Programação Orientada por Aspectos

Marco Túlio Valente
DCC – UFMG

Referências

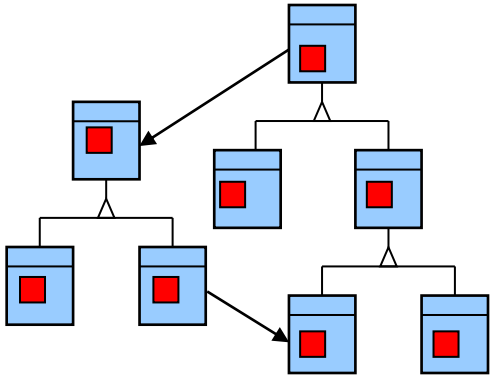
- Mini-curso:
  - Fabio Tirelo; Roberto da Silva Bigonha; Mariza Andrade da Silva Bigonha; Marco Túlio de Oliveira Valente. Desenvolvimento de Software Orientado por Aspectos. XXIII Jornada de Atualização em Informática (JAI), 2004.
- Livro:
  - Ramnivas Laddad. AspectJ in Action. Practical Aspect-Oriented Programming. July 2003, 512 pages.
- Artigos:
  - Gregor Kiczales et al. Aspect-Oriented Programming. ECOOP 1997.
  - Gregor Kiczales et al. An Overview of AspectJ. ECOOP 2001

Sugestão Inicial de Leitura

- Objects Never? Well, Hardly Ever!
- Mordechai Ben-Ari
- Communications of the ACM. Vol. 53 No. 9, Pages 32-35, 2010

Programação Orientada por Aspectos

- Orientação por objetos: paradigma dominante de programação
- Unidade básica de modularização: classes
- Desenvolvimento orientado por objetos:
  - Decomposição do sistema em classes
  - Classes implementam interesses (concerns) do sistema
- Concerns: requisitos, funcionalidades, propriedades etc
- Limitação de OO: determinados interesses não são facilmente implementados em uma ou em poucas classes do sistema
- São chamados de interesses transversais (crosscutting concerns)



Programação Orientada por Aspectos

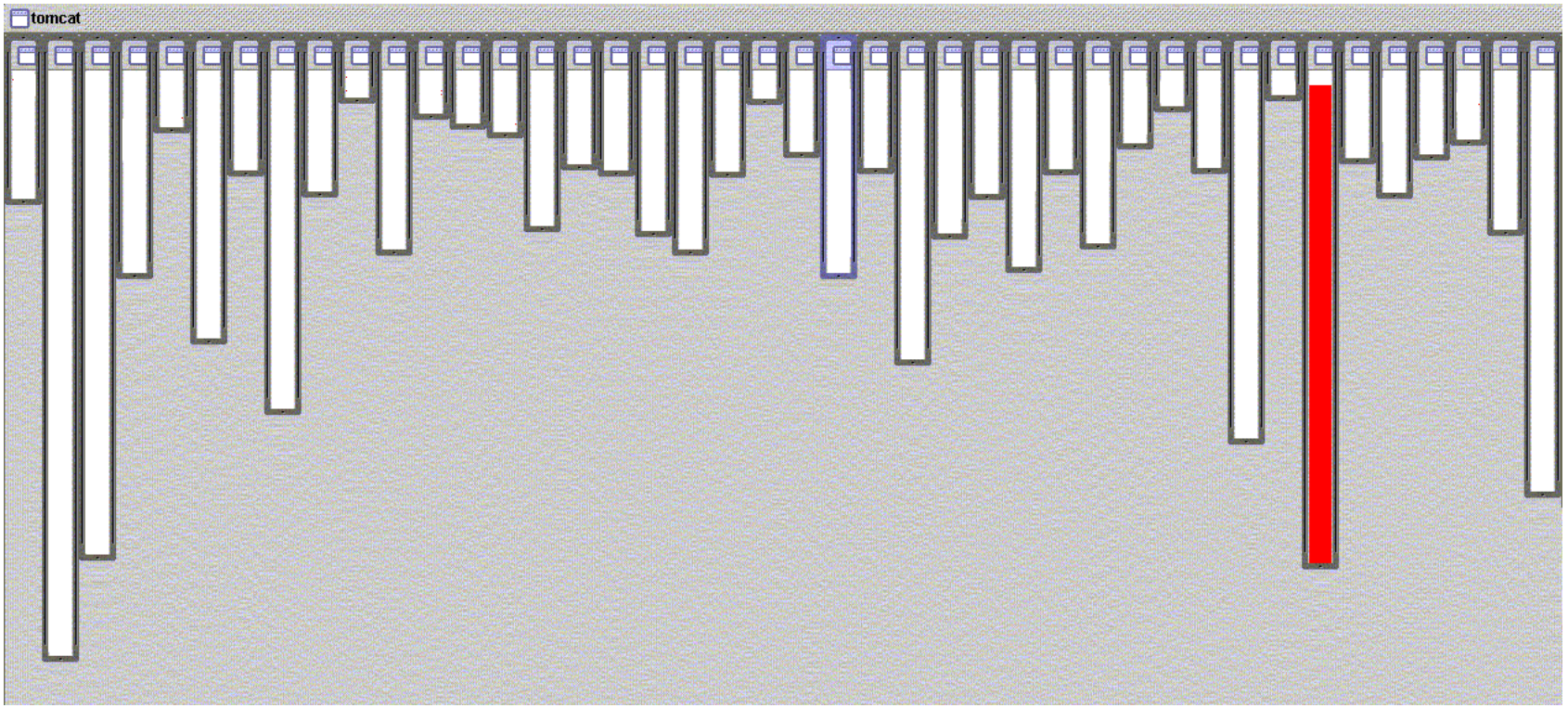
- Exemplos de interesses transversais: requisitos não-funcionais
  - Logging, autenticação, controle de acesso, distribuição, controle de concorrência, controle de transações, persistência

