

Desenvolvimento de Software Orientado a Aspectos

Eduardo Figueiredo

<http://www.dcc.ufmg.br/~figueiredo>
dcc603@gmail.com

28 Maio 2012

Agenda da Aula

- Introdução a desenvolvimento de software orientado a aspectos
- Interesses centrais e interesses transversais
- Elementos da linguagem AspectJ
 - Pontos de corte e adendos
 - Declarações inter-tipo

Motivação

- Queremos desenvolver software de qualidade
 - Reusável
 - Flexível
 - Confiável, etc.
- A complexidade de software é sempre crescente
 - Traz novos desafios para o desenvolvimento de software

Aspectos...

- Desenvolvimento de software orientado a aspectos (DSOA)
 - Abordagem para desenvolvimento baseada em uma nova abstração: aspecto
- DSOA é usado em conjunto com outras abordagens
 - Geralmente com orientação a objetos (OO)

Um pouco de história

Para contornar a complexidade de um problema, deve-se resolver uma questão importante por vez

Dijkstra, 1976

- Separação de interesses
 - Paradigmas de desenvolvimento de software evoluem para apoiar uma melhor separação de interesses

Programação Estruturada

- Funções e procedimentos
 - Abstrações para ações
- Dados
 - Informação manipulada pelas ações
- Ações e dados são projetados separadamente
 - Inaugurou a era da programação em alto nível

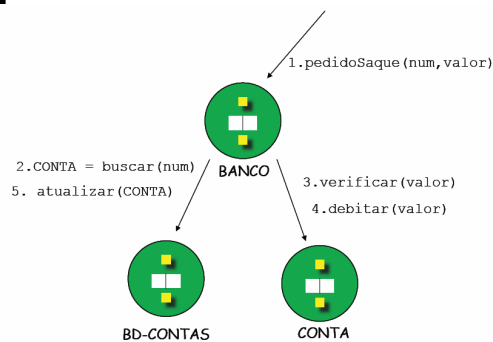
Programação OO

- Abstrações mais próximas do mundo real (problema do cliente)
 - Objetos, não funções
- Objetos reúnem dados e ações em uma mesma entidade
- Um programa OO é um monte de objetos dizendo aos outros o que fazer
 - Troca de mensagens

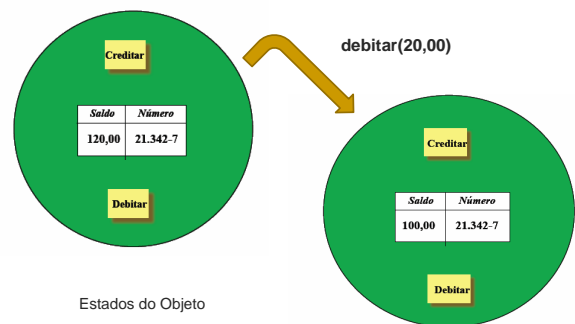
Exemplos de objetos

- Banco
 - Entidade financeira
- Conta
 - Uma conta bancária de um cliente
- Banco de Dados (de Contas)
 - Arquivo que armazena as contas dos clientes

