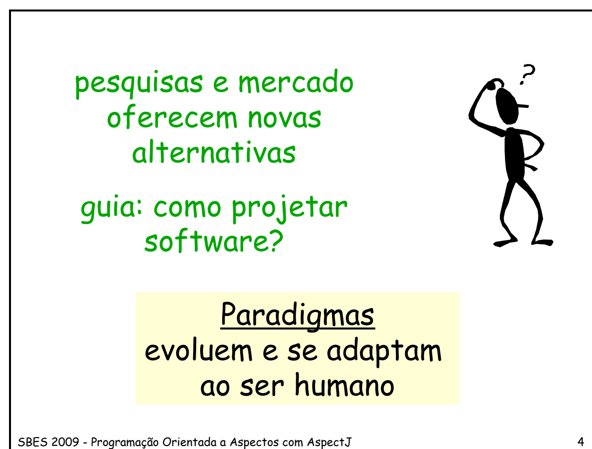
Objetivos

- Introduzir os principais conceitos da Programação Orientada a Aspectos
- Apresentar a linguagem AspectJ
  - exemplos em uma aplicação real
- Discutir a adoção de AspectJ em projetos reais

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 estruturado

- Funções e procedimentos
  - abstrações para ações
- Dados
  - abstrações para objetos do mundo real
- Funções e dados projetados separadamente
- Inaugurou era dos projetos de alto-nível

Exemplo: sistema bancário estruturado

debitar(numero,valor)
1. Acha conta no BD
2. Testa saldo disponível
3. Diminui saldo da conta
4. Atualiza mudança no BD

transferir(orig,dest,valor)
1. Acha conta **orig** no BD
2. Acha conta **dest** no BD
3. Testa saldo disponível em **orig**
4. Diminui saldo de **orig**
5. Aumenta saldo de **dest**
6. Atualiza **orig** e **dest** no BD

Projeto de qualidade

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

Voltando ao exemplo

E se a equipe que desenvolve as funções for separada da equipe que desenvolve a parte de banco de dados?

Se quisermos usar as operações em um sistema de bolsa de valores?

Erro na atualização de dados

<u>debitar(numero,valor)</u>
1. Acha conta no BD
2. Testa saldo disponível
3. Diminui saldo da conta
4. Atualiza mudança no BD

Mudança na política de débito/crédito

Mudança no meio de armazenamento em conta de dados

<u>transferir(orig,dest,valor)</u>
1. Acha conta **orig** no BD
2. Acha conta **dest** no BD
3. Testa saldo disponível em **orig**
4. Diminui saldo de **orig**
5. Aumenta saldo de **dest**
6. Atualiza **orig** e **dest** no BD

Mas, observando o mundo real...

- Existe um banco, que define as atividades bancárias
- Existem contas no banco, que possuem políticas de operação
- Existe um banco de dados do banco, que guarda e recupera contas e clientes

Programa orientado a objetos

- Abstrações bem mais próximas do mundo do problema
  - objetos, não funções
- Em um programa, “tudo” é objeto
- Um programa é um monte de objetos dizendo aos outros o que fazer
  - mensagens

Debitar OO

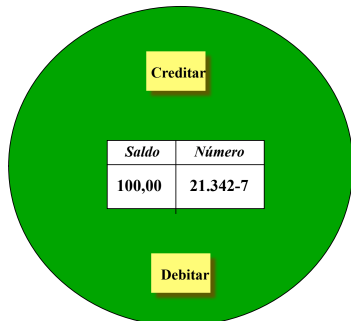
<u>Analisar de novo</u>
Política de débito;crédito
API de BD
Correção de erros
Reuso, modularidade

1.pedidoSaque(num,valor)
BANCO
2.CONTA = buscar(num)
5. atualizar(CONTA)
3.verificar(valor)
4.debitar(valor)
BD-CONTAS
CONTA

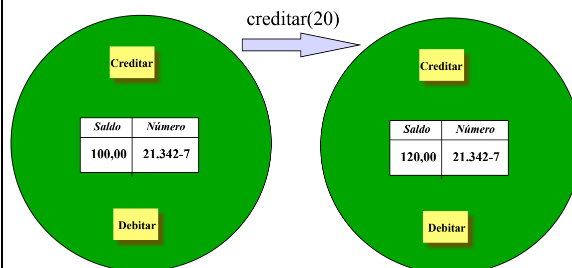
“A orientação a objetos tem por objetivo diminuir a distância conceitual entre o mundo real e o computacional.”

