Programação Orientada a Aspectos

Baseado no material do prof. Sérgio Soares

Paradigmas de Programação – BCC/UFRPE
Lucas Albertins – lucas.albertins@deinfo.ufrpe.br

Agenda

- Motivação
- Paradigma Orientado a Aspectos
- Separação de Interesses
- Implementação com AOP
- AspectJ
- Exemplos
- Exercícios

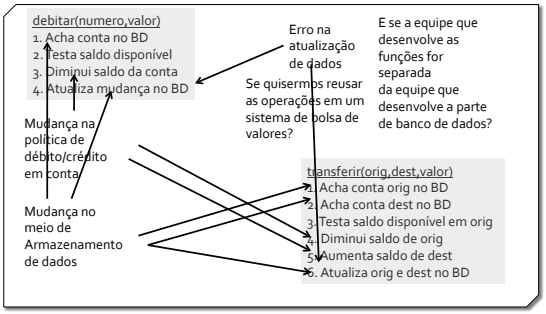
Motivação

- Queremos desenvolver software
  - de qualidade
  - capaz de se adaptar a mudanças
  - que lide com a complexidade
- A complexidade crescente dos sistemas de software gera novos desafios para as metodologias da Engenharia de Software

Projeto de qualidade

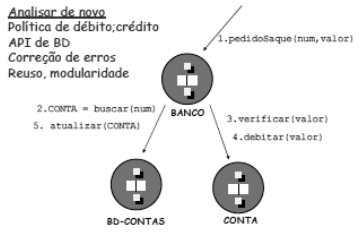
- Extensibilidade
  - software muda toda hora
- Facilidade de correção de erros
  - bug-free? nunca!
- Reusabilidade
  - produtividade total
- Modularidade
  - interfaces e separação

Paradigma Estruturado



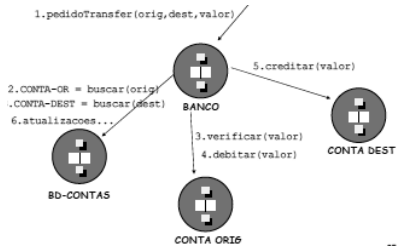
Paradigma Orientado a Objeto

- Debitar OO



Paradigma Orientado a Objeto

+ Transferir OO



Paradigma Orientado a Objeto

+ Em Java

```
public class Conta {
  private String numero;
  private double saldo;

  ...
  public void debitar (double valor) {
    if (this.getSaldo() >= valor)
      this.setSaldo(this.getSaldo()-valor);
    else
      //erro!
  }
  public void creditar (double valor) {
    this.setSaldo(this.getSaldo()+valor);
  }
}
```

OO resolve todos os problemas?

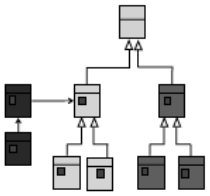
+ Ainda não!
+ Complexidade aumenta sem parar
+ Limitações com objetos
  + fatores de qualidade ainda são prejudicados

Interesse (*Concern*)

+ Em qualquer sistema, vários interesses precisam ser implementados
  + sem eles implementados, seu sistema não atende aos requisitos
+ Funcionais (negócio)
  + creditar, debitar, transferir
+ Não-funcionais (sistêmicos)
  + Logging
  + tratamento de exceções
  + Autenticação
  + Desempenho
  + Concorrência
  + Persistência ...

Problema

+ Código relacionado a certos interesses são transversais
+ Espalhados por vários módulos de implementação (classes)



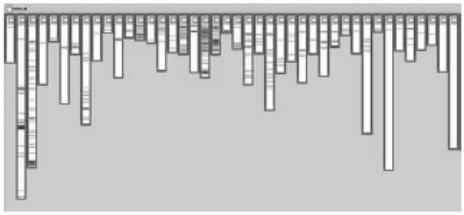
Logging em Conta

```
public class Conta {
  private String numero;
  private double saldo;
  ...
  public void debitar (double valor) {
    if (this.getSaldo() >= valor){
      this.setSaldo(this.getSaldo()-valor);
      System.out.println("ocorreu um debito!");
  }...
 }

// o mesmo para creditar e outros
// tipos de contas bancarias
```

