Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada
Programa de Pós-Graduação em Sistemas e Computação

JCML - Java Card Modeling Language: Definição e Implementação

Plácido Antônio de Souza Neto

Natal / RN
Novembro de 2007

Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada
Programa de Pós-Graduação em Sistemas e Computação

JCML - Java Card Modeling Language: Definição e Implementação

Dissertação de Mestrado submetido ao Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte como parte dos requisitos para a obtenção do grau de Mestre em Sistemas e Computação (MSc.).

Orientadora:
Profa. Dra. Anamaria Martins Moreira

Plácido Antônio de Souza Neto

Natal / RN, Novembro de 2007

JCML - JAVA CARD MODELING LANGUAGE: DEFINIÇÃO E IMPLEMENTAÇÃO

Plácido Antônio de Souza Neto

Esta dissertação de mestrado foi avaliada e considerada APROVADA pelo Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte.

_____

Profa. Dra. Anamaria Martins Moreira
Orientador

_____

Profa. Dra. Thaís Vasconcelos Batista
Coordenador do Programa

Banca Examinadora:

_____

Prof. Dra. Anamaria Martins Moreira
Universidade Federal do Rio Grande do Norte

_____

Prof. Dr. Martin Alejandro Musicante
Universidade Federal do Rio Grande do Norte

_____

Prof. Dr. Umberto Souza da Costa
Universidade Federal do Rio Grande do Norte

_____

Prof. Dr. Paulo Henrique Monteiro Borba
Universidade Federal de Pernambuco

Catalogação da Publicação na Fonte. UFRN / SISBI / Biblioteca Setorial Especializada do Centro de Ciências Exatas e da Terra - CCET.

SOUZA NETO, Plácido Antônio de

JCML - Java Card Modeling Language: Definição e Implementação / Plácido Antônio de Souza Neto. - Natal / RN, 2007.

135p. : il.

Orientador: Profa. Dra. Anamaria Martins Moreira

Dissertação (Mestrado) - Universidade Federal do Rio Grande do Norte. Centro de Ciências Exatas e da Terra. Departamento de Informática e Matemática Aplicada. Programa de Pós-Graduação em Sistemas e Computação.

1. Métodos formais - Dissertação. 2. Java Card - Dissertação. 3. Verificação Runtime - Dissertação. 4. Compilador - Dissertação. I. Moreira, Anamaria Martins. II. Título.

RN/UF/BSE-CCET CDU: 004.41

Dedico este trabalho ao Senhor da minha vida, Jesus Cristo, que é a verdadeira fonte de conhecimento e sabedoria, e que, pela sua morte de cruz, hoje tenho vida. Por me resgatar para a vida que Ele tem preparado para os que o amam e por me proporcionar momentos felizes ao Seu lado.

"Deus amou o mundo de tal maneira que deu seu filho unigênito para que todo aquele que nEle crê, não pereça, mas tenha vida eterna. (João 3:16)"

AGRADECIMENTOS

Agradeço a Deus, o Criador de todas as coisas e único dono da verdadeira sabedoria. Sem ele, confesso, não teria capacidade e forças para alcançar meus sonhos.

Agradeço a minha esposa, Ana Flávia L. M. Souza, pelo companheirismo, amor e preocupação. Por estar ao meu lado em todos os momentos. Amo você, minha linda!!!

Agradeço a todos os familiares que se fizeram presentes na minha vida, principalmente pela compreensão quando me fiz ausente, em especial a minha mãe, Ana Sueli de Souza, pelas constântes orações.

Agradeço também a minha orientadora, Anamaria Martins Moreira, pela dedicação na orientação, pelos conselhos e pelo aprendizado. Professora que se preocupa com seus alunos, fazendo com que o conhecimento seja verdadeiramente produzido. Sua ajuda foi de suma importância na conclusão deste trabalho. Muito obrigado.

Ao professor Umberto, pelas dicas na modelagem, desenvolvimento do compilador e otimização do código. O Sr. teve participação decisiva no desenvolvido do JCMLc.

Agradeço as colegas do PPgSC, pela amizade, força e até mesmo pela paciência em me transmitir alguns conteúdos importantes no decorrer desses 2 anos. Principalmente ao pessoal do *ConSiste*: Bruno, Cláudia, Itamir, Katia e Thiago.

Gostaria muito de agradecer aos professores do CEFET-RN, João Maria Filgueira, Cláudia Ribeiro, Francisco de Assis e Robinson Luís pelo exemplo e por se preocuparem com o ensino. E a instituição CEFET-RN, na qual trabalho, pela oportunidade de aplicar o aprendizado adiquirido nesta Pós-graduação.

Obrigado a todos!

Resumo

Métodos formais poderiam ser usados para especificar e verificar software *on-card* em aplicações Java Card. O estilo de programação para *smart cards* requer verificação em tempo de execução para condições de entrada em todos os métodos Java Card, onde o objetivo principal é preservar os dados do cartão. *Projeto por Contrato*, em particular, a linguagem JML, é uma opção para este tipo de desenvolvimento e verificação, pelo fato da verificação em tempo de execução ser parte da implementação pela JML. Contudo, JML e suas respectivas ferramentas para verificação em tempo de execução não foram projetadas com o foco nas limitações Java Card, sendo, dessa forma, não compatíveis com Java Card. Nesta dissertação, analisamos o quanto esta situação é realmente intrínseca às limitações Java Card e, se é possível re-definir a JML e suas ferramentas. Propomos requisitos para uma nova linguagem, a qual é compatível com Java Card e apresentamos como o compilador desta linguagem pode ser construído. JCML retira da JML aspectos não definidos em Java Card, como por exemplo, concorrência e tipos não suportados. Isto pode não ser o bastante, contudo, sem o esforço em otimização de código de verificação gerado pelo compilador, não é possível gerar código de verificação para rodar no cartão. O compilador JCML, apesar de ser bem mais restrito em relação ao compilador JML, está habilitado a gerar código de verificação compatível com Java Card, para algumas especificações *lightweight*. Como conclusão, apresentamos uma variante da JML compatível com Java Card, JCML (Java Card Modeling Language), com uma versão de seu compilador.

Área de Concentração: Engenharia de Software
Palavras-chave: Métodos Formais, Java Card, JML, JCML, Verificação *Runtime*, Compilador

Abstract

Formal methods should be used to specify and verify on-card software in Java Card applications. Furthermore, Java Card programming style requires runtime verification of all input conditions for all on-card methods, where the main goal is to preserve the data in the card. *Design by contract*, and in particular, the JML language, are an option for this kind of development and verification, as runtime verification is part of the *Design by contract* method implemented by JML. However, JML and its currently available tools for runtime verification were not designed with Java Card limitations in mind and are not Java Card compliant. In this thesis, we analyze how much of this situation is really intrinsic of Java Card limitations and how much is just a matter of a complete re-design of JML and its tools. We propose the requirements for a new language which is Java Card compliant and indicate the lines on which a compiler for this language should be built. JCML strips from JML non-Java Card aspects such as concurrency and unsupported types. This would not be enough, however, without a great effort in optimization of the verification code generated by its compiler, as this verification code must run on the card. The JCML compiler, although being much more restricted than the one for JML, is able to generate Java Card compliant verification code for some lightweight specifications. As conclusion, we present a Java Card compliant variant of JML, JCML (Java Card Modeling Language), with a preliminary version of its compiler.

Area of Concentration: Software Engineering
Key words: Formal Methods, Java Card, JML, JCML, Runtime Verification, Compiler

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Este trabalho tem como foco principal as tecnologias Java Card, a linguagem de especificação JML e o processo de verificação de especificações em tempo de execução. Especificação formal é um conceito baseado em fundamentos matemáticos para descrição de sistemas e suas propriedades. A verificação em tempo de execução tem como papel analisar se o sistema satisfaz ou não as funcionalidades previstas na especificação durante a sua execução. Dessa forma, o uso de especificação formal no desenvolvimento de sistemas, principalmente sistemas com restrição de recursos, como os *smart cards*, proporciona maior confiabilidade no que realmente está sendo desenvolvido.

Um *smart card* é capaz de processar dados, armazenar informações importantes, além de possuir elementos de segurança. A tecnologia Java Card implementa as características dos smart cards usando a linguagem Java. Isto se torna uma vantagem quando é levada em consideração a popularidade da linguagem Java, tornando o desenvolvimento mais fácil. Contudo, o desenvolvimento de aplicações Java Card ainda requer uma atenção especial, pelo fato de utilizar apenas um subconjunto da linguagem Java. Como consequência disto, existe uma máquina virtual Java específica para esta tecnologia, a JCVM (Java Card Virtual Machine).

A linguagem JML (Java Modeling Language) é utilizada para especificar classes e interfaces Java. O objetivo principal da JML é prover uma linguagem de especificações fácil de usar a programadores Java. O problema principal na aplicação de especificação JML no contexto de aplicações smart card é o ambiente restrito. Com processamento e memória limitados, a inserção de anotações JML a serem verificadas em tempo de execução normalmente ultrapassa os limites de processamento e memória permitidos pela

tecnologia Java para smart cards, Java Card. Além das restrições do cartão, a linguagem JML não foi projetada com o foco em Java Card. Dessa forma, nem todas as construções JML são compatíveis com Java Card.

Nas Seções 1.1, 1.2 e 1.3 serão apresentadas a motivação, objetivos e estrutura desta dissertação, respectivamente.

1.1 Motivação

A aplicação de métodos formais proporciona um desenvolvimento de software de forma precisa e correta de acordo com a sua especificação. A linguagem de especificação JML tem sido utilizada com sucesso no desenvolvimento de aplicações Java. Devido ao fato deste formalismo mostrar-se adequado para o desenvolvimento de aplicações Java, este trabalho tem como motivação o desenvolvimento de aplicações Java Card utilizado métodos formais.

Como a tecnologia Java Card utiliza apenas um subconjunto da linguagem Java, a verificação de código JML em tempo de execução torna-se inviável devido a linguagen JML utilizar recursos da linguagem Java não suportados em Java Card. Com isso, também como motivação deste trabalho, existe a necessidade de proporcionar maior confiabilidade às aplicações Java Card, a partir da aplicação de um formalismo.

1.2 Objetivos

O presente trabalho tem por objetivo: estudar a viabilidade de se oferecer mecanismos de verificação em tempo de execução em Java Card; definir e implementar uma linguagem de especificação, derivada de JML, para Java Card, com os mecanismos de verificação possíveis; propor um compilador que traduz especificação em código Java Card.

Para cumprir os objetivos desejados foi feito um estudo detalhado das tecnologias Java Card e JML, além de definição de um subconjunto JML, na construção da linguagem JCML (Java Card Modeling Language), junto com seu compilador, que é o produto proposto para esta dissertação.

1.3 Organização do Trabalho

No capítulo 2 são fornecidos os conceitos e teoria referentes às tecnologias *smart card* e Java Card. Os smart cards são apresentados segundo sua especificação e estrutura e os Java Card são apresentados de acordo com sua plataforma e arquitetura de desenvolvimento, além de restrições de memória e estrutura de desenvolvimento.

No capítulo 3, a JML será detalhada tem termos de estrutura da linguagem, compilação, ferramentas e tipos de especificação. O presente trabalho também apresenta, no Capítulo 4 uma análise da aplicação de construções JML em dispositivos *smart card*. Como resultado desta análise, uma nova linguagem, baseada na JML, é apresentada no capítulo 5. A JCML (Java Card Modeling Language) é uma linguagem que traduz especificação em código Java Card, com objetivo de verificar propriedades em tempo de execução. Neste capítulo, é apresentada a gramática da linguagem e uma versão de seu compilador, finalizando, com um comparativo do código gerado pelo compilador da linguagem JML e o compilador JCML. Por fim são apresentados os trabalhos relacionados e as conclusões do presente trabalho com uma descrição dos trabalhos futuros, nos Capítulos 6 e 7 respectivamente.

Capítulo 2

Smart Cards

Um *smart card* (cartão inteligente) é um computador [19]. Apesar de não incluir teclado ou monitor, um *smart card* tem os elementos que caracterizam um computador: processamento, memória e barramento de comunicação para entrada e saída de dados. Um *smart card* consiste fisicamente em um pequeno *chip*, envolvido em um revestimento plástico, que o deixa com a aparência de um cartão de crédito comum. Contudo, tem uma pequena placa de metal em sua parte inferior responsável pelo contato do cartão com o dispositivo de leitura. Um *smart card* é um dispositivo portátil capaz de executar pequenas aplicações e armazenar informações de forma segura.

Os *smart cards* são conhecidos como cartões de circuitos integrados, que contêm um microprocessador ou um *chip* de memória. Segundo Chen [8], os cartões com apenas *chips* de memória podem não ser considerados inteligentes, já que não realizam processamento.

Diferentemente dos cartões de crédito convencionais, a informação contida em um *smart card* não é facilmente copiada [8]. O poder de processamento do cartão inteligente, o mecanismo de criptografia de informações durante a comunicação com o cartão e assinatura digital, são utilizados para garantir a segurança de aplicações *smart cards* e dos dados armazenados.

Os *smart cards* (ver Figura 2.1) são padronizados para a indústria pela norma ISO (*Internation Organization Standardization*) 7816, assim como suas sete partes, que definem características físicas, dimensões e localização de contatos, sinais eletrônicos e protocolos de transmissões.

Figura 2.1: Smart Card.

A tecnologia Java Card [27] é uma adaptação da plataforma Java para ser utilizada em *smart cards* e outros dispositivos cujos ambientes têm memória limitada e restrições de processamento. Esta tecnologia possui sua propria máquina virtual, API (*Application Programming Interface*) [27], e especificação de tempo de execução (*Runtime*) [38].

Java Card possibilita uma maior produtividade no desenvolvimento de aplicações para cartões, uma vez que abstrai os detalhes de baixo nível do sistema *smart card*. Possui ainda, o suporte de ferramentas fornecidas por empresas como a SUN Microsystems (empresa desenvolvedora da tecnologia Java Card). Essas ferramentas são compostas por IDEs (*Integrated Development Environment*), plugins, simuladores, etc. que proporcionam um processo mais rápido de construção, teste e instalação desses tipos aplicações, fazendo com que exista um menor custo de produção.

2.1 Java Card

A tecnologia Java Card foi desenvolvida a partir de um subconjunto da linguagem Java. Dessa forma, a máquina virtual Java Card suporta apenas as propriedades que são requeridas pelo subconjunto da linguagem.

O desafio da tecnologia Java para *smart cards* é fazer com que a estrutura da linguagem Java seja utilizada para desenvolver sistemas que rodem em cartões. Java Card permite que os *smart cards* e outros dispositivos com memória limitada rodem pequenas aplicações chamadas *applets*. Java Card oferece:

- independência de plataforma - os sistemas Java para cartões podem rodar em qualquer sistema operacional;

- habilidade para guardar e atualizar múltiplas aplicações - em um único cartão é possível instalar várias aplicações *smart card*;

- compatibilidade com o padrão *smart card* existente - a plataforma Java Card implenta as características para desenvolvimento de aplicações definidas pela norma ISO 7816.

A arquitetura básica de uma Java Card consiste em: *Applets*, API Java Card, Máquina Virtual Java Card (JCVM), Java Card Runtime Environment (JCRE), e o sistema operacional nativo do cartão.

A máquina virtual, que é executada sobre o sistema operacional do cartão, pode ser vista como uma abstração. Sobre a máquina virtual encontra-se o Java Card Framework, que é um conjunto de classes necessárias para o funcionamento do JCRE e a API Java Card, como também as classes ou extensões proprietárias do fabricante do cartão. No topo, encontram-se os applets que utilizam todos os demais componentes.

Os componentes da tecnologia Java Card são:

- *JCVM (Java Card Virtual Machine)* [39] - A *JCVM* é uma Máquina Virtual dedicada, de uma única thread. A máquina virtual Java Card, versão 2.2.1 [1], não dá suporte às seguintes características de Java: carregamento dinâmico de classes; String e threads; variáveis do tipo double, float e char; arrays multidimensionais; classe *java.lang.System*; coleta de lixo. Isso ocorre pelo fato da JCVM definir um subconjunto da linguagem Java e ser adaptada às aplicações smart card.

  A principal diferença entre a JCVM e a JVM (*Java Virtual Machine*) é que a JCVM é implementada como duas partes separadas, o conversor e o interpretador. O conversor Java Card (Figura 2.2) é executado em qualquer estação de trabalho. O conversor faz parte da estrutura *off-card* (é executada fora do cartão). Após a criação do arquivo CAP (*Converted Applet* - Seção 2.1.1), a verificação das propriedades do *applet* é feita pelo interpretador *on-card* (executado dentro do cartão) quando o *smart card* estiver em contato com a aplicação. O interpretador reconhece o arquivo CAP gerado pelo conversor.

[1] A versão 2.2.1 da plataforma Java Card [37] é a versão utilizada como referência durante todo o trabalho.

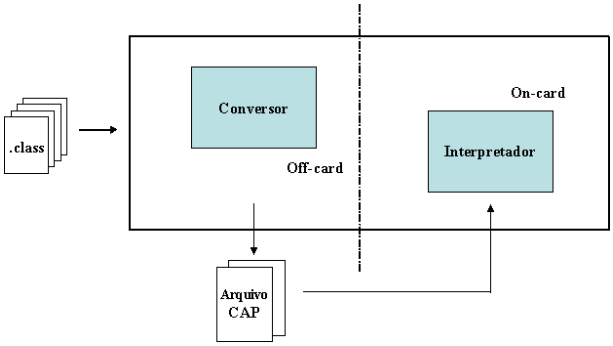

Figura 2.2: JCVM - Java Card Virtual Machine.

- *JCRE (Java Card Runtime Environment)* [38] - O *JCRE* consiste nos componentes do sistema Java Card que rodam dentro de um *smart card*. A propriedade mais significante do *JCRE* é que este provê uma separação clara entre o *smart card* e as aplicações desenvolvidas. A *JCRE* encapsula os detalhes da complexidade de um *smart card*, facilitando o desenvolvimento de aplicações. O JCRE define como se comporta o ambiente de execução, incluindo gerenciamento de memória, de aplicações, segurança e outras características da execução [38].

- *JC-API (API Java Card)* [37] - A API Java Card [27] descreve as interfaces de desenvolvimento para aplicação da tecnologia Java Card. A API contém definições de classe que são necessárias para o suporte da *JCVM e JCRE*.

A JCVM utiliza dois tipos de arquivos, independente de plataforma, que são aplicados no desenvolvimento de um software Java Card, são eles: Arquivos CAP (*Converted Applet*) e Arquivos *Export*. Os CAPs contém uma representação binária das classes executáveis Java (arquivos *.class*) e são como arquivos JAR que contém um conjunto de componentes. Cada arquivo contém informações de classes, bytecodes executáveis e informações de verificação, como tipos de dados e visibilidade. Após a construção dos arquivos CAP (*off-card*), estes são inseridos no cartão para serem executados (*on-card*) como mostra a Figura 2.2.

O arquivo *Export* não é executado pelo interpretador. Este arquivo é utilizado pelo conversor para a verificação da estrutura Java Card [8], contendo informações de APIs públicas de um pacote de classes. Com isso é definido o tipo de acesso e nomes das classes. Também é definido neste arquivo, o escopo de acesso e assinatura de métodos e atributos de cada classe. O arquivo *Export* não contém implementação, com isso não

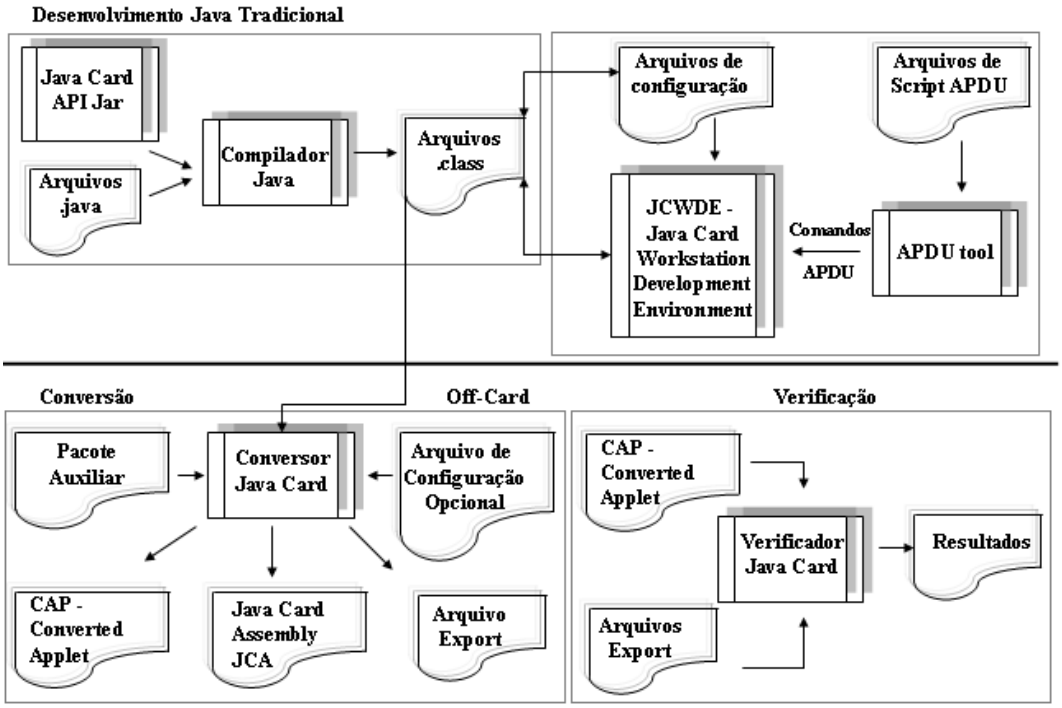

Figura 2.3: Desenvolvimento Java Card - *off-card*.

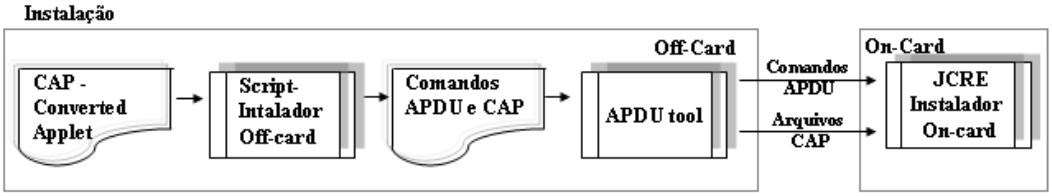

Figura 2.4: Desenvolvimento Java Card - *on-card*.

possui *bytecodes*. Ele pode ser distribuído pelos desenvolvedores Java Card sem revelar detalhes de implementação, podendo ser comparado com uma interface de comunicação Java, onde é definida a estrutura da classe.

As Figuras 2.3 e 2.4 apresentam os passos para desenvolvimento dos componentes necessários para o desenvolvimento de um *applet* Java Card [28]. A Figura 2.3 mostra a sequência de passos *off-card*, onde os arquivos CAP e Export são definidos. A Figura 2.4 apresenta a estrutura *on-card*, no qual através dos comandos APDU (*Application Protocol Data Unit*) a aplicação Java Card é inserida no cartão.

Os passos para o desenvolvimento de uma aplicação Java Card são:

- Escrever o *Applet* Java Card - Como resultado têm-se arquivos *.java*;
- Compilar o *Applet* - Como resultado têm-se arquivos *.class*;

- Testar o *Applet* Java Card com o simulador JCWDE - *Java Card Workstation Development Environment* (opcional);
- Converter as classes geradas em um arquivo CAP, usando a ferramenta de conversão. Opcionalmente, os arquivos *Export* são adicionados para prover informações sobre os pacotes e classes;
- Verificar o arquivo CAP;
- Instalar o arquivo CAP no cartão (ver Figura 2.4) e
- Testes dos *applets* no cartão.

2.1.1 *Applets* Java Card

Uma aplicação *smart card* é composta por uma aplicação *host* cliente, que pode residir em um computador, e pela aplicação contida no cartão. Em Java Card, esta aplicação é denominada *applet*. Os *applets* Java Card são classes Java que estendem a classe *javacard.framework.Applet*, seguindo as especificações da tecnologia Java Card. A comunicação entre o *applet* e uma aplicação *host* ocorre através de mensagens APDU (*Application Protocol Data Unit*). Essas mensagens são pacotes de dados lógicos trocados entre o dispositivo de leitura, denominado de CAD (*Card Aceptance Device*), e o cartão. O CAD fornece energia ao cartão e serve como o meio de comunicação entre a aplicação *host* e o *applet* que está no cartão.

Todo *applet* Java Card deve poder executar os métodos *install()* e *process()*; o JCRE [38] chama o método *install()* para instalar o *applet*, e o método *process()* é executado cada vez que é enviado um comando da APDU para o *applet*.

O *applet* tem início ao ser carregado na memória do cartão. Ao ser carregado, o JCRE invoca o método *install()* e o *applet* registra-se no JCRE com o método *register()*. Depois deste processo (instalação e registro) o *applet* está em estado "unselected" e em condições de ser selecionado para execução . O *applet* normalmente existe durante o resto da vida do cartão. A Figura 2.5 apresenta os métodos do ciclo de vida[2] de um *applet*.

[2]Neste contexto, ciclo de vida significa o ciclo de desenvolvimento de um *applet* Java Card

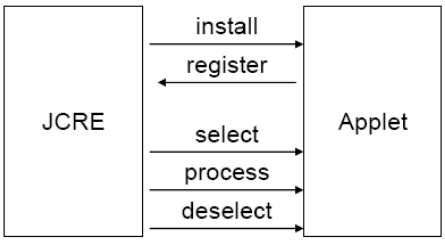

Figura 2.5: Ciclo de Vida de um *Applet*.

```
01 abstract public class Applet  {
02    public static void install
03       ( byte[] bArray, short bOffset, byte bLength )
04            throws ISOException{}
05    public abstract void process(APDU apdu)
06      throws ISOException{}
07    public boolean select(){}
08    public void deselect(){}
09    public Shareable getShareableInterfaceObject
10      (AID clientAID, byte parameter){}
11    protected final void register()
12       throws SystemException{}
13    protected final void register
14      (byte[] bArray, short bOffset, byte bLength )
15          throws SystemException{}
16    protected final boolean selectingApplet(){}
17 }
```

Figura 2.6: Classe *Applet* [37].

A seleção de um *applet* instalado no cartão ocorre quando a aplicação *host* requisita ao JCRE para que este selecione o mesmo no cartão, através de um comando APDU denominado *select*. O JCRE procura um AID (*Application Identifier*) em suas tabelas, correspondente ao *applet* requisitado.

O método *select* informa ao *applet* que o mesmo está sendo selecionado. Após esta operação, o *applet* encontra-se pronto para receber as requisições aos comandos APDUs da aplicação *host*. Isto ocorre quando o método *process* é chamado.

É importante observar que o método *select* possibilita que o desenvolvedor do *applet* realize alguma operação de inicialização necessária para que o mesmo torne-se apto a processar os comandos APDU da aplicação *host*.

As classes e os *applets* Java Card são empacotados em um único arquivo denominado CAP, semelhante a um arquivo Jar (*Java Archive*). Esse arquivo é instalado no cartão.

2.1.2 Restrições Java Card

Devido à pouca capacidade de recursos de hardware, se fez necessária a definição de uma nova plataforma baseada em Java, a qual viabilizasse o desenvolvimento e utilização da tecnologia em cartões inteligentes. As restrições para o Java Card previnem a máquina virtual de suportar todas as funcionalidades de Java. Algumas características são preservadas, contudo a máquina virtual para Java Card limita uma quantidade de operações possíveis, quantidade esta menor que as aplicadas em Java.

