

Listagem 6-5 (continuação)

Classe shape procedimental

```
        else if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle)shape;
            return r.height * r.width;
        }
        else if (shape instanceof Circle) {
            Circle c = (Circle)shape;
            return PI * c.radius * c.radius;
        }
        throw new NoSuchShapeException();
    }
```

Programadores de orientação a objeto talvez torçam o nariz e reclamem que isso é procedural—e estão certos. Mas nem sempre. Imagine o que aconteceria se adicionássemos uma função `perimeter()` à `Geometry`. As classes `shape` não seriam afetadas! Assim como quaisquer outras classes que dependessem delas!

Por outro lado, se adicionarmos uma nova classe shape, teremos de alterar todas as funções em Geometry. Leia essa frase novamente. Note que as duas situações são completamente opostas.

Agora, considere uma solução orientada a objeto na Listagem 6.6. O método `área()` é *polifórmico*. Não é necessária a classe `Geometry`. Portanto, se eu adicionar uma nova forma, nenhuma das funções existentes serão afetadas, mas se eu adicionar uma nova função, todas as classes `shape` deverão ser alteradas¹.

Listagem 6-6

Classes shape polifórmicas

```
public class Square implements Shape {  
    private Point topLeft;  
    private double side;  
  
    public double area() {  
        return side*side;  
    }  
  
}  
  
public class Rectangle implements Shape {  
    private Point topLeft;  
    private double height;  
    private double width;  
  
    public double area() {  
        return height * width;  
    }  
}
```

1. Desenvolvedores orientados a objeto experiente conhecem outras maneiras de se contornar isso. O padrão Visitor, com dual-dispatch, por exemplo.

Listagem 6-6 (continuação)**Classes shape polifôrmicas**

```
public class Circle implements Shape {
    private Point center;
    private double radius;
    public final double PI = 3.141592653589793;

    public double area() {
        return PI * radius * radius;
    }
}
```

Novamente, vemos que a natureza complementar dessas duas definições: elas são praticamente opostas! Isso expõe a dicotomia fundamental entre objetos e estruturas de dados:

O código procedural (usado em estruturas de dados) facilita a adição de novas funções sem precisar alterar as estruturas de dados existentes. O código orientado a objeto (OO), por outro lado, facilita a adição de novas classes sem precisar alterar as funções existentes.

O inverso também é verdade:

O código procedural dificulta a adição de novas estruturas de dados, pois todas as funções teriam de ser alteradas. O código OO dificulta a adição de novas funções, pois todas as classes teriam de ser alteradas.

Portanto, o que é difícil para a OO é fácil para o procedural, e o que é difícil para o procedural é fácil para a OO!

Em qualquer sistema complexo haverá vezes nas quais desejaremos adicionar novos tipos de dados em vez de novas funções. Para esses casos, objetos e OO são mais apropriados. Por outro lado, também haverá vezes nas quais desejaremos adicionar novas funções em vez de tipos de dados. Neste caso, estruturas de dados e código procedural são mais adequados.

Programadores experientes sabem que a ideia de que tudo é um objeto é *um mito*. Às vezes, você realmente *deseja* estruturas de dados simples com procedimentos operando nelas.

A lei de Demeter

Há uma nova heurística muito conhecida chamada Lei de Demeter²: um módulo não deve enxergar o interior dos objetos que ele manipula. Como vimos na seção anterior, os objetos escondem seus dados e expõem as operações. Isso significa que um objeto não deve expor sua estrutura interna por meio dos métodos acessores, pois isso seria expor, e não ocultar, sua estrutura interna.

Mais precisamente, a Lei de Demeter diz que um método *f* de uma classe *C* só deve chamar os métodos de:

- *C*
- Um objeto criado por *f*
- Um objeto passado como parâmetro para *f*
- Um objeto dentro de uma instância da variável *C*

² http://en.wikipedia.org/wiki/Law_of_Demeter

O método *não* deve chamar os métodos em objetos retornados por qualquer outra das funções permitidas. Em outras palavras, fale apenas com conhecidos, não com estranhos.

O código³ seguinte parece violar a Lei de Demeter (entre outras coisas), pois ele chama a função `getScratchDir()` no valor retornado de `getOptions()` e, então, chama `getAbsolutePath()` no valor retornado de `getScratchDir()`.

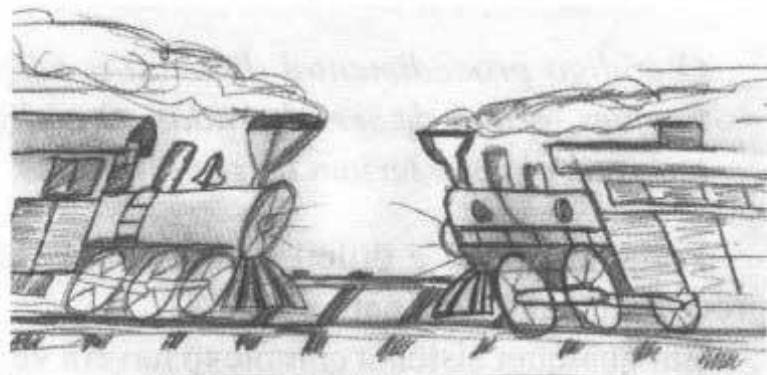
```
final String outputDir = ctxt.getOptions().getScratchDir().  
getAbsolutePath();
```

Carrinhos de trem

Esse tipo de código costuma ser chamador de *carrinho de trem*, pois parece com um monte de carrinhos de trem acoplados. Cadeias de chamadas como essa geralmente são consideradas descuidadas e devem ser evitadas [G36]. Na maioria das vezes é melhor dividi-las assim:

```
Options opts = ctxt.getOptions();  
File scratchDir = opts.getScratchDir();  
final String outputDir = scratchDir.getAbsolutePath();
```

Esses dois pedaços de código violam a Lei de Demeter? Certamente módulo que os contém sabe que o objeto `ctxt` possui opções (*options*), que contêm um diretório de rascunho (*scratchDir*), que tem um caminho absoluto (*AbsolutePath*). É muito conhecimento para uma função saber. A função de chamada sabe como navegar por muitos objetos diferentes.



Se isso é uma violação da Lei de Demeter depende se `ctxt`, `Options` e `ScracthDir` são ou não objetos ou estruturas de dados. Se forem objetos, então sua estrutura interna deveria estar oculta ao invés de exposta, portanto o conhecimento de seu interior é uma violação clara da lei. Por outro lado, se forem apenas estruturas de dados sem atividades, então eles naturalmente expõem suas estruturas internas, portanto aqui não se aplica a lei.

O uso de funções de acesso confunde essas questões. Se o código tiver sido escrito como abaixo, então provavelmente não estariamos perguntando sobre cumprimento ou não da lei.

```
final String outputDir = ctxt.options.scratchDir.getAbsolutePath();
```

Essa questão seria bem menos confusa se as estruturas de dados simplesmente tivessem variáveis públicas e nenhuma função, enquanto os objetos tivessem apenas variáveis privadas e funções públicas. Entretanto, há frameworks e padrões (*e.g.*, “beans”) que exigem que mesmo estruturas de dados simples tenham métodos acessores e de alteração.

³. Está em algum lugar no framework do Apache.

Híbridos

De vez em quando, essa confusão leva a estruturas híbridas ruins que são metade objeto e metade estrutura de dados. Elas possuem funções que fazem algo significativo, e também variáveis ou métodos de acesso e de alteração públicos que, para todos os efeitos, tornam públicas as variáveis privadas, incitando outras funções externas a usarem tais variáveis da forma como um programa procedural usaria uma estrutura de dados⁴.

Esses híbridos dificultam tanto a adição de novas funções como de novas estruturas de dados. Eles são a pior coisa em ambas as condições. Evite criá-los. Eles indicam um modelo confuso cujos autores não tinham certeza – ou pior, não sabiam – se precisavam se proteger de funções ou tipos.

Estruturas ocultas

E se ctxt, options e scratchDir forem objetos com ações reais? Então, como os objetos devem ocultar suas estruturas internas, não deveríamos ser capazes de navegar por eles. Então, como conseguiríamos o caminho absoluto de scratchDir ('diretório de rascunho')?

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

ou

```
ctx.getScratchDirectoryOption().getAbsolutePath()
```

A primeira opção poderia levar a uma abundância de métodos no objeto ctxt. A segunda presume que `getScratchDirectoryOption()` retorna uma estrutura de dados, e não um objeto. Nenhuma das opções parece boa.

Se ctxt for um objeto, devemos dizê-lo para fazer algo; não devemos perguntá-lo sobre sua estrutura interna. Por que queremos o caminho absoluto de scratchDir? O que faremos com ele? Considere o código, muitas linhas abaixo, do mesmo módulo:

```
String outFile = outputDir + "/" + className.replace('.', '/') +
    ".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

A mistura adicionada de diferentes níveis de detalhes [G34][G6] é um pouco confusa. Pontos, barras, extensão de arquivos e objetos `File` não devem ser misturados entre si e nem com o código que os circunda. Ignorando isso, entretanto, vimos que a intenção de obter o caminho absoluto do diretório de rascunho era para criar um arquivo de rascunho de um determinado nome. Então, e se disséssemos ao objeto ctxt para fazer isso?

```
BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);
```

That seems like a reasonable thing for an object to do! Isso permite ao ctxt esconder sua estrutura interna e evitar que a função atual viole a Lei de Demeter ao navegar por objetos os quais ela não deveria enxergar.

4. Às vezes chama-se de *Feature Envy* em [Refatoração].

Objetos de transferência de dados

A forma perfeita de uma estrutura de dados é uma classe com variáveis públicas e nenhuma função.

Às vezes, chama-se isso de objeto de transferência de dados, ou DTO (sigla em inglês). Os DTOs são estruturas muitos úteis, especialmente para se comunicar com bancos de dados ou analisar sintaticamente mensagens provenientes de sockets e assim por diante. Eles costumam se tornar os primeiros numa série de estágios de tradução que convertem dados brutos num banco de dados em objetos no código do aplicativo.

De alguma forma mais comum é o formulário “bean” exibido na Listagem 6.7. Os beans têm variáveis privadas manipuladas por métodos de escrita e leitura. O aparente encapsulamento dos beans parece fazer alguns puristas da OO sentirem-se melhores, mas geralmente não oferece vantagem alguma.

Listagem 6-7

address.java

```
public class Address {  
    private String street;  
    private String streetExtra;  
    private String city;  
    private String state;  
    private String zip;  
  
    public Address(String street, String streetExtra,  
                  String city, String state, String zip) {  
        this.street = street;  
        this.streetExtra = streetExtra;  
        this.city = city;  
        this.state = state;  
        this.zip = zip;  
    }  
  
    public String getStreet() {  
        return street;  
    }  
  
    public String getStreetExtra() {  
        return streetExtra;  
    }  
  
    public String getCity() {  
        return city;  
    }  
  
    public String getState() {  
        return state;  
    }  
  
    public String getZip() {  
        return zip;  
    }  
}
```

O Active Record

Os Active Records são formas especiais de DTOs. Eles são estruturas de dados com variáveis públicas (ou acessadas por Beans); mas eles tipicamente possuem métodos de navegação, como `save` (salvar) e `find` (buscar). Esses Active Records são traduções diretas das tabelas de bancos de dados ou de outras fontes de dados.

Infelizmente, costumamos encontrar desenvolvedores tentando tratar essas estruturas de dados como se fossem objetos, colocando métodos de regras de negócios neles. Isso é complicado, pois cria um híbrido entre uma estrutura de dados e um objeto.

A solução, é claro, é tratar o Record Active como uma estrutura de dados e criar objetos separados que contenham as regras de negócio e que ocultem seus dados internos (que provavelmente são apenas instâncias do Active Record).

Conclusão

Os objetos expõem as ações e ocultam os dados. Isso facilita a adição de novos tipos de objetos sem precisar modificar as ações existentes e dificulta a inclusão de novas atividades em objetos existentes. As estruturas de dados expõem os dados e não possuem ações significativas. Isso facilita a adição de novas ações às estruturas de dados existentes e dificulta a inclusão de novas estruturas de dados em funções existentes.

Em um dado sistema, às vezes, desejaremos flexibilidade para adicionar novos tipos de dados, e, portanto, optaremos por objetos. Em outras ocasiões, desejaremos querer flexibilidade para adicionar novas ações, e, portanto, optaremos tipos de dados e procedimentos.

Bons desenvolvedores de software entendem essas questões sem preconceito e selecionam a abordagem que melhor se aplica no momento.

Bibliografia

[Refatoração] *Refatoração - Aperfeiçoando o Projeto de Código Existente*, Martin Fowler et al., Addison-Wesley, 1999.

Tratamento de Erro

por Michael Feathers



Pode parecer estranho ter uma seção sobre tratamento de erro num livro sobre código limpo, mas essa tarefa é uma das quais todos temos de fazer quando programamos. A entrada pode estar errada e os dispositivos podem falhar. Em suma, as coisas podem dar errado, e quando isso ocorre, nós, como programadores, somos responsáveis por certificar que nosso código faça o que seja preciso fazer.

A conexão com um código limpo, entretanto, deve ser clara. O tratamento de erro domina completamente muitos códigos-fonte. Quando digo “domina”, não quero dizer que eles só fazem tratamento de erro, mas que é quase impossível ver o que o código faz devido a tantos tratamentos de erros espalhados. Esse recurso é importante, *mas se obscurecer a lógica, está errado*.

Neste capítulo ressaltarei uma série de técnicas e considerações que você pode usar para criar um código que seja limpo e robusto, que trate de erros com elegância e estilo.

Use exceções em vez de retornar códigos

Num passado longínquo havia muitas linguagens que não suportavam exceções. Nelas, as técnicas para tratar e informar erros era limitada. Ou você criava uma flag de erro ou retornava um código de erro que o chamador pudesse verificar. O código na Listagem 7.1 ilustra essas abordagens.

Listagem 7-1

DeviceController.java

```
public class DeviceController {
    ...
    public void sendShutDown() {
        DeviceHandle handle = getHandle(DEV1);
        // Check the state of the device
        if (handle != DeviceHandle.INVALID) {
            // Save the device status to the record field
            retrieveDeviceRecord(handle);
            // If not suspended, shut down
            if (record.getStatus() != DEVICE_SUSPENDED) {
                pauseDevice(handle);
                clearDeviceWorkQueue(handle);
                closeDevice(handle);
            } else {
                logger.log("Device suspended. Unable to shut down");
            }
        } else {
            logger.log("Invalid handle for: " + DEV1.toString());
        }
    }
    ...
}
```

O problema era que essas técnicas entupiam o chamador, que devia verificar erros imediatamente após a chamada. Infelizmente, facilmente se esqueciam de fazer isso. Por esse motivo, é melhor lançar uma exceção quando um erro for encontrado. O código de chamada fica mais limpo e sua lógica não fica ofuscada pelo tratamento de erro.

A Listagem 7.2 mostra o código depois de termos optado por lançar exceções em métodos que podem detectar erros.

Listagem 7-2

DeviceController.java (com exceções)

```
public class DeviceController {
    ...
    public void sendShutDown() {
        try {
            tryToShutDown();
        } catch (DeviceShutDownError e) {
            logger.log(e);
        }
    }
}
```

Listagem 7-2 (continuação):**DeviceController.java (com exceções)**

```

private void tryToShutdown() throws DeviceShutDownError {
    DeviceHandle handle = getHandle(DEV1);
    DeviceRecord record = retrieveDeviceRecord(handle);

    pauseDevice(handle);
    clearDeviceWorkQueue(handle);
    closeDevice(handle);
}

private DeviceHandle getHandle(DeviceID id) {
    ...
    throw new DeviceShutDownError("Invalid handle for: " + id.toString());
    ...
}

...
}

```

Observe como fica muito mais claro. Isso não é apenas uma questão estética. O código fica melhor porque duas preocupações que estavam intrincadas, o algoritmo para o desligamento do dispositivo e o tratamento de erro, agora estão separadas. Você pode pegar cada uma delas e estudá-las independentemente.

Crie primeiro sua estrutura try-catch-finally

Uma das coisas mais interessantes sobre exceções é que elas definem um escopo dentro de seu programa. Ao executar o código na parte try da estrutura try...catch...finally, você declara que aquela execução pode ser cancelada a qualquer momento e, então, continuar no catch.

De certa forma, os blocos try são como transações. Seu catch tem de deixar seu programa num estado consistente, não importa o que aconteça no try. Por essa razão, uma boa prática é começar com uma estrutura try...catch...finally quando for escrever um código que talvez lance exceções. Isso lhe ajuda a definir o que o usuário do código deve esperar, independente do que ocorra de errado no código que é executado no try.

Vejamos um exemplo. Precisamos criar um código que acesse um arquivo e consulte alguns objetos em série.

Começamos com um teste de unidade que mostra como capturar uma exceção se o arquivo não existir:

```

@Test(expected = StorageException.class)
public void retrieveSectionShouldThrowOnInvalidFileName() {
    sectionStore.retrieveSection("invalid - file");
}

```

O teste nos leva a cria esse stub:

```

public List<RecordedGrip> retrieveSection(String sectionName) {
    // retorno ficticio ate que tenhamos uma implementacao real
    return new ArrayList<RecordedGrip>();
}

```

Nosso teste falha porque ele não lança uma exceção. Em seguida, mudamos nossa implementação de modo a tentar acessar um arquivo inválido. Essa operação lança uma exceção:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
    try {
        FileInputStream stream = new FileInputStream(sectionName)
    } catch (Exception e) {
        throw new StorageException("retrieval error", e);
    }
    return new ArrayList<RecordedGrip>();
}
```

Nosso teste funciona agora porque capturamos a exceção. Neste momento, podemos refatorar. Podemos reduzir o tipo de execução que capturamos para combinar com aquele que realmente é lançado pelo construtor `FileInputStream: FileNotFoundException`:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
    try {
        FileInputStream stream = new FileInputStream(sectionName);
        stream.close();
    } catch (FileNotFoundException e) {
        throw new StorageException("retrieval error", e);
    }
    return new ArrayList<RecordedGrip>();
}
```

Agora que definimos o escopo com uma estrutura `try...catch`, podemos usar o TDD para construir o resto da lógica que precisamos, que será adicionada na criação do `FileInputStream` do `close` e poderá fingir que nada de errado aconteceu.

Experimente criar testes que forcão exceções e, então, adicione a ação ao seu tratador para cumprir seus testes. Isso fará com que você crie primeiro o escopo de transação do bloco `try` e lhe ajudará a manter essa natureza de transação daquele escopo.

Use exceções não verificadas

A discussão acabou. Por anos, programadores Java têm discutido sobre as vantagens e desvantagens de exceções verificadas. Quando a verificação exceções surgiu com a primeira versão do Java, parecia uma ótima ideia. A assinatura de todo método listaria todas as exceções que ele passaria ao seu chamador. Ademais, essas exceções eram parte do tipo do método. Seu código literalmente não seria compilado se a assinatura não fosse a mesma da que seu código podia fazer.

Naquela época, pensamos que exceções verificadas fosse uma ideia ótima; e era, elas tinham *algumas* vantagens. Entretanto, ficou claro agora que elas não são necessárias para a produção de um software robusto. O C# não verifica exceções, e, apesar das tentativas, nem o C++. Nem mesmo Python ou Ruby. Ainda assim é possível criar um software robusto em todas essas linguagens, porque, nesse caso, temos de decidir se realmente as exceções verificadas valem o preço que se paga.

Que preço? O de verificar exceções é a violação do Princípio de Aberto-Fechado¹.

Se você lançar uma exceção a ser verificada a partir de um método em seu código e o

catch estiver três níveis acima, *será preciso declará-la na assinatura de cada método entre você e o catch*. Isso significa que uma modificação em um nível mais baixo do software pode forçar a alteração de assinaturas em muitos níveis mais altos. Os módulos alterados podem ser reconstruídos e redistribuídos, mesmo que nada inerente a eles tenha sido mudado.

Considere a hierarquia de chamadas de um sistema grande. As funções no topo chamam as abaixo delas, que chamam outras abaixo delas e *ad infinitum*. Agora digamos que uma das funções dos níveis mais baixos seja modificada de uma forma que ela deva lançar uma exceção. Se essa exceção for verificada, então a assinatura da função deverá adicionar uma instrução throws. Mas isso significa que cada função que chamar nossa função modificada também deverá ser alterada para capturar a nova exceção ou anexar a instrução throws apropriada a sua assinatura. *Ad infinitum*. O resultado aninhado é uma cascata de alterações que vão desde os níveis mais baixo do software até o mais alto! Quebra-se o encapsulamento, pois todas as funções no caminho de um lançamento (*throw*) devem enxergar os detalhes daquela exceção de nível mais baixo. Segundo o propósito de exceções de que elas lhe permitem tratar erros distantes, é uma pena que as exceções verificadas quebrem dessa forma o encapsulamento.

As exceções verificadas podem às vezes ser úteis se você estiver criando uma biblioteca crítica: é preciso capturá-las. Mas no desenvolvimento geral de aplicativo, os custos da dependência superam as vantagens.

Forneça exceções com contexto

Cada exceção lançada deve fornecer contexto o suficiente para determinar a fonte e a localização de um erro. Em Java, você pode pegar um *stack trace* de qualquer exceção; entretanto, ele não consegue lhe dizer o objetivo da operação que falhou.

Crie mensagens de erro informativas e as passe juntamente com as exceções. Mencione a operação que falhou e o tipo da falha. Se estiver registrando as ações de seu aplicativo, passe informações suficientes para registrar o erro de seu catch.

Defina as classes de exceções segundo as necessidades do chamador

Há muitas formas de classificar erros. Pode ser pela origem: eles vieram desse componente ou daquele? Pelo tipo: são falhas de dispositivos, de redes ou erros de programação? Entretanto, quando definimos as classes de exceção num aplicativo, nossa maior preocupação deveria ser *como elas são capturadas*.

Vejamos um exemplo de uma classificação ruim de exceção. Aqui, há uma estrutura try...catch...finally para uma chamada a uma biblioteca de outro fabricante. Ela cobre todas as exceções que a chamada talvez lance:

```
ACMEPort port = new ACMEPort(12);

try {
    port.open();
} catch (DeviceResponseException e) {
    reportPortError(e);
    logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
    reportPortError(e);
```

```
        logger.log("Unlock exception", e);
    } catch (GMXError e) {
        reportPortError(e);
        logger.log("Device response exception");
    } finally {
        ...
    }
```

A estrutura possui muita duplicação, e não deveríamos estar surpresos. Na maioria dos casos de tratamento de exceções, o que fazemos é relativamente padrão, independente da situação no momento. Temos de registrar um erro e nos certificar que podemos prosseguir.

Neste caso, como sabemos que a tarefa que estamos fazendo é basicamente a mesma independente da exceção, podemos simplificar nosso código consideravelmente. Para isso, pegamos a API que estamos chamando e garantimos que ela retorne um tipo comum de exceção.

```
LocalPort port = new LocalPort(12);
try {
    port.open();
} catch (PortDeviceFailure e) {
    reportError(e);
    logger.log(e.getMessage(), e);
} finally {
    ...
}
```

Nossa classe `LocalPort` é um simples wrapper (“empacotador”) que captura e traduz as exceções lançadas pela classe `ACMEPort`:

```
public class LocalPort {
    private ACMEPort innerPort;

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

    public void open() {
        try {
            innerPort.open();
        } catch (DeviceResponseException e) {
            throw new PortDeviceFailure(e);
        } catch (ATM1212UnlockedException e) {
            throw new PortDeviceFailure(e);
        } catch (GMXError e) {
            throw new PortDeviceFailure(e);
        }
    }
    ...
}
```

Wrappers como o que definimos para a `ACMEPort` podem ser muito úteis. Na verdade, empacotar APIs de terceiros é a melhor prática que existe. Ao fazer isso, você minimiza as dependências nelas: você pode escolher migrar para uma biblioteca diferente no futuro sem

muitos problemas. Empacotar também facilita a simulação de chamadas de terceiros quando for testar seu próprio código.

Uma última vantagem de empacotar (wrapping) é que você não fica preso às escolhas do modelo de API de um fornecedor em particular. Você pode definir a API que preferir. No exemplo anterior, definimos um único tipo de exceção para a falha do dispositivo `port` e descobrimos que poderíamos escrever um código muito mais limpo.

Geralmente, uma única classe de exceção está bom para uma parte específica do código. As informações enviadas com a exceção podem distinguir os erros. Use classes diferentes apenas se houver casos em que você queira capturar uma exceção e permitir que a outra passe normalmente.

Defina o fluxo normal

Se você seguir os conselhos das seções anteriores, acabará com uma boa quantidade de divisão entre sua lógica do negócio e seu tratamento de erro. A maioria de seu código começará a parecer um algoritmo limpo e sem apetrechos. Entretanto, esse processo eleva ao máximo a detecção de erro em seu programa. Você empacota suas APIs de modo que você possa lançar suas próprias exceções e definir um controlador acima de seu código para que você possa lidar com qualquer computação cancelada. Na maioria das vezes, essa é uma abordagem ótima, mas há situações nas quais você talvez não queira cancelar.



Vejamos um exemplo. Abaixo está um código confuso que soma as despesas em um aplicativo de finanças:

```
try {
    MealExpenses expenses = expenseReportDAO.getMeals(employee.
getID());
    m_total += expenses.getTotal();
} catch(MealExpensesNotFound e) {
    m_total += getMealPerDiem();
}
```

Neste negócio, se as refeições (meals) forem um custo, elas se tornam parte do total. Caso contrário, o funcionário (employee) recebe uma quantia para ajuda de custos (PerDiem) pela refeição daquele dia. A exceção confunde a lógica.

Não seria melhor se não tivéssemos de lidar com o caso especial? Dessa forma, nosso código seria muito mais simples. Ele ficaria assim:

```
MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
m_total += expenses.getTotal();
```

Podemos tornar o código mais simples? Parece que sim. Podemos alterar a `ExpenseReportDAO` de modo que ela sempre retorne um objeto `MealExpense`. Se não houver gastos com refeições,

ela retorna um objeto `MealExpense` que retorna a ajuda de custos como seu total:

```
public class PerDiemMealExpenses implements MealExpenses {
    public int getTotal() {
        // retorna a ajuda de custos padrao
    }
}
```

Isso se chama o Padrão Special Case ('padrão do caso especial'), de Fowler. Você cria uma classe ou configura um objeto de modo que ele trate de um caso especial para você. Ao fazer isso, o código do cliente não precisa lidar com o comportamento diferente. Este fica encapsulado num objeto de caso especial.

Não retorne null

Acho que qualquer discussão sobre tratamento de erro deveria incluir as coisas que fazemos que levam a erros. A primeira da lista seria retornar `null`. Já perdi a conta dos aplicativos que já vi que em quase toda linha verificava por `null`. Abaixo está um exemplo:

```
public void registerItem(Item item) {
    if (item != null) {
        ItemRegistry registry = persistentStore.getItemRegistry();
        if (registry != null) {
            Item existing = registry.getItem(item.getID());
            if (existing.getBillingPeriod().hasRetailOwner()) {
                existing.register(item);
            }
        }
    }
}
```

Se você trabalhar num código-fonte como esse, ele pode não parecer tão ruim assim para você, mas ele é! Quando retornamos `null`, basicamente estamos criando mais trabalho para nós mesmos e jogando problemas em cima de nossos chamadores. Só basta esquecer uma verificação `null` para que o aplicativo fique fora de controle.

Percebeu que não havia uma verificação de `null` na segunda linha do `if` aninhado? O que teria acontecido em tempo de execução se `persistentStore` fosse `null`? Teríamos uma `NullPointerException` em tempo de execução, e ou alguém está capturando-a no nível mais alto ou não. Em ambos os casos isso é péssimo. O que você faria exatamente em resposta a um lançamento de `NullPointerException` das profundezas de seu aplicativo?

É fácil dizer que o problema com o código acima é a falta de uma verificação de `null`, mas, na verdade, o problema é que ele tem muitos. Se você ficar tentado a retornar `null` de um método, em vez disso, considere lançar uma exceção ou retornar um objeto SPECIAL CASE. Se estiver chamando um método que retorne `null` a partir de uma API de terceiros, considere empacotá-lo com um método que lance uma exceção ou retorne um objeto de caso especial.

Em muitos casos, objetos de casos especiais são uma solução fácil. Imagine que seu código fosse assim:

```
List<Employee> employees = getEmployees();
if (employees != null) {
    for (Employee e : employees) {
```

```
        totalPay += e.getPay();
    }
}
```

Neste momento, `getEmployees` pode retornar `null`, mas ele precisa? Se alterássemos `getEmployee` de modo que ele retornasse uma lista vazia, poderíamos limpar o código:

```
List<Employee> employees = getEmployees();
for(Employee e : employees) {
    totalPay += e.getPay();
}
```

Felizmente, Java possui o `Collections.emptyList()`, e ele retorna uma lista predefinida e imutável que podemos usar para esse propósito:

```
public List<Employee> getEmployees() {
if( ... there are no employees ... )
    return Collections.emptyList();
}
```

Se programar dessa forma, você minimizará a chance de `NullPointerExceptions` e seu código será mais limpo.

Não passe null

Retornar `null` dos métodos é ruim, mas passar `null` para eles é pior. A menos que esteja trabalhando com uma API que espere receber `null`, você deve evitar passá-lo em seu código sempre que possível.

Vejamos um exemplo do porquê. Abaixo está um método simples que calcula a distância entre dois pontos:

```
public class MetricsCalculator
{
    public double xProjection(Point p1, Point p2) {
        return (p2.x - p1.x) * 1.5;
    }
    ...
}
```

O que acontece quando alguém passa `null` como parâmetro?

