Capítulo

6

Design by Contract com JML

Rogério Dourado Silva Júnior
Jorge César Abrantes de Figuereido
Dalton Dario Serey Guerrero

Abstract

Design by Contract (DBC) is a development methodology that aims the construction of more reliable object-oriented systems. It provides an assertion violation mechanism capable to check the conformance of the code against its specification. The main idea of DBC is the explicit establishment of a contract between the classes of the system. The ones in the client role must guarantee certain conditions before invoking others methods (pre-conditions), that in turn must satifies some properties after their execution (post-conditions). JML (Java Modeling Language) is a formal behavioral interface specification language for Java that contains the essential notations used in DBC as a subset. In this course, the DBC concepts are introduced using JML notations to specify Java classes. We also give emphasis in examples showing how to use effectively the method and tools that supports it.

Resumo

Design by Contract (DBC) é um método de implementação que visa a construção de sistemas OO mais confiáveis, na medida em que provê mecanismos para detecção de violações da sua especificação. A principal idéia de DBC 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). JML (Java Modeling Language) por sua vez, é uma linguagem formal de especificação comportamental para Java que contém as notações essenciais de DBC como seu subconjunto. Neste minicurso, os conceitos de DBC são abordados com ênfase em como utilizar JML para anotar no código o design detalhado de classes e interfaces Java. Exemplos de utilização desse método são também introduzidos através do uso de algumas ferramentas.

6.1 Introdução

Uma das características mais importantes para qualquer sistema computacional é que este seja confiável, ou seja, robusto e correto. Teoricamente um sistema é dito correto caso ele desempenhe todas as funções conforme sua especificação estabelece e robusto quando este lida com situações inesperadas (não especificadas) de uma maneira tal a não comprometer todo seu funcionamento. O paradigma OO sem dúvida representou um avanço na busca pela qualidade de software, porém demorou um pouco para se perceber que apenas sua utilização não era suficiente para construir softwares melhores. Foi necessário então um tempo para que os desenvolvedores assimilassem efetivamente as boas práticas do desenvolvimento orientado a objetos. No entanto, tais práticas - apesar de fundamentais para um bom projeto - não são suficientes por si só para a obtenção de sistemas sem erros. Neste contexto, *Design by Contract* (DBC) surge como um método que visa a construção de sistemas OO mais confiáveis, na medida em que provê mecanismos para checar a correção de um sistema.

Vale ressaltar que é impossível julgar esta correção isoladamente, uma vez que o software só pode ser considerado correto - ou incorreto - quando confrontado com uma definição do comportamento esperado. Em outras palavras, toda a discussão sobre assunto resume-se ao problema de checar a consistência entre o software e sua especificação. Veremos como DBC e seus conceitos associados fornecem os meios pelos quais é possível expressar tal especificação e sobretudo avaliar automaticamente se a implementação a satisfaz. Porém este é apenas o mais direto dos benefícios desta abordagem, muitos outros surgem quando estes conceitos são devidamente inseridos em uma metodologia de programação. A principal idéia do método é que entre as classes e seus clientes seja estabelecido um contrato, o qual deve ser explicitamente definido através de anotações no seu próprio código. 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). Além de descrever individualmente os métodos, há também a necessidade de expressar propriedades globais das instâncias de uma determinada classe. Essas propriedades, chamadas de invariantes de classe, devem ser preservadas por todos os seus métodos.

O uso de pré e pós-condições no desenvolvimento de software não é recente, tais conceitos foram originalmente introduzidos ainda no final da década de 60 por Hoare, Floyd e Dijkstra. No entanto, o que vem chamando atenção atualmente acerca de DBC são as novas técnicas de verificação de software que confrontam os contratos com o código do programa. Uma forma de fazer isto é checar as asserções em tempo de execução. Depois de devidamente instrumentado (obviamente com suporte ferramental), o software torna-se capaz dele mesmo notificar violações de sua especificação. Uma variação desta abordagem é a geração automática de testes de unidade a partir dos contratos definidos, poupando o programador de implementar o código que efetivamente decide quando o teste deve ser considerado bem sucedido ou não. Uma técnica mais ambiciosa é realizar essa verificação estaticamente. Nesse caso, tenta-se estabelecer a conformidade com a especificação em todos os possíveis caminhos de execução, enquanto que a checagem em tempo de execução fica limitada pelos caminhos sendo executados pela suíte de testes sendo usada.

DBC acaba sendo também uma boa forma de documentar o software. Torna-se

mais fácil compreender seu funcionamento analisando os contratos que ele estabelece do que através do código ou até mesmo de comentários no programa. Isto se deve ao fato dos contratos serem mais abstratos do que o código, uma vez que estes não estabelecem nenhum algoritmo, se concentrando portanto em expressar o que é assumido e o que deve ser alcançado sem se preocupar em como isto deve ser feito. Diferentemente de comentários, um contrato é uma documentação formal, que define sem ambigüidades o comportamento esperado do sistema. Graças à essa característica é possível construir uma série de ferramentas automatizadas de suporte, o que acaba por aumentar as chances desta documentação ser mantida atualizada. Uma outra vantagem de DBC é que este facilita enormemente a tarefa de depuração do código. Em alguns casos, localizar a falta causadora de um possível mal funcionamento do software se resume em identificar o contrato que foi quebrado. Caso a violação seja na pré-condição de um determinado método, fica claro que o problema está no código que o invocou. Caso seja na pós-condição o erro se encontra na implementação do próprio método que foi incapaz em garanti-la após ter sido executado.

Neste curso, iremos utilizar a linguagem JML (*Java Modeling Language*) para exemplificar todos esses conceitos. JML foi concebida para permitir a especificação formal e detalhada de programas Java através da adição de anotações no próprio código fonte. Ela contém todas as notações necessárias à aplicação da técnica de DBC como seu subconjunto, além de ter sido projetada para ser de fácil uso pelos programadores. Para isto a linguagem tenta se manter mais próxima possível da sintaxe e semântica de Java. No entanto, falta às construções Java um pouco de expressividade em alguns aspectos que facilitaria a escrita de asserções. Por isso JML estende as expressões Java, adicionando algumas notações especializadas à especificação, tal como quantificadores. Assim como UML, JML também não impõe nenhuma metodologia particular para quem faz uso dela. De fato, a linguagem têm se tornado o padrão no que se refere à especificação de programas Java, prova disso é a já grande quantidade de ferramentas existentes que a utilizam. Vale ressaltar que atualmente JML é mantido por uma cooperação entre vários grupos de pesquisas, o que de certa forma tem contribuído para sua contínua evolução.

Na próxima seção são introduzidos os conceitos básicos que constituem a teoria de DBC, através do uso de exemplos independentes de qualquer linguagem de especificação. A seção 3 apresenta especificamente a linguagem JML e como esta pode ser utilizada para concretizar os conceitos apresentados na seção anterior. Na seção 4 aspectos mais avançados da linguagem são apresentados. A seção 5 demonstra a utilização de algumas ferramentas para JML. E por último, a seção 6 mostra um estudo de caso englobando a especificação, implementação e verificação de uma classe Java. Neste material é assumido que o leitor tenha familiaridade com a linguagem Java e sobretudo um bom conhecimento sobre orientação a objetos.

6.2 Design by Contract

6.2.1 Contratos e Asserções

Quando o programador codifica um método de uma classe, ele o faz com o intuito que este cumpra uma determinada tarefa em tempo de execução. A menos que esta tarefa seja trivial, ela engloba várias outras sub-tarefas. Cada uma destas sub-tarefas pode ser

codificada diretamente no próprio método ou pode-se criar um outro para comportar tal funcionalidade. No segundo caso, o método principal delega a um outro a responsabilidade de cumprir determinada tarefa através de uma chamada de método. Um número muito grande de chamadas de método causa uma fragmentação excessiva do código, enquanto que o contrário torna cada método em particular demasiadamente complexo. Apesar de ambas abordagens terem suas vantagens e desvantagens, **geralmente** um código com muitos métodos e invocações é mais fácil de entender (e conseqüentemente manter) do que um código com métodos enormes, cuja coesão provavelmente é baixíssima. A opção pelo uso da chamada de método para uma destas sub-tarefas é análogo a situação na qual preferimos contratar o serviço de terceiros à fazermos a tarefa nós mesmos. Por exemplo, se quisermos enviar uma correspondência a alguém nós podemos entregá-la pessoalmente ou contratar o serviço dos correios para tal. Cada parte espera algum benefício do contrato, ao mesmo tempo em que aceita cumprir algumas obrigações para os obter. Normalmente tais benefícios e obrigações são devidamente expressos em um documento que deve proteger ambas as partes, conforme mostra a Tabela 6.1. 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 o escopo do que foi contratado.

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

Tabela 6.1: Exemplo: O que constitui uma obrigação para uma das partes geralmente torna-se um benefício para a outra.

Aplicando essa mesma idéia no contexto de software, temos que caso a execução de uma determinada tarefa dependa de um outro método, torna-se necessário especificar precisamente a relação entre cliente (quem invoca) e contratado (método que é invocado). Em DBC, o mecanismo para expressar as condições que devem reger o contrato de software são chamadas de **asserções**. Pré-condições e pós-condições são asserções que definem respectivamente os direitos e obrigações individuais de cada método. Invariantes de classes constituem um outro tipo de asserção, que como o próprio nome denota, são propriedades que sempre são válidas ao longo do ciclo de um objeto. As invariantes de classe serão discutidas na próxima seção.

Tomemos como exemplo o contrato de um método que recebe um número qualquer como entrada e retorna sua raiz quadrada. Este contrato pode ser documentado como

mostra o Código 6.1. A pré-condição é usada para especificar o que deve ser verdadeiro antes do método ser chamado, enquanto que a pós-condição estabelece qual o compromisso do método depois de ter sido executado. A obrigação do cliente nesse caso é passar um número positivo como argumento (x). Por outro lado, este tem o direito de receber uma aproximação[1] da raiz quadrada como resultado.

Código 6.1: Pré e pós-condições

```
...
pré-condição x >= 0.0;
pós-condição aproximadamenteIgual(x, resultado * resultado, 0.1)
...
```

Como podemos observar, asserções não devem descrever casos especiais e sim situações esperadas do uso de determinado método. Ou seja, o uso somente de asserções não são apropriadas para lidar, por exemplo, com a passagem de referências nulas no argumento. Se for desejo do programador tratar tal situação como um caso aceitável (apesar de ser especial), este deveria fazê-lo através do uso de estruturas condicionais. O que precisa ser observado é que qualquer violação de uma especificação deve ser considerada como a manifestação de um erro no software. Em outras palavras:

- A violação de uma pré-condição indica um erro em quem invocou o método;
- A violação de uma pós-condição representa um erro no próprio método

Estas observações constituem uma regra básica:

Princípio da Não-Redundância

> Ou certa propriedade é colocada na pré-condição ou em uma instrução `if` no corpo do método, mas nunca em ambas.

Isso significa que a responsabilidade por tratar determinada situação deve ficar ou a cargo do cliente do método ou do próprio método, evitando com isso testes redundantes desnecessários. Apesar dessa escolha de certa forma ser uma questão de preferência pessoal do programador, a busca pela simplicidade e a divisão justa do trabalho são critérios que não podem ser negligenciados. No entanto, a experiência com o desenvolvimento de alguns sistemas demonstraram que a opção por fortalecer a pré-condição constitui uma boa estratégia. Nessa abordagem cada método pode se concentrar em desempenhar um papel bem definido, e o faz melhor por não precisar se preocupar com todos os casos imagináveis. Os clientes, por sua vez, não esperam de antemão que os métodos aos quais eles utilizam estejam preparados para todas as situações. Mas é preciso usar o bom senso, caso contrário o esforço para se escrever um método poderia se tornar a mais simples das tarefas. Bastaria o programador começar todos os métodos com a pré-condição `false` para fazer com que qualquer implementação fosse considerada correta. Isto equivale a dizer que sempre o cliente está errado, uma vez que tal pré-condição nunca poderá ser satisfeita.

[1] É utilizado um método fictício que testa se a diferença entre dois argumentos do tipo double é menor do que um terceiro.

Uma crítica ao estilo no qual as pré-condições são fortalecidas é que ele parece fazer com que cada cliente precise checar de forma redundante a mesma propriedade antes de invocar determinado método. Essa observação não se justifica na prática, uma vez que a presença de uma pré-condição *p* em um método *r* não implica que toda chamada a ele deve testar *p*, como mostrado no Código 6.2. O que a pré-condição exige é que o cliente garanta a propriedade *p*, o que não significa testá-la antes de cada chamada. Em muitos casos o próprio contexto da chamada implica necessariamente que *p* seja verdadeiro, o que torna o teste completamente desnecessário.

Código 6.2: Cliente de um método checando sua pré-condição

```
...
if (x.isConditionPTrue()) {
    x.r();
} else {
    ... Tratamento Especial ...
}
...
```

Porém, se mesmo assim alguns clientes ainda tiverem que checar a pré-condição, o que vai importar mesmo é se o tratamento especial dado quando a condição não for verificada é o mesmo em todas as chamadas ou específica para cada uma. Caso seja a mesma, fica claro que trata-se de um problema no *design*, no qual o método *r* impõe um contrato demasiadamente rígido a seus clientes. Uma solução seria torná-lo mais tolerante, de forma a incluir o tratamento especial como parte parte do próprio método. Porém se o tratamento for diferente para cada cliente é inevitável que estes individualmente verifiquem *p*, não sendo portanto uma conseqüência do estilo escolhido.

Nesse ponto pode surgir a dúvida sobre o que acontece quando algum desses contratos é violado em tempo de execução. Iremos abordar esse assunto posteriormente, mas de antemão pode-se dizer que depende do que o programador deseja. É possível inclusive tratar as asserções puramente como comentários, sem nenhum efeito na execução do software, o que alguns recomendam ser feito quando o software for colocado em plena produção. O motivo alegado é que tal mecanismo degrada a performance do sistema como um todo - o que de fato é verdadeiro. No entanto, existe uma outra corrente que alega que tal degradação geralmente é um preço que vale a pena ser pago tendo em vista os benefícios que este mecanismo oferece, principalmente para um software que não se encontra mais em um ambiente de desenvolvimento. Neste caso, as asserções são usadas justamente para verificar se tudo está funcionando como planejado. Em caso de ocorrência de alguma violação, a execução é interrompida com uma mensagem identificando claramente o que aconteceu. Existem outros meios de usar as especificações no estilo DBC para aumentar a confiança do software, como por exemplo através da técnica de análise estática.

