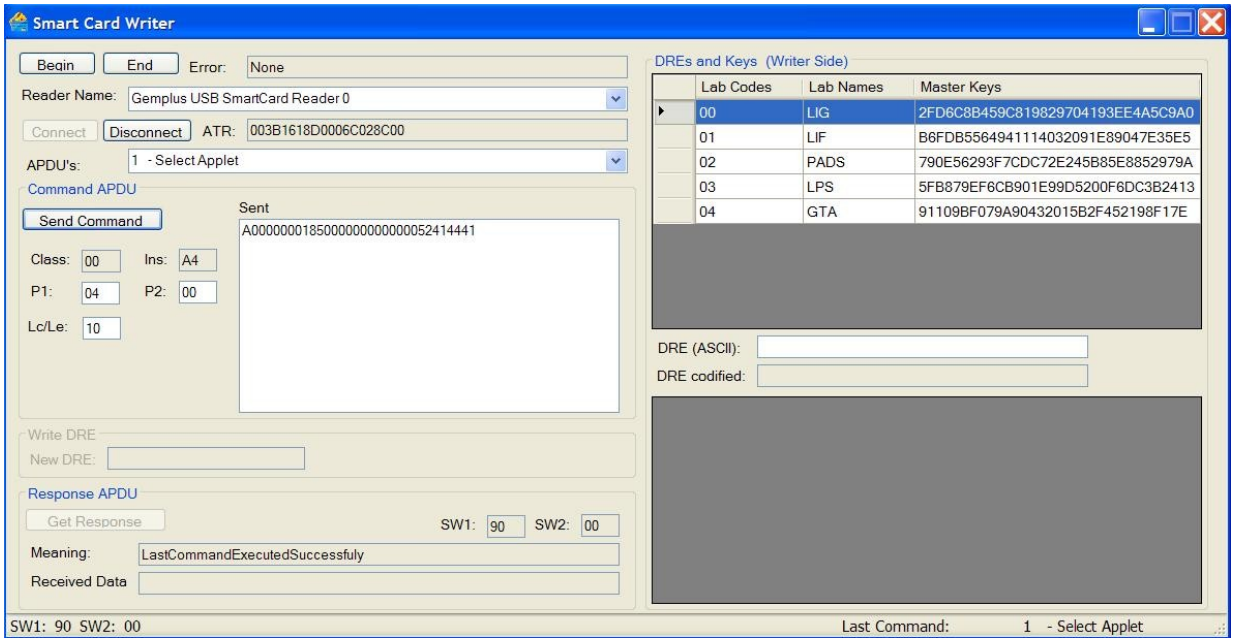


FIGURA 5.5 – SELECT *APPLET*

Ao enviar o comando a resposta pode ser visualizada nos campos da caixa “*Response APDU*”, onde são indicados os *bytes* retornados pelo *smart card* “*SW1*” e “*SW2*” e seu significado representado no campo “*Meaning*”.

Pelo aplicativo *host* de escrita é possível alterar o *PIN* (até 8 *bytes*) do cartão, cujo valor default é “30303030” (0000 em *ASCII*), por questões de segurança é recomendável que o aluno altere o valor por outro ao receber o cartão. Para alterar o valor do *PIN* deve-se primeiramente apresentar o código atual enviando o comando “*Verify PIN*” e posteriormente enviar o comando “*Update PIN*” com o novo valor. Caso haja uma tentativa de alterar o *PIN* antes de ser apresentado o código atual o *applet* retorna um erro. As figuras, 5.6 a 5.10 ilustram o procedimento da troca de *PIN* de “30303030” (0000) para “31313131”. A figura 5.11 mostra um diagrama de sequência para o *Update PIN*.

