Especialização em Engenharia de Software

Programação Orientada a Aspectos Parte 1

Sérgio Soares
DSC – UPE
sergio@dsc.upe.br

Departamento de Sistemas Computacionais

Objetivos

- Introduzir os principais conceitos da Programação Orientada a Aspectos
- Apresentar a linguagem de programação AspectJ
- Apresentar exemplos de uso de aspectos em AspectJ
- Contextualizar a Programação Orientada a Aspectos em um processo de desenvolvimento de software

Conteúdo Programático

1. Introdução à programação orientada a aspectos
   - Contexto, motivação, histórico
   - Interesses transversais, espalhamento e entrelaçamento de código
   - O modelo de aspectos
2. Programação com AspectJ
   - Sintaxe e Semântica
   - Ambiente de programação
   - Exemplos
3. Aplicações
4. Desenvolvimento de software orientado a aspectos

Dia 1

1. Introdução à programação orientada a aspectos
   - Contexto, motivação, histórico
   - Interesses transversais, espalhamento e entrelaçamento de código
   - O modelo de aspectos: aspectos, pontos de combinação, combinação, quantificação e transparência
   - Demonstração e Exercícios

Dia 2

2. Programação com AspectJ
   - Sintaxe e Semântica
   - Ambiente de programação
   - Exemplos

- Exercícios

Dia 3

3. Aplicações
   - aspectos de desenvolvimento
   - aspectos de produção
   - aspectos de reutilização

- Exercícios

Dia 4

4. Desenvolvimento de software orientado a aspectos
   - processo de desenvolvimento
   - atividades
     - —gereniamento
     - —requisitos
     - —análise e projeto
     - —testes

(Especificação do trabalho em grupo)

Metodologia

- Aulas teóricas para apresentação de conceitos
- Aulas práticas para experimentação com programação orientada a aspectos através de exemplos
  - Linguagem AspectJ
  - Ambiente Eclipse
- Exercício em pares

Software

- JDK 1.5
  - http://java.sun.com/j2se (1.5.0_01-b08)
- Eclipse
  - http://www.eclipse.org (3.1.2)
- AspectJ 1.5
  - http://www.eclipse.org/aspectj (1.5.1a)
- AJDT
  - http://www.eclipse.org/ajdt (1.3.1)
- http://sergio.dsc.upe.br/aspectos

Avaliação de alunos

- Avaliação com base em freqüência, participação e apresentação de trabalhos.
  - trabalho em grupo
    - —no máximo 4 integrantes por grupo

Normas

- Serão considerados aprovados os alunos que, em cada disciplina, obtiverem 75% de freqüência e média de proficiência igual ou superior a 7,0 (sete) nos trabalhos.
- Quando obtiver nota entre 5,0 (cinco) e 6,9 (seis vírgula nove), terá direito a nova avaliação, determinada pelo professor, na qual o aluno deverá obter média final igual ou superior a 7,0 (sete).

Referências Bibliográficas

- G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier and J. Irwin. "Aspect-Oriented Programming". June 1997.
- R. Filman and D. Friedman. Aspect-oriented programming is quantication and obliviousness. In OOPSLA 2000 Workshop on Advanced Separation of Concerns, Minneapolis, MN, Oct. 2000.
- Elrad, T.; Aksit, M.; Kiczales, G.; Lieberherr, K.; Ossher, H.: "Discussing Aspects of AOP". Communications of the ACM 44 (10), pp. 33 – 38, October 2001.
- C. Lopes. "Aspect-Oriented Programming: An Historical Perspective (What's in a Name?)". ISR Technical Report #UCI-ISR-02-5, University of California, Irvine, December 2002.

Referências Bibliográficas

- G. Kiczales et al. "An Overview of AspectJ". ECOOP'2001, Budapest, Hungary, 2001.
- Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. "Getting Started with AspectJ". Communication of the ACM. October 2001.
- Laddad, R., AspectJ in Action: Practical Aspect-Oriented Programming, Manning Publications Company, 2003. ISBN: 1930110936
- AspectJ Project. http://eclipse.org/aspectj

Referências Bibliográficas

- AspectJ Programming Guide
  - http://eclipse.org/aspectj/doc/released/progguide/index.html
- AOSD
  - http://aosd.net
- Este curso e outros materiais em
  - http://sergio.dsc.upe.br/aspectos

Introdução à Orientação a Aspectos

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

pesquisas e mercado oferecem novas alternativas

guia: como projetar software?

Paradigmas evoluem e se adaptam ao ser humano

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

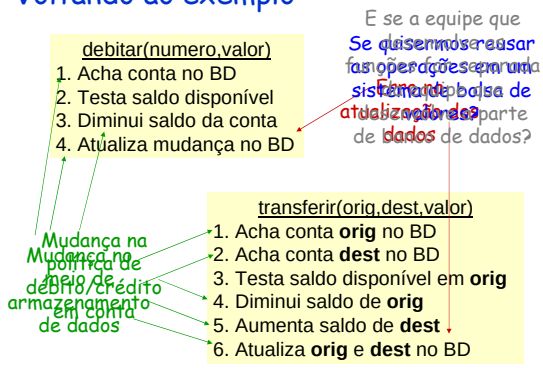
debitar(numero,valor)
1. Acha conta no BD
2. Testa saldo disponível
3. Diminui saldo da conta
4. Atualiza mudança no BD

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



Mas, observando o mundo real...

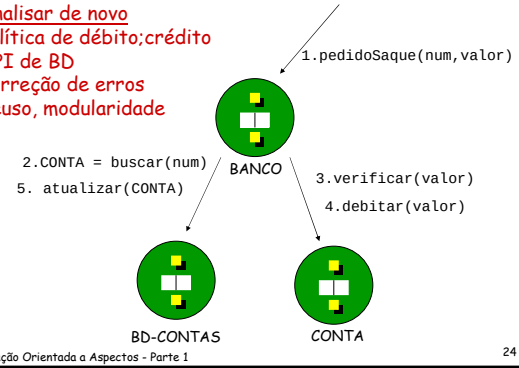
- Existe um banco, que define as atividades bancárias
- Existem contas no banco, que possuem políticas de operação
- Existe um banco de dados do banco, que guarda e recupera contas e clientes

Programa orientado a objetos

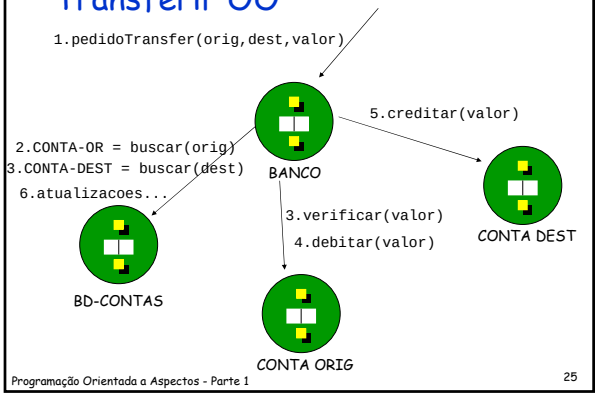
- Abstrações bem mais próximas do mundo do problema
  - objetos, não funções
- Em um programa, "tudo" é objeto
- Um programa é um monte de objetos dizendo aos outros o que fazer
  - mensagens

Debitar OO

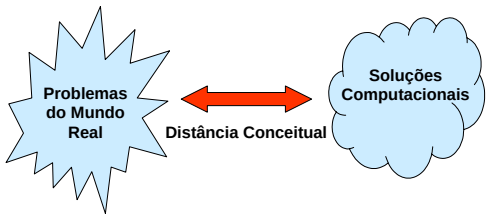
Analisar de novo
Política de débito;crédito
API de BD
Correção de erros
Reuso, modularidade



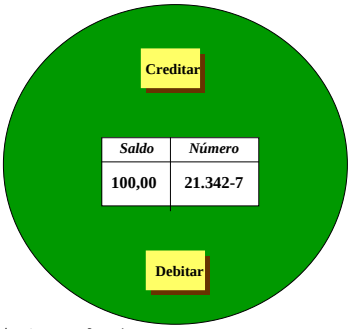
Transferir OO



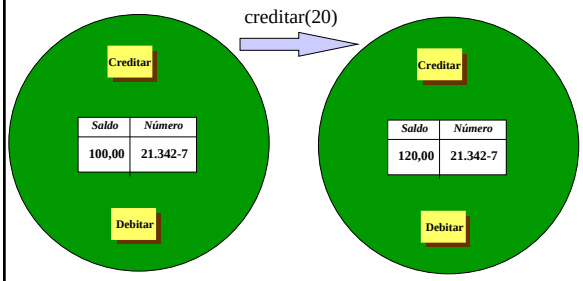
"A orientação a objetos tem por objetivo diminuir a distância conceitual entre o mundo real e o computacional."