```
calculator.xProjection(null, new Point(12, 13));
```

Receberemos uma `NullPointerException`, é claro.

Podemos consertar isso? Poderíamos criar um novo tipo de exceção e lançá-lo:

```
public class MetricsCalculator
{
    public double xProjection(Point p1, Point p2) {
        if (p1 == null || p2 == null) {
```

```
        throw new InvalidArgumentException(
            "Invalid argument for
MetricsCalculator.xProjection");
    }
    return (p2.x - p1.x) * 1.5;
}
}
```

Ficou melhor? Pode ser um pouco melhor do que uma exceção de ponteiro `null`, mas lembre-se de que temos de definir um tratador para `InvalidArgumentException`. O que ele deve fazer? Há algum procedimento bom?

Há uma alternativa. Poderíamos usar uma série de confirmações:

```
public class MetricsCalculator
{
    public double xProjection(Point p1, Point p2) {
        assert p1 != null : "p1 não pode ser nulo";
        assert p2 != null : "p2 não pode ser nulo";
        return (p2.x - p1.x) * 1.5;
    }
}
```

É uma boa informação, mas não resolve o problema. Se alguém passar `null`, ainda teremos um erro em tempo de execução.

Na maioria das linguagens de programação não há uma boa forma de lidar com um valor nulo passado acidentalmente para um chamador. Como aqui este é o caso, a abordagem lógica seria proibir, por padrão, a passagem de `null`. Ao fazer isso, você pode programar com o conhecimento de que um `null` numa lista de parâmetros é sinal de problema, e acabar com mais alguns erros por descuido.

Conclusão

Um código limpo é legível, mas também precisa ser robusto. Esses objetivos não são conflitantes. Podemos criar programas limpos e robustos se enxergarmos o tratamento de erro como uma preocupação à parte, algo que seja visível independentemente de nossa lógica principal. Na medida em que somos capazes de fazer isso, podemos pensar nisso de forma independente e dar um grande passo na capacidade de manutenção de nosso código.

Bibliografia

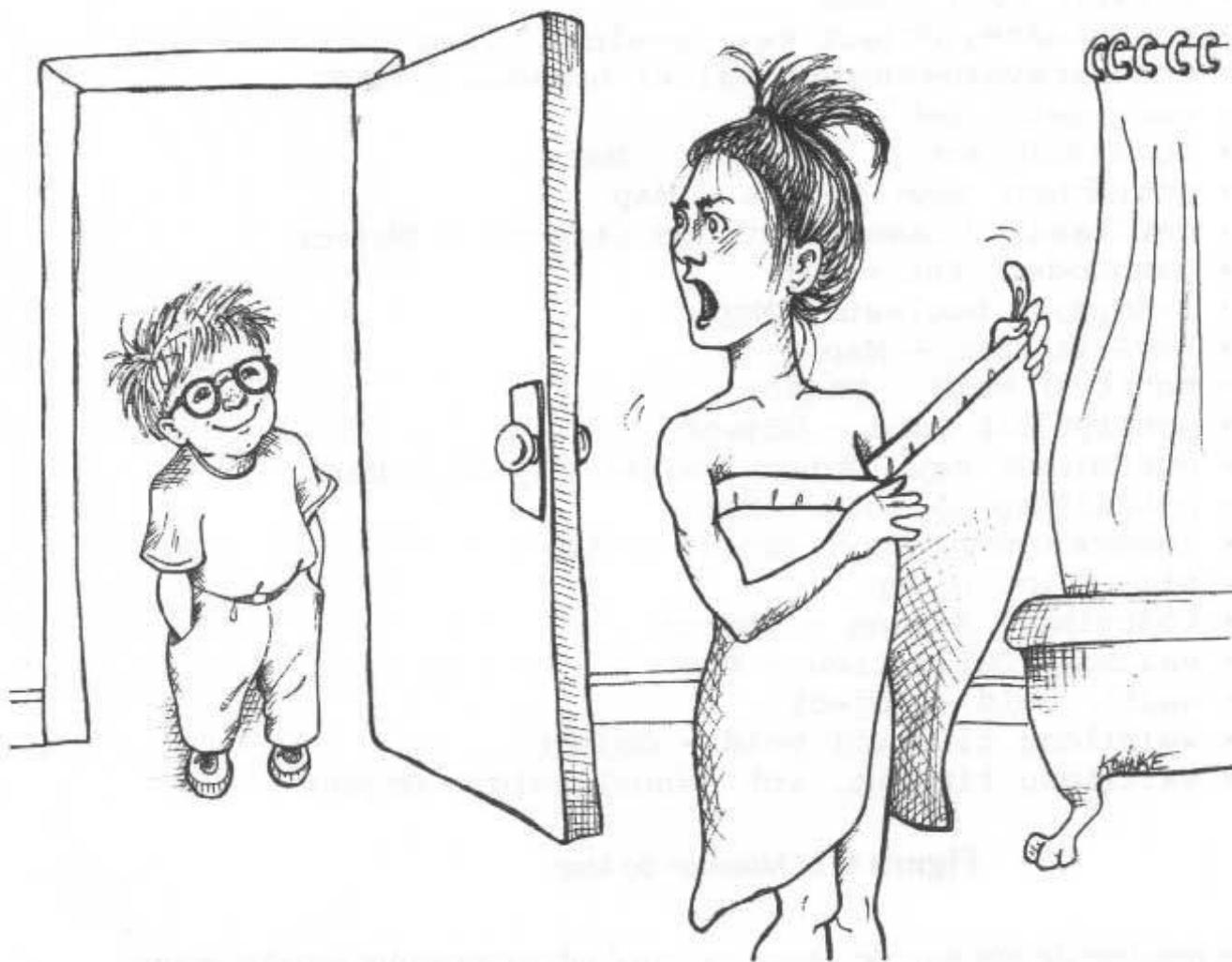
[Martin]: Agile Software Development: Principles, Patterns, and Practices, Robert C. Martin, Prentice Hall, 2002.

8. Limite os softwares que

8

Limites

por James Grenning



Raramente controlamos todos os softwares em nossos sistemas. De vez em quando compramos pacotes de outros fabricantes ou usamos códigos livres, ou dependemos de equipes em nossa própria empresa para construir componentes ou subsistemas para nós. De algum modo, devemos integrar, de forma limpa, esse código externo ao nosso. Neste capítulo veremos as práticas e técnicas para manter limpos os limites de nosso software.

O uso de códigos de terceiros

Há uma tensão natural entre o fornecedor de uma interface e seu usuário. Os fornecedores de pacotes e frameworks de outros fabricantes visam a uma maior aplicabilidade de modo que possam trabalhar com diversos ambientes e atender a um público maior. Já os usuários desejam uma interface voltada para suas próprias necessidades. Essa tensão pode causar problemas nos limites de nossos sistemas.

Tomemos o `java.util.Map` como exemplo. Como pode ver na Figura 8.1, os Maps tem uma interface bastante ampla com diversas capacidades. Certamente esse poder e flexibilidade são úteis, mas também pode ser uma desvantagem. Por exemplo, nosso aplicativo pode construir um Map e passá-lo adiante. Nossa objetivo talvez seja que nenhum dos recipientes de nosso Map não exclua nada do Map. Mas logo no início da lista está o método `clear()`. Qualquer usuário do Map tem o poder de apagá-lo. Ou talvez, segundo a convenção que adotamos, o Map pode armazenar apenas certos tipos de objetos, mas não é certo que ele restrinja os tipos que são adicionados a ele. Qualquer usuário determinado pode adicionar itens de qualquer tipo a qualquer Map.

Se nosso aplicativo precisa de um Map de Sensors, você pode se deparar com um Sensors assim:

- `clear() void - Map`
- `containsKey(Object key) boolean - Map`
- `containsValue(Object value) boolean - Map`
- `entrySet() Set - Map`
- `equals(Object o) boolean - Map`
- `get(Object key) Object - Map`
- `getClass() Class<? extends Object> - Object`
- `hashCode() int - Map`
- `isEmpty() boolean - Map`
- `keySet() Set - Map`
- `notify() void - Object`
- `notifyAll() void - Object`
- `put(Object key, Object value) Object - Map`
- `putAll(Map t) void - Map`
- `remove(Object key) Object - Map`
- `size() int - Map`
- `toString() String - Object`
- `values() Collection - Map`
- `wait() void - Object`
- `wait(long timeout) void - Object`
- `wait(long timeout, int nanos) void - Object`

Figura 8.1: Métodos do Map

Se nosso aplicativo precisar de um Map de sensors, você talvez encontre sensors assim:

```
Map sensors = new HashMap();
```

E, quando alguma outra parte do código precisa acessar o Sensor, você vê isso:

```
Sensor s = (Sensor)sensors.get(sensorId);
```

Não vemos isso apenas uma vez, mas várias ao longo do código. O cliente deste código fica com a responsabilidade de obter um `Object` do `Map` e atribuí-lo o tipo certo. Apesar de funcionar, não é um código limpo. Ademais, esse código não explica muito bem o que ele faz. Pode-se melhorar consideravelmente sua legibilidade com o uso de tipos genéricos, como mostra abaixo:

```
Map<Sensor> sensors = new HashMap<Sensor>();  
...  
Sensor s = sensors.get(sensorId);
```

Entretanto, isso não resolve o problema de que `Map<sensor>` oferece mais capacidade do que precisamos ou queremos.

Passar adiante pelo sistema uma instância de `Map<Sensor>` significa que haverá vários lugares para mexer se a interface para o `Map` mudar. Você talvez pense que uma mudança seja pouco provável, mas lembre-se de que houve uma quando o suporte a genéricos foi adicionado no Java 5. De fato, já vimos sistemas que impedem o uso de genéricos devido à magnitude das alterações necessárias para manter o uso abrangente dos `Maps`.

Abaixo está uma forma limpa de usar o `Map`. Nenhum usuário do `Sensors` se importaria se um pouco de genéricos for usado ou não. Essa escolha se tornou (e sempre deve ser) um detalhe da implementação.

```
public class Sensors {  
    private Map sensors = new HashMap();  
  
    public Sensor getById(String id) {  
        return (Sensor) sensors.get(id);  
    }  
    //codigo  
}
```

A interface no limite (`Map`) está oculta. É possível alterá-la causando muito pouco impacto no resto do aplicativo. O uso de tipos genéricos não é mais uma questão tão problemática assim, pois o gerenciamento de declarações e de tipos é feito dentro da classe `Sensors`.

Essa interface também foi personalizada para satisfazer as necessidades do aplicativo. Seu resultado é um código mais fácil de se entender e mais difícil de ser utilizado incorretamente. A classe `Sensors` pode forçar regras de modelo e de negócios.

Não estamos sugerindo que cada uso do `Map` seja encapsulado dessa forma. Mas lhe aconselhando para não passar os `Maps` (ou qualquer outra interface num limite) por todo o sistema. Se usar uma interface, como a `Map`, no limite, a mantenha numa classe ou próxima a uma família de classes em que ela possa ser usada. Evite retorná-la ou aceitá-la como parâmetro em APIs públicas.

Explorando e aprendendo sobre limites

Códigos de terceiros nos ajudam a obter mais funcionalidade em menos tempo. Por onde começar quando desejamos utilizar pacotes de terceiros? Não é tarefa nossa testá-los, mas pode ser melhor para nós criar testes para os códigos externos que formos usar.

Suponha que não esteja claro como usar uma biblioteca de terceiros. Podemos gastar um dia ou dois (até mais) lendo a documentação e decidindo como vamos usá-la. Então, escreveríamos

nosso código para usar o código externo e vemos se ele é ou não o que achávamos. Não ficaríamos surpresos de acabar em longas sessões de depuração tentando descobrir se os bugs que encontramos são do nosso código ou no deles.

Entender códigos de terceiros é difícil. Integrá-lo ao seu também é. Fazer ambos ao mesmo tempo dobra a dificuldade. E se adotássemos uma outra abordagem? Em vez de experimentar e tentar o novo código, poderíamos criar testes para explorar nosso conhecimento sobre ele. Jim Newkirk chama isso de *testes de aprendizagem*¹.

Nesses testes, chamamos a API do código externo como o faríamos ao usá-la em nosso aplicativo.

Basicamente estariamos controlando os experimentos que verificam nosso conhecimento daquela API.

O teste se focaliza no que desejamos saber sobre a API.

Aprendendo sobre log4j

Digamos que queremos usar o pacote log4j do Apache em vez de nosso próprio gravador de registro interno.

Baixaríamos o pacote e então abriríamos a página de documentação inicial. Sem ler muito, criamos nosso primeiro caso de teste, esperando que seja impresso “oi” no console.

```
@Test
public void testLogCreate() {
    Logger logger = Logger.getLogger("MyLogger");
    logger.info("oi");
}
```

Quando o executamos, o registrador (logger) produz um erro o qual nos diz que precisamos de algo chamado Appender. Após ler um pouco mais, descobrimos que existe um ConsoleAppender. Então, criamos um ConsoleAppender e vemos se desvendamos os segredos de registro no console.

```
@Test
public void testLogAddAppender() {
    Logger logger = Logger.getLogger("MyLogger");
    ConsoleAppender appender = new ConsoleAppender();
    logger.addAppender(appender);
    logger.info("oi");
}
```

Desta vez, descobrimos que o Appender não possui fluxo de saída. Estranho... Parecia lógico ter um. Depois de buscar ajuda no Google, tentamos o seguinte:

```
@Test
public void testLogAddAppender() {
    Logger logger = Logger.getLogger("MyLogger");
    logger.removeAllAppenders();
    logger.addAppender(new ConsoleAppender(
        new PatternLayout("%p %t %m%n"),
        ConsoleAppender.SYSTEM_OUT));
    logger.info("oi");
}
```

Funcionou. Uma mensagem de registro com “oi” apareceu no console! Parece estranho ter de dizer ao `ConsoleAppender` o que ele precisa escrever no console. Mais interessante ainda é quando removemos o parâmetro `ConsoleAppender.SYSTEM_OUT` e ainda é exibido “oi”. Mas quando retiramos o `PatternLayout`, mais uma vez há mensagem de falta de um fluxo de saída. Esse comportamento é muito estranho.

Lendo a documentação com um pouco mais de atenção, vimos que o construtor `ConsoleAppender` padrão vem “desconfigurado”, o que não parece muito óbvio ou prático. Parece um bug, ou pelo menos uma inconsistência, no log4j. Recorrendo novamente ao Google, lendo e testando, acabamos chegando à Listagem 8.1. Descobrimos bastante coisa sobre como funciona o log4j, e colocamos esse conhecimento numa série de testes simples de unidade.

Listagem 8-1

`LogTest.java`

```
public class LogTest {
    private Logger logger;

    @Before
    public void initialize() {
        logger = Logger.getLogger("logger");
        logger.removeAllAppenders();
        Logger.getRootLogger().removeAllAppenders();
    }

    @Test
    public void basicLogger() {
        BasicConfigurator.configure();
        logger.info("basicLogger");
    }

    @Test
    public void addAppenderWithStream() {
        logger.addAppender(new ConsoleAppender(
            new PatternLayout("%p %t %m%n"),
            ConsoleAppender.SYSTEM_OUT));
        logger.info("addAppenderWithStream");
    }

    @Test
    public void addAppenderWithoutStream() {
        logger.addAppender(new ConsoleAppender(
            new PatternLayout("%p %t %m%n")));
        logger.info("addAppenderWithoutStream");
    }
}
```

Agora sabemos como obter um console simples e inicializado de registro, e podemos encapsular esse conhecimento em nossas classes de registro de modo que o resto de nosso aplicativo fique isolado da interface limite do log4j.

Os testes de aprendizagem são melhores que de graça

Os testes de aprendizagem acabam não custando nada. Tivemos de aprender sobre a API mesmo, e escrever aqueles testes foi uma forma fácil e separada de obter o conhecimento que

conseguimos. Os testes de aprendizagem são experimentos precisos que ajudam a aumentar nosso entendimento.

Esses testes não só saem de graça como geram um retorno positivo de nosso investimento. Quando houver nossas distribuições daquele pacote externo, podemos executar os testes para ver se há diferenças nas atividades.

Os testes de aprendizagem verificam se os pacotes de terceiros que estamos usando se comportam como desejamos. Uma vez integrados, não há garantias de que o código se manterá compatível com as nossas necessidades. Os autores originais sofrerão pressão para alterarem seus códigos para satisfazer suas próprias necessidades. Eles consertarão bugs e adicionarão novos recursos. Cada distribuição vem com um novo risco. Se o pacote for alterado de uma forma que fique incompatível com nossos testes, descobriremos de imediato.

Você precise ou não do conhecimento proporcionado pelos testes de aprendizagem, deve-se definir um limite claro por meio de uma série de testes externos que experimentem a interface da mesma forma que seu código faria. Sem esses *testes limite* para facilitar a migração, poderemos ficar tentados a manter por mais tempo do que deveríamos a versão antiga.

O uso de código que não existe ainda

Há outro tipo de limite, um que separa o conhecido do desconhecido. Geralmente há lugares no código onde nosso conhecimento parece sumir. Às vezes, o que está do outro lado nos é desconhecido (pelo menos agora). Às vezes, optamos não olhar além do limite.

Alguns anos atrás fiz parte do time do desenvolvimento de software para um sistema de comunicação de rádios. Havia um subsistema, o “Transmissor”, que eu pouco sabia a respeito, e as pessoas responsáveis por ele não tinham ainda definido sua interface. Não queríamos ficar parados, então começamos a trabalhar longe daquela parte desconhecida do código.

Sabíamos muito bem onde nosso mundo terminava e onde começava o novo. Conforme trabalhávamos, às vezes chegávamos a esse limite entre os dois mundos. Embora névoas e nuvens de ignorância ofuscassem nossa visão para além do limite, nosso trabalho nos mostrou o que queríamos que fosse a interface limite. Desejávamos dizer ao transmissor algo assim:

Configure o transmissor na frequência fornecida e emita uma representação analógica dos dados provenientes desde fluxo.

Não tínhamos ideia de como isso seria feito, pois a API ainda não havia sido desenvolvida. Portanto, decidimos trabalhar nos detalhes depois.

Para não ficarmos parados, definimos nossa própria interface. Demos um nome fácil de lembrar, como `Transmitter`. Criamos um método chamado `transmit` que pegava uma frequência e um fluxo de dados. Essa era a interface que gostaríamos de ter.

O bom de criar a interface que desejamos é que podemos controlá-la. Isso ajuda a manter o código do lado do cliente mais legível e centralizado na função para a qual foi criado.

Na Figura 8.2 você pode ver que preenchemos as classes do `CommunicationsController` a partir da API do `Transmitter`, a qual não controlávamos e havia sido definida. Ao usar nossa própria interface para o aplicativo, mantivemos limpo e expressivo nosso código do `CommunicationsController`. Uma vez definida a API do `Transmitter`, criamos o

`TransmitterAdapter` para fazer a conexão. O ADAPTER² encapsulou a interação com a API e forneceu um único local para ser modificado se a API for aperfeiçoada.

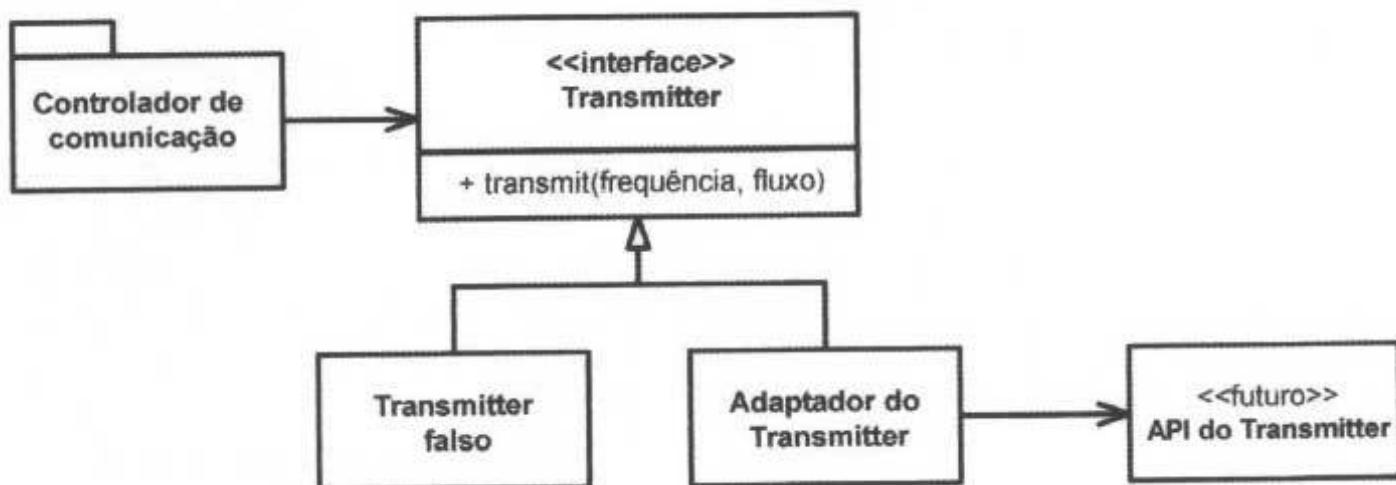


Figura 8.2: Adivinhando deverá ser o transmitter

Esse modelo também nos dá um seam³ bastante conveniente no código para testarmos. Ao usar um `FakeTransmitter` (transmissor falso) adequado, podemos testar as classes do `CommunicationsController`. Além de podermos criar testes limite uma vez que temos a `TransmitterAPI` para garantir que estamos usando corretamente a API.

Limites limpos

Coisas interessantes ocorrem nos limites. A alteração é uma delas. Bons projetos de software acomodam modificações sem muito investimento ou trabalho. Quando usamos códigos que estão fora de controle, deve-se dar uma atenção especial ao nosso investimento e garantir que uma mudança futura não seja muito custosa.

O código nos limites precisa de uma divisão clara e testes que definem o que se deve esperar. Devemos evitar que grande parte de nosso código enxergue as particularidades dos de terceiros. É melhor depender de algo que *você* controle do que pegar algo que acabe controlando você. Lidamos com os limites de códigos externos através de alguns poucos lugares em nosso código que fazem referência a eles. Podemos empacotá-los como fizemos com o Map, ou talvez usar um ADAPTER para converter nossa interface perfeita na que nos for fornecida. De qualquer forma, nosso código se comunica melhor conosco, provê uso consistente interno pelo limite e possui poucos pontos para serem mexidos quando o código externo sofrer alteração.

Bibliografia

[BeckTDD]: *Test Driven Development*, Kent Beck, Addison-Wesley, 2003.

[GOF]: *Padrões de Projeto, Soluções Reutilizáveis de Software Orientado a Objetos*, Gamma et al., Addison-Wesley, 1996.

[WELC]: *Working Effectively with Legacy Code*, Addison-Wesley, 2004.

² Consulte o padrão Adapter no [GOF].

Testes de Unidade



Nossa profissão evoluiu bastante ao longo dos últimos dez anos. Em 1997 não se ouvia falar em Desenvolvimento Dirigido a Testes (TDD, sigla em inglês). Para a grande maioria de nós, os testes de unidade eram um pequeno pedaço de código descartável que escrevíamos para nos certificar que nossos programas funcionavam. Meticulosamente criávamos nossas classes e métodos e, então, improvisávamos um código para testá-los. Tipicamente, isso envolvia um programa simples de controle que nos permitisse interagir manualmente com o programa que havíamos escrito.

Lembro-me de ter criado em meados da década de 1990 um programa em C++ para um sistema integrado em tempo real. Era um simples contador com a seguinte assinatura:

