

MBA - Desenvolvimento
de Soluções Corporativas
Java (SOA)

AOP
Programação
Orientada a Aspectos

Prof. Msc. Marcos Macedo
marcos@synapsystem.com.br

Fevereiro/2014

Aspect J

Aspectos

Aspects são os artefatos do AspectJ para criação dos componentes de natureza crosscutting, é como se fosse uma classe Java porém, para necessidades globais.

Aspectos tem a seguinte estrutura básica:

```
package <<Nome do pacote se existir>>;

import <<Import java/AspectJ se existir>>;

public aspect <<Nome do aspecto padrão java >> {

    // Pointcuts

    // Advices

}
```

Pointcuts

Pointcuts são alguns join points de um programa que são capturados pelo mecanismo de AOP do AspectJ. No AspectJ existem 11 possíveis join points, veja:

- Chamada de método
 - Execução de método
 - Chamada de construtor
 - Execução de construtor
 - Getter de uma propriedade
 - Setter de uma propriedade
 - Pre-inicialização
 - Inicialização
 - Inicialização estática
 - Handler
 - Advice de execução de join points
-

Pointcuts

Sintaxe básica

Um ponto de corte (*pointcut*) é uma construção sintática do AspectJ para se agrupar um conjunto de pontos de junção. Sua sintaxe básica é a seguinte (ilustrada com um exemplo):

```
public pointcut nome() : call (void Receita.print())
```

- A primeira parte é a declaração de restrição de acesso - nesse caso, *public*, mas pointcuts podem ser *private* ou *protected*.
- A palavra-chave *pointcut* denota que estamos declarando um ponto de corte.
- Todo pointcut tem um nome qualquer, e pode receber parâmetros - nesse caso, o pointcut não recebe parâmetros.
- Depois dos dois pontos (:) obrigatórios vem o tipo dos pontos de junção agrupados pelo pointcut - nesse caso temos um pointcut do tipo *call*, que indica uma chamada de método.
- Finalmente vem a assinatura do pointcut, uma especificação dos pontos de junção aos quais o pointcut se refere.

Caracteres especiais

Na descrição da assinatura, pode-se usar alguns caracteres especiais para incluir mais de um ponto de junção no pointcut. Os caracteres são:

| Caractere | Significado |
|-----------|---|
| * | Qualquer sequência de caracteres não contendo pontos |
| .. | Qualquer sequência de caracteres, inclusive contendo pontos |
| + | Qualquer subclasse de uma classe. |

Expressões lógicas

Um pointcut pode conter várias assinaturas ligadas através de operadores lógicos. Os operadores são:

| Operador | Significado | Exemplo | Interpretação do exemplo |
|----------|-------------|------------------------|--|
| ! | Negação | ! Receita | Qualquer classe exceto Receita |
| | "ou" lógico | Receita Ingrediente | Classe receita ou classe ingrediente |
| && | "e" lógico | Cloneable && Runnable | Classes que implementam ambas as interfaces Cloneable e Runnable |

Pointcuts

`pointcut <<nome>>(): <<expressao>;`

Exemplo:

```
package br.com.targettrust.aop.pointcuts;

public aspect PointcutSample {
    pointcut nomePointcut(): call (void metodoX());
}
```

Pointcuts

```
package br.com.targettrust.aop.pointcuts;

public aspect PointcutLogicalOperators {

    pointcut operadorAnd(): call ( String
br.com.targettrust.aop.java.domain.model.Pessoa.getNome() )
    &&
    call ( String
br.com.targettrust.aop.java.domain.model.PessoaFisica.getNome() );

    pointcut operadorOr(): call ( String
br.com.targettrust.aop.java.domain.model.Pessoa.getNome() )
    ||
    call ( String
br.com.targettrust.aop.java.domain.model.Pessoa.getEndereco() );

    pointcut operadorNot(): call ( ! String
br.com.targettrust.aop.java.domain.model.Pessoa.getNome() );

}
```

Pointcuts

Pointcuts por tipo de ponto de junção

Já vimos pointcuts do tipo *call*, que se refere a chamadas de métodos. Além desse, existem outros tipos, listados a seguir com suas sintaxes:

| Categoria de ponto de junção | Sintaxe do pointcut |
|----------------------------------|-------------------------------|
| Chamada de método ou construtor | call(AssinaturaDoMetodo) |
| Execução de método ou construtor | execution(AssinaturaDoMetodo) |
| Inicialização de classe | staticinitialization(Classe) |
| Leitura de dado de classe | get(AssinaturaDoCampo) |
| Escrita de dado de classe | set(AssinaturaDoCampo) |
| Tratamento de exceção | handler(Exceção) |

Pointcuts de fluxo de controle

Há dois tipos de pointcuts que se referem a um conjunto de pontos de junção que ocorrem em um **fluxo de controle**. Um fluxo de controle consiste de todos os comandos que são executados entre a entrada em um método e a saída deste, inclusive considerando comandos em outros métodos chamados pelo primeiro.