As restrições listadas estão relacionadas à perspectiva do programador Java Card.

Memória: Os *smart cards* tem 3 tipos de memória: ROM, EEPROM e RAM.

ROM é uma memória para leitura apenas. Dados e programas são armazenados na ROM durante a fabricação do *smart card*. Tanto a EEPROM quanto a memória RAM podem ser lidas e escritas, contudo, essas memórias tem características diferentes. O conteúdo da memória RAM é preservado apenas quando o cartão está alimentado. Na memória EEPROM, os dados são preservados mesmo quando o cartão não está em contato com o leitor, e não existe energia para deixar o cartão ativo. A memória RAM é mais rápida que a memória EEPROM. Os *smart cards* normalmente oferecem 1 kbytes de RAM, 16 kbytes EEPROM e 24 kbytes de ROM [8]. O processamento do cartão pode variar entre 5 e 40MHz dependendo do tipo de tecnologia do cartão. Com a evolução da tecnologia para desenvolvimento de cartões, existem cartões com uma capacidade de memória um pouco maior que as definidas em [8], por exemplo em [36] é definido um cartão com memória RAM com 12 kbytes, ROM entre 374 e 394 kbytes e 144 kbytes de memória EEPROM. A capacidade de processamento em [36] é de 33 MHz.

Apesar do constante progresso no que diz respeito à capacidade de memória, capacidade de armazenamento e processamento de dados ainda existe uma grande limitação no desenvolvimento de sistemas Java Card.

Threads: A Máquina Virtual Java Card não suporta múltiplas *threads*. Programas Java Card não podem usar a classe *Thread* definida na API Java ou qualquer palavra chave relacionada com a execução de *threads* Java. Os *applets* Java Card são executados um de

```
01 import javacard.framework.ISOException;
02 public class AcessoNegadoException extends ISOException{
03    public static final short SW_ACESSO_NEGADO = (short) 0x6322 ;
04    public AcessoNegadoException(short arg0) {
05         super(arg0);
06    }
07 }
```

Figura 2.7: Classe de Tratamento de Exceção.

cada vez. Os programas Java Card previamente instalados no cartão podem ser chamados para serem executados através de comandos específicos, citados na seção 2.1.1.

Exceções em Java Card: Devido às limitações de memória em Java Card, as classes que tratam exceções na API Java Card possuem um método estático chamado *throwIt*. Este método possibilita lançar uma exceção sem instanciar um objeto, por meio de uma instância da classe. Pode-se fazer *NomeDaClasseExcecao.throwIt(MENSAGEM)* ao invés de usar as palavras reservadas *throw new Exception()* para lançar uma nova exceção. Evitar instanciar objetos que tratam exceções em Java Card otimiza o uso de memória.

Throwable é a classe pai para todas as classes de exceção em Java Card. A classe *Exception* herda diretamente de *Throwable*. Em Java Card, todas as exceções de runtime descendem de *CardRuntimeException*, que é subclasse de *RuntimeException* que é subclasse de *Exception*. Outro tipo de exceção é a *CardException*, na qual define erros específicos de aplicação que devem ser tratados ou especificados em um método.

A classe mais frequentemente utilizada para tratar exceções Java Card é a classe *ISOException*, do pacote *javacard.framework.**. Uma *ISOException* é uma exceção em tempo de execução usada para relatar que um erro ocorreu no processamento de um comando APDU. Existe uma outra classe do pacote *javacard.framework.** usada para especificar erros de programação que podem ocorrer em um *applet* Java Card, esta classe é chamada de *UserException*.

Na Figura 2.7 é apresentada uma classe de exceção. A classe tem uma constante que é lançada ao usuário. É importante perceber que a classe herda as características de *ISOException*, podendo fazer uso do método estático *throwIt*.

Classes e Objetos: Todas as classes da API Java Card descendem da classe *java.lang.Object*, exatamente igual à linguagem Java. Contudo nem todos os métodos estão disponíveis na API Java Card, como por exemplo o método *toString()* de *Object*. A classe Object existe para prover a mesma estrutura hieráquica definida em Java. Apenas as classes *Object e Throwable* são suportadas em Java Card.

Tipos: Seria interessante se programas para *smart card* podessem ser escritos usando toda a estrutura da linguagem Java, principalmente os tipos primitivos. A especificação da máquina virtual Java Card implementa apenas um subconjunto dos tipos primitivos definidos em Java. Tipos como *float, string, double e char* não fazem parte da gramática Java Card.

O tipo *int* não é totalmente suportado. Ele pode ser usado dependendo da versão da máquina virtual. O tipo inteiro é uma característica opcional. Apenas os subtipos de inteiro, *byte e short* são garantidos. Propriedades como referência nula (*null*) e o tipo *boolean* também são permitidos em Java Card.

Criar objetos em Java Card também é possível, tanto referências simples quanto arrays. Arrays podem conter os tipos suportados por Java Card ou objetos.

Criação Dinâmica de Objetos: A plataforma Java Card suporta criação dinâmica de objetos, sejam instâncias de classe ou de arrays. Isso é feito usando o operador *new*. É normal programadores Java utilizarem e instanciarem vários tipos de objetos em uma aplicação. Ao se utilizar Java Card, o programador tem que se preocupar com a quantidade de objetos instanciados, devido ao pouco recurso de memória RAM do cartão. O uso de membros (variável, construtor ou métodos) estáticos em Java Card resolve problemas de espaço de memória de cada objeto instanciado, contudo, utilizar apenas membros estáticos não é uma boa prática de programação orientada a objetos.

A máquina virtual Java Card utiliza, de forma opcional, o conceito de *coleta de lixo* de Java, na qual objetos que perdem a referência são desalocados da memória automaticamente.

2.2 Desenvolvendo uma Aplicação Java Card

No desenvolvimento de uma aplicação Java Card, o programador tem que se preocupar com aspectos de comunicação *smart card* por meio do protocolo APDU ou RMI (*Remote Method Invocation*). O desenvolvimento de aplicações utilizando a plataforma Java Card pode ser dividido em etapas. Essas etapas de desenvolvimento estão descritas na Seção 2.1.

Uma aplicação RMI é utilizada tanto no cliente como no servidor. A aplicação servidora cria e torna acessíveis objetos que serão invocados remotamente por meio de uma interface. A aplicação cliente obtém uma referência ao objeto, e então invoca os seus métodos. No caso de Java Card RMI, o applet Java Card atua como servidor e a aplicação host como cliente. Como o objetivo principal deste trabalho consiste aplicações utilizando o protocolo APDU, a estrutura RMI para Java Card não será detalhada.

Apesar da possibilidade de se implementar componentes *smart card* através de APDU e RMI, um problema enfrentado por desenvolvedores Java Card é a ampla diversidade de fabricantes de cartões, cada um com suas interfaces próprias, o que dificulta a criação de aplicações que possam ser utilizadas entre *hardware* de diferentes fabricantes sem necessidade de manutenção.

O exemplo a ser utilizado para mostrar detalhes do desenvolvimento Java Card será apresentado na seção a seguir.

2.2.1 *Applet* para Controle de Usuário

A aplicação de controle de usuário tem como objetivo apresentar como os membros de um applet são implementados. A aplicação armazena informações de um usuário, são elas: tipo, podendo ser um estudante ou um professor; matrícula; locais de acesso autorizados; e controle de crédito que podem ser utilizados pelo usuário do cartão.

A aplicação deve armazenar unidades de crédito. Deve ser possível a adição de novos créditos, consulta à quantidade de créditos armazenados e o débito de valores disponíveis no cartão. É possível armazenar locais de acessos, adicionar novos locais e verificar se é

```
01 import javacard.framework.APDU;
02 import javacard.framework.Util;
03 public class UsuarioApplet extends Applet {
04
05   public static final byte CLA_SMARTESCOLA = (byte) 0x80;
06   public static final byte INS_VERIFICAR_SENHA = (byte) 0x00;
07   public static final byte INS_SET_MATRICULA = (byte) 0x11;
08   public static final byte INS_SET_TIPO = (byte) 0x12;
09   public static final byte INS_GET_MATRICULA = (byte) 0x21;
10   public static final byte INS_GET_TIPO = (byte) 0x22;
11   public static final byte INS_ADICIONAR_LOCAL = (byte) 0x13;
12   public static final byte INS_VERIFICAR_LOCAL = (byte) 0x23;
13   public static final byte INS_REMOVER_LOCAL = (byte) 0x33;
14   public static final byte INS_ADICIONAR_CREDITOS = (byte) 0x14;
15   public static final byte INS_VERIFICAR_CREDITOS = (byte) 0x24;
16   public static final byte INS_REMOVER_CREDITOS = (byte) 0x34;
17
18   public static final short SW_SENHA_INVALIDA = (short) 0x6301;
19   public static final short SW_AUTENTICACAO_INVALIDA = (short) 0x6302;
20
21   public static final byte MAXIMO_TENTATIVAS_PIN = 3;
22   public static final byte TAMANHO_MAXIMO_PIN = 10;
23   private OwnerPIN pin;
24   private Usuario u;
25
26    //Métodos UsuarioApplet;
27 }
```

Figura 2.8: Constantes UsuarioApplet.

permitido o usuário ter acesso ao local desejado. O usuário também pode ter o seu tipo modificado.

Deve existir um comando APDU para cada funcionalidade fornecida pelo applet. Na aplicação de controle de usuário temos os seguintes comandos APDU: verificar senha; controlar matrícula do usuário; controlar tipo; adicionar local; verificar local; remover local; adicionar crédito; verificar crédito e remover crédito. Cada constante é do tipo *byte*.

O código da Figura 2.8 define a estrutura da classe e dos comandos APDU através de constantes. Cada constante tem um valor que representará o comando correspondente e começa com a definição de instrução *INS*, por exemplo, *INS_VERIFICAR_SENHA*. O comando para verificar senha é representado pelo valor *0x00*. Os dados do applet, como os códigos identificadores das operações e das exceções, são usualmente definidos como constantes.

As constantes *SW_SENHA_INVALIDA* e *SW_AUTENTICACAO_INVALIDA* repre-

```
01 public static void install(byte[] bArray, short bOffset,
02             byte bLength) {
03         new UsuarioApplet(bArray, bOffset, bLength);
04 }
```

Figura 2.9: Método *install* de UsuarioApplet.

```
01 private UsuarioApplet(byte[] bArray, short bOffset, byte bLength) {
02         super();
03         u = new Usuario();
04         pin = new OwnerPIN(MAXIMO_TENTATIVAS_PIN, TAMANHO_MAXIMO_PIN);
05         pin.update(bArray, bOffset, bLength);
06         register();
07 }
```

Figura 2.10: Construtor UsuarioApplet.

sentam valores que são retornados ao usuário caso alguma exceção ocorra, por exemplo, problemas com a senha inválida ou quantidades de tentativas na verificaçao de senha (constante *MAXIMO_TENTATIVAS_PIN*). Outros tipos de constantes também podem aparecer na estrutura inicial de um applet, com o objetivo de auxiliar a implementação.

O construtor do applet deve ser privado, uma vez que a criação de um do applet deve ser feita apenas pelo JCRE através da chamada do método install, que chama o construtor do applet (Figura 2.9).

O método install é definido na classe *javacard.framework.Applet*, e é neste método que se deve colocar o código para criação do applet. A chamada ao método register, para que o applet seja registrado no JCRE, pode ser realizada no método construtor.

Na Figura 2.10 é apresentado o construtor da classe *UsuarioApplet*. Nele, é criada uma instância da entidade de domínio *Usuario*, que encapsula as propriedades e comportamentos de um usuário e uma instância do objeto *pin*.

Um PIN (*Personal Identification Number*) é um número de segurança, que deve ser fornecido antes do applet ser selecionado. Caso o número de tentativas de autenticação exceda o valor da constante que representa o número máximo de tentativas, o cartão é bloqueado. Caso o usuário seja bem sucedido na autenticação, este poderá executar as funções que são de direito.

O método *process* recebe os comandos APDU enviados pela aplicação *host*. O campo INS do APDU é recebido e verificado com o objetivo de determinar qual operação deve

```
01 public void process(APDU apdu) throws ISOException {
02       byte[] buffer = apdu.getBuffer();
03       byte cla = buffer[ISO7816.OFFSET_CLA];
04       byte ins = buffer[ISO7816.OFFSET_INS];
05
06       if (cla != CLA_SMARTESCOLA) {
07           ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
08       }
09       switch (ins) {
10           case INS_VERIFICAR_SENHA:
11               this.verificarSenha(apdu); break;
12           case INS_SET_MATRICULA:
13               this.setMatricula(apdu); break;
14           case INS_SET_TIPO:
15               this.setTipo(apdu); break;
16           case INS_GET_MATRICULA:
17               this.getMatricula(apdu); break;
18           case INS_GET_TIPO:
19               this.getTipo(apdu); break;
20           case INS_ADICIONAR_LOCAL:
21               this.adicionarLocal(apdu); break;
22           case INS_VERIFICAR_LOCAL:
23               this.verificarLocal(apdu); break;
24           case INS_REMOVER_LOCAL:
25               this.removerLocal(apdu); break;
26           case INS_ADICIONAR_CREDITOS:
27               this.adicionarCreditos(apdu); break;
28           default:
29               ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
30       }
31 }
```

Figura 2.11: Método *process* de UsuarioApplet.

ser executada. A implementação do método *process* é obrigatória pelo fato deste método ser herdado da classe *javacard.framework.Applet* e ser abstrato.

A Figura 2.11 apresenta o método *process* de *UsuarioApplet*. Este método tem uma APDU como parâmetro. Inicialmente, é recuperado o *buffer* da instrução APDU, recuperando respectivamente, a classe e o valor da instrução. É verificada a correspondência da classe APDU com o valor da constante *CLA_SMARTESCOLA*. Caso seja diferente, uma exceção é lançada. O mesmo ocorre com a instrução. Caso o campo INS não corresponda ao código de nenhum dos métodos do applet, a exceção de instrução não suportada é lançada (*SW_INS_NOT_SUPPORTED*).

A seguir, serão analisados os métodos *verificarSenha* e *getTipo* para facilitar o entendimento do applet desenvolvido. Como os métodos têm estrutura semelhante, não se faz necessária a explicação de todos. O Apêndice A apresenta a codificação completa das

```
01 private void verificarSenha(APDU apdu) {
02     byte tam_senha = (byte) apdu.setIncomingAndReceive();
03     byte[] buffer = apdu.getBuffer();
04     if (!pin.check(buffer, ISO7816.OFFSET_CDATA, tam_senha)) {
05         ISOException.throwIt(SW_SENHA_INVALIDA);
06     }
07 }
```

Figura 2.12: Método *verificarSenha* de UsuarioApplet.

classes *Usuario* e *UsuarioApplet* que são utilizadas no desenvolvimento deste exemplo.

Os métodos que tratam os comandos APDU devem ser preferencialmente privados, para que outros objetos não tenham acesso. Como todos os métodos são invocados durante a execução do método *process*, estes recebem um objeto APDU como parâmetro.

Método *verificarSenha(apdu)*: O métoto *verificarSenha* é responsável pela autenticação de um usuário. O método recebe um objeto APDU como parâmetro. O objeto APDU invoca o método *setIcomingAndReceive* para que o JCRE torne o(s) dado(s) acessível(is) através do *buffer* APDU recebido. A quantidade de *bytes* recebidos é retornado por este método e pode ser usado para verificar se o número de bytes recebidos é igual ao número de bytes esperado. O método *verificarSenha* de *UsuarioApplet* na Figura 2.12 recebe os dados através do *buffer* APDU, e depois verifica se a senha é válida. Se a senha não for válida ou o número de tentativas exceder a quantidade máxima permitida uma exceção é lançada com o valor *SW_SENHA_INVALIDA*. Esta verificação é feita pelo objeto *pin* do tipo *OwnerPIN*.

Método *getTipo(apdu)*: Quando o método necessita enviar dados de resposta para a aplicação *host*, o objeto APDU deve invocar o método *setOutgoing*. Este método notifica o JCRE que um dado será enviado no campo data da resposta APDU. O método *getTipo* retorna o tipo de usuário do cartão. Primeiramente é necessário carregar o *buffer* de dados da APDU, para que seja verificado se o tamanho da saída APDU obedece o máximo estabelecido.

É importante verificar que a referência ao objeto *u*, do tipo Usuário (entidade de domínio da aplicação) é quem tem as informações do usuário do cartão. O objeto usuário

```
01 private void getTipo(APDU apdu) {
02        byte[] buffer = apdu.getBuffer();
03        byte tam_saida = (byte) apdu.setOutgoing();
04        if (tam_saida != Usuario.TAMANHO_TIPO) {
05            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
06        }
07        byte indice_inicio = 0;
08        buffer[indice_inicio] = u.getTipo();
09        apdu.setOutgoingLength(Usuario.TAMANHO_TIPO);
10        apdu.sendBytes(indice_inicio, Usuario.TAMANHO_TIPO);
11 }
```

Figura 2.13: Método *getTipo* de UsuarioApplet.

retorna o valor do tipo que é armazenado no *buffer* de saída para que a resposta APDU seja enviada.

O método *setOutgoing* também retorna o tamanho do dado que a aplicação host espera receber. O método *setOutgoingLength* (Ver Figura 2.13) é chamado para informar o número de bytes a serem enviados. Por fim, o método *sendBytes* (Ver Figura 2.13) envia os dados de resposta para o usuário do cartão.

2.3 Conclusão

Neste capítulo, os conceitos e padrões para cartões inteligentes foram tratados com o objetivo de apresentar a estrutura básica, restrições e características de implementação de aplicações Java Card.

Desenvolvedores de aplicações Java para cartões podem usar frameworks, como OpenCard Framework, para que detalhes de comunicação possam ser abstraídos. Com isso, diferentes distribuições de cartões podem conter aplicações semelhantes, proporcionando portabilidade para as aplicações Java Card.

Detalhes com relações a restrições de *hardware* também devem ser levados em consideração durante o desenvolvimento dos applets Java Card. Como a Máquina Virtual Java para cartões implementa apenas um subconjunto da plataforma Java, é necessário conhecer quais objetos e classes podem ser utilizados.

Por fim, um exemplo de applet foi desenvolvido para que fossem apresentadas funções expecíficas necessárias para a comunicação e troca de mensagens utilizando o protocolo

APDU.

Capítulo 3

JML - Java Modeling Language

Java Modeling Language (JML) [20] é uma linguagem de especificação de interfaces e comportamentos (*BISL, Behavioral Interface Specification Language*) desenvolvida para a especificação de módulos de programas Java (classes, interfaces e métodos). Dizer que JML é uma BISL significa que com ela é possível especificar tanto a interface de um programa para o seu usuário como também detalhar o comportamento que vai ser oferecido para esse cliente. JML combina particularidades de linguagens DbC [26] (Design by Contract), como Eiffel [25], e a abordagem de especificação baseada em modelos, como VDM [2], B [1] e Z [44]. JML usa a sintaxe Java para escrever predicados, como invariantes, pré-condições e pós-condições.

A JML foi desenvolvida por Gary Leavens [20], estudantes e pesquisadores da Universidade de Iowa. Hoje, alguns grupos de pesquisa [5, 10, 23, 43, 3, 29, 6, 9] têm construído ferramentas que dão suporte à notação JML. O esforço cooperativo para o desenvolvimento de um padrão de especificação Java é importante tanto para desenvolvedores de ferramentas, como para usuários [4].

Além da JML, existem outras linguagem e métodos, como, por exemplo Z [44] e B [1], que são utilizados na atividade de especificação. Sistemas críticos, como programas para automação industrial, sistemas de tempo real e aplicações que lidam com valores monetários, que requerem um maior nível de segurança e corretude, são beneficiados com o uso desses formalismos.

3.1 Design by Contract - DbC

Design by Contract, ou Projeto por Contrato, é um método para o desenvolvimento de software. Design by Contract é o princípio pelo qual interfaces entre módulos de software devem ser estruturadas utilizando especificações precisas [18]. O princípio por traz de DbC é que uma classe e o seu cliente têm um contrato um com o outro. Os contratos definem obrigações e benefícios mútuos (pré-condições e pós-condições, respectivamente), além de restrições (invariante). Os contratos são definidos no código do programa através de comentários da própria linguagem de programação e são traduzidos em código executável pelo compilador. As propriedades (predicados) do contrato são conhecidas como asserções (*assertions*).

Asserções são representações de estados. São expressões verdadeiras em determinados pontos de um código [9], tornando-se importante quando se deseja provar a corretude de um programa. Para que seja possível provar a corretude de um determinado programa ou um trecho de código, é necessário definir expressões lógicas que representem estados desejados. Se um dos estados não for satisfeito, algo de errado ocorreu, seja com a assertiva definida ou com a própria definição do programa.

Meyer propôs, em seu trabalho sobre Design by Contract [26], asserções simples para a linguagem Eiffel [25]. Sua idéia era usar as asserções para definir uma interface entre os módulos do programa, e também entre o módulo e o cliente que usa o módulo.

Um contrato envolve 2 (duas) partes, o fornecedor e o cliente de um serviço. Cada uma das partes esperam algum benefício do contrato. Meyer em [26] apresenta duas propriedades características no desenvolvimento de um contrato, são elas:

- Cada parte espera algum benefício do contrato e está preparado para ter obrigações com o objetivo de obter estes benefícios e,
- Os benefícios e obrigações devem estar devidamente documentados.

O documento de contrato protege ambas as partes para garantir o que foi definido como obrigação e benefício. O benefício de uma das partes do contrato é a obrigação da outra. A Tabela 3.1 mostra um exemplo de contrato.

Parte	Obrigações	Benefícios
Cliente	1. As cartas e pacotes não podem ter mais que 5kg. 2. A dimensão dos pacotes não pode exceder 2 metros. 3. Pagar 40 reais.	Ter os pacotes entreges ao destinatário em 1 dia ou menos.
Fornecedor	1. Entregar os pacotes e cartas ao destinatário em 1 dia ou menos.	Não precisa se preocupar com pacotes pesados, pacotes muito grandes ou serviço não pago.

Tabela 3.1: Exemplo de Contrato.

A Tabela 3.1 apresenta um contrato para entrega de correspondências. O cliente do serviço tem como obrigação pagar 40 reais pelo serviço, e entregar pacotes menores que 2 metros e com peso abaixo de 5 kilos. Como benefício, o cliente terá seu pacote entregue ao destino em no máximo 1 dia. O fornecedor do serviço tem como benefício não se preocupar com pacotes muito grandes e pesados. Como obrigação, o fornecedor deverá entregar a correspondência em no máximo 1 dia.

Os benefícios [15] de se utilizar projeto por contrato incluem: uma abordagem sistemática de construir sistemas Orientados a Objetos com menos *bugs*; um método para documentar componentes de software; um *framework* que facilita a depuração, teste e proporciona garantia de qualidade ao produto desenvolvido.

Projeto por contrato é uma forma de atribuir responsabilidades aos métodos, o que deve ser feito, quais as saídas possíveis e que argumentos são necessários para as chamadas do método. Com isso, é possível atribuir culpa a uma parte das partes do contrato, caso haja problema. Assim, caso uma pré-condição seja violada, a culpa será atribuída ao cliente, semelhantemente, caso uma pós-condição seja violada, a culpa será do método fornecedor do serviço [20]. Projeto por contrato também é um mecanismo auxiliar aos comentários do programa e documentação não-formais. Apesar de usar formalismo em seu comentário, o uso de DbC não exclui os outros tipos de comentários de programa.

Além dos benefícios descritos sobre o uso de contratos, é possível garantir robustez e reusabilidade. Tratamento de exceções é um mecanismo de robustez no uso de projeto por contrato [33]. Dessa forma, quando uma asserção não for satisfeita, o tratamento da exceção deve garantir o estado desejado do objeto para a pós-condição excepcional. Definir

contrato para pós-condições não normais (excepcionais) proporciona robustez para o sistema.

A reusabilidade é alcançada no uso de DbC de 2 maneiras segundo Scott [33]. Não é necessário analizar o código da implementação para ver se realmente está sendo feito o que é sugerido. Este comportamento é garantido pelo contrato. A segunda maneira é no uso de herança. No desenvolvimento de sistemas podem ocorrer heranças entre as entidades de domínio. O tratamento de herança também pode ser feito da definição do contrato. Clientes que herdam classes com contratos já definidos podem ter certeza que os princípios definidos na classe pai serão garantidos. A subclasse deve herdar e obedecer as obrigações dos contratos da superclasse.

É importante incluir as pré e pós-condições como parte na declaração de cada método (rotina). Da mesma forma, o invariante de classe representa os estados desejáveis antes e após a execução de cada método. As asserções são expressões booleanas que expressam quais valores podem ser atribuídos às variáveis globais e locais.

A seguir serão apresentados os conceitos de invariante, pré e pós-condição, e herança, definidos em projeto por contrato.

3.1.1 Invariante

O conceito de invariante de classe na orientação a objetos tem origem com Hoare [7] na definição de invariante de dados. A mudança do conceito de invariante de dados para o desenvolvimento de software orientado a objetos, na forma de invariante de classe, ocorreu na definição da linguagem Eiffel, por Bertrand Meyer [25]. O invariante de classe é uma propriedade que se aplica a todas as instâncias de uma classe, não apenas a uma rotina específica [26].

O invariante deve ser preservado por todos os métodos da classe. O invariante deve ser satisfeito após o retorno de cada chamada de método, caso ele também seja satisfeito na chamada do método. O invariante é “adicionado” a cada pré e pós-condição de cada método, contudo, a sua caracteristica não é específica de cada método, mas sim à classe como um todo.

```
//@ requires true;
DEFINIÇÃO DO MÉTODO

//@ ensures true;
DEFINIÇÃO DO MÉTODO
```

Figura 3.1: Pré e Pós-Condições Verdadeiras.

3.1.2 Pré e Pós-Condições

As pré e pós-condições são estruturas base para a especificação do comportamento de uma função. São elas que determinam o que uma função espera para funcionar corretamente, bem como o que vai acontecer no final de sua execução. A pré-condição expressa requisitos que devem ser satisfeitos (*verdade*) antes das chamadas dos métodos. A pós-condição expressa propriedades que devem ser verdadeiras após a execução do método.

A falta da definição de uma pré-condição equivale a uma declaração verdadeira (*true*). Desta forma, não existe nenhuma propriedade a ser satisfeita antes da chamada do método (Ver Figura 3.1). O mesmo ocorre para a falta da definição de uma pós-condição. Uma asserção definida como *true* significa que qualquer estado a satisfaz, seja para uma cláusula de pré-condição, ou para uma cláusula de pós-condição. O contrato é escrito logo antes do cabeçalho do método. No exemplo da Figura 3.1, a palavra *requires* representa a pré-condição do método, e a palavra *ensures* representa a sua pós-condição.

3.1.3 Herança de Contrato

Na especificação de contratos para linguagens Orientadas a Objetos, é necessário observar as classes que herdam ou implementam propriedades e comportamentos de outras classes. O conceito de DbC definido por Meyer [25, 26] inclui herança de especificação. Se uma classe X estiver estendendo uma outra classe Y, então todas as especificações para os métodos de Y continuam sendo válidas na classe X.