Problemas do Mundo Real
Distância Conceitual
Soluções Computacionais

Objeto conta bancária



Estados do objeto conta



Definição classes - 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);
  }
}
```


Como projetar OO?

- Mais difícil
- Experiência
- Padrões

OO resolve nosso problema?

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

Conceito novo – *CONCERN* (interesse)

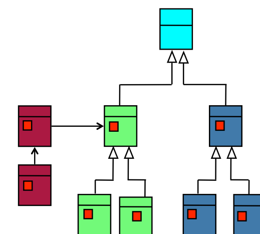
- Requisitos que devem ser implementados em um sistema
- Em qualquer sistema, vários interesses precisam ser implementados
  - sem eles implementados, seu sistema não atende aos requisitos

Tipos de interesses

- Funcionais (negócio)
  - creditar, debitar, transferir
- Não-funcionais (sistêmicos)
  - *logging*
  - tratamento de exceções
  - autenticação
  - desempenho
  - concorrência e sincronização
  - persistência...

Problema (mesmo com OO)

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



Cada cor um interesse diferente

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
```


Então, ao problema

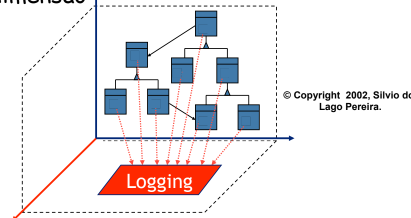
- Código entrelaçado (tangling)
  - código de *logging* é misturado com código de negócio
- Código espalhado (spread)
  - código de *logging* em várias classes
- *Logging* é um interesse transversal (*crosscutting concern*)

Fatores de qualidade

- Extensibilidade e correções
  - mudanças misturadas e espalhadas
  - mudança na API de logging
  - mudança na política de logging
- Reusabilidade
  - usar contas e clientes em sistemas sem logging

Causa do problema

- Implementação mapeia os requisitos em uma única dimensão



- Interesse é transversal à dimensão de implementação

Ex: Logging no Apache Tomcat

- Linhas vermelhas: logging
  - não estão no mesmo lugar
  - nem mesmo em poucos lugares

Sistema bancário com logging

Código de logging é vermelho…

O ideal seria se…

Sistema bancário sem *logging*

Código de logging

Como conseguir isso?

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*)

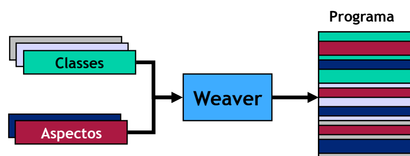
- *Para contornar a complexidade de um problema, deve-se resolver uma questão importante ou interesse (concern) por vez [Dijkstra 76].*

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

Em uma linguagem orientada a aspectos

- Parte OO
  - objetos modularizam interesses não-transversais
- Parte de aspectos
  - aspectos modularizam interesses transversais



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)

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


Para começar...

- 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
- uso de filtros de AspectJ

Todas as chamadas a creditar de qualquer conta

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

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

Identificamos os pontos...

- agora precisamos decidir duas coisas

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

SBES 2009 - Programação Orientada a Aspectos com AspectJ 37

Advice (adendo)

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

SBES 2009 - Programação Orientada a Aspectos com AspectJ 38

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

SBES 2009 - Programação Orientada a Aspectos com AspectJ 39

Para terminar, onde colocar tudo isto?

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

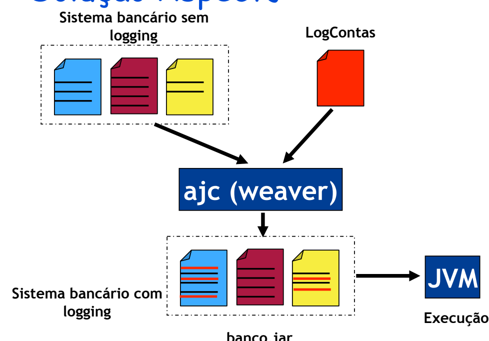
SBES 2009 - Programação Orientada a Aspectos com AspectJ 40