Os dois pointcuts são *cflow()* e *cflowbelow()*. A única diferença entre eles é que *cflow()* inclui entre os pontos de junção resultantes o próprio ponto de junção dado como parâmetro, enquanto *cflowbelow()* inclui apenas os pontos de junção que estão "abaixo" do ponto dado. A tabela abaixo mostra um exemplo de cada tipo:

| Ponto de corte | Descrição |
|--|--|
| <i>cflow(call (* Receita.print()))</i> | Todos os pontos de junção que ocorrem durante a chamada ao método <i>print</i> da classe <i>Receita</i> , incluindo a própria chamada. |
| <i>cflowbelow(execution(void Receita.print()))</i> | Todos os pontos de junção que ocorrem durante a execução do método <i>print</i> da classe <i>Receita</i> , NÃO incluindo a própria execução. |

Pointcuts baseados na estrutura léxica

Dois tipos de pointcuts levam em consideração trechos do código do programa, sem levar em consideração a execução. O primeiro é *within(Tipo)*. Esse pointcut inclui todos os pontos de junção dentro de uma classe ou aspecto passada como parâmetro. O segundo é *withincode(Método)*, que inclui todos os pontos de junção dentro de um método ou construtor.

| Ponto de corte | Descrição |
|--------------------------------------|---|
| <i>within(Receita+)</i> | Todos os pontos de junção que ocorrem dentro da classe <i>Receita</i> ou de qualquer de suas subclasses. |
| <i>withincode(* Receita.set*...)</i> | Todos os pontos de junção que ocorrem dentro de métodos da classe <i>Receita</i> cujos nomes começam com "set". |

Pointcuts

identifica código
dentro da classe ...

nome do
pointcut

```
pointcut facadeMethodsCall() :  
    within(HttpServletRequest) &&  
    call(* IFacade+.*(..));
```

identifica
chamadas de ...

qualquer
método

com quaisquer
argumentos

Declaração de Erros e Warnings

É possível gerar warnings e até mesmo erros com aspectos, isso se torna muito interessante por exemplo para evitar que desenvolvedores acessem métodos @Deprecated. Esse recurso pode ser utilizado para o reforço de contratos e até mesmo para evitar o acoplamentos de código.

```
package br.com.targettrust.aop.pointcuts.annotation.erroswarnings;

public aspect JoinPointMacthWithAnnotationErrosWarnings {

    declare warning :
    within(br.com.targettrust.aop.pointcuts.annotation.erroswarnings.java.*)
    && call(* faz*(..))
    : "Você não deve chamar metodos com o prefixo faz*";

    declare error :
    within(br.com.targettrust.aop.pointcuts.annotation.erroswarnings.java.*)
    && call(* deleteAll*(..))
    : "Você não deve deletar tudo !!!";

}
```

Advices - Before

```
public aspect SimpleAdvice {  
  
    before(int x):  
        call (* *.*.exec*(int)) && args(x) {  
            System.out.println("[SimpleAdvice]modificado: " + (x+1));  
        }  
}
```

```
before(Object x):  
    call(* *.*.save(Object)) &&  
    args(x)  
{  
    if (x==null)  
        throw new RuntimeException("Não pode salvar um objeto nulo!");  
}
```

Advices - After

```
public aspect AfterSimple {  
  
    after():  
        call(* *.save(Object)){  
            System.out.println("Simples execucao depois do metodo.");  
        }  
}
```

```
after()  
    returning(Object o):  
        call(public Object *.*.find()){  
            String x = (String)o;  
            x += "123";  
            System.out.println("retorno: " + x);  
        }
```

```
after()  
    throwing(Exception e):  
        call(public void *.*.delete()) {  
            System.out.println("Erro! Mensagem: " + e.getMessage());  
        }
```

Advices - Around

```
int around():  
call(public int *.*.soma(int,int)) {  
    System.out.println("Around advice modificando valores constante 1");  
    return 1;  
}
```

```
int around(int x,int y):  
call(public int *.*.soma2(int,int)) &&  
args(x,y) {  
    System.out.println("Around advice modificando valores dinamicos");  
    return proceed(x,y);  
}
```

ThisJoinPoint

```
package br.com.targettrust.aop.advice.thisJoinPoint.java;

public class ThisJoinPointTest {

    public static void execute(){};

    public static void main(String[] args) {
        execute();
    }
}
```

```
package br.com.targettrust.aop.advice.thisJoinPoint;

public aspect ThisJoinPoint {

    void around(): call(void *.*.execute()) &&
    within(br.com.targettrust.aop.advice.thisJoinPoint.java.*){
        System.out.println("Executando: " + thisJoinPoint.toString());
    }

}
```

Executando: call(void
br.com.targettrust.aop.advice.thisJoinPoint.java.ThisJoinPointTest.execute())

ThisJoinPoint

| Método | Descrição |
|----------------------------|---|
| getArgs() | Retorna os argumentos do método |
| getSignature() | Retorna a assinatura do método |
| getSourceLocation() | Fonte e linha do código Java |
| getClass() | Retorna a classe de implementação do JoinPoint |
| getKind() | Tipo de join point. Ex.: method-call |
| getTarget() | Retorna o objeto (Java) em execução |
| getStaticPart() | É um helper do join point que contém as informações que podem ser acessadas de forma estática. |
| getThis() | Semelhante ao target, porém sempre irá retornar o objeto que estiver relacionado ao this() de expressão pointcut. |

Quando estamos em um advice do tipo after ou before e existe uma expressão call(), é possível acessar informações do joint point passado quando ele existe, para isso usamos a variável **thisEnclosingJoinPointStaticPart**.