Com relação à especificação, existem dois possíveis casos:

1. A especificação vai ser totalmente reescrita;
2. A especificação anterior ainda é válida, e é necessário estende-la.

No primeiro caso é necessário refazer a especificação do método que está sendo reescrito normalmente. No segundo caso não é necessário redefinir a especificação, basta apenas estender as características do pai e acrescentar novas propriedades ao filho, além de satisfazer as especificações da classe pai, também vai satisfazer as que estão sendo escritas na nova classe.

3.2 Estrutura da Linguagem JML

Nesta seção será descrita a estrutura da linguagem JML, suas principais características, cláusulas e sintaxe.

JML foi projetado para ser usado em conjunto com uma variedade de ferramentas. Essas ferramentas suportam Design by Contract, checagem em tempo de execução, descoberta de invariante, verificação estática e verificação formal, que usa provadores de teoremas. Tais ferramentas serão descritas na seção 3.4.

O foco central de uma especificação JML são a pré-condição e a pós-condição (para os métodos) e o invariante (para a classe). As asserções JML são escritas como um comentário especial JAVA. Os comentários são ignorados pelo compilador Java, mas podem ser usados pelas ferramentas que dão suporte à JML [5]. Tais comentários extendem a sintaxe Java, adicionando algumas palavras chaves (reservadas) e construções sintáticas. Dentre algumas palavras reservadas, têm-se: *invariant, requires, assignable, ensures, signals, pure*; e alguns operadores como: *\forall, \exists, \old e \result* [21]. A JML ainda possui outras cláusulas que auxiliam no processo de especificação de software em Java. Tais cláusulas podem ser vistas com detalhes no manual de referência JML [21] e em [9] e podem ser encontradas na página oficial[1] da linguagem JML.

A especificação JML é escrita na forma de comentários Java (*Annotation*), tendo como marca inicial da especificação //@, para comentários de 1 (uma) linha, ou marcações entre /*@ .. @*/, para comentários de múltiplas linhas.

A Tabela 3.2 apresenta algumas cláusulas usadas em uma especificação JML.

[1] http:\\www.jmlspecs.org

Expressões JML	**Significado**
invariant E_1*;*	invariante
constraint E_1*;*	constraint
requires E_1*;*	pré-condição
ensures E_1*;*	pós-condição
signals (Exception) E_1	pós-condição excepcional
assignable α *;*	variáveis que podem ser modificadas
$\backslash old(\alpha)$;	valor de estado inicial de α
$\backslash result$;	define o valor a ser retornado
$\backslash forallD;\ E_1(opt);\ E_2$;	$\forall\, D \bullet E_1 \Rightarrow E_2$
$\backslash existsD;\ E_1(opt);\ E_2$;	$\exists\, D \bullet E_1 \Rightarrow E_2$

Tabela 3.2: Algumas Expressões JML.

```
public abstract class Invariant {
01    boolean[] b;
02    //@ invariant b != null && b.length == 6;
03
04   //@ assignable b;
05   Invariant() {
06      b = new boolean[6];
07   }
08 }
```

Figura 3.2: Exemplo de Invariante [21].

3.2.1 Cláusula *invariant*

Em JML, a declaração de um invariante (ver Seção 3.1.1) é feita através da palavra reservada *invariant*. Todo invariante é predicado, como apresentado na gramática abaixo.

invariant ::= *invariant-keyword predicate* **;**
invariant-keyword ::= **invariant**

As propriedades do invariante devem ser satisfeitas em todos os estados do objeto. Um estado é válido em um objeto *o* se este estado pode ocorrer em algum momento da execução do programa [21], por exemplo, após a execução do construtor da classe e, antes e depois da execução dos métodos.

No exemplo da Figura 3.2, a variável *b* tem visibilidade *default*, podendo ser acessada pela especificação JML. A declaração do invariante define que para todos os estados do objeto a variável *b* não é nula (*null*) e que o tamanho do array é de exatamente 6 posições. Pode-se perceber que no construtor da classe nunhum parâmetro é passado, contudo, o valor do tamanho de *b* é 6.

```
01 public abstract class Constraint{
02    int a;
03    //@ constraint a == \old(a);
04
05    boolean[] b;
06    //@ invariant b != null;
07    //@ constraint b.length == \old(b.length) ;
08
09    boolean[] c;
10    //@ invariant c != null;
11    //@ constraint c.length >= \old(c.length) ;
12
13    Constraint(int bLength, int cLength) {
14        b = new boolean[bLength];
15        c = new boolean[cLength];
16    }
17 }
```

Figura 3.3: Exemplo de Constraint [21].

3.2.2 Cláusula *constraint*

A cláusula *constraint* é utilizada quando se necessita restringir os possíveis estados do objeto. Esta cláusula denota uma relação entre o pré e pós-estado do método, restringindo como a variável pode ser modificada. É possível definir que uma variável é constante ou que a mesma pode apenas receber valores maiores que o atual atribuído. As *constraints* restringem a forma como os valores dos atributos podem mudar durante a execução do programa, diferente do invariante, que assumem um caráter estático. As *constraints* podem ser vistas como sendo pós-condições para todos os métodos.

Por exemplo, na Figura 3.3, a *constraint* diz que o valor da variável *a* e o tamanho do *array* representado pela variável *b* nunca irão mudar. Já o valor do *array* representado pela variável *c* irá apenas aumentar de valor.

Todos os métodos em classes que herdam de uma classe que contém a cláusula *invariant ou constraint*, devem respeitar a especificação de tais cláusulas definidas na classe original. Diferentemente do invariante, a cláusula *constraint* pode utilizar o operador \old, devido ao fato das *constraints* possuírem um caráter dinâmico. O operador *\old()* faz referência a um pré-estado de uma variável, como por exemplo: *\old(*α*)* <= α, define que ao término (pós-condição) de uma rotina (método), o valor de α é maior ou igual ao seu valor inicial no início da rotina. A linha 11 da Figura 3.3 apresenta um exemplo do uso de *old*. O invariante define apenas quais estados devem ser satisfeitos após a execução

```
01  public class Person{
02    private String name;
03    private int weight;
04
05    public int getWeight() {
06       return weight;
07    }
08
09    public void addKgs(int kgs) {
10      if (kgs >= 0) {
11         weight += kgs;
12      }
13    }
14
15    public Person(String n) {
16      name = n; weight = 0;
17    }
18  }
```

Figura 3.4: Implementação da Classe *Person* (adaptada de [20]).

de cada método, não como as variáveis podem ser modificadas. Devido a isto, é possível utilizar o operador $\backslash old()$ em uma cláusula *constraints* e não em um *invariant*.

3.2.3 Cláusulas *requires* e *ensures*

O primeiro passo para escrever contratos é organizar os comentários de programas de modo a descrever os contratos dos métodos. Isto é, definir o comportamento dos métodos em pré e pós-condições. Em JML as cláusulas *requires* e *ensures* representam pré e pós-condições, respectivamente. Os métodos são anotados com o objetivo de poder ter uma representação do que realmente deve ser implementado na função.

Será feita uma descrição informal com o objetivo de expressar o exemplo da Figura 3.4. Esse exemplo define uma implementação para a classe *Person*, que representa uma entidade pessoa, tendo como variáveis o nome e o peso. Como comportamentos, são definidos métodos, o primeiro para recuperar o valor do peso (*weight*) e o segundo para adicionar um valor inteiro ao peso. São os métodos *getWeight e addKgs*, respectivamente, além do construtor da classe *Person*.

O construtor da classe tem um objeto *String* como parâmetro. Este parâmetro atribui um valor à variável global *name*. O peso é definido com o valor inicial zero. O método *getWeight* apenas recupera o valor de *weight*, e o método *addKgs* adiciona um valor maior

```
01  public class Person {
02     private /*@ spec_public non_null @*/ String name;
03     private /*@ spec_public @*/  int weight;
04
05   /*@ invariant !name.equals("")
06     @ && weight >= 0;
07     @*/
08
09   //@ ensures \result == weight;
10   public /*@ pure @*/ int getWeight();
11
12   /*@ requires weight + kgs >= 0;
13     @ ensures kgs >= 0
14     @ && weight == \old(weight + kgs);
15     @*/
16   public void addKgs(int kgs);
17
18   /*@ requires n != null && !n.equals("");
19     @ ensures n.equals(name)
20     @ && weight == 0; @*/
21     public Person(String n);
22 }
```

Figura 3.5: Especificação JML da classe *Person* .

que zero ao peso.

Na Figura 3.5 é definida a especificação JML para a classe *Person*. É usada a cláusula *spec_public* para as variáveis da classe. Isso ocorre pelo fato destas serem declaradas como privadas (*private*). A cláusula *spec_public* representa que esta variável é pública no escopo da especificação. Se as variáveis não fossem definidas como *spec_public*, não seria possível serem utilizadas na especificação JML. Se uma das variáveis fosse definida com visibilidade pública não seria necessário utilizar a cláusula *spec_public* para ela. A cláusula *non_null* determina que a variável *name* não pode ser nula. A construção *non_null* é utilizada para objetos em variáveis globlais e parâmetros de métodos. A definição de uma variável não nula na contrução de um invariante assume a forma *name != null*.

É possivel utilizar expressões ou chamada de métodos Java nas especificações JML. Por exemplo, o objeto *name* é do tipo *String*. Este objeto pode executar o método *equals*, método este que é utilizado na definição do invariante. O invariante da classe *Person* define propriedades como o valor do *name*, que não pode ser vazio, e o valor de *weight*, que deve ser sempre maior que zero.

A especificação do método *getWeight* utiliza a cláusula *ensures* para definir a pós-

condição do método. A especificação define que o valor de retorno do método tem que ser a variável global *weight*. Na definição do predicado é utilizado o operador *result*. O operador *result* define qual deve ser o valor de retorno do método. Esta cláusula deve ser usada apenas dentro de uma cláusula *ensures*.

O método *getWeight* também é definido como puro (/*@ *pure* @*/). Um método puro em JML é aquele que não altera o estado do objeto.

A cláusula *requires* é utilizada na especificação do método *addKgs* para definir que o valor de *weight* não pode ser negativo. O valor da variável inteira *kgs*, passada como parâmetro do método, mais o valor de *weight* deve ser maior ou igual a zero. Esta pré-condição poderia ser omitida, pois, como esta propriedade já está definida no invariante, o valor de *weight* é verificado 2 vezes na execução do método *addKgs*.

Na definição da pós-condição do mesmo método é utilizado o operador *old* para definir que o valor final de *weight* após a execução do método deve ser o seu valor inicial mais o valor passado como parâmetro em *kgs*.

3.2.4 Cláusula *also*

A especificação de um método, seja em uma classe ou interface, deve começar com a cláusula *also* se, e somente se, este método tiver sido declarado na classe pai. Utilizar a cláusula *also* define que está sendo adicionada alguma especificação à que já existe na superclasse. Dessa forma, a classe filha deve obedecer a especificação previamente definida na superclasse, mais a nova especificação definida após a cláusula *also*.

A gramática JML abaixo apresenta uma parte da definição de uma especificação de método. Uma *extending-specification* é uma especificação normal JML, contudo, a cláusula **also** faz com que quem está lendo a especificação entenda que as propriedades da superclasse também devem ser garantidas durante a execução do método na subclasse.

method-specification ::= *specification* | *extending-specification*
extending-specification ::= **also** *specification*

A Figura 3.6 apresenta um exemplo onde, na entidade *SuperClasse*, os parâmetros *x* e *y* do *metodoA* têm que ser maior que 4, e maior ou igual a 2 vezes *x*, respectivamente.

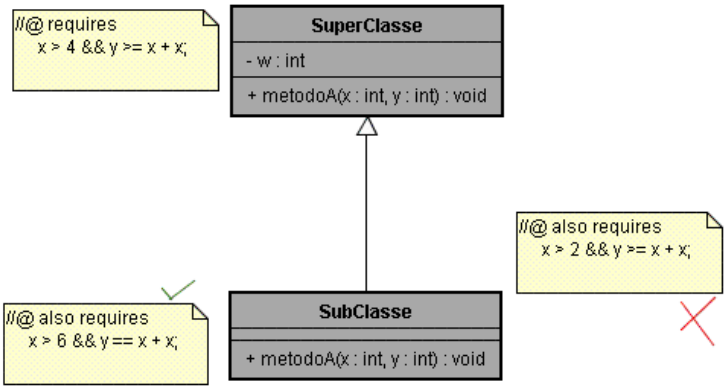

Figura 3.6: Exemplo de Uso da Cláusula *also*.

Na entidade *SubClasse* o *métodoA* é sobrecarregado e são definidas 2 especificações. A especificação válida restringe a especificação da superclasse, definindo um valor maior que 6 para x, e para y o valor igual a 2 vezes o valor de x. Se o valor de x for 7 e o valor de y for 14, a pré-condição é válida tanto para a superclasse como para a subclasse, pois, 7 é maior que 4 e também maior que 6, e 14 é igual a 2 vezes o valor de x.

A especificação menos restrita define um valor para x maior que 2. Se, na subclasse, o valor de x for 3, a pré-condição da superclasse é quebrada, pois x deve ser maior que 4.

Em uma construção *also*, a especificação da superclasse é adicionada a especificação da subclasse [21]. Primeiro é verificada a especificação da subclasse, caso tenha resultado verdadeiro, em seguida é verificada a especificação da superclasse. Apesar da especificação para pré-condição da subclasse da Figura 3.6 ser mais forte que a especificação da superclasse, o objetivo deste exemplo é apresentar a sintaxe da construção JML para herança de especificação.

3.2.5 Cláusula *signals*

A especificação de um método pode conter pós-condicão excepcional. Uma pós-condicão excepcional define que propriedades devem ser satisfeitas caso o método termine abruptamente, lançando uma exceção.

Na Figura 3.7, ao método *addKgs* foi acrescrentado o controle de exceção no escopo da especificação. Caso o valor do parâmetro seja negativo, a exceção *IllegalArgumentException* é lançada. A cláusula *signals* e *signals_only* define quais exceções podem ser

```
01   /*@ requires weight + kgs >= 0;
02     @ ensures kgs >= 0
03     @ && weight == \old(weight + kgs);
04     @ signals_only IllegalArgumentException;
05     @ signals (IllegalArgumentException)
06     @    kgs < 0;
07     @*/
08    public void addKgs(int kgs){
09       if (kgs >= 0) {
10          weight += kgs;
11       } else {
12         throw new IllegalArgumentException();
13       }
14    }
```

Figura 3.7: Exemplo de Cláusula *Signal*.

lançadas. No exemplo da Figura 3.7, mais especificamente, apenas a exceção *IllegalArgumentException* pode ser lançada.

3.2.6 Cláusula *assignable*

A especificação de um método pode conter uma cláusula que defina quais variáveis podem ser modificadas em uma chamada de método. A notação em JML é *assignable, modifies ou modifiable*. A cláusula *assignable* é mais comumente utilizada. Por padrão, se a cláusula *assignable* não for declarada, esta assume o valor *everything*, que indica que todas as variáveis podem ser modificadas. Outro valor para esta cláusula é *nothing*, onde se define que nenhum valor pode ser atribuído a qualquer variável. Os métodos declarados como *pure*, por exemplo, o método *getWeight* na Figura 3.5, tem o valor *nothing* para *assignable* por padrão.

//@ assignable \nothing ;
//@ assignable \everything ;

Se à espeficicação do método *addKgs* fosse acrescida a cláusula *assignable* para a variável *weight*, sua especificação seria mais restrita. A adição desta cláusula faria com que apenas o estado de *weight* pudesse ser modificado. A Figura 3.8 apresenta a especificação do método *addKgs* modificada.

```
01   /*@ requires weight + kgs >= 0;
02     @ ensures kgs >= 0
03     @ && weight == \old(weight + kgs);
04     @ assignable weight;
05     @ signals_only IllegalArgumentException;
06     @ signals (IllegalArgumentException)
07     @    kgs < 0;
08     @*/
09    public void addKgs(int kgs);
```

Figura 3.8: Exemplo de Cláusula *Assignable*.

```
01   (\sum int a; 0 <= a && a < 7; a) == 21
02   (\product int a; 0 < i && i < 7; a) == 720
03   (\max int a; 0 <= i && a < 7; a) == 6
04   (\min int a; 0 <= i && a < 7; a-1) == -1
```

Figura 3.9: Exemplo das Cláusulas *sum, product, max e min*.

3.2.7 Operadores *max, min, sun, product* e *num_of*

Os operadores \sum, \product, \min e \max retornam a soma, produto, mínimo e máximo de valores em um determinado conjunto. A Figura 3.9 apresenta um exemplo para as construções \sum, \product, \max e \min, respectivamente. Todos os operadores requerem um *inteiro* ou *float*.

O quantificador numérico \num_ of, retorna os valores de uma variável que se encontra dentro da faixa definida. A Figura 3.10, apresenta um invariante que estabelece que o número máximo de alunos com menos de 18 anos em uma determinada turma é 7. O quantificador utiliza o tipo *long* como retorno.

3.2.8 Cláusulas *work_space* e *when*

A cláusula *work_space* define o tamanho de um heap em bytes usado por um método quando este é chamado. O retorno deste operador é do tipo *long*. A cláusula *when* anota um método no que se refere a concorrência. Métodos que podem ser submetidos a acesso concorrênte é anotado com a cláusula *when*.

```
01  //@ public invariant (\num_of Estudante e;
02                        turma.contains(e); e.idade < 18)
03                        <= 7;
```

Figura 3.10: Exemplo da Cláusula *num_of*.

```
01 /*@ public invariant
02   @ !(\exists Estudante e;
03   @    alunos.contains(e);
04   @    e.getOrientador() == null)
05   @ */
```

Figura 3.11: Exemplo de Cláusula *exists*.

3.2.9 Cláusula *forall* e *exists*

JML incorpora os quantificadores universal (*forall*) e existencial (*exists*) como cláusulas da linguagem. A especificação de uma expressão quantificada em JML é definida na gramática a seguir:

spec-quantified-expr ::= *(quantifier quantified-var-decls* **;** *[[predicate]* **;** *]*
spec-expression)

O não-terminal *predicate* define o domínio dos valores na qual a expressão definida em *spec-expression* pode ser aplicada. Por exemplo, na cláusula *forall* definida a seguir, têm-se: Sejam *i* e *j*, em um domínio dos inteiros. Para todo *i* maiores ou iguais a zero e para todo *j* menores que 10, sendo *i* menor que *j*, o valor do índice *i* no vetor *a* é menor que o valor do índice *j* no vetor *a*.

\forall int i,j; 0 <= i && i < j && j < 10; a[i] < a[j];

$$\equiv$$

$$\forall\, i, j \in \mathbf{Z} \bullet 0 <= \mathrm{i} \land \mathrm{i} < \mathrm{j} \land \mathrm{j} < 10 \Rightarrow a[i] < a[j]$$

Ocorre o mesmo da lógica de predicados: $\forall\, x : X \in D \bullet P(x)$, (\forall X x ; D ; P - em JML), onde X é o tipo, x é a variável, D é o domínio e P é o predicado.

A estrutura sintática da cláusula (*exists*) é semelhante a cláusula (*forall*). Em ambos, a definição de domínio é opcional.

Na Figura 3.11 é apresentado um exemplo do uso da cláusula \exists no invariante. A expressão define que não existe uma entidade *Estudante* que não tenha um orientador. Isto equivale a dizer que todo objeto estudante em alunos tem um orientador, como mostrado na Figura 3.12.

```
01 /*@ public invariant
02   @ (\forall Estudante e;
03   @    alunos.contains(e);
04   @    e.getOrientador() != null)
05   @ */
```

Figura 3.12: Exemplo de Cláusula *forall*.

```
01  //@ requires value >= 0;
02  public static int isqrt(int value)
03  {
04    return (int) Math.sqrt(value);
05  }
```

Figura 3.13: Exemplo de Especificação Leve.

3.3 Tipos de Especificação de Métodos JML

Existem dois tipos de especificação em JML, são elas: especificação *lightweight* (leve) e especificação *heavyweight* (pesada). Uma especificação *lightweight* é uma especificação incompleta, onde pré-condições para métodos Java são obrigatórias, contudo, a definição de pós-condição pode ser omitida.

Uma omissão de cláusula em JML é interpretada como sendo *true* ou *false*. Uma especificação *lightweight* define pós-condições como verdadeiras. Uma especificação *heavyweight* é uma especificação completa, onde cada método tem uma definição clara de pós-condição.

Uma especificação *heavyweight* é marcada por 3 tipos de comportamentos, na qual cada comportamento é representado por uma palavra-chave, como, **behavior**, **normal_behavior** e **exceptional_behavior**. Neste tipo de especificação o comportamento dos métodos tem um significado bem definido pela especificação [9]. A omissão de uma ou das 3 dessas palavras-chave caracteriza a definição de uma especificação leve. A especificação *lightweight* não usa as palavras-chave que expressam comportamento.

Por exemplo, na Figura 3.13, a única regra definida na pré-condição é que o valor do argumento deve ser positivo, contudo, não define regras para as variáveis que podem mudar de estado ou em relação a qual deve ser o valor do resultado, ou ainda se o método lança alguma exceção. Note que a especificação é dada apenas com uma simples cláusula *requires*. Como a especificação de *isqrt* não tem nenhuma palavra que determine o com-

```
01  /*@ public normal_behavior
02    @ requires ...;
03    @ assignable ...;
04    @ ensures ...;
05    @ also public exceptional_behavior
06    @ requires ...;
07    @ assignable ...;
08    @ signals(...) ...;
09    @*/
```

Figura 3.14: Modelo de Especificação Pesada.

```
01  /*@ normal_behavior
02    @ requires value >= 0;
03    @ assignable value;
04    @ ensures \result * \result <= \old{value};
05    @*/
06  public static int isqrt(int value)
07  {
08    return (int) Math.sqrt(value);
09  }
```

Figura 3.15: Exemplo de Especificação Pesada.

portamento do método, esta é uma especificação leve.

Um comportamento normal e um comportamento excepcional são exemplos de especificações pesadas. Para cada comportamento pode-se especificar: visibilidade, *assignable* e pós-condições. O modelo da Figura 3.14 apresenta uma especificação com múltiplos comportamentos que definem tanto um comportamento normal como um comportamento excepcional.

Na Figura 3.15, o método *isqrt* tem a estrutura da especificação modificada, sendo definido um comportamento normal. A única variável que pode ter seu valor modificado na função é *value*, e a parte inteira retornada deve ser igual ou menor ao argumento do parâmetro, no seu estado inicial. Note também que foi utilizada a palavra **normal_behavior** para determinar um tipo de comportamento.

3.4 Ferramentas de Verificação JML

Para uma linguagem de especificação, da mesma forma que para uma linguagem de programação, algumas ferramentas são necessárias para suprir as necessidades dos usuários

da linguagem, como leitura, escrita e verificação de tipos [5]. As ferramentas de verificação JML podem ser classificadas em 2 (duas) categorias:

- Verificação em tempo de execução e,
- Verificação estática.

Checagem em tempo de execução envolve o teste da especificação durante a execução do programa. Os erros são apresentados em tempo de execução. A idéia de verificação de contratos no momento da execução teve início com a linguagem Eiffel [25] no final da década de 80. A principal ferramenta com esta característica para a linguagem JML é a *jmlc* [9].

Yoonsik Cheon [9] definiu o verificador *runtime* para JML. Desenvolver uma estrutura para verificar código de especificação em tempo de execução é uma tarefa que tem como objetivo apresentar sua viabilidade e eficiência quando aplicada a programação. Um requisito essencial na verificação *runtime* é a transparência, pois o comportamento do programa original escrito não é modificado. A especificação deve garantir estados finais a determinadas variáveis, caso algum problema ocorra, ou uma exceção seja lançada. Outro fator a ser levado em consideração é a semântica da linguagem (JML), pois é importante que o código da especificação reflita realmente o que necessita ser representado. Por fim, a verificação *runtime* deve detectar erros a partir das assertivas.

Na verificação estática, a verificação é feita antes da execução do programa. O adjetivo *estático* enfatiza que a verificação ocorre através de uma análise estática do código. Uma importante ferramenta de verificação estática para JML é o ESC/Java [22].

Serão apresentadas nas seções a seguir ferramentas que dão suporte a JML.

3.4.1 jmlc - JML Compiler

O compilador JML é uma extensão do compilador Java. O compilador inclui verificação dinâmica das instruções JML como pré condições, pós condições normais e com exceções, invariante e as cláusulas definidas na seção 3.2.

Semelhantemente ao compilador Java, o compilador JML produz bytecode a partir do código-fonte e especificação, adicionando às declarações que verificam o código original.

A tradução da especificação JML em código para verificação *runtime* é feita em 3 passos.

1. Simplificar a especificação JML em uma forma que facilite a tradução. Esta tarefa identifica e extrai asserções executáveis dos métodos que contém especificação.

2. Traduzir as asserções simplificadas em código *runtime*.

3. Adicionar o código gerado no código original. Este passo insere o código de verificação em tempo de execução no local apropriado do código original. Por exemplo, o código da pré e da pós-condição devem ser executados antes e após a execução do corpo do método, respectivamente.

Cheon [9] define uma abordagem para geração de código de especificação, esta abordagem é chamada de aboradagem *wrapper*. O método original se transforma em um método que tem por responsabilidade checar as regras da especificação definida na classe. Usando o método *wrapper* as asserções definidas na especificação dão origem a métodos que checam as pré-condições ou pós-condições, sejam elas normais ou excepcionais. O método wrapper substitui o método original.

O método original, após ser compilado com o *JMLc*, torna-se privado. Contudo o método original é chamado pelo método wrapper gerado para que o retorno do que foi programado ocorra corretamente. Se existir um método original *m* com x_n parâmetros e T_n tipos, será gerado um método com a mesma quantidade de parâmetros. Estes parâmetros serão utilizados quando o método original for chamado (Ver Figura 3.16). A visibilidade do método wrapper será a mesma do método original; por exemplo, será publica se o método original for público.

O método *wrapper* é chamado quando os clientes necessitam utilizar o método original que tem a mesma assinatura. Cada método pode lançar uma exceção, caso os parâmetros ou retornos não satisfaçam a especificação do método. A Figura 3.17 apresenta a estrutura em tempo de execução da abordagem *wrapper*.

```
01  T m(T1 x1, : : :, Tn xn) {
02    checkInv$S();
03    checkPre$m$S(x1, : : :, xn);
04    T rac$result;
05    try {
06       rac$result = orig$m(x1, :::, xn);
07       checkPost$m$S(x1, : : :, xn, rac$result);
08       return rac$result;
09    }
10    catch (JMLEntryPreconditionError rac$e) {
11       throw new JMLInternalPreconditionError(rac$e);
12    }
13    catch (JMLAssertionError rac$e) {
14       throw rac$e;
15    }
16    catch (Throwable rac$e) {
17       checkXPost$m$S(x1, : : :, xn, rac$e);
18    }
19    finally {
20         if (/* no postcondition violation? */) {
21             checkInv$S();
22             checkHC$S();
23         }
24    }
25  }
```

Figura 3.16: Estrutura do método wrapper [9].