```
void Timer::ScheduleCommand(Command* theCommand, int milliseconds)
```

A idéia era simples: o método `execute` de `Command` seria executado numa nova thread após um número específico de milésimos de segundos. O programa era como testá-lo.

Criei um simples programa de controle que esperava alguma ação no teclado. Sempre que um caractere era pressionado, ele agendaria um comando que escreveria o mesmo caractere cinco segundos depois. Então eu digitava uma melodia no teclado e esperava que ela fosse reproduzida na tela cinco segundos depois.

“Eu ... quero-uma-mulher ... igual ... a-com-quem-me-ca ... sei ... querido ... pa ... pai.”

Realmente cantei essa melodia enquanto digitava o ponto “.” e, então, a cantava novamente quando aparecia na tela.

Esse era meu teste! Depois que o vi funcionar e o mostrei aos meus colegas, joguei o código do teste fora.

Como eu disse, nossa profissão evoluiu. Atualmente eu criaria um teste que garantisse que cada canto do código funcionasse como eu esperava. Eu isolaria meu código do resto do sistema operacional em vez de apenas invocar as funções padrão de contagem; simularia aquelas funções de modo que eu tivesse controle absoluto sobre o tempo; agendaria comandos que configurassem flags booleanas; e, então, adiantaria o tempo, observando as flags e verificando se elas mudavam de falsas para verdadeiras quando eu colocasse o valor correto no tempo.

Quando pego uma coleção de testes para passar adiante, eu me certifico se eles são adequados para serem executados por qualquer pessoa que precise trabalhar com o código, e se eles e o código estavam juntos no mesmo pacote de origem.

Isso, progredimos bastante, mas ainda podemos ir mais longe. Os movimentos do Agile e do TDD têm incentivado muitos programadores a criarem testes de unidade automatizados, e muitos outros estão se unindo a cada dia. Mas nessa correria para adicionar testes ao nosso ofício, muitos programadores têm se esquecido de alguns dos pontos mais sutis e importantes de se escrever bons testes.

As três leis do TDD

Hoje em dia todos sabem que o TDD nos pede para criar primeiro os testes de unidade antes do código de produção. Mas essa regra é apenas o início. Considere as três leis¹ abaixo:

Primeira Lei Não se deve escrever o código de produção até criar um teste de unidade de falhas.

Segunda Lei Não se deve escrever mais de um teste de unidade do que o necessário para falhar, e não compilar é falhar.

Terceira Lei Não se deve escrever mais códigos de produção do que o necessário para aplicar o teste de falha atual.

Essas três leis lhe colocam numa rotina que talvez dure trinta segundos. Os testes e o código de produção são escritos juntos, com os testes apenas alguns segundos mais adiantados.

Se trabalharmos dessa forma, criariamos dezenas de testes a cada dia, centenas a cada mês e milhares a cada ano; os testes cobririam praticamente todo o nosso código de produção. O tamanho completo desses testes, que pode competir com o tamanho do próprio código de produção, pode apresentar um problema de gerenciamento intimidador.

Como manter os testes limpos

Há alguns anos, pedi para orientar uma equipe que tinha decidido explicitamente que seus códigos de testes *não deveriam ser* preservados segundo os mesmos padrões de qualidade que seu código de produção. Eles se deram autorização para violar as leis em seus testes de unidade. “Rápida e porcamente” era o lema. As variáveis não precisavam de nomes bem selecionados, as funções de teste não tinham de ser curtas e descritivas. Os códigos de testes não precisavam ser bem desenvolvidos e divididos de modo pensado. Se que o resto dos códigos de testes funcionasse e que cobrisse o código de produção, já era o suficiente.

Alguns de vocês lendo isso talvez simpatizem com tal decisão. Talvez, lá no passado, você criou testes tipo aquele que fiz para aquela classe Timer. É um grande passo ir da criação daquele tipo de teste descartável para uma coleção de testes de unidade automatizados. Portanto, assim como a equipe que eu orientara, você talvez decida que testes feitos “porcamente” sejam melhores do que nada.

O que aquela equipe não percebera era que ter testes daquele tipo é equivalente, se não pior, a não ter teste algum. O problema é que muitos testes devem ser alterados conforme o código de produção evolui. Quanto pior o teste, mais difícil será de mudá-lo. Quanto mais confuso for o código de teste, são maiores as chances de você levar mais tempo espremendo novos testes para dentro da coleção do que na criação do código de produção. Conforme você modifica o código de produção, os testes antigos começam a falhar e a bagunça no código de teste dificulta fazê-los funcionar novamente. Sendo assim, os testes começam a ser vistos como um problema em constante crescimento.

De distribuição em distribuição, o custo de manutenção da coleção de testes de minha equipe aumentou. Com o tempo, isso se tornou a maior das reclamações entre os desenvolvedores. Quando os gerentes perguntaram o motivo da estimativa de finalização estava ficando tão grande, os desenvolvedores culparam os testes. No final, foram forçados a descartar toda coleção de testes.

Mas, sem uma coleção de testes, eles perderam a capacidade de garantir que, após alterações no código-fonte, ele funcionasse como o esperado e que mudanças em uma parte do sistema não afetariam as outras partes. Então, a taxa de defeitos começou a crescer. Conforme aumentava o número de bugs indesejáveis, começaram a temer fazer alterações e pararam de limpar o código de produção porque temiam que isso poderia fazer mais mal do que bem. O código de produção começou a se degradar. No final das contas, ficaram sem testes, com um código de produção confuso e cheio de bugs, com consumidores frustrados e o sentimento de que o esforço para criação de testes não valeu de nada.

De certa forma estavam certos. Tal esforço tinha sido em vão. Mas fora decisão deles permitir que os testes ficassem uma bagunça e se tornasse a origem do fracasso. Se tivessem mantido os testes limpos, o esforço para a criação dos testes não teria os deixado na mão. Posso dizer isso com um pouco de certeza, pois participei de tudo, e já orientei muitas equipes que obtiveram sucesso com testes de unidade limpos.

A moral da história é simples: *Os códigos de testes são tão importantes quanto o código de produção.* Ele não é componente secundário. Ele requer raciocínio, planejamento e cuidado. É preciso mantê-lo tão limpo quanto o código de produção.

Os testes habilitam as “-idades”

Se não mantiver seus testes limpos, irá perdê-los. E, sem eles, você perde exatamente o que mantém a flexibilidade código de produção. Isso mesmo, você leu certo. São os *testes de unidade* que mantêm seus códigos flexíveis, reutilizáveis e passíveis de manutenção. A razão é simples. Se você tiver testes, não terá medo de alterar o código! Sem os testes, cada modificação pode gerar um bug. Não importa o grau de flexibilidade de sua arquitetura ou de divisão de seu modelo, pois sem os testes você ficará relutante em fazer mudanças por temer gerar bugs não detectados.

Mas *com* os testes esse medo praticamente some. Quanto maior a cobertura de seus testes, menor o medo. Você pode fazer mudanças quase sem penalidade ao código que tenha uma arquitetura emaranhada e um modelo confuso e opaco. De fato, você pode *improvisar* essa arquitetura e esse modelo sem temer!

Portanto, ter uma coleção de testes de unidade automatizados que cubram o código de produção é o segredo para manter seu projeto e arquitetura os mais limpos possíveis. Os testes habilitam todas as “-idades”, pois eles possibilitam as *alterações*.

Dessa forma, caso seus testes estejam ruins, então sua capacidade de modificar seu código fica comprometida e você começa a perder a capacidade de melhorar a estrutura dele. Quanto piores forem os testes, pior o código se torna. No final, você perde os testes e seu código se degrada.

Testes limpos

O que torna um teste limpo? Três coisas: legibilidade, legibilidade e legibilidade. Talvez isso seja até mais importante nos testes de unidade do que no código de produção. O que torna os testes legíveis? O mesmo que torna todos os códigos legíveis: clareza, simplicidade e consistência de expressão. Num teste você quer dizer muito com o mínimo de expressões possíveis.

Considere o código do FitNesse na Listagem 9.1. Esses três testes são difíceis de entender e certamente podem ser melhorados. Primeiro, há uma quantidade terrível de código duplicado [G5] nas chamadas repetidas a `addPage` e `assertSubString`. Mais importante ainda é que este código está carregado de detalhes que interferem na expressividade do teste.

Listagem 9-1

SerializedPageResponderTest.java

```
public void testGetPageHierachyAsXml() throws Exception
{
    crawler.addPage(root, PathParser.parse("PageOne"));
    crawler.addPage(rroot, PathParser.parse("PageOne.ChildOne"));
    crawler.addPage(root, PathParser.parse("PageTwo"));
```

Listagem 9-1 (continuação)**SerializedPageResponderTest.java**

```
request.setResource("root");
request.addInput("type", "pages");
Responder responder = new SerializedPageResponder();
SimpleResponse response =
    (SimpleResponse) responder.makeResponse(
        new FitNesseContext(root), request);
String xml = response.getContent();

assertEquals("text/xml", response.getContentType());
assertSubString("<name>PageOne</name>", xml);
assertSubString("<name>PageTwo</name>", xml);
assertSubString("<name>ChildOne</name>", xml);
}

public void testGetPageHieratchyAsXmlDoesntContainSymbolicLinks()
throws Exception
{
    WikiPage pageOne = crawler.addPage(root, PathParser.parse("P
    crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
    crawler.addPage(root, PathParser.parse("PageTwo"));

    PageData data = pageOne.getData();
    WikiPageProperties properties = data.getProperties();
    WikiPageProperty symLinks = properties.set(SymbolicPage.PROP
    symLinks.set("SymPage", "PageTwo");
    pageOne.commit(data);

    request.setResource("root");
    request.addInput("type", "pages");
    Responder responder = new SerializedPageResponder();
    SimpleResponse response =
        (SimpleResponse) responder.makeResponse(
            new FitNesseContext(root), request);
    String xml = response.getContent();

    assertEquals("text/xml", response.getContentType());
    assertSubString("<name>PageOne</name>", xml);
    assertSubString("<name>PageTwo</name>", xml);
    assertSubString("<name>ChildOne</name>", xml);
    assertNotSubString("SymPage", xml);
}

public void testDataAsHtml() throws Exception
{
    crawler.addPage(root, PathParser.parse("TestPageOne"), "test
    request.setResource("TestPageOne");
    request.addInput("type", "data");
```

Por exemplo, veja as chamadas para `PathParser`. Elas transformam strings em instâncias `PagePath` pelos `crawler`. Essa transformação é completamente irrelevante na execução do teste e serve apenas para ofuscar o objetivo. Os detalhes em torno da criação do `responder` e a definição do tipo de `response` também são apenas aglomerados. E tem a inepta forma pela qual

é construído URL requisitado a partir de um `resource` e um parâmetro. (Ajudei a escrever esse código, portanto me sinto completamente no direito de criticá-lo).

No final, esse código não foi feito para ser lido. O pobre leitor é inundado com um monte de detalhes que devem ser entendidos antes que os testes façam algum sentido. Agora, considere os testes improvisados na Listagem 9.2. Eles fazem exatamente o mesmo, mas foram refatorados de uma forma muito mais clara e descriptiva.

Listagem 9-2

`SerializedPageResponderTest.java (refatorado)`

```
public void testGetPageHierarchyAsXml() throws Exception {
    makePages("PageOne", "PageOne.ChildOne", "PageTwo");

    submitRequest("root", "type:pages");

    assertResponseIsXML();
    assertResponseContains(
        "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
    );
}

public void testSymbolicLinksAreNotInXmlPageHierarchy() throws Exception {
    WikiPage page = makePage("PageOne");
    makePages("PageOne.ChildOne", "PageTwo");

    addLinkTo(page, "PageTwo", "SymPage");

    submitRequest("root", "type:pages");

    assertResponseIsXML();
    assertResponseContains(
        "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
    );
    assertResponseDoesNotContain("SymPage");
}

public void testDataAsXml() throws Exception {
    makePageWithContent("TestPageOne", "test page");

    submitRequest("TestPageOne", "type:data");

    assertResponseIsXML();
    assertResponseContains("test page", "<Test");
}
```

A estrutura desses testes tornou óbvio o padrão CONSTRUIR-OPERAR-VERIFICAR². Cada um dos testes está claramente dividido em três partes. A primeira produz os dados do teste, a segunda opera neles e a terceira verifica se a operação gerou os resultados esperados.

Note que a grande maioria dos detalhes maçantes foi eliminada. Os testes vão direto ao ponto e usam apenas os tipos de dados e funções que realmente precisam. Quem ler esses testes dev ser capaz de descobrir rapidamente o que fazem, sem se deixar enganar ou ficar sobrepujado pelos detalhes.

Linguagem de testes específica ao domínio

Os testes na Listagem 9.2 mostram a técnica da construção de uma linguagem específica a um domínio para seus testes. Em vez de usar as APIs que os programadores utilizam para manipular o sistema, construímos uma série de funções e utilitários que usam tais APIs e que tornam os testes mais convenientes de se escrever e mais fáceis de ler. Esses funções e utilitários se tornaram uma API especializada usada pelos testes. Eles testam a linguagem que os programadores usam para auxiliá-los a escrever seus testes e para ajudar aqueles que precisem ler esses testes mais tarde.

Essa API de teste não foi desenvolvida de uma só vez; ela evoluiu de uma contínua refatoração de códigos de testes que ficaram demasiadamente ofuscados por detalhes confusos. Assim como você me viu refatorar a Listagem 9.1 para Listagem 9.2, desenvolvedores disciplinados também refatoraram seus códigos de teste para formas mais sucintas e expressivas.

Um padrão duplo

De um certo modo, a equipe que mencionei no início deste capítulo estava certa. O código dentro da API de teste tem um conjunto diferente de padrões de engenharia em relação ao código de produção. Ele pode ser simples, sucinto e expressivo, mas não precisa ser mais eficiente do que o do código de produção. Apesar de tudo, ele roda num ambiente de teste, e não de um de produção, e esses dois ambientes possuem requisitos diferentes.

Considere o teste na Listagem 9.3 que escrevi como parte de um sistema de controle de ambiente que eu estava fazendo a prototipagem. Sem entrar em detalhes, posso lhe dizer que o teste verifica se o alarme de temperatura baixa, o aquecedor e o ventilador estão todos ligados quando a temperatura estiver “fria demais”.

Listagem 9-3

`EnvironmentControllerTest.java`

```
@Test
public void turnOnLoTempAlarmAtThreshold() throws Exception {
    hw.setTemp(WAY_TOO_COLD);
    controller.tic();
    assertTrue(hw.heaterState());
    assertTrue(hw.blowerState());
    assertFalse(hw.coolerState());
    assertFalse(hw.hiTempAlarm());
    assertTrue(hw.loTempAlarm());
}
```

Há, é claro, muitos detalhes aqui. Por exemplo, o que faz a função `tic`? Na verdade, prefiro que você não se atenha a isso ao ler esse teste, mas que se preocupe apenas se você concorda ou não se o estado final do sistema é consistente com a temperatura sendo “fria demais”.

Note que, ao ler o teste, seus olhos precisam ler e reler entre o nome do estado sendo verificado e a medida do estado sendo verificado. Você vê `heaterState` e, então, seus olhos deslizam para a direita, para `assertTrue`. Você vê `coolerState` e seus olhos devem se voltar para a

esquerda, para `assertFalse`. Isso é entediante e falível, além de dificultar a leitura do teste.

Aperfeiçoei a legibilidade deste teste consideravelmente na Listagem 9.4.

Listagem 9-4

`EnvironmentControllerTest.java (refatorado)`

```
@Test
public void turnOnLoTempAlarmAtThreshold() throws Exception {
    wayTooCold();
    assertEquals("HBchL", hw.getState());
}
```

É claro que criei uma função `wayTooCold` para ocultar o detalhe da função `tic`. Mas o que se deve notar é a estranha string em `assertEquals`. As letras maiúsculas significam “ligado” e as minúsculas “desligado”, e elas sempre seguem a seguinte ordem: {heater, blower, cooler, hi-temp-alarm, lo-temp-alarm}.

Mesmo que esse código esteja perto de violar a regra do mapeamento mental³, neste caso parece apropriado. Note que, uma vez conhecendo o significado, seus olhos deslizam sobre a string e você logo consegue interpretar os resultados. Torna-se quase uma satisfação ler o teste. Dê uma olhada na Listagem 9.5 e veja como é fácil entender esses testes.

Listagem 9-5

`EnvironmentControllerTest.java (seleção maior)`

```
@Test
public void turnOnCoolerAndBlowerIfTooHot() throws Exception {
    tooHot();
    assertEquals("hBChl", hw.getState());
}

@Test
public void turnOnHeaterAndBlowerIfTooCold() throws Exception {
    tooCold();
    assertEquals("HBchL", hw.getState());
}

@Test
public void turnOnHiTempAlarmAtThreshold() throws Exception {
    wayTooHot();
    assertEquals("hBCHl", hw.getState());
}

@Test
public void turnOnLoTempAlarmAtThreshold() throws Exception {
    wayTooCold();
    assertEquals("HBchL", hw.getState());
}
```

A função `getState` está na Listagem 9.6. Note que esse código não é muito eficiente. Para isso, provavelmente eu deveria ter usado um `StringBuffer`.

3. Evite o mapeamento mental, p. 25.

Listagem 9-6**MockControlHardware.java**

```
public String getState() {  
    String state = "";  
    state += heater ? "H" : "h";  
    state += blower ? "B" : "b";  
    state += cooler ? "C" : "c";  
    state += hiTempAlarm ? "H" : "h";  
    state += loTempAlarm ? "L" : "l";  
    return state;  
}
```

As `StringBuffers` são um pouco feias. Mesmo no código de produção irei evitá-las se o custo for pequeno; e você poderia argumentar que o custo do código na Listagem 9.6 é bastante pequeno. Contudo, esse aplicativo está claramente em um sistema integrado em tempo real, e é mais provável que os recursos do computador e da memória sejam muito limitados. O ambiente de teste, entretanto, não costuma ter limitações alguma.

Essa é a natureza do padrão duplo. Há coisa que você talvez jamais faça num ambiente de produção que esteja perfeitamente bem em um ambiente de teste. Geralmente, isso envolve questões de eficiência de memória e da CPU. Mas nunca de clareza.

Uma confirmação por teste

Há uma escola de pensamento⁴ que diz que cada função de teste em um teste JUnit deve ter um e apenas uma instrução de confirmação (`assert`). Essa regra pode parecer perversa, mas a vantagem pode ser vista na Listagem 9.5. Aqueles testes chegam a uma única conclusão que é fácil e rápida de entender.

Mas e a Listagem 9.2? Parece ilógico que poderíamos de alguma forma facilmente adicionar a confirmação se a saída está em XML e se ela contém certas substrings. Entretanto, podemos dividir o teste em dois, cada um com sua própria confirmação, como mostra a Listagem 9.7.

Listagem 9-7**SerializedPageResponderTest.java (Confirmação Única)**

```
public void testGetPageHierarchyAsXml() throws Exception {  
    givenPages("PageOne", "PageOne.ChildOne", "PageTwo");  
  
    whenRequestIsIssued("root", "type:pages");  
  
    thenResponseShouldBeXML();  
}  
  
public void testGetPageHierarchyHasRightTags() throws Exception {  
    givenPages("PageOne", "PageOne.ChildOne", "PageTwo");  
  
    whenRequestIsIssued("root", "type:pages");  
  
    thenResponseShouldContain(  
        "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"  
    );  
}
```

Note que mudei os nomes das funções para usar a convenção comum *dado-quando-então*. Isso facilita ainda mais a leitura dos testes. Infelizmente, como pode ver, dividir os testes pode gerar muito código duplicado.

Podemos usar o padrão Template Method para eliminar a duplicação e colocar as partes *dado/quando* na classe base e as partes *então* em derivadas diferentes. Ou poderíamos criar uma classe de teste completamente separada e colocar as partes *dado e quando* na função @Before e as partes *quando* em cada função @Test. Mas isso parece muito trabalho para uma questão tão pequena. No final, prefiro as confirmações múltiplas na Listagem 9.2.

Acho que a regra da confirmação única é uma boa orientação⁷. Geralmente tento criar uma linguagem de teste para um domínio específico que a use, como na Listagem 9.5. Mas não tenho receio de colocar mais de uma confirmação em um teste. Acho que a melhor coisa que podemos dizer é que se deve minimizar o número de confirmações em um teste.

Um único conceito por teste

Talvez a melhor regra seja que desejamos um único conceito em cada função de teste. Não queremos funções longas que saiam testando várias coisas uma após a outra. A Listagem 9.8 é um exemplo disso. Esse teste deve ser dividido em três independentes, pois ele testa três coisas distintas. Juntá-los todos na mesma função obriga o leitor compreender o objetivo de cada função presente e o que ela testa.

Listagem 9-8

```
/**
 * Miscellaneous tests for the addMonths() method.
 */
public void testAddMonths() {
    SerialDate d1 = SerialDate.createInstance(31, 5, 2004);

    SerialDate d2 = SerialDate.addMonths(1, d1);
    assertEquals(30, d2.getDayOfMonth());
    assertEquals(6, d2.getMonth());
    assertEquals(2004, d2.getYYYY());

    SerialDate d3 = SerialDate.addMonths(2, d1);
    assertEquals(31, d3.getDayOfMonth());
    assertEquals(7, d3.getMonth());
    assertEquals(2004, d3.getYYYY());

    SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1, d1));
    assertEquals(30, d4.getDayOfMonth());
    assertEquals(7, d4.getMonth());
    assertEquals(2004, d4.getYYYY());
}
```

As três funções de teste devem seguir provavelmente assim:

- *Dado o último dia de um mês com 31 dias (como maio):*

⁷. "Faça a manutenção do código!"

1. Quando você adicionar um mês cujo último dia seja o 30º (como junho), então a data deverá ser o dia 30 daquele mês, e não a 31.

2. Quando você adicionar dois meses àquela data cujo último dia seja o 31º, então a data deverá ser o dia 31.

• *Dado o último dia de um mês com 30 dias (como junho):*

1. Quando você adicionar um mês cujo último dia seja o 31º, então a data deverá ser dia 30 e não 31.

Explicado dessa forma, você pode ver que há uma regra geral oculta nos testes diversos. Ao incrementar o mês, a data não pode ser maior do que o último dia daquele mês. Isso implica que incrementar o mês em 28 de fevereiro resultaria em 28 de março. Esse teste está faltando e seria útil criá-lo.

Portanto, não são as confirmações múltiplas em cada seção da Listagem 9.8 que causa o problema. É o fato de que há mais de um conceito sendo testado. Assim, provavelmente, a melhor regra seja minimizar o número de confirmações por conceito e testar apenas um conceito por função de teste.

F.I.R.S.T.*

Testes limpos seguem outras cinco regras que formam o acrônimo em inglês acima (*Fast, Independent, Repeatable, Self-validating, Timely*):

Rapidez os testes devem ser rápidos. Devem executar com rapidez. Quando os testes rodam devagar, você não desejará executá-los com frequência. E, consequentemente, não encontrará problemas cedo o bastante para consertá-los facilmente. E você não se sentirá livre para limpar o código, que acabará se degradando.

Independência os testes não devem depender uns dos outros. Um teste não deve configurar as condições para o próximo. Você deve ser capaz de executar cada teste de forma independente e na ordem que desejar. Quando eles dependem uns dos outros, se o primeiro falhar causará um efeito dominó de falhas, dificultando o diagnóstico e ocultando os defeitos abaixo dele.

Repetitividade deve-se poder repetir os testes em qualquer ambiente. Você deve ser capaz de efetuar testes no ambiente de produção, no de garantia de qualidade e no seu notebook enquanto volta para casa de trem sem uma rede disponível. Caso seus testes não possam ser repetidos em qualquer ambiente, então você sempre terá uma desculpa para o motivo das falhas. E também perceberá que não consegue rodar os testes fora o ambiente adequado.

Autovalidação os testes devem ter uma saída booleana. Obtenham ou não êxito, você não deve ler um arquivo de registro para saber o resultado. Você não deve ter de comparar manualmente dois arquivos texto para ver se os testes foram bem sucedidos. Se os testes não possuírem autovalidação, então uma falha pode se tornar subjetiva, e executar os testes pode exigir uma longa validação manual.

Pontualidade os testes precisam ser escritos em tempo hábil. Devem-se criar os testes de unidade *imediatamente antes* do código de produção no qual serão aplicados. Se criá-los depois, o teste do código de produção poderá ficar mais difícil. Ou talvez você ache que um pouco do código de produção seja complexo demais para testar. Ou talvez você não crie o código de produção de maneira que possa ser testado.

Conclusão

Aqui abordamos apenas o início deste tópico. De fato, acho que deveria ter um livro inteiro sobre *Testes limpos*. Os testes são tão importantes para a saúde de um projeto quanto o código de produção. Talvez até mais, pois eles preservam e aumentam a flexibilidade, capacidade de manutenção e reutilização do código de produção. Portanto, mantenha seus testes sempre limpos. Trabalhe para torná-los sucintos e expressivos. Invente APIs de teste que funcionem como uma linguagem específica a um domínio que lhe ajude a criar seus testes.

Se deixar os testes de degradarem, seu código também irá. Mantenha limpos os seus testes.

Bibliografia

[RSpec]: *RSpec: Behavior Driven Development for Ruby Programmers*, Aslak Hellesøy, David Chelimsky, Pragmatic Bookshelf, 2008.

[GOF]: *Padrões de Projeto, Soluções Reutilizáveis de Software Orientado a Objetos*, Gamma et al., Addison-Wesley, 1996.

O que é uma classe?

Até agora, este livro se focou em como escrever bem linhas e blocos de código. Mergulhamos na composição apropriada para funções e como elas se interrelacionam. Mas apesar de termos falado bastante sobre a expressividade das instruções do código e as funções que o compõem, ainda não teremos um código limpo até discutirmos sobre os níveis mais altos da organização do código. Vamos falar agora sobre classes limpas.

10

Classes

com Jeff Langr



Até agora, este livro se focou em como escrever bem linhas e blocos de código. Mergulhamos na composição apropriada para funções e como elas se interrelacionam. Mas apesar de termos falado bastante sobre a expressividade das instruções do código e as funções que o compõem, ainda não teremos um código limpo até discutirmos sobre os níveis mais altos da organização do código. Vamos falar agora sobre classes limpas.

Organização das classes

Seguindo uma convenção padrão Java, uma classe deve começar com uma lista de variáveis. As públicas (*public*), estáticas (*static*) e constantes (*constants*), se existirem, devem vir primeiro. Depois vêm as variáveis estáticas privadas (*private static*), seguidas pelas instâncias privadas. Raramente há uma boa razão para se ter uma variável pública (*public*).

As funções públicas devem vir após a lista de variáveis. Gostamos de colocar as tarefas privadas chamadas por uma função pública logo depois desta. Isso segue a regra de cima para baixo e ajuda o programa a ser lido como um artigo de jornal.

Encapsulamento

Gostaríamos que nossas variáveis e funções fossem privadas, mas não somos radicais.

Às vezes, precisamos tornar uma variável ou função protegida (*protected*) de modo que possa ser acessada para testes. Para nós, o teste tem prioridade. Se um teste no mesmo pacote precisa chamar uma função ou acessar uma variável, a tornaremos protegida ou apenas no escopo do pacote. Entretanto, primeiro buscaremos uma forma de manter a privacidade. Perder o encapsulamento sempre é o último recurso.

As classes devem ser pequenas!

A primeira regra para classes é que devem ser pequenas. A segunda é que devem ser menores ainda. Não, não iremos repetir o mesmo texto do capítulo sobre *Funções*. Mas assim como com as funções, ser pequena também é a regra principal quando o assunto for criar classes. Assim como com as funções, nossa questão imediata é “O quanto pequena?”. Com as funções medimos o tamanho contando as linhas físicas. Com as classes é diferente. Contamos as *responsabilidades*¹.

A Listagem 10.1 realça uma classe, a *SuperDashboard*, que expõe cerca de 70 métodos públicos.

A maioria dos desenvolvedores concordaria que ela é um pouco grande demais em tamanho. Outros diriam que a *SuperDashboard* é uma “classe de deus”.

Listagem 10-1 Responsabilidades em excesso

```
public class SuperDashboard extends JFrame implements MetaDataUser
    public String getCustomizerLanguagePath()
    public void setSystemConfigPath(String systemConfigPath)
    public String getSystemConfigDocument()
    public void setSystemConfigDocument(String systemConfigDocument)
    public boolean getGuruState()
    public boolean getNoviceState()
    public boolean getOpenSourceState()
    public void showObject(MetaObject object)
    public void showProgress(String s)
```

1. [RDD].

Listagem 10-1 (continuação)
Responsabilidades em excesso

```
public boolean isMetadataDirty()
public void setIsMetadataDirty(boolean isMetadataDirty)
public Component getLastFocusedComponent()
public void setLastFocused(Component lastFocused)
public void setMouseSelectState(boolean isMouseSelected)
public boolean isMouseSelected()
public LanguageManager getLanguageManager()
public Project getProject()
public Project getFirstProject()
public Project getLastProject()
public String getNewProjectName()
public void setComponentSizes(Dimension dim)
public String getCurrentDir()
public void setCurrentDir(String newDir)
public void updateStatus(int dotPos, int markPos)
public Class[] DataBaseClasses()
public MetadataFeeder getMetadataFeeder()
public void addProject(Project project)
public boolean setCurrentProject(Project project)
public boolean removeProject(Project project)
public MetaProjectHeader getProgramMetadata()
public void resetDashboard()
public Project loadProject(String fileName, String projectName)
public void setCanSaveMetadata(boolean canSave)
public MetaObject getSelectedObject()
public void deselectObjects()
public void setProject(Project project)
public void editorAction(String actionPerformed, ActionEvent event)
public void setMode(int mode)
public FileManager getFileManager()
public void setFileManager(FileManager fileManager)
public ConfigManager getConfigManager()
public void setConfigManager(ConfigManager configManager)
public ClassLoader getClassLoader()
public void setClassLoader(ClassLoader classLoader)
public Properties getProps()
public String getUserHome()
public String getBaseDir()
public int getMajorVersionNumber()
public int getMinorVersionNumber()
public int getBuildNumber()
public MetaObject pasting(
    MetaObject target, MetaObject pasted, MetaProject project)
public void processMenuItems(MetaObject metaObject)
public void processMenuSeparators(MetaObject metaObject)
public void processTabPages(MetaObject metaObject)
public void processPlacement(MetaObject object)
public void processCreateLayout(MetaObject object)
public void updateDisplayLayer(MetaObject object, int layerIndex)
public void propertyEditedRepaint(MetaObject object)
public void processDeleteObject(MetaObject object)
public boolean getAttachedToDesigner()
public void processProjectChangedState(boolean hasProjectChanged)
public void processObjectNameChanged(MetaObject object)
public void runProject()
```

Listagem 10-1 (continuação)**Responsabilidades em excesso**