- No entanto, certos requisitos funcionais também podem apresentar um comportamento transversal

- Problemas com requisitos transversais:
  - *Code tangling* (código entrelaçado): requisito transversal misturado com código de negócio
  - *Code spreading* (código espalhado): requisito transversal implementado em diversas classes

Exemplo 1: Tomcat

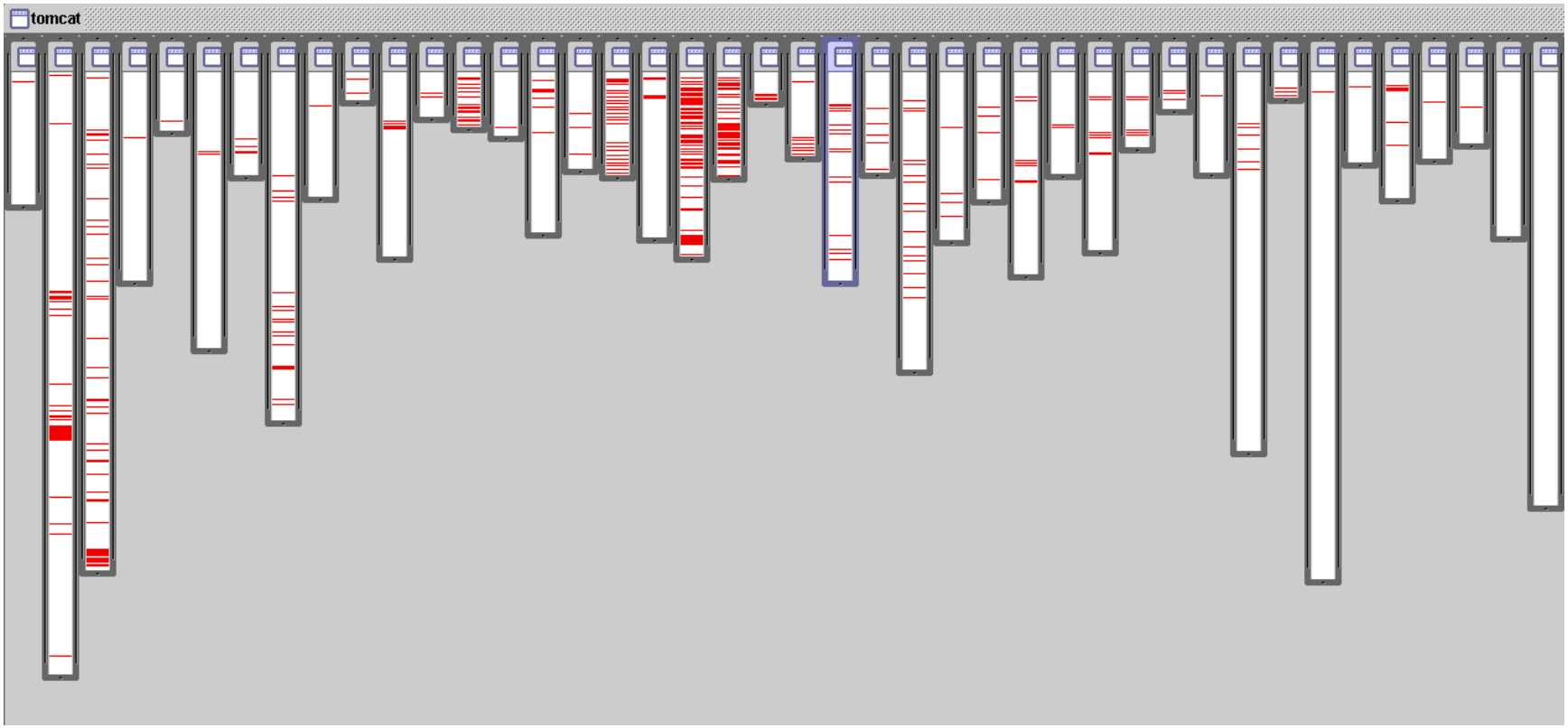
- Exemplo de requisito não-transversal: parsing de documentos HTML



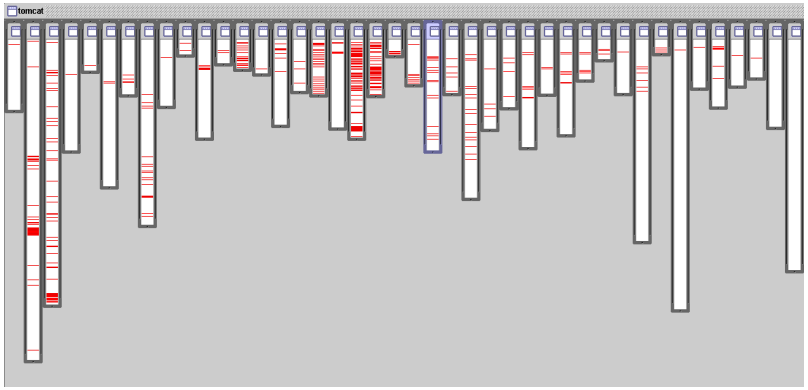
Fonte: Erik Hilsdale and Mik Kersten. *Aspect-Oriented Programming with AspectJ*, Mini-course at OOPSLA-2001.