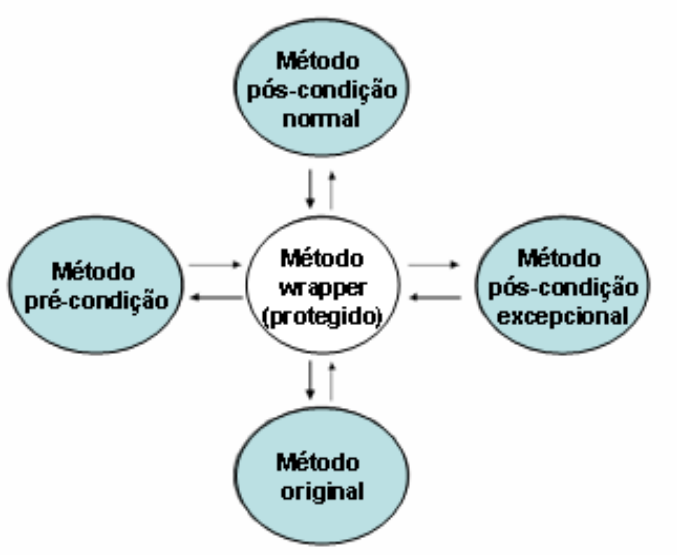

Figura 3.17: Abordagem *Wrapper* - Controle de Fluxo.

As regras para a geração de código para verificação em tempo de execução a partir das anotações JML de acordo com a abordagem *wrapper* são definidas com detalhes no trabalho de Yoonsik Cheon [9] e também será detalhada, no que interessar a esse trabalho, no Capítulo 4.

3.4.2 Outras Ferramentas JML

Existem ferramentas JML que tem como foco a verificação estática, são elas: *ESC/Java, Loop* e *Jack*.

ESC/Java A ferramenta ESC/Java [22] faz verificação estática a partir da especificação, e foi desenvolvida pela Compaq Systems Research Center. A ferramenta pode checar assertivas simples e pode checar alguns tipos de erros comuns no código Java, como referência nula ou índices de arrays fora do limite.

LOOP A ferramenta LOOP [43], desenvolvida na Universidade de Nijmegen - Holanda, traduz código Java anotado com JML em obrigações de prova para o provador de teoremas PVS [34]. LOOP permite verificação de propriedades mais complicadas que o ESC/Java.

Jack - *Java Applet Correctness Kit* A partir de programas Java anotados com JML, o JACK também gera obrigações de prova para as ferramentas Coq [12], PVS [35] e Simplify [14]. Esta ferramenta foi desenvolvida pela Gemplus e pela *Everest team* em INRIA Sophia Antipolis. JACK tem como objetivo prover um ambiente de desenvolvimento para verificação de programas Java e Java Card utilizando anotações JML.

3.5 Conclusão

Neste capítulo, foi apresentado o que é JML, seus conceitos, estrutura e ferramentas. JML é baseado na especificação de modelos e interface de comportamento, utilizando conceitos de projeto por contrato. JML facilita o desenvolvimento de aplicações orientada

a objetos Java, pois, além de ter sintaxe semelhante à tecnologia, não interfere diretamente na programação. A definição da especificação JML é feita através de comentários.

Como a linguagem JML é baseada nos conceitos definidos por Meyer [26], o significado e a aplicação de projeto por contrato foi detalhado neste capítulo. Conceitos como invariante de classe, pré e pós-condições e herança de contratos foram tratados. A estrutura da linguagem JML, cláusulas e sua utilização também fizeram parte deste capítulo.

Por fim, os tipos de especificação e as principais ferramentas que dão apoio à linguagem JML foram descritas.

Capítulo 4

Aplicabilidade de JML a Java Card

Sistemas Java Card se prestam bem para a aplicação de métodos formais, especificamente, projeto por contrato, por algumas razões:

- *applets* Java Card são usados frequentemente em aplicações críticas, na qual erros de programação podem ter conseqüências sérias;
- Como a linguagem dos *applets* é simples, com uma API pequena, em comparação ao Java convencional, a aplicação de métodos formais é algo praticável e útil;
- Os contratos podem ser usados para especificar propriedades da API Java Card, como também checar propriedades dos *applets* ou entidades de domínio que usam a API.

O uso de JML em Java Card teve início com a especificação das próprias classes da API Java Card [31] [24], onde os métodos de cada classe foram anotados com JML. A validação da especificação foi feita com ferramentas de verificação estática que dão suporte a JML.

A especificação da API Java Card é de interesse de desenvolvedores de *applets*, pois define cada função de cada classe da API e qual é a responsabilidade de cada método. Contudo, descreve apenas o comportamento das várias interfaces Java Card. Dessa forma, não é possível garantir nenhuma verificação *runtime* de sua especificação. A modelagem de *applets* Java Card no contexto de verificação estática tem sido feita com uso de ferramentas como ESC/Java [22] e LOOP [42]. A ferramenta ESC/Java não notifica todos

os erros, apenas referências nulas de objetos e índices de arrays. O ESC/Java tem compatibilidade com a versão 1.4 de Java. Sua utilização é bastante simples. É necessário apenas executar o comando *escj* passando como parâmetro o arquivo *.java* que se deseja verificar. A versão do Esc/Java2 apresenta uma interface amigável com o usuário, não sendo necessário a execução em linhas de comando. O desenvolvimento desta ferramenta ainda está em evolução. A ferramenta LOOP não foi testada durante o desenvolvimento deste trabalho.

Ferramentas JML, como JMLRAC (*JML Runtime Assertion Checker*) para verificação em tempo de execução, não foram desenvolvidas com foco nas limitações Java Card.

Neste capítulo será feita uma análise da aplicação de especificação JML em Java Card. Para isso no Capítulo 5, mostraremos as características do compilador JML, o código gerado e sua aplicação na plataforma Java Card. Por fim, apresentaremos um subconjunto da linguagem JML compatível com a plataforma Java para cartões, e a especificação da estrutura de um compilador para esta variante de JML.

4.1 Compatibilidade da Linguagem JML com Java Card

Projeto por contrato pode ser usado em *applets* Java Card com o objetivo de especificar componentes que rodam no cartão. Contudo, desenvolvedores têm que levar em consideração o poder de processamento do cartão, linguagem de desenvolvimento utilizada e restrições de memória dos cartões inteligentes, onde os componentes de software serão instalados e executados.

Baseado no manual de referência JML [21] e na especificação da máquina virtual Java Card, é possível identificar propriedades na linguagem JML que não são compatíveis com Java Card. Tais incompatibilidades estão relacionadas principalmente a regras da gramática JML e ao código gerado por seu compilador.

Construções JML Não-Compatíveis com Java Card: A JML herda a estrutura da linguagem Java, utilizando as mesmas regras da gramática. A gramática JML define algumas regras que não podem ser aplicadas a Java Card, como por exemplo, na definição

de um não-terminal para representação dos tipos primitivos e referência a objetos. O não-terminal *java-literal* representa os literais Java para especificação JML. A definição deste não-terminal em JML é exatamente igual à linguagem Java, como mostra a regra a seguir.

java-literal ::= *integer-literal*
| *floating-point-literal* | *boolean-literal*
| *character-literal* | *string-literal*
| *null-literal*

Os não-terminais *floating-point-literal*, *character-literal* e *string-literal* não fazem parte da gramática Java Card. O não-terminal *integer-literal* , da mesma forma, não é totalmente suportado. Apenas os tipos *byte* e *short*, que representam valores inteiros com um tamanho menor, são utilizados. Os tipos *boolean-literal* e *null-literal* também são suportados em Java Card. Dessa forma, *java-literal*, definido na gramática JML, não é compatível com Java Card. Uma variante da gramática JML para representação de tipos Java Card pode ser definida como a gramática abaixo. A regra descrita pelo não-terminal *javacard-literal* não está definida na JML, contudo, sua aplicação tornaria possível a utilização de tipos compatíveis.

javacard-literal ::= *byte-literal*
| *short-literal* | *boolean-literal*
| *null-literal*

Em Java Card, objetos (instâncias de classe) e *arrays* unidimensionais são suportados. Os *arrays* podem conter os tipos primitivos definidos em *javacard-literal* e objetos. Uma representação da gramática para *arrays* pode ser feita a partir do não-terminal *type-spec*, como definido abaixo.

type-spec ::= *type [dims]*
type ::= *reference-type* | *built-in-type*
reference-type ::= *name*
built-in-type ::= *void* | *boolean* | *byte* | *short*
dims ::= **[ ]**

Outras construções JML não aplicáveis a linguagem Java Card são, por exemplo, as cláusulas *when, work_space e num_of* e operadores, *max, min, sum e product*. A cláusula

```
JMLAssertionError
   JMLPreconditionError
     JMLEntryPreconditionError
     JMLInternalPreconditionError
   JMLPostconditionError
     JMLNormalPostconditionError
     JMLExceptionalPostconditionError
   JMLIntraconditionError
     JMLAssertError
     JMLAssumeError
     JMLHenceByError
     JMLLoopInvariantError
     JMLLoopVariantError
     JMLUnreachableError
   JMLInvariantError
   JMLConstraintError
```

Figura 4.1: Hierarquia das classes de tratamento de erros em JML.

when é usada no tratamento de métodos concorrentes. Como Java Card não define tratamento de concorrência em sua API, esta cláusula não é utilizada na especificação. A cláusula *work_space* define o tamanho de uma *heap* de bytes de memória a ser utilizada por um método quando este é chamado. Como o retorno deste operador é do tipo *long*, a JCRE não tem como tratar o resultado. A principal restrição dos demais operadores também é em relação ao tipo utilizado, os operadores têm como retorno na função o tipo *long*. Uma construção compatível com Java Card poderia ser restrita ao uso de *byte* e *short* pelos operadores.

Tratando Exceções JML em Java Card: Java Card dá suporte a todo o mecanismo de exceção da plataforma Java. Qualquer implementação de classes Java Card para tratamento de exceções pode ser inserida no cartão.

JML define classes específicas para tratamento de erros de invariante, pré e pós-condições. A violação de alguma propriedade é direta ou indiretamente instância da classe *java.lang.Error* ou *java.lang.RuntimeException*. JML define uma hierarquia das classes para tratamento de erros em tempo de execução, como mostrado na Figura 4.1. Esta definição é compatível com Java Card.

Todas as classes para tratamento de erros JML obedecem estrutura semelhante da definida da Seção 2.1.2, para tratamento de exceções.

```
01 public class Applet{
02   /*@
03     @      requires bArray != null &&
04     @               bOffset > 0 && bLength > 0 &&
05     @               bOffset + bLength <= bArray.length;
06     @      ensures true;
07     @      signals (Exception) true;
08    @*/
09   public static void install(byte[] bArray, short bOffset,
10                              byte bLength) throws ISOException;
11
12   // Métodos Process e Register
13 }
```

Figura 4.2: Especificação da classe *Applet* [30].

4.1.1 Especificação da Classe *Applet* Java Card

A especificação da classe *Applet* tem como objetivo formalizar a principal classe do pacote *javacard.framework* no desenvolvimento de aplicações para cartões. Erik Poll em [30], anotou com JML cada método da classe *Applet*. Ao anotar cada método da classe *Applet* com especificação JML, sempre que um novo *applet* for ser desenvolvido, será possível observar as regras definidas nesta especificação. O novo *applet* poderá herdar a classe *Applet* com a especificação JML, contudo não terá a garantia de verificação em tempo de execução devido as restrições da tecnologia Java Card. Dessa forma, a especificação da classe *Applet*, feita em [30], serve de orientação para a criação de novas classes *applet*. A seguir será apresentada a especificação JML para os métodos da classe *Applet*.

Método *install* O método *install* (Figura 4.2) define uma pré-condição com 4 expressões, representadas pela cláusula requires. A pós-condição normal e excepcional são definidas, ambas, como *true*. O estado do parâmetro *bArray* (linha 1) deve ser diferente de nulo antes do método ser executado, *bOffset* e *bLength* (linhas 2 e 3) devem ser maiores que zero, e a soma de *bOffset* e *bLength* (4) deve ser menor ou igual ao tamanho do array *bArray*. Não existe nenhuma restrição para os estados após a execução do método. Para ambos, pós-condição normal e excepcional, não existe nenhuma representação de estado.

Método *process* O método *process* (Figura 4.3) tem apenas uma expressão para a pré-condição do método, na qual o parâmetro *apdu* não pode ter referência nula. Da mesma

```
01   /*@      requires apdu != null;
02     @      ensures  true;
03     @      signals  (ISOException) true;
04     @      signals  (Exception) true;
05     @*/
06   public abstract void process(APDU apdu)
07     throws ISOException;
```

Figura 4.3: Especificação do método *process*.

```
01  /*@      requires bArray != null &&
02    @               bOffset >= 0 && 5 <= bLength && bLength <= 16 &&
03    @               bOffset+bLength <= bArray.length;
04    @*/
05   protected final void register(byte[] bArray,
06                                 short bOffset, byte bLength)
07      throws SystemException,
08             NullPointerException,
09             ArrayIndexOutOfBoundsException,
10             TransactionException;
```

Figura 4.4: Especificação *lightweight* do método *register*.

forma que no método *install*, as pós-condições (*requires e signals*) são trivialmente satisfeitas. Ambas são definidas como *true*.

Método *register* A especificação para o método *register* (Figura 4.4) tem uma pré-condição com cinco expressões. As propriedades definidas na pré-condição devem ser verdadeiras no momento em que o *applet* é registrado. O método *install* determina que o parâmetro *bLength* deve ter tamanho entre 5 e 16.

É possível perceber que todos os parâmetros dos métodos da classe *Applet* são dos tipos *byte* ou *short*, um objeto (*apdu*) ou *array*. Esses tipos estão previstos no não-terminal *javacard-literal* proposto.

4.1.2 Especificando uma Entidade Java Card

Como exemplo de aplicação de especificação JML em entidades Java Card, se tem um *applet* Java Card para representação de uma carteira (*Wallet*). Neste exemplo é utilizado construções JML que são compatíveis com a gramática Java Card. Este exemplo será utilizado nas seções a seguir, com o objetivo de explicar o código de verificação gerado a partir da especificação JML. A Figura 4.5 apresenta as variáveis e construtor da

```
01 import javacard.framework.*;
02 public class Wallet extends Applet {
03
04  final static byte CREDIT = (byte) 0x30;
05  final static byte DEBIT = (byte) 0x40;
06  final static byte GET_BALANCE = (byte) 0x50;
07
08  final static short MAX_BALANCE = 0x7FFF;
09  short balance;
10
11  /*@ invariant balance >= 0 &&
12                balance <= MAX_BALANCE;
13    @*/
14  private Wallet (byte[] bArray,short bOffset,byte bLength){
15    pin = new OwnerPIN(PIN_TRY_LIMIT,   MAX_PIN_SIZE);
16    pin.update(bArray, bOffset, bLength);
17    register();
18  }
19  //Outros Métodos
20 }
```

Figura 4.5: Classe Wallet.

classe *Wallet*. A classe é anotada com o invariante JML, que tem 2 expressões atômicas. O valor da variável *balance* deve ser sempre maior que zero, e também deve ter sempre seu valor menor ou igual à constante *MAX_BALANCE*. As constantes *CREDIT, DEBIT, GET_BALANCE*, serão utilizadas para verificar qual instrução deve ser executada no método do *process*.

A classe *Wallet* implementa os métodos definidos na classe *Applet*, descrita na seção anterior. A especificação dos métodos de *Applet* pode ser herdada pelos desenvolvedores de aplicações Java Card.

No código da Figura 4.6, o método *process* verifica a operação a partir da instrução *INS* para executar um dos métodos privados (*credit, debit ou getBalance*). O valor de cada variável verificada na instrução *switch* é definida inicialmente como constante do *applet*, como pode ser visto na Figura 4.5.

A Figura 4.7 mostra a especificação e implementação do método *credit*. A pré-condição do método *credit* tem três (3) expressões que definem que, o objeto *apdu* não pode ser nulo, o valor a ser creditado (*creditAmount*) não pode ser maior que o máximo permitido por transação e o valor creditado somado ao saldo não pode exceder o saldo máximo permitido por cada carteira.

```
01 public void process(APDU apdu) {
02    byte[] buffer = apdu.getBuffer();
03    if (buffer[ISO7816.OFFSET_CLA] != Wallet_CLA)
04       ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
05
06    switch (buffer[ISO7816.OFFSET_INS]) {
07      case GET_BALANCE:   getBalance(apdu);
08                          return;
09      case DEBIT:         debit(apdu);
10                          return;
11      case CREDIT:        credit(apdu);
12                          return;
13      default:            ISOException.throwIt
14                         (ISO7816.SW_INS_NOT_SUPPORTED);
15    }
16 }
```

Figura 4.6: Método *process* - Wallet.

```
01 /*@ requires apdu != null
02    && ((byte[])apdu.getBuffer())
03          [ISO7816.OFFSET_CDATA] < MAX_TRANSACTION_AMOUNT
04    &&  balance + ((byte[])apdu.getBuffer())
05          [ISO7816.OFFSET_CDATA] <= MAX_BALANCE;
06   @*/
07 private void credit(APDU apdu) {
08
09    byte[] buffer = apdu.getBuffer();
10
11    byte creditAmount =
12                  buffer[ISO7816.OFFSET_CDATA];
13
14    balance = (short)(balance + creditAmount);
15 }
```

Figura 4.7: Método *credit* da classe *Wallet*.

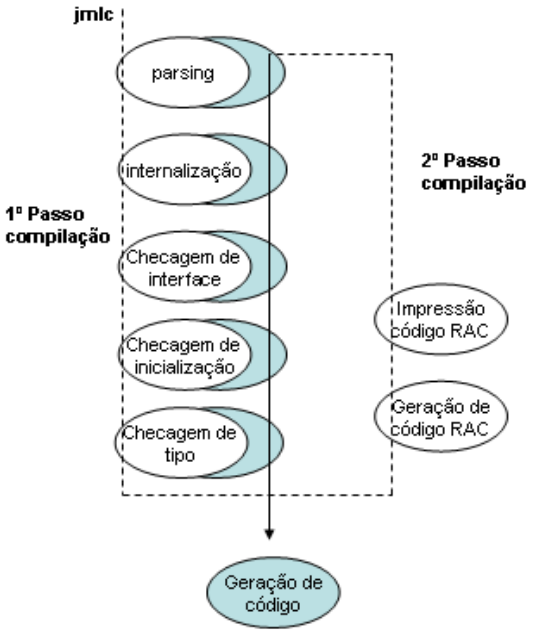

Figura 4.8: Passos de Compilação JML [9].

4.2 Verificação em Tempo de Execução da Especificação JML

O código para a verificação em tempo de execução de especificação JML é gerado pelo compilador da linguagem, o JMLc, que a partir da especificação, gera código de verificação para checar a validade da especificação em tempo de execução.

Uma particularidade da verificação em tempo de execução é que um erro é encontrado no momento em que a especificação do contrato é violada. O erro não é propagado. Isso significa uma vantagem no desenvolvimento de sistemas críticos, pois o erro é detectado e uma exceção é lançada.

O compilador JML usa uma abordagem de compilação em dois (2) passos, chamada *double-round compilation*. Um para o código da especificação e outro para o código original. O primeiro passo de compilação consiste de uma geração de código (*RAC - Runtime Assertion Checking*). Este passo gera código de acordo com o que foi definido na especificação JML. O código gerado é escrito em um arquivo temporário. No segundo passo de compilação, o novo código gerado é checado e compilado pelo compilador Java. O código da especificação é inserido no código Java original e é gerado o executável Java (*.class*). A Figura 4.8 mostra a estrutura dos passos de compilação do JML.

```
01 import java.io.PrintStream;
02 import java.lang.reflect.*;
03 import java.util.*;
04 import javacard.framework.*;
05 import org.jmlspecs.jmlrac.runtime.*;
06
07 public class Wallet extends Applet
08    implements JMLCheckable
09 {
10    private Wallet(byte abyte0[], short word0, byte byte0)
11    {
12        Block$();
13        . . .
14    }
15    . . .
16 }
```

Figura 4.9: Parte da classe *Wallet* decompilada.

Análise do Código Gerado a partir do JMLc Uma classe compilada com o JMLc sempre implementa a interface *JMLCheckable* e usa objetos de pacotes Java, com o *java.io, java.lang.reflect, java.util*, e objetos de pacotes JML como, *org.jmlspecs.jmlrac.runtime*.

Usando o exemplo da classe *Wallet* na Figura 4.5, após decompilar o executável gerado pelo JMLc (DJ Java Decompiler[1]), têm-se como resultado um código semelhante ao da Figura 4.9. O código consiste na definição da classe com a implementação da interface *JMLCheckable*, junto com os novos pacotes e a nova estrutura para o construtor da classe.

O compilador JML gera código verificável para as propriedades da classe, como o invariante, e também para cada especificação de método. O JMLc gera, ao:

- Seis (6) métodos para checagem de invariante e propriedades de instância da classe (*para toda classe*);
- Um (1) método para checar pré-condição (*para cada construtor ou método*);
- Um (1) método para checar pós-condição normal (*para cada construtor ou método*);
- Um (1) método para checar pós-condição excepcional (*para cada construtor ou método*) e;

[1]disponível em http://javatoolbox.com/authors/atanas-emilov-neshkov

- Cinco (5) métodos auxiliares (*para toda classe*).

Cada método tem a mesma estrutura para a definição de seus nomes, construídos de acordo com o tipo de especificação, a qual podem ser: *pre, post* ou *xpost*, respectivamente, o nome do método e o nome da classe.

Os métodos serão gerados independentes de existir o item de especificação correspondente. Por exemplo, para a pré-condição do método *credit*, são gerados, pelo compilador JML, três métodos para a verificação em tempo de execução. Os métodos são: *checkPre$credit$Wallet(APDU)*, *checkPost$credit$Wallet(APDU apdu, Object obj)* e *checkXPost$credit$Wallet(APDU apdu, Throwable throwable)*. Mesmo sem ter sido definido pós-condição normal ou excepcional, os métodos para tratamento desses tipos de especificação são gerados. Os métodos tem conteúdo vazio. Contudo, quando aplicados aos recursos de um cartão inteligente, podem ocasionar uma sobrecarga para a pouca memória existente.

A biblioteca usada pela JML na geração de código, principalmente o pacote JML *runtime*, não é suportado pela API Java Card. Outra incompatibilidade é em relação à quantidade de código gerado pelo compilador JML. A classe *Wallet*, inicialmente com 7 Kbytes de tamanho, torna-se um arquivo executável de 35 Kbytes. Considerando exemplos do mundo real, onde um arquivo com código fonte Java tem 15 Kbytes em média, a capacidade da memória EEPROM Java Card certamente será excedida.

O aumento no tamanho do arquivo se dá pelo fato de ser adicionado código após o primeiro passo de compilação. Os métodos de checagem de especificação são inseridos no código original. É possivel identificar, na Figura 4.10, a chamada aos métodos que checam o invariante, pré e pós-condições. O exemplo apresenta apenas uma parte das 119 linhas de código geradas para o método *credit*. O código original mais a especificação têm, como apresentado na Figura 4.7, 23 linhas de código. Dessa forma, houve um aumento de mais de 400% de tamanho no código do método *credit*, antes de ser compilado e após o uso do JMLc.

Além da quantidade excessiva de código gerado pelo JMLc, objetos não suportados pela plataforma Java Card também são utilizados. Um exemplo é o objeto *String* e o

```
01 private void credit(APDU apdu){
02    . . .
03    checkInv$instance$Wallet(
04         "credit@pre<File \"Wallet.java\", line 83, character 14>");
05            JMLChecker.exit();
06    }
07    if(JMLChecker.isActive(1))
08    {
09       checkPre$credit$Wallet(apdu);
10    }
11    try
12    {
13       internal$credit(apdu);
14       . . .
15       checkPost$credit$Wallet(apdu, null);
16       JMLChecker.exit();
17     }
18       . . .
19       checkXPost$credit$Wallet(apdu, throwable);
20       . . .
21 }
```

Figura 4.10: Método *credit* compilado com JMLc.

objeto *HashSet*, utilizados para geração de mensagens de erro e *warning* para o usuário do JML.

4.3 Subconjunto JML compatível com Java Card

Um requisito importante na definição de um subconjunto JML compatível com Java Card, é que esta variante deve prover tanto quanto possível as propriedades JML. Deve suportar os 2 tipos de especificação, tanto especificações *lightweight*, como especificações *heavy-weight*. Obedecer também aos diferentes níveis de verificação em tempo de execução. Semelhante ao JML e seu compilador, algumas especificações não são verificadas em tempo de execução, mas em tempo de compilação. Esse é o caso da cláusula *pure*. O usuário deve ser avisado que partes da sua especificação podem não ser checadas. Isso ocorre para algumas propriedades que podem ser custosas em um ambiente Java Card.

É importante que a seguinte estrutura JML seja preservada.

1. especificação de classes e interfaces: invariante de classe;
2. pré-condições, pós-condição normal e excepcional para os métodos;

3. condições *assignable* dos métodos (variáveis que podem ser modificadas pelos métodos;

4. operadores de implicação: $==>$;

5. tradução de expressões JML em código para a sintaxe Java: os quantificadores existencial e universal, \forall e \exists, e

6. propriedades JML referentes a definição de nível 0 [2] [21] JML, no manual de referência. As propriedades do nível 0 devem ser adaptadas para Java Card.

Gramática Com o objetivo de tornar JML compatível com Java Card, a primeira adaptação que deve ser feita é em relação à gramática. A gramática precisa ter como foco as restrições Java Card, retirando tudo aquilo que pertence à API Java, definida em JML, que não é reconhecido em Java Card. Neste sentido, as seguintes propriedades, que não são reconhecidas em Java Card, devem ser excluídas da especificação da gramática (Seção 2.1.2):

- Tipos primitivos Java: *int*, *long*, *double*, *float* and *char*;
- *arrays* multidimensionais;
- propriedades de concorrência;
- terminais e palavras-chave Java que não são suportadas pela plataforma Java Card;
- todos os itens da API *standard* Java que não fazem parte da API Java Card.

Compilador: O código gerado a partir de uma especificação, que segue as regras definidas na seção 4.2, deve ser otimizado, em relação ao compilador JML, ou então não será possível inserir e rodar este código em um cartão inteligente. Este é o principal requisito para o compilador. O código gerado não pode conter construções da linguagem que não são compatíveis com Java Card.

O compilador deve satisfazer as propriedades a seguir:

[2] O JML nível 0 deve ser suportado por todas as ferramentas JML e consiste na base da JML. As propriedades definidas no nível 0 são de fundamental importância para todos os usuários JML, para o entendimento da linguagem. Uma descrição detalhada da JML nível 0 pode ser encontrada em [21].

```
01 private void checkPre$Nome_Metodo$Nome_Classe(byte x)
02       throws RequiresException
03 {
04  if(!(x > 0))
05    RequiresException.throwIt(RequiresException.SW_REQUIRES_ERROR);
06 }
```

Figura 4.11: Condições para geração de código

- Não gerar métodos de verificação para especificações vazias. Por exemplo, se uma classe não contém especificação para o invariante, o método de checagem de invariante não deve ser gerado. Da mesma forma, o método *wrapper* não deve incluir a chamada a este método, onde há intenção de verificar a satisfação do invariante.

- Não gerar e chamar métodos de verificação para especificações definidas como *true*.

- Se a especificação a ser checada é expressa apenas em termos de expressões puramente Java, então, o código de verificação usado deve ser um condicional, negando a expressão. Por exemplo, uma pré-condição do tipo $x > 0$, na qual x é o parâmetro do método, o método a ser gerado é definido como na Figura 4.11.

- O usuário pode escolher o que deve ser gerado para verificação no cartão. Por exemplo, se apenas a pré-condição for escolhida, então, nenhum método de verificação para invariante ou pós-condição será gerado.

