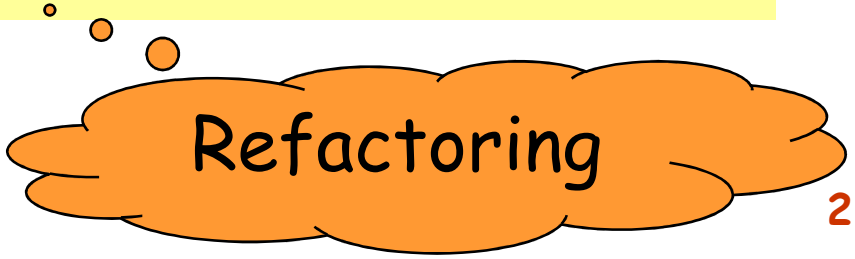
Introdução à Programação

Mais sobre Refactoring, Mudanças de ordem de comandos e Padrões de Codificação

Primeira e Segunda lei de Lehman

"A large program that is used undergoes continuing change or becomes progressively less useful"

"As a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it"



Tópicos da aula

- Hoje veremos a importância de se utilizar técnicas de Refactoring
  - Criação de métodos para Reuso
  - Criação de métodos para Legibilidade
  - Mais um tipo de Refactoring
    - Extract parameter
- Analisaremos o efeito da ordem de declarações e comandos
  - Leis para mudança de ordem de comandos
  - Mudança de estados efetuados por comandos
- Examinaremos também padrões de codificação
  - Nome de classes, métodos e atributos
  - Papel da indentação em um programa

Revisando Refactoring

- ***Refactoring*** consiste em uma série de técnicas que reestruturam o código do software, aumentando o potencial de reuso, extensibilidade e legibilidade
  - O comportamento do software continua o mesmo, só muda a estrutura
  - Facilita a manutenção do sistema
    - Mudança de requisitos
    - Correção de bugs
  - Deve ser usado acelerar o desenvolvimento do sistema

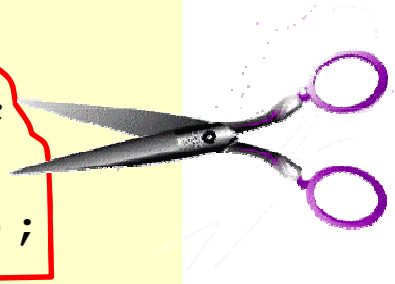
Revisando Refactoring

- Aplicar técnicas de ***Refactoring*** nem sempre aumenta todos os fatores de qualidade
  - Pode ser, por exemplo, somente a reusabilidade ou somente a extensibilidade ou ainda a legibilidade
  - Qualquer refactoring deve ser bem pensado para que realmente melhore algum fator de qualidade

Criação de Método para Aumento da Legibilidade

Antes do Refactoring

```
public void clickEvent(Component c) {
    if (c == botaoCalcular) {
        int entrada;
        entrada = campoEntrada.getInt();
        int resultado = 2 * entrada;
        campoResultado.setInt(resultado);
        campoEntrada.setText("");
    } else if (c == botaoSaida) {
        sairDoSistema();
    }
}
```



Criação de Método para Maior Legibilidade

Depois do Refactoring

```
public void clickEvent(Component c)
{
     if (c == botaoCalcular) {
        duplicarValor();
     } else if (c == botaoSaida) {
        sairDoSistema();
     }
}
```

```
void duplicarValor() {
   int entrada;
   entrada = campoEntrada.getInt();
   int resultado = 2 * entrada;
   campoResultado.setInt(resultado);
   campoEntrada.setText("");
}
```

Dificilmente, este método será reutilizado, porém o código ficou mais **legível**

Criação de Método para Maior Reusabilidade

- Evita duplicação de código
- Se o método é utilizado com uma certa frequência, compensa o tempo gasto com o refactoring do código
- Geralmente, acelera o desenvolvimento do software

Problemas da duplicação de código

- Código mais difícil de entender e de manter
  - Se um bug for encontrado, terá de ser corrigido em todos os outros códigos repetidos
  - Um programador desavisado pode esquecer de alterar os outros códigos replicados e o bug irá persistir
  - Se uma alteração for requerida, esta também deverá ser propagada por todos os trechos repetidos

Duplicação de código == duplicação de esforço!!!