A operação sacar



Objeto Conta



Definição da Classe (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);
    }
}
```

Problemas de OO

- Complexidade de software sempre aumenta
- Fatores de qualidade não são adequadamente tratados
- Separação de interesses em OO
 - Oferece suporte a implementação de requisitos funcionais
 - Não se preocupa com os requisitos não funcionais

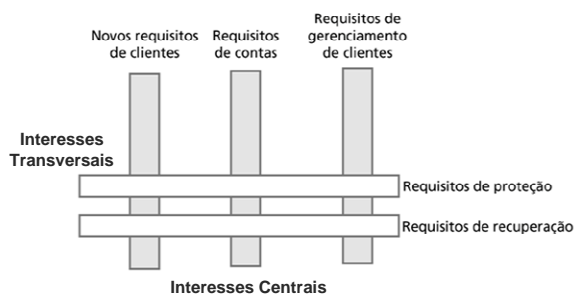
Alguns tipos de interesses

- Interesse funcional
 - Relacionado à uma funcionalidade específica a ser incluída em um sistema
- Interesse de qualidade de serviço
 - Relacionado ao comportamento não funcional
- Interesse organizacional
 - Relacionado aos objetivos e às prioridades da organização
- Interesse de sistema
 - Relacionado à atributos do sistema como um todo

Interesses transversais

- São interesses cuja implementação atravessa uma série de componentes de programa
- Interesses transversais resultam em dois problemas no código
 - Espalhamento (*scattering*)
 - Entrelaçamento (*tangling*)
- Causam problemas quando mudanças devem ser feitas em um interesse

Representação

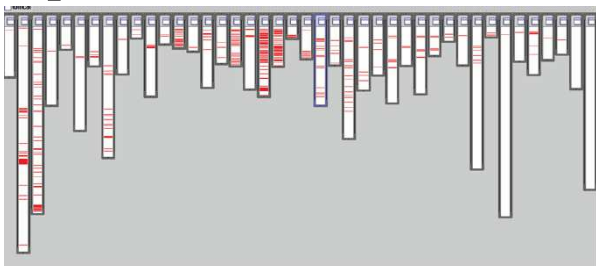


Exemplo: Logging