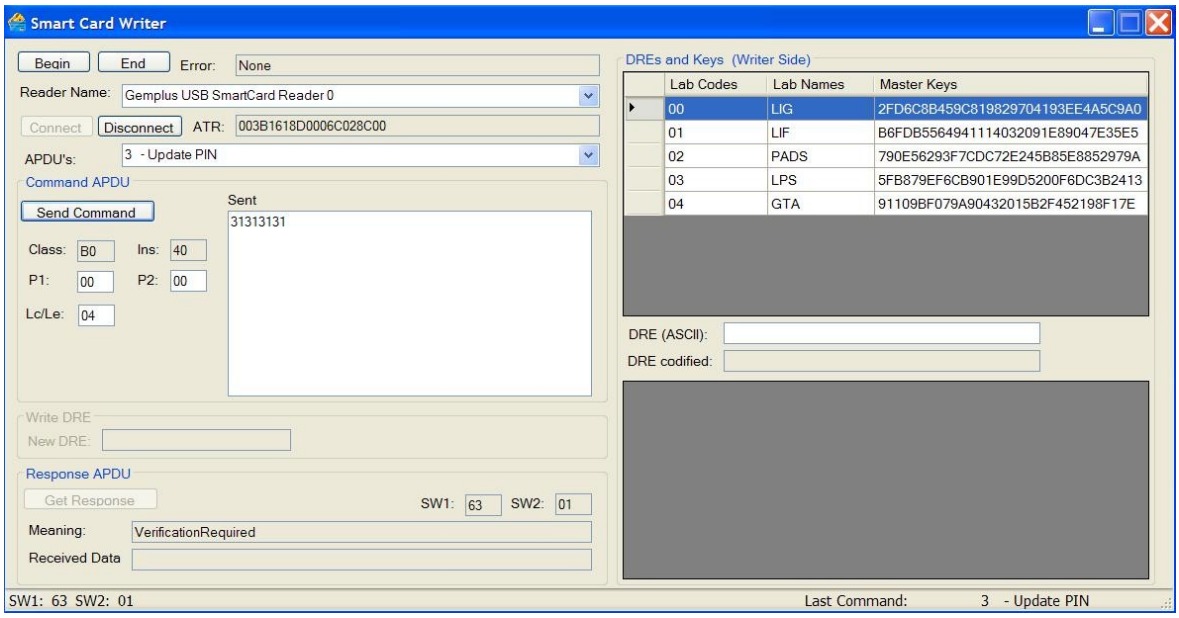


FIGURA 5.6 – COMANDO UPDATE PIN EXECUTADO ANTES DE VERIFY PIN

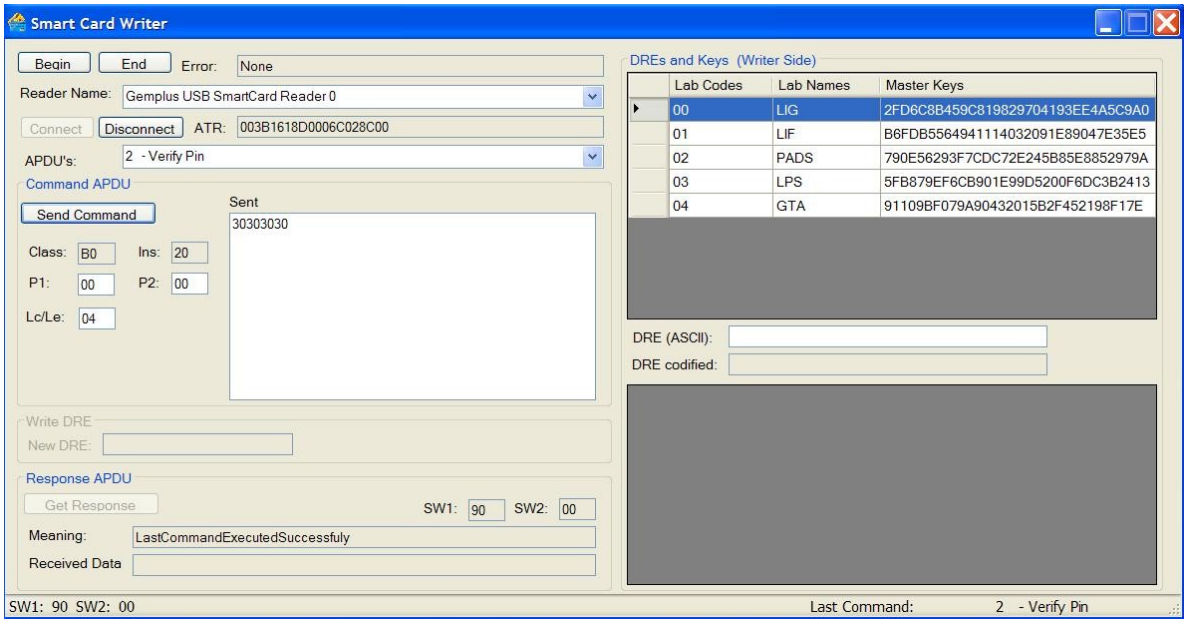


FIGURA 5.7 – COMANDO VERIFY PIN

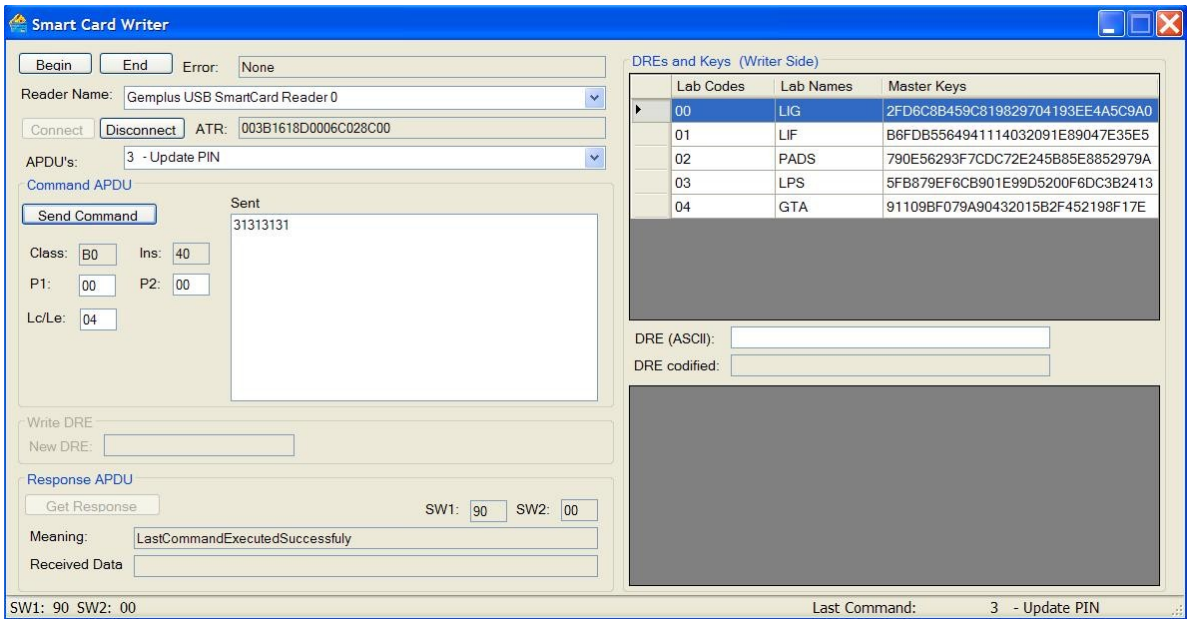


FIGURA 5.8 – COMANDO UPDATE PIN

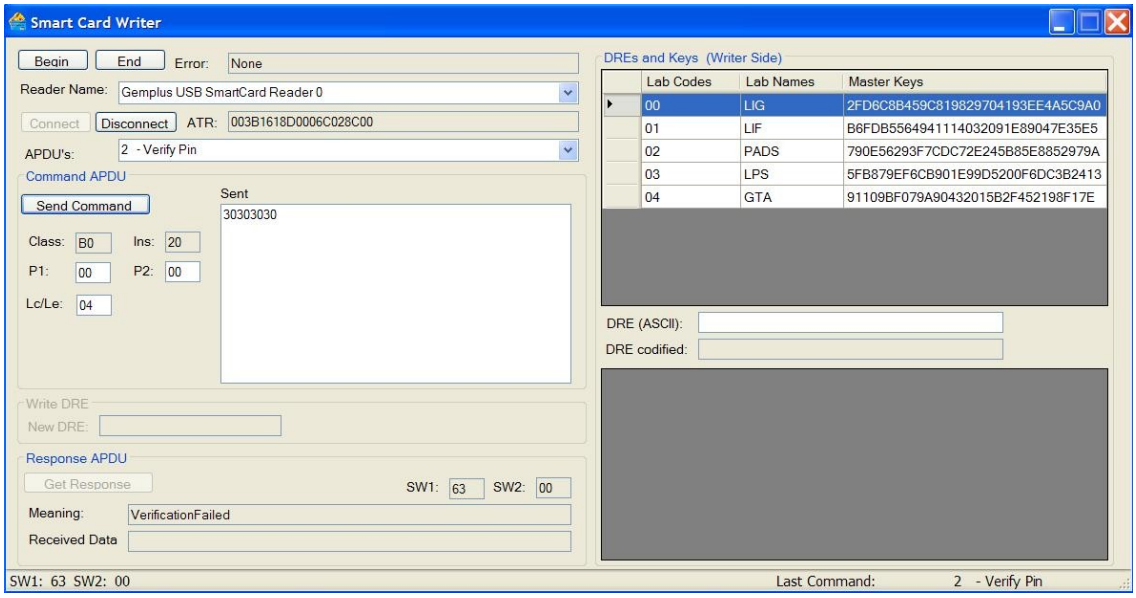


FIGURA 5.9 – COMANDO VERIFY PIN APÓS UPDATE PIN

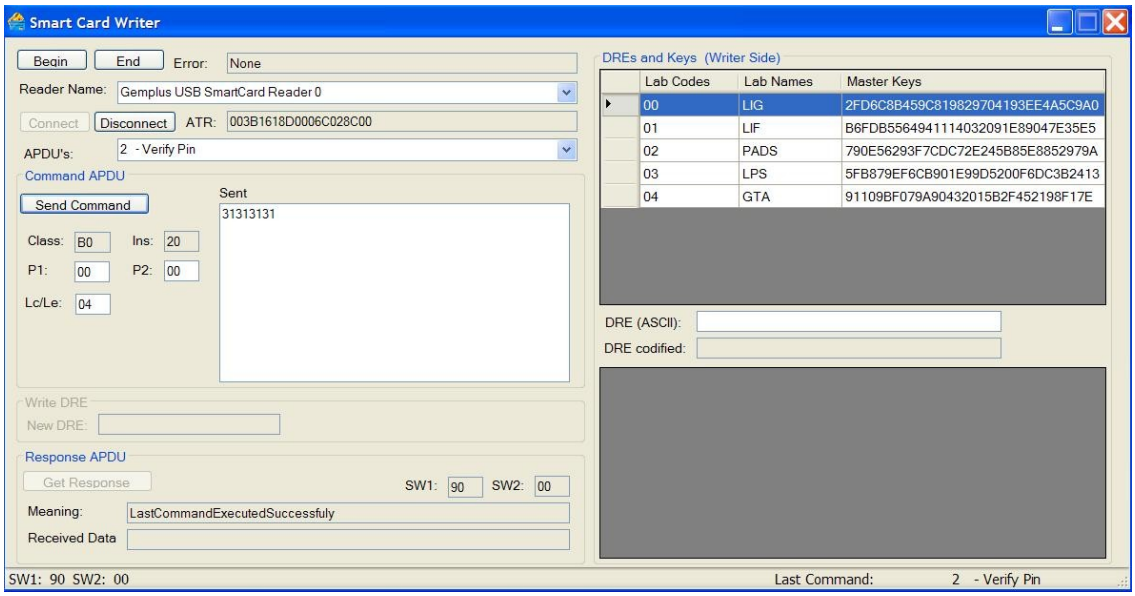


FIGURA 5.10 – COMANDO VERIFY PIN APÓS UPDATE PIN

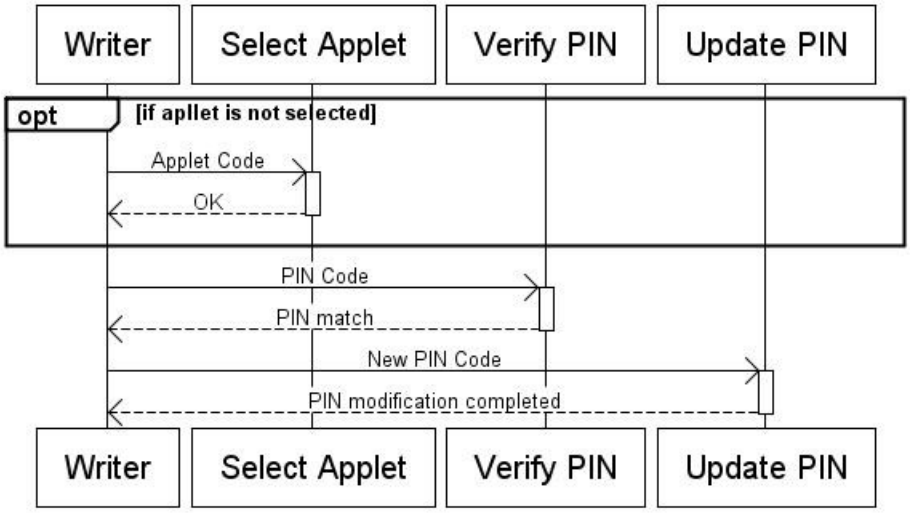
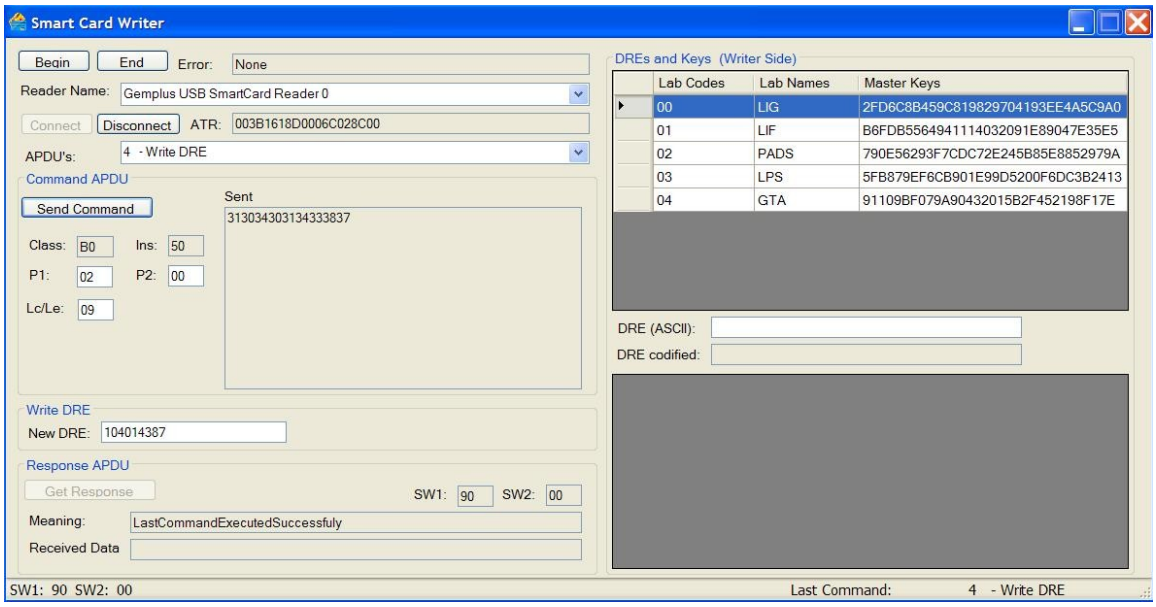


FIGURA 5.11 – DIAGRAMA DE SEQUÊNCIA DO PROCESSO DE ATUALIZAÇÃO DO PIN

O próximo passo é inserir o DRE do aluno no *smart card* pelo comando "*write DRE*", como mostrado na figura 5.12.



DRE.

As chaves devem ser armazenadas no cartão somente quando o DRE do aluno já estiver gravado, isto se deve pelo fato de que as chaves de cada laboratório são geradas a partir do DRE do aluno e da chave "Mãe" de cada laboratório que o aluno tem acesso. Deste modo, para gerar e armazenar as chaves no *smart card* e no banco de dados dos laboratórios primeiramente é enviado a *APDU* "*Request* DRE", que retorna como resposta o DRE armazenado no cartão. O resultado deste comando é mostrado na figura 5.13, onde o DRE recebido em *ASCII* é exibido no campo "DRE". O campo "DRE *codified*" mostra a codificação do DRE que será utilizada para a geração da chave "Filha", detalhes desse processo estão descritos no apêndice B.

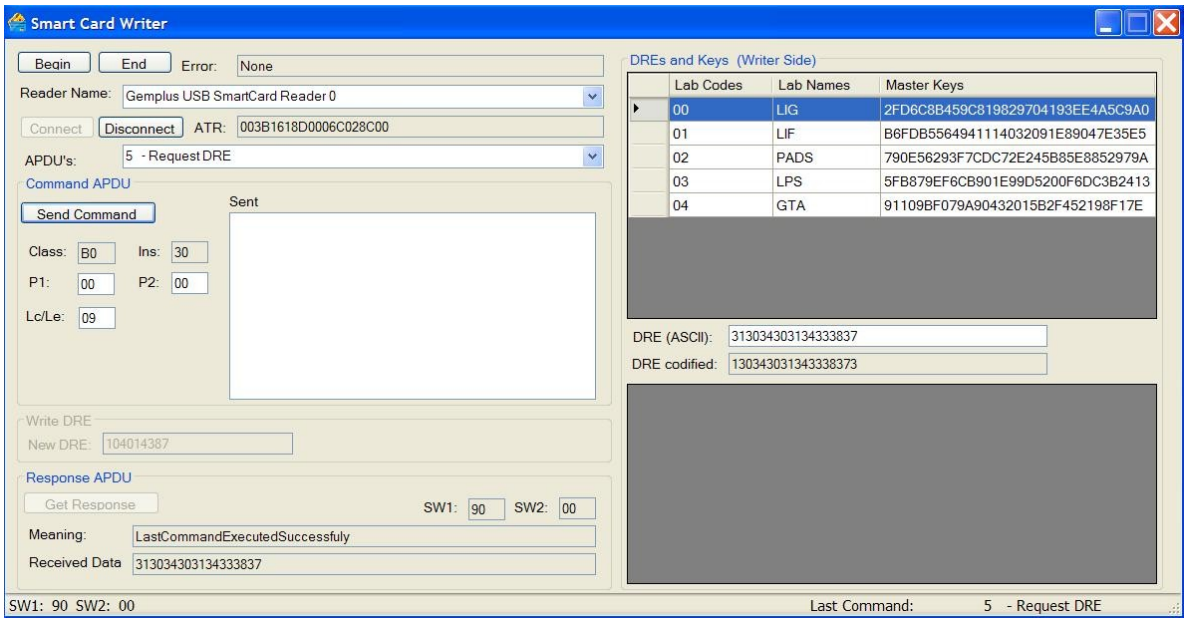


FIGURA 5.13 – COMANDO REQUEST DRE

Após o recebimento do DRE que está armazenado no *smart card* a aplicação *host* pode enviar o comando "*Write Default Keys*" que irá armazenar no cartão as chaves "Filhas" geradas. A aplicação *host* deve atualizar também os arquivos de cada laboratório onde o aluno tem acesso com as chaves geradas e seu DRE. Na figura 5.15 é mostrado o comando "*Write Default Keys*" sendo enviado ao cartão a *APDU* como dado, seguindo a estrutura da figura 5.1, as chaves geradas para os laboratórios *default* que estão selecionados na tabela (LIG, PADS e GTA). No caso exemplificado, a aplicação deve armazenar também as chaves geradas e o DRE do aluno nos arquivos "*DREContentDataLab00*.txt", "*DREContentDataLab02*.txt" e "*DREContentDataLab04*.txt". A figura 5.16 mostra como ficou o arquivo "*DREContentDataLab00*.txt" depois da execução do comando.

Esse processo de escrever o DRE no cartão e atualizar todas as chaves logo em seguida pode ser entendido como um processo de criação do cartão, cuja seqüência está detalhada na figura 5.14