Algumas maneiras de remover a duplicação...

- Identifique os pontos que são variáveis
- Se for entre vários métodos similares, escolha um e generalize-o adicionando parâmetros
  - Altere as chamadas para que recebam os novos parâmetros
  - Use um nome mais adequado para o nome do método
  - Eventualmente, é necessário também alterar seu tipo de retorno
- Se não for, identifique o código repetitivo e use um método para substitui-lo

Removendo a duplicação passo a passo

- Identificando os pontos variáveis

```
void criarBotaoEntrada(){
  botaoEntrada = new Button();
  botaoEntrada.setText("Entrada");
  this.include(botaoEntrada, 250,65);
}
```

Os dois trechos de código são idênticos. Apenas nome de variável e alguns parâmetros são diferentes

```
void criarBotaoResultado(){
  botaoResultado = new Button();
  botaoResultado.setText("Resultado");
  this.include(botaoResultado , 360,65);
}
```

Generalizando o método

Parametrização

```
void criarBotaoEntrada(String nome, int x, int y){
  botaoEntrada = new Button();
  botaoEntrada.setText(nome);
  this.include(botaoEntrada, x,y);
}
```

Alteração do tipo de retorno

```
Button criarBotaoEntrada(String nome, int x, int y){
  Button b = new Button();
  b.setText(nome);
  this.include(b,x,y);
  return b;
}
```

Adaptando as chamadas

```
MinhaClasseComDoisBotoes() {
  criarBotaoEntrada();
  criarBotaoResultado();
}
```

```
MinhaClasseComDoisBotoes()
  botaoEntrada = criarBotaoEntrada("Entrada", 250, 65);
  botaoResultado = criarBotaoEntrada("Resultado", 360,
   65);
}
```

Renomeando o método e removendo os métodos desnecessários

```
MinhaClasseComDoisBotoes()
  botaoEntrada = criarBotaoEntrada("Entrada", 250, 65);
  botaoResultado = criarBotaoEntrada("Resultado", 360, 65);
}
Button criarBotaoEntrada(String nome, int x, int y){...}
void criarBotaoResultado(){...}
```

Nome não é adequado!

```
MinhaClasseComDoisBotoes()
  botaoEntrada = criarBotao("Entrada", 250, 65);
  botaoResultado = criarBotao("Resultado", 360, 65);
}
Button criarBotao(String nome, int x, int y){...}
```

Adaptando as chamadas e renomeando o método antigo

```
Button criarBotao(String nome, int x, int y){
  Button b = new Button();
  b.setText(nome);
  this.include(b,x,y);
  return b;
}
```

```
MinhaClasseComDoisBotoes()
  botaoEntrada = criarBotao("Entrada", 250, 65);
  botaoResultado = criarBotao("Resultado", 360, 65);
}
```

Criação de Método para Maior Reusabilidade

Antes do Refactoring

```
MinhaClasseComDoisBotoes() {
   criarBotaoEntrada();
   criarBotaoResultado();
}
void criarBotaoEntrada(){
   botaoEntrada = new Button("Entrada");
   this.include(botaoEntrada, 250,65);
}
 void criarBotaoResultado(){
    botaoResultado = new Button ("Resultado");
    this.include(botaoResultado , 360,65);
 }
```

Generalizando com parâmetros

Criação de Método para Maior Reusabilidade

Depois do Refactoring

```
MinhaClasseComDoisBotoes() {
  botaoEntrada = criarBotao("Entrada", 250, 65);
  botaoResultado = criarBotao("Resultado", 360, 65);
}
```

```
Button criarBotao(String nome, int x, int y){
  Button b = new Button(nome);
  this.include(b,x,y);
  return b;
}
```

Aumentamos a **reusabilidade**

Refactoring: Extract Parameter

- Este refactoring deve ser usado quando se quer generalizar um método com parâmetros
  - Métodos que fazem as mesmas ações, porém que atuam com valores diferentes (parâmetros)
  - O Eclipse faz isto automaticamente com a opção de refactor *Introduce Parameter*

