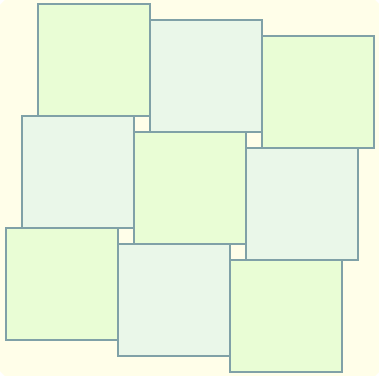
XXIII Jornadas de Atualização em Informática

Desenvolvimento de Software Orientado por Aspectos



Fabio Tirelo (PUC Minas)
Roberto S. Bigonha (UFMG)
Mariza A. S. Bigonha (UFMG)
Marco Túlio O. Valente (PUC Minas)

Programação Orientada por Aspectos

- Criada por Gregos Kiczales em 1997, como um projeto PARC/Xerox
- Na Programação Orientada por Objetos, a implementação de alguns requisitos violam a modularização natural da implementação
- Objetivo da AOP: modularizar elementos de projeto que não são adequadamente modelados na POO

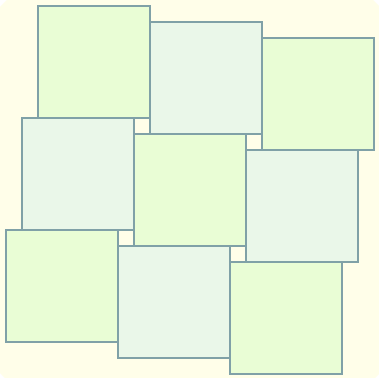
Objetivos

- Apresentar os princípios e as técnicas do desenvolvimento de software orientado por aspectos.
- Mostrar a aplicação de seus conceitos na linguagem AspectJ
- Realizar um estudo comparativo entre o desenvolvimento orientado por aspectos e o desenvolvimento puramente orientado por objetos.
- Exibir estudos de caso que mostrem a aplicabilidade da orientação por aspectos.

Sumário

- Primeiro dia:
  - Introdução: Modularidade
  - Requisitos transversais
  - Limitações da POO
  - Metodologia de Orientação por Aspectos
- Segundo dia:
  - Linguagem AspectJ
- Terceiro dia:
  - Aplicações de orientação por aspectos

Introdução: Modularidade



Modularidade

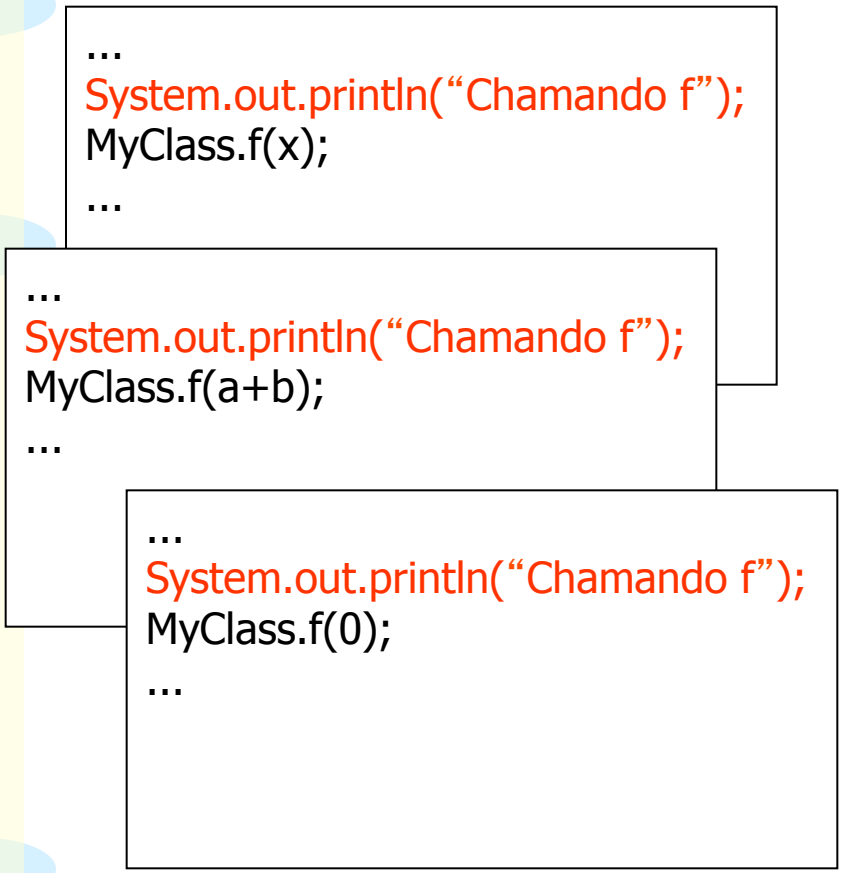
- Um *módulo* é um artefato de programação que pode ser desenvolvido e compilado separadamente de outras partes de compõem o programa

- Estratégias para modularização:
  - Programação estruturada
  - Programação orientada por objetos

Modularidade

- Programação Estruturada:
  - Implementação de abstrações dos comandos
- Programação Orientada por Objetos:
  - Implementação de interfaces
  - Herança
  - Polimorfismo

Modularidade

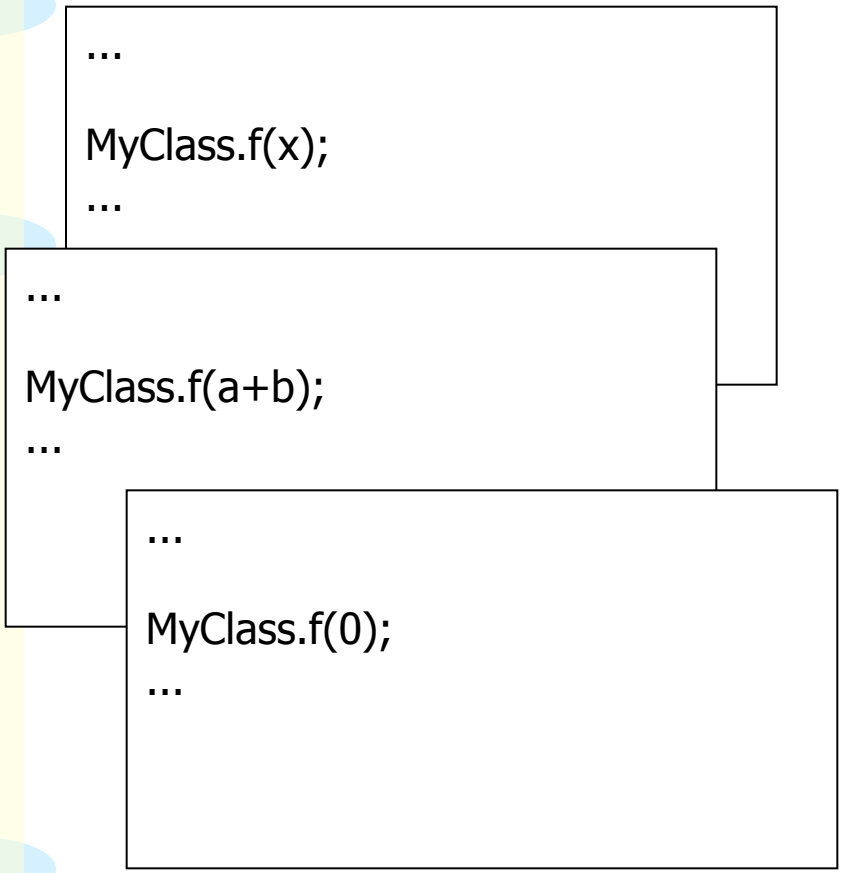


Ex: Toda chamada de f() é precedida por um registro de operação

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

        ...
    }
    ...
}
```

Modularidade



Solução: procedimentos

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

Modularidade

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());
    }
}
```

Modularidade

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) { ... }


}
```

Modularidade

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();
    }
    ...
}
```

Modularidade

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;
    }
    ...
}
```

Modularidade

- Exemplo: Descrever um carro por meio da Programação Orientada por Objetos

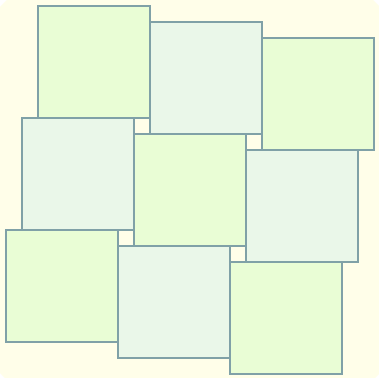
- Algumas classes possíveis:
  - Carro, Peça, Vela, Distribuidor, Correia, Motor, Injeção Eletrônica, Arrefecimento

- Possuem relações:
  - É-PARTE-DE (Composição)
  - É-UM (Herança)

Modularidade

- 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

Requisitos Transversais e Limitações da POO



Requisitos Transversais

- Requisitos de Sistemas:
  - Funcionais: objetivo final do sistema
  - Não-funcionais: elementos de projeto muitas vezes sem relação direta com o problema em questão

Requisitos Transversais

- Exemplo: requisitos de um sistema de processamento de cartões de crédito
  - Processamento de compras
  - Processamento de pagamentos
  - Geração de boletos
  - Registro de operações (*logging*)
  - Garantia de integridade de transações
  - Segurança
  - Desempenho

Requisitos Transversais

- Requisitos não-funcionais geralmente não são mapeáveis diretamente em uma ou poucas classes do programa

- Estes requisitos são denominados Requisitos Transversais ou Preocupações Ortogonais (*crosscutting concerns*)

Requisitos Transversais

- Sistemas orientados por objetos:
  - Unidade de modularização é a CLASSE
  - Requisitos transversais são espalhados em muitas classes do programa

- Requisitos transversais são difíceis de modularizar na POO

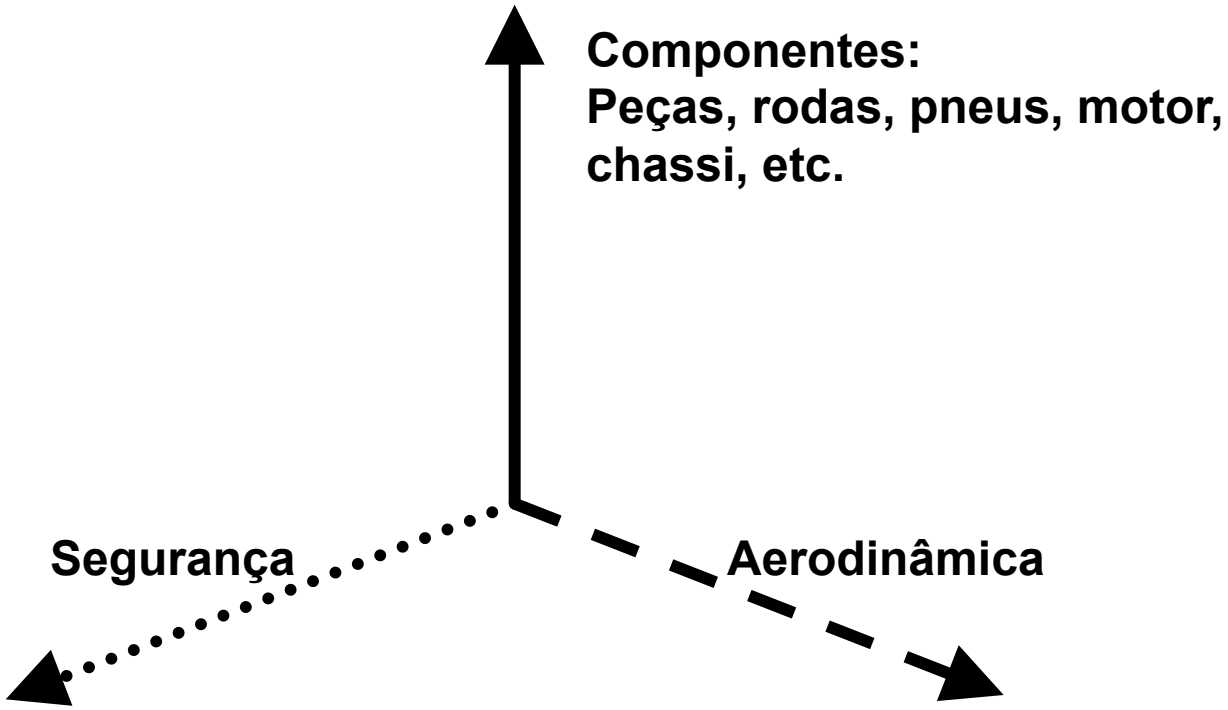
Requisitos Transversais

- Tratamento inadequado dos requisitos transversais pode resultar no baixo grau de modularização do sistema.

- Espalhamento dificulta:
  - Concepção
  - Implementação
  - Manutenção

  de requisitos transversais

Requisitos Transversais



- Espaço de Requisitos: multidimensional
- Devem ser implementados em espaço unidimensional

Requisitos Transversais