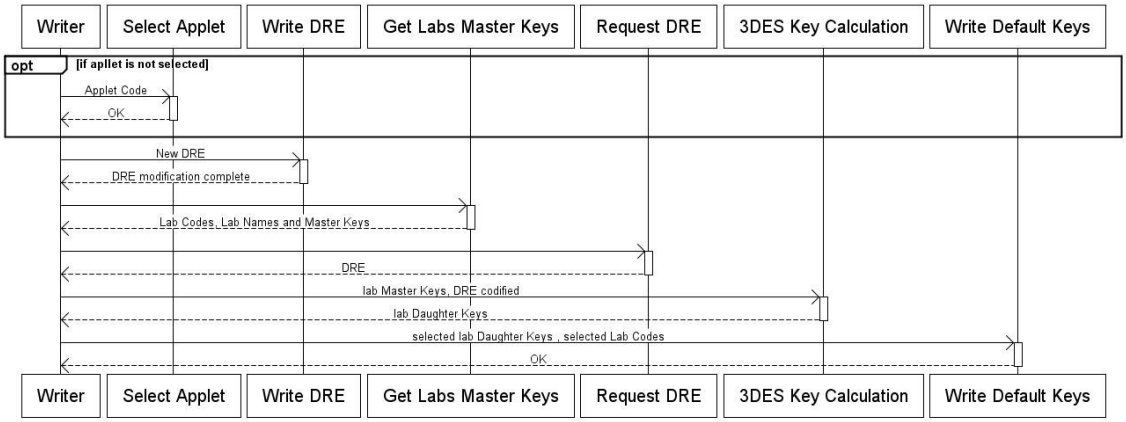


FIGURA 5.14 – DIAGRAMA DE SEQUÊNCIA DE INSERÇÃO DE USUÁRIOS

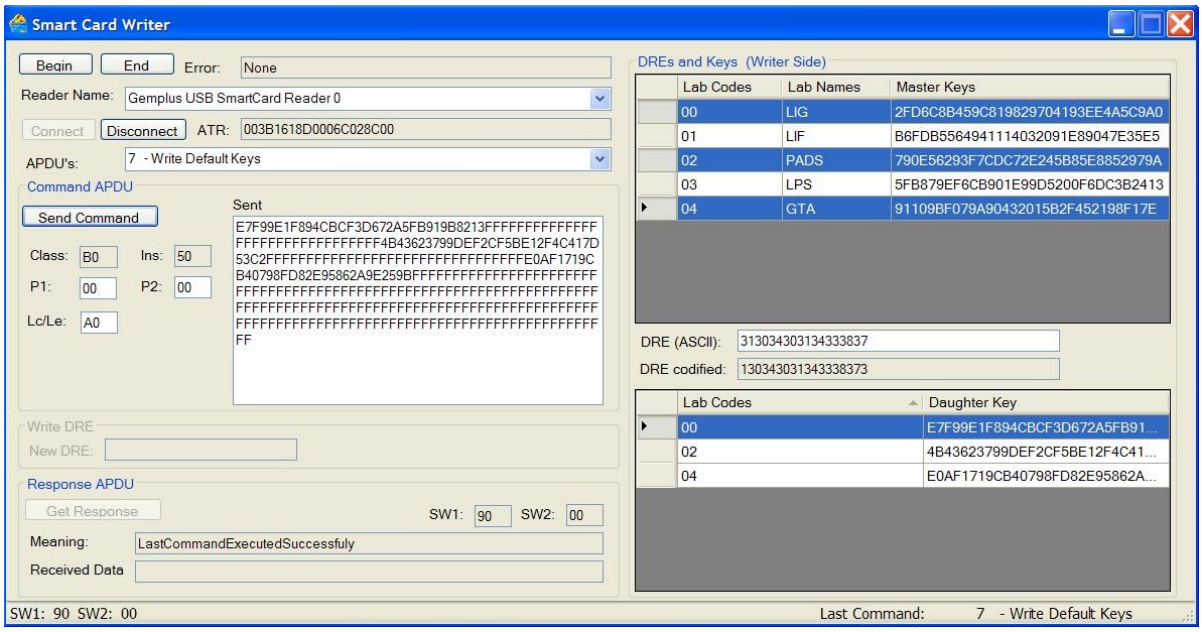


FIGURA 5.15 – COMANDO WRITE DEFAULT KEYS

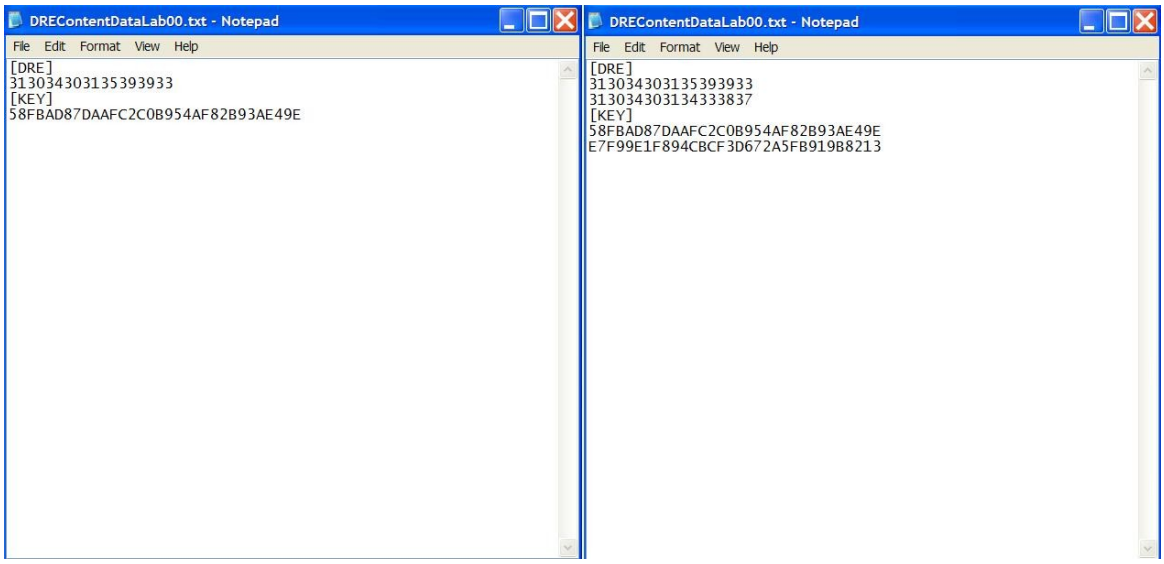


FIGURA 5.16 – DRECONTENTDATALAB00 ANTES E DEPOIS, RESPECTIVAMENTE

Uma chave também pode ser atualizada individualmente através do comando "*Write Lab Key*" no caso em que o aluno se associar ao laboratório posteriormente, por exemplo. Deve-se selecionar um dos laboratórios da lista para que a sua chave filha seja gerada a partir do DRE. A figura 5.17 ilustra o procedimento para inclusão da chave do LIF, cujo código é 01, indicado no parâmetro "P2" da *APDU* e a figura 5.18 descreve seqüencialmente o processo.

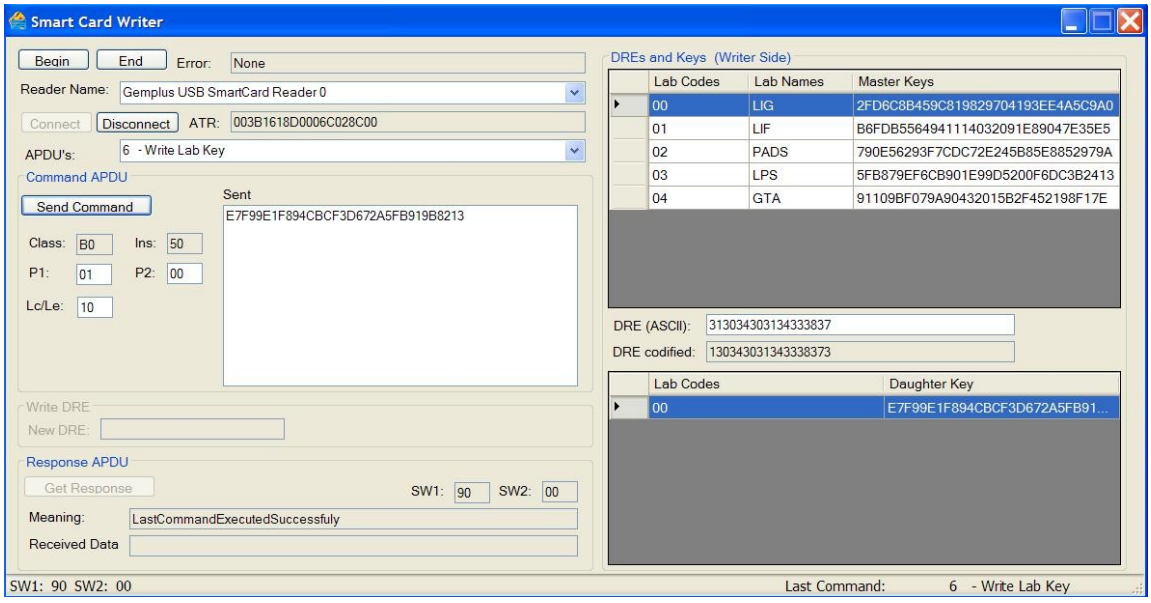


FIGURA 5.17 – COMANDO WRITE LAB KEY

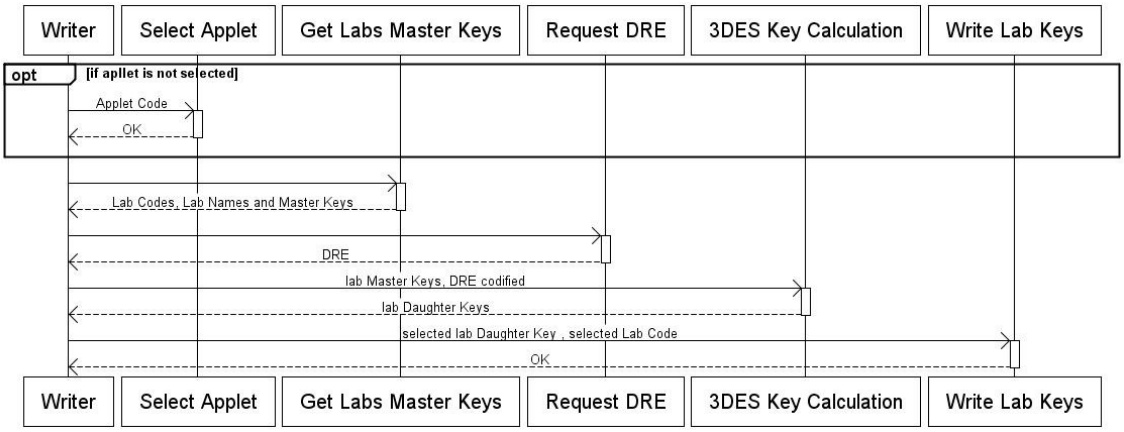


FIGURA 5.18 – DIAGRAMA DE SEQUÊNCIA DE ESCRITA DE CHAVE

Capítulo 6

Programa *host* leitor

Neste capítulo será apresentado o programa *host* que é responsável por ler os dados necessários no *applet*, que reside no *smart card*, para autenticação dos usuários nos laboratórios do DEL.

6.1 – Contextualização

A aplicação *host* leitor será responsável pela comunicação entre o usuário, o *applet Java Card* e o aplicativo *back-end* e visa oferecer um mecanismo simples e ágil para a realização de atividades de manipulação de um cartão, no caso em âmbito de aluno, podendo realizar basicamente a operação de autenticação nos laboratórios do DEL.

A aplicação residente no cartão é a responsável por armazenar as informações necessárias para identificação, atualização e autenticação do aluno nos laboratórios do DEL e realizar os cálculos necessários em todo o processo de autenticação utilizando métodos de criptografia de dados. O *applet* usado na comunicação com o *host* leitor é o mesmo utilizado para o aplicativo *host* escritor, o *applet* SmartDEL, que foi detalhado no capítulo anterior.

6.2 – Aplicação *Host* Leitor

A aplicação *host* leitora é responsável por ler os dados necessários e realizar os cálculos para identificação e autenticação do usuário com os laboratórios.

Para o seu desenvolvimento foi utilizada a ferramenta *Visual Studio 2005*, cedido por Luís Renato Azevedo de Araujo Silva (usuário registrado no *Academic Alliance*), sendo a linguagem de programação vb.net. Para a conexão com o *CAD* foram utilizados métodos da biblioteca *Winscard.dll*, biblioteca nativa do *Windows*. Mais detalhes da biblioteca, seus métodos e utilização podem ser encontrados no anexo A.

Na figura 6.1 pode ser vista a interface gráfica da aplicação *host* leitora.

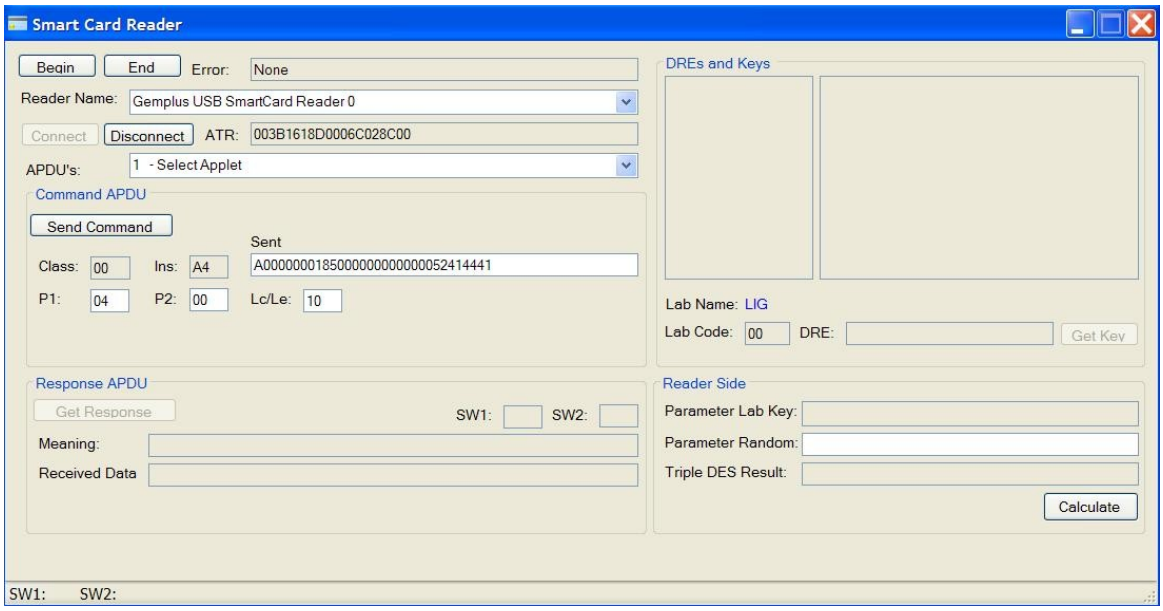


FIGURA 6.1 – INTERFACE GRÁFICA DA APLICAÇÃO *HOST* LEITORA

Para que o processo de autenticação seja possível, os dados referentes as chaves do laboratório devem estar armazenados no *smart card* e no banco de dados do laboratório. Por questões didáticas todas as informações necessárias para o funcionamento da aplicação de leitora foram armazenadas em arquivos especiais, cuja estrutura é descrita no apêndice C. A lista de todos os arquivos e a informação armazenada está descrita na tabela 6.1.

Arquivo	Conteúdo
SmartCardReader.config	Nome da *APDU* e do Laboratório acessado (XX) e Códigos da *APDU* e do Laboratório acessado (XX)
DREContentDataLabXX.txt	DRE dos alunos permitidos no laboratório xx e suas respectivas chaves filhas.

TABELA 6.1 – ARQUIVOS DO PROGRAMA *HOST* ESCRITOR

O arquivo "*SmartCardReader.config*" relaciona o nome do comando *APDU* ao seu respectivo código binário, com base neste arquivo a programa *host* preenche todos os comandos *APDU*s na combo "*APDU*'s" e que futuramente serão enviados ao cartão.

Cada item da combo "*APDU*s" representa uma chamada ao método do *Applet* SmartDEL, com exceção do "*Select Applet*", que representa a própria seleção do a*pplet* e obrigatoriamente deve ser feita.

A tabela 6.2 a seguir relaciona os nomes na combo "*APDU*s" com os métodos do SmartDEL.

Método	Nome de *APDU* relacionado
Verify	Verify PIN
sendID	Request DRE
sendResponse	Request RES e Get Response
updatePIN	Update PIN
keysInitialization	Não utilizado
updateSpecificKey	Não utilizado
writeID	Não utilizado

TABELA 6.2 – RELAÇÃO ENTRE OS MÉTODOS E SUAS *APDU*S

Nesse arquivo também estão armazenados o código e nome do laboratório que está sendo acessado e com base nisso são preenchidos os campos "*Lab Name*" e "*Lab Code*" na interface da leitora.

Sabendo o código XX do laboratório podemos abrir o arquivo "*DREContentDataLabXX.txt*" e obter a relação aluno-chave filha que será utilizada no processo de autenticação.

