 

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

INSTITUTO METRÓPOLE DIGITAL

LÓGICA APLICADA A ENGENHARIA DE SOFTWARE

Thaynan Paulo Fernandes Bezerra de Mendonça

Alison Jhonas de Sousa Silva

Natal, 2016

Thaynan Paulo Fernandes Bezerra de Mendonça

Alison Jhonas de Sousa Silva

**Relatório de Lógica Aplicada a Engenharia de Software: Aplicação dos conceitos JML a projeto Java**

Relatório referente às atividades passadas em sala de aula sobre a disciplina Lógica Aplicada a Engenharia de Software.

Professor: Marcel Vinícius Medeiros Oliveira.

Natal, 2016

MENDONÇA, Thaynan Paulo Fernandes Bezerra de

SILVA, Alison Jhonas de Sousa

Título: Relatório de Lógica Aplicada a Engenharia de Software: Aplicação dos conceitos JML a projeto Java.

Relatório apresentado à Universidade Federal do Rio Grande do Norte com o objetivo de obtenção de conceito para terceira unidade.

**SUMÁRIO**

**1 EQUIPE DE DESENVOLVIMENTO.......................................................................05**

**2 INTRODUÇÃO...........................................................................................................07**

**3 APRESENTAÇÃO DO PROJETO..........................................................................07**

**4 IMPLEMENTAÇÃO..................................................................................................20**

4.1 DESCRICAO DA SOLUÇÃO...............................................................09

**5 AVALIAÇÃO DOS METODOS................**

5.1 INSERÇÃO

5.2 REMOÇÃO

5.3 BUSCA

5.4 ENESIMO

5.5 POSICAO

5.6 MEDIANA

5.7 CHEIA

5.8 COMPLETA

5.9 IMPRIMA

**6 AVALIAÇÃO DO PROGRAMA**

**7 CONCLUSÃO..................................................................................................20**

**8 BIBLIOGRAFIA...............................................................................................21**

**1. EQUIPE DE DESENVOLVIMENTO**

A equipe de desenvolvimento é composta pelos discentes:

Thaynan Paulo Fernandes Bezerra de Mendonça

Alison Jhonas de Sousa Silva

**2 INTRODUÇÃO**

Este relatório foi elaborado com referência nas aulas apresentadas pelo professor Marcel Vinícius Medeiros Oliveira em sala de aula na matéria Lógica Aplicada a Engenharia de Software, visando a obtenção de conceito para a terceira unidade.

O projeto visa à implementação em um projeto da linguagem de programação Java dos conceitos da Design By Contract (DBC) através do uso da linguagem formal de especificação comportamental JML (Java Modeling Language) que contém as notações de DBC.

Nesse projeto serão anotados nas classes, nos contrutores e nos métodos as condições necessárias, em JML, para o estabelecimento dos conceitos do DBC, bem como analisar as diversas vantagens do seu uso.

**3 CONTEXTUALIZAÇÃO**

Nesse tópico informaremos um pouco sobre o método Design By Contract e JML.

3.1DESIGN BY CONTRACT

O paradigma da Orientação a Objetos garantia um avanço importante na qualidade de software. Entretanto, Sómente esse paradigma não é suficiente para garantir a inexistência de erros. Para isso há a necessidade de se criar novos métodos, nas quais possa se observar a corretude (garantir que o sistema execute o que está na sua especificação) e robustez (como sistema deve reagir para situações inesperadas). Um desses métodos é o Design By Contract (DBC).

O DBC tem como base a idéia de gerar contrados entre o módulo (classe ou interface) e os seus clientes de tal forma que se garante determinadas pré-condições e pós-condições. Esses contratos são definidos através de anotações no próprio código.

Além disso, cada membro do contrato espera um benefício ao mesmo tempo que cumpre algumas obrigações. Geralmente, os benefícios de um consistem na obrigação do outro sendo expressos no contrato. É importante enfatizar que o cliente apresenta a especificação do que deve ser realizado, e o contrato observa no contrato as condições para considerar o serviço concluido, não sendo obrigado a cumprir nada além do que está no seu escopo.

Todas essas condições são estabelecidas no contrato, de tal forma que esse permitiria definir sem ambiguidades o comportamento esperado para esse método, atraves da responsabilização do que os módulos e clientes devem cumprir.

O cliente deve garantir determindas condições, chamadas de pré-condições, ao chamar algum método, e esse último deve, ao finalizar, garantir algumas propriedades. Finalmente, devem existir propriedades que devem ser mantidas ao longo de toda a classe, restrições, denomindas invariantes de classe e devem ser mantidos por todos os métodos. O invariante da classe possui as seguintes características: os construtores devem estabelecer a invariante e todo método não estática deve presevar o invariante após a sua execução, se a mesma estava mantida antes de sua execução. Todos esses mecanismos são chamados de asserções e representam o estado da classe.