6.2.2 Invariantes

Como mostrado na seção anterior, pré e pós-condições são utilizadas para estabelecer os direitos e obrigações de cada método individualmente. No entanto, faz-se necessário também algum mecanismo para expressar propriedades globais sobre as instâncias de uma classe, as quais todos os métodos devem preservar. Para isso existe o conceito de invariante, que define as propriedades que devem ser válidas em todos os estados “visíveis”

dos objetos de uma determinada classe. Podemos considerar que uma instância qualquer está em um estado visível quando o controle de execução não está localizado em nenhum de seus métodos. Com isso temos as seguintes regras:

- A invariante deve ser satisfeita logo após a criação de toda instância da classe. Ou seja, seus construtores devem **estabelecer** a invariante;
- Todo método não-estático deve garantir que a invariante seja **preservada** após sua execução, se esta o for imediatamente antes de sua invocação.

No Código 6.3, podemos observar a descrição de uma invariante que estabelece que o atributo *nome* não pode ser vazio e que *peso* deve ser maior ou igual a zero.

Código 6.3: Invariante

```
...
invariante !nome.equals("") && peso >=0;
...
```

Todos os métodos e construtores devem preservar e estabelecer respectivamente as invariantes mesmo se estes terminem abruptamente, por exemplo em caso de uma exceção ser lançada. Por último vale ressaltar que enquanto as pré e pós-condições podem ser aplicadas em outros paradigmas, o conceito de invariante de classe é inconcebível fora da abordagem orientada a objetos.

6.2.3 Herança

Uma das conseqüências da prática de DBC é o melhor controle sobre um dos mais importantes fundamentos da orientação a objetos: herança e polimorfismo. Apesar destes recursos serem um dos alicerces da flexibilidade do paradigma OO, muitos ainda têm dificuldade de utilizá-los corretamente. Através do mecanismo de herança, por exemplo, podemos criar novas classes a partir de outras já existentes. Nesses casos, o comportamento dos métodos herdados não precisam necessariamente ser mantidos pelas suas sub-classes: é possível redefiní-los com um comportamento parcial ou completamente distinto. Nesse ponto, surge uma questão interessante em relação ao uso do método DBC: como evitar que a redefinição de um método produza um efeito incompatível com a especificação comportamental do método definido na super-classe? Fica claro que esse um problema causado pelo mal uso destes conceitos, o qual felizmente DBC ajuda a evitar na medida em que obriga aos métodos redefinidos honrarem os compromissos estabelecidos no contrato original.

Para exemplificar esta questão, suponha a existência de duas classes: `Pessoa` e `Estudante`. Sendo que a segunda é uma sub-classe da primeira. Devido ao polimorfismo, uma classe `C` qualquer 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. O que pode acontecer, por exemplo, é que quando `C` passe a se comunicar com uma instância de `Estudante`, o que só pode ser descoberto em tempo de execução, o contrato de um determinado método herdado seja distinto da especificação que consta na super-classe. Ou seja, `C` espera estar chamando um método

> Mesmo que todos os objetos de uma classe sejam substituídos por instâncias de suas sub-classes, deve continuar sendo possível executar o sistema

sob um determinado contrato, enquanto de fato está se comunicando com outro completamente diferente. No contexto de DBC, esta situação fere o princípio da substitutibilidade de OO que define que:

Nessa situação como podemos evitar que o cliente `C` seja "enganado" com uma chamada a um método que pode não realizar aquilo que ele espera? Na verdade, existem duas formas que uma classe pode deteriorar o contrato especificado na sua super-classe:

- A sub-classe poderia fazer com que a pré-condição se tornasse mais restritiva, gerando o risco que algumas chamadas consideradas anteriormente corretas na perspectiva de `C` (uma vez que estas satisfazem as obrigações originais impostas ao cliente) passasse a violar as regras do contrato;
- A sub-classe poderia fazer com que a pós-condição se tornasse mais permissiva, retornando um resultado menos satisfatório do que havia sido prometido a `C`.

O que DBC impõe é que nenhuma dessas situações devem ser permitidas. Porém, vale ressaltar que a redefinição de métodos no sentido oposto a esses citados logo acima é válida. Ou seja, 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 super-classe. Isso significa dizer que a sub-classe faz um trabalho no mínimo tão bom quanto o seu ascendente, motivo pelo qual não há nenhum motivo para inibir tal prática. Temos a partir de então um mecanismo eficiente para garantir que a redefinição de métodos seja utilizada da maneira correta, evitando que um procedimento seja transformado em outro completamente diferente através de herança. De toda essa discussão concluímos que toda especialização deve ser mantida compatível com a especificação original, embora reste a sub-classe o direito de melhorá-la.

Além das regras aplicadas às pré e pós-condições, o mecanismo de herança também tem efeito sobre as invariantes: estas são sempre passadas aos seus descendentes. Isso é resultado do próprio conceito de herança, no qual toda instância de uma classe é também instância de todos os seus ascendentes. Com isso, nada mais lógico do que as restrições de consistência definidas nas super-classes também se apliquem à própria classe. O que na prática significa dizer que a invariante real de uma determinada classe é a conjunção entre suas invariantes locais e todas as invariantes em sua hierarquia ascendente de herança.

6.2.4 Lidando com Exceções

Como um método não é mais visto apenas como um trecho de código e sim como a implementação de uma certa especificação, é possível definir melhor a noção de falha. Uma falha ocorre quando a execução de um método não consegue cumprir o seu contrato. Existem possíveis causas para o acontecimento de falhas, tais como mal funcionamento do *hardware*, erros de implementação ou algum evento externo inesperado. Falha aqui é o

conceito básico. Exceção é uma noção derivada, que ocorre quando uma certa estratégia para cumprir o contrato não obteve sucesso. A partir dessa definição é possível estabelecer a seguinte regra:

Princípio da Exceção

> Um método não deve lançar uma exceção quando sua pré-condição não for satisfeita, pois isso não denota uma quebra de contrato do método, e sim de quem o invocou. Na situação do método satisfazer sua pós-condição, também nenhuma exceção deve ser lançada.

DBC propõe que um método seja invocado em um estado no qual sua pré-condição seja satisfeita, caso contrário nada pode ser garantido, nem mesmo que a execução do método termine. Se o método é chamado nas condições ideais (com as pré-condições válidas), este deve ou retornar normalmente ou lançar uma exceção. Caso o método termine normalmente, ele é obrigado a satisfazer as pós-condições normais definidas no seu contrato, assim como mostramos anteriormente. Se este terminar abruptamente lançando uma exceção, um outro conjunto de regras, chamadas aqui de pós-condições excepcionais, é que devem ser atendidas. Diante deste cenário é possível fazer uma analogia entre o método e um quarto no qual existem duas portas de saída: uma normal e outra excepcional. No entanto, cada porta exige que algumas condições sejam atendidas antes de serem utilizadas. O método pode portanto, sair ou pela porta normal ou pela excepcional, porém para isto suas respectivas pós-condições devem ser satisfeitas.

Em relação às propriedades globais de classe, métodos e construtores devem preservar e estabelecer as invariantes tanto em caso de terminação normal ou abrupta. Em outras palavras, invariantes estão implicitamente incluídas tanto na pós-condição normal quanto na excepcional. O requisito de que invariantes sejam sempre satisfeitas, mesmo em caso de uma exceção ocorrer, pode ser demasiadamente exigente à primeira vista. No entanto, esta é a única opção que resta, uma vez que quando uma invariante é violada nenhuma garantia mais pode ser feita sobre as subseqüentes invocações de métodos ao objeto.

6.3 Conceitos Básicos de JML

Em JML, as especificações são escritas como um tipo especial de comentário, no qual cada linha comentada deve começar com o sinal de arroba (@). Normalmente, esses comentários não teriam nenhuma influência sobre a execução de um programa, servindo apenas ao próprosito de documentação. A finalidade de usar o @ é justamente permitir que as ferramentas automatizas possam diferenciar corretamente um comentário de uma especificação anotada em código. Todos os exemplos daqui por diante seguirão esta regra.

Até aqui vimos a teoria que envolve a técnica de DBC, com alguns poucos exemplos de especificações. A partir de agora, daremos ênfase em apresentar detalhadamente a linguagem JML e como esta pode ser utilizada para concretizar os conceitos apresentados na seção anterior.

6.3.1 Pré e pós-condições

A cláusula `requires` é usado para especificar as pré-condições de um método ou construtor. A cláusula deve preceder um predicado expresso na sintaxe Java, no qual os parâmetros do método sendo especificado, assim como os atributos visíveis da classe (ver 6.4.2), podem ser referenciados. Métodos que não tenham efeito colateral, como por exemplo métodos *get* que apenas consultam o estado do objeto (ver Seção 6.3.8), podem também ser utilizados no predicado. Múltiplas cláusulas `requires` podem ser utilizadas em uma especificação. Estas por sua vez, têm o mesmo significado de uma única cláusula cujo predicado seja a conjunção de todos os predicados anteriores. Por exemplo,

```
//@ requires x > 0;
//@ requires y < =0;
```

significa o mesmo que:

```
//@ requires x > 0 && y <= 0;
```

Quando nenhuma cláusula `requires` é definida em uma especificação, assume-se o uso de uma cláusula `requires` padrão, cujo predicado é definido como sendo `\not_specified`. Este mecanismo existe para que as técnicas de verificação possam escolher qual o significado mais apropriado para a ausência de um determinado aspecto na especificação. No entanto, do ponto de vista de quem está escrevendo a especificação `\not_specified` assume o valor de *true*, uma vez que nesse caso sempre a pré-condição será satisfeita.

A cláusula `ensures` especifica a propriedade que deve ser satisfeita ao final da execução de um método ou construtor no caso destes retornarem o controle sem que uma exceção ocorra. As mesmas regras que valem para `requires`, também valem para a especificação da pós-condição com o uso de `ensures`. A diferença é que no caso da cláusula `ensures`, podem ser usados ainda expressões como `\result` (se o método não for *void*) e `\old(E)`. A construção `\result` é usado para referenciar o resultado que é retornado pelo método. Por sua vez, `\old(E)` faz com que o valor da expressão *E* seja avaliado no momento anterior ao método ser executado. No Código 6.4, temos o uso da cláusula `\old(peso)` para impor que o atributo peso, ao final da execução, deve ter valor igual ao do próprio peso antes da chamada do método somado ao parâmetro *kgs*. Note, que foi preciso utilizar o modificador `spec_public` antes da declaração dos atributos da classe pois, conforme veremos na Seção seguinte, uma especificação pública (como esta do método *adicionaKgs(int kgs)*) só pode acessar atributos públicos. Porém, este modificador faz com que os atributos da classe passem a ter visibilidade pública do ponto de vista da especificação, sem no entanto interferir em nada nas regras estabelecidas entre os elementos do software.

Perceba que o mais correto seria que a cláusula `old` englobasse também o parâmetro *kgs*, para garantir que mesmo se o seu valor fosse modificado ao longo da computação do método, o valor original é que deveria ser considerado na avaliação da pós-condição. No entanto, ambas as formas têm o mesmo significado pois, por padrão, todo parâmetro do método que é utilizado na pós-condição implicitamente é envolto por uma cláusula `old`. A justificativa para esta convenção é que os clientes só conhecem os valores iniciais dos parâmetros que eles fornecem, não tendo portanto nenhuma informação a

Código 6.4: Uso da expressão `old(E)`

```
public class Pessoa {
  private /*@ spec_public @*/ String nome;
  private /*@ spec_public @*/ int peso;

  /*@ requires peso + kgs >= 0;
    @ ensures kgs>=0 && peso==\verb!\!old(peso) + kgs;
    @ signals (IllegalArgumentException e) kgs < 0;
    @*/
  public void adicionaKgs(int kgs) throws IllegalArgumentException{
    if(kgs < 0)
          throw new IllegalArgumentException();
    this.peso += kgs;
  }
}
```

respeito dos possíveis valores que estes podem assumir após a execução do método. Sem esta convenção, os métodos *errado1()* e *errado2()* do Código 6.5 estariam de acordo com suas especificações. No entanto, fica claro que do ponto de vista do cliente, somente o método *correto()* satisfaz o contrato estabelecido.

Código 6.5: Justificativa para o `old` implícito nos parâmetros

```
public abstract class OldImplicito {
    /*@ ensures 0 <= \result && \result <= x;
      @ signals (Exception) x < 0;
      @*/
    public static int errado1(int x) throws Exception {
        x = 5;
        return 4;
    }

    /*@ ensures 0 <= \result && \result <= x;
      @ signals (Exception) x < 0;
      @*/
    public static int errado2(int x) throws Exception {
        x = -1;
        throw new Exception();
    }

    /*@ ensures 0 <= \result && \result <= x;
      @ signals (Exception) x < 0;
      @*/
    public static int correto(int x) throws Exception {
        if (x < 0) throw new Exception();
        else return 0;
    }
}
```

6.3.2 Cláusula `signals`

Para expressar as pós-condições excepcionais, JML utiliza a cláusula `signals`, sendo uma para cada tipo de exceção que o método pode lançar em tempo de execução. Vale

lembrar aqui, que pelo conceito de herança uma classe qualquer também é do tipo de suas super-classes. Ou seja, definindo as pós-condições para uma exceção do tipo *java.lang.Exception* (da qual todas outras são sub-classes), automaticamente estará se englobando todos os outros tipos.

Código 6.6: Pós-condição para uma exceção de um determinado tipo

```
/*@ signals (IllegalArgumentException e) e.getMessage() != null &&
  @                                     x < 0;
  @*/
...
```

O Código 6.6 mostra uma possível pós-condição excepcional para o nosso método que calcula a raiz quadrada. Nela, exige-se que quando uma exceção do tipo `IllegalArgumentException` for lançada, a mensagem associada à ela não pode ser nula e que o argumento passado como parâmetro ao método deve ser negativo. Porém, perceba que neste exemplo outros tipos de exceções poderiam ser lançados e nesse caso, nenhuma pós-condição precisaria ser observada. Para restringir os tipos de exceção que o método pode lançar, deve-se generalizar o tipo da exceção para `java.lang.Exception` e impor que *e* seja uma instância de `IllegalArgumentException`, conforme mostra o Código 6.7. É possível também especificar que um método nunca deva lançar exceção alguma, bastando para isso trocar todo o predicado do Código 6.7 pela expressão `false`.