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");
  }
}
```

SBES 2009 - Programação Orientada a Aspectos com AspectJ 41

Solução AspectJ



SBES 2009 - Programação Orientada a Aspectos com AspectJ 42

Exercício

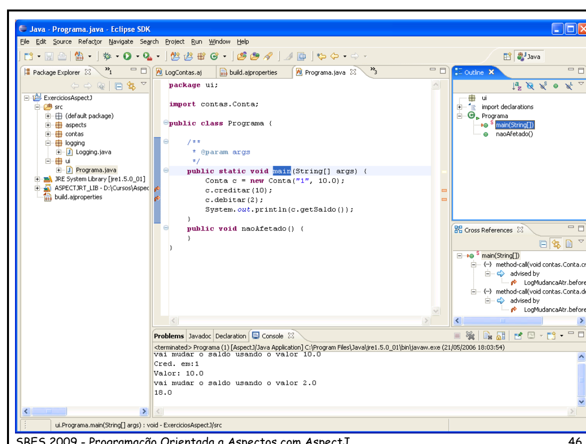
```
public class Conta {
  private String numero;
  private double saldo;
  ...
  public void setNumero(String numero) {
    System.out.println("Vai mudar o num.");
    this.numero = numero;
  }
  public void creditar(double valor) {
    System.out.println("Vai mudar o saldo");
    this.saldo = this.saldo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


Exercício

- Faça um aspecto que modularize este concern de logging
- Dica: usar dois pointcuts e dois advices
- Usem o notebook...

Solução

```
public aspect LogMudancaAtr {
  pointcut logNumero():
    call (* Conta*.setNumero(String));
  pointcut logSaldo():
    call (* Conta*.creditar(double)) ||
    call (* Conta*.debitar(double));

  before (): logNumero(){
    System.out.println("vai mudar o num.");
  }
  before (): logSaldo(){
    System.out.println("vai mudar o saldo");
  }
}
```

AspectJ e exemplos

Parte 2

AspectJ

- Modelo de pontos de junção
- *Pointcut*
- *Advice*
- *Inter-type declaration*
- Aspectos

Pontos de junção (*join points*)

- Temos que identificar os pontos de interesse na execução
- Pontos de junção
  - pontos na execução de um programa
  - chamadas de métodos
  - acesso a atributos (escrita, leitura)
  - etc.

SBES 2009 - Programação Orientada a Aspectos com AspectJ 49

Pontos de junção

- Pontos de execução de um programa
- Tipos disponíveis
  - chamada e execução de método
  - chamada e execução de construtor
  - acesso de escrita e leitura de um atributo
  - execução do tratamento de exceções
  - execução da inicialização de classes e objetos

SBES 2009 - Programação Orientada a Aspectos com AspectJ 50

Pointcut (conjunto de pontos de junção)

- Meio de identificar um conjunto de pontos de junção
- Filtro de pontos usando condições lógicas e padrões de nomes
- Exemplo: chamadas ao método `creditar` de `Conta`

```
call(void Conta.creditar(double))
```

casa com os pontos onde se **chama** o método `creditar` de `Conta` ≠executar

SBES 2009 - Programação Orientada a Aspectos com AspectJ 51

Pointcuts primitivos

- *Pointcuts* primitivos **baseados em nomes**
  - especificação baseada nomes de métodos, tipos de parâmetros, etc.)
  - enumeração explícita de nomes
- *Pointcuts* primitivos **baseados em propriedades**
  - especificação baseada em outras propriedades de métodos
  - uso de *wildcards* (padrões)

call (void Figure.make*(..))

SBES 2009 - Programação Orientada a Aspectos com AspectJ 52

Pointcuts – designadores

- Para cada tipo de ponto de junção, existe um designador
- Indica os tipos de pontos que queremos interceptar
- Principais
  - chamadas de métodos
  - execução de métodos
  - acesso a atributos

SBES 2009 - Programação Orientada a Aspectos com AspectJ 53

Designadores - exemplos

```
call(void Conta.creditar(double))
```

diferentes!

```
execution(void Conta.creditar(double))
```

```
get(double Conta.saldo)
```

se atributos privados, aspectos não podem fazer este acesso!

```
set(String Conta.numero)
```

SBES 2009 - Programação Orientada a Aspectos com AspectJ 54

Assinaturas de pontos de junção

- Métodos

<tipo-acesso> <retorno> <tipo>.<nome>(tipos-parametros)