```

public void setAçowDragging(boolean allowDragging)
public boolean allowDragging()
public boolean isCustomizing()
public void setTitle(String title)
public IdeMenuBar getIdeMenuBar()
public void showHelper(MetaObject metaObject, String propertyName)
// ... many non-public methods follow ...
}

```

Mas e se SuperDashboard possuísse apenas os métodos da Listagem 10.2?

Listagem 10-2**Pequena o suficiente?**

```

public class SuperDashboard extends JFrame implements MetaDataUser
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}

```

Cinco não é muito, é? Neste caso é, pois apesar da pequena quantidade de métodos, a SuperDashboard possui muitas *responsabilidades*.

O nome de uma classe deve descrever quais responsabilidades ela faz. Na verdade, selecionar um nome é a primeira forma de ajudar a determinar o tamanho da classe. Se não derivarmos um nome conciso para ela, então provavelmente ela ficará grande. Quanto mais ambíguo for o nome da classe, maiores as chances de ela ficar com muitas responsabilidades. Por exemplo, nomes de classes que possuam palavras de vários sentidos, como Processador ou Gerenciador ou Super, geralmente indicam um acúmulo lastimável de responsabilidades.

Devemos também poder escrever com cerca de 25 palavras uma breve descrição da classe, sem usar as palavras “se”, “e”, “ou” ou “mas”. Como descreveríamos a SuperDashboard? “A SuperDashboard oferece acesso ao componente que foi o último utilizado e também nos permite acompanhar os números da versão e da compilação”. O primeiro “e” é uma dica de que a classe possui responsabilidades em excesso.

O Princípio da Responsabilidade Única

O Princípio da Responsabilidade Única (SRP, sigla em inglês)² afirma que uma classe ou módulo deve ter um, e apenas um, motivo para mudar. Este princípio nos dá uma definição de responsabilidade e uma orientação para o tamanho da classe. Estas devem ter apenas uma responsabilidade e um motivo para mudar.

Uma classe pequena como essa, a SuperDashboard, na Listagem 10.2, possui duas razões para ser alterada.

Primeiro, ela acompanha a informação sobre a versão, que precisa ser atualizada sempre que surgir uma nova distribuição do software. Segundo, ela gerencia os componentes Swing do Java

2. Pode-se ler muito mais sobre este princípio em [PPP].

(é um derivado do `JFrame`, a representação do Swing de uma janela GUI de alto nível). Sem dúvida iremos querer atualizar o número da versão se alterarmos qualquer código do Swing, mas o oposto não é necessariamente verdadeiro: podemos mudar as informações da versão baseando-nos nas modificações em outros códigos no sistema.

Tentar identificar as responsabilidades (motivos para alteração) costuma nos ajudar a reconhecer e criar abstrações melhores em nosso código. Podemos facilmente extrair todos os três métodos `SuperDashboard` que lidam com as informações de versão em uma classe separada chamada `Version`. (Veja a Listagem 10.3). A classe `Version` é um construtor que possui um alto potencial para reutilização em outros aplicativos!

Listagem 10-3 Classe com responsabilidade única

```
public class Version {
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}
```

O SRP é um dos conceitos mais importantes no desenvolvimento OO. É também um dos mais simples para se entender e aprender. Mesmo assim, estranhamente, o SRP geralmente é o princípio mais ignorado na criação de classes. Frequentemente, encontramos classes que faz muitas coisas. Por quê?

Fazer um software funcionar e torná-lo limpo são duas coisas bem diferentes.

A maioria de nós tem uma mente limitada, por isso, nós tentamos fazer com que nosso código possua mais do que organização e clareza. Isso é totalmente apropriado. Manter uma separação de questões é tão importante em nossas *atividades* de programação como em nossos programas.

O problema é que muitos de nós achamos que já terminamos se o programa funciona.

Esquecemo-nos da *outra* questão de organização e de clareza. Seguimos para o próximo problema em vez de voltar e dividir as classes muito cheias em outras com responsabilidades únicas.

Ao mesmo tempo, muitos desenvolvedores temem que um grande número de classes pequenas e de propósito único dificulte mais o entendimento geral do código. Eles ficam apreensivos em ter de navegar de classe em classe para descobrir como é realizada uma parte maior da tarefa.

Entretanto, um sistema com muitas classes pequenas não possui tantas partes separadas a mais como um com classes grandes. Há também bastante a se aprender num sistema com poucas classes grandes. Portanto, a questão é: você quer suas ferramentas organizadas em caixas de ferramentas com muitas gavetas pequenas, cada um com objetos bem classificados e rotulados? Ou poucas gavetas nas quais você coloca tudo?

Todo sistema expansível poderá conter uma grande quantidade de lógica e complexidade. O objetivo principal no gerenciamento de tal complexidade é organizá-la de modo que o desenvolvedor saiba onde buscar o que ele deseja e que precise entender apenas a complexidade que afeta diretamente um dado momento. Em contrapartida, um sistema com classes maiores de vários propósitos sempre nos atrasa insistindo que percorramos por diversas coisas que não precisamos saber no momento.

Reafirmando os pontos anteriores: desejamos que nossos sistemas sejam compostos por muitas classes pequenas, e não poucas classes grandes. Cada classe pequena encapsula uma única

responsabilidade, possui um único motivo para ser alterada e contribui com poucas outras para obter os comportamentos desejados no sistema.

Coesão

As classes devem ter um pequeno número de instâncias de variáveis. Cada método de uma classe deve manipular uma ou mais dessas variáveis. De modo geral, quanto mais variáveis um método manipular, mais coeso o método é para sua classe. Uma classe na qual cada variável é utilizada por um método é totalmente coesa.

De modo geral, não é aconselhável e nem possível criar tais classes totalmente coesas; por outro lado, gostaríamos de obter uma alta coesão. Quando conseguimos, os métodos e as variáveis da classe são co-dependentes e ficam juntas como um todo lógico.

Considere a implementação de uma `Stack` (pilha) na Listagem 10.4. A classe é bastante coesa. Dos três métodos, apenas `size()` não usa ambas as variáveis.

Listagem 10-4

`Stack.java` – uma classe coesa.

```
public class Stack {  
    private int topOfStack = 0;  
    List<Integer> elements = new LinkedList<Integer>();  
  
    public int size() {  
        return topOfStack;  
    }  
  
    public void push(int element) {  
        topOfStack++;  
        elements.add(element);  
    }  
  
    public int pop() throws PoppedWhenEmpty {  
        if (topOfStack == 0)  
            throw new PoppedWhenEmpty();  
        int element = elements.get(--topOfStack);  
        elements.remove(topOfStack);  
        return element;  
    }  
}
```

A estratégia para manter funções pequenas e listas de parâmetros curtas às vezes pode levar à proliferação de instâncias de variáveis que são usadas por uma sequência de métodos. Quando isso ocorre, quase sempre significa que há pelo menos uma outra classe tentando sair da classe maior na qual ela está. Você sempre deve tentar separar as variáveis e os métodos em duas ou mais classes de modo que as novas classes sejam mais coesas.

Manutenção de resultados coesos em muitas classes pequenas

Só o ato de dividir funções grandes em menores causa a proliferação de classes. Imagine uma função grande com muitas variáveis declaradas. Digamos que você deseje extrair uma pequena parte dela para uma outra função. Entretanto, o código a ser extraído usa quatro das variáveis declaradas na função. Você deve passar todas as quatro para a nova função como parâmetros?

Absolutamente não! Se convertêssemos aquelas quatro variáveis em instâncias de variáveis da classe, então poderíamos extrair o código sem passar *qualquer* variável. Seria fácil dividir a função em partes menores.

Infelizmente, isso também significa que nossas classes perderiam coesão, pois acumulariam mais e mais instâncias de variáveis que existiriam somente para permitir que as poucas funções as compartilhassem. Mas, espere! Se há poucas funções que desejam compartilhar certas variáveis, isso não as torna uma cada uma classe? Claro que sim. Quando as classes perdem coesão, divida-as!

Portanto, separar uma função grande em muitas pequenas geralmente nos permite dividir várias classes também. Isso dá ao nosso programa uma melhor organização e uma estrutura mais transparente.

Como exemplo do que quero dizer, usemos um exemplo respeitado há anos, extraído do maravilhoso livro *Literate Programming*³, de Knuth. A Listagem 10.5 mostra uma tradução em Java do programa PrintPrimes de Knuth. Para ser justo com Knuth, este não é o programa que ele escreveu, mas o que foi produzido pela sua ferramenta WEB. Uso-o por ser um grande ponto de partida para dividir uma função grande em muitas funções e classes menores.

Listagem 10-5 PrintPrimes.java

```
package literatePrimes;

public class PrintPrimes {
    public static void main(String[] args) {
        final int M = 1000;
        final int RR = 50;
        final int CC = 4;
        final int WW = 10;
        final int ORDMAX = 30;
        int P[] = new int[M + 1];
        int PAGENUMBER;
        int PAGEOFFSET;
        int ROWOFFSET;
        int C;
```

Listagem 10-5 (continuação)**PrintPrimes.java**

```

int J;
int K;
boolean JPRIME;
int ORD;
int SQUARE;
int N;
int MULT[] = new int[ORDMAX + 1];
J = 1;
K = 1;
P[1] = 2;
ORD = 2;
SQUARE = 9;
while (K < M) {
    do {
        J = J + 2;
        if (J == SQUARE) {
            ORD = ORD + 1;
            SQUARE = P[ORD] * P[ORD];
            MULT[ORD - 1] = J;
        }
        N = 2;
        JPRIME = true;
        while (N < ORD && JPRIME) {
            while (MULT[N] < J)
                MULT[N] = MULT[N] + P[N] + P[N];
            if (MULT[N] == J)
                JPRIME = false;
            N = N + 1;
        }
    } while (!JPRIME);
    K = K + 1;
    P[K] = J;
}
{
    PAGENUMBER = 1;
    PAGEOFFSET = 1;
    while (PAGEOFFSET <= M) {
        System.out.println("The First " + M +
                           " Prime Numbers --- Page " + PAGENUMBER);
        System.out.println("");
        for (ROWOFFSET = PAGEOFFSET; ROWOFFSET < PAGEOFFSET + RR; ROWOFFSET++) {
            for (C = 0; C < CC; C++)
                if (ROWOFFSET + C * RR <= M)
                    System.out.format("%10d", P[ROWOFFSET + C * RR]);
            System.out.println("");
        }
        System.out.println("\f");
        PAGENUMBER = PAGENUMBER + 1;
        PAGEOFFSET = PAGEOFFSET + RR * CC;
    }
}
}

```

Este programa, escrito como uma única função, é uma zona; possui uma estrutura com muitas endentações, uma abundância de variáveis estranhas e uma estrutura fortemente acoplada. No mínimo essa grande função deve ser dividida em algumas menores.

Da Listagem 10.6 a 10.8 mostra o resultado da divisão do código na Listagem 10.5 em classes e funções menores, e seleciona nomes significativos para as classes, funções e variáveis.

Listagem 10-6

PrimePrinter.java (refatorado)

```
package literatePrimes;

public class PrimePrinter {
    public static void main(String[] args) {
        final int NUMBER_OF_PRIMES = 1000;
        int[] primes = PrimeGenerator.generate(NUMBER_OF_PRIMES);

        final int ROWS_PER_PAGE = 50;
        final int COLUMNS_PER_PAGE = 4;
        RowColumnPagePrinter tablePrinter =
            new RowColumnPagePrinter(ROWS_PER_PAGE,
                                    COLUMNS_PER_PAGE,
                                    "The First " + NUMBER_OF_PRIMES +
                                    " Prime Numbers");

        tablePrinter.print(primes);
    }
}
```

Listagem 10-7

RowColumnPagePrinter.java

```
package literatePrimes;

import java.io.PrintStream;

public class RowColumnPagePrinter {
    private int rowsPerPage;
    private int columnsPerPage;
    private int numbersPerPage;
    private String pageHeader;
    private PrintStream printStream;

    public RowColumnPagePrinter(int rowsPerPage,
                                int columnsPerPage,
                                String pageHeader) {
        this.rowsPerPage = rowsPerPage;
        this.columnsPerPage = columnsPerPage;
        this.pageHeader = pageHeader;
        numbersPerPage = rowsPerPage * columnsPerPage;
        printStream = System.out;
    }
}
```

Listagem 10-7 (continuação)

RowColumnPagePrinter.java

```
public void print(int data[]) {  
    int pageNumber = 1;  
    for (int firstIndexOnPage = 0;  
         firstIndexOnPage < data.length;  
         firstIndexOnPage += numbersPerPage) {  
        int lastIndexOnPage =  
            Math.min(firstIndexOnPage + numbersPerPage - 1,  
                    data.length - 1);  
        printPageHeader(pageHeader, pageNumber);  
        printPage(firstIndexOnPage, lastIndexOnPage, data);  
        printStream.println("\f");  
        pageNumber++;  
    }  
}  
  
private void printPage(int firstIndexOnPage,  
                      int lastIndexOnPage,  
                      int[] data) {  
    int firstIndexOfLastRowOnPage =  
        firstIndexOnPage + rowsPerPage - 1;  
    for (int firstIndexInRow = firstIndexOnPage;  
         firstIndexInRow <= firstIndexOfLastRowOnPage;  
         firstIndexInRow++) {  
        printRow(firstIndexInRow, lastIndexOnPage, data);  
        printStream.println("");  
    }  
}  
  
private void printRow(int firstIndexInRow,  
                     int lastIndexOnPage,  
                     int[] data) {  
    for (int column = 0; column < columnsPerPage; column++) {  
        int index = firstIndexInRow + column * rowsPerPage;  
        if (index <= lastIndexOnPage)  
            printStream.format("%10d", data[index]);  
    }  
}  
  
private void printPageHeader(String pageHeader,  
                           int pageNumber) {  
    printStream.println(pageHeader + " --- Page " + pageNumber);  
    printStream.println("");  
}  
  
public void setOutput(PrintStream printStream) {  
    this.printStream = printStream;  
}
```

Listagem 10-8

PrimeGenerator.java

```
package literatePrimes;

import java.util.ArrayList;

public class PrimeGenerator {
    private static int[] primes;
    private static ArrayList<Integer> multiplesOfPrimeFactors;

    protected static int[] generate(int n) {
        primes = new int[n];
        multiplesOfPrimeFactors = new ArrayList<Integer>();
        set2AsFirstPrime();
        checkOddNumbersForSubsequentPrimes();
        return primes;
    }

    private static void set2AsFirstPrime() {
        primes[0] = 2;
        multiplesOfPrimeFactors.add(2);
    }

    private static void checkOddNumbersForSubsequentPrimes() {
        int primeIndex = 1;
        for (int candidate = 3;
             primeIndex < primes.length;
             candidate += 2) {
            if (isPrime(candidate))
                primes[primeIndex++] = candidate;
        }
    }

    private static boolean isPrime(int candidate) {
        if (isLeastRelevantMultipleOfNextLargerPrimeFactor(candidate)) {
            multiplesOfPrimeFactors.add(candidate);
            return false;
        }
        return isNotMultipleOfAnyPreviousPrimeFactor(candidate);
    }

    private static boolean
    isLeastRelevantMultipleOfNextLargerPrimeFactor(int candidate) {
        int nextLargerPrimeFactor = primes[multiplesOfPrimeFactors.size()];
        int leastRelevantMultiple = nextLargerPrimeFactor * nextLargerPrimeFactor;
        return candidate == leastRelevantMultiple;
    }

    private static boolean
    isNotMultipleOfAnyPreviousPrimeFactor(int candidate) {
        for (int n = 1; n < multiplesOfPrimeFactors.size(); n++) {
            if (!isMultipleOfNthPrimeFactor(candidate, n))
                return false;
        }
    }
}
```

Listagem 10-8 (continuação)

PrimeGenerator.java

```

        return true;
    }

    private static boolean
    isMultipleOfNthPrimeFactor(int candidate, int n) {
        return
            candidate == smallestOddNthMultipleNotLessThanCandidate(candidate, n);
    }

    private static int
    smallestOddNthMultipleNotLessThanCandidate(int candidate, int n) {
        int multiple = multiplesOfPrimeFactors.get(n);
        while (multiple < candidate)
            multiple += 2 * primes[n];
        multiplesOfPrimeFactors.set(n, multiple);
        return multiple;
    }
}

```

A primeira coisa que você deve perceber é que o programa ficou bem maior, indo de uma página para quase três. Há diversas razões para esse aumento. Primeiro, o programa refatorado usa nomes de variáveis mais longos e descriptivos.

Segundo, ele usa declarações de funções e classes como uma forma de adicionar comentários ao código. Terceiro, usamos espaços em branco e técnicas de formatação para manter a legibilidade.

Note como o programa foi dividido em três responsabilidades principais. O programa principal está sozinho na classe `PrimePrinter` cuja responsabilidade é lidar com o ambiente de execução. Ela será modificada se o método de chamada também for. Por exemplo, se esse programa fosse convertido para um serviço SOAP, esta seria a classe afetada.

O `RowColumnPagePrinter` sabe tudo sobre como formatar uma lista de números em páginas com uma certa quantidade de linhas e colunas. Se for preciso modificar a formatação da saída, então essa seria a classe afetada.

A classe `PrimeGenerator` sabe como gerar uma lista de números primos. Note que ela não foi feita para ser instanciada como um objeto. A classe é apenas um escopo prático no qual suas variáveis podem ser declaradas e mantidas ocultas. Se o algoritmo para o cálculo de números primos mudar, essa classe também irá.

Não a reescrivemos! Não começamos do zero e criamos um programa novamente. De fato, se olhar os dois programas mais de perto, verá que usam o mesmo algoritmo e lógica para efetuar as tarefas.

A alteração foi feita criando-se uma coleção de testes que verificou o comportamento *preciso* do primeiro programa. Então, foram feitas umas pequenas mudanças, uma de cada vez. Após cada alteração, executava-se o programa para garantir que o comportamento não havia mudado. Um pequeno passo após o outro e o primeiro programa foi limpo e transformado no segundo.

Como organizar para alterar

Para a maioria dos sistemas, a mudança é constante. A cada uma, corremos o risco de o sistema não funcionar mais como o esperado. Em um sistema limpo, organizamos nossas classes de modo a reduzir os riscos nas alterações.

A classe `Sql` na Listagem 10.9 gera strings no formato SQL adequado dado um metadados apropriado. É um trabalho contínuo e, como tal, ainda não suporta funcionalidade SQL, como as instruções `update`. Quando chegar a hora da classe SQL suportar uma instrução `update`, teremos de “abrir” essa classe e fazer as alterações. Qualquer modificação na classe tem a possibilidade de estragar outro código na classe. É preciso testar tudo novamente.

Listagem 10-9

Classe que precisa ser aberta para alteração

```
public class Sql {  
    public Sql(String table, Column[] columns)  
    public String create()  
    public String insert(Object[] fields)  
    public String selectAll()  
    public String findByKey(String keyColumn, String keyValue)  
    public String select(Column column, String pattern)  
    public String select(Criteria criteria)  
    public String preparedInsert()  
    private String columnList(Column[] columns)  
    private String valuesList(Object[] fields, final Column[] columns)  
    private String selectWithCriteria(String criteria)  
    private String placeholderList(Column[] columns)  
}
```

A classe `Sql` deve ser alterada quando adicionamos um novo tipo de instrução ou quando mudarmos os detalhes de um único tipo de instrução – por exemplo, se precisarmos modificar a funcionalidade do `select` para suportar “sub-selects”. Esses dois motivos para alteração significam que a classe `Sql` viola o SRP.

Podemos ver essa quebra da regra num simples ponto horizontal. O método realçado da SQL mostra que há métodos privados, como o `selectWithCriteria`, que parecem se relacionar apenas às instruções `select`.

O comportamento do método privado aplicado apenas a um pequeno subconjunto de uma classe por ser uma heurística útil para visualizar possíveis áreas para aperfeiçoamento. Entretanto, o indício principal para se tomar uma ação deve ser a alteração do sistema em si. Se a classe `Sql` for considerada logicamente completa, então não temos de nos preocupar em separar as responsabilidades. Se não precisamos da funcionalidade `update` num futuro próximo, então devemos deixar a `Sql` em paz. Mas assim que tivermos de “abrir” uma classe, devemos considerar consertar nosso projeto.

E se considerássemos uma solução como a da Listagem 10.10? Cada método de interface pública definido na `Sql` anterior na Listagem 10.9 foi refatorado para sua própria classe derivada `Sql`. Note que os métodos privados, como `valuesList`, vão diretamente para onde são necessários. O comportamento privado comum foi isolado para um par de classes utilitárias, `Where` e `ColumnList`.

Listagem 10-10

Várias classes fechadas

```
abstract public class Sql {  
    public Sql(String table, Column[] columns)  
    abstract public String generate();  
}  
  
public class CreateSql extends Sql {  
    public CreateSql(String table, Column[] columns)  
    @Override public String generate()  
}  
  
public class SelectSql extends Sql {  
    public SelectSql(String table, Column[] columns)  
    @Override public String generate()  
}  
  
public class InsertSql extends Sql {  
    public InsertSql(String table, Column[] columns, Object[] fields)  
    @Override public String generate()  
    private String valuesList(Object[] fields, final Column[] columns)  
}  
  
public class SelectWithCriteriaSql extends Sql {  
    public SelectWithCriteriaSql(  
        String table, Column[] columns, Criteria criteria)  
    @Override public String generate()  
}  
  
public class SelectWithMatchSql extends Sql {  
    public SelectWithMatchSql(  
        String table, Column[] columns, Column column, String pattern)  
    @Override public String generate()  
}  
  
public class FindByKeySql extends Sql {  
    public FindByKeySql(  
        String table, Column[] columns, String keyColumn, String keyValue)  
    @Override public String generate()  
}  
  
public class PreparedInsertSql extends Sql {  
    public PreparedInsertSql(String table, Column[] columns)  
    @Override public String generate()  
    private String placeholderList(Column[] columns)  
}  
  
public class Where {  
    public Where(String criteria)  
    public String generate()  
}  
  
public class ColumnList {  
    public ColumnList(Column[] columns)  
    public String generate()  
}
```

O código em cada classe se torna absurdamente simples. O tempo necessário para entendermos qualquer classe caiu para quase nenhum. O risco de que uma função possa prejudicar outra se torna ínfima. Do ponto de vista de testes, virou uma tarefa mais fácil testar todos os pontos lógicos nesta solução, já que as classes estão isoladas umas das outras.

Tão importante quanto é quando chega a hora de adicionarmos as instruções update, nenhuma das classes existentes precisam ser alteradas! Programamos a lógica para construir instruções update numa nova subclasse de Sql chamada UpdateSql. Nenhum outro código no sistema sofrerá com essa mudança.

Nossa lógica reestruturada da Sql representa o melhor possível. Ela suporta o SRP e outro princípio-chave de projeto de classe OO, conhecido como Princípio de Aberto-Fechado OCP⁴ (sigla em inglês). As classes devem ser abertas para expansão, mas fechadas para alteração. Nossa classe Sql reestruturada está aberta para permitir novas funcionalidades através da criação de subclasses, mas podemos fazer essa modificação ao mesmo tempo em que mantemos as outras classes fechadas. Simplesmente colocamos nossa classe UpdateSql em seu devido lugar. Desejamos estruturar nossos sistemas de modo que baguncemos o mínimo possível quando os atualizarmos. Num sistema ideal, incorporaríamos novos recursos através da expansão do sistema, e não alterando o código existente.

Como isolar das alterações

As necessidades mudarão, portanto o código também. Aprendemos na introdução à OO que há classes concretas, que contêm detalhes de implementação (código), e classes abstratas, que representam apenas conceitos. Uma classe do cliente dependente de detalhes concretos corre perigo quando tais detalhes são modificados. Podemos oferecer interfaces e classes abstratas para ajudar a isolar o impacto desses detalhes.

Depender de detalhes concretos gera desafios para nosso sistema de teste. Se estivermos construindo uma classe Portfolio e ela depender de uma API TokyoStockExchange externa para derivar o valor do portfolio, nossos casos de testes são afetados pela volatilidade dessa consulta. É difícil criar um teste quando obtemos uma resposta diferente a cada cinco minutos! Em vez de criar a Portfolio de modo que ela dependa diretamente de TokyoStockExchange, podemos criar uma interface StockExchange que declare um único método:

Desenvolvemos TokyoStockExchange para implementar essa interface. Também nos certificamos se o construtor de Portfolio recebe uma referência a StockExchange como parâmetro:

```
public Portfolio {  
    private StockExchange exchange;  
    public Portfolio(StockExchange exchange) {  
        this.exchange = exchange;  
    }  
    // ...  
}
```

Agora nosso teste pode criar uma implementação para testes da interface StockExchange que simula a TokyoStockExchange. Essa implementação fixará o valor para qualquer símbolo

que testarmos. Se nosso teste demonstrar a compra de cinco ações da Microsoft para nosso portfolio, programamos a implementação do teste para sempre retornar U\$ 100 dólares por ação. Nossa implementação de teste da interface StockExchange se reduz a uma simples tabela de consulta. Podemos, então, criar um teste que espere U\$ 500 dólares como o valor total do portfolio.

