Resumo

A plataforma Java 2 MicroEdition - J2ME - tem por objetivo definir um conjunto de tecnologias centradas no desenvolvimento de soluções para dispositivos com baixa capacidade de recursos, mas, pelo fato de carregarem algum tipo de micro-processador, são potencialmente capazes de realizar operações computacionais, necessitando na maioria dos casos, de uma implementação que garanta a confiabilidade dos sistemas desenvolvidos ou no mínimo uma certeza que essas aplicações funcionam de acordo com sua especificação.

Java Modeling Language (JML) é uma linguagem de especificação de comportamento de interface baseada na técnica de programação conhecida como Design By Contract, sendo usada para prover mecanismos de checagem de contratos para aplicações escritas em Java. Porém, existem alguns problemas na implementação de JML que não permite que este seja aplicado a outros domínios como o de dispositivos móveis. Para eliminar alguns desses problemas, esse trabalho de monografia propõe o uso de AspectJ, uma extensão Orientada a Aspectos de Java, para implementar os benefícios de Design by Contract. Nossa principal contribuição será tornar JML extensível bastante para que se possam aplicar as vantagens da técnica de Design by Contract na especificação de aplicativos para dispositivos móveis.

Abstract

The Java 2 MicroEdition platform - J2ME - aims to define a set of technologies focused on the development of solutions for devices with low capacity of resources, but for carrying some type of micro-processor, they are potentially able to perform computing operations, requiring in most cases, an implementation that ensures the reliability of the systems developed or at least a certainty that these applications work according to its specification.

The Java Modeling Language (JML), a behavioral interface especification language (BISL) designed to Java, is used to provide functional software correctness of Java applications. However ,there are a few drawbacks in JML implementation that it does not work properly when applied to others domains, such as small devices(PDAs and mobile phones) .This domains are constrained resources with high level necessity of functional correctness. In order to eliminate these drawbacks, this paper proposes the use of AspectJ, a general-purpose aspect-oriented extension to Java, to implement the features of design by contract (DBC) technique embedded in JML. Our main contribution is to make JML extensible in order to apply the benefits of design by contract technique to small devices domains.

Sumário

Índice de Figuras

Índice de Tabelas

Tabela de Símbolos e Siglas

J2ME – *Java 2 Micro Edition*
JML – *Java Modeling Language*
PDA – *Personal Digital Assistant*
BISL - *Behavioral Interface Especification Language*
DBC – *Design by Contract*
JVM – *Java Virtual Machine*
API – *Application Programming Interface*
MID – *Mobile Information Device*
CLDC - *Connected Limited Device Configuration*
MID – *Mobile Information Device Profile*
KVM – *Kilo Virtual Machine*
VM – *Virtual Machine*
WTK – *Wireless Too Kit*

Agradecimentos

Agradeço primeiramente a minha família que me apoiou durante esses cinco suados anos de vida universitária e que suportaram todos os momentos ruins por ela gerados.

Ao meu orientador, Ricardo Massa, pode ter acreditado na minha capacidade de realizar um bom trabalho e por compartilhar suas idéias afim de que pudéssemos contribuir de forma prática na melhoria da qualidade das atividades realizadas durante a realização desta monografia.

Ao meu companheiro, Henrique Mostaert, que me deu um grande apoio técnico na utilização das ferramentas utilizadas no trabalho, compartilhando juntos momentos de dedicação e esforço que resultaram na conclusão deste trabalho.

A todos os meus amigos e a minha namorada que, de forma direta ou indireta, contribuíram para que eu concluísse meu curso de graduação.

Capítulo 1

Introdução

A plataforma Java 2 MicroEdition - J2ME - tem por objetivo definir um conjunto de tecnologias centradas no desenvolvimento de soluções para dispositivos com baixa capacidade de recursos, como memória, energia, capacidade de processamento, etc. Por esta razão, o desenvolvimento de aplicações voltadas para tais dispositivos necessitam de cuidados especiais para evitar falhas por falta de recursos de hardware. Neste sentido, é importante o uso de mecanismos elevem os patamares de confiabilidade dos sistemas desenvolvidos ou que, ao menos, garantam que essas aplicações funcionam de acordo com sua especificação.

JML (Java Modeling Language) é uma linguagem formal de especificação comportamental para Java que contém notações essenciais no estilo design by contract. A principal idéia do design by contract é que entre as classes e seus clientes seja estabelecido explicitamente um contrato. Nele, o cliente deve garantir certas condições antes de invocar os métodos da classe (pré-condições), que por sua vez deve garantir algumas propriedades após ter sido executado (pós-condições).

Até o momento, as ferramentas desenvolvidas para JML tratam apenas de geração de *bytecodes* específicos para a máquina virtual JVM, ou seja, bytecodes gerados contém tipos específicos para a plataforma J2SE, portanto incompatível com os tipos de dados presentes na plataforma J2ME (máquina virtual KVM).

O objetivo deste trabalho é de criar um compilador de um subconjunto de assertivas JML . Esse compilador será baseado no reuso do compilador JML (jmlc), presente na suíte de ferramentas desenvolvida na Universidade de Iowa. Este reuso consiste em aproveitar o código

responsável por realizar a compilação das anotações presentes no código Java (frontend), adicionar uma nova verificação do subconjunto escolhido e implementar um nova geração de código baseada na técnica de programação conhecida como Orientação a Aspectos. Com essa estratégia, permitiremos tanto a compilação para a máquina virtual KVM, quanto verificação em tempo de execução de aplicações J2ME utilizando seu próprio ambiente de execução (emuladores de dispositivos móveis). É importante salientar que a estratégia de utilizar Orientação a Aspectos para implementação de JML é inédita.

A principal contribuição deste trabalho é de possibilitar a verificação durante a execução (runtime assertion checker) de asserções JML para aplicações da plataforma J2ME, proporcionando assim os benefícios do Design by Contract associados ao *runtime assertion checker* como uma ferramenta poderosa de especificação e verificação que será utilizada na prática pelos programadores de aplicativos J2ME.

Este trabalho faz parte de um projeto em conjunto com o aluno de Mestrado da Escola Politécnica de Pernambuco, Henrique Mostaert. No trabalho de Mostaert[22], foi utilizado uma estratégia de implementação que utiliza o próprio código do compilador JMLC para implementar os contratos de JML em AspectJ. Porém, esse trabalho propõe a utilização do gerador de analisador sintático JavaCC[23] para gerar o código em AspectJ das asserções JML.

Esta monografia possui a seguinte estrutura: no Capítulo 2 descrevemos a plataforma alvo envolvidas nas especificações comportamentais. Neste, capítulo descrevendo os conceitos relativos à plataforma J2ME como, as configurações presentes na plataforma, o perfil para os dispositivos, a máquina virtual Java utilizada, a arquitetura das aplicações, dentre outras.

No Capítulo 3 descreveremos as principais estruturas presentes em JML juntamente com uma visão da metologia de desenvolvimento de software adotada por ele conhecida como Projeto por Contrato. No Capítulo 4 mostraremos as principais características presentes no paradigma de Orientação à Aspectos, juntamente com os aspectos motivadores do uso de AspectJ em projetos de software.

No Capítulo 5, foi apresentada a proposta de mapeamento entre as estruturas de JML para AspectJ que serviu de base para a construção do compilador JMLC4ME, desenvolvido durante a construção deste trabalho. Finalmente, no Capítulo 6, foi apresentada a conclusão para nosso trabalho, bem como, as limitações, as dificuldades encontradas durante o seu desenvolvimento, trabalhos relacionados, e trabalhos futuros.

Capítulo 2

Plataforma J2ME

Neste capítulo, serão abordados conceitos sobre a plataforma J2ME, a qual tem sido largamente utilizada para desenvolvimento de aplicações para celulares. Descrevemos a plataforma J2ME, o MIDP (Mobile Information Device Profile), a configuração CLDC, a máquina virtual KVM, a ferramenta de simulação de dispositivos móveis *Wireless Toolkit* e suas funcionalidades.

2.1 Definição

J2ME[2][3] é uma tecnologia desenvolvida pela *Sun Micro Systems* que possibilita o desenvolvimento de aplicações para sistemas embarcados com recursos limitados de processamento e memória. Esta plataforma ficou muito famosa por não ser somente um bloco de software ou tampouco apenas uma especificação, ela se destaca principalmente por ser um conjunto de tecnologias e especificações que tem como alvo disponibilizar uma JVM (*Java Virtual Machine*), API(Application Programming Interface) e ferramentas para equipamentos portáteis em diferentes segmentos do mercado de dispositivos. Na Figura 1 podemos identificar a camada onde J2ME se aplica.

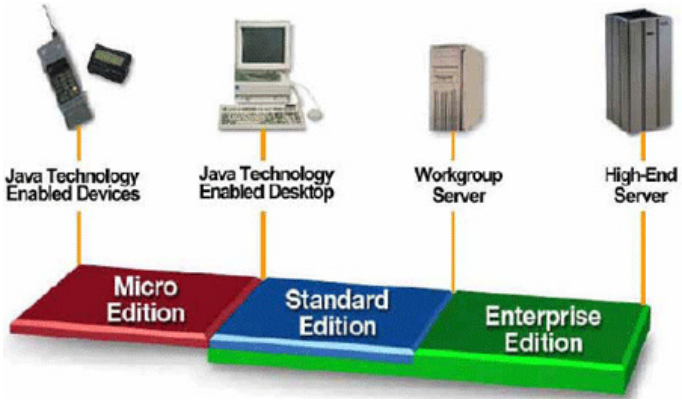

Figura 1. Camada de Aplicação de J2ME

O grande avanço na utilização e também no desenvolvimento de serviços providos para dispositivos móveis com J2ME pode ser explicado por vários motivos. O principal dele é que a plataforma da SUN é hoje a tecnologia que alcança o maior número de micro dispositivos disponíveis do mundo, independentemente do fabricante. Outra vantagem importante é que por ser uma plataforma gratuita tanto para download quanto para o uso comercial, os custos com o desenvolvimento são minimizados. Por fim, possui um banco de informações muito variado com uma documentação (API) muito bem elaborada.

2.2 Arquitetura

A arquitetura J2ME para dispositivos móveis é uma arquitetura modular e escalável que define camadas de configuração, perfil e uma camada que possui bibliotecas específicas fornecidas pelo fabricante.

No nível mais baixo a camada MID (*Mobile Information Device*) representa os recursos de hardware. Logo acima temos o Sistema Operacional nativo do sistema que geralmente vem embutido no dispositivo desde sua fabricação.

Na Camada de CLDC (*Connected Limited Device Configuration*), são especificadas a máquina virtual e um conjunto mínimo de recursos que devem ser implementados nos dispositivos que atendem à especificação J2ME.

Na camada de MIDP (Mobile Information Device), é fornecida uma série de API's de alto nível que combinadas com o CLDC prove um serviço completo para que as aplicações possam ser rodadas. Em outras palavras, o MIDP fornece uma API contendo classes Java que satisfazem os critérios de implementação descritos pela especificação do CLDC.

Na última camada dos componentes que representam a arquitetura de um aplicativo J2ME temos três sub-componentes: os aplicativos MIDP, os aplicativos específicos do OEM e os aplicativos nativos. Os aplicativos MIDP são aqueles que utilizam apenas classes específicas da plataforma J2ME como, por exemplo, as classes de interface gráfica do MIDP. Os aplicativos específicos do OEM são aqueles construídos com as API's auxiliares disponibilizadas pelos fabricantes de dispositivos. Os aplicativos nativos são aqueles que já pertencem ao dispositivo e que, em alguns casos, podem ser integrados aos sistemas operacionais nativos. A Figura 2 mostra a distribuição das camadas discutidas anteriormente.