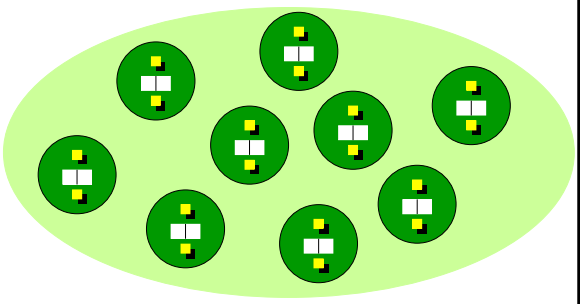
Objeto conta bancária



Estados do objeto conta



Classe de Contas Bancárias



objetos semelhantes moldados por uma classes

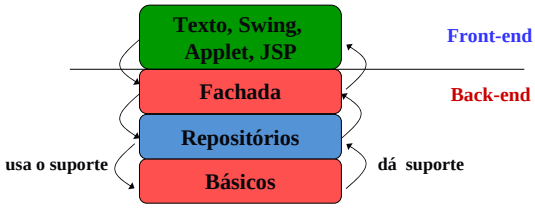
Definição classes - Java

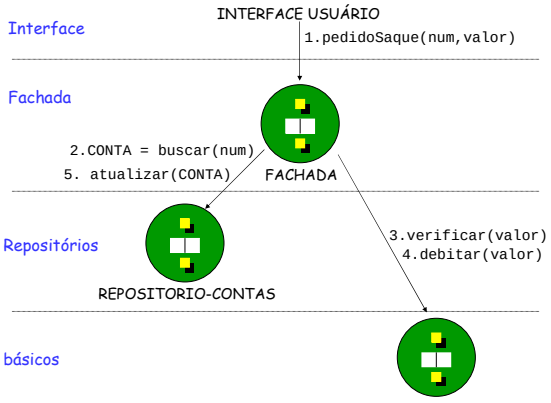
```
public class Conta {
  private String numero;
  private double saldo;

  ...
  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

Camadas de objetos





OO resolve nosso problema?

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

Conceito novo – CONCERN (interesse)

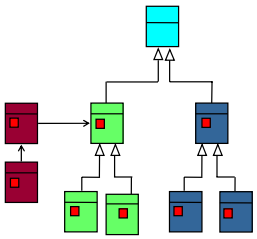
- Requisitos que devem ser implementados em um sistema
- Em qualquer sistema, vários interesses precisam ser implementados
  - sem eles implementados, seu sistema não atende aos requisitos

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

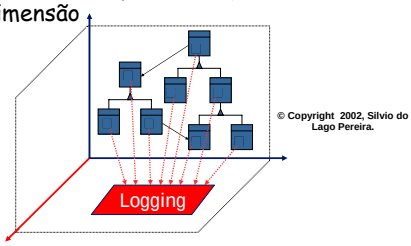
- Código entrelaçado (tangling)
  - código de *logging* é misturado com código de negócio
- Código espalhado (spread)
  - código de *logging* em várias classes
- *Logging* é um interesse transversal (crosscutting concern)

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

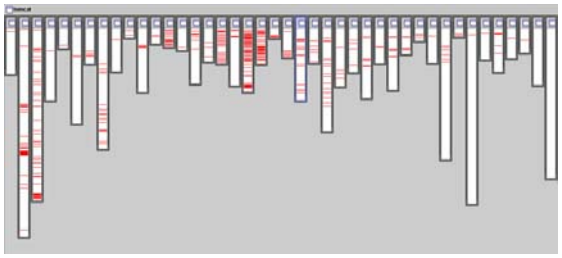
Causa do problema

- Implementação mapeia os requisitos em uma única dimensão



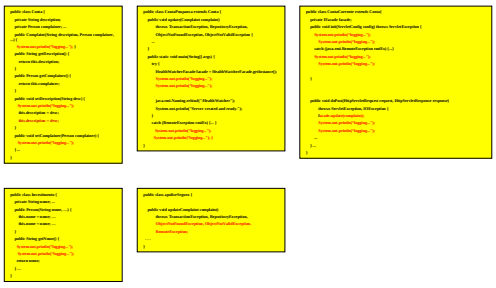
- Interesse é transversal à dimensão de implementação

Ex: Logging no Apache Tomcat



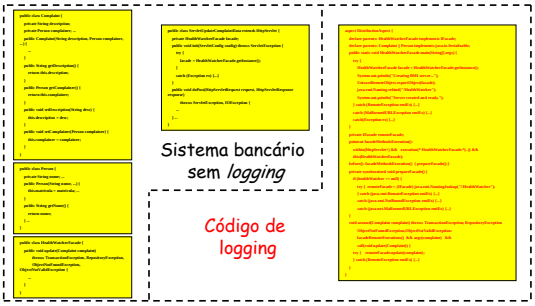
- Linhas vermelhas: logging
  - não estão no mesmo lugar
  - nem mesmo em poucos lugares

Sistema bancário com logging



Código de logging é vermelho...

O ideal seria se...



Como conseguir isso?

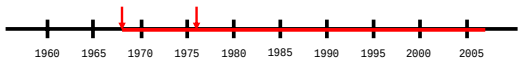
Programação orientada a aspectos (AOP)

- 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].*

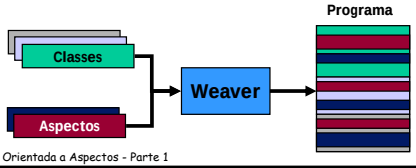


Implementação com AOP

- 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?

Advices (adendo)

- 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

Para terminar, onde colocar tudo isto?

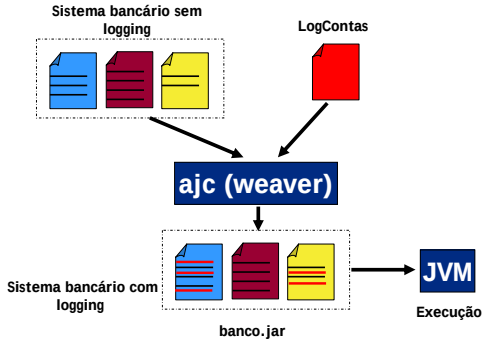
Aspectos

- 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 AspectJ



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
- Façam no notepad! ;-)

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");
  }
}
```


Solução

```
public class Conta {
  private String numero;
  private double saldo;
  ...
  public void setNumero(String numero) {
    this.numero = numero;
  }
  public void creditar(double valor) {
    this.saldo = this.saldo + valor;
  }
  public void debitar(double valor) {
    this.saldo = this.saldo - valor;
  }
```


Solução

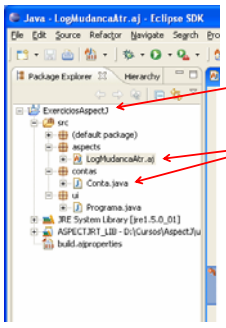
```
public class Programa {
     public static void main(String[] args)
{
    Conta c = new Conta("1", 10.0);
       c.creditar(10);
       c.debitar(2);
       System.out.println(c.getSaldo());
  }
}
```

O que será impresso no console?

Agora vamos para o ambiente de desenvolvimento

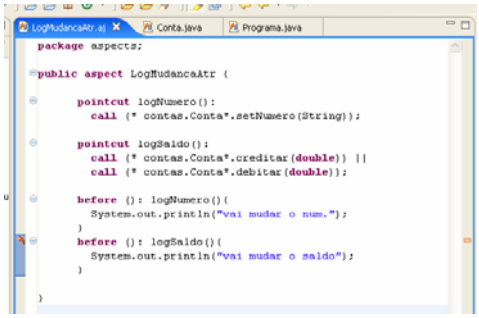
Abram o Eclipse e
executem o Roteiro 1
implementem o exercício anterior

Identificando partes do Eclipse

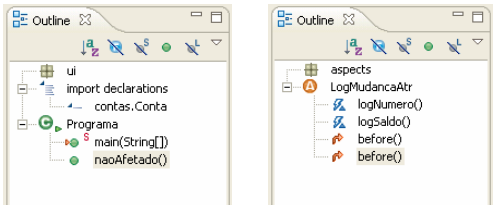


- Projeto
- Classes e aspectos

Editor de classes e aspectos



Outline view

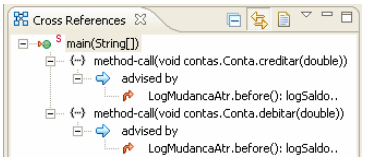


- Decora membros afetados por advice com uma seta laranja
- Mostra membros estruturais de aspectos

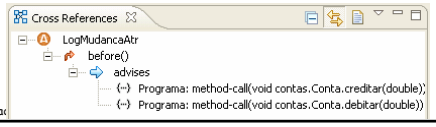
Cross References view

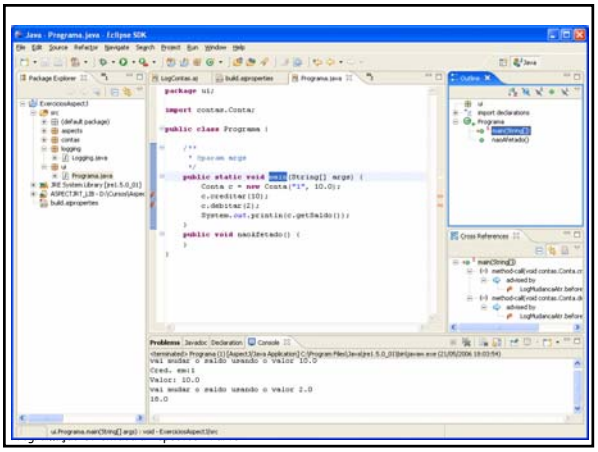
- Mostra relacionamentos de e para o membro selecionado
- Um clique duplo do mouse sobre uma das extremidades abre e mostra o arquivo fonte no editor

Classe



Aspecto



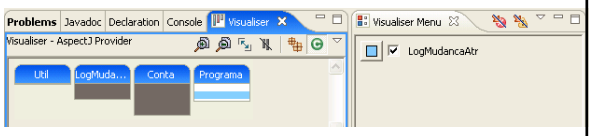


Visualização

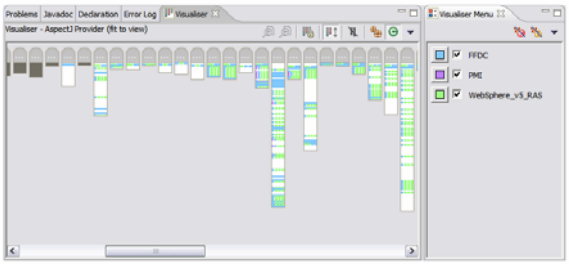
- No menu *Window* escolha a opção *Show view -> Other...*
- Em seguida na pasta *Visualiser* escolha as opções (segurando a tecla Shift) *Visualizer* e *Visualizer Menu*. Estas janelas mostram como os aspectos afetam as classes do projeto.

Impacto de Crosscutting

- Visualizador de aspectos: mostra o impacto de crosscutting dos aspectos de um projeto, pacote ou classe



Outro exemplo



Referências

- Gregor Kiczales et. al. Aspect-Oriented Programming. European Conference on Object-Oriented Programming, ECOOP'97
- http://groups.yahoo.com/group/asoc-br/
- http://www.aosd.net/
- http://www.eclipse.org/aspectj

Software Productivity Group
http://spg.dsc.upe.br
CIn-UFPE e DSC-UPE

Especialização em Engenharia de Software

Programação Orientada a Aspectos Parte 1

Sérgio Soares
DSC – UPE
sergio@dsc.upe.br

Departamento de Sistemas Computacionais

Especialização em Engenharia de Software

Programação Orientada a Aspectos Parte 2

Sérgio Soares
DSC – UPE
sergio@dsc.upe.br

Departamento de Sistemas Computacionais

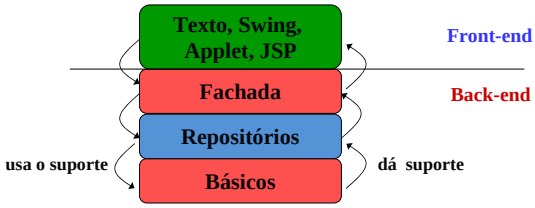
AspectJ

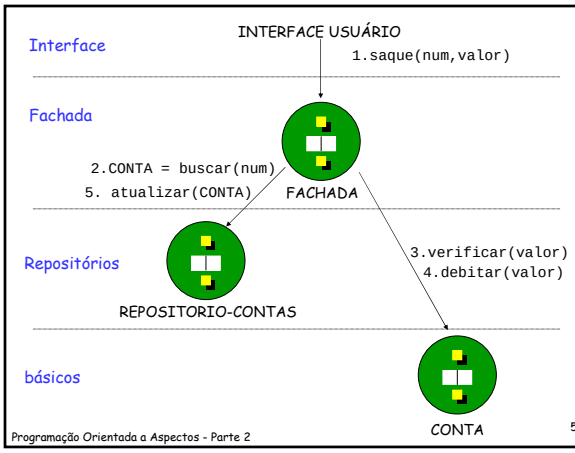
- Modelo de pontos de junção
- Pointcuts
- Advices
- Inter-type declarations
- Aspectos

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.

Sistema em camadas...de volta



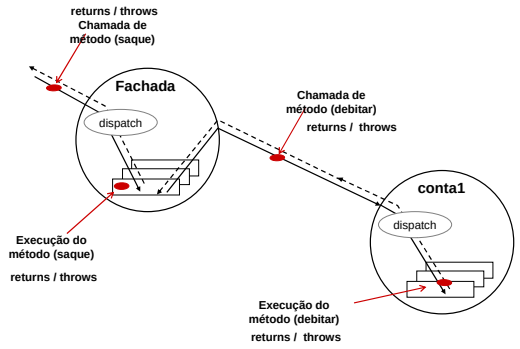



```
InterfaceGrafica(){ ...
//usa fachada
fach.saque("12",100.00);
...
```

```
public class Fachada { ...
  public void saque(String num, double valor){
    Conta c= repContas.buscar(num);
    c.debitar(valor);
    repContas.atualizar(c);
  }
}
```

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

Pontos de junção



pontos de acesso a atributos

```
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);
  }
  ...
  private void setSaldo(double saldo) {
      this.saldo = saldo;
  }

  public double getSaldo() {
      return saldo;
  }
```

escrita

leitura

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

Pointcuts (conjuntos 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

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)

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

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)
```


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

```
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

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

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

- podemos compor estes pointcuts

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

OU

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


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...