Código 6.7: Apenas exceção `IllegalArgumentException` pode ser lançada

```
...
/*@ signals (Exception e) e instanceof IllegalArgumentException &&
  @                       e.getMessage() != null && x < 0;
  @*/
...
```

Vale lembrar mais uma vez que a cláusula `signals`, da forma como usada nos exemplos anteriores, define as propriedades que devem ser válidas na ocorrência de uma determinada exceção. O que é completamente diferente de dizer que quando certa propriedade for válida tal exceção deve ocorrer. Essa é a causa de muita confusão na especificação das pós-condições excepcionais. Para expressar a obrigatoriedade de uma exceção em alguma circunstância deve-se compor o predicado da cláusula `ensures` com a negação do predicado *p*, definido na pós-condição excepcional. Dessa forma, se *p* for válido o método fica obrigado a lançar a exceção, caso contrário a pós-condição normal será violada. É mais fácil entender este conceito a partir do exemplo mostrado no Código 6.8. Nele, caso o método termine normalmente com *x* sendo negativo a pós-condição será violada, o que indica um erro na implementação do próprio método. A única opção para que isto não ocorra é lançar uma exceção diante de tal situação.

É freqüente também o mal uso da cláusula `signals`, devido a não observância do Princípio da Exceção discutida na Seção 6.2.4. Tal princípio quando aplicado ao contexto de JML exige que:

1. Sendo *p* o predicado utilizado na cláusula `requires`, *!p* não deve ser o predicado de nenhuma cláusula `signals`

Código 6.8: Caso x ¡ 0 o método obrigatoriamente deve lançar a exceção

```
/*@ ensures JMLDouble.approximatelyEqualTo(x, \result * \result, 0.1) &&
  @             x >= 0;
  @ signals (IllegalArgumentException e) e.getMessage() != null &&
  @                                      x < 0;
  @*/
...
```

2. Sendo *p* o predicado utilizado na cláusula `ensures`, *p* não deve constar no predicado de nenhuma cláusula `signals`

O Código 6.9 mostra exemplos de violação das regras acima. No primeiro caso, a especificação é por si só contraditória: a pré-condição estabelece que o método não deveria tratar a situação de x ser negativo (segundo o Princípio da Não-Redundância), o que é desmentido pela cláusula `signals`. No segundo caso, a ocorrência da exceção não indica nenhuma falha no cumprimento do contrato, uma vez que nesta situação a pós-condição normal também é satisfeita.

Código 6.9: Erros comuns no uso de `signals`

```
// VIOLA A REGRA 1
/*@ requires x >= 0;
  @ ensures JMLDouble.approximatelyEqualTo(x, \result * \result, 0.1);
  @ signals (IllegalArgumentException e) e.getMessage() != null &&
  @                                      x < 0;
  @*/
  public double raizQuadrada1(int x) throws IllegalArgumentException{...}

// VIOLA A REGRA 2
/*@ requires x >= 0;
  @ ensures JMLDouble.approximatelyEqualTo(x, \result * \result, 0.1);
  @ signals (IllegalArgumentException e)
  @         JMLDouble.approximatelyEqualTo(x, \result * \result, 0.1) &&
  @         e.getMessage() != null;
  @*/
  public double raizQuadrada2(int x) throws IllegalArgumentException{...}
```

6.3.3 Invariantes

Apesar de termos dedicado a Seção 6.2.2 ao tema invariante, cabe ainda alguns aspectos a serem discutidos, principalmente no contexto de JML. Uma invariante qualquer deve ser interpretada como sendo um compromisso da classe em não permitir que determinado predicado assuma o valor falso antes ou depois da execução de seus métodos e construtores. Com base nisso, podemos afirmar que a especificação do Código 6.10 está inconsistente com o código, pois a invariante seria facilmente violada caso o objeto fosse instanciado passando um argumento nulo ao seu construtor. Ou seja, uma invariante não deve expressar simplesmente um desejo e sim o que a classe realmente é capaz de garantir. Para resolver esse problema especificamente, bastaria adicionar a pré-condição `aNome!=null` ao construtor do método. Uma outra possível solução, apesar de incomum, seria fazer com que caso o construtor recebesse como parâmetro um objeto nulo, este atribuísse um valor padrão qualquer à variável *nome*.

As invariantes em JML podem ser precedidas pelos modificadores `static` ou `instance`. Assim como um método estático, uma invariante estática não pode referenciar o objeto atual através da palavra reservada `this`, o que conseqüente impede também

Código 6.10: Inconsistência entre especificação e código

```
public class Pessoa{
  private /*@ spec_public @*/ String nome;

  //@ public invariant nome!=null;

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

seu acesso a campos e métodos não estáticos da classe. Uma invariante declarada dentro de uma classe, assim como no Código 6.3, é por *default* uma invariante de instância, enquanto que quando declarada em uma interface é por sua vez considerada uma invariante estática. A distinção entre invariantes estática e de instância afeta também em que momento as invariantes devem ser verificadas. Para as invariantes estáticas valem as seguintes regras:

- O bloco de inicialização estática é obrigado a **estabelecer** a invariante, de modo que após sua execução a propriedade seja válida;
- Os construtores, assim como todos os métodos (sejam eles estáticos ou não) devem **preservar** a invariante.

6.3.4 Cláusula `also`

JML usa uma convenção simples para preservar a compatibilidade (discutida na Seção 6.2.3) entre os contratos de métodos herdados e a especificação original definida em sua super-classe: em uma redefinição de método, não é permitido usar as cláusulas `requires` e `ensures` para definir pré e pós-condições da mesma forma que vimos anteriormente, nesta situação obriga-se que todo contrato seja iniciado pela palavra `also`. Justamente para deixar explícito que o contrato que se segue está submetido ao que foi definido na super-classe. A ausência de pré ou pós-condições significa que a versão redefinida do método mantém a especificação exatamente igual à da versão original.

Código 6.11: Contrato de uma redefinição do Código 6.1

```
...
/*@ requires x >= 0.0;
  @ ensures JMLDouble
  @         .approximatelyEqualTo(x,\result * \result,0.1)
  @*/
  public double raizQuadrada(){...};        //MÉTODO ORIGINAL

/*@ also
  @   requires x > 0.0;
  @   ensures JMLDouble
  @           .approximatelyEqualTo(x,\result * \result,0.01)
  @*/
  public double raizQuadrada(){...};       //MÉTODO HERDADO
...
```

No Código 6.11 podemos observar o caso de uma redefinição, na qual o método herdado faz um trabalho “melhor” do que o original, na medida em que aceita também o

zero como argumento e retorna um resultado mais preciso do que o anterior. Fica claro nessa situação que qualquer cliente da especificação original também iria se satisfazer com este refinamento.

6.3.5 Descrições Informais

Apesar de todo o potencial para a verificação automatizada, as asserções são sobretudo uma ótima forma de deixar registrado informações importantes a respeito do comportamento esperado dos métodos de uma classe. Em alguns casos o programador pode querer propositalmente abrir mão da verificação de um determinado aspecto (para não precisar especificá-lo formalmente, por exemplo) mas mesmo assim desejar expressar a relação deste para com o restante do código por meio do conceito de contratos. Por conta dessa necessidade JML suporta o uso de descrições informais em suas especificações, mas o mais importante é que estas podem ser normalmente combinadas com trechos formais. Nesse caso, as descrições informais são tratadas como expressões booleanas que para efeito de verificação têm sempre o valor `true`. Uma propriedade informal é expressa da seguinte forma:

```
(* algum texto descrevendo a propriedade *)
```

Este mecanismo de fuga do mundo formal muitas vezes pode ser útil, mesmo que as propriedades definidas dessa maneira não sejam passíveis de serem verificadas através de ferramentas. Uma outra desvantagem é justamente a possível ambigüidade introduzida por descrições em linguagem natural. No Código 6.12 é mostrada uma variação da especificação anterior em termos de descrições formais e informais.

Código 6.12: Descrições formais e informais combinadas

```
...
//@ requires (* x é positivo *);
/*@ ensures (* \result é uma aproximação da raiz quadrada de x *)
  @         && \result >=0.0;
  @*/
...
```

Porém, vale lembrar que as descrições informais não devem ser usadas com o intuito de atenuar o esforço necessário à descrição dos contratos. Até mesmo porque as especificações não precisam ser completas. Ou seja, muitas vezes elas são deixadas intencionalmente incompletas de modo que se tenha mais liberdade na implementação. Em geral, essa é uma boa prática na medida em que reúne no contrato somente o que é importante a respeito do comportamento do método deixando os demais detalhes de lado. Assim, a atividade de codificação se torna mais rápida e fácil de ser realizada.

6.3.6 Extensões de Expressões Java

Um dos objetivos da linguagem JML é que esta seja de fácil utilização. Para isto, sua notação deveria ficar mais próxima possível da sintaxe Java. No entanto, apesar de JML se valer do uso de expressões Java na formalização de propriedades, ela incorporou algumas extensões no intuito de conceder maior expressividade à especificação. Tais extensões são mostradas na Tabela 6.2.

Sintaxe	Significado
a ==> *b*	*a* implica *b*
a <==> *b*	*a* se e somente se *b*
a <=!=> *b*	negação de *a* <==> *b*

Tabela 6.2: Algumas extensões incorporadas pelo JML

Código 6.13: Uso do operador de implicação

```
/*@ ensures (kgs>=0 ==> peso==\old(peso) + kgs) &&
  @         (kgs<0  ==> peso==\old(peso));
  @*/
  public void adicionaKgs(int kgs){
    if (kgs >= 0) this.peso += kgs;
  }
```

No Código 6.13, utilizamos esses operadores para modificar a especificação do Código 6.4. Essa modificação estabelece que se o parâmetro *kgs* for negativo, o atributo *peso* deve se manter inalterado, caso contrário ao seu valor deve ser adicionado *kgs*. O exemplo, apesar de simples, serve para demonstrar o potencial deste recurso.

6.3.7 Referências Não-Nulas

A anotação `non_null` é uma forma abreviada de definir que uma determinada referência não pode ser nula em qualquer estado do objeto visível externamente. É portanto uma forma simples de se definir este tipo de invariante. Com isso temos que a seguinte especificação

```
private /*@ spec_public @*/ String nome;
     public invariant nome!=null;
```

é equivalente a

```
private /*@ spec_public non_null @*/ String nome;
```

6.3.8 Métodos Puros

A principal restrição à especificação de propriedades em JML é que as expressões usadas nas asserções não podem ter nenhum efeito colateral. Ou seja, expressões de atribuição em Java (=, +=, etc...) e operadores de incremento (++) ou decremento (–) não são permitidos. Além disso, somente chamadas à métodos puros podem ser realizadas. Um método é dito puro se sua execução não altera o estado do programa. Alguns autores denominam tais métodos de métodos de consulta, pois estes podem ser usados para se obter o estado de um objeto sem modificá-lo. Em JML, deve-se utilizar o modificador `pure` na declaração do método para defini-lo como puro. Métodos que não são explicitamente declarados como puros, são considerados como sendo não puros e conseqüentemente não podem ser usados nas expressões das especificações.

No Código 6.14, temos uma demonstração envolvendo um método puro e sua utilização na pós-condição do método *isObeso()*. O contrato de *isObeso()* especifica que se o IMC (Índice de Massa Corporal) for maior que 25 este deve retornar `true`, caso contrário deve retornar `false`. O IMC é calculado pelo método *getIMC()*, com base nos

valores dos atributos *peso* e *altura*. Só foi possível utilizar o método *getIMC()* ao longo da especificação de *isObeso()*, por este ser um método puro.

Código 6.14: Uso da cláusula `pure`

```
public class Pessoa {
  protected /*@ spec_public @*/ double peso,altura;

  /*@ requires aPeso > 0 && aAltura > 0;
    @ ensures peso == aPeso && altura == aAltura;
    @*/
  public Pessoa(double aPeso, double aAltura){
    this.peso = aPeso; this.altura = aAltura;
  }

  /*@ ensures (getIMC()>25  ==> \result == true) &&
    @         (getIMC()<=25 ==> \result == false);
    @*/
  public boolean isObeso(){
    return (getIMC()>25);
  }

  //@ ensures \result == peso/(altura*altura);
  public /*@ pure @*/ double getIMC(){
    return peso/(altura*altura);
  }
}
```

6.3.9 Cláusula `assignable`

A cláusula `assignable` é utilizada para restringir que atributos podem ter seu valor ou referência modificados por um determinado método. Variáveis locais ao método ou que são criadas ao longo de sua execução não estão sujeitas a essa restrição. Na verdade, o uso do modificador `pure` (vide Seção 6.3.8) é uma forma abreviada de especificar um método como `assignable \nothing`. No Código 6.15, temos o método *mudaMedidas()*, cuja especificação estabelece que apenas as variáveis *peso* e *altura* podem ser alteradas.

Código 6.15: Uso da cláusula `assignable`

```
...
//@ assignable peso,altura;
public void mudaMedidas(int aPeso, double aAltura){
  this.peso = aPeso;
  this.altura = aAltura;
}
...
```

6.3.10 *Constraints*

Constraint é um recurso de JML que está intimamente ligado ao conceito de invariante. A diferença é que enquanto os predicados definidos nas invariantes devem ser satisfeitos em todos os estados visíveis de um objeto (ver Seção 6.2.2), as propriedades especificadas como *constraints* devem valer entre a combinação de cada um desses estados e o próximo

estado visível da execução. Ou seja, as invariantes assumem um caráter estático na medida em que estas são verificadas em determinados estados do objeto. Enquanto que as *constraints* possuem um caráter dinâmico, já que elas devem ser satisfeitas ao longo da computação que se inicia e termina em dois estados visíveis subseqüentes. *Constraints* podem ser usadas portanto, para restringir a forma com que os valores dos atributos mudam ao longo do tempo.

Para ajudar na compreensão, suponhamos que uma classe do nosso sistema possua uma estrutura de dados qualquer que armazena uma coleção de objetos. Em um contexto hipotético, queremos impor que essa coleção só possa aumentar de tamanho, nunca reduzir. Perceba que apenas com o conceito de invariante não seria possível expressar tal condição. No Código 6.16, podemos observar o uso da cláusula `constraint` para definir a regra citada acima. Valer ressaltar, que justamente pelo seu caráter dinâmico é que podemos fazer uso da cláusula `old` para estabelecer que em um estado visível qualquer o tamanho de *colecao* é sempre menor ou igual ao seu próprio tamanho no estado seguinte.

Código 6.16: Uso da cláusula `constraint`

