DSL

Quebre a barreira entre desenvolvimento e negócios

Casa do Código

LEONARDO OTTO

ISBN

Impresso e PDF: 978-85-5519-282-1

EPUB: 978-85-5519-283-8

MOBI: 978-85-5519-284-5

Você pode discutir sobre este livro no Fórum da Casa do Código: http://forum.casadocodigo.com.br/.

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

AGRADECIMENTOS

Agradeço à minha mãe, Angela Marisa Goslar. Sem ela, nem o autor e muito menos este livro poderiam ter existido.

Não poderia esquecer também dos meus amigos que revisaram o draft, em especial ao Michael Siegwarth e ao Ricardo Lecheta. Também à Cynthia Naomi Iizuka, por todo o apoio dado enquanto estava escrevendo este livro.

Algumas pessoas nos ajudam a focar em outros aspectos da vida, obrigado Alesandra Selma Martins pelas ótimas dicas profissionais. Agradeço também as empresas Rede e Livetouch, pois nelas conheci e aprendi muito com um grupo enorme de pessoas fantásticas.

Para finalizar, agradeço à Casa do Código pelo ótimo trabalho em tudo que precisei.

SOBRE O AUTOR

Leonardo Otto é Engenheiro de Software desde 2005, e é apaixonado por software bem feito e que funciona. Curioso por natureza, já trabalhou com uma infinidade de coisas diferentes, desde terminais de pagamento, microcontroladores até servidores na nuvem.

Adora testar novas ferramentas e tenta sempre utilizar as melhores disponíveis. Desde 2012, desenvolve aplicativos mobile para iOS e Android. Suas áreas de interesse são design de software, mobilidade, linguagens de programação e agilidade.

PREFÁCIO

Quando comecei a programar — e isso já faz algum tempo —, sempre procurei novas e melhores formas de me expressar em termos de linguagens. Tentava torcer a linguagem de modos novos para extrair formas mais elegantes e eficientes. Já falhei miseravelmente. Confundi mais que esclareci. Mas apesar de tudo, obtive alguns sucessos. Consegui modelar minhas ideias em forma de linguagem de forma ótima.

Trabalhamos essencialmente com ideias e linguagem, e limitações nestas áreas são nossos paradoxos. Vivemos dentro de nossos paradoxos de como as coisas devem ser. E pior ainda, vivemos dentro de um paradoxo de como pensar para resolver nossos problemas.

O paradoxo é quebrado quando encontramos novas formas de pensar, novos algoritmos, novos métodos e, sobretudo, novas linguagens.

São as linguagens, principalmente, que fazem você pensar diferente. Sua linguagem nativa (por exemplo, o português ou o inglês) é mais complexa e flexível em várias ordens de grandeza do que uma linguagem de programação, mas ela também tem suas limitações. Você só descobre que estas limitações existem quando aprende uma nova língua.

Existe um ponto de vista que diz que a profissão de programação é uma profissão de tradução. Estamos sempre traduzindo o que os especialistas estão dizendo para uma língua que os computadores entendam. Ou seja, capturamos as ideias de

especialistas, as completamos (pois normalmente eles eliminam detalhes) e traduzimos para outra linguagem.

Manter o código o mais próximo dos conceitos dos especialistas é um passo importante, mas este pode ser expandido ainda mais com o uso de *Linguagens Específicas de Domínio* (DSLs). Aprendendo como construí-las, ganhamos a capacidade criar softwares mais simples, mais comunicativos e com uma melhor manutenção.

Este livro vai lhe ensinar como utilizar e criar DSLs internas para obter o código mais expressivo possível. A comunicação com os especialistas de domínio se tornará um desafio muito mais interessante depois de conhecer estas técnicas.

Em muitas áreas, e principalmente em tecnologia, não existe uma teoria completa. Este livro não é o mais completo guia sobre DSLs, se é que algum dia algum livro vai estar totalmente completo.

É um livro focado, e introdutório, a um assunto fascinante, que são DSLs. É um passeio guiado em um lugar totalmente novo.
E espero que você volte muitas vezes a ele.

A quem se destina este livro?

Este livro se destina principalmente a desenvolvedores de software com experiência intermediária e/ou avançada. Desenvolvedores que desejam entrar no mundo das DSLs internas e também adicionar mais uma ferramenta de expressão ao seu trabalho.

Ele é particularmente útil para desenvolvedores que querem

melhorar suas APIs, usuários do domínio que queiram se comunicar melhor com os times de desenvolvimento, como também para desenvolvedores que queiram verificar, de uma forma mais clara, se as implementações de regras de negócios estão corretas.

Se o leitor já tiver uma formação acadêmica relacionada ou alguma experiência em compiladores, terá uma visão adicional sobre o conteúdo.

Os exemplos do livro utilizam as linguagens de programação Java e Scala. É necessário apenas ter uma experiência intermediária com a linguagem Java, mas não é necessário ter experiência prévia com Scala. É necessário também ter conhecimento básico da linguagem XML.

O livro possui um apêndice demostrando as principais características da linguagem Scala, e esta demonstração já é suficiente para aproveitar os exemplos. Devido a algumas similaridades entre a linguagem Java e a Scala, o leitor vai sentir uma transição suave quando começar a desenvolver utilizando Scala. Se o leitor já tiver experiência em Scala, pode pular este apêndice sem problemas.

NOMES E SIGLAS

Neste livro, uso sempre a sigla do termo em inglês para definir Linguagens Específicas de Domínio, ou seja, *Domain Specific Language* (DSL). Para especificar as linguagens como um conjunto, utilizo a sigla DSLs.

Todos os exemplos deste livro podem ser encontrados em: https://bitbucket.org/leonardootto/dsl-exemplos/src.

Sumário

Capítulo 1

BATENDO UMA BOLA COM SUA DSL

O Monty Python é um grupo humorístico inglês da década de 70, criador do programa *Monty Python's Flying Circus*. Este grupo tem vários esquetes bem humoradas e inteligentes sobre diversos assuntos. Em uma delas, é narrado um hipotético jogo de futebol entre filósofos falecidos.

Por exemplo, Karl Marx que faz o aquecimento, mas nunca entra em campo — uma analogia sobre a vida de Karl Marx que sempre teorizou, mas nunca fez nada prático. Como gosto de Monty Python, vamos utilizar uma analogia similar ao seu Futebol dos Filósofos para demonstrar um exemplo do que seria uma linguagem específica de domínio ou, para simplificar, DSL.

Conhecer este esquete do Monty Python não é pré-requisito para entender este exemplo. Mas como curiosidade, saibam que vem daí o nome da linguagem Python

1.1 COPA DO MUNDO DOS PROGRAMADORES

Imagine que, em uma realidade paralela, existe algo como: *A*

Copa do mundo de Futebol dos Programadores! Copa é uma versão dos jogos que você faz com seus amigos, só que com muito mais gente, mas não tão divertida.

Se é que dizer isto é necessário, futebol é um esporte jogado com os pés. Nele se tenta vencer as leis físicas de um objeto poder estar em apenas um lugar. Cada time conflitantemente tenta deixar o objeto em uma área do time oposto. Seria mais fácil se cada time tivesse seu próprio objeto, mas não teria muita graça.

Com o óbvio já esclarecido, podemos voltar ao exemplo. Nesta Copa do Mundo de Futebol, os mais famosos programadores competem em busca do título. Normalmente, você acompanha este evento como espectador, mas este ano vai ser diferente. Agora você participará da estruturação do evento: a empresa na qual você trabalha fechou um contrato com a organização da copa, e você foi escalado para desenvolver uma parte do sistema.

Como os programadores são exigentes, vai existir um conjunto grande de jogos com diversos times e vários jogadores. No final de cada jogo, é necessário compilar um resumo da partida e enviar estes dados para um servidor remoto.

Os eventos que podem ser enviados durante uma partida são os seguintes:

Início da partida;
Gol;
Substituição;
Cartão;
Intervalo;
Início do segundo tempo;

Fim do jogo.

Como você resolveria este problema? Como mapear este modelo de negócio para um modelo de software de forma mais clara?

1.2 UM MODELO PARA O PROBLEMA

O primeiro passo para tratarmos do problema é criar um modelo do nosso domínio. Para os não iniciados, um modelo de domínio é o modelo conceitual de todos os tópicos que envolvem nosso problema.

Este modelo de domínio possui um vocabulário próprio. Por exemplo, não precisamos dizer a um especialista de futebol o que é um jogador, o que é uma bola, um tempo, uma falta etc. Este vocabulário já é conhecido pelos especialistas, e utilizar a linguagem que os especialistas utilizam é a ideia por trás das linguagens específicas de domínio.

Para o nosso problema, então, podemos ter a seguinte modelagem:

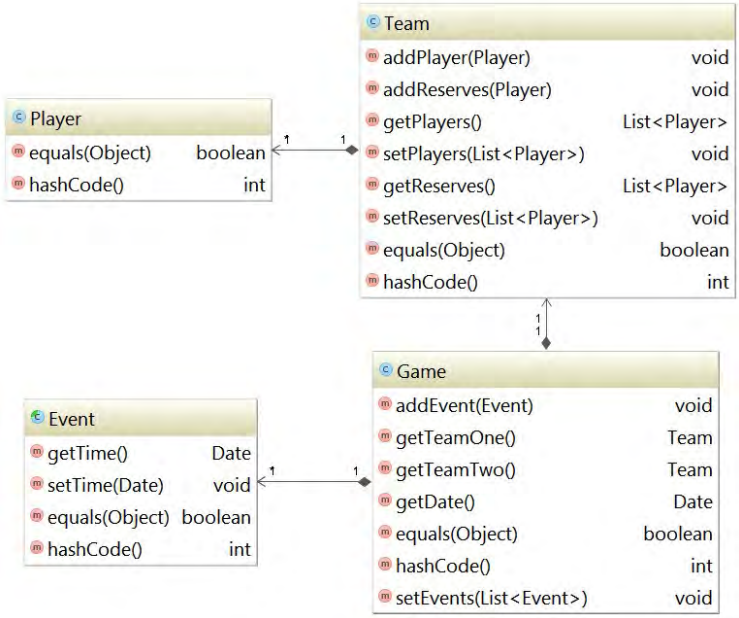

Figura 1.1: Modelo do Futebol dos Programadores

O modelo é bem direto. A classe `Game` representa um jogo entre dois times em uma determinada data. A classe `Event` representa todo um conjunto de eventos dentro de um jogo. Temos também a classe `Team`, que representa um time com seus jogadores e reservas.

Dentro do jogo, podemos ter os seguintes acontecimentos:

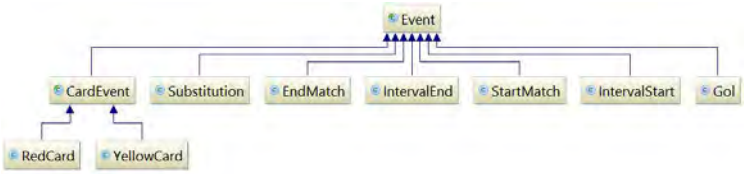

Figura 1.2: Eventos do Futebol dos Programadores

Cada uma destas classes representa um evento. Temos `CardEvent` para cartões, `Interval` para o intervalo entre os tempos, `StartMatch` para iniciar a partida e assim por diante. Todos estes eventos são tipos mais especializados do supertipo `Event`.

Este modelo é bem simples e você mesmo pode desenvolvê-lo em seu computador. Mas se você quiser, pode baixar este exemplo diretamente do repositório com os exemplos do livro, em https://bitbucket.org/leonardootto/dsl-exemplos/src.

1.3 JAVA E SCALA

Utilizei neste livro as linguagens Java e Scala. Java é a segunda mais usada em tecnologia, segundo o site Tiobe (http://tiobe.com). Além disso, é uma das linguagens com que estou bem familiarizado.

A linguagem C# foi amplamente baseada na Java, e os exemplos em Java podem ser facilmente convertidos para C#. Esta conversão pode até melhorar a DSL, visto que C# tem uma maior flexibilidade do que Java para algumas construções. Não é necessário conhecer C#, mas se você tiver conhecimento nesta linguagem, acredito que não terá problemas de compreensão nos exemplos em Java, ou até em Scala.

Scala também foi escolhida para representar as DSL internas aqui presentes, principalmente porque é uma linguagem que vem ganhando muito destaque dentro da comunidade Java. É estaticamente tipada e combina de forma muito transparente com programas escritos em Java.

A migração de Java para Scala é muito fluida, apesar de criticarem sua complexidade. Em boa parte, é uma linguagem bem simples para quem já tem conhecimento intermediário ou avançado em Java.

1.4 UTILIZANDO O MODELO

Definido o modelo, como o utilizamos? Primeiramente, precisamos usar alguma linguagem para implementar este modelo. Neste livro, escolhi usar Java e Scala. Para simplificar, vamos usar Java para fazer esta modelagem.

Inicialmente, criamos os times que vão se enfrentar. O código poderia ser algo como o seguinte:

```
public class Main {
    public static void main(String[] args) {
        //Define Teams
        Team languageTeam = new Team("Language Team");
        Team doItTeam = new Team("Do it Team");
    }
}
```

Uma vez que temos os times, precisamos dos jogadores que participarão deles:

```
public class Main {
    public static void main(String[] args) {
        //Define Teams
        Team languageTeam = new Team("Language Team");
        Team doItTeam = new Team("Do it Team");

        // Define languageTeam Players

        // C Creator
        Player ritchie = new Player("Dennis Ritchie");
        // Java
        Player gosling = new Player("James Gosling");
```

```
        // Python
        Player guido = new Player("Guido van Rossum");

        //Define doItTeam Players

        // Tdd creator
        Player beck = new Player("Kent Beck");
        // Many Many Books
        Player fowler = new Player("Martin Fowler");
        // Stackoverflow
        Player atwood = new Player("Jeff Atwood");
    }
}
```

Nesse momento, já temos alguns jogadores, mas eles não fazem parte de nenhum time. Vamos escalá-los nos times corretos pelo método `addPlayer` da classe `Team`:

```
public class Main {
    public static void main(String[] args) {
        // Define Teams
        // ...

        // Define languageTeam Players
        // ...

        // Define doItTeam Players
        // ...

        //Wire Players
        languageTeam.addPlayer(ritchie);
        languageTeam.addPlayer(gosling);
        languageTeam.addPlayer(guido);

        //Wire Players
        doItTeam.addPlayer(beck);
        doItTeam.addPlayer(fowler);
        doItTeam.addPlayer(atwood);
    }
}
```

Se quisermos escalar algum jogador como reserva, podemos chamar o método `addReserves`:

```
languageTeam.addReserves(bjarne);
```

Veja como fica a escalação completa dos times. O código é longo, mas também bem simples:

```
public class Main {
    public static void main(String[] args) {
        //Define Teams
        Team languageTeam = new Team("Language Team");
        Team doItTeam = new Team("Do it Team");

        //Define languageTeam Players
        // C Creator
        Player ritchie = new Player("Dennis Ritchie");
        // Java
        Player gosling = new Player("James Gosling");
        // Python
        Player guido = new Player("Guido van Rossum");
        // LISP
        Player mcCarthy = new Player("John McCarthy");
        // Ruby
        Player matsumoto = new Player("Yukihiro Matsumoto");
        // Swift
        Player lattner = new Player("Chris Lattner");
        // Perl
        Player wall = new Player("Larry Wall");
        // Javascript
        Player eich = new Player("Brendan Eich");
        // SmallTalk
        Player kan = new Player("Alan kan");
        // Scala
        Player odersky = new Player("Martin Odersky");
        // Fortran
        Player backus = new Player("John Backus");
        // C++ creator;
        Player bjarne = new Player("Bjarne Stroustrup");

        //Define doItTeam Players
        // Tdd creator
        Player beck = new Player("Kent Beck");
        // Many Many Books
        Player fowler = new Player("Martin Fowler ");
        // Stackoverflow
        Player atwood = new Player("Jeff Atwood");
```

```
        // Stackoverflow
        Player spolsky = new Player("Joel Spolsky");
        // LISP Hacker
        Player graham = new Player("Paul Graham");
        // Unix co-creator
        Player thompson = new Player("Ken Thompson");
        // Unix co-creator
        Player kernighan = new Player("Brian Kernighan");
        // Clean X
        Player martin = new Player("Robert C. Martin");
        // The art of Unix programming
        Player raymond = new Player("Eric C. Raymond");
        // GNU creator
        Player stallman = new Player("Richard Stallman");
        // Linux and Git creator
        Player linus = new Player("Linus Torvalds");

        //Wire Players
        languageTeam.addPlayer(ritchie);
        languageTeam.addPlayer(gosling);
        languageTeam.addPlayer(guido);
        languageTeam.addPlayer(mcCarthy);
        languageTeam.addPlayer(matsumoto);
        languageTeam.addPlayer(lattner);
        languageTeam.addPlayer(wall);
        languageTeam.addPlayer(eich);
        languageTeam.addPlayer(kan);
        languageTeam.addPlayer(odersky);
        languageTeam.addPlayer(backus);
        //Wire Reserves
        languageTeam.addReserves(bjarne);

        //Wire Players
        doItTeam.addPlayer(beck);
        doItTeam.addPlayer(fowler);
        doItTeam.addPlayer(atwood);
        doItTeam.addPlayer(spolsky);
        doItTeam.addPlayer(graham);
        doItTeam.addPlayer(thompson);
        doItTeam.addPlayer(kernighan);
        doItTeam.addPlayer(martin);
        doItTeam.addPlayer(raymond);
        doItTeam.addPlayer(stallman);
        doItTeam.addPlayer(linus);
    }
```

```
}
```

Uma vez que temos os times prontos e escalados, o jogo pode acontecer. No decorrer dele, alguns eventos vão acontecer, como faltas, gols, cartões, impedimentos etc. Nosso código precisa refletir isso também. Vamos usar a classe `Game` para adicionar seus diferentes eventos:

```
public class Main {
    public static void main(String[] args) {
        // Define players and teams

        // what happened in the game ?
        Game game = new Game("06/07/2014", languageTeam, doItTeam
);
        game.addEvent(new StartMatch("14:00"));
        game.addEvent(new Gol(odersky, "14:33"));
        game.addEvent(new YellowCard("14:35", backus));
        game.addEvent(new IntervalStart("14:47"));
        game.addEvent(new IntervalEnd("15:00"));
        game.addEvent(new Gol(linus, "15:05"));
        game.addEvent(new YellowCard("15:15", stallman));
        game.addEvent(new Substitution("15:25", ritchie, bjarne))
;
        game.addEvent(new EndMatch("15:48"));
    }
}
```

Pronto, agora temos a partida completamente definida.

Este exemplo demonstra como é a utilização do modelo de software. Ele não tem nada de muito interessante, é um modelo direto e muito verboso. Se ele fosse apresentado para alguma pessoa da área deste domínio, no caso futebol, seria bem provável que ela não teria um entendimento muito claro.

Como aumentamos a clareza da nossa solução? Podemos representar esta informação de outras formas. Existem vários modelos de representação de informação, e conhecer mais

modelos agrega no conhecimento como desenvolvedor e é um passo em direção à construção de DSLs.

Vamos conhecer mais um modelo na próxima seção. Este é muito utilizado para representar informações e vai ajudar a deixar mais claro o que estamos fazendo.

Representação XML

O XML é um formato comum muito usado para representação de informações. Poderíamos usá-lo para representar a informação dos nossos jogos também. Uma possível representação seria a seguinte:

```
<teamOne>
    <name>Language Team</name>
</teamOne>
<teamTwo>
    <name>Do it Team</name>
</teamTwo>
```

Depois, como no exemplo anterior, definimos os jogadores que participarão de cada time:

```
<teamOne>
    <name>Language Team</name>
    <players>
        <player>Dennis Ritchie</player>
        <player>James Gosling</player>
        <player>Guido van Rossum</player>
    <players>
</teamOne>
<teamTwo>
    <name>Do it Team</name>
    <players>
        <player>Kent Beck</player>
        <player>Martin Fowler</player>
        <player>Jeff Atwood</player>
    </players>
```

```
</teamTwo>
```

Como o XML já define a estrutura dos dados, cada jogador está adicionado corretamente em seu respectivo time. Para adicionar os reservas, fazemos de forma similar, da seguinte maneira:

```
<teamOne>
    <name>Language Team</name>
    <players>
        ...
    <players>
    <reserves>
        <player>Bjarne Stroustrup</player>
    </reserves>
</teamOne>
```

A escalação completa é um pouco longa, mas menor do que a representação em Java que tínhamos anteriormente. Podemos ver toda a estrutura representada a seguir:

```
<teamOne>
    <name>Language Tools</name>
    <players>
        <player>Dennis Ritchie</player>
        <player>James Gosling</player>
        <player>Guido van Rossum</player>
        <player>John McCarthy</player>
        <player>Yukihiro Matsumoto</player>
        <player>Chris Lattner</player>
        <player>Larry Wall</player>
        <player>Brendan Eich</player>
        <player>Alan kan</player>
        <player>Martin Odersky</player>
        <player>John Backus</player>
    </players>
    <reserves>
        <player>Bjarne Stroustrup</player>
    </reserves>
</teamOne>
<teamTwo>
    <name>Do it Team</name>
    <players>
```

```
        <player>Kent Beck</player>
        <player>Martin Fowler</player>
        <player>Jeff Atwood</player>
        <player>Joel Spolsky</player>
        <player>Paul Graham</player>
        <player>Ken Thompson</player>
        <player>Brian Kernighan</player>
        <player>Robert C. Martin</player>
        <player>Eric C. Raymond</player>
        <player>Richard Stallman</player>
        <player>Linus Torvalds</player>
    </players>
</teamTwo>
```

Um jogo é formado por dois times. Já definimos os times, então falta definir a partida que os contém.

```
<game>
    <date>06/07/2014</date>
    <teamOne>
    ...
    </teamOne>
    <teamTwo>
    ...
    </teamTwo>
</game>
```

Nenhuma partida é interessante se não tiver nenhum acontecimento. Então, vamos adicionar os eventos (acontecimentos) da partida.

```
<game>
    <date>06/07/2014</date>
    <teamOne> ... </teamOne>
    <teamTwo> ... </teamTwo>
    <events>
       <startMatch>14:00</startMatch>
       <endMatch>15:47</endMatch>
       ...
    </events>
</game>
```

Uma lista mais completa de eventos poderia ser a seguinte:

```
...
    <events>
        <startMatch>14:00</startMatch>
        <gol>
            <time>14:33</time>
            <player>Martin Odersky</player>
        </gol>
        <yellowCard>
            <time>14:35</time>
            <player>John Backus</player>
        </yellowCard>
        <intervalStart>14:47</intervalStart>
        <intervalEnd>15:00</intervalEnd>
        <gol>
            <time>15:05</time>
            <player>Linus Torvalds</player>
        </gol>
        <yellowCard>
            <time>15:15</time>
            <player>Richard Stallman</player>
        </yellowCard>
        <substitution>
            <time>15:25</time>
            <in>
                <player>Bjarne Stroustrup</player>
            </in>
            <out>
                <player>Dennis Ritchie</player>
            </out>
        </substitution>
        <endMatch>15:48</endMatch>
    </events>
...
```

Uma representação compacta seria então:

```
<game>
    <date>06/07/2014</date>
    <teamOne>
        ...
    </teamOne>
    <teamTwo>
```

```
        ...
    </teamTwo>
    <events>
        ...
    </events>
</game>
```

A representação em XML é bastante familiar para a maioria dos leitores. XML é uma linguagem bem comum para estruturar dados. Representar a estrutura de um jogo desta maneira teria suas vantagens, como por exemplo, não precisar de uma nova compilação para alterar configurações, ou transmitir estas configurações por outros meios mais facilmente.

Utilizei XML para exemplificar como descrever o modelo de forma mais simples. Para casos simples, modelar em XML é uma melhor escolha de implementação. Mas se for necessário mais expressividade e flexibilidade, podemos melhorar a implementação.

Melhorando o modelo

Podemos começar a utilizar XML para descrever todos os modelos dentro de nosso sistema. Mas será que isso resolve nossos problemas de expressividade?

Um problema do XML para representar DSL é que XML é um formato de dados, e ele não foi feito para descrever comportamento. Então, precisamos das ideias de modelagem da informação do XML e da flexibilidade de uma linguagem de propósito geral.

Para simplificar, poderíamos modificar nosso modelo, deixando-o de uma forma mais fluida dentro do Java. Algo que

poderia ser próximo do seguinte:

```
game()
.schedule("06/07/2014")
.team("Language Team",
        player("Dennis Ritchie"),
        player("James Gosling"),
        player("John Backus"),
        reserve("Bjarne Stroustrup")
).team("Do it Team",
        player("Kent Beck"),
        player("Martin Fowler"),
        player("Jeff Atwood")
).events(
        start("14:00"),
        yellowCard("15:05").to("John Backus"),
        intervalStart("14:47"),
        intervalEnd("15:00"),
        substitution("15:25",
                     out("Dennis Ritchie"),
                     in("Bjarne stroustrup")),
        end("15:27")
);
```

Neste exemplo, utilizamos funções aninhadas com encadeamento de métodos, mas não se preocupe com isso por enquanto. Posteriormente, vamos retornar a cada uma destas técnicas para você compreendê-las mais profundamente. Removi alguns jogadores apenas para melhorar a representação.

Esta versão de código é muito mais compacta, direta e simples de ser usada. Entretanto, existe um pouco de ruído por causa da sintaxe Java. Esta fluência é encontrada em algumas APIs, como por exemplo, a JMock ou a Fluent Mail Api. Mas será que conseguimos simplificar mais ainda esta API se utilizássemos alguma linguagem mais flexível e com menos ruído?