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


Mais designadores

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

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);
```


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

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

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

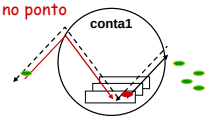
- `call` — `call(void Conta.creditar(..))`
- `execution` — `execution(void Conta.*(double))`
- `handler` — `handler(IOException)`
- `get, set` — `get(double Conta.saldo)`
- `within, withincode` — `within(contas.*)` `withincode(Programa.main(..))`
- `cflow, cflowbelow` — `cflow(call(Banco.saque(..)))`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

Advices

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

before

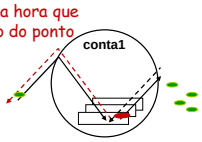
before executa antes de chegar no ponto




```
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

Mudando o exemplo de logging

```
public class Logging {
  public static final int CREDITO = 0;
  public static final int DEBITO  = 1;
  public void log(String n,double v,int op){
    switch(op){
      case(Logging.CREDITO):
        System.out.println("Cred. em:"+n);
        System.out.println("Valor: "+v); break;
      case(Logging.DEBITO):
        System.out.println("Deb. em:"+n);
        System.out.println("Valor:"+v); break;
      default: break;
}}}
```

Nova classe responsável pelo logging

Mudando o exemplo de logging

```
public class Conta {
  private String numero;
  private double saldo;
  private Logging logger = new Logging();
  ...
  public void debitar (double valor) {
    if (this.getSaldo() >= valor)
      this.setSaldo(this.getSaldo()-valor);
    this.logger.log(numero,valor,Logging.DEBITO);
  }
  public void creditar (double valor) {
    this.setSaldo(this.getSaldo()+valor);
    this.logger.log(numero,valor,Logging.CREDITO);
  }
}
```


Colocando logging em aspectos

A classe **Conta** ficaria assim...

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

teríamos que adicionar o atributo...como?

Inter-type declarations (introductions)

- Declara novos membros a classes
  - atributos
  - método
  - construtores
- Muda a hierarquia de tipos
  - declara que uma classe implementa novas interfaces
  - declara que uma classe estende de uma nova classe

Aspecto de Logging

```
public aspect LogContas {

  private Logging Conta.logger = new Logging();

  pointcut ...

  after (Conta c,double v): logCredito(c,v){
    c.logger.log(c.getNumero(),v,Logging.CREDITO);
  }

  after (Conta c,double v): logDebito(c,v){
    c.logger.log(c.getNumero(),v,Logging.DEBITO);
  }
}
```

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

Outro exemplo

```
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;

  }
  void setP2(Point p2) {
    this.p2 = p2;

  }
}

class Point {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;

  }
  void setY(int y) {
    this.y = y;

  }
}
```

- Sempre que um ponto ou uma linha forem modificados
  - atualizar o display
  - chamando método de atulização

Atualizando display

```
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
    Display.update();
  }
  void setP2(Point p2) {
    this.p2 = p2;
    Display.update();
  }
}

class Point {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
    Display.update();
  }
  void setY(int y) {
    this.y = y;
    Display.update();
  }
}
```

- Duplicação de código
  - difícil evolução
  - mudanças em várias classes

Atualizando display com AspectJ

```
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
  }
  void setP2(Point p2) {
    this.p2 = p2;
  }
}

class Point {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
  }
  void setY(int y) {
    this.y = y;
  }
}
```

```
aspect DisplayUpdating {

  pointcut move():
    call(void FigureElement.moveBy(int, int) ||
    call(void Line.setP1(Point))             ||
    call(void Line.setP2(Point))             ||
    call(void Point.setX(int))               ||
    call(void Point.setY(int));

  after() returning: move() {
    Display.update();
  }
}
```


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
}
```


Precedência entre aspectos

- Um aspecto pode declarar explicitamente uma relação de precedência entre aspectos concretos:
  `declare precedence: TypePatternList`

```
declare precedence: LoggingAtt, TrataExcecao;
```

Logging tem a maior precedência

Erros e warnings em tempo de compilação

- Um aspecto pode declarar erros e *warnings* a serem verificados em tempo de compilação associados a pontos de junção:
  `declare error`
  `declare warning`

```
declare warning :
    call(void Persistence.save(Object)) :
      "Consider using Persistence.saveOptimized()";
```

Acesso reflexivo a informações do ponto de junção

- Acesso a informação estática e dinâmica associada a pontos de junção
  - uso de reflexão
  - acesso a nomes de argumentos e nome de métodos afetados por advice
- O contexto dinâmico que poderia ser capturado é similar ao capturado por **`this`**, **`target`** e **`args`**

Acesso reflexivo a informações do ponto de junção

- AspectJ oferece acesso reflexivo através de 3 objetos especiais
  - **`thisJoinPoint`**
  - **`thisJoinPointStaticPart`**
  - **`thisEnclosingJoinPointStaticPart`**
- Estes objetos podem ser usados no corpo de qualquer advice

Acesso reflexivo a informações do ponto de junção

- Informação dinâmica
  - mudar a cada invocação distinta de um mesmo ponto de junção
    - objetos, argumentos
- Informação estática
  - não se modifica em tempo de execução
    - nome de métodos, tipos de parametros, métodos de uma classe

Acesso reflexivo a informações do ponto de junção

- Objeto que contém informação dinâmica no ponto de junção
  - **`thisJoinPoint`**
- Objetos que contêm informação estática sobre (a) o ponto de junção e (b) o contexto que envolve o ponto de junção
  - **`thisJoinPointStaticPart`**
  - **`thisEnclosingJoinPointStaticPart`**

`thisJoinPoint`

- Acesso reflexivo a informação dinâmica relativa ao ponto de junção:
  - **`this`**, **`target`** e **`args`**
- Acesso reflexivo a informação estática relativa ao ponto de junção
  - Uso do método **`getStaticPart`**
- Exemplo
  - Guardar valores de objeto corrente e argumentos através de aspecto de *logging*

`thisJoinPoint`

```
...
after(... {
  System.out.println(
      thisJoinPoint.getKind());          1
  System.out.println(
      thisJoinPoint.getSignature());     2
  System.out.println(
      thisJoinPoint.getTarget());        3
  System.out.println(
      thisJoinPoint.getThis());          4
}
```

```
1 method-execution
2 void contas.Conta.creditar(double)
3 contas.Conta@8813f2
4 contas.Conta@8813f2
```

Associação de aspectos

- Por default, apenas uma instância de aspecto existe para cada JVM
  - como um singleton
- Pode-se desejar uma instância de aspecto diferente associada a cada objeto, fluxo de controle, etc.

Instâncias de aspectos

- **`pertarget(<pointcut>)`**
  `perthis(<pointcut>)`
  - uma instância de aspecto associada a cada objeto que seja representado por **`target`** ou **`this`** nos pontos de junção descritos por <pointcut>
- **`percflow(<pointcut>)`**
  `percflowbelow(<pointcut>)`
  - uma instância de aspecto associada a cada ponto de junção em <pointcut>, está disponível para todos os pontos de junção de **`cflow`** ou **`cflowbelow`**

Exemplo: Aspectos de controle de concorrência

```
aspect ConcurrencyManagerAspect {
  private ConcurrencyManager manager;
  // uma instancia do aspecto é compartilhada por
  // todos os objeto "this", logo, o atributo é
  // acessado concorrentemente
}
```

```
aspect ConcurrencyManagerAspect
    perthis(synchronizationPoints(Object)) {
  private ConcurrencyManager manager;
  // uma instancia do aspecto é associada a cada
  // objeto "this", logo, há um objeto manager
  // para cada, não sendo acessado
  // concorrentemente
}
```


Referencias

- Programming guide
  - www.eclipse.org/aspectj/doc/released/progguide/index.html

Exercícios

Executar Roteiro 2

Especialização em Engenharia de Software

Programação Orientada a Aspectos Parte 2

Sérgio Soares
DSC – UPE
sergio@dsc.upe.br

Especialização em Engenharia de Software

Programação Orientada a Aspectos Parte 3

Sérgio Soares
DSC – UPE
sergio@dsc.upe.br

Departamento de Sistemas Computacionais

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

Aspectos de Desenvolvimento

- Verificação de Contratos
  - Pré-condições e Pós-condições
  - Garantia de Restrições
- Outros exemplos
  - *Logging*
  - *Profiling*

Aspectos de Desenvolvimento

- Verificação de Contratos [Eiffel/Meyer]
  - **pré-condições**
    - parâmetros são válidos?
  - **pós-condições**
    - valores foram atualizados adequadamente?
  - **garantia de restrições**
    - força parâmetros serem válidos
- Java não dá suporte a contratos
  - com AspectJ ...

Editor de Figuras

```
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
  }
  void setP2(Point p2) {
    this.p2 = p2;
  }
}

class Point {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
  }
  void setY(int y) {
    this.y = y;
  }
}
```

- Como garantir que os movimentos seguem os limites mínimo e máximo da tela?

Pré-Condições

...usando before

```
aspect PointBoundsChecking {
  before(int newX):
      call(void Point.setX(int)) && args(newX) {
    assert(newX >= MIN_X);
    assert(newX <= MAX_X);
  }
  before(int newY):
      call(void Point.setY(int)) && args(newY) {
    assert(newY >= MIN_Y);
    assert(newY <= MAX_Y);
  }
  private void assert(boolean v) {
    if (!v) throw new RuntimeException();
  }
}
```

Pós-Condições

...usando after

```
aspect PointBoundsChecking {
  after(Point p, int newX):
      call(void Point.setX(int)) &&
      target(p) && args(newX) {
    assert(p.getX() == newX);
  }
  after(Point p, int newY):
      call(void Point.setY(int)) &&
      target(p) && args(newY) {
    assert(p.getY() == newY);
  }
  private void assert(boolean v) {
    if (!v) throw new RuntimeException();
  }
}
```


Garantia de Restrições

...usando around