Aspectos Abstratos

```
package br.com.targettrust.aop.aspect;

public abstract aspect PaiBurro {
    protected String nome = "Aspecto Contador";
}
```

```
package br.com.targettrust.aop.aspect;
import org.aspectj.lang.Aspects;

public aspect Singleton extends PaiBurro{

    private int cont = 0;

    after(): call(* *.*.func()){
        cont++;
        Singleton o = (Singleton)Aspects.aspectOf(Singleton.class);
        System.out.println("Metodo:" +
thisJoinPoint.getSignature().getName() +
        " invocado: " + cont + " vezes! Monitor: " + o.nome);
    }
}
```

Inner Types Declarations

Recurso do AspectJ que permite adicionar membros nas classes de forma dinâmica. Com este recurso podemos adicionar atributos a um classe Java e utilizar essa informação para algum processamento em algum advice de um aspecto.

Não é difícil adicionar métodos a classes Java com os recursos de inner-type declarations do AspectJ, podemos inclusive fazer com que determinada classe implemente um set de interfaces.

```
declare parents: br.com.B extends br.com.A;
```

```
declare parents: br.com.B implements br.com.A;
```

Inner Types Declarations

Para adicionar atributos a uma classe Java podemos fazer conforme o exemplo abaixo:

```
private List<Porta> Casa.portas = new ArrayList<Porta>();
```

Para adicionar métodos a uma classe (Isso é necessário quando implementamos uma interface via aspectos) podemos fazer conforme o exemplo abaixo.

```
public void Casa.pintar(Cor c){  
    System.out.println("Pintando a casa de cor:" + c);  
}
```

Outro recurso que podemos utilizar é a adição de construtores a classes Java através de inner-type declarations, confira o exemplo abaixo.

```
public Casa.new(String rua) {  
    super();  
    System.out.println(rua);  
}
```

Inner Types Declarations

```
declare parents:
    HealthWatcherFacade implements IFacade;

declare parents: Complaint || Person
    implements java.io.Serializable;

public static void
    HealthWatcherFacade.main(String[] args) {
    try {...
        java.rmi.Naming.rebind("/HW");
    } catch ...
}
```

Adicionando
o método
main na classe
fachada

Alterando a hierarquia de tipos

Recursos Avançados: Annotations

```
@Pointcut("call(* *.*(int)) && args(i) && if()")
public static boolean chamadaIf(int i) {
    return i > 0;
}
```

```
pointcut chamadaIf(int i) : call(* *.*(int)) && args(i) && if(i > 0);
```

```
@AfterReturning("criticalOperation()")
public void phew() {
    System.out.println("phew");
}

@AfterReturning(pointcut="call(Foo+.new(..))", returning="f")
public void itsAFoo(Foo f) {
    System.out.println("It's a Foo: " + f);
}
```

Recursos Avançados: Declare Soft

Esse recurso foi introduzido apartir do AspectJ 1.5.X. É um recurso muito importante com ele podemos fazer com que as Exceptions do Java sejam encapsuladas em uma Exception do AspectJ que é do tipo unchecked e por consequência estende RuntimeException. Se a exceção que for utilizada para o soft for uma Exception que estende RuntimeException você receberá um warning e o soft não irá ocorrer.

A Exception do AspectJ que vai ser a encapsuladora de sua exeção será a `org.aspectj.lang.SoftException`. Esse recurso é útil para tratar os erros provocados porventura por aspectos.

Recursos Avançados: Declare Soft

```
package br.com.targettrust.aop.declaresoft;

public class MinhaExcpetionChata extends Exception {

    private static final long serialVersionUID = 1L;

    public MinhaExcpetionChata(String msg) {
        super(msg);
    }

    public MinhaExcpetionChata(String msg, Throwable rootCause) {
        super(msg, rootCause);
    }
}
```

```
package br.com.targettrust.aop.declaresoft;

public aspect DeclareSoft {
    declare soft : MinhaExcpetionChata : execution(* *(..));
}
```