Na figura 6.2 pode ser vista a modelagem dos arquivos mencionados.

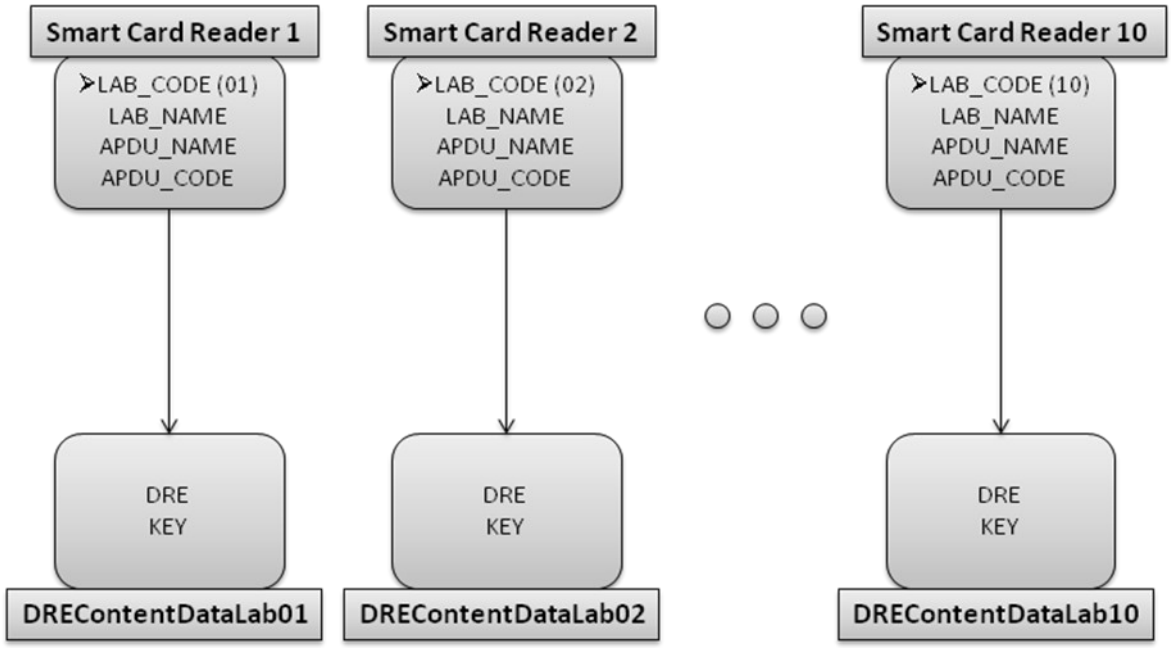


FIGURA 6.2 – MODELAGEM DOS ARQUIVOS.

Para iniciar o processo de comunicação do cartão com a aplicação *host* deve-se inserir um *smart card* a um *CAD* conectado ao *PC* em seguida clicar no botão "*Begin*" e escolher a leitora na qual foi inserida o cartão. Na figura 6.3 é ilustrado o resultado do procedimento no caso em que foi escolhida a leitora "*Gemplus USB SmartCard Reader 0"*, cedida gentilmente pela empresa Gemalto Cartões e Terminais Ltda.

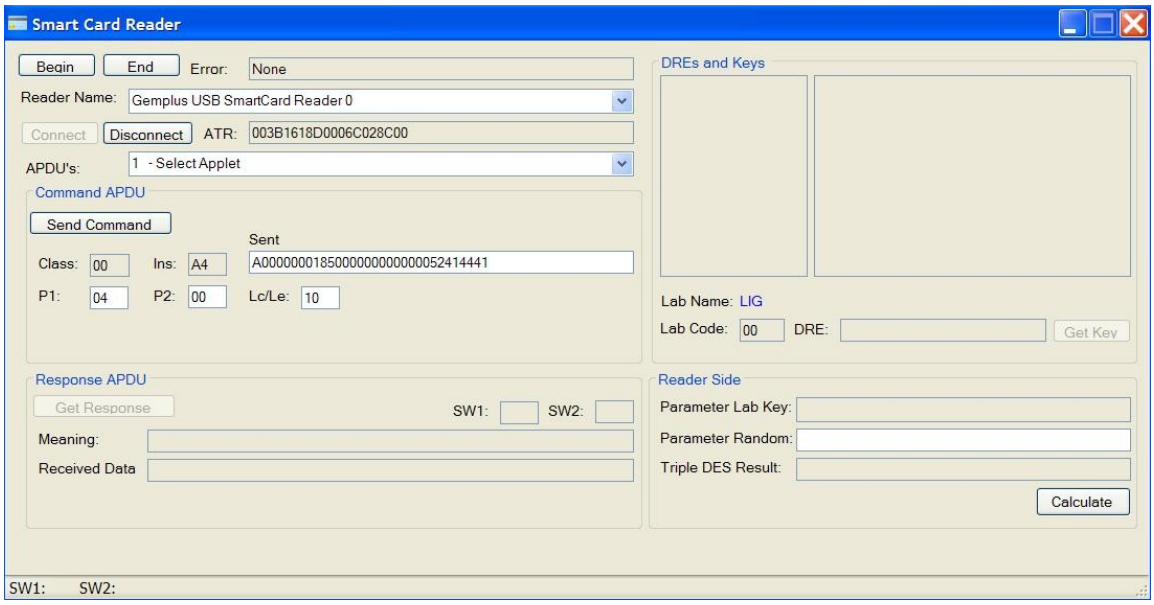


FIGURA 6.3 – ESCOLHA DA LEITORA

Antes que qualquer outro comando seja enviado ao *applet* deve-se selecioná-lo previamente enviado o comando "*Select Applet*" para que o *JCRE* encaminhe os comandos *APDU*s enviados ao cartão para o *applet* selecionado. Este procedimento é realizado ao clicar no botão "*Send Command*" após selecionar o comando desejado. O resultado do comando "*Select Applet*" ao selecionar SmartDEL é ilustrado na figura 6.4.

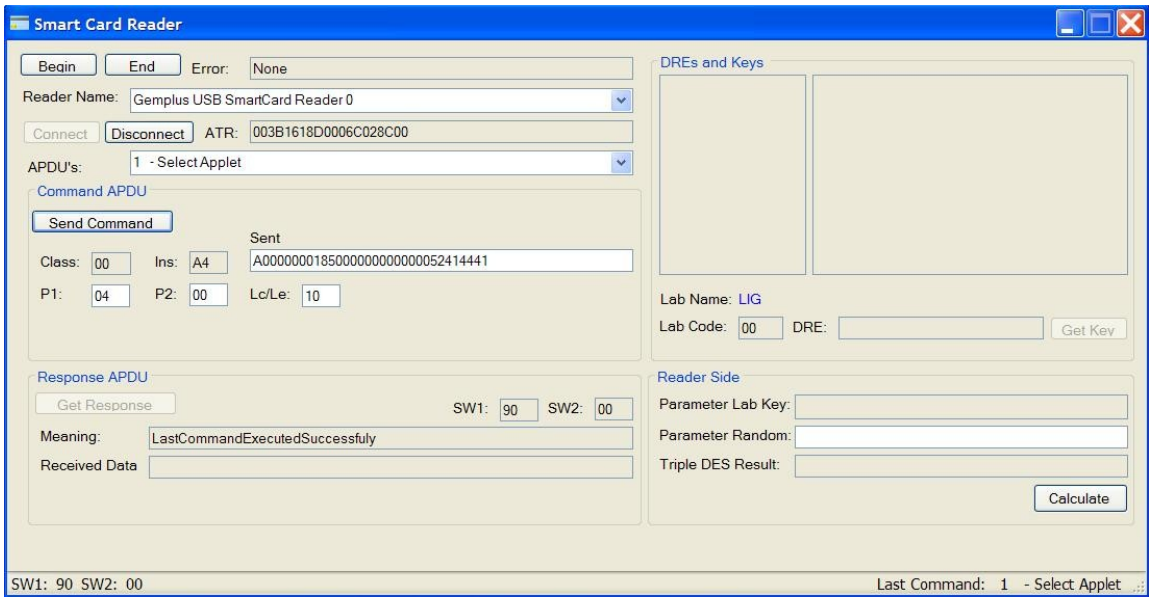


FIGURA 6.4 – RESULTADO DO COMANDO SELECT *APPLET*

Em um primeiro momento, antes que seja realizada qualquer interação entre a leitora do laboratório e o cartão, o usuário deve provar que é o legítimo portador do *smart card*. O usuário deve inserir o código *PIN* do seu cartão e através do comando "*Verify PIN*" a leitora envia o valor inserido ao cartão. No caso de resposta afirmativa o usuário prova que é o legitimo portador. Existem outras formas mais eficientes de provar a legitimidade do usuário como a verificação biométrica, onde informações das digitais do usuário são armazenadas no cartão e são comparadas com as informações coletadas no momento em que o usuário realizará o acesso. Apesar de sua incontestável eficiência, este tipo de verificação foge do escopo do projeto devido a sua complexidade. Na figura 6.5 é mostrado o resultado de enviar o comando "*Verify PIN*" com o código *default* "30303030".

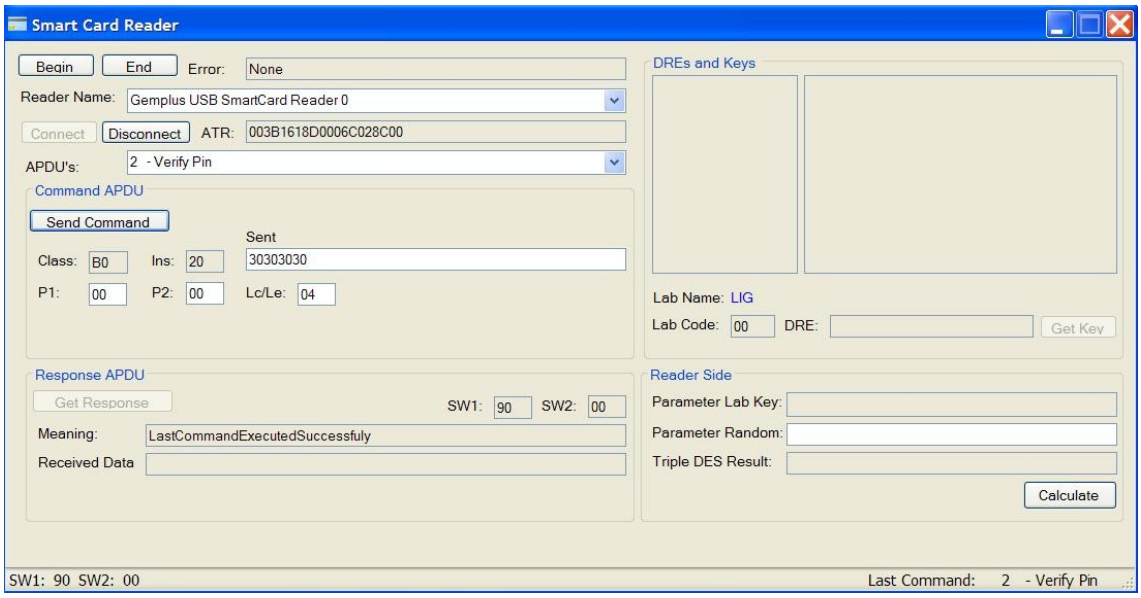


FIGURA 6.5 – RESULTADO DO COMANDO VERIFY PIN

Por questões de segurança é recomendado que o valor default do *PIN* seja alterado pelo aluno. Para alterar o valor do *PIN* deve-se primeiramente apresentar o código atual enviando o comando "*Verify PIN*" e posteriormente enviar o comando "*Update PIN*" com o novo valor. Caso haja uma tentativa de alterar o *PIN* antes de ser apresentado o código atual o *applet* retorna um erro. As figuras, 6.6 a 6.10 ilustram o procedimento da troca de *PIN* de "30303030" (0000) para "31313131". A figura 6.11 mostra um diagrama de sequência para esse processo.