```
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”
}
```

```
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) { "..." }
}
```

Como alterar a política de registro de operações?

Requisitos Transversais

- **Espalhamento** (*scattering*): implementação de um requisito transversal estar disperso no programa

- **Intrusão** (*tangling*): confusão gerada por códigos de mais de um requisito transversal presentes em uma região do programa

Conseqüências

- **Rastreamento do programa é difícil**

Implementação simultânea de vários requisitos em um único método

Correspondência obscura entre os requisitos e suas implementações

Mapeamento pobre entre requisito e implementação

Conseqüências

- **Baixa produtividade**

Implementação simultânea de vários requisitos em um único módulo

Desvio da atenção do desenvolvedor do requisito principal para os requisitos periféricos

Ocorrência constante de erros

Conseqüências

- **Baixo grau de reúso**

Implementação simultânea de vários requisitos em um único módulo

Outros sistemas que precisarem implementar a mesma funcionalidade não poderão reutilizar prontamente o módulo

Conseqüências

- **Pouca qualidade interna**

Implementação simultânea de vários requisitos em um único módulo

Baixa coesão e
alto grau de acoplamento

Conseqüências

- **Difícil evolução**

Modificações posteriores no sistema

Alteração de muitos módulos

Exemplo I: Registro de Operações

- Registro de todas as operações realizadas durante a execução do programa:
  - Chamadas e retorno de métodos
  - Criação de objetos
  - Lançamento de exceções
  - Tratamento de exceções

- Para isso, define-se uma classe Logger com as primitivas de registro

Exemplo I: Registro de Operações

```
public class Logger {
    public static void logEntry(String message) {
        System.out.println("Entering " + message);
    }
    public static void logExit(String message) {
        System.out.println("Exiting " + message);
    }
    ... Outros métodos semelhantes a estes
}
```

Exemplo I: Registro de Operações

```
public class SomeClass {
    public void doSomething(int par) {
        Logger.logEntry("doSomething(" + par + ")");
        // Realiza a operação principal do sistema
        Logger.logExit("doSomething");
    }
}
```

Intrusão e espalhamento?

Exemplo I: Registro de Operações

- Onde está implementado o requisito?
- Como acrescentar registros de tratamentos de exceção?
- Como alterar a política de registro de operações?

Exemplo II: Padrão Observador

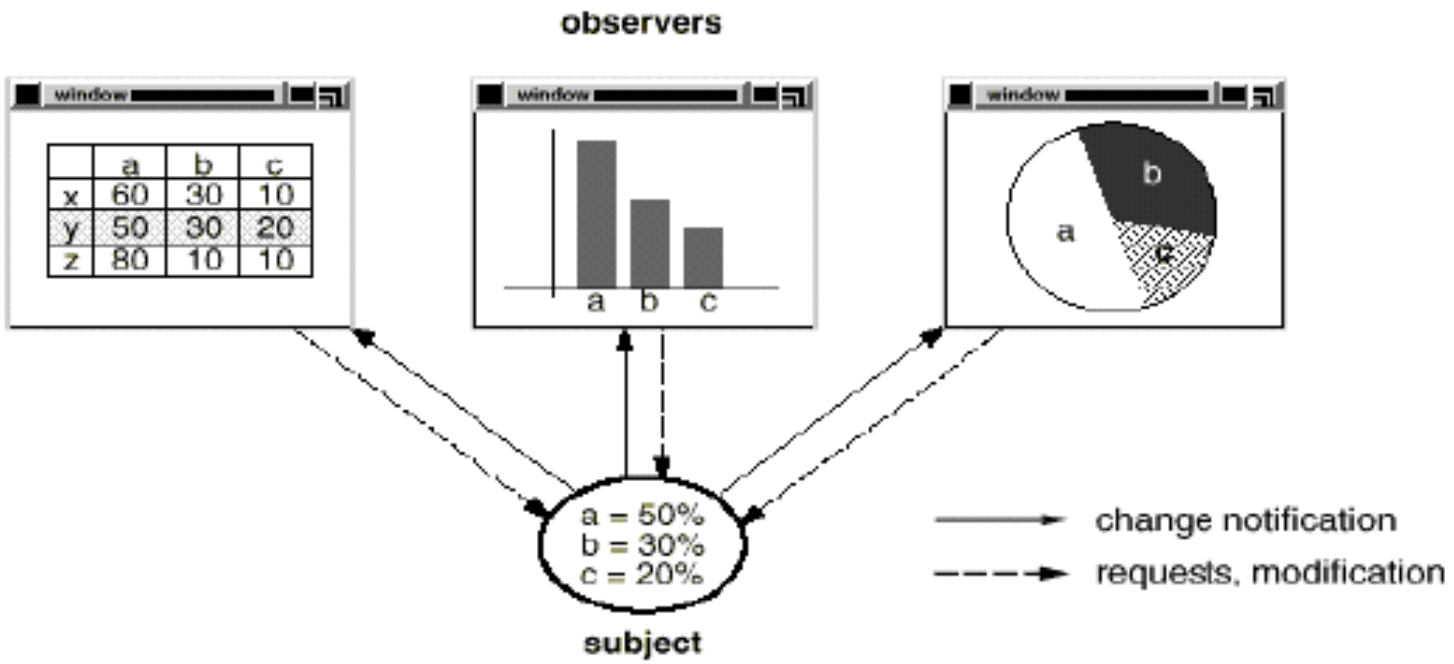
- Notificar e atualizar objetos dependentes sempre que o estado interno de um objeto é alterado
- A dependência é de um para muitos

- Utilizado para desacoplar objeto alvo de observação de seus observadores

  ➡ Maior grau de reúso

Exemplo II: Padrão Observador

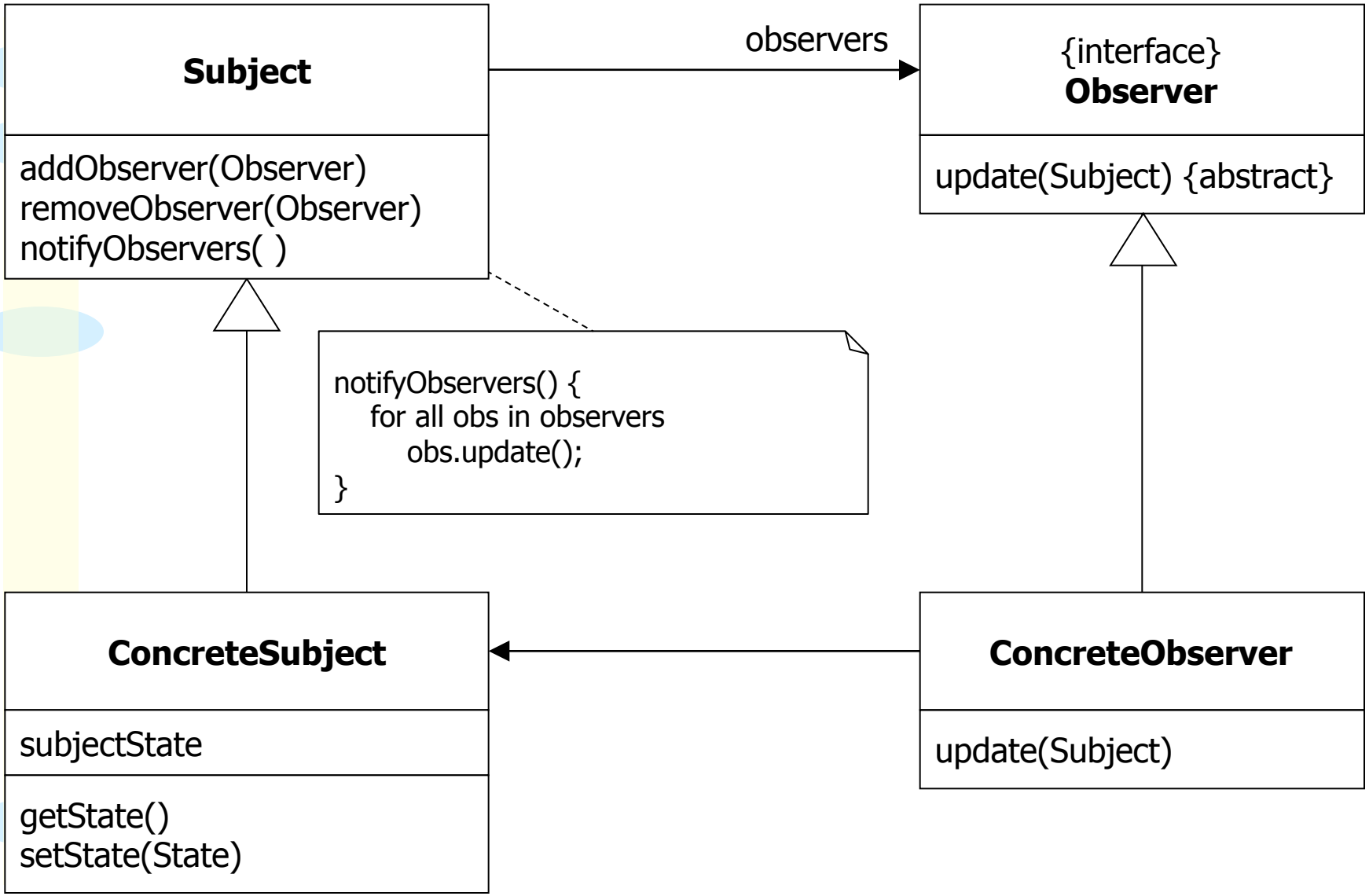
- Aplicações:
  - Tratamento de eventos em interfaces gráficas com o usuário
  - Separação de dados e apresentação



Exemplo II: Padrão Observador

- O padrão Observador é utilizado quando:
  - Uma abstração tem dois aspectos, um dependente do outro:
    - ➡ encapsular estes aspectos permite variar e reusar independemente
  - A mudança em um objeto requer modificações em outros
  - Um objeto deve fazer notificações a outros sem fazer considerações a respeito de suas implementações

Exemplo II: Padrão Observador



Exemplo II: Padrão Observador