Além disso, o DBC provoca algumas implicações para conceitos de OO, como por exemplo a herança e controle de exceções. A herança garante que a subclasse tenha que garantir as asserções estabelecidas na superclasse, fazendo com que a subclasse seja no mínimo tão bom quanto a super, e impossibilitando que um procedimento se transforme em outro diferente por causa da herança. Entretanto, desde que compatível com a especificação original da superclasse, a subclasse tem o direito de melhorá-la. Finalmente, a invariante deve ser mantido nas subclasses.

Para o caso do controle de exceções, é possível definir melhor a condição de falha de tal forma que estabelece-se quais as pós-condições (chamadas pós-condições excepcionais) para o caso de exceção. É importante que caso alguma pré-condição seja quebrada ou caso nao se proceda com exceção, pois foi quem invocou a condição que se equivocou. Também deve-se evitar usar de exceção quando se contempla todas as pós-condições normais. Os uso de exceção equivale a duas portas nos quais é necessário determinadas condições para que se possa utilizar cada uma delas. Para o caso de invariantes, deve-se garantir nos dois tipos de pós-condição.

Além disso, existem várias vantagéns no uso de DBC: melhor controle da corretude e robustez do código, documentação, possibilidade do próprio código encontrar violações a algum contrato estabelecido, uso de ferramentas para encontrar essas violações, facilidade de depuração do código (basta observar se o erro foi na pré-condição, ou seja, no código que invocou o método ou na pós-condição, na qual o método que nao conseguiu atender a alguma propriedade exigida), melhor acomplamento e coesão, maior facilidade no reuso de componentes entre outros.

Finalmente, a análise dessas violações pode ser em tempo de execução, na qual para determinadas entradas verifica-se se alguma asserção foi descumprida, ou geração automática de testes por um suporte ferramental ou, ainda, de forma estática, na qual o sistema pecorre todo os caminhos de execução possiveis de variáveis tentando encontrar todas as violações possíveis.

3.2 JAVA MODELING LANGUAGE

O Java Modeling Language (JML) é uma linguagem formal de especificação comportamental para a linguagem Java através da anotação no próprio código fonte e que possui as notações necessárias para a aplicação da técnica DBC. Para tanto, se mantém próxima da linguagem Java, mas acresce notações inexistentes como no caso dos quantificadoes, model, fields, pré-condições, pós-condições, invariantes, visibilidade de escopo para asserções, herança de contrato, comportamento normal versus comportamento excepcional.

Deve ser anotadas antes dos métodos, contrutores e no início da classe com o acréscimo do arroba (//@ para uma linha e /\*@ e @\*/ para múltiplas linhas), visando a diferenciação dos comentários existentes na própria linguagem Java.

As asserções são ignorados pelo compilador Java, mas podem ser usadas pelas ferramentas que dão suporte à JML.

Nesse texto comentaremos algumas dessas notações dessa linguagem que serão implementadas no projeto.

3.2.1 Cláusula Requires

Equivale as pré-condições. Devem ser satisfeitas quando um método é chamado e devem ser analisadas sempre que um método é chamado. Usa-se a palavra requires seguida de um leva de operadores lógicos, tais como predicados.

Se não é especificado é considerado true, ou seja, a pré-condição sempre será aceita.

3.2.2 Cláusula Ensures

Equivale as pós-condições. Devem ser satiafeitas após a execução de algum método. Nese caso é direcionadas as pós-condições normais, nas quais não se chama nenhuma classe de exceção. Usa-se a palavra requires seguida de um leva de operadores lógicos, tais como predicados.

Permite o uso de todoas as expressões do requires mais o uso do /result (retorna o valor após a operação, não possível em métodos void) e /old(e) (na qual se refere ao valor antes da operação).

Se não é especificado é considerado true, ou seja, a pós-condição sempre será aceita.

3.2.3 Cláusula Signal

Equivale as pós-condições excepcionais, sendo utilizado um signal pra cada exceção lançada. Após o signal, informa-se quais são as pós-condições especifícas para essa situação. Pode-se usar a classe Exception e instanciar (instanceof) qual a exceção está sendo necessária.

Deve levar em consideração: se p é um predicado do requires, !p não deve fazer parte da exceção (pois o método não deveria tratar esses valores) e se p é um predicado ensures, p não deve ser tratado numa exceção (pois já está sendo estabelecida pela via normal do contrato).

3.2.4 Clausula Invariant

Deve garantir que determinado predicado não assuma valor falso antes ou após algum método.

Pode ser precedidia pelos modificadores static e ou instance. O primeiro caso indica que não pode usar a palavra reservada this, sendo os casos de invariantes definidas dentro de interfaces, sendo as de uma classe, por default, um invariante de instância.

Com o invariante temos o seguinte: o construtor é obrigado a estabelecer o invariante e cada método deve preservar o invariante (antes e depois do método).

3.3.5 Cláusula Also

Usado para os caso de herança contratual, deve ser utilizada para acrescer alguns predicados específicos da subclasse para alguma especificação de algum método com contrato na superclasse, garantindo assim que essa classe esteja submetida as mesmas condições da classe ascendente. A ausência do also, indica a manutenção de exatamente as mesmas cláusulas da classe superior.

Primeiro a especificação da subclasse é verificada, se o resultado é verdadeiro, depois a da superclasse.

