O Programador Pragmático:

De Aprendiz a Mestre

•••

Capítulo 6: Enquanto Você Está Codificando

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

Introdução

Quando um projeto está na fase de codificação, é comum pensarmos que o trabalho é todo mecânico, transcrevendo o projeto para instruções executáveis.

A codificação não é mecânica, há decisões a serem tomadas a cada minuto. Estas requerem ponderação e julgamento cuidadoso caso o objetivo seja um programa com vida longa e produtiva.

Desenvolvedores que não pensam em seu código estão programando a base do acaso, o código pode funcionar, mas não há uma razão específica para isso.

Um programador pragmático deve considerar criticamente todos os códigos, incluindo os próprios. Manter-se alerta pode evitar um desastre.

Programação baseada no acaso

"Imagine um soldado em uma guerra. Ele chega a uma clareira: haveria uma mina terrestre ou é seguro? Não há indicações de que seja um campo minado, o soldado examina cuidadosamente o terreno com sua baioneta e recua, esperando uma explosão. Nada acontece. Após avançar cuidadosamente por algum tempo ele se convence de que é seguro e começa a andar orgulhosamente, só para acabar sendo feito em pedaços."

A procura inicial do soldado não revelou nada, mas isto foi apenas sorte. Ele foi levado por conclusões falsas com resultados catastróficos.

A programação é como um campo minado. Deve-se estar sempre atento para não chegar em conclusões falsas. Evite a **programação por coincidência** e favoreça a **programação deliberada** .

Como programar com base no acaso

A programação ao acaso ocorre quando um código é feito, testado e funciona. Então o programador já segue para o próximo passo sem buscar entender o porque o código funcionou.

Ao confiar nesta forma de programar, quando um erro ocorrer será extremamente difícil encontrá-lo. Visto que o programador não entende sequer o que está funcionando.

Acidentes de implementação: Parte 1

Acidentes de implementação correm simplesmente porque é desta maneira que o código está escrito atualmente. Supondo uma rotina com dados inválidos:

```
paint(g);
invalidate();
validate();
revalidate();
repaint();
paintImmediately(r);
```

Esta responde de uma maneira diferente da imaginada pelo autor. Quando esta rotina for "corrigida" seu código pode travar.

Estas rotinas não foram planejadas para serem usadas assim. Embora pareçam funcionar é apenas uma coincidência que pode levar ao pensamento "agora está funcionando, melhor deixar assim".

Acidentes de implementação: Parte 2

É fácil pensar “agora está funcionando, melhor deixar assim”. Por que mudar algo que está funcionando? Podemos pensar em várias razões:

- Pode não estar funcionando realmente – pode apenas parecer que está funcionando.
- A condição limítrofe em que você está se baseando pode ser apenas acidental. Em outras circunstâncias, ela pode se comportar diferentemente.
- O comportamento não documentado pode mudar na próxima versão da biblioteca.
- Chamadas adicionais e desnecessárias tornam o código mais lento.
- Chamadas adicionais também aumentam o risco de introdução de novos erros causados por elas próprias.

Acidentes de contexto e Suposições Implícitas

“Acidentes de contexto” surgem quando se confia em pressupostos não garantidos. Por exemplo: ao programar para uma GUI, todos os usuários falam portugues? Todos são alfabetizados?

Suposições implícitas ocorrem quando, por exemplo, supomos que X é resultado de Y, mas sem provas. Tal resultado pode ser apenas uma coincidência. Assim, é importante provar, não supor.

Dicas

DICA 44: Não programe por coincidência

Como programar deliberadamente

É importante passar menos tempo codificando indiscriminadamente, detectar e corrigir erros o mais cedo possível no ciclo de desenvolvimento e, acima de tudo, gerar menos erros.

Programar deliberadamente envolve:

- Estar sempre consciente do que está fazendo.
- Não codificar às cegas.
- Agir de acordo com um plano.
- Confiar apenas em coisas confiáveis.
- Documentar suas suposições.