Exemplo 1: Tomcat

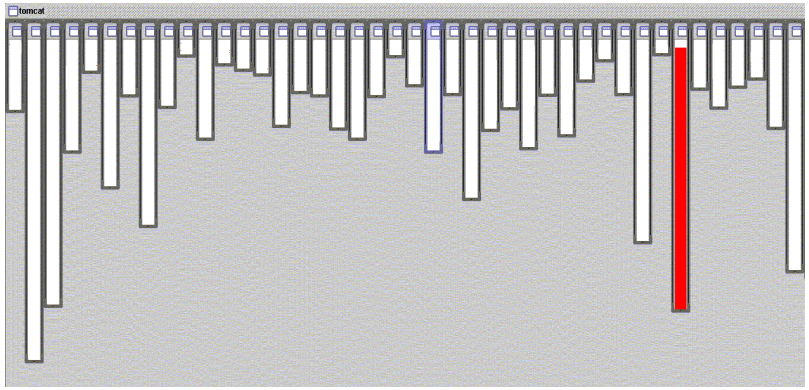
- Exemplo de requisito transversal: logging
  - Espalhamento (spreading) e Entrelaçamento (tangling)



OOP vs AOP



⇒



OOP **AOP**

- Uma linguagem orientada por aspectos permite confinar requisitos transversais em módulos
- Em AOP, tais módulos são chamados de aspectos
- AOP, portanto, é um complemento a OOP
- AOP não veio substituir OOP

Kiczales, Aspect Oriented Programming (ECOOP 97)

- “We have found many programming problems for which neither procedural nor object-oriented programming techniques are sufficient to clearly capture some of the important design decisions the program must implement. This forces the implementation of those design decisions to be scattered throughout the code, resulting in tangled code that is excessively difficult to develop and maintain.”
- “We present an analysis of why certain design decisions have been so difficult to clearly capture in actual code. We call the properties these decisions address aspects, and show that the reason they have been hard to capture is that they cross-cut the system’s basic functionality.”
- “We present the basis for a new programming technique, called aspect oriented programming, that makes it possible to clearly express programs involving such aspects, including appropriate isolation, composition and reuse of the aspect code.”

Exemplo 2

- Exemplo: Descrever um carro por meio de OO
- Algumas classes possíveis:
  - Carro, Peça, Vela, Distribuidor, Correia, Motor, Injeção Eletrônica, Arrefecimento
- Possuem relações:
  - É-PARTE-DE (Composição)
  - É-UM (Herança)
- Não são adequadamente modelados por meio de herança e composição:
  - Conforto
  - Aerodinâmica
  - Segurança
- Problema: estes requisitos “atravessam” as decisões de implementação de diversos itens do carro

Exemplo 3

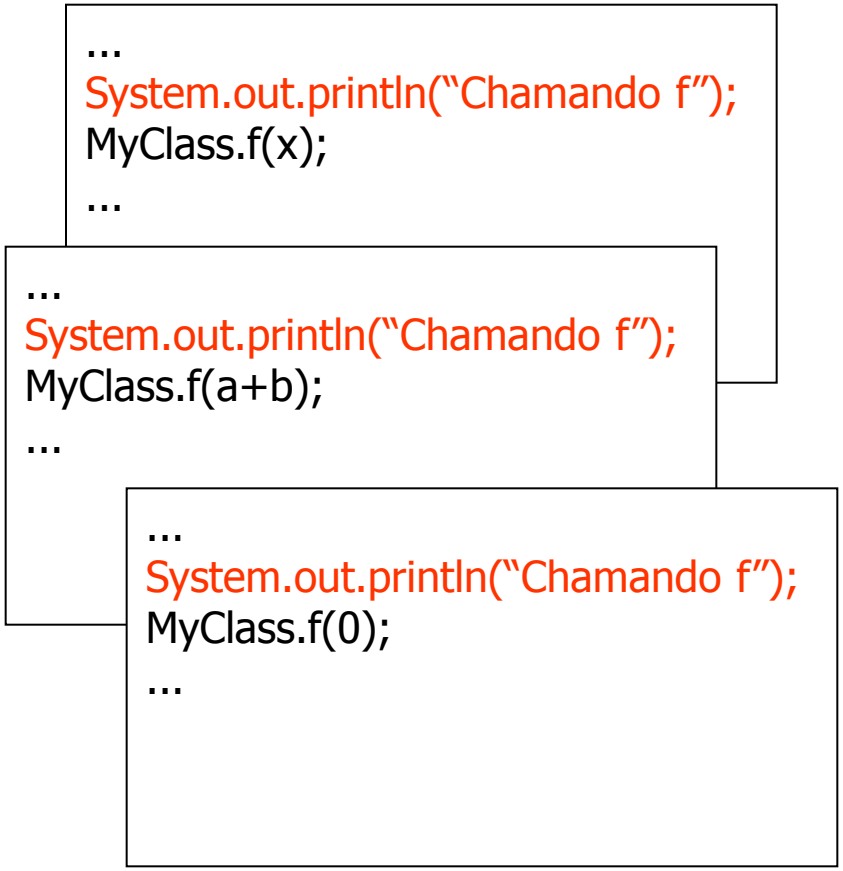
```
public class SomeBusinessClass extends OtherBusinessClass {
   “Dados membros do módulo”
   “Métodos redefinidos da superclasse”
   public void performSomeOperation(OperationInformation info) {
      “REALIZA A OPERAÇÃO OBJETIVO"
   }
   “Outras operações semelhantes à anterior”
}
```