```
public class PortfolioTest {  
    private FixedStockExchangeStub exchange;  
    private Portfolio portfolio;  
  
    @Before  
    protected void setUp() throws Exception {  
        exchange = new FixedStockExchangeStub();  
        exchange.fix("MSFT", 100);  
        portfolio = new Portfolio(exchange);  
    }  
  
    @Test  
    public void GivenFiveMSFTTotalShouldBe500() throws Exception {  
        portfolio.add(5, "MSFT");  
        Assert.assertEquals(500, portfolio.value());  
    }  
}
```

Se o um sistema estiver desacoplado o bastante para ser testado dessa forma, ele também será mais flexível e terá maior capacidade de reutilização. A falta de acoplamento significa que os elementos de nosso sistema ficam melhores quando isolados uns dos outros e das alterações, facilitando o entendimento de cada elemento.

Ao minimizar o acoplamento dessa forma, nossas classes aderem a outro princípio de projeto de classes conhecido como Princípio da Inversão da Independência ,DIP⁵ (sigla em inglês). Basicamente, o DIP diz que nossas classes devem depender de abstrações, não de detalhes concretos.

Em vez de depender da implementação de detalhes da classe TokyoStockExchange, nossa classe Portfolio agora é dependente da interface StockExchange, que representa o conceito abstrato de pedir o preço atual de um símbolo. Essa isola todos os detalhes específicos da obtenção de tal preço, incluindo sua origem.

Bibliografia

[RDD]: *Object Design: Roles, Responsibilities, and Collaborations*, Rebecca WirfsBrock et al., Addison-Wesley, 2002.

[PPP]: *Agile Software Development: Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002.

[Knuth92]: *Literate Programming*, Donald E. Knuth, *Center for the Study of language and Information*, Leland Stanford Junior University, 1992.

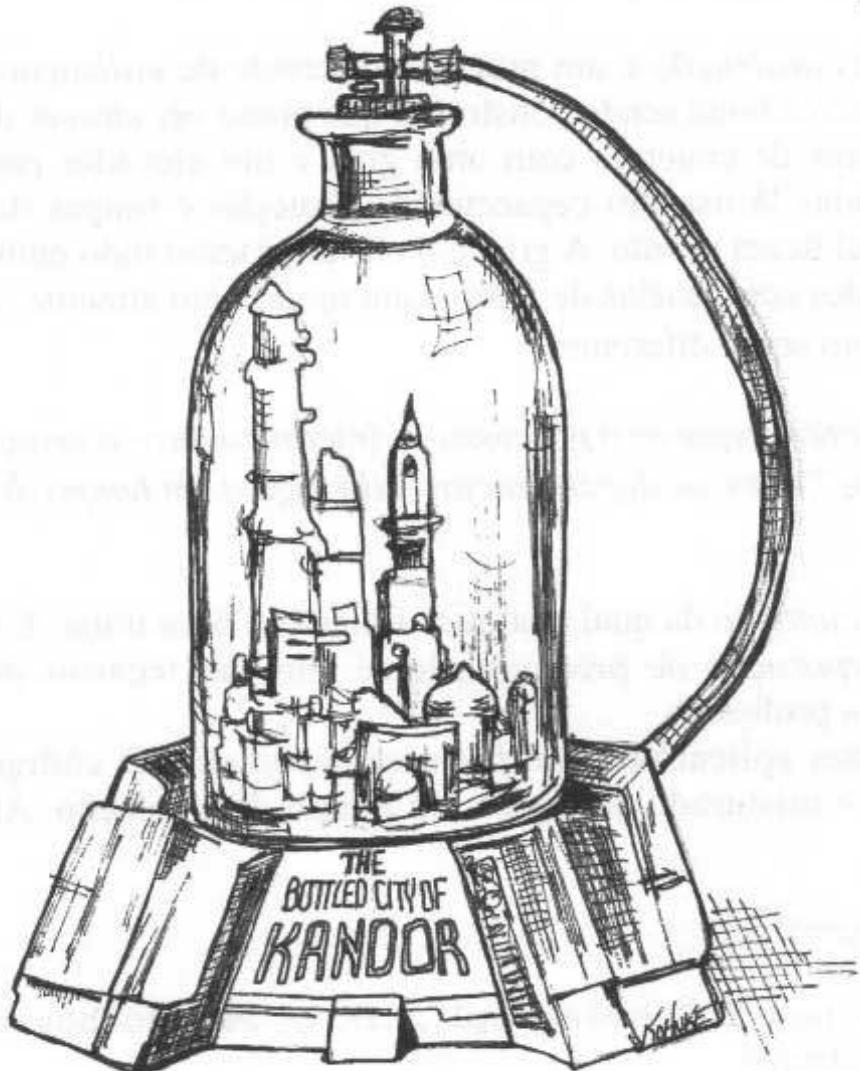
“Sistemas com sistemas dentro”

“Sistemas com sistemas dentro” é o nome que os engenheiros de software dão para aquela estrutura de software que é “difícil de entender”, ou seja, aquela que é composta por muitos módulos que dependem uns dos outros e que não se pode entender facilmente. Essa estrutura é muitas vezes resultado da evolução contínua de um sistema ao longo do tempo.

11

Sistemas

por Dr. Kevin Dean Wampler



“Complexidade mata. Ela suga a vida dos desenvolvedores, dificulta o planejamento, a construção e o teste dos produtos”.

—Ray Ozzie, CTO, Microsoft Corporation

Como você construiria uma cidade?

Conseguiria tratar de todos os detalhes sozinho? Provavelmente não. Até mesmo o gerenciamento de uma cidade é muito para uma pessoa só. Mesmo assim, as cidades funcionam (na maioria das vezes). Isso porque possuem equipes de pessoas que gerenciam partes específicas da cidade, o sistema de abastecimento de água, de energia, de trânsito, a segurança pública, as normas de construção, e assim por diante. Algumas dessas pessoas são responsáveis pela visão geral, enquanto outros se focam nos detalhes.

As cidades também funcionam porque progrediram em níveis apropriados de modularidade os quais possibilitaram que indivíduos e os “componentes” pudessem trabalhar de forma eficiente, mesmo sem ter noção da visão geral.

Embora equipes de software geralmente sejam organizadas dessa forma também, os sistemas nos quais trabalham não costuma ter a mesma divisão de preocupações e níveis de . Um código limpo nos ajuda a alcançar esses níveis mais baixos de . Neste capítulo, consideremos como manter o código limpo nos níveis de mais altos, o nível do *sistema*.

Separe a construção e o uso de um sistema

Primeiramente, considere que *construção* é um processo diferente de *utilização*. Na época em que escrevo este livro, há um novo hotel sendo construído que posso ver através da minha janela em Chicago. Hoje só há pilares de concreto com uma grua e um elevador preso do lado de fora. Todas as pessoas ocupadas lá usavam capacetes de proteção e roupas de trabalho. Em mais ou menos um ano, o hotel ficará pronto. A grua e o elevador terão indo embora. O edifício estará limpo, envolto em paredes com janelas de vidro e um tingimento atraente. As pessoas que trabalharão e ficarão ali também serão diferentes.

Os sistemas de software devem separar o processo de inicialização – a criação dos objetos do aplicativo e a “conexão” entre as dependências – da lógica em tempo de execução que vem após a inicialização.

O processo inicial é uma *preocupação* da qual qualquer aplicativo deva tratar. E será a primeira analisada neste capítulo. A *separação de preocupações* é uma das técnicas de projeto mais antigas e importantes em nossa profissão.

Infelizmente, a maioria dos aplicativos não faz essa separação. O código do processo de inicialização é específico e misturado na lógica em tempo de execução. Abaixo está um exemplo típico:

```
public Service getService() {
    if (service == null)
        service = new MyServiceImpl(...); // Padrão bom o suficiente
    para a maioria dos casos?
    return service;
}
```

Essa é a expressão INICIALIZAÇÃO/AVALIAÇÃO TARDIA, digna de vários méritos. Não visualizamos a operação geral da construção a menos que realmente usemos o objeto, e, como resultado, nosso tempo de inicialização pode ser mais rápido. Também garantimos que nunca seja retornado null.

Entretanto, agora temos uma dependência codificada permanentemente em `MyServiceImpl` e tudo o que seu construtor exige (o que omiti). Não podemos compilar sem resolver essas dependências, mesmo se nunca usarmos um objeto desse tipo tempo de execução.

Efetuar testes pode ser um problema. Se `MyServiceImpl` for um objeto grande, precisamos garantir que um TEST DOUBLE¹ ou MOCK OBJECT seja atribuído à área de operação antes deste método ser chamado no teste de unidade. Como temos lógica da construção misturada ao processamento normal em tempo de execução, devemos testar todos os caminhos da execução (por exemplo, o teste `null` e seu bloco). Ter essas duas responsabilidades significa que o método faz mais de uma coisa, portanto estamos violando, de certa forma, o *Princípio da Responsabilidade Única*.

Talvez o pior de tudo é que não sabemos se `MyServiceImpl` é o objeto correto em todas as classes, e foi isso que indiquei no comentário. Por que a classe que possui este método precisa enxergar o contexto global? Realmente *jamais* poderemos saber qual o objeto certo usar aqui? É possível que um tipo seja o certo para todos contextos?

É claro que uma ocorrência de INICIALIZAÇÃO-TARDIA não é um problema sério. Entretanto, costuma-se ter muitas instâncias de pequenas expressões como essa nos aplicativos. Devido a isso, a estratégia de configuração global (se houver uma) fica espalhada pelo aplicativo, com pouca modularidade e, geralmente, duplicação considerável.

Se formos cuidadosos ao construir sistemas bem estruturados e robustos, jamais devemos deixar que expressões *convenientes* prejudiquem a modularidade. O processo de inicialização da construção e atribuição de um objeto não são exceções. Devemos modularizar esse processo de separadamente da lógica normal em tempo de execução, e nos certificar que tenhamos uma estratégia global e consistente para resolver nossas dependências principais.

Separação do Main

Uma maneira de separar a construção do uso é simplesmente colocar todos os aspectos dela no `main` ou em módulos chamados por ele, e modelar o resto do sistema assumindo que todos os objetos foram construídos e atribuídos adequadamente (veja a Figura 11.1).

É fácil acompanhar o fluxo de controle. A função `main` constrói os objetos necessários para o sistema e, então, os passa ao aplicativo, que simplesmente os usa. Note a direção das setas de dependência cruzando a barreira entre o `main` e o aplicativo.

Todas apontam para a mesma direção, para longe do `main`. Isso significa que o aplicativo não enxerga o `main` ou o processo de construção, mas apenas espera que tudo seja devidamente construído.

Factories

É claro que de vez em quando precisamos passar o controle para o aplicativo *quando* um objeto for criado. Por exemplo, em um sistema de processamento de pedidos, o aplicativo deve criar instâncias do `LineItem` e adicionar a um `Order`. Neste caso, podemos usar o padrão ABSTRACT FACTORY² para passar o controle ao aplicativo quando for preciso criar os objetos `LineItem`, mas mantenha os detalhes dessa construção separada do código do aplicativo (veja a Figura 11.2).

1. [Mezzaros07].

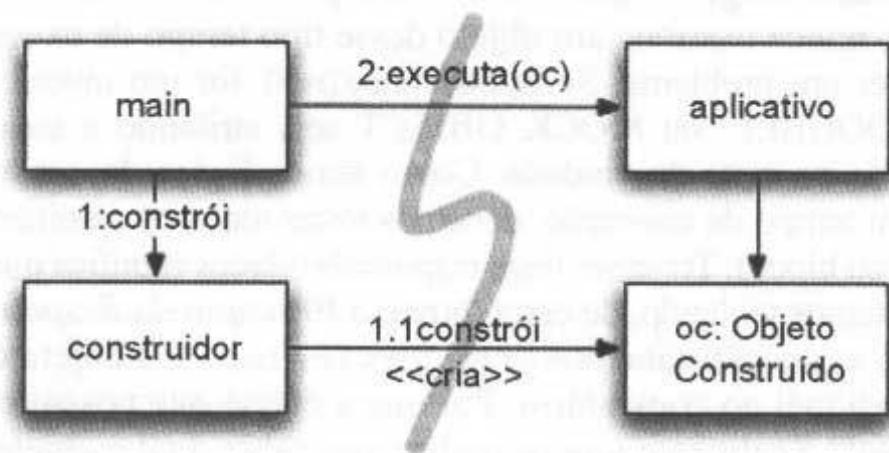


Figura 11.1: Separando o main() da construção de objetos

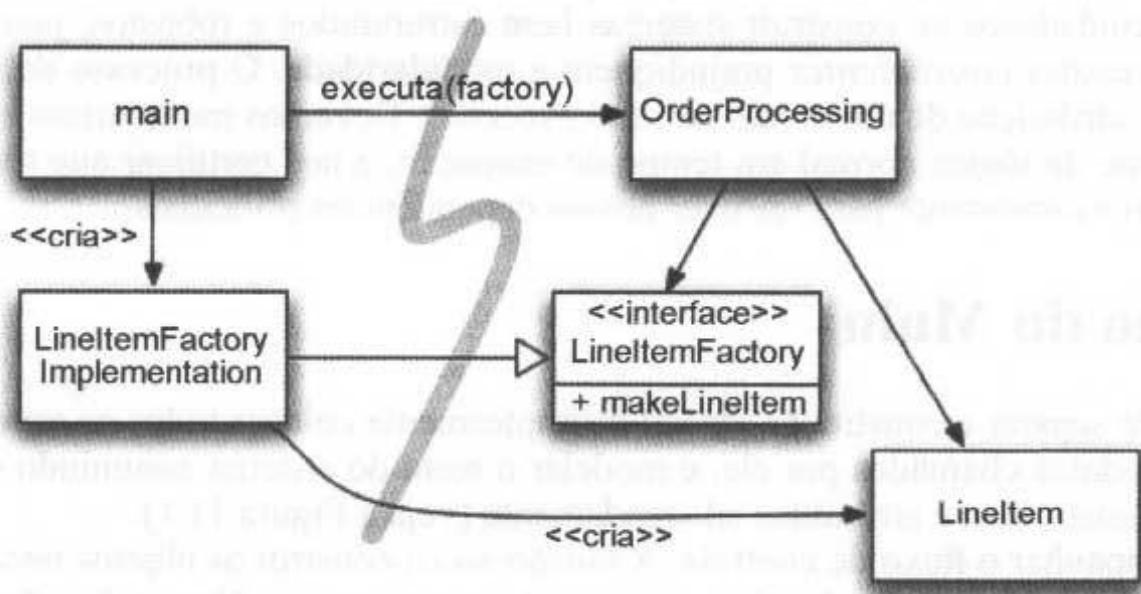


Figura 11.2: Separação da construção com uma factory

Novamente, observe que todas as dependências apontam de **main** para o aplicativo **OrderProcessing**. Isso significa que ele está desacoplado dos detalhes de como criar um **LineItem**. Essa capacidade é mantida em **LineItemFactory Implementation**, que está no mesmo lado da linha que o **main**. E mesmo assim o aplicativo possui controle total quando na criação de instâncias de **LineItem** e pode até oferecer parâmetros do construtor específicos ao aplicativo.

Injeção de dependência

Um mecanismo poderoso para separar a construção do uso é a *Injeção de Dependência* (DI, sigla em inglês), a aplicação da *Inversão de Controle* (IoC, sigla em inglês ao gerenciamento de dependência³). A IoC move as responsabilidades secundárias de um objeto para outros dedicados

3. Veja, [Fowler], por exemplo.

ao propósito que se deseja, dessa forma suportando o Princípio da *Responsabilidade Única*. Em vez disso, ela deve passar essa responsabilidade para outro mecanismo “dominante”, com isso invertendo o controle. Como essa configuração é uma preocupação global, esse mecanismo dominante geralmente será ou a rotina “principal” ou um *contêiner* de tarefa específica.

As consultas ao JNDI são implementações “parciais” da DI, na qual um objeto pede a um servidor de diretórios um “serviço” com um nome específico.

```
MyService myService = (MyService)(jndiContext.lookup("NameOfMyService"));
```

O objeto chamador não controla qual tipo será realmente retornado (contanto que ele implemente a interface apropriada, é claro), mas ele ainda determina ativamente a dependência.

A Injeção de Dependência Verdadeira vai mais além. A classe não determina diretamente suas dependências; ela fica completamente passiva e oferece métodos de escrita (*setters*) ou parâmetros de construtores (ou ambos) que serão usados para injetar as dependências. Durante o processo de construção, o contêiner de DI instancia os objetos necessários (geralmente sob demanda) e usa os parâmetros do construtor ou os métodos de escrita fornecidos para conectar as dependências.

Quais objetos dependentes são usados realmente são especificados por um arquivo de configuração ou diretamente programando-se no módulo de construção de tarefa específica.

O framework Spring oferece o melhor e mais conhecido contêiner de DI para Java⁴. Você define quais objetos conectar um ao outro em um arquivo de configuração, e depois solicita objetos específicos pelo nome no código Java. Logo veremos um exemplo.

Mas e sobre as virtudes da INICIALIZAÇÃO-TARDIA? Essa expressão, de vez em quando, ainda é útil com a DI. Primeiro, a maioria dos contêineres de DI não criará objetos até que sejam necessários. Segundo, muitos desses contêineres oferecem mecanismos para invocar factories ou construir proxies, que poderiam ser usados para sanar a AVALIAÇÃO-TARDIA e *otimizações*⁵ semelhantes.

Desenvolvimento gradual

Vilarejos viram pequenas cidades, que viram cidades grandes. No início as ruas são estreitas e quase não existem e, com tempo, elas são pavimentadas e, então, alargadas. Pequenos edifícios e terrenos vazios são substituídos por edificações maiores, algumas das quais acabarão virando arranha-céus.

No começo, não há serviços, como abastecimento de energia, água, esgoto e Internet (opa!). E só serão adicionados quando aumentar a densidade da população e de edificações.

Mas esse desenvolvimento não está livre de problemas. Quantas vezes, devido a um projeto de “melhoria” das avenidas você dirigiu por engarrafamentos e se perguntou “Por que não construíram as ruas largas o bastante desde o início?”.

Mas não podia ter sido de outra forma. Como justificar o custo de construção de uma via expressa de seis faixas passando no meio de uma cidade pequena já antecipando seu desenvolvimento? Quem *desejaria* tal avenida passando por sua cidade?

É mito dizer que podemos conseguir um sistema “correto de primeira”. Em vez disso, devemos implementar apenas os *fatos* de hoje e, então, refatorar e expandir o sistema, implementando novos

⁴ Veja o [Spring]. Também há uma framework Spring.NET.

fatos amanhã. Essa é a essência das agilidades iterativa e incremental. O desenvolvimento dirigido a testes, a refatoração e o código limpo que produzem fazem com que isso tudo funcione em nível de código. Mas e em nível de sistema? A estrutura do sistema não requer um pré-planejamento? Claro que sim, *ele* não pode crescer gradualmente do simples para o complexo, pode?

Se comparados aos sistemas físicos, os de software são únicos, e suas arquiteturas podem crescer gradualmente se mantivermos uma separação devida de preocupações.

Como veremos, é a natureza efêmera dos sistemas de software que possibilitam isso. Consideremos primeiro um contra-exemplo de uma arquitetura que não separa as preocupações de forma adequada.

As arquiteturas EJB1 e EJB2 originais são um bom exemplo e, devido a isso, geram obstáculos desnecessários para o crescimento orgânico. Considere uma *Entity Bean* para uma classe Bank frequente. Uma entity bean é uma representação, na memória, dos dados relacionais, ou seja, a linha de uma tabela.

Primeiro, você define uma interface local (no processo) ou remota (separada da JVM), que os clientes usariam. A Listagem 11.1 mostra uma possível interface local:

Listagem 11-1

Interface EJB2 local para um EJB da classe Bank

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;

public interface BankLocal extends java.ejb.EJBLocalObject {
    String getStreetAddr1() throws EJBException;
    String getStreetAddr2() throws EJBException;
    String getCity() throws EJBException;
    String getState() throws EJBException;
    String getZipCode() throws EJBException;
    void setStreetAddr1(String street1) throws EJBException;
    void setStreetAddr2(String street2) throws EJBException;
    void setCity(String city) throws EJBException;
    void setState(String state) throws EJBException;

    void setZipCode(String zip) throws EJBException;
    Collection getAccounts() throws EJBException;
    void setAccounts(Collection accounts) throws EJBException;
    void addAccount(AccountDTO accountDTO) throws EJBException;
}
```

Exibi diversos atributos para o endereço do Bank e uma coleção de contas (*account*) que há no banco (*bank*), cada uma com seus dados manipulados por um EJB de *Account* separado. A Listagem 11.2 mostra a classe de implementação correspondente para o bean de Bank.

Listagem 11-2**Implementação da Entity Bean do EJB2 correspondente**

```

package com.example banking;
import java.util.Collections;
import javax.ejb.*;

public abstract class Bank implements javax.ejb.EntityBean {
    // Business logic...
    public abstract String getStreetAddr1();
    public abstract String getStreetAddr2();
    public abstract String getCity();
    public abstract String getState();
    public abstract String getZipCode();
    public abstract void setStreetAddr1(String street1);
    public abstract void setStreetAddr2(String street2);
    public abstract void setCity(String city);
    public abstract void setState(String state);
    public abstract void setZipCode(String zip);
    public abstract Collection getAccounts();
    public abstract void setAccounts(Collection accounts);
    public void addAccount(AccountDTO accountDTO) {
        InitialContext context = new InitialContext();
        AccountHomeLocal accountHome = context.lookup("AccountHomeLocal");
        AccountLocal account = accountHome.create(accountDTO);
        Collection accounts = getAccounts();
        accounts.add(account);
    }
    // EJB container logic
    public abstract void setId(Integer id);
    public abstract Integer getId();
    public Integer ejbCreate(Integer id) { ... }
    public void ejbPostCreate(Integer id) { ... }
    // The rest had to be implemented but were usually empty:
    public void setEntityContext(EntityContext ctx) {}
    public void unsetEntityContext() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbLoad() {}
    public void ejbStore() {}
    public void ejbRemove() {}
}

```

Não mostrei a interface *LocalHome* correspondente – basicamente uma factory usada para criar objetos – e nem um dos possíveis métodos de localização (consulta, ou *queries*) que você pode adicionar. Por fim, você teve de criar um ou mais descritores de implementação que especifiquem os detalhes do mapeamento de objetos relacionais para um armazenamento permanente de dados, para a ação de transação desejada, para os limites de segurança, etc.

A lógica de negócio está fortemente acoplada ao “container” do aplicativo EJB2. Você precisa criar subclasses dos tipos do container e fornecer muitos métodos do tipo *lifecycle* exigidos pelo container.

Esse acoplamento ao container pesado dificulta o teste de unidade isolado.

É necessário fazer uma simulação do container, o que é difícil, ou gastar muito tempo implementando EJBs e testes em um servidor real. Reutilizar externamente e de modo eficiente a arquitetura EJB2 é impossível devido ao forte acoplamento.

Por fim, mesmo a programação orientada a objeto foi prejudicada. Um bean não pode herdar de outro. Note a lógica para adicionar uma nova conta. É comum nos beans do EJB2 definir os “objetos de transferência de dados” (DTOs, sigla em inglês) que são basicamente “*structs*” (estruturas) sem atividade alguma.

Isso costuma levar a tipos redundantes que possuem praticamente os mesmos dados, e requer códigos padronizados para copiar dados de um objeto para outro.

Preocupações transversais

A arquitetura EJB2 chega perto de uma divisão real de preocupações em algumas áreas. Por exemplo, a segurança, a comunicação desejada e alguns dos comportamentos de persistência são declarados dos descritores de implementação, independentemente do código-fonte.

Note que *preocupações* como persistência tendem a atravessar os limites naturais dos objetos de um domínio. Você deseja manter todos os seus objetos através da mesma estratégia, por exemplo, usando um SGBD⁶ versus bancos de dados não-relacionais, seguindo certas convenções de nomenclatura para tabelas e colunas, usando semânticas transacionais consistentes, etc.

Em princípio, você pode pensar em sua estratégia de persistência de uma forma modular e encapsulada. Mesmo assim, na prática, é preciso basicamente espalhar por vários objetos o mesmo código que implementa tal estratégia. Usamos o termo *preocupações transversais* para preocupações como essa. Novamente, o framework de persistência e a lógica de domínio (isolada) podem ser modulares. O problema é a minuciosa interseção desses domínios.

De fato, o modo como a arquitetura EJB trata da persistência, da segurança e das transações “antecipa” a *Programação Orientada a Aspecto* (POA)⁷ – uma abordagem de propósito geral para restaurar a modularidade para preocupações transversais.

Na POA, construções modulares chamadas *aspectos* especificam quais pontos no sistema devem ter seus comportamentos alterados de uma forma consistente para suportar uma determinada preocupação. Essa especificação é feita através de um mecanismo sucinto declarativo ou programático.

Usando a persistência como exemplo, você declararia quais objetos e atributos (ou *padrões* do tipo) devem ser mantidos e, então, delegar as tarefas de persistência ao seu framework de persistência. O framework da POA efetua as alterações de comportamento de modo *não-invasivo*⁸ no código a ser alterado. Vejamos três aspectos ou mecanismos voltados a aspectos em Java.

Proxies para Java

Os proxies para Java são adequados para situações simples, como empacotar chamadas de métodos em objetos ou classes individuais. Entretanto, os proxies dinâmicos oferecidos no JDK só funcionam com interfaces. Para usar proxies em classe, é preciso usar uma biblioteca de manipulação de Bytecode, como CGLIB, ASM ou Javassist⁹.

A Listagem 11.3 mostra o esqueleto para um Proxy do JDK que ofereça suporte à persistência para nosso aplicativo Bank, cobrindo apenas os métodos para obter e alterar a lista de contas.

Listagem 11-3

Exemplo de proxy do JDK

```
// Bank.java (suppressing package names...)
import java.util.*;

// The abstraction of a bank.
public interface Bank {
    Collection<Account> getAccounts();
    void setAccounts(Collection<Account> accounts);
}

// BankImpl.java
import java.util.*;

// The "Plain Old Java Object" (POJO) implementing the abstraction.
public class BankImpl implements Bank {
    private List<Account> accounts;

    public Collection<Account> getAccounts() {
        return accounts;
    }
    public void setAccounts(Collection<Account> accounts) {
        this.accounts = new ArrayList<Account>();
        for (Account account: accounts) {
            this.accounts.add(account);
        }
    }
}

// BankProxyHandler.java
import java.lang.reflect.*;
import java.util.*;

// "InvocationHandler" required by the proxy API.
public class BankProxyHandler implements InvocationHandler {
    private Bank bank;

    public BankHandler (Bank bank) {
        this.bank = bank;
    }

    // Method defined in InvocationHandler
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        String methodName = method.getName();
        if (methodName.equals("getAccounts")) {
            bank.setAccounts(getAccountsFromDatabase());
            return bank.getAccounts();
        } else if (methodName.equals("setAccounts")) {
            bank.setAccounts((Collection<Account>) args[0]);
            setAccountsToDatabase(bank.getAccounts());
            return null;
        } else {
            ...
        }
    }
}
```

Listagem 11-3 (continuação)

Exemplo de proxy do JDK

```
// Lots of details here:
protected Collection<Account> getAccountsFromDatabase() { ... }
protected void setAccountsToDatabase(Collection<Account> accounts) { ... }
}

// Somewhere else...

Bank bank = (Bank) Proxy.newProxyInstance(
    Bank.class.getClassLoader(),
    new Class[] { Bank.class },
    new BankProxyHandler(new BankImpl()));
```

Definimos uma interface `Bank`, que será *empacotada* pelo proxy, e um *Objeto Java Antigo e Simples* (POJO, sigla em inglês), chamado `BankImpl`, que implementa a lógica de negócio. (Falaremos depois sobre POJOs).

A API do Proxy requer um objeto `InvocationHandler` ao qual ela chama para implementar quaisquer chamadas ao método `Bank` feita pelo proxy. Nossa `BankProxyHandler` usa a API de Reflexão do Java para mapear as chamadas aos métodos genéricos correspondentes em `BankImpl`, e assim por diante.

Há *bastante* código aqui que é relativamente complicado, mesmo para esse caso simples¹⁰. Usar uma das bibliotecas de manipulação de bytes é igualmente desafiador. Este “volume” de código e sua complexidade são duas das desvantagens de usar proxies. Eles dificultam a criação de um código limpo! Ademais, os proxies não oferecem um mecanismo para a especificação de certas “partes” para execução através de todo o sistema – necessário para uma solução de POA real¹¹.

Frameworks de POA puramente Java

Felizmente, há ferramentas que podem tratar automaticamente da maioria dos proxies padronizados. Estes são usados internamente em diversos frameworks como, por exemplo, POA com Spring e com JBoss, para implementar aspectos puramente em Java¹². No Spring, você escreve sua lógica de negócio como *Objeto Java Antigo e Simples*. Os POJOs estão simplesmente centrados em seus domínios. Eles não possuem dependências nas estruturas empresariais (ou qualquer outro domínio). Dessa forma, em tese, eles são mais simples e fáceis de testar. A simplicidade relativa facilita a garantia de que você esteja implementando as *user stories* correspondentes de modo correto e a manutenção e o desenvolvimento do código em *user stories* futuras.