Como programar deliberadamente: Continuação

Ao programar deliberadamente é sempre bom ter em mente:

- Não testar apenas o código, testar também as suposições. Não suponha. Teste realmente.
- Priorizar o esforço. Gaste tempo nos aspectos importantes: quase sempre, essas são as partes difíceis.
- Não ser escravo da história. Não deixe códigos existentes ditarem regras para códigos futuros. Todos os códigos podem ser substituídos se não forem mais apropriados.

Da próxima vez que algo parecer estar funcionando e você não souber porque, verifique se não é apenas uma coincidência.

Velocidade do algoritmo

Há um tipo de estimativa que os programadores pragmáticos usam quase todo dia: estimar os recursos que os algoritmos usam – tempo, processador, memória e assim por diante.

Esse tipo de estimativa com frequência é crucial. Saber como um programa se adapta a diferentes quantidades de registros, assim como identificar quais partes do código precisam de otimização.

Sabe-se que essas questões podem ser respondidas com o uso do bom senso, alguma análise e uma maneira de escrever aproximações chamada notação do “Big-O”.

O que queremos dizer com estimativa de algoritmos?

A maioria dos algoritmos incomuns manipula algum tipo de entrada variável. E seu tamanho afeta o algoritmo: quanto maior a entrada, maior o tempo de execução ou maior o uso de memória.

Algoritmos variam na relação com o tamanho da entrada. Alguns são lineares, outros, exponenciais. Um que processa 10 itens em um minuto, pode levar uma vida para 100.

Às vezes, os desenvolvedores se veem realizando uma análise detalhada em relação aos requisitos de tempo de execução e memória. É nesse momento que a notação O() se torna útil.

A notação O( )

A notação O() pode ser interpretada como *da ordem de*. Por exemplo: o tempo no pior caso varia conforme o quadrado de n para uma rotina de classificação com tempo $O(n^2)$.

Ao dizer que uma função leva o tempo $O(n^2)$, significa que o limite superior do tempo gasto por ela não aumentará mais rápido do que n^2.

O uso da notação O() não é limitado apenas ao tempo, ela pode ser usada para representar qualquer outro recurso de um algoritmo. Como por exemplo o consumo de memória.

Algumas notações O( ) comuns

O(1): Constante (acessa elemento em matriz, instruções simples),

O(lg(n)) : Logarítmica (busca binária) [A notação lg(n) é a abreviatura de $\log^2(n)$],

O(n): Linear (busca sequencial),

O(n lg(n)) : Pior que linear, mas não muito pior (tempo de execução médio da classificação rápida, classificação por heap),

O(n^2): Lei do quadrado (classificação por seleção e inserção),

O(n^3): Cúbica (multiplicação de 2 matrizes n x n),

O(c^n): Exponencial (problema do caixeiro viajante, divisão do conjunto).

Estimativa por bom senso

Você pode estimar a ordem de muitos algoritmos básicos usando o bom senso:

- **Loops simples** : Se um loop simples for executado de 1 a n, é provável que o algoritmo seja O(n)
- **Loops aninhados** : Se você aninhar um loop dentro de outro, o algoritmo será O(m x n),
- **Divisão binária** : Se seu algoritmo divide igualmente o conjunto de coisas que ele considera a cada vez que percorre o loop, provavelmente ele é logarítmico, O(lg(n))
- **Dividir para conquistar** : Algoritmos que dividem sua entrada, trabalham nas duas metades independentemente e então combinam o resultado podem ser O(n lg(n)).
- **Análise combinatória:** Sempre que os algoritmos começam a examinar as permutações das coisas, seus tempos de execução podem sair do controle. Isso ocorre porque as permutações envolvem fatoriais (há 5! = 5x4x3x2x1 = 120 permutações dos dígitos de 1 a 5).

Dicas

DICA 45: Estime a ordem de seus algoritmos.

Velocidade do algoritmo na prática

Se não tiver certeza de quanto tempo seu código levará ou quanta memória ele usará, execute-o, variando a contagem de registros de entrada ou outros fatores que afetam o tempo de execução.

Em seguida, represente os resultados graficamente. O formato do gráfico dará uma boa ideia do comportamento do código em relação ao tempo de execução.