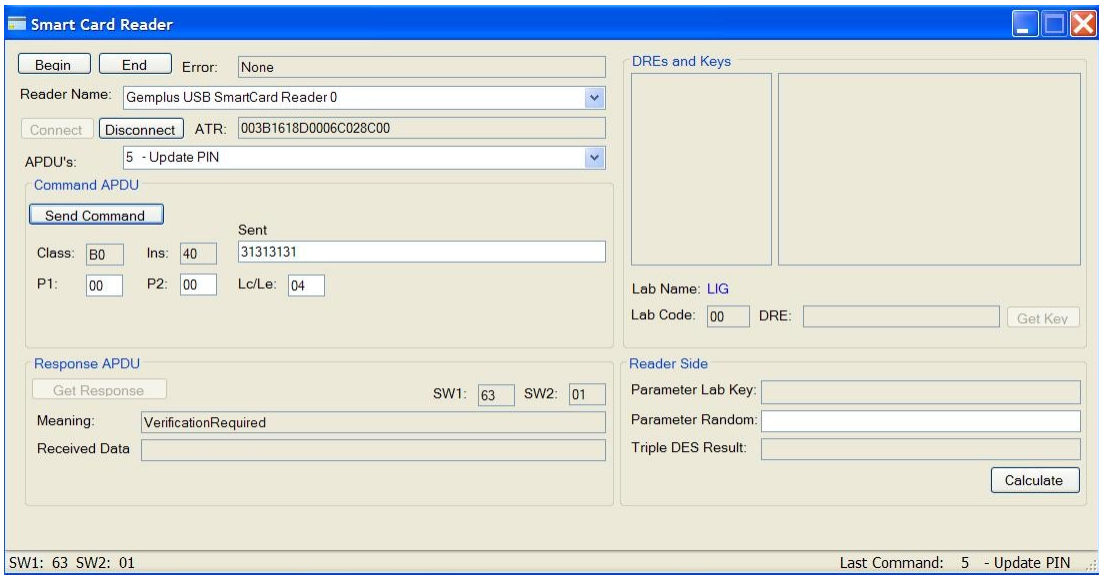


FIGURA 6.6 – RESULTADO DO UPDATE PIN ANTES DE VERIFY PIN

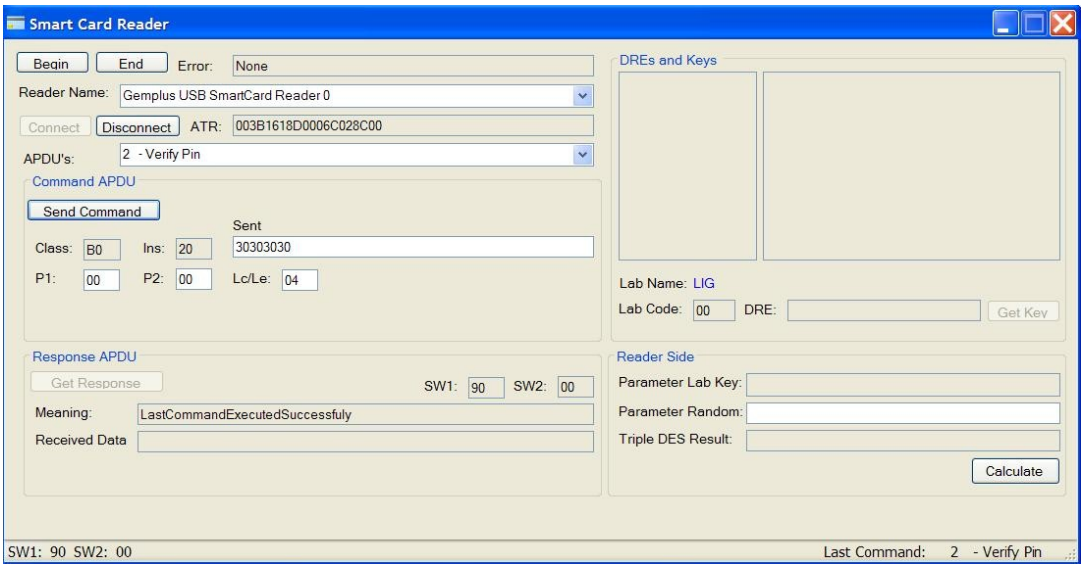


FIGURA 6.7 – COMANDO VERIFY PIN

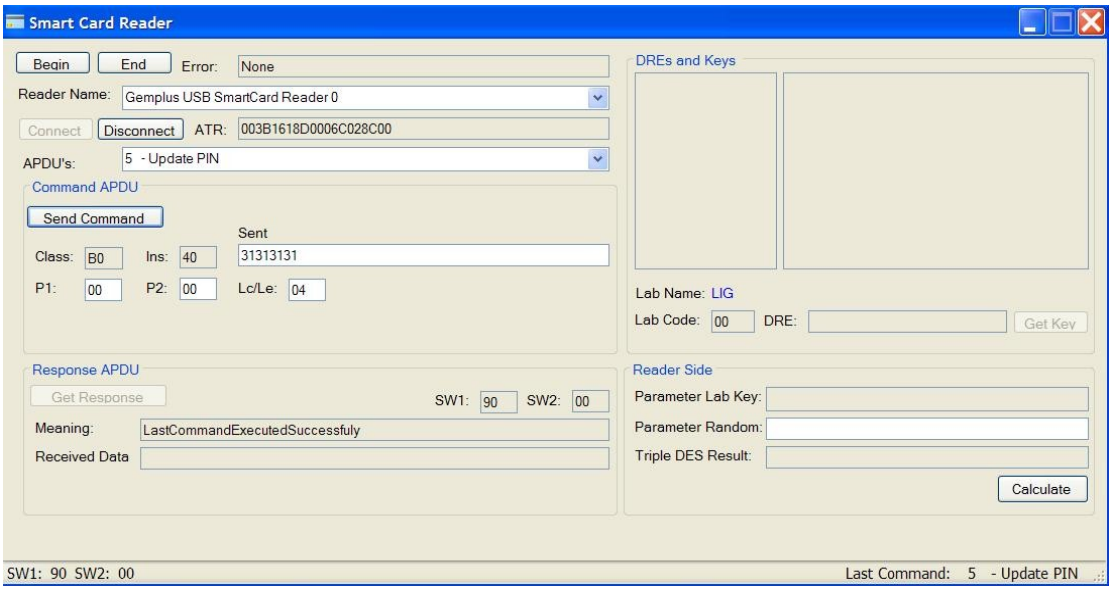


FIGURA 6.8 – COMANDO UPDATE PIN DEPOIS DE VERIFY PIN

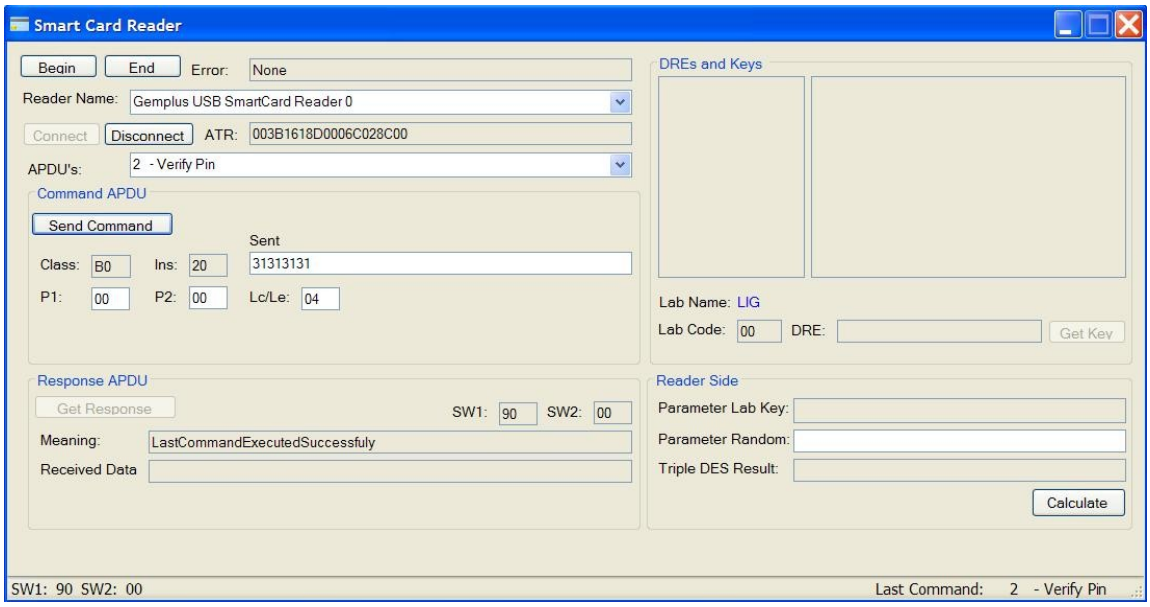


FIGURA 6.9 – COMANDO VERIFY PIN VALIDANDO O NOVO PIN

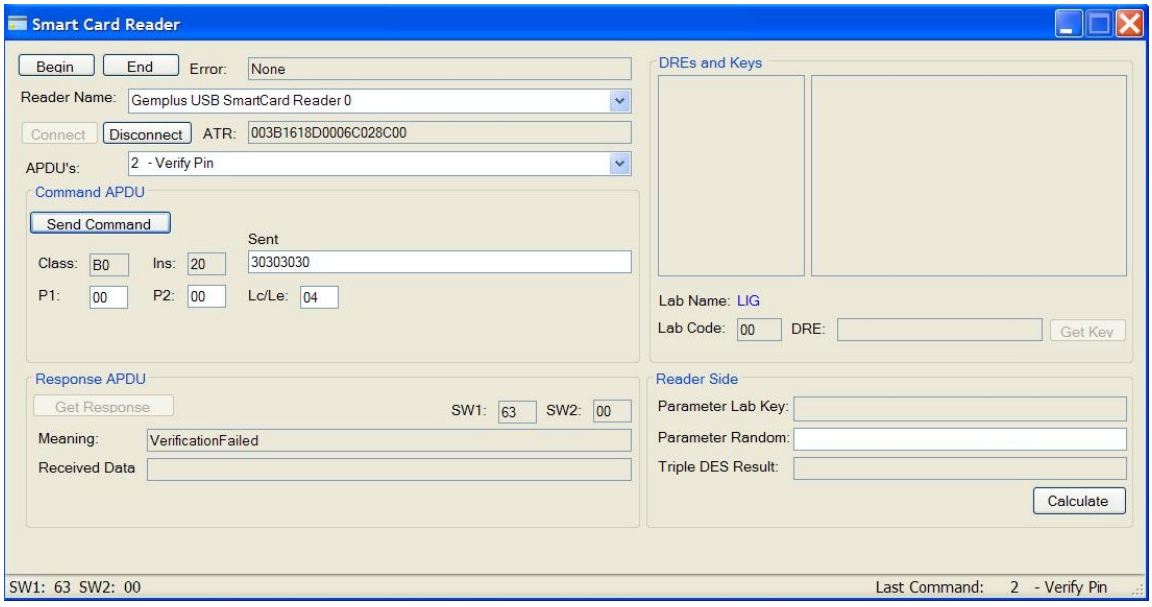


FIGURA 6.10 – COMANDO VERIFY PIN COM O PIN ANTERIOR

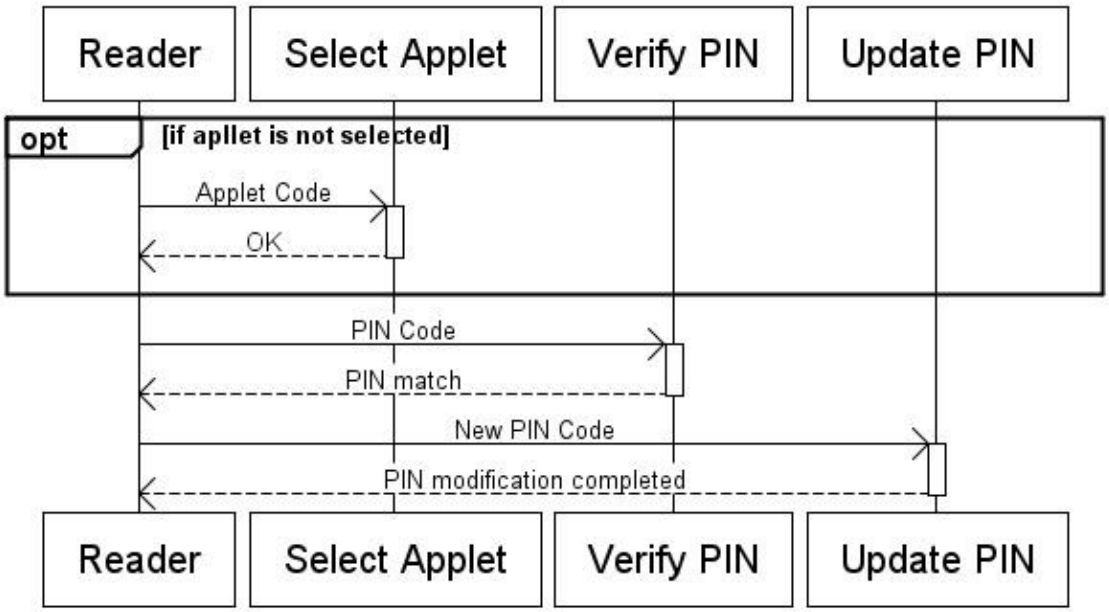


FIGURA 6.11 – DIAGRAMA DE SEQUÊNCIA DO PROCESSO DE ATUALIZAÇÃO DO PIN

Uma vez que o usuário prove a sua legitimidade o passo a seguir é a identificação do usuário por parte do laboratório. Este processo é realizado através da leitura do DRE armazenado no *smart card* através do comando "*Request* DRE". Em posse do DRE do usuário o laboratório que está sendo acessado procura em sua base (no caso "*DREContentDataLabXX*.txt") se esta pessoa possui permissão de acesso, ou seja, se ela consta em seu banco de dados. Se for encontrada uma ocorrência do DRE a pessoa tem permissão de acesso e o processo de autenticação continua, no entanto, todavia deve-se verificar se a chave que está no banco de dados do laboratório é a mesma que está armazenada no *smart card* para este laboratório. A figura 6.12 ilustra o resultado deste procedimento, neste caso foi retornado o DRE "104014387" pelo cartão. Ao pressionar o botão "*Get Key*" a chave relacionada ao DRE para o laboratório escolhido é obtido no *DREContentDataLabXX*.txt.

Na figura 6.13 é mostrado o exemplo da chave obtida para o laboratório, cujo código é 03 para o DRE "104014387".