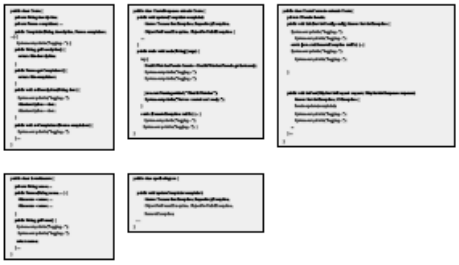
Tipos de Problemas

- Código entrelaçado (*tangling*)
  - código de *logging* é misturado com código de negócio
- Código espalhado (*spread*)
  - código de *logging* em várias classes
- *Logging* é um interesse transversal (*crosscutting concern)*

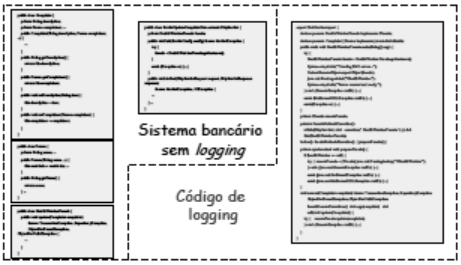
Ex: Logging no Apache Tomcat



Sistema Bancário com *logging*



Sistema bancário Ideal



Programação Orientada a Aspectos (AOP)

- Modularização de interesses transversais
- Promove separação de interesses
- Implementação de um interesse dentro de uma unidade...

Aspecto
nova unidade de
modularização e abstração

Separação de Interesses (*separation of concerns*)

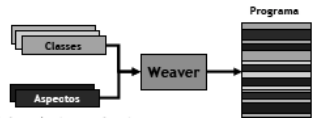
- Para contornar a complexidade de um problema, deve-se resolver uma questão importante ou interesse (concern) por vez
  [Dijkstra 76].
  - Separação de interesses
    - Paradigmas de desenvolvimento de software evoluem para apoiar uma melhor separação de interesses

Implementação com AOP

- Complementa a orientação a objetos
  - novo paradigma?
- Melhoria em reuso e extensibidade
- Separação de interesses
  - relação entre os aspectos e o resto do sistema nem sempre é clara
- Normalmente menos linhas de código

Implementação com AOP

- Parte OO
  - Objetos modularizam interesses não-transversais
- Parte de Aspectos
  - Aspectos modularizam interesses transversais



Aspectos em Java: AspectJ

- Extensão simples e bem integrada de Java
  - gera arquivos .class compatíveis com qualquer máquina virtual Java
- Linguagem orientada a aspectos
- Inclue suporte de ferramentas
  - JBuilder, Eclipse
- Implementação disponível
  - open source
- Comunidade de usuários ativa
  - Mantida por uma grande empresa (IBM)

AspectJ para *logging*

```
public class Conta {
 private String numero;
 private double saldo;
 ...
 public void debitar (double valor) {
  if (this.getSaldo() >= valor){
      this.setSaldo(this.getSaldo()-valor);
      System.out.println("ocorreu um debito!");
 }
 public void creditar (double valor) {
      this.setSaldo(this.getSaldo()+valor);
      System.out.println("ocorreu um credito!");
 }...
```

Inicialmente...

- Temos que identificar os pontos de interesse na execução
- Neste exemplo: ao fazer um crédito ou débito em qualquer conta
- Pontos de junção (join points)
  - pontos na execução de um programa
  - chamadas de métodos
  - acesso a atributos (escrita, leitura)
  - etc

Precisamos ainda agrupar!

- Interesse logging ocorre em todas as chamadas a creditar e debitar
- Precisamos agrupar todos os pontos de junção
- Em AspectJ
  - pointcut (conjunto de pontos de junção)

Pointcut

- agrupamento de pontos de junção (*join points*)
- uso de filtros de AspectJ

Todas as chamadas a creditar de qualquer conta

```
pointcut logCredíto():
     call (* Conta*.creditar(double));

pointcut logDebito():
     call (* Conta*.debitar(double));
```

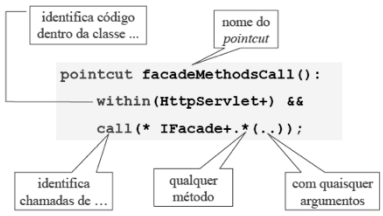
Pointcut

- Identificam join points de um sistema
  - chamadas e execuções de métodos (e construtores)
  - acessos a atributos
  - tratamento de exceções
  - inicialização estática e dinâmica
  - expõe o contexto nos join points
- Composição de join points
  - &&, || e !

Pointcut

- Identificam join points de um sistema
  - chamadas e execuções de métodos (e construtores)
  - acessos a atributos
  - tratamento de exceções
  - inicialização estática e dinâmica
  - expõe o contexto nos join points
- Composição de join points
  - &&, || e !