Considere também o que você está fazendo no próprio código. Um loop $O(n^2)$ simples pode ser executado melhor do que um loop $O(n \lg(n))$ complexo para valores de n menores.

Considere: Prática varia. Pequenas entradas podem parecer lineares, mas com milhões, o tempo subitamente baixará a medida que o sistema começar a executar o thrashing.

Dicas

DICA 46: Teste suas estimativas.

O melhor nem sempre é melhor

É necessário ser pragmático na escolha de algoritmos apropriados – o mais rápido nem sempre é o melhor para o trabalho.

Também é importante estar atento ao custo de inicialização, no caso de pequenos conjuntos de dados essa instalação pode degradar o tempo de execução e tornar o algoritmo inapropriado.

Tome cuidado também com a otimização prematura. É sempre uma boa ideia se certificar se um algoritmo é realmente um gargalo antes de investir seu precioso tempo melhorando-o.

Refatoração

À medida que um programa evolui, decisões anteriores têm de ser repensadas e partes do código retrabalhadas. Esse processo é perfeitamente natural. O código precisa evoluir, ele não é estático.

Apesar do termo “Construção de Software” ser bastante comum, a sugestão de que construir um software é como construir um prédio é errônea.

Um termo mais condizente seria **jardinagem** . Um software é orgânico e concreto, é possível plantar várias coisas. Com o tempo algumas florescem e outras viram adubo.

O cuidador observa e ajusta o crescimento das plantas. Da mesma forma, no desenvolvimento de software, refatorar é reescrever, retrabalhar e replanejar códigos.

Quando você deve refatorar?

Quando deparar com algo que não funciona porque o código não é mais adequado, notar dois itens que na verdade deveriam ser um ou algo mais lhe parecer “errado”, não hesite em alterar.

Isso se aplica aos seguintes exemplos:

- **Duplicação** : Você descobriu uma violação do princípio NSR.
- **Projeto não ortogonal** : Você descobriu alguma parte do código ou projeto que poderia ser mais ortogonal.
- **Conhecimento desatualizado** : As coisas mudam, os requisitos variam e seu entendimento do problema aumenta. O código tem de acompanhar.
- **Desempenho** : Você precisa mover uma funcionalidade de uma área do sistema para outra para melhorar o desempenho.

Complicações do mundo real

Restrições de tempo costumam ser usadas para não refatoração. Essa desculpa não se sustenta: não refatore agora e precisará de um investimento maior para a correção do problema mais tarde.

Uma boa analogia é considerar um código que precisa de refatoração como um “tumor”. É melhor remover enquanto ainda é pequeno do que esperar ficar grande e ser ainda mais difícil de se lidar.

Dicas

DICA 47: Refatore cedo, refatore sempre.

Como refatorar?

Em sua essência, refatorar é replanejar. Refatoração é uma atividade que precisa ser executada lenta, deliberada e cuidadosamente. Algumas dicas são:

- Não tente refatorar e adicionar funcionalidades ao mesmo tempo.
- Certifique-se de ter bons testes antes de começar a refatorar.
- Execute etapas curtas e deliberadas.
- Quando encontrar um trecho de código que não estiver exatamente como deveria, além de corrigi-lo, corrija também tudo que depender dele. Não viva com janelas quebradas.

Código que seja fácil de testar

O circuito integrado de Software é uma metáfora que as pessoas gostam de usar quando discutem a capacidade de reutilização e o desenvolvimento baseado em componentes.

A ideia é que os componentes de software devem ser combinados da mesma forma que os chips de circuito integrado. Isso só funcionará se os componentes que você estiver usando forem realmente confiáveis.

Teste de unidade

Testes de unidade são feitos em cada módulo, isoladamente, para verificar seu comportamento. Podendo assim simular como um módulo reagirá no mundo real uma vez que for implementado.

Um teste de unidade de software é um código que avalia um módulo específico, criando um ambiente controlado e executando rotinas nesse módulo para verificar seu comportamento.

Após a conclusão, são verificados os resultados retornados. Comparando-os em relação a valores conhecidos ou a resultados de execuções anteriores do mesmo teste.