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

- Atributos

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

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


Padrões

- Podemos usar os padrões * e + para acolher mais pontos de junção
- Pode ter resultados perigosos se não utilizados com cuidado!
  - alta expressividade

Quantificação

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

```
call(void Conta.*(double))
```


Pointcuts primitivos baseados em propriedades

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

Exemplos de padrões

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

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

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

Todas as chamadas a qualquer método de `Cliente` que tenha apenas um parâmetro qualquer

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

Todas as execuções a métodos set de `Conta` que tenham qualquer número de parâmetros, de qualquer tipo

Mais padrões

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

Execuções a qualquer método `set` de `Conta` e suas subclasses

```
get(String Cliente.*)
```

Todos os acessos de leitura a atributos de `Cliente` do tipo `String`

```
set(* *.*)
```

Todos os acessos de escrita a qualquer atributo de qualquer classe – do sistema todo

Outros designadores

```
call(Cliente.new(..))
```

Todas as chamadas a qualquer construtor de `Cliente`

```
handler(IOException)
```

Todos os lançamentos da exceção `IOException`

Composição de *pointcuts*

- *pointcuts* primitivos

```
pointcut setCliente():call(* Cliente.set*(*));
```

```
pointcut setConta():call(* Conta.set*(*));
```

- podemos compor estes *pointcuts*

```
pointcut sets(): setCliente() || setConta();
```

OU

```
pointcut sets():
     call(* Cliente.set*(*)) ||
     call(* Conta.set*(*));
```

SBES 2009 - Programação Orientada a Aspectos com AspectJ 61

Operadores

- `&&` **(and)** – intercepta quando ambos são `true`
- `||` **(or)** – intercepta quando um dos dois é `true`
- `!` **(not)** – intercepta todos que não estão no *pointcut* negado

Pergunta:

o que aconteceria se tentássemos...

```
pointcut sets():
     call(* Cliente.set*(*)) &&
     call(* Conta.set*(*));
```

SBES 2009 - Programação Orientada a Aspectos com AspectJ 62

Mais designadores

- Alguns designadores especiais podem ser usados
- Definem condições de filtro mais avançadas sobre os pontos de junção
- Exemplos:
  - `this`
  - `target`
  - `cflow`
  - `within`

SBES 2009 - Programação Orientada a Aspectos com AspectJ 63

`this`, `target`

restringe tipo de onde vem a chamada

```
poincut setContadaFachada():
     call(* Conta.creditar(*)) &&
     this(Fachada);
```

diferentes!

```
poincut setContaPoupanca():
     execution(* Conta.creditar(*)) &&
     this(Poupanca);
```

restringe tipo de quem recebe a chamada

```
poincut setContaPoupanca():
     call(* Conta.creditar(*)) &&
     target(Poupanca);
```

SBES 2009 - Programação Orientada a Aspectos com AspectJ 64

`cflow`

```
poincut setSaldoNoSaque():
     call(* Conta.setSaldo(..)) &&
     cflow(execution(* Fachada.saque(..)));
```

só queremos as chamadas a setSaldo que forem feitas no fluxo de execução do método saque da Fachada

SBES 2009 - Programação Orientada a Aspectos com AspectJ 65

Explicando o `cflow`...

```
public class Fachada { ...
  public void saque(String num, double valor){
     ...c.debitar(valor);...
  }
```

```
public class Conta { ...
  public void debitar(double valor){
    if (this.getSaldo() >= valor)
      this.setSaldo(this.getSaldo()-valor);
  }
  ...
  public void setSaldo(double saldo) {
      this.saldo = saldo;
  }
}
```

Seria interceptado apenas quando saque for chamado

SBES 2009 - Programação Orientada a Aspectos com AspectJ 66

within

```
poincut setSaldoEmConta():
     call(* Conta.setSaldo(..)) &&
     within(Conta);
```

só queremos as chamadas a `setSaldo` que forem feitas de dentro da classe `Conta`

