6. Programação Defensiva

Programação defensiva não significa ser defensivo quanto à sua programação - "Ele faz, portanto funciona!" A ideia é baseada na noção de direção defensiva. Na direção defensiva, você adota a mentalidade de nunca ter certeza do que os outros motoristas poderão fazer no trânsito. Desse modo, você garante que, se eles fizerem alguma manobra perigosa, você não será prejudicado. Você assume a responsabilidade por se proteger, mesmo que a culpa seja de outro motorista. Na programação defensiva, a ideia principal é que, se uma rotina recebe dados incorretos, eles não causarão problemas, mesmo que tenham origem em outra rotina. De um modo mais geral, é o reconhecimento de que os programas terão problemas e modificações, e que um programador esperto desenvolverá código adequadamente.

6.1 Protegendo seu programa de entradas inválidas

Você já deve ter ouvido a expressão “entra lixo, sai lixo”. Em relação a *software* de produção, a expressão “Entra lixo, sai lixo” não é suficientemente adequada. Um bom programa nunca gera lixo na saída, independentemente do que receba. Em vez disso, um bom programa usa “Entra lixo, nada sai”, “Entra lixo, sai mensagem de erro” ou “Nenhum lixo pode entrar”. Pelos padrões atuais, “Entra lixo, sai lixo” é sinal de um programa desleixado e pouco seguro.

Existem três maneiras gerais de tratar o lixo que entra:

Verifique os valores de todos os dados de fontes externas - Quando receber dados de um arquivo, de um usuário, da rede ou de alguma outra interface externa, certifique-se de que eles estejam dentro do intervalo permitido. Certifique-se de que os valores numéricos estejam dentro das tolerâncias e que as *strings* tenham o comprimento adequado para serem manipuladas. Se uma *string* se destina a representar um intervalo de valores restrito (como uma identificação de transação financeira ou algo semelhante), certifique-se de que ela seja válida para seu propósito planejado; caso contrário, rejeite-a. Se você estiver trabalhando em um aplicativo seguro, desconfie de dados que possam atacar seu sistema: tentativas de estouros de *buffer,* comandos SQL, código HTML ou XML embutidos, estouros de inteiros, dados passados para chamadas de sistema e assim por diante.

Verifique os valores de todos os parâmetros de entrada da rotina - Verificar os valores dos parâmetros de entrada da rotina é basicamente o mesmo que verificar dados provenientes de uma fonte externa, exceto pelo fato de que, nesse caso, os dados vêm de outra rotina, em vez de uma interface externa.

Decida como irá tratar de entradas incorretas - Quando você detectar um parâmetro inválido, o que fará com ele?

A melhor forma de codificação defensiva é não inserir erros. Usar projeto iterativo, escrever pseudocódigo antes do código, escrever casos de teste antes de escrever o código, e ter inspeções de projeto de baixo nível; todos esses procedimentos ajudam a evitar a inserção de defeitos. Assim, eles devem receber uma prioridade mais alta do que a programação defensiva.

6.2 Assertivas

Assertiva é utilizada durante o desenvolvimento - normalmente uma rotina ou uma macro - que permite a um programa fazer sua própria verificação enquanto é executado. Quando uma assertiva é verdadeira significa que tudo está funcionando conforme o esperado. Quando ela é falsa, significa que ela detectou um erro inesperado no código. Por exemplo, se o sistema presume que um arquivo de informação do cliente nunca terá mais do que 50.000 registros, ele poderia conter uma assertiva dizendo que o número de registros é menor ou igual a 50.000. Desde que o número de registros seja menor ou igual a 50.000, a assertiva permanecerá em silêncio. Entretanto, se encontrar mais do que 50.000 registros, ela deve acusar que existe um erro no programa.

As assertivas são especialmente úteis em programas grandes e complexos, e em programas de alta confiabilidade. Elas permitem que os programadores descubram mais rapidamente suposições de interface mal combinadas, erros que entram furtivamente quando o código é modificado e etc.

Normalmente, uma assertiva recebe dois argumentos na expressão booleana, que descreve a suposição que deve ser verdadeira, e uma mensagem a ser exibida, caso não seja verdadeira. Veja a seguir como ficaria uma assertiva em Java, dentro da expectativa de que a variável *denominador* tivesse valor diferente de zero:

```
assert denominador != 0 : “denominador é igual a 0”
```

Essa assertiva afirma que denominador não é igual a 0. O primeiro argumento, denominador != , é uma expressão booleana que é avaliada como verdadeira ou falsa. O segundo argumento é uma mensagem a ser impressa se o primeiro argumento for falso - isto é, se a assertiva for falsa.

Use assertivas para documentar as suposições feitas no código e para descobrir condições inesperadas. As assertivas podem ser usadas para verificar suposições como as seguintes:

- Se o valor de um parâmetro de entrada (ou de saída) cai dentro de seu intervalo esperado
- Se um arquivo ou fluxo está aberto (ou fechado) quando uma rotina começa a ser executada (ou quando ela acaba de ser executada)
- Se um arquivo ou fluxo está no início (ou no fim) quando uma rotina começa a ser executada (ou quando ela acaba de ser executada)
- Se um arquivo ou fluxo está aberto somente para leitura, somente para gravação ou tanto para leitura quanto para gravação
- Se o valor de uma variável somente de entrada não é alterado por uma rotina
- Se um ponteiro é não-nulo
- Se um *array* ou outro contêiner passado para uma rotina pode conter pelo menos um número X de elementos de dados
- Se uma tabela foi inicializada para conter valores reais
- Se um contêiner está vazio (ou cheio) quando uma rotina começa a ser executada (ou quando acaba de ser executada)
- Se os resultados de uma rotina altamente otimizada e complicada correspondem aos resultados de uma rotina mais lenta, porém escrita de forma clara

Normalmente, você não quer que os usuários vejam mensagens de assertiva no código de produção; as assertivas servem principalmente para uso durante o desenvolvimento e durante a manutenção. As assertivas normalmente são compiladas no código no momento do desenvolvimento e retiradas da compilação do código para produção. Durante o desenvolvimento, as assertivas encontram suposições contraditórias, condições inesperadas, valores incorretos passados para rotinas e coisas do gênero. Durante a produção, elas podem ser retiradas da compilação do código, para que não degradem

o desempenho do sistema.

Construindo seu próprio mecanismo de assertiva

Muitas linguagens possuem suporte interno para assertivas, incluindo C++, Java e Microsoft Visual Basic. Se sua linguagem não suporta rotinas de assertiva diretamente, saiba que elas são fáceis de escrever. A macro *assert* padrão da linguagem C++ não fornece mensagens de texto. Exemplo de rotina *ASSERT*, implementada como uma macro C++:

```
#define ASSERT (condicao, mensagem) {
        if (!(condicao)) {
              LogError("Falha: ", #condicao, mensagem);
              exit(EXIT_FAILURE);
        }
}
```

Diretrizes para o uso de assertivas

Aqui estão algumas diretrizes para o uso de assertivas:

Use código de tratamento de erro para as condições que você espera que ocorram; use assertivas para as condições que **nunca** ***devem ocorrer*** - As assertivas verificam as condições que *nunca* devem ocorrer. O código de tratamento de erro verifica as circunstâncias anormais que podem não ocorrer com muita frequência, mas que foram previstas pelo programador que escreveu o código e precisam ser tratadas pelo código de produção. Normalmente, o tratamento de erro verifica a existência de dados de entrada incorretos; as assertivas verificam a existência de erros no código.