```
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!");  
        }...  
    }...  
}
```

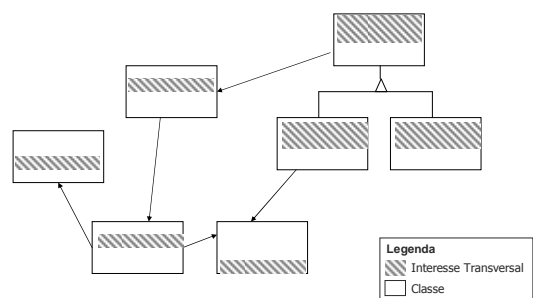
O código em vermelho implementa *Logging* e se espalha por várias partes do sistema

Logging no Tomcat

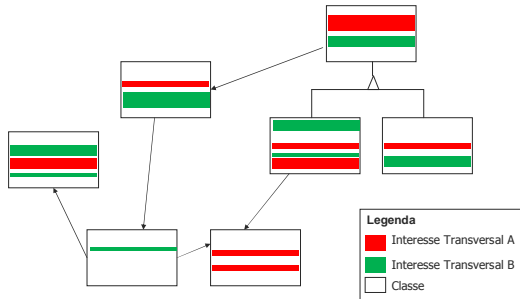


O código em vermelho implementa *Logging* e se espalha por várias partes do sistema

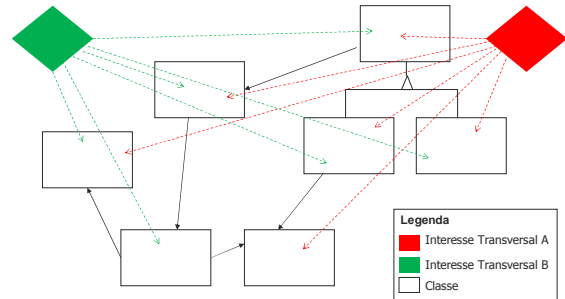
Espalhamento



Entrelaçamento



Solução com Aspectos



Definições de DSOA

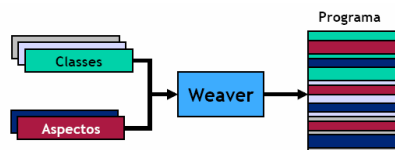
- Um aspecto é uma abstração que implementa um interesse transversal
 - Contém informações de onde deve ser incluído código do interesse
- Um ponto de junção é o local no programa onde pode ser incluído código do interesse
- Um ponto de corte é um conjunto de pontos de junção

Definições de DSOA

- Adendo é o código que implementa o comportamento do interesse
 - Semelhante a um método em classes
- Declarações inter-tipo são métodos, atributos e relacionamentos que implementam o interesse
 - Elementos de associação estática com as classes

DSOA

- Complementa a orientação a objetos
 - Não existe programa que só possuem aspectos
- Classes e aspectos são “costurados” juntos pelo *Weaver*



AspectJ

[AspectJ]

- Linguagem orientada a aspectos mais madura e difundida
- Extensão simples da linguagem Java
 - Gera arquivos .class compatíveis com a máquina virtual Java (JVM)
- Possui bom suporte de ferramenta
 - AspectJ Development Tools (AJDT)
 - Integrado a plataforma Eclipse

[Extensões de AspectJ]

- Pontos de Junção
- Pontos de Corte
- Adendos
- Declarações Intertipo

[Extensões de AspectJ]

- Pontos de Junção
- Pontos de Corte
- Adendos
- Declarações Intertipo

[Modelo de Pontos de Junção]

- Define como e onde será feita a junção entre classes e aspectos
- Exemplos
 - Chamada a métodos e construtores
 - Execução de métodos e construtores
 - Instanciação de objetos
 - Acesso e atualização de dados, etc.

[Exemplos de Pontos de Junção]

- Chamadas ao método **creditar** da classe **Conta**
 - `call(void Conta.creditar(double))`
- Acessos para leitura do atributo **numero** na classe **Conta**
 - `get(String Conta.numero)`

[Quantificação]

- Podemos usar coringas para indicar mais pontos de junção
 - Maior expressividade
- Tipos de coringas
 - * denota qualquer tipo e qualquer quantidade de caractere
 - + denota qualquer subclasse da classe
 - .. denota qualquer quantidade de parâmetros

Exemplos de Quantificação

- Todas as chamadas a qualquer método da classe Cliente que tenham apenas um parâmetro e retorne void
 - `call(void Cliente.*(*))`
- Todas as chamadas a métodos começando com set e qualquer quantidade de parâmetros na classe Conta
 - `call(* Conta.set*(..))`

Outros Exemplos

- Chamadas a qualquer método set* da classe conta e suas subclasses
 - `call(* Conta+.set*(..))`
- Acessos de leitura de qualquer atributo da classe Cliente com tipo String
 - `get(String Cliente.*)`
- Acessos de escrita a qualquer atributo de qualquer classe (de todo o sistema)
 - `set(* *.*)`

Extensões de AspectJ

- Pontos de Junção
- **Pontos de Corte**
- Adendos
- Declarações Intertipo

Ponto de Corte

- Um ponto de corte é um conjunto de pontos de junção
 - Meio de identificar pontos de junção
- Pontos de junção podem ser compostos usando operadores lógicos
 - `&&` (*and*) intercepta um ponto quando ambas as condições são satisfeitas
 - `||` (*or*) intercepta um ponto quando uma das condições é satisfeita
 - `!` (*not*) intercepta todos os pontos que não estão negados na condição

Exemplos de Composição

- Chamadas à métodos set das classes Cliente ou Conta (anônimos)
 - `call(* Cliente.set*(*)) || call(* Conta.set*(*))`
- Ou equivalente (nomeados)
 - `pointcut setCliente() : call(* Cliente.set*(*))`
 - `pointcut setConta() : call(* Conta.set*(*))`
 - `pointcut sets() : setCliente() || setConta()`

Voltando ao exemplo de 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!");
    }...
}
```

Queremos separar o código em vermelho que implementa logging

Definindo Pontos de Corte

Palavra reservada que identifica um ponto de corte

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

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

Definindo Pontos de Corte

Todas as chamadas ao método `creditar` de todas as classes de nome começando com `Conta`

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

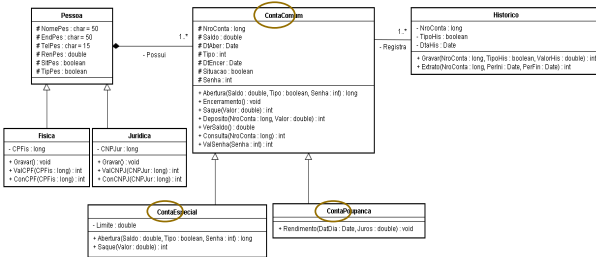
↑

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

↓

Todas as chamadas ao método `debitar` de todas as classes de nome começando com `Conta`