```
aspect PointBoundsChecking {
  void around(Point p, int newX):
      call(void Point.setX(int)) &&
      target(p) && args(newX) {
    proceed(p, clip(newX, MIN_X, MAX_X));
  }
  void around(Point p, int newY):
      call(void Point.setY(int)) &&
      target(p) && args(newY) {
    proceed(p, clip(newY, MIN_Y, MAX_Y));
  }
  private int clip(int val, int min, int max) {
    return Math.max(min, Math.min(max, val));
  }
}
```


Aspectos de Produção

- Distribuição
- Persistência
- Tratamento de Erros

Health Watcher

- Sistema de informação baseado na Web
  - Distribuição (Java RMI)
  - Persistência (BD relacional)

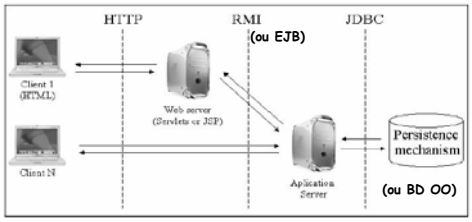
http://www.dsc.upe.br:8080/hw/

Distribuição e Persistência

- Sistema de informação baseado na Web
  - reclamações de saúde
  - melhorar serviços de vigilância sanitária
- Java → AspectJ

Distribuição e Persistência

- Configurações possíveis



(Java RMI)

Distribuição e Persistência

- Distribuição
  - acesso remoto de serviços usando Java RMI
- Persistência
  - comunicação com BDs relacionais
    - controle de conexões e transações
    - sincronização de estado de objetos com BDs (consistência)

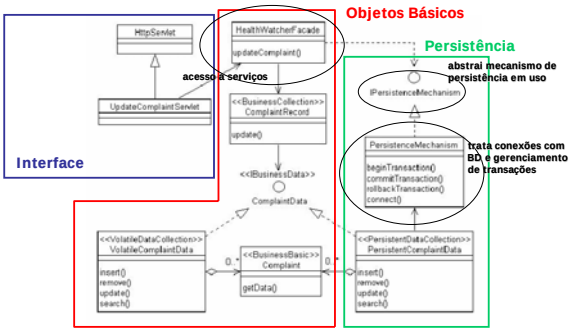
Distribuição e Persistência

- Arquitetura em Camadas
  - persistência (gerenciamento de dados)
  - objetos (*business*)
  - distribuição (comunicação)
  - apresentação (interface)
- Uso de padrões de projeto
- Melhorar separação de interesses

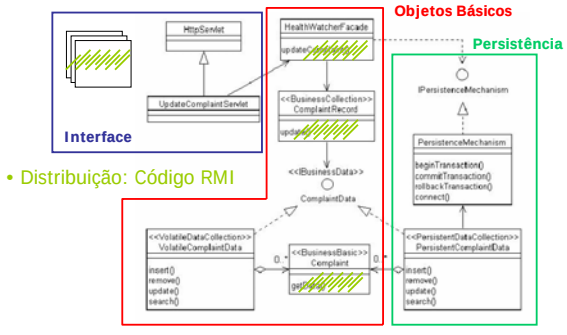
Distribuição e Persistência

- Alguns interesses transversais
  - código para iniciar e terminar transações
  - código de acesso a dados
  - código RMI
  - código especificando classes serializáveis para comunicação remota de seus objetos
  - código de transações se replica em todos métodos da classe **Fachada**

Distribuição e Persistência



Distribuição

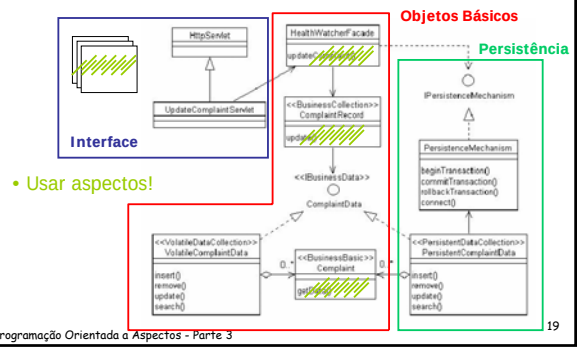


- Distribuição: Código RMI

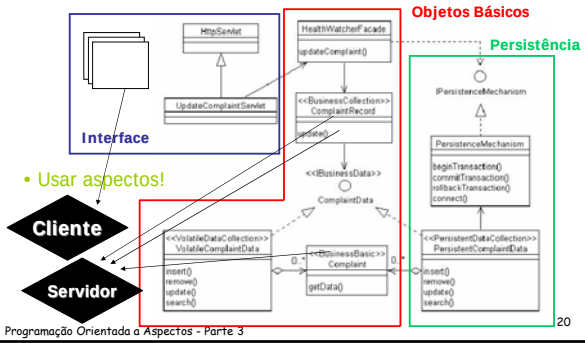
Distribuição

- Código RMI
  - Lado cliente
    - classes de interface do usuário
  - Lado servidor
    - classe "**HWFacade**"
    - argumentos e valores de retorno dos métodos da fachada que são executados remotamente

Distribuição

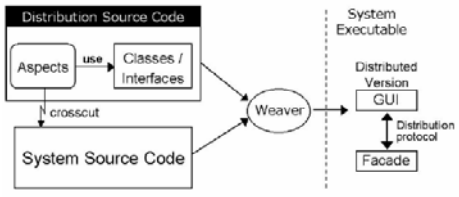


Distribuição



Distribuição

- Aspectos de Distribuição



Distribuição

- **Problema 1** (servidor): tornar a fachada remota
  - objetos devem implementar a interface **`Remote`**
    - classe **`HWFacade`** deve implementar **`Remote`**
  - todos métodos públicos devem incluir **`RemoteException`** na cláusula **`throws`**
  - classe **`HWFacade`** deve implementar **`UnicastRemoteObject`** que torna referências dos objetos remotamente disponíveis

Distribuição

- Aspecto Distribuição (lado servidor)
  - tornar remota a fachada

```
aspect ServerSideHWDistribution {
    declare parents: HWFacade implements
                     IHWFacade, UnicastRemoteObject;
```

- **`IHWFacade`**
  - tipo auxiliar aos aspectos
  - estende a interface **`Remote`**
  - contém as assinaturas dos métodos públicos da fachada
    - "**`throws RemoteException`**"

Distribuição

- Serializar tipos
  - parâmetros e valores de retorno

```
declare parents: healthGuide.HealthUnit || ...
                 complaint.Complaint ||
                 complaint.DiseaseType ||
                 complaint.Symptom implements
                 java.io.Serializable;
```

  - ferramentas de geração de código podem ajudar

Distribuição

- **Problema 2** (cliente): acesso transparente a nova fachada remota
- Todas classes cliente fazem referência a instância da fachada local
  - possuem um atributo **HWFacade** que armazena a instância local
  - devido a convenções do RMI, o tipo da referência remota é **IHWFacade**

Distribuição

- Aspecto Distribuição (lado cliente)
  - 1a. Opção: mudar o tipo HWFacade de todas as classes
    - mudança invasiva!
  - 2a. Opção: interceptar chamadas da fachada e substituir a referência local pela remota

```
pointcut facadeCalls(HWFacade f):
    target(f) && call(* *(..)) &&
    !call(static * *(..)) && this(HttpServlet);
Object around(HWFacade f) throws /*...*/:
    facadeCalls(f) {
  return proceed(remoteHW);
}
```

…entretanto, AspectJ requer que o tipo do retorno (IHWFacade) deve ser o mesmo da assinatura do advice (HWFacade)

Distribuição

- Solução: redirecionar chamadas de método
  - escrever um advice para cada método da fachada, por exemplo:

```
int around(Complaint c) throws /*...*/:
    facadeCalls() && args(c) &&
    call(int registerComplaint(Complaint)) {
  return remoteHW.registerComplaint(c);
} ...
```

  - problema de produtividade e manutenção!

Distribuição

- Solução alternativa usa API de reflection de Java e AspectJ

```
Object around(): facadeLocalCalls()  {
  Object[] args = thisJoinPoint.getArgs();
  Signature sig = thisJoinPoint.getSignature();
  String mName = signature.getName();
  return MExe.invoke(remoteFacade,mName,args);
}
```

- Falta repensar alternativas para Aspect

Aspectos reusáveis

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

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
  }
}
```

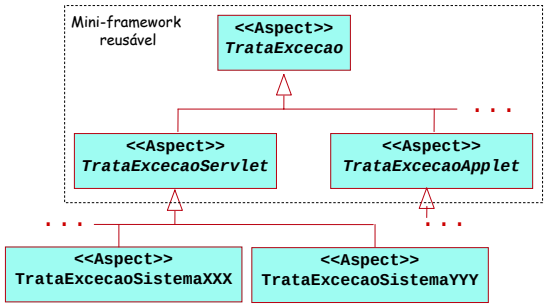
```
abstract aspect TrataExcecaoApplet
      extends TrataExcecao {
  void handling(Throwable e) {
     // trata exceção em applets
  }
}
```


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
}
```


Hierarquia de aspectos



Até aqui...

- Exemplos apresentados de aspectos no design
  - intuições para identificar aspectos
- Implementações em AspectJ
  - como o suporte da linguagem pode ajudar
- Quando os aspectos são apropriados?
  - existe um interesse que:
    - espalha-se através da estrutura de várias classes e/ou operações
    - é benéfico separá-los...

... Benefícios de separar

- A separação melhora o código realmente?
  - Separação de Interesses
    - pensar sobre os serviços do sistema sem distribuição
  - Descreve interações, reduz espalhamento
    - protocolo é completamente localizado nos aspectos de distribuição
  - Fácil de modificar/estender
    - mudar o protocolo de distribuição
  - "Plug and play"
    - o aspecto de distribuição pode ser facilmente 'desplugado' sem ser deletado

... Desvantagens de Aspectos

- "Inversões de dependência" podem dificultar compreensão
  - comportamento final de uma classe pode ser difícil de visualizar
- Fragmentação de código
- Cuidado!
  - Muitas vezes a separação afeta negativamente a qualidade do design (acoplamento, coesão, tamanho...)

AspectJ: pontos positivos

- modularidade, reuso, e extensibilidade
- inconsciência (o*bliviousness*)
- produtividade
- permite implementação e testes progressivos
  - plugabilidade

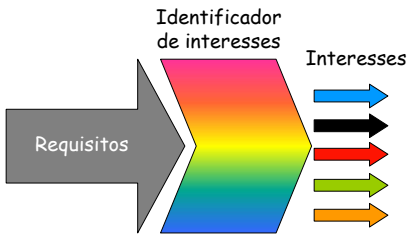
AspectJ: pontos negativos

- Novo paradigma
  - aprendizado
  - relação entre classes e aspectos deve ser minimizada
  - inconsciência (o*bliviousness*)
- Projeto da linguagem
  - tratamento de exceções
  - conflitos entre aspectos
- Requer suporte de ferramentas
- Combinação (apenas) estática

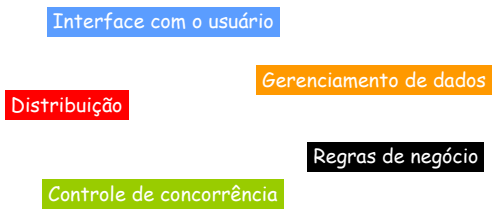
AO não é apenas P!

- 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)



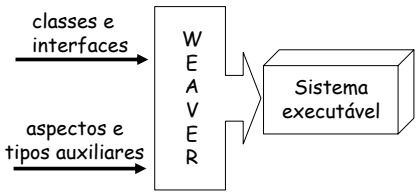
Interesses do Health Watcher



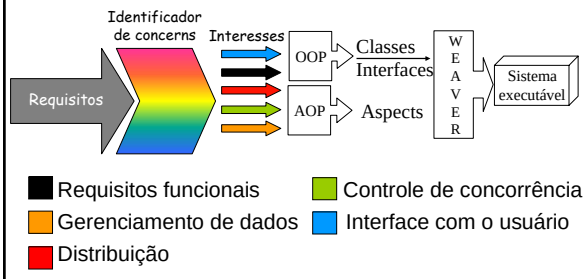
Passo 2: Implementar o sistema, separando os interesses

- 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

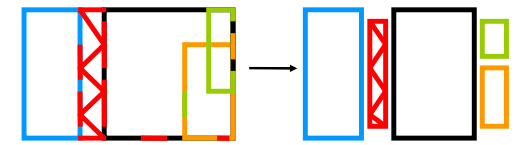
Passo 3: Recompor o sistema



Desenvolvimento de Software Orientado a Aspectos



OOP vs AOP



Requisitos funcionais
Controle de concorrência
Gerenciamento de dados
Interface com o usuário
Distribuição

AOSD: desafios

- projeto de linguagens
  - *crosscutting* mais dinâmico, sem depender de nomes
  - *weaving* dinâmico
  - evolução
- engenharia de software
  - extensões para UML, encontrar aspectos (componentes)
- teoria
  - compilação rápida, princípios de modularidade
- Veja mais em aosd.net

Exercícios

Continuem os exercícios do Roteiro 2

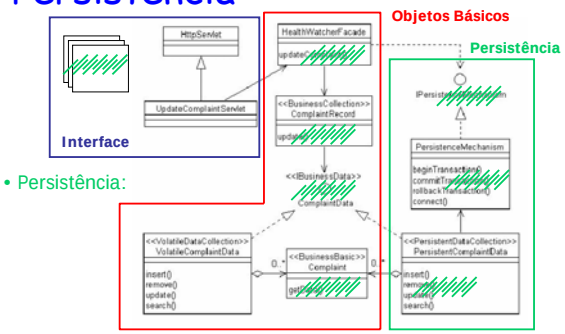
Extra

Outro exemplo de aspectos de produção

Persistência

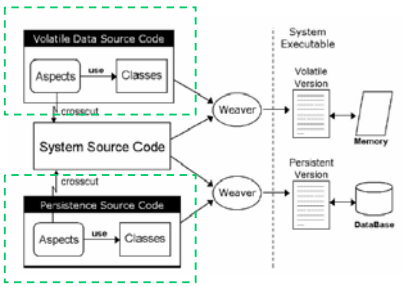
- Classes e interfaces para lidar com:
  - controle de conexões e transações
  - carregamento parcial de objetos
  - *caching* de objetos para melhora de performance
  - sincronização de estado de objetos com BDs (consistência)
- Uma generalização seriam aspectos de gerenciamento de dados
  - persistência em BDR, BDOO, arquivo e outros
  - armazenamento em memória
    - implementação progressiva (vide Parte 4 do curso)

Persistência



Persistência

- Aspecto Persistência



Persistência

Um aspecto reusável

```
abstract aspect AbstractPersistenceControl {
  private IPersistenceMechanism pm = null;
  abstract pointcut initSystem();
  abstract IPersistenceMechanism pmInit();
  public IPersistenceMechanism getPm() {
    if (pm == null) { pm = pmInit(); }
    return pm;
  }
  before(): initSystem() {
    getPm().connect();
  }
  after() throwing: initSystem() {
    getPm().disconnect();
  }
}
```


Persistência

- Aspecto concreto
  - no sistema Health Watcher (HW)...