```
public interface Observer {
    public void update(Subject s);
}
```

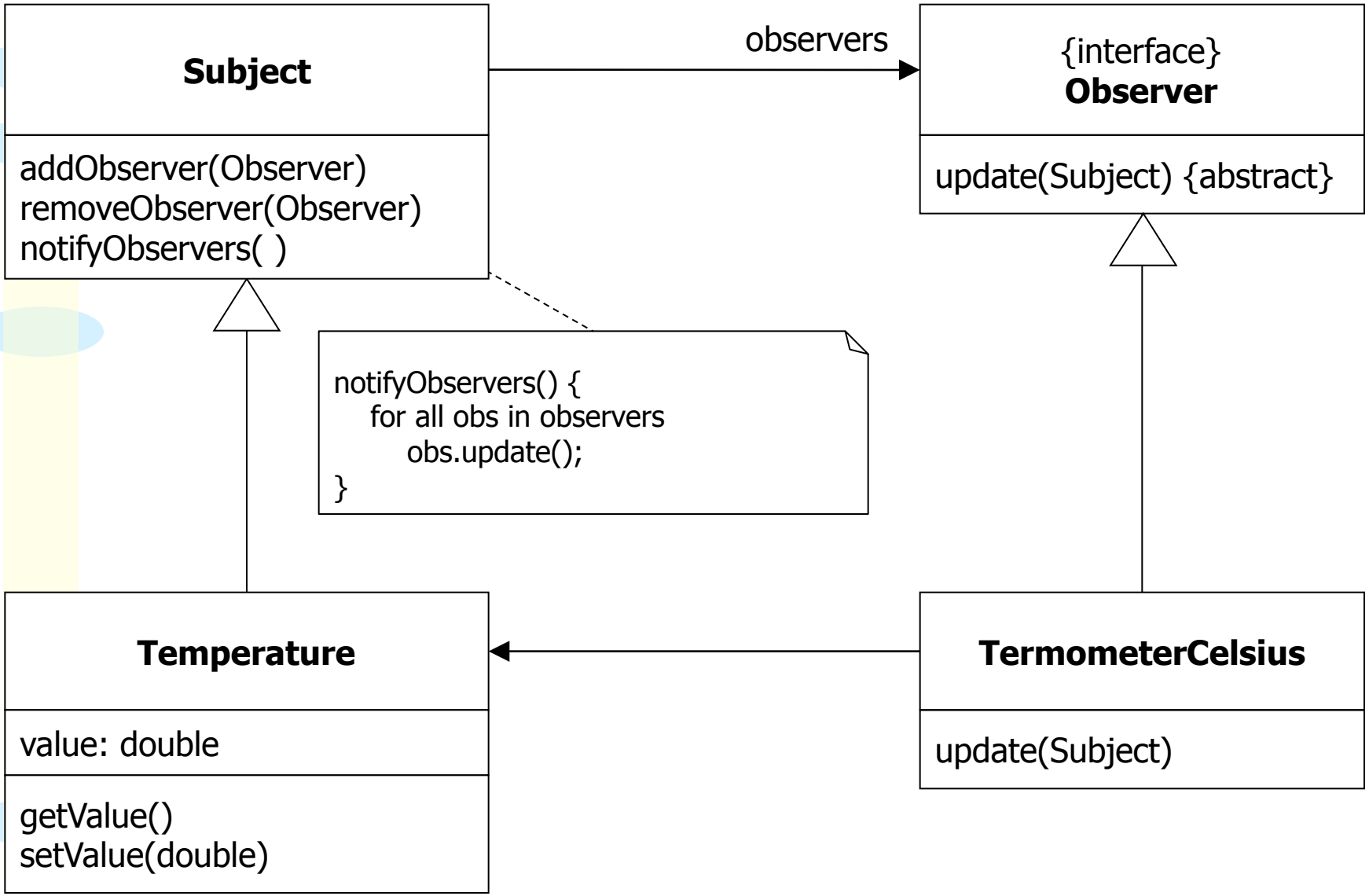
Exemplo II: Padrão Observador

```
public class Subject {
    private List observers = new LinkedList();
    public void addObserver(Observer observer) {
        observers.add(observer);
    }
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }
    public void notifyObservers() {
        ... // código no próximo slide
    }
}
```

Exemplo II: Padrão Observador

```
public void notifyObservers() {
    Iterator it = observers.iterator();
    while (it.hasNext()) {
        Observer obs = (Observer) it.next();
        obs.update(this);
    }
}
```

Exemplo II: Padrão Observador



Exemplo II: Padrão Observador

```
public class TesteTemperature {
    public static void main(String[] args) {
        Temperature t = new Temperature(10);
        t.addObserver(new CelsiusTermometer());
        t.addObserver(new FahrenheitTermometer());
        t.setValue(100);
    }
}
```

Saída:

```
Celsius: 100.0
Fahrenheit: 212.0
```

Exemplo II: Padrão Observador

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

Intrusão

Intrusão

Exemplo II: Padrão Observador

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

Intrusão

Intrusão

Exemplo II: Padrão Observador

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

```
public class TermometerCelsius {
    public void update(double value) {
        System.out.println("Celsius: " + value);
    }
}
```

Implementação sem intrusão

Exemplo II: Padrão Observador

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

        }
}
public class TermometerCelsius {
        public void update(double value) {

                System.out.println("Celsius: " + value);
        }
}
public class TermometerFahrenheit {
        public void update(double value) {

                System.out.println("Fahrenheit: " + value);
        }
}
```

Versão sem Padrão Observador

Exemplo II: Padrão Observador

```
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);
        }
}
public class TermometerFahrenheit implements Observer {
        public void update( Subject s ) {
                double value = 1.8 * ((Temperature) s).getValue() + 32;
                System.out.println("Fahrenheit: " + value);
        }
}
```

Versão com Padrão Observador

Espalhamento

Exemplo II: Padrão Observador

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

Intrusão e Espalhamento

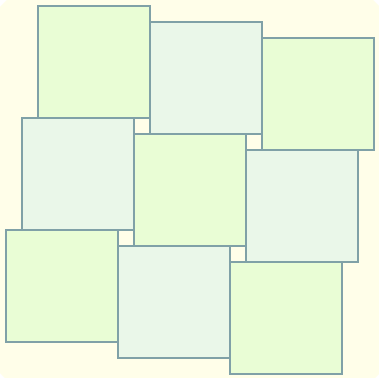
Exemplo II: Padrão Observador

- O padrão “desaparece no código”
- A sua implementação não é modular

- Adicionar ou remover o padrão no sistema é geralmente uma substituição invasiva e de difícil retrocesso

- Especificação do padrão é reusável
- Implementação do padrão não pode ser totalmente modularizada
- **Classes participantes do padrão não são totalmente reusáveis**

Metodologia de Desenvolvimento Orientado por Aspectos



Metodologia de Orientação por Aspectos

- Fases do desenvolvimento orientado por aspectos:
  - Decomposição de requisitos
  - Implementação separada dos requisitos
  - Recomposição de requisitos

Metodologia de Orientação por Aspectos

- Decomposição de requisitos:
  - separação dos requisitos em funcionais e transversais.

```
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) { "..." }
}
```

Metodologia de Orientação por Aspectos

- Classificação de requisitos transversais:
  - Infra-estrutura
  - Serviços
  - Paradigmas

Metodologia de Orientação por Aspectos

- Exemplos de requisitos de infra-estrutura:
  - Coordenação: escalonamento e sincronização
  - Distribuição: tolerância a falhas, comunicação, serialização e replicação
  - Persistência

Metodologia de Orientação por Aspectos

- Exemplos de requisitos de serviços:
  - Segurança
  - Recuperação de erros
  - Operações de retrocesso (undo)
  - Rastreamento
  - Registro de operações

Metodologia de Orientação por Aspectos

- Exemplos de requisitos de paradigma:
  - Padrão visitor
  - Padrão observer

Metodologia de Orientação por Aspectos

- Após a decomposição, implementam-se os requisitos **separadamente**
- Utiliza-se POO para a implementação
- Para cada requisito, pode-se:
  - Implementar classes e interfaces específicas para o problema
  - Utilizar bibliotecas externas

Metodologia de Orientação por Aspectos

- Após a implementação separada de cada requisito, especificam-se as **regras de recomposição** do sistema.
- Estas regras são implementadas como **aspectos**.
- Aspectos definem como compôr requisitos em um processo denominado costura (*weaving*).

```
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) { "..." }
}
```

Metodologia de Orientação por Aspectos

- Conceitos fundamentais:
  - Pontos de junção (*joinpoints*)
  - Conjuntos de junção (*pointcuts*)
  - Regras de junção (*advices*)

Metodologia de Orientação por 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 campo de classe

Metodologia de Orientação por Aspectos

- Conjuntos de junção são conjuntos de pontos de junção
- Conjuntos de junção reunem informações de contexto 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

Metodologia de Orientação por Aspectos

- Regras de junção 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

Metodologia de Orientação por Aspectos

- Recursos básicos de linguagens orientadas por aspectos:
  - definição de pontos de junção;
  - identificação pontos de junção;
  - interferência na execução de pontos de junção.

Metodologia de Orientação por Aspectos

- Aspectos encapsulam os recursos das linguagens orientadas por aspectos
- **Aspecto = unidades de implementação modular de requisitos transversais**
- Um aspecto contém definições de conjuntos e regras de junção

Metodologia de Orientação por Aspectos

```
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());
    }
}
```