Você incorpora a estrutura necessária do aplicativo, incluindo as preocupações transversais, como persistência, transações, segurança, fazer cache, transferência automática por falha (*failover*), etc., usando arquivos de configuração com declarações ou APIs. Em muitos casos, você realmente especifica os aspectos da biblioteca do Spring ou do JBoss, onde o framework trata dos mecanismos do uso dos proxies em Java ou de bibliotecas de Bytecode transparentes ao usuário. Essas declarações controlam o contêiner de injeção de dependência, que instancia os objetos principais e os conecta sob demanda.

A Listagem 11.4 mostra um fragmento típico de um arquivo de configuração do Spring V2.5, o `app.xml`¹³:

10. Para exemplos mais detalhados do API do Proxy e de seu uso, consulte, por exemplo, [Goetz].

Listagem 11-4

Arquivo de configuração do Spring 2.X

```

<beans>
  ...
  <bean id="appDataSource"
    class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close"
    p:driverClassName="com.mysql.jdbc.Driver"
    p:url="jdbc:mysql://localhost:3306/mydb"
    p:username="me"/>

  <bean id="bankDataAccessObject"
    class="com.example.banking.persistence.BankDataAccessObject"
    p:dataSource-ref="appDataSource"/>

  <bean id="bank"
    class="com.example.banking.model.Bank"
    p:dataAccessObject-ref="bankDataAccessObject"/>
  ...
</beans>
```

Cada “bean” é como uma parte de uma “Boneca Russa”, com um objeto domain para um Bank configurado com um proxy (empacotado) por um objeto de acesso a dados (DAO, sigla em inglês), que foi configurado com um proxy pela fonte de dados do driver JDBC. (Veja a Figura 11.3).

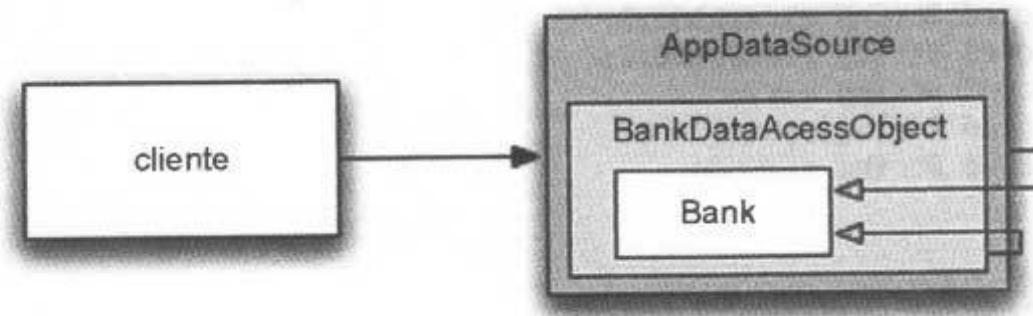


Figura 11-3
A “Boneca Russa” do decorators

O cliente acha que está chamando `getAccounts()` em um objeto Bank, mas na verdade está se comunicando com a parte mais externa de um conjunto de objetos DECORATOR¹⁴ aninhados que estendem o comportamento básico do POJO Bank. Poderíamos adicionar outros DECORATOR para transações, efetuação de cache, etc.

No aplicativo, são necessárias algumas linhas para solicitar o container da DI para os objetos no nível mais superior do sistema, como especificado no arquivo XML.

```

XmlBeanFactory bf =
new XmlBeanFactory(new ClassPathResource("app.xml", getClass()));
Bank bank = (Bank) bf.getBean("bank");
```

Devido à necessidade de tão poucas linhas do código Java específico para o Spring, o *aplicativo* está quase completamente desacoplado do Spring, eliminando todos os problemas de acoplamento de sistemas como o EJB2.

Embora o XML possa ser detalhado e difícil de ler¹⁵, a “política” especificada nesses arquivos de configuração é mais simples do que o proxy complicado e a lógica do aspecto que fica oculta e é criada automaticamente. Esse tipo de arquitetura é tão urgente que levou frameworks como o Spring a efetuarem uma reformulação completa do padrão EJB para a versão 3. O EJB3 majoritariamente segue o modelo do Spring de suporte a declarações a preocupações transversais usando os arquivos de configuração XML e/ou as anotações do Java 5.

A Listagem 11.5 mostra o EJB3¹⁶ com nosso objeto Bank reescrito.

Listagem 11-5

Um EJB do Bank no EJB3

```
package com.example.banking.model;
import javax.persistence.*;
import java.util.ArrayList;
import java.util.Collection;

@Entity
@Table(name = "BANKS")
public class Bank implements java.io.Serializable {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;

    @Embeddable // An object "inlined" in Bank's DB row
    public class Address {
        protected String streetAddr1;
        protected String streetAddr2;
        protected String city;
        protected String state;
        protected String zipCode;
    }

    @Embedded
    private Address address;

    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER,
               mappedBy="bank")
    private Collection<Account> accounts = new ArrayList<Account>();

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public void addAccount(Account account) {
        account.setBank(this);
        accounts.add(account);
    }

    public Collection<Account> getAccounts() {
        return accounts;
    }
}
```

¹⁵ Pode-se simplificar o exemplo através de mecanismos que explorem a convenção acima da configuração e as anotações do Java 5 de modo a reduzir a

Listagem 11-5 (continuação)**Um EJB do Bank no EBJ3**

```
public void setAccounts(Collection<Account> accounts) {  
    this.accounts = accounts;  
}
```

O código está muito mais limpo do que o do EJB2 original. Alguns dos detalhes da entidade ainda estão lá, contidos nas anotações. Entretanto, como nenhuma daquelas informações está fora das anotações, o código fica limpo, claro e, portanto, fácil de testar, fazer a manutenção, etc.

Se desejar, você pode mover algumas ou todas as informações nas anotações sobre a persistência para os descritores de implementação XML, deixando um POJO puro. Se os detalhes de mapeamento da persistência não mudarem frequentemente, muitas equipes podem optar por manter as anotações, entretanto com muito menos efeitos prejudiciais se comparado ao modo invasivo do EJB2.

Aspectos do AspectJ

Finalmente, a ferramenta mais famosa para a separação de preocupações através de aspectos é a linguagem AspectJ¹⁷, uma extensão do Java que oferece suporte de “primeira-classe” a aspectos como construtores de modularidade. A abordagem puramente Java oferecida pela POA em Spring e em JBoss são o suficiente para 80-90% dos casos nos quais os aspectos são mais úteis. Entretanto, o AspectJ proporciona uma série de ferramentas rica e poderosa para separar preocupações. Sua desvantagem é ter de usar várias ferramentas novas e aprender a estrutura e o uso das expressões de uma nova linguagem.

Uma “forma de anotação” do AspectJ recentemente lançada, na qual usam-se anotações do Java 5 para definir aspectos usando um código puramente Java, ameniza parcialmente essa questão do uso de novas ferramentas. Ademais, o Framework do Spring possui uma série de recursos que facilita ainda mais, para uma equipe com experiência limitada em AspectJ, a inclusão de aspectos baseados em anotações.

Uma discussão completa sobre o AspectJ foge do escopo deste livro. Para isso, consulte [AspectJ], [Colyer] e [Spring].

Testes na arquitetura do sistema

A capacidade de separar preocupações através de abordagens voltadas a aspectos não pode ser exagerada. Se puder escrever a lógica de domínio de seu aplicativo usando POJOs, desacoplados de qualquer preocupações acerca da arquitetura em nível de código, então é possível testar sua arquitetura. Você pode desenvolvê-la do simples ao sofisticado, se for preciso, através da adoção de novas tecnologias sob demanda. Não é necessário fazer um BDUF (*Big Design Up Front*). De fato, um BDUF é até mesmo prejudicial por inibir a adaptação a mudanças devido à resistência psicológica de descartar os esforços anteriores e devido à forma pela qual a escolha da arquitetura influencia as ideias seguintes sobre o design.

Arquitetos de construções têm de fazer um BDUF porque não é prático fazer alterações radicais na arquitetura para uma estrutura física grande após o início da construção¹⁸.

17. Consulte [AspectJ] e [Colyer].

18. Não há uma quantidade significativa de pesquisa repetitiva e de discussão de detalhes, mesmo após o início da construção.

Embora o software tenha sua própria mecânica²⁰, é economicamente prático fazer alterações radicais se a estrutura separa de forma eficiente suas preocupações.

Isso significa que podemos iniciar um projeto de Software com uma arquitetura simples, porém bem desacoplada, fornecendo rapidamente *user stories* que funcionem e, então, adicionando mais infra-estrutura conforme o desenvolvemos. Alguns dos maiores sites da Web alcançaram um grau de disponibilidade e performance muito altos através do uso de cache sofisticado de dados, segurança, virtualização, e assim por diante, tudo feito de forma eficiente e flexível porque os projetos com acoplamento mínimo são apropriadamente *simples* em cada nível de e de escopo.

É claro que isso não quer dizer que iniciamos um projeto sem algum planejamento. Temos certas expectativas a respeito do escopo, objetivos e programação gerais para o projeto, assim como para a estrutura geral do sistema final. Todavia, devemos manter a capacidade de alterações durante o desenvolvimento no caso de aperfeiçoamentos.

A arquitetura anterior EJB é uma das muitas APIs bem conhecidas que foram desenvolvidas de modo exagerado e que comprometeram a separação de preocupações. Mesmo tais APIs podem ser destrutivas se realmente não forem necessárias. Uma boa API deve ficar *oculta* na maioria das vezes, portanto a equipe expande a maioria de seus esforços criativos centralizados nas *user stories* sendo implementadas. Caso contrário, os limites da arquitetura inibirão a entrega eficiente do melhor serviço ao consumidor.

Para recapitular:

Uma arquitetura de sistema adequada consiste em domínios modularizados de preocupações, cada um implementado com POJOs – Objetos Java Antigos e Simples, ou outros. Os diferentes domínios são integrados uns aos outros com ferramentas de Aspectos ou voltadas para eles pouco invasivas. Pode-se testar essa arquitetura da mesma forma que se faz com o código.

Otimize a tomada de decisões

Modularidade e separação de preocupações descentralizam o gerenciamento e possibilitam a tomada de decisões. Em um sistema consideravelmente grande, seja uma cidade ou um projeto de software, ninguém pode tomar todas as decisões.

Todos sabemos que o melhor é designar responsabilidades às pessoas mais qualificadas. Geralmente nos esquecemos que também é melhor adiar as decisões até o último momento. Isso não é ser preguiçoso ou irresponsável, mas permitir-nos tomar decisões quando tivermos as melhores informações possíveis.

Uma decisão prematura é tomada sem muitas informações adequadas. Teremos muito menos retorno do consumidor, reflexão sobre o projeto e experiência com nossas escolhas de implementação se decidirmos muito cedo.

A agilidade oferecida por um sistema POJO com preocupações modularizadas nos permite uma otimização – decisões na hora certa – baseando-se nas informações mais recentes. Além de também reduzir complexidade dessas decisões.

20. O termo *mecânica de software* foi usado primeiramente por [Kolence].

Use padrões sabiamente quando eles adicionarem um valor demonstrativo

É uma coisa maravilhosa assistir à construção de uma infra-estrutura devido ao ritmo com que as novas estruturas são construídas (mesmo num inverno rigoroso) e aos projetos extraordinários possíveis com a tecnologia atual. Uma construção é um mercado maduro com partes altamente otimizadas, métodos e padrões que evoluíram sob pressão por séculos.

Muitas equipes usavam a arquitetura EJB2 porque ela era padrão, mesmo com o advento de planejamentos mais diretos e leves. Já vi equipes ficarem obcecadas com diversos padrões muito populares e perderem o foco no quesito de implementação voltado para seus consumidores.

Os padrões facilitam a reutilização de ideias e componentes, recrutam pessoas com experiência considerável, encapsulam boas ideias e conectam os componentes. Entretanto, o processo de criação de padrões pode, às vezes, ser muito longo para que o mercado fique à espera deles, e alguns padrões acabam se desviando das necessidades reais das pessoas a quem eles pretendem servir.

Sistemas precisam de linguagens específicas a um domínio

A construção de infra-estruturas, assim como a maioria dos domínios, desenvolveu uma linguagem rica com vocabulário, expressões e padrões²¹ que expressam informações essenciais de maneira clara e concisa. Houve recentemente na área de softwares uma retomada do interesse pela criação de *Linguagens Específicas a um Domínio* (DSLs, sigla em inglês)²², que são linguagens separadas, pequenos scripts ou APIs em linguagens padrão que permitem a escrita do código num formato que possa ser lido como uma prosa redigida por um especialista em domínio.

Uma boa DSL minimiza a “distância de comunicação” entre o conceito de um domínio e o código que o implementa, assim como as práticas flexíveis otimizam a comunicação entre os membros de uma equipe e a desta com suas partes interessadas. Se estiver implementando a lógica de um domínio na mesma linguagem usada pelo especialista em domínios, haverá menos risco de erro na tradução do domínio para a implementação.

As Linguagens Específicas a um Domínio permitem todos os níveis de e todos os domínios no aplicativo a ser expresso como POJOs, desde um nível mais alto até os detalhes de baixo nível.

Conclusão

Os sistemas também devem ser limpos. Uma arquitetura invasiva afeta a agilidade e sobrepuja a lógica do domínio que, quando ofuscada, perde-se qualidade porque os bugs se escondem mais facilmente e dificulta a implementação. Se a agilidade for comprometida, a produtividade também será e se perderão as vantagens do TDD. EM todos os níveis de , o objetivo deve estar claro. Isso só acontecerá se você criar POJOs e usar mecanismos voltados a aspectos para incorporar de modo não invasivo outras preocupações de implementação.

21. O trabalho de [Alexander] exerceu influência especialmente na comunidade de softwares.

22. Um IDEL é, por exemplo, UMLModel é um bom exemplo de uma API JAVA que cria uma DSL.

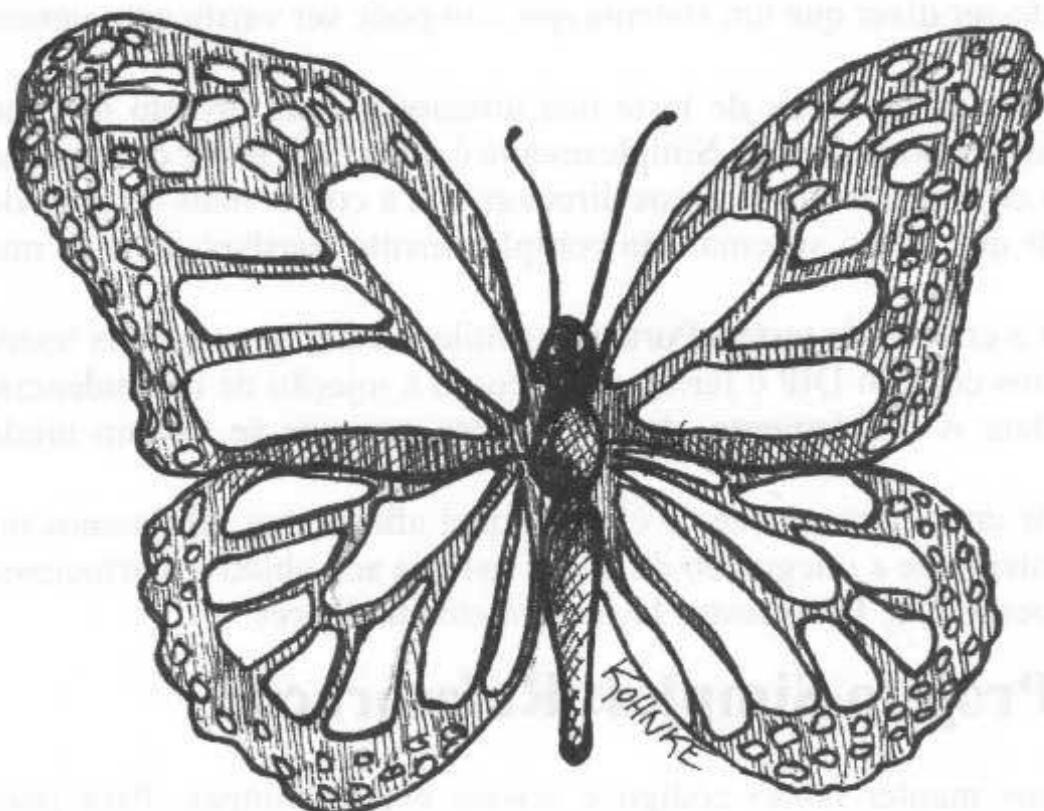
Esteja você desenvolvendo sistemas ou módulos individuais, jamais se esqueça de usar a coisa mais simples que funcione.

Bibliografia

- [Alexander]: Christopher Alexander, *A Timeless Way of Building*, Oxford University Press, New York, 1979.
- [AOSD]: Aspect-Oriented Software Development port, <http://aosd.net>
- [ASM]: Página do ASM, <http://asm.objectweb.org/>
- [AspectJ]: <http://eclipse.org/aspectj>
- [CGLIB]: Code Generation Library, <http://cglib.sourceforge.net/>
- [Colyer]: Adrian Colyer, Andy Clement, George Hurley, Mathew Webster, *Eclipse AspectJ*, Person Education, Inc., Upper Saddle River, NJ, 2005.
- [DSL]: Domain-specific programming language, http://en.wikipedia.org/wiki/Domain-specific_programming_language
- [Fowler]: Inversion of Control Containers and the Dependency Injection pattern, <http://martinfowler.com/articles/injection.html>
- [Goetz]: Brian Goetz, Java Theory and Practice: Decorating with Dynamic Proxies, <http://www.ibm.com/developerworks/java/library/j-jtp08305.html>
- [Javassist]: Página do Javassist, <http://www.csg.is.titech.ac.jp/~chiba/javassist/>
- [JBoss]: Página do JBoss, <http://jboss.org>
- [JMock]: JMock—A Lightweight Mock Object Library for Java, <http://jmock.org>
- [Kolence]: Kenneth W. Kolence, Software physics and computer performance measurements, Proceedings of the ACM annual conference—Volume 2, Boston, Massachusetts, pp. 1024–1040, 1972.
- [Spring]: The Spring Framework, <http://www.springframework.org>
- [Mezzaros07]: XUnit Patterns, Gerard Mezzaros, Addison-Wesley, 2007.
- [GOF]: Padrões de Projeto, Soluções Reutilizáveis de Software Orientado a Objetos, Gamma et al., Addison-Wesley, 1996

Emergência

por Jeff Langr



Obtendo clareza através de um processo de emergência

E se houvesse quatro regras simples que você pudesse usar para lhe ajudar na criação de bons projetos enquanto trabalhasse? E se ao seguir essas regras você obtivesse conhecimentos sobre a estrutura e o modelo de seu código, facilitando a aplicação de princípios, como o SRP e o DIP? E se essas quatro regras facilitassem a emergência de bons projetos? Muitos de nós achamos que as quatro regras do Projeto Simples¹ de Kent Beck sejam de ajuda considerável na criação de um software bem projetado.

De acordo com Kent, um projeto é “simples” se seguir as seguintes regras:

- Efetuar todos os testes;
- Sem duplicação de código;
- Expressar o propósito do programador
- Minimizar o número de classes e métodos

Essas regras estão em ordem de relevância.

Regra 1 de Projeto Simples: Efetue todos os testes

Primeiro e acima de tudo, um projeto deve gerar um sistema que funcione como o esperado. Um sistema pode ter um projeto perfeito no papel, mas se não há uma maneira simples de verificar se ele realmente funciona como o planejado, então o que está escrito é dubitável.

Um sistema que é testado detalhadamente e que passa em todos os testes é um sistema passível de testes. Isso pode parecer óbvio, mas é importante. Os sistemas que não podem ser testados não podem ser verificados. Logo, pode-se dizer que um sistema que não pode ser verificado, jamais deveria ser implementado.

Felizmente, tornar nossos sistemas passíveis de teste nos direciona a um projeto no qual nossas classes sejam pequenas e de propósito único. Simplesmente é mais fácil testar classes que sigam o SRP. Quanto mais testes criarmos, mais seremos direcionados a coisas mais simples de serem testadas. Portanto, garantir que nosso sistema seja completamente passível de teste nos ajuda a criar projetos melhores.

O acoplamento forte dificulta a criação de testes. Portanto, similarmente, quanto mais testes criarmos, usaremos mais princípios como o DIP e ferramentas como a injeção de dependência, interfaces e de modo a minimizar o acoplamento. Assim, nossos projetos se tornam ainda melhores.

O interessante é que ao seguir uma regra simples e óbvia a qual afirma que precisamos ter testes e executá-los, afeta constantemente a integração de nosso sistema aos objetivos principais da OO de acoplamento fraco e coesão alta. Criar testes leva a projetos melhores.

Regras de 2 a 4 de Projeto Simples: Refatoração

Agora que temos testes, podemos manter nosso código e nossas classes limpas. Para isso, refatoraremos gradualmente o código. Para cada linha nova que adicionarmos, paramos e refletimos sobre o novo projeto. Acabamos de prejudicá-lo? Caso positivo, nós o limpamos e rodamos nossos testes para nos certificar de que não danificamos nada. O fato de termos esses testes elimina o receio de que, ao limpá-los, podemos danificá-lo.

Durante a fase de refatoração, podemos aplicar qualquer conceito sobre um bom projeto de software. Podemos aumentar a coesão, diminuir o acoplamento, separar preocupações, modularizar as preocupações do sistema, reduzir nossas classes e funções, escolher nomes melhores, e por aí vai. Ademais, também podemos aplicar as três regras finais do projeto simples: eliminar a duplicação, garantir a expressividade e minimizar o número de classes e métodos.

Sem repetição de código

A repetição de código é o inimigo principal para um sistema bem desenvolvido. Ela representa trabalho, risco e complexidade desnecessária extras. A duplicação se apresenta de várias formas. Linhas de código que parecem idênticas são, é claro, duplicações.

Linhas de código semelhantes geralmente podem ser modificadas de modo que fiquem mais parecidas ainda para serem refatoradas mais facilmente. Há outros tipos de duplicação também, como a de implementação. Por exemplo, podemos ter dois métodos na classe de uma coleção:

```
int size() {}  
boolean isEmpty() {}
```

Poderíamos ter implementações separadas para cada método. O `isEmpty` poderia usar um booleano, enquanto `size` poderia usar um contador. Ou poderíamos eliminar essa duplicação colocando `isEmpty` na declaração de `size`:

```
boolean isEmpty() {  
    return 0 == size();  
}
```

Criar um sistema limpo requer a eliminação de código repetido, mesmo que sejam algumas linhas. Por exemplo, considere o código abaixo:

```
public void scaleToOneDimension(  
    float desiredDimension, float imageDimension)  
{  
    if (Math.abs(desiredDimension - imageDimension) <  
        errorThreshold)  
        return;  
    float scalingFactor = desiredDimension / imageDimension;  
    scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01f);  
    RenderedOp newImage = ImageUtilities.getScaledImage(  
        image, scalingFactor, scalingFactor);  
    image.dispose();  
    System.gc();  
    image = newImage;  
}  
  
public synchronized void rotate(int degrees) {  
    RenderedOp newImage = ImageUtilities.getRotatedImage(  
        image, degrees);  
    image.dispose();  
    System.gc();  
    image = newImage;  
}
```

A fim de manter esse código limpo, devemos eliminar a pequena quantidade de duplicação entre os métodos `scaleToOneDimension` e `rotate`:

```
public void scaleToOneDimension(  
    float desiredDimension, float imageDimension)
```

```

{
    if (Math.abs(desiredDimension - imageDimension) <
errorThreshold)
        return;
    float scalingFactor = desiredDimension / imageDimension;
    scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01f);
    replaceImage(ImageUtilities.getScaledImage(
        image, scalingFactor, scalingFactor));
}

public synchronized void rotate(int degrees) {
    replaceImage(ImageUtilities.getRotatedImage(image, degrees));
}

private void replaceImage(RenderedOp newImage) {
    image.dispose();
    System.gc();
    image = newImage;
}

```

Ao extraímos a semelhança neste nível baixíssimo, começamos a notar as violações ao SRP.

Dessa forma, podemos mover um método recém-extraído para outra classe, o que aumentaria sua visibilidade. Outra pessoa na equipe talvez perceba a oportunidade para abstrair mais a frente o novo método e usá-lo em um contexto diferente. Essa “pequena reutilização” pode reduzir consideravelmente a complexidade do sistema. Entender como reutilizar uma pequena parte do código é fundamental para fazer uso da utilização num escopo maior.

O padrão TEMPLATE METHOD² é uma técnica comum para a remoção de duplicação em alto nível. Por exemplo:

```

public class VacationPolicy {
    public void accrueUSDivisionVacation() {
        // código para calcular as férias baseando-se nas horas
        trabalhadas até a data
        //
        // código para garantir que as férias cumpram o tempo
        mínimo nos EUA
        //
        // código para aplicar vaction ao registro de folha de
        pagamento
        //
    }

    public void accrueEUDivisionVacation() {
        // código para calcular as férias baseando-se nas horas
        trabalhadas até a data
        //
        // código para garantir que as férias cumpram o tempo
        mínimo nos EUA
        //
        // código para aplicar vaction ao registro de folha de
        pagamento
        //
    }
}

```

```
    }  
}
```

O código ao longo de `accrueUSDivisionVacation` e `accrueEuropeanDivisionVacation` é praticamente o mesmo, com exceção do cálculo do tempo mínimo legal. Aquele pequeno algoritmo é alterado baseando-se no cargo do funcionário.

Podemos eliminar essa duplicação óbvia aplicando o padrão Template Method.

```
abstract public class VacationPolicy {  
    public void accrueVacation() {  
        calculateBaseVacationHours();  
        alterForLegalMinimums();  
        applyToPayroll();  
    }  
  
    private void calculateBaseVacationHours() { /* ... */ };  
    abstract protected void alterForLegalMinimums();  
    private void applyToPayroll() { /* ... */ };  
}  
  
public class USVacationPolicy extends VacationPolicy {  
    @Override protected void alterForLegalMinimums() {  
        // Logica usada nos EUA  
    }  
}  
  
public class EUVacationPolicy extends VacationPolicy {  
    @Override protected void alterForLegalMinimums() {  
        // Logica usada na Uniao Europeia  
    }  
}
```

As subclasses preenchem o “buraco” no algoritmo `accrueVacation`, fornecendo os únicos bits de informações que não são repetidos.

Expressividade

A maioria de nós já teve de trabalhar num código intrincado. Muitos de nós mesmos já produzimos alguns códigos confusos. Escrever códigos que nós entendamos é fácil, pois quando o fazemos, possuímos um conhecimento profundo do problema que desejamos resolver. Mas outras pessoas que pegarem esse mesmo código não terão esse mesmo grau de conhecimento.

A maioria dos custos de um projeto de software está na manutenção em longo prazo. A fim de minimizar os possíveis danos conforme fazemos alterações, é crucial que sejamos capazes de entender o que o sistema faz. Conforme os sistemas se tornam mais complexos, um desenvolvedor leva cada vez mais tempo para comprehendê-lo, e sempre há uma chance ainda maior de mal entendidos.

Portanto, o código deve expressar claramente o propósito de seu autor. Quando mais claro o autor tornar seu código, menos tempo outras pessoas terão de gastar para comprehendê-lo. Isso reduz os defeitos e o custo de manutenção. Você pode se expressar através da escolha de bons nomes. Queremos ser capazes de ler o nome de uma classe ou função e não ficarmos surpresos

quando descobrirmos o que ela faz. Você também pode se expressar mantendo pequenas suas classes e funções, que costumam ser fáceis de nomear, criar e entender.