Que tal tentarmos com Scala? É uma linguagem superflexível e amigável para programadores Java.

Scala

Scala será a outra linguagem usada neste livro. É uma linguagem compilada, estaticamente tipada e com muitas funcionalidades interessantes. Temos um apêndice para explicar as funcionalidades usadas neste livro, então não é necessário compreender todos os detalhes agora. Se tiver alguma dúvida pode consultar o *Apêndice — Scala*. Lá teremos uma descrição um pouco mais detalhada sobre as funcionalidades da linguagem que utilizamos aqui.

Estamos convertendo o código Java para Scala para reduzir a verbosidade do Java e aproveitar a flexibilidade do Scala para fazer um modelo muito próximo da linguagem do especialista de domínio. O exemplo anterior em Java poderia ser escrito da seguinte maneira, utilizando Scala:

```
game("06/07/2014",
    team("Language Tools") {
        players(
          "Dennis Ritchie",
          "James Gosling",
          "Guido van Rossum",
          "John McCarthy",
          "Yukihiro Matsumoto",
          "Chris Lattner",
          "Larry Wall",
          "Brendan Eich",
          "Alan kan",
          "Martin Odersky",
          "John Backus"
        ) reserves (
          "Bjarne Stroustrup"
        )
    },
    team("Do it Team") {
        players(
          "Kent Beck",
```

```
            "Martin Fowler",
            "Jeff Atwood",
            "Joel Spolsky",
            "Paul Graham",
            "Ken Thompson",
            "Brian Kernighan",
            "Robert C. Martin",
            "Eric C. Raymond",
            "Richard Stallman",
            "Linus Torvalds"
        )
    },
    events(
        start at "14:00",
        gol from "John Backus" at "14:33",
        yellowCard at "14:35" to "John Backus",
        intervalStart at "14:47",
        intervalEnd at "15:00",
        gol from "Linus Torvalds" at "15:05",
        yellowCard at "15:15" to "Richard Stallman",
        substitution at "15:25" out "Dennis Ritchie" in "Bjarne s
troustrup",
        end at "15:48"
    )
```

O exemplo Scala simplifica um pouco do ruído da linguagem. Os ruídos de linguagem são todos estes símbolos extras que temos de adicionar em um código para ele funcionar dentro da nossa linguagem. Veremos mais sobre isso por todo o livro, mas por enquanto imagine que, quanto menor a quantidade de símbolos extras que adicionamos, mais clara fica a leitura.

Por exemplo, anteriormente tínhamos no código Java o seguinte:

```
...
substitution("15:25",
             out("Dennis Ritchie"),
             in("Bjarne stroustrup")),
...
```

E posteriormente, convertemos este código para Scala da seguinte maneira:

```
  ...
  substitution at "15:25" out "Dennis Ritchie" in "Bjarne stroust
rup",
  ...
```

Este exemplo também nos aproxima mais de algo próximo do que um especialista de domínio usaria para representar este caso. Por exemplo, se ele fosse um comentarista esportivo, ele diria algo assim em linguagem nativa:

```
    Ocorreu uma substituição aos 15 minutos de jogo trocando o jo
gador Dennis Ritchie
    e entrando o jogardor Bjarne stroustrup
```

Temos, por fim, o seguinte exemplo:

```
    game
        schedule 06/07/2014
        team Language Team
            Dennis Ritchie
            James Gosling
            Guido van Rossum
            John Backus
        reserves
            Bjarne Stroustrup
        team Do it Team
            Kent Beck
            Martin Fowler
            Jeff Atwood
        event
          start 14:00
          gol 15:05 Linus Torvalds
          yellowCard 15:05 John Backus
          intervalStart 14:47
          intervalEnd 15:00
          substitution 15:25 out Dennis Ritchie in Bjarne Stroust
rup
          end 15:47
```

Este exemplo simplifica ao máximo o que queremos modelar, praticamente da mesma forma que um especialista utilizaria para descrever o evento. Estes últimos exemplos, ao contrário do primeiro, são exemplos de Linguagens Específicas de Domínio (DSLs).

Estas linguagens não fazem nada além de descrever um domínio específico, que no nosso caso é uma partida de futebol. Mas não se preocupe em entender a fundo ainda sobre DSLs, teremos uma descrição mais detalhada e formal no *capítulo 2*.

Esta simplicidade torna os programas mais fáceis de serem compreendidos e editados. Algumas vezes, ela também torna o comportamento visível para além dos desenvolvedores, chegando até aos especialistas de domínio.

> Neste exemplo, estamos realizando apenas a modelagem da DSL. Não adicionei ou fiz menção a códigos que tratam de requisições para servidores, banco de dados ou qualquer outro assunto que não seja relacionado a DSL. Isso apenas atrapalharia o entendimento com coisas desnecessárias e fora do escopo do livro.

1.5 FOCO NAS DSLS INTERNAS

O exemplo em XML e o último têm características especiais, porque eles não executam dentro de uma linguagem, não estão dentro do arquivo fonte — são arquivos separados. Ou seja, eles não possuem uma linguagem que os hospeda, não têm linguagem

hospedeira. Estes exemplos não seguem a sintaxe da linguagem hospedeira, no caso Java e Scala, e utilizam sua própria sintaxe.

Isso cria a primeira divisão dentro das DSLs. Dizemos que **DSLs internas** são as que estão dentro de uma linguagem hospedeira, enquanto as **DSLs externas** são arquivos separados de arquivos fonte e processados de forma diferenciada.

Um outro termo que talvez você tenha escutado é interface fluente (*Fluent interface*), que designa DSLs internas que possuem uma certa fluência na sua utilização. Um termo usado para contrastar com a interface fluente seria a interface padrão que usamos normalmente, nomeada por alguns autores como API Comando consulta. Para deixar mais claro, esta interface padrão, ou comando consulta, é onde temos métodos `get and set` para definir atributos junto com outros métodos que realizam comportamentos.

As DSL externas necessitam de um conhecimento teórico maior e são mais complexas para criar, por isso este livro é focado principalmente em **DSLs internas**. Isso é feito para trazer um conteúdo mais leve e fácil de absorver. Entretanto, linguagens externas e internas têm uma sinergia e, quando necessário, discuto como elas interagem.

1.6 E AGORA?

Agora que já conhecemos o que são DSLs em linhas gerais, podemos entrar mais a fundo no assunto. No próximo capítulo, apresentarei várias DSLs em que você talvez nunca tenha reparado, como também definições um pouco mais concretas e técnicas de

construção. Além disso, vamos diferenciar DSL de outros tipos de APIs, e se devemos ou não usá-las para resolver um problema.

Capítulo 2

ADENTRANDO NAS DSLS

Neste capítulo, vamos demonstrar alguns exemplos e uma definição mais casual de DSL. Entraremos mais a fundo no conceito de modelo de domínio e modelo semântico, e veremos como a DSL se relaciona com este último.

Para um leitor que já conhece um pouco sobre DSLs, pode passar diretamente para o tópico que deseja aprender ou reforçar. A um leitor que está começando agora, isso permitirá que ele perceba e absorva melhor cada estilo de DSL para adentrar nos detalhes nos capítulos posteriores.

Vamos demonstrar várias técnicas de construção de DSL para guiar o desenvolvedor quando estiver criando a sua. Será uma apresentação aos conceitos, então não se preocupe em entender todos os detalhes de implementação. Posteriormente, vamos nos aprofundar em cada um deles.

2.1 MODELO DE DOMÍNIO

A modelagem de domínio é uma atividade na qual analisamos e identificamos as entidades envolvidas nos problemas para compreendermos seu funcionamento. É necessário entender como as entidades interagem entre si para cumprir seus objetivos.

No exemplo introdutório, analisamos e identificamos as entidades relacionadas a um jogo de futebol para compreender como ele deve ser mapeado. Os especialistas de domínio conhecem muito sobre o domínio relacionado e utilizam o mesmo vocabulário para explicar o modelo em qualquer lugar.

Por exemplo, o objeto que os jogadores chutam se chama bola, o lugar que se tem de colocar a bola se chama gol etc. Tudo relacionado ao jogo tem nomes distintos e bem definidos por todos os especialistas deste domínio do futebol.

Os modeladores sabem como usar este vocabulário, documentar, compartilhar e implementá-lo em software. Eles também devem entender o mesmo vocabulário usado pelos especialistas.

Para fazer esta modelagem, podemos usar ferramentas e técnicas. Quando se está trabalhando com domínios complexos, estas ferramentas e técnicas se fazem ainda mais necessárias.

Não tratamos profundamente sobre modelagem de domínio neste livro, mas se quiser saber mais, recomendo os livros:

> *Padrões de Arquitetura de Aplicações Corporativas*, do Martin Fowler;
> *Domain-Driven Design: Tackling Complexity in the Heart of Software*, do Eric Evans;
> *Design Patterns: Elements of Reusable Object-Oriented Software*, da Gang of Four (Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides);
> *Clean code: A Handbook of Agile Software Craftsmanship*, do Robert Martin (Uncle Bob);

Desenvolvimento de Software Orientado a Objetos, Guiado por Testes, do Steve Freeman e Nat Pryce.

A Casa do Código também tem vários livros que auxiliam uma melhor modelagem de software, como *Design Patterns com Java: Projeto orientado a objetos guiado por padrões*, *Introdução à Arquitetura e Design de Software: Uma visão sobre a plataforma Java*, *Desbravando Java e Orientação a Objetos: Um guia para o iniciante da linguagem*, entre outros.

Como as linguagens Java e Scala são orientadas a objetos, você usará classes, métodos e objetos, e mapeará os problemas utilizando estes artefatos fornecidos por elas.

Quando o modelo de domínio e o modelo em software compartilham um vocabulário similar, dizemos que existe uma interação forte dos modelos. Como por exemplo, no exemplo inicial. Lá temos objetos representando evento, jogo, jogadores e times, e estes nomes são os mesmos usados pelos especialistas de domínio para representar um jogo.

2.2 MODELO SEMÂNTICO

O objetivo de uma DSLs, seja ela interna ou externa, é preencher um modelo. O nome dado por Fowler (2013) a este modelo é *Modelo semântico*.

Sintaxe e semântica são normalmente palavras que andam juntas. A sintaxe é a gramática de uma linguagem, definindo quais palavras devem estar em que lugar. Já a semântica é o que aquele conjunto de palavras significa. Em programação, seria o que ele faz quando é executado.

Neste caso, é o modelo que define a semântica, por isso este nome. Esta distinção do modelo de domínio e modelo semântico existe, pois o de domínio tende a ser mais rico e complexo, enquanto o semântico pode ser apenas uma estrutura de dados simples. Os dois modelos também podem ser definidos em classes diferentes, ou ser a mesma classe.

Por exemplo, poderíamos ter um modelo de software que representasse um conceito complexo com classes. Digamos uma classe chamada `ClasseComplexa`. Posteriormente, para simplificar a interação de criação da `ClasseComplexa`, resolveríamos criar uma DSL

Ou seja, criamos uma classe que possui significado apenas na DSL para representar este conceito. Este vai ser nosso modelo semântico.

2.3 UMA DSL PARA CADA PROBLEMA

Existe um conjunto grande de DSLs comumente utilizadas em programação. Algumas delas você pode ter usado mesmo sem saber que eram DSLs. Vou demonstrar algumas delas para posteriormente criar uma categorização segundo suas semelhanças.

CSS

O exemplo talvez mais comum de uma DSL é o CSS, por exemplo:

```
p {
    text-align: center;
    color: red;
```

```
}
```

CSS é a linguagem para definir estilos dentro do HTML. Possivelmente, é a DSL mais usada atualmente. Ela é limitada a um problema específico e tem uma expressividade reduzida, apesar de bem extensa.

As últimas inovações dentro desta área são as linguagens intermediárias, como LESS ou SASS. Estas, quando compiladas, se transformam em arquivos CSS. A utilidade principal é trazer mais flexibilidade, trazendo construções de linguagem de programação como variáveis, mixins, aninhamentos, entre outros.

Essas linguagens intermediárias também são DSLs. E como elas não dependem da sintaxe da linguagem hospedeira, elas podem ser consideradas DSL externas.

Expressões regulares

Expressões regulares são um conjunto de textos e símbolos usados para encontrar padrões dentro de textos, e são outro exemplo óbvio de DSL. Um exemplo simples de expressão regular para encontrar um e-mail seria algo como:

```
.+@.+\.[a-z]+
```

Expressões regulares são muito usadas, mas normalmente não estão dentro da linguagem na qual são utilizadas. Por exemplo, em Java, para usarmos esta expressão regular, utilizaríamos uma biblioteca da seguinte forma:

```
    //Lembre-se de que precisamos fazer o scape do '\' por isso '
\\'
    Pattern padrao = Pattern.compile(".+@.+\\.[a-z]+");
```

```
Matcher pesquisa = padrao.matcher("leonardootto@gmail.com");

if (pesquisa.matches()){
    System.out.println("O email é valido!");
}else{
    System.out.println("O email não é valido!");
}
```

Logo, existem DSLs que estão definidas dentro da linguagem e outras que são definidas fora dela. O caso das expressões regulares é um pouco diferente, pois elas podem estar em um arquivo separado. Elas não dependem da sintaxe da linguagem, logo, podemos considerar que são uma DSL externa.

Sed

Sed é uma ferramenta para realizar modificações dentro de arquivos. Ela é uma ferramenta muito antiga, criada em 1973, e continua a ser usada até hoje em dia. Sed vem da palavra inglesa *Stream EDitor*, que quer dizer *editor de streams*.

Ela foi uma das primeiras ferramentas a suportar expressões regulares internamente, e é bem compacta em suas construções. Por exemplo, vamos dizer que temos o script:

```
>> echo "Olá Leonardo. Tudo bem?" | sed s/Leonardo/José/
Olá José. Tudo bem?
```

Neste script, fazemos a substituição da palavra `Leonardo` pela palavra `José` dentro do texto. O caractere `s` é um comando do Sed que informa que vamos fazer uma substituição, e a barra é um delimitador qualquer que podemos utilizar. O comando `echo` é apenas para mostrar o texto, e a barra é para colocar a saída de um comando dentro de outro comando.

Podemos usar qualquer caractere como delimitador. Por

exemplo, poderíamos ter o seguinte comando similar:

```
>> echo "Olá Leonardo. Tudo bem?" | sed s:Leonardo:José:
   Olá José. Tudo bem?
```

Algumas vezes você pode querer adicionar alguma coisa ao texto. Vamos dizer que você gostaria de adicionar parênteses junto ao nome. Então, você faria o seguinte:

```
>> echo "Olá Leonardo. Tudo bem?" | sed 's:Leonardo:(&):'
   Olá (Leonardo). Tudo bem?
```

Sed possui uma linguagem que pode ser descrita como uma DSL. Ela tem um domínio específico e é uma ferramenta interessante para se aprender.

Fluent API

Outro exemplo bem interessante foi o usado pela Fluent Mail API (CHAPIEWSKI, 2008), que é uma DSL construída sobre a API padrão de envio e recebimento de e-mails do Java.

```
new EmailMessage()
    .from("demo@guilhermechapiewski.com")
    .to("destination@address.com")
    .withSubject("Fluent Mail API")
    .withBody("Demo message")
    .send();
```

Esta DSL é muito mais compacta que a utilização da API padrão de e-mail do Java, facilitando o trabalho de enviar e-mails. É um ótimo exemplo de uso de DSL interna para tratar um problema específico, neste caso o envio de e-mail.

Build

Existem dois tipos de linguagens, as compiladas e as

interpretadas. Como se vê pelos nomes, as compiladas sofrem o processo de compilação, enquanto as interpretadas executam utilizando um interpretador.

O processo de compilação é a transformação da linguagem entendida por pessoas em uma linguagem que pode ser entendida por computadores. A maioria das linguagens compiladas dá ferramentas que auxiliam no processo de compilação, fornecendo facilidades como estruturas padrão, plugins com utilitários, entre outros.

Dentro da linguagem Java, temos várias ferramentas de build, sendo que uma muito conhecida é a Ant. Um exemplo de uso desta ferramenta seria o seguinte:

```
    <project>
        <target name="clean">
            <delete dir="build"/>
        </target>

        <target name="compile">
            <mkdir dir="build/classes"/>
            <javac srcdir="src" destdir="build/classes"/>
        </target>

        <target name="jar">
            <mkdir dir="build/jar"/>
            <jar destfile="build/jar/HelloWorld.jar" basedir="bui
ld/classes">
                <manifest>
                    <attribute name="Main-Class" value="oata.Hell
oWorld"/>
                </manifest>
            </jar>
        </target>

        <target name="run">
            <java jar="build/jar/HelloWorld.jar" fork="true"/>
        </target>
```

```
</project>
```

Este arquivo do Ant representa uma linguagem de construção e é também uma DSL.

SQL

O SQL (*Structured Query Language*) é a linguagem padrão usada para trabalhar com banco de dados relacionais. Ela é declarativa, ou seja, informa como deve ser o resultado, e não o caminho para chegar até ele.

Um exemplo de seu uso poderia ser o seguinte. Tendo a seguinte tabela `'T'`:

```
c1  c2
1    a
2    b
```

Se executássemos o seguinte comando:

```
SELECT c1 FROM T;
```

Obteríamos o seguinte resultado:

```
c1
1
2
```

A sua utilização é bem simples e, na carreira de um desenvolvedor de software, é improvável que ele não vá se deparar com SQL em algum momento. Esta linguagem é um exemplo clássico de DSLs, pois trata de um domínio específico.

Como disse anteriormente, existem vários tipos diferentes de DSLs. Mas será que poderíamos defini-las de alguma forma mais

prática?

2.4 DEFINIÇÕES

Uma DSL deve ser focada em um domínio específico, ou seja, ela não tenta resolver todos os problemas de domínios múltiplos.

Como toda linguagem de programação, a DSL é construída para ser entendida por computadores, mas também deve facilitar o entendimento de humanos. Linguagens de propósito geral possuem muitos recursos, e dificultam o aprendizado e o uso. Uma DSL, ao contrário, deve suportar um mínimo de recursos necessários para resolver o problema do domínio.

Então, podemos definir uma DSL como:

> **Linguagens específica de domínio**
>
> Uma linguagem de programação de computadores de expressividade limitada, focada em um domínio específico.

Esta definição é útil, pois é bem abrangente e restringe principalmente os pontos principais da linguagem.

O valor das DSLs está exatamente neste ponto: serem focadas para resolver apenas um problema da forma mais clara possível. O propósito por trás delas é trazer uma interface mais humana para os usuários finais. E a melhor forma de fazer isso é com um modelo que fala a língua do domínio.

Categorizações

Como apresentado na seção *Foco nas DSLs Internas*, existem duas divisões principais dentro das DSLs:

> **DSL internas:** uma DSL que é descrita usando a sintaxe de uma linguagem de propósito geral, de uma maneira especial, para ser vista como uma linguagem customizada. De uma forma mais simples, ele está dentro de um arquivo fonte de uma linguagem de propósito geral.
>
> **DSL externas:** uma linguagem separada da linguagem da aplicação. Logo, possui uma sintaxe que pode ser totalmente diferente da linguagem da aplicação, e não está limitada a esta sintaxe. Pode ainda usar uma sintaxe comum, como por exemplo, XML.

Utilizando esta definição geral, já podemos categorizar as DSLs anteriormente apresentadas. A Fluent Mail API usa a própria sintaxe da linguagem, ou seja, ele utiliza os mesmos elementos de uma linguagem já existente para criar sua própria DSL. Então, ela é uma DSL interna.

O SQL é uma linguagem totalmente separada da linguagem da aplicação, e não utiliza os mesmos elementos da linguagem da aplicação. Logo, ela é uma DSL externa. Já as expressões regulares são outro exemplo de DSL externa. Apesar de estarem dentro da linguagem da aplicação, elas normalmente são tratadas de forma especial.

O Sed e o Build também são exemplos bem poderosos de DSLs

externas. Aliás, fico feliz que não tenhamos de construir aplicações utilizando apenas expressões regulares. Imagine o caos que isso seria.

Um outro ponto de vista que se pode ter sobre as DSLs é elas serem uma fina camada de acesso a um modelo semântico, sendo que este pode ser uma biblioteca ou um modelo de domínio. O exemplo da Fluent Mail API combina bem neste caso, já que ela é uma fina camada para a DSL acessar a biblioteca de e-mail padrão do Java.

Uma dúvida que pode ocorrer com o leitor é como a API da DSL combina com a API padrão que usamos. Para entender isso, precisamos entender sobre DSL e comando consulta.

DSL e comando consulta

Podemos categorizar dois tipos principais de formas de construção de API dentro de Orientação a Objetos: a API Comando Consulta, na qual o código funciona como se fosse uma máquina; e a DSL, em que o código funciona como se fosse uma linguagem sendo utilizada.

Para as APIs comando consulta, os atributos dos objetos são inseridos e recuperados para fazer as operações. Este tipo de API é muito usado atualmente, e normalmente é o que se encontra na maioria das APIs. Métodos de acesso e de atribuição de valores, ou no padrão Java, `gets e sets` .

O que difere uma DSL de uma API comando consulta é o fato de que a preocupação da API é resolver o problema, enquanto a preocupação da DSL é fazer declarações de como deve ser

resolvido o problema. A DSL tende a se parecer muito com a linguagem natural neste caso.

Alguns métodos como `.with()`, `.and()` e `.or()` não seriam nada adequados para uma API padrão comando consulta. Mas seriam muito bem usados em uma DSL, já que a preocupação dela é grande em relação a como aquela ação deve ser definida linguisticamente.

Como descrito anteriormente, o foco deste livro é em DSL internas, já que elas são o primeiro passo para quem quer ter os benefícios de DSLs. Mas alerto ao leitor para não acreditar que DSLs internas devem ser utilizadas para todos os problemas. Para alguns deles, criar DSLs externas é mais simples do que DSLs internas.

Pilha de abstrações

Como explicado anteriormente, as DSLs focam em problemas de um domínio específico, enquanto linguagens gerais servem para múltiplos domínios. Outro ponto a se destacar é que elas possuem sintaxe e semântica para resolver os problemas do domínio, usando o mesmo nível de abstração que ele utiliza. O foco é resolver o problema do domínio, e não detalhes de implementação, ou elementos de fora que não sejam essenciais.

Uma forma de visualizar a relação entre as duas é imaginar que as DSLs são uma abstração sobre a implementação do domínio, e o modelo é uma abstração sobre a linguagem de propósito geral. Existem várias características sobre abstrações que são interessantes de se discutir.

Toda abstração provê uma ou mais funcionalidades para os agentes externos, e essas funcionalidades são conhecidas como interfaces que os clientes podem utilizar. Estas devem possuir algumas características para facilitar o design e a abstração ser de melhor qualidade.

Uma das características é que ela deve ser minimalista. Ou seja, a abstração deve expor em sua interface o mínimo possível para cumprir o seu propósito. Por exemplo, vamos dizer que possuímos uma interface da seguinte forma:

```
interface AcmeAbstraction{
    public HashMap<String,String> namesByType();
}
```

Neste código, a interface retorna um tipo `HashMap`, acoplando o código que utiliza a implementação concreta. Se futuramente a implementação mudar de `HashMap` para `TreeMap`, isso quebraria o código do cliente. Então, você deve abstrair este retorno com um tipo mais genérico.

A abstração também deve ser destilada, ou seja, ela não deve conter detalhes não essenciais ao domínio específico. O que distingue os detalhes essenciais dos não essenciais é o conhecimento claro do domínio, e os especialistas do domínio podem ajudar neste caso.

Outro ponto importante é que a abstração deve ser extensível e dividida em componentes. Ou seja, você deve poder estender sua abstração para suportar novas funcionalidades no futuro, e deve também poder criar composições destas abstrações para criar outras de mais alto nível.

Bancada de linguagens

Existe um outro ambiente popular de utilização e criação de DSLs, que é a **bancada de linguagens**. Uma bancada de linguagem é uma IDE customizada para especificar DSLs e criar scripts utilizando estas DSLs.

Alguns autores, como o Martin Fowler, consideram bancadas de linguagens como uma outra categoria de DSLs. Não gosto desta definição, pois elas são apenas um conjunto de ferramentas em volta de uma DSL externa. Logo, seriam um subconjunto das DSL externas, e não uma categoria diferente de DSL.

Um ambiente que poucas pessoas consideram de programação são as planilhas do Excel. Se fizermos uma análise, provavelmente o Excel é o ambiente de programação mais bem-sucedido atualmente. Ele é uma ferramenta ilustrativa de desenvolvimento, em que os números de cálculos são demonstrados visualmente. As planilhas servem tanto como ambiente de desenvolvimento quanto como visualização das informações.

Podemos considerar o Excel como uma bancada de linguagem. Provavelmente, teremos em um futuro mais ferramentas deste tipo, apesar de atualmente estarmos ainda iniciando neste foco.

2.5 DEVO UTILIZAR UMA DSL?

Existe um ponto de vista que diz que a profissão de programação é uma profissão de tradução. Estamos sempre traduzindo o que os especialistas de domínio estão dizendo para uma língua que os computadores entendam. Ou seja, capturamos as ideias de especialistas, as completamos (pois normalmente eles

eliminam detalhes) e traduzimos para outra linguagem. Este é o modelo ideal de desenvolvimento: utilizar uma linguagem similar para descrever as mesmas ideias para todos os envolvidos.

Eric Evans (2011) fala muito sobre isto em seu livro sobre DDD. Ele dá o nome desta linguagem única de Linguagem Ubíqua. Uma linguagem para todos utilizarem. Se o software é desenvolvido desta maneira, ele fica claro para o desenvolvedor, para quem mantém o código e até para os especialistas do domínio.