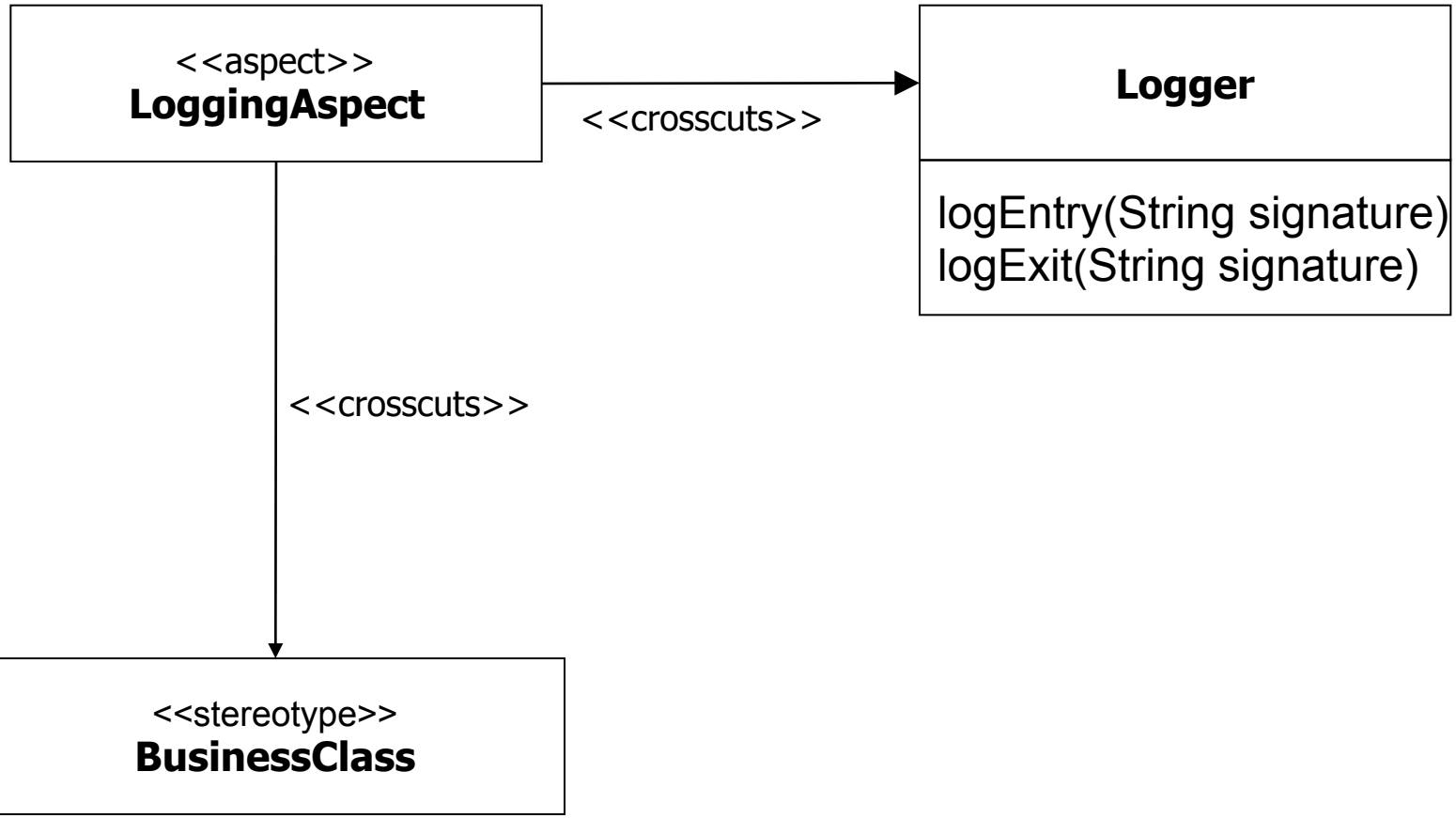
Metodologia de Orientação por Aspectos

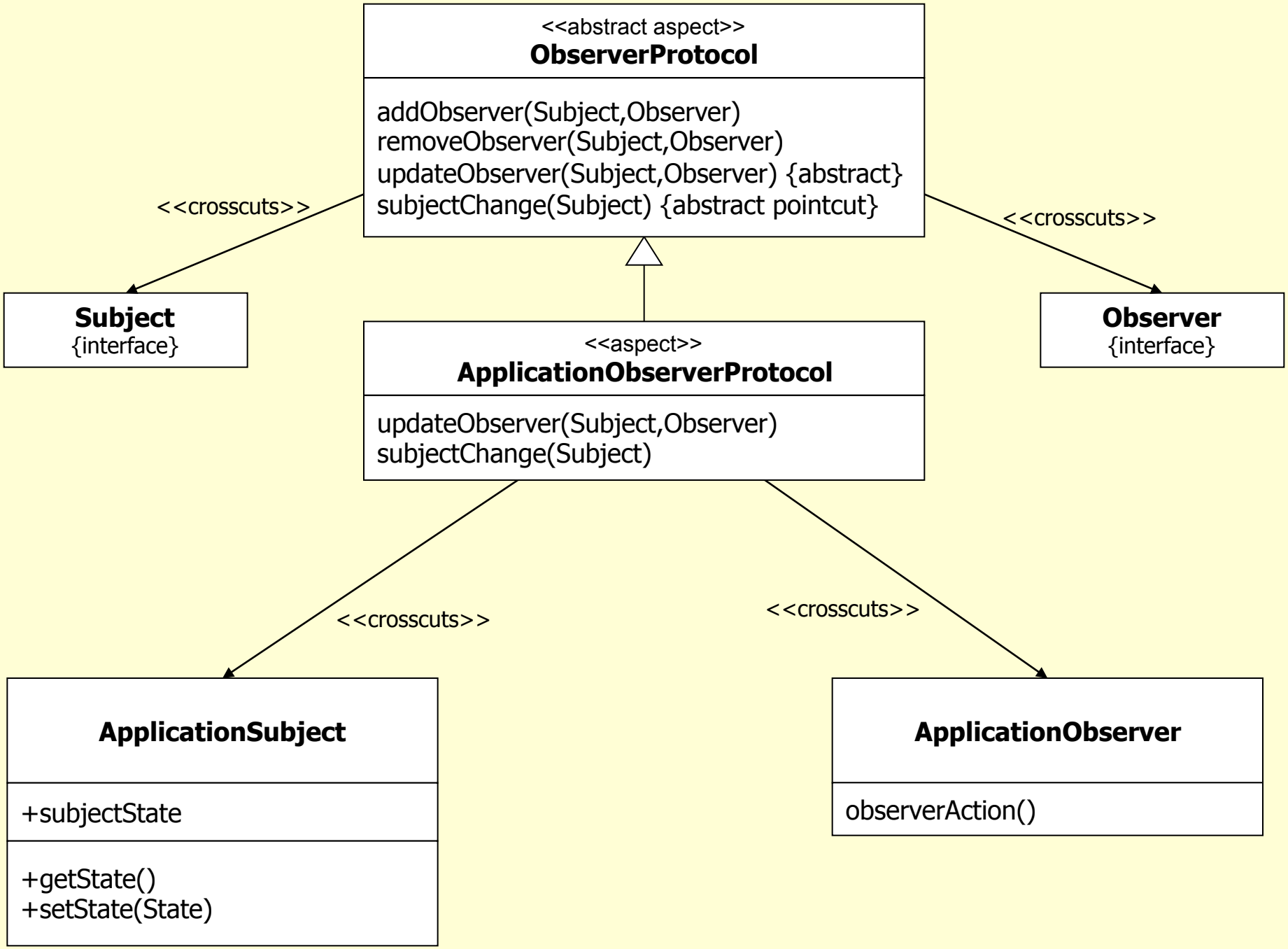
- O ***Weaver*** é o compilador de aspectos
- Objetivo: realizar o processo de costura de código (*weaving*)
- O processo de costura entrelaça códigos de regras de junção com códigos dos pontos de junção.
- Costura = instrumentação das classes do programa

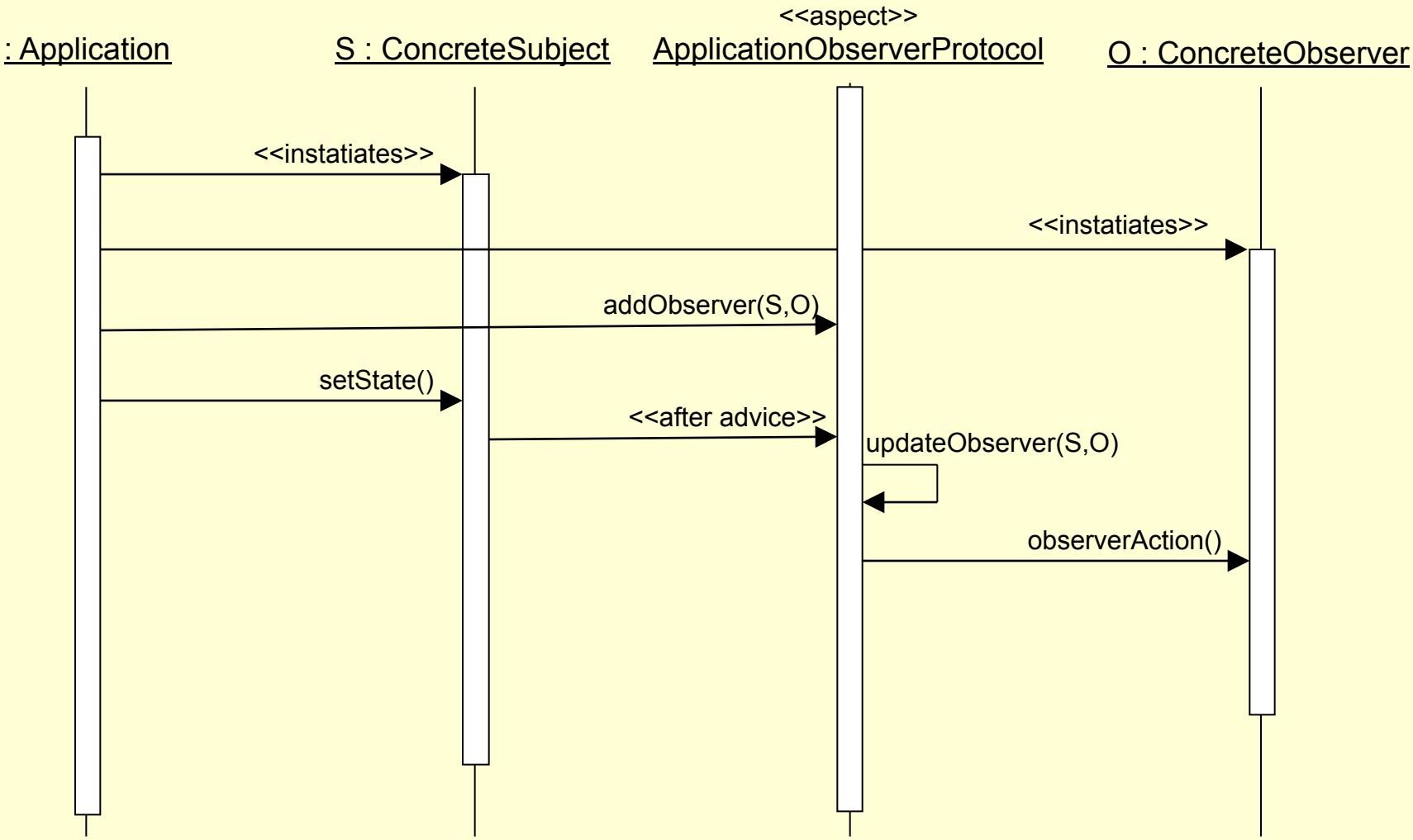
Metodologia de Orientação por Aspectos

- Para modelagem de aspectos, utilizam-se variações dos diagramas da UML
- Diagramas geralmente utilizados:
  - Classes
  - Seqüência
  - Colaboração

Metodologia de Orientação por Aspectos




<<abstract aspect>>
ObserverProtocol
addObserver(Subject,Observer)
removeObserver(Subject,Observer)
updateObserver(Subject,Observer) {abstract}
subjectChange(Subject) {abstract pointcut}
<<crosscuts>>
<<crosscuts>>
Subject
{interface}
Observer
{interface}
<<aspect>>
ApplicationObserverProtocol
updateObserver(Subject,Observer)
subjectChange(Subject)
<<crosscuts>>
<<crosscuts>>
ApplicationSubject
+subjectState
+getState()
+setState(State)
ApplicationObserver
observerAction()


: Application
S : ConcreteSubject
<<aspect>>
ApplicationObserverProtocol
O : ConcreteObserver
<<instatiates>>
<<instatiates>>
addObserver(S,O)
setState()
<<after advice>>
updateObserver(S,O)
observerAction()

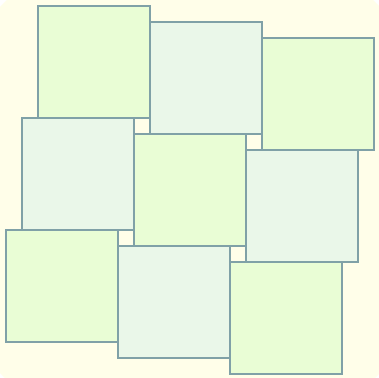
Metodologia de Orientação por Aspectos

- O desenvolvimento orientado por aspectos é bastante semelhante ao orientado por objetos
- Proposta metodológica:
  - definir e implementar os requisitos do sistema por meio de orientação por objetos;
  - definir regras de costura que façam o entrelaçamento dos requisitos em diferentes módulos.

Metodologia de Orientação por Aspectos

- Para modelagem de aspectos, pode-se utilizar a própria linguagem UML.
- Dentre os diagramas da UML, os mais utilizados são os diagramas de classes, de seqüência e de colaboração.
- Poucos conceitos são introduzidos nestes diagramas.
- A orientação por aspectos, no nível de projeto do sistema, é bastante acessível aos desenvolvedores de sistemas de software.