Exemplo 3

```
public class SomeBusinessClass extends OtherBusinessClass {
   "Dados membros do módulo"
   "Outros dados membros: stream de logging, flag de consistência"
   "Métodos redefinidos da superclasse"
   public void performSomeOperation(OperationInformation info) {
      "Garante autenticidade"
      "Garante que info satisfaça contrato"
      "Bloqueia o objeto para garantir consistência entre threads"
      "Garante que a cache está atualizada"
      "Faz o logging do início da operação"
      "REALIZA A OPERAÇÃO OBJETIVO"
      "Faz o logging do final da operação"
      "Desbloqueia o objeto"
   }
   "Outras operações semelhantes à anterior"
   public void save(PersistenceStorage ps) { "..." }
   public void load(PersistenceStorage ps) { "..." }
}
```

Exemplo 4: Métodos

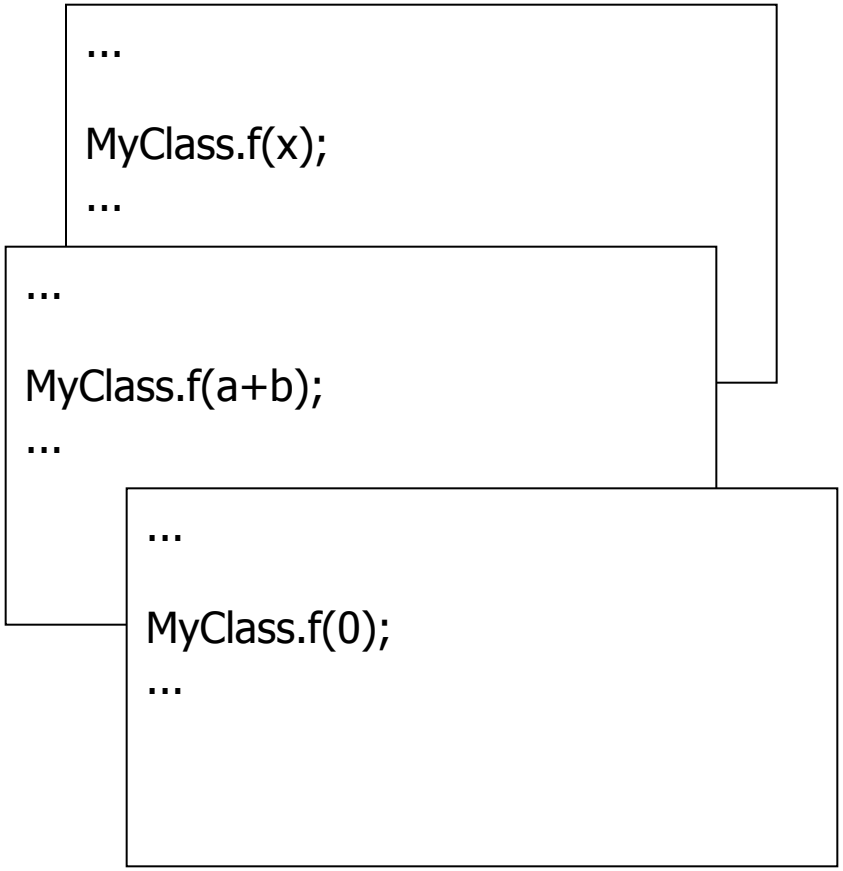
Ex: Toda chamada de f() é precedida por um registro de operação




```
class MyClass {
    ...
    public static void f(int n) {

        ...
    }
    ...
}
```

Exemplo 4: Métodos

Solução: procedimentos




```
class MyClass {
    ...
    public static void f(int n) {
        System.out.println("Chamando f");
        ...
    }
    ...
}
```

Exemplo 4: Herança

Ex: métodos comuns em classes

```
class Ponto {
    private int x, y;
    public int getX() { ... }
    public int getY() { ... }
    public void setX(int x) { ... }
    public void setY(int y) { ... }
    public void draw(Graphics g) { ... }
    public void refresh() {
        draw(Canvas.getGraphics());
    }
}
```

```
class Linha {
    private Ponto p1, p2;
    public int getP1() { ... }
    public int getP2() { ... }
    public void setP1(Ponto p) { ... }
    public void setP2(Ponto p) { ... }
    public void draw(Graphics g) { ... }
    public void refresh() {
        draw(Canvas.getGraphics());
    }
}
```

Exemplo 4: Herança

Solução: herança como mecanismo de reúso

```
abstract class Figura {
    public abstract void draw(Graphics g);
    public void refresh() {
        draw(Canvas.getGraphics());
    }
}
```

```
class Ponto extends Figura {
    private int x, y;
    public int getX() { ... }
    public int getY() { ... }
    public void setX(int x) { ... }
    public void setY(int y) { ... }
    public void draw(Graphics g) { ... }



}
```

```
class Linha extends Figura {
    private Ponto p1, p2;
    public int getP1() { ... }
    public int getP2() { ... }
    public void setP1(Ponto p) { ... }
    public void setP2(Ponto p) { ... }
    public void draw(Graphics g) { ... }



}
```