```
...
private /*@ spec_public @*/ Collection colecao = new ArrayList();
//@ public constraint colecao.size() >= \old(colecao.size());
...
```

Diferentemente das invariantes que são obrigatoriamente globais, as *constraints* podem ainda ser definidas para um ou mais métodos específicos, conforme é mostrado logo abaixo:

`//@ public constraint` *predicado* `for` *metodo1, metodo2,...;*

Na prática, nós poderíamos pensar as *constraints* como sendo uma pós-condição implícita para todos os métodos ou para alguns em particular (caso assim seja especificado). Ou seja, este mecanismo é uma forma abreviada de impor uma determinada pós-condição a um conjunto de métodos. No entanto, lembre-se que *constraints* não se aplicam aos construtores, uma vez que os objetos não possuem um estado anterior à sua chamada.

6.3.11 Cláusula `initially`

A cláusula `initially` é mais uma abreviação utilizada em JML para simplificar a especificação de certas propriedades. Ela define a condição que deve ser satisfeita para todos os objetos de uma classe logo após sua criação (instanciação). Ela é implementada pela linguagem através da adição implícita de tal propriedade como pós-condição de cada construtor da classe. No Código 6.17, estabelece-se que todo o construtor do método deve tornar a referência para o atributo *colecao* não-nula.

6.4 Conceitos Avançados de JML

Depois de termos visto as principais construções de JML na Seção anterior, podemos agora aprofundar a discussão sobre outros aspectos importantes da linguagem, tais como a cláusula `model`, regras de visibilidade e por último o uso de quantificadores. A seguir, abordaremos detalhadamente cada um destes tópicos.

Código 6.17: Uso da cláusula `initially`

```
...
private /*@ spec_public @*/ Collection colecao = new ArrayList();
//@ initially colecao != null;
...
```

6.4.1 Cláusula `model`

Em JML, existe o modificador `model` que pode ser usado para declarar vários tipos de anotações: atributos, métodos e até mesmo classes. O significado de uma anotação declarada como `model` é que esta existe apenas para ser utilizada na especificação, não existindo portanto do ponto de vista do programa em si. Por exemplo, um atributo de modelo (aquele declarado como `model`) pode ser pensado como um atributo imaginário que serve para auxiliar a especificação de certas propriedades, mas que no entanto, não é visível fora do contexto de especificação. Outro exemplo é a cláusula `model import`, que assim como a diretiva `import` de Java é utilizada para permitir a utilização de classes de outros pacotes ao longo das anotações - esta cláusula, assim como qualquer outra que é precedida pelo modificador `model`, não tem nenhuma influência sobre o software.

Apesar dessas possibilidades, o uso mais comum deste modificador é realmente na declaração de atributos de modelo, o que torna possível criar abstrações sobre os atributos concretos da classe. Ou seja, o valor de um atributo de modelo é determinado pelos atributos concretos dos quais ele abstrai. Essa relação é definida pela cláusula `represents`. Diferentemente de atributos, métodos e classes declarados com `model` não são abstrações de métodos e classes concretas, respectivamente. Eles são simplesmente métodos ou classes que imaginamos fazer parte do programa, cujo único propósito é ajudar nas atividades de anotação. Apesar de todos esses recursos serem utilizados somente para fins de especificação, o escopo para as declarações é o mesmo de Java. Isso implica na impossibilidade de se declarar com o mesmo nome, um atributo de modelo e um atributo normal simultaneamente em uma classe Java.

O Código 6.18, exemplifica bem o uso da cláusula `model`. Ele é uma variação do Código 6.14, no qual o método *getIMC()* não é mais puro, pois este agora atualiza o atributo *qtdImcCalculado*. Seria incômodo se tivéssemos que criar um outro método para calcular o IMC (que não incremente a variável e que seja declarado como puro) somente para fins de especificação. Para resolver este problema, utilizamos o modificador `model` para declarar o atributo de modelo *imc*, que nada mais é do que a representação do cálculo do IMC sobre os atributos *peso* e *altura*. Perceba que ambos os métodos *isObeso()* e o próprio *getIMC()* são agora especificados com base na definição imposta pela abstração *imc*.

Esse mecanismo torna-se possível através da cláusula `represents`, que exige que:

- O lado esquerdo da cláusula referencie uma atributo de modelo;
- O tipo da expressão do lado direito deve ser compatível com o do atributo declarado no lado direito.

Uma cláusula `represents` também pode ser declarada como `static`. Nesse caso, apenas elementos `static` podem ser referenciados, tanto do lado esquerdo quanto do lado direito da cláusula.

Código 6.18: Exemplo do uso de atributos de modelo

```
public class PessoaModelo {
  private /*@ spec_public @*/ double peso,altura;
  private /*@ spec_public @*/ int qtdImcCalculado = 0;

  //@ public model double imc;
  //@ private represents imc <- peso/(altura*altura);

  /*@ requires aPeso > 0 && aAltura > 0;
    @ ensures peso == aPeso && altura == aAltura;
    @*/
  public PessoaModelo(double aPeso, double aAltura){
    this.peso = aPeso;
    this.altura = aAltura;
  }

  /*@ ensures (imc>25  ==> \result == true) &&
    @         (imc<=25 ==> \result == false);
    @*/
  public boolean isObeso(){
    if(getIMC()>25) return true;
    else return false;
  }

  /*@ ensures \result == imc &&
    @         qtdImcCalculado == \old(qtdImcCalculado) + 1;
    @*/
  public double getIMC(){
    qtdImcCalculado++;
    return peso/(altura*altura);
  }
}
```

6.4.2 Modificadores de Visibilidade

A linguagem Java impõe uma série de regras ao controle de acesso à atributos, métodos e construtores declarados em uma classe. Tais regras dependem exclusivamente da visibilidade destes elementos, que por sua vez podem ser classificados como `private`, `protected`, `public` ou visibilidade padrão (*package*). Um nome declarado em uma classe *P.C* (onde *P* é o pacote e *C* é a classe) pode ser referenciado fora de *P* se e somente se ele é declarado como `public`, ou se este é declarado como `protected` e a referência ocorra em alguma sub-classe de *P.C*. Este nome pode ser referenciado por qualquer outra classe do pacote *P* se for declarado como `public` ou com a visibilidade padrão (quando não é usado nenhum modificador). Além disso, tal nome pode sempre ser utilizado dentro da própria classe, mesmo que este seja declarado como `private`. Maiores detalhes sobre as regras de visibilidade, consulte a especificação da linguagem Java.

Além destas, JML estabelece uma regra adicional: uma anotação não pode referenciar elementos (atributo, método ou construtor) que estão “mais escondidos” do que ela própria. Sendo que os possíveis níveis de visibilidade das anotações são os mesmos de Java. Ou seja, para a referência à uma variável *x* ser válida, a visibilidade do contexto de anotação que a referencia deve ser tão permissiva quanto a própria declaração de *x*. Mais

detalhadamente, suponha que *x* seja uma variável declarada em uma classe *P.C*. Temos as seguintes regras:

- Uma expressão em um contexto `public` de especificação (um contrato de um método, por exemplo) pode referenciar *x* somente se *x* for declarado como `public`.
- Uma anotação em um contexto `protected` pode referenciar *x* somente se x for declarado como `public` ou `protected`. Além disso, se *x* for `protected`, a referência deve ser feita de dentro da própria classe ou a partir de alguma de suas sub-classes.
- Uma anotação cuja visibilidade seja *package* (caso desta não ser declarada com nenhum modificador de visibilidade) pode referenciar *x* somente se *x* for declarado como `public`, `protected`, ou *package*. Além disso, se *x* tiver visibilidade *package*, a referência deve ser feita de alguma classe do seu pacote.
- Por fim, uma anotação com visibilidade `private` pode referenciar *x* somente se *x* também for `private`.

O motivo desta restrição se baseia no fato de que teoricamente as pessoas permitidas a ler uma determinada anotação, deveriam também poder olhar cada uma das referências usadas em sua especificação, caso contrário elas não conseguiriam entendê-la. Por exemplo, os usuários de uma biblioteca de software deveriam ser capazes de ver todas as declarações de métodos e atributos utilizados por uma anotação pública, uma vez que esta não pode conter nenhum acesso a elementos com visibilidade `protected`, `private` ou *package*.

Apesar do Código 6.19 referir-se somente à invariantes, poderia ter sido usado qualquer outro tipo de anotação, tais como *constraints* ou a até mesmo a especificação de um método, para exemplificar as regras citadas logo acima.

No caso da especificação de métodos, vale ressaltar que o nível de visibilidade da anotação é implicitamente definido como sendo o mesmo do método que esta especifica. Ou seja, considera-se que a especificação de um método `protected` por exemplo, também possua um contexto `protected` para fins de checagem das regras de visibilidade. As palavras reservadas da linguagem JML `spec_public` e `spec_protected` são o meio pelo qual torna-se possível gerar especificações em casos que não há compatibilidade entre a visibilidade da anotação e do código Java. Por exemplo, o seguinte código declara um atributo do tipo inteiro como sendo `private` para Java, mas que JML a considera como `public`.

```
private /*@ spec_public @*/ int tamanho;
```

Com isso, torna-se possível utilizar o atributo `tamanho` em uma invariante pública, por exemplo. No entanto, `spec_public` é mais do que uma forma de modificar a visibilidade de um elemento do código Java para fins de especificação. Quando aplicados a atributos, este pode ser considerado um forma abreviada de declarar um atributo de modelo de mesmo nome. Ou seja, a declaração acima pode ser pensada como sendo equivalente às seguintes declarações (caso o código Java que referencia o atributo `tamanho` fosse reescrito para passar a referenciar *_tamanho*).

```
//@ public model int tamanho;
```

Código 6.19: Uso dos modificadores de visibilidade

```
public class DemoPrivacidade{

  public int pub;
  protected int prot;
  private int priv;
  int def;

  //@ public invariant pub > 0; // legal
  //@ public invariant prot > 0; // ilegal!
  //@ public invariant def > 0; // ilegal!
  //@ public invariant priv < 0; // ilegal!

  //@ protected invariant pub > 1; // legal
  //@ protected invariant prot > 1; // legal
  //@ protected invariant def > 1; // ilegal!
  //@ protected invariant priv < 1; // ilegal!

  //@ invariant pub > 1; // legal
  //@ invariant prot > 1; // legal
  //@ invariant def > 1; // legal
  //@ invariant priv < 1; // ilegal!

  //@ private invariant pub > 1; // legal
  //@ private invariant prot > 1; // legal
  //@ private invariant def > 1; // legal
  //@ private invariant priv < 1; // legal
}
```

```
private int _tamanho; //@ in tamanho;
//@ private represents tamanho <- _tamanho;
```

Conforme citado na Seção anterior, esse mecanismo permite a mudança do código Java sem que isto afete os leitores da especificação. A mesma equivalência vale para `spec_protected`, bastando para isso utilizar o modificador `protected` na primeira linha, ao invés de `public`.

6.4.3 Quantificadores

JML suporta vários tipos de quantificadores: um quantificador universal (`\forall`), um quantificador existencial (`\exists`), quantificadores de generalização (`\sum`, `\product`, `\min` e `\max`) e um quantificador numérico (`\num_of`). No Código 6.20, a pré-condição exige que o *array* não seja nulo, e que para todos os inteiros *i*, se *i* é maior que 0 e menor que o tamanho do *array a*, então o elemento *a[i-1]* não é maior que do que o elemento *a[i]*. Em outras palavras, esta asserção define que o *array a* deve estar ordenado de forma crescente.

Este tipo de construção se assemelha a um laço `for` de Java, com uma declaração (sem inicialização), um predicado que descreve a faixa de valores da variável declarada e finalmente uma expressão booleana (que deve ser verdadeira para todos os valores dentro da faixa especificada). A definição da faixa é opcional, mas, caso omitida, a especificação não pode ser verificada - apesar de ainda ser útil para fins de documentação.

Código 6.20: Quantificador universal `forall`

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

Os quantificadores `\sum`, `\product`, `\min` e `\max` retornam, respectivamente, a soma, produto, mínimo e máximo de um determinado conjunto de valores que satisfazem certa expressão. No exemplo abaixo, todas as asserções são verdadeiras.