Recursos Avançados: Declare Soft

```
package br.com.targettrust.aop.declaresoft.java;

import org.aspectj.lang.SoftException;
import br.com.targettrust.aop.declaresoft.MinhaExcpetionChata;

public class DeclareSoftTest {

    public void fazAlgo() throws Exception {
        throw new MinhaExcpetionChata("Exception chata mesmo");
    }

    public static void main(String[] args) {
        try {
            DeclareSoftTest dst = new DeclareSoftTest();
            dst.fazAlgo();
        } catch (SoftException se) {
            System.out.println(" 1 - SoftExcetion: " +
se.getWrappedThrowable());
        } catch (Exception e) {
            System.out.println(" 2 - Exception: " + e);
        }
    }
}
```

```
1 - SoftExcetion:
br.com.targettrust.aop.declaresoft.MinhaExcpetionChata: Exception
chata mesmo
```

Atividades em Grupo

(2 a 3 alunos)

Atividades de AspectJ – Sala de Aula

- 1. Especificar casos / problemas para fazer uso da Programação Orientada a Aspectos**
 - 2. Escrever os programas Java e AspectJ usando o Design Patterns *Observer / Observable***
 - 3. Desenvolver um AspectJ para implementação do Design Patterns *Singleton***
 - 4. Analisar os programas usando Java e AspectJ**
 - 5. Realizar as alterações desejadas de acordo com o Diagrama de Classes**
 - 6. Aplicação Completa: Regras de Negócios + Interface Gráfica + Banco de Dados**
-

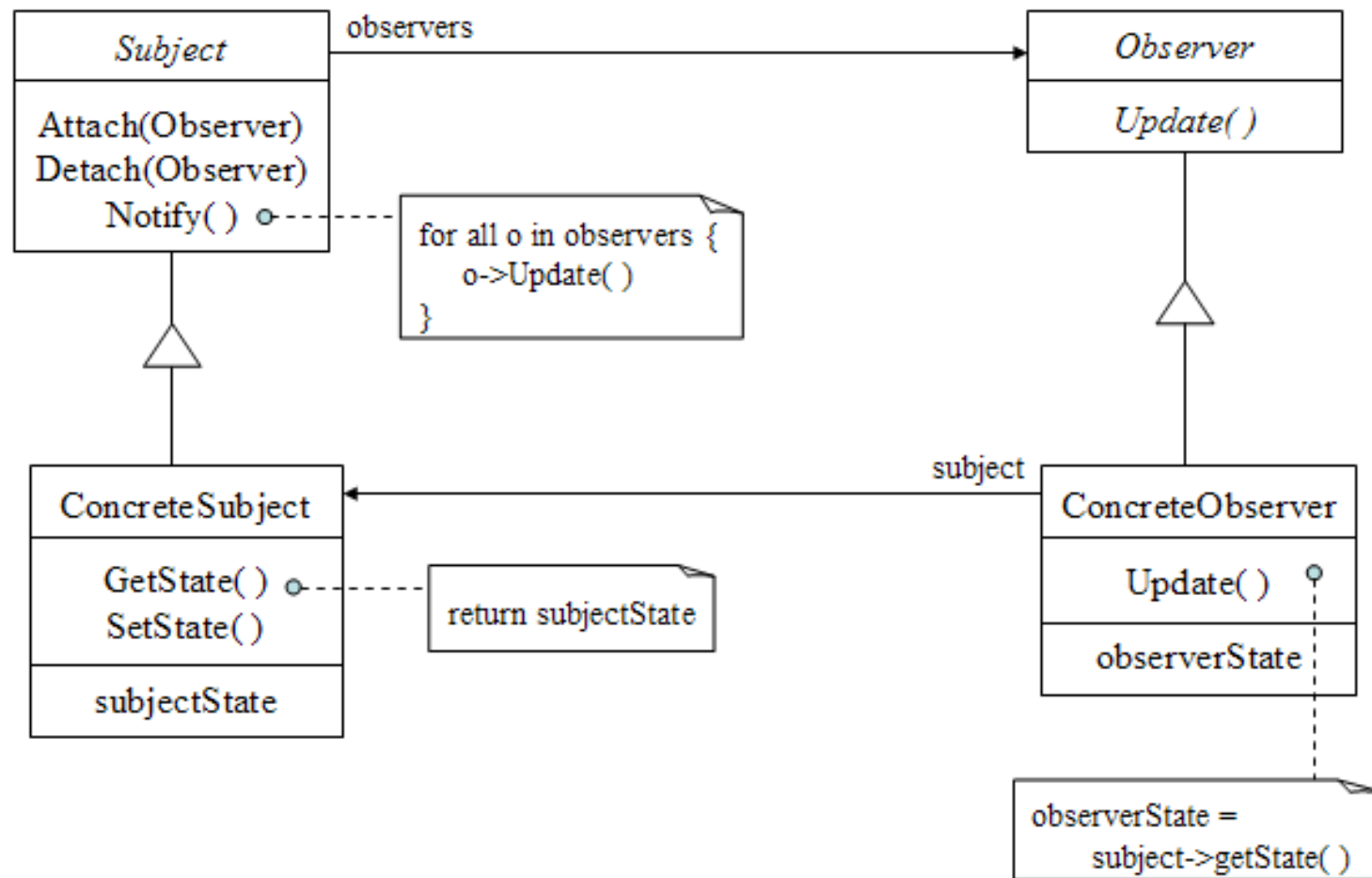
Atividade 1

Atividade 1 - Sistemas usando Aspectos

Descrever três casos em que aspectos podem ser usados para melhorar um sistema. Escrever um parágrafo com 20 linhas no mínimo para cada caso. Os casos devem ser precisos e específicos, ou seja, você deve detalhar todo o cenário de como seria a utilização dos conceitos de aspectos, assim como também descrever um sistema possivelmente real, e não um sistema fictício.