Refactoring Rename

- Este refactoring deve ser usado sempre que o nome da entidade (seja classe, variável ou método) não condiz com o seu propósito

```
void configurarBotaoSair(String nome, int x, int y){
  Button b = new Button(nome);     this.include(b,x,y);
  return b;
}
```

```
void configurarBotao(String nome, int x, int y){
  Button b = new Button(nome);     this.include(b,x,y);
  return b;
}
```

Ordem de Declarações e Comandos

- Dentro de uma classe, a ordem de declaração de métodos, atributos e construtores não importa
- Dentro de métodos ou construtores, a ordem de declarações podem gerar erros de compilação
- Dentro de métodos ou construtores a ordem de comandos podem ou não influenciar a lógica do programa

Mudança de Ordem de Declarações em uma Classe

```
Button botaoEntrada;
MinhaJanelaComBotao(){
  botaoEntrada = criarBotaoEntrada("Entrada", 250, 65);
}
Button criarBotaoEntrada(String nome, int x, int y){...}
```

Pode dificultar legibilidade, porém não afeta programa

```
Button criarBotaoEntrada(String nome, int x, int y){...}
MinhaJanelaComBotao(){
  botaoEntrada = criarBotaoEntrada("Entrada", 250, 65);
}
Button botaoEntrada;
```

Mudança de Ordem de Declarações em um Método

```
MinhaJanelaComBotao(){
  Button botaoEntrada;
  botaoEntrada = criarBotaoEntrada("Entrada", 250, 65);
}
```

Neste caso, gera erro de compilação

```
MinhaJanelaComBotao(){
  botaoEntrada = criarBotaoEntrada("Entrada", 250, 65);
  Button botaoEntrada;
}
```

Mudança de Ordem de Comandos em um Método

```
MinhaJanelaComBotao(){
  Button botao = new Button();
  botao.setText("Fechar");
  this.include(botao, 250, 65);
}
```

```
MinhaJanelaComBotao(){
  Button botao = new Button();
  this.include(botao, 250, 65);
  botao.setText("Fechar");
}
```

Indiferente, pois apenas estado do objeto referenciado por `botao` é alterado após inclusão

Mudança de Ordem de Comandos em um Método

```
MinhaJanelaComBotao(){
  Button botao = new Button("Fechar");
  this.include(botao, 250, 65);
  botao.setText("Abrir");
  botao = new Button();
}
```

```
MinhaJanelaComBotao(){
  Button botao = new Button("Fechar");
  this.include(botao, 250, 65);
  botao = new Button();
  botao.setText("Abrir");
}
```

Diferente, pois a referência passada para o include é diferente da que está armazenada atualmente em `botao`

Mudança de Ordem de Comandos em um Método

```
MinhaJanelaComBotao(){
  Button botao = null;
  botao = new Button();
  this.include(botao, 250, 65);
  botao.setText("Abrir");
}
```

```
MinhaJanelaComBotao(){
  Button botao = null;
  this.include(botao, 250, 65);
  botao.setText("Abrir");
  botao = new Button();
}
```

Neste caso um erro em tempo de execução será gerado

Mudanças de Estado Efetuados por Comandos

- O estado de um objeto pode ser modificado se:
  - Executa-se um método do objeto que altere seu estado
  - Passa-se uma referência do objeto para um método, e este método modifica seu estado
    - Lembre-se, neste caso, de que só o estado é modificado, e não a referência

Mudanças de Estado Efetuados por Comandos

```
class Referencia {
    void setTextoPadrao(Button a)   {
      a.setText("Abrir");

    }
}
```

```
Referencia r = new Referencia();
Button botao = new Button();
botao.setText("Fechar");

r.setTextoPadrao(botao);
```

Estado alterado por método da Classe Button

Estado alterado por método da Classe Referencia

Padronização de Codificação

- Quanto mais fácil for o entendimento (legibilidade) do código do sistema, mais produtiva será a equipe de desenvolvimento
- Freqüentemente as pessoas que escrevem o código não são as mesmas que o mantém
- Um padrão de codificação visa minimizar esses problemas
  - Estabelece regras, definindo como o código deve ser escrito para favorecer a impessoalidade do programa
  - Facilita a integração de novos desenvolvedores ao ambiente de desenvolvimento