Resumo dos designadores

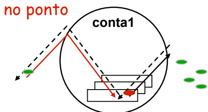
- call — `call(void Conta.creditar(..))`
- execution — `execution(void Conta.*(double))`
- handler — `handler(IOException)`
- get, set — `get(double Conta.saldo)`
- within, withincode — `within(contas.*)`, `withincode(Programa.main(..))`
- cflow, cflowbelow — `cflow(call(Banco.saque(..)))`
- this — `this(Banco)`
- target — `target(Conta)`
- args — `args(x,y)`

Advice

- Define o que fazer nos pontos de junção indicados (comportamento ortogonal)
  - **before** (antes)
  - **after** (depois)
  - **around** (em volta...)

before

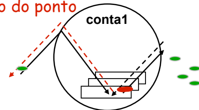
before executa antes de chegar no ponto




```
poincut setSaldo():
  call(* Conta.setSaldo(..));

before(): setSaldo(){
  System.out.println("vai mudar o saldo");
}
```


after

after executa na hora que estiver voltando do ponto




```
poincut setSaldo():
  call(* Conta.setSaldo(..));

after(): setSaldo(){
  System.out.println("mudou o saldo");
}
```


Variantes de after

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

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

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

Executado quando ocorrer a exceção em alguma chamada de debitar

Voltando ao *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 (): logDebito(){
    System.out.println("ocorreu um debito");
  }
}
```

Sem graça! gostaria de registrar a conta e o valor do debito!

Expondo o contexto de execução

- *pointcut* pode expor alguns valores
  - parâmetros de chamadas e execuções de métodos
  - objeto que executa ou chama um método
  - objeto no qual um método é executado ou chamado
- *advice* pode usar valor
  - e até substitui alguns destes!

parâmetros – número da conta

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

conta c que recebeu a chamada

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

imprime numero da conta creditada

parâmetros – valor

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

define o parametro - valor do credito definido

```
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 valor

Alguns pontos de junção e seus contextos

Ponto de junção	Objeto "corrente" "this"	Objeto "alvo" "target"	Argumentos "args"
Method Call	executing object	target object	method args
Method Execution	executing object	executing object	method args
Constructor Call	executing object	------	constructor args
Constructor Execution	executing object	executing object	constructor args
Field reference	executing object	target object	------
Field assignment	executing object	target object	assigned value
Handler execution	executing object	executing object	caught exception

around

- Substitui o ponto de junção escolhido por um código definido
- Pode escolher se o ponto original será executado

```
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);
}
```

Tratamento de exceções com aspectos

Outras declarações *inter-type*

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

Aspecto introduz novo método `main` na classe `Conta`

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

Aspecto define que `Conta` é uma subclasse de `ContaAbstrata`

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

Aspecto define que `Conta` deve implementar uma interface `Serializable` para envio das contas pela rede

Herança e especialização de aspectos

- *pointcuts* podem ter *advice* adicional
  - aspecto abstrato com
    - —*pointcut* concreto
    - —sem advice definido para o *pointcut*
  - módulo pode expor *pointcuts* bem-definidos

```
abstract aspect AtualizaDado {
  pointcut alteracao(Conta c):
    set(Conta.saldo) && this(c);
}
```


Especialização de aspectos

```
aspect AtualizaDadoBD extends AtualizaDado {
  after(Conta c): alteracao(c) {
    // atualiza no BD
  }
}
```

```
aspect AtualizaDadoArq extends AtualizaDado {
  after(Conta c): alteracao(c) {
    // atualiza no arquivo
  }
}
```

Dependendo do aspecto selecionado no momento do weaving, a atualização será feita no BD ou no arquivo

Herança e especialização de aspectos

- *pointcuts* abstratos podem ser especializados
  - aspecto abstrato com
    - —*pointcut* abstrato
    - —*advice* concreto definido sobre *pointcut* abstrato

Tratamento de exceção reusável 1

```
abstract aspect TrataExcecao {
  abstract pointcut exceptionJoinPoint();
  Object around(): exceptionJoinPoints() {
    Object o = null;
    try { o = proceed(); }
    catch (Throwable ex) {
      this.handling(ex);
    }
    return o;
  }
  abstract void handling(Throwable e);
}
```


Tratamento de exceção reusável 2

```
abstract aspect TrataExcecaoServlet
      extends TrataExcecao {
  void handling(Throwable e) {
    // trata exceção em servlets
  }
}
```

```
abstract aspect TrataExcecaoApplet
      extends TrataExcecao {
  void handling(Throwable e) {
    // trata exceção em applets
  }
}
```