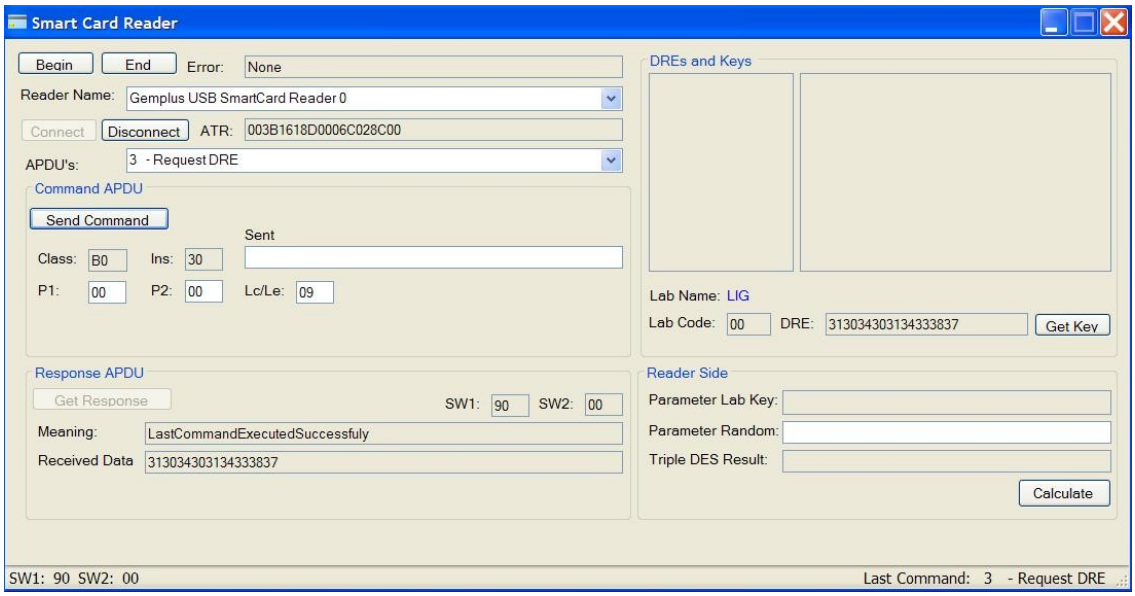


FIGURA 6.12 – RESULTADO DO COMANDO REQUEST DRE

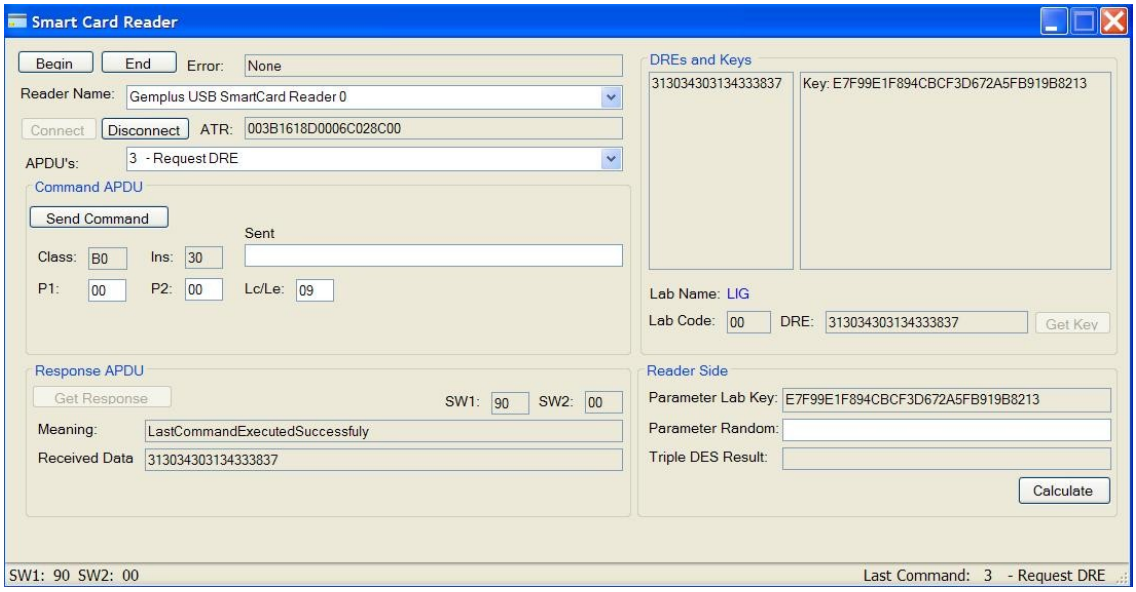


FIGURA 6.13 – CHAVE RELACIONADA AO DRE ENCONTRADA PELA APLICAÇÃO *HOST*

Para realizar a comparação das chaves o laboratório envia ao *smart card* o comando *APDU* "*Request RES*", que possui como dado um número randômico de 16 *bytes* e o código do laboratório como parâmetro "P2". Ao receber o dado randômico o cartão o encripta utilizando o algoritmo *T-DES* em modo CBC com a chave relacionada ao laboratório identificado em "P2" e espera o comando "*Get Response*" para enviar como resposta os 8 primeiros *bytes* do resultado, os 8 *bytes* restantes são mantidos para serem utilizados como chave de sessão para uma eventual troca de dados. O laboratório, por sua vez, realiza a mesma operação com a chave relacionada ao DRE obtido

anteriormente (figura 6.12) e o dado randômico enviado e compara os 8 primeiros *bytes* da resposta obtida com o resultado que foi informado pelo *smart card*. Se o resultado observado são equivalentes, significa que as entidades possuem a mesma chave e o acesso é concedido.

As figuras 6.14 a 6.16 ilustram o processo de comparação das chaves entre o laboratório e *smart card*. Na figura 6.17 é mostrado um digrama de seqüência mostrando toda a operação.

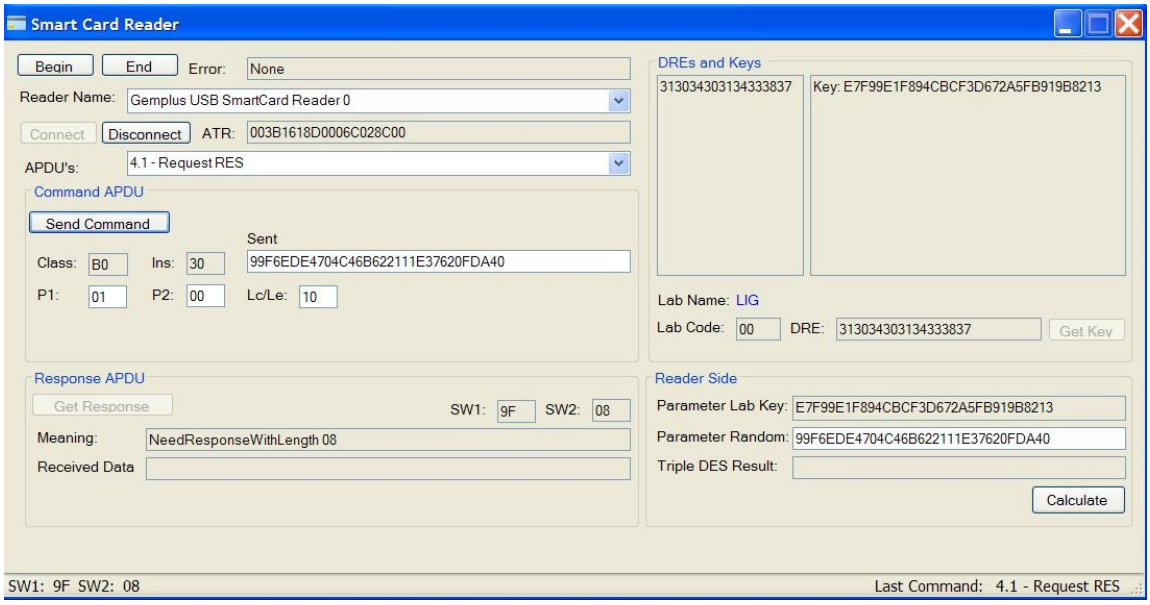


FIGURA 6.14– COMANDO REQUEST RES ENVIA DADO RANDOMICO E CODIGO DO LAB

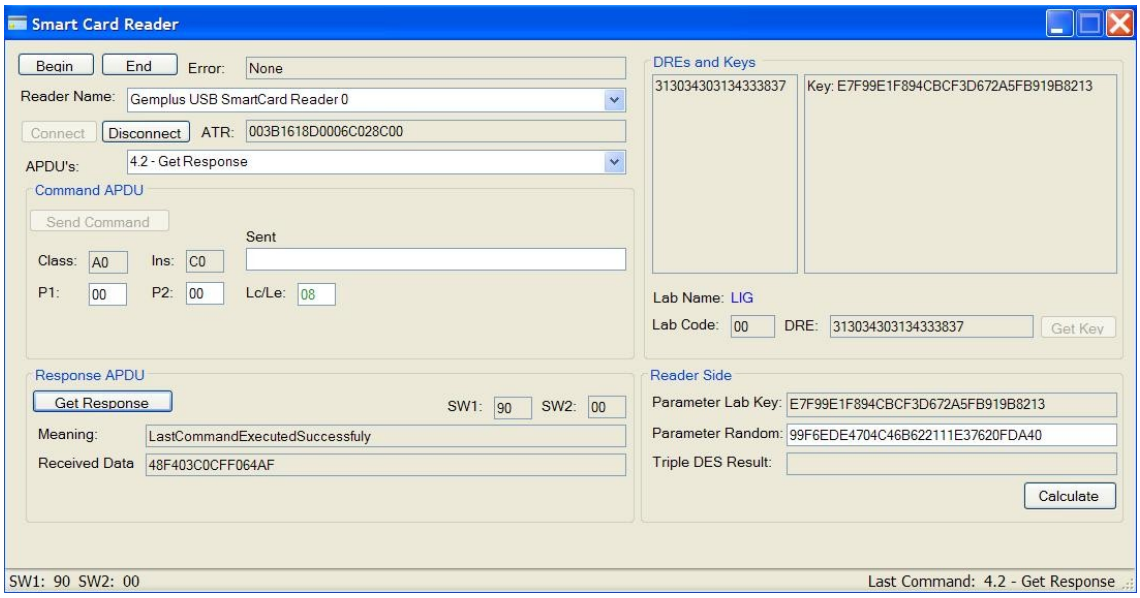


FIGURA 6.15 – COMANDO GET RESPONSE

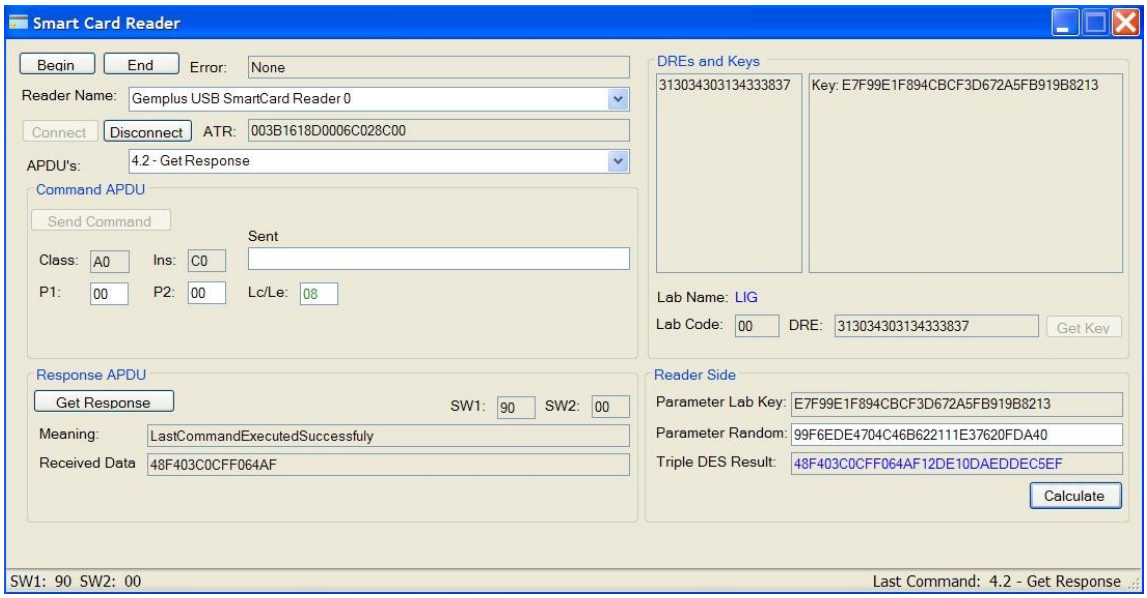


FIGURA 6.16 – COMPARAÇÃO DOS RESULTADOS DO CARTÃO E APLICAÇÃO *HOST*

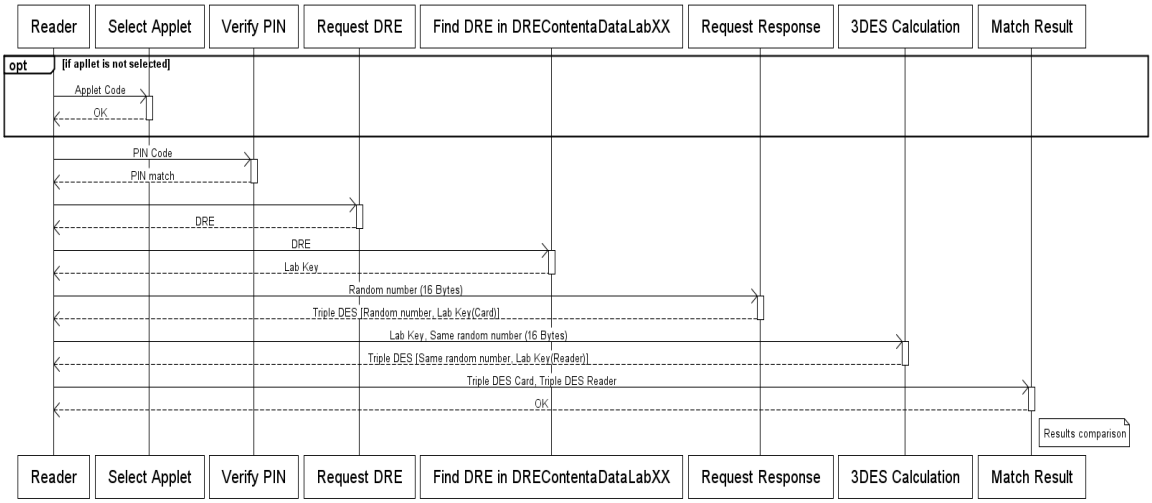


FIGURA 6.17 –DIAGRAMA DE SEQÜÊNCIA DE AUTENTICAÇÃO

Capítulo 7

Conclusão

Os atuais sistemas de segurança possuem uma tecnologia ultrapassada, visto que seus componentes limitados não garantem transações tão seguras e nem tão abrangentes como as possibilitadas pelos *Smart cards* combinadas com a linguagem *Java Card*. A permanência desses sistemas só se dá devido à infra-estrutura já existente, sendo que as adaptações para receber *Smart cards*, cuja semelhança com um micro-computador é bastante evidente, estão em ascensão.