A Linguagem AspectJ



A Linguagem AspectJ

- A linguagem AspectJ é uma extensão de Java para implementação de AOP
- Definida pelo criador de AOP, Gregor Kiczales

- Informações e ambientes de execução:
  - eclipse.org/aspectj
  - eclipse.org/ajdt

A Linguagem AspectJ

- Um **Aspecto** é uma unidade modular de implementação de requisitos transversais.
- Aspectos são definidos por declarações de aspectos, que são similares a declarações de classes em Java.
- Declarações de aspectos podem:
  - declarar conjuntos de junção e regras
  - incluir métodos e campos
  - estender classes ou outros aspectos.

A Linguagem AspectJ

- AspectJ suporta dois tipos de implementação de requisitos transversais:
  - **Transversalidade dinâmica**

    permite definir implementação adicional em pontos bem definidos do programa
  - **Transversalidade estática**

    afeta as assinaturas estáticas das classes e interfaces de um programa Java.

A Linguagem AspectJ

- **Pontos de junção** são posições bem definidas da execução de um programa
- Exemplo: chamadas de métodos ou execução de blocos de tratamento de exceções.
- Pontos de junção podem também possuir contexto associado.
- Exemplo: em uma chamada de método, o contexto contém objeto alvo da chamada e a lista de argumentos da chamada do método

A Linguagem AspectJ

- Um **conjunto de junção** é formado por um conjunto de pontos de junção e as informações de contexto destes pontos.
- AspectJ utiliza conjuntos de junção para
  - Especificar coleções de pontos de junção
  - Expôr contexto de pontos de junção para regras de junção
- Sintaxe para definição de conjuntos de junção:

  pointcut nome(arg1, ..., argn): definição;

A Linguagem AspectJ

- Tipos fundamentais de pontos de junção:
  - Chamadas de métodos e construtores
  - Execuções de métodos e contrutores
  - Acesso a campos de classe
  - Tratamento de exceções
  - Inicializações estáticas de classe

A Linguagem AspectJ

- Para chamadas de métodos e construtores, utiliza-se o operador **call**, com a sintaxe:

  call(assinatura)

- Exemplos:
  - **call**(**void** Point.setX(**int**))
  - **call**(Point.new(**int**,**int**))

  - **pointcut** callConstrutorPonto():

    call(**void** Point.new(**int**,**int**));

A Linguagem AspectJ

- Para execuções de métodos e construtores, utiliza-se o operador **execution**, com a sintaxe:

  execution(assinatura)

- Exemplos:
  - **execution**(**void** Point.setX(**int**))
  - **execution**(Point.new(**int**,**int**))

  - **pointcut** execConstrutorPonto():
    execution(**void** Point.new(**int**,**int**));

A Linguagem AspectJ

- Diferenças entre pontos de junção de chamadas e de execuções:
  - o contexto no qual estão inseridos
  - o local onde o código do aspecto é costurado

A Linguagem AspectJ

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

```
pointcut execF() :
        execution(void MyClass.f(int));
```

A Linguagem AspectJ

```
class CallerClass {
    public static void g(...) {
        CHAMADA
        ... MyClass.f(10); ...
    }
    ...
}
```

```
pointcut callF() :
        call(void MyClass.f(int));
```

A Linguagem AspectJ

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

- Exemplos:
  - **get**(**int** Point.x)

  - **pointcut** alteraX():
    set(**int** Point.x);

A Linguagem AspectJ

- Para tratamento de exceção, utiliza-se o operador
  - **handler**(classe-exceção)

- Exemplos:
  - **handler**(NumberFormatException)

  - **pointcut** trataExcecaoIO():
    handler(IOException);

A Linguagem AspectJ

- Para execução do bloco **static** de classes, utiliza-se o operador
  - **staticinitialization**(nome-classe)

- Exemplo
  - **staticinitialization**(MyClass)

A Linguagem AspectJ

- Modificadores
  - Operações em conjuntos: **||** **&&** **!**
  - Símbolos curingas: ***** **+** **..**
  - Visibilidade, **static**

A Linguagem AspectJ

```
pointcut loggableCalls():
        execution(public * *(..))
          && !execution(* Logger.*(..))

pointcut setPointField():
        call(void Point.setX(int))
          || call(void Point.setY(int))

pointcut setFigureField():
        call(void Figure+.set*(..));
```

A Linguagem AspectJ

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

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

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

A Linguagem AspectJ

- Definição baseada em contexto:
  - **this**(classe ou objeto)
  - **target**(classe ou objeto)
  - **args**(argumentos)

A Linguagem AspectJ

```
pointcut updatingFigure(Figure f):
   call(void Figure+.set*(..)) && target(f);

pointcut updatingFigure(Figure f):
   execution(void Figure+.set*(..)) && this(f);

pointcut updatingPoint(Point p, int val):
   execution(void Point.set*(int))
        && this(p) && args(val);
```

A Linguagem AspectJ

- Diferença entre **this** e **target**:
  - **this**(*obj*): *obj* é o objeto que originou a chamada do método onde ocorre o ponto de junção
  - **target**(*obj*): *obj* é o objeto receptor de chamada de método

```
pointcut movingPoint(Line l, Point p):
        call(void Point.moveBy(int,int))
                && this(l) && target(p);
```

A Linguagem AspectJ

- Diferença entre **this** e **within**:
  - **this**(*Classe*): *Classe* é o tipo dinâmico do objeto
  - **within**(*Classe*): *Classe* é o tipo estático do objeto

A Linguagem AspectJ

```
public abstract class Figure {
    public abstract void moveBy(int x, int y);
}

public class Point extends Figure {
    private int x, y;
    public void moveBy(int x, int y) {
        this.x += x;
        this.y += y;
    }
}
```

A Linguagem AspectJ

```
public class Line extends Figure {
    Point p1, p2;
    public void moveBy(int x, int y) {
        p1.moveBy(x,y);
        p2.moveBy(x,y);
    }
}
```

A Linguagem AspectJ

```
pointcut withinLine():
        call(void Point.moveBy(int,int))
                && within(Line);

pointcut thisLine():
        call(void Point.moveBy(int,int))
                && this(Line);
```

A Linguagem AspectJ

```
pointcut withinFigure():
        call(void Point.moveBy(int,int))
                && within(Figure);

pointcut thisFigure():
        call(void Point.moveBy(int,int))
                && this(Figure);
```

A Linguagem AspectJ

- Definição baseada no fluxo de execução:
  - **cflow**(*conjunto-junção*)

    todos os pontos de junção gerados pertencentes ao fluxo de execução de *conjunto-junção*
  - **cflowbelow**(conjunto-junção)

    todos os pontos de junção pertencentes ao fluxo de execução de *conjunto-junção, sem considerar o próprio ponto de junção originador*

A Linguagem AspectJ

```
public class ClientClass {
    public void someMethod() {
        ...
        Line L = new Line(
                        new Point(10,6),
                        new Point(30,5));
        ...
        L.moveBy(10,16);
        ...
    }
}

cflow(execution(void ClientClass.someMethod()))
```

A Linguagem AspectJ

- Condicionais:
  - **if** (condição)

```
pointcut invalidSetX(int x):
    call(void Point.setX(int))
      && args(x) && if (x < 0);
```

A Linguagem AspectJ

- A linguagem AspectJ permite examinar informações nos pontos de execução de algum ponto de junção.
- Cada corpo de regra de junção possui um objeto especial, *thisJoinPoint*, que contém informações sobre o ponto de junção.

A Linguagem AspectJ

- Operações possíveis de *thisJoinPoint*:
  - getArgs()
  - getKind()
  - getSignature()
  - getStaticPart()
  - getTarget()
  - getThis()

A Linguagem AspectJ

- Regras de junção definem as regras de costura de um programa orientado por aspectos.

- Regras de junção definem ações a serem executadas antes, depois ou ao redor de pontos de junção.

A Linguagem AspectJ

- Sintaxe:

```
before (argumentos) : conjunto-junção {
    Código a ser costurado
}

after (argumentos) : conjunto-junção {
    Código a ser costurado
}
```

A Linguagem AspectJ

```
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());
    }
}
```

Todos os métodos públicos

Métodos da classe Logger

Métodos registrados

Entrada de método

Saída de método

A Linguagem AspectJ

```
public aspect UpdatingAspect {

    pointcut updatingFigure(Figure f):
        call(void Figure+.set*(..)) && target(f);

    after(Figure f): updatingFigure(f) {
        f.refresh();
    }

}
```

Atualizações de Figuras

Chamada do método de atualização após a modificação da figura

A Linguagem AspectJ

- Regras **after** podem ser executadas após retorno ou lançamento de exceção

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

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

A Linguagem AspectJ

```
pointcut gettingX() :
        execution(int Point.getX());

after() returning (int val) : gettingX() {
        System.out.println("Value of x: " + val);
}
```

A Linguagem AspectJ

- Sintaxe da regra *around*:

```
tipo around (argumentos) : conjunto-junção {
    Código a ser costurado
}
```

- Substitui o ponto de junção pelo código a ser costurado
- O código do ponto de junção pode ser executado por meio da construção **proceed**

A Linguagem AspectJ

```
void around(Point p, int val) :
        call(void Point.set*(int))
                && target(p)
                && args(val)
{
        if (val >= 0 && val < 100)
            proceed(p, val);
        else
            throw new IllegalArgumentException();
}
```

Tipo da regra

Chamadas capturadas

Continua execução

Lança uma exceção

A Linguagem AspectJ

```
void around(Point p, int val) :
        call(void Point.set*(int))
                && target(p)
                && args(val)
{
        if (val >= 0 && val < 100)
            proceed(p, val);
        else
            proceed(p, 0);
}
```

Ou continua
com outro valor

A Linguagem AspectJ

```
pointcut invalidPointConstruction(int x, int y):
    call(Point.new(int,int))
        && args(x,y) && (if(x < 0) || if(y < 0));

Point around(int x, int y):
        invalidPointConstruction(x,y)
{
    int xa = Math.abs(x), ya = Math.abs(y);
    return proceed(xa,ya);
}
```

Valores inválidos

Tipo de retorno

Conjunto de junção

Chama o construtor com valores corrigidos

A Linguagem AspectJ

- Algumas implementações de aspectos necessitam de recursos para alterar:
  - o comportamento das classes em tempo de execução
  - estrutura estática das classes.
- Exemplo: padrões de projeto

A Linguagem AspectJ

- A transversalidade dinâmica permite modificar o comportamento da execução do programa.
- A transversalidade estática permite redefinir a estrutura estática dos tipos e o seu comportamento em tempo de execução.

A Linguagem AspectJ

- 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.

A Linguagem AspectJ

- O mecanismo de introdução de AspectJ permite a inclusão de campos e métodos em classes e interfaces.
- É possível também adicionar métodos não-abstratos em interfaces, definindo comportamento *default*.