Se todas as partes do projeto, das funcionalidades, dos testes, das funções e objetos por todo o sistema usarem uma linguagem única, então teremos um sistema muito mais fácil de aprender e de se guiar por cada uma das suas diferentes fases ou artefatos.

Manter o código o mais próximo dos conceitos dos especialistas é um passo importante, mas este passo pode ser expandido ainda mais com o uso de DSLs. Utilizando DSL, podemos nos aproximar ainda mais da linguagem que os especialistas de domínio usam.

As DSLs também aprimoram entendimento e a produtividade no desenvolvimento. Compreender uma DSL é normalmente muito mais fácil do que compreender de que forma associar ou combinar uma API comando consulta. A leitura de uma DSL costuma ser muito mais clara. Esta clareza faz com que seja mais difícil cometer enganos, e encontrar onde podemos ter cometido algum deles.

Um modelo para um problema de software já fornece uma melhoria na produtividade, fornecendo uma abstração para se

trabalhar. E é sobre esta abstração que uma DSL trabalharia, melhorando ainda mais os benefícios que o modelo já dispõe.

Outro ponto positivo é a melhora na comunicação com os especialistas do domínio, que podem ler código DSL, apontando informações incorretas. Aliás, envolver especialistas de domínio na criação do modelo é um benefício apontado por vários desenvolvedores para torná-lo mais preciso e expressivo para o problema.

Os especialistas de domínio sem conhecimentos de programação não vão se tornar programadores por usarem DSLs, mas serão capazes de entender as abstrações usadas para implementar as regras do negócio, e se elas são adequadas para cobrir todos os seus cenários. Como elas são desenvolvidas em um alto nível de abstração, quem as utiliza não precisa se preocupar com detalhes de implementação de baixo nível.

O retorno na utilização delas costuma ser muito alto, ou seja, o retorno que elas trazem no ciclo de vida de todo software costuma ser muito maior. Além disso, estender o projeto costuma ser fácil, pois como temos uma abstração de alto nível, é mais simples compreender e, posteriormente, estender as implementações.

Para DSL externas, outra vantagem seria desacoplar o contexto de compilação para execução. Ou seja, poder mudar configurações da aplicação apenas modificando arquivos externos.

Uma razão para não se utilizar uma DSL é o custo da construção ser muito alto para se obter um pequeno benefício. As DSLs não sobrecarregam as linguagens de um sistema, muito pelo contrário. Elas facilitam o entendimento de uma abstração que

teria de ser aprendida.

Um problema que pode ser encontrado é quando a DSL cresce demasiadamente, se transformando em uma linguagem de propósito geral. Então, tente deixar a sua DSL o mais simples e focada possível.

Outra desvantagem de seu uso é que a implementação de linguagens é uma atividade difícil, e é um assunto que não pode ser gerenciado por programadores sem muita experiência. DSLs têm um custo de construção que talvez não seja pago se o projeto tem um modelo de domínio não muito complexo. Isso vai fazer com que o custo de construção seja mais alto do que seu benefício no projeto como um todo.

Então, podemos usar também quando temos um modelo de domínio que está crescendo em complexidade e queremos passar por um processo de melhoria na expressividade. Você também pode sofrer com alguma degradação de performance, já que é outra via de indireção que estamos adicionando. Assim, é necessário analisar se o projeto tem requisitos fixos de performance, e se o uso de DSL não vai impactar em sua utilização.

Ou seja, se seu software tem uma carga de acesso extensivamente grande, as respostas devem ser dadas no menor tempo possível. Além de a paralelização não poder ser usada, DSLs podem causar problemas em seu código. Mesmo neste caso, você pode implementar a DSL para avaliação, pois, como veremos posteriormente, a troca pode ser transparente se for bem construída.

Mas como implementar então a DSL de forma correta?

2.6 COMO IMPLEMENTAR DSLS INTERNAS

Já foi dito anteriormente que o objetivo de uma DSL é preencher um modelo semântico. Um modelo semântico costuma ser um subconjunto do modelo de domínio.

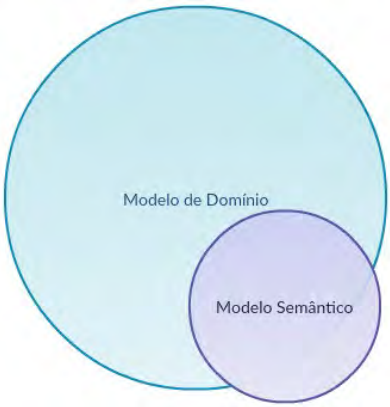

Figura 2.1: Modelo semântico

Uma boa prática para desenvolver sua DSL é deixá-la independente do modelo semântico. Desta maneira, podemos trabalhar com a DSL e o modelo independentemente.

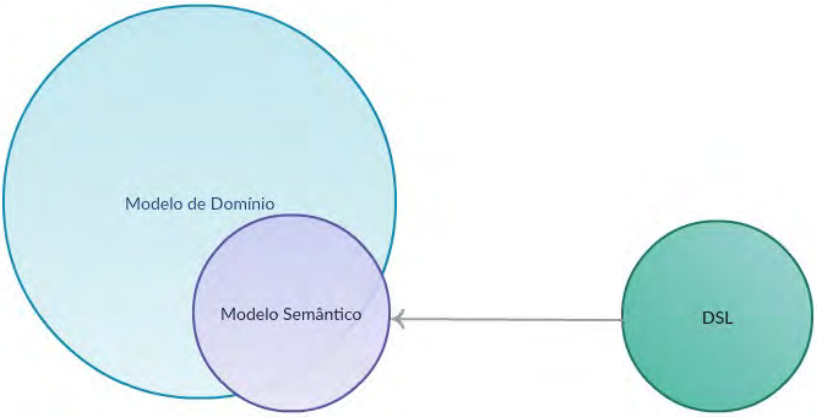

Figura 2.2: DSL acessando o modelo semântico

Algumas DSLs são construídas sem preencher um modelo semântico, mas recomendo quase sempre a utilização deste modelo de separação, pois ele isola as responsabilidades. Outra vantagem nesta independência é poder testar melhor como uma DSL funciona.

Como a resposta de uma DSL é um modelo semântico, podemos fazer o acesso pela DSL e verificar se o resultado será o mesmo ao usar a API comando consulta padrão. Apesar de a DSL e o modelo semântico ainda estarem ligados de alguma maneira, essa separação ajuda na evolução de ambas as partes.

Tudo o que você tem para implementar já está a sua disposição em sua linguagem nas DSLs internas. Boa parte do interesse sobre DSLs atualmente veio da comunidade Ruby, que incentiva a criação de DSL para resolver problemas. Algumas das técnicas usadas em Ruby podem também serem usadas em outras linguagens. Falando em linguagens, Lisp é uma das mais antigas e, desde sua criação, já se falava no uso de DSL.

Assim como DSL internas ou interfaces fluentes, existem vários outros modos de construir DSL. Nas próximas seções, vamos verificar estas variações e como elas se parecem.

2.7 VARIAÇÕES DE DSLS

Funções são as unidades essenciais para trabalhos em programação. Elas podem ter outros nomes, como métodos, procedimentos e sub-rotinas. A diferença entre uma interface comando consulta e uma DSL está em como estas funções são combinadas.

Nesta seção, vamos ver vários tipos de técnicas e/ou combinações de função para criar suas DSLs. Alguns exemplos serão encadeamento de métodos, sequência de funções, símbolos, parâmetros nomeados, tabela de símbolos, closures, entre outros. Também veremos quais dessas técnicas seriam mais interessantes para cada caso.

Encadeamento de métodos

Um padrão geral para interfaces fluentes é o **encadeamento de métodos** (FOWLER, 2013). Nele, o retorno das funções inicia a chamada para outra função. Ela é usada para facilitar a definição de vários valores de uma forma mais compacta.

Para uma DSL de atalhos hipotética, usando encadeamento de métodos, seria próximo do seguinte:

```
group()
    .shortcut().ctrl().plus('z')
    .shortcut().ctrl().shift().plus('z')
    .shortcut().ctrl().alt().del()
```

```
.shortcut().alt().f4();
```

Normalmente, em um código orientado a objetos, isso pareceria uma grande violação de princípios, e realmente é uma violação. Não se deve utilizar este tipo de encadeamento normalmente, pois causa fragilidade em relação às mudanças nas interfaces.

Mas na visão de uma DSL, este encadeamento é perfeitamente aplicável, pois dá a sensação de que se trata mesmo de uma linguagem de definição de atalhos. Vamos compreender melhor como desenvolver uma DSL utilizando encadeamento de métodos no próximo capítulo.

Sequência de funções

Outra forma de se encontrar esta fluência é aplicar a **sequência de funções** (FOWLER, 2013) da seguinte maneira. Em uma sequência de funções, temos métodos globais sendo chamados em uma sequência.

Ela é utilizada quando queremos chamar funções de forma mais independente. Um exemplo poderia ser o seguinte:

```
group();
shortcut();
    ctrl();
    plus('z');
shortcut();
    ctrl();
    shift();
    plus('z');
shortcut();
    ctrl();
    alt();
    del();
shortcut();
```

```
    alt();
    f4();
end();
```

A diferença aqui é que, no lugar de uma API que envia comandos, temos uma API que parece mais uma linguagem para descrever o que queremos. Existem algumas diferenças entre trabalhar com encadeamento de métodos ou sequência de funções.

A primeira coisa é que você precisa definir funções globais para poder acessar as funções. Talvez isso polua um pouco o espaço de nomes, mas em Java isso pode ser resolvido com importação estática.

Outra questão da sequência de funções é que você precisa definir em que contexto está trabalhando. No exemplo anterior, tínhamos o método `ctrl()` e o programa precisava saber em que atalho colocar essa tecla, então temos de manter o contexto em que estamos executando o código.

Posteriormente, veremos com mais profundidade funções globais. Entretanto, se você precisa utilizá-las agora, você pode pular para o capítulo *Sequência de funções e funções aninhadas*, para compreender melhor seu uso.

Uma forma de resolver o problema da globalidade é colocar as funções dentro de um contexto de algum objeto. Isso costuma resolver a maioria dos problemas, apesar do incomodo de ter de utilizar um contexto de objeto.

Funções aninhadas

Um outro tipo de DSL que pode ser utilizado é o de funções aninhadas. Em funções aninhadas, temos função aninhadas dentro

de outras funções.

Esta forma de construção faz com que as funções de nível mais baixo sejam passadas para os níveis mais altos, criando uma certa hierarquia de invocações. Esta hierarquia segue o padrão da hierarquia da linguagem.

Esta técnica também minimiza a necessidade de utilização de variáveis para se manter o contexto, já que as funções mais internas são resolvidas antes das de mais alto nível. A linguagem LISP, uma das mais antigas ainda sendo usada, combina muito com este tipo de estrutura. E o código que utiliza funções aninhadas se assemelha muito a códigos LISP.

Um exemplo de utilização poderia ser como o seguinte:

```
group(
    shortcut(
            ctrl(),
            plus('z')
    ),
    shortcut(
            ctrl(),
            shift(),
            plus('z')
    ),
    shortcut(
            ctrl(),
            alt(),
            del()
    ),
    shortcut(
            alt(),
            f4()
    )
);
```

Apesar de ter várias vantagens, este tipo de estrutura também possui algumas desvantagens. Uma delas é o problema da

globalidade de funções. Como temos muitas funções globais, isso pode acabar poluindo o espaço de nomes.

Em algumas linguagens, este problema é minimizado com algumas técnicas, como por exemplo, em Java, com importação de funções globais (utilizando `import static ...`). Outro problema existente neste tipo de construção é que ela cria alguns ruídos sintáticos, como as vírgulas para separar os métodos internos.

Existem outras características desta construção, e veremos mais sobre elas posteriormente. Mas se quiser aprender agora, é só pular para o capítulo *Sequência de funções e funções aninhadas*.

Combinações

Anteriormente, demonstramos algumas técnicas, mas elas podem também ser usadas de forma agrupada. Veja o exemplo a seguir:

```
group(
    shortcut().ctrl().plus('z'),
    shortcut().ctrl().shift().plus('z'),
    shortcut().ctrl().alt().del(),
    shortcut().alt().f4()
);
```

Neste exemplo, temos todos os tipos de estruturas apresentadas. O encadeamento de métodos para definir um atalho, a sequência de funções para definir um grupo de atalhos, e as funções aninhadas para colocar os atalhos dentro de um grupo.

Este tipo de estrutura híbrida tenta obter as melhores vantagens de cada uma das estruturas anteriores, mas acaba criando um pouco de ruído sintático. Existem várias decisões para

se escolher uma estrutura ou outra, ou ainda uma combinação. Apesar de não existir uma resposta totalmente certa, mostrarei mais à frente algumas indicações de qual utilizar para cada necessidade em sua linguagem.

Símbolos

Algumas linguagens permitem um outro tipo de construção usando símbolos. Java não tem suporte a símbolos, mas Scala possui suporte a este recurso. Por exemplo, em Scala, poderíamos ter o seguinte:

```
group(
    shortcut('ctrl -> 'z'),
    shortcut('ctrl -> 'shift -> 'z'),
    shortcut('ctrl -> 'alt -> 'del),
    shortcut('alt -> 'f4)
)
```

Neste código, temos uma linguagem para criação de grupos de atalhos. Posteriormente, veremos mais sobre símbolos, mas caso queira compreender melhor como desenvolver uma DSL utilizando-os, consulte o capítulo *Outras técnicas* (seção *Símbolos*).

Parâmetros nomeados

Outra estrutura que nem todas as linguagens suportam como o Scala é a chamada de funções com parâmetros nomeados. Parâmetros nomeados é uma funcionalidade em que, para uma função com vários parâmetros, podemos passá-los na chamada da função pelos seus nomes.

Nosso exemplo não suporta muito bem este tipo de configuração, já que ele é uma estrutura um pouco mais flexível.

Então, poderíamos ter uma DSL para descrever empresas da bolsa de valores, como por exemplo:

```
quote(
  exchange= "NASDAQ",
  name = "Google Inc",
  symbols = ('GOOG,'GOOGL),
  industry = "Technology"
)
```

Neste exemplo temos o método `quote` e os parâmetros `exchange`, `name`, `symbols` e `industry`. Estes podem são chamados pelos seus nomes.

Caso queira compreender melhor como desenvolver uma DSL usando parâmetros nomeados, consulte a seção *Parâmetros nomeados*.

Tabela de símbolos

O objetivo de uma tabela de símbolos é realizar o mapeamento entre o símbolo usado para se referir a um objeto e o objeto referente a este símbolo. Já utilizamos uma tabela de símbolos no capítulo anterior, no seguinte trecho de código:

```
...
.team("Language Team",
        player("Dennis Ritchie"),
        player("James Gosling"),
        player("John Backus"),
        reserve("Bjarne Stroustrup")
).events(
         start("14:00"),
         yellowCard("15:05").to("John Backus"),
         intervalStart("14:47"),
         intervalEnd("15:00"),
         substitution("15:25",out("Dennis Ritchie"),in("Bjarne st
roustrup")),
         end("15:27")
```

```
);
```

Como se pode perceber, estamos criando um jogador (`player`) de nome `"John Backus"` no seguinte trecho de código:

```
...
player("John Backus"),
...
```

E referenciando este mesmo jogador usando seu nome no seguinte trecho:

```
...
yellowCard("15:05").to("John Backus"),
...
```

A tabela de símbolos facilita sua vida na construção de DSLs mais flexíveis, pois possivelmente elas serão comuns na grande maioria das construções.

Closures

Closures é um recurso de linguagem bastante conhecido, mas não suportado em todas as linguagens de programação. A linguagem Java, por exemplo, só foi obter esta capacidade na sua versão 8.

Closures podem aparecer com diversos nomes, como blocos, lambdas e funções anônimas, mas funcionam essencialmente de uma forma semelhante. Um exemplo de DSL utilizando closures na versão do Java 8 seria a seguinte:

```
Builder.build(quote -> {
            quote.exchange = "NASDAQ";
            quote.name = "Google Inc";
            quote.symbols(s -> {
```

```
                s.add("GOOG");
                s.add("GOOGL");
            });
            quote.industry = "Technology";
        });
```

Este código é o mesmo apresentado anteriormente que descreve uma ação na bolsa de valores. Podemos também utilizar Scala para melhorar este exemplo, da seguinte maneira:

```
StockBuilder(stock => {
      stock exchange "NASDAQ"
      stock name "Google Inc"
      stock symbols("GOOG", "GOOGL")
      stock industry "Technology"
    })
```

Existem três características nas closures. A primeira é que eles suportam melhor aninhamento, pois é possível colocar funções mais complicadas dentro do fecho. A maioria das linguagens limita o que se pode colocar nos argumentos de funções.

Outra vantagem é a avaliação tardia. Ao contrário da função aninhada, em que os parâmetros são resolvidos antes de se chamar a função, nos fechos se tem avaliação tardia destes argumentos da função. Desta maneira, você tem controle sobre quando chamar a avaliação do fecho, e isso pode ser útil em alguns casos.

Por exemplo, digamos que você tenha o seguinte código que modela um farol:

```
Farol(
  verde = {
    println("Verde")
  },
  amarelo = {
    println("Amarelo")
  },
  vermelho = {
```

```
    println("Vermelho")
  }).on()
```

Neste código, queremos que seja executada cada uma das funções em um tempo determinado, e os fechos combinam muito bem.

A última vantagem é a criação de um escopo limitado para algumas variáveis de seu código. Isso pode ser interessante quando temos estruturas de configuração repetidas. Também elimina o uso de escopo de objetos e funções globais que podem ser eliminadas para os fechos.

Implicits

Uma funcionalidade interessante em algumas linguagens é a extensão de literais. Este recurso pode ter outros nomes em outras linguagens, como em C# com métodos de extensão, em C++ com Sobrescrita de operadores, ou em Ruby com adição de métodos nas classes. Ela consiste em adicionar métodos que antes não existiam a classes ou tipos.

Em Scala, esta funcionalidade é conhecida como conversões implícitas (*Implicit*). A linguagem Java não tem suporte a esta funcionalidade, então, você ainda não pode utilizar este recurso para sua DSL.

Implicits não costumam ser usados muito frequentemente, mas são muito bons quando precisamos usar tipos inteiros em algum processamento. Um exemplo de extensão de literais em Scala para uma DSL de medidas seria como o seguinte:

```
1.kg + 1.kg + 500.g + 220.hectogram
1.m + 1.cm + 10.decimeter
```

Recepção dinâmica

Em linguagens dinâmicas, como Ruby e JavaScript, existe um recurso conhecido como recepção dinâmica de métodos. O que este recurso faz é permitir chamar um método em um objeto e ele ser resolvido em tempo de execução em vez de em tempo de compilação. Por exemplo, poderíamos ter o código:

```
aCar.turnOn
```

Mesmo não tendo definido o método `turnOn`, este código compilaria sem problemas. Esta característica funciona muito bem em linguagens dinâmicas, mas algumas linguagens estáticas, como C# e Scala, suportam um mecanismo similar.

Em Scala, a partir da versão 2.10, temos um objeto que simula a recepção dinâmica. O uso mais comum de recepção dinâmica é mover informações de parâmetros de um método para o nome do método. Por exemplo, digamos que se tenha uma DSL que faz uma listagem de arquivos, poderíamos fazer o seguinte:

```
Ribbon(f => {
  f.`directory C:\\Windows\\Logs`
  f.extension_log
}).foreach(println)
```

Apesar de todo o poder da recepção dinâmica, sua utilização tem de ser criteriosa. É muito fácil acabar com uma complicação extra dentro da DSL e cometer enganos. Então, recomendo que utilize apenas pontualmente.

Gerador de construções

Algumas vezes, é necessário criar um objeto imutável, mas de maneira incremental. Quando temos estes casos, podemos utilizar

um gerador de construções.

Por exemplo, vamos dizer que precisamos construir um emissor de tickets de shows. Um ticket tem vários dados, como número de identificação, número de confirmação, data do evento, localização, valor e descrição. Podemos modelar este objeto da seguinte maneira:

```
case class Ticket(
    id: String,
    confirmation: String,
    date: Date,
    place: String,
    amount: Float,
    description: String)
```

E criar um gerador como o seguinte:

```
class TicketBuilder {
  var id: Option[String] = None
  var confirmation: Option[String] = None
  var date: Option[Date] = None
  var place: Option[String] = None
  var amount: Option[Float] = None
  var description: Option[String] = None

  def withId(id: String) = {
    this.id = Some(id)
    this
  }

  def withConfirmation(confirmation: String) = {
    this.confirmation = Some(confirmation)
    this
  }

  def withDate(date: Date) = {
    this.date = Some(date)
    this
  }

  def withPlace(place: String) = {
```

```
        this.place = Some(place)
        this
      }

      def withAmount(amount: Float) = {
        this.amount = Some(amount)
        this
      }

      def withDescription(description: String) = {
        this.description = Some(description)
        this
      }

      def build(): Ticket = new Ticket(
                                          id.get, confirmation.get,
 date.get,
                                          place.get, amount.get, de
scription.get)
    }
```

Na utilização, faríamos da seguinte maneira:

```
    class Main {
      def main(args: Array[String]) {

        val builder = new TicketBuilder()
        builder.withId("1")
          .withConfirmation("xptoconfirmation")
          .withDate(new Date())
          .withPlace("Credicard Hall")
        .withAmount(65.00f)
        .withDescription("Offspring Show")

        val ticket: Ticket = builder.build()
      }
    }
```

Podemos ter vários construtores aninhados para construir um objeto final. Podemos utilizar sempre um gerador de construções quando queremos construir parcialmente um objeto que precisa de todos os dados para construção.

2.8 QUAL TÉCNICA UTILIZAR PARA CONSTRUIR SUA DSL

Com todas estas possibilidades, pode ser que você fique um pouco confuso com quais elementos usar para construir sua DSL. A maioria das DSL no mundo real usa mais de um conjunto de técnicas, pois alguns tipos de expressões combinam melhor com alguns elementos dentro da DSL. Vou demonstrar alguns exemplos para indicar quais elementos podem ser mais apropriados.

Lista obrigatória

Quando existir dentro da DSL elementos que obrigatoriamente precisam vir de forma fixa, as funções aninhadas são uma estrutura que casam muito bem. Por exemplo, vamos dizer estamos construindo uma DSL simples para enviar um lembrete diário. Poderíamos começar com a linguagem natural descrevendo o seguinte:

```
diariamente enviar email
    para leonardootto@gmail.com
    as 18:00
    contendo "Desligar a cafeteira do escritório"
```

Lembre-se de que estamos trabalhando com uma linguagem de computador, então podemos sintetizar a linguagem e remover algumas coisas supérfluas. Simplificando a linguagem natural, teríamos algo assim:

```
 enviar email
    para leonardootto@gmail.com
    as 18:00
    contendo "Desligar a cafeteira do escritório"
```

Vamos assumir que algumas palavras dentro da nossa linguagem sempre devem aparecer. Isso é uma escolha de projeto e flexibilização, mas vamos escolher as palavras que denotam o caminho do e-mail, que horas ele deve ser enviado e qual o seu conteúdo. Partindo deste princípio e usando funções aninhadas, teríamos algo assim:

```
enviar(
    email("leonardootto@gmail.com"),
    as("18:00"),
    contendo("Desligar a cafeteira do escritório")
)
```

Funções aninhadas é uma técnica muito interessante e na seção *Funções aninhadas*, veremos mais detalhes sobre esta estrutura e como desenvolver utilizando estes conceitos.

Lista opcional

Quando dentro da DSL existir elementos que são opcionais, uma estrutura que combina muito bem são o encadeamento de métodos e também os parâmetros nomeados.

Utilizando a mesma ideia do exemplo anterior (o envio de e-mails), temos alguns campos que são requeridos. O e-mail de quem está enviando e o destinatário são requeridos, enquanto outros campos são opcionais, como a cópia (cc), a cópia oculta (bcc) e até o conteúdo.

Uma possível DSL seria a seguinte:

```
emailto("billgames@hotmail.com")
       .from("torvalds@klaava.Helsinki.Fi")
       .cc("stevejobs@apple.com")
       .bcc("stallman@fsf.org")
       .content("Linux Rocks!!!");
```

O encadeamento de métodos combina bem com elementos opcionais, pois não precisamos tratar de todas as combinações possíveis, como seria o caso da utilização de funções aninhadas. Posteriormente no capítulo *Encadeamento de métodos e Composite*, veremos mais detalhes sobre o uso deste método.

Múltiplos filhos homogêneos

Quando temos de usar uma construção que possui um pai para muitos filhos similares, algo que casa bem é utilizar uma lista de literais.

Por exemplo, uma turma tem vários alunos. Poderíamos descrever usando uma DSL como a seguinte:

```
turma.contemOsAlunos("Harry Potter","Ronald Weasley","Hermione Granger")
```

Você já deve ter usado esta técnica em alguns de seus programas mesmo sem perceber, e ela é autoexplicativa.

Múltiplos filhos heterogêneos

Quando se tem muitos filhos heterogêneos, a melhor forma de tratar é voltar para encadeamento de métodos. Por exemplo, para uma DSL para tratar itens da feira, de forma simplificada, poderia ser o seguinte:

```
feira.fruta("maça")
     .queijo("prato")
     .presunto("italiano")
     .hortalicas("cebola")
```

A implementação de DSLs internas possui a vantagem de o desenvolvedor não precisar entender sobre gramáticas e análise

sintática de linguagens, e também de não precisar de ferramentas externas. Mas apesar de ele não precisar conhecer estes assuntos para criar sua DSL interna, este conhecimento não é descartável.

Você pode querer ter uma pequena DSL focada, e eu recomendo isso, mas pode ser que com o tempo algumas partes dela se tornem um pouco mais complexas. O conhecimento sobre gramática e análise sintática vai auxiliar a organizar esta complexidade crescente, e é isso que vamos aprender a seguir.