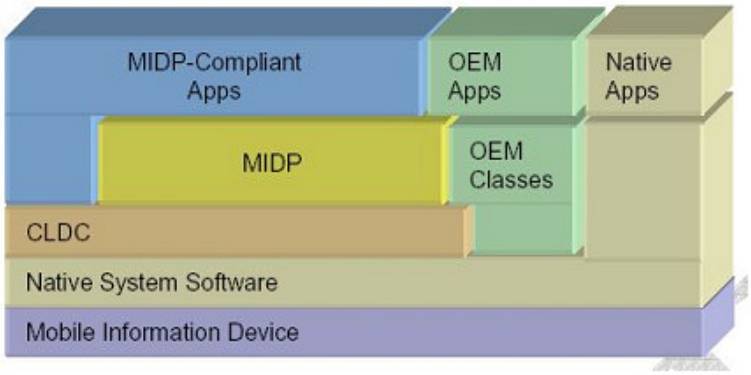

Figura 2. Arquitetura da plataforma J2ME

2.3 Especificação CLDC

A especificação CLDC é a camada base da plataforma de J2ME que rege as configurações para aparelhos extremamente pequenos como celulares, pagers, PDA's, ou qualquer outro que possua características comuns de produção em grande escala com restrições de capacidade de processamento, consumo de energia, quantidade de memória, entre outros. Ela especifica uma máquina virtual projetada para rodar em dispositivos com no mínimo de 160kb de memória,

processador de 16MHZ de processamento e possui uma conexão extremamente lenta, tipicamente 9.600 bps para aparelhos TDMA/CDMA e de 128kbps para GPRS/CDMA 1xRTT[1].

A configuração CLDC oferece a possibilidade de execução simultânea de múltiplas aplicações utilizando os recursos de processamento multitarefa do processamento nativo ou também pela criação de duas máquinas virtuais lógicas. Por outro lado, esta aplicação pode ser carregada em um sistema bem simples, onde só existe uma aplicação sendo rodada na JVM por vez. O mecanismo de execução de aplicações é independente para cada implementação da configuração CLDC, porém existe um requisito que é imposto pela máquina virtual de existir recursos capazes de fazer a carga dinâmica das classes que formam a aplicação. Opcionalmente, o sistema operacional pode oferecer a facilidade de salvar na memória as aplicações, eliminando a necessidade de carregar a aplicação através da rede sempre antes de iniciar a execução.

2.4 Perfil MIDP

O perfil MIDP (Mobile Information Device Profile) é uma extensão da API base fornecida pela configuração CLDC, implementa em software as restrições impostas por ela imposta. O conjunto de bibliotecas definidas por esse perfil aumenta os requisitos de quantidade de recursos disponíveis nos dispositivos determinados pela configuração CLDC. Essas extensões de bibliotecas são capazes de fornecer suporte ao desenvolvimento de aplicações utilizando diversos aspectos tais como: gerência de sons, transações seguros, interface com o usuário, sistema de armazenamento persistente.

Uma aplicação escrita em J2ME é composta por uma ou mais MIDlets. Um MIDlet é uma unidade de software que pode ser disponibilizada para utilização por um dispositivo e que pode realizar um objetivo específico. É comum definirmos durante a implementação da aplicação o ciclo de vida de um MIDlet, que pode ser listado nos seguintes estados conforme a Figura 3:

- **Ativada:** a MIDlet está em execução;
- **Pausada:** a MIDlet fica pausada quando há uma chamada ao seu construtor através do método pauseApp();

[1] É uma tecnologia de telefonia móvel 3G baseada em CDMA que dobra a capacidade dos atuais telefones móveis CDMAs, aumentando a capacidade de transmissão de dados para download de até 123Kbps.

- **Destruída**: quando a MIDlet libera os recursos e é desligada pelo gerenciador de aplicativos.

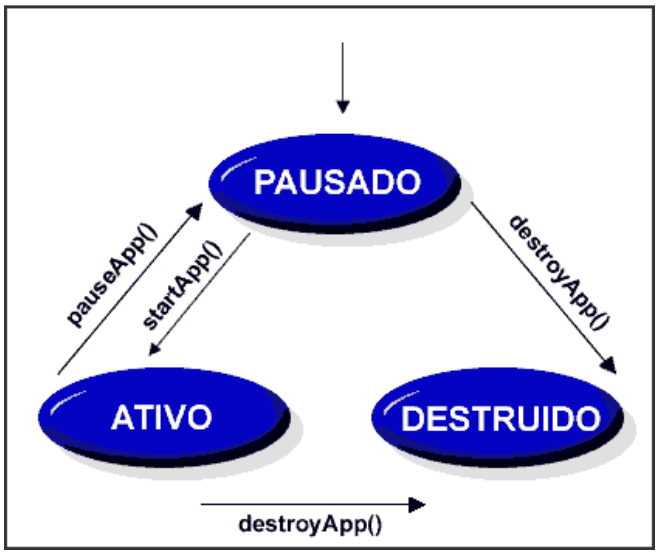

Figura 3. Clico de vida de um MIDlet

Assim que a MIDlet é invocada, o Application Manager (AM) invoca o método startApp(), o qual coloca o midlet no estado Ativo. Enquanto ela estiver executando o AM pode pausar ela invocando o método pauseApp() no caso de uma chamada sendo recebida, ou SMS chegando. A aplicação pode pausar a si mesma, bastando invocar o notifyPaused(). Assim como a AM pode pausar a aplicação e esta a si mesma, o mesmo pode ocorrer com o DestroyApp() que é invocado pela AM para fechar a aplicação ou ainda, pode ser fechada através da própria aplicação invocando notifyDestroyed().

As MIDlets devem ser projetadas para serem executadas em qualquer dispositivo contendo uma VM, porém na prática isso se torna muito difícil devido a grande variação de tamanho de tela, cores, teclados e outros aspectos existentes nos diferentes dispositivos Por isso, as aplicações são desenvolvidas com uma certa abstração na tela, pois os comandos de inserção de dados são feitos através dos botões dos dispositivos, e isto não é previamente sabido. As aplicações só descobrem isso em tempo de execução e se comportam de maneira apropriada para cada dispositivo. Já no desenvolvimento de jogos a aplicação é precisa ser bem mais específica, pois o desenvolvedor precisa conhecer o dispositivo previamente para melhor aproveitamento dos recursos, como a disposição dos objetos na tela por exemplo.

2.5 Kilo Virtual Machine

A KVM (Kilo Virtual Machine) é uma implementação da VM (Virtual Machine) otimizada para ser usada em dispositivos limitados e foi desenvolvida para ser facilmente portável. Essa Máquina Virtual foi projetada para obter os principais aspectos da máquina virtual clássica de Java, ou seja, ela aceita o mesmo conjunto de *bytecodes* e o mesmo formato para o arquivo ".class", porém com um conjunto restrito de instruções otimizado para dispositivos móveis.

A KVM foi projetada para ter uma quantidade mínima de memória, entre 40 a 80 kilobytes e ser tão completa e potente quanto possível sem afetar nenhum outro objetivo. Ela é implementada na linguagem C, o que torna possível ser portada para outras plataformas que possuam um compilador C. Todavia, muitos fabricantes adaptam seus produtos colocando instruções específicas na KVM ao invés de implementar sua própria VM.

A configuração CLDC adotou a KVM como sua máquina virtual de referência e a SUN a disponibiliza como parte desta configuração. Todavia, algumas características disponíveis na KVM que não são suportadas pela configuração CLDC são determinadas como opcionais, pois para uma configuração particular do dispositivo de determinada aplicação alguma característica específica pode ser desativada reduzindo consideravelmente a quantidade de memória que está sendo requerida. Além disso, a não inclusão ou exclusão total de características permite que a KVM seja alterada para outros sistemas ou para outras configurações poupando espaço de armazenamento e recursos de processamento que podem estar sendo desperdiçados. Dentre as principais características, podemos citar:

- Long Integer e Float
- Array Multidimensionais
- Threads e tratamento de eventos
- Start-up
- Class Loaders

2.6 Wireless Toolkit

A Sun Java Wireless Toolkit (anteriormente conhecida como J2ME Wireless Toolkit) é um conjunto de ferramentas para a criação de aplicações Java que são executados em dispositivos compatíveis com a tecnologia J2ME. O J2ME Wireless Toolkit contém ferramentas para criação de aplicações, utilitários e um emulador de dispositivo.

Com essa ferramenta é possível emular o comportamento esperado de um MIDlet utilizando um simulador de dispositivos móvel genérico oferecido pela SUN ou ainda obter um simulador específico que sua aplicação exija. Ainda é possível configurar toda a plataforma a ser utilizada escolhendo a versão tanto do MIDP quanto do CLDC apropriada e ainda definir quais as API's que serão disponibilizados durante a execução da aplicação. Isso é muito importante principalmente para poder reconstituir fielmente a simulação, antecipando e prevenindo possíveis falhas na execução do aplicativo.

Figura 4. Emulação de um MIDlet utilizando a WTK com dois diferentes simuladores.

Capítulo 3

Java Modeling Language

Neste capítulo apresentaremos os principais conceitos que estão por trás da linguagem de especificação Formal de Java.

3.1 Projeto por Contrato

Uma das características mais importante de um sistema computacional é que esse seja confiável, ou seja, robusto e correto. A corretude está intimamente ligada com a capacidade de um sistema de executar todas suas funções conforme o especificado e a robustez quando trata de situações inesperadas de uma maneira que não comprometa o seu funcionamento [12].

Sem dúvida o paradigma OO representou um grande avanço na melhoria das práticas do processo de desenvolvimento de software. Por outro lado, demorou-se um pouco para perceber que somente sua utilização não seria suficiente para a construção de melhores produtos de software. Foi necessário um certo tempo de amadurecimento para que os desenvolvedores percebessem que somente a utilização do paradigma OO não seria suficiente para garantir as características de corretude e robustez exigidas. Nesse contexto, *Design by Contract (DBC)* [17] surge como uma técnica de programação que vista construir sistemas OO mais confiáveis.

Escrever especificações formais de módulos de programas como classes e interfaces proporciona melhorias na qualidade do software auxiliando no processo de identificar as responsabilidades dos módulos (classes e interfaces) e as obrigações dos seus clientes. Conseqüentemente, identificar e especificar as responsabilidades dos módulos geralmente conduz para um melhor design que é fracamente acoplado e possui maior coesão. Estas características

são importantes na metodologia de construção de componentes, pois promovem maior reusabilidade e confiabilidade através das especificações formais.

Esse processo de identificar responsabilidades e obrigações dos seus respectivos módulos e clientes é a metodologia de desenvolvimento de software conhecida como *Design by Contract* (DBC) introduzida por Meyer na linguagem Eiffel. A principal idéia por trás do DBC é o fato de um módulo (classe ou interface) e seus clientes possuírem um "contrato" entre eles. O cliente deve garantir certas condições antes da chamada de um método definido em uma particular classe. Da mesma forma no retorno da chamada do método deve garantir certas propriedades que devem ser satisfeitas.

O uso de pré e pós-condições no desenvolvimento de software não é recente, tais conceitos foram originalmente introduzidos por Hoare, Floyd e Dijkstra no final da década de 60[24]. No entanto, o que mais vem chamando atenção neste método são as novas técnicas ferramentais de verificação que confrontam a especificação com o código do programa. Uma das formas de se fazer essa verificação é checar asserções em tempo de execução. Após o código ser devidamente instrumentado, com a ajuda de algum suporte ferramental, o software é capaz de identificar violações no seu contrato definido previamente. Outra técnica mais ambiciosa seria de realizar verificações estaticamente, onde a ferramenta de verificação estática tenta traças todos os caminhos possíveis de execução.

Por esses contratos serem mais abstratos do que o código, DBC torna-se uma forma de documentar software de maneira mais elegante do que se basear apenas no entendimento do código ou dos comentários feitos no programa. O contrato é uma especificação formal que define sem ambigüidades o comportamento esperado do sistema. Graças a essa característica é possível construir uma série de ferramentas automatizadas de suporte, fazendo com o que essa documentação possa ser mantida atualizada sempre.