```
(\sum int i; 0 <= i && i < 5; i) == 0 + 1 + 2 + 3 + 4
(\product int i; 0 < i && i < 5; i) == 1 * 2 * 3 * 4
       (\max int i; 0 <= i && i < 5; i) == 4
      (\min int i; 0 <= i && i < 5; i-1) == -1
```

O quantificador numérico `\num_of`, retorna o número de valores da variável quantificada que se encontra dentro da faixa definida e cujo predicado é verdadeiro. O Código 6.21, apresenta uma invariante que estabelece que o número máximo de alunos com menos de 18 anos em uma determinada turma é 10.

Código 6.21: Quantificador numérico `num_of`

```
...
public static final int MAX_MENORES = 10;
Turma turma = new Turma();
//@ public invariant (\num_of Estudante e;
                              turma.contains(e); e.idade < 18)
                                     <= MAX_MENORES;
...
```

6.5 Técnicas e Ferramentas JML

Assim como qualquer linguagem de programação, as linguagens de especificação (tal como JML) necessitam de um suporte ferramental apropriado, sem o qual torna-se quase impossível a sua efetiva utilização. De modo geral, essas ferramentas devem automatizar ou no mínimo facilitar as tarefas de leitura, escrita e verificação das anotações. Como veremos nesta Seção, JML conta com uma variedade de soluções abrangendo cada um desses aspectos, motivo pelo qual seu uso é de fato viável em projetos reais de desenvolvimento de software. A ferramenta básica, da qual quase todas as outras fazem uso, é o (JML *checker*), responsável por verificar a conformidade da sintaxe e semântica das anotações. A existência desta ferramenta por si só já é uma grande vantagem sobre os comentários informais, na medida em que fornece meios para manter a especificação minimamente consistente.

A maneira mais óbvia de checar anotações JML é verificá-las dinamicamente, de forma que caso uma violação da especificação seja detectada em tempo de execução, o desenvolvedor possa ser notificado. As principais ferramentas que funcionam desta forma são *JMLRAC* (JML Runtime Assertion Checker) e *JMLUnit*. É possível também verificar a conformidade entre especificação e código sem que para isso seja necessário executar o software. Esta abordagem, chamada aqui de checagem estática, é muito mais audaciosa pois tenta estabelecer a correção do código para todos os possíveis caminhos de execução, o que não acontece na checagem dinâmica. Embora este problema não seja decidível para propriedades mais complexas, as ferramentas de checagem estática têm conseguido avanços importantes nesse sentido. A mais importante delas é a ferramenta *Esc/Java2*.

Por último, não podemos deixar de citar a ferramenta *JMLDoc*, uma extensão do tão conhecido gerador de documentação (em formato html) para projetos Java - *Javadoc*. Sua finalidade é adicionar à documentação gerada pelo *Javadoc*, as informações oriundas das anotações JML.

Código 6.22: Classe contendo um erro para demonstração das ferramentas

```
1 public class Pessoa {
2   private /*@ spec_public non_null @*/ String nome;
3   private /*@ spec_public @*/ int peso;
4
5   //@ initially peso == 0;
6   //@ public invariant !nome.equals("") && peso >= 0;
7
8   /*@ requires aNome!=null && !aNome.equals("");
9     @ ensures nome.equals(aNome);
10    @*/
11  public Pessoa(String aNome){
12    this.nome = aNome; this.peso = 0;
13  }
14
15  //@ ensures nome == aNome;
16  public void setNome(String aNome){
17    this.nome = aNome;
18  }
19
20  /*@ requires peso + kgs > 0;
21    @ ensures peso == \old(peso) + kgs;
22    @ signals (Exception e) kgs < 0 && peso == \old(peso); @*/
23  public void adicionaKgs(int kgs) throws Exception{
24    if (kgs >= 0)
25      this.peso += kgs;
26    else throw new Exception("O peso não pode ser nulo!");
27  }
28
29  /*@ also
30    @ ensures \result !=null && !\result.equals("") &&
31    @         (* \result imprime todos seus atributos *); @*/
32  public String toString(){
33    return "nome="+ nome +" peso=" + peso;
34  }
35 }
```

Abordaremos a seguir, cada uma dessas ferramentas detalhadamente. Para isto, iremos utilizar a classe `Pessoa`, mostrada no Código 6.22, que propositalmente contém um erro a ser identificado pelas ferramentas de verificação. Esta classe é a representação de uma pessoa com dois atributos: nome e peso. O nome nunca pode ser nulo ou vazio e o peso deve ser sempre maior ou igual a 0. Além disso, é possível alterar seu peso através de uma chamada ao método *adicionaKgs()*. Este por sua vez, pode vir a lançar uma exceção caso o parâmetro *kgs* seja negativo. Se isto ocorrer, significa que o peso se manteve inalterado. E finalmente, a pessoa deve ser capaz de retornar uma representação textual (também não-nula e não vazia) de si mesma através do método *toString()*.

6.5.1 *JML Runtime Assertion Checker*

O objetivo desta ferramenta é encontrar inconsistências entre a especificação e o código através da execução das asserções - enquanto o software está sendo executado, um monitor checa se houve alguma violação da especificação e notifica o usuário. Assim como em qualquer técnica de verificação, espera-se concluir que um determinado código está incorreto em relação à sua especificação. No entanto, pode ser que a própria especificação é que esteja incorreta (em relação ao que o usuário tem em mente), enquanto o código na verdade não tenha nenhum problema. Portanto, queremos ressaltar que encontrar problemas na especificação de um software é tão importante quanto encontrar erros em seu código. A especificação é um artefato do projeto que deve ser mantido consistente e atualizado. Este esforço é drasticamente reduzido graças a característica de JML de não ser ambígua, o que conseqüentemente torna possível a automação de diversas tarefas.

Um dos requisitos fundamentais a um verificador desta natureza, é que ele seja capaz de dar informações que permitam a rápida identificação do que deve ser feito para corrigir uma inconsistência. Por conta disso, o *JMLRAC* procura fornecer dados que efetivamente ajudem os usuário nesse sentido, englobando tanto informações de caráter estático quanto dinâmico. Informações estáticas incluem coisas como que partes da especificação foram violadas e a localização no código, do erro. Por outro lado, informações dinâmicas incluem os valores das variáveis e quais chamadas de métodos levaram à violação.

O verificador deve ser acima de tudo confiável, no sentido de não notificar falsas violações. Por exemplo, JML permite o uso de descrições informais nas asserções para fins de documentação. No entanto, caso a ferramenta atribua algum valor particular à esse trecho da especificação, é possível que uma violação seja relatada sem que de fato ela exista. Por isso, em tais situações o *JMLRAC* simplesmente ignora a existência dessas descrições informais.

Outra característica importante é que a execução do software deve ser transparente ao usuário enquanto nenhuma violação for encontrada. Ou seja, exceto pelo seu desempenho, um programa com o mecanismo de checagem de asserções habilitado e consistente com sua especificação deve se comportar exatamente da mesma forma como se este mecanismo não existisse. Este aspecto é garantido pela própria linguagem JML que impede que as asserções tenham algum efeito colateral (vide Seção 6.3.8).

Apesar da ferramenta não impor nenhuma metodologia particular, existem algumas diretrizes que podem ser úteis na sua utilização. Uma técnica básica é especificar primeiramente as pré-condições para o comportamento normal do método, de modo a garantir que ele esteja sendo invocado da maneira esperada. Para facilitar a depuração é

recomendável que cada classe do sistema implemente o método *toString()*. Desse forma o *JMLRAC* pode exibir o estado do objeto nas mensagens de violação. Em seqüência, defini-se as pós-condições do método para certificar que este está desempenhando sua função corretamente. Em paralelo, as invariantes podem ser especificadas gradualmente na medida em que, durante a implementação dos métodos, novas propriedades vão sendo descobertas.

Funcionamento

Tendo como entrada um programa Java anotado com JML, o uso da ferramenta *JMLRAC* engloba as seguintes etapas:

1. Checagem sintática das anotações;
2. Compilação do programa com inserção automática de instruções que checam as asserções em tempo de execução;
3. Execução do código instrumentado.

O comando *jmlc* é o responsável pelas duas primeiras etapas. Na verdade, ele é um compilador Java que interpreta as anotações JML e gera os *bytecodes* (*.class*) devidamente instrumentados. Essa instrumentação tem por objetivo fazer com que o próprio código seja capaz de detectar violações de sua especificação. Portanto, sua utilização é bastante similar ao uso do compilador *javac* de Java, conforme é mostrado logo abaixo:

```
jmlc Pessoa.java
```

Este comando produz um arquivo correspondente, `Pessoa.class`, no diretório corrente. Algumas vezes é conveniente redirecionar a saída para que as classes compiladas sejam colocadas em um diretório diferente. Para fazer isto, basta utilizar a opção *-d* seguido do caminho do diretório destino.

```
jmlc -d ../bin Pessoa.java
```

Para maiores detalhes sobre as opções disponíveis para compilação, leia o manual da ferramenta que faz parte da distribuição JML. Alternativamente, há uma interface gráfica chamada *jmlc-gui*, que pode ser utilizada para executar o compilador. Ele torna o processo de compilação mais fácil através do uso de *menus* para a escolha dos arquivos e das opções desejadas. Para executar o programa compilado com o *jmlc*, basta incluir a biblioteca *jmlruntime.jar* no *classpath* e depois invocar normalmente a máquina virtual Java com o comando *java*. Tudo isto é feito automaticamente se o comando *jmlrac* for utilizado. Para demonstrar o uso da ferramenta na classe `Pessoa`, mostrada no Código 6.22, optamos por colocar o método `main` da aplicação em uma outra classe chamada `PessoaMain` (Código 6.23). Esta não possui nenhuma anotação JML, uma vez que seu objetivo é exclusivamente invocar os métodos da classe `Pessoa` sob algumas circunstâncias, as quais denominamos de cenários de uso.

O trecho abaixo é uma transcrição que mostra como compilar e executar as classes `Pessoa` e `PessoaMain`.

```
> jmlc -Q Pessoa.java PessoaMain.java
> jmlrac PessoaMain 1
```

```
Exception in thread "main"
        org.jmlspecs.jmlrac.runtime.JMLEntryPreconditionError:
        by method Pessoa.Pessoa regarding specifications at
File "Pessoa.java", line 9, character 29 when
        'aNome' is null
        at Pessoa.checkPre$$init$$Pessoa(Pessoa.java:303)
        at Pessoa.<init>(Pessoa.java:31)
        at Pessoa.main(Pessoa.java:118)
```

Código 6.23: Classe responsável por exercitar os cenários de uso do Código 6.22

```
1  public class PessoaMain {
2    public static void main(String[] args){
3      try {
4        Pessoa p;
5         int cenario = new Integer(args[0]).intValue();
6         switch(cenario){
7           case 1:{
8             p = new Pessoa(null);
9           }
10           case 2:{
11             p = new Pessoa("Joao");
12             p.adicionaKgs(70);
13             p.adicionaKgs(-40);
14           }
15           case 3:{
16             p = new Pessoa("Joao");
17             p.setNome(null);
18           }
19         }
20     } catch (Exception e) {}
21   }
22 }
```

A opção *-Q* do comando *jmlc*, faz com que a compilação ocorra sem que sejam impressas na tela informações detalhadas sobre o andamento do processo. A escolha do cenário de uso a ser utilizado na execução é determinado pelo argumento passado à classe `PessoaMain` na linha de comando. Para o cenário número 1, a saída indica que houve uma violação da pré-condição do construtor *Pessoa()*, especificada na linha 9. O problema é que este cenário de uso tenta instanciar um objeto *p* com o nome nulo - o que faz com que haja a quebra do contrato do construtor. No entanto, esta violação não indica um erro na classe Pessoa (seja na sua especificação ou implementação) e sim que o cliente não atendeu às condições necessárias na instanciação.

Quando executamos novamente a classe `PessoaMain` para o cenário de uso 2, nenhuma violação é detectada. No entanto, para o cenário 3 a seguinte saída é gerada:

```
> jmlrac PessoaMain 3
Exception in thread "main"
        org.jmlspecs.jmlrac.runtime.JMLInvariantError:
        by method Pessoa.setNome@post<File "Pessoa.java",
        line 16, character 20> regarding specifications at
File "Pessoa.java", line 6, character 46 when
```

```
'this' is nome=null peso=0
at Pessoa.checkInv$instance$Pessoa(Pessoa.java:197)
at Pessoa.setNome(Pessoa.java:551)
at PessoaMain.main(PessoaMain.java:27)
```

Dessa vez, houve a violação da invariante especificada na linha 6. Ou seja, a classe não foi capaz de manter a invariante ao longo de sua execução. A violação de uma pós-condição ou de uma invariante, assim como esta, isenta o cliente da culpa e indica que o erro (de especificação ou implementação) se encontra na própria classe `Pessoa`. Nesse caso especificamente, de forma proposital omitimos a pré-condição do método *setNome()* que define que, assim como no construtor, o argumento *aNome* não pode ser nulo nem vazio. Vale ressaltar que a implementação do método *toString()*, torna ainda mais fácil identificar a causa do problema uma vez que este mostra o estado do objeto no exato momento da violação - *'this' is nome=null peso=0*.

Essa demonstração de uso do *JMLRAC* deixa claro que se o software não for devidamente estimulado, algumas violações poderão nunca ser detectadas. Por isso, essa ferramenta geralmente não é utilizada para verificar classes isoladas mas para verificar interação real entre os objetos do sistema. Ou seja, o programador ao implementar uma classe como `Pessoa` poderia até utilizar esta ferramenta para tentar capturar erros básicos (assim como mostrado logo acima), porém a verificação verdadeira se daria quando o módulo a qual a classe pertence fosse sendo utilizado, de preferência ao longo de seu desenvolvimento e antes de ser posto em produção. De toda essa discussão podemos concluir que:

> Na checagem dinâmica, as chances de alguma violação ser detectada depende diretamente da qualidade dos cenários de uso escolhidos.

Um outro aspecto importante é que não é necessário compilar todos os fontes com *jmlc*. Por exemplo, a classe `PessoaMain` poderia ter sido compilada com *javac*, mas mesmo assim a classe `Pessoa` (esta sim compilada com o *jmlc*) continuaria tendo suas asserções verificadas em tempo de execução. Para facilitar ainda mais o uso de JML, está disponível gratuitamente na Internet um *plugin* para a plataforma de desenvolvimento Eclipse que provê dentre outras coisas:

- Checagem sintática das anotações
- *Highlighting* do código de especificação
- Interface para a ferramenta de verificação de asserções em tempo de execução (*jmlrac*)
- Integração com a ferramenta *jmldoc*

6.5.2 Teste de Unidade - *jmlunit*

O principal objetivo da ferramenta *jmlunit* é utilizar as anotações JML para gerar automaticamente as classes responsáveis pelos testes de unidade do sistema. Com isso, pretende-se fazer com que o desenvolvedor não precise mais codificar os critérios que

decidem se um determinado teste de unidade passou ou não (este procedimento é comumente chamado de oráculo de teste). Mais especificamente, as classes são geradas para o *framework* JUnit, que nada mais é do que uma plataforma de execução para testes de classes Java. A idéia é que essas classes ao enviarem mensagens aos objetos Java do sistema sob teste, previamente compilado com o comando *jmlc*, possam capturar exceções derivadas da violação de algum contrato. Essa violação não constitui necessariamente uma falha do teste, pois podem existir três situações:

- Quando a pré-condição de um método sendo testado é violada, não podemos concluir que a classe contém um erro. A violação indica somente que esta foi utilizada da maneira incorreta. Nesse caso, o teste é considerado como bem-sucedido.
- Quando a pré-condição é satisfeita, mas alguma outra asserção é violada, podemos concluir que a implementação falhou em cumprir o que foi especificado. Nesse caso, o teste deve indicar uma falha.
- Existe ainda a possibilidade do método sob teste, que teve sua própria pré-condição satisfeita, invocar um outro método sem no entanto satisfazer a pré-condição deste último, o que claramente também constitui um erro.

Em outras palavras, o código gerado é usado para exercitar as classes sob teste e decidir se estas cumprem ou não o que sua especificação estabelece. O desenvolvedor continua sendo o responsável por fornecer os dados de teste, apesar desta tarefa ser enormemente facilitada pelas classes geradas. A ferramenta inclui ainda um conjunto padrão de dados de teste para os tipos primitivos de Java. Além disso, é possível codificar manualmente outros testes nas classes geradas para explorar cenários de uso não contemplados.

Funcionamento

Mostraremos agora como utilizar a ferramenta para testar a classe do Código 6.22. O primeiro passo é executar o comando *jmlunit* na classe `Pessoa` de forma a produzir os arquivos: `Pessoa_JML_TestData.java` (no qual os dados de testes serão colocados) e `Pessoa_JML_Test.java` (que contém os testes propriamente ditos). Na classe `Pessoa_JML_TestData.java` é criado um atributo para cada tipo de dado utilizado como argumento nos métodos da classe sob teste. Por exemplo, a classe `Pessoa` precisa de dados de teste do tipo inteiro (para usar no método *adicionaKgs*) e do tipo `String` (para usar no construtor e no método *setNome*). Conseqüentemente, a classe `Pessoa_JML_TestData.java` será gerada com os atributos *vintStrategy* e *vStringStrategy*, respectivamente. Como havíamos falado anteriormente, para os tipos de Java a ferramenta utiliza como padrão de dados de teste os valores -1, 0 e 1 para o tipo `int` e os valores `""` e `null` para o tipo `String`. Isto significar dizer que os métodos e construtores que recebem como parâmetro quaisquer desses tipos serão testados com esses valores. Caso o desenvolvedor queira fornecer dados adicionais, basta sobrescrever o método *addData()* correspondente ao seu atributo, conforme mostrado no Código 6.24. Vale lembrar que a adição de dados de testes para tipos de Java é opcional.

Resta ainda fornecer à ferramenta que objetos da classe `Pessoa` serão testados. Isto é feito de maneira análoga aos tipos primitivos - existe um atributo com o nome

Código 6.24: Dados de teste adicionais em `Pessoa_JML_TestData.java`

