

Edição 47



JAVA 8 NA PRÁTICA

Desenvolva uma aplicação desktop



ISSN 2179625-4



DESENVOLVA PARA CORPORAÇÕES OU PARA O DIA A DIA COM JAVA

FORMAÇÃO DESENVOLVEDOR JAVA

DOMINE O DESENVOLVIMENTO DE APLICAÇÕES ORIENTADAS A OBJETO COM **UML** E APRENDA TÓPICOS AVANÇADOS DE PROGRAMAÇÃO COMO O USO DE **SERVLETS, JSP, FRAMEWORKS WEB DENTRO DO PADRÃO MVC.**

➤ A linguagem Java está presente em complexos sistemas como em aplicativos mobile e desktop, servindo a milhares de pessoas. Inicie sua carreira e certifique-se no instituto que valoriza seu currículo. Faça Infnet!

CERTIFICAÇÕES:

➤ Oracle Certified Professional, Java SE 6 Programmer

➤ Oracle Certified Professional, Java EE 5 Web Component Developer



Sumário

Conteúdo sobre Boas Práticas

06 – Anti-padrões de Projeto: o que são, como identificar e evitar

[Alessandro Jatobá]

Artigo no estilo Solução Completa

15 – Criando aplicações Desktop em Java

[Carlos Alberto Silva e Lucas de Oliveira Pires]

Artigo no estilo Solução Completa

23 – Programação paralela em Java

[Renato Alexandre Justo e Fabrício Martins Lopes]

CURSOS ONLINE

A Revista Java Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.



CONHEÇA ALGUNS DOS CURSOS:

- Curso de noSQL (Redis) com Java
- Curso Básico de JDBC
- Java Básico: Aplicações Desktop
- JSF com Primefaces
- Conhecendo o Apache Struts

Para mais informações :

www.devmedia.com.br/curso/javamagazine

(21) 3382-5038



Edição 46 • 2014 • ISSN 2173625-4



Assine agora e tenha acesso a todo o conteúdo da DevMedia:
www.devmedia.com.br/mvp

EXPEDIENTE

Editor

Eduardo Spínola (eduspinola@gmail.com)

Consultor Técnico Davi Costa (davigc_08@hotmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa e Diagramação Romulo Araujo

Atendimento ao leitor

A DevMedia possui uma Central de Atendimento on-line, onde você pode tirar suas dúvidas sobre serviços, enviar críticas e sugestões e falar com um de nossos atendentes. Através da nossa central também é possível alterar dados cadastrais, consultar o status de assinaturas e conferir a data de envio de suas revistas. Acesse www.devmedia.com.br/central, ou se preferir entre em contato conosco através do telefone 21 3382-5038.

Publicidade

publicidade@devmedia.com.br – 21 3382-5038

Anúncios – Anunciando nas publicações e nos sites do Grupo DevMedia, você divulga sua marca ou produto para mais de 100 mil desenvolvedores de todo o Brasil, em mais de 200 cidades. Solicite nossos Media Kits, com detalhes sobre preços e formatos de anúncios.

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique à vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site Easy Java Magazine, entre em contato com o editor, informando o título e mini-resumo do tema que você gostaria de publicar:



EDUARDO OLIVEIRA SPÍNOLA

eduspinola.wordpress.com

@eduspinola / @Java_Magazine

CURSOS ONLINE

A Revista SQL Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

Feita para Desenvolvedores de Software e DBAs



CONHEÇA OS CURSOS MAIS RECENTES:

- **Cursos: Curso de noSQL (Redis) com Java**
- **Desenvolvimento para SQL Server com .NET**
- **Curso PostgreSQL - Treinamento de banco de dados (Curso Básico)**

Para mais informações :

<http://www.devmedia.com.br/cursos/banco-de-dados>
(21) 3382-5038



DEVMEDIA



CRIANDO APLICAÇÕES INOVADORAS QUE VÃO MUDAR O MUNDO?

DESENVOLVA COM AS FERRAMENTAS QUE OS MELHORES DESENVOLVEDORES
USAM: GIT, REDMINE, JENKINS, SONAR, MAVEN E MUITO MAIS.

A suite ToolsCloud é um conjunto integrado das melhores ferramentas para desenvolvimento de software do mercado.

Um ambiente pronto para usar e com suporte, que os desenvolvedores curtem e os gerentes se surpreendem.

- Aumenta a produtividade;
- Facilita a colaboração e comunicação;
- Maior visibilidade e acompanhamento;
- Melhor planejamento e mais agilidade.

Na suite ToolsCloud você tem a sua disposição:

- As ferramentas mais usadas, integradas e de uso imediato;
- Repositório de código (git e svn), integração contínua (Jenkins/Hudson), Gerência de pendências (Redmine), e mais Maven, Nexus, Sonar;
- Infraestrutura robusta, em ambiente de nuvem, suporte e segurança;
- Facilita práticas de agilidade e lean startups;
- Ambiente isolado, customizável e flexível;
- Colaboração garantida: wiki, fóruns de discussão, chat (IRC).

Acompanhe a Toolscloud:

 toolscloud@toolscloud.com



twitter.com/toolscloud



ToolsCloud
toolscloud.com

Anti-padrões: o que são, como identificar e evitar

Aprenda neste artigo como maus hábitos de programação produzem riscos significativos aos projetos de software

Embara seja uma prática bastante aceita na comunidade de desenvolvedores, o uso de padrões de projeto na engenharia de software não é isento de críticas. Muito pelo contrário, há uma forte corrente de desenvolvedores que defende a ideia de que o uso de determinados padrões arquiteturais é, na verdade, um fator limitante para o programador.

Algo curioso nessa corrente de pensamento é o fato dela ter ganhado força quase que imediatamente após a publicação do famoso livro da “Gang-of-Four” (GoF) de Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, intitulado “Design Patterns: Elements of Reusable Object-Oriented Software”, em 1994.

O que aconteceu foi que em 1995 Andrew Koenig, ex-programador e pesquisador em técnicas de programação, cunhou o termo “anti-padrões”, como uma resposta quase que imediata à publicação do famoso catálogo de padrões que prometia descrever soluções arquiteturais confiáveis para problemas comuns de programação.

Para Koenig, muitos (senão todos) os padrões catalogados, quando não aplicados de maneira consciente, poderiam não só se tornar inefetivos, mas também representar grandes riscos aos projetos de software. E Koenig não para por aí. Em 1998 ele publicou o livro “AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis” e estendeu o uso desse conceito para além da engenharia de software, em áreas como a sociologia, a administração e a gestão de projetos.

Nesse livro, Koenig define anti-padrões de software como sendo uma “prática comum em arquitetura de software, porém inefetiva e contra-produtiva para um problema recorrente, causando de forma geral mais problemas do que benefícios”. Vale ressaltar que, embora Koenig tenha listado e nomeado um conjunto de más práticas em seu livro, não há especificamente um catálogo de anti-padrões. Sua existência está normalmente relacionada a exageros dos desenvolvedores ao projetar

Fique por dentro

Muito se discute a respeito do benefício da adoção de reconhecidos padrões de projeto, no entanto, não se discute com a mesma intensidade como algumas práticas bastante recorrentes de desenvolvedores trazem enormes prejuízos aos projetos. Esse artigo é útil por demonstrar como essas práticas se formam e por que elas devem ser evitadas, estimulando mudanças em alguns hábitos recorrentes de programadores e analistas.

uma solução, ao escrever código ou ao aplicar um reconhecido padrão. Ainda assim, há tentativas de catalogar anti-padrões, algumas dando a essas más práticas nomes bastante pitorescos, como “fluxo de lava” ou “marcha da morte”.

Além disso, em alguns casos o simples uso de técnicas obsoletas da engenharia de software, como o modelo em cascata, é considerado anti-padrão. Para outros, o uso de qualquer padrão catalogado, por si só, é um anti-padrão. Apesar disso, no que diz respeito a anti-padrões de engenharia de software, as seguintes categorias são comumente aceitas:

- Anti-padrões de projeto de software;
- Anti-padrões de análise orientada a objetos;
- Anti-padrões de programação;
- Anti-padrões metodológicos.

Há ainda, embora não tão discutida, uma categoria para anti-padrões de gerência da configuração, onde podemos encontrar problemas relacionados à gestão de versões ou de empacotamento.

O tema da existência de anti-padrões tem sido bastante explorado, seja pela comunidade de desenvolvedores em geral ou especificamente pela comunidade de desenvolvedores Java. Uma publicação que pode ser considerada seminal sobre anti-padrões de arquitetura de software é o livro de 1998 “AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis”, de William Brown, Raphael Malveau, Skip McCormick, Tom Mowbray,

sugestivamente autointitulados de “Upstart Gang-of-Four” e considerados “evangelistas” dos anti-padrões. É um livro importante por ressaltar, além dos anti-padrões em si, as forças ou tentações que levam à recorrência em sua aplicação.

Especificamente para a comunidade Java, em 2003, o livro “J2EE Design patterns”, de William Crawford e Jonathan Kaplan, dedica um capítulo ao tema anti-padrões, listando alguns dos considerados mais presentes nas aplicações Java EE até aquele momento.

Em seu livro, Crawford e Kaplan destacam o quanto importante é estudar não apenas as boas práticas de arquitetura de software, mas também o que deu errado na aplicação dessas práticas. Esses autores se concentraram em anti-padrões que prejudicam a escalabilidade, que afetam a camada de apresentação das aplicações Java, mas, principalmente, no mau uso da especificação Enterprise JavaBeans (EJBs – ver **BOX 1**), que se popularizava à época.

BOX 1. Enterprise JavaBeans (EJB)

EJB é um modelo presente na especificação Java Enterprise Edition (Java EE) desde 1997 para a construção de componentes server-side que encapsulem lógica de negócios, sendo responsáveis por operações como persistência, integridade transacional, segurança etc., especialmente entre aplicações distribuídas.

Também é um livro importante por discutir, na mesma publicação, padrões GoF e anti-padrões, estimulando a reflexão do desenvolvedor. Complementando essa discussão, no mesmo ano foi lançado o livro “J2EE AntiPatterns”, de Bill Dudney, Stephen Asbury, Joseph K. Krozak e Kevin Wittkopf. Nesse livro, os autores, estabelecem, de certa forma, um catálogo de anti-padrões J2EE, que embora não formalmente categorizados, apresentam alguma classificação, como anti-padrões de escalabilidade, de persistência, e, novamente, de EJBs.

É preciso, no entanto, deixar claro que há uma distinção entre um anti-padrão e um simples mau的习惯 de programação, uma vez que toda solução envolve riscos, que são maiores ou menores dependendo da forma como ela é implementada. Por exemplo, a iniciativa de muitos programadores, especialmente iniciantes, em resolver todos os problemas aplicando padrões catalogados resulta diversas vezes em anti-padrões.

Nesse artigo vamos demonstrar anti-padrões que aparecem de forma bastante recorrente. Para cobrir os casos mais comuns e, especialmente, sedimentar conhecimento a respeito de como essas ocorrências se formam, exploraremos quatro anti-padrões classificados como metodológicos (“Otimização prematura”, “Bala de prata”, “Martelo de ouro” e “Programação por permuta”), além de outros seis anti-padrões de programação (“Spinning”, “Se repita”, “Escondendo erros”, “Sufoco”, “Sessão estufada” e “Código espaguete”).

No caso dos anti-padrões metodológicos, mais do que código-fonte, demonstraremos como a intenção do programador acaba sendo refletida em aumento de riscos e prejuízos à qualidade. Já os padrões de programação têm reflexo imediato em código que, embora não seja necessariamente ruim, acrescenta riscos

catastróficos e desnecessários ao projeto. Acreditamos que os exemplos analisados nesse artigo são suficientes para despertar no desenvolvedor a precaução necessária à boa prática de desenvolvimento.

Quando um simples mau hábito se torna um anti-padrão metodológico

Maus hábitos não são necessariamente anti-padrões. Não há, por exemplo, um problema intrínseco na atitude de copiar código ou de testar iterativamente um algoritmo até achar uma forma em que ele funcione. No entanto, a linha que separa uma má-prática ou um hábito ruim de um anti-padrão perigoso é bastante tênue e deve ser observada cuidadosamente.

Por isso, nas subseções a seguir apresentaremos alguns casos para ajudar o leitor a reavaliar seus hábitos como desenvolvedor e, dessa forma, evitar a criação ou reprodução de anti-padrões que podem trazer prejuízos consideráveis a seus projetos.

A “premonição” do desenvolvedor: Otimização prematura

A necessidade de resolver todos os problemas da maneira mais rápida possível é uma característica humana, inerente não apenas aos profissionais de computação, mas a qualquer pessoa. Ninguém gosta de conviver com a possibilidade da catástrofe.

Isso nos leva à tentativa constante e equivocada de tentar prever o que deve ser implementado, antes mesmo das etapas de análise serem completadas. Em seu livro, Koenig chama essa característica de “Otimização prematura”. Esse anti-padrão, certamente um dos mais recorrentes, ocorre quando o desenvolvedor, na tentativa de garantir níveis elevados de eficiência ou desempenho, implementa trechos de código que não necessariamente estarão definidos no projeto da solução. Muitas vezes boas práticas de design são sacrificadas, por isso implementando “tiros de canhão em formigas”, sem relação direta com requisitos funcionais estabelecidos ou em análise.

As consequências não são somente essas. É sabido que a otimização, mesmo quando bem implementada, pode reduzir a legibilidade do código em prol do aumento do desempenho. Porém, quando o código implementado perde sua relação com os resultados da análise, essa situação fica ainda mais grave, pois a rastreabilidade se perde e não há de fato aumento justificável do desempenho.

Esse anti-padrão é extremamente perigoso e difícil de localizar, pois pode ser implementado com código aparentemente de boa qualidade. Isso pode ser demonstrado, por exemplo, no uso exaustivo de especificações como *Enterprise JavaBeans* (EJB). Inicialmente projetados para a computação distribuída, EJBs têm sido fartamente utilizados nos mais diversificados tipos de aplicação, principalmente depois que a comunidade de desenvolvedores reconheceu que seu uso em aplicações distribuídas não explorava todo o potencial da especificação.

Não há nenhum pecado em utilizar esse tipo de componente em casos que não envolvem computação distribuída. Como acabamos de dizer, usar EJBs somente dessa forma é subutilizar a especificação.

Anti-padrões: o que são, como identificar e evitar

No entanto, é possível notar que seu uso se massificou de tal forma que alguns desenvolvedores, em especial aqueles menos experientes, perderam a real noção do propósito do uso de EJBs.

Por exemplo, como a *Java Persistence API* (JPA – ver **BOX 2**) já incorpora um *Entity Bean* que fornece acesso a dados de maneira bastante simples, a arquitetura dos EJBs tem sido fartamente – e bem – utilizada em substituição ao padrão DAO para acesso à camada de persistência. No entanto, é preciso destacar que essa não é, obviamente, a única forma de se implementar acesso a dados em Java e, claro, não é sempre a mais adequada maneira de implementar a persistência, beirando o exagero em alguns casos.

BOX 2. Java Persistence API (JPA)

A Java Persistence API é a especificação que descreve a implementação do acesso a persistência em bancos de dados relacionais em Java. Embora tenha sido desenvolvida inicialmente para a especificação EJB, pode ser utilizada diretamente por qualquer cliente, aplicações web, e até mesmo fora da plataforma Java EE, como por exemplo, aplicações Java SE.

Sendo assim, podemos encontrar o anti-padrão “Otimização prematura” implementado de diversas maneiras. Como exemplos comuns, podemos citar o uso desnecessário de cache ou a persistência de dados completamente desnecessários e que não estão presentes no projeto da solução.

Da Bala de Prata ao Martelo de Ouro: complexidade desnecessária

Como afirmamos na introdução desse artigo, não deve ser difícil encontrar viciados em padrões, até mesmo porque alguns deles são verdadeiramente viciantes. Vamos, por exemplo, resgatar o padrão GoF (**BOX 3**) Singleton. Ele foi criado para que se pudesse controlar a quantidade de instâncias (originalmente, deveria manter apenas uma instância ativa) que uma determinada classe disponibilizaria.

Esse padrão foi tão adaptado que muitas vezes podemos vê-lo controlando a instanciação de classes em contextos ou momentos totalmente desnecessários. Por mais exagerado que possa parecer, é possível encontrar projetos em que todos os *beans* são singletons, gerando questionamentos sobre a aplicação desse padrão.

BOX 3. Catálogo de padrões Gang of Four (GoF)

O catálogo da Gang-of-Four é o repositório de padrões para a programação orientada a objetos publicado por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides em 1994, no livro “Design Patterns: Elements of Reusable Object-Oriented Software”. Esse livro se tornou um clássico no ensino de programação e descreve 23 padrões de projeto considerados essenciais.