Uma outra vantagem de se utilizar DBC é o fato dessa técnica facilitar a depuração do código. Com o uso dessa técnica, fica fácil localizar a causa de um possível mau funcionamento do software. Casa a violação de um contrato for na pré-condição de um determinado método, fica claro que o problema está localizado no código que o invocou. Caso a violação se localiza na pós-condição de um método, fica claro que o problema está na implementação deste método que não conseguiu garantir o contrato durante sua execução.

3.1.2 Herança

Com o uso da técnica de Projeto por Contrato podemos controlar de maneira eficiente um dos conceitos mais importantes do paradigma de Orientação a Objetos: polimorfismo e herança. Através desses conceitos, podemos criar novas classes a partir de classes existentes, porém o comportamento dos métodos herdados não precisa ser necessariamente o mesmo, ele pode ser redefinido totalmente ou parcialmente. Nesse contexto surge uma questão: como evitar que a definição do novo comportamento do método herdado vá de encontro à especificação contida no método da superclasse?

Para exemplificar essa situação, considere duas classes: Pessoa e Estudante, onde a última é subclasse da primeira. Devido à característica de polimorfismo, uma classe C que se comunica com o tipo Pessoa pode na verdade está lidando com uma instância da classe Estudante. O desenvolvedor da classe C está ciente que deve respeitar o contrato definido em Pessoa, porém desconhece completamente a existência de outras classes. Portanto, em tempo de execução, quando C passar a se comunicar com uma instância de Estudante, o contrato de um determinado método herdado pode ser distinto da especificação que consta na superclasse. Ou seja, C espera estar chamando um método respeitando um contrato, quando na verdade está se comunicando com outro completamente diferente. Na verdade existem duas formas que uma classe pode ferir o contrato especificado na superclasse:

- A subclasse poderia fazer com que a pré-condição se tornasse mais restritiva;
- A subclasse poderia fazer com que a pós-condição se tornasse mais permissiva, retornando um resultado menos satisfatório do que havia sido prometido a C;

Na verdade, o que DBC impõe é que nenhuma dessas situações impostas seja permitida. No Projeto por Contrato, uma redefinição pode enfraquecer a pré-condição e/ou fortalecer a pós-condição original sem que com isso haja algum prejuízo aos clientes da especificação da superclasse. Isso significa dizer que a subclasse faz um trabalho no mínimo tão bom quanto da superclasse, garantindo um mecanismo eficiente de redefinição de método, evitando que um procedimento seja transformado em outro completamente diferente através da herança.

3.1.3 Contratos e Asserções

Quando um desenvolvedor codifica um método de uma classe, ele o faz com a intenção que este cumpra uma tarefa específica em tempo de execução. A menos que essa tarefa seja trivial, a definição desse método pode incluir várias sub-tarefas. Cada uma dessas sub-tarefas pode ser codificada no próprio método ou pode se criar outros métodos para comportar tais funcionalidades. No segundo caso, o método principal atribui a outro a responsabilidade de cumprir determinada função através de uma chamada de método. Um número grande de chamada de métodos torna o código excessivamente fragmentado, enquanto que o contrário torna o código demasiadamente complexo. Apesar das duas abordagens possuírem vantagens e desvantagens, geralmente um código com muitos métodos e invocações é mais fácil de entender e de manter do que um código com métodos enormes.

Essa opção pelo uso de chamada de método é análoga a situação pela qual preferimos contratar um serviço de terceiros a fazermos a tarefa nós mesmos. Por exemplo, se quisermos entregar uma correspondência a alguém podemos fazer isso pessoalmente ou contratar um serviço de correio para tal. Cada parte espera algum benefício do contrato, ao mesmo tempo em que aceita cumprir algumas obrigações para obtê-los. Normalmente tais benefícios e obrigações estão expostos em um documento que deve proteger ambos as partes. A elaboração deste documento traz as seguintes vantagens:

- O cliente passa a contar com a especificação do que deve ser realizado;
- O contratado estabelece o mínimo aceitável para que o serviço seja considerado concluído. Ele não é obrigado a fazer nada que extrapole os limites da especificação;

Tabela 1. O que constitui uma obrigação para uma das partes geralmente torna-se um benefício para a outra.

Parte	Obrigações	Benefícios
Cliente	Encomendas com no máximo 5 kg. Cada dimensão não deve ultrapassar 2 metros. Pagar R$1,00 por grama.	Ter sua encomenda entregue no destinatário dentro de no máximo 4 horas.
Fornecedor	Entregar a encomenda no destinatário dentro de no máximo 4 horas.	Não tem o dever de lidar com encomendas muito grandes, muito pesadas ou que não foram pagas.

No contexto de desenvolvimento de software, temos que caso a execução de um método dependa de sub-tarefas implementadas em outros métodos, torna-se necessário especificar precisamente a relação entre cliente (quem invoca) e contratado (quem é invocado). Na técnica de Projeto por Contrato, o mecanismo para expressar as condições que devem reger o contrato de software é chamado de **asserções.** Pré-condições e pós-condições são asserções que definem respectivamente os direitos e as obrigações individuais de cada método. Invariantes de classes constituem outro tipo de asserção, que como o próprio nome denota, são propriedades que sempre são válidas ao longo do ciclo de um objeto.

3.2 Java Modeling Language

JML (Java Modeling Language) [8] [7] [6] é uma linguagem de especificação de interfaces e comportamentos (BISL, Behavioral Interface Specification Language) desenvolvida para a especificação de módulos Java (classes, interfaces e métodos).

JML segue a iniciativa do DBC presente na linguagem Eiffel e utiliza expressões da própria linguagem de programação para especificar o comportamento dos módulos dos programas [12]. Além disso, JML incorpora várias características e conceitos da abordagem de especificações orientadas a modelo (model-oriented) como VDM, Z e Larch. Por fim JML prover também algumas idéias do cálculo de refinamentos [5] [10].

A linguagem JML introduz um número de construtores para descrever o comportamento. Estes incluem *model, fields*, quantificadores, visibilidade de escopo para as asserções, pré-condições, pós-condições, invariantes, herança de contrato e especificações de comportamento normal versus comportamento excepcional [11]. Ao utilizar estes construtores para descrever o comportamento desejado dos módulos de classes e interfaces ,estaremos obtendo um ganho significativo no processo de desenvolvimento do software[9]. Dentre eles podemos citar:

- Descrição mais precisa do que o código se propõe a fazer;
- Eficiência na descoberta e correção de bugs;
- Descoberta antecipada de uso incorreto das classes pelos clientes;
- Documentação mais precisa que geralmente confirma a realidade do código da aplicação. (condizente);

3.2.1 Assertiva Requires

Através do uso da assertiva r*equires* podemos descrever pré-condições para execuções de métodos. Pré-condições são asserções que devem ser satisfeitas quando um método é chamado, ou seja, elas devem ser checadas antes que qualquer código do método seja executado.

A sintaxe para a utilização da assertiva requires é o uso da palavra-chave *requires* seguida de uma lista de operações lógicas. Normalmente esse tipo de assertiva é usado juntamente com outras assertivas JML, tais como predicados, para que juntos, detalhem melhor o comportamento de um determinado componente.

No exemplo abaixo, o método binarySearch() exige que o array ‘a’ não seja nulo e que esteja ordenado, ou seja, dado um elemento i qualquer do array, a[i-1] <= a[i] é sempre válido. Vale notar que o uso do predicado \forall retorna um booleano, que completa a expressão lógica

presente no *requires.* Se estas duas condições forem satisfeitas, o método pode enfim ser executado.

```
/*@ requires a != null
  @       && (\forall int i;
  @           0 < i && i < a.length;
  @           a[i-1] <= a[i]);
@*/
int binarySearch(int[] a, int x) {
//    ...
}
```

Figura 5. Uso da assertiva *requires* e o predicado \forall na construção da pré-condição do método binarySearch().

3.2.2 Assertivas Ensures

Pós-condições são asserções que devem ser satisfeitas quando um método termina. Essas pós-condições devem ser checadas depois que método em questão termina a execução. Entretanto, em uma linguagem como Java que suporta exceções, temos que realizar uma distinção entre pós-condição normal e excepcional.

Uma pós-condição normal de método verifica o que deve ser verdadeiro quando ele retorna normalmente sem lançar nenhuma exceção. Por outro lado, uma pós-condição excepcional de método verifica o que deve ser verdadeiro para que ele retorne lançando uma exceção especificada utilizando a cláusula *signals*.

A sintaxe para a utilização da assertiva ensures é o uso da palavra-chave *ensures* seguida de uma lista de operações lógicas que devem ser satisfeitas logo após a execução do bloco de código. Como na cláusula *requires,* esse tipo de assertiva é também pode ser usado juntamente com outras assertivas JML, tais como predicados, para que juntos, detalhem melhor o comportamento de um determinado componente.

No exemplo da figura 6, mostramos o uso da pós-condição normal através da palavra-chave ensures, exigindo que o retorno do método soma() seja a soma dois inteiros ‘a’ e ‘b’. Na linha seguinte, mostramos o uso da pós-condição excepcional, onde caso o retorno do método não for a soma dos inteiros ‘a’ e ‘b’ será lançado uma exceção do tipo *Exception* em tempo de execução.

```
//@ ensures \result == a + b;
//@ signals (Exception) \result != a + b;
   public int soma( int a, int b ){
    return a + b;
}
```

Figura 6. O uso da palavra-chave *ensures*

3.2.3 Invariantes

As invariantes descrevem propriedades que são sempre verdadeiras em todo estado visível de um bloco de código. Uma invariante qualquer deve ser interpretada como um compromisso assumido pela classe em não permitir que o valor de um determinado predicado assuma falso antes ou depois da execução de seus métodos e construtores. Ou seja, uma invariante não deve simplesmente expressar um desejo e sim o que a classe realmente é capaz de garantir. De acordo com JML, as invariantes podem ser precedidas dos modificadores *static* e *instance*. Da mesma maneira que um método estático na linguagem Java, uma invariante estática não pode referenciar um objeto através da palavra reservada *this,* o que conseqüentemente também impede seu acesso a campos e métodos não estáticos da classe.

Se uma invariante for declarada dentro de uma classe, como na Figura 7, essa é por *default* uma invariante de instância. Por outro lado, se sua declaração estiver dentro do escopo de uma interface a invariante é considerada uma invariante de estática. Essa distinção entre os tipos de invariantes afetam também a ordem de verificação em tempo de execução. Para invariantes estáticas valem as seguintes regras: (1) bloco de inicialização estática é obrigado a estabelecer a invariante, de modo que após sua execução a propriedade seja válida; (2) construtores, assim como todos os métodos (estáticos e não-estáticos) devem preservar a invariante.

```
public class Pessoa {

    private /*@ spec_public @*/ String nome;

    //@ public invariant nome!=null;
    //@ ensures nome == aNome;
    public Pessoa(String aNome){
        this.nome=aNome;
        }
}
```

Figura 7. Exemplo do uso de invariante de estância..

3.2.4 Compilador JMLC

O processo de compilação de um arquivo Java anotado em JML ocorre através do uso da ferramenta chamada JML Compiler (JMLC) [14]. Diferentemente do compilador tradicional de Java, que reconhece as especificações contidas no código como simples comentários de linha, o JMLC reconhece as anotações JML, verifica possíveis erros sintáticos e insere nos *bytecodes* gerados código específico JML.

O arquivo gerado pelo compilador é um arquivo *.class* normalmente de tamanho maior do que se fosse compilado utilizando o compilador tradicional de Java. Pelo fato deste arquivo final ser usado em dispositivos móveis com baixa capacidade de memória e de processamento, este se torna um dos grandes problemas que devem ser tratados nessa final de geração de código. Na figura 8 abaixo, mostramos a compilação de um código Java utilizando o console do DOS e utilizando a interface gráfica fornecida na ferramenta JMLC respectivamente.