Até agora, analisamos várias técnicas para implementação, mas ainda não fizemos a implementação mais a fundo destas técnicas. Antes de entrarmos nos próximos capítulos nestes assuntos, vamos aprender uma forma de descrever linguagens e vermos como isso facilita a implementação posterior.

2.9 BNF

A BNF (ou *Backus-Naur Form*) é uma sintaxe para descrever linguagens. Ela foi criada para descrever a linguagem ALGOL e, atualmente, é amplamente usada para comunicar um grande conjunto de sintaxes. Sua forma padrão é a seguinte:

```
<símbolo> ::= <expressão>
```

Um exemplo de uso da BNF para descrição do código de CPF seria algo como:

```
<cpf> ::= <bloco3>.<bloco3>.<bloco3>-<bloco2>
<bloco3> ::= <digito><digito><digito>
<bloco2> ::= <digito><digito>
<digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Esta BNF possui 4 regras de produção. A primeira diz que um

CPF é constituído de 3 blocos de 3 números separados por . (ponto), seguido por um - (hífen) e, posteriormente, um bloco de dois números.

A segunda regra diz que um bloco de 3 números são 3 números um após o outro. A regra seguinte é semelhante à regra anterior, mas apenas para dois números. E a última regra de produção nos diz que um dígito pode ser qualquer um dos dígitos de 0 até 9.

A BNF pode aparecer em diversas sintaxes e formas diferentes, então se atenha à ideia por trás da linguagem no lugar da sua sintaxe.

Utilizando a BNF para construir uma DSL

Para ilustrar como utilizar a BNF para ajudar na construção de uma DSL, imagine o seguinte cenário. Você foi contratado para desenvolver uma DSL para facilitar o acesso a um banco de dados usado exclusivamente por esta empresa. Vamos chamar este banco de dados de YesSql.

Tudo que o YesSql faz é selecionar colunas de tabelas e filtrar de alguma forma simples. Vamos começar com uma descrição simples da linguagem:

```
    <select-clause> ::= SELECT <result-column> FROM <table-clause
>
    <result-column> ::=  * | <expr> | <result-column> , <result-c
olumn>
    <expr>          ::= <column-name> | <table-name>.<column-name
>
    <table-clause>  ::= <table-name> | <table-name> , <table-clau
se>
```

Como vimos no capítulo anterior, existem algumas

construções que casam melhor com algumas técnicas. Vamos começar pela regra de produção `<table-clause>`, e assumir que a regra `<table-name>` pode ser uma string simples qualquer.

```
    <table-clause>  ::= <table-name> | <table-name> , <table-clau
se>
```

Nesta regra, temos dois elementos: o nome da tabela ou vários nomes separados por vírgula, ou seja, temos vários filhos homogêneos. Para este caso, a melhor construção seria a lista de literais. Um exemplo que ilustra isso seria como o seguinte:

```
TableClause tb = table("pessoa","cidade","pais")
```

Continuando, vamos analisar a regra de produção `<expr>`.

```
<expr>  ::= <column-name> | <table-name>.<column-name>
```

Nestas regras, temos elementos opcionais. O usuário da nossa biblioteca pode utilizar apenas o nome da coluna ou, para evitar ambiguidades, o nome da tabela e a coluna. Um código para esta expressão poderia ser o seguinte:

```
Expression e1 = table("pessoa").column("nome")
//ou apenas a coluna
Expression e2 = column("nome")
```

A próxima regra de produção que temos é a `<result-column>`:

```
    <result-column> ::=  * | <expr> | <result-column> , <result-c
olumn>
```

Esta regra é um pouco mais complexa, e pode utilizar 3 tipos diferentes de elementos. Quando temos vários tipos heterogêneos de elementos, a melhor técnica para se utilizar é a de encadeamento de métodos. No nosso exemplo, seria algo como o

seguinte:

```
     ResultColumn rc = new ResultColumn(all(),column("nome"),colu
mn("cpf"))
```

Agora analisamos a primeira regra geral, que é a seguinte:

```
    <select-clause> ::= SELECT <result-column> FROM <table-clause
>
```

Temos dois elementos requeridos e dois que produzimos pelas regras anteriores. Uma possível DSL seria algo como:

```
    select(all(),column("nome"),column("cpf")).from(table("pessoa
"))
```

Preferi utilizar encadeamento de métodos para facilitar a clareza na leitura da DSL, e por ela seguir mais a estrutura de como o SQL é lido. A BNF mostra muita da sua utilidade para descrever de forma mais cara como implementar uma DSL, seja ela interna ou externa.

Dominando seus conceitos, além de possibilitar a construção de DSL internas mais complexas, ajuda no entendimento do funcionamento inicial das DSL externas, caso o leitor se interesse por este tópico em especial.

2.10 E AGORA?

Nos próximos capítulos, descreveremos cada uma das técnicas em mais detalhes, com os exemplos apresentados anteriormente. Se o leitor tiver interesse em apenas uma implementação em particular, pode partir direto para esta parte. Caso o leitor queira aprender mais, pode seguir capítulo por capítulo, para dominar melhor cada uma das técnicas de construção.

CAPÍTULO 3

ENCADEAMENTO DE MÉTODOS E COMPOSITE

O encadeamento de método é visto por muitos como sinônimo de DSL. Apesar de ser uma técnica muito boa, ela não é a única dentro de todo o conjunto de técnicas de DSL. Depois de aprendê-la, vamos aprender um padrão estrutural chamado Composite, e como ele auxilia na construção de não apenas no encadeamento de método, mas de várias outras técnicas de construção.

Utilizamos no capítulo anterior um exemplo de uma DSL usando o encadeamento de métodos, e agora vamos construir passo a passo a mesma DSL usando duas formas diferentes de construção. Isso dará uma visão bem ampla sobre o uso do encadeamento de métodos e do padrão Composite.

3.1 ENCADEAMENTO DE MÉTODOS

Para compreender como construir uma DSL utilizando encadeamento de métodos, usaremos a mesma DSL apresentada na seção *Variações de DSLs* do capítulo anterior. Ela se apresentava da seguinte forma:

```
group()
    .shortcut().ctrl().plus('z')
```

```
.shortcut().ctrl().shift().plus('z')
.shortcut().ctrl().alt().del()
.shortcut().alt().f4();
```

Esta DSL é utilizada para descrever um domínio que trata de atalhos e grupos de atalhos. Primeiramente, como no capítulo *Batendo uma bola com sua DSL*, vamos criar nosso modelo.

Um atalho é constituído de uma ou mais teclas conjuntas. Não defino como duas teclas, pois em algumas aplicações pode ser que apenas uma tecla ative algum atalho específico. Como, por exemplo, no editor Vim, que tem atalhos definidos com apenas uma tecla.

Ele é em um dos editores de textos mais famosos do Linux, criado em 1991, e está instalado em quase todas as máquinas que rodam Linux. Neste editor, para modificar seu comportamento de entrada, digitamos a tecla `i`, um atalho de apenas uma tecla. Logo, para flexibilizar, vamos assumir que um atalho pode ser de apenas uma tecla.

Para fazer essa modelagem, precisamos construir um domínio. Mas como fazer isso?

Construindo o domínio

Para construir o domínio, precisamos de um conjunto de classes para representar nosso problema, classes que representem atalhos e uma classe que contenha o conjunto deles. Uma modelagem poderia ser como a seguinte:

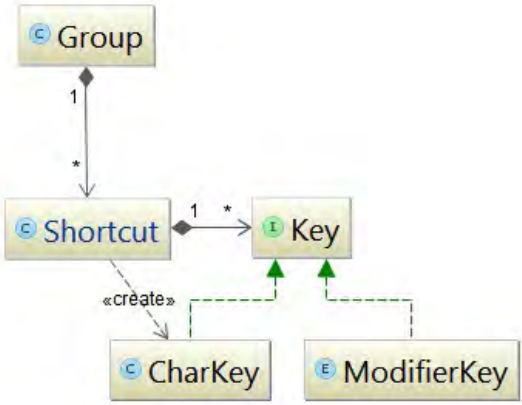

Figura 3.1: Uma modelagem para a DSL de atalhos

Para iniciar, vamos começar pela linguagem Java, que é aquela com que o leitor provavelmente está mais habituado. Posteriormente, usaremos Scala em um exemplo que une várias técnicas.

Então, vamos começar criando uma classe `Shortcut` para representar os atalhos, e uma interface `Key` para representar as teclas, como o seguinte:

```
public class Shortcut {...}
public interface Key{}
```

Uma tecla de atalho pode ser de dois tipos: ou ela é uma tecla modificadora, ou é uma de caractere. Para representar uma tecla de caractere, criamos a classe `CharKey` e, para as teclas modificadoras, a classe `ModifierKey` , como a seguir:

```
public class CharKey implements Key {...}
public enum ModifierKey implements Key {...}
```

A `CharKey` poderia ser uma classe simples que contém um caractere como o seguinte:

```
public class CharKey implements Key {
    public final char key;

    public CharKey(char key){
        this.key = key;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false
;

        CharKey charKey = (CharKey) o;
        if (key != charKey.key) return false;

        return true;
    }

    @Override
    public int hashCode() {
        return (int) key;
    }
}
```

Como se pode observar, implementei os métodos `equals` e `hashCode`, pois eles vão ser úteis quando quisermos comparar se uma tecla é semelhante à outra. A classe que representa as teclas modificadoras pode ser algo como:

```
public enum ModifierKey implements Key {
    Alt, Ctrl, Del, Shift,
    F1, F2, F3, F4, F5, F6,
    F7, F8, F9, F10, F11, F12
}
```

Na classe `ModifierKey`, não precisamos definir os métodos `equals` e `hashCode`, pois eles já estão implícitos dentro do

`enum`. Agora podemos voltar à classe `Shortcut`, e adicionar a característica de um atalho ter uma ou mais teclas, e implementar também os métodos padrões.

```
public class Shortcut {
    private Set<Key> keys = new HashSet<Key>();

    public Shortcut addKey(Key key) {
        boolean add = keys.add(key);
        if(!add){
            throw new IllegalStateException("Duplicate key:"+key)
;
        }
        return this;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false
;
        Shortcut shortcut = (Shortcut) o;
        if (keys != null
               ? !keys.equals(shortcut.keys)
               : shortcut.keys != null) return false;
        return true;
    }
    @Override
    public int hashCode() {
        return keys != null ? keys.hashCode() : 0;
    }
}
```

Temos de definir também uma classe para o grupo de atalhos. Para isso, vamos criar a classe `Group`, que será nosso agrupador de atalhos. A implementação seria a seguinte:

```
public class Group {
    private ArrayList<Shortcut> list = new ArrayList<>();

    public Group() {
    }
```

```
    public boolean addShortcut(Shortcut shortcut) {
        return this.list.add(shortcut);
    }
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false
;

        Group group = (Group) o;
        if (list != null
                ? !list.equals(group.list)
                : group.list != null) return false;

        return true;
    }

    @Override
    public int hashCode() {
        return list != null ? list.hashCode() : 0;
    }
}
```

Agora que já definimos as teclas, os atalhos e os grupos de atalhos, podemos usar nossa API de atalhos. Como discutido anteriormente, esta API é do padrão do tipo comando consulta, e ainda não é nossa finalidade. Mas vamos utilizá-la para quesito de comparação, e porque vamos utilizar esta API para construção e testes da DSL posteriormente.

No código a seguir, podemos ver um exemplo de utilização desta API.

```
public class Main {
    public static void main(String[] args) {
        Group g = new Group();
        Shortcut s = new Shortcut();
        s.addKey(Ctrl);
        s.addKey(new CharKey('z'));
        g.addShortcut(s);
```

```
        s = new Shortcut();
        s.addKey(Ctrl);
        s.addKey(Shift);
        s.addKey(new CharKey('z'));
        g.addShortcut(s);

        s = new Shortcut();
        s.addKey(Ctrl);
        s.addKey(Alt);
        s.addKey(Del);
        g.addShortcut(s);
    }
}
```

A API é bem clara: ela cria grupos de atalhos e adiciona cada um deles dentro destes grupos. Cada atalho é composto de um conjunto de teclas. Apesar de ser bem clara, ela é de comando consulta, e não é nosso objetivo. Então, como utilizaríamos o encadeamento de métodos em nosso exemplo para criar a nossa DSL?

Expandindo o domínio para uma DSL

Para converter nosso exemplo anterior para uma DSL, usando encadeamento de métodos, teríamos de encadear cada um dos métodos em um fluxo. Por exemplo, tendo o seguinte código apresentado anteriormente:

```
Shortcut s = new Shortcut();
s.addKey(Ctrl);
s.addKey(new CharKey('z'));
```

Faríamos o seguinte encadeamento:

```
new Shortcut().addKey(Ctrl).addKey(new CharKey('z'));
```

Para usarmos este encadeamento, precisamos modificar a

classe `Shortcut` para retornar sua referência a cada chamada de método de definição de propriedades. Este tipo de construção quebra as separações entre métodos de APIs comando consulta, pois nelas só podemos modificar o objeto, ou consultar o estado de um método. Ou seja, métodos de modificação não possuem retorno.

Este princípio é quebrado para construção do encadeamento de métodos. Porém, como esta é uma API um pouco diferente, isso é permitido. A implementação disso seria algo como:

```
public class Shortcut {
    private Set<Key> keys = new HashSet<Key>();

    public Shortcut addKey(Key key) {
        boolean add = keys.add(key);
        if(!add){
            throw new IllegalStateException("Duplicate key:"+key)
;
        }
        return this;
    }
    ...
}
```

A modificação anterior criou um encadeamento de métodos, mas ele ainda é um pouco estranho para o leitor do código. Sua utilização direta seria algo como o seguinte código:

```
    Group g = new Group();
    Shortcut s = new Shortcut();
    s.addKey(Ctrl).addKey(new CharKey('z'))
    g.addShortcut(s);
```

Esse uso não modifica muita coisa do código original. Podemos então realizar uma limpeza nas chamadas, modificando para que elas fiquem mais limpas. Normalmente, quando vemos algum atalho descrito graficamente, ele vem assim: `Ctrl + Alt +`

z .

Logo, podemos modificar um pouco nossa DSL para fazê-la chegar mais próxima do exemplo a seguir:

```
new Shortcut().ctrl().plus('z')
```

Para realizar esta modificação, considerei a criação de métodos especiais para teclas modificadoras, e um método exclusivo para as teclas de caracteres, da seguinte maneira:

```
public class Shortcut {
    private Set<Key> keys = new HashSet<Key>();

    public Shortcut addKey(Key key) {
        boolean add = keys.add(key);
        if(!add){
            throw new IllegalStateException("Duplicate key:"+key)
;
        }
        return this;
    }

    public Shortcut ctrl() {
        return addKey(Ctrl);
    }
    public Shortcut plus(char key) {
        return addKey(new CharKey(key));
    }
    ...
}
```

Adicionando esta nova construção à API de grupos anteriores, temos o seguinte:

```
new Group().addShortcut(new Shortcut().ctrl().plus('z'));
```

Como fizemos anteriormente, podemos simplificar mais ainda os métodos, da seguinte maneira:

```
group().shortcut().ctrl().plus('z')
```

Para realizar esta modificação, primeiramente é necessário criar o método `group()` dentro da classe `Group`, e também os métodos do acesso.

```
public class Group {
...
    public static Group group() {
        return new Group();
    }
    public Shortcut lastShortcut() {
        if (list.size() == 0) {
            return null;
        }
        return list.get(list.size() - 1);
    }
    public Group shortcut() {
        Shortcut shortcut = new Shortcut();
        addShortcut(shortcut);
        return this;
    }
    public Group ctrl() {
        lastShortcut().ctrl();
        return this;
    }

    public Group plus(char key) {
        lastShortcut().plus(key);
        return this;
    }
...
}
```

Também é necessário adicionar a importação estática quando for usar a DSL, para simplificar as chamadas. Estas modificações são demonstradas a seguir:

```
...
import static dsl.cap03.example1.direct.model.Group.*;
...
public class Main {
    public static void main(String[] args) {...}
    private static Group getGroupFromCommandQuery() {...}
```

```
    private static Group getGroupFromDsl() {
        return group()
                .shortcut().ctrl().plus('z')
    }
}
```

Como podemos ver no código, misturamos nosso modelo que possui uma API comando consulta com os métodos da DSL. Isso pode causar um pouco de confusão, principalmente para DSL maiores e mais complexas.

Mas será que existe alguma outra forma de desenvolver o encadeamento de métodos de forma mais organizada? Sim, existe! E vamos aprender na próxima seção sobre ele, o padrão Composite.

3.2 COMPOSITE OR DON'T COMPOSITE?

Em um programa de manipulação de arquivos, podemos ter toda uma estrutura de representação de arquivos e diretórios. Esta estrutura pode ser representada na forma de uma árvore, esta que contém nós que são os diretórios e folhas que são os arquivos. Ambos, diretórios e arquivos, podem ser copiados, movidos, renomeados, entre outras operações.

Tratar os objetos desta árvore como tipos de elementos similares facilita na reutilização de algoritmos. Pode ser que mais pessoas tenham tentado resolver este mesmo problema de design de software de forma semelhante criando padrões de design. Existem muitos destes padrões já catalogados.

Os desenvolvedores usam estes padrões sempre que encontram problemas cuja solução parece se encaixar bem com algum deles.

Composite é um destes padrões, e é usado quando precisamos tratar um conjunto de objetos de uma forma uniforme, como nosso exemplo anterior do programa de manipulação de arquivos.

Em DSLs, normalmente é usado este padrão para não ter de colocar os métodos da DSL dentro do modelo semântico. Fazer isso acaba misturando interesses distintos dentro dos objetos, causando alguma confusão. Além disso, misturar mais de um tipo de acesso dentro dos objetos, DSL e comando consulta também não é uma boa prática.

Logo, recomendo utilizar o padrão Composite sempre que possível. Para saber mais sobre padrões de desenvolvimento, recomendo o clássico *Padrões de Projeto - Soluções de Software Orientado a Objetos* de Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994).

Na seção anterior, desenvolvemos uma DSL usando encadeamento de métodos, sem usar o padrão Composite. Agora vamos aprender como organizar mais nosso código utilizando este padrão e centralizando o código referente a DSL.

Exemplo com Composite

A grande diferença entre o exemplo anterior sem e com Composite é que precisamos de uma classe exclusiva para realizar o tratamento da DSL e dos encadeamentos. Logo, vamos criar uma classe chamada `DSL` para realizar estas funções, como segue:

```
public class DSL {

    private final Group group;

    private DSL() {
```

```
        this.group = new Group();
    }

    public static DSL group() {
        return new DSL();
    }

    public DSL shortcut() {
        Shortcut shortcut = new Shortcut();
        addShortcut(shortcut);
        return this;
    }

    public boolean addShortcut(Shortcut shortcut) {
        return group.addShortcut(shortcut);
    }

    public DSL ctrl() {
        lastShortcut().ctrl();
        return this;
    }

    public DSL plus(char key) {
        lastShortcut().plus(key);
        return this;
    }
    ...
}
```

Esta classe `DSL` contém um grupo de atalhos instanciados diretamente em sua criação. Temos o método `group()` que faz a criação da nossa DSL. Temos também os métodos de criação dos atalhos com o método `shortcut`, `ctrl` etc.

Agora realizamos o acesso diretamente por esta classe, separando o modelo semântico e a DSL por completo.

```
group()
.shortcut().ctrl().plus('z')
.shortcut().ctrl().shift().plus('z')
.shortcut().ctrl().alt().del()
.shortcut().alt().f4()
```

```
    .build();
```

A diferença é que existe uma separação mais clara sobre o que é a DSL e o que é a API comando consulta. Sem Composite, quem faz todo controle são as classes do domínio; já no modelo usando Composite, a DSL utiliza o modelo e este não conhece sobre a DSL. A separação do modelo semântico e da DSL é muito importante, e é sempre recomendada como uma boa prática para separar as responsabilidades e não poluir o modelo do domínio.

3.3 INTERFACES PROGRESSIVAS

Algumas vezes queremos restringir alguns encadeamentos para que a leitura da DSL fique mais clara e fácil. Nestes casos, podemos usar interfaces de retorno entre os métodos. Isso permite não só que a restrição exista, mas também, nas IDEs modernas, um autocompletar que facilita o desenvolvimento.

Não vamos utilizar o exemplo anterior, pois ele complicaria um pouco as coisas por ter muitas opções. Então, demonstrarei com um outro exemplo como usar esta técnica.

Até agora aprendemos um conjunto de técnicas diferentes, que tal aplicá-las em um exemplo um pouco mais complexo? Que tal um jogo simples? Vamos ver isso na próxima seção, como aplicar algumas destas técnicas que aprendemos em um exemplo único.

3.4 GAME OF LIFE

Até agora vimos um conjunto de técnicas para se agrupar ao conjunto de técnicas do construtor de DSLs. Para consolidar este conhecimento, agora vamos adicionar um pouco mais de

complexidade com um exemplo real. Escolhi Scala para este exemplo para dar um pouco mais de flexibilidade.

Game of life é uma simulação de vida artificial muito conhecido no campo da matemática e computação. De forma simplificada, para quem não conhece, é uma simulação simples de pequenos serem vivos que vivem em uma matrix quadrada.

Esta simulação é usada para ensinar algumas teorias da computação e também programação. Ela possui regras muito simples, como demonstrado a seguir:

1. Qualquer célula viva com menos de dois vizinhos vivos morre de solidão.
2. Qualquer célula viva com mais de três vizinhos vivos morre de superpopulação.
3. Qualquer célula morta com exatamente três vizinhos vivos se torna uma célula viva.
4. Qualquer célula viva com dois ou três vizinhos vivos continua no mesmo estado para a próxima geração.

Para iniciar, vamos modelar nosso problema. Temos dois tipos de célula, vivas ou mortas. Então, modelamos da seguinte maneira:

```
trait CellType

object DeadCell extends CellType {
  override def toString: String = "D"
}

object LiveCell extends CellType {
  override def toString: String = "L"
}
```

Temos também um conjunto de quatro regras para definir o

que ocorre em cada iteração. Cada regra define um estado inicial e um estado final, e tem um conjunto de proposições. Por exemplo, para primeira regra:

```
Qualquer célula viva com menos de dois vizinhos vivos morre de so
lidão.
```

Temos o estado inicial **célula viva**, o estado final **morre** e também uma proposição, **com menos de dois vizinhos**. Os estados, celular viva ou morta, já foram modelados, então modelaremos agora a proposição.

Uma proposição no nosso modelo é composta de uma sentença (maior, menor, igual etc.) e alguns operadores lógicos (**ou**, **e**) Assim, vamos definir estas sentenças de uma forma única e modelá-las. Escolhi utilizar o nome `Predicate` para esta abstração, pois predicado são coisas relacionadas a um sujeito.

Modelei da seguinte forma:

```
  trait Predicate {
    def apply(types: Seq[CellType]): Boolean
  }

  case class Or(first: Predicate, second: Predicate) extends Pred
icate {
    def apply(types: Seq[CellType]): Boolean = {
      first.apply(types) || second.apply(types)
    }
  }

  case class Equal(number: Int, cellType: CellType) extends Predi
cate {
    def apply(types: Seq[CellType]): Boolean = {
      number == types.count(_ == cellType)
    }
  }

  case class Greater(number: Int, cellType: CellType) extends Pre
```

```
dicate {
    def apply(types: Seq[CellType]): Boolean = {
      number < types.count(_ == cellType)
    }
  }

  case class Less(number: Int, cellType: CellType) extends Predic
ate {
    def apply(types: Seq[CellType]): Boolean = {
      number > types.count(_ == cellType)
    }
  }
```

Agora, para definir as regras utilizando nosso modelo de domínio, faríamos:

```
    val rules = List(
      Rule(LiveCell, List(Less(2, LiveCell)), DeadCell)
      Rule(LiveCell, List(Greater(3, LiveCell)), DeadCell),
      Rule(DeadCell, List(Equal(3, LiveCell)), LiveCell),
      Rule(LiveCell,
           List(Or(Equal(2, LiveCell),
           Equal(3, LiveCell))), LiveCell)
    )
```

Até agora, fizemos apenas uma modelagem do domínio e sua implementação. Mas não utilizamos nenhuma DSL para expressar as regras que apresentamos. Como será que seria uma implementação deste modelo em uma DSL? Vamos ver na próxima seção.

DSL do GameOfLife

Agora que já temos nosso modelo de domínio definido, usaremos uma DSL para preenchê-lo. Isso vai deixar nossa intenção muito mais explícita. Também vamos utilizar interfaces progressivas demonstrando como usar esta técnica.

Primeiramente, convertemos diretamente a linguagem natural

para encadeamento de métodos, da seguinte maneira:

```
    //solidão
    qualquer().celula().viva().com().menos().de(2).vizinhos().mor
re();
    //superpopulação
    qualquer().celula().viva().com().mais().de(3).vizinhos().morr
e();
    qualquer().celula().morta().com().exatamente(3).vizinhos().vi
ve();
    qualquer().celula().viva().com(2).ou().com(3).vizinhos().vivo
s().vive();
```

Para modelar isso, vamos criar um objeto para ser nosso construtor da DSL, vou chamá-lo de `Dsl`. Criaremos também o seu `Companion object`.

```
class Dsl{ }
object Dsl{ }
```

O método de entrada para nossa construção de regras é o `qualquer`, então vamos implementá-lo em nosso `object Dsl`, como a seguir:

```
class Dsl{}
object Dsl{
   def qualquer: Unit = new Dsl()
}
```

Nossa primeira interface é a que recebemos do método `qualquer`, então vamos criar uma interface para isso. Mas em Scala, não temos interfaces, logo, usaremos o similar e mais poderoso `Trait`.