Tratamento de exceção concreto

```
aspect TrataExcecaoSistemaXXXX
      extends TrataExcecaoServlet {
  pointcut exceptionJoinPoint():
    call(... // pontos onde as exceções
             // devem ser tratadas
}
```

```
aspect TrataExcecaoSistemaYYYY
      extends TrataExcecaoServlet {
  pointcut exceptionJoinPoint():
    call(... // pontos onde as exceções
             // devem ser tratadas
}
```


Aspectos reusáveis

- Tratamento de exceções
  - uso de *pointcuts* abstratos
  - hierarquia de aspectos
- Reuso ainda é uma incógnita em AOP
  - mas em OO também! ☺

Categorias de Aspectos

- Aspectos de **Desenvolvimento**:
  - facilitam tarefas durante o processo de desenvolvimento
  - são facilmente removidos do sistema
  - ex.: auditoria e contratos
- Aspectos de **Produção**:
  - implementam interesses que se espalham através de classes
  - tendem a afetar um número menor de classes
  - ex.: tratamento de exceções e segurança
- Aspectos **Reutilizáveis**:
  - requerem mais experiência com POA
  - especialização de aspectos
- Categorias não são disjuntas

AspectJ: pontos positivos

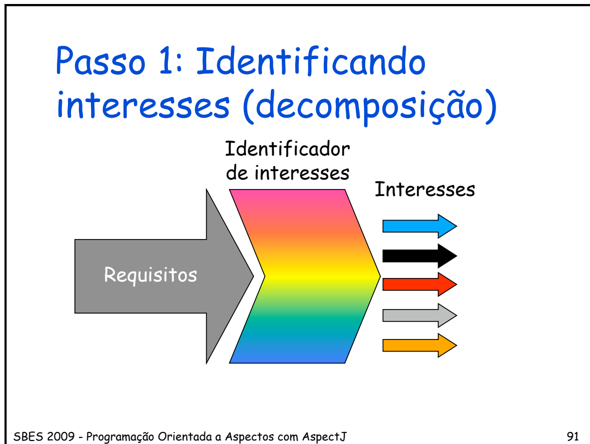
- Separação de interesses
  - modularidade, reuso, e extensibilidade
  - diminuição da complexidade
- Inconsciência (*obliviousness*)
- Produtividade
  - paralelismo
- Permite implementação e testes progressivos
  - plugabilidade

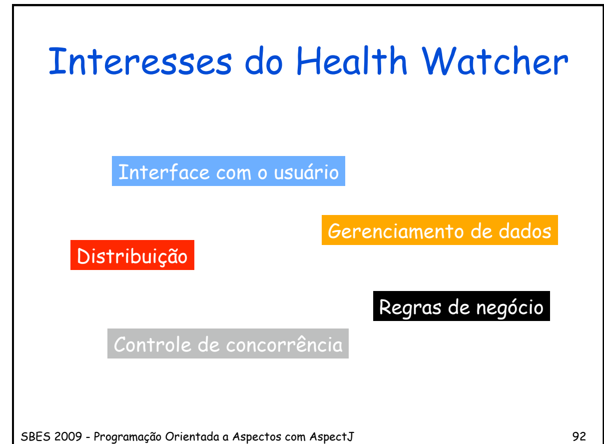
AspectJ: pontos negativos

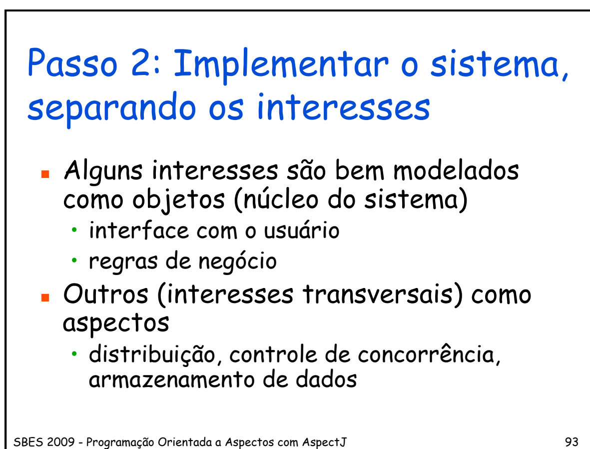
- Novo paradigma
  - aprendizado, entendimento do comportamento de classes
  - relação entre classes e aspectos deve ser minimizada
  - inconsciência (*obliviousness*)
- Fragmentação de código
- Projeto da linguagem
  - tratamento de exceções
  - conflitos entre aspectos
- Requer suporte de ferramentas
- Combinação (apenas) estática