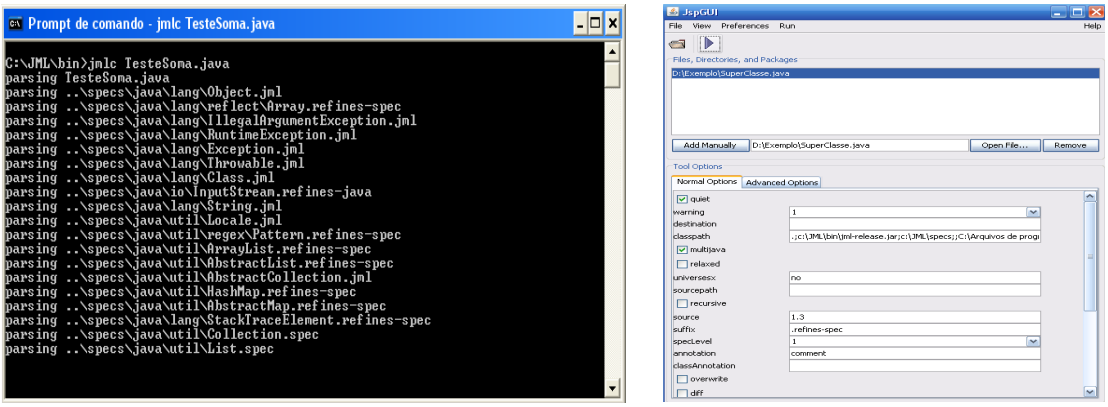

Figura 8. Utilização do compilador JMLC através do console(Direita) e da interface gráfica(Esquerda).

3.2.5 JML Runtime Assertion Checker

Essa ferramenta é responsável por identificar, em tempo de execução, possíveis inconsistências entre o código fonte e a sua especificação. Durante a execução do bloco especificado, o JMLRAC [14] (JML Runtime Assertion Checker) procura verificar se alguma condição viola o contrato que foi especificado anteriormente. Caso haja alguma violação, a ferramenta retorna uma mensagem de erro específica que ajuda ao desenvolvedor descobrir a origem do problema, que pode partir de uma pré-condição não satisfeita ou de uma pós-condição não garantida.

Considere um exemplo de especificação de um método ***testeSoma (int a, int b)*** que realiza a soma de dois inteiros positivos. O contrato estabelecido para o método garante que o resultado final será a soma dos parâmetros de 'a' e 'b'. Se por algum motivo, a implementação do método ferir o contrato estabelecido e retorna um resultado diferente do especificado, a ferramenta **jmlrac** é capaz de identificar a origem do erro através do lançamento da exceção **JMLNormalPostconditionError,** como ilustrado na figura 9**.**

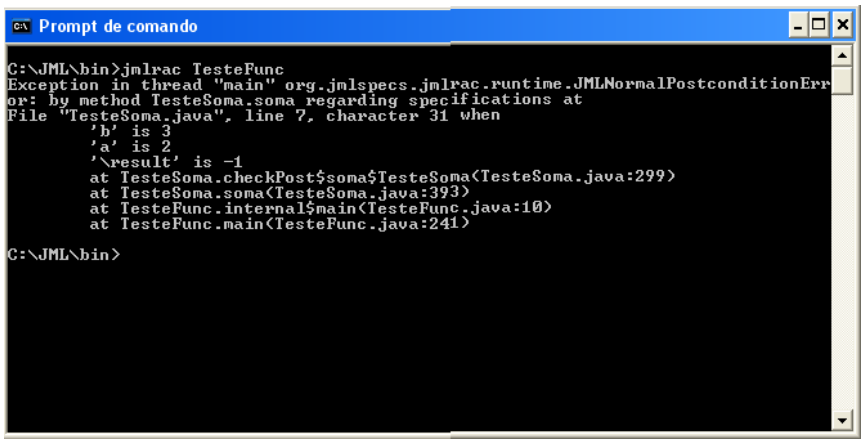

Figura 9. JMLRAC identificando um erro de pós-condição

3.2.6 Tipos de Especificações em JML

Em JML, não é necessário que o comportamento de um método ou classe seja especificado por completo. De fato, as especificações de software são classificadas em dois estilos dependo do nível de especificação requerido: *lightweight* e *heavyweight.*

O primeiro estilo descreve as especificações leves. Nesse estilo não é usado nenhuma palavra-chave que identifique o comportamento especificado. É a forma mais simples possível de especificação onde a sua principal característica é a incompletude, ou seja, não é necessário especificar todas as asserções disponíveis para descrever o comportamento desejado.

O segundo estilo descreve as especificações pesadas. Esse estilo é caracterizado por apresentar uma modelagem de comportamento mais completo do que o anterior. Porém nesse estilo é necessário que se use uma das três palavras-chaves que descrevem o tipo de comportamento desejado: **normal_behaviour, behaviour, exceptional_behaviour.**

A palavra chave "normal_behaviour" descreve um tipo de comportamento onde as situações excepcionais que podem ocorrer em tempo de execução não podem ser descritas na

especificação. A Figura 5 mostra um trecho de um código Java onde é demonstrado o uso de uma especificação "pesada" utilizando normal_behaviour:

```
/*@ normal_behaviour
  @ ensures \result == a + b;
  @*/
public int soma( int a, int b ){
    return a + b;
}
```

Figura 10. Uso da palavra chave "normal_behavior"

Por outro lado, se utilizarmos a palavra-chave exceptional_behaviour, deveremos apenas especificar o comportamento excepcional que o código poderá assumir. Na Figura 6 é mostrado um exemplo de que como se deve especificar um trecho de código quando a pós-condição do método soma () não é satisfeita:

```
/*@ exceptional_behaviour
  @ signals (Exception) \result != a + b;
  @*/
public int soma( int a, int b ){
    return a + b;
}
```

Figura 11. Uso da palavra chave "exceptional_behavior"

Caso a palavra-chave usada for do tipo behaviour, é necessário que o desenvolvedor construa uma especificação completa do funcionamento do código, tratando tanto condições normais de execução (pré e pós-condições) quanto condições excepcionais. Na Figura 7 é mostrado um exemplo de uma especificação completa do método soma() descrevendo todo comportamento esperado do método em tempo de execução:

```
/*@ behaviour
  @ requires ( a >= 0 ) && ( b >= 0 );
  @ assignable \nothing;
  @ ensures \result == a + b;
  @ signals (Exception) \result != a + b;
  @*/
public int soma( int a, int b ){
    return a + b;
}
```

Figura 12. Uso da palavra chave "behavior"

Capítulo 4

Programação Orientada a Aspectos

Neste capítulo apresentaremos os conceitos referente a Programação Orientada a Aspectos e o suporte ferramental utilizada na elaboração desse trabalho.

4.1 Introdução

A grande demanda no desenvolvimento de software de qualidade que atendesse níveis de reuso, manutenibilidade e mudanças de requisitos aumentaram o uso do paradigma de Orientação a Objetos (OA). Porém OA possui sérias limitações como o entrelaçamento e espalhamento do código que dificultam principalmente na manutenção do código quando há, por exemplo, mudanças grandes nos requisitos. Por outro lado, partes dessas limitações podem ser compensadas com uso da técnica de Padrões de Projetos ou com extensões do paradigma de Orientação a Objetos. Essas técnicas visam principalmente obter uma maior modularidade do software em situações que somente a OA não seja suficiente.

Uma das extensões de OA mais populares é conhecida como Programação Orientada a Aspectos (AOP). AOP procura solucionar a ineficiência em capturar importantes decisões de projeto que um sistema deve implementar que resulta no entrelaçamento e espalhamento do código com diferentes propósitos, tornando o sistema extremamente difícil de manter.

4.2 Interesses e Aspectos

Um dos elementos centrais da AOP é o conceito de Interesse, que são as características relevantes de uma aplicação. Um interesse pode ser dividido em uma série de aspectos que representam os requisitos. Os aspectos podem ser agrupados no domínio da aplicação, compondo os interesses funcionais, que formam a lógica de negócio, ou podem ser agrupados em elementos que prestam suporte aos interesses funcionais nomeados por interesses sistêmicos, e também chamados de ortogonais ou transversais. Dentro os principais interesses sistêmicos podemos citar: distribuição, gerenciamento de dados, controle de concorrência, tratamento de exceções, logging, debugging. A figura 13 abaixo mostra como seria a separação dos interesses não funcionais do sistema utilizando uma linguagem Orientada a Aspectos em comparação com uma linguagem Orientada a Objetos.

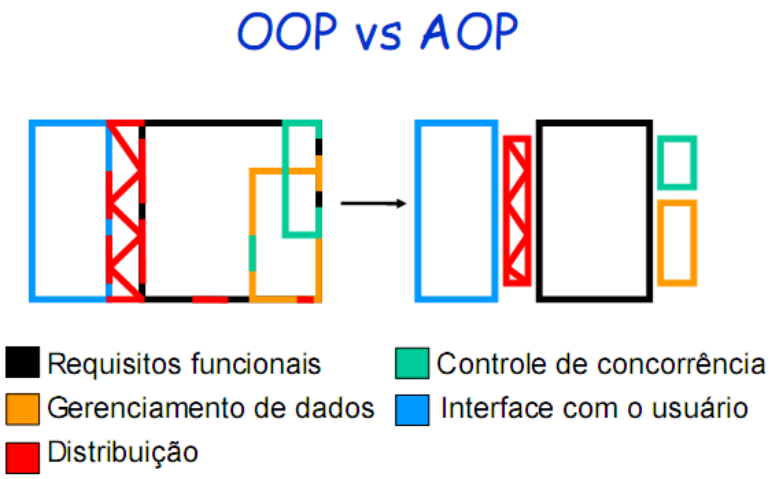

Figura 13. Separação de *concerns* utilizando OOP(esq.) e AOP(dir.)

Cada aspecto define uma função específica que pode afetar várias partes de um mesmo sistema. Ainda mais, um aspecto por ser uma classe Java, pode definir atributos e métodos ou ainda participar de uma hierarquia de aspectos. Aspectos podem alterar, por exemplo, a estrutura estática de um sistema através da técnica *static crosscutting,* onde é possível adicionar membros a uma classe (métodos, atributos, construtores) alterando a hierarquia do sistema.

Além de afetar o sistema estaticamente, aspectos também são capazes de modificar o sistema dinamicamente através da interceptação de pontos no fluxo da execução ou ainda obter o

controle total sobre a execução, como por exemplo, invocar a execução de métodos, execução de construtores, tratamento de exceções e acesso à atributos dentre outros.

4.3 Combinação Aspectual

A combinação aspectual, ou *weaving,* é o processo responsável por combinar os elementos escritos em linguagem de componentes (Ex. Java) com os elementos escritos em linguagem de aspectos. É um processo que antecede a compilação, gerando um código intermediário na linguagem de componentes capaz de produzir a operação desejada, ou de permitir a sua realização durante a execução do programa. Porém as classes referentes ao código do negócio nos sistemas não sofrem qualquer alteração para suportar a programação orientada a aspectos. Isso é feito no momento da combinação entre os componentes e os aspectos.

Weaver é o compilador responsável por gerar o aspecto de distribuição através da combinação aspectual do código fonte do sistema com seus respectivos aspectos, ou seja, compõe o núcleo do sistema com os aspectos relacionados. Esse processo de compilação é ilustrado na figura 14 abaixo.