3.3.6 Descrições Informais

São comentários, que não são processados pelo compilador, mas ajudam a deixar registrado informações importantes a respeito do comportamtento de algum método da classe.

3.3.7 Extenções de expressão Java

O JML incorporou algumas expressões para conceder maior expressividade, como por exemplo implicação (==>), se somente se (<==>) e negação de se somente se (<=!=>), podendo ser utilizada para maior interpretação dos dados.

3.3.8 Spec\_public

Representa que esta variável é pública no escopo da especificação, permitindo o seu uso no JML. Caso uma das váriaveis seja pública, não há a necessidade dessa especificação.

3.3.9 Referências Não-nulas

É uma forma abreviada de dizer que uma referência não pode ser nulo em qualquer estado visível:

private /\*@ spec\_public non\_null @\*/

3.3.10 Métodos Puros

Usadas para métodos que não tem efeitos colaterais, como por exemplo, a existência de atribuições (=, +=, -= e etc) e incrementos (++) e decrementos (--). Além disso, são permitidas chamadas a esses métodos nas especificações do JML, pois possibilitam saber o estado de um objeto sem modificá-lo.

public /\*@ pure @\*/ int ganhar()

3.3.11 Cláusula Assignable

Indica quais atributos podem ter seu valor ou referência modificados. Variáveis locais ao método ou que são criadas ao longo de sua criação não são sujeitas a essa restrição. O modificador pure é nada mais do que especificar um método como Assignable \nothing.

Por padrão, se nada é especificado, assignable assume o valor /everything que indica que todas as variáveis podem ser modificadas.

3.3.12 Constraints

Semelhantemente ao invariante, os constraints também indicam a forma que o predicado deve satisfazer todos os métodos, entretanto, o constraints leva em consideração a combinação de estados seguidos, enquanto o invariante tem caráter estático para toda a classe. Acaba sendo um mecanismo que impõe uma pós-condição a um conjunto de métodos, restrigindo a forma que os atributos podem mudar durante a execução desses métodos.

Por seu carater dinâmico, possibilita o uso da cláusula old, na qual faz a interação entre dados antes e posterior a execução do método e podem ser definidas para cada método ao contrário do invariante que é global.

É importante resaltar, finalmente, que constraint não se aplica a construtores, pois os mesmo não possuem estado anterior a sua chamada

3.3.13 Cláusula Initially

Indica a condição de uma variável após a sua especificação, sendo implementada no JML atrávés do acréscimo de tal propriedade como pós condição de todos os contrutores.

3.3.14 Model e Cláusula Presents

Anotação presente apenas na especificação, não sendo visível para o programa em si e pode ser utilizado para ajudar a especficar algumas propriedades que não é visível além do contexto da especificação.

Também pode ser usada para atributos de modelos, o que permite criar abstraçãoes sobre atributos do concretos dos quais ele abstrai através da cláusula Represents. Apesar de ser usado apenas para especificação, usa o mesmo escopo do java, o que implica na impossibilidade de declarar o mesmo nome.

3.3.15 Quantificadores

O JML permite o uso de quantificadores como: quantificador universal (\forall), existêncial (\exist), de generalização (\sum, \product, \max, \min) e quantificador numérico (\num\_of). Esse último indica a quantidade de vezes que determinado atributo dentro da faixa definida e para que o predicado seja verdadeiro.

3.4 TIPO DE ESPECIFICAÇÃO

A especificação pode ser leve (lightweight) ou pesada (heavyweight). Na especificação leve, exigie-se as pré-condições, mas não as pós-condições, considerando-as nesse caso verdadeiras.

Para as pesadas temos os 3 tipos de comportamentos:behavior, normal\_behaviour e expectional\_behaviour.

**4. IMPLEMENTAÇÃO**

**5. ANÁLISE DOS RESULTADOS**

**6 AVALIAÇÃO DO PROGRAMA**

**7 CONCLUSÃO**

Após o término do exercício pudemos aprimorar o conhecimento sobre a implementação e uso das ABBs.

A implementação dos exemplos mostrou-se um interessante desafio quanto à manutenção e acesso aos diversos nós que compõem a árvore.

Através destes exemplos, fica também evidente o quão dependente da busca é a árvore binária de busca, e o quão vantajoso é o armazenamento do tamanho das sub-árvores à esquerda e à direita, quando utilizando-se da busca em operações além das básicas (busca, inserção e remoção).

Finalmente, a análise da complexidade dos métodos consistiu numa importante prática para entendimento desse conteúdo.

**8 BIBLIOGRAFIA**

CORMEN, T. H. et.al. Algoritmos: teoria e prática, 3a edição. Ed. Campus, 2002.

SZWARCFITER , J. L., MARKENZON, L. Estruturas de dados e seus algoritmos. Editora LCT. 3a. Edição, 2010.

<http://www.univasf.edu.br/~marcelo.linder/arquivos_ed1/aulas/aula21.pdf>

<http://professor.ufabc.edu.br/~leticia.bueno/classes/aa/materiais/resumoAA.pdf>