Isso traz à tona o fato de que muitos desenvolvedores acreditam que uma determinada solução, por ser popular, pode ser aplicada a um grande número de problemas. Esse tipo de anti-padrão é comumente referenciado pelos nomes de “Bala de prata” e “Martelo de ouro”. Anti-padrões classificados dessa forma têm uma relação intrínseca com outro anti-padrão, chamado popularmente de “Complexidade accidental” ou “Complexidade desnecessária”,

uma vez que tudo que se consegue é aumentar a complexidade da implementação de forma completamente desnecessária, sem nenhum benefício aparente.

Tanto o termo “Bala de prata” quanto “Complexidade acidental” são descritos em um artigo do vencedor do prêmio Turing (considerado o prêmio Nobel da computação) Fred Brooks. Brooks afirma que é preciso distinguir a complexidade acidental da complexidade essencial, pois embora a complexidade acidental possa ser consertada, nada pode remover a complexidade essencial, que está relacionada fortemente ao problema a ser resolvido.

A tentação da “Programação por permuta”

Por mais óbvio que pareça, não custa lembrar: copiar e colar não é reusar. Embora essa frase possa parecer ridícula para programadores experientes, é importante ressaltar que esse cenário não é o mesmo para equipes menos experientes. De qualquer maneira, é difícil – ou impossível – encontrar um desenvolvedor que nunca copiou código, muitas vezes multiplicando problemas de maneira viral.

Também é fácil destacar a abordagem contraproducente da tentativa-e-erro na solução de um problema. Primeiramente, nesses casos em que o desenvolvedor não se dá o trabalho de compreender o problema e o algoritmo, não há como comprovar que uma vez tendo funcionado, a solução encontrada não apresentará mais erros. Mais do que isso, há uma grande probabilidade de que o conjunto de pequenas modificações introduzidas no código, com o intuito de corrigí-lo, possa carregar novos erros.

Esse tipo de vício de programação é considerado um anti-padrão, sendo batizado de forma bastante irônica como “Programação por permuta”, ou seja, indica que o desenvolvedor não deu a devida importância à corretude do código (ou sequer para seu entendimento), na medida em que o algoritmo se mostrou funcional para um determinado número de opções de entrada.

O que ocorre nesses casos é algo bastante frequente na comunidade de desenvolvedores: erros intermitentes, ou seja, problemas que não serão detectados até que testes apurados sejam realizados ou que, quando ocorrerem, serão difíceis de serem reproduzidos e corrigidos devidamente.

Embora esse seja um anti-padrão metodológico, tentamos demonstrá-lo em um exemplo de código na **Listagem 1**. Nesse exemplo, o programa deveria exibir como saída somente os caracteres numéricos em uma determinada palavra.

O trecho de código mostrado, além de não apresentar a saída correta (recebendo a entrada “abc123”, ele exibe “13”, quando deveria exibir “123”), induz o programador a cometer o erro de reduzir os incrementos da variável *i*. No entanto, se removermos a linha onde *i* é incrementada, o algoritmo entra em *loop* infinito.

Também é importante destacar que, embora nesse exemplo – que usa uma entrada hipotética – o problema esteja claro para os leitores, com entradas reais o erro pode demorar ainda mais para ser detectado.

Nesse exemplo, tentamos demonstrar a má prática da alteração iterativa das linhas de código até que ele produza o resultado

esperado, uma vez que a implementação é de difícil compreensão. Sendo assim, como o programador aparentemente não se preocupa em entender o código antes de alterá-lo, erros dessa natureza só são descobertos (quando o são) em estágios avançados de verificação e testes, em geral com custos maiores para o projeto.

Anti-padrões de programação

Os anti-padrões de projeto ou aqueles associados à análise ou metodologia, por estarem relacionados a problemas conceituais, são realmente bastante difíceis de diagnosticar. No entanto, anti-padrões de programação, ligados diretamente com a escrita do código, são igualmente perigosos.

Dessa forma, padrões de programação vão desde simples vícios como deixar atributos com visibilidade pública (intitulado de forma bastante debochada de “orgia de objetos”), a mal uso de estruturas de código simples, como laços e condições (também com nomenclatura debochada de “código espaguete” ou “código lasanha”).

Portanto para ilustrar um pouco desse universo, daremos foco a três anti-padrões bastante disseminados, sendo portanto muito perigosos e cujos prejuízos passam muitas vezes despercebidos.

Locks incrementais com o anti-padrão Spinning

O anti-padrão “spinning”, também conhecido como “espera ocupada” ou “espera ativa”, é um dos mais comumente encontrados. Detectamos a sua existência em porções de código em que uma determinada instrução realiza verificações repetidas até que uma determinada condição seja atendida, consumindo recursos desnecessariamente.

Esse anti-padrão pode ser implementado de infinitas maneiras, como a ilustrada por Obi Ezechukwu, desenvolvedor e autor em técnicas de programação orientada a objetos, em um artigo para a JavaWorld em 2009 (ver seção **Links**), que ele intitulou de “Lock incremental”.

Nesse exemplo, a existência de *deadlocks* e a forma desnecessária como eles são provocados é demonstrada, assim como a gravidade de sua ocorrência para o funcionamento de qualquer aplicação. No entanto, mesmo com farta documentação sobre esse tipo de problema, ainda é fácil se deparar com código que provoca *deadlocks*, às vezes, de forma incremental.

Um *lock* incremental ocorre quando um objeto requisita um determinado conjunto de recursos, mas, quando o último desses recursos é obtido, o primeiro recurso ainda está em uso por algum objeto e, portanto, indisponível. A principal consequência desse tipo de ocorrência é a espera por recursos, em muitos casos por tempo indefinido, uma vez que em Java é difícil (ou impossível) forçar um processo a liberar recursos que estão bloqueados, especialmente no caso de um *deadlock*.

Esse tipo de anti-padrão é bastante difícil de ser detectado pela complexidade da implementação e do uso de threads, que requer experiência e atenção do desenvolvedor. Para ilustrá-lo, vamos imaginar que uma determinada aplicação precisa compartilhar os recursos de um objeto da classe **RecursoCompartilhado**, como podemos ver na **Listagem 2**.

Listagem 1. Código-fonte apresentando o anti-padrão “programação por permuta”.

```
package br.com.devmedia.antipadroes;

public class ProgramacaoPorPermuta {

    public static void main(String[] args){

        StringBuffer entrada = new StringBuffer("123abc");
        char[] saida = new char[10];

        int i = 0;
        int j = 0;
        int l = entrada.length();

        while (i < l) {
            if (Character.isDigit(entrada.charAt(i))) {
                saida[j++] = entrada.charAt(i++);
            }
            i++;
        }

        saida[j] = '^0';
        System.out.println(saida);

    }
}
```

Listagem 2. Código-fonte da classe **RecursoCompartilhado**.

```
package br.com.devmedia.antipadroes.spinning;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class RecursoCompartilhado {

    private Lock lockA;
    private Lock lockB;

    public RecursoCompartilhado(){
        lockA = new ReentrantLock();
        lockB = new ReentrantLock();
    }

    public Lock getLockA() {return lockA;}
    public Lock getLockB(){return lockB;}

}
```

Em seguida, vamos implementar a classe **Spinning**, criando duas threads no método **main()** que tentarão compartilhar os recursos de um objeto do tipo **RecursoCompartilhado**. O código-fonte da classe **Spinning** pode ser visto na **Listagem 3**.

Ao executar essa classe, além de vermos que o *lock* secundário não é obtido, podemos notar que sua execução não é terminada. O problema nesse caso é que **thread0**, quando obtém o **lockA**, também tenta obter o **lockB**, que por sua vez já está preso por **thread1**. O mesmo ocorre quando **thread1** tenta obter o **lockA**, levando a aplicação a um *deadlock*.

Embora esse exemplo pareça mais complicado, sua solução é bastante simples. Basta que cada thread, ao invés de pedir o segundo *lock*, execute um método como “*tryLock()*”, que retorna “*false*” quando o *lock* secundário está em uso, liberando o *lock* primário e, consequentemente, seus recursos.

Anti-padrões: o que são, como identificar e evitar

Listagem 3. Código-fonte da classe Spinning, com o respectivo anti-padrão.

```
package br.com.devmedia.antipadroes.spinning;

import java.util.concurrent.locks.Lock;

public class Spinning extends Thread{

    protected int threadID;
    protected Lock lockPrimario, lockSecundario;

    public Spinning(RecursoCompartilhado recursoCompartilhado, boolean inverterOrdem, int threadID){
        this.threadID = threadID;

        if (inverterOrdem) {
            this.lockPrimario = recursoCompartilhado.getLockA();
            this.lockSecundario = recursoCompartilhado.getLockB();
        }else{
            this.lockPrimario = recursoCompartilhado.getLockB();
            this.lockSecundario = recursoCompartilhado.getLockA();
        }
    }

    @Override
    public void run(){
        try{
            lockPrimario.lock();
            System.out.println("Thread " + getName() + " está em modo de leitura");
        }catch(InterruptedException exce){
            Thread.sleep(1000*threadID);

            lockSecundario.lock();
            System.out.println("Thread " + getName() + " está em modo de escrita");
        }finally{
            lockSecundario.unlock();
            lockPrimario.unlock();
        }

        System.out.println("Thread " + getName() + " finalizada");
    }
}

public static void main(String[] args) {
    Spinning thread1;
    Spinning thread2;
    RecursoCompartilhado recurso;

    recurso = new RecursoCompartilhado();

    thread1 = new Spinning(recurso, false, 1);
    thread1.start();

    thread2 = new Spinning(recurso,true, 2);
    thread2.start();
}
```

Não reuse, se repita!

Um dos conceitos mais difundidos na programação ágil é o DRY (ver **BOX 4**). Esse conceito tem relação bastante estreita com o princípio da abstração, uma das bases fundamentais da orientação a objetos (ver **BOX 5**). O anti-padrão “Se repita” vai diretamente de encontro a esse conceito. Sua adoção significa escrever código que contém repetições de estruturas que poderiam ser encapsuladas e reusadas.

A palavra “DRY”, que é traduzida para o português como “seco”, permite uma analogia com código otimizado e enxuto. Sendo assim, aplicações do anti-padrão “se repita” são comumente designadas pela palavra “WET” (“molhado” em português),

BOX 4. Don't Repeat Yourself (DRY)

É um conceito de programação, descrito por Andy Hunt e Dave Thomas – cofundadores do Manifesto Ágil –, no livro “The Pragmatic Programmer”, que estabelece como meta a redução das repetições de informação de todo e qualquer tipo, especialmente entre as camadas da aplicação. De acordo com esse princípio, na medida em que cada elemento de negócio ou conhecimento do software é implementado univocamente, a modificação de um elemento não acarreta a necessidade de modificar outros.

BOX 5. Princípio da Abstração

O princípio da abstração, enquanto base do paradigma da orientação a objetos, determina que a análise e o projeto devem considerar os aspectos essenciais do contexto, em detrimento de características que possam ser consideradas menos importantes. Cabe ao analista, de acordo com a análise do contexto – ou do domínio –, definir quais as características essenciais para o projeto da solução e quais elementos podem ser deixados de fora da análise.

permitindo um trocadilho que remete a código repetitivo, como mostra a **Listagem 4**.

Nesse código podemos ver um método para cada operação Matemática (com corpo bastante semelhante), enquanto poderíamos ter um único método flexível o suficiente para realizar todas as operações.

Listagem 4. Código-fonte WET.

```
package br.com.devmedia.antipadroes;

public class SeRepita {

    public int somar(int x, int y){
        int z = x + y;
        return z;
    }

    public int multiplicar(int x, int y){
        int z = x * y;
        return z;
    }

    public int subtrair(int x, int y){
        int z = x - y;
        return z;
    }
}
```

Enfim, repetição é algo bastante frequente em projetos mal documentados e com longo ciclo de vida, o que nos faz deduzir que esse anti-padrão é por muitas vezes resultante de projetos mal elaborados ou mal conduzidos, podendo, inclusive, ser provocado por outros anti-padrões.

Dificultando a correção de problemas com o anti-padrão “Escondendo erros”

A exibição de mensagens de erros está presente em qualquer plataforma de desenvolvimento e, de forma geral, essas mensagens não são amigáveis. No entanto o público alvo dessas mensagens, enviadas pelo compilador ou pelo interpretador, é o desenvolvedor e, por isso, elas tendem a exibir o maior número de detalhes possível sobre a ocorrência, tornando sua compreensão bastante difícil para um usuário comum.

Sendo assim, a customização das mensagens de erro é uma tendência natural para qualquer desenvolvedor, ocultando o máximo possível de informações desnecessárias ou incompreensíveis para o usuário final.

É justamente nesse ponto que reside o maior perigo, uma vez que a definição a respeito do quanto de informação sobre o erro deve ser exibida é bastante subjetiva. Mais do que isso, devemos deixar claro que um erro do sistema vai necessariamente ser resolvido por um desenvolvedor. Sendo assim, é útil que a mensagem de erro exiba detalhes importantes sobre sua ocorrência ou sua origem. No entanto é muito comum a ocultação total das informações sobre os erros em prol de uma mensagem amigável para o usuário final, como vemos na **Listagem 5**.

Listagem 5. Código-fonte do anti-padrão “Escondendo erros”.

```
package br.com.devmedia.antipadroes;

import java.io.*;

public class EscondendoErros {

    public static void main(String[] args) {

        File file = new File("C:\\arquivo.txt");
        FileInputStream fis = null;
        BufferedInputStream bis = null;
        DataInputStream dis = null;

        try {
            fis = new FileInputStream(file);
            bis = new BufferedInputStream(fis);
            dis = new DataInputStream(bis);

            while (dis.available() != 0) {
                System.out.println(dis.readLine());
            }

            fis.close();
            bis.close();
            dis.close();

        }catch (Exception e) {
            System.out.println("Erro ao importar arquivo");
        }
    }
}
```

Nesse exemplo, fizemos uma pequena aplicação que realiza a tarefa de importar um arquivo de texto (*arquivo.txt*). Contudo, embora esse tipo de problema esteja claramente sujeito a erros de importação como, por exemplo, inexistência do arquivo ou de seu caminho físico, o desenvolvedor ocultou toda e qualquer informação a respeito do erro ocorrido.

No caso em questão o desenvolvedor poderia facilitar a correção de eventuais problemas, por exemplo, exibindo na mensagem da exceção informações sobre o tipo do erro, nome e caminho do arquivo, ou fazer múltiplos tratamentos de exceções com mensagens exclusivas.

Pelo contrário, tudo que é exibido na ocorrência de uma exceção – seja ela qual for – é o texto “Erro ao importar arquivo”, que não diz absolutamente nada a respeito do erro, dificultando bastante a depuração e a devida correção do problema.

“Sufocando” a persistência com “Stifle”

Imagine uma operação comum como inserir registros num banco de dados relacional. Algumas vezes é necessário inserir diversos registros de uma só vez para atualizar/alimentar a base de dados, operação bastante comum em diversos tipos de aplicação.

É nesse momento que o anti-padrão “Sufoco” (que em inglês é batizado de “Stifle”) se faz presente. Esse anti-padrão é muito comum em aplicações Java EE (aparece no livro de Dudney, Asbury, K. Krozak e Wittkopf) e sua principal consequência é a redução drástica da performance na camada de persistência, podendo causar *deadlocks*.

De forma geral, ele aparece quando, para resolver o problema de múltiplas atualizações no banco de dados, o programador escreve a execução das instruções SQL dentro de um laço, como mostra a **Listagem 6**.

Na classe **Sufoco**, diversos objetos do tipo **Usuario** devem ser inseridos em uma só chamada ao método **adicionarUsuarios()**. Como os objetos a serem persistidos são passados dentro do

Listagem 6. Código-fonte do anti-padrão “Sufoco”.

```
package br.com.devmedia.antipadroes.sufoco;

import java.sql.*;
import java.util.*;

public class Sufoco {

    public void adicionarUsuarios (ArrayList<Usuario> usuarios) throws Exception{

        Iterator<Usuario> it = usuarios.iterator();

        Connection con = Conexao.getConexao();
        Statement stmt = con.createStatement();

        while (it.hasNext()) {
            Usuario usuario = (Usuario) it.next();
            stmt.execute("INSERT INTO TB_USUARIO (NOME) VALUES
            ('"+ usuario.getNome() +"')");
        }
    }
}
```

Anti-padrões: o que são, como identificar e evitar

ArrayList usuarios, o desenvolvedor escreveu um laço que percorre esse **ArrayList** por meio de um **Iterator** e, a cada volta nesse laço, faz uma inserção.

As consequências para a performance da aplicação são desastrosas, uma vez que é impossível prever o tamanho desse **ArrayList**. Além disso, mesmo que fosse possível determinar previamente o tamanho desse **ArrayList**, o desempenho da aplicação continua comprometido e, dependendo da configuração e estrutura da camada de persistência, *locks* no banco de dados podem atrapalhar o funcionamento do sistema.