```
trait Qualquer{ }
class Dsl extends Qualquer{
    def qualquer:Qualquer = this
}
```

Depois do método `qualquer`, temos o método `celula`,

então criamos mais um `Trait` para representá-lo, como a seguir:

```
trait Celula{}
trait Qualquer{
     def celula:Celula
}
class Dsl extends Qualquer with Celula{
    def qualquer:Qualquer = this
    override def celula: Celula = this
}
```

Agora já podemos fazer o encadeamento inicial:

```
Dsl.qualquer().celula()
```

Agora temos duas possibilidades para depois do método `Celula` : referir-se às células vivas ou às mortas, sem falar nas possibilidades posteriores. Para simplificar, teremos o seguinte conjunto de classes da DSL:

```
trait Ou{
  def com(n:Int):ComNumero
}
trait ComNumero{
  def ou():Ou
  def com
  def vizinhos:Vizinhos
}
trait Vizinhos{
  def morre
  def vive
}
trait De{
  def vizinhos:Vizinhos
}
trait Mais{
  def de(n:Int):De
}
trait Menos{
  def de(n:Int):De
}
trait Com{
```

```
    def mais:Mais
    def menos:Menos
    def exatamente(n:Int):De
  }
  trait CelulaViva{
    def com(n:Int):ComNumero
    def com:Com
  }
  trait CelulaMorta{
    def com:Com
  }
  trait Celula{
    def viva:CelulaViva
    def morta:CelulaMorta
  }
  trait Qualquer{
    def celula:Celula
  }
```

E nossa classe vai construir as regras a partir do conjunto de chamadas o objeto `Dsl` , como a seguir:

```
  class Dsl extends Qualquer
      with Celula with CelulaViva with CelulaMorta
      with Com with Mais with Menos with De with Vizinhos with
ComNumero with Ou {
  {
    override def celula: Celula = {this}

    override def viva: CelulaViva = {this}

    override def morta: CelulaMorta = {this}

    override def com(n: Int): ComNumero = {this}

    override def com: Com = {this}

    override def menos: Menos = {this}

    override def exatamente(n: Int): De = {this}

    override def mais: Mais = {this}

    override def morre: Dsl = {this}
```

```
    override def vive: Dsl = {this}

    override def ou(): Ou = {this}

    override def vizinhos: Dsl = {this}

    override def de(n: Int): De = {this}
}
```

No final, faremos as chamadas da seguinte maneira:

```
qualquer.celula.viva.com.menos.de(2).vizinhos.morre
qualquer.celula.viva.com.mais.de(3).vizinhos.morre
qualquer.celula.morta.com.exatamente(3).vizinhos.vive
qualquer.celula.viva.com(2).ou().com(3).vizinhos.vive
```

Por enquanto, não fizemos nenhuma implementação além da DSL. Mas você se interessou pelo jogo, pode consultar o código completo da página com os exemplos do livro.

3.5 E AGORA?

Neste capítulo, aprendemos a técnica de encadeamento de métodos e o padrão Composite para o desenvolvimento. Apesar de esta técnica ser muito conhecida, ela não é a única.

No próximo capítulo, continuaremos expandindo nossas opções e aprenderemos uma que lembra muito do estilo de desenvolvimento da linguagem LISP.

Capítulo 4

SEQUÊNCIA DE FUNÇÕES E FUNÇÕES ANINHADAS

Neste capítulo, vamos aprender mais sobre a sequência de funções, no que ela consiste e suas características. Também demonstraremos um exemplo inicial de utilização e um exemplo de uso em um framework real.

Além disso, vamos aprender sobre funções aninhadas, suas características e como implementá-las, junto de exemplos para melhorar a compreensão da utilização desta técnica.

4.1 SEQUÊNCIA DE FUNÇÕES

A sequência de funções é uma das mais simples técnicas para construir DSLs. Ela consiste em criar chamadas de acesso global para descrever a DSL. Como as chamadas são globais, não existe uma relação direta entre cada uma delas, e é necessário criar um mecanismo para associá-las.

Sua simplicidade extrema e a necessidade de associação fazem com que sua utilidade não seja tão grande. Mas apesar disso, é importante seu aprendizado para podermos aplicar estas técnicas em casos particulares que veremos posteriormente.

Uma característica das linguagens Java e Scala que facilita a importação deste tipo de funções são as importações estáticas. Por exemplo, para usarmos o seguinte código:

```
double r = Math.cos(Math.PI * 2);
```

É necessário realizar a importação da classe `Math` da seguinte forma:

```
import java.lang.Math;
```

Mas poderíamos fazer a importação estática assim:

```
import static java.lang.Math.*;
```

E simplificar a chamada desta maneira:

```
double r = cos(PI * 2);
```

Se a linguagem não permite importações estáticas, é necessário utilizar métodos de classe para realizar as chamadas, e isso adiciona ruído à sua DSL. Como vimos anteriormente, ruídos são os códigos extras que você tem de adicionar para suportar uma DSL.

O grande problema da sequência de funções é que ela o obriga a utilizar dados estáticos. O problema com dados estáticos é que eles podem causar problemas em ambientes com várias threads, já que a sequência de execução não é conhecida.

Utilizando sequência de funções

Para entender melhor, vamos usar o exemplo que usamos no capítulo *Adentrando nas DSLs*, que era o seguinte:

```
group();
 shortcut();
```

```
        ctrl();
        plus('z');
    shortcut();
        ctrl();
        shift();
        plus('z');
    shortcut();
        ctrl();
        alt();
        del();
    shortcut();
        alt();
        f4();
    end();
```

Como podemos ver, temos a chamada de vários métodos em sequência. Para este exemplo, foi usada a importação estática. Para iniciar, utilizaremos o padrão Composite como apresentado no capítulo *Encadeamento de métodos e Composite*. Assim, separamos o modelo semântico que temos do modelo da nossa DSL.

Iniciamos primeiro por uma classe para representar a DSL do nosso exemplo, como segue:

```
public class DSL {
}
```

Depois, definimos a função global `group()`. Esta função é importada estaticamente:

```
public class DSL {
    private static Group actualGroup = new Group();

    private DSL() {}

    public static void group() {
        actualGroup = new Group();
    }

    private static Group actualGroup() {
        if (actualGroup == null) {
```

```
                throw new IllegalArgumentException("group() not c
alled");
            }
            return actualGroup;
        }
    }
```

Além da função `group`, definimos também uma função `actualGroup` para testar e retornar o valor do grupo atual. E o construtor da classe foi deixado como privado para evitar que alguém acabe instanciando este objeto e utilizando-o de forma incorreta.

Posteriormente, criamos o método global `shortcut` da seguinte maneira:

```
    public static void shortcut() {
          actualGroup.addShortcut(new Shortcut());
    }
```

Depois, os outros métodos de acesso às teclas como segue:

```
    public static void ctrl() {
          actualGroup.lastShortcut().ctrl();
        }

        public static void shift() {
            actualGroup.lastShortcut().shift();
        }

        public static void alt() {
            actualGroup.lastShortcut().alt();
        }

        public static void plus(char key) {
            actualGroup.lastShortcut().plus(key);
        }

        public static void del() {
            actualGroup.lastShortcut().del();
        }
```

```
    public static void f1() {
        actualGroup.lastShortcut().f1();
    }
    ...
    public static end(){
        Group group = actualGroup;
        actualGroup = null;
        return group;
    }
```

O método `lastShortcut` retorna o último atalho do grupo atual, e o método `end` termina o fluxo, retornando o grupo. Este código é bem simples e fácil de acompanhar. Talvez o leitor já tenha percebido que o código tem uma pequena falha: ele não suporta múltiplas threads.

Para resolver este problema, precisamos fazer com que nossas variáveis de contexto não sejam modificadas ao mesmo tempo por threads diferentes, ou de forma mais simples, entregar variáveis de contexto diferente para cada uma das threads.

A plataforma Java já tem uma classe específica para tratar desta questão de variáveis estáticas em ambientes de múltiplas threads. Esta classe se chama `ThreadLocal` . Ela funciona basicamente como um Wrap para algum tipo que queiramos guardar.

Um Wrap é um invólucro de alguma coisa, tipo um papel alumínio que colocamos sobre alguma coisa. É apenas uma classe dentro da qual há uma outra classe.

Se quiséssemos guardar a varíavel `group` , usaríamos o seguinte código:

```
public class DSL {
    private static ThreadLocal<Group> groupThreadLocal = new Thre
adLocal<Group>();
```

```
    private DSL() {
    }

    public static void group() {
        groupThreadLocal.set(new Group());
    }

    private static Group actualGroup() {
        Group group = groupThreadLocal.get();
        if (group == null) {
            throw new IllegalArgumentException("group() not calle
d");
        }
        return group;
    }
    ...
}
```

Toda vez que uma thread diferente chamar `actualGroup`, cada uma vai receber seu respectivo `group` sem problemas, com a utilização em múltiplas threads.

Será que temos algum exemplo no mundo real de utilização de sequência de funções ou algo similar? Temos sim, e vamos ver isso na próxima sessão.

JMock

Existe uma biblioteca usada para testes muito conhecida chamada JMock. Em sua segunda versão (JMock 2), a sua sintaxe de utilização é algo como o seguinte:

```
    Mockery context = new Mockery();
    context.checking(new Expectations() {{
        oneOf (clock).time();
        will(returnValue(loadTime));
        oneOf (loader).load(KEY);
        will(returnValue(VALUE));
    }});
```

Esta sintaxe é um pouco incomum à primeira vista. Mas olhando com mais cuidado, podemos encontrar algumas similaridades.

O `context` é o objeto interno do framework de testes. Utilizamos este objeto para fazer algumas checagens de expectativa. A parte um pouco estranha vem agora: estamos criando uma classe anônima que deriva de `Expectations` e, de modo inline, estamos chamando alguns métodos da própria classe.

É uma forma de chamar sequência de funções, mas de uma forma um pouco mais organizada e com contexto fechado. Não precisa se preocupar com os outros métodos, este exemplo é apenas para ilustrar uma outra forma de criar DSLs utilizando sequência de funções, sem se preocupar com que contexto para chamar suas funções.

Alguns programadores Java (eu incluso), depois de alguns contatos com algumas linguagens, gostam de definir valores padrões, da seguinte maneira:

```
Map<String,String> ordem = new HashMap<String,String>(){{
     put("Kent Beck","1");
     put("Martin Fowler","2");
     put("Jeff Atwood","3");
     put("Joel Spolsky","4");
     put("Paul Graham","5");
     put("Ken Thompson","6");
     put("Brian Kernighan","7");
     put("Robert C. Martin","8");
     put("Eric C. Raymond","9");
     put("Richard Stallman","10");
     put("Linus Torvalds","11");
 }};
```

É uma forma um pouco mais compacta e agradável de olhar a

criação de coleções. Mas por favor, não use isso indisciplinadamente em seu código.

O JMock usa esta sintaxe de criação de expectativas e internamente faz uso de métodos utilizando sequência de funções. Esta biblioteca é um exemplo muito interessante e flexível de uso de DSL. A princípio, ela parece muito complicada para construir, mas ela usa várias das técnicas apresentadas neste livro, como sequência de função, encadeamentos de métodos e funções aninhadas.

No exemplo inicial do JMock, temos:

```
context.checking(new Expectations() {{
        oneOf (clock).time();
        will(returnValue(loadTime));
        oneOf (loader).load(KEY);
        will(returnValue(VALUE));
    }});
```

Dentro da classe `Expectations`, estamos criando um contexto fechado para fazer uma sequência de funções. E no seguinte código:

```
        ...
        oneOf (clock).time();
        ...
```

Temos aqui um exemplo de encadeamento de métodos. E no seguinte código:

```
        ...
        will(returnValue(loadTime));
        ...
```

Temos um exemplo de funções aninhadas.

Vimos no capítulo *Adentrando nas DSLs* como era a aparência

de funções aninhadas. Na próxima seção, veremos ver mais a fundo como implementá-las.

4.2 FUNÇÕES ANINHADAS

Uma das vantagens das funções aninhadas é poder ter a estrutura da informação em uma forma visual. A ordem de avaliação das funções aninhadas é feita das funções mais internas para, posteriormente, as mais externas.

Vejamos um exemplo hipotético. Para imprimir a seguinte frase: "Um exemplo hipotético", isto seria impresso utilizando algo como o seguinte:

```
hipotetico(exemplo(um()));
```

Esta ordem de avaliação trocada pode causar algum impacto na hora da implementação da DSL e devemos analisar com cuidado estes casos. Uma vantagem encontrada nas funções aninhadas é que a avaliação das funções é feita por último, o que minimiza a utilização de variáveis de contexto necessárias em alguns outros tipos de técnicas.

Um problema que você pode encontrar no uso de funções aninhadas são parâmetros opcionais. Se sua linguagem suporta argumentos padrão, isso facilita; mas se não existe suporte, então você terá de criar várias funções para diferentes combinações. Por exemplo, digamos que temos o seguinte código em Scala:

```
def vida(saude:Boolean, dinheiro:Int = 0){
   println(
      (if(saude) "Tenho" else "Não tenho")
      + " saúde e " +
      (if(dinheiro > 0) "tenho" else "não tenho")
```

```
        + " dinheiro."
    )
}
```

Neste pequeno trecho de código, temos que a saúde é um parâmetro obrigatório, enquanto o dinheiro é um opcional.

No lugar deste método receber dados primitivos (`Boolean` e `Int`), podemos fazê-los receberem objetos, da seguinte maneira:

```
case class Saude(saudavel:Boolean)
case class Dinheiro(valor:Int)

def S2(b:Boolean) = Saude(b)
def $(valor:Int) = Dinheiro(valor)

def vida(saude:Saude, dinheiro:Dinheiro = Dinheiro(0)){
  println(
      (if(saude.saudavel) "Tenho" else "Não tenho")
      + " saúde e " +
      (if(dinheiro.valor > 0) "tenho" else "não tenho")
      + " dinheiro."
    )
}
```

Desta forma, podemos chamar o método `vida` com ou sem parâmetros, dessa maneira:

```
scala>  vida(S2(true))
Tenho saúde e não tenho dinheiro.

scala> vida(S2(true), $(1000))
Tenho saúde e tenho dinheiro.
```

Se Scala não suportasse parâmetros opcionais para executar este código, eu teria de criar dois métodos `vida` .

Agora que já entendemos um pouco sobre o funcionamento de funções aninhadas, vamos mais a fundo, voltando ao nosso exemplo anterior de atalhos que já conhecemos.

Utilizando funções aninhadas

Para compreender melhor como usar as funções aninhadas, vamos voltar ao exemplo do capítulo *Adentrando nas DSLs*. Lá descrevemos o mesmo exemplo da biblioteca de atalhos, e o código era o seguinte:

```
group(
    shortcut(
            ctrl(),
            plus('z')
    ),
    shortcut(
            ctrl(),
            shift(),
            plus('z')
    ),
    shortcut(
            ctrl(),
            alt(),
            del()
    ),
    shortcut(
            alt(),
            f4()
    )
);
```

Como se pode ver no código anterior, existe um agrupamento das funções de uma forma hierárquica, uma dentro das outras. Para construir esta DSL, primeiramente criamos nosso objeto `Dsl`.

```
public class DSL {}
```

Posteriormente, criamos a primeira chamada referente ao método `group`.

```
public class DSL {
     public static Group group() {
```

```
        Group group = new Group();
        return group;
    }
}
```

Este método `group` pode receber um conjunto de `shortcuts`, então vamos adicionar em nossa classe também.

```
public class DSL {
      public static Group group(Shortcut... shortcut) {
         Group group = new Group();
         for (Shortcut s : shortcut) {
             group.addShortcut(s);
         }
         return group;
      }
}
```

Um atalho é constituído de várias teclas, logo adicionamos esta característica também.

```
public class DSL {
    ...
    public static Shortcut shortcut(Key... keys) {
        Shortcut shortcut = new Shortcut();
        for (Key key : keys) {
            shortcut.addKey(key);
        }
        return shortcut;
    }
}
```

Temos um conjunto de várias teclas em um teclado. Estas podem ser de caractere, especiais ou ainda modificadores. Elas são mapeadas na classe `ModifierKey`, então vamos fazer a importação estática desta classe e criar os métodos para cada uma das teclas, da seguinte maneira:

```
...
import static dsl.cap02.example1.model.shortcut.ModifierKey.*;
...
```

```
public class DSL {
    ...
    public static Key ctrl() {
        return Ctrl;
    }
    public static Key alt() {
        return Alt;
    }
    public static Key shift() {
        return Shift;
    }
    public static Key del() {
        return Del;
    }
    public static Key plus(char c) {
        return new CharKey(c);
    }
    public static Key f1() {
        return F1;
    }
    ...
}
```

Agora, de forma simples, conseguimos usar funções aninhadas:

```
    group(
        shortcut(
                ctrl(),
                plus('z')
        ),
        shortcut(
                ctrl(),
                shift(),
                plus('z')
        ),
        shortcut(
                ctrl(),
                alt(),
                del()
        ),
        shortcut(
                alt(),
                f4()
        )
```

```
);
```

Neste exemplo, podemos ver como as funções aninhadas combinaram de forma muito prática com nosso exemplo, deixando a estrutura mais explícita e simples de compreender. Até agora, vimos várias formas de como usar sequência de funções e funções aninhadas em vários códigos, mas talvez o leitor ainda esteja precisando de algum embasamento um pouco mais profundo para entender o que pode ser feito com estas técnicas.

Para isso, criei uma pequena biblioteca em Scala, chamada *Refine*. Vamos ver como ela funciona?

4.3 REFINE

A Refine é uma pequena biblioteca para filtragem e seleção de arquivos. Ela utiliza funções aninhadas para construir as seleções e os filtros dos arquivos. Por exemplo, para filtrar todos os arquivos de algum diretório qualquer, temos:

```
implicit val testDir = new File(resDirectory, "list_01")
val fileList = files(testDir)
```

Podemos também fazer um filtro nos arquivos do diretório por algum nome específico, como mostrado no seguinte trecho de código:

```
implicit val testDir = new File(resDirectory, "list_01")

var list = filter(files, name("file"))
```

Outra opção da biblioteca é filtrar arquivos maiores que um tamanho específico, da seguinte maneira:

```
implicit val testDir = new File(resDirectory, "list_01")
val list = filter(files, name("file"), size(bigger(1))) // bi
```

```
gger that 1 byte
```

Outra opção da biblioteca é carregar todos os arquivos recursivamente, ou seja, pegando os arquivos dentro dos diretórios. Podemos, então, fazer a mesma coisa filtrando todos os arquivos da pasta, como mostrado no seguinte trecho de código:

```
    implicit val testDir = new File(resDirectory, "list_01")
    val list = filter(recursive(files), name("file"), size(bigge
r(1)))
```

Existem outras opções nesta biblioteca, mas acredito que já deu para ter uma pequena compreensão de como funções aninhadas são usadas no mundo real.

> A biblioteca Refine pode ser encontrada em: https://github.com/leonardootto/refine.

4.4 E AGORA?

Neste capítulo, aprendemos sobre sequência de funções e funções aninhadas. Mas será que existem outras construções além das apresentadas até agora?

Sim, existem outras construções menos complexas! Mas não menos importantes. Aprenderemos mais sobre elas nos próximos capítulos.

Capítulo 5

OUTRAS TÉCNICAS

Além das técnicas aprendidas nos capítulos anteriores, existem outras não tão comuns que ajudam na construção de DSLs. Apesar de não serem tão usuais, elas são utilizadas em casos específicos e podem ser necessárias para melhorar sua construção.

Neste capítulo, aprenderemos como usá-las e detalharemos cada uma delas. Vamos aprender sobre símbolos, parâmetros nomeados, closures, extensão de literais, recepção dinâmica e testes. E elas podem ou não ser usadas em conjunto.

5.1 SÍMBOLOS

Às vezes, a DSL precisa referenciar objetos em vários pontos no código. Por exemplo, no seguinte código:

```
Group g = new Group();
Shortcut s = new Shortcut();
CharKey z = new CharKey('z');

s.addKey(Ctrl);
s.addKey(z);
g.addShortcut(s);

s = new Shortcut();
s.addKey(Ctrl);
s.addKey(Shift);
s.addKey(z);
```

```
g.addShortcut(s);
```

Criamos a referência à tecla `z` e a usamos duas vezes. Às vezes, podemos criar uma string que está associada a essa referência para podermos utilizá-la em vários lugares do código. Na maioria das linguagens, nomeamos os objetos usando `strings`; em outras linguagens, existe o conceito de símbolo, que podemos usar no lugar de strings.

Em Java, não temos este tipo especial de dados, mas, em Scala, temos. Em Scala, ele é nomeado com uma aspa simples, seguido do identificador. Como símbolos são utilizados principalmente para buscas, ao nos depararmos com eles, temos uma compreensão mais clara do comportamento do código. Veja um exemplo:

```
val symbol = 'meu_simbolo
val igual = symbol == 'meu_simbolo
println("São iguais? " + igual)
```

Símbolos diferem de strings em seu comportamento, pois são constantes. Não faz muito sentido concatenar ou remover partes de símbolos. Eles são usados principalmente para buscas, logo, são mais adequados para uso como chaves em mapas, por exemplo.

Por exemplo, no shell do Scala, podemos fazer o seguinte:

```
scala> //criando um símbolo

scala> val sym = 'a
sym: Symbol = 'a

scala> //comparando simbolos

scala> sym == 'a
res0: Boolean = true

scala> //buscando via símbolos
```

```
    scala> val map = Map(('a,1),('b,2))
    map: scala.collection.immutable.Map[Symbol,Int] = Map('a -> 1
, 'b -> 2)

    scala> map('a)
    res1: Int = 1

    scala> map('b)
    res2: Int = 2
```

Usamos símbolos no exemplo dos atalhos no capítulo *Adentrando nas DSLs*, como no exemplo a seguir:

```
group(
   shortcut('ctrl -> 'alt -> "z")
   shortcut('ctrl -> 'shift -> "z")
   shortcut('ctrl -> 'alt -> 'del)
   shortcut('alt  -> 'f4)
)
```

Para construir esta DSL, primeiramente adicionamos os métodos padrões para grupos, como:

```
object Dsl {
  def group(shortcuts: Shortcut*): Group = {
    val group = new Group()
    shortcuts.foreach(group.addShortcut)
    group
  }
}
```

Isso permite ter um método global `group` com vários atalhos dentro. Posteriormente, adicionamos um método similar para os atalhos, como no código seguinte:

```
object Dsl {
  def group(shortcuts: Shortcut*): Group = {
    val group = new Group()
    shortcuts.foreach(group.addShortcut)
    group
  }
```

```
  def shortcut(keys: Seq[Key]): Shortcut = {
    val shortcut = new Shortcut
    keys.foreach(shortcut.addKey)
    shortcut
  }
  // podemos ter apenas uma tecla no atalho
  def shortcut(key: Key): Shortcut = {
    shortcut(List(key))
  }
}
```

Os símbolos não precisam ser adicionados, pois já são os próprios nomes das teclas. Por exemplo, o símbolo para as teclas especiais `shift`, `ctrl`, `alt` e `f1` são respectivamente `'shift`, `'ctrl`, `'alt` e `'f1`.

Então, de alguma forma, precisamos adicionar o método `->` aos símbolos. Fazemos isso utilizando métodos implícitos:

```
object Dsl{

 ...
 implicit class ImplicitSymbol(sym: Symbol) {
    def ->(otherSymbol: Symbol): Seq[Symbol] = {
      Seq(sym,otherSymbol)
    }
    def ->(char: Char): Seq[Key] = {
      val key = SymbolToKey(sym)
      Seq(sym,new CharKey(char))
    }
  }
}
```

Neste código, adicionamos duas possibilidades para o método `->`: um recebendo outro símbolo, e outro recebendo um caractere - assim, o retorno deste método será sempre uma sequência.

O método `symbolToKey` faz uma conversão direta de símbolos usados em nossa DSL para os atalhos do modelo semântico, como segue:

```
def symbolToKey(sym: Symbol): Key = {
  sym.name.toLowerCase match {
    case "ctrl" => ModifierKey.Ctrl
    case "alt" => ModifierKey.Alt
    case "shift" => ModifierKey.Shift
    case "del" => ModifierKey.Del
    case "f1" => ModifierKey.F1
    case "f2" => ModifierKey.F2
    case "f3" => ModifierKey.F3
    case "f4" => ModifierKey.F4
    case "f5" => ModifierKey.F5
    case "f6" => ModifierKey.F6
    case "f7" => ModifierKey.F7
    case "f8" => ModifierKey.F8
    case "f9" => ModifierKey.F9
    case "f10" => ModifierKey.F10
    case "f11" => ModifierKey.F11
    case "f12" => ModifierKey.F12
    case _ => null
  }
}
```

Vamos continuar. Criamos métodos implícitos para a sequência de símbolos.

