02

Refatorando

Download

Caso queira começar o treinamento a partir desse vídeo, pode baixar o projeto aqui (https://s3.amazonaws.com/caelum-online-public/auron/auron-stage7.zip). **Só baixe este arquivo se não tiver feito os exercícios dos capítulos anteriores.**

Refatoração

Bem-vindo ao capítulo sobre refatoração.

A classe Sorteador: será que podemos deixá-la melhor do que já é?

Abrindo nossa classe `Sorteador`, não precisamos meditar muito para vermos a repetição do código que instancia e adiciona um `Par` na lista de pares:

```
Par par = new Par(participantes.get(indiceAtual), participantes.get(0), sorteio);
sorteio.adicionaPar(par);
```

Sabemos que código duplicado é inimigo da manutenção: imagine que tenhamos a necessidade de alterar essa lógica. Será que lembraremos de alterá-la em todos os locais em que se repete?

Atacando código duplicado

Uma maneira de resolvermos isso é **refatorar** nosso código. Refatoração consiste em melhorar o design de nosso código favorecendo sua legibilidade e sua manutenção, **mas sem alterar seu comportamento original.**

O Eclipse possui uma série de ferramentas que auxiliam neste processo, inclusive já podemos selecionar o código duplicado e com o botão direito usaremos a opção `Extract do Method`. Esta ferramenta extrairá o código que selecionamos para um novo método, utilizando nos locais onde nosso código duplicado aparecia:

```
private void extracted(int indiceAtual) {
    Par par = new Par(participantes.get(indiceAtual), participantes.get(0), sorteio);
    sorteio.adicionaPar(par);
}
```

Métodos expressivos

O nome do método `extracted` não nos diz muita coisa sobre o que o método faz, é por isso que escolheremos um nome mais elegante para ele, em nosso caso, `criaEAdicionaOParNoSorteio`, extremamente mais expressivo:

```
private void criaEAdicionaOParNoSorteio(int indiceAtual) {
    Par par = new Par(participantes.get(indiceAtual), participantes.get(0), sorteio);
    sorteio.adicionaPar(par);
}
```

Porém, precisamos ajustar os nos nomes dos parâmetros do método, também tornando-os mais expressivos:

```
private void criaEAdicionaOParNoSorteio(Sorteio sorteio, int indiceAtual, int indiceFinal) {
    Par par = new Par(participantes.get(indiceAtual), participantes.get(indiceFinal), sorteio);
    sorteio.adicionaPar(par);
}
```

Repare que o método é privado, pois não temos interesse que utilizadores da classe chamem este método. Só faz sentido ele ser chamado pela própria classe.

Agora, precisamos chamar nosso método com os parâmetros corretos nos dois locais em que são chamados pela aplicação:

```
public void sortear() throws SorteioException {
        int indiceAtual = 0;
        int totalDeParticipantes = participantes.size();

        if(totalDeParticipantes < 2)
            throw new SorteioException("Por favor, insira uma lista de participantes com no mín

        while(indiceAtual < totalDeParticipantes) {
            if(indiceAtual == totalDeParticipantes - 1) {
                criaEAdicionaOParNoSorteio(sorteio, indiceAtual, 0);
                break;
            }
            criaEAdicionaOParNoSorteio(sorteio, indiceAtual, indiceAtual + 1);
            indiceAtual++;
        }
}
```

Vamos rodar nossa suite de testes para verificar se quebramos nosso código. Tudo continua funcionando.