Vale ainda ressaltar que a solução para esse problema pode ser encontrada na própria API JDBC. Nesse caso, tudo que o desenvolvedor precisaria fazer para não resultar no anti-padrão “Sufoco” é fazer uma inserção em “Batch”. A classe **Statement**, presente no pacote **java.sql**, possui o método **addBatch()**, que recebe instruções SQL. Bastava ao programador chamar esse método ao invés do método **executeUpdate()**.

Feito isso, restaria definir o momento, tendo cuidado para que o *batch* não fique grande demais, e executá-lo por meio de uma chamada ao método **executeBatch()**.

Embora o uso de *batch* seja mais rústico e simples, existem outras formas mais sofisticadas de resolver o problema em questão, como por exemplo, utilizar filas de mensagens JMS (Java Message Service).

Estufando a sessão

Esse talvez seja o mais simples dos anti-padrões que mostramos nesse artigo, porém um dos mais recorrentes entre desenvolvedores web em Java. Devemos lembrar que o escopo de sessão deve ser utilizado para rastrear o fluxo seguido pelo usuário durante a navegação, no entanto, por mais conveniente que armazenar dados na sessão possa parecer, colocar informação demais nesse escopo – ou pior, o tipo errado de informação – leva o desenvolvedor a se deparar com o anti-padrão intitulado “Sessão estufada”.

Esse anti-padrão é especialmente perigoso para programadores pouco experientes e com pouco conhecimento sobre os fundamentos da arquitetura web em Java. Por exemplo, é preciso saber que, em Java, a sessão é implementada como uma **Collection**, que nada mais é do que uma organização de referências para outros objetos. Por isso, uma *collection*, por si só, costuma ter um ciclo de vida bastante longo, mesmo que os objetos dentro dela não fiquem ativos por tanto tempo.

Sendo assim, se colocarmos objetos no escopo de sessão de maneira inadequada – que provavelmente não serão removidos da memória durante toda a navegação do usuário – podemos causar um erro normalmente chamado de *memory leak* (vide **BOX 6**). Não podemos esquecer que, embora o gerenciamento automatizado da memória seja uma das mais brilhantes características da plataforma Java, nós desenvolvedores não temos controle de quando o *garbage collector* retirará os objetos inutilizados da memória.

Sendo assim, pode ocorrer de alguns objetos que não estão sendo mais utilizados ficarem armazenados na memória por muito mais tempo do que imaginamos.

Por isso é importante que o programador tenha o cuidado de selecionar o escopo correto e, principalmente, o tipo de objeto que deve efetivamente ser colocado no escopo de sessão.

Esse anti-padrão é ilustrado no código fonte do servlet **SessaoEstufada**, mostrado na **Listagem 7**.

BOX 6. Memory leak

Memory leak (ou vazamento de memória) é um erro que ocorre quando uma aplicação utiliza memória de maneira inadequada, por exemplo, alocando um determinado espaço de memória e não liberando-o quando seu uso não é mais necessário. Esse tipo de problema geralmente acarreta perdas consideráveis de performance – ou até mesmo interrupção do funcionamento da aplicação – por reduzir a quantidade de memória disponível.

Listagem 7. Código-fonte do anti-padrão “Sessão estufada”.

```
package br.com.devmedia.antipadroes;

import java.io.IOException;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.annotation.*;

@WebServlet("/SessaoEstufada")
public class SessaoEstufada extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        HttpSession session = request.getSession();

        Curso curso = new Curso();
        curso.setCodigo(request.getParameter("FORM_CODIGO"));
        curso.setTitulo(request.getParameter("FORM_TITULO"));

        session.setAttribute("antipatterns.curso", curso);

        RequestDispatcher dispatcher = getServletContext().getRequestDispatcher(
            "/Resposta.jsp");
        dispatcher.forward(request, response);
    }
}
```

Nesse trecho de código vemos que um objeto do tipo **Curso** é colocado na sessão. A menos que haja um requisito específico para isso, não há nada que indique que um objeto desse tipo precise ficar disponível por toda a navegação do usuário. No entanto, no caso demonstrado na **Listagem 7**, seu ciclo de vida se estenderá por todo o tempo de sessão – enquanto o usuário estiver conectado – de maneira desnecessária.

Esse tipo de erro é bastante difícil de ser detectado, uma vez que a decisão do escopo de cada objeto instanciado na aplicação dificilmente é feita em etapas de análise e projeto, ficando portanto totalmente a cargo do desenvolvedor. Talvez por isso podemos encontrá-lo em aplicações bastante comuns, como carrinhos de compras em sites de comércio eletrônico, ou para armazenar preferências de usuário. Sua solução, no entanto, não é complicada. Basta ter atenção ao tempo de vida que cada objeto deve ter. Para dados com curto ciclo de vida, podemos usar o escopo

de *request*, deixando o escopo de sessão apenas para dados que efetivamente precisam estar disponíveis por toda a navegação do usuário. Também é importante ter em mente que, embora não seja muito comum, é possível remover dados da sessão de forma explícita com poucas instruções em Java, de maneira relativamente simples.

Também como forma de prevenir esse anti-padrão, Crawford e Kaplan afirmam em seu livro que objetos com longo ciclo de vida devem ser instanciados na camada de negócio, nunca na camada de visualização. Para os autores, as principais vantagens dessa abordagem é que esses dados estariam persistentes mesmo que a conexão do usuário se perca.

Exagerando no código e preparando um “Espaguete”

Aplicações web fazem, de forma bastante regular, uso de pré-processamento (por exemplo, verificar se o usuário está logado) e pós-processamento (enviar respostas às requisições dos usuários). Em versões mais antigas da Servlet API, o código referente a esses processamentos era, de forma geral, escrito nos métodos do *servlet* – mais comumente nos métodos **doGet()** e **doPost()**. No entanto, especificações mais recentes permitem que esse tipo de código fique em outros componentes, reduzindo a complexidade e flexibilizando a manutenção.

Dessa forma, manter código que faça com que um *servlet* realize tarefas de processamento das requisições para gerar respostas ao usuário acaba materializando o anti-padrão conhecido como “Código espaguete”, uma vez que fazem com que o *servlet* encapsule mais tarefas – e consequentemente mais código – do que deveria. Nos casos mais graves, o código é escrito diretamente nos métodos **doPost()** e **doGet()**. Vemos um exemplo desse anti-padrão na [Listagem 8](#).

Nesse exemplo, o Servlet possui os métodos **efetuarLogin()** e **validarUsuario()**. O primeiro é responsável por autenticar o usuário, recuperando seu registro no banco de dados. O segundo, por validá-lo, verificando se o usuário encontrado na sessão é o mesmo recebido pelo método. Podemos ver também que ambos os métodos são chamados por **doPost()**. Nesse caso, uma solução simples seria escrever esses métodos (**efetuarLogin()** e **validarUsuario()**) em uma outra classe, em outra camada da aplicação. Isso não só tornaria o Servlet mais enxuto como preservaria suas características e promoveria o reuso dos métodos **efetuarLogin()** e **validarLogin()**.

O anti-padrão “Espaguete”, embora geralmente de fácil detecção, é de difícil refatoração, uma vez que parte considerável do fluxo da aplicação é incorporada em servlets. Resolver esse tipo de problema pode demandar alterações no projeto e a criação de novas classes.

FÓRUM DEVMEDIA

**O lugar perfeito para você ficar por dentro
de tudo o que acontece nas tecnologias do
mercado atual**

No fórum da DevMedia você irá encontrar uma equipe disponível e altamente qualificada com consultores e colaboradores prontos para te ajudar a qualquer hora e sobre qualquer assunto. Temos as salas de Java, .NET, Delphi, Banco de Dados, Engenharia de Software, PHP, Java Script, Web Design, Automação comercial, Ruby on Rails e muito mais!



ACESSE AGORA
www.devmedia.com.br/forum

Anti-padrões: o que são, como identificar e evitar

Listagem 8. Código-fonte do anti-padrão “Código espaguete”.

```
package br.com.devmedia.antipadroes;

import java.io.IOException;
import javax.servlet.*;
import javax.servlet.http.*;
import br.com.devmedia.antipadroes.sufoco.*;

@WebServlet("/Espaguete")
public class Espaguete extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {}

    protected void doPost(HttpServletRequest request, HttpServletResponse response, HttpSession session) throws ServletException, IOException, Exception {

        Usuario usuario = efetuarLogin(request);
        try {
            efetuarLogin(request);
            validarUsuario(usuario, session);
        } catch (Exception e) {
            throw e;
        }
    }

    private void validarUsuario(Usuario usuario, HttpSession session) throws Exception{
        if (((Usuario)session.getAttribute("SESSION_USUARIO")).getId() != (usuario.getId()))
        {
            throw new Exception ("Usuario Invalido!");
        }
    }

    private Usuario efetuarLogin(HttpServletRequest request){
        Usuario usuario = UsuarioController.getUsuario((String)request.getAttribute("FORM_NOME"), (String)request.getAttribute("FORM_SENHA"));
        return usuario;
    }
}
```

Há um famoso ditado que diz: “se a única ferramenta que você tem é um martelo, vai acabar achando que tudo que vê é prego”. Nós desenvolvedores estamos sempre lidando com esse tipo de pensamento devido ao paradoxo inerente à nossa profissão: ao mesmo tempo em que a tarefa de programar pode ser bastante repetitiva, há uma enorme variabilidade, na medida em que não é possível afirmar com que problemas teremos que lidar diariamente.

É por isso que o uso de padrões pode ser ao mesmo tempo a salvação e uma maldição, se mostrando como uma maneira de simplificar o trabalho, mas exigindo um enorme conhecimento sobre fundamentos de desenvolvimento orientado a objetos.

Embora também tenhamos mostrado nesse artigo exemplos de como os anti-padrões se refletem na escrita do código, gostaríamos

de ressaltar que a sua existência tem uma relação muito maior com a forma como os profissionais de computação aplicam os conceitos fundamentais da área. A existência de anti-padrões, assim como a aplicação de boas práticas e padrões de arquitetura, se relaciona intrinsecamente com a forma de abordar um problema, que não deve, de forma alguma, ser minimalista nem preciosista. Cabe ao desenvolvedor adquirir os conceitos necessários, refletir a respeito de sua aplicação e achar os limites para a adoção de quaisquer técnicas ou padrões.

Dessa forma, podem ser evitadas situações desagradáveis como o uso de padrões sem saber o motivo (anti-padrão intitulado de forma bem humorada como “Culto da programação”) ou manias como evitar retirar código redundante ou de baixa qualidade devido ao custo ou imprevisibilidade das consequências de sua remoção (também intitulado de maneira bem humorada como “Fluxo de lava”).

Além disso, ao contrário dos nomes engraçados que esses anti-padrões recebem na comunidade de desenvolvedores, suas consequências não são nada divertidas, tanto para os projetos de software em si quanto para as carreiras daqueles que os utilizaram.

Talvez por esses motivos, Donald Knuth, ilustre ganhador do Prêmio Turing, tenha escrito em seu famoso artigo “Structured programming with GoTo Statements” (“Programação estruturada com instruções GoTo”, em tradução livre) que os programadores gastam um tempo enorme se preocupando com o desempenho de determinadas partes de seus programas, criando geralmente código de difícil depuração e manutenção, causando impactos negativos no projeto de software.

Autor



Alessandro Jatobá

jatoba@jatoba.org

É Mestre em Ciência da Computação pela UFRJ, Doutorando em Engenharia também pela UFRJ com Doutorado-Sanduíche pelo Departamento de Engenharia e Projeto de Sistemas da Universidade de Waterloo, no Canadá. Trabalha com Java há mais de 15 anos, a maior parte deles liderando equipes de desenvolvedores em projetos Web. Também é professor universitário, lecionando temas ligados ao desenvolvimento orientado a objetos.



Links:

Página de Andrew Koenig no Dr. Dobb's Journal.

<http://www.drdobbs.com/author/Andrew-Koenig>

Página de Obi Ezechukwu na Java World.

<http://www.javaworld.com/author/Obi-Ezechukwu/>

Criando aplicações Desktop em Java

Aprenda sobre os novos recursos do Java 8 na prática

A nova versão do Java trouxe mudanças significativas e marcantes com relação à linguagem. Essas mudanças não se relacionam apenas com a inclusão ou alteração de APIs ou mesmo mudanças discretas na máquina virtual (JVM). Elas foram além e causaram impactos até mesmo na sintaxe da linguagem. Sendo assim, neste artigo mostraremos como o Java 8 agora incorpora conceitos provenientes de linguagens funcionais, como Lisp e Haskell, para tornar ainda mais fácil o desenvolvimento de determinadas tarefas que antes necessitavam de mais complexidade e muitas linhas de código.

Para grande parte daqueles que lidam com Java no dia a dia essas mudanças precisarão de um tempo para serem totalmente absorvidas. A nova *feature* mais significante é a adição das Expressões Lambdas (EL) como uma maneira alternativa para escrever classes internas anônimas. Certamente, para aqueles que já se aventuraram em linguagens como Scala, que executa na JVM, estas mudanças não causarão tanto impacto.

Outra mudança é a inclusão de uma nova API para trabalhar com datas, a Date and Time. Esse novo recurso é um clamor antigo dos desenvolvedores Java que sempre criticaram a API antiga, caracterizando-a como complexa e cansativa. Felizmente, agora teremos um novo conjunto de classes e interfaces totalmente reescrito, melhor e mais fácil de usar.

Para manipulação de coleções em Java a solução Stream API, também lançada com o Java 8, fornece um estilo de programação funcional. Com relação a este recurso, veremos como é feita sua integração com as coleções do Java e demonstraremos algumas facilidades que essa API oferece para realizar diferentes operações.

Outra novidade que abordaremos nesse artigo é chamada de Default Methods. Esse recurso permite que interfaces presentes na linguagem Java disponibilizem métodos novos sem que as classes que as implementem tenham que fornecer uma implementação para esses novos métodos.

Portanto, nesse tutorial será mostrado como desenvol-

Fique por dentro

Neste artigo mostraremos como desenvolver uma aplicação desktop utilizando recursos do Java 8. Aprenderemos de forma básica e simples como criar um CRUD sobre um cadastro de clientes. Por meio deste exemplo, veremos algumas das novas funcionalidades da versão mais recente do Java associadas à utilização do padrão de projeto Facade e também como construir GUI (interface gráfica) lançando mão do JavaFX, biblioteca padrão do Java 8 para construção de interfaces. A partir disso o leitor terá uma base sólida para iniciar o desenvolvimento de suas aplicações com a versão mais recente do Java.

ver uma aplicação desktop em Java utilizando os novos recursos mencionados e também teremos a oportunidade de trabalhar com a biblioteca JavaFX, padrão no Java 8, que será responsável pela camada de front-end e servirá como substituta do Swing.

Teremos a oportunidade de ver esses novos recursos empregados em um contexto desktop. Apesar da grande maioria das aplicações hoje em dia serem web por diversos motivos como: facilidade de atualização, alcance maior, compatibilidade com qualquer sistema operacional que possua um browser e diversas outras, as aplicações desktop ainda têm seu espaço no mercado. Por exemplo, aplicações PDV (Ponto de Venda) geralmente são feitas para rodar como uma aplicação desktop, principalmente pela necessidade de tempo de resposta curto. Outro exemplo que pode ser citado é a aplicação do governo para declaração de imposto de renda. A versão atual dessa aplicação é desktop. Além disso, é sempre bom dar uma relembrada no bom e velho desenvolvimento de aplicativos standalone e aplicativos para desktops.

Para evitar a complexidade de integração com banco de dados e não sair do escopo, a aplicação que desenvolveremos armazena os objetos em uma estrutura de dados (Collection) que “simula” um banco de dados.

Preparando o ambiente de desenvolvimento

Antes de iniciarmos o desenvolvimento da aplicação é necessário ter o Java 8 instalado (veja na seção **Links** o endereço para

Criando aplicações Desktop em Java

download). Na página de downloads do Java, temos versões específicas do JDK para vários sistemas operacionais. Escolha aquela compatível com o seu ambiente de trabalho.

Após o download do JDK, basta executar sua instalação para que possamos iniciar os estudos das mudanças e novos recursos do Java 8.

Com o intuito de aumentar a produtividade, utilizaremos como IDE o NetBeans 8.0.1, que suporta as funcionalidades do JDK 8 (veja na seção **Links** o endereço para download). Quando alguma das novas construções do Java é utilizada no código, o IDE reconhece-as, realça os erros corretamente e permite que o desenvolvedor corrija a sintaxe automaticamente. Dito isso, após efetuar o download, basta instalar o NetBeans.