A Linguagem AspectJ

```
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

public aspect MouseListenerAspect {

    public void MouseListener.mouseClicked(MouseEvent e) { }
    public void MouseListener.mouseEntered(MouseEvent e) { }
    public void MouseListener.mouseExited(MouseEvent e) { }
    public void MouseListener.mousePressed(MouseEvent e) { }
    public void MouseListener.mouseReleased(MouseEvent e) { }

}
```

Métodos “introduzidos” na interface MouseListener

A Linguagem AspectJ

```
public interface Nameable {
    public void setName(String name);
    public String getName();
}

public interface Identifiable {
    public long getId();
    public void setId(long id);
}
```

A Linguagem AspectJ

```
public class Entity
        implements  Nameable,
                    Identifiable
{
    private String name;
    public void setName(String name) {
        this.name = name;
    }
    public String getName() { return name; }
    private long id;
    public void setId(long id) {
        this.id = id;
    }
    public long getId() { return id; }
}
```

A Linguagem AspectJ

- Para reutilizar as implementações:
  - A classe Entity deve estender uma classe que forneça as implementações default
  - Utilizar composição: restringe o uso de polimorfismo
- Caso contrário, haverá duplicidade de código

A Linguagem AspectJ

```
public interface Nameable {
    public void setName(String name);
    public String getName();
    static aspect DefaultImplementation {
        private String Nameable.name;
        public void Nameable.setName(String name) {
            this.name = name;
        }
        public String Nameable.getName() {
            return name;
        }
    }
}
```

A Linguagem AspectJ

```
public interface Identifiable {
    public void setId(long id);
    public long getId();
    static aspect DefaultImplementation {
        private long Identifiable.id;
        public void Identifiable.setId(long id) {
            this.id = id;
        }
        public long Identifiable.getId() {
            return id;
        }
    }
}
```

A Linguagem AspectJ

```
public class Entity
        implements      Nameable,
                        Identifiable
{
    ... Outros membros
}
```

Simulação de Herança Múltipla em Java

A Linguagem AspectJ

- Em AspectJ, é possível definir superclasses e implementação de interfaces para classes e interfaces já existentes, desde que isto não viole as regras de herança de Java.

  declare parents:

  tipo-filho **implements** lista-interfaces;

  declare parents:

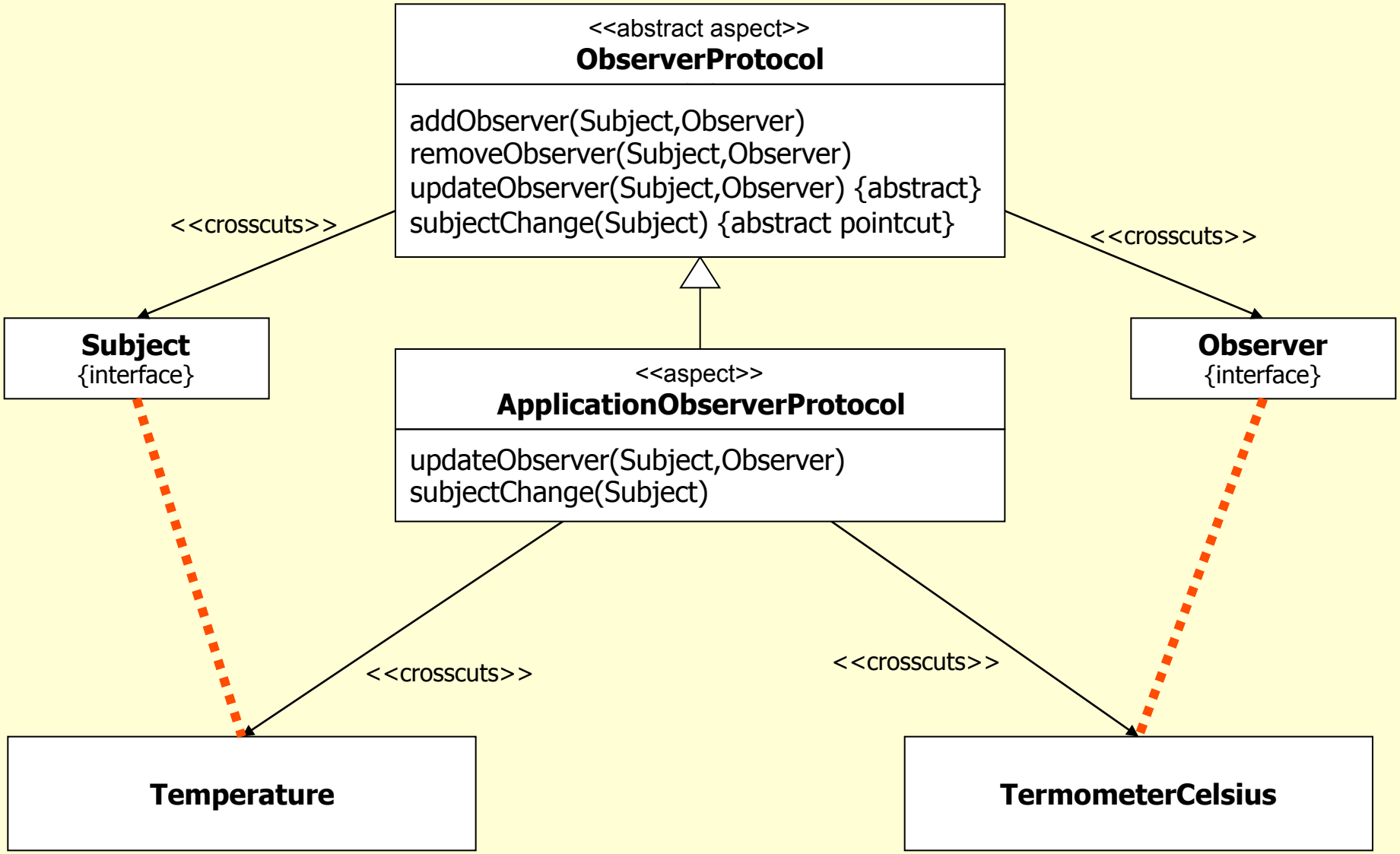
  tipo-filho **extends** classe-ou-lista-interfaces;

- Conseqüência: desacoplamento na hierarquia de classes dos requisitos funcionais e transversais

A Linguagem AspectJ

- Exemplos:

```
declare parents:
    Temperature extends Subject;
declare parents:
    TermometerCelsius implements Observer;
```



Desacoplamento dos requisitos funcionais e não-funcionais

A Linguagem AspectJ

- Em AspectJ, é possível declarar erros e advertências de compilação.
- Este mecanismo é similar às diretivas #error e #warning da linguagem C.

- Sintaxe:

  declare error: conjunto-junção :
  mensagem-erro;

  declare warning: conjunto-junção :
  mensagem-advertência;

A Linguagem AspectJ

- Exemplo: impedir o acesso direto aos campos de uma classe, mesmo dentro da classe

declare error :

Conjunto de junção

```
set(int Point.*)
&& !withincode(void Point.set*(int)) :
"Use setter method, even inside Point class.";
```

Mensagem de erro

A Linguagem AspectJ

- Em Java, todas as exceções devem ser declaradas na assinatura dos métodos ou capturadas e tratadas.
- Vantagem: lembra o desenvolvedor de tratar condições de erro no programa.
- Desvantagem: se tivermos certeza de que uma exceção nunca ocorre, o código de tratamento é inútil e polui a implementação.

A Linguagem AspectJ

- Exemplo: método escreve informações em um arquivo temporário e, logo em seguida, abra este arquivo para leitura.
- A exceção **FileNotFoundException** deve ser tratada.
- Entretanto, o arquivo existe, pois acabou de ser criado pelo próprio programa.

A Linguagem AspectJ

- Em AspectJ, é possível silenciar exceções de forma seletiva e relançá-las como exceções não checadas.
- Sintaxe:

  declare soft: tipo-exceção :
  conjunto-de-junção;

- Efeito na compilação:
  - enfraquece a exceção para o conjunto de junção especificado;
  - o compilador Java não exija o seu tratamento.

A Linguagem AspectJ

- Exemplo:

Exceção

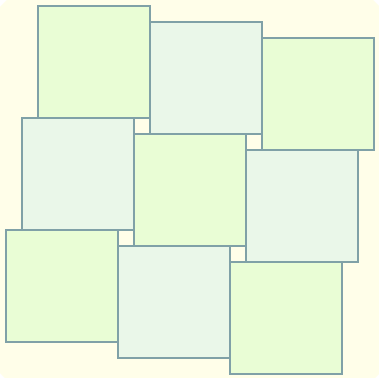
```
declare soft: FileNotFoundException :
  withincode(void SomeClass.someMethod());
```

Conjunto de junção

A Linguagem AspectJ

- A linguagem AspectJ é uma extensão de Java para implementação de AOP
- Em AspectJ é possível:
  - Alterar o comportamento da execução de programas
  - Alterar as assinaturas de classes e interfaces de um programa

Exemplos de Aplicação de Orientação por Aspectos



Exemplo I: Registros de Operações

- Objetivo: registrar as operações mais importantes da execução de um programa
- Exemplos:
  - Chamadas e retorno de métodos
  - Lançamento e tratamento de exceções
  - Alteração de campos de classe
  - Chamadas e retorno de construtores
  - Chamadas de métodos de classes específicas

Exemplo I: Registros de Operações

- Organização:
  - Classe **Logger**

    implementação separada da funcionalidade de geração de registros

  - Aspecto **LoggingAspect**

    responsável pela costura da funcionalidade de geração de registros pelas classes do programa

Exemplo I: Registros de Operações

```
public class Logger {
    public static void logEntry(String signature) {
        System.out.println("Entering " + signature);
    }
    public static void logNormalExit(String signature) {
        System.out.println("Normal return of " + signature);
    }
    public static void logExitWithException(
                    String signature,
                    Exception exc) {
        System.out.println("Return of " + signature
                                + " with exception " + exc);
    }
    public static void logExceptionHandling(Exception exc) {
        System.out.println("Handling exception " + exc);
    }
}
```

Entrada de método

Saída normal de método

Saída de método com exceção

Tratamento de exceção

Exemplo I: Registros de Operações

```
public aspect LoggingAspect {

    pointcut loggableCalls() : ... ;

    before(): loggableCalls() { ... }

    after() returning: loggableCalls() { ... }
    after() throwing (Exception exc) : loggableCalls() { ... }

    pointcut excHandling(Exception exc): ... ;

    before(Exception exc): excHandling(exc) { ... }

}
```

Conjunto de junção para método

Regra para entrada de método

Regras para saída de método

Conjunto de junção de tratamento de exceções

Regra para tratamento de exceções

Exemplo I: Registros de Operações

```
pointcut publicMethods():
        execution(public * *(..));
```

Todos os métodos públicos

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

Todos os métodos da classe Logger

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

Métodos públicos
não pertencentes à classe Logger

Exemplo I: Registros de Operações

```
before() : loggableCalls() {
    Signature sig = thisJoinPoint.getSignature();
    Logger.logEntry(sig.toString());
}
```

[illegible] registrados

Exemplo I: Registros de Operações

```
after() returning: loggableCalls() {
    Signature sig = thisJoinPoint.getSignature();
    Logger.logNormalExit(sig.toString());
}
```

Realiza o registro de métodos registrados

```
after() throwing (Exception e) : loggableCalls() {
    Signature sig = thisJoinPoint.getSignature();
    Logger.logExitWithException(sig.toString(), e);
}
```

Realiza o registro de métodos registrados

Exemplo I: Registros de Operações

```
pointcut exceptionHandling(Exception exc):
    handler(Exception+) && args(exc);
```

Captura de todas as exceções

Exposição do objeto exceção tratado

```
before(Exception exc):
        exceptionHandling(exc) {
    Logger.logExceptionHandling(exc);
}
```

Antes do tratamento da exceção

Realiza o registro da operação

Exemplo I: Registros de Operações

```
public class MyClass {
    public void aMethod(int x) {
        String sig = "MyClass.aMethod(int)";
        Logger.logEntry(sig);
        if (x < 0) {
            IllegalArgumentException exc;
            exc = new IllegalArgumentException();
            Logger.logExitWithException(sig, exc);
            throw exc;
        }
        REQUISITO FUNCIONAL DO MÉTODO
        Logger.logNormalExit(sig);
    }
}
```

Exemplo I: Registros de Operações

```
public class MyClass {
    public void aMethod(int x) {


        if (x < 0)
            throw new IllegalArgumentException();



         REQUISITO FUNCIONAL DO MÉTODO


    }
}
```

Exemplo I: Registros de Operações

```
public class TesteLogger {
    public static void main(String[] args) {
        String sig = "TesteLogger.main(String[])";
        Logger.logEntry(sig);
        try {
            MyClass myObject = new MyClass();
            myObject.aMethod(5);
            myObject.aMethod(-1);
        } catch (IllegalArgumentException exc) {
            Logger.logExceptionHandling(exc);
            TRATAMENTO DA EXCEÇÃO
        }
        Logger.logNormalExit(sig);
    }
}
```

Exemplo I: Registros de Operações

```
public class TesteLogger {
    public static void main(String[] args) {


        try {
            MyClass myObject = new MyClass();
            myObject.aMethod(5);
            myObject.aMethod(-1);
        } catch (IllegalArgumentException exc) {

            TRATAMENTO DA EXCEÇÃO
        }

    }
}
```

Exemplo I: Registros de Operações

- O *weaver* adiciona chamadas de métodos de registro nos pontos indicados no aspecto

- Todo código de registro de operação pode ser retirado dos locais onde é intruso

- Modularização da política de geração de registros de operação do programa.