- Se a especificação não for satisfeita durante a execução, uma exceção, que herda da classe *CardRuntimeException*, é lançada.

4.4 Conclusão

Este capítulo analisou a aplicação de especificação JML em Java Card, mostrando incompatibilidades referentes à linguagem e ao compilador JML. O objetivo principal nesta análise foi mostrar que a estrutura atual do JML não é compatível com o ambiente Java Card no âmbito da verificação em tempo de execução. Isso ocorre pois componentes de código gerado pelo compilador JML não estão definidos no ambiente de execução Java Card.

Devido às restrições da JML e ao tipo de código gerado por seu compilador, se fez

necessário a definição de um subconjunto JML compatível com Java Card. Este subconjunto consiste, principalmente, em retirar da gramática JML, recursos Java que não são suportadas em Java Card. Foi descrito o que deve ser checado em tempo de execução, a partir da nova gramática, e como estas regras devem ser implementadas, pois a quantidade de código gerado pode gerar problemas relacionados a memória.

Por fim, é necessário otimizar o código gerado a partir de uma especificação JML. A descrição de uma nova linguagem, baseada na JML, e respectiva ferramenta de compilação foi a solução proposta para atingir o objetivo que é a aplicação de checagem *runtime* em cartões inteligentes. Nossa conclusão é que é possível prover verificação JML a *applets* Java Card em tempo de execução. Contudo é necessário retirar construções JML não compatíveis com Java Card. Da mesma forma, a geração de código do novo compilador deve ir ao encontro das restrições da tecnologia Java Card, sempre mantendo o significado da especificação.

Capítulo 5

JCML - Java Card Modeling Language

Este trabalho propõe a JCML (Java Card Modeling Language), uma linguagem de especificação de classes e interfaces Java Card. A linguagem foi definida a partir de um subconjunto JML. A partir de uma especificação JCML é possível gerar código compatível com as restrições da plataforma Java Card, para verificação em tempo de execução.

Como visto no capítulo 4, o compilador JML gera código que não pode ser inserido no cartão. Isto ocorre porque o arquivo gerado pelo compilador JMLc pode ultrapassar o limite de memória permitido pelo cartão e gerar, também, código que não é reconhecido pela plataforma Java Card. Com isso, além da definição de um subconjunto JML compatível com Java Card (JCML), é necessária a implementação de um compilador que dê suporte a esta nova linguagem.

Este capítulo irá apresentar a estrutura da linguagem JCML e seu respectivo compilador, o qual gera código de verificação em tempo de execução para especificações JCML.

5.1 Descrição da Gramática JCML

A gramática da linguagem JCML segue a estrutura da JML descrita em [21], manual de referência. Para definição das regras da gramática é usado o formalismo *EBNF* (*Extended Backus-Naur Form*) [32]. As regras JCML são adaptadas, quando necessário, às restrições Java Card. Cláusulas JML não compatíveis com Java Card são excluídas da linguagem JCML e o motivo de sua não utilização é apresentado. A estrutura, para o lado direito das regras da gramática JCML, é:

- $Prod$ - Exatamente uma ocorrência da construção *Prod*;
- $(Prod)*$ - Zero ou mais ocorrências da construção *Prod*;
- $(Prod)+$ - Uma ou mais ocorrências da construção *Prod*;
- $[Prod]$ - A construção *Prod* pode ocorrer ou não;
- $Prod_1 \mid Prod_2$ - Ou a construção $Prod_1$, ou a construção $Prod_2$ deve ocorrer;
- **term** - Terminal;

5.1.1 Unidade de Compilação

A unidade de compilação (*jcml-compilation-unit*) é a definição inicial da gramática JCML. Uma estrutura ou unidade de compilação em JCML é a mesma de Java Card, contudo, utiliza algumas cláusulas adicionais referentes à especificação.

Uma *jcml-compilation-unit*, como mostrado nas regras a seguir, é composta por, (1) zero ou uma definição de pacote (*package-definition*), (2) seguida por zero ou mais cláusulas import Java Card (*import-definition*) e, (3) zero ou mais definições de tipo (*type-definition*), que podem ser classes ou interfaces Java Card.

jcml-compilation-unit ::= *[package-definition]*
*(import-definition)**
*(type-definition)**

package-definition ::= **package** *name* **;**
import-definition ::= **import** *name-star* **;**

name ::= *ident (. ident)** **;**
name-star ::= *ident (. ident)* [.**]* **;**

Um código válido para uma unidade de compilação pode ser visualizado na Figura 5.1. O comentário da linha 7, *type-definition*, representa o não-terminal para declaração de classe ou interface Java Card, que será detalhado na seção 5.1.2.

```
01 package br.dimap.mestrado.jcml;
02
03 import javacard.framework.APDU;
04 import javacard.framework.Applet;
05 import javacard.security.*;
06
07 //type-definition
```

Figura 5.1: Código Válido: Unidade de Compilação.

Comparação com JML: Não foi utilizada a regra para refinamento (*refine-prefix*) e definição de alto nível (*top-level-definition*) para a unidade de compilação. Isto ocorre porque a linguagem JCML não define, refinamento de especificação. O não-terminal *top-level-definition* também não foi aplicado devido sua construção utilizar declarações da linguagem MultiJava. O JCML não utilizada o *parser* MultiJava, o qual é utilizado pelo compilador JML e descrito no manual de referência [21]. O JCML, por sua vez, é derivado do *parser* Java definido pela ferramenta JavaCC [17].

5.1.2 Definindo Tipos (Classes e Interfaces)

Uma definição de tipo é representada como segue, a qual pode ser uma definição de classe ou interface, ou então apenas a representação de vazio através do terminal ponto e vírgula (;).

type-definition ::= *class-definition* | *interface-definition* | **;**

class-definition ::= *[doc-comment] modifiers* **class** *ident*
[class-extends-clause] [implements-clause]
class-block

interface-definition ::= *[doc-comment] modifiers* **interface** *ident*
[interface-extends]
class-block

class-block ::= **{** *(field)** **}**

As definições de classe e interface são semelhantes. Em ambas, não pode existir comentário de especificação, apenas comentário Java ou JavaDoc. Uma diferença entre a definição de interface e classe Java Card é que, em uma interface é permitido apenas declaração de instâncias abstratas dos métodos (assinatura). Pode existir herança a nível

```
01 /**
02  * Representa um Usuário no sistema.
03  */
04 public class UsuarioApplet extends Applet
05 { /* field */ }
```

Figura 5.2: Código Válido: Tipos de Estrutura Java Card.

de interface, desse modo a cláusula *interface-extends* faz parte da estrutura da interface. A principal diferença entre os não-terminais *class-definition* e *interface-definition* é que, na definição de uma classe são utilizadas as regras *class-extends-clause* e *implements-clause*, enquanto que na definição de uma interface, uma ou várias interfaces podem ser herdadas, pelo não-terminal *interface-extends*. Uma interface não pode implementar outra interface. Um código válido para a definição de classe pode ser visualizado na Figura 5.2.

O não-terminal *field* descreve todos os membros (variáveis, construtores e métodos) da classe e interface. O bloco de uma classe tem como marcação inicial e final as chaves ({ .. }). O não-terminal *field* será detalhado na seção 5.1.3. A seguir serão apresentadas contruções que compõem o cabeçalho de uma classe ou interface.

class-extends-clause ::= **extends** *name*
implements-clause ::= **implements** *name-list*
name-list ::= *name (, name)**
interface-extends ::= **extends** *name-list*
modifiers ::= *(modifier)**
modifier ::= **public | protected | private**
| abstract | static
| final | *jcml-modifier*

Uma classe, utilizando JCML, pode implementar *N* interfaces. Toda classe pode herdar apenas de uma classe. Herança múltipla é feita utilizando interfaces e apenas uma declaração de herança. Os não-terminais *class-extends-clause, implements-clause e interface-extends* definem as regras de herança e implementação de interfaces.

O não-terminal *jcml-modifier* representa palavras-chave JCML a serem reconhecidas na declaração de uma especificação, como por exemplo, **spec_public, pure, helper** e **non_null**, descritas na seção 5.1.4.

A construção **spec_public** define que se uma determinada variável for privada, ela se torna pública para a especificação. A construção **pure** define que uma função não muda o

estado do objeto. A construção **helper** define que um determinado método pode quebrar o invariante. Este método deve ser privado e utilizado por um método público da classe. Por fim, a construção **non_null** define que um objeto não pode ser nulo. O modificador *non_null* quando definido em um parâmetro de método, representa que este não pode ser nulo (*null*). O tipo do parâmetro que tem a marcação do terminal *non_null* deve ser *reference-type* (Seção 5.1.3).

Comparação com JML: Palavras reservadas como **native, synchronized, transient** e **volatile**, são recursos não suportados em Java Card. Dessa forma, estes terminais não foram utilizados na construção do não-terminal *modifier* para JCML. A gramática JML utiliza os terminais citados como construções possíveis para *modifier*.

5.1.3 Membros de Classe - Declarando Classes e Interfaces

Na seção 5.1.2 foi descrita a estrutura inicial de um bloco para classes e interfaces Java Card. Os membros de classe representam toda a estrutura de uma classe Java Card, desde a declaração de variáveis globais e locais dos métodos até as cláusulas de especificação JCML. A construção *field* define: uma declaração de membros da classe - atributos e métodos (*member-decl*), ou uma declaração de especificação JCML (*jcml-declaration*) ou uma construção de inicialização para membros de classe (*class-initializer-decl*). A construção *field* é definida como segue.

field ::= *member-decl*
| *jcml-declaration*
| *class-initializer-decl*

class-initializer-decl ::= *[method-specification] [**static**] compound-statement*
| *method-specification*

Declarações de métodos, variáveis globais ou classes e interfaces aninhadas têm sua construção a partir do não-terminal *member-decl*, o qual define: uma declaração de método (*method-decl*), ou uma declaração de variável global (*variable-definition*), ou a construção de uma classe aninhada (*class-definition*) ou uma construção de interface aninhada (*interface-definition*). Uma classe aninhada é a definição de uma classe, dentro de outra classe, e sua construção é a apresentada na seção 5.1.2. A construção de um *member-decl* é feita de acordo com a gramática a seguir.

member-decl ::= *method-decl*
| *variable-definition*
| *class-definition*
| *interface-definition*

Uma declaração de método (*method-decl*) inicia por: comentários Javadoc ou comentários Java (*[doc-comment]*), pela definição de uma especificação para o método *jcml-method-specification*, opcionais. Os não-terminais *modifiers, type-spec, method-head e method-body* completam a representação de *method-decl*. Esses não-terminais definem o cabeçalho e corpo de um método Java Card. A construção de um *method-decl* é mostrada na regra a seguir.

method-decl ::= *[doc-comment]*
[jcml-method-specification]
[modifiers] [type-spec] method-head
method-body

Na regra *build-in-type* para JCML apenas os tipos *boolean, short, byte e void* foram definidos, pois são os tipos suportados pelo Java Card. Os demais não-terminais que definem a estrutura JCML estão descritos no Apêndice C deste trabalho. Os tipos primitivos em Java Card são representados pelo não-terminal *built-in-type*. Nele podem ser utilizados os tipos primitivos *boolean, byte e short*, além do identificador *void* que significa que o método não retorna valor. O tipo $\backslash TYPE$ representa todos os tipos Java Card. Este não-terminal é usado apenas em anotações JCML, diferente do não terminal *build-in-type* que pode ser utlizado no escopo da classe ou método.

type-spec ::= *type [dim]* | $\backslash$*TYPE [dims]*
type ::= *reference-type* | *built-in-type*
reference-type ::= *name*
built-in-type ::= **void** | **boolean** | **byte** | **short**
dim ::= **[ ]**

A Figura 5.3 mostra uma estrutura válida de classe Java Card. Entre as linhas 4-12 são definidas variáveis Java Card e seus respectivos comentários. Esta definição é definida na gramática pelo não-terminal *variable-definition*. Abaixo da declaração de variável, a classe *UsuarioApplet* define 2 métodos. O método *install* e o método *process*. Internamente a ambos os métodos, são definidos outros membros, como variáveis locais

```
01 public class UsuarioApplet extends Applet
02 {
03
04   public static final byte CLA_SMARTESCOLA = (byte) 0x80;
05   /** Instrução para verificar a senha. */
06   public static final byte INS_VERIFICAR_SENHA = (byte) 0x00;
07   /** Instrução para alterar o nome. */
08   public static final byte INS_SET_MATRICULA = (byte) 0x11;
09   /** Instrução para alterar o tipo do usuário. */
10  public static final byte INS_SET_TIPO = (byte) 0x12;
11  /** Instrução para recuperar o nome. */
12  public static final byte INS_GET_MATRICULA = (byte) 0x21;
13             .
14             .
15             .
16  public static void install(byte[] bArray, short bOffset,
17          byte bLength) {
18      new Usuario(bArray, bOffset, bLength);
19  }
20
21  //@ requires apdu != null;
22
23  public void process(APDU apdu) throws ISOException {
24      byte[] buffer = apdu.getBuffer();
25      byte cla = buffer[ISO7816.OFFSET_CLA];
26      byte ins = buffer[ISO7816.OFFSET_INS];
27
28      // Tratamento da classe do comando
29      if (cla != CLA_SMARTESCOLA) {
30          ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
31      }
32            .
33            .
34            .
35  }
36 }
```

Figura 5.3: Código Válido: Membros Java Card.

e estruturas de controle como *if*. O não-terminal *method-body* tem como definição outro não-terminal *compound-statement*. O *compound-statement* (Apêndice C) representa as estruturas Java Card (linhas 24-30).

As seções seguintes tratam da gramática da especificação JCML. Alguns dos não-terminais de especificação já foram declarados junto a definição das regras Java Card.

Comparação com JML: As regras para declaração de membros de classe em JCML, preservam ao máximo as regras definidas pela linguagem JML. A principal diferença é em relação a definição do tamanho do array, através da construção *dim*, o qual não pode ser um array multidimensional, pois Java Card não utiliza este tipo de construção. Outra diferença em relação entre a JCML e a JML é o não-terminal *built-in-type*, que define apenas os tipos primitivos permitidos em Java Card, excluindo as construções para *float, double* e *int*. Por fim, os não-terminais *jcml-declaration* e *jcml-method-specification* utilizam apenas construções compatíveis com Java Card para definição de especificação. Estas construções serão melhor detalhadas na seção 5.1.4.

5.1.4 Tipos de Especificação JCML

Uma declaração de especificação para Java Card (*jcml-declaration*) é uma das opções do não-terminal *field* (Seção 5.1.3). Um membro Java Card pode ser uma especificação JCML. O não-terminal *jcml-declaration* pode ser representado por um *invariant*, por uma *history-constraint* ou por uma variável de especificação através do não-terminal *represents-decl*. A cláusula invariant funciona como uma pré-condição e uma pós-condição incluída em todas as funções (incluindo o construtor Java Card) enquanto que *history-constraint* introduz uma relação entre os estados depois e antes da execução dos métodos. O valor de uma construção de modelo abstrato (*model*, Apêndice C) é determidado por um atributo da classe. Este relacionamento é especificado pela cláusula *represents*. A sintaxe para declaração de *invariant, constraint* e *represents* é mostrada a seguir.

jcml-declaration ::= *modifiers invariant*
| *modifiers history-constraint*
| *modifiers represents-decl*

invariant ::= *invariant-keyword predicate* **;**

invariant-keyword ::= **invariant**

history-constraint ::= *constraint-keyword predicate* **;**

constraint-keyword ::= **constraint**

represents-decl ::= *represents-keyword store-ref-expression*
l-arrow-or-eq spec-expression **;**
|
represents-keyword store-ref-expression **such_that*
predicate **;**

represents-keyword ::= **represents**
l-arrow-or-eq ::= **<-** | **=**

Os modificadores JCML *(jcml-modifier)* são reconhecidos apenas dentro de uma construção de especificação. Os terminais **spec_public** e **spec_protected** estão relacionados com a visibilidade da variável nas regras de especificação, **model** e **ghost** são terminais que definem variáveis de modelo, as construções **pure** e **helper** definem respectivamente que, o método não pode mudar o estado do objeto e que o método tem permissão de quebrar o invariante. O terminal **uninitialized**, define que a variável pode não ser inicializada, e por fim, as construções **non_null**, **nullable** e **nullable_by_default** estão relacionadas a referências nulas dos objetos.

jcml-modifier ::= **spec_public**
| **spec_protected**
| **model** | **ghost** | **pure**
| **helper** | **uninitialized**
| **non_null** | **nullable** | **nullable_by_default**

A Figura 5.4 mostra uma estrutura válida para especificação de invariante em JCML (linhas 7-12). Em cada declaração de variável, pode ser utilizada a construção **spec_public** (linhas 1-5), onde representa que as variáveis não publicas para especificação JCML.

Comparação com JML: Inicialmente foram definidas apenas as construções de tipos de especificação para os não-terminais *invariant, constraint* e *represents* em JCML. Em relação ao modificadores JCML, não são utilizados os modificadores relacionados a construções para tipos não permitidos em Java Card, por exemplo **spec_bigint_math** e outros descritos em [21].

```
01    private /*@ spec_public @*/ byte indiceLocal;
02    private /*@ spec_public @*/ byte[] locais;
03    private /*@ spec_public @*/ byte[] matricula;
04    private /*@ spec_public @*/ byte tipo;
05    private /*@ spec_public @*/ short creditos;
06
07    /*@ invariant indiceLocal >= 0 &&
08                    indiceLocal <= 10; @*/
09    /*@ invariant creditos >= 0 &&
10                  creditos <= VALOR_MAXIMO_CREDITOS; @*/
11    /*@ invariant tipo == TIPO_PROFESSOR ||
12                    tipo == TIPO_ESTUDANTE; @*/
```

Figura 5.4: Código Válido: Tipos de Especificação.

Na seção a seguir serão descritos os tipos de especificação de métodos para Java Card.

5.1.5 Especificação de Método JCML

Na Seção 5.1.3 o não-terminal *method-decl* define, como regra opcional na declaração do método, a especificação através do não-terminal *jcml-method-specification*. Antes do cabeçalho de cada definição de método Java Card é facultada a escolha do uso da uma especificação. Uma specificação pode ser herdada de uma superclasse. Para a definição de uma herança na especificação é utilizado o não-terminal *extending-specification*. O terminal **also** é definido para representar que a especificação é herdada.

jcml-method-specification ::= *specification* | *extending-specification*

extending-specification ::= **also** *specification*

Uma cláusula *jcml-method-specification* em um método ou interface deve iniciar com a palavra-chave **also**, se somente se, este método já foi declarado na superclasse. O não-terminal **also** representa que a especificação declarada é uma adição a alguma especificação do método que existe na superclasse, mesmo que a especificação seja *true*.

Um método também pode ter uma especificação local, que não é herdada. Esta declaração é feita pala regra *specification*, que também é utilizada pelo não-terminal *extending-specification*. Uma *specification* pode incluir um número indefinido de *spec-case* separados pela pala palavra **also**. Neste caso o terminal **also** significa uma conjunção e define que todas as cláusulas de especificação devem ser satisfeitas.

Uma especificação de método (*method-specification*) pode incluir *N* membros *spec-case* e deve satisfazer todas as propriedades especificadas.

specification ::= *spec-case (* **also** *spec-case)**

spec-case ::= *lightweight-spec-case* | *heavyweight-spec-case*

privacy ::= **public** | **protected** | **private**

Uma *spec-case* pode ser representada por uma *lightweight-spec-case*, definida como especificações leves, ou *heavyweight-spec-case* definida como especificação pesada.

5.1.5.1 Especificação Leve

Uma especificação *lightweight* é uma especificação que pode ser incompleta. Por padrão, uma cláusula de especificação omitida é representada pelo não terminal **not_specified**. Por exemplo, se em uma especificação *lightweight*, não é definida uma cláusula de pré-condição (não-terminal *requires-clause*), está omissão não afetará a verificação em tempo de execução. Se uma cláusula foi omitida, ela não será verificada. Isto ocorre apenas para as especificações leves.

JCML herda as caracteristicas de *specification privacy* da JML. A idéia de uma *specification privacy* é limitar a visibilidade da especificação. As regras de visibilidade de especificação são semelhantes as regras de visibilidade Java, onde se pode definir estruturas privadas, públicas e protegidas (*private, public e protected*).

lightweight-spec-case ::= *generic-spec-case*

generic-spec-case ::= *spec-header [generic-spec-body]*

generic-spec-body ::= *(simple-spec-body-clause)+*

spec-header ::= *(requires-clause)+*

simple-spec-body-clause ::= *diverges-clause*
| *assignable-clause*
| *ensures-clause*
| *signals-only-clause*
| *signals-clause*

requires-clause ::= *requires-keyword pred-or-not* **;**

```
01    /*@
02       requires m != null;
03       requires m.length <= TAMANHO_MAXIMO_MATRICULA;
04      @*/
05    public void setMatricula(byte[] m) throws ISOException {
06        if (m.length > TAMANHO_MAXIMO_MATRICULA) {
07            ISOException.throwIt(SW_TAMANHO_MATRICULA_INVALIDO);
08        }
09        matricula = m;
10    }
```

Figura 5.5: Código Válido: Especificação de Método.

Uma especificação leve é definida por um *spec-header*, que pode ser 1 (uma) ou várias definições de pré-condição (*requires-clause*), e uma ocorrência opcional do não-terminal *generic-spec-body*.

Uma especificação leve pode ser entendida como uma simplificação de uma especificação heavyweight, contudo a omissão de uma cláusula de especificação em uma especificação leve é tratada de forma diferente quando comparada uma uma especificação pesada.

O padrão para uma cláusula que é omitida em uma especificação leve é o terminal **not_specified**. Em relação à exceção, a omissão da cláusula *signals* tem como padrão apenas permitir exceções declaradas no cabeçalho do método. A não declaração de especificação também é considerada uma especificação leve.

O não-terminal *assignable-clause* determina quais variáveis podem ser modificadas no contexto do método. Uma omissão deste não terminal em uma especificação leve significa que qualquer variável pode sofrer modificação em seu estado.

As cláusulas *ensures e signals* são construções JCML para pós-condição normal e excepcional, respectivamente. Estas cláusulas definem estados desejados após a execução de um método. A Figura 5.5 mostra uma estrutura válida para especificação de método Java Card em JCML. As linhas 1-4 definem os predicados para a pré-condição do método.

Como o foco principal deste trabalho é o tratamento de alguns tipos de especificações leve para Java Card, as demais cláusulas na linguagem não são detalhadas nesta seção, contudo, todas as regra JCML estão definidas no Apêndice C. A próxima seção descreverá o compilador JCML, sua estrutura, tipo de código gerado e restrições.

Comparação com JML: Em relação à especificação de métodos Java Card, a gramática JCML preserva as construções da gramática JML. O uso do terminal *at* (@) no início de cada linha de especificação também é permitido. Entretanto, na versão inicial do compilador JCML, este terminal não é utilizado. É utilizado apenas no início e final da especificação (/*@ .. @*/) de múltiplas linhas, e em construção para linhas simples (//@). Contudo, não impede que essa construção seja implementada nas próximas iterações. A construção apresentada a seguir não foi desenvolvida na versão inicial do compilador devido a dificuldade de implementação.

```
/*@ requires E1 && E2 &&
  @          E3 && E4 &&
  @          E5 && E6;
  @*/
```

Na seção 5.2 será apresentada a estrutura do compilador JCMLc (JCML Compiler) para as regras da gramática descritas nesta seção.

5.2 JCMLc - JCML Compiler

O compilador JCML, apesar de ser bem mais restrito que o JML, está habilitado a gerar código compatível com Java Card para algumas especificações *lightweight*. Apesar da sintaxe definir construções de outros tipos, a versão inicial do compilador gera código apenas para construções de invariante e pré-condição. O compilador JCML não estende o compilador original da JML. Não é realizada análise semântica das construções JCML, é feita uma tradução das especificações em funções de verificação. A organização do código gerado pelo *JCMLc* preserva, tanto quanto possível, a estrutura do JMLc, onde são gerados métodos de verificação baseados a abordagem *wrapper* proposta em [9]. A abordagem *wrapper* transforma asserções de especificação em métodos Java Card de verificação. Cada asserção é verificada através da chamada de um método.

Nesta seção será apresentada a estrutura do compilador para JCML. Mostraremos a abordagem utilizada para tradução de especificação em código executável, cláusulas que são reconhecidas na versão atual do compilador e descrição das iterações para inclusão de novas cláusulas JCML.

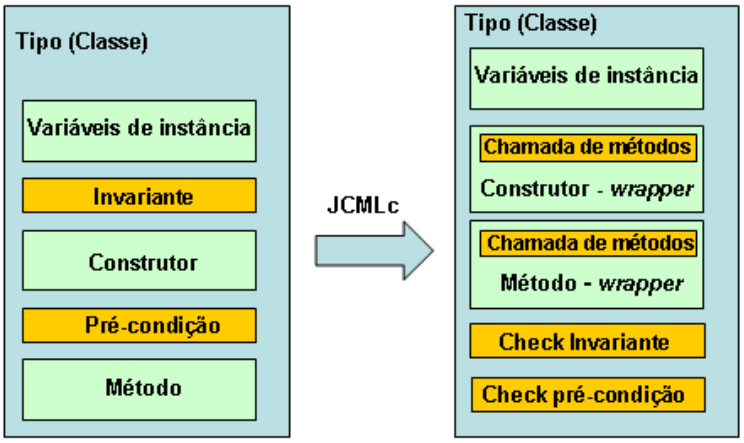

Figura 5.6: Estrutura para Geração de Código *wrapper*.

5.2.1 Abordagem *wrapper* para JCMLc

Seja uma classe com um conjunto de predicados $P_1..P_n$, os quais representam as propriedades da especificação, pré-condição, pós-condição ou invariante. Primeiro, as propriedades são transformadas em um único predicado, como por exemplo, $P \equiv P_1 \wedge P_2 \wedge P_3 \wedge P_n$, depois, são gerados os métodos de verificação. Os predicados de uma pré-condição serão transformados em métodos de verificação de pré-condição, o mesmo ocorrendo para as demais construções. As contruções para invariante geram métodos de verificação de invariante. Os predicados de construções diferentes geram métodos de verificação diferentes.

Semelhante ao JMLc, um método *wrapper* tem o mesmo nome e assinatura do método original. O método *wrapper* chama os métodos para checar o invariante ou pré-condição e depois executa o método original. A Figura 5.6 apresenta a estrutura da classe após a compilação do código com JCMLc. Os predicados da especificação são transformados em código Java Card e inseridos no código original. São gerados métodos de verificação e os respectivos comandos para execução desses métodos. Como a abordagem utilizada pelo compilador JML é uma abordagem na qual são criadas funções de verificação, não foi utilizada a abordagem de geração de código e verificação *in-line*.