Se o código de tratamento de erro for usado para resolver uma condição anormal, o tratamento de erro permitirá que o programa responda ao erro normalmente. Se uma assertiva é ativada para uma condição anormal, a ação corretiva normalmente não é apenas tratar de um erro – a ação corretiva é alterar o código-fonte do programa, recompilar e lançar uma nova versão do *software.*

Uma boa maneira de interpretar as assertivas é como uma documentação executável - você não pode contar com elas para fazer o código funcionar, mas elas podem documentar as suposições mais ativamente do que os comentários de linguagem de programação.

Evite colocar código executável em assertivas - Colocar código em uma assertiva cria a possibilidade de que o compilador elimine o código quando você desativar as assertivas. Suponha que você tenha uma assertiva como o código a seguir. O problema desse código é que, se você não compilar as assertivas, não compilará o código que executa a ação.

```
Debug.Assert ( PerformAction() )
```

Coloque as instruções executáveis em suas próprias linhas, atribua os resultados a variáveis de status e teste essas variáveis. Eis um exemplo de uso seguro de uma assertiva:

```
actionPerformed = PerformAction();
Debug.Assert ( actionPerformed );
```

Use assertivas para documentar e verificar pré-condições e pós-condições - As pré-condições e pós-condições fazem parte de uma estratégia de projeto e desenvolvimento de programas conhecida como "projeto por contrato" (Meyer 1997). Quando pré-condições e pós-condições são usadas, cada rotina ou classe estabelece um contrato com o restante do programa.

Pré-condições são as propriedades que o código-cliente de uma rotina ou classe promete que serão verdadeiras, antes de chamar a rotina ou instanciar o objeto. As pré-condições são as obrigações do código cliente com o código que ele chama.

Pós-condições são as propriedades que a rotina ou classe promete que serão verdadeiras quando concluir a execução. As pós-condições são as obrigações da rotina ou da classe com o código que a utiliza.

As assertivas são uma ferramenta útil para documentar pré-condições e pós-condições. Comentários poderiam ser usados para documentar pré-condições e pós-condições, mas, ao contrário dos comentários, as assertivas podem verificar dinamicamente a sua validade.

No exemplo a seguir, são usadas assertivas para documentar as pré-condições e a pós-condição da rotina *Velocity.*

Se as variáveis *latitude, longitude* e *elevation* fossem provenientes de uma fonte externa, valores inválidos deveriam ser verificados e manipulados pelo código de tratamento de erro e não pelas assertivas. No entanto, se as variáveis estiverem vindo de uma fonte interna confiável e o projeto da rotina for baseado na suposição de que esses valores estarão dentro de seus intervalos válidos, então as assertivas serão apropriadas.

```
Exemplo em VB de uso de assertivas para documentar pré e pós-condições

Private Function Velocity (
  ByVal latitude As Single,
  ByVal longitude As Single,
  ByVal elevation As Single
  ) As Single

  'Pré-condições
  Debug.Assert ( -90 <= latitude And latitude <= 90 )
  Debug.Assert ( 0 <= longitude And longitude < 360 )
  Debug.Assert ( -500 <= elevation And elevation <= 75000 )
  ...

  'Pós-condições
  Debug.Assert (0 <= returnVelocity And returnVelocity <= 600 )

  'valor de retorno
  Velocity = returnVelocity
End Function
```

Para código altamente robusto, faça uma assertiva e depois trate do erro de qualquer maneira - Para qualquer condição de erro dada, uma rotina geralmente usará uma assertiva ou um código de tratamento de erro, mas não ambos. Alguns especialistas argumentam que apenas um tipo é necessário (Meyer 1997).

No entanto, os programas e projetos do mundo real tendem a ser desorganizados demais para contar unicamente com as assertivas. Em um sistema grande e com longa expectativa de vida, diferentes partes podem ser desenvolvidas por diferentes *designers* em um período de 5 a 10 anos ou mais. Os *designers* estarão separados no tempo e dispersos por numerosas versões. Seus projetos focalizarão diferentes tecnologias em diferentes pontos ao logo do ciclo de vida do sistema. Os *designers* estarão

geograficamente separados, principalmente se partes do sistema forem adquiridas de fontes externas. Os programadores, igualmente, terão trabalhado segundo diferentes padrões de codificação, em diferentes momentos na vida do sistema. Em uma equipe de desenvolvimento numerosa, alguns programadores inevitavelmente serão mais conscienciosos do que outros e algumas partes do código serão examinadas com maior rigor do que outras. Alguns programadores farão testes unitários mais completos de seus códigos do que outros. Com equipes de teste dispersas em diferentes regiões geográficas e sujeitas às pressões comerciais que resultam em variações na cobertura de teste a cada lançamento, você também não pode contar, sempre, com testes de regressão abrangentes em nível de sistema.

Em tais circunstâncias, tanto assertivas como código de tratamento de erro poderiam ser usados para tratar do mesmo erro. No código-fonte do Microsoft Word, por exemplo, as condições que sempre devem ser verdadeiras têm assertivas, mas tais erros também são abordados por código de tratamento de erro, para o caso de a assertiva falhar. Para aplicativos extremamente grandes, complexos e de longa duração, como o Word, as assertivas são valiosas, pois elas ajudam a descobrir o máximo de erros possível durante o desenvolvimento. Mas o aplicativo é tão complexo (- milhões de linhas de código) e tem passado por tantas gerações de modificação que não é realista supor que cada erro concebível seja detectado e corrigido antes que o *software* seja distribuído; assim, os erros também precisam ser tratados na versão de produção do sistema.

Aqui está uma amostra de como isso funcionaria no exemplo da rotina *Velocity.*

```
Exemplo em Visual Basic do uso de assertivas para documentar pré e pós-condições

Private Function Velocity ( _
     ByRef latitude As Single, _
     ByRef longitude As Single, _
     ByRef elevation As Single _
     ) As Single

     'Pré-condições
     Debug.Assert ( -90 <= latitude And latitude <= 90 )
     Debug.Assert ( 0 <= longitude And longitude < 360 )
     Debug.Assert ( -500 <= elevation And elevation <= 75000 )

     'Desinfeta os dados de entrada. Os valeres deves estar dentro dos inter-
     valos acima, mas se um valor não estiver dentro de seu intervalo válido,
     ele será alterado para o valor válido mais próximo

     If ( latitude < -90 ) Then
          latitude = -90
     ElseIf ( latitude > 90 ) Then
          latitude = 90
     End If
     If ( longitude < 0 ) Then
          Longitude = 0
     ElseIf ( longitude > 360 ) Then
     ...
```

6.3 Técnicas de tratamento de erro

As assertivas são usadas para tratar de erros que nunca devem ocorrer no código. Como você trata de erros que espera que ocorram? Dependendo das circunstâncias específicas, talvez você queira retornar um valor neutro, substituir pelos próximos dados válidos, retornar a mesma resposta da vez anterior, substituir pelo valor válido mais próximo, registrar uma mensagem de alerta em um arquivo de log, retornar um código

de erro, chamar uma rotina ou objeto de processamento de erro, exibir uma mensagem de erro ou terminar a execução - ou talvez você queira usar uma combinação dessas respostas.