Exemplos de Contas



Extensões de AspectJ

- Pontos de Junção
- Pontos de Corte
- Adendos
- Declarações Intertipo

Adendo

- Especifica o código que será executado quando um ponto de corte for atingido
- O adendo pode ser executado
 - Antes do ponto de corte (*before*)
 - Depois do ponto de corte (*after*)
 - Antes e depois do ponto de corte (*around*)

Definindo um Adendo

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

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

O código do adendo será executado após cada chamada ao método `creditar`

Variações do Adendo *After*

■ After returning

- É interceptado quando o método retorna normalmente sua execução

■ After throwing

- É interceptado quando o método retorna uma situação excepcional (*exception*)

Exemplos de adendo *After*

```
pointcut debitos():
    call(* Conta.debitar(..));

after() returning: debitos(){
    System.out.println("debito deu certo!");
}

after() throwing: debitos(){
    System.out.println("debito deu errado!");
}
```

Expondo o Contexto

- Pontos de corte podem expor alguns valores para a execução do adendo
 - **args()** parâmetros de chamadas de métodos
 - **target()** objeto que recebe a chamada do método ou acesso ao atributo
 - **this()** objeto que efetua a chamada ao método ou acesso ao atributo

Exemplo de Uso (*target*)

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

Conta que recebeu a chamada

```
after(Conta c): logCredito(c){
    System.out.println("ocorreu credito");
    System.out.println("num: "+c.getNumero());
}
```

Imprime o número da conta

Exemplo de Uso (*args*)

```
pointcut logCredito(Conta c, double v):
    call (* Conta.creditar(double)) &&
    target(c) && args(v);
```

Valor do crédito (parâmetro de creditar)

```
after(Conta c, double v): logCredito(c,v){
    System.out.println("ocorreu credito");
    System.out.println("num: "+c.getNumero());
    System.out.println("valor: " + v);
}
```

Imprime o valor creditado

Adendo do tipo *Around*

- Substitui o ponto de junção interceptado
 - **proceed()** permite executar o ponto interceptado

```
pointcut logDebito(Conta c, double v):
    call (* Conta.debitar(double)) &&
    target(c) && args(v);

void around(Conta c, double v): logDebito(c,v){
    if(v > c.getSaldo())
        System.out.println("Sem saldo!");
    else
        proceed(c,v);
}
```

Extensões de AspectJ

- Pontos de Junção
- Pontos de Corte
- Adendos
- **Declarações Intertipo**

Declarações Intertipo

- Ponto de corte e adendo afetam o comportamento dinâmico do sistema
- AspectJ também fornece mecanismos para alterar a estrutura estática
- Tipos de declarações intertipo
 - Introduzir membros (métodos, atributos e construtores)
 - Modificar a hierarquia de herança

Exemplos: Introduzir Membros

```
private float Conta.chequeEspecial;
```

Introduz um novo atributo chamado `chequeEspecial` do tipo `float` na classe `Conta`

```
public static void Conta.main(String[] args){...}
```

Introduz um novo método `main` na classe `Conta`

Exemplos: Modificar a Herança

```
declare parents: Conta extends ContaAbstrata;
```

A classe `Conta` passa a estender `ContaAbstrata`

```
declare parents: Conta implements Serializable;
```

A classe `Conta` passa a implementar a interface `Serializable`

```
declare parents banco.entidades.*  
implements Serializable;
```

Toda classe e interface do pacote `banco.entidades` passa a estender/implementar `Serializable`

Exemplos de Aspectos

Aspecto

- Entidade modular semelhante a uma classe
 - Além de métodos e atributos, reúne também pontos de corte, adendos e declarações intertipo
- Um aspecto pode interceptar uma ou várias classes do sistema
- Tem como objetivo implementar um interesse transversal
 - Classes implementam os interesses centrais

O aspecto Logging

```
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");  
    }  
}
```

Outro Exemplo (Java)