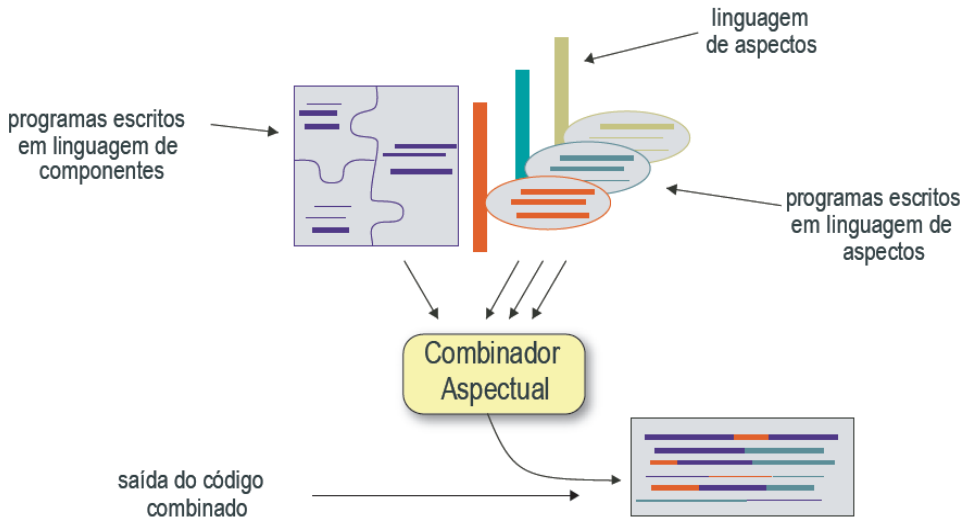

Figura 14. Processo de combinação aspectual

4.4 AspectJ

AspectJ [15] [16] é uma extensão orientada a aspectos de propósito geral da linguagem Orientada a Objetos Java. Foi criada pela Xerox Palo Alto Research Center em 1997 e posteriormente agregada ao projeto Eclipse da IBM em 2002. Além dos elementos oferecidos pela POO como classes, métodos, atributos e etc., são acrescentados novos conceitos e construções ao AspectJ, tais como: aspectos (aspects), conjuntos de junção (point cuts), pontos de junção (join points).

Por ser uma extensão de Java, o código gerado por programas AspectJ são bytecodes compatíveis com a especificação da JVM. Essa compatibilidade só é possível porque os compiladores de AspectJ transformam as construções específicas de AspectJ em código Java puro, todas as construções de AspectJ que afetam classes Java modificarão o código destas classes utilizando construções de Java. A seguir, são apresentados cada um dos conceitos e construções mais importantes que compõem o AspectJ.

4.4.1 Join Point

Um elemento crítico na concepção de qualquer Linguagem Orientada a Aspectos é a definição do modelo de join point, ou pontos de junção. Os pontos de junção são responsáveis por definir a estrutura dos interesses transversais do sistema [19].

Em AspectJ, pontos de junção são pontos bem definidos no fluxo de execução do programa. Podemos citar como exemplos de pontos de junção: chamada e execuções de métodos, acesso a atributos, tratamento de exceções, inicialização estática e dinâmica entre outros. A figura 15 abaixo identifica alguns pontos de junção no fluxo de execução de dois objetos.

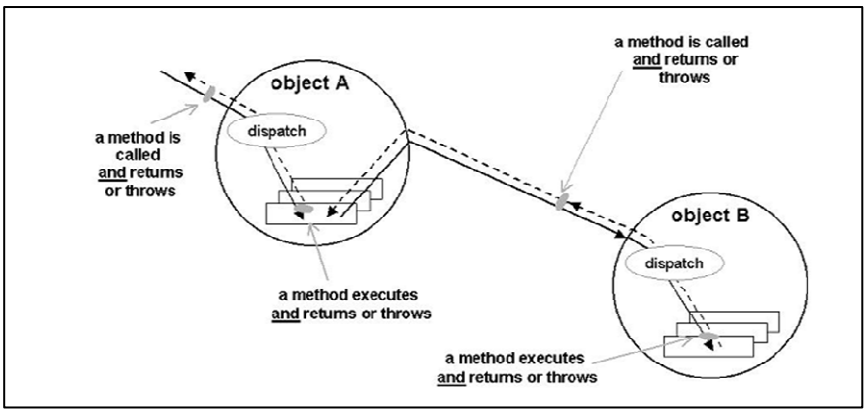

Figura 15. Identificação de pontos de junção

O primeiro *joinpoint* é a chamada de um método do objeto A, que pode retornar normalmente ou lançar uma exceção em tempo de execução. Outro *joinpoint* seria a execução desse método, que também poderia retornar com sucesso ou gerar uma exceção. Por outro lado, um método do objeto A poderia invocar um método do objeto B, definindo assim outro *joinpoint.* Esse é apenas um pequeno exemplo que demonstra o poder do uso de AspectJ no desenvolvimento de software, pois o desenvolvedor passa a obter o controle total da execução do sistema, podendo interferir em qualquer ponto que seja necessário utilizando as estruturas definidas na linguagem, como veremos nos tópicos a seguir.

4.4.2 Pointcut

Elementos usados para definir um ponto de junção, como uma espécie de regra usada para especificar eventos que serão considerados pontos de junção. Os *pointcuts*, ou pontos de atuação, têm como objetivo criar regras genéricas para definir os eventos que serão considerados pontos de junção, sem precisar defini-los individualmente (o que tornaria a POA quase sem sentido). Outra função dos pontos de atuação é apresentar dados do contexto de execução de cada ponto de junção, que serão utilizados pela rotina disparada pela ocorrência do ponto de junção mapeado no ponto de atuação [19].

Pointcuts podem ser formados pela combinação de joinpoints através dos operadores &&(E), || (OU), ! (não). Através dessa combinação, podemos, por exemplo, acessar valores dos argumentos dos métodos, atributos e exceções dos joinpoints, retornos de métodos e etc.

Para definir os *pointcuts,* AspectJ provê algumas estruturas conhecidas como designadores de pontos de junção como os apresentados na tabela abaixo:

Tabela 2. Designadores de pontos de junção (*pointcuts*)

Designador	Semântica
Call (Assinatura)	Invocação de método/construtor identificado por Assinatura
Execution (Assinatura)	Execução de método/construtor identificado por Assinatura
Get (*Assinatura*)	Acesso a atributo identificado por Assinatura

Set(*Assinatura*)	Atribuição de atributo identificado por Assinatura
This (*PadrãoTipo*)	O objeto em execução é instância de PadrãoTipo
Target (*PadrãoTipo*)	O objeto de destino é instância de PadrãoTipo
Args (*PadrãoTipo*)	Os argumentos são instâncias de PadrãoTipo
Within (*PadrãoTipo*)	O código em execução está definido em PadrãoTipo

4.4.3 Advices

Advices são pedaços da implementação de um aspecto executados em pontos bem definidos do programa principal (pontos de junção), definindo o código adicional que deverá executar nos pontos de junção definidos [19].

Os advices são compostos de duas partes: a primeira delas é o ponto de atuação (*pointcuts*), que define as regras de captura dos pontos de junção; a segunda é o código que será executado quando ocorrer o ponto de junção definido pela primeira parte. O advice é um mecanismo bastante similar a um método (quando comparamos com a programação orientada a objetos), cuja função é declarar o código que deve ser executado a cada ponto de junção em um ponto de atuação.

AspectJ provê algumas estruturas que auxiliam o desenvolvedor na definição dos *advices*. Essas estruturas estão relacionadas na tabela abaixo.

Tabela 3. Advices de AspectJ

Designador	Semântica
Before	Executa antes da computação de um *joinpoint*
After returning	Executa o *joinpoint* computar com sucesso
After throwing	Executa quando um *joinpoint* lança uma exceção
After	Executa após a computação de um *joinpoint*, em qualquer situação
Around	Executa quando um *joinpoint* é alcançado e tem total controle sobre sua execução

4.5 Static crosscutting

Até o momento, só analisamos construtores que nos permitem modificar a estrutura dinâmica de um sistema, ou seja, interferir no fluxo de execução de um sistema em tempo de execução. Porém, como mencionado na seção 4.2, AspectJ provêm também mecanismos de implementar *crosscutting* estático, afetando a estrutura estática do sistema através da técnica de *introduction.*

Um *introduction* é um membro de um aspecto, porém ele define ou modifica um membro de outra classe. Com esta técnica podemos, por exemplo:

- Adicionar métodos a uma classe existente;
- Adicionar atributos a uma classe existente;
- Extender uma classe existente;
- Implementar uma interface em uma classe existente;
- Converter exceções verificadas em exceções não verificadas;

Para exemplificar o uso de *introduction*, vamos supor que queremos modificar a classe *Point* para suportar *cloning*. Usando a técnica de *introduction* podemos efetuar essa modificação sem modificar a classe original. No exemplo abaixo o aspecto *CloneablePoint* realiza três funções:

- Declara que a classe Point implementa a interface *Cloneable;*
- Declara que os métodos em *Point,* cujas assinaturas são do tipo *Object clone(),* devem ter suas exceções checadas convertidas em exceções não-checadas;
- Adiciona um método que sobrescreve o método *Clone()* da classe *Point* que foi herdado de *Object*;

```
class Point {
       private int x, y;

       Point(int x, int y) { this.x = x; this.y = y; }

       int getX() { return this.x; }
       int getY() { return this.y; }

       void setX(int x) { this.x = x; }
       void setY(int y) { this.y = y; }

       public static void main(String[] args) {

          Point p = new Point(3,4);
          Point q = (Point) p.clone();
       }
   }
```

Figura 16. Classe *Point*

```
aspect CloneablePoint {
     declare parents: Point implements Cloneable;

     declare soft: CloneNotSupportedException: execution(Object clone());

     Object Point.clone() { return super.clone(); }
    }
```

Figura 17. Aspecto utilizando *introduction* que modifica estaticamente a classe *Point*

Capítulo 5

Estratégia de Especificação Formal

Neste Capítulo apresentaremos a estratégia de especificação formal proposta por esse trabalho na instrumentação de código Java destinado a plataforma J2ME.

5.1 Estratégia de Compilação

O compilador tradicional de JML, o jmlc, foi construído sobre o compilador MultiJava e reusa código fonte de outras ferramentas JML [14], especialmente do JML Type Checker que funciona como o front-end do jmlc. Por essa razão, somente dois passos de compilação são adicionados ao back-end dessa estrutura: o *Runtime assertion checker code generation* e o *Runtime assertion Checker code printing*.

JMLC utiliza um mecanismo de compilação chamado de "Double-round" compilation. Primeiramente, esse mecanismo é responsável por realizar um parser no arquivo com extensão ".java" usando uma extensão do compilador do compilador tradicional de Java, o MultiJava, e então verificar a sintaxe e a semântica das anotações JML do código e finalmente produzir uma árvore de sintaxe abstrata (AST). Essa árvore é utilizada pelos dois novos passos inseridos na compilação para gerar um arquivo Java temporário, completando a primeira metade do double-round compilation.

Para cada método Java presente no código passado para o compilador, três novos métodos de asserção são gerados nesse arquivo temporário: um para a checagem de pré-condição, e dois

para a checagem de pós-condição normal e excepcional. Finalmente esse arquivo temporário é compilado pelo MultiJava, completando assim o ultimo passo do double-round compilation. Os bytecodes instrumentados não só contem estruturas tradicionais ,que seriam gerados caso o código fonte fosse compilado utilizando o *Javac*, mas também métodos de asserção que serão utilizados para checagem de violação de contratos em tempo de execução.

O problema surge quando o código Java anotado foi desenvolvido utilizando a plataforma J2ME. Os bytecodes instrumentados e gerados pelo compilador JMLC são exclusivamente destinado ao uso da plataforma J2SE. Existem dois motivos que impedem o aproveitamento dessa ferramenta na instrumentação de código Java J2ME: uso de reflexão e existência de estruturas incompatíveis.