Um método verificador de pré-condição checa as propriedades definidas na especificação. Caso uma das propriedades de pré-condição seja violada, uma exceção de pré-condição é lançada. Seja m um método declarado na classe C com assinatura T_0 m($T_1 a_1$, $T_2 a_2$, .. , $T_n a_n$), assume-se que m pode lançar uma exceção do tipo *RequiresException*,

```
01  private void checkInv$Nome_Classe$() {
02    if (!(P1, P2, . . ., Pn)) {
03         InvariantException.throwIt(InvariantException.SW_INVARIANT_ERROR);
04    }
05  }
06
07  private void checkPre$Nome_Metodo$Nome_Classe(T1 a1, . . ., Tn an) {
08    if (!(P1, . . ., Pn)) {
09         RequiresException.throwIt(RequiresException.SW_REQUIRES_ERROR);
10    }
11 }
```

Figura 5.7: Estrutura dos Métodos de verificação para Java Card.

caso uma das propriedades seja violada. O método de verificação de pré-condição para m é o método *checkPre$m$C*, com $T_1 a_1$, $T_2 a_2$, .., $T_n a_n$ parâmetros, os mesmos parâmetros definidos em m. São geradas notações únicas para os métodos de verificação, de acordo com a assinatura de cada método. Métodos para verificação de invariante são semelhantes, em estrutura, aos métodos para checagem de pré-condições.

Na Figura 5.7 apresentamos a estrutura para os métodos de verificação JCML. O compilador traduz as especificações JCML em métodos, seja referente a invariante, ou pré-condição. Inicialmente é feita a negação dos predicados da especificação, e a partir de cada declaração, com P_n predicados, são geradas condições de verificação (*if*) para checar a validade das expressões, por exemplo o invariante na Figura 5.7, se o predicado $P_1..P_n$ for verdadeiro (*true*), sua negação (!(*true*)), faz com que a verificação continue, não executando a exceção na cláusula *if* (linhas 3). Caso contrário, se o predicado $P_1..P_n$ for falso (*false*), sua negação (!(*false*)) faz com que o condicional seja verdadeiro, fazendo com que a exceção *InvariantException* seja lançada. O método *checkPre$Nome_Metodo$Nome_Classe(*T_1 a_1, . . ., T_n a_n*)* (linhas 6-9) tem uma estrutura de verificação similar, considerando declarações para pré-condição sob os parâmetros $a_1, a_2, .., a_n$ e predicados correspondentes em P_1 ... P_n.

Na seção a seguir será apresentada a estrutura de tradução de cláusulas JCML em código de verificação Java Card.

5.2.2 Traduzindo Cláusulas JCML

Nesta seção apresentaremos a sequência para a tradução de especificações JCML em código Java Card, e como é feita a transformação de cada tipo de especificação em código de verificação *runtime*.

Como exemplo para tradução de regras JCML em código Java Card será utilizado a classe de domínio descrita no Capítulo 2 e Apêndices A e B. A classe implementa informações de um usuário, as quais podem ser, tipo, matrícula, locais de acesso autorizados, e controle de crédito que podem ser utilizados pelo usuário do cartão.

Estrutura de Arquivos: Após compilar uma classe (*Classe.java*) com JCMLc, um novo arquivo é criado com o código original, mais o código Java Card gerado a partir da especificação. O nome do novo arquivo é uma junção do nome inicial da classe, mais JCML (*ClasseJCML.java*). A partir da nova classe (*ClasseJCML.java*), o executável Java (*ClasseJCML.class*) é gerado com o compilador Java Card. A verificação de contexto de variáveis é feita pelo compilador Java após a geração do arquivo *ClasseJCML.java*. Esta prevista a análise semântica e verificação de contexto em iterações futuras do compilador.

Na Figura 5.8 é apresentada a sequência de compilação. A partir de uma classe Java Card, o compilador JCML verifica o que é código de especificação, e o que é código Java Card. Para cada cláusula de especificação é gerado um método de verificação correspondente, e este é inserido no código original. Caso não exista nenhuma anotação JCML, o código final é igual ao código original, pois, como a especificação é vazia, não há necessidade de gerar código para verificação.

Definição de Exceções: Como cada método de verificação gerado pode lançar uma exceção, como mostrado na Figura 5.7, se fez necessário a criação de classes para representação de cada uma das exceções. As exceções herdam da classe *CardRuntimeException* do pacote *javacard.framework*, e seguem a mesma estrutura da classe *ISOException*, utilizada frequentemente em aplicações Java Card. As classes para tratamento de erros de especificação JCML são: *InvariantException* e *RequiresException*, como mostram as Figuras 5.9 e 5.10.

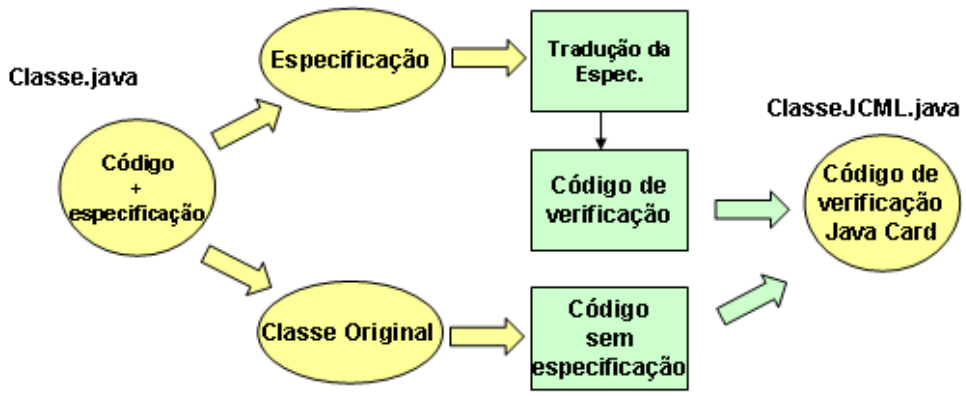

Figura 5.8: Estrutura de Compilação.

```
01 import javacard.framework.CardRuntimeException;
02
03 public class InvariantException extends CardRuntimeException{
04
05    public static final short SW_INVARIANT_ERROR = (short) 0x6377 ;
06    public InvariantException(short word){
07        super(word);
08    }
09
10    public static void throwIt(short word){
11        throw new InvariantException(word);
12    }
13 }
```

Figura 5.9: Classe *InvarianteException*.

```
01 import javacard.framework.CardRuntimeException;
02
03 public class RequiresException extends CardRuntimeException{
04
05    public static final short SW_REQUIRES_ERROR = (short) 0x6388 ;
06    public RequiresException(short word){
07        super(word);
08    }
09
10    public static void throwIt(short word){
11        throw new RequiresException(word);
12    }
13 }
```

Figura 5.10: Classe *RequiresException*.

Cada classe implementa o método *throwIt* e um construtor que recebe como parâmetro uma variável do tipo *short*, que tem o objetivo de identificar o tipo de exceção que foi lançada. Caso o valor do parâmetro seja a constante *0x6377* , ocorreu uma violação de invariante, caso o valor seja *0x6388*, ocorreu um erro de pré-condição.

As classes para tratamento de exceção de invariante e pré-condição foram criadas com o objetivo de auxiliar o código de verificação gerado pelo JCMLc. Cada classe deve ser inserida no cartão, para que as exceções sejam reconhecidas durante a execução das aplicações Java Card.

Invariante: Inicialmente, serão apresentados exemplos para tratamento de propriedades de invariante, dessa forma, apenas predicados deste tipo serão descritos. Logo após, propriedades para as pré-condições serão acrescidas na especificação dos métodos.

A especificação da classe *Usuario* tem predicados sobre as variáveis de instância da classe. As expressões de invariante definem que: a variável *indiceLocal* deve ser maior ou igual a 0 e menor ou igual a 10, o valor da variável *creditos* deve ser menor ou igual à constante *VALOR_MAXIMO_CREDITOS* e também maior ou igual a zero, e o valor do tipo de cada usuário deve ser *TIPO_ESTUDANTE* ou *TIPO_PROFESSOR*, estudante ou professor, respectivamente.

As propriedades do invariante valem para todos os métodos definidos na classe. A Figura 5.11 mostra a especificação do invariante da classe, variáveis globais e interface dos métodos. A partir desta especificação, o compilador JCMLc irá gerar um método

```
01 public class Usuario {
02
03    //CONSTANTES
04
05    private byte indiceLocal;
06    private byte[] locais;
07    private byte[] matricula;
08    private byte tipo;
09    private short creditos;
10
11    /*@ invariant indiceLocal >= 0 && indiceLocal <= 10; @*/
12    /*@ invariant creditos >= 0 && creditos <= VALOR_MAXIMO_CREDITOS; @*/
13    /*@ invariant tipo == TIPO_PROFESSOR || tipo == TIPO_ESTUDANTE; @*/
14
15    public Usuario();
16
17    public void setMatricula(byte[] m) throws ISOException;
18    public byte[] getMatricula();
19    public void setTipo(byte t) throws ISOException;
20    public byte getTipo();
21    public void adicionarLocal(byte codigo_local) throws ISOException;
22    public void removerLocal(byte codigo_local) throws ISOException;
23    public boolean temAcessoLocal(byte codigo_local);
24    public void adicionarCreditos(short valor);
25    public void removerCreditos(short valor);
26    public short getCreditos();
27
28 }
```

Figura 5.11: Invariante da Classe Usuario.

de verificação, *checkInv$Usuario$()*, e realizará a chamada a esta função em todos os métodos da classe.

Para cada cláusula de invariante, é criado um condicional, o qual verifica o valor do estado de cada variável durante a chamada de qualquer método da classe *Usuario*. As linhas 6-7, 9-10 e 12-13 do código da Figura 5.12 são consequência da especificação do invariante feita nas linhas 11, 12 e 13 da Figura 5.11, após serem compilados com JCMLc. Na criação do método de verificação, o compilador JCMLc nega os predicados definidos para o invariante, com o objetivo de saber se uma das propriedades foi quebrada. Se o predicado de algum dos condicionais for verdade, significa que o invariante da classe não foi satisfeito, lançando assim uma exceção do tipo *InvariantException*.

A checagem de invariante é feita no início e no final de cada método, dentro de um bloco para tratamento de exceção, devido à possibilidade de quebra do invariante. Como a chamada ao método *checkInv$Usuario$* pode lançar uma *InvarianteException*, é aplicado o bloco *try .. catch* na estrutura do método de verificação. O código do método original é

```
01 public class Usuario {
02
03   // Variáveis, Construtor e Métodos;
04
05   private void checkInv$Usuario$() throws InvariantException {
06     if (!(indiceLocal >= 0 && indiceLocal <= 10 ))
07        InvariantException.throwIt(InvariantException.SW_INVARIANT_ERROR);
08
09     if (!(creditos >= 0 && creditos <= VALOR_MAXIMO_CREDITOS ))
10        InvariantException.throwIt(InvariantException.SW_INVARIANT_ERROR);
11
12     if (!(tipo == TIPO_PROFESSOR || tipo == TIPO_ESTUDANTE ))
13         InvariantException.throwIt(InvariantException.SW_INVARIANT_ERROR);
14     }
15 }
```

Figura 5.12: Método *checkInv$Usuario$* para Verificação do Invariante.

```
01 public void adicionarLocal(byte codigo_local) throws ISOException {
02   try{
03        checkInv$Usuario$();
04
05        //Código do Método Original
06        if (indiceLocal >= QUANTIDADE_MAXIMA_LOCAIS ) {
07            ISOException.throwIt (SW_QUANTIDADE_MAXIMA_LOCAIS_EXCEDIDA ) ;
08        }
09        else if (! this.temAcessoLocal (codigo_local ) ) {
10           locais [indiceLocal ]= codigo_local ;
11           indiceLocal ++ ;
12        }
13        checkInv$Usuario$();
14  }catch (InvariantException invEx) {
15       InvariantException.throwIt(InvariantException.SW_INVARIANT_ERROR);
16   }
17 }
```

Figura 5.13: Método *adicionarLocal* com Checagem de Invariante.

inserido após a chamada do método de checagem de invariante e antes do final do bloco *try*. Caso o invariante seja violado, o método original não será executado, pois executará o bloco *catch*. O método *adicionarLocal* (Apêndice A) apresentado na Figura 5.13 tem a sua estrutura modificada após a compilação da classe Usuário com o JCMLc. Este método antes de ser compilado com JCMLc tinha como implementação apenas o código entre as linhas 6-12.

Uma forma alternativa de geração de código de verificação é criar um método para execução do método original. O novo método contém o código original, contudo tem seu nome modificado, com o objetivo de não causar duplicidade de métodos. Por exemplo, a chamada ao método protegido de *adicionarLocal* (linha 5 Figura 5.14). Sua assi-

```
01 public void adicionarLocal(byte codigo_local) throws ISOException {
02   try{
03        checkInv$Usuario$();
04
05        internal$adicionarLocal(codigo_local);
06        checkInv$Usuario$();
07  }catch (InvariantException invEx) {
08       InvariantException.throwIt(InvariantException.SW_INVARIANT_ERROR);
09   }
10 }
```

Figura 5.14: Método *adicionarLocal* Protegido.

natura é modificada para *metodoOriginal$adicionarLocal$(codigo_local)* é o seu conteúdo é o mesmo apresentado nas linhas 5-12 da Figura 5.13, que é o código original do método. O compilador original JML renomeia o método original, o chamando de *internal$adicionarLocal(codigo_local)*. Este método é chamado por um novo método que tem o mesmo nome do método original. Este método faz a chama aos métodos de verificação de invariante e pré-condição, e também ao método *internal$adicionarLocal()*, como mostra a Figura 5.14. O compilador JCML não cria o método interno com o código origianl. Os métodos de verificação são inseridos no método original, como mostra a Figura 5.15.

Todos os 10 métodos da classe *Usuario* terão a mesma estrutura modificada após serem compilados com JCMLc, como mostrado na Figura 5.15. Sua nova estrutura corresponderá a:

- Assinatura do método;
- Inicio do bloco *try*;
- Chamada ao método de verificação de invariante;
- Código do método original;
- Chamada ao método de verificação de invariante;
- Final do bloco *try*;
- Bloco *catch*;
- Final do método.

```
01 public void Nome_Metodo(){
02   try{
03        checkInv$Usuario$();
04
05        //CÓDIGO ORIGINAL DO MÉTODO
06        checkInv$Usuario$();
07   }catch (InvariantException invEx) {
08       InvariantException.throwIt(InvariantException.SW_INVARIANT_ERROR);
09   }
10}
```

Figura 5.15: Estrutura de um Método após Compilação com JCMLc.

```
01  /*@
02    requires valor >= 0;
03    requires valor + getCreditos() <= VALOR_MAXIMO_CREDITOS;
04  @*/
05  public void adicionarCreditos(short valor) {
06     creditos += valor;
07 }
```

Figura 5.16: Especificação do Método *adicionarCreditos* .

Pré-Condição: A geração de código de verificação para pré-condição é feita, pelo JCMLc, de forma semelhante à geração de código para o invariante. Como no invariante as variáveis a serem checadas são variáveis globais, não existe necessidade do método de verificação conter parâmetros, pois todos os métodos da classe têm acesso às variáveis da classe. Contudo, para as pré-condições, os valores a serem checados incluem as variáveis passadas como parâmetro de cada método. Dessa forma, para a geração do método de verificação de pré-condição, é necessário que os mesmos parâmetros do método original sejam os parâmetros do método de verificação. Por exemplo, se o método m contém a_1, a_2 .. a_n como parâmetros, o método de verificação *checkPre$m$T*, terá os mesmos parâmetros a_1, a_2 .. a_n definidos em m.

A especificação do método *adicionarCreditos* (Figura 5.16) define como pré-condição que o valor da variável *valor* do tipo *short*, passada como parâmetro, deve ser maior ou igual a 0 e que a soma de *valor* com a quantidade de creditos atual, fornecida pelo método *getCreditos* (Figura 5.17), não pode ser maior que a constante *VALOR_MAXIMO_CREDITOS*. A variável *valor* é adicionada a *creditos*, na implementação do método original.

Após compilar a classe *Usuario* com a especificação das pré-condições do método *adicionarCreditos*, é gerado o método de verificação *check-*

```
01 public short getCreditos() {
02     return creditos;
03 }
```

Figura 5.17: Método *getCreditos*.

```
01 private void checkPre$adicionarCreditos$Usuario(short valor)
02                          throws RequiresException{
03   if(!(valor >= 0))
04       RequiresException.throwIt(RequiresException.SW_REQUIRES_ERROR);
05   if(!(valor + getCreditos () <= VALOR_MAXIMO_CREDITOS ))
06        RequiresException.throwIt(RequiresException.SW_REQUIRES_ERROR);
07 }
```

Figura 5.18: Método *checkPre$adicionarCreditos$Usuario*.

Pre$adicionarCreditos$Usuario, o qual tem o mesmo parâmetro *valor* do tipo short. Este método verifica se os predicados definidos como pré-condição são satisfeitos antes da execução do método. A Figura 5.18 apresenta o método de verificação para a pré-condição de *adicionarCreditos*.

O método original *adicionarCreditos* também tem sua estrutura modificada após o uso do JCMLc, semelhante ao método *adicionarLocal*. Além da checagem do invariante, descrita anteriormente, é feita a checagem da pré-condição do método, através da chamada do método *checkPre$adicionarCreditos$Usuario*. O método *wrapper* chama o método para checar o invariante e pré-condição (linhas 3 e 4), antes da execução do código original. Como foi adicionada a verificação de pré-condição ao método, foi gerado um bloco *catch* para tratamento da exceções do tipo *RequiresException* (linhas 10-12 da Figura 5.19), caso alguma propriedade de pré-condição não seja satisfeita. O código original do método *adicionarCreditos*, tinha 7 linhas de código. Após ser compilado com o JCMLc este método passou a ter 13 linhas de código, incluindo espaçamentos para legibilidade.

Toda a especificação da classe *Usuario*, invariante e pré-condição, pode ser vista no Apêndice B. Da mesma forma, a classe *UsuarioJCML*, produto da compilação da classe *Usuario* com o JCMLc, com todos os métodos de verificação gerados, também pode ser encontrada no Apêndice B.

```
01 public void adicionarCreditos ( short valor) {
02    try{
03       checkInv$Usuario$();
04       checkPre$adicionarCreditos$Usuario(valor) ;
05
06       creditos += valor ;
07
08     }catch (InvariantException invEx) {
09       InvariantException.throwIt(InvariantException.SW_INVARIANT_ERROR);
10     }catch (RequiresException reqEx) {
11        RequiresException.throwIt(RequiresException.SW_REQUIRES_ERROR);
12     }
13  }
```

Figura 5.19: Método *adicionarCreditos* após Compilação.

```
01 void CheckMetodo() Exception{
02     if (Exp1 &&  !Exp2)
03          Exception.throwIt(Exception.ERROR);
04 }
```

Figura 5.20: Checagem de Expressões com Implicação.

5.3 Suporte a Operadores não Definidos em Java Card

Nesta seção apresentaremos o modelo de tradução para construções JCML que não são diretamente suportadas pela plataforma Java Card, na tradução de especificação JCML para código *runtime* com do compilador JCML.

5.3.1 Implicação e Equivalência

Dada uma expressão de implicação Exp_1 ==> Exp_2, a qual é verdade (*true*) se a expressão booleana Exp_1 for falsa ou, a expressão booleana Exp_2 for verdadeira. Podemos traduzir uma expressão que contenha esse tipo cláusula em código Java Card, usando o método de verificação descrito na Figura 5.20. Se !Exp_2 e Exp_1, o resultado será verdade, sendo necessário lançar uma exceção. Da mesma forma, em uma implicação reversa Exp_2 <== Exp_1, deve ser traduzida com o mesmo método descrito na Figura 5.20, pelo fato de ter o mesmo significado.

Os operadores de equivalência e negação de equivalência (<==> *e* <=*!*=>), na qual, tem o mesmo significado de == e *!*=, respectivamente, podem ser diretamente traduzidos em código Java Card.

```
01 void CheckMetodo_Forall() Exception{
02  for (short v0 = 1; v0 < value; v0++) {
03    for (short v1 = v0 - 1; v1 < v0; v1++) {
04       . . .
05       for (short v_n = v_n - v_(n-1); vn < v(n-1); vn++) {
06          if (!(spec-expression))
07                Exception.throwIt(Exception.ERROR);
08       }
09    }
10  }
11 }
```

Figura 5.21: Checagem de Expressões com Cláusula *forall*.

5.3.2 Quantificadores

A especificação de uma expressão com quantificador universal é dada a partir da seguinte estrutura gramatical:

spec-quantified-expr ::= *(quantifier quantified-var-decls* **;**
[[predicate] ;] spec-expression)

onde o predicado (*predicate*) apresenta o domínio, onde os valores da expressão, definida pelo não-terminal *spec-expression*, deve aplicar a verificação da variável declarada no não-terminal *quantified-var-decls*. Considere a expressão abaixo:

```
\forall short i,j; 0 <= i && i < j && j < 10; a[i] < a[j]
```

a qual, usa o operador de quantificador universal e especifica que o vetor *a* é ordenado de forma crescente. De acordo com o manual JML, caso haja a ausência do domínio (*[predicate]*), a expressão deve ser verificada no domínio da variável declarada através do *quantified-var-decls*.

O modelo de tradução de cláusula *forall* em método de verificação Java Card tem a estrutura descrita da Figura 5.21. A expessão pode ser traduzida em um operador de laço Java, *for*, na qual será feita a checagem da expressão definida em *spec-expression*, de acordo com o domínio de verificação. Caso a negação da expressão seja verdade, uma exceção será lançada.

O operador *forall* pode verificar a expressão definida pelo não-terminal *spec-expression N* vezes. Apesar desta verificação poder ser exponencial, o número de iterações pode ser reduzido com a definição do domínio, representado pelo não-terminal

predicate. Como em JCML apenas os tipos *byte* e *short* são suportados como domínio, o espaço de busca pode ser reduzido. O operador existencial pode ser traduzido para Java Card de maneira similar ao operador universal.

Outras construções JCML que não são diretamente traduzidas para Java Card, como por exemplo *assignable, pure, not_specified, etc.* não estavam no escopo inicial deste trabalho, contudo, tais construções serão produto das próximas iterações de desenvolvimento do compilador.

Na seção 5.4 será feita uma análise entre o código gerado pelo compilador JCML e o código gerado pelo compilador JML.

5.4 Análise do Código Gerado pelo Compilador JCML

Como descrito no Capítulo 4, o tipo e a quantidade de código gerado pelo compilador JML, faz com que não seja possível aplicar verificação de JML em Java Card.

Esta seção tem por objetivo mostrar um comparativo entre o código gerado pelos compiladores JML e JCML. A análise se dará em relação a quantidade de linhas de código gerados para cada método, e também, quantos métodos de verificação são criados.

A Tabela 5.1 apresenta, em *Lines of Code* (LOC), a quantidade de código gerado para cada método. Cada linha da tabela descreve o nome do método da classe Usuário, o tamanho original, um identificador (sim ou não) que representa se este método tem especificação para pré-condição, tamanho do método após compilar com JCMLc e tamanho do método após compilar com JMLc. É possível notar na Tabela 5.1 que para cada método compilado com JMLc, é gerado uma média de 119 linhas de código. Usando o JCMLc, se o método não for anotado com especificação, este permanecesse sem modificação, tendo a mesma quantidade de linhas de código após a compilação. Como exemplos há os métodos: *getMatricula, getTipo e getCreditos* . Como consequência do uso do compilador JML, o tamanho total da classe *Usuario* compilada com JMLc é de 3692 linhas de código, enquanto que usando o compilador JCML, o tamanho total, em linhas de código, é de 283.

A Tabela 5.2 apresenta, em porcentagem, o aumento da quantidade de código gerado

Método	Tam. Original (LOC)	Tem Espec.?	JCMLc (LOC)	JMLc (LOC)
setMatricula	10	Sim	19	119
getMatricula	3	Não	3	113
setTipo	10	Sim	18	119
getTipo	3	Não	3	113
adicionarLocal	12	Sim	22	120
removerLocal	14	Sim	25	119
temAcessoLocal	11	Sim	22	117
adicionarCreditos	7	Sim	16	118
removerCreditos	7	Sim	16	118
getCreditos	3	Não	3	113
Classe Total	134	Sim	283	3692

Tabela 5.1: Análise da Classe *Usuario* em Linhas de Código - Diferença entre JCMLc e JMLc .

para cada método. Cada linha da tabela descreve o nome do método da classe Usuário, o tamanho original, um identificador (sim ou não) que representa se este método tem especificação para pré-condição, o aumento da quantidade de linhas após compilar com JCMLc e o aumento da quantidade de linha após compilar com JMLc. Apesar do compilador JCML apesentar, em alguns casos, um aumento de mais de 100% em relação ao arquivo original, este aumento é justificável quando comparado ao compilador JML que apresenta, na maioria dos casos um aumento de mais de 1000%. A aplicação de especificação JCML, apesar do aumento percentual elevado, é suficiente para o desenvolvimento Java Card. O aumento total em JCML é de 111,19%, enquanto o aumento em JML é de 2655,22%.

Método	Tam. Original (LOC)	Tem Espec.?	JCMLc (%)	JMLc (%)
setMatricula	10	Sim	90%	1090%
getMatricula	3	Não	0%	3666%
setTipo	10	Sim	80%	1090%
getTipo	3	Não	0%	3666%
adicionarLocal	12	Sim	83%	900%
removerLocal	14	Sim	78,57%	750%
temAcessoLocal	11	Sim	100%	963,63%
adicionarCreditos	7	Sim	128,57%	1585,71%
removerCreditos	7	Sim	128,57%	1585,71%
getCreditos	3	Não	0%	3666%
Classe Total	134	Sim	111,19%	2655,22%

Tabela 5.2: Análise do Aumento de Linhas de Código da Classe *Usuario* (em %) - Diferença entre JCMLc e JMLc .

Em relação ao tamalho do arquivo, o executável gerado a partir do JMLc tem 37

Kbytes de tamanho, enquanto o gerado pelo JCMLc tem 5 Kbytes. O arquivo original com 5 Kbytes de tamanho, após ser compilado com o compilador JCML passa a ter 8 Kbytes (arquivo *.java*). O arquivo gerado pelo JCMLc (*UsuarioJCML.java*) ao ser compilado com Java Card produz um executável com 5 Kbytes de tamanho (*.class*), enquanto que o arquivo original, sem especificação, tem 3 Kbytes após ser compilador com Java Card, um aumento de 66%. Ao se utilizar o compilador JML, este produz um executável de 37 Kbytes, 1133,33% de aumento em relação ao arquivo original. Um aumento considerável em relação as restrições Java Card. A Tabela 5.3 mostra a diferença entre os tamanhos dos arquivos.

Tipo de Arquivo	**Java Card**	**JCML**	**JML**
.java	5 Kb	8 Kb	5 Kb
.class	3 Kb	5 Kb	37 Kb

Tabela 5.3: Análise do Tamanho do Arquivo Gerado pelo Compilador - Diferença entre JCMLc e JMLc .

Outra comparação possível, entre JCML e JML, é a quantidade de métodos gerados a partir de cada especificação. A Tabela 5.4 apresenta quantos métodos são criados por cada compilador. O objetivo dos novos métodos gerados é de auxiliar na verificação da especificação durante a execução do programa. O compilador JML gera 41 métodos de verificação para a classe *Usuario*. Nestes 41 métodos estão inclusos verificação de pós-condição e pós-condição excepcional. Como o compilador JCML não gera funções de verificação para métodos sem especificação e pós-condição, existem apenas 8 métodos de verificação após a classe *Usuario* ser compilada com JCMLc, além dos 10 métodos originais.