Exemplo 4: Aspectos

Ex: chamada a refresh() no final de métodos

```
class Ponto extends Figura {
    ...
    public void setX(int x) {
        this.x = x; refresh();
    }
    public void setY(int y) {
        this.y = y; refresh();
    }
    ...
}
```

```
class Linha extends Figura {
    ...
    public void setP1(Ponto p) {
        this.p1 = p; refresh();
    }
    public void setP2(Ponto p) {
        this.p2 = p; refresh();
    }
    ...
}
```

Exemplo 4: Aspectos

Solução: aspectos

```
aspect RefreshingAspect {
    after(): execution(void Figura+.set*(..)) {
        refresh();
    }
}
```

```
class Ponto extends Figura {
    ...
    public void setX(int x) {
        this.x = x;
    }
    public void setY(int y) {
        this.y = y;
    }
    ...
}
```

```
class Linha extends Figura {
    ...
    public void setP1(Ponto p) {
        this.p1 = p;
    }
    public void setP2(Ponto p) {
        this.p2 = p;
    }
    ...
}
```

Separation of Concerns

- Principal objetivo de orientação por aspectos
- Idéia básica: um interesse, um módulo
- Dijkstra (1968): “we know that a program must be correct and we can study it from that viewpoint only; we also know that is should be efficient and we can study its efficiency on another day […] But nothing is gained – on the contrary – by tackling these various aspects simultaneously. It is what I sometimes have called “the separation of concerns.”
- Parnas (1972): “we have tried to demonstrate by these examples that it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others.”

Desenvolvimento Orientado por Aspectos

- Identificação e caracterização dos requisitos
- Implementação de requisitos não-transversais em classes
- Implementação de requisitos transversais em aspectos
- Weaver: ferramenta que combina código OO e OA para gerar sistema final

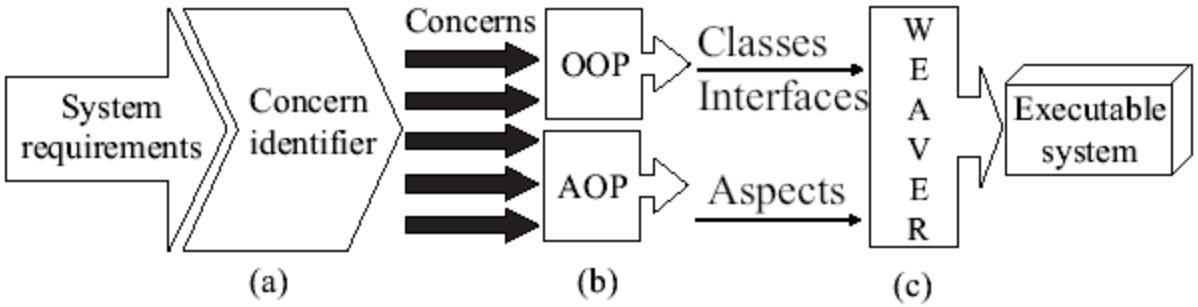


Figure 1. Aspect-Oriented development phases.

Fonte: Soares et al. Distribution and Persistence as Aspects, SPE, 2005.

AOP: Histórico

- 1968: Separation of Concerns (Dijkstra)
- 1972: Separation of Concerns (Parnas)
  - David Parnas. On the Criteria To Be Used in Decomposing Systems into Modules, CACM, December 1972.
- 1992: Adaptive Programming (Lieberherr)
- 1992: Composition Filters (Aksit)
- 1997: Aspect Oriented Programming
  - Kiczales G. et al. Aspect-Oriented Programming. ECOOP 1997.
- 1998: AspectJ (primeira versão pública, Xerox PARC)
- 2001: AspectJ
  - Gregor Kiczales et al. An Overview of AspectJ. ECOOP 2001
- Mais detalhes:
  - Cristina Videira Lopes. AOP: A Historical Perspective, 2004

AOP: Conceitos Fundamentais

- Conceitos fundamentais:
  - Pontos de junção (*joinpoints*)
  - Conjuntos de junção (*pointcuts*)
  - Regras de junção (*advices*)
  - Aspectos
- Pontos de junção são pontos da execução do programa:
  - Chamadas e execuções de métodos
  - Chamadas e execuções de construtores
  - Retorno de métodos
  - Retorno de construtores
  - Lançamento de exceções
  - Tratamento de exceções
  - Alteração de campos de classe

AOP: Conceitos Fundamentais

- Conjuntos de junção são conjuntos de pontos de junção
- Exemplos:
  - Todas as chamadas de métodos públicos
  - Toda criação de objetos da classe Point
  - Toda chamada do método Point.setX ou Point.setY
  - Toda alteração do campo x da classe Point
- Advices definem como interferir nos pontos de junção
- Exemplo:
  - Chamar o método Logger.logEntry antes de toda chamada de método público do programa
  - Chamar o método notifyObservers após toda execução dos métodos Point.setX e Point.setY
- Aspectos: coleção de pointcuts e advices