Retornar um valor neutro - Às vezes, a melhor resposta para dados incorretos é continuar funcionando e simplesmente retornar um valor que sabidamente seja inofensivo. Um cálculo numérico poderia retornar `0`. Uma operação de *string* poderia retornar uma *string* vazia, uma operação de ponteiro poderia retornar um ponteiro vazio. Uma rotina de desenho que receba um valor de entrada incorreto para cor em um videogame poderia usar a cor de fundo ou a cor de primeiro plano padrão. Entretanto, uma rotina de desenho que exibe dados de raio X para pacientes com câncer não poderia exibir um "valor neutro". Nesse caso, seria melhor terminar o programa do que exibir dados incorretos do paciente.

Substituir pelos próximos dados válidos - No processamento de um fluxo de dados, algumas circunstâncias exigem simplesmente o retorno dos próximos dados válidos. Se você estiver lendo registros de um banco de dados e encontrar um registro corrompido, poderá simplesmente continuar a ler, até encontrar um registro válido. Se você estiver verificando leituras de um termômetro 100 vezes por segundo e não obtiver uma leitura válida uma vez, poderá simplesmente esperar mais 1/100 segundos e levar em consideração a próxima leitura.

Retornar a mesma resposta da vez anterior - Se o software de leitura de termômetro não obtiver uma leitura uma vez, ele poderá simplesmente retornar o mesmo valor da última leitura. Dependendo da aplicação, as temperaturas podem não mudar muito em 1/100 segundos. Em um videogame, se você detectar um pedido para pintar parte da tela com uma cor inválida, pode simplesmente retornar a mesma cor usada anteriormente. Mas se você estivesse autorizando transações em um caixa automático, provavelmente não desejaria usar a "mesma resposta da última vez" - ela seria o número da conta bancária do usuário anterior!

Substituir pelo valor válido mais próximo - Em alguns casos, você poderia optar por retomar o valor válido mais próximo, como no exemplo da rotina *Velocity.* Normalmente, essa é uma estratégia razoável ao se fazer leituras em um instrumento calibrado. O termômetro poderia ser calibrado entre `0` e `100` graus Celsius, por exemplo. Se você detectar um leitura menor do que `0`, pode substituir por `0`, que é o valor válido mais próximo. Se você detectar um valor maior do que `100`, pode substituir por `100`. Para uma operação de *string,* se constasse que o comprimento de uma *string* é menor do que `0`, você poderia substituir por `0`. Meu carro usa essa estratégia de tratamento de erro quando engato marcha a ré. Como meu velocímetro não mostra velocidades negativas, quando dou a ré ele mostra simplesmente uma velocidade igual a `0` - o valor válido mais próximo.

Registrar uma mensagem de alerta em um arquivo de `log` - Quando dados incorretos são detectados, você pode optar por registrar uma mensagem de alerta em um arquivo e, então, continuar. Essa estratégia pode ser usada em conjunto com outras técnicas, como a substituição pelo valor válido mais próximo ou a substituição pelos próximos dados válidos. Se você usar um *log,* considere se pode torná-lo publicamente disponível ou se precisa criptografá-lo ou protegê-lo de alguma outra maneira.

Retornar um código de erro - Você poderia decidir que apenas certas panes de um sistema tratarão dos erros. Outras panes não tratarão dos erros de forma local;

elas simplesmente informarão que um erro foi detectado e esperarão que alguma outra rotina, que esteja mais alto na hierarquia de chamadas, trate do erro. O mecanismo específico para notificar o restante do sistema de que um erro ocorreu poderia ser qualquer um dos seguintes:

- Configurar o valor de uma variável de status
- Retornar o status como valor de retorno da função
- Lançar uma exceção usando o mecanismo de exceção interno da linguagem

Nesse caso, o mecanismo de relato de erro específico tem menos importância do que a decisão sobre quais partes do sistema tratarão de erros diretamente e quais apenas informarão que eles ocorreram. Se a segurança for um problema, certifique-se de que as rotinas de chamada sempre verifiquem os códigos de retorno.

Chamar uma rotina/objeto de processamento de erro - Outra estratégia é centralizar o tratamento de erro em uma rotina de tratamento de erro global ou em um objeto de tratamento de erro. A vantagem dessa estratégia é que a responsabilidade pelo processamento do erro pode ser centralizada, o que pode tornar a depuração mais fácil. A contrapartida é que o programa inteiro saberá a respeito desse recurso central e será acoplado a ele. Se você quiser reutilizar qualquer parte do código do sistema em outro sistema, terá que arrastar o mecanismo de tratamento de erro com o código a ser reutilizado.

Esta estratégia tem uma implicação importante na segurança. Se seu código tiver encontrado um estouro de *buffer,* é possível que um invasor tenha comprometido o endereço da rotina ou do objeto manipulador. Assim, quando um estouro de buffer tiver ocorrido durante a execução de um aplicativo, não será mais seguro utilizar esta estratégia.

Exibir uma mensagem de erro quando o erro for encontrado - Esta estratégia minimiza a sobrecarga do tratamento de erro; entretanto, ela tem o potencial de espalhar mensagens da interface com o usuário por todo o aplicativo, o que pode gerar desafios quando você precisar de uma interface com o usuário consistente, quando tentar separar claramente a interface com o usuário do restante do sistema ou quando tentar traduzir o *software* para um idioma diferente. Além disso, cuidado para não dar muitas informações para um potencial invasor do sistema. Às vezes, os invasores usam mensagens de erro para descobrir como atacar um sistema.

Tratar do erro da maneira que funcionar melhor no local - Alguns projetos de *software* exigem um tratamento local de todos os erros - a decisão sobre qual método de tratamento de erro específico será usado ficará por conta do programador que fizer o projetar e implementar a parte do sistema que encontrar o erro.

Essa estratégia proporciona muita flexibilidade aos desenvolvedores individuais, mas cria um risco significativo de que o desempenho global do sistema não satisfaça seus requisitos de correção ou robustez. Dependendo de como os desenvolvedores acabarem tratando dos erros específicos, esta estratégia também tem o potencial de espalhar código de interface com o usuário por todo o sistema, o que expõe o programa a todos os problemas associados à exibição de mensagens de erro.

Terminar a execução - Alguns sistemas terminam a execução quando detectam um erro. Essa estratégia é útil em aplicativos onde a segurança é crítica. Por exemplo, se o software que controla um equipamento de radiação para tratamento de pacientes com câncer receber dados de entrada inválidos para a dosagem de radiação, qual é sua melhor resposta para o tratamento do erro? Ele deve usar o mesmo valor da última vez?