Neste trabalho foram empregados cartões dotados de *chip* de contato e microprocessador. Sob esta ótica foi realizado um estudo das características principais dos cartões inteligentes (*Smart cards*), enfatizando a tecnologia *Java Card* como plataforma de *software*, e foram mostrados aspectos estruturais e algumas funcionalidades referentes a essa tecnologia. Neste sentido, foi demonstrado, pelos sistemas ilustrativos de autenticação de usuários nos laboratórios do DEL que a estrutura interna dos *Smart cards* garante um nível elevado de segurança em transações e confiança em processos de autenticação.

A tecnologia por trás dos *Smart cards* é, portanto, um campo promissor, visto que cartões fazem parte da vida das pessoas e fazer uso do mesmo é algo extremamente simples. Trata-se de uma tecnologia que influencia os indivíduos diretamente em diversas áreas, de diversas maneiras, seja no acesso a um escritório, no metrô, no trabalho e até mesmo em suas próprias casas.

Bibliografia

[1] CHEN, Zhiqun. ***Java Card* (tm) Technology for *Smart cards*: Architecture and Programmer's (The Java Series) (Paperback).** Addison-Wesley Professional, 1st edition. 2000. 400p.

[2] CHUCK, Wilson. **Get Smart The Emergence of *Smart cards* in the United States and their Pivotal Role in Internet Commerce.** Mullaney. 2001.

[3] GADELLAA, K.O. **Fault Attacks on *Java Card* (Master´s Thesis)**. Universidade de Eindhoven. 2005.

[4] GUILLOU, L. C. – UGON, M. e QUISQUATER, J-J. **Cryptographic authentication protocols for *smart cards***. Computer Networks. 2001.

[5] ISO/IEC, Int. Standards Organization. **7816-4:1995 Information technology - Identification cards – Integrated circuit(s) cards with contacts part 4: Inter-Industry commands for interchange**. Genebra, Suiça. 1995.

[6] ISO/IEC, Int. Standards Organization. **7816-3:1997 Information technology - Identification cards – Integrated circuit(s) cards with contacts part 3: Electronic signals and transmission protocols**. Genebra, Suiça. 1997.

[7] ISO/IEC, Int. Standards Organization. **7816-2:1999 Information technology - Identification cards – Integrated circuit(s) cards with contacts part 2: Dimensions and location of the contacts**. Genebra, Suiça. 1999.

[8] JONG, Eduard – PIETER, Hartel – PEYRET, Patrice – CATTANEO, Peter. ***Java Card*: An analysis of the most successful *smart card* operating system to date**. University Twente. 2005.

[9] SUN MICROSYSTEMS, Inc. ***Java Card* Platform, Runtime Environment Specification**. Disponível em <http://java.sun.com/products/javacard/specs.html>. Acesso em 23 de Outubro 2008.

[10] SUN MICROSYSTEMS, Inc. **Documentation about *Smart card* Overview**. Disponível em <http://java.sun.com/products/javacard/smartcards.html>. Acesso em 23 de Outubro 2008.

[11] SUN MICROSYSTEMS, Inc. **An Introduction to *Java Card* Technology - Part 1.** Disponível em <http://java.sun.com/javacard/reference/techart/javacard1/>. Acesso em 3 de Dezembro de 2008.

[12] SUN MICROSYSTEMS, Inc. **An Introduction to *Java Card* Technology - Part 2.** Disponível em

<http://java.sun.com/javacard/reference/techart/javacard2/>. Acesso em 3 de Dezembro de 2008.

[13] SUN MICROSYSTEMS, Inc. **An Introduction to *Java Card* Technology - Part 3.** Disponível em <http://java.sun.com/javacard/reference/techart/javacard3/>. Acesso em 3 de Dezembro de 2008.

[14] WOLFGANG, Rankl e WOLFGANG, Effing. ***Smart card* handbook**. John Wiley & Sons, 3rd edition. 2004. 1120p.

[15] ECLIPSE FOUNDATION, THE. Disponível em <http://www.eclipse.org/>. Acesso em 9 de Setembro de 2008.

[16] SUN MICROSYSTEMS, Inc. ***JAVA CARD* DEVELOPMENT KIT.** Disponível em <http://java.sun.com/j2se/1.4.2/download.html>. Acesso em 9 de Setembro de 2008.

[17] SourceForge, Inc. **EclipseJCDE.** Disponível em <http://sourceforge.net/projects/eclipse-jcde/>. Acesso em 9 de Setembro de 2008.

[18] IBM Research, Inc. **IBM Research Develops Technology to Protect GSM Cell Phones' ID Cards From Hacker Attacks.** Disponível em <http://domino.research.ibm.com/comm/pr.nsf/pages/news.20020507_simcard.html>. Acesso em 20 de Março de 2009.

[19] 3GPP, 3rd Generation Mobile Group. **3GPP TS 35.206 3G Security; Specification of the MILENAGE algorithm set: An example algorithm set for the 3GPP authentication and key generation functions f1, f1*, f2, f3, f4, f5 and f5*; Document 2: Algorithm specification**. Release 8.

Apêndice A

Processo de Autenticação

O processo de autenticação consiste na troca de chaves entre o cartão e o laboratório, se ambas as partes possuem a mesma chave para o usuário em questão a permissão é concedida. No entanto, por questões de segurança a chave não é trocada de forma direta entre as entidades. Deste modo, para que o processo de autenticação seja possível é necessário que ambas as partes possuam as mesmas informações, pois trata-se de uma autenticação simétrica. No *smart card* estão armazenadas as chaves de todos os laboratórios que o aluno tem acesso, esta chave é gerada a partir de seu DRE e a chave Mãe do laboratório, cujo processo de geração da chave é mostrado em detalhes no apêndice B. O laboratório, por sua vez, armazena as chaves geradas e as relacionam aos DREs dos alunos que tem permissão de acesso.

Em um primeiro momento, antes que seja realizada qualquer interação entre a leitora do laboratório e o cartão, o usuário deve provar que é o legítimo portador do *smart card*. O usuário deve inserir o código *PIN* do seu cartão e através do comando "*Verify PIN*" a leitora envia o valor inserido ao cartão. No caso de resposta afirmativa o usuário prova que é o legitimo portador. Existem outras formas mais eficientes de provar a legitimidade do usuário como a verificação biométrica, onde informações das digitais do usuário são armazenadas no cartão e são comparadas com as informações coletadas no momento em que o usuário realizará o acesso. Apesar de sua incontestável eficiência, este tipo de verificação foge do escopo do projeto devido a sua complexidade.

Uma vez que o usuário prove a sua legitimidade o passo a seguir é a identificação do usuário por parte do laboratório. Este processo é realizado através da leitura do DRE armazenado no *smart card* através do comando "*Request* DRE". Em posse do DRE do usuário o laboratório que está sendo acessado procura em sua base se esta pessoa possui permissão de acesso, ou seja, se ela consta em seu banco de dados. Se for encontrada uma ocorrência do DRE a pessoa tem permissão de acesso, no entanto, todavia deve-se verificar se a chave que está no banco de dados do laboratório é a mesma que está armazenada no *smart card* para este laboratório.

Para realizar a comparação das chaves o laboratório envia ao *smart card* o comando *APDU* "*Request* RES", que possui como dado um número randômico de 16 *bytes* e o código do laboratório como parâmetro "P2". Ao receber o dado randômico o cartão o encripta utilizando o algoritmo *T-DES* em modo CBC com a chave relacionada ao laboratório identificado em "P2" e envia como resposta os 8 primeiros *bytes* do resultado, os 8 *bytes* restantes são mantidos para serem utilizados como chave de sessão para uma eventual troca de dados. O laboratório, por sua vez, realiza a mesma operação com a chave relacionada ao DRE obtido anteriormente e o dado randômico enviado e compara os 8 primeiros *bytes* da resposta obtida com o resultado que foi informado pelo *smart card*. Se o resultado observado são equivalentes, significa que as entidades possuem a mesma chave e o acesso é concedido. Na figura abaixo é mostrado um diagrama de seqüência de todo o processo de autenticação.

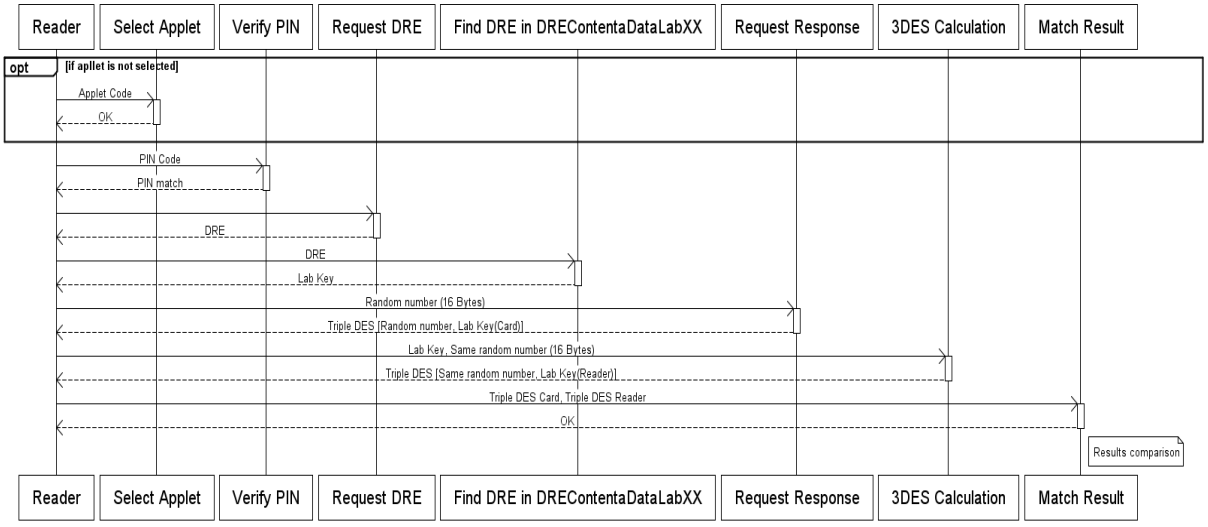


Diagrama de seqüência do processo de autenticação.

Apêndice B

Geração das Chaves Filhas

Para que nenhum aluno possua a mesma chave de autenticação para um dado laboratório foi desenvolvido um método de derivação de chaves. Cada laboratório possui uma chave, denominada "Chave Mãe", a partir da qual serão geradas as chaves para cada aluno utilizando seu DRE. O processo de geração da "Chave Mãe" é randômico. A tabela abaixo mostra a relação laboratório – "Chave Mãe".

Laboratório	Chave Mãe
LIG	2FD6C8B459C819829704193EE4A5C9A0
LIF	B6FDB5564941114032091E89047E35E5
PADS	790E56293F7CDC72E245B85E8852979A
LPS	5FB879EF6CB901E99D5200F6DC3B2413
GTA	91109BF079A90432015B2F452198F17E

Relação entre laboratórios e chaves mãe

Segue o exemplo do processo de derivação de uma chave filha para o aluno com DRE "104014387" no LIG:

1. Transformar o DRE em código ASCII:

104014387 = 313034303134333837

2. Realizar um *swap* entre os nibles do DRE

313034303134333837 = 130343031343338373

3. Aplicar um *Triple DES* em modo *CBC* utilizando a chave Mãe do laboratório.

- MK_LIG = 2FD6C8B459C819829704193EE4A5C9A0
- DRE_SWAPPED = 130343031343338373

TDES (MK_LIG, DRE_SWAPPED) = DK_1040387_LIG

Ao realizar a operação do item 3, temos o resultado "E7F99E1F894CBCF3D672A5FB919B8213", que é a chave filha do LIG para o aluno com o DRE "104014387". Esta chave deve ser armazenada no *smart card* na posição

destinada à chave do LIG (16 primeiros *bytes* do buffer) e no arquivo "*DREContentDataLab00*.txt", que será utilizado no processo de autenticação.

Apêndice C

Estrutura dos arquivos

Na aplicação *host* foi desenvolvida uma estrutura de arquivos para configuração da aplicação e armazenar dados referentes ao laboratório. A seguir será mostrada a estrutura de cada um dos arquivos com seu respectivo conteúdo.

1. SmartCardWriter.config

Este arquivo é responsável por armazenar as configurações da aplicação escritora, basicamente são os comandos *APDU* e seus respectivos nomes.