AOP: Conceitos Fundamentais

- Gregor Kiczales et al. An Overview of AspectJ. ECOOP 2001.
- “Join points are well-defined points in the execution of the program”
- “Pointcuts are a means of referring to collections of join points and certain values at those join points”
- “Advices are method-like constructs used to define additional behavior at join points”
- “Aspects are units of modular crosscutting implementation, composed of pointcuts, advice, and ordinary Java member declarations”

AspectJ

Marco Túlio Valente
DCC – UFMG

AspectJ

- Extensão de Java com recursos para AOP
- Principal conceito: aspectos
- Declarações de aspectos são similares a declarações de classes
- Em um aspecto, podem ser declarados:
  - Conjuntos de junção (pointcuts)
  - Advices
- AspectJ suporta dois tipos de implementações de requisitos transversais:
  - Transversalidade dinâmica: permite definir implementação adicional em pontos bem definidos do programa
  - Transversalidade estática (*intertype declarations*): permite alterar as assinaturas estáticas das classes e interfaces de um programa Java

Joinpoints: call e execution

- Joinpoints: “pontos bem definidos” da execução de um programa
- Pointcuts: conjuntos de pontos de junção
- Exemplo de sintaxe para definição de pointcuts:
  - `pointcut nome(arg1,..., argn): call (assinatura_metodo);`
  - `pointcut nome(arg1,..., argn): execution(assinatura_metodo);`
- call vs execution
  - call: ponto corresponde à chamada do método
  - execution: ponto corresponde à execução do método chamado
- Joinpoints podem ser combinados usando: || && !
- Assinaturas de métodos podem utilizar curingas (wildcard)
- Exemplos:

```
pointcut publicMethods(): execution(public * *(..));
pointcut LoggerCalls(): execution(* Logger.*(..));
pointcut loggableCalls(): publicMethods() && !logObjectCalls();
```

Advices

- Advice: bloco de código que é executado em um ponto de junção
- Podem ser de três tipos:
  - before: código é executado antes do ponto de junção
  - after: código é executado após o ponto de junção
  - around: código é executado “no lugar” do ponto de junção
- Sintaxe para declaração de advices:

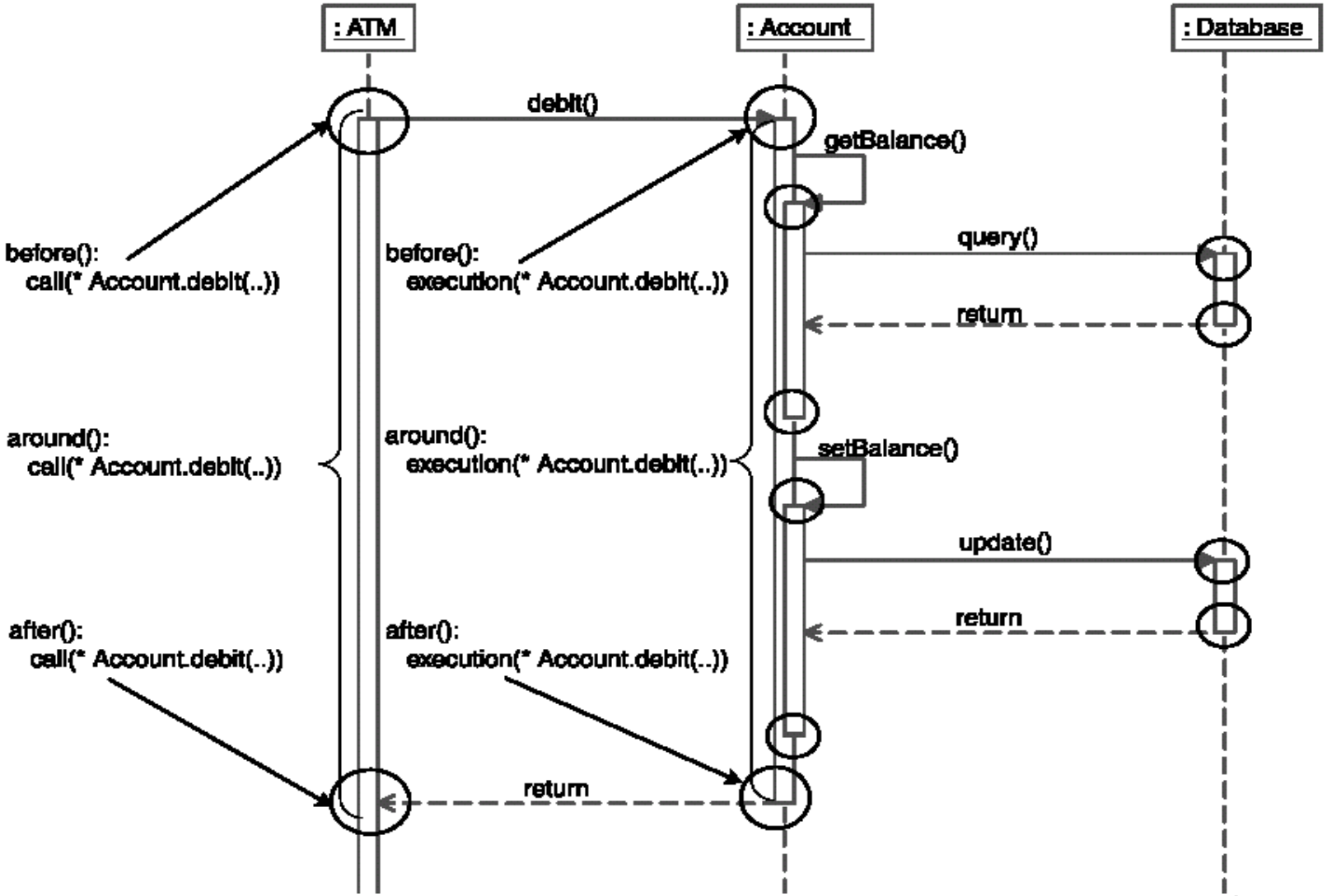
```
before/after/around (argumentos): nome_pointcut {
   “código a ser costurado”
}
```