```
...
private org.jmlspecs.jmlunit.strategies.IntStrategyType
    vintStrategy
    = new org.jmlspecs.jmlunit.strategies.IntStrategy()
        {
            protected int[] addData() {
                return new int[] {
                    100, -100 //dados de testes adicionais
                };
            }
        };
...
```

vPessoaStrategy cujo método *make()* (ao invés de *addData()*) deve ser obrigatoriamente implementado. Caso isto não seja feito, a ferramenta será incapaz de executar os testes, uma vez que esta não dispõe de objetos para testar. Com base no inteiro recebido como parâmetro pelo método *make()*, deve-se codificar os as instâncias a serem testadas, conforme escolha dos dados de teste. O Código 6.25 mostra que decidimos utilizar apenas um único objeto `Pessoa` (inicializado com o nome ”Joao”).

Código 6.25: Definição dos objetos que serão testados

```
...
private org.jmlspecs.jmlunit.strategies.StrategyType
    vPessoaStrategy
    = new org.jmlspecs.jmlunit.strategies.NewObjectAbstractStrategy()
        {
            protected Object make(int n) {
                switch (n) {
                case 0:
                          return new Pessoa("Joao"); //objeto de teste
                default:
                   break;
                }
                throw new java.util.NoSuchElementException();
             }
         };
...
```

Após definidos os dados de testes, está tudo pronto para testarmos nossa classe. Basta agora

1. Compilar as classes sob teste (nesse caso `Pessoa`) com *jmlc*
2. Compilar as classes geradas pela ferramenta com o compilador normal de Java (*javac*). É preciso incluir no *classpath* as bibliotecas `junit.jar`, `jmljunitruntime.jar` e `jmlruntime.jar` (as duas últimas fazem parte do pacote da ferramenta)
3. E finalmente rodar os testes através da execução da classe `Pessoa_JML_Test` usando o comando *jmlrac* (saída em modo texto) ou *jml-junit* (interface gráfica do JUnit).

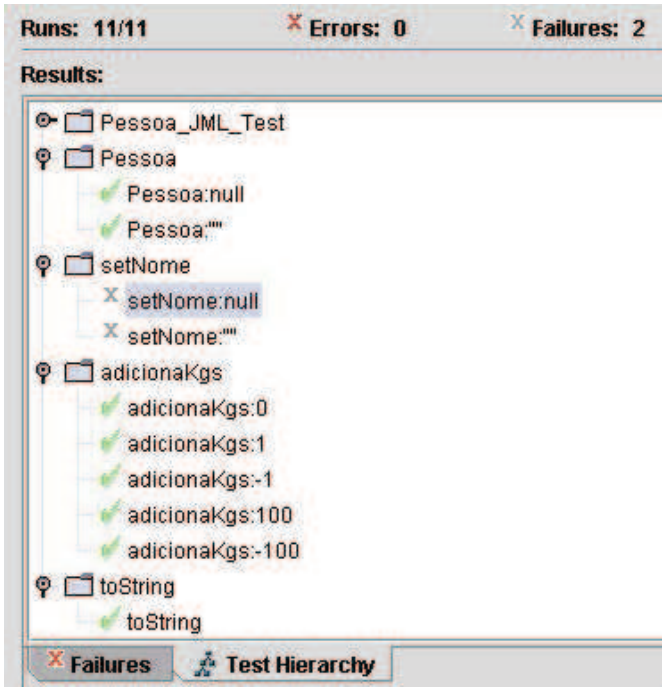


Figura 6.1: Resultado da execução dos testes

O resultado da execução dos testes pode ser visualizado na Figura 6.1. Ao total foram gerados 11 testes, sendo que um deles (*isRACCompiled()*) é um teste interno da ferramenta que checa antes de mais nada se a classe `Pessoa` foi realmente compilada com *jmlc*. Na Tabela 6.3 é mostrado um resumo da execução dos testes. Assim como na Seção anterior, o erro referente ao método *setNome()* foi detectado.

Método	**Testes**	**Tipo**	**Argumentos**	**Resultado**
isRACCompiled()	1	-	-	Passou
Pessoa()	2	String	"" e `null`	Passou
setNome()	2	String	"" e `null`	Falhou
adicionaKgs()	5	int	-1, 0, 1, 100 e -100	Passou
toString()	1	-	-	Passou

Tabela 6.3: Esquema de testes

Este tipo de teste é muito útil para encontrar pré-condições que por engano não tenham sido consideradas durante a especificação da classe. Quando um erro é encontrado devido à violação de uma invariante ou pós-condição, certamente ou a implementação está errada ou está faltando alguma pré-condição. Nosso exemplo mostra bem essa situação. Vale lembrar também que se todas as anotações da classe `Pessoa` fossem removidas, os testes de unidade gerados da forma descrita até aqui não seriam capazes de detectar nenhum erro. Isto porque a ferramenta utiliza justamente o mecanismo de violação de asserções para decidir se uma classe possui o comportamento esperado. Se não há nenhuma asserção, é impossível haver qualquer violação. Este fato nos mostra uma importante observação:

Experiências mostram que esta ferramenta permite a realização de testes de

A qualidade dos testes gerados pela ferramenta *jmlunit* é tão boa quanto a especificação JML produzida.

unidade com um mínimo esforço de codificação além de ser eficaz na identificação de diversos tipos de erros. Estudos mostram que cerca de metade das falhas encontradas são causadas por erros na especificação, o que mostra sua utilidade também na depuração das anotações. No entanto, a abordagem exige que a descrição do comportamento esperado seja razoavelmente completa, já que a qualidade dos testes gerados depende diretamente da qualidade da especificação. Isso não chega a ser um problema pois a técnica de certa forma substitui o esforço despendido na codificação dos testes pelo esforço de especificação.

Contudo, testes de unidades realizados com *jmlunit* possuem uma limitação: eles apenas detectam erros que são resultados de uma única chamada simples aos métodos da classe. Caso exista um erro que só ocorra depois que os métodos *a()* e *b()* sejam invocados em seqüência, a ferramenta não será capaz de encontrá-lo. Isso não necessariamente restringe sua capacidade de encontrar erros, no entanto impõe que os dados de testes sejam cuidadosamente escolhidos - os objetos que serão testados precisam se encontrar em estados que permitam exercitar todas as possibilidades. Por exemplo, suponhamos que se deseje testar uma classe que implementa uma pilha de tamanho limitado. Se fornecermos como dado de teste somente uma pilha vazia (de maneira análoga ao Código 6.25), a situação em que se tenta inserir um elemento em uma pilha cheia nunca será testada. Nesse caso, o ideal seria fornecer um pilha vazia, uma cheia e outra parcialmente cheia para exercitar todas as possíveis situações.

6.5.3 Checagem Estática com *ESC/Java2*

A ferramenta *ESC/Java2* realiza o que é chamado de checagem estática estendida, uma técnica de verificação em tempo de compilação que vai além da simples checagem de tipos realizada pelos compiladores convencionais (por isso o termo estendida). *ESC/Java2* suporta grande parte da linguagem JML (incluindo todas as construções vistas até aqui), e para este subconjunto checa a consistência entre o código e as anotações. A interação com usuário é similar a um compilador qualquer: codifica a especificação em JML e executa a ferramenta, esta por sua vez lista todos os possíveis erros existentes no programa.

As anotações JML são usadas pela ferramenta de duas maneiras. Primeiro elas ajudam o *ESC/Java2* a suprimir mensagens sobre falsos erros. Por exemplo, vamos supor que para uma pessoa ser considerada igual a outra, basta que elas tenham o mesmo nome. Para atender a este requisito adicionamos o trecho mostrado no Código 6.26 à classe `Pessoa`. Nesse caso, a pré-condição `p!=null` do método *equals()*, impede que a ferramenta produza uma advertência sobre uma possível referência nula ao objeto *p* na chamada ao método *getNome()*.

Em segundo lugar, anotações JML fazem com que aspectos adicionais sejam checados pela ferramenta. Por exemplo, quando alguma classe invoca o método *equals()*, a pré-condição `p!=null` induz o *ESC/Java2* a emitir um aviso caso o parâmetro *p* possa ser passado com o valor *null*. Diferentemente das outras ferramentas vistas até aqui, *ESC/Java2* não depende das anotações JML para ser utilizada - apesar destas contribuírem bastante para a sua eficácia.

Código 6.26: Definição dos objetos que serão testados

```
...
  //@ requires p!=null
  //@ ensures \result == nome.equals(p.getNome());
  public boolean equals(Pessoa p){
        return nome.equals(p.getNome());
  }

  public /*@ pure @*/ String getNome() {
        return nome;
  }
...
```

Um propriedade interessante da ferramenta é que ela não é nem sólida (conceito matemático de *soundness*) nem completa. Não ser sólida significa dizer que *ESC/Java2* não se compromete em notificar todos os erros. Por incompleta entenda-se que nem sempre esses avisos correspondem verdadeiramente a erros no software. Pode parecer estranho, mas esta foi uma escolha proposital imposta ao seu *design*. Sem essa característica, a ferramenta deixaria a desejar em outros aspectos como eficiência e automação (seria necessário a interação do usuário para guiar o processo de verificação). Portanto, para fazer com que a ferramenta fosse de fato viável de ser utilizada foi prudente ignorar a possibilidade de alguns tipos de erros. No entanto, isto não é suficiente para fazer com que a ferramenta deixe de ser de grande utilidade no desenvolvimento de software (como veremos adiante em um exemplo).

Em essência, *ESC/Java2* traduz os programas anotados com JML em fórmulas lógicas, que por sua vez são submetidas ao seu provador de teoremas (não interativo). Tais fórmulas são logicamente válidas se e somente se o código não contém o erro sendo analisado. Quando uma fórmula não é verificada, o contra-exemplo encontrado pelo provador de teorema é traduzido em mensagens legíveis ao desenvolvedor, o que inclui informações sobre o tipo e o local em que o erro em potencial foi localizado. Na verdade, *ESC/Java2* é o sucessor da ferramenta originalmente chamada de *ESC/Java*. Apesar de *ESC/Java2* manter-se fiel aos objetivos da versão anterior, ela traz importantes avanços tais como:

- Compatibilidade com a versão 1.4 de Java
- Capacidade de aceitar anotações englobando toda a linguagem JML
- Aumento do conjunto de construções JML passíveis de verificação estática

Funcionamento

A utilização da ferramenta é bastante simples. Basta executar o comando *escj* passando como parâmetro o arquivo .java que se deseja checar. Quando a ferramenta é aplicada ao Código 6.22, a seguinte saída é produzida:

```
> escj -Suggest Pessoa.java
ESC/Java version ESCJava-2.0a8
    [0.291 s 4369344 bytes]