```
aspect PersistenceControlHW
            extends AbstractPersistenceControl {
  pointcut initSystem(): call(HWFacade.new(..));
  IPersistenceMechanism pmInit() {
    IPersistenceMechanism pm;
    pm = PersistenceMechanismRDBMS.getInstance();
    return pm;
  }
}
```


Persistência

- Controle de Transações - Versão OO:
  - métodos explicitamente invocam métodos para controle de transações
  - código para tratar exceções
  - pelo menos, 6 linhas de código por método
  - ↓ manutenibilidade
  - ↓ reusabilidade

Controle de Transações – Versão OO

```
public int insert(Complaint c) throws ... {
  try {
    pm.beginTransaction();
    return complaintRecord.insert(complaint);
    pm.commitTransaction();
  } catch(...) {
    pm.rollbackTransaction();
  }
}
```


Persistência

Controle de Transações – Versão OA

```
abstract aspect AbstractTransactionControl {
  abstract pointcut transactionalMethods();
  before(): transactionalMethods() {
    getPm().beginTransaction();
  }
  after() returning: transactionalMethods()  {
    getPm().commitTransaction();
  }
  after() throwing: transactionalMethods()  {
    getPm().rollbackTransaction();
  }
  abstract IPersistenceMechanism getPm();
}
```

Outro aspecto reusável

Persistência

Controle de transações concreto

```
aspect TransactionControlHealthWatcher
            extends AbstractTransactionControl {
  declare parents: HealthWatcherFacade implements
                ITransactionalMethods;
  pointcut transactionalMethods():
         execution(* ITransactionalMethods.*(..));
  IPersistenceMechanism getPm() {
        return PersistenceControlHealthWatcher.
                              aspectOf().getPm();
  }
}
```


~~Persistência~~ → Sincronização de estado

- Sincronização de estado
  - atualizações nos objetos réplicas do cliente (distribuição)
  - atualizações nos objetos em memória no servidor
  - tais objetos não deveriam saber se são persistentes ou não
    - sincronização de estado é um interesse transversal (mesmo em relação a persistência)

Sincronização de estado 1

```
public aspect UpdateStateControl {

  private interface PersistentObject {
    public void aspectUpdating(int updateType);
  }

  declare parents: Complaint || HealthUnit || ...
                   implements PersistentObject;

  pointcut remoteUpdate(PersistentObject po):
    this(HttpServlet) && target(po) &&
    call(* set*(..));

  pointcut localUpdate(PersistentObject po):
    this(SystemRecord+) && target(po) &&
    call(* set*(..));
```


Sincronização de estado 2

```
after(PersistentObject po) returning:
    remoteUpdate(po) {
  po.aspectUpdating(REMOTE_UPDATE);
}

after(PersistentObject po) returning:
    localUpdate(po) {
  po.aspectUpdating(LOCAL_UPDATE);
}
```

Sincronização de estado 3