Atividade 2

Atividade 2 - Exemplo de Design Patterns



Atividade 2 - Observer/Observable usando Java

```
Observer.java X
1 package pattern.java;
2
3 abstract class Observer
4 {
5     protected Subject subj;
6
7     public abstract void update( ) ;
8 }
9
```

```
BinObserver.java X
1 package pattern.java;
2
3 class BinObserver extends Observer
4 {
5     public BinObserver( Subject s )
6     {
7         subj = s;
8         subj.attach( this );
9     }
10
11     public void update( )
12     {
13         System.out.println( "Numero Binário: " + Integer.toBinaryString( subj.getState() ) );
14     }
15 }
```

Atividade 2 - Observer/Observable usando Java

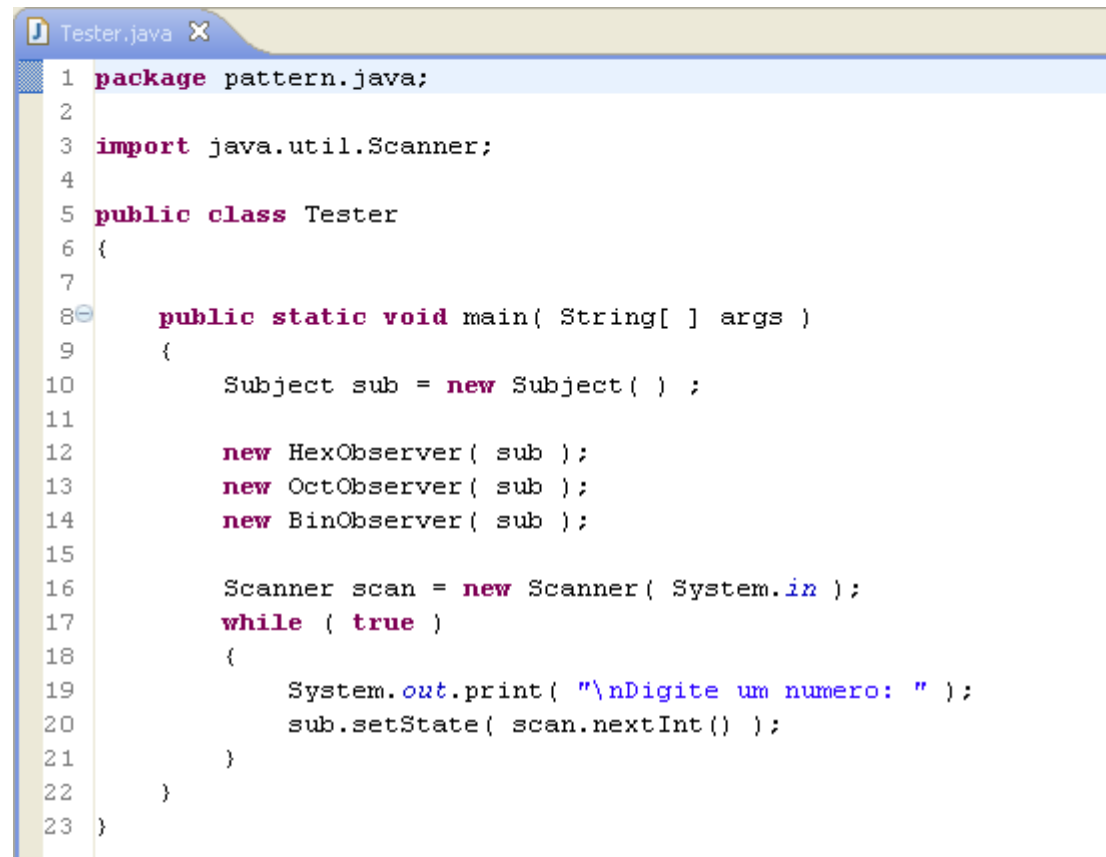
```
HexObserver.java X
1 package pattern.java;
2
3 class HexObserver extends Observer {
4
5     public HexObserver( Subject s )
6     {
7         subj = s;
8         subj.attach( this );
9     }
10
11     public void update( )
12     {
13         System.out.println( "Numero Hexadecimal: " + Integer.toHexString( subj.getState() ) );
14     }
15 }
```

```
OctoObserver.java X
1 package pattern.java;
2
3 class OctObserver extends Observer
4 {
5     public OctObserver( Subject s )
6     {
7         subj = s;
8         subj.attach( this );
9     }
10
11     public void update( )
12     {
13         System.out.println( "Numero Octal: " + Integer.toOctalString( subj.getState() ) );
14     }
15 }
```

