Código Limpo:

Habilidades Práticas do Agile Software

•••

Capítulo 7: Tratamento de Erro

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

Tratamento de erros: erros podem acontecer

Erros podem ocorrer, seja por entradas inválidas, por estados inválidos, por falhas em dispositivos ou por outros motivos.

Programadores devem garantir que, independentemente desses casos, o software faça o que ele precisa fazer.

Os softwares devem ser robustos e, para atingir tal ponto, a atividade de tratar erros é inevitável.

Mas robustez não deve custar as boas práticas de código limpo.

Tratamento de erros: robustez e código limpo

Tratamento de erro é uma tarefa necessária, mas há muitos códigos-fontes tão recheados com ele que é difícil compreender o que eles de fato fazem.

“Esse recurso é importante, mas se obscurecer a lógica, está errado.”

A raiz desse problema já foi observado anteriormente: o tratamento de erro polui o código quando não há uma devida separação de responsabilidades.

Use exceções e não códigos de erro

Por essa razão, não devemos retornar status de execução dos métodos.

Essa é a maneira utilizada em linguagens que não suportam exceções (como C) e deve ser restrita a elas.

O principal motivo para se evitar essa prática é que ela implica na existência de estruturas condicionais para tratar o erro. Isso força a mistura com a lógica.

Use exceções e não códigos de erro: código antes

O tratamento de erro usando status de erro tornou o código abaixo poluído e dificulta a compreensão dele:

```
public class StudentMenu {
   public void saveStudentMenu() {
       Student student = getStudentFromForm();
       if (student != null) {
           int savingStatus = studentManager.saveStudent(student);

           if (savingStatus == 0)
               System.out.println("Student successfully saved!");
           else if (savingStatus == -1000)
               System.err.println("Such student has already been saved!");
           else if (savingStatus == -2000)
               System.err.println("Such student has invalid data!");
       }
       else System.err.println("Student data must be given!");
}
```

Use exceções e não códigos de erro: repassando erros

É ainda pior se o tratamento estiver em métodos mais acima. Na pior das hipóteses, cada um entre eles terá estruturas semelhantes apenas para repassar o retorno

```
public int saveStudent(Student student) {
   if (studentDAO.findOneByName(student.getName()) != null)
       return -1000;
   return switch (studentDAO.save(student)) {
       case 0 -> 0;
       case -1000 -> -2000; // Se o DAO retornar -1000, significa que o student tem dados inválidos
       default -> -10000;
   };
}
```

Esse método pertence a classe StudentManager (usada anteriormente) e ele é obrigado a repassar códigos do DAO (que faz acesso ao banco de dados) acima.

Use exceções e não códigos de erro: as desvantagens

Além de obscuro, o código também se torna menos flexível a mudanças externas, tais como mudar o status de erro que será recebido.

Fora isso, tratar erros assim é uma atividade propensa a ser esquecida. Os erros podem passar despercebidos e surtirem efeito em contextos aparentemente desconexos.

Em vista a esses problemas, há um recurso mais atual que deve ser utilizado em linguagens mais modernas: Exceções.

Use exceções e não códigos de erro: alternativa atual

Exceções, permitem separar o que é lógica do que é erro.

Uma vez lançadas, seu gerenciamento – exceto o tratamento – deixa de ser responsabilidade do desenvolvedor. Não é necessário repetir isso a cada método.

Assim, pode-se facilmente separar o tratamento da lógica ao delegá-lo a métodos mais acima.

Adicionalmente, se elas não forem tratadas em nenhum momento após serem lançadas, elas irão encerrar o processo. Logo, erros não passarão despercebidos.

Use exceções e não códigos de erro: código depois

O código principal do exemplo anterior fica mais enxuto com o uso de exceções:

```
public void saveStudentMenu() {
    try {
        Student student = getStudentFromForm();
        studentManager.saveStudent(student);
        System.out.println("Student successfully saved!");
    }
    catch (Exception e) {
        System.err.println("Could not registry the student. Cause: " + e.getMessage());
    }
}
```

Apesar de conter uma estrutura try...catch...finally, a lógica principal se torna mais evidente e seu objetivo é melhor compreendido ao lê-la.

Use exceções e não códigos de erro: somente o necessário

Mesmo o método saveStudent() da classe StudentManager se torna mais limpo e compreensível:

```
public void saveStudent(Student student) {
   if (studentDAO.findOneByName(student.getName()) != null)
       throw new EntityAlreadyExistsException("Student name must be unique!");
   studentDAO.save(student);
}
```

Nota-se que ele não necessita repassar as exceções lançadas pelo StudentDAO e contém somente a lógica de sua responsabilidade.