Tendo o IDE e o Java 8 instalados no sistema, o próximo passo é registrar o Java no IDE conforme os passos a seguir:

1. Com o NetBeans aberto, selecione *Ferramentas > Plataformas Java* no menu principal;
2. Em seguida, clique em *Adicionar Plataforma* na caixa de diálogo *Gerenciador de plataforma Java*;
3. Na caixa de diálogo *Adicionar Plataforma Java*, selecione *Edição Padrão Java* e clique em *Próximo*;
4. Especifique o diretório que contém o JDK e clique em *Próximo*, assim como na **Figura 1**;
5. Clique em *Finalizar* para fechar a caixa de diálogo *Adicionar Plataforma Java*. Com isso o JDK 8 é registrado como uma plataforma no IDE;
6. Por fim, assegure-se que o JDK 1.8 esteja selecionado na lista *Plataformas* e clique em *Fechar*, conforme a **Figura 2**.

Deste modo o próprio IDE já será executado na versão 8 e nenhuma outra configuração adicional será necessária.

A aplicação Cadastro de Clientes

Para estudarmos as novidades e mudanças trazidas com o Java 8, criaremos uma pequena aplicação que realiza sobre um cadastro de clientes as seguintes operações: Cadastro, Deleção, Pesquisa e Edição.

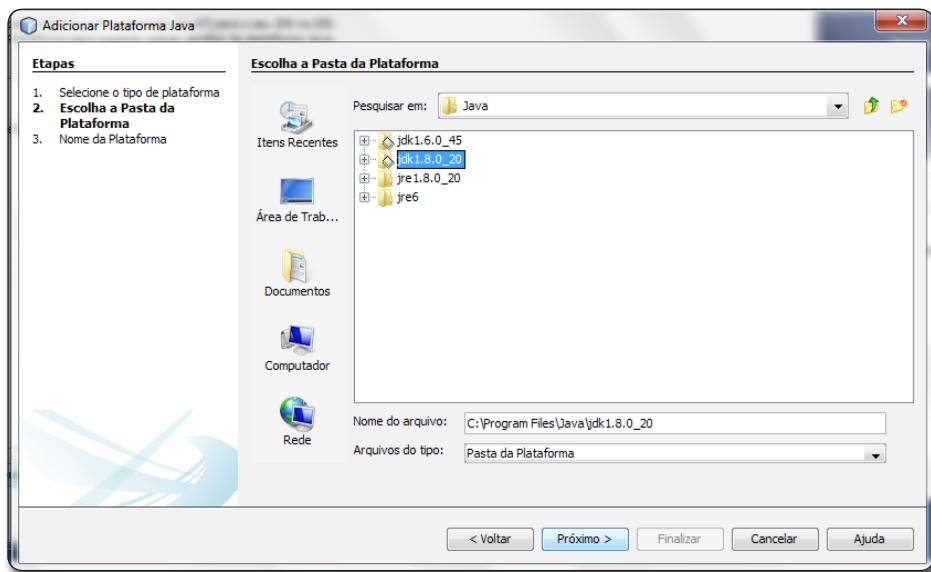


Figura 1. Especificando o diretório de instalação do JDK 8

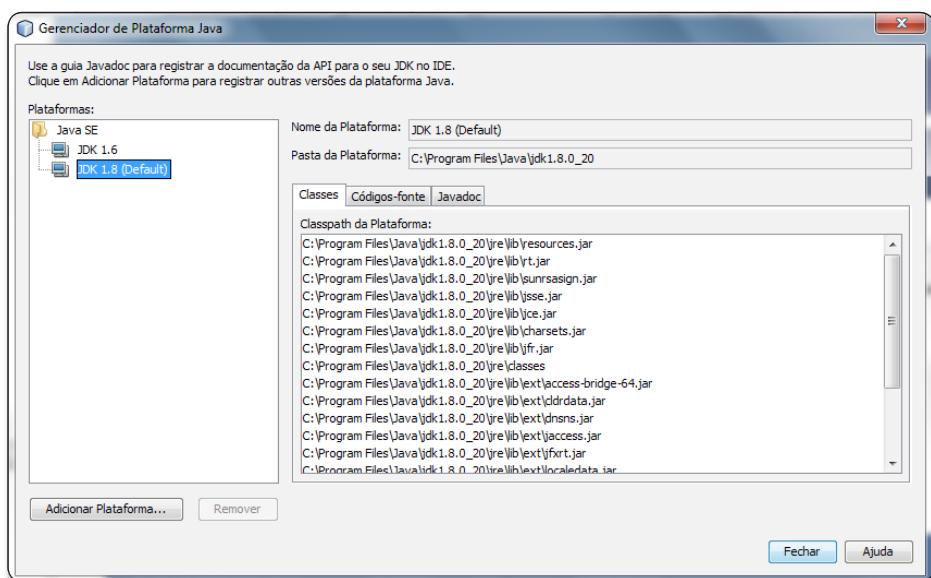


Figura 2. Confirmando seleção do JDK 1.8

Para isso, utilizaremos o padrão arquitetural MVC. Esse padrão define a divisão de uma aplicação em três componentes: Modelo, Visão e Controle. No primeiro componente temos o repositório de informações e as classes que manipulam essas informações. No segundo temos a interface com o usuário e no terceiro o controle do fluxo de todas as informações que passam pelo sistema. O principal foco dessa estrutura é dividir um grande problema em vários problemas menores e de menor complexidade.

Dessa forma, qualquer tipo de alteração em uma das camadas não interfere nas demais, facilitando a atualização de layouts, alteração nas regras de negócio e adição de novos recursos. Em caso de grandes projetos, o MVC facilita bastante a divisão de tarefas entre a(s) equipe(s).

Neste exemplo também faremos uso do padrão de projeto Facade. Através desse padrão é possível ocultar toda a complexidade de uma ou mais classes fazendo uso de uma fachada (Facade). Para construção da interface com o usuário será utilizado

o JavaFX, nova biblioteca gráfica da plataforma Java que dispõe de vários recursos para criação de aplicações ricas.

Criando a aplicação no NetBeans

A fim de iniciarmos o desenvolvimento, criaremos uma aplicação desktop no NetBeans clicando no botão *Novo Projeto*, selecionando a categoria *Java Desktop* e depois clicando em *Próximo*. Na tela seguinte, no campo *Nome do Projeto*, informe “*JavaApplicationCRUD*” e clique no botão *Finish* para confirmar a criação do projeto.

Com o projeto criado, é preciso configurá-lo para utilizar o JDK 8 para compilar, executar e depurar.

Para configurar o projeto, os seguintes passos são necessários:

1. Na janela *Projetos* do NetBeans, clique com o botão direito do mouse no projeto *JavaApplicationCRUD* e selecione as *Propriedades* no menu de contexto;
2. Na caixa de diálogo *Propriedades do Projeto*, escolha a categoria *Bibliotecas* e defina o *JDK 1.8* como a *Plataforma Java*, de acordo com a **Figura 3**;

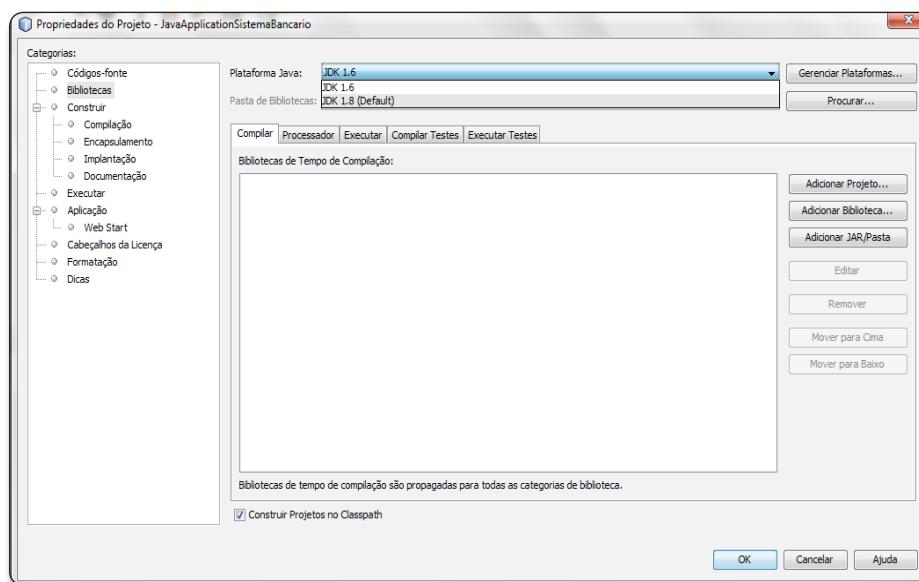


Figura 3. Definindo a versão do Java utilizada pelo projeto

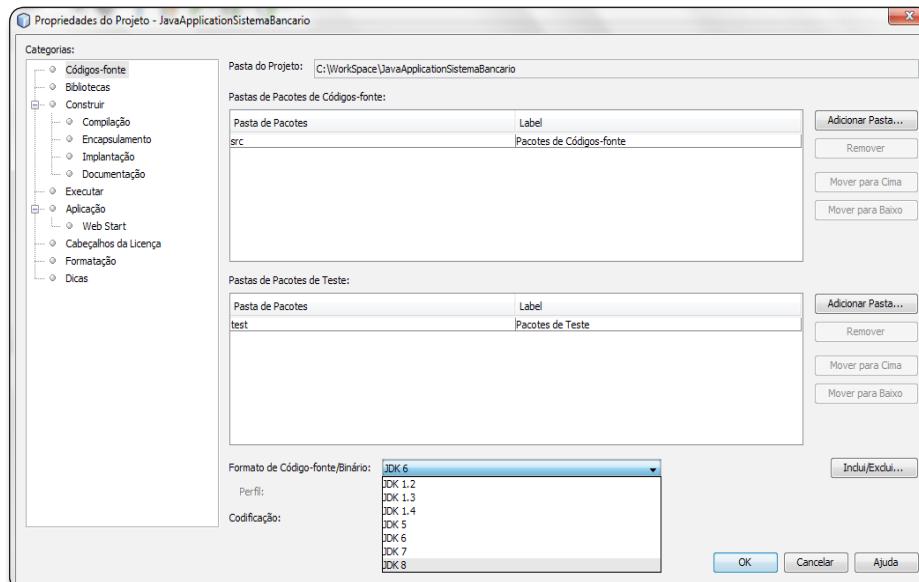


Figura 4. Definindo formato do código-fonte/binário

3. Feito isso, selecione a categoria *Código-fonte* e defina o *Formato de Código-origem/Binário* para *JDK 8*, conforme a **Figura 4**;

4. Por fim, clique em *OK* para salvar as alterações.

Modelo

Com a estrutura do projeto pronta, vamos criar a classe que representará o domínio do nosso sistema, ou seja, a entidade que a aplicação manipulará. Sendo assim, vamos começar criando a classe **Cliente**. No pacote **br.com.cadastro.model**, crie a classe **Cliente** e a codifique conforme a **Listagem 1**.

Nessa classe temos os atributos do cliente e os métodos getters e setters de acesso a esses atributos. A novidade nesse código fica por conta do método *getIdade()*, presente na linha 26, que faz uso da nova API de manipulação de datas, disponível no pacote **java.time**. Esse método recebe como parâmetro a data de nascimento do cliente e retorna sua idade.

Deixamos a data de nascimento no formato antigo, isto é, como instância da classe **java.util.date**, para demonstrar ao leitor como proceder no momento de realizar a conversão de uma data no formato antigo para a nova representação, com **LocalDate**.

Para executar a conversão deve-se utilizar a classe **Instant**. Assim, primeiro obtém-se uma instância dessa classe (**Date.toInstant()**) para, em seguida, criar o objeto **LocalDate** a partir dessa instância, usando o método **LocalDateTime.ofInstant()**. Esses passos podem ser vistos na linha 27.

Na linha 28, o método **now()** da classe **LocalDate** informa a data atual, sem a presença das horas ou minutos. Como temos a data atual e a data de nascimento do cliente, ambos como instância de **LocalDate**, basta subtrair o ano atual pelo ano de nascimento, obtido a partir do método **getYear()**, para conhecer a idade, o que é realizado na linha 29.

Depois de criada a entidade de nossa aplicação, vamos criar as classes e operações de persistência. Sendo assim, no pacote **br.com.cadastro.model.dao**, crie a classe **GenericAbstractCrudDao**

Criando aplicações Desktop em Java

conforme a **Listagem 2**. O objetivo dessa classe abstrata é criar uma camada genérica de acesso ao banco de dados, possibilitando o reaproveitamento e melhor organização do código. A variável **T**, presente na linha 5, representa o tipo da classe que será gerenciada. Já a variável **database**, vista na linha 10, simboliza o banco de dados da aplicação. Observe ainda que as quatro operações do CRUD foram declaradas.

Na **Listagem 3**, a classe **ClienteDAO**, que também deve ser criada no pacote **br.com.cadastro.model.dao**, estende a classe genérica. Assim, ela herda todas as operações declaradas pela superclasse, inclusive o método **listar()**, visto na linha 9, que cria uma lista de clientes da mesma forma como empregado no Java 7. Além disso, duas novas operações foram criadas, a saber: **listarPeloNome()**, na linha 19, e **listarMaioresDeIdade()**, na linha 24. Nesses dois métodos temos uso de EL em conjunto com as coleções do Java.

Agora, note que com apenas duas linhas de código em cada método foi possível filtrar as informações que queríamos sem prejudicar a legibilidade do código. Isso ocorreu graças ao método **filter()** da interface **Stream**, que consegue filtrar uma lista através de uma condição repassada por uma EL.

O método **listarMaioresDeIdade()** retorna uma lista contendo apenas os clientes cadastrados maiores de 18 anos. Para isso, obtemos uma lista de clientes a partir de uma Collection e chamamos o método **stream()** para obter uma instância da interface **Stream**. De posse dessa instância, invocamos o método **filter()** passando uma expressão lambda válida como parâmetro. Essa expressão simboliza uma condição que será aplicada sobre a coleção. Em nosso caso, temos: **c -> c.getIdade(c.getDataNascimento()) > 18**.

Como verificado, uma expressão lambda é dividida em duas partes pelo operador '**->**'. A parte da esquerda é somente a declaração dos parâmetros. No nosso caso, temos o parâmetro **c**, que representa uma instância da classe **Cliente**. A parte da direita representa o corpo da função, como se fosse a implementação do método em si.

Após essa etapa de filtro, foi executado o método **collect()**, para transformar a instância de **Stream** em um **List**.

Como podemos verificar, o uso de EL, juntamente com Collections, facilita bastante a manipulação de estruturas de dados e permite construções poderosas em poucas linhas de código.

Outro recurso que utilizamos na elaboração desse sistema foi o padrão de projeto Facade. Esse padrão permite encapsular todas as informações do sistema em apenas um ponto.

As classes e interfaces do Facade serão agrupadas no pacote **br.com.cadastro.model.facade**. Dito isso, crie a interface **ClienteFacade**, que deve se apresentar conforme a **Listagem 4**. Perceba que nessa interface estão declaradas todas as operações que necessitamos para um cliente. Nesse ponto chegamos a mais uma novidade do Java 8. Você se lembra que nas versões anteriores não era permitido ter métodos concretos declarados em uma interface? A partir dessa versão o desenvolvedor pode evoluir suas interfaces adicionando um *default method* sem se preocupar com a perda de compatibilidade com outras classes que implementam tais interfaces.

Listagem 1. Implementação da classe Cliente.

```
01. package br.com.cadastro.model.domain;
02.
03. import java.io.Serializable;
04. import java.util.Date;
05. import java.time.LocalDate;
06. import java.time.LocalDateTime;
07. import java.time.ZonedDateTime;
08.
09. public class Cliente implements Serializable {
10.
11.     private String nome;
12.     private String cpf;
13.     private Date dataNascimento;
14.     private String telefone;
15.
16.     public Cliente() {
17.     }
18.
19.     public Cliente(String nome, String cpf, Date dataNascimento, String telefone) {
20.         this.nome = nome;
21.         this.cpf = cpf;
22.         this.dataNascimento = dataNascimento;
23.         this.telefone = telefone;
24.     }
25.
26.     public int getIdade(Date dataNascimento) {
27.         LocalDate dataNasc = LocalDateTime.ofInstant(
28.             (dataNascimento.toInstant()), ZonedDateTime.systemDefault()).toLocalDate();
29.         LocalDate dataAtual = LocalDate.now();
30.         return dataAtual.getYear() - dataNasc.getYear();
31.     }
31. } //getters e setters omitidos
```

Listagem 2. Implementação da classe GenericAbstractCrudDao.

```
01. package br.com.cadastro.model.dao;
02.
03. //imports omitidos
04.
05. public abstract class GenericAbstractCrudDao<T extends Serializable> {
06.
07.     public abstract List<T> listar();
08.
09.     private static List<Object> database = null;
10.
11.     public GenericAbstractCrudDao() {
12.         database = new ArrayList<>();
13.     }
14.
15.     public T salvar(T entity) {
16.         getDatabase().add(entity);
17.         return entity;
18.     }
19.
20.     public T editar(T entity) {
21.         getDatabase().remove(entity);
22.         getDatabase().add(entity);
23.         return entity;
24.     }
25.
26.     public void remover(T entity) {
27.         getDatabase().remove(entity);
28.     }
29.
30.     public static List<Object> getDatabase() {
31.         return database;
32.     }
33. }
```

O método `isAniversariante()`, que tem sua declaração iniciada na linha 13, foi criado para demonstrar ao leitor como utilizar o recurso *default methods*. A única diferença em relação à implementação de um método tradicional é a presença da palavra reservada `default` na assinatura do método.