Testando em relação a um contrato: Parte 1

Testes em relação a um contrato tem como objetivo assegurar que uma determinada unidade honre seu contrato. Por exemplo:

```
require
   argument >= 0;
ensure
   ((Result * Result) - argument).abs <= epsilon*argument ;
```

Isso nos diz o que testar:

- Passe um argumento negativo e verifique se ele é rejeitado.
- Passe um argumento igual a zero para verificar se ele é aceito.
- Passe valores entre zero e o argumento máximo exprimível e verifique se a diferença entre o quadrado do resultado e o argumento original é menor do que alguma pequena fração do argumento.

Testando em relação a um contrato: Parte 2

Presumindo que a rotina faça sua própria verificação de pré e pós-condições, é possível escrever um script de teste básico para verificar o comportamento da função de raiz quadrada.

```
public void testValue(double num, double expected) {
   double result = 0.0;
   try { // Podemos lançar uma
       result = mySqrt(num) ; // exceção de pré-condição
   }
   catch (Throwable e) {
       if (num < 0.0) // Se a entrada for < 0,
           return; // estamos esperando a
       else // exceção, caso contrário
           assert(false); // force uma falha no teste
   }

   assert(Math.abs(expected-result) < epsilon*expected) ;
}
```

Testando em relação a um contrato: Parte 3

Agora, é possível chamar essa rotina para testar a função:

```
testValue(-4.0, 0.0);
testValue( 0.0, 0.0);
testValue( 2.0, 1.4142135624);
testValue(64.0, 8.0);
testValue(1.0e7, 3162.2776602);
```

Esse é um teste muito simples, no mundo real, é provável que qualquer módulo não trivial dependa de vários outros módulos, portanto, como testar esta combinação?

Testando em relação a um contrato: Parte 4

Supondo que exista um módulo A que usasse um objeto LinkedList e um objeto Sort. Em ordem, seriam testados:

- Todo o contrato de LinkedList.
- Todo o contrato de Sort.
- O contrato de A, que depende dos outros contratos, mas não os expõe diretamente.

Se os testes de LinkedList e Sort passarem, mas o de A falhar, podemos concentrar rapidamente nossos esforços de depuração em A ou em como A usa um desses subcomponentes, economizando tempo ao evitar revisões repetidas dos subcomponentes.

Testando em relação a um contrato: Parte 5

Todas estas etapas são importantes para evitar uma “bomba-relógio” – algo que permaneça despercebido e surja em um momento indesejado posteriormente no projeto.

Quando projetar um módulo, ou até mesmo uma única rotina, é necessário projetar tanto seu contrato quanto o código para testar esse contrato.

Não há uma maneira melhor de corrigir erros do que evitá-los. Na verdade, ao construir os testes antes de implementar o código, será possível testar a interface antes de decidir usá-la.

Dicas

DICA 48: Projete para testar.

Criando testes de unidade

Os testes de unidade de um módulo devem ser acessíveis. Em projetos pequenos, coloque o teste dentro do próprio módulo. Em projetos maiores, coloque-os em sub diretórios separados.

A regra é simples: se não for fácil encontrar, não será usado. Dessa forma serão fornecidos dois recursos inestimáveis:

- Exemplos de como usar toda a funcionalidade do módulo.
- Um meio de construir testes de regressão para validar qualquer alteração futura no código.

Apesar da conveniência, nem sempre é prático ter um teste de unidade para cada classe ou módulo. Em Java, por exemplo, todas as classes podem ter um método main, que pode ser usado para executar testes de unidade.

Construa uma janela de teste: Parte 1

Nem mesmo os melhores conjuntos de testes são capazes de encontrar todos os bugs. Assim será sempre necessário testar um software implementado, usando dados do mundo real.

Arquivos de log contendo mensagens de rastreamento são um mecanismo importante para visualizar o estado interno de um módulo.

As mensagens de log devem estar em um formato regular consistente. A partir delas é possível deduzir o tempo de processamento ou os caminhos lógicos adotados pelo programa.