Pointcut



Pointcut

- Métodos
  - <tipo-acesso> <retorno> <tipo>.<nome>(tiposparametros)

```
call(public void Conta.debitar(double))
```

- Atributos
  - <tipo> <tipo-classe>.<nome>

```
get(double Conta.numero)
```

Pointcut – Padrões

- Notação
  - * denota qualquer tipo e quantidade de caracteres, exceto '.'
  - .. denota qualquer quantidade de caracteres incluindo '.'
  - + denota qualquer subclasse de um dado tipo

```
call(void Cliente.*(*))
```

Todas as chamadas a qualquer método de Cliente que tenha apenas um parâmetro qualquer e retorne void

```
execution(void Conta+.set*(..))
```

Execuções a qualquer método set de Conta e suas subclasses

Próximo passo...

- agora precisamos decidir duas coisas

O que fazer nestes pontos?
Fazer isto antes ou depois do ponto?

Advices (adendos)

- Especifica o código que será executado quando acontecer os pontos indicados
  - parecido com métodos
- Comportamento executado
  - antes (before)
  - depois (after)

Advice para Logging

```
pointcut logCredito():
  call (* Conta*.creditar(double));

after(): logCredito(){
  System.out.println("ocorreu um credito");
}
```

DEPOIS de cada ponto de logCredito, executar o seguinte bloco

Onde colocar estas definições...

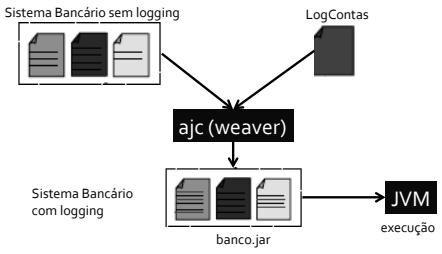
Aspectos

- Unidades de modularização e abstração
  - para interesses transversais
- Agrupa pointcuts e advices
- Parecidos com classes

Aspecto LogContas

```
public aspect LogContas {

  pointcut logCredito():
    call (* Conta.creditar(double));

  pointcut logDebito():
    call (* Conta.debitar(double));

  after (): logCredito(){
    System.out.println("ocorreu um credito");
  }

  after () returning: logDebito(){
    System.out.println("ocorreu um debito");
  }
}
```

Solução com AspectJ



Outro Exemplo

```
public class Conta {
  private String numero;
  private double saldo;
  ...
  public void setNumero(String numero) {
    System.out.println("Vai mudar o num.");
    this.numero = numero;
  }
  public void creditar(double valor) {
    System.out.println("Vai mudar o saldo");
    this.saldo = this.saldo + valor;
  }
  public void debitar(double valor) {
    System.out.println("Vai mudar o saldo");
    this.saldo = this.saldo - valor;
  }
```

Exercícios

1. Implemente a classe Conta e seu respectivo aspecto como mostrado em sala de aula e utilizando este exemplo, crie um aspecto para realizar tratamento de erro da classe conta nos métodos debitar() e creditar(). Em ambos os métodos, se o parâmetro valor for menor ou igual a zero, uma exceção ValorInvalidoException deverá ser lançada. No método debitar(), caso o saldo seja insuficiente para se debitar o valor, a exceção SaldoInsuficienteException deverá ser lançada. Lembre-se que quem deve lançar as exceções são os aspectos.

OBS: nas duas exceções você deve imprimir na mensagem da exceção o valor ou saldo que gerou a exceção (Dica: pesquise sobre o uso do construtor target() e args() em pointcuts)

Exercícios

2. Agora considere a seguinte classe CadastroContas

```
package contas;
public class CadastroContas {
        private RepositorioContas contas;
        ...
        public void transferir(String nDe, String nPara, double
valor) throws                   ContaNaoEncontradaException,
SaldoInsuficienteException {
                Conta de = contas.procurar(nDe);
                Conta para = contas.procurar(nPara);
                de.debitar(valor);
                para.creditar(valor);
        }
}
```

Defina as classes/interfaces necessárias para a classe CadastroContas funcionar e o aspecto `TrocaOrdemArgumentos` que troca a ordem dos números das contas na chamada do método transferir da classe CadastroContas. Use o advice *around*.

Leitura Adicional

- R. LADDAD. AspectJ in Action, 2ª Ed. 2010.
  - Part 1 Understanding AOP and AspectJ
- Sergio Soares. Programação Orientada a Aspectos com AspectJ. Minicurso CBSoft 2010.