Não use exceções verificadas

Exceções verificadas são aquelas que exigem serem tratadas. Do contrário, deve-se declará-las na assinatura do método.

```
public void methodThatDoesNotHandleTheException() throws SomeCheckedException {
    . . .
}
```

A ideia original de quando foram inseridas em Java – serem explicitamente declaradas na assinatura do método – parece boa, mas elas possuem um custo alto.

Não use exceções verificadas: o preço por usar

O preço é ferir o princípio Open-Closed, que diz que um módulo deve estar aberto a extensão de comportamento, mas fechado para alterações.

Todos os métodos entre quem lança a exceção e quem possui o tratamento devem possuí-la declarada na assinatura.

Se, por exemplo, essa exceção deixar de ser lançada em uma nova implementação todas essas assinaturas deverão ser adaptadas.

Em outras palavras, dezenas de métodos alheios a mudança deverão mudar também e serem reconstruídos.

Não use exceções verificadas: conclusão

Python, Ruby, C++, C# etc. não verificam exceções, mas é possível produzir sistemas de software robustos com essas linguagens.

Há situações em que se deseja e é vital que a exceção seja tratada a todo custo. São nessas situações em que exceções verificadas são justificáveis.

Seu uso deve, então, ser reservado somente a essas situações e sempre de maneira bem pensada.

Tratando exceções: por onde começar?

Começar pela estrutura *try...catch...finally* primeiro é uma prática que auxilia na criação de tratamentos de erro eficientes e limpos.

Essa estrutura possui um escopo próprio (o que nasce ali, só é visto ali). Graças a isso, podemos ver o *try* como uma transação – ou tudo funciona ou nada acontece.

Ao catch, então, é atribuída a responsabilidade de manter o programa em um estado consistente, independentemente do erro que ocorra.

Tratando exceções: aplique o TDD

Quando o código puder lançar exceções, desenvolva, primeiramente, os catches para os possíveis erros.

Para auxiliar a detecção dos erros, pode-se utilizar de TDD. Testes podem ser criados para forçar as exceções e ajudar a escolher o melhor tratamento para cada uma.

Isso ajuda a construir a API do programa, ou seja, a definir o que o usuário do código deve esperar.

Outra vantagem é que, ao se concentrar nos problemas primeiro, pode-se concentrar apenas na lógica depois. Isso permite separar um do outro mais facilmente.

Crie exceções mais genéricas: pense na captura delas

Às vezes, faz-se necessário a criação de exceções próprias.

Nessa situação, o que deve ser considerado, apesar de todas as diferentes formas de classificar exceções, é o como ela será capturada.

É comum que, em um contexto fechado (como o de um método ou de uma classe), o tratamento seja o mesmo para diferentes erros.

Por consequência, exceções mais genéricas – que englobam diferentes tipos de erros – se provam mais úteis ao contribuir à redução de código repetido.

Crie exceções mais genéricas: código antes

A API abaixo, por exemplo, define diferentes exceções para diferentes erros que podem ocorrer

```
public void openPort() {
   ACMEPort port = new ACMEPort(12);
   try {
       port.open();
   } catch (DeviceResponseException e) {
       System.err.println("Couldn't open the port. Cause" + e.getMessage());
   } catch (UnlockException e) {
       System.err.println("Couldn't open the port. Cause" + e.getMessage());
   } catch (GMXError e) {
       System.err.println("Couldn't open the port. Cause" + e.getMessage());
   }
}
```

A duplicação de código é evidente: o mesmo tratamento está presente em cada catch.

Crie exceções mais genéricas: código depois

Tendo uma exceção mais ampla, esse tratamento de erro pode ser reduzido a:

```
try {
   port.open();
}
catch (PortDeviceFailureException e) {
   System.err.println("Unable to open the port. Cause " + e.getMessage());
}
```

A exceção *PortDeviceFailureException* foi criada com a intenção de englobar todos os possíveis erros que podem ocorrer na operação. Assim, foi possível remover as duplicações.

Crie exceções mais genéricas: quando não usar?

Em contrapartida, definir diversas exceções diferentes é útil quando se espera que cada erro diferente seja capturado e tratado de maneiras diferentes.

Isso pode ser útil, caso necessário, para prover mais liberdade – a quem for usar API – de tomar as decisões relacionadas ao tratamento.

Entretanto, como já mencionado, em um contexto fechado, definir apenas uma exceção é comumente o suficiente.

Assim, mantém-se os erros internos encapsulados e terceiros só saberão o necessário. Esse cenário tende a ser mais preferível.