Deve usar o valor válido mais próximo? Deve usar um valor neutro? Nesse caso, desligar é a melhor opção. Obviamente, é preferível reiniciar a máquina do que correr o risco de descarregar a dosagem errada.

Uma estratégia semelhante pode ser usada para melhorar a segurança do Microsoft Windows. Por padrão, o Windows continua a operar, mesmo quando seu *log* de segurança está cheio. Mas você pode configurar o Windows para que pare o servidor, caso o *log* de segurança fique cheio, o que pode ser apropriado em um ambiente onde a segurança é crítica.

Robustez *versus* correção

Conforme nos mostram os exemplos do *videogame* e do raio X, o estilo de processamento de erro considerado o mais apropriado depende do tipo de *software* em que o erro ocorre. Esses exemplos também ilustram que o processamento de erro geralmente favorece mais a correção ou mais a robustez. Os desenvolvedores tendem a usar esses termos informalmente, mas, rigorosamente falando, eles estão em extremos opostos da escala. *Correção* significa nunca retornar um resultado impreciso; não retornar nenhum resultado é melhor do que retornar um resultado impreciso. *Robustez* significa sempre tentar fazer algo que permita ao *software* continuar funcionando, mesmo que às vezes isso leve a resultados imprecisos.

Os aplicativos em que a segurança é crítica tendem a privilegiar a correção em detrimento da robustez. É melhor não retornar nenhum resultado do que retornar um resultado errado. A máquina de radiação é um bom exemplo desse princípio.

Os aplicativos comerciais tendem a privilegiar a robustez em detrimento da correção. Qualquer resultado normalmente é melhor do que terminar a execução do *software.* O processador de textos que estou usando ocasionalmente exibe um fragmento de uma linha de texto na parte inferior da tela. Se ele detectar essa condição, eu vou querer que o processador de textos seja encerrado? Não. Eu sei que na próxima vez em que pressionar a tecla Page Up ou Page Down, a tela será atualizada e a exibição voltará ao normal.

Com tantas opções, você precisa tomar o cuidado de manipular parâmetros inválidos de maneiras consistentes ao longo de todo o programa. A maneira como os erros são tratados afeta a capacidade do *software* de atender aos requisitos relacionados com a correção, robustez e outros atributos não-funcionais. Definir uma estratégia geral para parâmetros inválidos é uma decisão arquitetônica ou de projeto de alto nível, e deve ser tratada em um desses níveis.

Quando você decidir sobre a estratégia, certifique-se de segui-la com firmeza. Se você optar por fazer com que um código de alto nível trate dos erros e com que um código de baixo nível apenas os informe, certifique-se de que o código de alto nível realmente trate dos erros! Algumas linguagens fornecem a opção de ignorar o fato de que uma função está retornando um código de erro - em C++, você não é obrigado a fazer nada com o valor de retorno de uma função –, mas não ignore informações de erro! Teste o valor de retorno da função. Se você espera que a função jamais produza um erro, verifique-a de qualquer maneira. O principal objetivo da programação defensiva é se precaver dos erros que você não espera.

Essa diretriz também vale para funções de sistema, assim como para suas próprias funções. A não ser que você tenha estabelecido uma diretriz arquitetônica para não verificar erros em chamadas de sistema, verifique os códigos de erro após cada chamada. Se você detectar um erro, inclua o número e a descrição do erro.

6.4 Exceções

As exceções são um meio específico pelo qual o código pode passar erros ou eventos

excepcionais ao código que o chamou. Se o código de uma rotina encontra uma condição inesperada que não sabe como tratar, ele lança uma exceção, basicamente levantando as mãos em resignação e gritando: "Eu não sei o que fazer em relação a isso - espero realmente que alguém saiba como tratar disso!" Um código que não tem compreensão de contexto de um erro pode retornar o controle para outras partes do sistema que possam ter uma capacidade melhor de interpretar o erro e fazer algo de útil a respeito dele.

As exceções também podem ser usadas para organizar uma lógica complicada dentro de um trecho de código. A estrutura básica de uma exceção é que uma rotina usa *throw* para lançar um objeto de exceção. O código de alguma outra rotina mais acima na hierarquia de chamada capturará a exceção dentro de um bloco try-catch.

As linguagens populares variam no modo de implementar exceções. A Tabela resume as principais diferenças entre três delas:

Tabela: Suporte para exceções em linguagens populares

Atributo de exceção	C++	Java	Visual Basic
Suporte para *try-catch*	sim	sim	sim
Suporte para *try-catch-finaly*	não	sim	sim
O que pode ser lançado	Objeto *Exception* ou objeto derivado da classe *Exception* ponteiro de objeto; referência de objeto; tipo de dados como *string* ou *int*	Objeto *Exception* ou objeto derivado da classe *Exception*	Objeto *Exception ou* objeto derivado da classe *Exception*
Efeito de exceção não capturada	Ativa *std:: unexpected(),* que por padrão ativa *std:: terminate(),* que por padrão ativa *abort()*	Termina a linha de execução, caso a exceção seja uma "exceção verificada"; nenhum efeito, caso a exceção seja uma "exceção em tempo de execução"	Termina o programa
As exceções lançadas devem ser definidas na interface de classe	Não	Sim	Não
As exceções capturadas devem ser definidas na interface de classe	Não	Sim	Não

As exceções têm um atributo em comum com a herança: usadas com sabedoria, elas podem reduzir a complexidade. Usadas imprudentemente, elas podem tornar o código quase impossível de seguir. Algumas sugestões para a compreensão das vantagens das exceções e para evitar as dificuldades frequentemente associadas a elas.

Use exceções para notificar outras partes do programa sobre erros que não devem ser ignorados - A principal vantagem das exceções é sua capacidade de sinalizar

condições de erro de tal maneira que elas não possam ser ignoradas. Outras estratégias de tratamento de erros criam a possibilidade de que uma condição de erro possa se propagar por uma base de código sem ser percebida. As exceções eliminam essa possibilidade.

Lance uma exceção somente para as condições que sejam realmente excepcionais - As exceções devem ser reservadas para as condições que são verdadeiramente excepcionais - em outras palavras, para as condições que não podem ser tratadas por outras práticas de codificação. As exceções são usadas em circunstâncias similares às assertivas - para eventos que não são apenas pouco frequentes, mas para eventos que *nunca* devem ocorrer.

As exceções representam um equilíbrio entre uma maneira poderosa de tratar de condições inesperadas, por um lado, e uma maior complexidade, por outro. As exceções enfraquecem o encapsulamento, exigindo que o código que chama uma rotina saiba quais exceções podem ser lançadas dentro do código chamado. Isso aumenta a complexidade do código, o que vai contra os padrões de projeto.

Não use uma exceção para se eximir da responsabilidade - Se uma condição de erro pode ser tratada no local, trate-a no local. Não lance uma exceção não-capturada em uma seção de código, se você puder tratar do erro de forma local.

Evite lançar exceções em construtores e destrutores, a não ser que você as capture no mesmo lugar - As regras sobre como as exceções são processadas se tornam complicadas rapidamente quando exceções são lançadas em construtores e destrutores. Em C++, por exemplo, os destrutores não são chamados, a não ser que um objeto seja totalmente construído, o que significa que, se o código dentro de um construtor lançasse uma exceção, o destrutor não seria chamado, configurando assim um possível erro de ausência de recurso.