Você também pode se expressar se usar uma nomenclatura padrão. Os Padrões de Projeto, por exemplo, são amplamente modos de comunicação e expressividade. Ao usar os nomes de padrões, como COMMAND ou VISITOR, no nome de classes que implementem tais padrões, você pode descrever de forma sucinta o seu projeto para outros desenvolvedores.

Testes de unidade bem escritos também são expressivos. Por exemplo, um objetivo principal dos testes é funcionar como um meio de documentação. Uma pessoa que leia nossos testes deverá ser capaz de obter um rápido entendimento do que se trata uma classe.

Mas a forma mais importante de ser expressivo é tentar. Muito freqüentemente, fazemos nosso código funcionar e, então, partimos para o problema seguinte, sem considerar o bastante em facilitar a leitura daquele código para outras pessoas. Lembre-se de que é muito mais provável que essa outra pessoa seja você.

Portanto, tenha um pouco de orgulho em seu trabalho. Gaste um tempo em cada uma das suas funções e classes. Escolha nomes melhores, divida funções grandes em menores e, de forma geral, cuide do que você mesmo criou. Cuidar é um recurso precioso.

Poucas classes e métodos

Podem-se exagerar mesmo nos conceitos mais fundamentais, como a eliminação de duplicação, expressividade do código e o SRP. Numa tentativa de tornar nossas classes e métodos pequenos, podemos vir a criar estruturas minúsculas. Portanto, essa regra sugere que também devamos manter a mínima quantidade de funções e classes.

Muitas classes e métodos, às vezes, são o resultado de um dogmatismo exagerado. Considere, por exemplo, um padrão de programação que insiste em criar uma interface para cada classe. Ou desenvolvedores que teimam em sempre separar campos e comportamentos em classes de dados e classes de comportamentos. Deve-se evitar tal dogmatismo e adotar uma abordagem mais pragmática.

Nosso objetivo é manter nosso sistema geral pequeno ao mesmo tempo em que também mantemos nossas classes e funções pequenas. Lembre-se, contudo, de que essa regra é a de menor prioridade das quatro de Projeto Simples. Portanto, embora seja importante manter baixa a quantidade de classes e funções, é mais importante ter testes, eliminar a duplicação e se expressar.

Conclusão

Há uma série de práticas simples que possam substituir a experiência? Obviamente que não. Por outro lado, as práticas descritas neste capítulo e neste livro são uma forma consolidada de muitas décadas de experiência adquiridas pelos autores. Seguir as regras de projeto simples pode realmente incentivar e possibilitar desenvolvedores a aderirem a bons princípios e padrões que, de outra forma, levariam anos para aprender.

Bibliografia

[XPE]: Extreme Programming Explained: Embrace Change, Kent Beck, Addison-Wesley, 1999.

[GOF]: Padrões de Projeto, Soluções Reutilizáveis de Software Orientado a Objetos, Gamma et al., Addison-Wesley, 1996

versão para o seu lado, com o que é sólido e seguro em um lado e instável e suscetível a quedas no outro. Ainda assim, é preciso ter paciência e tempo para que o resultado final seja bom. O que é certo é que é preciso ter paciência e tempo para que o resultado final seja bom.

13

Concorrência

por Brett L. Schuchert



"Objetos são abstrações de procedimento. Threads são abstrações de agendamento."
— James O. Coplien¹

Escrever programas concorrentes limpos é difícil, muito. É muito mais fácil criar um código que execute uma única thread, assim como um código multithread que pareça bom superficialmente, mas que esteja defeituoso em um nível mais baixo. Esse código funciona bem até que se use o sistema excessivamente.

Neste capítulo, discutiremos a necessidade da programação concorrente e das dificuldades que ela representa. Depois, daremos várias sugestões para lidar com tais dificuldades e escrever um código concorrente limpo. E, então, fechamos com as questões relacionadas aos códigos concorrentes de teste.

Concorrência limpa é um assunto complexo, válido um livro só para ele. Nossa estratégia aqui é apresentar uma visão geral e oferecer um tutorial mais detalhado em Concorrência II na página 317. Se você estiver apenas curioso sobre concorrência, este capítulo será o suficiente por enquanto. Se precisar de um conhecimento mais profundo, então leia também o tutorial.

Por que concorrência?

Concorrência é uma estratégia de desacoplamento. Ela nos ajuda a desacoplar o que é executado de quando é executado. Em aplicativos com apenas uma thread, o que e quando ficam tão fortemente acoplados que geralmente pode-se determinar o estado do aplicativo inteiro apenas ao olhar o rastreamento de retorno na pilha.

Desacoplar o que de quando pode melhorar consideravelmente tanto as estruturas quanto a taxa de transferência dos dados de um aplicativo. De uma perspectiva estruturada, o aplicativo seria mais como muitos minicomputadores trabalhando juntos do que um único e grande main(). Isso pode facilitar a compreensão do sistema e oferecer recursos melhores para separar preocupações.

Considere, por exemplo, o modelo “Servlet” padrão dos aplicativos da Web. Esses sistemas rodam sob o “guarda-chuva” de um contêiner Web ou EJB que gerencia parcialmente a concorrência para você. Os servlets são executados de forma assíncrona sempre que chega um pedido da Web.

O programador do servlet não tem de gerenciar todos os pedidos que chegam. Em princípio, cada execução de servlet ocorre em seu próprio mundinho e fica desacoplado de todas as execuções de outros servlets.

É claro que se fosse fácil assim, este capítulo não seria necessário. De fato, o desacoplamento oferecido pelos contêineres Web está longe de serem perfeitos. Os programadores de servlets têm se estar bem atentos de modo a garantir que seus programas concorrentes estejam corretos. Mesmo assim, as vantagens do modelo de servlet são significantes.

Mas a estrutura não é o único motivo para se adotar a concorrência. Alguns sistemas possuem limites de tempo de resposta e de taxa de transferência de dados que requerem soluções concorrentes programadas manualmente. Por exemplo, considere um agregador de informações com uma única thread que obtém os dados de diferentes sites da Web e os agrupa em um resumo diário. Como esse sistema só possui uma thread, ele consulta um site de cada vez, sempre terminando em um e seguindo para o próximo. A execução diária precisa ser feita em menos de 24 horas. Entretanto, conforme mais websites são adicionados, o tempo também aumenta, até que sejam necessárias mais do que 24 horas para obter todos os dados. Ter uma única thread implica em muito tempo de espera nos sockets da Web para que a E/S termine.

Poderíamos melhorar o desempenho usando um algoritmo multithread que consulte mais de um website por vez.

Ou pense num sistema que trate de um usuário de cada vez e exija apenas um segundo de tempo por usuário. Esse sistema é o suficiente para poucos usuários, mas conforme o número aumentar, também crescerá o tempo de resposta. Nenhum usuário quer entrar numa fila atrás de outros 150! Poderíamos melhorar o tempo de resposta tratando de vários usuários concorrentemente.

Ou, então, imagine um sistema que interprete extensas séries de dados, mas que só ofereça uma solução completa após processá-las todas. Talvez poderiam processar cada série em um computador diferente, de modo que muitas séries de dados fossem processadas paralelamente.

Mitos e conceitos equivocados

E também há motivos irrefutáveis para se adotar a concorrência. Entretanto, como dissemos anteriormente, usar a concorrência é difícil. Se não você não for muito cauteloso, poderá criar situações muito prejudiciais.

Considere os mitos e conceitos equivocados comuns abaixo:

- *A concorrência sempre melhora o desempenho.*

Isso pode ocorrer às vezes, mas só quando houver um tempo de espera muito grande que possa ser dividido entre múltiplas threads ou processadores. Nenhuma situação é trivial.

- *O projeto não muda ao criar programas concorrentes.*

De fato, o projeto de um algoritmo concorrente pode ser consideravelmente diferente do projeto de um sistema de apenas uma thread. O desacoplamento entre o que e quando costuma ter grande impacto na estrutura do sistema.

- *Entender as questões de concorrência não é importante quando se trabalha com um contêiner como um da Web ou um EJB.*

Na verdade, é melhor saber apenas o que o seu contêiner está fazendo e como protegê-lo das questões de atualização da concorrência e do deadlock (impasse) descrito mais adiante.

A seguir estão outras frases mais corretas em relação à criação de softwares concorrentes:

- *A concorrência gera um certo aumento, tanto no desempenho como na criação de código adicional.*

- *Uma concorrência correta é complexa, mesmo para problemas simples.*

- *Os bugs de concorrência geralmente não se repetem, portanto são frequentemente ignorados como casos únicos² em vez dos defeitos que realmente representam.*

- *A concorrência geralmente requer uma mudança essencial na estratégia do projeto.*

Desafios

O que torna a programação concorrente tão difícil? Considere a simples classe seguinte:

```
public class X {
```

```
private int lastIdUsed;
public int getNextId() {
    return ++lastIdUsed;
}
```

Digamos que criemos uma instância de `x`, atribuímos 42 ao campo `lastIdUsed` e, então a compartilhamos entre duas threads. Agora, suponha que ambas as threads chamem o método `getNextId()`. Haverá três resultados possíveis:

- Thread um recebe o valor 43, thread um recebe 44 e `lastIdUsed` é 44.
- Thread um recebe o valor 44, thread um recebe 43 e `lastIdUsed` é 44.
- Thread um recebe o valor 43, thread um recebe 43 e `lastIdUsed` é 43.

O surpreendente terceiro resultado3 ocorre quando ambas as threads se cruzam. Isso acontece porque há muitos caminhos que elas podem seguir naquela linha de código Java, e alguns dos caminhos geram resultados incorretos. Há quantos caminhos diferentes? Para poder responder a essa questão, precisamos entender o que o Just-In-Time Compiler faz com o Bytecode gerado e o que o modelo de memória do Java considera atômico.

Uma resposta rápida, usando o Bytecode gerado, é que há 12.870 caminhos possíveis de execução para aquelas duas threads executadas dentro do método `getNextId()`. Se o tipo de `lastIdUsed` mudar de `int` para `long`, o número de caminhos possíveis cresce para 2.704.156. É claro que a maioria desses caminhos gera resultados válidos. O problema é que alguns não.

Princípios para proteção da concorrência

A seguir está uma série de princípios e técnicas para proteger seus sistemas dos problemas de códigos concorrentes.

Princípio da Responsabilidade Única

O SRP5 declara que um dado método/classe/componente deve ter apenas um motivo para ser alterado. O modelo de concorrência é complexo o suficiente para ser uma razão e ter seu próprio direito de mudança e, portanto, merece ser separado do resto do código. Infelizmente, é muito comum que se incluam diretamente os detalhes de implementação de concorrência em outro código de produção. Abaixo estão alguns pontos a se levar em consideração:

- *O código voltado para a concorrência possui seu próprio ciclo de desenvolvimento, alteração e otimização.*
- *O código voltado para a concorrência possui seus próprios desafios, que são diferentes e mais difíceis do que o código não voltado para concorrência.*
- *O número de maneiras pelas quais um código voltado para concorrência pode ser escrito de*

forma errada é desafiador o bastante sem o peso extra do código de aplicação que o cerca.

Recomendação: Mantenha seu código voltado para a concorrência separado do resto do código⁶.

Solução: limite o escopo dos dados

Como vimos, duas threads que modificam o mesmo campo de um objeto compartilhado podem interferir uma na outra, causando um comportamento inesperado. Uma solução é usar a palavra reservada synchronized para proteger uma parte crítica do código que use aquele objeto compartilhado. É importante restringir a quantidade dessas partes. Em quantos mais lugares os dados compartilhados podem vir a ser alterados, maiores serão as chances de:

- Você se esquecer de proteger um ou mais daqueles lugares – danificando todos os códigos que modifiquem aqueles dados compartilhados
- Haver duplicação de esforços para garantir que tudo esteja protegido de forma eficiente (violação do Princípio do Não Se Repita⁷, DRY, sigla em inglês);
- Dificultar mais a determinação da origem das falhas, que já são difíceis de encontrar.

Recomendação: Leve a sério o encapsulamento de dados; limite severamente o acesso a quaisquer dados que possam ser compartilhados.

Solução: Use cópias dos dados

Essa é uma boa maneira de evitar que os dados compartilhados compartilhem seus dados. Em alguns casos podem-se fazer cópias dos objetos e tratá-los como somente-leitura. Em outros casos podem-se fazer cópias dos objetos, colocar os resultados de múltiplas threads naquelas cópias e, então, unir os resultados numa única thread.

Se houver uma maneira fácil de evitar o compartilhamento de objetos, será muito pouco provável que o código resultante cause problemas. Talvez você esteja preocupado com o custo de toda essa criação de objetos adicionais. Vale a pena experimentar para descobrir se isso é de fato um problema. Entretanto, se usar cópias dos objetos permitir ao código evitar a sincronização, o que se ganha provavelmente compensará pelas criações adicionais e o aumento da coleta de lixo.

Solução: as threads devem ser as mais independentes possíveis

Considere escrever seu código com threads de tal modo que cada thread exista em seu próprio mundo, sem compartilhamento de dados com qualquer outra thread. Cada uma processa um pedido do cliente, com todos os seus dados necessários provenientes de uma fonte não compartilhada e armazenada como variáveis locais. Isso faz com que cada uma das threads se comporte como se fossem a única thread no mundo, sem a necessidade de sincronização.

Por exemplo, as classes que criam subclasses a partir de HttpServlet recebem todas as suas informações passadas por parâmetros nos métodos doGet e doPost. Isso faz cada Servlet

se agir como se tivesse sua própria máquina. Contanto que o código no Servlet use apenas variáveis locais, não há como o Servlet causar problemas de sincronização. É claro que a maioria dos aplicativos usando Servlets acabará adotando recursos compartilhados, como conexões de bancos de dados.

Recomendação: *Tente dividir os dados em subsistemas independentes que possam ser manipulados por threads independentes, possivelmente em processadores diferentes.*

Conheça sua biblioteca

O Java 5 oferece muitas melhorias para o desenvolvimento concorrente em relação às versões anteriores. Há várias coisas a se considerar ao criar código com threads em Java 5:

- Use as coleções seguras para threads fornecidas.
- Use o framework Executor para executar tarefas não relacionadas.
- Use soluções non-blocking sempre que possível.
- Classes de bibliotecas que não sejam seguras para threads.

Coleções seguras para threads

Quando o Java estava no início, Doug Lea escreveu o livro precursor *Concurrent Programming*, com o qual juntamente ele desenvolveu várias coleções seguras para threads, que mais tarde se tornou parte do JDK no pacote `java.util.concurrent`. As coleções neste pacote são seguras para situações multithread e funcionam bem. Na verdade, a implementação de `ConcurrentHashMap` roda melhor do que a `HashMap` em quase todos os casos. Além de permitir leitura e escrita concorrente simultânea e possuir métodos que suportam operações compostas comuns que, caso contrário, não seriam seguras para thread. Se o Java 5 for o ambiente de implementação, comece com a `ConcurrentHashMap`.

Há diversos outros tipos de classes adicionados para suportar o modelo de concorrência avançado. Aqui estão alguns exemplos:

ReentrantLock	bloqueio que pode ser colocado em um método e liberado em outro.
Semaphore	implementação do semáforo clássico, um bloqueio com um contador.
CountDownLatch	bloqueio que espera por um número de eventos antes de liberar todas as threads em espera. Isso permite que todas as threads tenham a mesma chance de iniciar quase ao mesmo tempo.

Recomendação: *Revise as classes disponíveis para você. No caso do Java, familiarize-se com as classes `java.util.concurrent`, `java.util.concurrent.atomic`, `java.util.concurrent.locks`.*

Conheça seus métodos de execução

Há diversas maneiras de dividir o comportamento em um aplicativo concorrente. Para falarmos sobre eles, precisamos entender algumas definições básicas.

Recursos limitados (Bound Resources)	recursos de um tamanho ou número fixo usado em um ambiente concorrente.
Conexões de banco de dados e buffers de leitura/escrita de tamanho fixo são alguns exemplos.	
Exclusão mútua (Mutual Exclusion)	apenas uma thread de cada vez pode acessar dados ou recursos compartilhados.
Espera indefinida (Starvation)	uma thread ou um grupo de threads não pode prosseguir por um tempo excessivamente longo ou indefinidamente. Por exemplo: sempre deixar que threads de execução rápida rodem primeiro pode fazer com que threads que levem mais tempo tenham de esperar muito caso as de execução rápida forem infinitas.
Bloqueio infinito (Deadlock)	duas ou mais threads esperam que a outra termine. Cada thread possui um recurso que a outra precisa e nenhuma delas pode terminar até obter o tal recurso.
Livelock	threads num entrave, cada uma tentando fazer seu trabalho, mas se deparando com outras “no caminho”. Devido à repercussão, as threads continuam tentando progredir, mas não conseguem por um tempo excessivamente longo ou indefinido.

Dadas essas definições, agora podemos discutir os modelos de execução usados na programação concorrente.

Producer-Consumer⁹

Uma ou mais threads producer criam alguma tarefa e a colocam em um buffer ou fila de espera. Uma ou mais threads consumer pegam a tarefa da fila de espera e a finalizam. A fila de espera entre as producers e as consumers é um bound resource (recurso limitado). Isso significa que as producers devem esperar por espaço livre na fila de espera antes de colocar algo nela, e que as consumers devem esperar até que haja algo na fila de espera para ser recuperado. A coordenação entre as threads producers e consumers através da fila de espera requer que elas enviem sinais entre si. As producers escrevem na fila de espera e sinalizam que ela não está mais vazia. As consumers lêem a partir da fila de espera e sinalizam que ela não está mais cheia. Ambas ficam na espera pela sinalização para poderem continuar.

Leitores e escritores¹⁰

Quando você tem um recurso compartilhado que serve principalmente como uma fonte de informações para leitores, mas que de vez em quando é atualizada por escritores, a taxa de transferência dos dados é um problema. Isso porque ela pode gerar espera indefinida (starvation) e acúmulo de informações velhas. Permitir atualizações pode afetar a taxa de transferência dos dados. Coordenar os leitores de modo que não leiam algo que um escritor esteja atualizando, e vice-versa, é um equilíbrio difícil. Os escritores tendem a bloquear muitos leitores por bastante tempo, afetando assim a taxa de transferência dos dados.

O desafio é equilibrar as necessidades tanto dos leitores como dos escritores para satisfazer a operação correta, oferecer uma taxa de transferência de dados razoável e evitar a espera indefinida. Uma estratégia simples é fazer os escritores esperarem até que não haja mais leitores e, então, permitir que façam a atualização. Entretanto, se houver leitores constantes, os escritores ficarão numa espera indefinida. Por outro lado, se houver escritores frequentemente e eles tiverem prioridade, o impacto será na taxa de transferência de dados. Encontrar esse equilíbrio e evitar as questões de atualização concorrente é do que trata o problema.

Dining Philosophers (Problema dos Filósofos)¹¹

Imagine alguns filósofos sentados em uma mesa redonda. Coloca-se um garfo à esquerda de cada um. Há uma grande tigela de espaguete no meio da mesa. Os filósofos passam o tempo pensando, a menos que estejam com fome. Quando isso acontece, cada um pega seu garfo e come. Um filósofo não pode comer a menos que esteja segurando dois garfos. Se o filósofo à sua direita ou esquerda já estiver usando um dos garfos que ele precisa, ele deverá esperar até que aquele filósofo termine de comer e repouse o garfo novamente na mesa.

Quando um filósofo acaba de comer, ele devolve seu garfo à mesa e espera ficar com fome novamente. Substitua os filósofos por threads e os garfos por recursos. Esse problema é semelhante a muitos aplicativos corporativos nos quais os processos competem pelos recursos. A menos que tenha sidometiculosamente desenvolvido, os sistemas que competem dessa forma podem sofrer deadlock, livelock e queda na taxa de transferência de dados e no desempenho.

A maioria dos problemas de concorrência que você encontrará será uma variação desses três. Estude esses algoritmos e crie soluções usando-os à sua maneira, de modo que, ao se deparar com problemas de concorrência, você esteja mais preparado para resolvê-lo.

Recomendação: *Aprenda esses algoritmos básicos e entenda suas soluções.*

Cuidado com dependências entre métodos sincronizados

Dependências entre métodos sincronizados causam pequenos bugs no código concorrente. A linguagem Java possui a palavra reservada synchronized, que protege um único método. Entretanto, se houver mais de um método sincronizado na mesma classe compartilhada, então seu sistema pode ser sido escrito incorretamente¹².

Recomendação: evite usar mais de um método em um objeto compartilhado.

Haverá vezes em que você deverá usar mais de um método em um objeto compartilhado. Neste caso, há três formas de deixar o código certo:

- **Bloqueio voltado para o cliente:** faça o cliente bloquear o servidor antes de chamar o primeiro método e certifique-se de que o bloqueio inclua o código que chama o último método.
- **Bloqueio voltado para o servidor:** dentro do servidor, crie um método que bloquee o servidor, chame todos os métodos e, então, desbloqueie. Faça o cliente chamar o novo método.
- **Servidor extra:** crie um servidor intermediário que efetue o bloqueio. Esse é um exemplo de bloqueio voltado para o servidor, no qual o servidor original não pode ser alterado.

Mantenha pequenas as seções sincronizadas

A palavra reservada `synchronized` adiciona um bloqueio. Todas as seções do código protegidas pelo mesmo bloqueio garantem que há apenas uma thread em execução para todas elas num dado momento. Os bloqueios são prejudiciais, pois causam atrasos e adicionam trabalho extra. Portanto, não queremos amontoar nosso código com instruções `synchronized`. Por outro lado, devem-se proteger seções críticas¹³. Sendo assim, desejamos criar nosso código com o menor número possível de seções críticas.

Alguns programadores ingênuos tentam conseguir isso tornando as seções críticas muito grandes. Entretanto, estender a sincronização além da seção crítica mínima aumenta os conflitos e prejudica o desempenho¹⁴.

Recomendação: mantenha suas seções sincronizadas as menores possíveis.

É difícil criar códigos de desligamento corretos

Criar um sistema que deva ficar para sempre executando é diferente de criar algo que funcione por um tempo e, então, desligue de maneira adequada.

Obter um desligamento adequado pode ser difícil. Dentre os problemas comuns estão o deadlock¹⁵, com threads esperando por um sinal que nunca chega para continuar. Por exemplo, imagine um sistema com uma thread pai que gera diversas threads filhas e, então, antes de liberar seus recursos e desligar, espera que todas elas finalizem. E se uma das threads filhas sofrer deadlock? O pai esperará para sempre e o sistema jamais desligará.

Ou pense num sistema semelhante que tenha sido instruído a desligar. A thread pai diz a todas as suas filhas para abandonar suas tarefas e finalizar. Mas e se duas das filhas estiverem operando como um par producer/consumer? Suponha que a thread producer receba o sinal da thread pai e desligue imediatamente. A consumer talvez estivesse esperando uma mensagem da producer e bloqueada de modo que não consiga receber o sinal para desligamento.

Ela ficaria presa esperando pela producer e nunca finalizar, evitando que a thread pai também finalize.

Situações como essa não são tão incomuns assim. Portanto, se você precisar criar um código concorrente que exija um desligamento apropriado, prepare-se para passar a maior parte de seu tempo tentando fazer com que o desligamento ocorra com sucesso.

Recomendação: *Pense o quanto antes no desligamento e faça com que ele funcione com êxito. Vai levar mais tempo do que você espera. Revise os algoritmos existentes, pois isso é mais difícil do que você imagina.*

Teste de código com threads

Considerar que o código está correto, impossível. Testes não garantem que tudo esteja correto. Entretanto, eles podem minimizar os riscos. Isso tudo é válido para uma solução com uma única thread. Enquanto houver duas ou mais threads usando o mesmo código e trabalhando com os mesmos dados compartilhados, as coisas se tornam consideravelmente mais complexas.

Recomendação: *Crie testes que consigam expor os problemas e, então, execute-os frequentemente, com configurações programáticas e configurações e carregamentos de sistema. Se o teste falhar, rastreie a falha. Não ignore uma falha só porque o teste não a detectou no teste seguinte.*

É muito para se assimilar. Abaixo estão algumas recomendações mais detalhadas:

- Trate falhas falsas como questões relacionadas às threads.
- Primeiro, faça com que seu código sem thread funcione.
- Torne seu código com threads portátil.
- Torne seu código com threads ajustável.
- Rode com mais threads do que processadores.
- Rode em diferentes plataformas.
- Altere seu código para testar e forçar falhas.

Trate falhas falsas como questões relacionadas às threads.

O código que usa threads causa falhas em coisas que “simplesmente não falham”. A maioria dos desenvolvedores não entende como o uso de threads interage com outros códigos (incluindo seus autores). Os bugs em códigos com threads podem mostrar seus sintomas uma vez a cada mil ou milhares de execuções.

Tentativas para repetir os erros no sistema podem ser frustrantes. Isso geralmente leva os desenvolvedores a descartarem as falhas como raios cósmicos, uma pequena falha no hardware ou outro tipo de “casos isolados”. É melhor assumir que não existem casos isolados, os quais quanto mais forem ignorados, mais o código será construído no topo de uma abordagem possivelmente falha.

Recomendação: *Não ignore falhas de sistema como se fossem casos isolados.*

Primeiro, faça com que seu código sem thread funcione

Isso pode parecer óbvio, mas não custa nada repetir. Certifique-se de que o código funcione sem threads. Geralmente, isso significa criar POJOs que são chamados pelas suas threads.

Os POJOs não enxergam as threads e, portanto, podem ser testados fora do ambiente com threads. Quando mais locais no seu sistema você conseguir colocar tais POJOs, melhor.

Recomendação: *Não procure bugs não relacionados a threads com os relacionados a elas ao mesmo tempo. Certifique-se de que seu código funcione sem threads.*

Torne seu código com threads portátil

Criar o código que suporte concorrência de modo que possa ser executado em diversas configurações:

- Uma thread, várias threads, variações conforme a execução.
- O código com threads interage com algo que possa ser tanto real como artificial.
- Execute com objetos artificiais que rodem de forma rápida, lenta e variável.
- Configure testes de modo que possam rodar para um certo número de iterações.

Recomendação: *Faça de seu código com threads especialmente portátil de modo que possa executá-lo em várias configurações.*

Torne seu código com threads ajustável

Obter o equilíbrio certo entre as threads requer testar e errar. O quanto antes, encontre maneiras de cronometrar o desempenho de seu sistema sob variadas configurações. Possibilite para que 188 threads possam ser facilmente ajustadas. Considere permitir a alteração enquanto o sistema estiver em execução.

Considere permitir um auto-ajuste baseando-se na taxa de transferência de dados e no uso do sistema.

Rode com mais threads do que processadores.

Coisas acontecem quando o sistema alterna entre as tarefas. A fim de incentivar a troca (swapping) de tarefas, execute mais threads do que os processadores ou núcleos presentes. Quanto mais frequentemente suas tarefas alternarem, mais provavelmente você descobrirá partes do código que precisam de uma seção crítica ou que causa um deadlock.

Rode em diferentes plataformas

Em meados de 2007, elaboramos um curso sobre programação concorrente. O curso foi desenvolvido essencialmente sob a plataforma OS X. Apresentamos à turma usando o Windows XP rodando sob uma máquina virtual (VM, sigla em inglês). Criamos testes para demonstrar que as condições para falhas ocorriam mais frequentemente num ambiente OS X do que em um XP. Em todos os casos, sabia-se que o código testado possuía erros. Isso só reforçou o fato de que sistemas operacionais diferentes têm diferentes políticas de tratamento de threads, cada uma afetando a execução do código. O código multithread se comporta de maneira diferente em ambientes diversos¹⁶. Você deve rodar seus testes em cada possível ambiente de implementação.

Recomendação: Execute o quanto antes e frequentemente seu código com threads em todas as plataformas finais.

Altere seu código para testar e forçar falhas

É normal que as falhas se escondam em códigos concorrentes. Testes simples não costumam expô-las.