5.5 Conclusão

Este capítulo apresentou a JCML, Java Card Modeling Language, uma linguagem de especificação e verificação de aplicações Java Card em tempo de execução, e uma versão inicial do seu compilador. Devido às restrições Java Card, a implementação de verificação *runtime* é uma tarefa difícil. A motivação a implementação da JCML e seu compilador, é prover verificação automática, de propriedades de especificação, em tempo de execução. Este trabalho é inspirado na JML e JMLc, contudo, tem foco específico no desenvolvimento Java Card, ao invés de Java.

Método	Tem Espec.?	JCMLc	JMLc
Invariante	Sim	1	6
setMatricula	Sim	1	3
getMatricula	Não	0	3
setTipo	Sim	1	3
getTipo	Não	0	3
adicionarLocal	Sim	1	3
removerLocal	Sim	1	3
temAcessoLocal	Sim	1	3
adicionarCreditos	Sim	1	3
removerCreditos	Sim	1	3
getCreditos	Não	0	3
Total de Métodos Gerados	x	8	41

Tabela 5.4: Quantidade de Métodos de Verificação Gerados - JCMLc x JMLc .

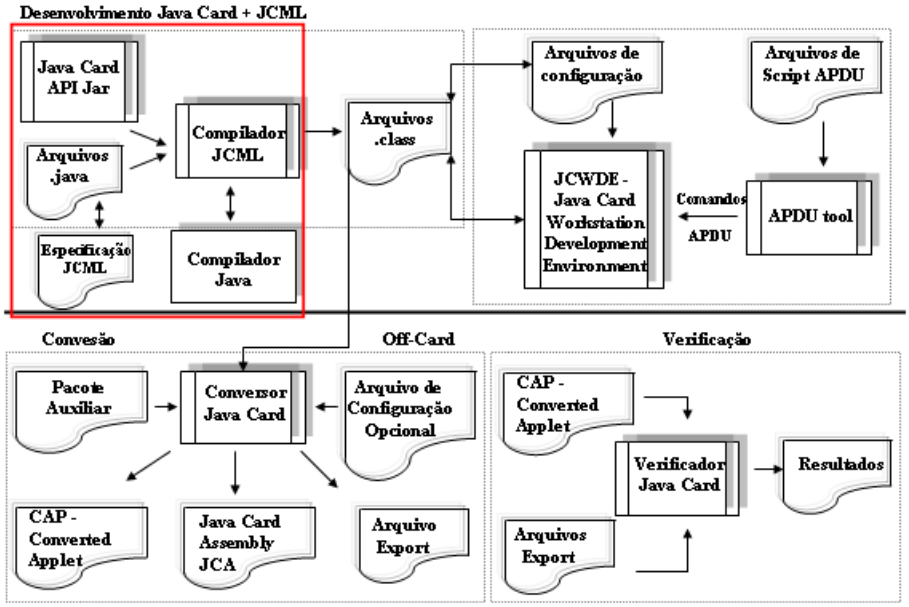

Figura 5.22: Desenvolvimento Java Card Aplicando Especificação JCML.

A primeira versão do compilador, apresentada neste capítulo, está habilitada a checar propriedades simples, as quais podem ser traduzidas diretamente para código Java Card. Foi apresentado, também neste capítulo, um modelo para a implementação de novas propriedades do compilador JCML.

Dessa forma, com o uso da linguagem JCML, são inseridos código adicional ao desenvolvimento Java Card tradicional. Os executáveis Java Card são criados a partir da especificação JCML. O código *runtime* é compatível com as restrições da tecnologia Java Card. A aplicação de especificação JCML no desenvolvimento Java Card segue a estrutura definida na Figura 5.22.

Foi utilizada a ferrameta JAVACC (*Java Compiler Compiler*) [17] para a construção do compilador JCML. O JavaCC é um gerador de analisador sintático que produz código Java. Ele permite que uma linguagem seja definida de maneira simples, por meio de uma notação semelhante à EBNF. Como saída produz o código-fonte para classes Java que implementam os analisadores léxico e sintático para aquela linguagem. O JavaCC define uma linguagem própria para descrição, em um único arquivo, do analisador léxico e do analisador sintático.

Capítulo 6

Trabalhos Relacionados

Neste capítulo, são expostos trabalhos que relacionam métodos formais e aplicações com restrição de recursos. Trabalhos com foco na verificação estática para geração de código a partir de especificação formal para dispositivos com restrições de recurso vêm sendo produto de pesquisas. Por exemplo, o método *BSmart* [16] e o projeto *BOM - B with Optimized Memory* [40]. Estes projetos trabalham com especificações *B*, em vez de JML, com foco na geração de código, não se preocupando com verificação em tempo de execução.

Da mesma forma, também relacionando especificação e geração de código para dispositivos com restrições de recurso, inspirado na JML, se tem a proposta do *CML - C Modeling Language* [13], o qual tem como foco especificação de programas C.

Em [30, 24] é descrita a especificação, por meio de anotações em JML, de classes da API Java Card, como forma de oferecer uma documentação adicional a esta API, bem como permitir que verificações sejam efetivadas por meio de ferramentas que dão suporte à linguagem JML. Verificação Java Card geralmente é efetuada fora do cartão para assegurar que a aplicação a ser instalada segue a especificação Java Card e não compromete a segurança da máquina virtual ou de outros *applets* instalados no cartão.

Trabalhos têm sido desenvolvidos com o objetivo definir técnicas e ferramentas para tradução de código de especificação para linguagem Java. Nas seções a seguir, serão apresentadas alguns características das linguagens e ferramentas que traduzem especificação em código.

6.1 B with Optimized Memory - BOM

O projeto BOM define uma tradução otimizada de código B para a linguagem Java Card, com o objetivo de prover aplicações para dispositivos com pouca capacidade de memória e processamento. O projeto tem como foco dividir a aplicação Java Card em duas classes, uma que implementa a lógica de negócio de uma aplicação *smart card* e outra que implementa o código da classe *applet*. BOM otimiza a geração de código do Atelier B [11].

Os objetivos do projeto BOM são:

- Otimizar a implementação da linguagem;
- Desenvolver um tradutor confiável a partir de uma especificação B, para código C e Java, na qual a otimização de mémoria é o principal foco;
- Prover um tradutor de código *Open Source*;
- Aplicar o método B na arquitetura Java Card;

A ferramenta JBtools [41] é um subproduto do projeto BOM, o qual implementa a tradução de código B em Java Card. Este projeto define um conjunto de ferramentas como: um type checker B, uma ferramenta para geração de documentação e um gerador de código de B para Java e C#.

A geração de código com o BOM é feita a partir de especificação B, enquanto a especificação com JCML é feita no próprio código Java Card através da inserção de comentários de especificação. Diferente do projeto BOM, o JCML ainda não realiza otimização de código. Contudo a aplicação do método B no desenvolvimento (arquitetura) de aplicações Java Card é realizada de forma semelhante ao JCML. Em JCML a verificação de código de especificação é realizada durante a execução do programa. No projeto BOM os métodos da aplicação são gerados a partir da especificação.

6.2 BSmart

O método BSmart [16] foi desenvolvido no laboratório *ConSiste* da *UFRN*, o qual tem como principal objetivo, fornecer um desenvolvimento rigoroso, a partir de especificações

B, de aplicações Java Card. O método consiste em uma metodologia de desenvolvimento, uma ferramenta (*Plugin Eclipse*), fornecendo uma interface gráfica que integra um conjunto de programas que auxilia e automatiza o processo de desenvolvimento Java Card.

A ferramenta *BSmart* possibilita a geração de código para Java Card, utilizando o protocolo APDU e, da mesma forma, também permite que seja gerado código para invocação remota (RMI) de métodos Java Card.

O ciclo de desenvolvimento *BSmart* tem início com a especificação de uma máquina B, a qual define a aplicação. Segue-se com os refinamentos, que introduzem os detalhes específicos e as características da aplicação Java Card. Por fim, é feita a geração do código Java Card com base na implementação dos refinamentos feitos em B.

O método BSmart e a linguagem JCML foram desenvolvidos no mesmo ambiente do projeto Smart (*Engineering of Smart Card Applications*). O projeto tinha como foco a plataforma Java Card e o desenvolvimento de métodos e tecnologias para *smart card*.

O método BSmart gera código Java e Java Card a partir de especificação B, faz análise de compatibilidade da especificação B e o código Java, inclui edição de especificação B e type checking. Diferente do BSmart, o JCML faz a geração de funções de verificação para as construções inseridas no código da aplicação Java Card. O método BSmart gera código para a aplicação a partir da especificação e o JCML gera código de verificação.

6.3 C Modeling Language - CML

A CML [13] é uma linguagem de especificação, desenvolvida em parceria entre a *UPE e Cin/UFPE*. Ela descreve requisitos não-funcionais para aplicações desenvolvidas na linguagem de programação C. CML também é usável em aplicações com restrição de tempo, memória, área e outros recursos limitados.

Semelhante a JML e JCML, a especificação CML é feita em blocos de comentário. O padrão de comentário em CML é entre */*! . . !*/*, limitando o início e fim a especificação.

O compilador CML traduz a especificação em um arquivo XML. Este arquivo é lido por uma outra aplicação que entende o formato gerado. A partir do arquivo XML, é gerado código C.

A sequência de compilação CML é semelhante ao método *BSmart*, que também gera um arquivo XML intermediário a partir da especificação, para que num passo posterior, seja gerado o código Java Card.

Capítulo 7

Considerações Finais

A plataforma Java Card fornece um ambiente favorável à aplicação de projeto por contrato no desenvolvimento de aplicações. Na sua maioria, as aplicações necessitam de funcionamento correto e de alguma garantia de segurança das informações a serem armazenadas. Geralmente, os códigos de aplicações para smart card são pequenos e possuem aspectos em comum, características que podem ser exploradas para a reutilização de especificações.

Os *applets* Java Card necessitam de segurança e garantia de que dados armazenados não são violados. A aplicação de formalismo no desenvolvimento deste tipo de software assegura corretude em relação à sua especificação.

Especificacões que gerem código Java Card a ser verificado em tempo de execução vêm ao encontro das necessidades críticas do ambiente *smart card* e garantir propriedades importantes para as aplicações smart card.

A JML é uma linguagem de especificação Java que é bastante usada e difundida. Apesar de garantir estados desejáveis em tempo de execução em um ambiente Java, sua aplicação em um ambiente de desenvolvimento como o Java Card, não viável. Isso ocorre pelo fato da plataforma Java para cartões ser bem mais restrita que a versão *standard* do Java. Dessa forma, existe uma necessidade de se adaptar a linguagem e ferramentas JML para que este desenvolvimento seja possível.

Este trabalho propôs uma linguagem de especificação para aplicações Java Card, baseada em projeto por contrato, chamada Java Card Modeling Language. A JCML é

uma linguagem baseada na JML, contudo, não utiliza construções incompatíveis com a plataforma Java Card.

Foram feitas análises de viabilidade do uso da JML no desenvolvimento Java Card e foi percebido que apesar de ser possível utilizar contruções JML que são reconhecidas pelo Java Card, o compilador JML gera código que não é possível inserir no cartão.

Após esta análise foi feita a descrição da linguagem JCML, e a descrição do compilador da linguagem JCML, o JCMLc. O compilador JCML gera código de verificação em tempo de execução para aplicações Java Card. O processo de tradução ocorre a partir da criação de métodos que são gerados com o objetivo de realizar a checagem durante a execução dos *applets* Java Card.

Por fim, foi feita uma análise comparativa do código gerado pelo compilador JCML e o compilador JML. O tamanho do executável gerado pelo JCMLc é 7 vezes menor que o gerado pelo JMLc. Em relação a quantidade de linhas geradas após a compilação, há um aumento de 111% com o uso do JCMLc e um aumento de mais de 2600% com o compilador JML. Apesar de verificar menos construções que a JML, o compilador da linguagem proposta gera código que pode ser inserido dentro do cartão, tanto pelos tipos de métodos e objetos gerados, quanto pelo tamanho do arquivo após a compilação. O compilador JCML é mais restrito, em sua versão inicial, que o compilador JML.

Existem tarefas importantes, para o compilador JCML, que ainda não foram executadas. Estas tarefas são dispostas como trabalhos futuros:

- Implementação de construções para toda especificação *ligthweigth*, como por exemplo, as cláusulas *assignable e signals*,
- Checagem de pós-condição normal e excepcional,
- Geração de código de verificação para construções de quantificadores *exists* e *forall*,
- Checagem de métodos puros,
- Verificação de propriedades em tempo de compilação,
- Adição de construções específicas para a plataforma Java Card,

- Transações e,

- Estudos de caso.

Referências Bibliográficas

[1] J. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[2] Stephen Bear. Structuring for the VDM specification language. In *VDM The Way Ahead (VDM '88)*, pages 2–25, Berlin - Heidelberg - New York, September 1988. Springer.

[3] A. Bhorkar. A run-time assertion checker for Java using JML. Technical Report 00-08, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011, May 2000. Available by anonymous ftp from ftp.cs.iastate.edu or by e-mail from almanac@cs.iastate.edu.

[4] Breunesse, Catano, Huisman, and Jacobs. Formal methods for smart cards: An experience report. *SCIPROG: Science of Computer Programming*, 55, 2005.

[5] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005.

[6] S. Burri. Design of a java/jml frontend for boogiepl. Master's thesis, 2005.

[7] Hoare C.A.R. *Proof od Correctness of Data Representations*, volume 1. Acta Informatica, 1972.

[8] Z. Chen. *Java Card technology for Smart Cards: architecture and programmer's guide*. Java series. Addison-Wesley, pub-AW:adr, 2000.

[9] Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Department of Computer Science, Iowa State University, April 2003.

[10] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In Boris Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Máalaga, Spain, Proceedings*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255, Berlin, June 2002. Springer-Verlag.

[11] ClearSy. Atelier B, user and reference manuals. 1996.

[12] The Coq. The coq proof assistant : Reference manual : Version 7.2, February 2002.

[13] F. de Oliveira Jr. etal. CML: C modeling language. In *Proceedings of the XI Brazilian Symposium on Programming Languages*, pages 5–18, 2007.

[14] Detlefs, Nelson, and Saxe. Simplify: A theorem prover for program checking. *JACM: Journal of the ACM*, 52, 2005.

[15] Eiffel Software. Building bug-free O-O software: An introduction to Design by Contract.

[16] B. Gomes, A. M. Moreira, and D. Déharbe. Developing java card applications with B. In *Proceedings of the Brazilian Symposium on Formal Methods (SBMF'2005)*, pages 63–77, 2005.

[17] Java compiler compiler. http://www.experimentalstuff.com/Technologies/JavaCC/.

[18] J. Jézéquel and B. Meyer. Design by contract: The lessons of ariane. *IEEE Computer*, 30(1):129–130, January 1997.

[19] T. M. Jurgensen and S. B. Guthery. *Smart Cards: A Developer's Toolkit*. Prentice Hall PTR, 2002.

[20] G. T. Leavens and Y. Cheon. Design by contract with JML. Draft, available from jmlspecs.org., 2006.

[21] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual*, May 2006. Draft revision 1.193.

[22] K. R. M. Leino, G. Nelson, and J. B. Saxe. ESC/Java user's manual. Technical note, Compaq Systems Research Center, October 2000.

[23] C. Marche, C. Paulin Mohring, and X. Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004.

[24] H. Meijer and E. Poll. Towards a full formal specification of the JavaCard API. *Lecture Notes in Computer Science*, 2140, 2001.

[25] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1991.

[26] B. Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, 1992.

[27] Sun Microsystems. Java card technology overview. <http://java.sun.com/products/javacard/overview.html>, Acessado em julho de 2006.

[28] C. E. Ortiz. An introduction to java card technology - part 2, the java card applet. <http://developers.sun.com/techtopics/mobility/javacard/articles/javacard2/>, 2003, Acessado em outubro de 2006.

[29] M. Pavlova and L. Burdy. Java bytecode specification and verification. *21st Annual ACM Symposium on Applied Computing*, 2006.

[30] E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Smart Card Research and Advanced Application Conference (CARDIS'2000)*, pages 135–154. Kluwer Acad. Publ., 2000.

[31] E. Poll, J. van den Berg, and B. Jacobs. Formal specification of the javacard API in JML: the APDU class. *Computer Networks*, 36(4):407–421, 2001.

[32] Kenneth H. Rosen. *Discrete Mathematics and its Applications*. McGraw-Hill Book Company, New York-St. Louis-San Francisco-Toronto-London-Sydney-Hamburg, 1995.

[33] Gavin Scott. Design by contract. Technical report, Dublin City University, 1999.

[34] N. Shankar. PVS: Combining specification, proof checking, and model checking. *Lecture Notes in Computer Science*, 1166, 1996.

[35] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[36] STMicroeletronics. Smartcard 32-bit risc mcu with 144 kbytes eeprom java card hw execution and cryptographic library. Technical report, ST Microeletronics, 2006.

[37] Sun Microsystems. *API Specification - Java Card Platform Application Programming Interface Specification, Version 2.2.1*, 2005.

[38] Sun Microsystems. *Runtime Environment Specification - Java Card Platform, Version 2.2.1*, 2005.

[39] Sun Microsystems. *Virtual Machine Specification - Java Card Platform, Version 2.2.1*, 2005.

[40] B. Tatibouet, A. Requet, J.C. Voisinet, and A. Hammad. Java Card code generation from B specifications. In *5th International Conference on Formal Engineering Methods (ICFEM'2003)*, volume 2885 of *LNCS*, pages 306–318, Singapore, November 2003.

[41] B. Tatibouët and J. C. Voisinet. jBTools and B2UML: a platform and a tool to provide a UML class diagram since a B specification. In *ICSSEA'2001 – 14th Int. Conf. on Software Systems Engineering and Their Applications*, volume 2, CNAM, Paris, France, December 2001.

[42] van den Berg and Jacobs. The LOOP compiler for java and JML. In *TACAS: International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, LNCS*, 2001.

[43] J. van den Berg and B. Jacobs. The LOOP compiler for java and JML. In Tiziana Margaria and Wang Yi, editors, *TACAS*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer, 2001.

[44] J. Woodcock and J. Davies. *Using Z. Specification, Refinement, and Proof*. Prentice Hall, London, 1995.

Apêndice A

Classes da Aplicação de Controle de Usuário

A.1 Classe Usuario.java

```
import javacard.framework.ISOException;


public class Usuario {
    public static final byte QUANTIDADE_MAXIMA_LOCAIS = 127;
    public static final short VALOR_MAXIMO_CREDITOS = 32767;
    public static final byte TAMANHO_TIPO = 1;
    public static final byte TAMANHO_LOCAL = 1;
    public static final byte TAMANHO_CREDITOS = 2;
    public static final byte TAMANHO_MAXIMO_MATRICULA = 15;

    public static final byte TIPO_ESTUDANTE = 0;
    public static final byte TIPO_PROFESSOR = 1;

    public static final short SW_TAMANHO_MATRICULA_INVALIDO =
            (short) 0x63A0;
    public static final short SW_LOCAL_INEXISTENTE = (short) 0x63A1;
    public static final short SW_QUANTIDADE_MAXIMA_LOCAIS_EXCEDIDA =
            (short) 0x63A2;
    public static final short SW_VALOR_MAXIMO_CREDITO_EXCEDIDO =
            (short) 0x63A3;
    public static final short SW_CREDITOS_INSUFICIENTES = (short) 0x63A4;
    public static final short SW_TIPO_USUARIO_INVALIDO = (short) 0x63A5;

    private byte indiceLocal;
    private byte[] locais;
    private byte[] matricula;
    private byte tipo;
    private short creditos;

    public Usuario() {
        tipo = TIPO_ESTUDANTE;
        matricula = new byte[TAMANHO_MAXIMO_MATRICULA];
        locais = new byte[QUANTIDADE_MAXIMA_LOCAIS];
```

```
        indiceLocal = 0;
        creditos = 0;
    }

    public void setMatricula(byte[] m) throws ISOException {
        if (m.length > TAMANHO_MAXIMO_MATRICULA) {
            ISOException.throwIt(SW_TAMANHO_MATRICULA_INVALIDO);
        }
        matricula = m;
    }

    public byte[] getMatricula() {
        return matricula;
    }

    public void setTipo(byte t) throws ISOException {
        if (t != TIPO_ESTUDANTE &&
                t != TIPO_PROFESSOR) {
            ISOException.throwIt(SW_TIPO_USUARIO_INVALIDO);
        }
        tipo = t;
    }

    public byte getTipo() {
        return tipo;
    }

    public void adicionarLocal(byte codigo_local) throws ISOException {
        if (indiceLocal >= QUANTIDADE_MAXIMA_LOCAIS) {
            ISOException.throwIt(SW_QUANTIDADE_MAXIMA_LOCAIS_EXCEDIDA);
        }
        else if (!this.temAcessoLocal(codigo_local)) {
            locais[indiceLocal] = codigo_local;
            indiceLocal++;
        }
    }

    public void removerLocal(byte codigo_local) throws ISOException {
        for (byte b = 0; b < indiceLocal; b++) {
            if (locais[b] == codigo_local) {
                for (byte c = b; c < indiceLocal - 1; c++) {
                    locais[c] = locais[c + 1];
                }
                indiceLocal--;
                return;
            }
        }
        ISOException.throwIt(SW_LOCAL_INEXISTENTE);
    }

    public boolean temAcessoLocal(byte codigo_local) {
        for (byte b = 0; b < indiceLocal; b++) {
            if (locais[b] == codigo_local) {
                return true;
            }
        }
        return false;
    }
```

```
    public void adicionarCreditos(short valor) {
        if ((short) (valor + creditos) > VALOR_MAXIMO_CREDITOS) {
            ISOException.throwIt(SW_VALOR_MAXIMO_CREDITO_EXCEDIDO);
        }
        creditos += valor;
    }

    public void removerCreditos(short valor) {
        if (valor > creditos) {
            ISOException.throwIt(SW_CREDITOS_INSUFICIENTES);
        }
        creditos -= valor;
    }

    public short getCreditos() {
        return creditos;
    }
}
```

A.2 Classe UserApplet.java

```
import javacard.framework.APDU;
import javacard.framework.Applet;
import javacard.framework.ISO7816;
import javacard.framework.ISOException;
import javacard.framework.OwnerPIN;
import javacard.framework.Util;

public class UsuarioApplet extends Applet {

    public static final byte CLA_SMARTESCOLA = (byte) 0x80;
    public static final byte INS_VERIFICAR_SENHA = (byte) 0x00;
    public static final byte INS_SET_MATRICULA = (byte) 0x11;
    public static final byte INS_SET_TIPO = (byte) 0x12;
    public static final byte INS_GET_MATRICULA = (byte) 0x21;
    public static final byte INS_GET_TIPO = (byte) 0x22;
    public static final byte INS_ADICIONAR_LOCAL = (byte) 0x13;
    public static final byte INS_VERIFICAR_LOCAL = (byte) 0x23;
    public static final byte INS_REMOVER_LOCAL = (byte) 0x33;
    public static final byte INS_ADICIONAR_CREDITOS = (byte) 0x14;
    public static final byte INS_VERIFICAR_CREDITOS = (byte) 0x24;
    public static final byte INS_REMOVER_CREDITOS = (byte) 0x34;

    public static final short SW_SENHA_INVALIDA = (short) 0x6301;
    public static final short SW_AUTENTICACAO_INVALIDA = (short) 0x6302;

    public static final byte MAXIMO_TENTATIVAS_PIN = 3;
    public static final byte TAMANHO_MAXIMO_PIN = 10;

    private OwnerPIN pin;
    private Usuario u;
```

```
public static void install(byte[] bArray, short bOffset,
        byte bLength) {
    new UsuarioApplet(bArray, bOffset, bLength);
}

private UsuarioApplet(byte[] bArray, short bOffset, byte bLength) {
    super();
    u = new Usuario();
    pin = new OwnerPIN(MAXIMO_TENTATIVAS_PIN, TAMANHO_MAXIMO_PIN);
    pin.update(bArray, bOffset, bLength);
    register();
}

public boolean select() {
    if (pin.getTriesRemaining() == 0) {
        return false;
    }
    return true;
}

public void deselect() {
    pin.reset();
}

public void process(APDU apdu) throws ISOException {
    byte[] buffer = apdu.getBuffer();
    byte cla = buffer[ISO7816.OFFSET_CLA];
    byte ins = buffer[ISO7816.OFFSET_INS];

    if (cla == ISO7816.CLA_ISO7816 && ins == ISO7816.INS_SELECT) {
        return;
    }

    if (cla != CLA_SMARTESCOLA) {
        ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
    }

    switch (ins) {
        case INS_VERIFICAR_SENHA:
            // Verificar senha
            this.verificarSenha(apdu);
            break;
        case INS_SET_MATRICULA:
            // Alterar a matrícula
            this.setMatricula(apdu);
            break;
        case INS_SET_TIPO:
            // Alterar o tipo
            this.setTipo(apdu);
            break;
        case INS_GET_MATRICULA:
            // Recuperar a matrícula
            this.getMatricula(apdu);
            break;
        case INS_GET_TIPO:
            // Recuperar o tipo
            this.getTipo(apdu);
            break;
```

```
            case INS_ADICIONAR_LOCAL:
                // Adicionar um local
                this.adicionarLocal(apdu);
                break;
            case INS_VERIFICAR_LOCAL:
                // Verificar se um local é permitido
                this.verificarLocal(apdu);
                break;
            case INS_REMOVER_LOCAL:
                // Remover um local permitido
                this.removerLocal(apdu);
                break;
            case INS_ADICIONAR_CREDITOS:
                // Adicionar créditos
                this.adicionarCreditos(apdu);
                break;
            default:
                ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
        }
    }

    private void verificarSenha(APDU apdu) {
        byte tam_senha = (byte) apdu.setIncomingAndReceive();
        byte[] buffer = apdu.getBuffer();
        if (!pin.check(buffer, ISO7816.OFFSET_CDATA, tam_senha)) {
            ISOException.throwIt(SW_SENHA_INVALIDA);
        }
    }

    private void setMatricula(APDU apdu) {
        if (!pin.isValidated()) {
            ISOException.throwIt(SW_AUTENTICACAO_INVALIDA);
        }

        byte[] buffer = apdu.getBuffer();
        byte tam_entrada = (byte) apdu.setIncomingAndReceive();
        byte lc = buffer[ISO7816.OFFSET_LC];
        if (tam_entrada != lc) {
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }
        if (tam_entrada > Usuario.TAMANHO_MAXIMO_MATRICULA) {
            ISOException.throwIt(Usuario.SW_TAMANHO_MATRICULA_INVALIDO);
        }
        byte[] nova_matricula = new byte[tam_entrada];
        Util.arrayCopy(buffer, ISO7816.OFFSET_CDATA, nova_matricula,
                (short) 0, tam_entrada);
        u.setMatricula(nova_matricula);
    }

    private void getMatricula(APDU apdu) {
        byte tam_saida = (byte) apdu.setOutgoing();
        if (tam_saida > Usuario.TAMANHO_MAXIMO_MATRICULA) {
            ISOException.throwIt(Usuario.SW_TAMANHO_MATRICULA_INVALIDO);
        }
        apdu.setOutgoingLength(tam_saida);
        apdu.sendBytesLong(u.getMatricula(), (short) 0, tam_saida);
    }
```

```
    private void setTipo(APDU apdu) {
        if (!pin.isValidated()) {
            ISOException.throwIt(SW_AUTENTICACAO_INVALIDA);
        }

        byte[] buffer = apdu.getBuffer();
        byte tam_tipo = (byte) apdu.setIncomingAndReceive();
        if (tam_tipo != Usuario.TAMANHO_TIPO) {
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }
        byte tipo = buffer[ISO7816.OFFSET_CDATA];
        if (tipo != Usuario.TIPO_ESTUDANTE && tipo != Usuario.TIPO_PROFESSOR) {
            ISOException.throwIt(ISO7816.SW_WRONG_DATA);
        }
        u.setTipo(tipo);
    }

    private void getTipo(APDU apdu) {
        byte[] buffer = apdu.getBuffer();
        byte tam_saida = (byte) apdu.setOutgoing();
        if (tam_saida != Usuario.TAMANHO_TIPO) {
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }
        byte indice_inicio = 0;
        buffer[indice_inicio] = u.getTipo();
        apdu.setOutgoingLength(Usuario.TAMANHO_TIPO);
        apdu.sendBytes(indice_inicio, Usuario.TAMANHO_TIPO);
    }

    private void adicionarLocal(APDU apdu) {
        if (!pin.isValidated()) {
            ISOException.throwIt(SW_AUTENTICACAO_INVALIDA);
        }

        byte[] buffer = apdu.getBuffer();
        byte tam_local = (byte) apdu.setIncomingAndReceive();
        if (tam_local != Usuario.TAMANHO_LOCAL) {
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }
        byte local = buffer[ISO7816.OFFSET_CDATA];
        u.adicionarLocal(local);
    }

    private void verificarLocal(APDU apdu) {
        if (!pin.isValidated()) {
            ISOException.throwIt(SW_AUTENTICACAO_INVALIDA);
        }

        byte[] buffer = apdu.getBuffer();
        byte tam_local = (byte) apdu.setIncomingAndReceive();
        if (tam_local != Usuario.TAMANHO_LOCAL) {
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }
        byte local = buffer[ISO7816.OFFSET_CDATA];
        if (!u.temAcessoLocal(local)) {
            ISOException.throwIt(Usuario.SW_LOCAL_INEXISTENTE);
        }
    }
```

```
    private void removerLocal(APDU apdu) {
        if (!pin.isValidated()) {
            ISOException.throwIt(SW_AUTENTICACAO_INVALIDA);
        }

        byte[] buffer = apdu.getBuffer();
        byte tam_local = (byte) apdu.setIncomingAndReceive();
        if (tam_local != Usuario.TAMANHO_LOCAL) {
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }
        byte local = buffer[ISO7816.OFFSET_CDATA];
        u.removerLocal(local);
    }

    private void adicionarCreditos(APDU apdu) {
        if (!pin.isValidated()) {
            ISOException.throwIt(SW_AUTENTICACAO_INVALIDA);
        }

        byte[] buffer = apdu.getBuffer();
        byte tam_creditos = (byte) apdu.setIncomingAndReceive();
        if (tam_creditos != Usuario.TAMANHO_CREDITOS) {
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }
        short valor_creditos = Util.getShort(buffer, ISO7816.OFFSET_CDATA);
        u.adicionarCreditos(valor_creditos);
    }

    private void removerCreditos(APDU apdu) {
        if (!pin.isValidated()) {
            ISOException.throwIt(SW_AUTENTICACAO_INVALIDA);
        }

        byte[] buffer = apdu.getBuffer();
        byte tam_creditos = (byte) apdu.setIncomingAndReceive();
        if (tam_creditos != Usuario.TAMANHO_CREDITOS) {
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }
        short valor_creditos = Util.getShort(buffer, ISO7816.OFFSET_CDATA);
        u.removerCreditos(valor_creditos);
    }

    private void verificarCreditos(APDU apdu) {
        byte[] buffer = apdu.getBuffer();
        byte tam_esperado = (byte) apdu.setOutgoing();
        if (tam_esperado != Usuario.TAMANHO_CREDITOS) {
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        }
        apdu.setOutgoingLength(Usuario.TAMANHO_CREDITOS);
        short indice_inicio = 0;
        Util.setShort(buffer, indice_inicio, u.getCreditos());
        apdu.sendBytes(indice_inicio, Usuario.TAMANHO_CREDITOS);
    }
}
```