Tornando código inline mais expressivo

Podemos melhorar ainda mais nosso código. Vejamos esta condição if:

```
if(indiceAtual == totalDeParticipantes - 1)
```

Por mais ínfima que seja, ela contém uma lógica. Você consegue identificar rapidamente o que ela faz? Pois é, podemos tornar nossa intenção mais evidente extraindo essa lógica para o método `participanteAtualEhOUltimo` , pois queremos saber se o participanteAtual de nossa lista é o último da lista:

```
private boolean participanteAtualEhOUltimo(int indiceAtual) {
    return indiceAtual == totalDeParticipantes -1;
}
```

E sua chamada:

```
public void sortear() throws SorteioException {
        int indiceAtual = 0;
        int totalDeParticipantes = participantes.size();

        if(totalDeParticipantes < 2)
            throw new SorteioException("Por favor, insira uma lista de participantes com no mín

        while(indiceAtual < totalDeParticipantes) {
            if(participanteAtualEhOUltimo(indiceAtual)) {
                criaEAdicionaOParNoSorteio(sorteio, indiceAtual, 0);
                break;
            }
            criaEAdicionaOParNoSorteio(sorteio, indiceAtual, indiceAtual + 1);
            indiceAtual++;
        }
}
```

Rodando nossos testes mais uma vez!

Nosso `totalDeParticipantes` é local ao método, vamos torná-la uma instância da classe.

```
public class Sorteador {
    // código anterior comentado
    private int totalDeParticipantes;
}
```

E não podemos esquecer de inicializar a nossa variável `totalDeParticipantes` no construtor:

```
public Sorteador(Sorteio sorteio, List<Participante> participantes) throws SorteioException {
    // código anterior comentado
    totalDeParticipantes = participantes.size();
}
```

Agora, vamos extrair o código que testa se a quantidade mínima de participantes é atendida para seu próprio método. Repare que ele consulta nossa variável de instância:

```
private void verificaTamanhoDaListaDeParticipantes() throws SorteioException {
    if(totalDeParticipantes < 2)
        throw new SorteioException("Por favor, insira uma lista de participantes com no mínimo
}
```

E claro, sua chamada em nosso código fica assim:

```
public void sortear() throws SorteioException {
        int indiceAtual = 0;

        verificaTamanhoDaListaDeParticipantes();

        while(indiceAtual < totalDeParticipantes) {
```

```
            if(participanteAtualEhOUltimo(indiceAtual)) {
                criaEAdicionaOParNoSorteio(sorteio, indiceAtual, 0);
                break;
            }
            criaEAdicionaOParNoSorteio(sorteio, indiceAtual, indiceAtual + 1);
            indiceAtual++;
        }
    }
```

Quebramos algo? Rodando nossa suite de testes

Vamos rodar mais uma vez nossos testes. Excelente, como alteramos apenas a organização do nosso código sem mudar seu comportamento, nossos testes passaram.

Podemos melhorar ainda mais nosso código, embaralhando a lista de participantes através de `Collections.shuffle` :

```
        int indiceAtual = 0;

        verificaTamanhoDaListaDeParticipantes();
        Collections.shuffle(participantes);

        while(indiceAtual < totalDeParticipantes) {
            if(participanteAtualEhOUltimo(indiceAtual)) {
                criaEAdicionaOParNoSorteio(sorteio, indiceAtual, 0);
                break;
            }
            criaEAdicionaOParNoSorteio(sorteio, indiceAtual, indiceAtual + 1);
            indiceAtual++;
        }
    }
```

Mas será que podemos deixar essa linha mais expressiva, usando um vocabulário mais perto do domínio do problema? Sim, podemos usar o termo `embaralhaParticipantes` . Novamente, criamos um método privado para tornarmos mais expressivo nosso código:

```
    private void embaralhaParticipantes() {
        Collections.shuffle(participantes);
    }
```

```
        int indiceAtual = 0;

        verificaTamanhoDaListaDeParticipantes();
        embaralhaParticipantes();

        while(indiceAtual < totalDeParticipantes) {
            if(participanteAtualEhOUltimo(indiceAtual)) {
                criaEAdicionaOParNoSorteio(sorteio, indiceAtual, 0);
                break;
            }
            criaEAdicionaOParNoSorteio(sorteio, indiceAtual, indiceAtual + 1);
            indiceAtual++;
```

```
        }
    }
```

Será que nosso código passa em nossos testes? Só rodando-os para sabermos! Excelente, nosso código passa no teste.