Reflexão é um recurso disponível no pacote "Java.lang.reflection" da API de J2SE que foi utilizado nas etapas de compilação descritos anteriormente para implementar a instrumentação e verificação de programas Java em tempo de execução para suportar especificação de herança. Porém, a API disponível para J2ME não oferece suporte a tal facilidade, impedindo o uso do código fonte nessa plataforma.

O outro problema trata do uso de estruturas incompatíveis com a plataforma J2ME. JML Compiler utiliza nos bytecodes instrumentados estruturas de dados, tais como *HashSet* e *Map*, ambas do pacote "Java.util", que não são suportados por J2ME, impedindo assim o aproveitamento do código instrumentado nesse tipo de plataforma.

Para solucionar os problemas descritos , esse trabalho propõe uma nova maneira de reestruturação dos dois passos inseridos por Cheon [13] no compilador JMLC. Essa nova estrutura consiste em modificar o *JML RAC Code Generation* para traduzir as anotações em JML para construções em AspectJ que são intuitavamente equivalentes. Por fim, o *JML RAC Code printing* é modificado para gerar um arquivo com a extensão ".aj" como um arquivo temporário.

Na nossa proposta, para cada método Java presente no código fonte será gerado o que chamamos de *aspect assertion methods* para checagem de pré-condições, pós-condições normais e excepcionais e invariantes. Finalmente, o arquivo gerado em AspectJ é finalmente compilado utilizando o processo de *Weaving* descrito na seção 4.3 deste trabalho.

Como os bytecodes gerados nesse processo são totalmente compatíveis com a plataforma J2ME, a instrumentação de código Java para dispositivos móveis utilizando JML se torna possível.

5.2 Mapeamento

Nesta seção, apresentaremos uma série de regras de tradução de notações escritas em JML para um código semanticamente equivalente em AspectJ.

No mapeamento proposto, iremos utilizar uma classe genérica **S** que declara um método **m()** com a assinatura **T m (T1 x1,...,xn Tn xn) throws E1,...,Em.** Considere também que este mesmo método **m()** possui uma especificação escrita em JML com as pré-condições **Pi**, pós-condições normais **Qi**, pós-condições excepcionais **Ri**, onde i varia de 1 até k.

5.2.1 Mapeamento de Pré-Condição

Na semântica de JML, uma pré-condição tem que ser satisfeita antes de qualquer código presente no método seja executado, caso contrário deve ser lançada uma exceção informando que houve uma violação no contrato da especificação. Todas as pré-condições locais do método ou herdadas de uma superclasse são combinas em JML por **disjunção.** Isso significa que a pré-condição resultante do método m() descrito anteriormente será dada por **P = Pi || .... || Pk** .

O código em AspectJ que verifica o comportamento das pré-condições do método m() utiliza um *advice* do tipo *before* para checar se as condições necessárias para a execução do método serão satisfeitas. Os pontos de execução que serão afetados são definidos pelos designadores *execution* e *within*. No primeiro designador, é descrito a assinatura do método que será interceptado, no nosso exemplo a assinatura do método m() é explicitada. O segundo designador indica que esse comportamento não afetará suas subclasses, ou seja, somente a classe S será interceptada por esse *advice*. Também é necessário declarar um parâmetro *current* para expor alguma informação do contexto da execução. No mapeamento proposto, usamos *current* para acessar o método que checa se a pré-condição foi satisfeita. Caso tais condições sejam violadas, será lançada uma exceção do tipo *JMLInternalPreconditionError*, similar ao comportamento do compilador tradicional de JML, o jmlc. A estrutura geral do código em AspectJ é apresentada abaixo.

```
before ( S current, T1 x1,...,Tn xn):
    execution (T S.m (T1,...,Tn)) &&
            within (S) && this (current) && args (x1,...,xn){
                if (!current.check$m$S (x1,...,xn)){
                    throw new JMLInternalPreconditionError();
                }
            }
```

O método check$m$S (x_1,...,x_n) responsável por checar se a pré-condição do método m() foi satisfeita é inserido na classe S através da técnica de AspectJ chamada de *introduction*, onde podemos inserir um método de uma maneira indireta com o auxilio de um aspecto do tipo *static crosscutting*. A definição desse método é tratada pelo compilador JMLC4ME de duas maneiras dependendo se o método herda especificações de uma superclasse. Caso o método m() seja herdado, além de checar as pré-condições locais do método, é necessário checar todas as especificações herdadas de supertipos por disjunção. O código em AspectJ que checa esse comportamento é apresentado abaixo.

```
public S.check$m$S  (T1 x1,...,Tn xn)  {
        return P1 || ... || Pk ||
        checkPre$m$SuperClass ||
        checkPre$m$SuperInterface 1,...,||
        checkPre$m$SuperInterface j;
}
```

Caso o método não tenha herdado especificações, ou seja, método não é sobrescrito, é necessário somente a checagem das pré-condições locais do método. O código em AspectJ que checa esse comportamento é apresentado abaixo.

```
public S.check$m$S  (T1 x1,...,Tn xn)  {
        return P1 || ... || Pk  ;
}
```

5.2.2 Mapeamento de Pós-Condição

Pós-Condições são propriedades que devem ser satisfeitas após a execução do método. Em JML existem dois tipos de pós-condições: pós-condições normais e pós-condições excepcionais. Pós-condições normais são propriedades que devem ser satisfeitas quando o método termina sua execução normalmente, sem lançar nenhuma exceção em tempo de execução.

Pós-condições normais avaliam o predicado em JML **\old(Pi) ==> Qi** , onde Qi é uma pós-condição normal e Pi é sua pré-condição correspondente que deverá ser satisfeita antes da avaliação de Qi . Note que a avaliação de Pi é feita num antes da execução do método, para evitar que alguma rotina do método altere o valor dos parâmetros originais passados no momento da chamada da função. Para realizar esse comportamento, JML usa o predicado **\old** na verificação de pós-condições. De acordo com a semântica de JML, diferentemente das pré-condições, todas as pós-condições normais de um método são combinadas por **conjunção** gerando uma única pós-condição representada pelo predicado **(\old(P) ==> Q) ≡ (\old(Pi) ==> Qi) .... (\old(Pk) ==> Qk).** Essa equação demonstra que pós-condição JML depende diretamente da avaliação da pré-condição em um estado anterior o da execução, ou seja, é necessário salvar os valores originais antes da execução do código especificado e checar o resultado da expressão para verificar se a pós-condição foi alcançada ou não.

Por outro lado, as pós-condições excepcionais são propriedades que devem ser satisfeitas quando um método lança alguma exceção em tempo de execução. Neste caso, o predicado a ser avaliado é **\old(Pi) ==> Ri** ,onde Ri é uma pós-condição excepcional e Pi é sua pré-condição correspondente que deverá ser satisfeita antes da avaliação de Ri. Cada pós-condição excepcional consiste em uma série de cláusulas *signals* da forma **(X11 e11) ... (Xkl ekl)** , onde **Xij** é um tipo de exceção e **eij** é uma variável que refere ao tipo de exceção a ser lançada. Analogamente a pós-condição normal, a pós-condição resultante também é feita por **conjunção** resultando no predicado **(\old(P) ==> R) ≡ (\old(Pi) ==> Ri) .... (\old(Pk) ==> Rk).**

Para exemplificar o mapeamento, suponha que o método m() definido na seção 5.3, possua uma série de expressões vi,...,vk do tipo \old(E), onde E é a avaliação da pós-condição normal ou excepcional do método. Esses valores precisam ser salvos em variáveis locais antes da execução do método, para evitar efeitos colaterais no valor das variáveis. O código em AspectJ que representa o comportamento da verificação de pós-condições normais e excepcionais é apresentado abaixo.

```
T around ( S current, T1 x1,...,Tn xn) throws E1,...,En :
    execution (T S.m (T1,...,Tn)) &&
        this (current) && args (x1,...,xn) {
            T rac$result; // representa o retorno do método
          << salvaOldValues >>                                    (1)
           try {
           // executa o método original
           rac$result = proceed(current, x1,...,xn);
           << checaPosCondiçãoNormal >>                           (2)

           } catch (Throwable rac$e) {
              << relança JMLException >>                          (3)
              << checaPosCondiçãoExcepcional >>                   (4)
           }
}
```

A estrutura geral do aspecto que checa as duas formas de pós-condições em JML usa o *advice* do tipo *around*, que obtém total controle sobre a execução do método referido. O método **proceed()** de AspectJ é utilizado para representar a chamada da execução do método e obter o valor de retorno, separando assim o fluxo de execução em dois estados: antes da chamada do método e depois da execução do método.

Em (1), salvamos os valores das *old expressions* antes da chamada do método através da declaração de variáveis locais old$$v_i$ da seguinte maneira:

```
<< salvaOldValues >> ≡
 old$v1 = v1
   .....
 old$vk = vk
```

O advice *around* é usado também para inserir o comportamento de checagem de lançamento de exceções usando um bloco try/catch, o que seria impossível se usássemos outro tipo de advice presente em AspectJ. Após a chamada do método utilizando **proceed(),** o

compilador gera o seguinte código em aspecto para checar a presença de violação da pós-condição normal (2):

```
<< checaPosCondiçãoNormal >> ≡
    If ( ! (  ! (P1[[old$vi]] ) || Q1[[old$vi]]  ) &&
                ... && ( ! (Pk[[old$vi]] )  || Qk[[old$vi]] ) )  {
        throw new JMLInternalNormalPostconditionError();
       }
```

O código acima representa a avaliação da pós-condição normal **\old(Pi) ==> Qi** . A expressão Java (!Pi || Qi) representa a tradução da implicação de JML usada na avaliação da pós-condição.

Caso o método termine de maneira anormal, ou seja, lança algum tipo de exceção, ela é capturada pelo bloco try/catch e tratada de duas maneiras: caso a exceção seja do tipo JMLInternalPostConditionError, relançamos a exceção para notificar o desenvolvedor que houve uma violação no contrato de pós-condição excepcional. O código que representa esse comportamento foi rotulado em (3).

```
<< relançaJMLException >>
if(rac$e instanceof JMLInternalNormalPostconditionError ) {
        throw (JMLInternalNormalPostconditionError) rac$e;
}
```

Por outro lado, se a exceção não for do tipo JMLInternalPostConditionError, então o compilador JMLC4ME realiza a avaliação de cada pós-condição excepcional Ri e combina os resultados por conjunção. Note que para cada Ri, sua exceção correspondente Xij tem que ser instância da exceção lançada.

A variável rac$v guarda o resultado da conjunção de todas as pós-condições avaliadas. Caso seja verdadeira, será lançada uma exceção do tipo *JMLInternalExcepcionalPostConditionError*, similar ao comportamento do compilador jmlc. Caso contrário, a exceção capturada é relançada. O código em AspectJ que representa o comportamento descrito anteriormente é apresentado abaixo.

```
<< checaPosCondiçãoExcepcional  >> ≡
        boolean rac$v = true;
        boolean rac$pre1 = P1[[old$vi]];
        if(rac$v && rac$pre1){
            if(rac$e instanceof X11) {
              boolean flag1 = true;
              X11 e11 = (X11)rac$e;
               flag1 = R1[[old$vi]];
                        rac$v = rac$v && flag1;
               }
              ...
              }
              boolean rac$prek = Pk[[old$vi]];
              if(rac$v && rac$prek) {
                 if(rac$e instanceof Xk1){
                    boolean flagk1 = true;
                    Xk1 ek1 = (Xk1) rac$e;
                    flagk1 = Rk[[old$vi]];
                     rac$v = rac$v && flagk1;
                    }
                  ...
               }
             if(!rac$v){
              throw new JMLInternalExceptionalPostconditionError();
              }
             else{
               if(rac$e instanceof X11){
                   throw (X11) rac$e;
                 }
               ...
                if(rac$e instanceof Xkj){
               throw (Xkj) rac$e;
                }
        }
```

A respeito da especificação de herança nas pós-condições, a ausência de **within (S)** na declaração do advice da página 42 força o aspecto a afetar as execuções de m() nos subtipos de **S**. Conseqüentemente, todas as pós-condições normais ou excepcionais serão automaticamente checadas e combinadas por conjunção. É importante ressaltar que o compilador só gerará código em Aspecto se o método possuir uma pós-condição anotada em JML. Caso contrário, nenhum código referente à pós-condição será gerado.