Exemplo II: Padrão Singleton

- Garante que uma classe possui uma única instância e provê um ponto global de acesso a sua instância
- Aplicações:
  - Quando for necessária somente uma instância de uma classe no sistema
    - Exemplos: fábrica de sessões de BD, gerenciador de interface gráfica, gerenciador de recursos
  - Quando precisarmos de acessar facilmente a instância única
- Considere que as subclasses de uma classe singleton não precise necessariamente respeitar a restrição de instância única

Exemplo II: Padrão Singleton

```
public class PrinterSingleton {
    private static int objectsSoFar = 0;
    private int id;
    public void print() {
        System.out.println("My ID is " + id);
    }
    public PrinterSingleton() {
        id = ++objectsSoFar;
    }
}
```

Construtor não pode ser público!

Exemplo II: Padrão Singleton

```
public class PrinterSingleton {
    private static int objectsSoFar = 0;
    private int id;
    public void print() {
        System.out.println("My ID is " + id);
    }
    private PrinterSingleton() {
        id = ++objectsSoFar;
    }
}
```

Não é possível criar subclasses!

Exemplo II: Padrão Singleton

```
public class PrinterSingleton {
    private static int objectsSoFar = 0;
    private int id;
    public void print() {
        System.out.println("My ID is " + id);
    }
    protected PrinterSingleton() {
        id = ++objectsSoFar;
    }
}
```

Classes no mesmo pacote podem criar múltiplas instâncias!

Exemplo II: Padrão Singleton

```
public class PrinterSingleton {
    ... // demais membros da classe
    private static PrinterSingleton theinstance;
    public static PrinterSingleton instance() {
        if (theinstance == null)
            theinstance = new PrinterSingleton();
        return theinstance;
    }
}
```

Método que retorna a instância da classe

Exemplo II: Padrão Singleton

```
public class PrinterSingleton {
    ... // demais membros da classe
    private static PrinterSingleton theinstance;
    public static PrinterSingleton instance() {
        if (theinstance == null)
            theinstance = new PrinterSingleton();
        return theinstance;
    }
}
```

Retorna instância criada ou não existir

Exemplo II: Padrão Singleton

```
public class PrinterSingleton {
    private static int objectsSoFar = 0;
    private int id;
    public void print() {
        System.out.println("My ID is " + id);
    }
    protected PrinterSingleton() {
        id = ++objectsSoFar;
    }
    private static PrinterSingleton uniqueInstance = null;
    public static PrinterSingleton instance() {
        if (uniqueInstance == null) {
            uniqueInstance = new PrinterSingleton();
        }
        return uniqueInstance;
    }
}
```

INTRUSÃO

Exemplo II: Padrão Singleton

- Problemas:
  - Violação do padrão dentro do pacote
  - Intrusão do código do padrão na classe
  - Código de definição do padrão não pode ser reutilizado
  - Como alterar a classe de singleton para não-singleton ou vice-versa?

Exemplo II: Padrão Singleton

```
public abstract aspect SingletonProtocol {

    public interface Singleton { }

    pointcut singletonCreation() : ... ;

    Object around() : singletonCreation() {
        ...
    }

}
```

Utilizada para garantir consistência de tipos

Conjunto de junção de criação de instância

Regra de junção de garantia do protocolo

Exemplo II: Padrão Singleton

```
abstract pointcut protectionExclusions();
```

Utilizado para definir pontos que podem violar o padrão

```
pointcut singletonCreation() :

        call(Singleton+.new (..))

                && ! protectionExclusions();
```

Construtor de subclasses de Singleton que não estejam protegidas do protocolo

Exemplo II: Padrão Singleton

```
private Hashtable singletons = new Hashtable();
```

Armazena as instâncias únicas de cada classe

```
Object around() : singletonCreation() {
    Signature sig = thisJoinPoint.getSignature();
    Class singleton = sig.getDeclaringType();
    if (singletons.get(singleton) == null)
        singletons.put(singleton, proceed() );
    return singletons.get(singleton);
}
```

Obtém a classe do objeto criado e armazena na tabela

Exemplo II: Padrão Singleton

```
public class Printer {
    private static int objectsSoFar = 0;
    private int id;
    public void print() {
        System.out.println("My ID is " + id);
    }
    public PrinterSingleton() {
        id = ++objectsSoFar;
    }
}
```

Construtor pode ser público!

Exemplo II: Padrão Singleton

```
public class PrinterSubclass extends Printer {
    public PrinterSubclass() {
        super();
    }
}
```

Exemplo II: Padrão Singleton

```
public aspect SingletonInstance
        extends SingletonProtocol {

    declare parents:
            Printer implements Singleton;

    public pointcut protectionExclusions():
            call((PrinterSubclass +).new (..));

}
```

Suficiente para garantir que Printer é singleton

Garante que PrinterSubclass pode ser instanciada

Exemplo II: Padrão Singleton

- Implementação não-intrusiva
- Implementação modularizada do padrão
- Código do padrão pode ser completamente reutilizado em outras implementações
- Permite que subclasses sejam instanciadas
- Facilita trocar de singleton para não-singleton ou vice-versa

Exemplo III: Padrão Observador

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

        }
}
public class TermometerCelsius {
        public void update(double value) {

                System.out.println("Celsius: " + value);
        }
}
public class TermometerFahrenheit {
        public void update(double value) {

                System.out.println("Fahrenheit: " + value);
        }
}
```

Versão sem Padrão Observador

Exemplo III: Padrão Observador

```
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);
        }
}
public class TermometerFahrenheit implements Observer {
        public void update( Subject s ) {
                double value = 1.8 * ((Temperature) s).getValue() + 32;
                System.out.println("Fahrenheit: " + value);
        }
}
```

Versão com Padrão Observador

Espalhamento

Exemplo III: Padrão Observador

- O padrão “desaparece no código”
- A sua implementação não é modular

- Adicionar ou remover o padrão no sistema é geralmente uma substituição invasiva e de difícil retrocesso

- Especificação do padrão é reusável
- Implementação do padrão não pode ser totalmente modularizada
- **Classes participantes do padrão não são totalmente reusáveis**

Exemplo III: Padrão Observador

```
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;
    }
}
```

Exemplo III: Padrão Observador

```
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 {
    ...
}
```

Exemplo III: Padrão Observador

```
public abstract aspect ObserverProtocol {

    public interface Subject { }
    public interface Observer { }

    public void addObserver(Subject s, Observer o) { … }
    public void removeObserver(Subject s, Observer o) { … }

    public abstract pointcut subjectChange(Subject s);

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

    after(Subject s): subjectChange(s) { … }

}
```

Garante consistência de tipos

Inclusão e exclusão de observadores

Mudanças de estado do objeto alvo

Notificação e atualização de observadores

Regra de costura do padrão

Exemplo III: Padrão Observador

```
private HashMap soHash = new WeakHashMap();
```

Armazena listas de observadores dos alvos

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

Obtém a lista de observadores de s

Cria a lista de observadores se não existir

Retorna a lista de observadores

Exemplo III: Padrão Observador

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

Adiciona observador à lista de s

Atualiza o observador

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

Remove o observador da lista de s

Exemplo III: Padrão Observador

```
abstract public pointcut subjectChange(Subject s);

abstract public 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);
    }
}
```

Remove o observador da lista de s

Remove o observador da lista de s

Exemplo III: Padrão Observador

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

Incluir as classes como participantes do padrão

Define o que é alteração do alvo

Atualização dos observadores

Exemplo III: Padrão Observador

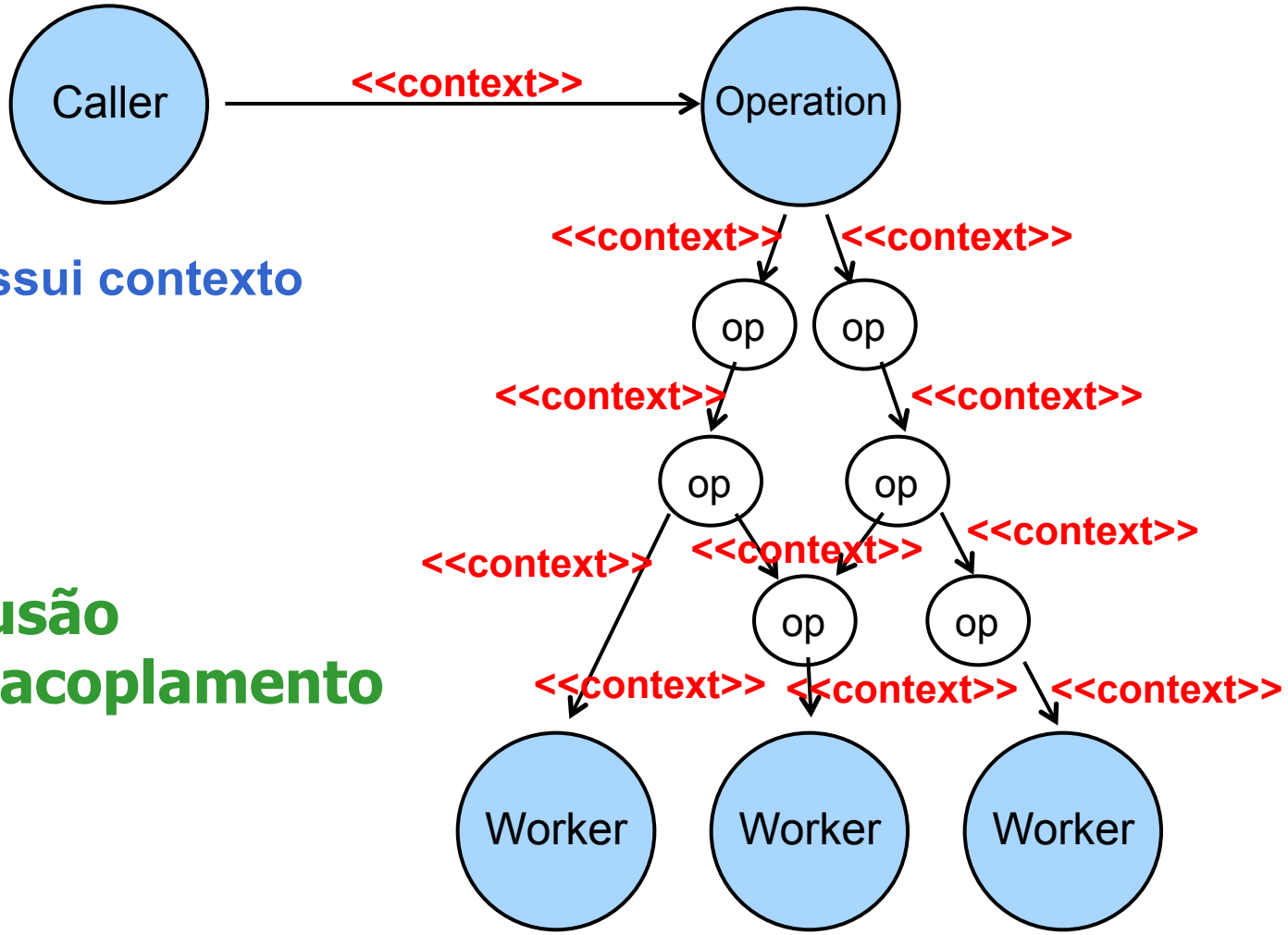
```
Temperature t = new Temperature();
Termometer term1 = new TermometerCelsius();
Termometer term2 = new TermometerFahrenheit();