Na verdade, elas costumam se ocultar durante o processamento normal e talvez só apareçam uma vez em algumas horas, ou dias, ou semanas!

O motivo que torna os bugs em threads raros, esporádicos e de rara reincidência é que muito poucos caminhos dos milhares possíveis através de uma seção realmente falham. Portanto, a probabilidade de se tomar um caminho falho é extraordinariamente pequena. Isso dificulta muito a detecção e a depuração.

Como você poderia aumentar suas chances de capturar tais raras ocorrências? Você pode alterar seu código e forçá-lo a rodar em diferentes situações através da adição de chamadas a métodos como `Object.wait()`, `Object.sleep()`, `Object.yield()` e `Object.priority()`.

Cada um deles pode afetar a ordem da execução, aumentando assim as chances de detectar uma falha. É melhor que o código falhe o quanto antes.

Há duas opções para alteração:

- Manualmente
- Automatizada

Manualmente

Você pode inserir manualmente as chamadas a `wait()`, `sleep()`, `yield()` e `priority()`. É bom fazer isso quando estiver testando uma parte capciosa do código.

O exemplo abaixo faz exatamente isso:

```
public synchronized String nextUrlOrNull() {  
    if(hasNext()) {  
        String url = urlGenerator.next();  
        Thread.yield(); // inserido para testar.  
        updateHasNext();  
        return url;  
    }  
    return null;  
}
```

A chamada ao `yield()` inserida mudará os caminhos de execução tomados pelo código e possivelmente fará o código falhar onde não havia erro antes. Se isso ocorrer, não foi porque você adicionou uma chamada ao `yield()`¹⁷, mas porque seu código já possuía a falha e isso simplesmente a tornou evidente.

Há muitos problemas com essa abordagem:

- É preciso encontrar manualmente os locais onde fazer isso.
- Como saber onde e qual tipo de chamada colocar?
- Deixar tal código em um código de produção sem necessidade atrasa o código.
- Essa abordagem é um tiro no escuro. Você pode ou não encontrar falhas. De fato, as probabilidades não estão a seu favor.

Precisamos de uma forma de fazer isso durante a fase de testes, e não na de produção. Também temos de misturar com facilidade as configurações entre as diferentes execuções, o que aumentará as chances de encontrar erros no todo.

Claramente, se dividirmos nosso sistema em POJOs que não saibam nada sobre as threads e as classes que controlam o uso daquelas, será mais fácil encontrar os locais apropriados para alterar o código. Ademais, poderíamos criar muitas variações de testes que invoquem os POJOs sob sistemas diferentes de chamadas a sleep, yield, e assim por diante.

Automatizada

Você poderia usar ferramentas como um framework orientado a aspecto, CGLIB ou ASM para alterar seu código de forma automática. Por exemplo, você poderia usar uma classe com um único método:

```
public class ThreadJigglePoint {  
    public static void jiggle() {  
    }  
}
```

Você pode adicionar chamadas a ele em vários lugares de seu código:

```
public synchronized String nextUrlOrNull() {  
    if(hasNext()) {  
        ThreadJigglePoint.jiggle();  
        String url = urlGenerator.next();  
        ThreadJigglePoint.jiggle();  
        updateHasNext();  
        ThreadJigglePoint.jiggle();  
        return url;  
    }  
    return null;  
}
```

Agora você usa um aspecto simples que selecione aleatoriamente entre fazer nada, dormir ou ficar passivo.

Ou imagine que a classe ThreadJigglePoint tem duas implementações. A primeira implementa jiggle, não faz nada e é usada na produção. A segunda gera um número aleatório para selecionar entre dormir, ficar passivo ou apenas prosseguir. Se executar seus testes mil vezes com essa

aleatoriedade, talvez você revele algumas falhas. Se o teste passar, pelo menos você pode dizer que teve a devida diligência. Embora um pouco simples, essa poderia ser uma opção razoável em vez de uma ferramenta mais sofisticada.

Há uma ferramenta chamada `ConTest`¹⁸, desenvolvida pela IBM que faz algo semelhante, mas com um pouco mais de sofisticação.

A questão é testar o código de modo que as threads executem em ordens diferentes em horas diferentes. A combinação de testes bem escritos e esse processo podem aumentar consideravelmente as chances de encontrar erros.

Recomendação: *Use essas estratégias de testes para desmascarar erros.*

Conclusão

É difícil conseguir um código concorrente correto. Um código simples de se seguir pode se tornar um pesadelo quando múltiplas threads e dados compartilhados entram em jogo. Se você estiver criando esse tipo de código, é preciso mantê-lo rigorosamente limpo, ou haverá falhas sutis e não frequentes.

Primeiro e acima de tudo, siga o Princípio da Responsabilidade Única. Divida seu sistema em POJOs que separem o código que enxerga threads daquele que as ignora. Certifique-se de que você está testando apenas seu código que usa threads e nada mais. Isso sugere que ele deva ser pequeno e centralizado. Tenha em mente as possíveis fontes de problemas com concorrência: múltiplas threads operando em dados compartilhados ou usando uma fonte de recursos em comum. Casos de limites, como desligar adequadamente ou terminar a iteração de um loop, pode ser um risco a mais.

Estude sua biblioteca e conheça os algoritmos essenciais. Entenda como alguns dos recursos oferecidos por ela dão suporte à solução de problemas semelhantes aos proporcionados pelos algoritmos essenciais.

Aprenda como encontrar partes do código que devam ser bloqueadas e as bloquie – faça isso apenas com as que realmente precisem ser. Evite chamar uma seção bloqueada a partir de outra, pois isso requer um grande entendimento se algo deve ou não ser compartilhado. Mantenha a quantidade de objetos compartilhados e o escopo do compartilhamento o mais curto possível. Altere os modelos dos objetos com os dados compartilhados para acomodar os clientes em vez de forçar estes a gerenciar o estado compartilhado.

Problemas surgirão. Os que não aparecem logo geralmente são descartados como casos isolados. Esses famosos “casos isolados” costumam ocorrer apenas na inicialização ou em momentos aleatórios. Portanto, é preciso ser capaz de rodar repetidamente e constantemente seu código com threads em muitas configurações e plataformas. A capacidade de ser testado, que vem naturalmente com as Três Leis do TDD, implica certo nível de portabilidade, o que oferece o suporte necessário para executar o código numa gama maior de configurações.

Você pode melhorar consideravelmente suas chances de encontrar erros se tomar seu tempo para manipular seu código. Isso pode ser feito manualmente ou com alguma ferramenta automatizada. Invista nisso o quanto antes. É melhor ter executado seu código com threads o máximo possível antes de colocá-lo na fase de produção.

Refinamento Sucessivo

Caso de estudo de um analisador sintático de parâmetro em uma linha de comando



Este capítulo é um caso de estudo sobre como obter um refinamento com êxito. Você verá um módulo que começará bem, mas não progredirá mais. Então, verá como ele será refatorado e limpo.

A maioria de nós, de tempos em tempos, tem de analisar a sintaxe dos parâmetros na linha de comando. Se não tivermos um utilitário adequado, então simplesmente deixamos passar o array de strings passado à função main. Há vários utilitários bons disponíveis, mas nenhum deles faz exatamente o que quero. Portanto, decidi criar o meu próprio, que chamarei de Args.

É muito simples usá-lo. Basta criar uma classe Args com os parâmetros de entrada e uma string de formato, e, então, consultar a instância Args pelos valores dos parâmetros. Por exemplo, veja o exemplo abaixo:

Listagem 14-1

Uso simples de Args

```
public static void main(String[] args) {
    try {
        Args arg = new Args("l,p#,d*", args);
        boolean logging = arg.getBoolean('l');
        int port = arg.getInt('p');
        String directory = arg.getString('d');
        executeApplication(logging, port, directory);
    } catch (ArgsException e) {
        System.out.printf("Argument error: %s\n", e.errorMessage());
    }
}
```

Viu a simplicidade? Apenas criamos uma instância da classe Args com dois parâmetros. O primeiro é a string do formato, ou esquema: "l, p#, d*. " Ela declara três parâmetros na linha de comando. O primeiro, -l, é booleano; o segundo, -p, é um inteiro; e o terceiro, -d, é uma string. O segundo parâmetro do construtor Args é um array simples passado como parâmetro na linha de comando para main.

Se o construtor retornar sem lançar uma ArgsException, então a linha de comando da entrada é analisada e a instância Args fica pronta para ser consultada. Métodos como getBoolean, getInt e getString nos permite acessar os valores dos argumentos pelos nomes.

Se houver um problema, seja na string de formato ou nos parâmetros da linha de comando, será lançada uma ArgsException. Para uma descrição adequada do que deu errado, consulte o método errorMessage da exceção.

Implementação de Args

A Listagem 14.2 é a implementação da classe Args. Leia-a com bastante atenção. Esforcei-me bastante no estilo e na estrutura, e espero que valha a pena.

Listagem 14-2

Args.java

```
package com.objectmentor.utilities.args;

import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;
import java.util.*;

public class Args {
    private Map<Character, ArgumentMarshaler> marshalers;
```

Listagem 14-2

Args.java

```
private Set<Character> argsFound;
private ListIterator<String> currentArgument;

public Args(String schema, String[] args) throws ArgsException {
    marshalers = new HashMap<Character, ArgumentMarshaler>();
    argsFound = new HashSet<Character>();

    parseSchema(schema);
    parseArgumentStrings(Arrays.asList(args));
}

private void parseSchema(String schema) throws ArgsException {
    for (String element : schema.split(","))
        if (element.length() > 0)
            parseSchemaElement(element.trim());
}

private void parseSchemaElement(String element) throws ArgsException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (elementTail.length() == 0)
        marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (elementTail.equals("*"))
        marshalers.put(elementId, new StringArgumentMarshaler());
    else if (elementTail.equals("#"))
        marshalers.put(elementId, new IntegerArgumentMarshaler());
    else if (elementTail.equals("##"))
        marshalers.put(elementId, new DoubleArgumentMarshaler());
    else if (elementTail.equals("[*]]"))
        marshalers.put(elementId, new StringArrayArgumentMarshaler());
    else
        throw new ArgsException(INVALID_ARGUMENT_FORMAT, elementId, elementTail);
}

private void validateSchemaElementId(char elementId) throws ArgsException {
    if (!Character.isLetter(elementId))
        throw new ArgsException(INVALID_ARGUMENT_NAME, elementId, null);
}

private void parseArgumentStrings(List<String> argsList) throws ArgsException {
    for (currentArgument = argsList.listIterator(); currentArgument.hasNext();)
    {
        String argString = currentArgument.next();
        if (argString.startsWith("-"))
            parseArgumentCharacters(argString.substring(1));
        } else {
            currentArgument.previous();
            break;
        }
    }
}
```

Listagem 14-2

Args.java

```

private void parseArgumentCharacters(String argChars) throws ArgsException {
    for (int i = 0; i < argChars.length(); i++)
        parseArgumentCharacter(argChars.charAt(i));
}

private void parseArgumentCharacter(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null) {
        throw new ArgsException(UNEXPECTED_ARGUMENT, argChar, null);
    } else {
        argsFound.add(argChar);
        try {
            m.set(currentArgument);
        } catch (ArgsException e) {
            e.setErrorArgumentId(argChar);
            throw e;
        }
    }
}

public boolean has(char arg) {
    return argsFound.contains(arg);
}

public int nextArgument() {
    return currentArgument.nextInt();
}

public boolean getBoolean(char arg) {
    return BooleanArgumentMarshaler.getValue(marshalers.get(arg));
}

public String getString(char arg) {
    return StringArgumentMarshaler.getValue(marshalers.get(arg));
}

public int getInt(char arg) {
    return IntegerArgumentMarshaler.getValue(marshalers.get(arg));
}

public double getDouble(char arg) {
    return DoubleArgumentMarshaler.getValue(marshalers.get(arg));
}

public String[] getStringArray(char arg) {
    return StringArrayArgumentMarshaler.getValue(marshalers.get(arg));
}
}

```

Note que você consegue ler este esse código de cima para baixo sem ter de saltar muito para lá e para cá. Uma coisa que você teve de olhar um pouco mais a à frente é a definição de ArgumentMarshaler, que deixei de fora propositalmente. Após ter lido com atenção esse código, você deve ser capaz de entender o que fazem a interface ArgumentMarshaler e seus derivados. Irei lhe mostrar alguns deles agora (Listagem 14-3 até 14.6).

Listagem 14-3**ArgumentMarshaler.java**

```
public interface ArgumentMarshaler {  
    void set(Iterator<String> currentArgument) throws ArgsException;  
}
```

Listagem 14-4**ArgumentMarshaler.java**

```
public class BooleanArgumentMarshaler implements ArgumentMarshaler {  
    private boolean booleanValue = false;  
  
    public void set(Iterator<String> currentArgument) throws ArgsException {  
        booleanValue = true;  
    }  
  
    public static boolean getValue(ArgumentMarshaler am) {  
        if (am != null && am instanceof BooleanArgumentMarshaler)  
            return ((BooleanArgumentMarshaler) am).booleanValue;  
        else  
            return false;  
    }  
}
```

Listagem 14-5**StringArgumentMarshaler.java**

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;  
  
public class StringArgumentMarshaler implements ArgumentMarshaler {  
    private String stringValue = "";  
  
    public void set(Iterator<String> currentArgument) throws ArgsException {  
        try {  
            stringValue = currentArgument.next();  
        } catch (NoSuchElementException e) {  
            throw new ArgsException(MISSING_STRING);  
        }  
    }  
  
    public static String getValue(ArgumentMarshaler am) {  
        if (am != null && am instanceof StringArgumentMarshaler)  
            return ((StringArgumentMarshaler) am).stringValue;  
        else  
            return "";  
    }  
}
```

Listagem 14-6**IntegerArgumentMarshaler.java**

```

import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class IntegerArgumentMarshaler implements ArgumentMarshaler {
    private int intValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgsException {
        String parameter = null;
        try {
            parameter = currentArgument.next();
            intValue = Integer.parseInt(parameter);
        } catch (NoSuchElementException e) {
            throw new ArgsException(MISSING_INTEGER);
        } catch (NumberFormatException e) {
            throw new ArgsException(INVALID_INTEGER, parameter);
        }
    }

    public static int getValue(ArgumentMarshaler am) {
        if (am != null && am instanceof IntegerArgumentMarshaler)
            return ((IntegerArgumentMarshaler) am).intValue;
        else
            return 0;
    }
}

```

O outro ArgumentMarshaler derivado simplesmente replica esse padrão para arrays tipo doubles e String, e serve para encher esse capítulo. Deixarei-os para que você pratique neles.

Outra coisa pode estar lhe incomodando: a definição das constantes dos códigos de erro. Elas estão na classe ArgsException (Listagem 14.7).

Listagem 14-7**ArgsException.java**

```

import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class IntegerArgumentMarshaler implements ArgumentMarshaler {
    private int intValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgsException {
        String parameter = null;
        try {
            parameter = currentArgument.next();
            intValue = Integer.parseInt(parameter);
        } catch (NoSuchElementException e) {
            throw new ArgsException(MISSING_INTEGER);
        } catch (NumberFormatException e) {
            throw new ArgsException(INVALID_INTEGER, parameter);
        }
    }

    public static int getValue(ArgumentMarshaler am) {
        if (am != null && am instanceof IntegerArgumentMarshaler)
            return ((IntegerArgumentMarshaler) am).intValue;
        else
            return 0;
    }
}

```

Listagem 14-7 (continuação)

ArgsException.java

```
public ArgsException(ErrorCode errorCode,
                     char errorArgumentId, String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
    this.errorArgumentId = errorArgumentId;
}

public char getErrorArgumentId() {
    return errorArgumentId;
}

public void setErrorArgumentId(char errorArgumentId) {
    this.errorArgumentId = errorArgumentId;
}

public String getErrorParameter() {
    return errorParameter;
}

public void setErrorParameter(String errorParameter) {
    this.errorParameter = errorParameter;
}

public ErrorCode getErrorCode() {
    return errorCode;
}

public void setErrorCode(ErrorCode errorCode) {
    this.errorCode = errorCode;
}

public String errorMessage() {
    switch (errorCode) {
        case OK:
            return "TILT: Should not get here.";
        case UNEXPECTED_ARGUMENT:
            return String.format("Argument -%c unexpected.", errorArgumentId);
        case MISSING_STRING:
            return String.format("Could not find string parameter for -%c.",
                                errorArgumentId);
        case INVALID_INTEGER:
            return String.format("Argument -%c expects an integer but was '%s'.",
                                errorArgumentId, errorParameter);
        case MISSING_INTEGER:
            return String.format("Could not find integer parameter for -%c.",
                                errorArgumentId);
        case INVALID_DOUBLE:
            return String.format("Argument -%c expects a double but was '%s'.",
                                errorArgumentId, errorParameter);
        case MISSING_DOUBLE:
            return String.format("Could not find double parameter for -%c.",
                                errorArgumentId);
        case INVALID_ARGUMENT_NAME:
            return String.format("'%c' is not a valid argument name.", errorArgumentId);
    }
}
```

Listagem 14-7 (continuação)

ArgsException.java

```

        case INVALID_ARGUMENT_FORMAT:
            return String.format("%s is not a valid argument format.",
                errorParameter);
    }
    return "";
}

public enum ErrorCode {
    OK, INVALID_ARGUMENT_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
    MISSING_STRING,
    MISSING_INTEGER, INVALID_INTEGER,
    MISSING_DOUBLE, INVALID_DOUBLE)
}

```

É impressionante a quantidade de código necessária para esclarecer os detalhes deste simples conceito. Um dos motivos disso é que estamos usando uma linguagem particularmente prolixo. Como Java é uma linguagem estática, é preciso digitar muitas palavras de modo a satisfazer o sistema de tipos. Em linguagens como Ruby, Python ou Smalltalk, esse programa fica muito menor¹.

Leia o código novamente. Atente especialmente para os nomes escolhidos, o tamanho das funções e o formato do código. Se você for um programador experiente, talvez faça uma crítica aqui e ali e em relação ao estilo ou à estrutura. De modo geral, entretanto, espero que perceba que este programa está bem escrito e possui uma estrutura limpa.

Por exemplo, deve ficar óbvio como adicionar um novo tipo de parâmetro, como uma data ou um número complexo, e que essa inserção exige pouco esforço. Em suma, simplesmente bastaria criar um derivado de `ArgumentMarshaler`, uma nova função `getXXX` e uma nova instrução `case` na função `parseSchemaElement`. Provavelmente haveria também um na nova `ArgsException.ErrorCode` e uma nova mensagem de erro.

Como fiz isso?

Permita-me tirar sua dúvida. Eu não criei simplesmente esse programa do início ao fim neste formato que ele se encontra agora. E, mais importante, não espero que você seja capaz de escrever programas limpos e elegantes de primeira. Se aprendemos algo ao longo das últimas décadas, é que programar é mais uma arte do que ciência. Para criar um código limpo, é preciso criar primeiro um “sujo” e, então limpá-lo.

Isso não deveria ser surpresa para você. Aprendemos essa verdade na escola quando nossos professores tentavam (em vão) nos fazer escrever rascunhos de nossas redações. O processo, eles diziam, era criar um rascunho e, depois, um segundo e, então, vários outros até que chegássemos à versão final. Escrever redações limpas, eles tentavam nos dizer, é uma questão de refinamento constante.

A maioria dos programadores iniciantes (assim como muitos alunos do ensino fundamental) não segue muito bem esse conselho. Eles acreditam que o objetivo principal é fazer o programa funcionar e, uma vez conseguido, passam para a tarefa seguinte, deixando o programa “que funciona” no estado em que estiver. A maioria dos programadores experientes sabe que isso é suicídio profissional.

Args: o rascunho

A Listagem 14.8 mostra uma versão anterior da classe Args. Ela “funciona” e está uma zona.

Listagem 14-8

Args.java (primeiro rascunho)

```
import java.text.ParseException;
import java.util.*;

public class Args {
    private String schema;
    private String[] args;
    private boolean valid = true;
    private Set<Character> unexpectedArguments = new TreeSet<Character>();
    private Map<Character, Boolean> booleanArgs =
        new HashMap<Character, Boolean>();
    private Map<Character, String> stringArgs = new HashMap<Character, String>();
    private Map<Character, Integer> intArgs = new HashMap<Character, Integer>();
    private Set<Character> argsFound = new HashSet<Character>();
    private int currentArgument;
    private char errorArgumentId = '\0';
    private String errorParameter = "TILT";
    private ErrorCode errorCode = ErrorCode.OK;

    private enum ErrorCode {
        OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT
    }

    public Args(String schema, String[] args) throws ParseException {
        this.schema = schema;
        this.args = args;
        valid = parse();
    }

    private boolean parse() throws ParseException {
        if (schema.length() == 0 && args.length == 0)
            return true;
        parseSchema();
        try {
            parseArguments();
        } catch (ArgsException e) {
        }
        return valid;
    }

    private boolean parseSchema() throws ParseException {
        for (String element : schema.split(",")) {
```

Listagem 14-8 (continuação)

Args.java (primeiro rascunho)

```
if (element.length() > 0) {
    String trimmedElement = element.trim();
    parseSchemaElement(trimmedElement);
}
return true;
}

private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
        parseBooleanSchemaElement(elementId);
    else if (isStringSchemaElement(elementTail))
        parseStringSchemaElement(elementId);
    else if (isIntegerSchemaElement(elementTail)) {
        parseIntegerSchemaElement(elementId);
    } else {
        throw new ParseException(
            String.format("Argument: %c has invalid format: %s.",
                          elementId, elementTail), 0);
    }
}

private void validateSchemaElementId(char elementId) throws ParseException {
    if (!Character.isLetter(elementId)) {
        throw new ParseException(
            "Bad character:" + elementId + "in Args format: " + schema, 0);
    }
}

private void parseBooleanSchemaElement(char elementId) {
    booleanArgs.put(elementId, false);
}

private void parseIntegerSchemaElement(char elementId) {
    intArgs.put(elementId, 0);
}

private void parseStringSchemaElement(char elementId) {
    stringArgs.put(elementId, "");
}

private boolean isStringSchemaElement(String elementTail) {
    return elementTail.equals("*");
}

private boolean isBooleanSchemaElement(String elementTail) {
    return elementTail.length() == 0;
}

private boolean isIntegerSchemaElement(String elementTail) {
    return elementTail.equals("#");
}
```

Listagem 14-8 (continuação)

Args.java (primeiro rascunho)

```
private boolean parseArguments() throws ArgsException {
    for (currentArgument = 0; currentArgument < args.length; currentArgument++) {
        String arg = args[currentArgument];
        parseArgument(arg);
    }
    return true;
}

private void parseArgument(String arg) throws ArgsException {
    if (arg.startsWith("-"))
        parseElements(arg);
}

private void parseElements(String arg) throws ArgsException {
    for (int i = 1; i < arg.length(); i++)
        parseElement(arg.charAt(i));
}

private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
        argsFound.add(argChar);
    else {
        unexpectedArguments.add(argChar);
        errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
        valid = false;
    }
}

private boolean setArgument(char argChar) throws ArgsException {
    if (isBooleanArg(argChar))
        setBooleanArg(argChar, true);
    else if (isStringArg(argChar))
        setStringArg(argChar);
    else if (isIntArg(argChar))
        setIntArg(argChar);
    else
        return false;
}

    return true;
}

private boolean isIntArg(char argChar) {return intArgs.containsKey(argChar);}

private void setIntArg(char argChar) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
        parameter = args[currentArgument];
        intArgs.put(argChar, new Integer(parameter));
    } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorCode = ErrorCode.MISSING_INTEGER;
    }
}
```

Listagem 14-8 (continuação)

Args.java (primeiro rascunho)

```
        throw new ArgsException();
    } catch (NumberFormatException e) {
        valid = false;
        errorArgumentId = argChar;
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw new ArgsException();
    }
}

private void setStringArg(char argChar) throws ArgsException {
    currentArgument++;
    try {
        stringArgs.put(argChar, args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
    }
}

private boolean isStringArg(char argChar) {
    return stringArgs.containsKey(argChar);
}

private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.put(argChar, value);
}

private boolean isBooleanArg(char argChar) {
    return booleanArgs.containsKey(argChar);
}

public int cardinality() {
    return argsFound.size();
}

public String usage() {
    if (schema.length() > 0)
        return "-[" + schema + "]";
    else
        return "";
}

public String errorMessage() throws Exception {
    switch (errorCode) {
        case OK:
            throw new Exception("TILT: Should not get here.");
        case UNEXPECTED_ARGUMENT:
            return unexpectedArgumentMessage();
        case MISSING_STRING:
            return String.format("Could not find string parameter for -%c.",
                errorArgumentId);
```

Listagem 14-8 (continuação)**Args.java (primeiro rascunho)**

```
        case INVALID_INTEGER:
            return String.format("Argument -%c expects an integer but was '%s'.",
                errorArgumentId, errorParameter);
        case MISSING_INTEGER:
            return String.format("Could not find integer parameter for -%c.",
                errorArgumentId);
    }
    return "";
}

private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(s) -");
    for (char c : unexpectedArguments) {
        message.append(c);
    }
    message.append(" unexpected.");
    return message.toString();
}

private boolean falseIfNull(Boolean b) {
    return b != null && b;
}

private int zeroIfNull(Integer i) {
    return i == null ? 0 : i;
}

private String blankIfNull(String s) {
    return s == null ? "" : s;
}

public String getString(char arg) {
    return blankIfNull(stringArgs.get(arg));
}

public int getInt(char arg) {
    return zeroIfNull(intArgs.get(arg));
}

public boolean getBoolean(char arg) {
    return falseIfNull(booleanArgs.get(arg));
}

public boolean has(char arg) {
    return argsFound.contains(arg);
}

public boolean isValid() {
    return valid;
}

private class ArgsException extends Exception {
```

Espero que sua reação inicial a esse pedaço de código seja “Realmente estou feliz por ele não tê-lo deixado assim!”. Se você se sentir assim, então lembre-se de como as outras pessoas ficarão ao ler um código que você deixou num formato de “rascunho”.

Na verdade, “rascunho” é provavelmente a coisa mais gentil que se possa falar desse código. Está claro que é um trabalho em progresso. Quantidade de instâncias de variáveis é assustadora. Strings estranhas, como “TILT”, “HashSets” e “TreeSets, e os blocos try...catch... catch ficam todas amontoadas.

Eu não queria criar um código amontoado. De fato, eu estava tentando manter as coisas razoavelmente bem organizadas. Você provavelmente pode deduzir isso dos nomes que escolhi para as funções e as variáveis, e o fato de que mantive uma estrutura bruta. Mas, obviamente, me afastei do problema.

A bagunça cresceu gradualmente. As versões anteriores não estavam tão bagunçadas assim. Por exemplo, a Listagem 14.9 mostra uma versão antiga na qual só funcionavam os parâmetros do tipo booleano.

Listagem 14-9

Args.java (Booleano apenas)

```
package com.objectmentor.utilities.getopts;

import java.util.*;

public class Args {
    private String schema;
    private String[] args;
    private boolean valid;
    private Set<Character> unexpectedArguments = new TreeSet<Character>();
    private Map<Character, Boolean> booleanArgs =
        new HashMap<Character, Boolean>();
    private int numberofArguments = 0;

    public Args(String schema, String[] args) {
        this.schema = schema;
        this.args = args;
        valid = parse();
    }

    public boolean isValid() {
        return valid;
    }

    private boolean parse() {
        if (schema.length() == 0 && args.length == 0)
            return true;
        parseSchema();
        parseArguments();
        return unexpectedArguments.size() == 0;
    }

    private void parseSchema() {
        for (String element : schema.split(",")) {
            parseSchemaElement(element);
        }
    }
```