5.2.3 Mapeamento de Invariantes

De acordo com a semântica de JML, invariantes expressam propriedades globais que devem ser preservadas por todas as rotinas da classe antes, durante, após a execução e até mesmo no lançamento de exceções. Em JML podemos ainda ter dois tipos de invariantes: estáticas e de instância.

Para complementar o nosso exemplo, suponhamos agora que a nossa classe S possui uma série de invariantes $InstInv_1,...,InstInv_k$. Da mesma maneira que as pós-condições, invariantes de classes são combinadas por conjunção. Logo a invariante resultante da classe S seria $InstInv \equiv InstInv_1 \ldots InstInv_k$.

O código em AspectJ que representa o comportamento de invariantes escritas em JML é um aspecto *advice* do tipo *before* , declarado para checar as invariantes antes da execução de qualquer método. Declaramos também, outros dois *advices* do tipo *after returning e after throwing* para checar invariantes quando o método termina normalmente e para quando o método lança algum tipo de exceção, respectivamente. A cláusula (!static * * (..)) define que o designador *execution* especifica que o *advice* se torna aplicável para todos os métodos não estáticos da classe. O código que descreve esse comportamento é mostrado abaixo.

```
before ( S current):
execution (!static * *(..)) &&
   this (current) {
     << checkInstanceInvariant >>
     }
     after (S current) returning (Object o):
     execution (!static * *(..)) &&
     this (current) {
```

```
    << checkInstanceInvariant >>
    }
  after ( S current) throwing (Throwable rac$thrown):
   execution (!static * *(..)) &&
       this (current) {
                <<rethrowJMLException>>
        else
                << checkInstanceInvariant >>
```

O primeiro rótulo << checkInstanceInvariant >> checa a corretude das invariantes antes da execução do método através da conjunção das invariantes locais. Caso essa conjunção retorne falsa, uma exceção do tipo JMLInvariantError é lançada indica que houve alteração no valor da invariante. O código que mostra esse comportamento é o mesmo gerado para o *advice after returning*.

```
<< checkInstanceInvariant >>
        if ( ! ( InstInv1 && ... && InstInvn))  {
         throw new JMLInvariantError();
        }
```

Caso o método termine de maneira anormal, lançando uma exceção em tempo de execução, o compilador é responsável por gerar o seguinte código em AspectJ.

```
<< rethrowJMLException >>
        if (rac$thrown instanceof JMLInternalPreconditionError) {
                throw (JMLInternalPreconditionError) rac$thrown;
        }
        ...
        else if (rac$thrown instanceof JMLInvariantError) {
        throw (JMLInvariantError) rac$thrown;
        }
```

Se a exceção lançada for uma instância de qualquer violação de JML, então o código gerado relança as exceções. Caso contrário, a invariante precisa ser checada novamente chamando o novamente o código definido no rótulo *<< checkInstanceInvariant >>* .

5.3 Compilador JMLC4ME

O compilador JMLC4ME foi desenvolvido com o auxílio de um gerador de parsers Java conhecido como JavaCC[23], com o propósito de realizar uma análise léxica e sintática de um arquivo Java anotado com JML e realizar a geração de um código em AspectJ semanticamente equivalente. O primeiro passo do processo de compilação é submeter o código Java ao *JML Type Checker* do compilador JMLC. Caso o código seja verificado corretamente, o mesmo é passado para a fase de extração das anotações JML e geração de código temporário em Aspectos (extensão .aj). É importante ressaltar que essa fase da compilação só aceita um arquivo JML válido, validado pelo *TypeChecker* da fase anterior.

Na próxima etapa do processo, o arquivo ".aj" e o arquivo original Java original é passado para um compilador de AspectJ ,da preferência do desenvolvedor, que realiza o processo de weaving dos arquivos. Os códigos JML presentes no arquivo original são ignorados completamente pelo compilador de AspectJ por se tratarem de notações de comentários do tipo linha única (//) ou linhas múltiplias (/* comentários */). Finalmente, os bytecodes são gerados com estruturas e tipos compatíveis com a plataforma J2ME, podendo então ser utilizado com algum emulador de dispositivos móveis, como a Wireless Toolkit discutido na seção 2.6. Todo processo de compilação funciona com uma caixa preta para o desenvolvedor. A princípio, a informação de como o código está sendo tratado e manipulado pelo JMLC4ME fica imperceptível. Portanto, possíveis mensagens de erros de compilação precisam ser referentes às anotações de JML e não de Aspectos. A Figura 18 explica graficamente o processo de compilação descrito nesta seção.

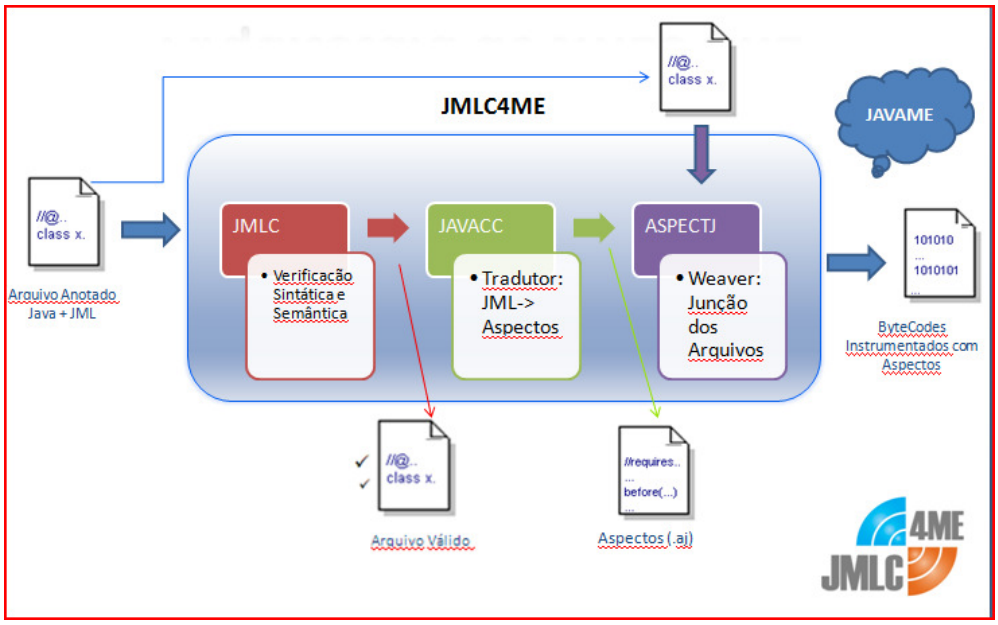

Figura 18. Estratégia de especificação formal para dispositivos móveis usando o compilador JMLC4ME.

5.3.1 Etapa de Geração de Código em Aspectos

A geração de código em aspectos no processo de compilação descrito anteriormente é feita da seguinte maneira: as classes Java são carregadas e organizadas hierarquicamente em níveis, partindo das classes que não são instanciadas até as superclasses. A geração de código parte do topo dessa hierarquia até os níveis mais baixos.

Nas superclasses, o parser procura métodos Java que possuam especificações JML e realiza a tradução para AspectJ no final da leitura do bloco (especificações + declaração do método). O arquivo produzido com extensão ".aj" é gerado para cada classe, portanto, se tivermos 10 classes Java especificadas, serão gerados 10 arquivos em Aspectos referentes aquelas classes. O processo de tradução segue os passos descritos na figura 19.

Figura 19. Fluxograma de geração de código em Aspectos

Nas subclasses o tratamento da geração de código segue duas regras: (1) quando o método é estático e sobrescrito, (2) quando método é não estático e sobrescrito. De acordo com a semântica de JML, métodos estáticos não herdam especificações de superclasses mesmo se o método for sobrescrito. Portanto, na geração do código em Aspecto que representa a checagem da pré-condição do método em questão, basta somente realizar a checagem das pré-condições locais de acordo com o mapeamento descrito na Seção 5.3.1. Caso o método seja não estático e sobrescrito, será necessário realizar a chamada explícita ao método que checa a pré-condição do método correspondente na superclasse. Os outros métodos da classe (não estáticos e não sobrescritos), que não seguem essas duas regras, são mapeados diretamente utilizando o mapeamento da superclasse.

5.3.2 Exemplo de Mapeamento

Para exemplificar o uso do compilador JMLC4ME, realizamos a especificação formal na linguagem JML do MIDlet CalculatorMIDlet disponível na página oficial da plataforma J2ME da Sun (Apêndice A). As especificações contidas nesse arquivo foram validadas pelo compilador JMLC e foram introduzidas como entrada para a etapa de geração de código em Aspecto do compilador proposto. No código abaixo, é mostrado o aspecto gerado referente à especificação do método Sub() da classe CalculatorMIDlet.java .

// Validação da pré-condição do método Sub

```
public boolean CalculatorMIDlet.check_precondition_sub(double a , double b){
            return (b<=a);
                  }


      before(CalculatorMIDlet current,double a , double b ) :
            execution(double CalculatorMIDlet.sub(double,double )) &&
              within(CalculatorMIDlet) &&
                  this(current) && args(a,b) {

              if (!current.check_precondition_sub(a,b)) {
                    throw new JMLInternalPreconditionError();
                  }
      }
```

// Validaçao da pós-condição do método Sub

```
public boolean CalculatorMIDlet.check_posCondition(double a , double b
               ,double result) {

            return !(b<=a) || (result == a-b);

       }

double around (CalculatorMIDlet current,double a , double b ) :
            execution(double CalculatorMIDlet.sub(double,double )) &&

            this(current) && args(a,b) {

               try {

               double oldA = a;
               double oldB = b;

               double result = proceed(current,a,b);

               if(!current.check_posCondition(oldA,oldB,result)){
                         throw new JMLInternalNormalPostConditionError();
                        }

               }
               catch (Throwable e) {
                   // Method does not throw exceptions
               }

               return result;
      }
```

Capítulo 6

Conclusões

O desenvolvimento de aplicativos para dispositivos com baixa capacidade de recursos, além das limitações impostas pelo hardware, em algumas situações, necessitam de uma implementação que prime pela corretude das aplicações construídas. Além disso, as aplicações precisam ser o mais simples possível pelo fato de que as aplicações complexas são geralmente maiores, e com isso, pode ocorrer um baixo desempenho da aplicação durante a sua execução.

Esse trabalho propõe um novo processo de compilação que possibilite que programas escritos em J2ME possam ser anotados utilizando uma linguagem de especificação formal como JML. A estratégia proposta irá adicionar mais uma fase no processo de compilação de programas escritos em JML que será a fase de geração de um código semanticamente equivalente ao escrito utilizando JML utilizando a técnica de programação chamada de Orientação a Aspectos.

Com essa abordagem, será possível desenvolver software para dispositivos móveis utilizando todo potencial de uma linguagem de especificação formal poderosa como JML para garantir que as aplicações funcionem exatamente da maneira como foram especificados.

6.1 Contribuições

A principal contribuição deste trabalho é de possibilitar a verificação durante a execução (runtime assertion checker) de asserções JML para aplicações da plataforma J2ME, proporcionando assim os benefícios do Design by Contract associados ao *runtime assertion checker* como uma

ferramenta poderosa de especificação e verificação que será utilizada na prática pelos programadores de aplicativos J2ME.