Nomenclatura de Classes

- Como na maioria das vezes, uma classe é uma abstração de uma classe de objetos do mundo real, deve-se lhe dar um nome compreensível e compatível com o que se quer modelar
- Nome da classe não deve ser abreviado
  - Economia de símbolos não justifica perda de expressividade
  - Excetuando-se os casos onde a abreviatura é mais conhecida do que o nome completo
- Recomenda-se colocar o nome de uma classe, começando com letra maiúscula
- Nome da classe não deve ser nem mais abrangente nem mais específico do que o conceito que ela representa
  - **Exemplos:** `Aluno`, `AlunoGraduacao`, `CPF`

Nomenclatura de Atributos e Variáveis

- A princípio, nomes não devem ser abreviados
  - Principalmente para atributos
  - Quando o fizer, usar bom senso para que não se perca expressividade
  - Usar bom senso para que nome da variável ou atributo não fique grande demais
    - Focar no que a variável ou atributo representam

- Recomenda-se colocar o nome de um atributo ou variável começando com letra minúscula
  - Se houver mais de uma palavra, as palavras seguintes devem começar com maiúscula
- Para nome de variáveis locais que representam contadores, pode-se colocar apenas uma letra do alfabeto

Exemplos de Atributos e Variáveis

- Para representar a quantidade de autores de um livro

```
quantidadeAutores //Bom

qtdAutores //Aceitável

qtdAut // Inaceitável

atributoQueArmazenaQuantidadeDeAutores //Ruim
```

- Declarando contador dentro de um laço

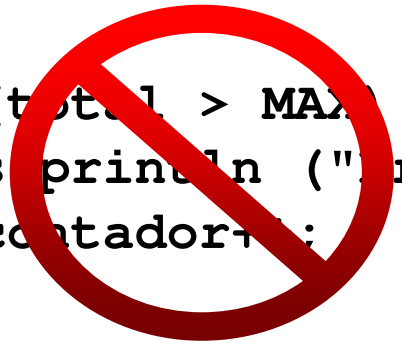
```
for (int i = 0; i < 5; i++) //Aceitável
```

Nomenclatura de Métodos

- Nomes não devem ser abreviados
- Recomenda-se colocar o nome começando letra minúscula
  - Se houver mais de uma palavra, as palavras seguintes devem começar com maiúscula
- Não se deve colocar nomes misturando palavras de línguas diferentes
  - Excetuando-se o caso de métodos de acesso (*get* e *set*)
- Muitas vezes, coloca-se na primeira palavra um verbo no infinitivo representando a utilidade do método
  - Ex: `adicionarAluno(Aluno aluno)`, `removerAluno(Aluno aluno)`

Importância da Indentação

- Como sabemos, o compilador Java ignora espaços e tabulações
- Porém, para aumentar a legibilidade do código é muito importante se preocupar com sua indentação
  - Torna mais claro o que será feito dentro de comandos condicionais e laços
  - Torna mais claro o limite de métodos, construtores e classes



```
if (total > MAX)
   s.println ("Erro!!");
contador++;
```

Que trecho de código é mais claro?

```
public class MinhaJanelaComBotao extends Window{
private Button botao;
private void criarBotao(String nome,int x, int y){
botao = new Button(nome);
this.include(botao,x,y);}
}
```

```
public class MinhaJanelaComBotao extends Window{
   private Button botao;
   private void criarBotao(String nome,int x, int y){
        botao = new Button(nome);
        this.include(botao,x,y);
   }
}
```

Resumindo ...

- Importância de Refactoring
  - Criação de métodos para Legibilidade
  - Problemas de duplicação
  - Criação de métodos para Reuso
  - Extract parameter
- Importância de ordem de declarações e comandos
  - Leis para mudança de ordem de comandos
  - Mudança de estados efetuados por comandos
- Importância de Padrões de Codificação