Atividade 2 - Observer/Observable usando Java

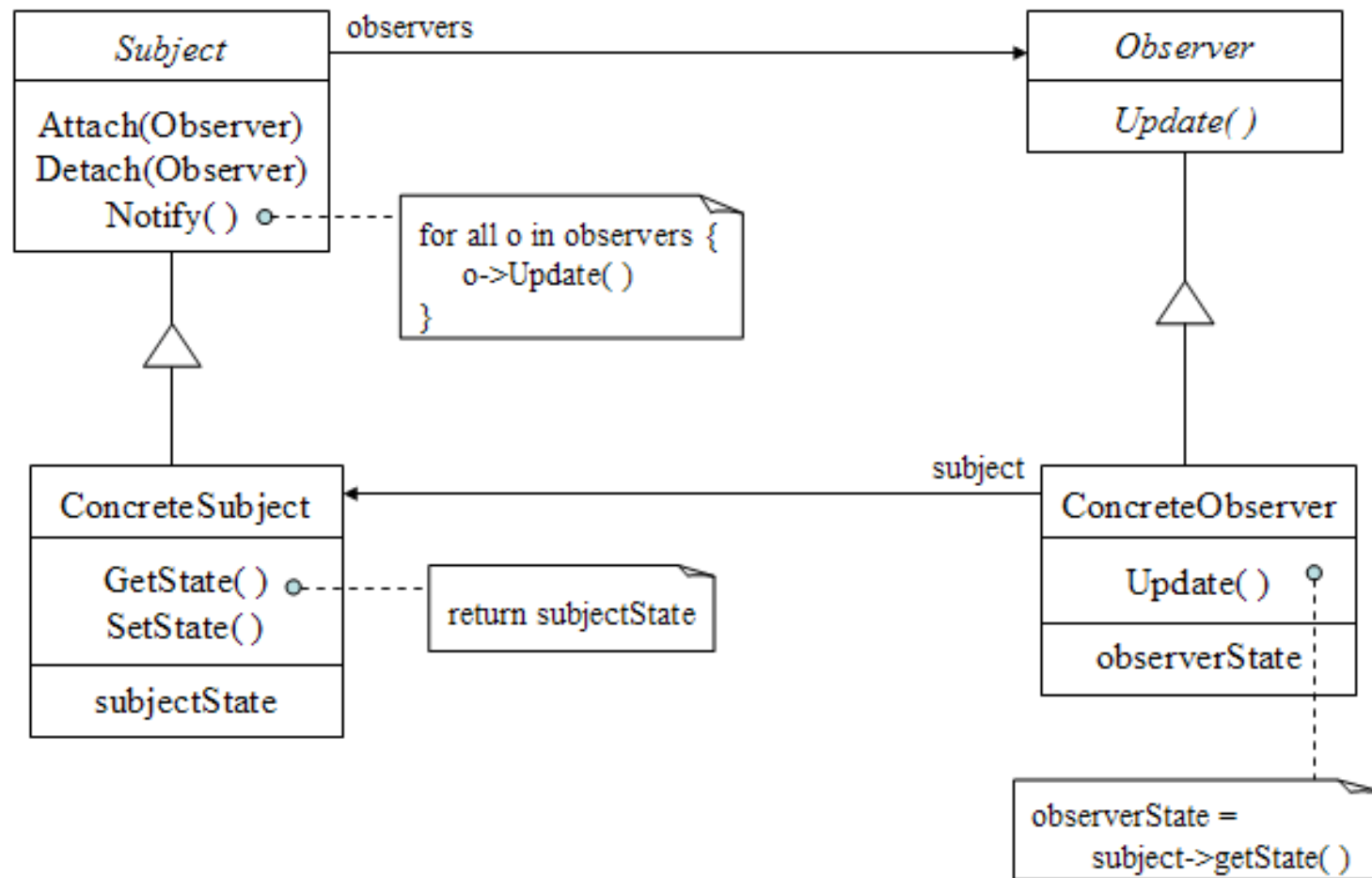
```
Subject.java X
1 package pattern.java;
2
3 class Subject
4 {
5     private Observer[ ] observers = new Observer[ 9 ] ;
6
7     private int totalObs = 0;
8
9     private int state;
10
11     public void attach( Observer o )
12     {
13         observers[ totalObs++ ] = o ;
14     }
15
16     public int getState( )
17     {
18         return state;
19     }
20
21     public void setState( int in )
22     {
23         state = in;
24         notifica( );
25     }
26
27     private void notifica( )
28     {
29         for (int i = 0; i < totalObs ; i++ )
30         {
31             observers[ i ].update( ) ;
32         }
33     }
34 }
```