A segunda contribuição esperada é introduzir uma nova abordagem para engenharia de um runtime assertion checking para JML. Ou seja, o uso dos benefícios da programação orientada a aspectos, especificamente AspectJ, no intuito de solucionar questões de otimização e geração de código compatível com aplicativos J2ME. Vale salientar que esta contribuição resolva o problema específico do *runtime assertion checker* de JML para J2ME, entretanto o subconjunto JML implementado e otimizado com a abordagem proposta poderá ser utilizado para aplicações J2SE de forma simétrica.

A limitação principal dessa ferramenta é de que ela abrange , por enquanto, algumas estruturas presentes em JML. Caso seja utilizada a ferramenta para especificar software com estruturas não presentes no mapeamento proposto, a ferramenta deverá informar qual estrutura utilizada não é compatível com a versão atual da ferramenta.

Finalmente, este projeto proverá para a comunidade Java/JML uma nova abordagem para *runtime assertion checker* para aplicativos J2ME, que consiste em uma ferramenta que auxilia na fase de desenvolvimento (*debugging tool*) ou mesmo na fase de produção, constituindo assim uma ferramenta poderosa de Design by Contract para Java.

6.2 Trabalhos Relacionados

Feldman[20] apresenta uma ferramenta de DBC para Java chamada *Jose*. Essa ferramenta adota uma linguagem de DBC para expressar contratos de software. Similar a nossa estratégia, *Jose* utiliza AspectJ para implementar DBC. Porém, a semântica de JML difere da linguagem de DBC descrita no trabalho de Feldman em dois aspectos: pós-condições e invariantes. Jose define que as pós-condições são combinadas por conjunção sem levar em consideração a pré-condição correspondente. Além disso, a ferramenta estabelece que métodos privados podem modificar invariantes. Como vimos, a semântica de JML não permite nenhuma das situações descritas acima.

Pipa[21] é uma linguagem de especificação de comportamento de interfaces (BISL) destinada a AspectJ. Ela extende a linguagem de JML para especificar classes de interfaces de AspectJ. O objetivo é usar as ferramentas existentes de JML para verificar programas escritos em

AspectJ. Diferentemente de Pipa, o trabalho proposto usa AspectJ para implementar asserções de JML em programas Java.

6.3 Trabalhos Futuros

O primeiro trabalho futuro seria implementar novas estruturas de JML em Aspectos, ou seja, aumentar a abrangência do subconjunto mapeado por esse trabalho. Com JML oferece novas estruturas como o uso de predicados, seria interessante poder expressar esse mesmo comportamento utilizando notações em AspectJ.

O segundo trabalho futuro seria realizar a integração de todas as ferramentas utilizadas em um único *plugin* para o Eclipse afim de evitar problemas de compatibilidade entre as suítes fornecidas. Com isso todo o processo de compilação funcionaria como uma caixa-preta para o desenvolvedor, facilitando e agilizando o trabalho de especificação do código a ser desenvolvido.

O terceiro trabalho futuro seria em trabalhar na otimização do mapeamento do código em AspectJ. Gerando menos código, ou ainda gerando somento o necessário, o *.jar* final da aplicação teria um tamanho reduzido , que em muitos casos, é primordial para a adaptação de uma aplicação para um dispositivo móvel específico.

Bibliografia

[1] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. ACM Computing Surveys, 1996.

[2] Sun Inc.MIDP 2.0. http://java.sun.com/products/midp/. Visitado pela última vez em 03/05/2008.

[3] Sun Inc. JSR 118. http://www.jcp.org/aboutJava/communityprocess/final/jsr118/. Visitado pela última vez em 03/02/2008.

[4] Sun Inc. API JavaCard Sun Inc. http://java.sun.com/products/javacard/index.jsp. Visitado pela última vez em 04/04/2008.

[5] POLL, Erick , BERG, Joachim v. d. , e JACOBS, Bart. Specification of the JavaCard API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, Smart Card Research and Advanced Application,

[6] LEAVENS G., BAKER Albert L., e RUBY Clyde. JML: A notation for Detailed Design. In Haim Kilov, Bernahard Rumpe, e Ian Simmonds(editors), Behavioral Specifications of Businesses and Systems, capítulo 12, páginas 175-158. Copyright Kluwer,1999.

[7] LEAVENS G., BAKER Albert L., e RUBY Clyde. Preliminary Design of JML: Behavioral Interface Specification language for Java. Department of Computer Science, Iowa State University, TR #98-06-rev27, Junho, 1998, revisado em Abril 2005.

[8] VERZULLI, JOE. Getting started with JML: Improve your Java programs with JML annotation. IBM DeveloperWorks article, Março 2003.

[9] LEAVENS G., e CLIFTON Curtis. Lessons from the JML Project. To appear in Verified Software: Theories, Tools, Experiments.Department of Computer Science, Iowa State University, TR #05-12a, Abril 2005, revisado em Julho de 2005.

[10] Gemplus. Electronic Purse Case Study. http://www-sop.inria.fr/lemme/verificard/electronic_purse/. Visitado pela última vez em 04/05/2008.

[11] Agarwal, P., Rubio-Medrano, C. E., Cheon, Y., and Teller, P. J. (2006). A formal specifcation in JML of the Java security package. Technical Report 06-13, Department of Computer Science, The University of Texas at El Paso.

[12] VERAS, R. C. Especificação Comportamental de um Subconjunto da Plataforma J2ME. Trabalho de Conclusão de Curso. Departamento de Sistemas Computacionais, Escola Politécnica de Pernambuco, Universidade de Pernambuco, Dezembro 2005.

[13] CHEON, Y. A Runtime Assertion Checker for the Java Modeling Language. Ph.D. thesis. Departament of Computer Science, Iowa State University, April 2003.

[14] BURDY, L et al. An Overview of JML tools and applications. **International Journal on Software Tools for Technology Transfer**, volume 7, número 3, p. 212-232, 2005.

[15] KICZALES, G. Aspect-oriented programming. ACM Computing Surveys, v.28, n.154, 1996.

[16] KICZALES, G. et al.. Getting Started with AspectJ. Communications of the ACM, 44(10):59–65, 2001.

[17] LEAVENS, T et.al. Design By Contract with JML . UTEP, December 2006.

[18] GOSLING, James et al.The Java Language Specification Second Edition.The Java Series. Addison-Wesley, Boston, MA, 2000.

[19] SOARES, S.,BORBA P. AspectJ - Programação Orientada a aspectos em Java . SBLP 2002.

[20] Y. A. Feldman et al. Jose : Aspects for design by contract 80-89. Sefm, 0:80-89, 2006.

[21] J. Zhao and M. C. Rinard. Pipa: A behavioral interface speci¯cation language for aspectj. In Proc. Fundamental Approaches to Software Engineering (FASE'2003) of ETAPS'2003, Lecture Notes in Computer Science, Apr. 2003.

[22] R. Henrique. Implementing JML Contracts with AspectJ. Master Degree Thesis. DSC, Pernambuco University, Maio 2008.

[23] DELAMARO M. E. Como Construir um Compilador Utilizando Ferramentas Java. NOVATEC.

[24] C. A. R. Hoare. "An axiomatic basis for computer programming". Communications of the ACM, 12(10):576–585, October 1969

Apêndice A

CalculatorMIDlet

Neste apendice é apresentado o código fonte de uma calculadora desenvolvida em J2ME anotada com asserções JML utilizada como base do exemplo da seção 5.4.2

Código Fonte

```
/**
 * The calculator demo is a simple floating point calculator
 * which powered by floating point support available in cldc1.1.
 *
 * @version
 */
public final class CalculatorMIDlet extends MIDlet implements CommandListener
{
    /** The number of characters in numeric text field. */
    private static final int NUM_SIZE = 20;

    /** Soft button for exiting the game. */
    private final Command exitCmd = new Command("Exit", Command.EXIT, 2);

    /** Menu item for changing game levels. */
    private final Command calcCmd = new Command("Calc", Command.SCREEN, 1);

    /** A text field to keep the first argument. */
    private final TextField t1 = new TextField(null, "", NUM_SIZE,
TextField.NUMERIC);

    /** A text field to keep the second argument. */
    private final TextField t2 = new TextField(null, "", NUM_SIZE,
TextField.NUMERIC);
```

```
    /** A text field to keep the result of calculation. */
    private final TextField tr = new TextField("Result", "", NUM_SIZE,
TextField.ANY);

    /** A choice group with available operations. */
    private final ChoiceGroup cg =
        new ChoiceGroup("", ChoiceGroup.EXCLUSIVE,
            new String[] { "add", "subtract", "multiply", "divide" }, null);

    /** An alert to be reused for different errors. */
    private final Alert alert = new Alert("Error", "", null, AlertType.ERROR);

    /** Indicates if the application is initialized. */
    private boolean isInitialized = false;

    /** Added for experiment. */

    //@ public instance invariant result >= 0;
    public double result = 0.0;

    /**
     * Creates the calculator view and action buttons.
     */
    protected void startApp() {
        if (isInitialized) {
            return;
        }
        Form f = new Form("FP Calculator");
        f.append(t1);
        f.append(cg);
        f.append(t2);
        f.append(tr);
        f.addCommand(exitCmd);
        f.addCommand(calcCmd);
        f.setCommandListener(this);
        Display.getDisplay(this).setCurrent(f);
        alert.addCommand(new Command("Back", Command.SCREEN, 1));
        isInitialized = true;
    }



        /**
     * Does nothing. Redefinition is required by MIDlet class.
     */
    protected void destroyApp(boolean unconditional) {
    }

    /**
     * Does nothing. Redefinition is required by MIDlet class.
     */
    protected void pauseApp() {
    }
```

```
//@ ensures \result == a + b;
  public double add(double a, double b){
    return a - b;
  }

  //@ requires b <= a;
  //@ ensures \result == a - b;
  public double sub(double a, double b){
    return a - b;
  }

  //@ ensures \result == a * b;
  public double mult(double a, double b){
    return a * b;
  }

  //@ requires b > 0;
  //@ ensures \result == a / b;
  public double div(double a, double b){
    return a / b;
  }

  /**
   * Responds to commands issued on CalculatorForm.
   *
   * @param c command object source of action
   * @param d screen object containing the item the action was performed on.
   */
  public void commandAction(Command c, Displayable d) {
      if (c == exitCmd) {
          destroyApp(false);
          notifyDestroyed();

          return;
      }

      try {
          double n1 = getNumber(t1, "First");
          double n2 = getNumber(t2, "Second");

          switch (cg.getSelectedIndex()) {
          case 0:
              result = add(n1, n2);

              break;

          case 1:
              result = sub(n1, n2);

              break;

          case 2:
              result = mult(n1, n2);

              break;

          case 3:
                result = div(n1, n2);
```

```
                break;

            default:
            }
        } catch (NumberFormatException e) {
            return;
        } catch (ArithmeticException e) {
            alert.setString("Divide by zero.");
            Display.getDisplay(this).setCurrent(alert);

            return;
        }

        /*
         * The resulted string may exceed the text max size.
         * We need to correct last one then.
         */
        String res_str = Double.toString(result);

        if (res_str.length() > tr.getMaxSize()) {
            tr.setMaxSize(res_str.length());
        }

        tr.setString(res_str);
    }

    /**
     * Extracts the double number from text field.
     *
     * @param t the text field to be used.
     * @param type the string with argument number.
     * @throws NumberFormatException is case of wrong input.
     */
    private double getNumber(TextField t, String type)
        throws NumberFormatException {
        String s = t.getString();

        if (s.length() == 0) {
            alert.setString("No " + type + " Argument");
            Display.getDisplay(this).setCurrent(alert);
            throw new NumberFormatException();
        }

        double n;

        try {
            n = Double.parseDouble(s);
        } catch (NumberFormatException e) {
            alert.setString(type + " argument is out of range.");
            Display.getDisplay(this).setCurrent(alert);
            throw e;
        }

        return n;
    }
} // end of class 'CalculatorMIDlet' definition
```

Anexo I

Implementing Java Modeling Language Contracts with AspectJ