Os especialistas em linguagens de programação podem dizer que lembrar de regras como essas é "trivial", mas os programadores que são simples mortais terão dificuldade de lembrar delas. É considerada uma prática melhor de programação simplesmente evitar a complexidade extra que tal código cria, não escrevendo esse tipo de código.

Lance exceções no nível de abstração correto - Uma rotina deve apresentar uma abstração consistente em sua interface: uma classe também. As exceções lançadas fazem parte da interface da rotina, assim como os tipos de dados específicos.

Quando você optar por passar uma exceção para a rotina que fez a chamada, certifique-se de que o nível de abstração da exceção seja consistente com a abstração da interface da rotina. Aqui está um exemplo do que não fazer:

```
//Mau exemplo em Java de uma classe que lança uma exceção em um nível de abstração
inconsistente

    class Employee {
     public TaxId GetTaxId( ) throws EOFException {
        ...
     }
     ...
    }
```

O código *GetTaxId()* devolve a exceção *EOFException,* de nível mais baixo, para seu chamador. Ele não toma posse da exceção em si; ele expõe alguns detalhes sobre como

é implementado, passando a exceção de nível mais baixo para a rotina que fez a chamada. Isso acopla efetivamente o código do cliente da rotina não com o código da classe *Employee,* mas com o código abaixo da classe *Employee*, que lança a exceção *EOFException,* o encapsulamento é quebrado e o controle intelectual começa a declinar.

Um vez disso, o código *GetTaxId()* deve devolver uma exceção compatível com a interface de classe da qual faz parte, como segue:

Bom exemplo em Java de uma classe que lança uma exceção em um nível compatível de abstração

```
claas Employee {
    ...
    public TaxId GetTaxId() throws EmployeeDataNovAvailable {
       ...
    }
    ...
}
```

O código de tratamento de exceção dentro de *GetTaxId()* provavelmente apenas mapeará a exceção *io_disk_not_ready* na exceção *EmployeeDataNotAvailable*, o que está certo, pois isso é suficiente para preservar a abstração da interface.

Inclua na mensagem de exceção todas as informações que levem à exceção - Toda exceção ocorre em circunstâncias específicas detectadas no momento em que o código lança a exceção. Essa informação é valiosa para a pessoa que lê a mensagem de exceção. Certifique-se de que a mensagem contenha as informações necessárias para entender porque a exceção foi lançada. Se a exceção foi lançada devido a um erro de índice *de array,* certifique-se de que a mensagem de exceção inclua os limites superior e inferior do *arrqy* e o valor do índice inválido.

Evite blocos **catch** ***vazios*** - Às vezes, é tentador se desfazer de uma exceção com a qual você não sabe o que fazer, como segue:

Mau exemplo em Java da maneira de ignorar uma exceção

```
try {
   ...
   // muito código
} catch ( AnException exception ) {
}
```

Tal estratégia estabelece que: ou o código dentro do bloco *try* está errado, porque ele lança uma exceção sem nenhum motivo, ou o código dentro do bloco *catch* está errado, porque ele não trata de uma exceção válida. Determine qual é a causa raiz do problema e, então, corrija o bloco *try* ou o bloco *catch.*

Conheça as exceções que seu código de biblioteca lança - Se você estiver trabalhando em uma linguagem que não exija que uma rotina ou classe defina as exceções que lança, certifique-se de saber quais exceções são lançadas por qualquer código de biblioteca utilizado. Deixar de capturar uma exceção gerada por código de biblioteca fará com que seu programa falhe tão rápido quanto o fato de deixar de capturar uma exceção que você mesmo gerou. Se o código de biblioteca não documenta as exceções que lança, crie um código de protótipo para colocar as bibliotecas em uso e descobrir as exceções.

Considere a construção de um relator de exceção centralizado - Uma

estratégia para garantir a consistência no tratamento de exceção é usar um relator de exceção centralizado. O relator de exceção centralizado fornece um repositório central de conhecimentos sobre os tipos de exceções existentes, como cada exceção deve ser tratada, a formatação de mensagens de exceção e etc.

Aqui está um exemplo de rotina de tratamento de exceção simples que apenas imprime uma mensagem de diagnóstico:

```
    Exemplo em Visual Basic de relator de exceção centralizado, parte 1

    Sub ReportException( _
        ByVal className, _
        ByVal thisException As Exception _
    )
        Dim message As String
        Dim caption As String

        message = “Exception: “ & thisException.Message & “”. & ControlChars.CrLf
& “Class: “ & className & ControlChars.CrLf & "Routine: ” &
thisException.TargetSite.Name & ControlChars.CrLf
        caption = "Exception”
        MessageBox.Show( message, caption, MessageBoxButtons.OK,
MessageBoxIcon-Exclamation )
```

Você usaria essa rotina de tratamento de exceção genérica com um código como o seguinte:

```
    Exemplo em Visual Basic de um relator de exceção centralizado, parte 2

    Try
        ...
    Catch exceptionObject As Exception
        ReportException ( CLASS_NAME, exceptionObject )
    End Try
```

O código dessa versão de *ReportException() é* simples. Em uma aplicação real, você poderia tornar o código tão simples ou tão elaborado quanto necessário para atender suas necessidades de tratamento de exceção.

Se você decidir construir um relator de exceção centralizado, certifique-se de considerar os problemas gerais envolvidos no tratamento de erro centralizado.

Padronize o uso de exceções em seu projeto - Para manter o tratamento de exceção o mais intelectualmente controlável possível, você pode padronizar o uso de exceções de várias maneiras:

- Se você estiver trabalhando em uma linguagem como a C++, que permite lançar uma variedade de tipos de objetos, dados e ponteiros, padronize especificamente o que você lançará. Por questões de compatibilidade com outras linguagens, considere o lançamento apenas de objetos derivados da classe de base *Exception.*
- Considere a criação de sua própria classe de exceção específica do projeto, a qual pode servir como classe de base para todas as exceções lançadas em seu projeto. Isso ajuda na centralização e na padronização de registros em *log, o* relato de erros e assim por diante.
- Defina as circunstâncias específicas sob as quais o código pode usar sintaxe *throw-catch* para realizar o processamento de erro de forma local.
- Defina as circunstâncias específicas sob as quais o código pode lançar uma exceção que não seria tratada de forma local.
- Determine se um relator de exceção centralizado será usado.
- Defina se são permitidas exceções em construtores e destrutores.

Pense em alternativas às exceções - Várias linguagens de programação têm suportado exceções há tempos, mas pouco critério convencional tem aparecido a respeito de como usá-las com segurança.