```
...

implicit class ImplicitSeqSymbol(syms: Seq[Symbol]) {
  def ->(char: Char): Seq[Key] = {
    val keys: Seq[Key] = SeqSymbolToSeqKey(syms)
    keys :+ new CharKey(char)
  }

  def ->(otherSymbol: Symbol): Seq[Symbol] = {
    syms :+ otherSymbol
  }
}
```

Se uma sequência de símbolos receber um caractere, eles viram uma sequência final de atalhos. Já se receber outro símbolo, apenas adiciona os símbolos na sequência e retorna esta nova sequência.

Criamos também um método implícito para conversão de sequências de símbolos para sequências de atalhos, da seguinte maneira:

```
implicit def SeqSymbolToSeqKey(list: Seq[Symbol]): Seq[Key] = {
  list.map(s => symbolToKey(s))
}
```

Assim, agora podemos aninhar os símbolos em sequência e, quando tivermos todos eles, convertemos para os atalhos do nosso modelo semântico.

5.2 PARÂMETROS NOMEADOS

Uma outra construção muito útil são os parâmetros nomeados. Eles nos ajudam a atribuir dados e dão uma noção de hierarquia das informações. Esta funcionalidade não está presente em várias linguagens, apesar de ser muito usada.

Por exemplo, C, C++ e Java não suportam, mas Scala, C# 4.0, Swift e Object-C sim. Como já utilizamos Scala anteriormente, vamos usá-la para o exemplo. Podemos fazer o seguinte no shell do Scala:

```
    scala> //Definimos um método de soma simples

    scala> def sum(primeiro:Int, segundo:Int):Int = { primeiro +
segundo }
    sum: (primeiro: Int, segundo: Int)Int

    scala> //Agora podemos chamar o 'primeiro' parâmetro primeiro
 e
    scala> //o 'segundo' parâmetro por segundo

    scala> sum(primeiro = 1, segundo = 1)
    res0: Int = 2
```

```
scala> //Ou o contrário

scala> sum(segundo = 1, primeiro = 1)
res1: Int = 2
```

Eles são usados principalmente para melhorar a legibilidade. São úteis também, inclusive, para deixar a legibilidade de parâmetros booleanos mais claros. Por exemplo, no Shell do Scala:

```
scala> def linguagemBoa(permitePhp:Boolean) = {
     |
     | println("Linguagem" + ( if(permitePhp) "" else "não")+
" boa permite PHP");
     |
     | }
linguagemBoa: (permitePhp: Boolean)Unit

scala> // Agora permitimos Php por que é só um exemplo. ;)

scala> linguagemBoa(permitePhp = true)
Linguagem boa permite PHP
```

Agora que você já entendeu como parâmetros nomeados funcionam, vamos voltar para o exemplo que tivemos no capítulo *Adentrando nas DSLs*. Naquele capítulo, tínhamos o seguinte trecho de código:

```
quote(
      exchange = "NASDAQ",
      name = "Google Inc",
      symbols = ('GOOG,'GOOGL),
      industry = "Technology"
    )
```

Para criar esta DSL, primeiramente criamos nosso `object DSL` como o método recebendo nossos parâmetros:

```
object Dsl {
  def quote(exchange: String,
            name: String,
```

```
            symbols: Seq[Symbol], industry: String): Quote = {
    val quote = new Quote()
    quote.exchange = exchange
    quote.name = name
    quote.symbols = symbols
    quote.industry = industry
    quote
  }
}
```

Para o parâmetro `symbols`, fazemos uma conversão para retornar uma sequência, da seguinte maneira:

```
object Dsl {
    ...
    implicit def convertTupleToList[A <: Product](tuples: A): Seq
Symbol] = {
        val arity = tuples.productArity
        var seq: Seq[Symbol] = Seq()
        for (idx <- 0 until arity) {
          val element: Any = tuples.productElement(idx)
          element match {
            case symbol: Symbol =>
              seq = seq :+ symbol
            case _ =>
          }
        }
        seq
    }
}
```

Utilizar parâmetros nomeados facilita na criação de algumas estruturas e é uma ótima técnica para se usar.

5.3 ANOTAÇÕES

Normalmente, quando estamos programando, classificamos nossos dados e criamos regras sobre como eles são usados. Mas algumas vezes queremos ter uma interação mais flexível com o código, fazendo com que se marque algumas estruturas para serem

tratadas posteriormente de alguma forma.

Anotação é um recurso utilizado por algumas linguagens para marcar estas informações extras nos dados. No Java, podemos anotar campos, classes e métodos para serem processados posteriormente, seja em tempo de compilação ou execução. No Scala, as construções que usam anotações são semelhantes.

Por exemplo, vamos dizer que temos o seguinte modelo em Java:

```
public class Pessoa{
    private int age;
    ...
}
```

Agora poderíamos ter um gerenciador de pessoas, que testa se as pessoas são maiores de idade ou não.

```
public PeopleHandler{

    public boolean isValid(People people){
        ...
    }
}
```

Esta validação é muito simples de fazer, como segue:

```
    public class PeopleHandler {
        public boolean isValid(People people) {
            return people.getAge() >= 18;
        }
    }
```

Mas e se quisermos um validador genérico para qualquer objeto? Como podemos fazer a validação? Para fazer um validador para qualquer objeto, podemos usar `Annotation` para anotar os métodos que queremos que sejam validados. E utilizar reflexão

para conseguir esta informação.

Reflexão é a capacidade que algumas linguagens têm, assim como Java e Scala, de conseguir ler características e valores de suas próprias classes. Por exemplo, no nosso validador temos:

```
public class Handler {
    public boolean isValid(Object obj) {
        for (Field f : obj.getClass().getDeclaredFields()) {
            f.setAccessible(true);
            PropertyAge column = f.getAnnotation(PropertyAge.clas
s);
            if(column != null) {
                try {
                    Integer age = (Integer) f.get(obj);
                    return age >= 18;
                } catch (Exception e) {
                    return false;
                }
            }
        }
        return false;
    }
}
```

Nele conseguimos ler os campos declarados de uma classe em:

```
obj.getClass().getDeclaredFields()
```

E verificamos se um campo tem uma anotação em:

```
PropertyAge column = f.getAnnotation(PropertyAge.class);
```

Agora conseguimos validar qualquer objeto que tenha esta anotação. Normalmente, as anotações são utilizadas para realização de mapeamentos de um tipo de objeto para outro tipo, ou ainda para realizar validações mais flexíveis.

Para construir este exemplo em Java, usamos como funcionalidade principal reflexão. Esta funcionalidade da

linguagem permite, em tempo de execução, pegarmos qualquer objeto ou classe e descobrirmos suas características.

Poderíamos mostrar este exemplo anterior na linguagem Scala, mas reflexão ainda é uma funcionalidade beta em Scala, então não vamos abordá-la.

5.4 CLOSURES

Closures já são usados em linguagem há muito tempo, entretanto, apenas recentemente atingiram as principais linguagens de programação. Closures possuem muitas nomeações diferentes, como lambdas, blocos, entre outros, mas essencialmente são a mesma coisa.

Java só veio adquirir este recurso a partir da versão 8. Já Scala possui este recurso desde sua concepção.

Uma boa definição de closures, dada por Fowler, é que: closures são fragmentos de código que podem ser tratados como objeto.

Um exemplo comum de uso de closures é obter um subconjunto de uma coleção maior de elementos. Por exemplo, temos uma lista de jogadores de basquete, suas respectivas alturas e nacionalidades, como no código a seguir:

```
object Main {
  def main (args: Array[String]) {
      case class Player(name:String,size:Double,nationality:Strin
)

      val players = List(
        Player("Gheorghe Mureşan",2.31,"ROU"),
        Player("Manute Bol",2.31,"SUD"),
```

```
        Player("Shawn Bradley",2.29,"USA"),
        Player("Yao Ming",2.29,"CHN"),
        Player("Chuck Nevitt",2.26,"USA"),
        Player("Pavel Podkolzin",2.26,"RUS"),
        Player("Slavko Vraneš",2.26,"MNE"),
        Player("Mark Eaton",2.24,"USA"),
        Player("Mark Eaton",2.24,"NED"),
        Player("Rik Smits",2.24,"USA"),
        Player("Ralph Sampson",2.24,"USA"),
        Player("Priest Lauderdale",2.24,"USA"),
        Player("Randy Breuer",2.21,"USA"),
        Player("Keith Closs",2.21,"USA"),
        Player("Swede Halbrook",2.21,"USA"),
        Player("Žydrūnas Ilgauskas",2.21,"LTU"),
        Player("Aleksandar Radojević",2.21,"BIH"),
        Player("Peter John Ramos",2.21,"PUR"),
        Player("Arvydas Sabonis",2.21,"LTU"),
        Player("Ha Seung-Jin",2.21,"KOR"),
        Player("Hasheem Thabeet",2.21,"TZA")
      )
  }
}
```

Poderíamos calcular, por exemplo, qual o tamanho médio destes jogadores assim:

```
    var total:Double = 0
    for(i <- 0 until players.length){
      val player = players(i)
      total = total + player.size
    }
    val midSize = total / players.length
```

Agora poderíamos querer filtrar os jogadores maiores que a média, da seguinte forma:

```
  //maiores
  var bigger: List[Player] = List()
  for (i <- 0 until players.length) {
    val player = players(i)
    if (player.size >= midSize) {
      bigger = bigger :+ player
    }
```

```
}
```

Também poderíamos querer filtrar os jogadores dos Estados Unidos, da seguinte maneira:

```
//dos USA
var fromUSA: List[Player] = List()
for (i <- 0 until players.length) {
  val player = players(i)
  if (player.nationality == "USA") {
    fromUSA = fromUSA :+ player
  }
}
```

Estes dois últimos códigos têm muitas linhas repetidas, e estamos sempre retornando uma lista com os elementos filtrados por uma determinada função. A forma mais simples de eliminar essa duplicação é criando um objeto para realizar o filtro. Por exemplo, poderíamos ter o objeto `Filter`, como no seguinte código:

```
class Filter[T](elements:List[T]){
  def filter(filter:FilterFunction[T]): Unit ={
    var list = List[T]()
    for(t <- elements){
      if(filter.apply(t)){
        list = list:+t
      }
    }
  }
}
```

E também uma classe que serve como função de filtro, como a seguir:

```
trait FilterFunction[T] {
  def filter(value: T): Boolean
}
```

Assim, poderíamos criar a busca pelos maiores jogadores dessa

forma:

```
    //maiores
    class PlayerBigSizeFilter(midSize: Double) extends FilterFunc
tion[Player] {
      override def filter(player: Player): Boolean = {
        if (player.size >= midSize) {
          true
        } else {
          false
        }
      }
    }
    val list = new Filter(players).filter(new PlayerBigSizeFilter
(midSize))
```

De forma semelhante, poderíamos buscar pelas nacionalidades americanas, como mostra o código:

```
    // dos estados unidos
    class PlayerFromNationality(nationality: String) extends Filt
erFunction[Player] {
      override def filter(player: Player): Boolean = {
        if (player.nationality == nationality) {
          true
        } else {
          false
        }
      }
    }
    var usaPlayers = new Filter(players).filter(new PlayerFromNat
ionality("USA"))
```

Como podemos ver, conseguimos remover a duplicação, mas ao custo de mais código. Usando closures, teríamos algo como o seguinte:

```
    //maiores jogadores
    val maiores = players.filter(p=> p.size > midSize)
    //americanos
    val americanos = players.filter(p=> p.nationality == "USA")
```

Agora temos um código muito mais conciso, simplificado e sem duplicações. A grande utilidade das closures é sua abstração e sua forma compacta, isso torna-os ferramentas muito interessantes para criação de DSLs.

Exemplo de closures

No exemplo do capítulo *Adentrando nas DSLs*, demonstramos a seguinte DSL.

```
StockBuilder(stock => {
   stock exchange "NASDAQ"
   stock name "Google Inc"
   stock symbols("GOOG", "GOOGL")
   stock industry "Technology"
 })
```

Para criar esta DSL, primeiramente criamos uma classe `StockBuilder`:

```
class StockBuilder {
  var exchange_ : String = ""
  var name_ : String = ""
  var symbols_ : Seq[String] = Seq()
  var industry_ : String = ""

  def exchange(exchange: String) = {
    exchange_ = exchange
  }

  def name(name: String) = {
    name_ = name
  }

  def symbols(symbols: String*) = {
    symbols_ = symbols
  }

  def industry(industry: String) = {
    industry_ = industry
```

```
  }

  def build():Stock = {
    new Stock(exchange_,name_,symbols_,industry_)
  }
}
```

Este objeto será responsável pela criação do nosso objeto `Stock`. Posteriormente, criamos um objeto `StockBuilder`:

```
object StockBuilder {
  def apply(bilderFunction: StockBuilder => Unit):Stock = {
    val builder = new StockBuilder()
    bilderFunction(builder)
    builder.build()
  }
}
```

Pronto, isso é tudo que precisamos fazer. Aproveitamos uma boa funcionalidade da linguagem Scala para remover um pouco do ruído, resultando em chamadas sem pontos ou parênteses. Como anteriormente, teremos o seguinte código:

```
    StockBuilder(stock => {
       stock exchange "NASDAQ"
       stock name "Google Inc"
       stock symbols("GOOG", "GOOGL")
       stock industry "Technology"
     })
```

5.5 EXTENSÃO DE LITERAIS

Em computação, temos dois termos parecidos para designar conceitos diferenciados. Quando temos algo explícito, estamos nos referindo a coisas ou ações tomadas diretamente por nós. Já quando temos algo implícito, estamos nos referindo a coisas ou ações tomadas indiretamente por nós.

De uma forma mais simples, em programação, chamar uma função é uma chamada explícita. Já o que essa função faz ou se outras funções são chamadas a partir dela é uma definição implícita da função.

Este conceito de coisas implícitas (implicits) é utilizado na linguagem Scala para fazer extensão de literais. Usamos a palavra reservada `implicits` para descrever isso. Java infelizmente não possui o recurso de extensão de literais, talvez seja adicionado em edições futuras.

A extensão de literais facilita a criação de algumas DSLs interessantes. Por exemplo no capítulo *Adentrando nas DSLs*, temos o seguinte código:

```
    val equalsGrams = (1.kg + 1.kg + 500.g + 220.hectogram) == 24
.5.kg
    val equalsMeters = (1.m + 50.cm + 50.decimeter) == 6.5.m
```

Para criar este exemplo, primeiramente criamos as classes dos nossos modelos. Para as gramas, criamos a seguinte classe do modelo:

```
class Gram(val value: Double, val tenExpo: Long) {
  def +(number: Int) {
    new Gram(value + number, tenExpo)
  }

  def +(other: Gram): Gram = {
    if (other.tenExpo == tenExpo) {
      new Gram(value + other.value, tenExpo)
    } else {
      val otherValueConverted: Double = other.toGram / pow(10, te
nExpo)
      new Gram(value + otherValueConverted, tenExpo)
    }
  }
```

```
  def toGram: Double = {
    value * pow(10, tenExpo)
  }
}
```

De forma similar, criamos uma classe para representar os metros, da seguinte forma:

```
class Meters(val value: Double, val tenExpo: Long) {
  def +(number: Int) {
    new Meters(value + number, tenExpo)
  }

  def +(other: Meters): Meters = {
    if (other.tenExpo == tenExpo) {
      new Meters(value + other.value, tenExpo)
    } else {
      val otherValueConverted: Double = other.toMeters / pow(10,
tenExpo)
      new Meters(value + otherValueConverted, tenExpo)
    }
  }

  def toMeters: Double = {
    value * Math.pow(10, tenExpo)
  }
}
```

Cada unidade de medida possui um valor e um múltiplo ou submúltiplo representado pela potência. Por exemplo, 1km tem o valor 1 e potência 3. Após criar estas classes, podemos adicionar os métodos aos nossos valores:

```
object Dsl {

  implicit class IntImplicitsGrams(number: Double) {
    def kg = kilogram
    def kilogram: Gram = {
      new Gram(number, 3)
    }
    def hg = hectogram
    def hectogram = {
```

```
      new Gram(number, 2)
    }
    def dag = dekagram
    def dekagram = {
      new Gram(number, 1)
    }
    def g = gram
    def gram = {
      new Gram(number, 0)
    }
    def dg = decigram
    def decigram = {
      new Gram(number, -1)
    }
    def cg = centigram
    def centigram = {
      new Gram(number, -2)
    }

    def mg = miligram
    def miligram = {
      new Gram(number, -3)
    }
  }
}
```

Para as medidas em metro, criamos outra classe similar, como segue:

```
implicit class IntImplicitsMeters(number: Double) {
    def km = kilometer
    def kilometer = {
      new Meters(number, 3)
    }

    def hm = hectometer
    def hectometer = {
      new Meters(number, 2)
    }

    def dam = dekameter
    def dekameter = {
      new Meters(number, 1)
    }
```

```
  def m = meter
  def meter = {
    new Meters(number, 0)
  }

  def dm = decimeter
  def decimeter = {
    new Meters(number, -1)
  }

  def cm = centimeter
  def centimeter = {
    new Meters(number, -2)
  }

  def mm = milimeter
  def milimeter = {
    new Meters(number, -3)
  }

}
```

Estas classes vão fazer a conversão implícita dos valores para seus tipos específicos. Se nosso problema de domínio fosse relacionado a cálculos com várias escalas, a utilização de uma DSL simples como esta seria de grande interesse. Ela facilitaria o entendimento e a comunicação entre os especialistas e os desenvolvedores, e tornaria explícito o comportamento do código de uma forma simples.

5.6 RECEPÇÃO DINÂMICA

Recepção dinâmica é a capacidade de realizar a chamada de método em tempo de execução e não em tempo de compilação. Em linguagens interpretadas, é muito simples realizar este tipo de chamada; já em compiladas, isso não é muito comum.

Apesar disso, algumas linguagens compiladas adicionaram um mecanismo similar para fazer chamadas dinamicamente. É interessante utilizar recepção dinâmica quando estamos construindo DSL, pois ela nos ajuda a reduzir a sintaxe da linguagem.

Java não suporta este recurso, então nosso exemplo será utilizando Scala. Por exemplo, no shell do Scala, podemos fazer o seguinte:

```
shell> //importamos a biblioteca

scala> import scala.language.dynamics
import scala.language.dynamics

shell> //Criamos a classe dinamica

scala> class DynImpl extends Dynamic {
     | def selectDynamic(name: String) = name
     | }
defined class DynImpl

shell> //Instanciamos a classe

scala> val d = new DynImpl
d: DynImpl = DynImpl@76c52298

shell> //Executamos o método dinamico

scala> d.nome_do_metodo
res1: String = nome_do_metodo
```

Neste exemplo, criamos uma classe dinâmica e executamos um *nome_do_metodo* que vai retornar este mesmo nome `"nome_do_metodo"`. Poderíamos ter usado qualquer nome para executar, já que este método é dinâmico.

Então, poderíamos ter algo como o seguinte:

```
objeto.metodo("parametro")
```

E simplificar para:

```
objeto.parametro
```

Com este conceito em mente, podemos retornar a um dos nossos exemplos iniciais. Lá tínhamos a seguinte DSL para consulta de arquivos:

```
Ribbon(f => {
    f.`directory C:\\Windows\\Logs`
    f.extension_log
  }).foreach(println)
```

Para construir essa DSL, primeiramente construímos um objeto `Ribbon`, da seguinte maneira:

```
object Ribbon {
  def apply(func: (Ribbon) => Unit): List[String] = {
    val rDef = new Ribbon
    func(rDef)
    rDef.list
  }
}
```

Este objeto receberá uma função que recebe um objeto do tipo `Ribbon` para configurar os filtros de arquivos. A classe `Ribbon` vai receber métodos dinâmicos com o diretório e a extensão e, com estas informações, retornará uma lista de caminhos dos arquivos, dessa forma:

```
class Ribbon extends Dynamic {
  var dir: String = ""
  var ext: String = ""

  def selectDynamic(methodName: String) {
    methodName match {
      case extension if methodName.startsWith("extension") =>
```

```
        this.ext = methodName.replace("extension_", "")
      case directory if methodName.startsWith("directory") =>
        this.dir = methodName.replace("directory ", "")
    }
  }

  def directory(dir: String) = {
    this.dir = dir
  }

   def list: List[String] = {
    new File(this.dir).list(new FilenameFilter {
      override def accept(dir: File, name: String): Boolean = {
        name.endsWith(ext)
      }
    }).toList
  }
}
```

O método `selectDynamic` vai ser chamado toda vez que um método da `class Ribbon` for chamado também. Scala permite chamar métodos com nomes que tenham espaços e outros caracteres diferentes. No exemplo, foi usado um método com espaço e outro para descrever o tipo da extensão dos arquivos, como no exemplo a seguir:

```
Ribbon(f => {
    f.`directory C:\\Windows\\Logs`
    f.extension_log
  }).foreach(println)
```

Então, como explicado no início, a utilização de recepção dinâmica nos permite neste exemplo reduzir a sintaxe, deixando a leitura mais simples e direta.

5.7 TESTES

Testes são uma parte essencial do desenvolvimento de

software, e não poderia ser diferente no desenvolvimento de DSLs. Tanto internas como externas, a importância de testar seu software é enorme, e existem muitos outros livros que cobrem este tema. Isso deve ser uma prática comum no desenvolvimento de software.

Como descrito anteriormente, o objetivo de uma DSL é fazer o preenchimento de um modelo semântico. Então, podemos utilizar duas estratégias para se testar: testar o modelo semântico, e se sua DSL preenche corretamente este modelo.

Testando o modelo semântico

Testar o modelo semântico garante que ele se comporta de maneira esperada. Os métodos fazem o que deviam fazer para todos os casos, e casos pouco comuns são avaliados.

Este teste é similar ao que se costuma fazer em outros tipos de modelos. Não precisamos da DSL para testar o modelo, podemos utilizar sua interface básica para isso.

Por exemplo, no capítulo *Adentrando nas DSLs*, mostramos um exemplo de biblioteca de atalhos. Tínhamos um fragmento de código da seguinte forma:

```
Group g = new Group();
Shortcut s = new Shortcut();
s.addKey(Ctrl);
s.addKey(new CharKey('z'));
g.addShortcut(s);
```

Como regra, um atalho não tem teclas repetidas, então poderíamos ter um teste para avaliar esta característica assim:

```
public class TestModel {

    @Test(expected = IllegalStateException.class)
```

```
    public void testSpecialRepetition() {
        Group g = new Group();
        Shortcut s = new Shortcut();
        s.addKey(Ctrl);
        s.addKey(Ctrl);
        g.addShortcut(s);
    }

    @Test(expected = IllegalStateException.class)
    public void testCharKeyRepetition() {
        Group g = new Group();
        Shortcut s = new Shortcut();
        s.addKey(new CharKey('k'));
        s.addKey(new CharKey('k'));
        s.addKey(new CharKey('k'));
        g.addShortcut(s);
    }
}
```

No primeiro teste, verificamos se a API proíbe o uso de teclas especiais repetidas. No segundo, vemos se existe a proibição de teclas normais repetidas. Estas proibições devem ser respeitadas neste modelo, por isso o teste.

Estamos testando as funcionalidades do modelo semântico externamente, e isso é ótimo para separar os testes e as responsabilidades. Também estamos desacoplando a execução com o preenchimento do modelo semântico.

Testando sua DSL

Você já deve estar cansado, mas vamos repetir mais uma vez: o papel de uma DSL é preencher o modelo semântico. E como vimos anteriormente, é uma boa prática criar um modelo semântico separado.

Então, para testar, podemos verificar se o modelo foi criado corretamente. Nos exemplos do livro, temos vários casos desta

verificação. No capítulo *Batendo uma bola com sua DSL*, vimos que podemos ter a criação de um XML para criar o modelo da seguinte forma:

```
public class Main {

    public static void main(String[] args) {
        try {
            Game game = createGame();
            Game game2 = loadFromXMLFile("game.xml");
            System.out.println("Iguais? " + game.equals(game2));

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    ...
```

Neste exemplo, temos a verificação se o modelo criado diretamente é exatamente o mesmo criado pelo carregamento do XML. No uso da DSL, no exemplo posterior, temos o seguinte grupo de código:

```
public class Main {

    public static void main(String[] args) {
        Game game = directModel();
        Game game2 = dslModel();
        System.out.println("Iguais? " + game.equals(game2));
    }
    ...
```

Este pequeno trecho de código testa para verificar se o modelo criado diretamente é semelhante ao criado pela DSL. Poderíamos migrar este código para o seguinte teste unitário:

```
public class TestMain {

    @Test
    public void test() {
        Game game = directModel();
```

```
        Game game2 = dslModel();
        Assert.assertEquals(game, game2);
    }
    ....
```

No exemplo anterior, testamos positivamente, ou seja, testamos se uma entrada positiva cria corretamente um modelo semântico adequado. Mas existem outros tipos de testes que você também deve realizar, que são os testes negativos.

Neles testamos entradas inválidas que podem causar erros em suas DSLs ou no modelo semântico. Testes negativos também ajudam a definir que nível de tratamento de erro que queremos, pois podemos detalhar em mais detalhes uma utilização incorreta da DSL.

Um bom motivo para testar é que os testes ajudam a verificar se quebramos algo em cada uma das versões diferentes das DSLs. Ou seja, ajudam nas migrações das versões; vamos ver mais sobre isso na próxima seção.

5.8 MIGRANDO DSL INTERNAS

As DSLs sofrem do mesmo problema de bibliotecas: uma vez publicadas suas interfaces, é necessário manter a compatibilidade. Se você atualiza estas interfaces, pode quebrar códigos que utilizam suas DSLs futuramente.

Uma forma de evitar este problema é fornecer ferramentas automatizadas para migrar a DSL de uma versão para outra. Com DSLs externas, este trabalho é um pouco mais simples do que com as internas. Normalmente, estas ferramentas não existem no mercado, você mesmo vai ter de construí-las.

Outra abordagem é criar várias DSLs versionadas. Como, por boa prática, teremos um modelo semântico, e já que o papel da DSL é preencher esse modelo, podemos ter várias versões que realizam esta tarefa. Se for possível e em tempo hábil, é interessante criar uma ferramenta que migra de uma versão para outra, fazendo as correções necessárias.

Vimos até agora vários tipos de formas de construção dentro das DSLs que seguem um fluxo similar. Será que existem alguns outros fluxos diferentes?

Sim, temos outros bem diferentes, e veremos isso na próxima seção. Estes novos modelos vão nos ajudar a pensar de formas diferentes na resolução de problemas.

5.9 MODELOS ALTERNATIVOS

Atualmente, o modelo de computação que reina em programação é o modelo imperativo, ou seja, um modelo que define computações como uma sequência de passos. Um dos motivos deste reinado é que ele é um modelo simples de ser entendido, uma sequência de passos é algo bem simples de se seguir.

Se o que você quer resolver for uma sequência de ações, tudo bem; mas se for algo diferente, talvez seja interessante tentar um modelo alternativo. Por exemplo, no shell do Scala, podemos fazer:

