O Programador Pragmático:

De Aprendiz a Mestre

•••

Capítulo 4: Paranóia Pragmática

Este material foi desenvolvido pelo grupo PET/ADS do IFSP São Carlos

Introdução

“Quando todos estão realmente atrás de você, a paranoia passa a ser uma mentalidade saudável.”

Woody Allen

Programação defensiva

Abordagem de desenvolvimento de software que se concentra na prevenção de erros e na mitigação de problemas em vez de simplesmente detectá-los e corrigi-los posteriormente.

Programação assertiva

Abordagem de desenvolvimento que se concentra na inclusão de verificações no código-fonte para garantir que as condições esperadas sejam atendidas durante a execução de um programa.

Dicas

DICA 30: Aceite que não é possível criar software perfeito.

Design by Contract (DBC)

Os módulos de software são usados para assegurar a precisão do programa. Eles são divididos em:

- **Pré-condições:** O que deve ser verdadeiro para a rotina ser chamada; os requisitos da rotina;
- **Pós-condições:** O que é certo que a rotina fará; o estado das coisas quando a rotina for concluída;
- **Invariantes de classe:** condição que deve sempre verdadeira do ponto de vista de um chamador;

Contrato de uma rotina no iContract para Java

Exemplo de contrato de inserção de um nó numa lista ordenada:

```
/**
 * @invariant forall Node n in elements() |
 * n.prev() != null
 * impliesao
 * n.value().compareTo(n.prev().value()) > 0
 */
public class dbc_list {
    /**
     * @pre contains(aNode) == false
     * @post contains(aNode) == true
     */
    public void insertNode(final Node aNode) {
        // ...
    }
}
```

O DBC e os parâmetros constantes

Pós-condições podem usar os parâmetros do método para verificar se um comportamento está correto, mas se houver permissão para alterá-los é possível burlar o contrato.

A palavra-chave **final** em Java pode ser utilizada para indicar que um parâmetro não deve ser alterado dentro do método. Entretanto, isso não impede que uma subclasse redeclare o parâmetro.

A sintaxe ***variável@pre*** da biblioteca iContract pode ser utilizada para garantir o valor original da variável no método.

Contrato e responsabilidades

O chamador deve garantir que todas as pré-condições da rotina sejam atendidas.

A rotina deve garantir que todas as pós-condições e invariantes sejam verdadeiras quando for concluída.

O não cumprimento do contrato leva a uma **exceção** ou ao encerramento do programa, o que **não** é considerado um **bug**.

Aceitar qualquer coisa em um contrato levará a um código extenso.

Ao fazer um contrato, deve-se dar preferência a um "código preguiçoso", rigoroso quanto ao que vai aceitar antes de começar e que prometa o mínimo possível em retorno.

Dicas

DICA 31: Projete com contratos.

Herança e Polimorfismo

“As subclasses devem poder ser usadas por meio da interface da classe base sem o usuário precisar saber a diferença.”

Princípio de Substituição de Liskov

Com o uso da herança, é possível especificar o contrato na classe base, para que seja aplicado automaticamente nas subclasses.

Sem um contrato, o compilador garante que uma subclasse siga uma assinatura de método específica.

Definindo um contrato na classe base, podemos garantir que as subclasses não alterem os significados dos métodos.

Implementando o DBC: Asserções

Asserções são **declarações** ou **expressões** colocadas em um programa para verificar se determinadas condições ou suposições são verdadeiras em um ponto do código.

Elas são usadas para fazer o compilador verificar o código do contrato automáticamente.

No entanto elas não podem substituir o DBC:

- Asserções não são propagadas em hierarquias de herança;
- Como elas não têm um conceito de valores "antigos", é preciso adicionar código à pré-condição para manter as informações da entrada do método;
- Bibliotecas e sistemas de tempo de execução não são projetados para dar suporte a contratos, resultando em chamadas de asserção não verificadas.

Suporte de Linguagem

Em linguagens que suportam DBC, caso ocorra um erro ou a pós-condição não seja atendida, uma mensagem de erro e um rastreamento de pilha são exibidos.

No entanto, em linguagens como Java, C e C++, que não tem suporte nativo, o programa pode apresentar comportamentos inesperados ao continuar executando sem cumprir o contrato.

Nesse caso, mesmo que as bibliotecas não sejam verificadas automaticamente, os contratos dão um ponto de partida para fazer as verificações necessárias.

Deve-se então inserir comentários especiais no código-fonte original e usar pré-processadores.

Exemplos de pré-processadores:

- Nana para C e C++,
- iContract para Java.

Outros usos de invariantes: invariantes semânticas

Invariantes semânticas são usadas para expressar requisitos invioláveis, atuando como um tipo de "contrato filosófico" no desenvolvimento de software.

Elas devem ser claras e concisas, para garantir que sejam bem compreendidas e aceitas por todos os envolvidos no desenvolvimento do sistema.

São normas fixas invioláveis, não sujeitas a mudanças políticas, e devem refletir o verdadeiro significado de algo no sistema.