Listagem 3. Implementação da classe ClienteDao.

```

01. package br.com.cadastro.model.dao;
02.
03. //imports omitidos
04.
05. public class ClienteDao extends GenericAbstractCrudDao<Cliente> {
06.
07.     public ClienteDao() {}
08.
09.     @Override
10.     public List<Cliente> listar() {
11.         List<Cliente> clientes = new ArrayList<>();
12.         for (Object o : getDatabase()) {
13.             if (o instanceof Cliente) {
14.                 clientes.add((Cliente) o);
15.             }
16.         }
17.         return clientes;
18.     }
19.
20.     public List<Cliente> listarPeloNome(String nome) {
21.         Stream<Cliente> streamClienteFiltro = this.listar().stream();
22.         return streamClienteFiltro.filter(c -> !c.getNome().isEmpty() && c.getNome().toLowerCase().contains(nome.toLowerCase())).collect(Collectors.toList());
23.     }
24.
25.     public List<Cliente> listarMaioresDedadade() {
26.         Stream<Cliente> streamClienteFiltro = this.listar().stream();
27.         return streamClienteFiltro.filter(c -> c.getIdade(c.getDataNascimento()) > 18).collect(Collectors.toList());
28.     }

```

Listagem 4. Implementação da interface ClienteFacade.

```

01. package br.com.cadastro.model.facade;
02.
03. //imports omitidos
04.
05. public interface ClienteFacade {
06.
07.     Cliente salvar(Cliente cliente);
08.     void remover(Cliente cliente);
09.     Cliente editar(Cliente cliente);
10.     List<Cliente> listarTodos();
11.     List<Cliente> listarPeloNome(String nome);
12.
13.     public default boolean isAniversariante(Cliente c) {
14.         LocalDate dataAtual = LocalDate.now();
15.         LocalDate dataNasc = LocalDateTime.ofInstant(c.getDataNascimento().toInstant(), ZoneId.systemDefault()).toLocalDate();
16.         Period periodo = Period.between(dataAtual, dataNasc);
17.         return periodo.getDays() == 0 && periodo.getMonths() == 0;
18.     }
19. }

```

Outro caso que ilustra esse recurso do Java 8 é o método `stream()`, adicionado na interface **Collection**. Isso foi feito para fornecer o suporte às expressões lambda. Para adicionar o método `stream()` na interface sem quebrar as implementações existentes

de **Collection** em todo o mundo, o Java adicionou o `stream()` como um método default da interface, fornecendo assim uma implementação padrão. Com isso, temos a opção de implementar o método `stream()` ou, se preferir, usar a implementação padrão já oferecida pelo Java.

No pacote `br.com.banco.model.facade.impl` deve ser criada a classe `ClienteFacadeImpl`, que implementa a interface `ClienteFacade`, conforme a **Listagem 5**. Nessa classe utilizada como fachada, encapsulamos toda a complexidade do CRUD sobre o cadastro de clientes oferecendo uma interface simples e unificada, evitando acoplamento e complexidade. Através dela, sabe-se, por exemplo, que o método `salvar()` recebe um objeto do tipo `Cliente` como parâmetro e internaliza esse objeto no banco de dados. Toda complexidade existente para realizar essa ação fica escondida através da fachada.

Controlador

Para darmos continuidade ao desenvolvimento, criaremos o controle do fluxo da aplicação. O controle vem para gerenciar a comunicação entre os demais componentes e controlar o fluxo de dados, regras de negócios e ações dos usuários.

Portanto, no pacote `br.com.cadastro.controller`, **Crie a classe ClienteController**. Um pequeno trecho de código, apresentado na **Listagem 6**, nos ajudará a entender a relação existente entre a camada de controle e a camada de visão. O restante do código pode ser obtido no site da revista Easy Java Magazine. A partir da compreensão desse trecho é possível estender esse conhecimento para as demais operações do CRUD.

A anotação `@FXML`, presente na linha 1, indica que essa operação tem um componente correspondente (de mesmo nome) na camada de visão. Já a variável `c`, definida na linha 3, armazena uma instância de um objeto do tipo `Cliente` e da linha 4 até a linha 11 as propriedades do cliente são recuperadas da GUI e armazenadas na instância criada. Perceba que na linha 10 é feita uma validação sobre o texto digitado no campo data de nascimento para verificar se a data inserida está no formato esperado. Na linha 12, o objeto `Cliente` é salvo no repositório através da chamada ao método `salvar()`. Em seguida, os campos de entrada de dados da interface com o usuário são limpos e a lista de clientes cadastrados é atualizada. Na seção seguinte veremos como o JavaFX faz para invocar o método `salvar()`.

Outra operação que analisaremos é a `selecionarCliente()`, exposta na linha 22. Note que o único valor passado como parâmetro para esse método é do tipo `MouseEvent`. Esse evento é disparado quando ocorre uma interação do mouse com algum componente da tela. Quando isso ocorre, a variável `tblClientes`, utilizada na linha 23 e do tipo `TableView`, a partir dos métodos `getSelectionModel()` e `getSelectedItem()`, recupera o `Cliente` selecionado pelo clique do mouse. Nas linhas seguintes as propriedades do cliente são repassadas para os campos de texto correspondentes.

Visão

Deste modo a camada de back-end da nossa aplicação está finalizada. Precisamos agora desenvolver o front-end, isto é, a

Criando aplicações Desktop em Java

Listagem 5. Implementação da classe ClienteFacadeImpl.

```
01. package br.com.cadastro.model.facade.impl;
02.
03. //imports omitidos
04.
05. public class ClienteFacadeImpl implements ClienteFacade {
06.
07.     private ClienteDao clienteDao;
08.
09.     public ClienteFacadeImpl() {
10.         clienteDao = new ClienteDao();
11.     }
12.
13.     @Override
14.     public Cliente salvar(Cliente cliente) {
15.         return clienteDao.salvar(cliente);
16.     }
17.
18.     @Override
19.     public List<Cliente> listarTodos() {
20.
21.
22.         @Override
23.         public void remover(Cliente cliente) {
24.             clienteDao.remover(cliente);
25.         }
26.
27.         @Override
28.         public Cliente editar(Cliente cliente) {
29.             return clienteDao.editar(cliente);
30.         }
31.
32.         @Override
33.         public List<Cliente> listarPeloNome(String nome) {
34.             return clienteDao.listarPeloNome(nome);
35.         }
36.     }
```

Listagem 6. Trecho da classe ClienteController.

```
01. @FXML
02. public void salvar() throws ParseException {
03.     Cliente c = new Cliente();
04.     c.setNome(txtNome.getText());
05.     c.setCpf(txtCPF.getText());
06.     c.setTelefone(txtTelefone.getText());
07.
08.     if (!txtDataNascimento.getText().isEmpty()) {
09.         Date releaseDate = dataFormatter.parse(txtDataNascimento.getText());
10.        c.setDataNascimento(releaseDate);
11.    }
12.    clienteFacade.salvar(c);
13.    txtNome.clear();
14.    txtCPF.clear();
15.    txtTelefone.clear();
16.    txtDataNascimento.clear();
17.    lblValidation.setError(null);
18.    tblClientes.getItems().addAll(clienteFacade.listarTodos());
19. }
20
21. @FXML
22. public void selecionarCliente(MouseEvent arg0) {
23.     Cliente c = tblClientes.getSelectionModel().getSelectedItem();
24.     txtNome.setText(c.getNome());
25.     txtCPF.setText(c.getCpf());
26.     txtTelefone.setText(c.getTelefone());
27.     txtDataNascimento.setText(dataFormatter.format(c.getDataNascimento()));
28. }
```

classe que será responsável pela interação com o usuário. Para isso, utilizaremos JavaFX. Apesar de não ser o foco deste artigo, o uso dessa tecnologia para construir GUIs pode agregar uma experiência visual mais interessante para o usuário e trazer mais produtividade para o desenvolvedor. Ela permite a criação de interfaces ricas com o uso de efeitos, além de dispor de uma API completa para criação de telas orientadas a componentes.

O JavaFX traz suporte ao FXML, uma forma de declarar todos os elementos de interface sem escrever uma linha de código Java. A grande vantagem disso está na possibilidade de usar uma ferramenta para geração da interface e a possibilidade de modificar o XML sem ter que recompilar a aplicação inteira.

Para criar o FXML os seguintes passos são necessários no NetBeans:

1. Clique com o botão direito na aplicação JavaApplicationCRUD;
2. Em seguida, clique em *Novo* e escolha a opção *Outros*;
3. Na coluna *Categoria*, escolha *JavaFX*, e na coluna *Tipos de Arquivos*, escolha *FXML vazio*.

O código desse arquivo está disponível na **Listagem 7**. Sobre o FXML produzido, alguns pontos merecem atenção; São eles:

- A tag **<AnchorPane>**, utilizada na linha 5, sinaliza um layout para a tela. Dentro dessa tag estão todos componentes adicionados ao cadastro (botão, campo de texto, etc.);
- O atributo **fx:controller**, visto na linha 6, é usado para associar uma classe de controle ao documento, sendo responsável por coordenar o comportamento dos componentes da interface;
- O controle **TableView**, na linha 17, indica que linhas e colunas serão desenhadas criando uma tabela. Nesta tabela, o evento **OnMouseClicked** define que a operação **selecionarCliente()** da classe **Controller** será chamada quando um botão do mouse for pressionado dentro dela.

Finalizada a criação do documento FXML, o próximo passo é criar a classe que conterá um objeto representando a hierarquia desse documento, isto é, a tela da aplicação. Assim sendo, os seguintes passos devem ser executados:

1. Clique com o botão direito na aplicação JavaApplicationCRUD e depois clique em *Novo*;
2. Em seguida, escolha a opção *Classe Principal do JavaFX*;
3. Por fim, defina o nome da classe como **CadastroView** e o pacote como **br.com.cadastro.view**.

Após criar a classe, a implemente conforme o código da **Listagem 8**. Nessa listagem, o **FXMLLoader** lê o arquivo de extensão **.fxml** e retorna um objeto do tipo **Parent**, como expõe a linha 12.

Com esse objeto é possível configurar a raiz da cena da aplicação. Na linha 13, observa-se a presença das classes **javafx.scene.Scene** (cena) e **javafx.stage.Stage** (palco). Estes nomes fazem uma analogia com apresentações artísticas e denotam como as interfaces de usuário são tratadas no JavaFX, onde as telas são exibidas por meio de “cenas” apresentadas em um “palco”, representado pelo parâmetro **Stage**.

Como o foco do artigo é outro e JavaFX é apenas um auxiliar no desenvolvimento de nosso sistema, não aprofundaremos no estudo desta tecnologia e mencionaremos apenas alguns pontos que julgamos de interesse do leitor.

O coração desta tecnologia está presente no documento FXML. Nele, estão todos os recursos visuais e de interação com o usuário.

Dependendo da complexidade do projeto, pode se tornar trabalho-só codificar o arquivo manualmente. Pensando nisso, foi criada uma ferramenta de design visual que auxilia o desenvolvedor a gerar o arquivo XML. Trata-se do Scene Builder. Com ele, o único trabalho do desenvolvedor é arrastar e soltar os componentes na tela.

A **Figura 5** mostra a tela de cadastro que obtemos a partir da execução da aplicação.

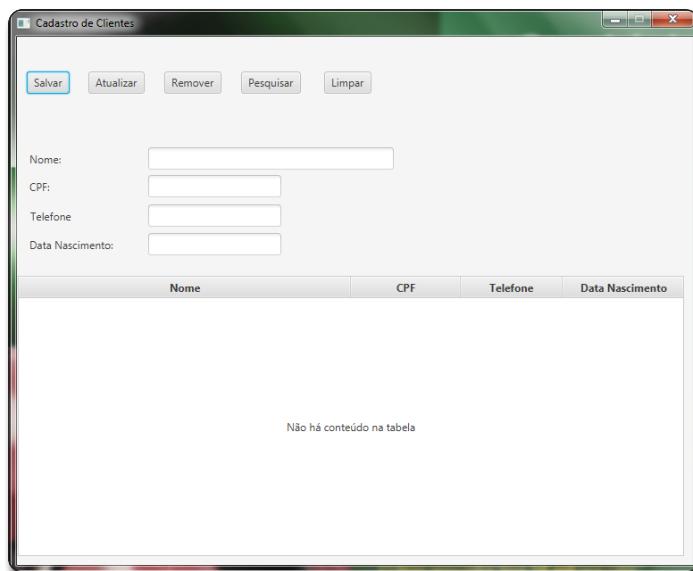


Figura 5. Tela de cadastro de clientes

Certamente o recurso mais aguardado da nova versão do Java foram as expressões lambda. Além de trazer conceitos de outras linguagens relacionadas ao paradigma funcional, esse recurso provocou alterações na forma de codificar e na sintaxe da linguagem. Isso, em um primeiro momento, pode gerar estranheza e resistência. Porém esse novo recurso veio para tornar mais fácil a implementação de determinadas tarefas que necessitavam de muitas linhas de código, viabilizando mais produtividade ao trabalho do desenvolvedor e resultando em código mais claro de ler e simples de manter.

Outra novidade que deixou os desenvolvedores Java satisfeitos foi a tão esperada Date and Time API, que conta com uma interface

Listagem 7. Implementação do Arquivo XML.

```

01. <?xml version="1.0" encoding="UTF-8"?>
02.
03. //imports omitidos
04.
05. <AnchorPane id="AnchorPane" prefHeight="579.0" prefWidth="746.0"
06. xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1"
07. fx:controller="br.com.cadastro.controller.ClienteController">
08.   <children>
09.     <Button id="btnSalvar" layoutX="11.0" layoutY="39.0"
10.       mnemonicParsing="false" onAction="#salvar" text="Salvar" />
11.     <Label layoutX="14.0" layoutY="129.0" text="Nome:" />
12.     <Label layoutX="14.0" layoutY="160.0" text="CPF:" />
13.     <Label layoutX="14.0" layoutY="225.0" text="Data Nascimento:" />
14.     <Label layoutX="15.0" layoutY="193.0" text="Telefone:" />
15.     <TextField fx:id="txtNome" layoutX="147.0" layoutY="124.0"
16.       prefHeight="25.0" prefWidth="275.0" />
17.     <TextField fx:id="txtCPF" layoutX="147.0" layoutY="155.0" />
18.     <TextField fx:id="txtTelefone" layoutX="147.0" layoutY="188.0" />
19.     <TextField fx:id="txtDataNascimento" layoutX="147.0" layoutY="220.0" />
20.     <TableView fx:id="tblClientes" layoutX="1.0" layoutY="268.0" onMouseClicked=
21.       "#selecionarCliente" prefHeight="314.0" prefWidth="746.0" />
22.     <columns>
23.       <TableColumn fx:id="clNome" prefWidth="372.0" text="Nome" />
24.       <TableColumn fx:id="clCpf" prefWidth="123.0" text="CPF" />
25.       <TableColumn fx:id="clTelefone" prefWidth="114.0" text="Telefone" />
26.       <TableColumn fx:id="clDataNascimento" prefWidth="136.0"
27.         text="Data Nascimento" />
28.     </columns>
29.   </AnchorPane>
30.   <children>
31.     <Label fx:id="lblValidationError" contentDisplay="CENTER"
32.       prefHeight="29.0" prefWidth="707.0" textAlignment="RIGHT"
33.       textFill="#ee0c0c" />
34.     <font>
35.       <Font size="16.0" />
36.     </font>
37.   </children>
38. </children>;

```

Listagem 8. Implementação da tela de cadastro.

```

01. //imports omitidos;
02.
03. public class CadastroView extends Application {
04.
05.   public static void main(String[] args) {
06.     launch();
07.   }
08.
09.   @Override
10.   public void start(Stage palco) throws Exception {
11.     URL arquivoFXML = getClass().getResource("./cadastroView.fxml");
12.     Parent fxmlParent = (Parent) FXMLLoader.load(arquivoFXML);
13.     palco.setScene(new Scene(fxmlParent));
14.     palco.setTitle("Cadastro de Clientes");
15.     palco.setResizable(false);
16.     palco.show();
17.   }
18.