```
public void Complaint.aspectUpdating(int uType) {
  try {
    if (uType == REMOTE_UPDATE) {
      HealthWatcherFacade facade = ...
      facade.update(this);
    } else {
      ComplaintRecord record = ...
      record.update(this);
    }
  } catch (Exception e) {
    throw new org.aspectj.lang.SoftException(e);
  }
}
...
```


~~Exercício extra~~ PROJETO

- Baseado nos slides apresentados sobe distribuição, crie aspectos para distribuir o processamento entre a interface com o usuário FrameBanco e a classe Fachada
  - Dica: Usem RMI
  - http://java.sun.com/products/jdk/rmi/

Avaliação

- Enviar por email
  - Projeto
  - Respostas dos 4 exercícios
  - sergio@dsc.upe.br
- Grupos com no máximo 4 integrantes
- Entrega deve acontecer até a sexta-feira antes do próximo curso

Especialização em Engenharia de Software

Programação Orientada a Aspectos Parte 3

Sérgio Soares
DSC – UPE
sergio@dsc.upe.br

Departamento de Sistemas Computacionais

Especialização em Engenharia de Software

Programação Orientada a Aspectos Parte 4

Sérgio Soares
DSC – UPE
sergio@dsc.upe.br

Departamento de Sistemas Computacionais

Como desenvolver softwares orientados a aspecto?

- Até aqui olhamos apenas a implementação, e as demais atividades do ciclo de desenvolvimento?

Desenvolvimento de Software Orientado a Aspectos utilizando RUP e AspectJ

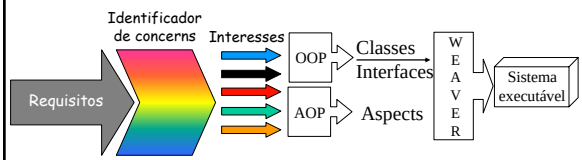
DSOA – Desenvolvimento de Software Orientado a Aspectos

- AspectJ implementa aspectos em Java . . .
- . . . mas como desenvolver sistemas orientados a aspectos?

Método de implementação OA

- Guias para o DSOA
- Complementar a técnicas de programação OO e padrões de projeto

DSOA

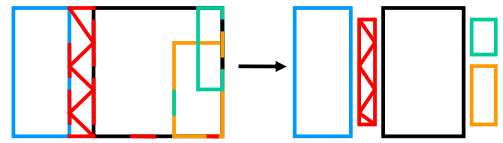


- UI e negócio usam OO
- Distribuição, gerenciamento de dados, e controle de concorrência usam OA

Derivando o método de implementação

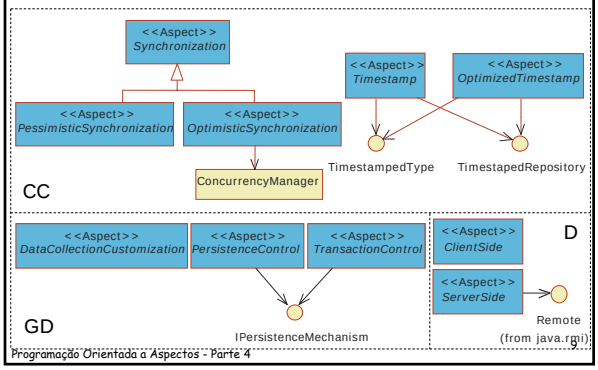
- Estudo de caso para reestruturar um programa OO em um programa OA
  - POA é útil
  - sugestão de melhorias em AspectJ
  - dependências e impactos entre os aspectos
    - distribuição vs. gerenciamento de dados
  - framework e padrões de aspectos para implementar os interesses

OOP vs AOP



Requisitos funcionais
Controle de concorrência
Gerenciamento de dados
Interface com o usuário
Distribuição

Framework de aspectos (AspectJ)



Método de implementação

- Definido para uma arquitetura de software específica
  - utilizada em vários sistemas de informação OO reais!
  - permire a definição de guias mais precisos
  - alguns tipos podem ser gerados automaticamente
    - ferramentas de suporte

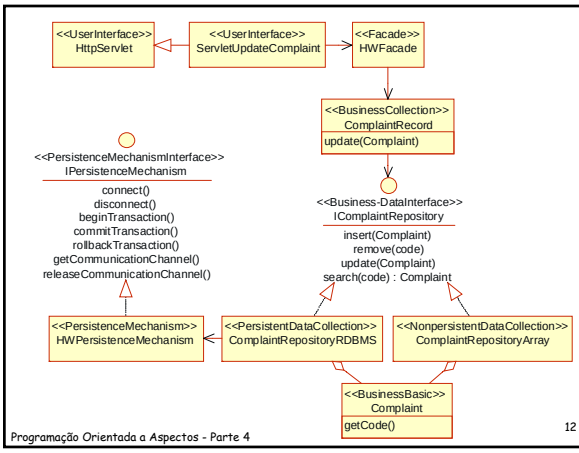
Arquitetura OO em camadas

Interfaçe com o usuário

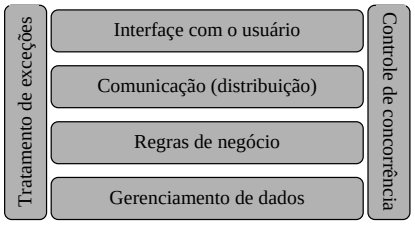
Comunicação (distribuição)

Regras de negócio

Gerenciamento de dados



Arquitetura OA em camadas



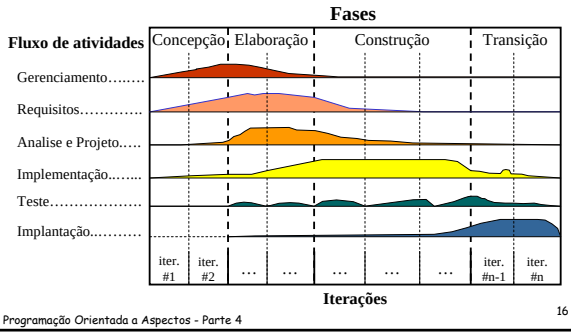
Método de implementação

- Como integrar com desenvolvimento guiado por casos de uso e com o RUP
  - impacto na dinâmica do processo
    - —uso de uma abordagem de implementação alternativa
  - impacto nas atividades do "processo"
    - —gerenciamento, requisitos, análise e projeto, implementação, e teste
    - —mudanças e definição de novas atividades

RUP

- Não é um processo!
  - mudanças no framework do processo permitindo refletir as mesmas nas suas instâncias

RUP



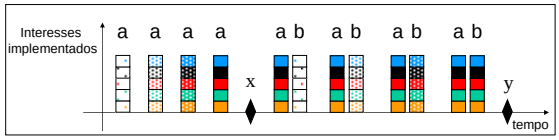
Uma abordagem alternativa de implementação

Implementação Progressiva

Implementação progressiva

- Abordagem alternativa
  - abstrai inicialmente parte dos requisitos não-funcionais
    - —ex.: distribuição, persistência, e controle de concorrência
  - permite validação antecipada de requisitos funcionais
    - —aumento em produtividade
  - diminui a complexidade de testes
    - —testes progressivos

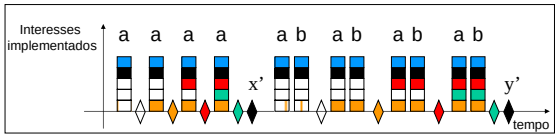
Abordagem regular



Interface com o usuário
Distribuição
Requisitos funcionais
Gerenciamento de dados persistente
Controle de concorrência
Milestone (fim de iteração)

a e b são casos de uso, conjuntos de casos de uso ou cenários

Abordagem progressiva



Interface com o usuário
Distribuição
Requisitos funcionais
Gerenciamento de dados persistente
Gerenciamento de dados não-persistente
Controle de concorrência
Milestone (fim de iteração)
Iteração functional

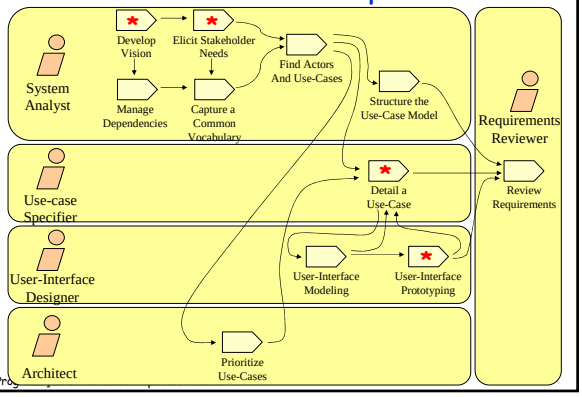
a e b são casos de uso, conjuntos de casos de uso ou cenários

Mudanças nas atividade do RUP

Gerenciamento

- No planejamento das iterações
  - considerar a abordagem progressiva
  - iterações funcionais e não-funcionais
- Análise de riscos
  - uso de um paradigma novo (DSOA)
  - uso de uma arquitetura de software específica

Atividades de requisito



Requisitos

- Desenvolver a visão / Elicitar necessidades dos stakeholders
  - a arquitetura de software a ser utilizada visa extensibilidade
    - —suporta separação de vários interesses
      - distribuição, controle de concorrência, gerênciamento de dados e tratamento de exceções
    - —possui suporte a geração automática de tipos

generalizar a arquitetura ...

Requisitos

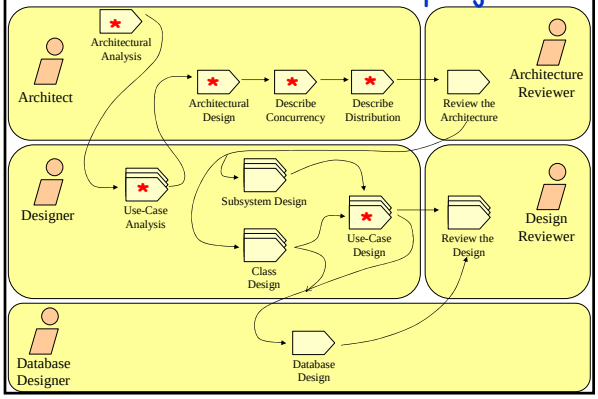
- Detalhar caso de uso
  - identificar aspectos candidatos
    - —requisitos não-funcionais
    - —e funcionais também?
    - —pesquisas na área
      - Early Aspects

http://www.early-aspects.net/

Requisitos

- Prototipar interface com o usuário
  - considerar o uso da abordagem de implementação progressiva
  - gerar um protótipo mais rapidamente e não o descartar

Atividades de análise e projeto



Análise e projeto

- Análise arquitetural
  - considerar o uso da arquitetura de software específica
- Análise de caso de uso
  - identificar tipos da arquitetura de software específica
  - classes de entidade -> classes básicas

Análise e projeto

- Projeto arquitetural
  - considerar o suporte oferecido pelo método
    - —persistência -> JDBC
    - —distribuição -> RMI
    - —controle de concorrência -> várias opções
  - classes de controle -> atributos da classe fachada
  - classes de fronteira -> interfaces com o usuário ou com subsistemas

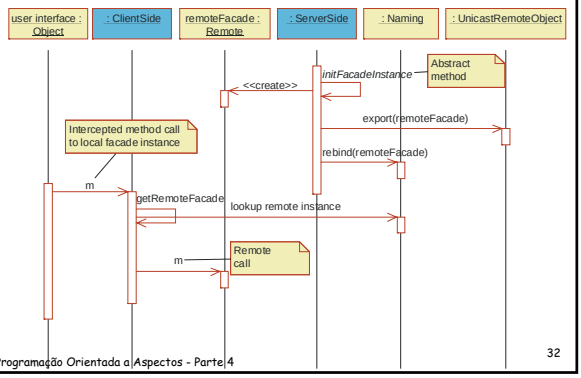
Análise e projeto

- Descrever concorrência
  - usar um método de controle de concorrência [Soares 2001]
- Descrever distribuição
  - o método suporta a distribuição entre a IU e a fachada
  - pequenas modificações no método para suportar a distribuição de outras partes

Análise e projeto

- Projetar caso de uso
  - diagramas de sequência com a iteração entre os objetos da arquitetura de software específica
  - o framework de aspectos já guia o impacto dos mesmos nos objetos do sistema
    - —diagramas mostrando a iteração dos aspectos pre-definidos [Soares 2004]

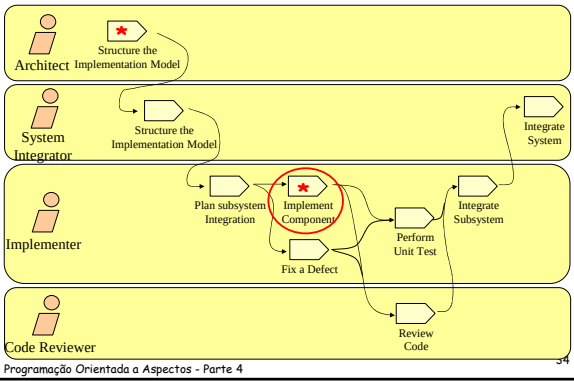
Diagrama de sequência – distribuição



Outras abordagens para APOA

- UML/Theme [Clarke03]
  - Siobhán Clarke, Elisa Baniassad . Aspect-Oriented Analysis and Design: The Theme Approach. Addison Wesley, 2005 .
- aSideML [Chavez04]
  - Christina Chavez, "Um Enfoque Baseado em Modelos para o Design Orientado a Aspectos", Tese de Doutorado, PUC-Rio, 2004.
- Em breve... Sérgio Soares ☺

Atividades de implementação



Implementação

- Estruturar o modelo de implementação
  - gerar classes da arquitetura
    - —ferramentas de modelagem geram as classes básicas
    - —ferramenta de suporte ao método gera coleções de negócio e de dados, interfaces negócio-dados e a classe fachada
  - aspectos relacionados aos interesses transversais também podem ser gerados
    - —progressiva vs. não-progressiva

Implementação

- Implementar componente
  - implementar a parte funcional e a interface com o usuário dos casos de uso selecionados
  - e . . .

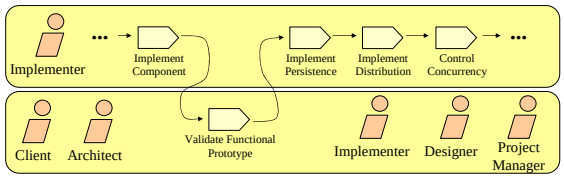
Implementação

- (versão não-progressiva)
  - gerar aspectos, específicos para o sistema em quetão, para lidar com os interesses transversais utilizando o framework de aspectos do método

Implementação

- (versão progressiva)
  - "desligar" persistência, distribuição e controle de concorrência
  - gerar coleções de dados não-persistentes e aspectos de não-persistência
  - e . . .

Novas atividades (abordagem progressiva)



Implementação

- Validar prototipo funcional
  - validar o protótipo com os stakeholders
  - anotar e executar mudanças de requisitos
  - validar os requisitos e suas eventuais mudanças

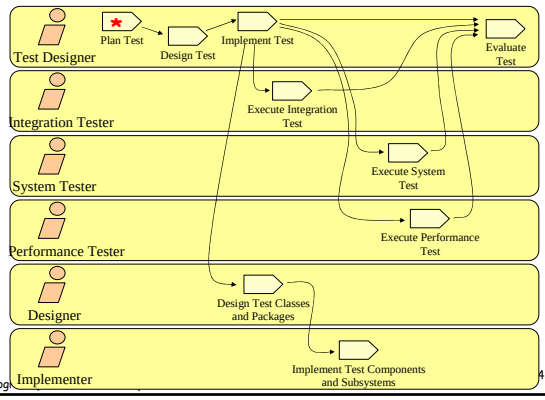
Implementação

- Implementar persistência
  - "desligar"
    - —aspectos de não-persistência
  - gerar / "ligar"
    - —aspectos de persistência
    - —coleções de dados persistentes
  - testar protótipo persistente

Implementação

- Implementar distribuição
  - gerar / "ligar" aspectos de distribuição
  - testar protótipo persistente e distribuído
- Controlar concorrência
  - gerar / "ligar" aspectos de controle de concorrência
    - —usando o método de controle de concorrência [Soares 2001]
  - testar protótipo com controle de concorrência

Atividades de teste



Teste

- Planejar teste
  - levar em conta a abordagem de implementação progressiva e o DSOA
  - testes progressivos
    - "ligar" e "desligar" os interesses transversais
    - testar várias combinações de interesses transversais
- Pesquisas na USP-São Carlos

Resumo

- Método de implementação
  - actividades, abordagem alternativa de implementação, guias
  - relacionado a um processo de desenvolvimento real
- Aspectos podem afetar outros
  - aspectos de distribuição afetam aspectos de gerenciamento de dados
  - analise e projeto são essenciais para identificar tais diferenças

Resumo

- Padrões e framework de aspectos
  - reuso, geração de aspectos dependentes da aplicação
- Suporte de ferramenta
  - aumento de produtividade
  - refactoring para aspectos
  - refactoring de aspectos

Referências

- Ivar Jacobson, Grady Booch, and James Rumbaugh. The Unified Software Development Process. Addison Wesley, 1999.
- Grady Booch, Ivar Jacobson, e James Rumbaugh. Unified Modeling Language - User's Guide, Addison-Wesley, 1999.

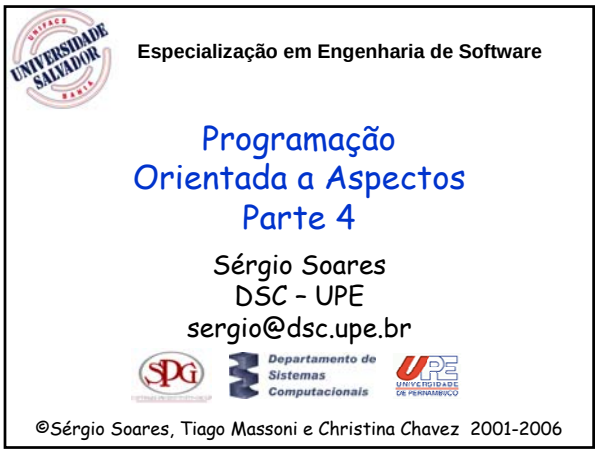
Referências

[Soares 2001] Sérgio Soares. Desenvolvimento Progressivo de Programas Concorrentes Orientados a Objetos. Dissertação de mestrado. Centro de Informática-UFPE, Fevereiro 2001.

[Soares 2004] Sérgio Soares. An Aspect-Oriented Implementation Method. Tese de Doutorado. Centro de Informática-UFPE, Outubro 2004.

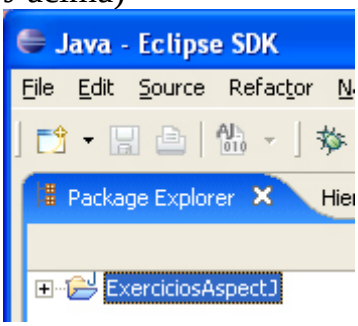
Software Productivity Group

http://spg.dsc.upe.br

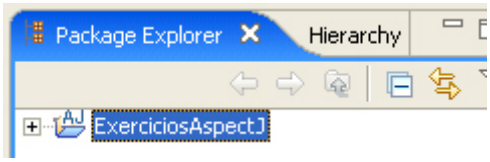

Especialização em Engenharia de Software
Programação
Orientada a Aspectos
Parte 4
Sérgio Soares
DSC – UPE
sergio@dsc.upe.br
Departamento de
Sistemas
Computacionais
©Sérgio Soares, Tiago Massoni e Christina Chavez 2001-2006

ROTEIRO 1 - Introdução ao ambiente de desenvolvimento Java - Eclipse

1. Dicas essenciais
   - Nomes de classes "devem" começar com letra maiúscula, para seguir o padrão de codificação de Java.