Pessoa ...
```

```
  Prover started:0.047 s 8027736 bytes
    [3.666 s 7737944 bytes]
Pessoa: Pessoa(java.lang.String) ...
    [1.387 s 8493952 bytes]  passed
Pessoa: setNome(java.lang.String) ...
------------------------------------------------------------
Pessoa.java:19: Warning: Possible assignment of null to
                         variable declared non_null (NonNull)
                this.nome = aNome;
                            ^
Associated declaration is "Pessoa.java", line 3, col 25:
        private /*@ spec_public non_null @*/ String nome;
                                ^
Suggestion [19,12]: perhaps declare parameter 'aNome' at 18,28
                    in Pessoa.java with 'non_null'
------------------------------------------------------------
    [0.61 s 7962464 bytes]  failed
Pessoa: equals(Pessoa) ...
    [0.526 s 8225416 bytes]  passed
Pessoa: getNome() ...
    [0.181 s 8666848 bytes]  passed
Pessoa: adicionaKgs(int) ...
    [0.872 s 8627888 bytes]  passed
  [7.249 s 8628768 bytes total]
1 warning
```

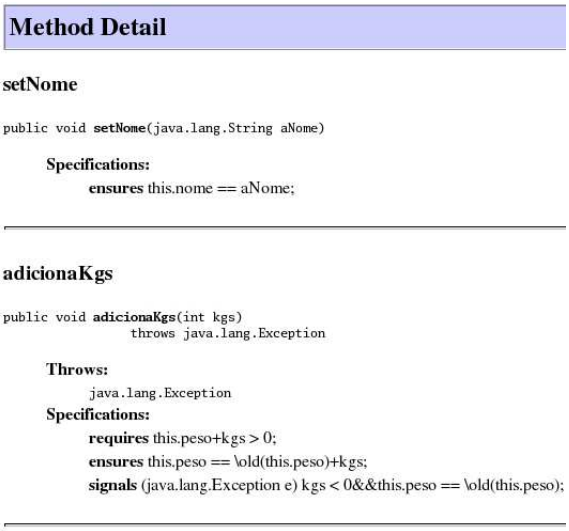
A opção -*Suggest* é utilizada para que em caso de um erro ser encontrado, seja sugerido se possível, o que precisa ser feito para corrigí-lo. Conforme esperado, a mesma falha detectada pelas ferramentas anteriores foi identificada. A ferramenta informa que existe a possibilidade de uma atribuição nula à variável *nome* na linha 19, que corresponde ao corpo do método *setNome()*. Esse fato violaria a cláusula `non_null` de sua declaração. O mais interessante é que para corrigir o problema, Esc/Java2 sugere que o parâmetro do método seja declarado também como `non_null`, o que tem o mesmo efeito de acrescentar a anotação `requires aNome != null` ao contrato do método. Ou seja, nesse caso especificamente *Esc/Java2* foi capaz de propor a solução correta para a falha.

6.5.4 Documentando as Especificações com *JMLDoc*

Resumidamente, a ferramenta *JMLDoc* gera automaticamente páginas HTML de maneira análoga à ferramenta *javadoc*, porém incluindo informações oriundas das anotações JML, conforme mostra a Figura 6.2. Para aqueles já acostumados em navegar na documentação gerada pelo *javadoc*, a inclusão da especificação em uma notação formal apenas contribui para a compreensão e documentação do projeto. O *plugin* para a plataforma Eclipse, citado no final da Seção 6.5.1, é integrado ao *jmldoc* de forma que com o apertar de um botão a documentação HTML de todo o projeto é gerada.

6.6 Estudo de Caso

Com o objetivo de sumarizar o conteúdo abordado neste material, iremos introduzir um estudo de caso envolvendo a especificação, codificação e verificação de uma classe que



Method Detail

setNome

```
public void setNome(java.lang.String aNome)
```

Specifications:

ensures this.nome == aNome;

---

adicionaKgs

```
public void adicionaKgs(int kgs)
            throws java.lang.Exception
```

Throws:

java.lang.Exception

Specifications:

requires this.peso+kgs > 0;

ensures this.peso == \old(this.peso)+kgs;

signals (java.lang.Exception e) kgs < 0&&this.peso == \old(this.peso);

Figura 6.2: Exemplo da saída do *jmldoc*

implementa o comportamento de uma pilha. Tentaremos contudo, dar uma visão prática da aplicação da técnica a partir do seguinte enunciado:

> *Implemente uma pilha de tamanho limitado, utilizando um array como estrutura de dado. Uma instância da classe deve ser capaz de empilhar e desempilhar um objeto não-nulo qualquer (tipo `java.lang.Object`), além de se comparar a uma outra pilha e se clonar. Deve ser possível também instanciar uma pilha com valores pré-determinados. Uma pilha é igual a outra se e somente se ambas apresentem o mesmo comportamento quando submetidas à uma seqüência qualquer de chamadas aos seus métodos. A pilha pode armazenar qualquer tipo de objeto. Porém, caso a pilha não esteja vazia, todos os elementos devem ter o mesmo tipo do primeiro objeto empilhado. Qualquer operação que viole essas regras deve lançar uma exceção.*

Esse estudo de caso produzirá duas versões da classe `Pilha`, ambas serão verificadas com a ferramenta *jmlunit*. No entanto, apesar da primeira conter um erro conceitual (propositalmente inserido), este não será detectado pelo *jmlunit*. Isso essencialmente caracteriza um erro sobretudo na especificação, uma vez que a ferramenta, se usada corretamente, nos dá a confiança necessária de que o código a satisfaz. O objetivo com isso é chamar a atenção de que tão importante quanto a consistência entre especificação e código, é que esta especificação atenda aos requisitos do problema. Na segunda versão a especificação anterior será estendida, o que nos possibilitará detectar tal erro e por fim corrigí-lo. Ainda em tempo, a primeira versão é dividida em duas etapas:

- **Etapa I** - abrange a especificação e implementação dos atributos, construtores e métodos auxiliares
- **Etapa II** - abrange a especificação e implementação da interface pública da classe. Interface esta extraída dos requisitos presentes no enunciado (vide Código 6.27). Nesta etapa, em particular, o erro será introduzido.

Seremos o mais preciso possível na especificação, o que não significa necessariamente que esta seja sempre a melhor maneira de utilizar a técnica. Por exemplo, o

Código 6.27: Interface da classe Pilha

```
public interface PilhaIF {

        public void push(Object o) throws
                PilhaCheiaException,
                ObjetoNuloException,
                TipoIncorretoException;
        public Object pop() throws PilhaVaziaException;
        public Object clone ();
        public boolean equals(Pilha aPilha);

}
```

desenvolvedor pode optar por gerar uma especificação menos completa para as classes periféricas (ganhando com isso agilidade) de um sistema enquanto adota uma postura mais rígida em relação a especificação das partes críticas.

6.6.1 Primeira versão

Etapa I

O produto desta primeira etapa é mostrado logo a seguir no Código 6.28. Para implementar uma pilha usando um *array* utilizaremos uma variável do tipo inteiro que aponta sempre para a próxima posição livre da pilha. Inicialmente seu valor é 0, já que ao ser instanciada a pilha se encontra vazia. Não faz sentido também existir uma pilha de tamanho menor ou igual a 0, o que a torna incapaz de armazenar qualquer elemento. Nossa classe deve conter ainda o próprio *array* no qual os elementos serão armazenados. Somente sobre estes dois atributos, já é possível especificar bastante coisa sobre o comportamento da classe, conforme mostra a Tabela 6.4.

Anotação	Descrição
`initially proximoLivre==0;`	Ao ser instanciado, a próxima posição livre é sempre 0
`invariant 0<=proximoLivre && proximoLivre<items.length;`	A próxima posição livre não deve apontar para além dos limites do *array*
`/*@ spec_public non_null @*/ Object[] items;`	O próprio *array* nunca pode ser nulo
`invariant (\forall int i; 0 <= i && i < proximoLivre; items[i] != null);`	Nenhum dos elementos da pilha pode ser nulo

Tabela 6.4: Especificação sobre os atributos da pilha

Código 6.28: Código produzido na Etapa I

```
public class Pilha{
  private /*@ spec_public @*/ int proximoLivre;
  private /*@ spec_public @*/ Object[] items;

  //@ initially proximoLivre == 0;
  //@ invariant 0 <= proximoLivre && proximoLivre <= items.length;
  //@ invariant items != null;
  /*@ invariant (\forall int i; 0 <= i && i < proximoLivre;
    @                    items[i] != null);
    @*/

  /*@ requires aTam > 0;
    @ ensures  items.length == aTam;
    @*/
  public Pilha(int aTam){
    this.items = new Object[aTam];
    this.proximoLivre = 0;
  }

/*@ requires conteudo!=null && conteudo.length <= aTam;
  @ ensures (\forall int i; 0 <= i && i < conteudo.length;
  @                  items[i] == conteudo[i]);
  @*/
  public Pilha(int aTam, Object[] conteudo){
    this(aTam);
    for(int i = 0; i < conteudo.length; i++){
      items[i] = conteudo[i];
      proximoLivre++;
    }
  }

  //@ ensures \result == (proximoLivre == 0);
  public /*@ pure @*/ boolean isEmpty(){
    return proximoLivre == 0;
  }

  //@ ensures \result == (proximoLivre == items.length);
  public /*@ pure @*/ boolean isFull(){
    return proximoLivre == items.length;
  }
}
```

Feito isto, pensemos agora no construtor da classe, cuja função básica deve ser criar uma pilha com um determinado tamanho (maior que 0). Este tamanho pode ser definido pelo argumento do construtor no momento da sua instanciação. Para satisfazer a primeira e terceira invariante da Tabela 6.4, o corpo do construtor deve inicializar o atributo *proximoLivre* com 0 e criar de fato o array *items*. Esse último fato nos sugere a formulação de uma pós-condição para atender o seguinte comportamento: o *array* criado deve possuir o tamanho definido pelo parâmetro recebido no construtor. Além disso, deve ser possível criar uma pilha já preenchida com determinados objetos. Para isso, criamos um construtor que recebe um *array* de nome *conteudo* como parâmetro (contendo os objetos a serem inseridos na pilha). Obviamente, esse *array* não pode ser nulo, e seu tamanho não pode ser maior que o tamanho da própria pilha (definido pelo outro argumento *aTam*). Após a chamada a esse construtor, a pilha deve estar devidamente inicializada com os elementos deste *array*.

Antes de codificar os métodos que compõem a interface da classe `Pilha`, vamos implementar dois métodos que nos serão úteis no restante da implementação: o método *isFull()* e *isEmpty()*, que têm a função é responder se a pilha está cheia ou vazia em

um determinado momento. Para verificar se a pilha está vazia, basta checar se o atributo *proximoLivre* é igual a 0. De maneira análoga, um método para checar se a pilha está cheia pode simplesmente testar se *proximoLivre* já se igualou ao tamanho do *array*. Como esses dois métodos não possuem nenhum efeito colateral (não modificam o estado do objeto), declaramos cada um deles com o modificador `pure` para que a especificação dos outros métodos possam utilizá-los. Como é possível observar, já temos código suficiente para uma primeira verificação do que foi produzido.Deixaremos para entrar em detalhes sobre a utilização da ferramenta *jmlunit* ao final da Etapa II, por enquanto basta saber que foram gerados 22 testes a partir dos dados de testes escolhidos, sendo que todos eles foram bem-sucedidos. Isto deve dar a segurança necessária de que até agora o código está consistente com a especificação.

Etapa II

Nesta etapa, iremos codificar a interface pública da classe definida na Etapa I. Para chegar à implementação mostrada no Código 6.29, comecemos pelo método *equals(Pilha p)*, responsável por verificar se a pilha passada como parâmetro é igual à própria instância. Perceba que o enunciado não exige que todos os elementos do *array* sejam iguais e sim que, caso apliquemos a mesma seqüência de empilhamentos e desempilhamentos em ambas, o comportamento produzido deve ser o mesmo.

Para atingir este objetivo basta que o tamanho máximo das duas pilhas seja o mesmo, que o atributo *proximoLivre* esteja apontando para a mesma posição do *array* e que todos elementos até esta posição sejam iguais. Porém para exercer essa função, o método impõe como pré-condição que a pilha *p* não seja nula. Agora que já temos disponível o método *equals(Pilha p)* (que também é puro), fica fácil especificar o método *clone()*. Ele deve retornar um pilha que seja igual no comportamento, mas que no entanto seja uma referência diferente da própria instância.

O método *pop()* é o responsável pela função de desempilhar. No caso da pilha estar vazia no momento da sua invocação, uma exceção deve ser lançada e o estado da pilha deve se manter inalterado. Caso contrário, o método deve retornar o topo da pilha. Uma outra observação interessante a ser feita é que para desempilhar um objeto só é preciso alterar o valor do atributo *posicaoLivre* e nada mais (por isso o uso da cláusula `assignable`). Como sempre o topo da pilha será igual a *items[proximoLivre - 1]*, foi criado o método *topo()* para facilitar a especificação. Note que ele é declarado dentro de um bloco de anotação JML com o modificador `model`, isso o torna visível somente para a especificação - não existindo portanto, para a classe em si. Porém, mesmo sendo um método de modelo (veja maiores detalhes na Seção 6.4.1), ainda há a necessidade de utilizar a palavra reservada `pure` para permitir que este seja utilizado, de fato, ao longo da especificação.

A especificação do método *push(Object o)* é um pouco mais extensa, porém é também facilmente formulada a partir dos requisitos expostos no enunciado. Quando invocado, a pilha não pode estar cheia, o objeto *o* não pode ser nulo e caso a pilha não esteja vazia o tipo de *o* deve ser igual ao tipo do primeiro elemento inserido. Caso alguma dessas condições não seja satisfeita uma exceção deve ser lançada e o estado da pilha deve

Código 6.29: Código produzido na Etapa II contendo um erro conceitual

```
//@ public pure model Object topo(){ return items[proximoLivre-1]; }
/*@ ensures topo() == o &&
  @         proximoLivre == \old(proximoLivre) + 1;
  @ signals (PilhaCheiaException e) isFull();
  @ signals (ObjetoNuloException e) o == null;
  @ signals (TipoIncorretoException e) (!isEmpty()) &&
  @              (o.getClass() != items[0].getClass());
  @ signals (Exception e) equals(\old(this));
  @ assignable proximoLivre,items[proximoLivre]; @*/
  public void push(Object o) throws Exception {
    if (isFull()) {
      throw new PilhaCheiaException("Pilha cheia!");
    } else if (o == null) {
      throw new ObjetoNuloException("Objeto nulo!");
    } else if ((!isEmpty())&&(o.getClass() != items[0].getClass())){
      throw new TipoIncorretoException("Tipo incorreto do objeto!");
    } else {
      items[proximoLivre] = o;
      proximoLivre++;
    }
  }

/*@ ensures \result == \old(topo());
  @ signals (PihaVaziaException e) isEmpty();
  @ signals (Exception e) equals(\old(this));
  @ assignable proximoLivre; @*/
  public Object pop() throws PihaVaziaException{
    if(!isEmpty()) return items[proximoLivre - 1];
    else throw new PihaVaziaException("Pilha vazia!");
  }

/*@ requires aPilha != null;
  @ ensures \result==true <==> proximoLivre == aPilha.proximoLivre &&
  @                            items.length == aPilha.items.length &&
  @                          (\forall int i; 0 <= i && i < proximoLivre;
  @                                  items[i].equals(aPilha.items[i])); @*/
  public /*@ pure @*/ boolean equals(Pilha aPilha){
    if( (proximoLivre!=aPilha.proximoLivre) ||
       (items.length != aPilha.items.length) )
      return false;
    for(int i = 0; 0 <= i && i < proximoLivre;i++){
      if(!items[i].equals(aPilha.items[i]))
        return false;
    }
  return true;
  }