Aspectos

- Aspecto: conjunto de poincuts e advices

```
public aspect LoggingAspect {
   pointcut publicMethods(): execution(public * *(..));

   pointcut logObjectCalls(): execution(* Logger.*(..));

   pointcut loggableCalls(): publicMethods() && !logObjectCalls();

   before(): loggableCalls() {
      Logger.logEntry(thisJoinPoint.getSignature().toString());
   }

   after(): loggableCalls() {
      Logger.logExit(thisJoinPoint.getSignature().toString());
   }
}
```

before x after x around; call x execution



Fonte: Laddad, R. AspectJ in Action (incluindo próximos slides).

Around advice

Listing 3.3 FailureHandlingAspect.java

```
import java.rmi.RemoteException;

public aspect FailureHandlingAspect {
    final int MAX_RETRIES = 3;

    Object around() throws RemoteException
        : call(* RemoteService.get*(..) throws RemoteException) {
        int retry = 0;
        while(true){
            try{
                return proceed();
            } catch(RemoteException ex){
                System.out.println("Encountered " + ex);
                if (++retry > MAX_RETRIES) {
                    throw ex;
                }
                System.out.println("\tRetrying...");
            }
        }
    }
}
```

❶ Method part of advice

❷ Pointcut (anonymous) part of advice

❸ Execution of captured join point

After Advice

- Exemplo 1: “after” resultado normal ou exceção

```
after(): loggableCalls() {
   Logger.logExit(thisJoinPoint.getSignature().toString());
}
```

- Exemplo 2: “after” resultado normal

```
after() returning (): loggableCalls() {
   Logger.doExit(thisJoinPoint.getSignature() +
   " with normal return");
}
```

- Exemplo 3: “after” exceção

```
after() throwing (Exception exc): loggableCalls() {
   Logger.doExit(thisJoinPoint.getSignature() +
   " with " + exc + " thrown");
}
```

Assinaturas de Métodos

Table 3.3 Examples of method signatures *(continued)*

Signature Pattern	Matched Methods
`public void Account.set*(*)`	All public methods in the `Account` class with a name starting with `set` and taking a single argument of any type.
`public void Account.*()`	All public methods in the `Account` class that return `void` and take no arguments.
`public * Account.*()`	All public methods in the `Account` class that take no arguments and return any type.
`public * Account.*(..)`	All public methods in the `Account` class taking any number and type of arguments.
`* Account.*(..)`	All methods in the `Account` class. This will even match methods with `private` access.
`!public * Account.*(..)`	All methods with nonpublic access in the `Account` class. This will match the methods with `private`, default, and `protected` access.
`public static void Test.main(String[] args)`	The `static main()` method of a `Test` class with `public` access.
`* Account+.*(..)`	All methods in the `Account` class or its subclasses. This will match any new method introduced in `Account`'s subclasses.
`* java.io.Reader.read(..)`	Any `read()` method in the `Reader` class irrespective of type and number of arguments to the method. In this case, it will match `read()`, `read(char[])`, and `read(char[], int, int)`.

Assinaturas de Construtores

Table 3.4 Examples of constructor signatures

Signature Pattern	Matched Constructors
`public Account.new()`	A public constructor of the `Account` class taking no arguments.
`public Account.new(int)`	A public constructor of the `Account` class taking a single integer argument.
`public Account.new(..)`	All public constructors of the `Account` class taking any number and type of arguments.
`public Account+.new(..)`	Any public constructor of the `Account` class or its subclasses.
`public *Account.new(..)`	Any public constructor of classes with names ending with `Account`. This will match all the public constructors of the `SavingsAccount` and `CheckingAccount` classes.
`public Account.new(..) throws InvalidAccountNumberException`	Any public constructors of the `Account` class that declare they can throw `InvalidAccountNumberException`.

Joinpoints: set, get, within, withincode

- Para acessos a campos de classe, utilizam-se os operadores:
  - **get**(campo)
  - **set**(campo)
- Exemplo:

```
pointcut alteraX(): set(int Point.x);
```

- Definição baseada na estrutura do programa:
  - **within**(classe)
  - **withincode**(método)
- Exemplos:

```
pointcut setFieldOutsideSetter(): set(int Point.x)
      && ! withincode(void Point.setX(int));

pointcut createPointOutsideFactory(): call(Point.new(..))
      && ! withing(PointFactory);
```

Assinaturas de Campos; Assinaturas baseadas em Escopo Léxico

Table 3.5 Examples of field signatures

Signature Pattern	Matched Fields
`private float Account._balance`	Private field `_balance` of the `Account` class
`* Account.*`	All fields of the `Account` class regardless of an access modifier, type, or name

Table 3.8 Examples of lexical-structure based pointcuts

Pointcut	Natural Language Description
`within(Account)`	Any join point inside the `Account` class's lexical scope
`within(Account+)`	Any join point inside the lexical scope of the `Account` class and its subclasses
`withincode(* Account.debit(..))`	Any join point inside the lexical scope of any `debit()` method of the `Account` class
`withincode(* *Account.get-Balance(..))`	Any join point inside the lexical scope of the `getBalance()` method in classes whose name ends in `Account`

Transversalidade Estática

- Transversalidade dinâmica permite modificar o comportamento da execução do programa (introduzindo advices em joinpoints)
- Transversalidade estática permite redefinir a estrutura estática dos tipos
- Transversalidade estática: *intertype declarations* ou *introductions*
- Transversalidade estática em AspectJ:
  - introdução de campos e métodos em classes e interfaces;
  - modificação da hierarquia de tipos;
  - declaração de erros e advertências de compilação;
  - enfraquecimento de exceções.