Apêndice B

Classes Usuário após ser Compilada com JCMLc

B.1 Classe Usuario.java com Especificação JCML

```
import javacard.framework.ISOException;

public class Usuario {
    public static final byte QUANTIDADE_MAXIMA_LOCAIS = 127;
    public static final short VALOR_MAXIMO_CREDITOS = 32767;
    public static final byte TAMANHO_TIPO = 1;
    public static final byte TAMANHO_LOCAL = 1;
    public static final byte TAMANHO_CREDITOS = 2;
    public static final byte TAMANHO_MAXIMO_MATRICULA = 15;

    public static final byte TIPO_ESTUDANTE = 0;
    public static final byte TIPO_PROFESSOR = 1;

    public static final short SW_TAMANHO_MATRICULA_INVALIDO =
            (short) 0x63A0;
    public static final short SW_LOCAL_INEXISTENTE = (short) 0x63A1;
    public static final short SW_QUANTIDADE_MAXIMA_LOCAIS_EXCEDIDA =
            (short) 0x63A2;
    public static final short SW_VALOR_MAXIMO_CREDITO_EXCEDIDO =
            (short) 0x63A3;
    public static final short SW_CREDITOS_INSUFICIENTES = (short) 0x63A4;
    public static final short SW_TIPO_USUARIO_INVALIDO = (short) 0x63A5;

    private /*@ spec_public @*/ byte indiceLocal;
    private /*@ spec_public @*/ byte[] locais;
    private /*@ spec_public @*/ byte[] matricula;
    private /*@ spec_public @*/ byte tipo;
    private /*@ spec_public @*/ short creditos;


    /*@ invariant indiceLocal >= 0 && indiceLocal <= 10; @*/
    /*@ invariant creditos >= 0 && creditos <= VALOR_MAXIMO_CREDITOS; @*/
    /*@ invariant tipo == TIPO_PROFESSOR || tipo == TIPO_ESTUDANTE; @*/

    public Usuario() {
```

```
        tipo = TIPO_ESTUDANTE;
        matricula = new byte[TAMANHO_MAXIMO_MATRICULA];
        locais = new byte[QUANTIDADE_MAXIMA_LOCAIS];
        indiceLocal = 0;
        creditos = 0;
    }

    /*@
       requires m != null;
       requires m.length <= TAMANHO_MAXIMO_MATRICULA;
    @*/
    public void setMatricula(byte[] m) throws ISOException {
        if (m.length > TAMANHO_MAXIMO_MATRICULA) {
            ISOException.throwIt(SW_TAMANHO_MATRICULA_INVALIDO);
        }
        matricula = m;
    }

    public byte[] getMatricula() {
        return matricula;
    }

    /*@
      requires t == TIPO_ESTUDANTE || t == TIPO_PROFESSOR;
    @*/
    public void setTipo(byte t) throws ISOException {
        tipo = t;
    }

    public byte getTipo() {
        return tipo;
    }


    /*@
      requires codigo_local >= 0;
    @*/
    public void adicionarLocal(byte codigo_local) throws ISOException {
        if (indiceLocal >= QUANTIDADE_MAXIMA_LOCAIS) {
            ISOException.throwIt(SW_QUANTIDADE_MAXIMA_LOCAIS_EXCEDIDA);
        }
        else if (!this.temAcessoLocal(codigo_local)) {
            locais[indiceLocal] = codigo_local;
            indiceLocal++;
        }
    }

    /*@
      requires temAcessoLocal(codigo_local);
    @*/
    public void removerLocal(byte codigo_local) throws ISOException {
        for (byte b = 0; b < indiceLocal; b++) {
            if (locais[b] == codigo_local) {
                for (byte c = b; c < indiceLocal - 1; c++) {
                    locais[c] = locais[c + 1];
                }
                indiceLocal--;
                return;
```

```
            }
        }
        ISOException.throwIt(SW_LOCAL_INEXISTENTE);
    }

    /*@
     requires codigo_local >= 0;
    @*/
    public boolean temAcessoLocal(byte codigo_local) {
        for (byte b = 0; b < indiceLocal; b++) {
            if (locais[b] == codigo_local) {
                return true;
            }
        }
        return false;
    }

    /*@
     requires valor >= 0;
     requires valor + getCreditos() <= VALOR_MAXIMO_CREDITOS;
    @*/
    public void adicionarCreditos(short valor) {
        creditos += valor;
    }

    /*@
     requires valor >= 0;
     requires valor <= getCreditos();
    @*/
    public void removerCreditos(short valor) {
        creditos -= valor;
    }

    public short getCreditos() {
        return creditos;
    }
}
```

B.2 Classe UsuarioJCML.java após Compilada com JCMLc

```
import javacard.framework.ISOException ;

public class UsuarioJCML {

    public static final byte QUANTIDADE_MAXIMA_LOCAIS = 127 ;
public static final short VALOR_MAXIMO_CREDITOS = 32767 ;
    public static final byte TAMANHO_TIPO = 1 ;
    public static final byte TAMANHO_LOCAL = 1 ;
    public static final byte TAMANHO_CREDITOS = 2 ;
    public static final byte TAMANHO_MAXIMO_MATRICULA = 15 ;
    public static final byte TIPO_ESTUDANTE = 0 ;
```

```
    public static final byte TIPO_PROFESSOR = 1 ;
    public static final short SW_TAMANHO_MATRICULA_INVALIDO = (short) 0x63A0 ;
    public static final short SW_LOCAL_INEXISTENTE = (short) 0x63A1 ;
    public static final short SW_QUANTIDADE_MAXIMA_LOCAIS_EXCEDIDA = (short) 0x63A2 ;
    public static final short SW_VALOR_MAXIMO_CREDITO_EXCEDIDO = (short) 0x63A3 ;
    public static final short SW_CREDITOS_INSUFICIENTES = (short) 0x63A4 ;
    public static final short SW_TIPO_USUARIO_INVALIDO = (short) 0x63A5 ;

    private byte indiceLocal;
    private byte [] locais;
    private byte [] matricula;
    private byte tipo;
    private short creditos;

    public UsuarioJCML ( ) {
        tipo = TIPO_ESTUDANTE ;
        matricula = new byte [TAMANHO_MAXIMO_MATRICULA ] ;
        locais = new byte [QUANTIDADE_MAXIMA_LOCAIS ] ;
        indiceLocal = 0 ;
        creditos = 0 ;
    }

    public void setMatricula ( byte [] m) throws ISOException {
        try{
            checkInv$Usuario$();
            checkPre$setMatricula$Usuario(m) ;

            if (m.length > TAMANHO_MAXIMO_MATRICULA ) {
                ISOException.throwIt (SW_TAMANHO_MATRICULA_INVALIDO ) ;
            }
            matricula = m ;
       }catch (InvariantException invEx) {
            InvariantException.throwIt(InvariantException.SW_INVARIANT_ERROR);
       }catch (RequiresException reqEx) {
            RequiresException.throwIt(RequiresException.SW_REQUIRES_ERROR);
       }
   }

private void checkPre$setMatricula$Usuario( byte[] m) throws RequiresException{
       if(!(m != null && m.length <= TAMANHO_MAXIMO_MATRICULA ))
          RequiresException.throwIt(RequiresException.SW_REQUIRES_ERROR);
    }

    public byte [] getMatricula ( ) {
       try{
          checkInv$Usuario$();
          return matricula ;
       }catch (InvariantException invEx) {
          InvariantException.throwIt(InvariantException.SW_INVARIANT_ERROR);
          return null;
       }
    }

    public void setTipo ( byte t) throws ISOException {
       try{
            checkInv$Usuario$();
            checkPre$setTipo$Usuario( t) ;
```

```
            tipo = t ;
        }catch (InvariantException invEx) {
            InvariantException.throwIt(InvariantException.SW_INVARIANT_ERROR);
        }catch (RequiresException reqEx) {
            RequiresException.throwIt(RequiresException.SW_REQUIRES_ERROR);
        }
    }

private void checkPre$setTipo$Usuario( byte t) throws RequiresException{
        if(!(t == TIPO_ESTUDANTE || t == TIPO_PROFESSOR ))
            RequiresException.throwIt(RequiresException.SW_REQUIRES_ERROR);
    }

    public byte getTipo ( ) {
       try{
           checkInv$Usuario$();
           return tipo ;
       }catch (InvariantException invEx) {
           InvariantException.throwIt(InvariantException.SW_INVARIANT_ERROR);
           return (byte)0;
       }
   }

   public void adicionarLocal ( byte codigo_local) throws ISOException {
        try{
            check$Invariant$Usuario();
            checkPre$adicionarLocal$Usuario( codigo_local) ;

            if (indiceLocal >= QUANTIDADE_MAXIMA_LOCAIS ) {
                ISOException.throwIt (SW_QUANTIDADE_MAXIMA_LOCAIS_EXCEDIDA ) ;
            }else if (! this.temAcessoLocal (codigo_local ) ) {
                locais [indiceLocal ]= codigo_local ;
                indiceLocal ++ ;
           }
        }catch (InvariantException invEx) {
            InvariantException.throwIt(InvariantException.SW_INVARIANT_ERROR);
        }catch (RequiresException reqEx) {
            RequiresException.throwIt(RequiresException.SW_REQUIRES_ERROR);
        }
   }

   private void checkPre$adicionarLocal$Usuario( byte codigo_local)
                                          throws RequiresException{
       if(!(codigo_local >= 0 ))
          RequiresException.throwIt(RequiresException.SW_REQUIRES_ERROR);
   }

  public void removerLocal ( byte codigo_local) throws ISOException {
      try{
            checkInv$Usuario$();
            checkPre$removerLocal$Usuario( codigo_local) ;

            for( byte b = 0 ;b < indiceLocal ;b ++ ){
               if (locais [b ]== codigo_local ) {
                  for( byte c = b ;c < indiceLocal - 1 ;c ++ ){
                      locais [c ]= locais [c + 1 ];
                  }
                  indiceLocal -- ;
```

```
                return ;
              }
            }
            ISOException.throwIt (SW_LOCAL_INEXISTENTE ) ;
        }catch (InvariantException invEx) {
           InvariantException.throwIt(InvariantException.SW_INVARIANT_ERROR);
        }catch (RequiresException reqEx) {
           RequiresException.throwIt(RequiresException.SW_REQUIRES_ERROR);
        }
   }

   private void checkPre$removerLocal$Usuario( byte codigo_local)
                                        throws RequiresException{
      if(!(temAcessoLocal(codigo_local)))
          RequiresException.throwIt(RequiresException.SW_REQUIRES_ERROR);
  }

    public boolean temAcessoLocal ( byte codigo_local) {
        try{
            checkInv$Usuario$();
            checkPre$temAcessoLocal$Usuario( codigo_local) ;

            for( byte b = 0 ;b < indiceLocal ;b ++ ){
               if (locais [b ]== codigo_local ) {
                   return true ;
               }
            }
            return false ;
        }catch (InvariantException invEx) {
            InvariantException.throwIt(InvariantException.SW_INVARIANT_ERROR);
            return false;
        }catch (RequiresException reqEx) {
            RequiresException.throwIt(RequiresException.SW_REQUIRES_ERROR);
            return false;
        }
    }

   private void checkPre$temAcessoLocal$Usuario( byte codigo_local)
                                          throws RequiresException{
       if(!(codigo_local >= 0 ))
            RequiresException.throwIt(RequiresException.SW_REQUIRES_ERROR);
   }

    public void adicionarCreditos ( short valor) {
       try{
checkInv$Usuario$();
            checkPre$adicionarCreditos$Usuario( valor) ;

            creditos += valor ;
       }catch (InvariantException invEx) {
           InvariantException.throwIt(InvariantException.SW_INVARIANT_ERROR);
       }catch (RequiresException reqEx) {
           RequiresException.throwIt(RequiresException.SW_REQUIRES_ERROR);
        }
    }

    private void checkPre$adicionarCreditos$Usuario( short valor) throws RequiresExcep
```

```
        if(!(valor >= 0 && valor + getCreditos () <= VALOR_MAXIMO_CREDITOS ))
            RequiresException.throwIt(RequiresException.SW_REQUIRES_ERROR);
    }

    public void removerCreditos ( short valor) {
        try{
             checkInv$Usuario$();
             checkPre$removerCreditos$Usuario( valor) ;

             creditos -= valor ;
        }catch (InvariantException invEx) {
             InvariantException.throwIt(InvariantException.SW_INVARIANT_ERROR);
        }catch (RequiresException reqEx) {
            RequiresException.throwIt(RequiresException.SW_REQUIRES_ERROR);
        }
    }

    private void checkPre$removerCreditos$Usuario( short valor)
                                        throws RequiresException{
        if(!(valor >= 0 && valor <= getCreditos () ))
            RequiresException.throwIt(RequiresException.SW_REQUIRES_ERROR);
    }

    public short getCreditos ( ) {
        try{
             checkInv$Usuario$();
             return creditos ;
        }catch (InvariantException invEx) {
             InvariantException.throwIt(InvariantException.SW_INVARIANT_ERROR);
             return (short)0;
        }
    }

    private void checkInv$Usuario$() throws InvariantException {
         if (!(indiceLocal >= 0 && indiceLocal <= 10 ))
            InvariantException.throwIt(InvariantException.SW_INVARIANT_ERROR);

         if (!(creditos >= 0 && creditos <= VALOR_MAXIMO_CREDITOS ))
            InvariantException.throwIt(InvariantException.SW_INVARIANT_ERROR);

         if (!(tipo == TIPO_PROFESSOR || tipo == TIPO_ESTUDANTE ))
            InvariantException.throwIt(InvariantException.SW_INVARIANT_ERROR);
    }

}
```

Apêndice C

Gramática JCML

C.1 Unidade de Compilação

jcml-compilation-unit ::= *[package-definition]*
*(import-definition)**
*(type-definition)**

package-definition ::= **package** *name* **;**
import-definition ::= **import** *name-star* **;**

name ::= *ident (* **.** *ident)** **;**
name-star ::= *ident (* **.** *ident)* [* **.***]* **;**

C.2 Definindo Tipos de Estrutura

type-definition ::= *class-definition* | *interface-definition* | **;**

class-definition ::= *[java-doc-comment] [java-comment] modifiers* **class** *ident*
[class-extends-clause] [implements-clause]
class-block

interface-definition ::= *[java-doc-comment] [java-comment] modifiers* **interface** *ident*
[implements-clause]
class-block

class-block ::= **{** *(field)** **}**

class-extends-clause ::= **extends** *name*
implements-clause ::= **implements** *name-list*
name-list ::= *name (* **,** *name)**
interface-extends ::= **extends** *name-list*
modifiers ::= *(modifier)**
modifier ::= **public** | **protected** | **private**
| **abstract** | **static**
| **final** | **synchronized**
| *jcml-modifier*

C.3 Membros de Classe - Declarando Classes e Interfaces

field ::= *member-decl*
| *jcml-declaration*
| *class-initializer-decl*

member-decl ::= *method-decl*
| *variable-definition*
| *class-definition*
| *interface-definition*

method-decl ::= *((java-doc-comment)* | (java-comment)*)*
[jcml-method-specification]
[modifiers] method-head
method-body

method-head ::= *ident formals [throws-clause]*
method-body ::= *compound-statement*
throws-clause ::= **throws** *name (, name)**
formals ::= **(** *[param-declaration-list]* **)**
param-declaration-list ::= *param-declaration (, param-declaration)*
param-declaration ::= *(param-modifier)* type-spec ident [dims]*
param-modifier ::= **final** | **non_null** | **nullable**

variable-definition ::= *((java-doc-comment)* | (java-comment)*)*
modifiers variable-decls

variable-decls ::= *[field] type-spec variable-declarators* **;**

variable-declarators ::= *variable-declarator (, variable-declarator)**

variable-declarator ::= *ident [dims] [* **=** *initializer]*

initializer ::= *expression* | *array-initializer*
array-initializer ::= **{** *[initializer-list]* **}**
initializer-list ::= *initializer (, initializer) **

type-spec ::= *type [dims]* | *\TYPE [dims]*
type ::= *reference-type* | *built-in-type*
reference-type ::= *name*
built-in-type ::= **void** | **boolean** | **byte** | **short**
dims ::= **[ ]**

class-initializer-decl ::= *[method-specification] compound-statement*
| *method-specification*

C.4 Tipos de Especificação JCML

jcml-declaration ::= *modifiers invariant*

| *modifiers history-constraint*
| *modifiers represents-decl*

invariant ::= *invariant-keyword predicate* **;**

invariant-keyword ::= **invariant**

invariant ::= *constraint-keyword predicate* **;**

constraint-keyword ::= **constraint**

represents-decl ::= *represents-keyword store-ref-expression*
l-arrow-or-eq spec-expression **;**
|
represents-keyword store-ref-expression **such_that**
predicate **;**

represents-keyword ::= **represents**

l-arrow-or-eq ::= **<-** | **=**

initially-clause ::= **initially** *predicate* **;**

axiom-clause ::= **axiom** *predicate* **;**

jcml-modifier ::= **spec_public**
| **spec_protected**
| **model** | **ghost** | **pure**
| **helper** | **uninitialized**
| **non_null** | **nullable** | **nullable_by_default**

C.5 Especificação de Método JCML

jcml-method-specification ::= *specification* |
extending-specification

extending-specification ::= **also** *specification*

specification ::= *spec-case-seq*

spec-case-seq ::= *spec-case (* **also** *spec-case)**

spec-case ::= *lightweight-spec-case* | *heavyweight-spec-case*

privacy ::= **public** | **protected** | **private**

C.5.1 Especificação Lightweight

lightweight-spec-case ::= *generic-spec-case*

generic-spec-case ::= *spec-header [generic-spec-body]*

generic-spec-body ::= *(simple-spec-body-clause)+*

spec-header ::= *(requires-clause)+*

simple-spec-body-clause ::= *diverges-clause*
| *assignable-clause*
| *ensures-clause*
| *signals-only-clause*
| *signals-clause*

requires-clause ::= *requires-keyword pred-or-not* **;**

requires-keyword ::= **requires** | **pre**

pred-or-not ::= *predicate* | **not_specified** | **same**

C.5.2 Especificação Heavyweight

heavyweight-spec-case ::= *behavior-spec-case*
| *exceptional-behavior-spec-case*
| *normal-behavior-spec-case*

behavior-spec-case ::= *behavior-keyword generic-spec-case*

behavior-keyword ::=**behavior** | **behaviour**

normal-behavior-spec-case ::= *[* **privacy** *]* *normal-behavior-keyword normal-spec-case*

normal-behavior-keyword ::= **normal_behavior** | **normal_behaviour**

normal-spec-case ::= *generic-spec-case*

exceptional-behavior-spec-case ::= *[* **privacy** *]* *exceptional-behavior-keyword*
exceptional-spec-case

exceptional-behavior-keyword ::= **exceptional_behavior** | **exceptional_behaviour**

exceptional-spec-case ::= *generic-spec-case*

requires-clause ::= *requires-keyword pred-or-not* **;**

requires-keyword ::= **requires** | **pre**

pred-or-not ::= *predicate* | **not_specified** | **same**

ensures-clause ::= *ensures-keyword pred-or-not* **;**

ensures-keyword ::= **ensures** | **post**

signals-clause ::= *signals-keyword* **(** *reference-type [ident]* **)**
[pred-or-not] **;**

signals-keyword ::= **signals**| **exsures**

signals-only-clause ::= *signals-only-keyword reference-type* **(**, *reference-type)** **;**

| *signals-only-keyword* **nothing* **;**

signals-only-keyword ::= **signals_only**

diverges-clause ::= *diverges-keyword pred-or-not* **;**

diverges-keyword ::= **diverges**

when-clause ::= *when-keyword pred-or-not* **;**

when-keyword ::= **when**

assignable-clause ::= *assignable-keyword (ident (, ident) * | store-ref-keyword)* **;**

assignable-keyword ::= **assignable**
| **modifiable**
| **modifies**

callable-clause ::= *callable-keyword callable-methods-list* **;**

callable-keyword ::= **callable**

callable-methods-list ::= *method-name-list* | *store-ref-keyword*

store-ref-keyword ::= **nothing** | **everything** | **not_specified**

C.6 Estrutura de Controle Java Card

compound-statement ::= **{** *statement (statement)** **}**

statement ::= *compound-statement*
| *local-declaration* **;**
| *expression* **;**
| **if (** *expression* **)**
statement [**else** *statement]*
| **break** *[ident]* **;**
| **continue** *[ident]* **;**
| **return** *[expression]* **;**
| *switch-statement*
| *try-block*
| *loop-stmt*
| **throw** *expression* **;**
| **synchronized (** *expression* **)** *statement*
| **;**

switch-statement ::= **switch (** *expression* **) {**
*(switch-body)** **}**

switch-body ::= *switch-label-seq (statement)**

switch-label-seq ::= *switch-label (switch-label)**

switch-label ::= **case** *expression* **:** | **default** :

try-block ::= **try** *compound-statement (handler)**

[**finally** *compound-statement]*

handler ::= **catch (** *param-declaration* **)** *compound-statement*

local-declaration ::= *local-modifiers variable-decls*

local-modifiers ::= *(local-modifier)**

local-modifier ::= **ghost | final | non_null | nullable**

loop-stmt ::= **while (** *expression* **)** *statement*
| do *statement* **while (** *expression* **) ;**
| for (*[for-init]* **;** *[expression]* **;** *[expression-list]* **)**
statement

for-init ::= *local-declaration* | *expression-list*

C.7 Predicados e Expressões de Especificação:

predicate::= *expressionList*

expressionList::= *expression (AND expressionList)**

expression ::= *assignment-expr*

assignment-expr ::= *conditional-expr (assignment-op assignment-expr)**

assignment-op ::= **= | += | -= | *= | /= | %=**

conditional-expr ::= *equivalence-expr [* **?** *conditional-expr* **:** *conditional-expr]*

equivalence-expr ::= *implies-expr (equivalence-op implies-expr)**

equivalence-op ::= **<==> | <=!=>**

implies-expr ::= *logical-or-expr [* **==>** *implies-non-backward-expr]*
|
logical-or-expr **<==** *logical-or-expr (* **<==** *logical-or-expr)**

implies-non-backward-expr ::= *logical-or-expr*
[**==>** *implies-non-backward-expr]*

logical-or-expr ::= *logical-and-expr (* **||** *logical-and-expr)**

logical-and-expr ::= *equality-expr (* **&&** *equality-expr)**

equality-expr ::= *relational-expr (* **==** *relational-expr)**
| *relational-expr (* **!=** *relational-expr)**

relational-expr::= *additive-expr* **<** *additive-expr*
| *additive-expr* **>** *additive-expr*
| *additive-expr* **>=** *additive-expr*
| *additive-expr* **<=** *additive-expr*
| *additive-expr [* **instanceof** *additive-expr]*

additive-op ::= **+** | **-**

additive-expr ::= *mult-expr (additive-op mult-expr)**

mult-expr ::= *unary-expr (mult-op unary-expr)**

mult-op ::= ***** | **/** | **%**

unary-expr ::= **(** *type-spec* **)** *unary-expr*
| **++** *unary-expr*
| **–** *unary-expr*
| **+** *unary-expr*
| **-** *unary-expr*
| *unary-expr-not-plus-minus*

unary-expr-not-plus-minus ::= **!** *unary-expr*
| **(** *built-in-type* **)** *unary-expr*
| **(** *reference-type* **)** *unary-expr-not-plus-minus*
| *primary-expr*

primary-expr::= *ident* | *constant* |
T | *F* | *null_constant*
(*expression* **)**