```

Criando aplicações Desktop em Java

fluente, além de ser totalmente imutável, ou seja, as datas não podem ser modificadas após sua criação. Assim, sempre que adicionamos ou subtraímos dias, por exemplo, um novo objeto é criado, do mesmo modo que ocorre com uma String.

Autor



Carlos Alberto Silva

casilvamg@hotmail.com

É Formado em Ciência da Computação pela Universidade Federal de Uberlândia (UFU), com especialização em Desenvolvimento Java pelo Centro Universitário do Triângulo (UNITRI). Trabalha na empresa Algar Telecom como Analista de Sistemas e atualmente é aluno do curso de especialização em Análise e Desenvolvimento de Sistemas Aplicados a Gestão Empresarial no Instituto Federal do Triângulo Mineiro (IFTM). Possui as seguintes certificações: OCJP, OCWCD e ITIL.



Autor



Lucas de Oliveira Pires

pires.info@gmail.com

É Formado em Sistemas de Informação pelo Centro Universitário do Triângulo (UNITRI) com especialização em Análise e Desenvolvimento de Sistemas Aplicados à Gestão Empresarial (IFTM). Trabalha na empresa Algar Telecom como Analista de Desenvolvimento.



Links:

Endereço para download do Java 8.

<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

Endereço para download do NetBeans 8.0.1.

<https://netbeans.org/downloads/>

Conteúdo sobre JavaFX.

<http://www.oracle.com/technetwork/pt/java/javafx/tech/index.html>

Exemplo de uso da API Date and Time.

<http://java.dzone.com/articles/java-8-apis-javauitime>

DÊ UM SALTO EM CONHECIMENTO!



Acesse o maior
portal para
desenvolvedores
da América
Latina!

20
mil
posts

430
mil
cadastrados

10
milhões de
page-views
por mês

Programação paralela em Java

Conheça as vantagens e ferramentas disponíveis no desenvolvimento de aplicações paralelas em Java

A evolução do poder de processamento de computadores e dispositivos cresceu muito em relação ao que possuímos alguns anos atrás. Por volta de 1965, ainda não existia uma expectativa real do mercado de hardware e seu crescimento, até que Gordon E. Moore, na época presidente da Intel, fez uma profecia sobre a capacidade de processamento, a qual dobraria a cada 24 meses e mantendo o mesmo custo, que ficou conhecida como **lei de Moore**. Poucos anos depois foi possível afirmar que a sua previsão estava correta e o padrão se manteve até os dias atuais.

Desta forma, por muitos anos, desenvolvedores de aplicativos têm se aproveitado desse progresso, não se preocupando em realizar melhorias no código para aumentar o desempenho e eficiência do *software*. Assim, os sistemas começaram a ficar cada vez mais rápidos apenas com a troca do hardware e as novas tecnologias de processador.

Nessa evolução do hardware, o aumento ficou direcionado ao *clock* do processador, isto é, à frequência que o processador executa as tarefas, passando de 500MHz, 1GHz, até os atuais processadores de 3GHz.

No entanto, o aumento da velocidade impacta em um aumento no consumo de energia e a dissipação de calor, o que vai na contramão da produção de processadores cada vez mais compactos para dispositivos também cada vez menores e que tenham um aproveitamento eficiente de energia.

Assim, não seria mais possível aumentar a velocidade dos processadores. Logo, a solução seria aumentar a quantidade de núcleos nos processadores. Essa transformação se deu com o surgimento da tecnologia Multicore. Com essa tecnologia, um único *chip* é capaz de acomodar duas ou mais unidades de processamento.

Para o sistema operacional, cada núcleo é visto como um processador diferente e a adição de novos núcleos

Fique por dentro

Este artigo apresenta os conceitos de programação paralela, bem como a implementação de código para execução paralela utilizando a linguagem Java. Para isso, serão abordados os frameworks Executor e Fork/Join e as novidades relacionadas a esse tipo de processamento considerando as últimas versões da linguagem. Com a correta utilização de frameworks como o Fork/Join, podemos melhorar o desempenho e a agilidade na escrita de códigos e sua execução em computadores multicore.

de processamento permite que as instruções das aplicações sejam processadas em paralelo, em vez da execução sequencial.

Com essa nova realidade, desenvolvedores de aplicativos, para acompanhar essa mudança de paradigma, precisarão alterar a maneira que escrevem seus códigos, passando da abordagem sequencial para a paralela.

Felizmente, a linguagem Java desde o seu início, foi projetada para dar suporte ao desenvolvimento de aplicações multitarefa e a facilitar essas implementações por parte dos desenvolvedores. A utilização de Threads, para criação de tarefas que deveriam executar ao mesmo tempo, foi e ainda continua sendo uma das maneiras mais simples de se escrever uma aplicação concorrente.

Com o crescimento da linguagem, novas características foram adicionadas visando melhorar o tratamento de tarefas concorrentes. A primeira grande mudança aconteceu na versão 5. Até então era o programador que controlava a criação, execução e o estado das *threads*. Com o Java 5 e a inclusão do framework *Executor*, a criação, gerência e finalização das *threads* passou a ficar a cargo da API.

Posteriormente no Java 7, a novidade veio com o lançamento do framework *Fork/Join*, que ajuda a simplificar ainda mais escrita de código paralelo, permitindo o aproveitamento de todos os recursos oferecidos por múltiplos processadores.

Programação paralela em Java

E em sua versão mais recente, o Java 8, com os novos recursos de programação funcional, a linguagem permite a produção de código paralelo, limpo e com qualidade.

Logo, o conhecimento e entendimento do funcionamento e desenvolvimento de aplicações paralelas deixa de ser apenas um diferencial e passa a ser uma necessidade. Com base nisso, este artigo irá descrever os conceitos da programação paralela, analisando e utilizando as ferramentas disponibilizadas pela linguagem Java.

Arquiteturas paralelas

A capacidade de processar informações e retornar resultados é o que torna os computadores a ferramenta ideal para auxiliar na resolução de cálculos e problemas complexos. Entretanto, quanto maiores os problemas ou a quantidade de informações a serem processadas, mais recursos serão necessários, assim como, computadores com maior poder de processamento. Com o intuito de solucionar ou amenizar esse problema, surgiram algumas arquiteturas computacionais que buscam parallelizar a execução de tarefas complexas e que demandam muito tempo. Para isso, essas arquiteturas são capazes de dividir um mesmo problema em problemas menores para serem resolvidos individualmente. Dentro dessas arquiteturas, nos subtópicos a seguir analisaremos algumas das mais utilizadas.

Arquitetura com multiprocessadores

Uma maneira de se adquirir o paralelismo é utilizar uma arquitetura de múltiplos processadores. Na **Figura 1**, cada P representa uma unidade de processamento, que se comunica com a memória principal (M). A memória principal, por sua vez, é compartilhada entre todos os processadores, podendo ser única ou dividida em vários módulos. O acesso à memória principal é realizado através da rede de interconexão.

Arquitetura com multicamputadores

Outra forma de arquitetura paralela é a de multicamputadores. Neste modelo, cada unidade de processamento tem sua própria memória principal. Desta forma, como observamos na **Figura 2**, o processador *P1* se liga com a memória principal *M1*. Nesta proposta, nenhum processador poderá ter acesso diretamente a um módulo de memória que não esteja diretamente relacionado a ele. Assim, a comunicação entre as unidades computacionais é realizada por troca de mensagens através da rede de interconexão.

A vantagem dessa arquitetura está na possibilidade de se adicionar um número grande de computadores à rede. Entretanto, o aumento de desempenho em nível de aplicação irá depender do desenvolvimento do software paralelo.

Arquitetura multicore

Para obtermos melhor desempenho podemos aumentar o número de processadores utilizando a arquitetura de Multicamputadores. Esta solução é adotada por várias empresas que possuem grandes datacenters, como Facebook, Google e Microsoft, que

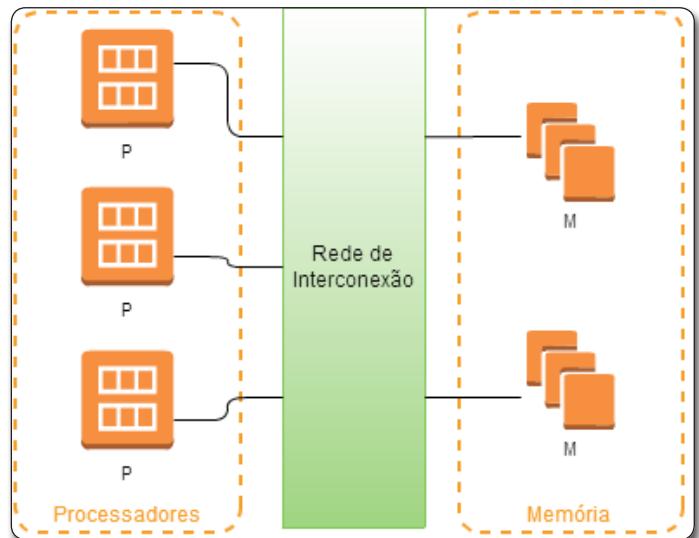


Figura 1. Arquitetura Multiprocessadores

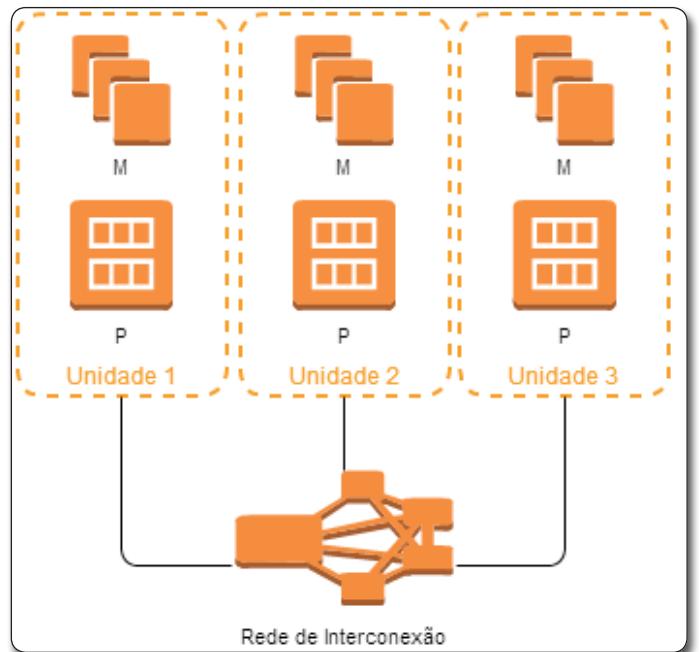


Figura 2. Arquitetura Multicomputadores

precisam processar uma grande quantidade de informações no menor tempo possível.

Atualmente, com computadores/dispositivos pessoais cada vez menores e com maior poder de processamento como: notebooks, tablets e smartphones, existe a necessidade de obter maior desempenho sem prejudicar o seu espaço físico, isto é, sem torná-los grandes demais. Para manter a criação de processadores menores e com um melhor desempenho computacional, surgiu a tecnologia multicore. Nesta tecnologia temos uma solução importante, que é a capacidade de inserir duas ou mais unidades de processamento (CPU) em um mesmo chip.

Para o sistema operacional, cada núcleo é tratado como se fosse um processador diferente, com seus próprios recursos de execução. Assim, diversos núcleos de processamento replicarão os recursos de um processador.

Na **Figura 3** podemos ver a capacidade de simultaneidade, em que duas aplicações são processadas ao mesmo tempo em núcleos diferentes. Com isso, pode-se ter uma janela do seu navegador aberta e estar navegando, enquanto outra está carregando e processando um vídeo.

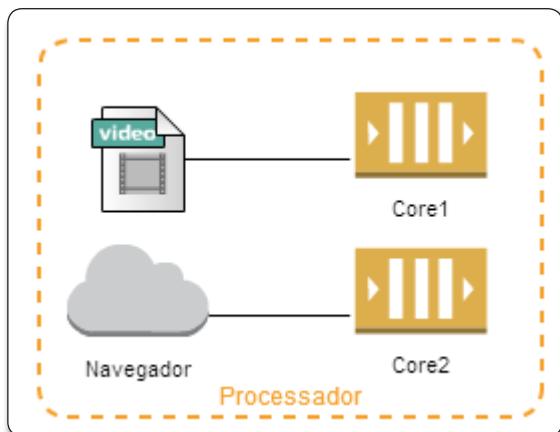


Figura 3. Arquitetura Multicore

Nos últimos anos tivemos uma grande popularização da arquitetura **multicore**. Mas, para podemos aproveitar os ganhos de performance trazidos por essa tecnologia, as aplicações devem ser desenvolvidas para execução em paralelo.

Programação paralela e concorrente

Em geral, quando desenvolvemos uma aplicação, a menos que seja um requisito essencial, não nos preocupamos muito com o seu tempo de processamento, paralelismo ou concorrência. No entanto, em aplicações que envolvem análise de dados, como processamento de imagens, visão computacional, bioinformática, astronomia e meteorologia, se preocupar com o desempenho passa a ser essencial.

O aumento cada vez mais surpreendente do volume de dados e a complexidade da informação envolvida tornam o desenvolvimento de software uma tarefa cada vez mais complexa, sendo necessário otimizar o código e aplicar de maneira correta algoritmos que possam dividir a carga de trabalho e o processamento dos dados.

Como sabemos, para executar um aplicativo de forma paralela ou concorrente, é necessário separar o código em subconjuntos de instruções independentes com o objetivo de serem executados por threads diferentes, uma para cada subconjunto.

Threads e Multithreading

A interface `java.lang.Runnable` é quem define a assinatura do método `run()`. Deste modo, por contrato, a implementação desse

método passa a ser uma condição necessária para que uma classe tenha objetos que possam ser executados em paralelo. Por sua vez, a classe concreta `java.lang.Thread` implementa a interface `java.lang.Runnable` e as instruções definidas no `run()` dessa classe é que serão executadas de forma paralela pela JVM.

Com as threads podemos realizar a divisão de tarefas e executar de maneira concorrente/simultânea trechos distintos de um mesmo aplicativo, com a finalidade de diminuir o tempo de retorno do processamento de tarefas grandes e/ou complexas.

A implementação de uma aplicação pode disparar a chamada de várias *threads*, sendo que todas elas compartilham o mesmo espaço de memória, inclusive as mesmas variáveis, podendo trocar mensagens entre si durante suas execuções.

Ao utilizarmos threads, passamos a ter a execução da aplicação de maneira concorrente ou paralela. A execução de maneira concorrente simula a simultaneidade, passando a impressão de que várias tarefas estão sendo processadas ao mesmo tempo. Na verdade, no entanto, o que ocorre é o escalonamento das threads no processador. Por sua vez, na execução de maneira paralela não existe o escalonamento, pois esta irá aproveitar todos os processadores disponíveis. Assim, as tarefas realmente serão computadas simultaneamente.

Neste contexto, seja usando execução concorrente ou paralela, é possível que uma *thread* crie novas *threads* que sejam responsáveis pela execução de partes da tarefa inicial como uma forma de aumentar o desempenho. A **Figura 4** ilustra o conceito de uma *thread* mestre delegando tarefas para *threads* filhas.

O código da **Listagem 1** demonstra a criação e início de execução de uma *thread* em Java que irá exibir uma simples mensagem.

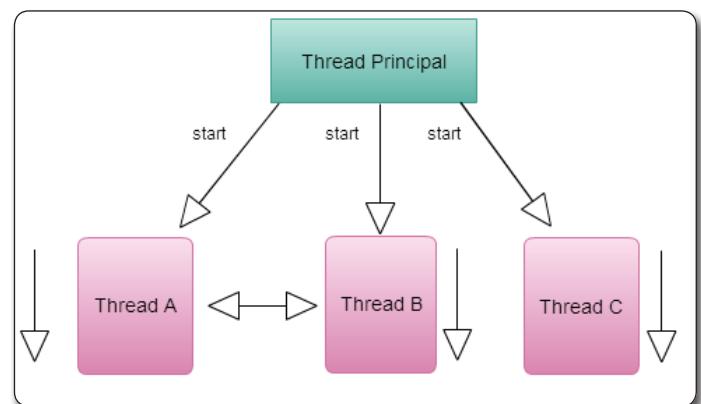


Figura 4. Delegação de tarefas entre Threads

Sincronização entre threads

Devido ao fato das *threads* compartilharem as mesmas variáveis e dados, quando uma *thread* estiver lendo o valor de uma variável e simultaneamente outra *thread* a estiver atualizando, inconsistências podem ser geradas nos dados.