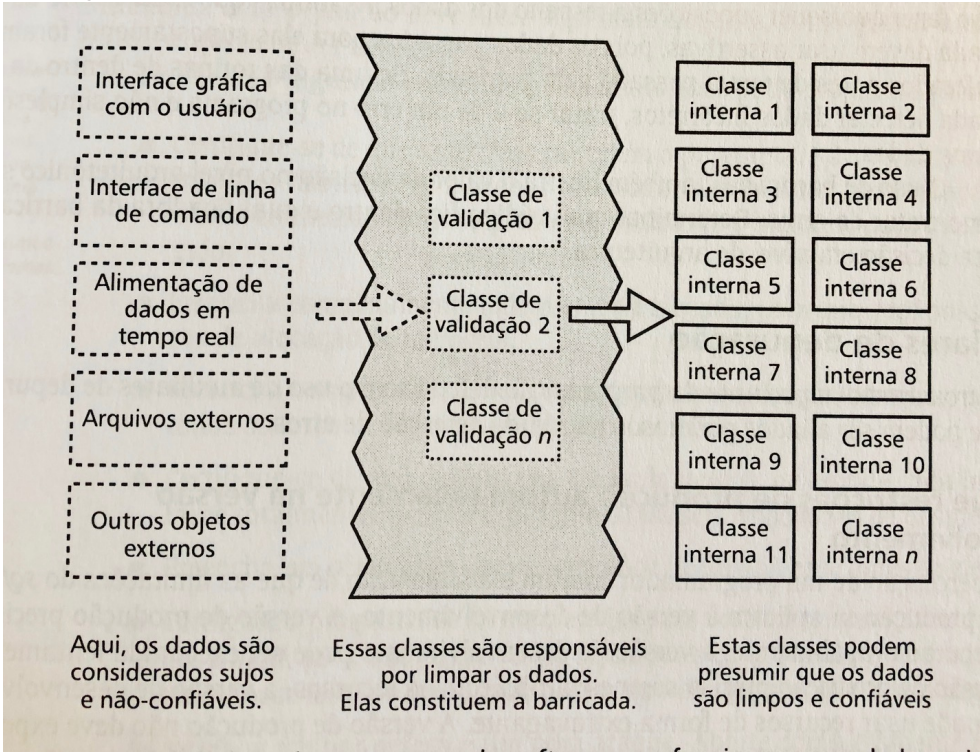
Alguns programadores usam exceções para tratar de erros apenas porque a linguagem que utilizam fornece esse mecanismo de tratamento de erro em particular. Você sempre deve considerar o conjunto completo de alternativas de tratamento de erro: tratamento de erro de forma local, propagação do erro por intermédio do uso de um código de erro, registro de informações de depuração em um arquivo de *log*, término da execução do sistema ou o uso de alguma outra estratégia. Tratar erros com exceções apenas porque sua linguagem fornece tratamento de exceção é um exemplo clássico de programação *em* uma linguagem, em vez de programação para uma linguagem.

Finalmente, verifique se seu programa realmente precisa tratar de exceções, ponto final. Às vezes a melhor resposta para um erro sério em tempo de execução é liberar todos os recursos adquiridos e cancelar a operação. Deixe o usuário executar o programa novamente, com a entrada correta.

6.5 Defenda seu programa com barricada para limitar o dano causado por erros

As barricadas são uma estratégia de confinamento de danos. O motivo é semelhante àquele de se ter compartimentos isolados no casco de um navio. Se o navio colidir com um *iceberg* e o casco se romper, esse compartimento será fechado e o resto do navio não será afetado. As barricadas também são semelhantes às paredes corta-fogo (*firewall*) de um prédio. As paredes corta-fogo impedem que o fogo passe de um recinto para outro do edifício. (As barricadas se chamavam *"firewalls"*, mas agora o termo *"firewall*" se refere ao bloqueio de tráfego de rede hostil.)

Uma maneira de usar barricada para propósitos de programação defensiva é designar certas interfaces como limites de áreas "seguras". Verifique a validade dos dados que ultrapassam os limites de uma área segura e responda sensatamente se eles não são válidos. A Figura ilustra esse conceito.



Definir algumas partes do *software* que funcionam com dados sujos e algumas que funcionam com dados limpos é uma maneira eficiente de retirar da maior parte do código a responsabilidade de verificar a existência de dados incorretos.

Essa mesma estratégia pode ser usada no nível da classe. Os métodos públicos da classe presumem que os dados são inseguros, e são responsáveis por examinar esses dados e torná-los limpos. Quando os dados tiverem sido aceitos pelos métodos públicos da classe, os seus métodos privados poderão considerar que eles são seguros.

Outra maneira de interpretar essa estratégia é como uma técnica de sala de cirurgia. Os dados são esterilizados antes de poderem entrar na sala de cirurgia. Tudo que estiver nessa sala será considerado seguro. A principal decisão de projeto é determinar o que colocar na sala de cirurgia, o que impedir que entre e onde colocar as portas - quais rotinas são consideradas internas na zona de segurança, quais são externas e quais desinfetam os dados. Normalmente, a maneira mais fácil de fazer isso é desinfetando os dados externos quando eles chegam, mas os dados frequentemente precisam ser desinfetados em mais de um nível; portanto, às vezes são necessários vários níveis de esterilização.

Converta os dados de entrada para o tipo correto no momento da entrada – Normalmente, a entrada chega na forma de uma *string* ou de um número. Algumas vezes, o valor será mapeado para um tipo booleano, como "sim" ou "não". Outras vezes, o valor será mapeado para um tipo enumerado, como *Color_Red, Color_Green* e *Color_Blue.* Transportar dados de tipo questionável por qualquer período de tempo em um programa aumenta a complexidade e a chance de que alguém possa danificar seu programa introduzindo uma cor como "Sim". Converta os dados de entrada para a forma correta, assim que for possível, após eles serem introduzidos.

Relacionamento entre barricadas e assertivas

O uso de barricadas torna clara e definida a distinção entre assertivas e tratamento de erro. As rotinas que estão fora da barricada devem usar tratamento de erro, pois não é seguro fazer quaisquer suposições a respeito dos dados. As rotinas que estão dentro da barricada devem usar assertivas, pois os dados passados para elas supostamente foram desinfetados antes de terem passado pela barricada. Se uma das rotinas de dentro da barricada detectar dados incorretos, tratar-se-á de um erro no programa e não simplesmente nos dados.

O uso de barricadas também ilustra o valor da decisão no nível arquitetônico sobre como tratar de erros. Determinar qual código fica dentro e qual fica fora da barricada é uma decisão em nível de arquitetura.

6.6 Auxiliares de depuração

Outro aspecto importante da programação defensiva é o uso de auxiliares de depuração, que podem ser aliados poderosos na rápida detecção de erros.

Não aplique restrições de produção automaticamente na versão de desenvolvimento

A percepção de um programador comum é a suposição de que as limitações do *software* de produção se aplicam à versão de desenvolvimento. A versão de produção precisa ser executada rapidamente; a versão de desenvolvimento pode ser executada lentamente. A versão de produção precisa ser mesquinha com os recursos; a versão de desenvolvimento pode usar recursos de forma extravagante. A versão de produção não

deve expor operações perigosas para o usuário; a versão de desenvolvimento pode ter operações extras, as quais você pode usar sem uma rede de segurança.

Um programa no qual trabalhei fazia uso extensivo de uma lista quadruplamente encadeada. O código da lista encadeada era propenso a erros e a lista encadeada tendia a se corromper. Eu acrescentei uma opção de menu para verificar a integridade da lista encadeada.