Exemplo: Sistema de transações de cartão de débito, no qual o requisito fundamental é que o mesmo débito nunca deve ser aplicado à conta duas vezes, independentemente das falhas.

Dicas

DICA 32: Encerre antecipadamente.

Programas mortos não contam mentiras: encerre, não falhe!

Detectar problemas o mais cedo possível permite encerrar o programa antes que resultados inesperados ocorram.

Encerrar o programa é muitas vezes a melhor ação a ser tomada quando algo inesperado acontece para evitar agravar a situação.

Em Java, exceções não capturadas levam ao encerramento do programa, exibindo um rastreamento de pilha.

Em outras linguagens, deve-se criar mecanismos para encerrar o programa quando ocorrem erros inesperados.

Deixe as asserções ativadas

É comum inferir que as asserções adicionam sobrecarga ao código e, por esse motivo, devem ser retiradas antes de enviar um programa para produção.

Entretanto, não será possível testar todas as permutações pelas quais seu código passará.

Além disso, quando o programa for executado em um ambiente de produção, será impossível prever todos os fatores aos quais o programa estará sujeito.

Caso realmente haja problemas de desempenho, é possível tornar as asserções que o afetam opcionais.

Asserções e seus efeitos colaterais

O trecho de código a seguir ilustra um efeito colateral adicionado por uma asserção, também chamado de Heisenbug:

```
while (iter.hasMoreElements()) {
    Test.ASSERT(iter.nextElement() !=null);
    //a asserção itera para o próximo elemento antes de usarmos o objeto
    Object obj = iter.nextElement();
    // ....
}
```

VS

```
while (iter.hasMoreElements()) {
    Object obj = iter.nextElement();
    Test.ASSERT(obj != null);
    // ....
}
```

Dicas

DICA 33: Se não pode acontecer, use asserções para assegurar que não aconteça.

Quando usar exceções

Lidar com muitos erros pode tornar o código desorganizado e difícil de manter. Veja um exemplo:

```
retcode = OK;
if (socket.read(name) != OK) retcode =BAD_READ;
else {processName(name)
    if (socket.read(address) != OK) retcode = BAD_READ;
    else {
        processAddress(address);
        if (socket.read(telNo) != OK)
        // etc, etc…
    }
}
return retcode;
```

Quando usar exceções

Exceções permitem escrever um código mais claro e legível, onde a lógica principal do programa não é obscurecida pela manipulação de erros.

```
retcode = OK;
try {
    socket.read(name);
    process(name);
    // etc, etc...
}
catch (IOException e) {
    retcode = BAD_READ;
    Logger.log("Error reading individual: "+ e.getMessage());
}
return retcode;
```

O que é Excepcional?

Se algo deveria estar presente no programa, como um arquivo obrigatório, e não está, uma exceção é justificada, pois algo inesperado ocorreu.

Por outro lado, se a presença de algo não é uma garantia, como um arquivo especificado pelo usuário, é mais apropriado usar retornos de erro convencionais em vez de exceções.

O uso excessivo de exceções para controlar o fluxo normal do programa pode prejudicar a legibilidade e a manutenção do código, além de romper o encapsulamento.

Programas que usam exceções como parte do processamento normal podem criar emaranhados de código complicados e acoplamento excessivo entre rotinas e chamadores.

Dicas

DICA 34: Use exceções para problemas excepcionais.

Os manipuladores de erros são uma alternativa

Um manipulador de erros é uma **rotina que é chamada quando um erro específico é detectado**.

Em linguagens que não possuem suporte a exceções, como C, os manipuladores de erros são uma das poucas opções disponíveis para lidar com erros.

Em linguagens que possuem um sistema de exceções, como Java, os manipuladores de erros podem ser úteis quando a manipulação de exceções se tornar tediosa ou difícil de implementar.

Como Balancear Recursos

O código a seguir não tem os recursos balanceados e, por isso, pode apresentar problemas de recursos não liberados, além de não ser claro.

```
void readCustomer(const char *fName, Customer *cRec) {
    cFile = fopen(fName, "r+");
    fread(cRec, sizeof(*cRec), 1, cFile);
}

void writeCustomer(Customer *cRec) {
    rewind(cFile);
    fwrite(cRec, sizeof(*cRec), 1, cFile);
    fclose(cFile);
}
void updateCustomer(const char *fName, double newBalance) {
    Customer cRec;
    readCustomer(fName, &cRec);
    cRec.balance = newBalance;
    writeCustomer(&cRec);
}
```

Como Balancear Recursos

Ao lidar com recursos, a rotinas ou objetos que os alocam devem ser responsáveis por desalocá-lo.

```
void readCustomer(FILE *cFile, Customer *cRec) {
    fread(cRec, sizeof(*cRec), 1, cFile);
}
void writeCustomer(FILE *cFile, Customer *cRec) {
    rewind(cFile);
    fwrite(cRec, sizeof(*cRec), 1, cFile);
}
void updateCustomer(const char *fName, double newBalance) {
    FILE *cFile;
    Customer cRec;
    cFile = fopen(fName, "r+");       // --->
    readCustomer(cFile, &cRec);       //    /
    if (newBalance >= 0.0) {          //    /
        cRec.balance = newBalance;    //    /
        writeCustomer(cFile, &cRec); //    /
    }                                 //    /
    fclose(cFile);                    // <---
}
```

Dicas

DICA 35: Acabe o que começou.