Para garantir a consistência, o gerenciamento da execução de várias *threads* em um mesmo aplicativo é realizado através da sincronização. Para que cada *thread* tenha acesso exclusivo a determinado recurso é necessário aplicar uma estratégia de sincro-

Programação paralela em Java

nização entre elas, como os monitores. Desta forma os dados que forem compartilhados permanecerão consistentes e cada *thread* realizará sua atividade corretamente.

Um monitor é um tipo especial de módulo ou objeto cujos métodos só podem ser executados por uma *thread* de cada vez. Na linguagem Java, a palavra reservada **synchronized** possibilita o uso de monitores e pode ser adotada de duas formas: na declaração de métodos ou no início de um bloco de código.

Listagem 1. Código da classe ThreadJava.

```
1.package thread;
2.
3.public class ThreadJava {
4.
5.    public static void main(String[] args) throws InterruptedException {
6.        Thread thread = new Thread() {
7.            @Override
8.            public void run() {
9.                System.out.println("Criando uma Thread em Java!");
10.            }
11.        };
12.        thread.start();
13.    }
14.}
```

O uso do termo **synchronized** sobre um bloco de código determina que uma *thread*, ao acessar esse bloco, deverá consultar o monitor e solicitar uma permissão para que ela possa ter acesso ao mesmo, já que apenas uma *thread* por vez poderá ter acesso a um bloco sincronizado para executar as rotinas ali definidas, evitando, por exemplo, que diversas *threads* tentem incrementar o valor de uma variável simultaneamente.

Além disso, ainda podemos controlar o estado das *threads* usando as instruções **notify()**, **notifyAll()** e **wait()**, disponíveis em **java.lang.Object**, como uma forma de gerenciar a execução das tarefas e possivelmente prever a ocorrência de *deadlocks*.

A invocação do método **wait()** em uma *thread* faz com que essa *thread* entre no estado de espera e aguarde até que seja invocado o método **notify()** ou **notifyAll()** a partir de uma outra *thread* do mesmo contexto. Com isso a *thread* com maior tempo de espera é liberada do bloqueio condicional ao qual foi submetida quando usado o **notify()** ou todas as *threads* são liberadas quanto utilizado o método **notifyAll()**.

Na **Listagem 2** temos um exemplo de uso da sincronização entre *threads*. Para isso, criamos uma classe denominada **SomaSincronizada** para calcular a soma dos valores de uma lista e armazenar o resultado na variável global **total**.

Inicialmente, criamos o método **getValues()** que irá retornar uma lista com três valores para obtermos alguns números para testes. Em seguida, definimos que a nossa classe seja herdeira da classe **Thread**, por meio do operador **extends**. Depois, sobrescrevemos o método **public void run()**, responsável pela execução da *thread*, no qual percorremos a lista de **getValues()**. Para cada elemento retornado por esse método, somaremos o seu valor na variável **total**. Esse processo deverá ser feito dentro de um bloco

synchronized, para o qual é definido o parâmetro **this**, que informa que a *thread* (objeto) corrente será usada como monitor. Quando terminamos de realizar o cálculo, invocamos o método **notify()** da *thread* em ação para devolver a execução a alguma *thread* que eventualmente esteja aguardando.

Listagem 2. Código da classe SomaSincronizada.

```
01.package thread;
02.
03.import java.math.BigDecimal;
04.import java.util.ArrayList;
05.import java.util.List;
06.
07.public class SomaSincronizada extends Thread {
08.
09.    protected BigDecimal total = BigDecimal.ZERO;
10.
11.    @Override
12.    public void run() {
13.        synchronized (this) {
14.            for (BigDecimal value: getValues()) {
15.                total = total.add(value);
16.            }
17.            notify();
18.        }
19.    }
20.
21.    private List<BigDecimal> getValues() {
22.        List<BigDecimal> list = new ArrayList<>();
23.        list.add(new BigDecimal(13));
24.        list.add(new BigDecimal(12));
25.        list.add(new BigDecimal(16));
26.        return list;
27.    }
28.
29.    public static void main(String[] args) throws InterruptedException {
30.        SomaSincronizada st = new SomaSincronizada();
31.        st.start();
32.        synchronized (st) {
33.            st.wait();
34.            System.out.println("O resultado é: " + st.total);
35.        }
36.    }
37.}
```

No método **main()** da classe **SomaSincronizada** criamos um objeto **st** com o objetivo de testar a implementação realizada. A nova *thread* **st** dispara o método **start()** e, em seguida, a *thread* principal entra em um bloco sincronizado. Neste bloco, a primeira instrução é a chamada do método **wait()**. Assim, a *thread* principal entra em estado de espera até que outra *thread*, neste caso a *thread* **st**, faça a chamada do método **notify()**, e então continua a sua execução, na linha 34, exibindo o resultado da soma. Em nosso exemplo, o valor apresentado será **41**.

Na **Listagem 3** implementamos o mesmo exemplo, mas sem a sincronização. Deste modo, se utilizarmos a classe **SomaNaoSincronizada**, o valor de saída poderá ser diferente a cada execução. Isso se deve ao fato que o método **main()**, como no caso anterior, executa a *thread* principal do programa, que poderá terminar antes da *thread* **st**, que calcula o resultado da soma. Com isso um valor incorreto será exibido.

Listagem 3. Código da classe SomaNaoSincronizada.

```
01.package thread;
02.
03.import java.math.BigDecimal;
04.import java.util.ArrayList;
05.import java.util.List;
06.
07.public class NotificaThreadNaoSincronizado extends Thread {
08.
09.    protected BigDecimal valor = BigDecimal.ZERO;
10.
11.    @Override
12.    public void run() {
13.        for (BigDecimal value: getValues()) {
14.            valor = valor.add(value);
15.        }
16.    }
17.
18.    private List<BigDecimal> getValues() {
19.        List<BigDecimal> list = new ArrayList<>();
20.        list.add(new BigDecimal(13));
21.        list.add(new BigDecimal(12));
22.        list.add(new BigDecimal(16));
23.        return list;
24.    }
25.
26.    public static void main(String[] args) throws InterruptedException {
27.        SomaNaoSincronizada st = new SomaNaoSincronizada();
28.        st.start();
29.        System.out.println("O resultado é: " + st.valor);
30.    }
31.}
```

Executor framework

Normalmente, ao desenvolver uma simples aplicação concorrente em Java, implementa-se objetos **Runnable** e também objetos da classe **Thread**. Deste modo, o desenvolvedor é quem controla a criação, execução e o estado das threads no programa. Essa manipulação, como visto na **Listagem 1**, é eficaz para resolver pequenos problemas. No entanto, códigos que implementam operações complexas podem ser mais difíceis de ler e gerenciar, e consequentemente, deixar a cargo do desenvolvedor o controle sobre as threads pode “abrir as portas” para diversos erros.

De olho nesse problema, na versão 5, a linguagem Java introduziu melhorias com o objetivo de simplificar a vida do desenvolvedor. Estas melhorias foram em um novo pacote, chamado **java.util.concurrent**.

Com este novo pacote temos o framework **Executor**, que separa a criação da execução das tarefas. Desta forma, o desenvolvedor apenas precisa implementar um objeto da classe **Runnable** e usar um objeto da classe **Executor**. O framework irá encapsular e será responsável pela criação, gerência e finalização das *threads* para a execução das tarefas.

O objetivo principal deste recurso é a separação de responsabilidades. O Executor abstrai todo o gerenciamento de atividades das *threads*, não sendo necessária a criação de uma *thread* por parte do desenvolvedor, deixando para este apenas a parte que lhe convém: o desenvolvimento do código concorrente, ou seja, a implementação da classe **Runnable**.

Também temos a introdução do *pool de threads*. Como sabemos, a criação de *threads* é um processo custoso em relação ao consumo de recursos, como memória e performance de execução. O *pool de threads* diminui o impacto da criação de *threads*, mantendo uma fila com tarefas a serem executadas e um conjunto de *threads* executoras reutilizáveis.

O pacote de concorrência oferece várias interfaces e novos recursos. Entre eles, podemos destacar:

- **Executor**: uma simples interface com o método **execute()**, que recebe como parâmetro uma implementação de **Runnable**. A interface **Executor** é a abstração principal do pacote **java.util.concurrent** e serve de base para todo o framework;
- **ExecutorService**: representa uma extensão da interface **Executor** que inclui diversos recursos para gerenciar o processo de execução das tarefas submetidas ao pool, desde o seu estado atual de execução até o cancelamento da atividade;
- **ScheduledExecutorService**: representa uma extensão da interface **java.util.concurrent.Executor** que permite agendar execuções de tarefas. É muito útil quando precisamos que uma determinada *thread* seja executada em ciclos programados, como a atualização ou sincronização automatizada do sistema, leitura de um arquivo ou comunicação com sistemas legados.

Para utilizarmos a interface **java.util.concurrent.Executor**, precisamos de uma implementação concreta da interface **java.util.concurrent.ExecutorService**. Com esse intuito, pode ser adotada a classe **java.util.concurrent.Executors**, que disponibiliza alguns métodos que retornam objetos da interface **ExecutorService**. Alguns desses métodos são analisados a seguir:

- **newSingleThreadExecutor()**: cria uma implementação de **ExecutorService** (como mostrado na **Listagem 4**) com apenas uma thread trabalhadora (*Worked Thread*). No nosso exemplo, como apenas exibiremos uma mensagem, nenhum processamento simultâneo foi necessário. Sendo assim, criamos um **ExecutorService** e invocamos o método **execute()** passando o trecho de código que deverá ser executado pela thread criada pelo executor. Por último, invocamos o método **shutdown()**, que irá avisar ao executor para não receber novas tarefas e iniciar a sua liberação. Caso esse método não fosse chamado, o programa principal (**main()**) nunca seria finalizado, pois as *threads* ativas de **ExecutorService** impedem que a JVM termine sua execução;
- **newFixedThreadPool()**: cria uma implementação de **ExecutorService** com um número fixo de threads trabalhadoras que é passado como parâmetro na chamada do método fábrica (ex.: **Executors.newFixedThreadPool(2)**). Desta forma podemos limitar a criação de threads. Se existirem mais tarefas para serem executadas do que threads livres, as tarefas aguardam em fila até que alguma thread esteja disponível. Ao mesmo tempo, threads podem ficar ociosas esperando novos trabalhos;
- **newCachedThreadPool()**: cria uma implementação de **ExecutorService** para atendimento das tarefas de maneira mais dinâmica. A criação de threads será feita de acordo com os novos trabalhos. Caso uma nova tarefa não encontre uma *thread* trabalhadora para atendê-la,

Programação paralela em Java

uma nova thread será criada. Além disso, threads que ficam muito tempo ociosas são destruídas. A desvantagem desta opção está na criação de diversas threads caso existam muitas tarefas a serem processadas, o que pode gerar um consumo indevido de recursos do sistema e até gerar um **OutOfMemoryError** por falta de memória;

- **newScheduleThreadPool()**: cria uma implementação de **ScheduleExecutorService**, que por sua vez é uma implementação de **ExecutorService**. Uma **ScheduleExecutorService** tem por objetivo criar tarefas que podem ser agendadas ou executadas com certa periodicidade. Na **Listagem 5** criamos uma tarefa que será agendada e só será executada depois de um segundo. Para determinar a tarefa e definir o período, utilizamos o método **schedule()**.

Listagem 4. Código da classe ExecutorJava.

```
01.import java.util.concurrent.ExecutorService;
02.import java.util.concurrent.Executors;
03
04.public class ExecutorJava {
05
06.    public static void main(String[] args) {
07.        ExecutorService executorService = Executors.newSingleThreadExecutor();
08.        executorService.execute(new Runnable() {
09.
10.            @Override
11.            public void run() {
12.                System.out.println("Criando um Executor em Java!");
13.            }
14.        });
15.        executorService.shutdown();
16.    }
17.
18.}
```

Listagem 5. Código da classe ScheduledJava.

```
01.import java.util.concurrent.Executors;
02.import java.util.concurrent.ScheduledExecutorService;
03.import java.util.concurrent.TimeUnit;
04.
05.public class ScheduledJava {
06.
07.    public static void main(String[] args) {
08.        ScheduledExecutorService scheduled = Executors.newSingleThreadScheduledExecutor();
09.        Runnable runnable = new Runnable() {
10.
11.            @Override
12.            public void run() {
13.                System.out.println("Criando um Scheduled em Java!");
14.            }
15.        };
16.        scheduled.schedule(runnable, 1, TimeUnit.SECONDS);
17.        scheduled.shutdown();
18.    }
19.
20.}
```

O novo pacote ainda introduziu uma nova interface chamada **Callable**. Quando trabalhamos com objetos **Runnable** não podemos retornar valores de sua execução. Para esses casos, teríamos que declarar variáveis fora do contexto da *thread*, o que pode causar sérios problemas de sincronização, limitando as possibilidades de divisão de trabalho.

Para solucionar esse problema implementamos objetos da interface **Callable**, que é similar a **Runnable**, mas que pode retornar um resultado ou disparar exceções checadas, através do método **call()**.

Na **Listagem 6** temos um exemplo com essa interface. Nele, primeiramente definimos um *pool de threads* e implementamos o método **call()** para realizar a soma dos dez primeiros números naturais e retornar o resultado dessa execução. Para isso, submetemos a tarefa ao pool utilizando o método **submit()** de **ExecutorService**.

A execução do método **submit()** irá retornar um objeto da classe **Future** que abstrai o retorno do método **call()**. Assim, para obtermos o valor da nossa soma, fazemos uma chamada ao método **get()** de **Future** e este retorna o valor da execução de **call()**. Como a chamada ao método **get()** é bloqueante, a execução da aplicação irá aguardar até que o processamento tenha sido finalizado, para só então exibir o resultado no console, como definido na linha 21.

Listagem 6. Código da classe CallableJava.

```
01.import java.util.concurrent.Callable;
02.import java.util.concurrent.ExecutorService;
03.import java.util.concurrent.Executors;
04.import java.util.concurrent.Future;
05.
06.public class CallableJava {
07.
08.    public static void main(String[] args) throws Exception {
09.        ExecutorService pool = Executors.newCachedThreadPool();
10.        Callable<Integer> callable = new Callable<Integer>() {
11.
12.            @Override
13.            public Integer call() throws Exception {
14.                int soma = 0;
15.                for (int i = 0; i <= 10; i++) {
16.                    soma += i;
17.                }
18.                return soma;
19.            }
20.        };
21.        Future<Integer> result = pool.submit(callable);
22.        System.out.println(result.get());
23.        pool.shutdown();
24.
25.    }
26.
27.}
```

Fork/Join Framework

Com a introdução do pacote **java.util.concurrent** no Java 5, melhoramos a forma como trabalhamos com código concorrente em Java. Entretanto, para podermos tirar proveito de múltiplos processadores e das novas tecnologias multicore, precisamos desenvolver instruções para serem executadas em paralelo.

Para isso, na versão 7, o Java avança com a inclusão de uma nova implementação de **ExecutorService**, o framework **Fork/Join**, que ajuda a simplificar ainda mais a escrita de código paralelo. Este novo framework permite ao desenvolvedor aproveitar todos os recursos oferecidos por múltiplos processadores.

O **Fork/Join** foi otimizado para tarefas que podem ser divididas recursivamente e executadas em pequena escala, e depois combinar

os resultados, utilizando a técnica de dividir e conquistar. Basicamente, dentro de uma tarefa, é verificado o tamanho do problema a ser resolvido. Se ele é pequeno o suficiente, então o problema é resolvido diretamente, senão o problema é dividido em partes. Nesse caso, cada parte irá gerar uma nova subtarefa, que após sua execução retornará um subresultado. Posteriormente, todos os subresultados são reunidos, devolvendo o resultado final do processamento. Dividir as tarefas em tarefas menores é um processo chamado de *fork*, e unir os resultados dessas pequenas tarefas é um processo chamado de *join* (Figura 5).

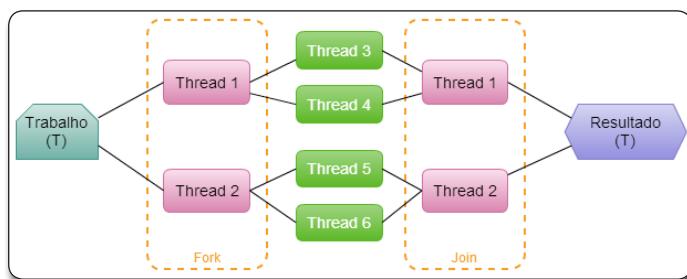


Figura 5. Modelo Fork-Join

A grande evolução e melhoria trazida por esse modelo se dá na redução da carga de trabalho. Se uma tarefa tem peso consideravelmente grande de execução, então o melhor é dividi-la e executá-la de forma separada. Com esta técnica é possível tirar proveito de todas as unidades de processamento, e assim uma tarefa complexa pode ter seu tempo de execução reduzido significativamente.