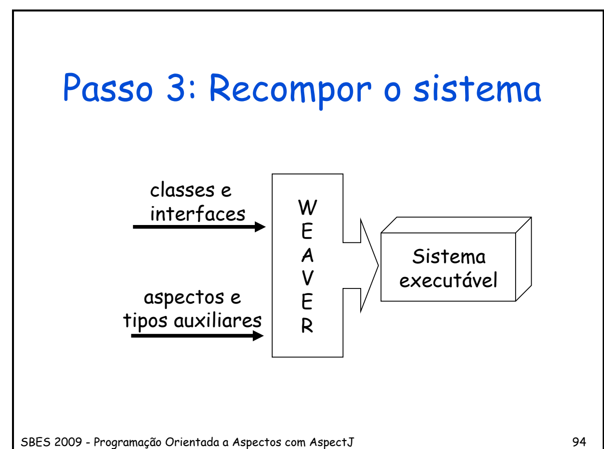
AO não é apenas P!

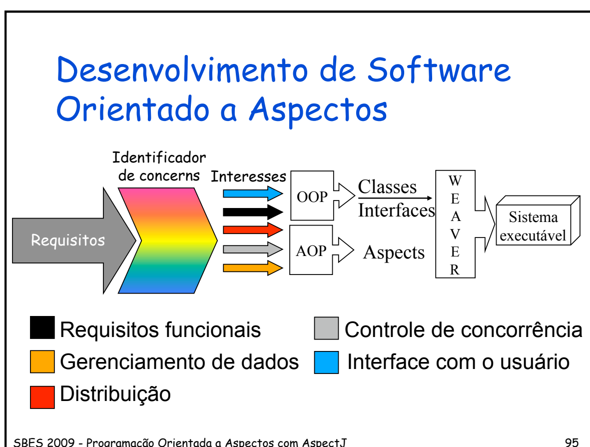
- Aspectos sugerem análise e projeto pensando em aspectos
- AOSD: tendência em engenharia de sofware
- Ao entender o problema e definir a solução, interesses transversais aparecem cedo

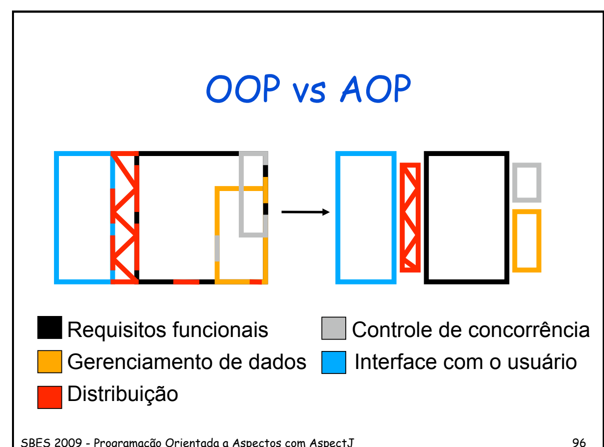

Passo 1: Identificando interesses (decomposição)
Identificador de interesses
Interesses
Requisitos
SBES 2009 - Programação Orientada a Aspectos com AspectJ
91


Interesses do Health Watcher
Interface com o usuário
Gerenciamento de dados
Distribuição
Regras de negócio
Controle de concorrência
SBES 2009 - Programação Orientada a Aspectos com AspectJ
92


Passo 2: Implementar o sistema, separando os interesses
Alguns interesses são bem modelados como objetos (núcleo do sistema)
interface com o usuário
regras de negócio
Outros (interesses transversais) como aspectos
distribuição, controle de concorrência, armazenamento de dados
SBES 2009 - Programação Orientada a Aspectos com AspectJ
93


Passo 3: Recompor o sistema
classes e interfaces
aspectos e tipos auxiliares
W E A V E R
Sistema executável
SBES 2009 - Programação Orientada a Aspectos com AspectJ
94