   - Nomes de métodos e variáveis "devem" começar com letra minúscula.
   - Caso o nome de métodos e variáveis seja uma palavra composta, o nome começa com letra minúscula e a primeira letra da proxima palavra é Maiúscula. Ex: `String numeroLido`, `int diaDaSemana`.
   - Não utilize caracteres especiais, como acentuação.
2. Crie um diretório chamado `aulaEclipse` e um subdiretório chamado `ExerciciosAspectJ` (ou outro nome qualquer) para armazenar o seu projeto.
3. Crie um projeto:
   - no menu *File* escolha a opção *New -> AspectJ Project*, um wizard iniciará
   - digite o nome do projeto (o mesmo do diretório criado, logo `ExerciciosAspectJ`) e defina o diretório do mesmo selecionando a opção *Create project from existing source* e em seguida usando o botão *Browse...*. Depois de selecionar o diretório pressione *Next >*.
   - Agora selecione o "ExerciciosAspectJ" em *Source folders on build path* escolha *Add Folder...*,
   - Informe *src* como o nome do diretório no qual o Eclipse colocará os arquivos fontes e clique em *OK*. Em seguida escolha a opção *Yes*.
   - Mude o caminho do *Default output folder* para "ExerciciosAspectJ/classes". Neste caminha serão colocados os arquivos .class gerados a partir da compilação dos arquivos fontes que estão em "ExerciciosAspectJ/src"
   - Clique em *Finish*
   - É possível que o Eclipse pergunte se deseja mudar para a pespectiva Java, uma vez que o projeto em questão é um projeto AspectJ. Selecione a caixa que permite que esta decisão seja lembrada e clique em *Yes*
   - Observe se o icone do projeto criado está como a figura abaixo (uma pasta com uma letra J acima)

     caso esteja clique com o botão direito do mouse sobre o projeto e escolha a opção *Convert to AspectJ Project*. O icone do projeto passará a ser a mesma pasta mas com as letras AJ acima.

4. Na janela *Package Explorer* você visualiza o seu projeto.
5. Criando uma classe:
   - no menu *File* escolha a opção *New -> Class*
   - informe o nome da classe (`Conta`) e o pacote (`contas`).
   - clique em *Finish*
   - o Eclipse criará o Arquivo *Conta.java* na pasta *src* e o abrirá para edição
   - Defina então na classe `Conta` os atributos `numero` (`String`) e `saldo` (`double`), ambos

privados.
   - Defina agora o construtor da classe `Conta` que inicializa o número da conta com um valor recebido como prarâmetro e o saldo com zero. (dica: clique com o botão direito na classe e escolha *Source* e em seguida *Generate constructor using fields...*, marque ambos os atributos e clique em OK)
   - Em seguida defina os métodos creditar, debitar, getNumero, e getSaldo (dica: clicando com o botão direito na classe e escolhendo *Source* existe uma opção *Generate Getters and Setters...*, marque os métodos que deseja gerar (não gere o setSaldo) e clique em OK)
   - Agora crie a classe `Programa` da mesma forma que foi criada `Conta`, mas depois de fornecer o nome da classe selecione a opção para gerar o método `main` antes de clicar em *Finish*
   - escreva no método `main` definições e comandos para manipular objetos do tipo `Conta` testando os métodos definidos anteriormente.
   - No Eclipse a compilação é incremental, isto é, a medida que você escreve código e salva o arquivo o Eclipse compila o código e já mostra eventuais erros de compilação.
6. Executando o programa:
   - Com o editor mostrando a classe `Programa` escolha no menu *Run* a opção *Run as* e em seguida *Java Application*
   - note que a execução ocorreu na pasta *Console* abaixo do editor de texto.
   - a partir de agora para executar o programa basta clicar no ícone de execução (um botão verde com o símbolo de *play* branco, abaixo do menu *Navigate*).
7. Dicas de escrita de código:
   - Digite um comando (ou parte dele) e pressione *Ctrl + barra de espaço*. O Eclipse mostratá opções de escrita de código para o comando, se existir.
   - Caso o código que vc está escrevendo tenha algum erro de sintaxe, o Eclipse sublinhará o mesmo como o word faz com palavras escritas erradamente. Coloque o mouse sobre a palavra e o Eclipse indicará o erro, e até mesmo dará sugestões de como corrigí-lo (selecione o erro e pressione *Ctrl + 1*).
8. Como ler informações do teclado
   - Baixe a classe `Util.java` e salve a mesma no diretório *src* do seu projeto.
   - Para que o Eclipse encontre a nova classe, clique no Eclipse sobre o diretório *src* com o botão direito do mouse e escolha a opção *Refresh* ou clique em *F5*
   - Utilize os comandos `Util.readStr()`, `Util.readInt()` e `Util.readDbl()` para ler valores `String`, `int` e `double`, respectivamente.
     Ex.:
     ```
     String numero = Util.readStr();
     int dia = Util.readInt();
     double valor = Util.readDbl();
     ```
     Há ainda o comando `Util.waitEnter()` que espera o usuário dar um enter para seguir a execução do programa.

ROTEIRO 2

Exercício 1

1. Crie um diretório chamado `ProjetoAspecto` para armazenar o seu projeto.
2. Copie os arquivos do exercício (pacotes `fachada`, `contas`, `gui`, e `util`) num subdiretório `src` dentro do diretório `ProjetoAspecto`. Outra opção para baixar os arquivos.
3. Crie um novo projeto AspectJ no diretório criado (vide Roteiro 1):
4. Agora vamos criar um aspecto
   - no menu *File* escolha a opção *New -> Other...*, selecione *Aspect* e em seguida clique em *Next*
   - escolha o *Source Folder* (o mesmo do projeto)
   - digite o nome do pacote (`aspectos`)
   - e o nome do aspecto (`Teste`)
5. Defina no aspecto um `pointcut` para identificar todas as chamadas ao método `creditar` de objetos do tipo `Conta`.
6. Agora defina um *advice* para imprimir a mensagem "Vou creditar" antes das chamadas ao método `creditar` de objetos do tipo `Conta`.
7. Abra o aquivo `build.ajproperties` que foi criado automaticamente no seu projeto. Este arquivo é responsável por informar que aspectos devem ser compostos com o sistema quando o weaver de AspectJ executar. Certifique-se do(s) aspecto(s) que está(ão) listado(s) no arquivo de configuração e que o mesmo está salvo.
8. Agora para gerar a versão do sistema composta pelo aspecto selecione o icone do weaver de AspectJ (o quarto ícone abaixo dos menus *Navigate* e *Search*) que comporá os aspectos ao sistema Java. Em seguida execute o programa e veja se o comportamento do mesmo foi alterado como desejado. Na verdade assim que o arquivo `build.ajproperties` for salvo (Ctrl + s) o Eclipse chama automaticamente o weaver de AspectJ.
9. Para gerar a versão original do programa, sem o efeito do aspecto, abra o arquivo de configuração e desmarque a seleção no pacote aspectos. Em seguida gere um build do sistema e execute-o novamente, observando que o aspecto não mais afeta o sistema.
10. No menu *Window* escolha a opção *Show view -> Other...* Em seguida na pasta *Visualiser* escolha as opções (segurando a tecla Ctrl) *Aspect Visualiser* e *Aspect Visualiser Menu*. Estas janelas mostram como os aspectos afetam as classes do projeto.
    - na janela *Aspect Visualiser Menu* você observa os aspectos que estão afetando o seu projeto e a cor que o representa
    - na janela *Aspect Menu* você observa, através das cores de cada aspecto, que aspectos afetam que classes. Quando esta janela estiver ativa você pode selecionar uma classe ou pacote através do *Package Explorer* para que seja mostrado que classes são afetadas.
      - nas classes que forem afetadas você pode clicar duas vezes com o mouse sobre a linha colorida para que o Eclipse mostra que método é afetado pelo aspecto representado pela cor.
11. Ainda com o projeto criado no primeiro roteiro crie outro aspecto (`Teste2`).
    - defina um `pointcut` para identificar todas as execuções do método `creditar` de um objeto do tipo `Conta` expondo o objeto do qual o método vai ser executado.
    - defina um *advice* para imprimir o saldo da conta após a execução do método creditar.
    - abra o arquivo de configuração, deselecione o aspecto `Teste`, selecione o novo aspecto, e peça para gerar um novo Build. Teste o sistema novamente.
    - o que deve acontecer caso ambos os aspectos forem compostos com o sistema? Modifique o arquivo `build` e faça o teste.

Exercício 2

1. Usando o mesmo projeto criado no exercício anterior vamos definir um exemplo que usa uma hierarquia de aspectos, favorecendo o reuso de aspectos.
   - crie o aspecto abstrato `AbstractLogging` e defina no mesmo:
     - um `pointcut` abstrato `loggedMethods`
     - dois *advice* que imprimem o valor da variável `thisJoinPoint` (contem informação sobre o *join point* corrente) antes e depois do pointcut `loggedMethods`
     - note que este aspecto define apenas qual o comportamento associado a logar *join points*, sem defini-los, o que deverá ser feito por um *subaspecto*
   - crie agora o aspecto `BancoLogging` que herda de `AbstractLogging` e define o `pointcut loggedMethods` identificando chamadas a métodos de classes dos pacotes `contas`, `fachada`, e `gui`.
     - note que este aspecto apenas define que join points devem ser logados
   - com essa estrutura podemos definir outros *subaspectos* abstratos de `AbstractLogging` para logar outros join points de outro sistema.
2. Para verificar de maneira mais fácil que classes um aspecto afeta selecione o mesmo, vá na janela *Outline* e clique nos símbolos + de modo a verificar que métodos de que classes o aspecto afeta.
3. Note que o caminho contrário também pode ser feito, caso selecionemos uma classe e na janela *Outline* cliquemos no + para descobrir que aspecto afeta que métodos da classe.

Exercício 3

1. Ainda utilizando o projeto de aplicação bancária, vamos escrever um aspecto para checar propriedades sobre a nossa aplicação.
   - defina um aspecto chamado `ValidaSistema` que deve verificar se algum atributo privado é alterado fora do método `set` para o mesmo. Caso positivo um *warning* deve ser informado em tempo de compilação.
   - adicione o aspecto na lista de arquivos, gere a nova versão do sistema e observe se em algum ponto da aplicação a propriedade é violada.
2. Provavelmente um dos pontos do sistema em que a propriedade é violada é no construtor das classes. Como o padrão de codificação sugere este tipo de programação, altere o aspecto definido para desconsiderar atribuições a atributos privados dentro do construtor da classe. Gere a aplicação e observe os *warnings*.
3. Mais uma vez refina o aspecto para não levar em conta métodos da interface negócio-dados (`RepositorioContas`), uma vez que suas implementações (`RepositorioContasArray` e `RepositorioContasLista`) manipulam os atributos nos seus métodos. Gere novamente a aplicação. Observe que apenas métodos da classe `Conta` ferem a propriedade. Modifique-os para alterar o atributo saldo através de um método `set`, porém privado. Gere a aplicação e observe que não há mais *warnings*.