No modo de depuração, o Microsoft Word contém código no *loop* inativo que verifica a integridade do objeto *Documento* a cada poucos segundos. Isso ajuda a detectar rapidamente qualquer corrupção de dados e facilita o diagnóstico de erro.

Durante o desenvolvimento, esteja preparado para trocar velocidade e utilização de recursos por ferramentas internas que possam ajudar o trabalho a fluir mais suavemente.

Introduza logo os auxiliares de depuração

Quanto antes você introduzir auxiliares de depuração, mais eles serão úteis. Normalmente, você não se dará ao trabalho de escrever um auxiliar de depuração até ter sido perturbado várias vezes por um problema. Entretanto, se você escrever o auxiliar após a primeira vez ou usar outro, de um projeto anterior, ele ajudará ao longo de todo o projeto.

Use programação ofensiva

Os casos excepcionais devem ser tratados de uma maneira que os tornem óbvios durante o desenvolvimento e recuperáveis quando o código de produção estiver sendo executado. Michael Howard e David LeBlanc se referem a essa estratégia conto "programação ofensiva" (Howard e LeBlanc 2003).

Suponha que você tenha uma instrução *case,* a qual espera tratar apenas cinco tipos de eventos. Durante o desenvolvimento, o caso-padrão deve ser usado para gerar um alerta, dizendo: "Ei! Existe outro caso aqui! Corrija o programa!". Durante a produção, entretanto, o caso-padrão deve fazer algo mais elegante, como gravar uma mensagem em um arquivo de *log* de erros.

Aqui estão algumas maneiras pelas quais você pode programar ofensivamente:

- Certifique-se de que *assertivas* cancelem o programa. Não permita que os programadores habituem-se a apenas pressionar a tecla Enter para escapar de um problema conhecido. Torne o problema incômodo o bastante para que ele seja corrigido.
- Preencha completamente toda memória alocada, para que você possa detectar erros de alocação de memória.
- Preencha completamente todos os arquivos ou fluxos alocados, para descobrir todos os erros de formato de arquivo.
- Certifique-se de que o código na cláusula *default* ou *else* de cada instrução *case* falhe totalmente (cancele o programa) ou seja impossível de desprezar.
- Preencha um objeto com dados sem valor, imediatamente antes de ele ser excluído.
- Configure o programa de forma a enviar para você mesmo arquivos de *log* de erros, por *e-mail,* para que possa ver os tipos de erros que estão ocorrendo no *software* lançado, se isso for apropriado para o tipo de *software* que está desenvolvendo.

Às vezes, a melhor defesa é um bom ataque. Aborte completamente durante o desenvolvimento para que você possa abortar pouco durante a produção.

Planeje remover os auxiliares de depuração

Se você está escrevendo código para seu próprio uso, é normal manter todo o código de depuração no programa. Porém, se estiver escrevendo código para uso comercial, a desvantagem no desempenho quanto a tamanho e velocidade pode ser proibitiva. Faça

planos para evitar o embaralhamento de código de depuração dentro e fora de um programa. Aqui estão várias maneiras de se fazer isso:

***Use ferramentas de controle de versão e ferramentas internas como* ant *e* make -** As ferramentas de controle de versão podem construir diferentes versões de um programa a partir dos mesmos arquivos-fonte. No modo de desenvolvimento, você pode configurar a ferramenta interna para incluir todo o código de depuração. No modo de produção, você pode configurá-la para excluir todo código de depuração que não desejar na versão comercial,

Use um pré-processador interno - Se seu ambiente de programação possui um pré-processador - como acontece com a linguagem C++, por exemplo –, você pode incluir ou excluir código de depuração ao toque de uma chave de compilador. Você pode usar o pré-processador diretamente ou escrevendo uma macro que trabalhe com definições de pré-processador. Aqui está um exemplo de escrita de código usando o pré-processador diretamente:

```
    Exemplo em C++ de uso direto do pré-processador para controlar código de depuração


    #define DEBUG
    ...

    #if defined(DEBUG)
      // código de depuração
      ...

    #endif
```

Esse tema apresenta diversas variações. Em vez de apenas definir *DEBUG,* você pode atribuir um valor a ele e depois testar esse valor, em vez de testar se *DEBUG* está definido. Desse modo, você pode distinguir entre diferentes níveis de código de depuração. Você poderia ter algum código de depuração que desejasse em seu programa o tempo todo, de modo que o circundaria por meio de uma instrução como *#if DEBUG > 0.* Outro código de depuração poderia ser apenas para propósitos específicos; então, você poderia circundá-lo por meio de uma instrução como *#if DEBUG == POINTER_ERROR.* Em outros locais, talvez você quisesse configurar níveis de depuração; portanto, poderia ter instruções como *#if DEBUG > LEVEL_A.*

Se você não gosta de ter instruções *#ifdefined()* espalhadas por todo o seu código, pode escrever uma macro de pré-processador para executar a mesma tarefa. Aqui está um exemplo:

```
    Exemplo em C++ de uso de uma macro de pré-processador para controlar código de depuração

    #define DEBUG
    #if defined( DEBUG )
    #define DebugCode( code_fragment ) { code_fragment }
    #else
    #define DebugCode( code_fragment )
    #endif
    ...

    DebugCode(
       statement 1;
       statement 2;
       statement n;
```

```
);
...
```

Assim como no primeiro exemplo de uso do pré-processador, essa técnica pode ser alterada de várias maneiras que a tornam mais sofisticada, o que é preferível a incluir completamente todo o código de depuração ou excluí-lo completamente.

Escreva seu próprio pré-processador Se uma linguagem não inclui um pré-processador, é muito fácil escrever um para incluir e excluir código de depuração. Estabeleça uma convenção para designar código de depuração e escreva seu pré-compilador seguindo essa convenção. Por exemplo, em Java, você poderia escrever um pré-compilador para responder às palavras-chave *//#BEGIN* DEBUG e *//#END DEBUG.* Escreva um *script* para chamar o pré-processador e, em seguida, compile o código processado. Você economizará tempo a longo prazo e não compilará erroneamente o código não processado.

6.7 Determinando o quanto de programação defensiva deve ser deixada no código de produção

Um dos paradoxos da programação defensiva é que, durante o desenvolvimento, você gostaria que um erro fosse perceptível - você preferiria irritar-se com ele do que correr o risco de não percebê-lo. No entanto, durante a produção, você preferiria que o erro fosse o mais discreto possível, para que o programa se recuperasse ou falhasse elegantemente. Aqui estão algumas diretrizes para decidir quais ferramentas de programação defensiva devem ser mantidas em seu código de produção e quais devem ser excluídas:

Mantenha o código que identifica erros importantes - Decida quais áreas do programa podem permitir a existência de erros não-detectados e quais áreas não podem. Por exemplo, se você estivesse escrevendo um programa de planilha eletrônica, poderia permitir a existência de erros não-detectados na área de atualização da tela do programa, porque a maior penalidade de um erro desse tipo é apenas uma tela desajeitada. O que você não poderia permitir seria a existência de erros não-detectados no mecanismo de cálculo, pois tais erros poderiam gerar resultados sutilmente incorretos na planilha eletrônica de alguém. A maioria dos usuários toleraria uma tela desajeitada, mas não admitiria cálculos de imposto incorretos e uma auditoria da Receita Federal.