/*@ also
  @ ensures equals((Pilha)\result) && (Pilha)\result != this; @*/
  public Object clone () {
    Pilha novaPilha = new Pilha(items.length);
      for (int k = 0; k < proximoLivre; k++) {
        novaPilha.items[k] = items[k];
        novaPilha.proximoLivre++;
      }
    return novaPilha;
  }

/*@ also
  @ ensures \result != null && !\result.equals("") &&
  @         (* Mostre o estado do objeto *); @*/
  public String toString(){
    return "proximoLivre="+proximoLivre + ", empty="+isEmpty() +
      ", full="+isFull()+"\n items="+items;
}
```

ser mantido. Se nenhuma dessas situações for verdadeira, o método deve fazer com que *o* passe a ser o topo da pilha e que o atributo *proximoLivre* seja incrementado em um. Por último, incluímos também o método *toString()*, que por ser herdado de `Object` deve ser especificado com `also`.

Tipo	Dado de Teste
int	`-1, 0, 1`
Object	`"", new Integer(1)`
Object[]	`{},{""},{"",new Integer(1)}`

Pilha	`Pilha pilhaVazia = new Pilha(1);`
	`Pilha pilhaQuaseCheia = new Pilha(2,new Object[]{""});`
	`Pilha pilhaCheia = new Pilha(1,new Object[]{""});`

Tabela 6.5: Dados de teste para a primeira versão

Vamos verificar agora a primeira versão com o *jmlunit*[2]. Para os dados de testes escolhidos (vide Tabela 6.5) foram gerados 55 casos de testes, sendo que nenhum deles falhou. Essa quantidade depende diretamente dos dados de testes selecionados, que em nosso caso, estabelecem os seguintes cenários:

- O construtor *Pilha(int)* será testado para os valores -1, 0 e 1;
- O construtor *Pilha(int,Object[])* será testado para todas as possíveis combinações (dois a dois) entre o conjunto de dados de teste do tipo *int* e do tipo *Object[]*;
- O método *push(Object)* será testado em todas as instâncias definidas para o tipo *Pilha*, para cada um dos parâmetros do conjunto *Object*;
- Os demais métodos serão testados cada um três vezes, pois definimos três possíveis estados da pilha (cheia, vazia e parcialmente cheia).

6.6.2 Segunda versão

Basicamente, a segunda versão da classe `Pilha` fortalece a pós-condição dos métodos *pop()* e *push()* com o objetivo de aumentar nossa confiança quanto a corretude da implementação. Essas novas propriedades foram percebidas a partir da seguinte constatação:

1. Logo após um objeto *o* ser empilhado, a pilha deve se manter inalterada no caso de invocarmos em seqüência o método *pop()* e novamente *push(o)*.
2. Logo após um objeto *o* ser desempilhado, a pilha deve se manter inalterada no caso de invocarmos em seqüência o método *push(o)* e novamente *pop()*.

De tão óbvia, essas propriedades parecem, à primeira vista, serem incapazes de revelar algum falha na nossa implementação. Porém, sua real utilidade é justamente confrontar a implementação do método *pop()* em relação ao método *push()*, e vice-versa.

[2]O funcionamento da ferramenta é mostrado na Seção 6.5.2

Como veremos a seguir, esse é o motivo pelo qual torna-se possível detectar o erro inserido propositalmente na versão anterior. Vamos então, voltar nossa atenção para como expressar tais propriedades. Sabemos, de antemão, que é preciso comparar o estado atual da pilha com uma outra. Esta funcionalidade por sinal, já foi implementada pelo método *equals(Pilha)*. O problema, portanto, se resume na criação de uma pilha idêntica à própria instância (na qual possamos aplicar os métodos para empilhar e desempilhar) e posteriormente testar sua igualdade utilizando o método *equals(Pilha)*. Para isso, criamos dois métodos de modelo que permitem observar o estado da pilha logo após a execução de *pop()* e *push()*, conforme mostrado no Código 6.30.

Código 6.30: Métodos auxiliares de observação para *pop()* e *push()*

```
/*@ public pure model Pilha pushObservavel(Object o) throws Exception{
  @   Pilha pilha = (Pilha) this.clone();
  @   pilha.push(o);
  @   return pilha;
  @ }
  @ public pure model Pilha popObservavel() throws Exception{
  @   Pilha pilhaClonada = (Pilha) this.clone();
  @   pilhaClonada.pop();
  @   return pilhaClonada;
  @ }
  @*/
```

Com esses métodos torna-se fácil expressar as propriedade 1 e 2 definidas anteriormente como sendo:

1. `equals(popObservavel().pushObservavel(o))`
2. `equals(pushObservavel(\old(topo())).popObservavel())`

Após adicionarmos essas propriedades na pós-condição dos métodos *push()* e *pop()* respectivamente, executamos novamente a suíte de testes (com os mesmo dados de testes da Tabela 6.5). Do total de 55 testes, 6 falharam, sendo 2 erros referentes ao método *pop()* e 4 referentes ao método *push()*. Abaixo se encontra a saída de apenas um dos erros identificados:

```
1 pop(Pilha_JML_Test$TestPop)junit.framework.AssertionFailedError:
2       Method 'pop' applied to
3       Receiver: proximoLivre=1, empty=false, full=false,
4                   items=[Ljava.lang.Object;@1754ad2
5 Caused by:
6        org.jmlspecs.jmlrac.runtime.JMLInternalNormalPostconditionError:
7        by method Pilha.pop regarding specifications at
8 File "Pilha.java", line 83, character 27 when
9       '\old(topo())' is
10      '\old(topo())' is
11      '\result' is
12      'this' is proximoLivre=1, empty=false, full=false,
13                  items=[Ljava.lang.Object;@1754ad2
14      at Pilha.pop(Pilha.java:2789)
15      at Pilha_JML_Test$TestPop.doCall(Pilha_JML_Test.java:520)
16      at Pilha_JML_Test$OneTest.runTest(Pilha_JML_Test.java:85)
17      at Pilha_JML_Test$OneTest.run(Pilha_JML_Test.java:75)
18      at org.jmlspecs.jmlunit.JMLTestRunner.doRun(JMLTestRunner.java:253)
19      at org.jmlspecs.jmlunit.JMLTestRunner.run(JMLTestRunner.java:228)
20      at Pilha_JML_Test.main(Pilha_JML_Test.java:22)
```

As linhas 3 e 12 mostram que não há nenhuma alteração do estado do objeto após a invocação do método *pop()*. Esta fato nos parece estranho, pois para as pós-condições adicionadas serem satisfeitas este método deve, de certa forma, ter um comportamento oposto à *push()* - o que é impossível acontecer se este não possui nenhum efeito colateral. Dessa observação, derivamos que o erro se encontra no fato do método *pop()* não decrementar o atributo *proximoLivre*. A não observância deste fato na especificação foi o motivo pelo qual o erro não foi detectado na versão anterior. Bastaria, portanto, ter acrescentado na pós-condição do método *pop()* a propriedade

```
proximoLivre == \old(proximoLivre) - 1.
```

Com isso, os testes executados ao final da Etapa II teriam sido suficientes para detectar tal erro na implementação.

6.7 Conclusão

Há décadas, métodos formais vêm sendo utilizados com relativo sucesso no desenvolvimento de sistemas críticos. De modo geral, seus benefícios são evidenciados pela melhoria da qualidade do software produzido, através principalmente da identificação precoce de erros. Esse fato é uma indicação de que ao menos uma parcela dos mesmos benefícios possam ser obtidos para outras classes de sistemas. Neste material, vimos os fundamentos básicos que envolvem a teoria de *Design by Contract* e como esta pode contribuir para a obtenção de sistemas mais confiáveis. A utilização do método DBC é uma forma suave de introduzir os principais conceitos da disciplina de métodos formais na prática de desenvolvimento.

Neste material, vimos os fundamentos básicos que envolvem DBC e como a linguagem JML pode ser utilizada para concretizá-los. Mostramos também diversas ferramentas de apoio que efetivamente viabilizam a utilização do método atualmente. Por último, um estudo de caso foi apresentado com o objetivo de demonstrar passo-a-passo a aplicação do que foi abordado ao longo do curso. A seguir, colocamos alguns apontadores sobre o assunto e temas relacionados.

6.7.1 Referências para Leitura

A principal fonte de consulta sobre DBC é o livro [Meyer, 1997], o qual aprofunda a teoria inicialmente publicada em [Meyer, 1992]. Um outro apontador importante sobre o assunto é o sítio mantido pela Eiffel Software [Eiffel, 2005], empresa que mantém a linguagem Eiffel, a precurssora no suporte de DBC. Existe ainda um bom livro [Mitchell and McKim, 2002] que aborda o tema através de exemplos. Todo material referente a JML [Burdy et al., 2003, Leavens et al., 1999, Leavens and Cheon, 2003, Leavens et al., 2000, Leavens et al., 2002], pode ser encontrado no endereço eletrônico *http://www.jmlspecs.org*. Vale citar que alguns exemplos utilizados neste curso foram extraídos destas referências.

As ferramentas *JMLRAC* [Bhorkar, 2000], *JMLUnit* [Cheon and Leavens, 2004] e *JMLDoc* podem ser obtidas através deste mesmo endereço, enquanto o *plugin* que integra o JMLRAC e o JMLDoc à plataforma Eclipse se encontra em [JMLEclipse, 2005]. A ferramenta *Esc/Java2* [Flanagan et al., 2002] está também disponível na Internet no sítio

[KindSoftware, 2005]. Caso o leitor tenha interesse em se aprofundar nos conceitos que deram origem à teoria de DBC, é imprescindível ler os artigos de Hoare [Hoare, 1969], Dijkstra [Dijkstra, 1997] e Floyd [Floyd, 1967].

Referências

Bhorkar, A. (2000). A run-time assertion checker for Java using JML. Technical Report 00-08, Department of Computer Science, Iowa State University.

Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G., Leino, K., and Poll, E. (2003). An overview of jml tools and applications.

Cheon, Y. and Leavens, G. T. (2004). The JML and JUnit way of unit testing and its implementation. Technical Report 04-02a, Department of Computer Science, Iowa State University.

Dijkstra, E. W. (1997). *A Discipline of Programming*. Prentice Hall PTR.

Eiffel (2005). Homepage sobre design by contract mantida pela eiffel software. Mai. http://archive.eiffel.com/doc/manuals/technology/contract/.

Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. (2002). Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245. ACM Press.

Floyd, R. W. (1967). Assigning meaning to programs. In Schwartz, J. T., editor, *Mathematical aspects of computer science: Proc. American Mathematics Soc. symposia*, volume 19, pages 19–31, Providence RI. American Mathematical Society.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

JMLEclipse (2005). Homepage do projeto, mantido por santos laboratory. Mai. http://jmleclipse.projects.cis.ksu.edu/.

JMLSpecs (2005). Homepage do projeto. FMai. http://www.jmlspecs.org.

KindSoftware (2005). Homepage do grupo de pesquisa kindsoftware. Mai. http://secure.ucd.ie/products/opensource/ESCJava2/download.html.

Leavens, G. and Cheon, Y. (2003). Design by contract with jml.

Leavens, G. T., Baker, A. L., and Ruby, C. (1999). JML: A notation for detailed design. In Kilov, H., Rumpe, B., and Simmonds, I., editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers.

Leavens, G. T., Baker, A. L., and Ruby, C. (2000). Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i.

Leavens, G. T., Poll, E., Clifton, C., Cheon, Y., and Ruby, C. (2002). Jml reference manual. Department of Computer Science, Iowa State University. Available from http://www.jmlspecs.org.

Meyer, B. (1992). Applying ”design by contract”. *Computer*, 25(10):40–51.

Meyer, B. (1997). *Object-Oriented Software Construction*. Prentice Hall, 2 edition.

Mitchell, R. and McKim, J. (2002). *Design by Contract, by Example*. Addison-Wesley.

Rogério Dourado Silva Júnior

Nascido em Salvador, BA, em novembro de 1979. É Bacharel em Ciência da Computação pela UFBA, BA, 2002 e Mestrando em Informática pela UFCG, PB. As principais áreas de interesse de pesquisa são: Engenharia de Software, Verificação de Software e Processos de Desenvolvimento.

Jorge Cesar Abrantes de Figueiredo

Nascido em Sousa, PB, em abril de 1965. É Engenheiro Eletricista (eletrônica) pela UFPB, PB, 1987, Mestre em Informática e Doutor em Ciência pela UFPB, PB, em 1989 e 1994, respectivamente. Realizou atividades de Pós-Doutorado na Aarhus University, Dinamarca, durante o período de agosto/98 a abril/2000. É Professor Adjunto do Departamento de Sistemas e Computação da Universidade Federal de Campina Grande. Foi coordenador do Curso de Ciência da Computação, DSC/UFCG, PB, em 1997 e 1998 e coordenador do curso de Pós-Graduação em Informática, COPIN/DSC/UFCG, PB, entre 2001 e 2003. Foi pesquisador visitante no Department of Computer Science, University of Pittsburgh, PA, USA, em 1992 e 1993. As principais áreas de interesse de pesquisa são: Métodos Formais, Redes de Petri e Engenharia de Software. Em especial, realiza pesquisa na aplicação de métodos formais no desenvolvimento rigoroso de software. Áreas nas quais publicou mais de 80 artigos a nível nacional e internacional.

Dalton Serey Guerrero

Nascido em Valparaiso, Chile, em agosto de 1971. É Bacharel em Ciências da Computação pela UFPB, Mestre em Informática pela UFPB e Doutor em Engenharia Elétrica pela UFCG. É Professor Adjunto do DSC/UFCG, Coordenador do Laboratório de Redes de Petri e membro do Grupo de Pesquisa em Métodos Formais da UFCG. É Vice-Coordenador do Curso de Ciência da Computação, DSC/UFCG, PB desde 2002. Foi pesquisador visitante no Department of Computer Science, University of Aarhus, Dinamarca, em 1999 e 2000. Suas áreas de interesse são: Engenharia de Software, Validação e Verificação Matemática de Software e Orientação a Objetos, áreas em que é autor de diversos trabalhos e artigos publicados em congressos e periódicos nacionais e internacionais.