4. Agora vamos definir um aspecto para checar uma pre-condição.
   - defina um aspecto para levantar a exceção `IllegalArgumentException` caso algum método seja chamado tendo uma referência não válida (`null`) em qualquer parte da aplicação.
   - comente a linha 10 da classe Banco, adicione o aspecto na lista de arquivos, gere e execute uma nova versão do sistema.
   - observe a quantidade de classes que o seu aspecto afeta.
5. Outra precondição que pode ser checada é se após a realização de um crédito o valor creditado foi somado ao saldo.

Exercício 4

1. Mais uma vez usando o projeto de aplicação bancária defina o aspecto `TrocaOrdemArgumentos` que troca a ordem dos números das contas na chamada do método `transferir`. Use o *advice* `around`. Gere e execute a aplicação usando o aspecto definido.
2. Agora defina o aspecto `TransacaoTransferencia`. Este aspecto deve imprimir uma mensagem após a execução **com sucesso** do método `creditar` e do método `debitar` de uma conta, mas somente se os mesmos forem chamados devido a execução de uma transferência. Use o *designator* `cflow`. Gere e execute a aplicação usando o aspecto definido.
3. No restante da aula crie aspectos de modo a fazer testes com outras construções de AspectJ. Em resumo: brinque com o Eclipse e com AspectJ ;-)

AspectJ Quick Reference

Aspects *at top-level (or* **static** *in types)*

aspect *A* { … }
defines the aspect *A*
privileged aspect *A* { … }
A can access private fields and methods
aspect *A* **extends** *B* **implements** *I, J* { … }
B is a class or abstract aspect, *I* and *J* are interfaces
aspect *A* **percflow**(*call(void Foo.m())*) { … }
an instance of *A* is instantiated for every control flow through calls to *m()*

general form:
[**privileged**] [*Modifiers*] **aspect** *Id*
[**extends** *Type*] [**implements** *TypeList*] [*PerClause*]
{ *Body* }
where *PerClause* is one of
pertarget (*Pointcut*)
perthis (*Pointcut*)
percflow (*Pointcut*)
percflowbelow (*Pointcut*)
issingleton ()

Pointcut definitions *in types*

private pointcut *pc*() : *call(void Foo.m())* ;
a pointcut visible only from the defining type
pointcut *pc*(*int i*) : *set(int Foo.x) && args(i)* ;
a package-visible pointcut that exposes an *int*.
public abstract pointcut *pc*() ;
an abstract pointcut that can be referred to from anywhere.
abstract pointcut *pc*(*Object o*) ;
an abstract pointcut visible from the defining package. Any pointcut that implements this must expose an *Object*.

general form:
abstract [*Modifiers*] **pointcut** *Id* (*Formals*) ;
[*Modifiers*] **pointcut** *Id* (*Formals*) : *Pointcut* ;

Advice declarations *in aspects*

before () : *get(int Foo.y)* { ... }
runs before reading the field *int Foo.y*
after () **returning** : *call(int Foo.m(int))* { ... }
runs after calls to *int Foo.m(int)* that return normally
after () **returning** (*int x*) : *call(int Foo.m(int))* { ... }
same, but the return value is named *x* in the body
after () **throwing** : *call(int Foo.m(int))* { ... }
runs after calls to *m* that exit abruptly by throwing an exception
after () **throwing** (*NotFoundException e*) : *call(int Foo.m(int))* { ... }
runs after calls to *m* that exit abruptly by throwing a *NotFoundException*. The exception is named *e* in the body
after () : *call(int Foo.m(int))* { ... }
runs after calls to *m* regardless of how they exit
before(*int i*) : *set(int Foo.x) && args(i)* { ... }
runs before field assignment to *int Foo.x*. The value to be assigned is named *i* in the body
before(*Object o*) : *set(* Foo.*) && args(o)* { ... }
runs before field assignment to any field of *Foo*. The value to be assigned is converted to an object type (*int* to *Integer,* for example) and named *o* in the body
int **around** () : *call(int Foo.m(int))* { ... }
runs instead of calls to *int Foo.m(int)*, and returns an *int*. In the body, continue the call by using **proceed**(), which has the same signature as the around advice.
int **around** () **throws** *IOException* : *call(int Foo.m(int))* { ... }
same, but the body is allowed to throw *IOException*
Object **around** () : *call(int Foo.m(int))* { ... }
same, but the value of **proceed**() is converted to an *Integer*, and the body should also return an *Integer* which will be converted into an *int*

general form:
[**strictfp**] *AdviceSpec* [**throws** *TypeList*] : *Pointcut* { *Body* }
where *AdviceSpec* is one of
before (*Formals*)
after (*Formals*)
after (*Formals*) **returning** [(*Formal*)]
after (*Formals*) **throwing** [(*Formal*)]
Type **around** (*Formals*)

Special forms *in advice*

thisJoinPoint
reflective information about the join point.
thisJoinPointStaticPart
the equivalent of **thisJoinPoint.getStaticPart()**, but may use fewer resources.
thisEnclosingJoinPointStaticPart
the static part of the join point enclosing this one.

proceed (*Arguments*)
only available in **around** advice. The *Arguments* must be the same number and type as the parameters of the advice.

Inter-type Member Declarations *in aspects*

int Foo . m (*int i*) { ... }
a method *int m(int)* owned by *Foo*, visible anywhere in the defining package. In the body, **this** refers to the instance of *Foo*, not the aspect.
private *int Foo . m* (*int i*) **throws** *IOException* { ... }
a method *int m(int)* that is declared to throw *IOException,* only visible in the defining aspect. In the body, **this** refers to the instance of *Foo*, not the aspect.
abstract *int Foo . m* (*int i*) ;
an abstract method *int m(int)* owned by *Foo*
Point . **new** (*int x, int y*) { ... }
a constructor owned by *Point*. In the body, **this** refers to the new *Point*, not the aspect.
private static *int Point . x* ;
a static *int* field named *x* owned by *Point* and visible only in the declaring aspect
private *int Point . x = foo()* ;
a non-static field initialized to the result of calling *foo()*. In the initializer, **this** refers to the instance of *Foo*, not the aspect.

general form:
[*Modifiers*] *Type Type . Id* (*Formals*)
[**throws** *TypeList*] { *Body* }
abstract [*Modifiers*] *Type Type . Id* (*Formals*)
[**throws** *TypeList*] ;
[*Modifiers*] *Type* . new (*Formals*)
[**throws** *TypeList*] { *Body* }
[*Modifiers*] *Type Type . Id* [= *Expression*] ;

Other Inter-type Declarations *in aspects*

declare parents : *C* **extends** *D*;
declares that the superclass of *C* is *D*. This is only legal if *D* is declared to extend the original superclass of *C*.
declare parents : C **implements** *I, J* ;
C implements *I* and *J*
declare warning : *set(* Point.*) && !within(Point)* : *"bad set"* ;
the compiler warns *"bad set"* if it finds a set to any field of *Point* outside of the code for *Point*
declare error : *call(Singleton.new(..))* : *"bad construction"* ;
the compiler signals an error *"bad construction"* if it finds a call to any constructor of *Singleton*
declare soft : *IOException* : *execution(Foo.new(..))*;
any IOException thrown from executions of the constructors of *Foo* are wrapped in **org.aspectj.SoftException**
declare precedence : *Security, Logging, ** ;
at each join point, advice from *Security* has precedence over advice from *Logging*, which has precedence over other advice.

general form
declare parents : *TypePat* **extends** *Type* ;
declare parents : *TypePat* **implements** *TypeList* ;
declare warning : *Pointcut* : *String* ;
declare error : *Pointcut* : *String* ;
declare soft : *Type* : *Pointcut* ;
declare precedence : *TypePatList* ;

Primitive Pointcuts

call (*void Foo.m(int)*)
a call to the method *void Foo.m(int)*
call (*Foo.new(..)*)
a call to any constructor of *Foo*
execution (* *Foo.*(..) throws IOException*)
the execution of any method of *Foo* that is declared to throw *IOException*
execution (*!public Foo .new(..)*)
the execution of any non-public constructor of *Foo*
initialization (*Foo.new(int)*)
the initialization of any *Foo* object that is started with the constructor *Foo(int)*
preinitialization (*Foo.new(int)*)
the pre-initialization (before the **super** constructor is called) that is started with the constructor *Foo(int)*
staticinitialization(*Foo*)
when the type *Foo* is initialized, after loading
get (*int Point.x*)
when *int Point.x* is read
set (*!private * Point.**)
when any non-private field of *Point* is assigned
handler (*IOException+*)
when an *IOException* or its subtype is handled with a catch block
adviceexecution()
the execution of all advice bodies
within (*com.bigboxco.**)
any join point where the associated code is defined in the package *com.bigboxco*
withincode (*void Figure.move()*)
any join point where the associated code is defined in the method *void Figure.move()*
withincode (*com.bigboxco.*.new(..)*)
any join point where the associated code is defined in any constructor in the package *com.bigoxco.*
cflow (*call(void Figure.move())*)
any join point in the control flow of each call to *void Figure.move()*. This includes the call itself.
cflowbelow (*call(void Figure.move())*)
any join point below the control flow of each call to *void Figure.move()*. This does not include the call.
if (*Tracing.isEnabled()*)
any join point where *Tracing.isEnabled()* is **true**. The boolean expression used can only access static members, variables bound in the same pointcut, and **thisJoinPoint** forms.

this (*Point*)
any join point where the currently executing object is an instance of *Point*
target (*java.io.InputPort*)
any join point where the target object is an instance of *java.io.InputPort*
args (*java.io.InputPort, int*)
any join point where there are two arguments, the first an instance of *java.io.InputPort,* and the second an *int*
args (*, int*)
any join point where there are two arguments, the second of which is an *int.*
args (*short, .., short)*
any join point with at least two arguments, the first and last of which are *short*s
Note: any position in **this**, **target**, and **args** can be replaced with a variable bound in the advice or pointcut.

general form:
call(*MethodPat*)
call(*ConstructorPat*)
execution(*MethodPat*)
execution(*ConstructorPat*)
initialization(*ConstructorPat*)
preinitialization(*ConstructorPat*)
staticinitialization(*TypePat*)
get(*FieldPat*)
set(*FieldPat*)
handler(*TypePat*)
adviceexecution()
within(*TypePat*)
withincode(*MethodPat*)
withincode(*ConstructorPat*)
cflow(*Pointcut*)
cflowbelow(*Pointcut*)
if(*Expression*)
this(*Type* | *Var*)
target(*Type* | *Var*)
args(*Type* | *Var* , …)
where *MethodPat* is:
[*ModifiersPat*] *TypePat* [*TypePat* .] *IdPat* (*TypePat* | .., …)
[**throws** *ThrowsPat*]
ConstructorPat is:
[*ModifiersPat*] [*TypePat* .] **new** (*TypePat* | .. , …)
[**throws** *ThrowsPat*]
FieldPat is:
[*ModifiersPat*] *TypePat* [*TypePat* .] *IdPat*
TypePat is one of:
IdPat [**+**] [**[]** …]
! *TypePat*
TypePat **&&** *TypePat*
TypePat **||** *TypePat*
(*TypePat*)