[*APDU*_NAME]
1 - Select *Applet*
2 - Verify Pin
3 - Update PIN
4 - Write DRE
5 - Request DRE
6 - Write Lab Key
7 - Write Default Keys
[*APDU*_COMMAND]
00A4040010A0000000185000000000000052414441
B02000000430303030
B04000000430303030
B050020009FFFFFFFFFFFFFFFFFF
B030000009
B05001LL10FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
B0500000A0FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFF

2. MasterKeyFile.txt

Este arquivo relaciona cada laboratório ao seu código e à sua chave Mãe

[LAB_NAME]
LIG
LIF
PADS
LPS
GTA
[LAB_CODE]
00
01
02
03
04
[MK]
2FD6C8B459C819829704193EE4A5C9A0
B6FDB5564941114032091E89047E35E5
790E56293F7CDC72E245B85E8852979A
5FB879EF6CB901E99D5200F6DC3B2413
91109BF079A90432015B2F452198F17E

3. *DREContentDataLabXX*.txt

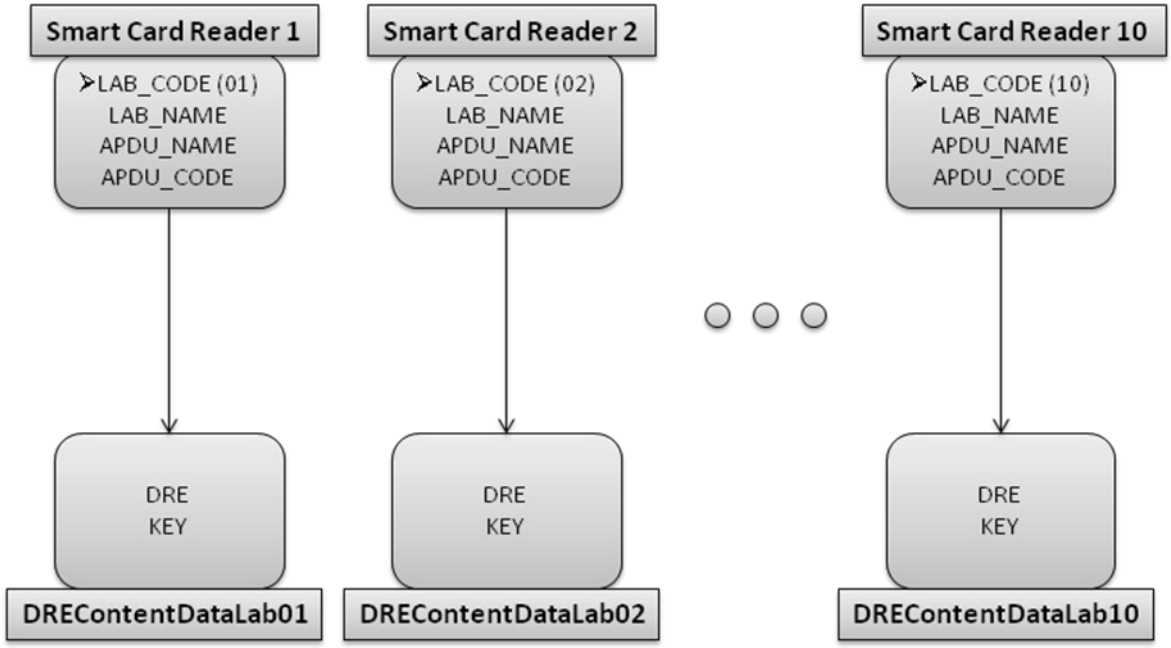
Este arquivo é especifico por laboratório, onde XX é o código do laboratório. Este arquivo relaciona todos os DREs que tem acesso ao laboratório com suas chaves filhas.

[DRE]
313034303135393933
313034303134333837
[KEY]
58FBAD87DAAFC2C0B954AF82B93AE49E
E7F99E1F894CBCF3D672A5FB919B8213

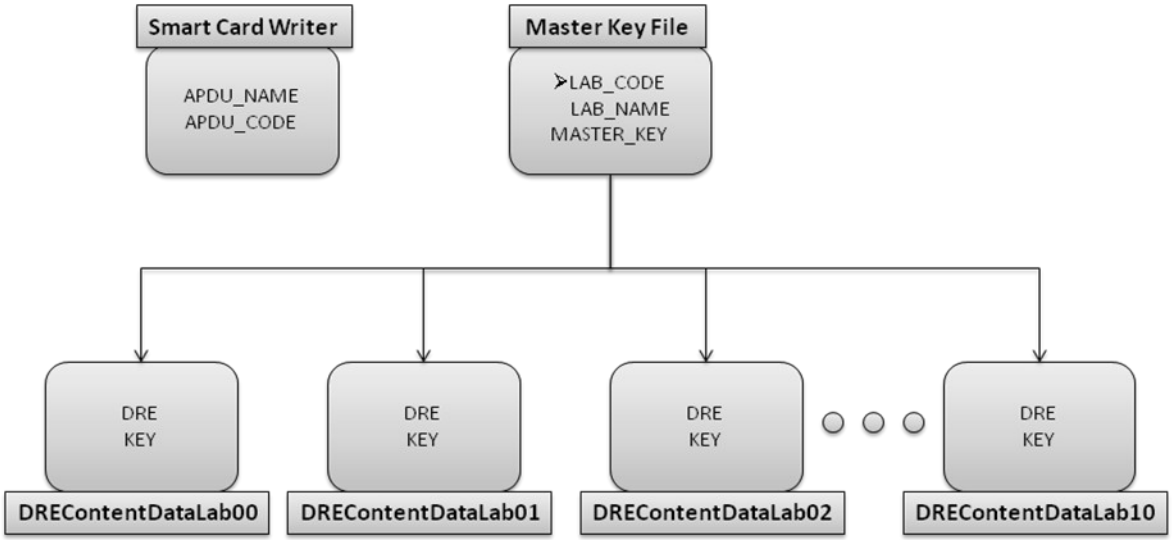
4. *SmartCardReader.config*

Este arquivo é responsável por armazenar as configurações da aplicação leitora, basicamente são os comandos *APDU* e seus respectivos nomes.

[LAB_CODE]
00
[LAB_NAME]
LIG
[*APDU*_NAME]
1 - Select *Applet*
2 - Verify Pin
3 - Request DRE
4.1 - Request RES
4.2 - Get Response
5 - Update PIN
[*APDU*_COMMAND]
00A4040010A000000018500000000000052414441
B02000000430303030
B030000009
B03001LL10RANDOM
A0C00000XX
B04000000430303030



Modelagem de arquivos na leitora.



Modelagem de arquivos no escritor

Anexo A

Biblioteca Winscard

PC/*SC* é, de fato, o padrão mais utilizado para fazer a interface entre *Smart cards*/leitores e *PC's*. Essa *API* padronizada e de alto-nível permite que o desenvolvedor se concentre no cartão propriamente dito, sem ter que se preocupar com vários aspectos e propriedades dos leitores de *Smart card*.

Para trabalhar com esses dispositivos de interface com os *Smart cards*, a *Microsoft* já oferece um rico conjunto de *API's*, contidas em uma biblioteca chamada *Winscard Dynamic Link Library*, nativa do *Windows*.

Esta parte da documentação é uma referência à algumas funções úteis dessa biblioteca e tem como objetivo introduzir o funcionamento de cada uma delas.

SCardEstabilishContext

A *função SCardEstablishContext* instancia um contexto para a aplicação dentro do gerenciador de recursos *PC* / *SC*. Este deve ser a primeira função utilizada numa aplicação.

Método:
LONG **SCardEstablishContext**(DWORD *dwScope*,
LPCVOID *pvReserved1*,
LPCVOID *pvReserved2*,
LPSCARDCONTEXT *phContext*);

Parâmetros:

dwScope	input	Explicação abaixo
pvReserved1	input	Deve ser NULL.
pvReserved2	input	Deve ser NULL.
PhContext	output	Manipulador do gerenciador de recursos PC/SC

Valores de *dwscope:*
SCARD_SCOPE_USER
SCARD_SCOPE_SYSTEM

SCardReleaseContext

A função *SCardReleaseContext* destrói a aplicação dentro do contexto do gerenciador de recursos PC / SC. Esta deve ser a última função chamada em uma aplicação *PC* / *SC.*

Método:
LONG **SCardReleaseContext**(SCARDCONTEXT *hContext*);

Parâmetro:

hContext	input	Contexto de conexão a ser liberado.

SCardListReaders

A função ***ScardListReaders*** retorna uma lista de leitores atualmente disponíveis no sistema, podendo ser filtradas opcionalmente por nomes de conjuntos de leitoras.

Método:
LONG **SCardListReaders**(SCARDCONTEXT *hContext*,
LPCSTR *mszGroups*,
LPSTR *mszReaders*,
LPDWORD *pcchReaders*);

mszReaders é um ponteiro para uma string de caracteres que é atribuída pela aplicação. Se a aplicação envia *szReaders* como *NULL* então esta função retorna o tamanho do buffer necessário para a alocação em *pcchReaders*.
Se o valor apontado pelo pcchReaders é especificado como *SCARD_AUTOALLOCATE*, então *mszReaders* é dimensionado como um ponteiro para um ponteiro, e recebe o endereço de um bloco de memória contendo a string de caracteres, atribuída pela *PC / SC API*.
Este bloco de memória deve ser desalocado pela aplicação, com a invocação de um método chamado ***SCardFreeMemory***.

Em caso de sucesso, *mszReaders* é uma string múltipla separada por caracteres nulos (‘\0’) e terminada por um duplo caractere nulo (Ex: "*Reader* A*0Reader* B\0\0")

Parâmetros:

hContext	input	Contexto de conexão para o gerenciador de recursos PC/SC
mszGroups	input	Grupo de listas para listar as leitoras
MszReaders	output	String múltipla para receber a lista de leitoras
pcchReaders	input	Tamanho máximo de *mszReaders*
pcchReaders	output	Tamanho real de *mszReaders*, incluindo tudo

SCardConnect

O método ***SCardConnect*** estabelece a conexão entre a aplicação requerente e o smartcard inserido numa leitora específica.
A primeira conexão liga e dá um *reset* no cartão. Se não há cartão na leitora, um erro é retornado.

Método:
LONG **SCardConnect**(SCARDCONTEXT *hContext*,
LPCSTR *szReader*,
DWORD *dwShareMode*,
DWORD *dwPreferredProtocols*,
LPSCARDHANDLE *phCard*,
LPDWORD *pdwActiveProtocol*);

Parâmetros:

hContext	input	Contexto de conexão com o gerenciador de recursos PC/SC

szReader	input	Nome da leitora que contém o cartão desejado.
dwShareMode	input	Conexão compartilhada ou exclusiva
dwPreferredProtocols	input	Máscara de bit especificando a lista de protocolos de cartão aceitáveis
phCard	output	Identificador para o *Smart card*
pdwActiveProtocol	output	Protocolo real selecionado pela leitora

Valores para *dwSharedMode:*
SCARD_SHARE_SHARED : A aplicação permite que outros compartilhem o *Smart card*.
SCARD_SHARE_EXCLUSIVE : A aplicação pede acesso exclusivo ao cartão.
SCARD_SHARE_DIRECT : A aplicação pede controle direto e exclusivo da leitora, mesmo sem cartão inserido.

Valores para *dwPreferredProtocols:*
SCARD_PROTOCOL_T0 : Faz uso do protocolo T=0.
SCARD_PROTOCOL_T1 : Faz uso do Protocolo T=1.
SCARD_PROTOCOL_T0| SCARD_PROTOCOL_T1 :Faz uso de ambos os protocolos.
SCARD_PROTOCOL_RAW : Para cartões de memória.
0 : Permitido somente se *dwShareMode* for usado como SCARD_SHARE_DIRECT. Nesse caso não há comunicação com o *Smart card*.

Valores para *pdwActiveProtocol:*
SCARD_PROTOCOL_T0 : Protocolo T=0 ativado.
SCARD_PROTOCOL_T1 : Protocolo T=1 ativado.
SCARD_PROTOCOL_RAW : Para cartões de memória.
SCARD_PROTOCOL_UNDEFINED : SCARD_SHARE_DIRECT foi especificado. Há a possibilidade de não existir cartão na leitora.

SCardDisconnect

O método ***SCardDisconnect*** termina uma conexão feita previamente entre o aplicativo requerente e um *Smart card*.

Método:
LONG **SCardDisconnect**(SCARDHANDLE *hCard*,
DWORD *dwDisposition*);

Parâmetros:

hCard	in	Contexto do cartão (retornado por **SCardConnect**)
dwDisposition	in	Ação a ser tomada no cartão.

Valores para *dwDisposition:*
SCARD_LEAVE_CARD : Não faz nada (Cartão permanece ativo).
SCARD_RESET_CARD : Reinicia o cartão (Reset brando, “Warm Reset”)
SCARD_UNPOWER_CARD : Desativa o cartão.
SCARD_EJECT_CARD : Ejeta fisicamente o cartão (Caso seja possível pela leitora).

SCardTransmit

A função *SCardTransmit* envia o comando *APDU* ao *Smart card*, que retorna uma resposta.

Método:
LONG **SCardTransmit**(SCARDHANDLE *hCard*,
LPCSCARD_IO_REQUEST *pioSendPci*,
LPC*BYTE pbSendBuffer*,
DWORD *cbSendLength*,
LPSCARD_IO_REQUEST *pioRecvPci*,
LP*BYTE pbRecvBuffer*,
LPDWORD *pcbRecvLength*);

Parâmetros:

hCard	input	Contexto do cartão (retornado por **SCardConnect**)
pioSendPci	in/out	Explicação abaixo
pbSendBuffer	in	Comando *APDU* a ser enviado para o cartão
cbSendLength	in	Tamanho do comando *APDU*
pioRecvPci	in/out	Explicação abaixo
pbRecvBuffer	out	Resposta vinda do cartão
pcbRecvLength	in	Tamanho máximo de *pbRecvBuffer*
pcbRecvLength	out	Tamanho real da resposta em *pbRecvBuffer*

Valores para *pioSendPci* e *pioRecvPci:*
Ambos os parâmetros são ponteiros para a estrutura, dizendo para o driver qual protocolo deve
ser utilizado para enviar o comando *APDU* e receber a resposta do cartão.
Na maioria dos casos, pode-se usar um dos valores globais pré-definidos para ambos:

SCARD_PCI_T0 se protocolo corrente for T=0.
SCARD_PCI_T1 se protocolo corrente for T=1.

Como alternativa, pode-se definir ambos os valores como *NULL*, e deixar a *API PC/SC* verificar que protocolo está sendo utilizado.

Além das funções aqui explicadas existe ainda um grupo seleto de outras funções, mas que não foram abordadas por fugirem ao escopo desse projeto.