Diagnósticos formatados de maneira pobre ou inconsistente são apenas verborragia – são difíceis de ler e impossíveis de analisar.

Construa uma janela de teste: Parte 2

Outra forma de verificar um código em execução é através de teclas de atalho. Ao pressionar uma combinação específica, uma janela de diagnóstico aparece, exibindo mensagens de status.

Para servidores complexos, uma solução é incluir um servidor Web interno. Isso possibilita aos usuários acessar facilmente informações do servidor via navegador na porta HTTP do aplicativo.

Uma cultura de teste

Todo software criado será testado, portanto é melhor planejar um teste integral. Prevenção pode ajudar a reduzir os custo de manutenção e necessidade de suporte técnico.

A comunidade Perl tem grande compromisso com teste de unidade e regressão. O procedimento padrão de instalação de módulos Perl dá suporte a um teste de regressão chamando

```
% make test
```

Ela torna mais fácil agrupar e analisar resultados de testes para assegurar a compatibilidade.

Testar é mais cultural do que técnico, é possível criar essa cultura de teste em um projeto independente da linguagem usada.

Dicas

DICA 49: Teste seu software ou seus usuários testarão.

Assistentes do mal: Parte 1

Com o passar do tempo a criação de aplicativos está se tornando cada vez mais complexa. Espera-se que estes programas sejam dinâmicos, flexíveis e compatíveis com outros aplicativos.

Com o crescimento da complexidade surgiu o *assistente.* Com apenas um clique e respondendo algumas perguntas o assistente gerará um código esqueleto pronto para ser usado.

Porém o uso de um assistente não torna um programado amador em um especialista, a menos que entenda completamente o que foi criado, ele estará apenas se enganando.

Assistentes apenas criam o código. Se estiver correto ou não o programador fica por conta própria.

Dicas

DICA 50: Não use um código de assistente que você não entender.

Assistentes do mal: Parte 2

Se um programador usar um assistente e não entender o que foi criado não terá seu aplicativo sob controle. Ele não será capaz de editá-lo e terá dificuldade quando tiver que depurá-lo.

Os assistentes não são como bibliotecas ou serviços padrão do sistema operacional em que os desenvolvedores podem confiar cegamente.

Uma vez que o código do assistente for integrado, ele se tornará parte do código do desenvolvedor. Mas será intercalado linha a linha, ao invés de ter uma interface bem definida.

Dessa forma, ninguém deve produzir códigos que não entende.

Sobre os autores

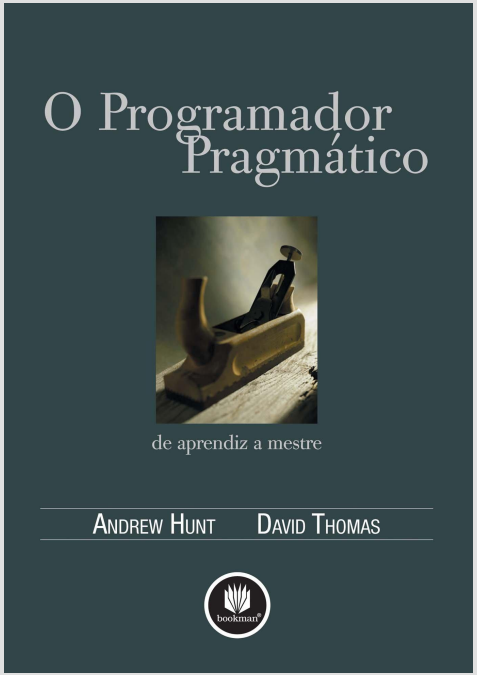
Andrew Hunt trabalhou em diversas áreas, como telecomunicações, serviços financeiros, artes gráficas etc. Hunt se especializou em combinar técnicas já consolidadas com tecnologias de ponta, criando soluções novas e práticas. Ele administra sua empresa de consultoria em Raleigh, Carolina do Norte.

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

Saulo Fernando, Autor

Aluno e bolsista do PET/ADS desde abril de 2023
LinkedIn

Lucas Oliveira, Revisor

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