Desenvolvimento de Software Orientado a Aspectos
Identificador de concerns
Interesses
Requisitos
OOP
Classes
Interfaces
AOP
Aspects
W E A V E R
Sistema executável
Requisitos funcionais
Controle de concorrência
Gerenciamento de dados
Interface com o usuário
Distribuição
SBES 2009 - Programação Orientada a Aspectos com AspectJ
95


OOP vs AOP
Requisitos funcionais
Controle de concorrência
Gerenciamento de dados
Interface com o usuário
Distribuição
SBES 2009 - Programação Orientada a Aspectos com AspectJ
96

Pesquisas em DSOA

- Em todas as atividades ligadas ao desenvolvimento de software
  - requisitos, analise, projeto, implementação e testes
- Falta integrar as diversas pesquisas na área
  - teremos um "Processo de DSOA"?

AOSD na indústria

- Uso de ferramentas
  - Spring AOP
  - JBoss AOP
  - AspectWerkz
  - PostSharp (.NET)
  - Compose*
  - EOS
  - Glassbox

vide referências no final

AOSD na indústria
Rápida consulta a listas de discussão em 2007

- Instituição financeira
  - projeto médio (8 desenvolvedores)
  - logging, transaction, **violação de código fonte**
- CarnegieLearning.com
  - projeto grande (12 desenvolvedores, +1000 classes Java)
  - **violação de código fonte**

AOSD na indústria

- Uma grande empresa (Dinamarca)
  - Projeto com mais de 80 pessoas
    - logging de desempenho, framework para GUI assíncrona e vários outros cc a implementar
  - Projeto com mais de 15 pessoas
    - uso de Spring para implementar cc
  - Projeto com 20 pessoas
    - usa C# e pretende introduzir AOP

AOSD na indústria

- 3 projetos no leste asiático
  - mais de 40 pessoas em cada por mais de 2 anos
  - redes domésticas, telecomunicações e desenvolvimento de produtos eletrônicos para o consumidor
  - não apenas gerenciou *crosscutting concerns*, mas separou código dependente de plataforma do código independente de plataforma
  - em dois deles, limitações de compilação obrigaram o uso de adaptações OO para simular AOP

AOSD na indústria

- Empresas conhecidas que usam AOP
  - IBM
    - mantém AspectJ
  - Motorola
    - WEAVR
  - Microsoft
    - Policy Injection Application Block
  - CESAR
    - Flip (Meantime) - SPL

Conclusão

- OO surgiu com Simula 67
  - começou a ser disseminado na indústria depois de ... 20 anos?
    - padrões de projetos
- AOP tem 10 anos
  - algum uso em grandes empresas

SBES 2009 - Programação Orientada a Aspectos com AspectJ 103

Será que AOP vai pegar?

Parece que vai...

cada empresa deve avaliar seus riscos
adotar AOP em projetos menores
situações menos críticas

SBES 2009 - Programação Orientada a Aspectos com AspectJ 104

Referências

- AspectJ (www.eclipse.org/aspectj)
- Spring AOP (www.springframework.org)
- JBoss AOP (labs.jboss.com/jbossaop)
- AspectWerkz (aspectwerkz.codehaus.org)
- PostSharp (www.postsharp.org)
- EOS (www.cs.iastate.edu/~eos)
- WEAVR (www.iit.edu/~concur/weavr)
  - www.jot.fm/issues/issue_2007_08/article3.pdf
- Policy Injection Application Block (msdn2.microsoft.com/en-us/library/ Bb410104.aspx)
- http://groups.yahoo.com/group/asoc-br/

SBES 2009 - Programação Orientada a Aspectos com AspectJ 105

Referências

- Gregor Kiczales et. al. Aspect-Oriented Programming. European Conference on Object-Oriented Programming, ECOOP'97
- AspectJ Programming Guide
  - http://eclipse.org/aspectj/doc/released/ progguide/index.html
- AOSD
  - http://aosd.net
- Outros materiais em
  - http://www.cin.ufpe.br/~scbs/aspectos

SBES 2009 - Programação Orientada a Aspectos com AspectJ 106

Software Productivity Group

http://www.cin.ufpe.br/spg

SBES 2009 - Programação Orientada a Aspectos com AspectJ 107

Programação Orientada a Aspectos com AspectJ

Sérgio Soares
scbs@cin.ufpe.br