Atividade 2 - Observer/Observable usando Java

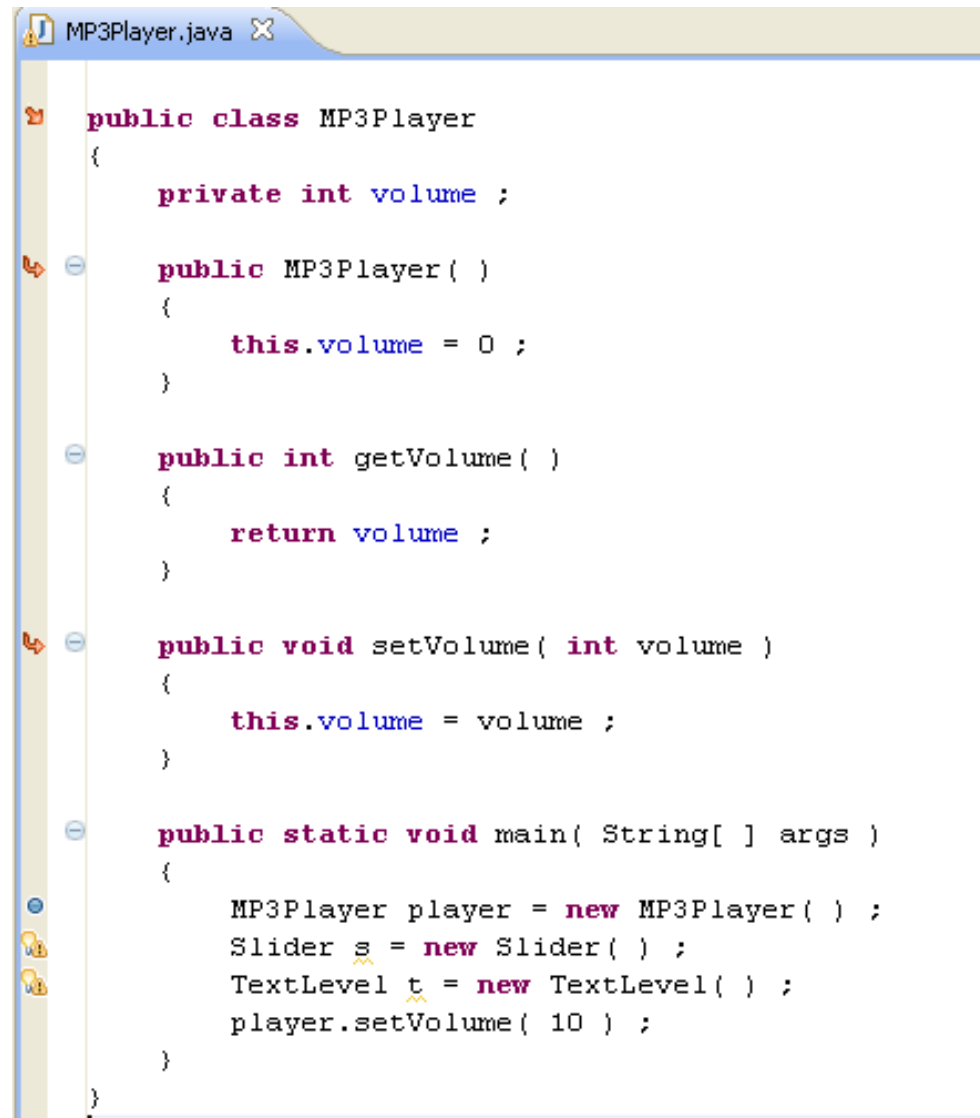
A screenshot of a Java IDE window titled 'Tester.java'. The code is as follows:

```
1 package pattern.java;
2
3 import java.util.Scanner;
4
5 public class Tester
6 {
7
8     public static void main( String[ ] args )
9     {
10         Subject sub = new Subject( ) ;
11
12         new HexObserver( sub );
13         new OctObserver( sub );
14         new BinObserver( sub );
15
16         Scanner scan = new Scanner( System.in );
17         while ( true )
18         {
19             System.out.print( "\nDigite um numero: " );
20             sub.setState( scan.nextInt() );
21         }
22     }
23 }
```

Atividade 2 - Usando Design Patterns (Observer/Observable) **FIAP**



Atividade 2 – Classes JAVA



```
MP3Player.java X
public class MP3Player
{
    private int volume ;

    public MP3Player( )
    {
        this.volume = 0 ;
    }

    public int getVolume( )
    {
        return volume ;
    }

    public void setVolume( int volume )
    {
        this.volume = volume ;
    }

    public static void main( String[ ] args )
    {
        MP3Player player = new MP3Player( ) ;
        Slider s = new Slider( ) ;
        TextLevel t = new TextLevel( ) ;
        player.setVolume( 10 ) ;
    }
}
```

Atividade 2 – Classes JAVA

```
Slider.java X
public class Slider
{
    public Slider( )
    {
    }

    public void update( )
    {
        System.out.println("::: Slider foi informado :::");
    }
}
```

```
TextLevel.java X
public class TextLevel
{
    public TextLevel( )
    {
    }

    public void update( )
    {
        System.out.println("::: TextLevel foi informado :::");
    }
}
```