TemperatureObservation.aspectOf().addObserver(t,term1);
TemperatureObservation.aspectOf().addObserver(t,term2);
```

Exemplo III: Padrão Observador

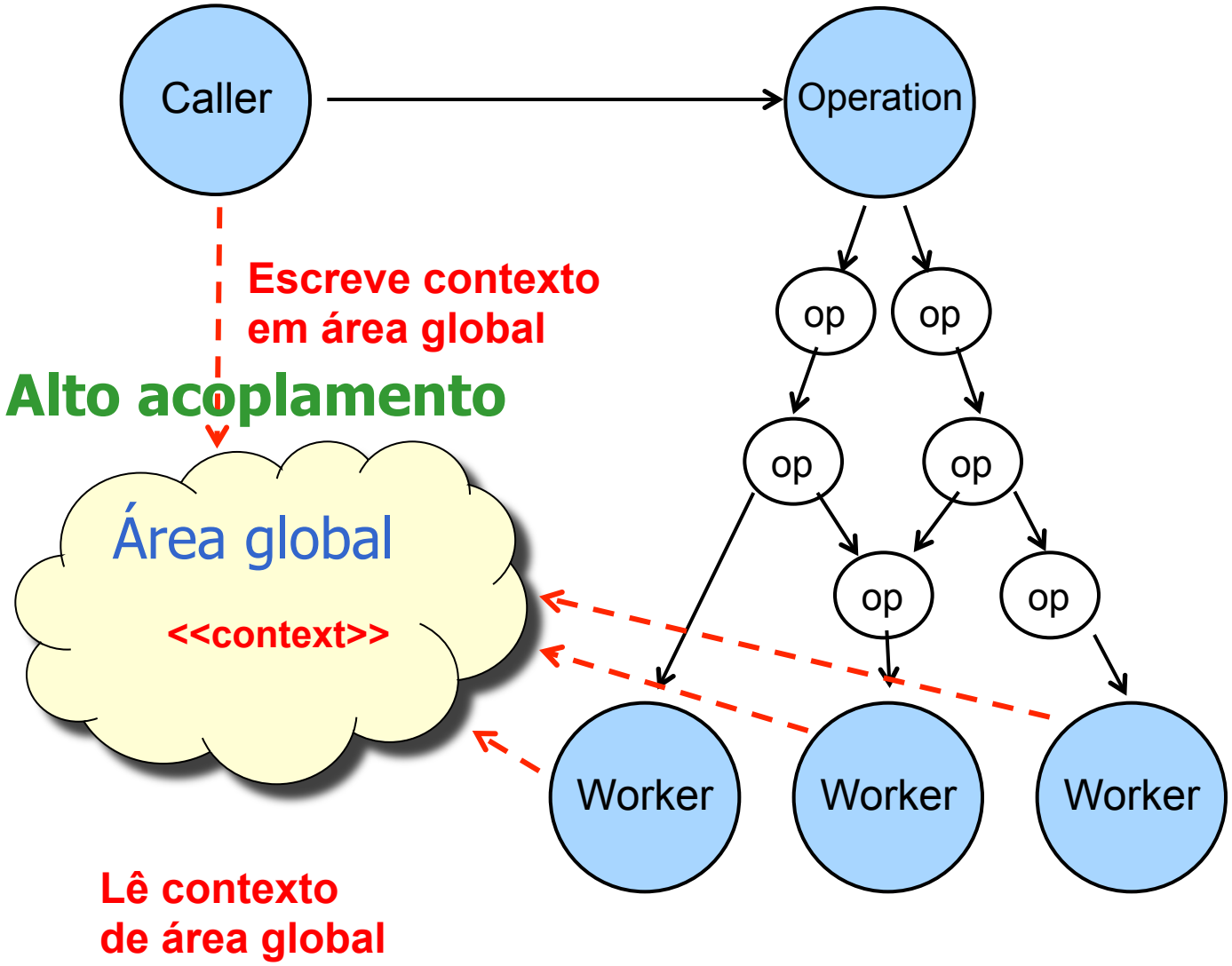
- Implementação não-intrusiva do padrão
- Padrão modularizado
- Implementação do padrão é reutilizável

Exemplo IV: Padrão *Wormhole*

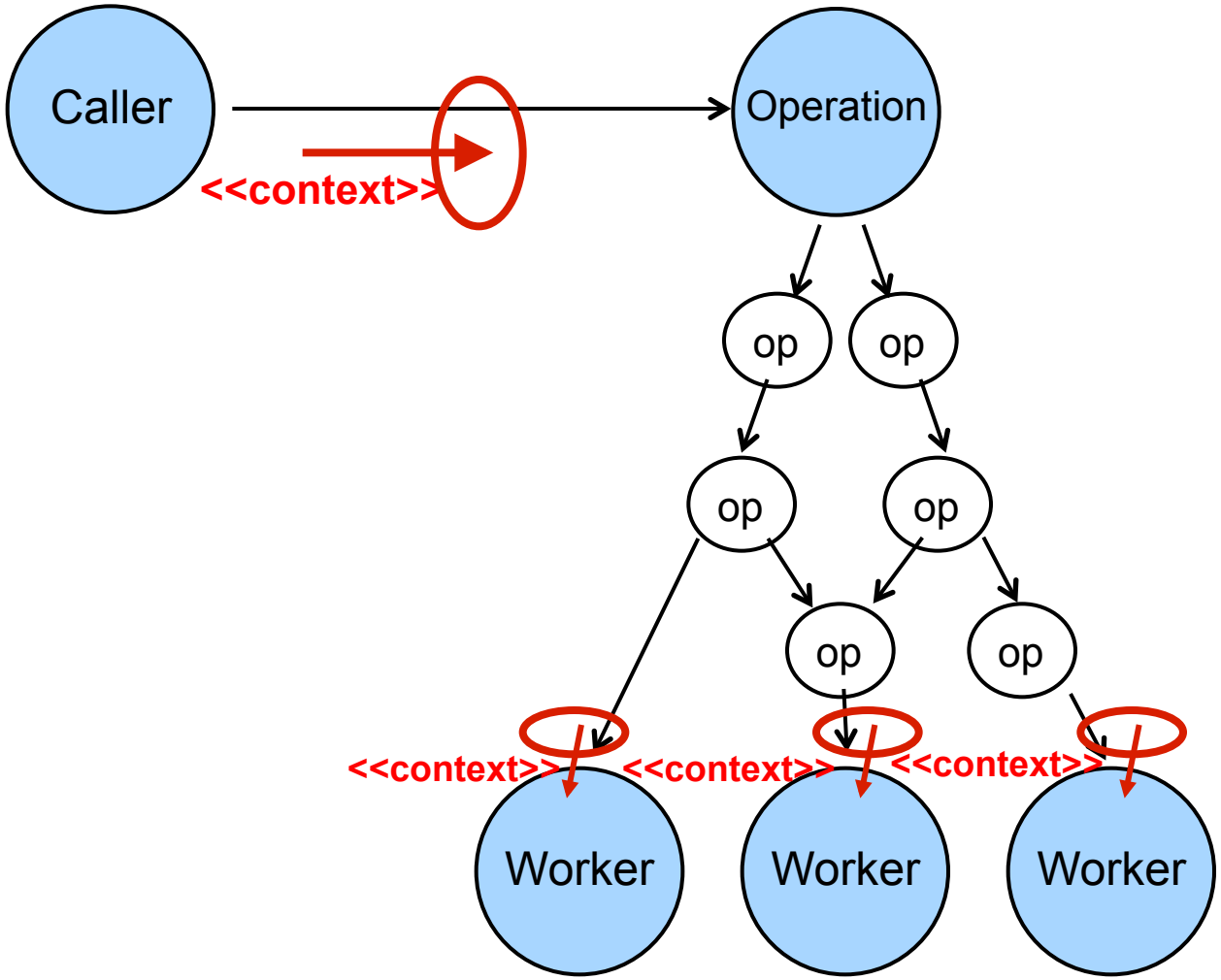


Intrusão
Alto acoplamento

Exemplo IV: Padrão *Wormhole*



Exemplo IV: Padrão *Wormhole*



Exemplo IV: Padrão *Wormhole*

```
public aspect WormholePattern {

    pointcut invocations(Caller c):
        this(c) && call(* Operation+.*(..));

    pointcut workPoints(Worker w):
        target(w) && call(void Worker.doWork());

    pointcut perCallerWork(Caller c, Worker w):
        cflow(invocations(c)) && workPoints(w);

    before (Caller c, Worker w): perCallerWork(c, w) {
        w.propagateCallerContext(c.getContext());
    }

}
```

Pontos de chamada das operações

Pontos de chamada da operação principal

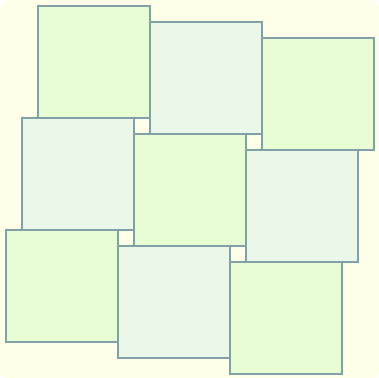
Pontos de trabalho disparados por Caller

Propagação do contexto

Exemplo IV: Padrão *Wormhole*

- Padrão *wormhole* faz a propagação direta de informações de contexto com baixo acoplamento
- Aspecto **WormholePattern** funciona como mediador entre as classes
- Toda alteração pode ser feita diretamente no aspecto
- Protocolo abstrato também é possível

Conclusões



Conclusões

- Objetivo do desenvolvimento orientado por aspectos: permitir melhor modularização de requisitos transversais de sistemas.
- A linguagem AspectJ é uma linguagem que permite a implementação orientada por aspectos em Java.
- Fontes importantes:
  - http://aosd.net
  - http://eclipse.org/aspectj

Conclusões

- AOP não é um substituto da POO
- Os requisitos funcionais de um sistema continuarão a ser implementados por meio da POO
- AOP adiciona novos conceitos à POO para:
  - facilitar a implementação de requisitos transversais
  - retirar grande parte da atenção dada a tais requisitos nas fases iniciais de desenvolvimento
- AOP permite ao desenvolvedor:
  - Concentrar-se na implementação dos módulos de requisitos funcionais do sistema
  - Implementar separadamente a distribuição dos requisitos transversais aos módulos.

Conclusões

- AOP permite definir claramente as responsabilidades de módulos individuais
- Melhoria do nível de modularização de sistemas:
  - Baixo acoplamento
  - Alta coesão modular
- Conseqüências:
  - Melhora o processo de manutenção
  - Aumenta o grau de reúso
  - Facilita o processo de evolução de sistemas
    - Novos aspectos não alteram classes já existentes
    - Novas classes não alteram aspectos já existentes

Conclusões

- Pontos negativos:
  - É possível quebrar o encapsulamento de módulos
  - Existem limitações na integração de módulos

Conclusões

- Orientação por aspectos ainda não resolve definitivamente a importante questão da modularidade de sistemas computacionais complexos

- Entretanto, é uma área de pesquisa promissora e representa um dos mais importantes avanços desde o advento da orientação por objetos.