Transversalidade Estática

- Modificando a superclasse de um tipo:
  - declare parents: TypePattern extends TypeList;
- Adicionando uma interface a um tipo:
  - declare parents: TypePattern implements TypeList;
- Exemplo:

```
aspect A {
  declare parents: SomeClass implements Runnable;
  public void SomeClass.run() { .... }
}
```

Transversalidade Estática: Exemplo

Listing 3.6 MinimumBalanceRuleAspect.java

```
public aspect MinimumBalanceRuleAspect {
    private float Account._minimumBalance;

    public float Account.getAvailableBalance() {
        return getBalance() - _minimumBalance;
    }

    after(Account account) :
        execution(SavingsAccount.new(..)) && this(account) {
        account._minimumBalance = 25;
    }

    before(Account account, float amount)
        throws InsufficientBalanceException :
        execution(* Account.debit())
        && this(account)  && args(amount) {
        if (account.getAvailableBalance() < amount) {
            throw new InsufficientBalanceException(
                "Insufficient available balance");
        }
    }
}
```

Introducing a data member

Introducing a method

Using the introduced data member

Using the introduced method

AspectJ: Exemplos

Marco Túlio Valente
DCC – UFMG

Exemplo 1: Padrão Observador

- Implementação em Java origina espalhamento e intrusão

```
public class Temperature extends Subject {
  private double value;
  public double getValue() {
    return value;
  }
  public void setValue(double value) {
    this.value= value;
    notifyObservers();
  }
}
public class TermometerCelsius implements Observer {
  public void update(Subject s) {
    double value= ((Temperature) s).getValue();
    System.out.println("Celsius: " + value);
  }
}
```

Aspecto ObserverProtocol (1/2)

- Modulariza protocolo sujeito/observador
- Abstrai “mecânica” do padrão
- Aspecto abstrato: possui métodos e/ou pointcuts abstratos

```
public abstract aspect ObserverProtocol {
  public interface Subject {}
  public interface Observer {}
  private HashMap soHash= new WeakHashMap();

  private List getObservers(Subject s) {
    List observers= (List) soHash.get(s);
    if (observers == null) {
       observers= new LinkedList();
       soHash.put(s, observers);
    }
    return observers;
  }
```

Aspecto ObserverProtocol (2/2)

```
  public void addObserver(Subject s, Observer o) {
    getObservers(s).add(o);
    updateObs(s,o);
  }
  public void removeObserver(Subject s, Observer o) {
    getObservers(s).remove(o);
  }

  public abstract pointcut subjectChange(Subject s);
  public abstract void updateObs(Subject s, Observer o);

  after(Subject s): subjectChange(s) {
    Iterator it = getObservers(s).iterator();
    while (it.hasNext()) {
      Observer o = (Observer) it.next();
      updateObs(s,o);
    }
  }
}
```

Sujeito

- Código do padrão “desaparece” do sujeito e dos observadores

```
public class Temperature {
  private double value;
  public Temperature(double initialValue) {
    value = initialValue;
  }
  public double getValue() {
    return value;
  }
  public void setValue(double value) {
    this.value = value;
  }
}
```

Observadores

```
public abstract class Termometer {
  public abstract void printTemperature(double value);
}

public class TermometerCelsius extends Termometer {
  public void printTemperature(double value) {
    System.out.println("Celsius: " + value);
  }
}

public class TermometerFahrenheit extends Termometer {
...
}
```

Observação de Temperatura

- Estende aspecto abstrato ObserverProtocol
- Implementa pointcut subjectChange e método updateObserver

```
public aspect TemperatureObservation
       extends ObserverProtocol {
  declare parents: Temperature implements Subject;
  declare parents: Termometer implements Observer;

  public pointcut subjectChange(Subject s):  target(s) &&
     execution(void Temperature.setValue(double));

  public void updateObs(Subject s, Observer o) {
    Temperature temp= (Temperature) s;
    Termometer term= (Termometer) o;
    term.printTemperature(temp.getValue());
  }
}
```

Instanciação e Configuração

- Normalmente, aspectos funcionam como um singleton
- São automaticamente criados e inicializados antes do primeiro uso
- "Método" aspectOf permite acesso à instância única de um aspecto
- Exemplo:

```
Temperature t= new Temperature();
Termometer term1= new TermometerCelsius();
Termometer term2= new TermometerFahrenheit();
TemperatureObservation.aspectOf().addObserver(t,term1);
TemperatureObservation.aspectOf().addObserver(t,term2);
```

- Conclusões:
  - Implementação não-intrusiva do padrão
  - Padrão modularizado
  - Implementação do padrão é reutilizável