Código da classe ATM que possui código relacionado a Logging

```
public class ATM {  
    ...  
    private int displayMainMenu() {  
        screen.displayMessageLine( "nMain menu:" );  
        screen.displayMessageLine( "1 - View my balance" );  
        screen.displayMessageLine( "2 - Withdraw cash" );  
        screen.displayMessageLine( "3 - Deposit funds" );  
        screen.displayMessageLine( "4 - Exit\n" );  
        screen.displayMessage( "Enter a choice: " );  
        int option = keypad.getInput();  
        Logger.log("User option: " + option);  
        return option;  
    }  
}
```

Aspecto Logging: Opção 1

```
public aspect Logging { Código parcial do aspecto Logging  
    ...  
    public pointcut displayMainMenuPC() :  
        call (private int ATM.displayMainMenu());  
    after() returning (int option): displayMainMenuPC() {  
        Logger.log("User option: " + option);  
    }  
}
```

O aspecto Logging pega o retorno da chamada ao método displayMainMenu

O que fazer com esta classe?

```
public class Logger { Código da classe Logger  
    private static Vector<String> logs = new Vector<String>();  
  
    public static void log(String text) {  
        logs.add(text);  
    }  
  
    public static void printLog() {  
        for (int i=0; i<logs.size(); i++) System.out.println( logs.get(i) );  
    }  
}
```

Este código pode ser movido para o aspecto Logging, certo?

Aspecto Logging: Opção 2

```
public aspect Logging { Código parcial do aspecto Logging  
    private Vector<String> logs = new Vector<String>();  
    private static final int LOG = 0;  
  
    public void log(String text) { O código da classe Logger foi movido para o aspecto Logging  
        logs.add(text);  
    }  
    ...  
    public pointcut displayMainMenuPC() :  
        call (private int ATM.displayMainMenu());  
    after() returning (int option): displayMainMenuPC() {  
        log("User option: " + option);  
    }  
}
```

Aspecto Logging: Opção 3

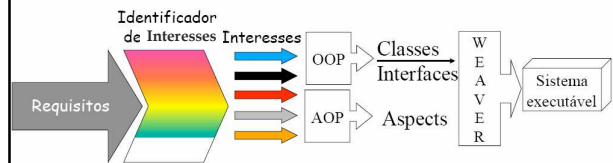
Código parcial do aspecto Logging

```
public aspect Logging extends Logger {  
    ...  
    public pointcut displayMainMenuPC() :  
        call (private int ATM.displayMainMenu());  
    after() returning (int option): displayMainMenuPC() {  
        log("User option: " + option);  
    }  
}
```

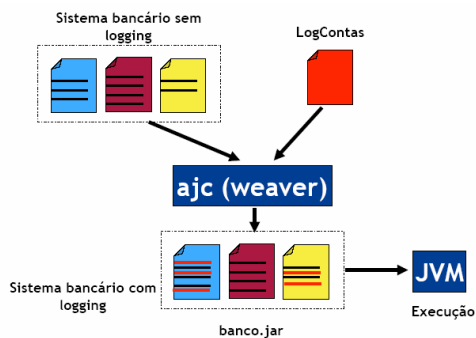
Aspectos podem herdar de outra classe ou de outro aspecto.

O Processo DSOA

O processo de DSOA



Em AspectJ



Vantagens do DSOA

- Separação de interesses transversais
 - Melhor modularidade, diminui a complexidade dos componentes, mais fácil o reuso, extensão, etc.
- Inconsciência e produtividade
 - As classes não conhecem os aspectos
 - Interesses podem ser implementados em paralelo
- Configuração (plugabilidade)

Desvantagens de DSOA

- Novo paradigma
 - Requer aprendizado e treinamento (nova forma de pensar)
- Fragmentação de código
 - Muitos componentes pequenos
- Ausência de suporte ferramental
 - Poucas ferramentas e poucos recursos
- Linguagens ainda pouco maduras

Bibliografia Principal

- Ian Sommerville. Engenharia de Software, 9ª Edição. Pearson Education, 2011.
 - Cap. 21 Eng. de Soft. Orientada a Aspectos
- 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.