Lidando com exceções de terceiros

Porém, a API mencionada no exemplo pertence a desenvolvedores externos. Portanto, não é possível alterá-las diretamente, adicionando e/ou removendo exceções.

Assim, quando se estiver usando API’s de terceiros, pode ser mais complexo evitar a dependência de múltiplas exceções.

Nesses casos, apesar do tratamento ser o mesmo, seria necessário adicionar um catch para cada exceção de todo modo.

Além disso, haveria um grande acoplamento do sistema às características particulares de um código que está alheio ao controle do desenvolvedor.

Wrappers: uma ótima solução

Uma excelente solução para esse desafio é o uso de *Wrappers*. Essas são classes criadas para empacotar outras classes, no caso, de terceiros.

Empacotar classes de terceiros é uma técnica que permite traduzir as funcionalidades dessas API’s para termos conhecidos e sob controle do desenvolvedor.

Isso reduz o acoplamento com o código de terceiros e aumenta o encapsulamento.

Portanto, trocas de dependências tornam-se menos prejudiciais e mudanças colaterais desnecessárias são evitadas.

Wrappers: encapsulando o código

O exemplo a seguir demonstra a aplicação desse conceito na API do exemplo anterior

```
public class LocalPort {
   private final ACMEPort innerPort;

   public LocalPort(int portNumber) { innerPort = new ACMEPort(portNumber); }

   public void open() {
       try { innerPort.open(); }
       catch (DeviceResponseException e) { throw new PortDeviceFailureException(e); }
       catch (UnlockException e) { throw new PortDeviceFailureException(e); }
       catch (GMXError e) { throw new PortDeviceFailureException(e); }
   }
}
```

Forneça exceções com contexto

É uma prática importante fornecer contexto e descrição às exceções quando lançá-las. Isso facilita uma busca eficiente pelos erros e a manutenção.

O Stack Trace do Java lista o caminho dos métodos por onde a exceção passou. Todavia, ele não é capaz de dizer a razão pela qual o método falhou.

Ao lançar uma exceção, devem ser incluídas informações sobre qual operação falhou e o porquê ela falhou.

Adicionalmente, exceções também são classes, então as regras de nomeação também são aplicáveis ao criá-las.

Defina o fluxo normal

Utilizar TDD, como citado anteriormente, eleva a detecção de erros ao máximo. As estruturas try...catch...finally foram montadas de forma a capturar todos os possíveis erros.

Nesses casos, a operação contida no try é cancelada e a execução prossegue em um catch para o erro ser tratado de acordo.

Porém, há casos em que esse processo – de cancelar a operação – não é a solução mais adequada.

Defina o fluxo normal: código antes

O exemplo abaixo trata-se de um código que busca as despesas por refeições de um funcionário e retorna o total:

```
public double getMealExpensesByEmployee(Employee employee) {
    try {
        MealExpenses expenses = expensesDAO.findMealExpensesByEmployeeId(employee.getId());
        return expenses.getTotal();
    }
    catch (EntityNotFoundException e) {
        return getMealPerDiem();  // quantia padrão para ajuda de custos
    }
}
```

Caso não sejam encontradas despesas no repositório, o funcionário recebe uma quantia padrão para ajuda de custos - getMealPerDiem().

Defina o fluxo normal: o problema

O tratamento posto no catch no código anterior incluiu uma regra de negócio, uma parte da lógica principal.

Isso pode facilmente obscurecer a lógica do método, pois mistura responsabilidades. Esse código separa coisas que deveriam estar juntas.

O catch deve ser responsável somente por manter o estado consistente e/ou reportar um erro.

Para começar a eliminar esse problema, é importante sempre definir bem o fluxo normal do programa.

Defina o fluxo normal: special case

O padrão Special Case pode ser utilizado para eliminar esse problema. Esse padrão se apoia fortemente nos conceitos de polimorfismo e encapsulamento.

Defina uma classe abstrata ou interface e assine o método para retorná-la. Em seguida, crie a implementação padrão – aquela que contém a lógica do *sunny day*.

Adicione outras implementações, que contenham as lógicas dos *rainy days* (as que seriam colocadas no catch). Retorne-as de acordo com o que acontecer de errado.

O método chamador não saberá a diferença da implementação, entretanto, ele tem a certeza de que recebeu o que precisa corretamente.