Remova o código que identifica erros simples Se um erro tem consequências realmente simples, remova o código que o verifica. No exemplo anterior, você poderia remover o código que verifica a atualização de tela da planilha eletrônica. “Remover” não significa remover o código fisicamente. Significa usar controle de versão, chaves de pré-compilador ou alguma outra técnica para compilar o programa sem esse código em particular. Não havendo problema de espaço, você até pode manter o código de verificação de erro, fazendo-o registrar mensagens reservadamente, em um arquivo de *log* de erros.

Remova o código que resulta em falhas sérias - Conforme já foi mencionado, durante o desenvolvimento, quando seu programa detecta um erro, você deseja que esse erro seja o mais perceptível possível, para poder corrigi-lo. Na maior parte das vezes, a melhor maneira de atingir esse objetivo é fazer o programa imprimir uma mensagem de depuração e terminar ao detectar um erro. Isso é útil mesmo para erros

secundários.

Durante a produção, seus usuários precisam de uma oportunidade para salvar seus trabalhos antes que o programa termine; provavelmente, os usuários estarão dispostos a tolerar umas poucas anomalias em troca de manterem o programa em funcionamento por um período longo o suficiente para fazerem isso. Os usuários não gostam de nada que resulte na perda de seus trabalhos, independentemente do quanto isso ajude na depuração e, em última análise, melhore a qualidade do programa. Se seu programa contém código de depuração que poderia causar perda de dados, retire-o da versão de produção.

Mantenha o código que ajuda o programa a terminar elegantemente - Caso seu programa contenha código de depuração que detecta erros potencialmente fatais, mantenha o código que permite a ele terminar elegantemente. Na Mars Pathfinder, por exemplo, os engenheiros mantiveram, por decisão de projeto, parte do código de depuração. Um erro ocorreu após a Pathfinder ter pousado. Usando os auxiliares de depuração que tinham sido mantidos, os engenheiros da JPL foram capazes de diagnosticar o problema e carregar o código revisado na Pathfinder, e esta completou sua missão perfeitamente (março de 1999).

Registre os erros em* log *para seu pessoal de suporte técnico - Considere o fato de manter os auxiliares de depuração no código de produção, mas alterando seu comportamento, para que ele se torne adequado à versão de produção. Se você tiver carregado seu código com assertivas que interrompem o programa durante o desenvolvimento, pode considerar a possibilidade de alterar rotina de assertiva para registrar mensagens em um arquivo de *log* durante a produção, em vez de eliminá-las completamente.

Certifique-se de que as mensagens de erro deixadas sejam amigáveis - Se você mantiver mensagens de erro internas no programa, verifique se elas estão em uma linguagem amigável para o usuário. Em um de meus primeiros programas, recebi um telefonema de uma usuária relatando que tinha recebido uma mensagem que dizia: “Você obteve uma alocação de ponteiro inválida, Bafo de Onça!”. Felizmente para mim, a tal usuária tinha senso de humor. Uma estratégia comum e eficaz é notificar o usuário de um “erro interno” e listar um endereço de *e-mail* ou número de telefone que ele possa usar para relatar o problema.

6.8 Sendo defensivo quanto à programação defensiva

Programação defensiva demais cria seus próprios problemas. Se você verificar os dados passados como parâmetros de cada maneira concebível, em cada local concebível, seu programa será enorme e lento. E, o que é pior, o código adicional necessário para a programação defensiva aumenta a complexidade do *software.* O código instalado para a programação defensiva não é imune a defeitos e a probabilidade de você encontrar um erro no código da programação defensiva é igual à de qualquer outro código - principalmente, se você não escrever o código com muito critério. Pense a respeito de onde você precisa ser defensivo e estabeleça suas prioridades de programação defensiva adequadamente.

Lista de verificação: programação defensiva
Geral

- A rotina se protege de dados de entrada inválidos?
- Você usou assertivas para documentar suposições, incluindo pré-condições e pós-condições?
- As assertivas foram usadas apenas para documentar condições que nunca devem ocorrer?
- A arquitetura ou o projeto de alto nível especificam um conjunto determinado de técnicas de tratamento de erro?
- A arquitetura ou o projeto de alto nível especificam se o tratamento de erro deve privilegiar a robustez ou a correção?
- Foram criadas barricadas para conter o efeito danoso de erros e reduzir o volume de código que precisa se preocupar com processamento de erro?
- Foram usados auxiliares de depuração no código?
- Os auxiliares de depuração foram instalados de tal maneira que possam ser ativados ou desativados sem muito tumulto?
- O volume de código de programação defensiva é apropriado - nem muito, nem pouco?
- Você usou técnicas de programação ofensiva para tornar os erros difíceis de ignorar durante o desenvolvimento?

Exceções

- Seu projeto definiu uma estratégia padronizada para o tratamento de exceções?
- Você considerou alternativas para o uso de uma exceção?
- Sempre que possível, o erro é tratado de forma local, em vez de lançar uma exceção, não-local?
- O código evita o lançamento de exceções em construtores e destrutores?
- Todas as exceções estão nos níveis de abstração apropriados para as rotinas que as lançam?
- Cada exceção inclui todas as informações de base relevantes?
- O código está livre de blocos *catch* vazios? (Ou, então, se um bloco *catch* vazio é realmente apropriado, ele está documentado?)

Problemas de segurança

- O código que verifica dados de entrada inválidos verifica tentativas de estouros de *buffer,* injeção de SQL, injeção de HTML, estouros de inteiro e outras entradas mal-intencionadas?
- Todos os códigos de retorno de erro são verificados?
- Todas as exceções são capturadas?
- As mensagens de erro evitam o fornecimento de informações que ajudariam um invasor a penetrar no sistema?

Pontos-chave

❖ O código de produção deve tratar de erros de uma maneira mais sofisticada do que afirma a estratégia "Entra lixo, sai lixo".

❖ As técnicas de programação defensiva tornam os erros mais fáceis de encontrar, mais fáceis de corrigir e menos danosos para o código de produção.

❖ As assertivas podem ajudar a detectar erros antecipadamente, em especial em sistemas grandes, sistemas de alta confiabilidade e bases de código que mudam com muita frequência.

❖ A determinação sobre como tratar de entradas incorretas é uma decisão importante do tratamento de erro e também do projeto de alto nível.

❖ As exceções fornecem uma maneira de tratamento de erros que opera em uma dimensão diferente do fluxo normal do código. Quando usadas com cuidado, elas são um acréscimo valioso na caixa de ferramentas intelectual do programador e devem ser

ponderadas com relação às outras técnicas de processamento de erro.

❖ As restrições que se aplicam ao sistema de produção não se aplicam necessariamente à versão de desenvolvimento. Você pode usar isso em seu beneficio, adicionando na versão de desenvolvimento um código que o ajude a descobrir erros rapidamente.