```
scala>//Criamos uma lista de elementos

scala> val list = List(1,2,3,4,5,6,7,8,9)
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala>//no modelo imperativo dizemos exatamente como encontrar
```

```
o número

  scala> for( i <- 0 to list.length - 1){
       |    if(list(i) == 5) print("find 5 ")
       | }
  find 5 in 4 position

  scala>//no modelo declarativo dizemos o que queremos mas não co
mo

  scala> if(list.contains(5)) print("find 5")
```

Este é um exemplo simples, mas demonstra tipos diferentes de computação.

Agora, depois de entendido que podemos ter mais de um tipo de modelo de computação, vamos ver três tipos de modelos alternativos: tabelas de decisões, rede de dependências e máquinas de estado. Também veremos como cada um deles executa estes modelos alternativos.

Tabela de decisões

Um tipo de modelo computacional muito usado para quando temos diversas condições resultando em diferentes ações são as tabelas de decisão.

Por exemplo, digamos que queremos descobrir se, em um determinado jogo de futebol, ocorreu ou não o impedimento de um jogador segundo as regras oficiais. Uma tabela de decisão para impedimento seria aproximadamente, como a seguinte:

Impedimento

Condições					
Mais perto da linha de meta que penultimo	S				
Própria metade de compo	N		S		
Mesma linha penultimo	N			S	
Mesma linha dois ultimos	N				S
Tiro de meta, Arremesso lateral, ou Tiro de canto	N	S			
Consequências					
Impedido ?	S	N	N	N	N

Figura 5.1: Tabela de decisão

Elas são divididas entre condições e consequências. As condições descrevem todas as condições que temos para chegar às consequências, e as consequências dizem o que devemos realizar quando tivermos as condições atingidas.

Em alguns casos, para simplificar o tamanho da tabela, criamos um terceiro estado além do sim e não, que é "não importa". Na tabela que criamos, os espaços em branco são estes estados.

Uma vantagem de se utilizar este modelo é que ele é simples de ser seguido e se comunica bem com os especialistas de domínio. Estes especialistas costumam ter habilidades de trabalhar com planilhas, então é uma boa tática permitir que se importe planilhas para o sistema.

Uma desvantagem deste modelo é que ele consegue tratar apenas de certo grau de complexidade. Para problemas mais complexos, é mais interessante utilizar um sistema de regras de

produção, que veremos posteriormente.

Se quisermos transformar esta tabela de decisão em código, poderíamos começar criando algumas classes. Vamos utilizar Scala, pois, como é uma linguagem mais compacta (ou seja, produz códigos-fontes menores), vai ser mais fácil para o leitor acompanhar. Poderíamos começar criando uma classe para os valores da tabela, da seguinte maneira:

```
object TableValue {
  case object S extends TableValue
  case object N extends TableValue
  case object X extends TableValue

  sealed class TableValue() {
    override def equals(any: scala.Any): Boolean = {
      if (!any.isInstanceOf[AnyRef]) return false

      val ref = any.asInstanceOf[AnyRef]
      if (this.eq(X) || ref.eq(X)) {
        true
      } else {
        this eq ref
      }
    }
  }
}
```

Neste exemplo, temos os valores possíveis para a tabela de decisão, que são `Sim`, `Não` e `Não Importa`. Criei um método `equals` para conseguir que o object `X` aceitasse comparações com as outras duas classes `N` e `S`.

Posteriormente, podemos criar uma classe para representar uma linha da seguinte maneira:

```
case class Row(values: List[TableValue], string: String)
```

Em uma linha da tabela, temos um texto e os valores da tabela.

Inverti a ordem para deixar a tabela mais agradável de ser vista. Agora já podemos criar uma classe que engloba estas linhas. Uma tabela de decisão é formada por linhas de decisões e consequências, então podemos modelar esta classe da seguinte maneira:

```
case class Table
(
  rows: List[Row],
  cons: List[Row]
)
```

Com `rows` sendo as linhas da tabela, e `cons` sendo as consequências, precisamos também de um método para verificar, em uma determinada entrada, qual saída teremos:

```
case class Table
(
  rows: List[Row],
  cons: List[Row]
) {
  def result(input: List[TableValue]): List[TableValue] = {
    for (column <- 0 to rows.head.values.size) {
      val colList = getColList(column, rows)
      if (input.equals(colList)) {
        return getColList(column, cons)
      }
    }
    List()
  }

  def getColList(col: Int, list: List[Row]): List[TableValue] = {
    list.collect({
      case row: Row if col < row.values.length => row.values(col)
    })
  }
}
```

Um exemplo de utilização deste código seria o seguinte:

```
object Main {
  def main(args: Array[String]) {
    import TableValue._;
```

```
    val table = Table(
      List(
        Row(List(S, N, N, N, N), "Mais perto da linha da meta que
 penultimo"),
        Row(List(X, X, X, X, S), "Própria metade de campo"),
        Row(List(X, S, X, X, X), "Mesma linha que penúltimo"),
        Row(List(X, X, S, X, X), "Mesma linha dois últimos"),
        Row(List(X, X, X, S, X), "Tiro de meta, Arremesso lateral
 ou tiro de canto")
      ),
      List(
        Row(List(S, N, N, N, N), "Impedido")
      ))

    val result = table.result(List(S, S, S, N, N))
    println(result)
  }
}
```

Neste exemplo, criamos a tabela de decisão, as condições e as consequências, e verificamos para cada entrada qual seria a saída. É um exemplo simples, mas demonstra como seria a criação de um modelo alternativo de computação.

Rede de dependências

Outro modelo alternativo muito usado é o da rede de dependências. Neste modelo, temos um conjunto de tarefas que podem ser executadas em ordem, ou fora de ordem, mas que dependem umas das outras.

No Java e Scala, temos a ferramenta Ant, que serve para este propósito de executar um conjunto de tarefas relacionadas. Nesta ferramenta, temos um arquivo de build e várias tarefas interdependentes para executar compilação, limpeza e empacotamento das classes Java.

Por exemplo, um arquivo de build poderia ser descrito como:

```
<project name="ProjetoName" default="init" basedir=".">
    <target name="init">
        <echo message="init"/>
    </target>
    <target name="prepare" depends="init">
        <echo message="prepare"/>
    </target>
    <target name="compile" depends="prepare">
        <echo message="compile"/>
    </target>
    <target name="package" depends="clean,compile">
        <echo message="packing"/>
    </target>
    <target name="run" depends="compile">
        <echo message="run"/>
    </target>
    <target name="clean" depends="init">
        <echo message="clean"/>
    </target>
</project>
```

Neste arquivo, descrevemos um conjunto de tarefas utilizando a tag XML `target`, e cada uma destas tarefas podem ser dependentes de outra tarefa. Neste exemplo, a tarefa `compile` depende da tarefa `prepare`, e assim por diante.

Não cobriremos como instalar a ferramenta Ant aqui. Porém, um manual de instalação do Ant pode ser consultado em sua página oficial, em http://ant.apache.org/manual/install.html.

Depois de instalada, com este arquivo de build, poderíamos executar o seguinte comando:

```
shell> ant compile
```

E teríamos o seguinte resultado final:

```
 ant compile
```

```
Buildfile: ../build.xml

init:
     [echo] init

prepare:
     [echo] prepare

compile:
     [echo] compile

BUILD SUCCESSFUL
Total time: 0 seconds
```

Podemos ver como a ferramenta, antes de executar a tarefa de compilação, executa as tarefas dependentes dela. E a rede de dependências funciona exatamente da mesma forma.

Um detalhe importante a se ressaltar é que as tarefas não são executadas de forma duplicada. Se duas tarefas tiverem dependências entre si e dependerem também de uma terceira, a terceira só será executada uma única vez.

Poderíamos ter uma DSL similar para construção de execução de tarefas. Seria algo similar ao seguinte código:

```
object Main extends TaskBuilder {

  def main(args: Array[String]) {
    task("init") {
      println("init")
    }
    task("prepare", depends = "init") {
      println("prepare")
    }
    task("compile", depends = "prepare") {
      println("compile")
    }
    task("package", depends = "clean,compile") {
      println("package")
    }
```

```
    task("run", depends = "compile") {
      println("run")
    }
    task("clean", depends = "init") {
      println("clean")
    }

    run("package")
  }
}
```

Neste exemplo, temos um caso similar ao Ant, mas em código nativo. Para fazer a modelagem desta rede de dependência, começamos modelando uma tarefa da seguinte maneira:

```
object Task{
  def apply(name: String, body: => Unit): Task = new Task(name, b
ody)
}

class Task(val name: String, body: => Unit) {
  def call(): Unit = {
    body
  }
}
```

Aqui temos a classe `Task`, constituída de um bloco e um nome. Posteriormente, modelamos as dependências entre uma tarefa e as outras, do seguinte modo:

```
case class Dependency(task: Task, dependsOn: String)
```

Depois, fazemos a implementação da classe `TaskBuilder`:

```
trait TaskBuilder {

  var list = List[Dependency]()
  var alreadyRun = List[Task]()

  def task(name: String, depends: String = "")(body: => Unit): Un
it = {
    list = list :+ Dependency(new Task(name, body), depends)
```

```
  }

  def run(taskName: String): Unit = {

    getTaskDependencies(taskName).foreach(task => {
      task.dependsOn.split(",").foreach(taskName => {
        run(taskName)
      })
    })
    if (getTaskByName(taskName).isDefined) {
      val task = getTaskByName(taskName).get

      if (!alreadyRun.contains(task)) {
        task.call()
        alreadyRun = alreadyRun :+ task
      }
    }
  }

  def getTaskDependencies(name: String): Option[Dependency] = {
    list.find(_.task.name == name)
  }

  def getTaskByName(name: String): Option[Task] = {
    getTaskDependencies(name).map(_.task)
  }
}
```

A implementação feita com `Trait` é a que faz mais sentido para o contexto. Ela tem uma lista de tarefas e uma lista de tarefas já executadas. Tem também um método para adicionar as tarefas e duas dependências no método `task`.

E tem também um método `run` que faz a execução do código. Este método faz duas coisas principais: primeiramente, executa todas as dependências e, depois, executa a tarefa final.

Este código é reentrante, pois cada tarefa pode ter suas próprias dependências. Uma coisa que foi implementada é uma lista das tarefas já executadas, pois não podemos executar uma tarefa de

forma duplicada.

Máquinas de estado

Máquinas de estado são um modelo matemático para representar programas de computador e circuitos lógicos. Ela modela uma máquina abstrata que está em um estado dentro de vários possíveis.

Elas são usadas em vários problemas computacionais, como parser de linguagens de computador, design de protocolos de comunicação, na criação de algoritmos de inteligência artificial, entre outros. Uma representação visual do que seria uma máquina de estados pode ser a seguinte:

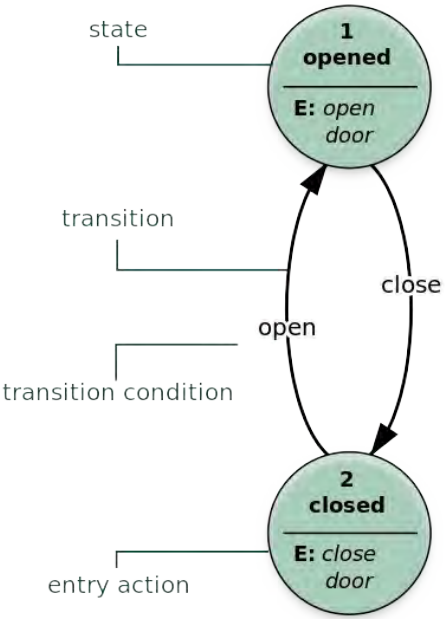

Figura 5.2: Tabela de decisão

Podemos utilizar DSLs para representar seus estados e transições, Para exemplificar o uso da máquina de estados, podemos criar uma simulação de vidas, em que pessoas passam entre vários estados de vida do nascimento até sua morte.

Esta simulação poderia retornar o seguinte:

```
    Ayden Everett transitou de Live para Single com  16 ano(s)
    Nathan Rosales transitou de Live para Single com  17 ano(s)
    Ayden Everett transitou de Single para MakeOut com  17 ano(s)
    Blake Meyer transitou de Live para Single com  17 ano(s)
    Arianna Wilkins transitou de Live para Single com  17 ano(s)
    Cooper Holden transitou de Live para Single com  18 ano(s)
    Zoey Barber transitou de Live para Single com  18 ano(s)
    Nathan Rosales transitou de Single para MakeOut com  18 ano(s
)
    Taylor Aguilar transitou de Live para Single com  19 ano(s)
    Brooklyn Bell transitou de Live para Single com  19 ano(s)
    Gabriel Wilder transitou de Live para Single com  19 ano(s)
    Allison Ayala transitou de Live para Single com  19 ano(s)
    Blake Meyer transitou de Single para MakeOut com  20 ano(s)
    Zoey Barber transitou de Single para MakeOut com  21 ano(s)
    Kennedy Walker transitou de Live para Single com  21 ano(s)
    Ayden Everett transitou de MakeOut para Dating com  21 ano(s)
    Zoey Barber transitou de MakeOut para Single com  22 ano(s)
    Nathan Rosales transitou de MakeOut para Dating com  22 ano(s
)
    Blake Meyer transitou de MakeOut para Dating com  23 ano(s)
    Nathan Rosales transitou de Dating para Fiance com  25 ano(s)
    Ayden Everett transitou de Dating para Fiance com  27 ano(s)
    Blake Meyer transitou de Dating para Fiance com  27 ano(s)
    Nathan Rosales transitou de Fiance para Married com  28 ano(s
)
    Blake Meyer transitou de Fiance para Married com  28 ano(s)
    Nathan Rosales transitou de Married para Widower com  47 ano(
s)
    Gabriel Wilder transitou de Single para Died com  77 ano(s)
    Nathan Rosales transitou de Widower para Died com  78 ano(s)
    Cooper Holden transitou de Single para Died com  79 ano(s)
    Ayden Everett transitou de Fiance para Died com  79 ano(s)
    Blake Meyer transitou de Married para Died com  81 ano(s)
    Allison Ayala transitou de Single para Died com  92 ano(s)
```

```
    Brooklyn Bell transitou de Single para Died com  93 ano(s)
    Taylor Aguilar transitou de Single para Died com  95 ano(s)
    Kennedy Walker transitou de Single para Died com  95 ano(s)
```

Esta simulação demonstra um conjunto de pessoas hipotéticas e seus relacionamentos pela vida até seu falecimento. Para implementar algo assim, primeiro criaríamos uma classe para representar as pessoas, como:

```
class Person(age: Int,
                  name: String,
                  sex: Sex,
                  state: State)...
```

Dentro desta classe, temos uma máquina de estados para representar as transições entre os estados:

```
  ...
  def machine(): Machine = {
    def transite(start: State, end: State, dist: NormalDistributi
on)
      :() => (Boolean, State) = () =>
      {
      val probability = dist.probability(age - 1, age)
      if (Math.random() < probability) {
        (true, end)
      } else {
        (false, start)
      }
    }

    Machine(
      //start state
      state,
      //all states can go to died
      Transition(Live, transite(Live, Died, deadDistribution(sex)
)),
      Transition(Single, transite(Single, Died, deadDistribution(
sex))),
      Transition(MakeOut, transite(MakeOut, Died, deadDistributio
n(sex))),
      Transition(Dating, transite(Dating, Died, deadDistribution(
```

```
sex))),
      Transition(Fiance, transite(Fiance, Died, deadDistribution(
sex))),
      Transition(Married, transite(Married, Died, deadDistributio
n(sex))),
      Transition(Widower, transite(Widower, Died, deadDistributio
n(sex))),
      Transition(Live, transite(Live, Single, liveToSingleDistrib
ution())),
      Transition(Single, transite(Single, MakeOut, singleDistribu
tion())),
      Transition(MakeOut, transite(MakeOut, Single, makeOutToDati
ngDistribution())),
      Transition(MakeOut, transite(MakeOut, Dating, makeOutToDati
ngDistribution())),
      Transition(Dating, transite(Dating, Fiance, datingToFianceD
istribution())),
      Transition(Dating, transite(Dating, Single, singleDistribut
ion())),
      Transition(Fiance, transite(Fiance, Single, singleDistribut
ion())),
      Transition(Fiance, transite(Fiance, Married, fianceToMarrie
dDistribution())),
      Transition(Married, transite(Married, Single, singleDistrib
ution())),
      Transition(Married, transite(Married, Widower, marriedToWid
owerDistribution())),
      Transition(Widower, transite(Widower, Single, singleDistrib
ution())),
      Transition(Widower, transite(Widower, Died, singleDistribut
ion()))
    )
  }
```

Neste código, temos a descrição da máquina de estados, seu estado inicial, suas transições e as funções de transição dos estados. As definições dos estados são feitas desta forma:

```
...
object States {
  def Live = State("Live")
  def Single = State("Single")
  def MakeOut = State("MakeOut")
```

```
  def Dating = State("Dating")
  def Fiance = State("Fiance")
  def Married = State("Married")
  def Widower = State("Widower")
  def Died = State("Died")
}

case class State(name: String)
...
```

E a máquina de estados é construída assim:

```
...
case class Machine(state: State, transitions: Transition*) {
  def tick(): Machine = {

    val allFunctions = transitions.filter(t => t.startState == st
ate)

    allFunctions.foreach(transition => {
      val ret = transition.func()
      if (ret._1) {
        return Machine(ret._2, transitions: _*)
      }
    })
    //if not encounter any transition stay
    this
  }
}
...
```

As transições são definidas da seguinte forma:

```
...
case class Transition(startState: State, func: () => (Boolean, St
ate))
...
```

Este código detalha como utilizar máquina de estados em um exemplo simples, mas funcional. Você pode encontrar muitos casos de uso de máquinas de estados, e ela é uma ferramenta muito interessante para se trabalhar.

5.10 E AGORA?

Neste capítulo, aprendemos sobre várias construções diferentes para criação de DSLs. Aprendemos sobre símbolos, parâmetros nomeados, anotações, closures, implícita, recepção dinâmica, testes, migrações e modelos alternativos, completando o grupo de técnicas apresentadas no livro.

Este aprendizado vai enriquecer a qualidade final das DSLs criadas pelo desenvolvedor, e fazê-lo encontrar novos caminhos para expressão. Isso finaliza nosso aprendizado.

Para completar o conhecimento do leitor, foi adicionado um apêndice sobre a linguagem Scala para sanar algumas dúvidas que você possa ter sobre o uso da linguagem.

Capítulo 6
CONCLUSÃO

E chegamos ao final do livro! Vimos um conjunto grande de técnicas e conceitos. Iniciamos com um exemplo simples, mas muito interessante, para nos familiarizarmos sobre como as DSLs são desenvolvidas.

Posteriormente, no capítulo *Adentrando nas DSLs*, nos aprofundamos neste estudo mostrando conceitos de domínio, os tipos de modelos e as motivações de uso. Também apresentamos cada uma das técnicas para termos uma visão global do conteúdo.

No capítulo *Encadeamento de métodos e Composite*, vimos encadeamentos de métodos e padrões de construção. Também aprendemos a modelar um domínio, e depois utilizar estas técnicas. No capítulo *Sequência de funções e funções aninhadas*, aprendemos sobre sequência de função e funções aninhadas, com mais exemplos práticos e um exemplo do mundo real. Já no capítulo *Outras técnicas*, vimos um conjunto de técnicas gerais para construção de DSLs, como símbolos, parâmetros nomeados, closures, extensão de literais, entre outros.

Tenho certeza de que o leitor ganhou algum conhecimento e uma nova visão sobre como modelar e resolver problemas de domínio de uma forma mais natural. Se o leitor não tinha

experiência com a linguagem Scala, tenho certeza também de que ganhou mais conhecimento nesta interessante linguagem.

Acredito que se Scala não se tornar a próxima grande linguagem da JVM, com certeza influenciou positivamente com seus conceitos o futuro das linguagens sobre a JVM.

Com este livro, acredito que o leitor, na próxima vez que demonstrar seu código para alguém que não tenha experiência em programação, possa tornar esta experiência incrivelmente natural para ambos, com o cliente escrevendo o que deseja e os desenvolvedores tornando isso possível exatamente como descrito.

A internet é um mar de conhecimento e o aprendizado nunca termina em um só conteúdo. Partindo deste livro, o leitor pode encontrar muitas informações em vários lugares diferentes sobre diversos assuntos.

Para dúvidas relacionadas ao livro, temos sempre o Fórum da Casa do Código (http://forum.casadocodigo.com.br). Lá eu e a comunidade poderemos ajudá-lo de uma forma mais direta.

Para informações mais gerais, recomendo o conhecido Stack Overflow (http://stackoverflow.com.br/) que, desde que surgiu, se tornou a referência para dúvidas sobre tecnologia. Recomendo também, se o leitor tiver conhecimento em alguma tecnologia específica, se cadastrar no site e ajudar a comunidade. Isso é muito mais do que ajudar, é uma forma de competição e aprendizagem muito saudável.

O uso de DSLs externas tem uma relação muito íntima com linguagens formais e compiladores. Se o leitor quiser saber mais sobre estes assuntos, recomendo fortemente uma leitura mais

profunda do tema com o livro *Compiladores: Princípios, Técnicas e ferramentas*, de Alfred Aho (editora Addison-Wesley). Ele é um clássico sobre o tema de compiladores.

Um grande abraço a todos. Espero encontrá-los novamente em um próximo livro.

Capítulo 7

APÊNDICE — SCALA

Em todos os capítulos, aprendemos várias técnicas de implementação de DSLs utilizando as linguagens Java e Scala. A linguagem Java é amplamente conhecida para desenvolvimento, mas a Scala não é tão conhecida, apesar de mais poderosa.

Para resolver este problema, neste apêndice vamos aprender algumas funcionalidades dessa linguagem, focando nos exemplos de DSLs apresentados no livro. O core da linguagem é razoavelmente simples e, sendo um programador Java com alguma experiência, a transição entre uma linguagem a outra é bem transparente.

Linguagem escalável

O nome da linguagem Scala vem de Linguagem Escalável (*Scalable Language*). Dizer que uma linguagem é escalável significa que ela pode crescer, mas crescer pode ter muitos significados diferentes.

Um dos significados é permitir que a linguagem cresça para se moldar a novas linguagens, com a adição de novas bibliotecas. O outro significado é que a linguagem pode ser utilizada tanto para pequenos sistemas quanto para sistemas maiores, permitindo um

crescimento suave do menor para o maior. Esta característica torna bem simples a construção de DSLs em cima da linguagem.

Scala executa sobre a plataforma Java e funciona de forma transparente com as bibliotecas existentes. Você pode também mesclar códigos Java e Scala de forma transparente, ou seja, sem que você tenha de se preocupar muito com isso. Os programas são bem concisos e tendem a ser bem curtos, mais legíveis e menos repetitivos.

Ela adota grande parte da sintaxe do Java e do C#. Além disso, também adota os tipos básicos do Java, sua biblioteca de classes e seu modelo de execução.

A linguagem é estaticamente tipada e combina os paradigmas de programação orientada a objetos e funcional. Paradigmas de programação são classificações de linguagens de programação segundo seu estilo e funcionalidade. No paradigma orientado a objetos, temos os dados e os métodos para manipulação deles mantidos em um lugar unitário, conhecido como objeto. Já no funcional, temos as funções — blocos de código que têm a intenção de se comportar como funções matemáticas.

Existem muitos comparativos entre estes dois paradigmas, e cada um deles tem pontos fortes e fracos. Uma definição interessante, mas simplificada, destes assuntos é a de Robert C. Martin (2012), que define os paradigmas como restrições de possibilidades dentro das linguagens.

O paradigma orientado a objetos tem sido muito bem-sucedido desde sua criação, e atualmente é suportado pela grande maioria das linguagens de programação. Já o paradigma funcional

tem ganhado grande destaque nestes últimos anos, e muitas ideias criadas em linguagens funcionais foram migradas para linguagens orientadas a objetos. Um exemplo a se destacar são lambdas.

Enquanto Orientação a Objetos é uma restrição sobre como usar os ponteiros, as linguagens funcionais têm a restrição sobre como os valores são modificados. Scala combina os dois paradigmas de forma elegante e expande, de forma escalável, as capacidades dos programadores principalmente acostumados com linguagens apenas orientadas a objetos, como Java.

Instalação

Atualmente, existem três maneiras diferentes de utilizar a linguagem Scala. A primeira delas é instalando a linguagem diretamente em seu sistema. A segunda é usando o *Lightbend Activator*, uma ferramenta web de linha de comando que ajuda na criação de aplicações. A terceira opção é a instalação de uma IDE pré-configurada para o Scala.

Inicialmente, vamos instalar o Scala por linha de comando, pois é a forma mais básica de instalação e exercita melhor o aprendizado da linguagem. Mas se o leitor já tiver familiaridade com as IDEs, não afetará em nada o resultado final do código.

> Você pode ver todas as possibilidades de executar o Scala e baixar as ferramentas seguindo o link: http://www.scala-lang.org/download/.

Instalando o SDK

Primeiramente, entre no site do download da linguagem Scala, em: http://www.scala-lang.org/download/.

Na página, baixe os binários para seu sistema. No momento em que o livro foi escrito, a última versão era a do link http://downloads.typesafe.com/scala/2.11.8/scala-2.11.8.tgz. Mas você pode baixar uma mais atualizada, sem problemas.

Depois de baixar o SDK, descompacte-o em uma pasta de sua preferência e configure o PATH do seu ambiente para apontar para a pasta `bin` do SDK. Por exemplo, nossas variáveis de ambiente ficariam da seguinte maneira, dependendo do sistema:

```
Environment        Variable                Value (example)
Unix             $SCALA_HOME               /usr/local/share/scala
$PATH             $PATH:$SCALA_HOME/bin

Windows            %SCALA_HOME%            c:\Progra~1\Scala
%PATH%            %PATH%;%SCALA_HOME%\bin
```

Depois de instalado o SDK, podemos executar o interpretador REPL Scala:

```
> scala
```

O REPL é simplesmente um shell de uma determinada linguagem em que podemos testar pequenos trechos de código e ver seu resultado em loop. Esta sigla vem de **Read-Eval-Print-Loop**, ou em tradução livre **Leia-Faça-Mostre-Continue**

E o resultado será próximo ao seguinte:

```
Welcome to Scala version 2.11.8 (OpenJDK 64-Bit Server VM, Java 1
.8.0_91).
Type in expressions to have them evaluated.
```

```
Type :help for more information.

scala>
```

Para testar, vamos fazer uma soma. Digite o seguinte na linha de comando:

```
scala> 1+1
res0: Int = 2

scala>
```

Como esperado, o resultado é 2. Neste caso, o tipo do dado retornado é `Int` . Ele também atribui automaticamente a resposta a uma variável, neste caso `res0` .

Podemos utilizar este valor retornado da seguinte maneira:

```
scala> 1+1
res0: Int = 2

scala> 1+res0
res1: Int = 3
```

Existem várias formas de executar um código Scala. Esta que acabamos de executar é a que chamamos de REPL. Um REPL é um programa que vai ler suas entradas, interpretar e produzir uma saída como resultado. O REPL do Scala é bem poderoso, e podemos fazer praticamente tudo o que fazemos normalmente na linguagem.

A maioria dos programadores Java não está acostumada com REPLs, e interagir com a linguagem assim pode parecer meio estranho. Entretanto, é uma ferramenta ótima para testar pequenos trechos de código.

Além do REPL, existem outras formas de executar código

Scala. Por exemplo, você pode ter o arquivo `OlaScala.scala` , como o seguinte:

```
println("Olá Casa do Scala")
```

E podemos executar este código da seguinte maneira no shell:

```
shell> scala OlaScala.scala
```

E teremos o seguinte resultado, como o esperado:

```
shell> scala OlaScala.scala
Olá Casa do Scala
```

Você pode ainda executar diretamente pela JVM, do seguinte modo:

```
shell > java -cp $SCALA_HOME/lib/scala-library.jar;. OlaScala
```

O `$SCALA_HOME` é o diretório de instalação do Scala.

Além destas formas, podemos também executar dentro de uma IDE. Mas para aproveitar melhor o aprendizado, vamos utilizar diretamente o REPL para entender melhor como a linguagem funciona.

Apesar disso, não recomendo o uso para grandes quantidades de código apenas pelo REPL, pois você acaba perdendo várias funcionalidades excelentes das IDEs, como sintaxe colorida e validações de sintaxe automáticas.

Depois que dominarmos a sintaxe básica da linguagem, o uso da IDEs só soma no aprendizado. Possivelmente, na versão 9 do Java, teremos um shell na linguagem que vai tornar mais comum o uso de REPLs.

7.1 FUNCIONALIDADES

A partir de agora, vamos descrever várias funcionalidades da linguagem e como ela funciona. Isso ajudará na criação de suas DSLs futuramente. Você não precisa seguir cada uma das funcionalidades passo a passo se não quiser, mas elas são um guia precioso para utilização e ajudam na construção da linguagem.

Valores e variáveis

A primeira coisa que precisamos saber antes de programar em Scala são duas palavrinhas simples: `val` e `var`. Enquanto `val` define valores, `var` define variáveis. Valores são imutáveis; variáveis são mutáveis.

Por exemplo, para definir um valor fazemos:

```
scala> val a = 1
a: Int = 1
```

Se tentarmos modificar o valor, teremos:

```
scala> a = 2
<console>:8: error: reassignment to val
       a = 2
```

Já se definirmos uma variável, podemos trocar o seu valor:

```
scala> var b = 1
b: Int = 1

scala> b = 2
b: Int = 2
```

Inferência de tipos

Scala possui mecanismos que inferem os tipos que queremos

utilizar. Isso poupa tempo, pois não precisamos escrever diretamente, e deixa a sintaxe mais limpa, pois o código é menor. Por exemplo, para definir o valor de uma `String` :

```
scala> val a = "texto"
a:String = texto
```

Não precisamos dizer que a variável vai ser do tipo String, pois o compilador infere o tipo automaticamente. Mas se quisermos, podemos definir explicitamente o tipo da variável, por exemplo:

```
scala> var a:String = "texto"
a: String = texto
```

Se tentarmos colocar outro tipo na variável, teremos o seguinte erro:

```
scala> a = 1
<console>:8: error: type mismatch;
 found   : Int(1)
 required: String
       a = 1
           ^
```

Como no Java, existem outros tipos de variáveis e você pode utilizar os mesmos tipos da linguagem Java, da mesma forma que utilizei anteriormente para os tipos `String` e `Int` .

Métodos

No Java, a forma padrão para se criar um método é o seguinte:

```
public class  Main {

    public int soma(int x, int y) {
        return x+y;
    }
}
```

Para criar métodos no Scala, é muito simples. O mesmo método de soma do Java seria escrito da seguinte maneira no Scala.

```
scala> def soma(x:Int, y:Int):Int ={
     | return x+y
     | }
soma: (x: Int, y: Int)Int
```

A barra `|` apenas indica que estamos em uma nova linha, ela não faz parte do código final. Nos exemplos posteriores, removerei propositalmente a barra para simplificar o código. Definido o método, já podemos usá-lo:

```
scala> soma(1,2)
res1: Int = 3
```

Utilizando a inferência de tipos, podemos simplificar nosso método `soma` da seguinte maneira:

```
scala> def soma(x:Int, y:Int) ={
      x+y
      }
soma: (x: Int, y: Int)Int
```

Perceba que o compilador inferiu qual era o tipo do retorno e também que a última linha era o retorno do método. Logo podemos remover a palavra reservada `return` para o retorno do método em Scala.

Imperativo e funcional

Normalmente, é utilizado em linguagem como Java o estilo de programação imperativo. Neste paradigma, um comando é executado por vez, iterando com loops e objetos mutáveis são compartilhados entre as funções.

Já no estilo funcional, objetos imutáveis são transferidos entre

o código. Em linguagens funcionais, as funções são de primeira classe, ou seja, podem ser tratadas como valores, passadas, manipuladas e retornadas por outras funções.

Por exemplo, tendo um array de palavras:

```
scala> val palavras = "banana abacate abacaxi maça".split(" "
)
palavras: Array[String] = Array(banana, abacate, abacaxi, maç
a)
```

Para iterar e imprimir os elementos de forma iterativa, teríamos o seguinte:

```
scala> var i = 0
i: Int = 0

scala> while(i < palavras.length){
         println(palavras(i))
         i = i + 1
       }
banana
abacate
abacaxi
maça
```

Em uma forma mais funcional, teríamos o seguinte:

```
scala> palavras.foreach(palavra => println(palavra))
banana
abacate
abacaxi
maça
```

Desta forma chamamos o método `foreach` passado uma função para este método. A sintaxe com `=>` é uma sintaxe inline, para definir funções e tornar o código mais enxuto e simplificado. Ou seja, uma sintaxe mais curta para descrever algo.

Uma forma de reconhecer um estilo mais funcional dentro do

Scala é o uso quase inexistente de variáveis (`var`), mas cabe ao programador escolher a melhor ferramenta para cada propósito.

Operações

Todas as operações em Scala são feitas com chamadas de métodos, por exemplo:

```
scala> 1 + 1
res0: Int = 2
```

Esse código é transformado na chamada de método:

```
scala> 1.+(1)
res1: Int = 2
```

Scala não tem sobrecarga de operadores, já que não tem operadores no sentido tradicional. No lugar dos operadores, você pode criar métodos com caracteres especiais com `+` , `-` e `*` . Logo, uma operação como `1+2` será interpretada como uma chamada de métodos assim: `1.+(2)` .

Outro conceito importado do Scala é que, de forma semelhante aos métodos, se os parênteses forem aplicados a uma variável, teremos a chamada do método `apply(...)` . Por exemplo, se tivermos o uso dos parênteses:

```
scala> "texto"(0)
res21: Char = t
```

Isso será transformado no seguinte:

```
scala> "texto".apply(0)
res22: Char = t
```

Este princípio é aplicado para todos os objetos que definem o método `apply` . De forma similar, objetos que definem o método

`apply` normalmente são instanciados da seguinte maneira:

```
val palavras = Array("banana", "abacate", "abacaxi", "maça")
```

Isso é semelhante a chamar o método `apply` diretamente, como:

```
val palavras = Array.apply("banana", "abacate", "abacaxi", "m
aça")
```

Classes

Classes em Scala são definidas de forma similar ao Java, da seguinte maneira:

```
scala> class PrimeiraClasse{}
defined class PrimeiraClasse
```

E a instanciação de classes é similar à do Java:

```
scala> new PrimeiraClasse
res25: PrimeiraClasse = PrimeiraClasse@3cdc3901
```

Para criar membros de classe, usamos da mesma maneira as palavras reservadas `val` e `var`. E para criar métodos, utilizamos a palavra reservada `def`. Por exemplo, poderíamos criar a seguinte classe com campos:

```
scala> class SegundaClasse {
         val numero = 1
         var delta = 0
         def diff(){
           numero + delta
         }
       }
defined class SegundaClasse

scala> val sc = new SegundaClasse
sc: SegundaClasse = SegundaClasse@284bb560
```

```
scala> sc.diff()
res1: Int = 1

scala> sc.delta = 42
sc.delta: Int = 42

scala> sc.diff()
res1: Int = 1

scala> sc.diff
res2: Int = 43
```

No método `diff`, não colocamos o retorno, pois o compilador infere o retorno pela última linha de código do método.

Podemos definir um método mais explicitamente, da seguinte maneira:

```
scala> def diff(a:Int,b:Int):Int = {
         a-b
       }
diff: (a: Int, b: Int)Int
```

Este método recebe dois parâmetros `a` e `b`, e retorna um tipo `Int`. Para executar o método, podemos chamar diretamente em:

```
scala> def diff(a:Int,b:Int):Int = {
         a-b
       }
diff: (a: Int, b: Int)Int
```

Agora podemos executá-lo da seguinte maneira:

```
scala> diff(1,2)
res1: Int = -1
```

Objetos singletons

Em Scala, diferentemente do Java, não existem membros de

classe estáticos. Isto é, não temos uma classe com métodos que podem ser acessados globalmente por todo o código. No lugar disso, Scala possui *objetos singletons*.

A definição de um objeto singleton é a mesma para classes. A única diferença é que, no lugar da palavra-chave `class`, utilizamos a palavra-chave `object`. Ou seja, no lugar de criar uma classe com métodos estáticos, como fazemos no Java, no Scala criamos um objeto estático singleton.

Por exemplo, podemos ter o seguinte:

```
scala> :past
// Entering paste mode (ctrl-D to finish)

class Teste{}
object Teste{
    def calcular() = {
        1 + 1
    }
}

// Exiting paste mode, now interpreting.

defined class Teste
defined object Teste
```

Neste exemplo, podemos ver a criação da classe `Teste` e do objeto Singleton `Teste`.

> Quando quisermos inserir dentro do REPL múltiplos comandos, usamos o comando `:past`. De depois de digitar todos que você achar necessário, utilize o atalho `ctrl-D` para finalizar.

Quando temos uma classe com o mesmo nome do objeto Singleton, chamamos este objeto de `companion object`. A classe e o `companion object` têm de estar definidos no mesmo arquivo, e eles podem acessar membros privados entre eles.

Para acessar os métodos estáticos, fazemos da mesma forma que em Java, da seguinte maneira:

```
scala> Teste.calcular()
res0: Int = 2
```

Símbolos

Símbolos literais são escritos como `'simbolo`, em que `simbolo` pode ser um identificador alfanumérico. Por exemplo, para se criar um símbolo, fazemos o seguinte:

```
scala> 'meu_simbolo
res1: Symbol = 'meu_simbolo
```

Ou ainda, podemos fazer da seguinte maneira:

```
scala> Symbol("meu_simbolo")
res1: Symbol = 'meu_simbolo
```

Estes literais são mapeados para instâncias da classe `scala.Symbol`. Símbolos são usados tipicamente em situações nas quais se deseja um identificador.

Operadores como métodos

Como visto anteriormente em Scala, operadores são métodos, mas é possível aplicar este conceito também a métodos. Ou seja, é possível fazer com que métodos se pareçam com operadores.

Podemos chamar o método `indexOf` de uma string, por

exemplo:

```
scala> "123".indexOf('2')
res2: Int = 1
```

Ou pode fazer de forma similar à seguinte chamada:

```
scala> "123" indexOf '2'
res4: Int = 1
```

Você pode utilizar esta notação para qualquer método.

Implicits

Conversões implícitas, ou *implicits*, são um poderoso recurso da linguagem Scala. Este recurso é usado para realizar conversões automáticas entre diferentes tipos de objetos. Para exemplificar, digamos que temos o seguinte código:

```
scala> class Pizza
defined class Pizza

scala> class Pedaco(quantidade:Int)
defined class Pedaco
```

Definimos uma pizza e pedaços de pizza. Podemos somar pedaços de pizza com outros pedaços de pizza, como mostrado no seguinte código:

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

class Pedaco(val quantidade:Int){
   def +(outro:Pedaco):Pedaco = {
    return new Pedaco(quantidade+outro.quantidade)
   }
}

scala> val p1 = new Pedaco(1)
p1: Pedaco = Pedaco@9e2d2
```

```
scala> val p2 = new Pedaco(1)
p2: Pedaco = Pedaco@467c1624

scala> p1 + p2
res0: Pedaco = Pedaco@1f2305e5

scala> (p1 + p2).quantidade
res1: Int = 2
```

Mas se quisermos somar uma pizza com se fossem pedaços de pizza, como faríamos? Utilizando conversões implícitas, claro.

Definindo que uma pizza é normalmente cortada em 8 pedaços, poderíamos fazer o seguinte:

```
scala> implicit def pizzaToPedaco(p:Pizza)={ new Pedaco(8) }
warning: there was one feature warning; re-run with -feature
for details
pizzaToPedaco: (p: Pizza)Pedaco
```

Não precisa ligar para o warning; ele só indica que, para este tipo de uso de `implicit`, precisamos fazer uma importação. Para resolver este *warning*, utilize a importação como mostrado a seguir:

```
scala> import scala.language.implicitConversions
import scala.language.implicitConversions

scala> implicit def pizzaToPedaco(p:Pizza)={ new Pedaco(8) }
pizzaToPedaco: (p: Pizza)Pedaco
```

Agora conseguimos somar pedaços com pizzas completas, como mostra o código a seguir:

```
scala> val p1 = new Pedaco(1)
p1: Pedaco = Pedaco@352d8328

scala> val p2 = new Pedaco(1)
p2: Pedaco = Pedaco@3217a2f5
```

```
scala> val pz = new Pizza
pz: Pizza = Pizza@870769

scala> (p1 + p2 + pz).quantidade
res1: Int = 10
```

Implicit classes

Algumas vezes, precisamos adicionar métodos em classes preexistentes. Para isso, podemos usar `Implicit classes` . Por exemplo, adicionaremos um método especial dentro da classe `String` , da seguinte maneira:

```
scala> implicit class StringIsSmile(str:String){
          def isSmile(){
            str.equals(":)")
          }
       }
defined class StringIsSmile

scala> ":(".isSmile()
res0: Boolean = false

scala> ":)".isSmile()
res1: Boolean = true
```

Funções e closures

Em Java, temos funções definidas dentro dos objetos, também chamadas de métodos. Em Scala, temos mais possibilidades, como funções aninhadas com funções, funções literais e valores de funções.

A forma mais simples de definir funções é dentro de classes, como já foi mostrado anteriormente. Podemos também definir funções dentro de funções:

```
scala> :past
```

```
// Entering paste mode (ctrl-D to finish)

def retorna2() = {
   def retorna1() = 1
   retorna1() + retorna1()
}

// Exiting paste mode, now interpreting.

retorna2: ()Int

scala> retorna2()
res0: Int = 2
```

Funções também podem ser representadas como valores. Veja um exemplo:

```
scala> //Criando uma função simples

scala> def soma(a:Int,b:Int) = a+b
soma: (a: Int, b: Int)Int

scala> soma(1,1)
res1: Int = 2
```

É possível também definir a função de uma forma literal, por exemplo:

```
scala> var soma = (a:Int, b:Int) => a + b
soma: (Int, Int) => Int = <function2>

scala> soma
res0: (Int, Int) => Int = <function2>

scala> soma(1,1)
res1: Int = 2
```

A notação `=>` converte da parte esquerda, no caso `(a:Int, b:Int)`, para uma saída à direita `a+b`. Quando estamos atribuindo um valor de função internamente, o compilador vai transformar esta função em uma classe `FunctionN`.

Podemos definir também funções com mais declarações, fazendo o seguinte:

```scala
    scala> var somaPrint = (a:Int, b:Int) => {
              println("somando...")
              a + b
          }
    somaPrint: (Int, Int) => Int = <function2>

    scala> somaPrint(1,1)
    somando...
    res0: Int = 2
```

A biblioteca padrão do Scala utiliza muito estes recursos de funções literais, também conhecidos como funções anônimas. Por exemplo, na iteração de coleções:

```
    scala> val numbers = List(1,2,3,4)
    numbers: List[Int] = List(1, 2, 3, 4)

    scala> numbers.foreach((n:Int) => println("encontrei o "+n))
    encontrei o 1
    encontrei o 2
    encontrei o 3
    encontrei o 4
```

Até agora criamos funções que têm todos os seus parâmetros passados diretamente na assinatura. Porém, é possível fazer a captura de variáveis de fora do escopo da função. Veja um exemplo:

```scala
    scala> var tamanho = 10
    tamanho: Int = 10

    scala> val diff = (x:Int) => tamanho - x
    diff: Int => Int = <function1>

    scala> diff(1)
```

```
    res10: Int = 9

    scala> diff(2)
    res11: Int = 8

    scala> diff(5)
    res12: Int = 5
```

A função `diff` está fazendo a captura da variável `tamanho` para calcular o resultado final. Esta captura é feita em tempo de execução, e não de compilação. Por exemplo, podemos modificar o valor da variável `tamanho` e obter uma resposta diferente:

```
    scala> tamanho = 22
    tamanho: Int = 22

    scala> diff(2)
    res14: Int = 20
```

Este tipo de função, que faz a captura de variáveis externas, é conhecida como *closure*.

Repetição de parâmetros

Scala, assim como Java, permite a repetição de parâmetros. Por exemplo, podemos ter a seguinte função:

```
    scala> def printAll(args:String*) = args.foreach(a=>println(a
))
    printAll: (args: String*)Unit

    scala> printAll("1","2","3","4")
    1
    2
    3
    4
```

Scala também permite parâmetros nomeados, como:

```
scala> def distancia(x1:Int,x2:Int) = x2-x1
distancia: (x1: Int, x2: Int)Int

scala> distancia(x2 = 1, x1 = 0)
res1: Int = 1
```

Traits

Traits são muito usados em Scala. Eles encapsulam definições de métodos e campos. Diferentemente das classes, é possível fazer a extensão de qualquer número de Traits que se desejar.

A definição de Traits é similar à de classes. Por exemplo, podemos ter um Trait para `logs` como a seguinte

```
scala> trait Logger{
      def log(str:String) = println(str)
      }
defined trait Logger

scala> class SuperComplexAlgorithm extends Logger {
          def start() = {
            log("Start HiperProcessing")
          }
      }
defined class SuperComplexAlgorithm

scala> new SuperComplexAlgorithm().start()
Start HiperProcessing
```

Case classes e pattern matching

Scala tem algumas facilidades para criação de classes. Implementando uma classe `case class`, temos a implementação automática dos métodos `toString`, `hashCode` e `equals`. Além disso, temos a implementação automática de métodos de construção também. Podemos ver isso nos seguintes exemplos:

```
scala> case class Pessoa(name:String)
```

```
defined class Pessoa

scala> val pessoa = Pessoa("John") // método de construção
pessoa: Pessoa = Pessoa(John)

scala> println(pessoa) // toString
Pessoa(John)

scala> println(pessoa.hashCode)
-840087766

scala> pessoa.equals(Pessoa("John2"))
res6: Boolean = false

scala> pessoa.equals(Pessoa("John"))
res7: Boolean = true
```

Definir `case classes` tem a desvantagem de tornar as classes maiores, por ter todos estes métodos utilitários. Apesar disso, temos a vantagem deste tipo de construção que é suportar *pattern matching*. Para compreender melhor o conceito de *pattern matching*, podemos criar algo como o seguinte:

```
scala> trait HtmlElem
defined trait HtmlElem

scala> case class Text(content:String) extends HtmlElem
defined class Text

scala> case class Tag(name:String, subtags:HtmlElem*) extends
HtmlElem
defined class Tag
```

A trait `HtmlElem` representa elementos dentro de um documento HTML. `Text` são os possíveis textos dentro do documento e `Tag` são as tags. Logo, poderíamos ter a seguinte estrutura representando um HTML:

```
scala> val doc = Tag("html",Tag("body",Tag("h1",Text("Hello")
)))
doc: Tag = Tag(html,WrappedArray(Tag(body,WrappedArray(Tag(h1
```

```
WrappedArray(Text(Hello)))))))
```

Vamos imaginar que agora teríamos de pegar o texto dentro do corpo do documento. Em uma linguagem que não suporta pattern matching, poderíamos ter de fazer o seguinte:

```
    scala> :paste
    :paste
    // Entering paste mode (ctrl-D to finish)

    def forEachTag(elems:Seq[HtmlElem], name:String, f:(Tag)=>Uni
) = {
        elems.foreach(e=>{
            if(e.isInstanceOf[Tag]){
                if(e.asInstanceOf[Tag].name == name){
                    f(e.asInstanceOf[Tag])
                }
            }
        })
    }

    def forEachText(elems:Seq[HtmlElem], f:(String)=>Unit) = {
        elems.foreach(e=>{
            if(e.isInstanceOf[Text]){
                f(e.asInstanceOf[Text].content)
            }
        })
    }

    forEachTag(doc.subtags,"body", body =>{
        forEachTag(body.subtags,"h1", h1 =>{
            forEachText(h1.subtags, text=>{
                println(text)
            })
        })
    })

    // Exiting paste mode, now interpreting.

    Hello
    forEachTag: (elems: Seq[HtmlElem], name: String, f: Tag => Un
it)Unit
    forEachText: (elems: Seq[HtmlElem], f: String => Unit)Unit
```

Temos um código que navega pela estrutura e retorna o resultado se encontrar o padrão esperado. Ele é um pouco complicado e foi simplificado bastante para ficar mais compacto. Agora, em uma linguagem que suporta `Pattern matching`, teríamos algo como o seguinte:

```
    doc match {
        case Tag("html",Tag("body",Tag("h1",text:Text))) => print
(text.content)
        case _ => print("Não encontrado")
    }
```

Como você pode ver, esta representação é muitíssimo mais simples que a anterior e é utilizada em vários lugares dentro da biblioteca da linguagem.

Dynamics

Apesar de a linguagem Scala ser estaticamente tipada, algumas vezes podemos aproveitar algumas características das linguagens dinamicamente tipadas. Por exemplo, utilizando Dynamics, poderíamos ter o seguinte:

```
scala> import scala.language.dynamics
import scala.language.dynamics

scala> class Test extends Dynamic{
         def applyDynamic(name:String)(args:Any*)={
             println(name)
         }
       }
defined class Test

scala> new Test().mostraNomeDoMetodo()
mostraNomeDoMetodo
```

Com Dynamics, conseguimos simular a chamada dinâmica de métodos dentro de linguagens estáticas.

7.2 E AGORA?

Neste apêndice, aprendemos o essencial sobre a linguagem Scala. Este conhecimento é muito importante para construirmos DSLs. Espero que, além das DSLs, esta parte do livro sirva como uma introdução a conceitos das novas linguagens dentro da `JVM`, como Groovy, Kotlin, Ceylon, entre outras. Estes conhecimentos só têm a agregar ao desenvolvimento do profissional.

Capítulo 8

REFERÊNCIAS BIBLIOGRÁFICAS

CHAPIEWSKI, Guilherme. *Fluent Mail API.* 2008. Disponível em: https://github.com/guilhermechapiewski/fluent-mail-api.

EVANS, Eric. *Domain-Driven Design.* Alta Books, 2011.

FOWLER, Martin. DSL: Linguagens Específicas de Domínio. Bookman, 2013.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Padrões de Projeto - Soluções de Software Orientado a Objetos.* Addison-Wesley, 1994.

MARTIN, Robert C. *Three Paradigms.* Dez. 2012. Disponível em: http://blog.8thlight.com/uncle-bob/2012/12/19/Three-Paradigms.html.

ODERSKY, Martin; SPOON, Lex; VENNERS, Bill. *Programming in Scala.* 3. ed. 2016. Disponível em: http://www.artima.com/shop/programming_in_scala_2ed.