Doug Lea, líder do projeto, diz que o paralelismo com Fork/Join está entre as melhores e mais simples técnicas de design de software quando se precisa obter boa performance.

Dividir e Conquistar

O principal objetivo na utilização do Framework Fork/Join é a divisão de trabalho para ganho de desempenho. Para isso, este algoritmo se baseia na ideia da divisão e conquista, utilizando recursividade. Com o intuito de demonstrar o seu funcionamento, podemos aplicá-lo para solucionar o cálculo de fatoriais.

Este cálculo é um exemplo clássico de recursão. Na Matemática, o fatorial de um número é definido como a multiplicação de todos os seus antecessores naturais excetuando-se o zero e incluindo a si próprio. Sua representação formal é $n!$. Logo, observa-se que $n! = n * (n-1)!$, ou ainda $n! = n * (n-1) * (n-2)!$.

Com isso, percebemos que o mesmo problema pode ser dividido em várias partes, que podem ainda ser decompostas em novas outras partes.

Para exemplificar, imagine que precisamos calcular o fatorial do número 3. Para este caso a solução é simples e rápida, mas se precisássemos calcular o fatorial de 5000. Computacionalmente, quanto maior o número, maior a quantidade de processamento e consumo de recursos. Agora e se pudéssemos calcular cada parte do problema separadamente e depois unir os resultados?

São em cenários como esses que a estratégia de dividir e conquistar pode ser utilizada.

A sua vantagem é que não mais limitamos a resolução do problema a uma forma sequencial. Também podemos executá-la em paralelo. Deste modo, ao invés de apenas iterarmos utilizando uma única *thread*, conseguimos separar a execução da solução em várias *threads*, que podem utilizar todos os recursos computacionais disponíveis.

No entanto, para obtermos uma vantagem significativa devemos observar os limites do processamento paralelo e sequencial. No nosso pequeno exemplo, calcular o fatorial de 3 poderia ser muito mais rápido de maneira sequencial do que paralela, visto que temos que contabilizar também os tempos de alocação e inicialização de objetos.

ForkJoinPool

A principal diferença entre os frameworks Fork/Join e Executor é o algoritmo de roubo de trabalho (*work-stealing*) presente em Fork/Join. Com esse algoritmo cada *thread* mantém uma fila de tarefas, e caso uma *thread* esteja sem trabalho, ela roubará uma tarefa de outra *thread*. Desta maneira se evita o problema de que *threads* que já terminaram a execução de suas tarefas fiquem ociosas, enquanto existem outras com trabalho a executar.

Para essa solução, o pacote `java.util.concurrent` fornece uma nova implementação de `ExecutorService`: `ForkJoinPool`. O `ForkJoinPool` pode trabalhar com objetos `Runnable` ou `Callable`. No entanto, possui uma abstração própria para definição de tarefas que irão trabalhar seguindo o modelo de dividir e conquistar: `ForkJoinTask` e suas subclasses `RecursiveAction` e `RecursiveTask`.

Uma `ForkJoinTask` representa uma tarefa abstrata em um pool de Fork/Join. Seus dois métodos principais são `fork()` e `join()`. O método `fork()` é utilizado para criação de novas tarefas, enquanto `join()` aguarda a execução para posterior união dos resultados.

A subclasse `RecursiveTask` é utilizada quando é necessário retornar resultados e a subclasse `RecursiveAction`, por outro lado, é utilizada quando nenhum resultado deve ser retornado.

Na [Listagem 7](#) implementamos um exemplo utilizando `RecursiveAction` para calcular o quadrado de cada elemento de um determinado array. Deste modo, primeiramente declaramos um `ForkJoinPool`. Quando instanciado com o construtor sem parâmetros, como no exemplo, o pool irá atribuir, por padrão, o número de processadores identificado no sistema operacional. Esse número pode ser alterado por meio da passagem de um parâmetro de tipo inteiro ao método construtor da classe, por exemplo: `ForkJoinPool(3)`.

Em seguida, criamos um array de tamanho 10.000 com números aleatórios de 0 a 10 e instanciamos a classe `QuadradoDoNumero`, que estende `RecursiveAction`.

Na implementação da classe `QuadradoDoNumero` criamos um array chamado `numeros[]` que receberá no construtor todos os números que desejamos elevar ao quadrado, e ainda duas

variáveis, chamadas **inicio** e **fim**, que serão auxiliares para que possamos parallelizar a execução do nosso cálculo.

Ao iniciar a execução da classe principal, no método **compute()** será verificado se devemos dividir o array para processamento paralelo. Caso positivo, criamos uma nova instância de **QuadradoDoNumero** contendo um array de números. Esse array será sempre do mesmo tamanho, mas será percorrido considerando apenas os índices que estiverem dentro do intervalo definido pelas variáveis **inicio** e **fim**. Dessa maneira dividimos o cálculo do quadrado dos números em várias subtarefas.

Ainda na implementação de **QuadradoDoNumero**, precisamos sobreescriver o método **compute()**. Esse método será executado quando a tarefa for invocada pelo pool. Nele, verificamos a cada chamada se o problema é pequeno o suficiente para ser executado de forma sequencial, utilizando para isso as variáveis **inicio** e **fim**. Caso o tamanho do array (**fim - inicio**) seja menor que 100, não serão criadas novas threads trabalhadoras e será executado o cálculo. Caso contrário, dividimos o problema em dois novos objetos, **esquerda** e **direita**, da classe **QuadradoDoNumero**, e assim sucessivamente até chegarmos à condição de parada (**fim - inicio < 100**).

Cada um desses objetos será responsável pela execução do algoritmo considerando uma parte dos dados. Assim, enquanto uma tarefa está calculando o quadrado dos elementos do array com índices de 0 a 99, outra estará calculando o quadrado dos elementos do array com índices de 100 a 199.

Depois de criada a tarefa, devemos invocar o método **fork()** da classe **RecursiveAction**. Esse método faz com que uma nova atividade

seja submetida ao pool de threads, sendo incluída na fila de tarefas do pool para ser executada. Em seguida, invocamos o método **join()**, que garante que a tarefa apenas retorne quando todas as suas subtarefas estiverem completas, ou seja, assim que as subdivisões, a partir da primeira chamada, forem concluídas, bloqueando a finalização até que a execução do método **compute()** seja concluída.

Ao apurar os tempos de execução desse exemplo, talvez a implementação sequencial tenha um desempenho melhor que a execução paralela. Isso acontece porque quando parallelizamos a execução do nosso algoritmo, fazemos uma verificação para saber se é necessário criar uma nova tarefa ou se o código já pode ser computado, o que gera um tempo adicional de execução, que acumula o tempo de leitura, de carregamento de novas classes em memória e de alocação das variáveis que foram utilizadas adicionais para a execução paralela. Como criar objetos gera custos extras para a memória e também para o processamento, deve-se ficar atento para dividir uma tarefa apenas quando essa opção for vantajosa.

Para expor como o desempenho pode ser aprimorado, na **Listagem 8** implementamos um exemplo de algoritmo que realiza a soma dos valores de um determinado número de elementos presentes em um array. Para isso, criamos uma classe chamada **Soma** estendendo **RecursiveTask**. Utilizamos essa abstração (**RecursiveTask**) porque queremos que nossa execução nos retorne um resultado.

Vale observar nesse exemplo que limitamos no construtor de **ForkJoinPool()** a quantidade de processos paralelos que podem ser executados. Assim, caso o sistema possua mais de dois processadores, apenas dois poderão ser utilizados.

Listagem 7. Código da classe QuadradoDoNumeroRecursiveAction.

```
01.import java.util.Random;
02.import java.util.concurrent.ForkJoinPool;
03.import java.util.concurrent.RecursiveAction;
04.
05.
06.public class QuadradoDoNumeroRecursiveAction {
07.
08.    public static void main(String[] args) {
09.        ForkJoinPool pool = new ForkJoinPool();
10.        Random random = new Random();
11.        int limite = 10000;
12.        int[] numeros = new int[limite];
13.        for (int i = 0; i < limite; i++) {
14.            numeros[i] = random.nextInt(10);
15.        }
16.        QuadradoDoNumero quadrado = new QuadradoDoNumero
17.            (numeros, 0, numeros.length);
18.        pool.invoke(quadrado);
19.        for (int numero : numeros) {
20.            System.out.println(numero);
21.        }
22.
23.    }
24.
25.class QuadradoDoNumero extends RecursiveAction {
26.
27.    int[] numeros;
28.    int inicio;
29.    int fim;
30.
31.    public QuadradoDoNumero(int[] numeros, int inicio, int fim) {
32.        this.numeros = numeros;
33.        this.inicio = inicio;
34.        this.fim = fim;
35.    }
36.
37.    @Override
38.    protected void compute() {
39.        if ((fim - inicio) < 100) {
40.            for (int i = inicio; i < fim; i++) {
41.                numeros[i] = numeros[i] * numeros[i];
42.            }
43.        } else {
44.            int meio = (inicio + fim) / 2;
45.            QuadradoDoNumero esquerda = new QuadradoDoNumero
46.                (numeros, inicio, meio);
47.            QuadradoDoNumero direita = new QuadradoDoNumero
48.                (numeros, meio, fim);
49.            esquerda.fork();
50.            direita.fork();
51.            esquerda.join();
52.            direita.join();
53.
54.        }
55.
56.    }
57.
```

Listagem 8. Código da classe SomaRecursiveTask.

```
01.import java.util.ArrayList;
02.import java.util.List;
03.import java.util.concurrent.ForkJoinPool;
04.import java.util.concurrent.RecursiveTask;
05.
06.public class SomaRecursiveTask {
07.
08.    public static void main(String[] args) {
09.        ForkJoinPool pool = new ForkJoinPool(2);
10.        int limite = 10000;
11.        int[] numeros = new int[limite];
12.        for (int i = 0; i < limite; i++) {
13.            numeros[i] = 1;
14.        }
15.        int soma = pool.invoke(new Soma(numeros, 0, numeros.length));
16.        System.out.println(soma);
17.    }
18.
19.}
20.
21.class Soma extends RecursiveTask<Integer> {
22.
23.    int[] numeros;
24.    int inicio;
25.    int fim;
26.
27.    public Soma(int[] numeros, int inicio, int fim) {
28.        this.numeros = numeros;
29.        this.inicio = inicio;
30.        this.fim = fim;
31.    }
32.
33.    @Override
34.    protected Integer compute() {
35.        int soma = 0;
36.        List<RecursiveTask<Integer>> forks = new ArrayList<>();
37.        if ((fim - inicio) < 20) {
38.            int somaTemp = 0;
39.            for (int i = inicio; i < fim; i++) {
40.                somaTemp += numeros[i];
41.            }
42.            return somaTemp;
43.        } else {
44.            int meio = (inicio + fim) / 2;
45.            Soma somaEsquerda = new Soma(numeros, inicio, meio);
46.            Soma somaDireita = new Soma(numeros, meio, fim);
47.            forks.add(somaEsquerda);
48.            forks.add(somaDireita);
49.            somaEsquerda.fork();
50.            somaDireita.fork();
51.        }
52.        for (RecursiveTask<Integer> fork : forks) {
53.            soma += fork.join();
54.        }
55.        return soma;
56.    }
57.
58.}
```

Durante o desenvolvimento de seus sistemas, lembre-se que o acesso aos processadores é compartilhado com outros aplicativos do sistema. Por isso, nem sempre a quantidade informada será realmente a utilizada.

No método `compute()`, como no exemplo da [Listagem 7](#), é escrito o algoritmo a ser submetido ao pool. Neste caso, cada nova tarefa irá conter uma sub lista de outras tarefas (`RecursiveTask`), para que ao final de cada execução possamos somar os resultados obtidos individualmente. A instrução de divisão verifica se a quantidade de elementos no vetor é menor que 20 elementos. Caso positivo, executa o algoritmo de soma, caso negativo, inicia uma nova tarefa. Por fim, quando essa tarefa e suas subtarefas forem finalizadas, o resultado é somado e retornado.

Nestes exemplos, pudemos perceber que a implementação de código Java paralelo evoluiu bastante desde a simples criação de threads. Com as novas ferramentas disponíveis no Java 7, uma gama de outras possibilidades passa a ser possível de maneiras muito mais simples e com total aproveitamento dos recursos computacionais.

Java 8

Neste momento, com a disponibilização da versão 8, a linguagem Java passa por uma de suas maiores evoluções. São vários os novos recursos, muitos deles voltados a técnicas de programação funcional, no entanto, como não é objetivo desse artigo entrar em todos os conceitos e novidades relacionados, vamos destacar apenas um. Nesta versão foi incluída a possibilidade do Java parallelizar automaticamente a execução de algumas rotinas.

Para apresentar essa opção, na [Listagem 9](#) implementamos o mesmo exemplo analisado neste artigo utilizando uma das novas opções disponibilizadas pela linguagem. A partir da chamada ao método `parallel()`, passa a ser possível parallelizarmos a execução de determinadas tarefas, como a soma de valores presentes em um array. Para mais informações sobre o Java 8, leia os artigos relacionados já publicados na Java Magazine e Easy Java Magazine.

Listagem 9. Código da classe SomaNoJava8.

```
01.import java.util.Arrays;
02.
03.public class SomaNoJava8 {
04.
05.    public static void main(String[] args) {
06.        int limite = 10000;
07.        int[] numeros = new int[limite];
08.        for (int i = 0; i < limite; i++) {
09.            numeros[i] = 1;
10.        }
11.        int soma = Arrays.stream(numeros).parallel().sum();
12.        System.out.println(soma);
13.    }
14.}
```

As facilidades trazidas pelas novas versões do Java visam motivar o uso de código paralelo e concorrente, além de deixar para trás velhos conceitos de que somente programadores bastante experientes conseguem desenvolver sistemas multitarefas.

Todos os conceitos analisados valem para uma série de aplicações. Por exemplo, quando precisamos enviar um e-mail, na

Programação paralela em Java

importação de arquivos de texto, na sincronização da aplicação com outros sistemas, na leitura de dados de balanças eletrônicas, na comunicação com web services da Receita Federal para Notas Fiscais Eletrônicas, entre outras possibilidades. Tudo isso sem o grande problema da tela travar para o usuário. Portanto, aproveite os recursos analisados para enriquecer a sua caixa de ferramentas e aprimorar a qualidade e desempenho de suas aplicações.

Autor



Renato Alexandre Justo

renatoalexandrejusto@gmail.com

É graduado em Tecnologia em Desenvolvimento de Jogos Digitais, trabalha há seis anos como desenvolvedor Java, em soluções integradas. Atualmente cursando Especialização em Tecnologia Java pela Universidade Tecnológica Federal do Paraná.



Autor



Fabrício Martins Lopes

fabricio@utfpr.edu.br

É mestre em Informática e doutor em Bioinformática, atuando principalmente nos temas de pesquisa em reconhecimento de padrões, bioinformática, processamento de imagens e visão computacional. Atualmente, é docente da Universidade Tecnológica Federal do Paraná atuando nos cursos de Engenharia da Computação e Tecnologia em Análise e Desenvolvimento de Sistemas.



Links:

CORREA, Jefferson dos Santos. Estudo sobre linguagens de programação paralela e os conceitos chave de paralelismo em nível de programação. Dissertação (Trabalho de Conclusão de Curso) – Fundação Educacional do Município de Assis – FEMA, Assis, São Paulo, 2013.

GONZÁLEZ, Javier Fernández. Java 7 Concurrency Cookbook. Packt Publishing, 2012.

JAVABEAT. Simple Introduction To Fork Join Framework in Java 7.

<http://www.javabeat.net/simple-introduction-to-fork-join-framework-in-java-7/>

LEA, Doug. A Java Fork/Join Framework.

<http://dl.acm.org/citation.cfm?id=337465>

PONGE, Julien. Fork and Join: Java Can Excel at Painless Parallel Programming Too!

<http://www.oracle.com/technetwork/articles/java/fork-join-422606.html>

SILVA, Jackson Amaral. Um Gerador Automático de Algoritmos Evolutivos Paralelos em Java.

<http://www.mestrado.engcomp.uema.br/wp-content/uploads/2014/03/dissertacao.pdf>

SUTTER, Herb. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software.

http://www.cs.utexas.edu/~lin/cs380p/Free_Lunch.pdf

Conhecimento faz diferença!

Automação de Testes
+ de 290 vídeos para assinantes

Faça já sua assinatura digital! | www.devmedia.com.br/es

Faça um *upgrade* em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software. Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviços); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**



DEV MEDIA

Somos tão apaixonados por tecnologia que o nome da empresa diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia. Somos uma equipe composta de gente que entende e gosta do que faz,
assim como você.



Estrutura

100% NACIONAL.
Servidores de primeira linha, links de alta capacidade.



Supporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.



Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.



1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
Conheça!



Porta 80

WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce |
Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486