Defina o fluxo normal: criado o special case

Reformulando a classe MealExpenses de acordo com esse padrão, tem-se o resultado:

```
interface MealExpenses {
    double getTotal();
}

public class SunnyDayMealExpenseImpl implements MealExpenses {
   @Override public double getTotal() {
       // Implementação do sunny day
   }
}

public class RainyDayMealExpenseImpl implements MealExpenses {
   @Override public double getTotal() {
       // Implementação do rainy day - contém o valor padrão para ajuda de custos
   }
}
```

Defina o fluxo normal: método refatorado

Dessa forma, pode-se reduzir o método getMealExpensesByEmployee() para o que ele realmente faz: recuperar as despesas e retornar o total.

```
public double getMealExpensesByEmployee(Employee employee) {
   MealExpenses expenses = expensesDAO.findMealExpensesByEmployeeId(employee.getId());
   return expenses.getTotal();
}
```

O código fica mais limpo e não há misturas de responsabilidades. Nesse caso, inclusive, foi possível remover até mesmo o *try…catch…finally*.

O velho conhecido: NullPointerException

NullPointerException é, sem dúvidas, umas das exceções mais recorrentes durante o desenvolvimento.

Quando ocorrem, possivelmente é devido à falta de verificação por *null*. Isso é compreensível.

Essa atividade é muitas vezes incidental e facilmente esquecida. Quando não são, estão presentes a cada uma ou duas linhas de código. O resultado é claro: código obscuro.

Mas basta esquecer uma delas para o cliente receber um *NullPointerException*.

O velho conhecido: e se...?

Voltando ao método *saveStudent()*, o que aconteceria na execução do código abaixo se *student* fosse null?

```
public void saveStudent(Student student) {
    if (studentDAO.findOneByName(student.getName()) != null)
        throw new EntityAlreadyExistsException("Student name must be unique!");
    studentDAO.save(student);
}
```

Ocorreria um *NullPointerException* com uma mensagem genérica e que precisaria ser tratada em métodos acima.

O velho conhecido: driblando o problema

Que práticas, então, podem ser seguidas para evitar essa situação? Não retornar *null* dos métodos é um começo.

Optional do Java é uma excelente alternativa. Ainda é necessário uma verificação, claro, mas além de ser mais declarativa e agradável, ela é obrigatória.

A depender do contexto, o Special Case também é uma boa opção.

Quando trabalhando com *Collections* ou *Arrays*, a melhor alternativa ao *null* é retornar uma coleção vazia ou um array vazio. Se estiverem vazios, apenas não haverá iteração.

Um velho conhecido: não passe *null*

De forma análoga, não passe *null* como argumento a menos que a API sendo usada o exija.

Ainda nesse contexto, ao criar os próprios métodos, é importante lembrar de "sanitizar" as entradas. Em outras palavras: verificar por valores inválidos, como *null*.

Kotlin e Typescript são exemplos de linguagens que não permitem que a passagem de argumentos nulos, a menos que seja explicitamente permitido pelo desenvolvedor.

Geralmente, não há muito o que fazer nessas verificações. Uma solução comum seria lançar uma exceção própria, que pode ser mais descritiva e contextualizada que uma *NullPointerException*.

Conclusão

Como erros são inevitáveis, sistemas de software devem ser robustos e estarem preparados para eles. Mas o tratamento de erro não deve obscurecer o código.

O bloco try deve conter a lógica e o catch apenas manter a consistência do programa.

TDD auxilia na construção das estruturas *try…catch…finally*.

Wrappers são úteis quando lidar com API’s de terceiros.

Fornecer contexto às exceções facilita a manutenção.

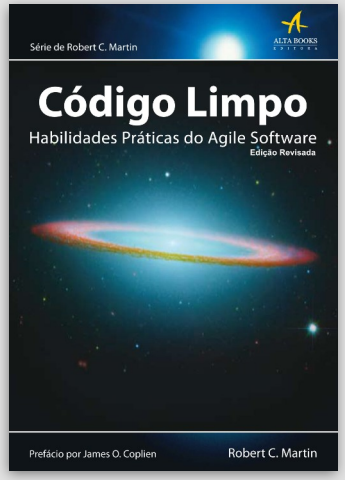
Não retornar ou passar *null* como argumento ajuda a evitar *NullPointerExceptions*.

Sobre o autor

Robert C. “Uncle Bob” Martin é desenvolvedor e consultor de software desde 1990. Ele é o fundador e o presidente da Object Mentor, Inc., uma equipe de consultores experientes que orientam seus clientes no mundo todo em C++, Java, C#, Ruby, OO, Padrões de Projeto, UML, Metodologias Agile e eXtreme Programming.

Este conjunto de slides foi elaborado a partir da obra:

MARTIN, Robert. **Código Limpo:** Habilidades Práticas do Agile Software. 1. ed. Rio de Janeiro: Alta Books, 2009.



A equipe

Lucas Almeida, Autor

Aluno e bolsista do PET/ADS

Github

Lucas Oliveira, Revisor

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

LinkedIn