Aninhe alocações

O aninhamento de alocações é o ato de alocar recursos dentro de recursos já alocados. Evitando deixar recursos órfãos e reduzindo a possibilidade de deadlocks

Diretrizes:

- Desalocação na Ordem Reversa da alocação.
- Ordem de alocação consistente: alocar os recursos sempre na mesma ordem, em diferentes partes do código.

Objetos e Exceções

Em C++, as exceções podem interferir na desalocação de recursos.

Para que haja equilíbrio entre alocação e desalocação deve-se fazer um procedimento semelhante ao construtor no destrutor de uma classe.

Quando um recurso for necessário, instancie um objeto em uma classe com recursos encapsulados, assim o destrutor desalocará o recurso quando o objeto sair do escopo.

A forma de gerenciar exceções varia de acordo com a linguagem. Mas, em todas, o desafio é manter a consistência no tratamento de exceções para evitar vazamentos de recursos.

Balanceando Recursos em Java

Em Java, a coleta de lixo desaloca objetos não referenciados, facilitando o gerenciamento de recursos.

O bloco **finally** é usado para garantir que recursos externos sejam liberados corretamente, mesmo se ocorrer uma exceção.

```
public void doSomething() throws IOException {
    File tmpFile = new File(tmpFileName);
    FileWriter tmp = new FileWriter(tmpFile);
    try {
        // executa alguma tarefa
    }
    finally {
        tmpFile.close();
    }
}
```

Balanceando Recursos em Java

Desde Java 7 foi introduzido o try-with-resources, que segue as diretrizes de aninhamento de exceções mesmo sem o uso explícito de finally.

```
public void doSomething() throws IOException {
    File tmpFile = new File(tmpFileName);
    try (FileWriter tmp = new FileWriter(tmpFile)){
        // executa alguma tarefa
    }
    catch (IOException e){
        e.printStackTrace();
    }
}
```

Quando não é possível balancear os recursos

Em programas que usam estruturas de dados dinâmicas, a alocação de recursos pode ser complicada, uma vez que as estruturas agregadas podem durar mais tempo.

Nesses casos, estabelecer uma invariante semântica para alocação de memória é essencial. Deve-se, também, decidir quem é responsável pelos dados em estruturas agregadas.

Há três opções principais para estruturas agregadas:

- A estrutura de nível superior é responsável por liberar qualquer subestrutura contida;
- A estrutura de nível superior é desalocada, tornando órfãs as subestruturas não referenciadas;
- A estrutura de nível superior se recusa a ser desalocada se houver subestruturas.

A escolha depende das circunstâncias, mas deve ser explícita e consistente para cada estrutura de dados.

Sobre os autores

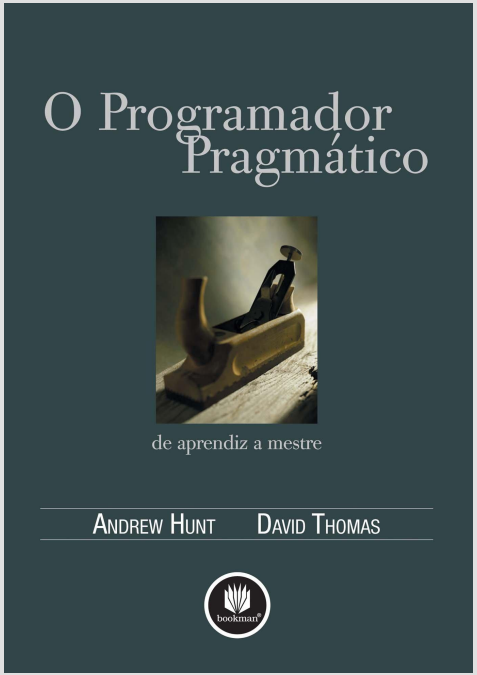
Andrew Hunt trabalhou em diversas áreas, como telecomunicações, serviços financeiros, artes gráficas etc. Hunt se especializou em combinar técnicas já consolidadas com tecnologias de ponta, criando soluções novas e práticas. Ele administra sua empresa de consultoria em Raleigh, Carolina do Norte.

Em 1994 , David Thomas, fundou na Inglaterra uma empresa de criação de software certificada pela ISO9001 que distribuiu mundialmente projetos sofisticados e personalizados. Hoje, Thomas é consultor independente e vive em Dallas, Texas.

Atualmente, David e Andrew trabalham juntos em The Pragmatic Programmers, L.L.C

Este conjunto de slides foi elaborado a partir da obra:

HUNT, Andrew ; THOMAS, David. **O Programador Pragmático:** de aprendiz a mestre.Bookman, 2010.



A equipe

Bárbara Branco, Autora

Aluna e bolsista do PET/ADS desde maio de 2023
LinkedIn

Lucas Oliveira, Revisor

Professor de Computação, é tutor do PET/ADS desde janeiro de 2023.
LinkedIn