Atividade 2 – Aspectos (PadraoObserver.aj)

```
PadraoObserver.aj x
+import java.util.Iterator;

public abstract aspect PadraoObserver
{
    protected interface Subject { }

    protected interface Observer
    {
        void update( ) ;
    }

    Subject sujeito;
    List<Observer> observadores = new ArrayList<Observer>( );

    abstract pointcut criarSujeito( Subject s ) ;
    abstract pointcut criarObservador( Observer o ) ;
    abstract pointcut modificarSujeito( ) ;

    after( Subject s ) : criarSujeito( s )
    {
        if ( sujeito == null )
        {
            System.out.println( " foi criado um sujeito " + s.getClass().getName( ) );
            sujeito = s;
        }
    }

    after( Observer o ) : criarObservador( o )
    {
        System.out.println( " foi criado um observador " + o.getClass().getName( ) );
        observadores.add( o ) ;
    }
}
```

```
after( ) : modificarSujeito( )
{
    Iterator<Observer> i = observadores.iterator( );
    while ( i.hasNext( ) )
    {
        Observer o = ( Observer ) i.next( ) ;
        System.out.println( "\n foi executado o método update() do observador " + o.getClass().getName( ) );
        o.update( ) ;
    }
}
```

Atividade 2 – Aspectos (ObserverConcreto.aj)

```
ObserverConcreto.aj X

public aspect ObserverConcreto extends PadraoObserver
{
    pointcut criarSujeito( Subject s ) : execution( MP3Player.new( ) ) && this( s ) ;

    pointcut criarObservador( Observer o ) :
        ( execution( Slider.new( ) ) || execution( TextLevel.new( ) ) ) && this( o ) ;

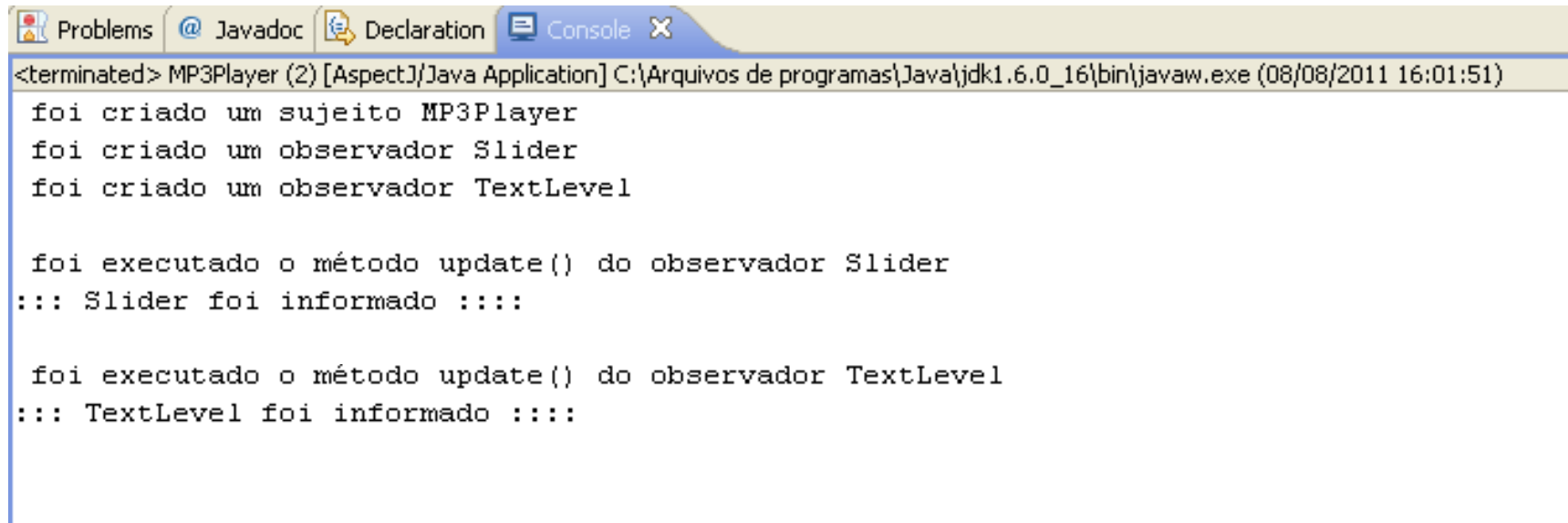
    pointcut modificarSujeito() : execution(* MP3Player.setVolume(..) ) ;

    declare parents: MP3Player implements Subject;
    declare parents: Slider implements Observer;
    declare parents: TextLevel implements Observer;

    public void Slider.update( Subject s )
    {
        MP3Player p = ( MP3Player ) s;
        System.out.println( " foi informado Slider " );
    }

    public void TextLevel.update( Subject s )
    {
        MP3Player p = ( MP3Player ) s;
        System.out.println( " foi informado TextLevel " );
    }
}
```

Atividade 2 – Aspectos (execução)



The screenshot shows an IDE console window with the following tabs: Problems, Javadoc, Declaration, and Console. The Console tab is active, displaying the output of a Java application. The output is as follows:

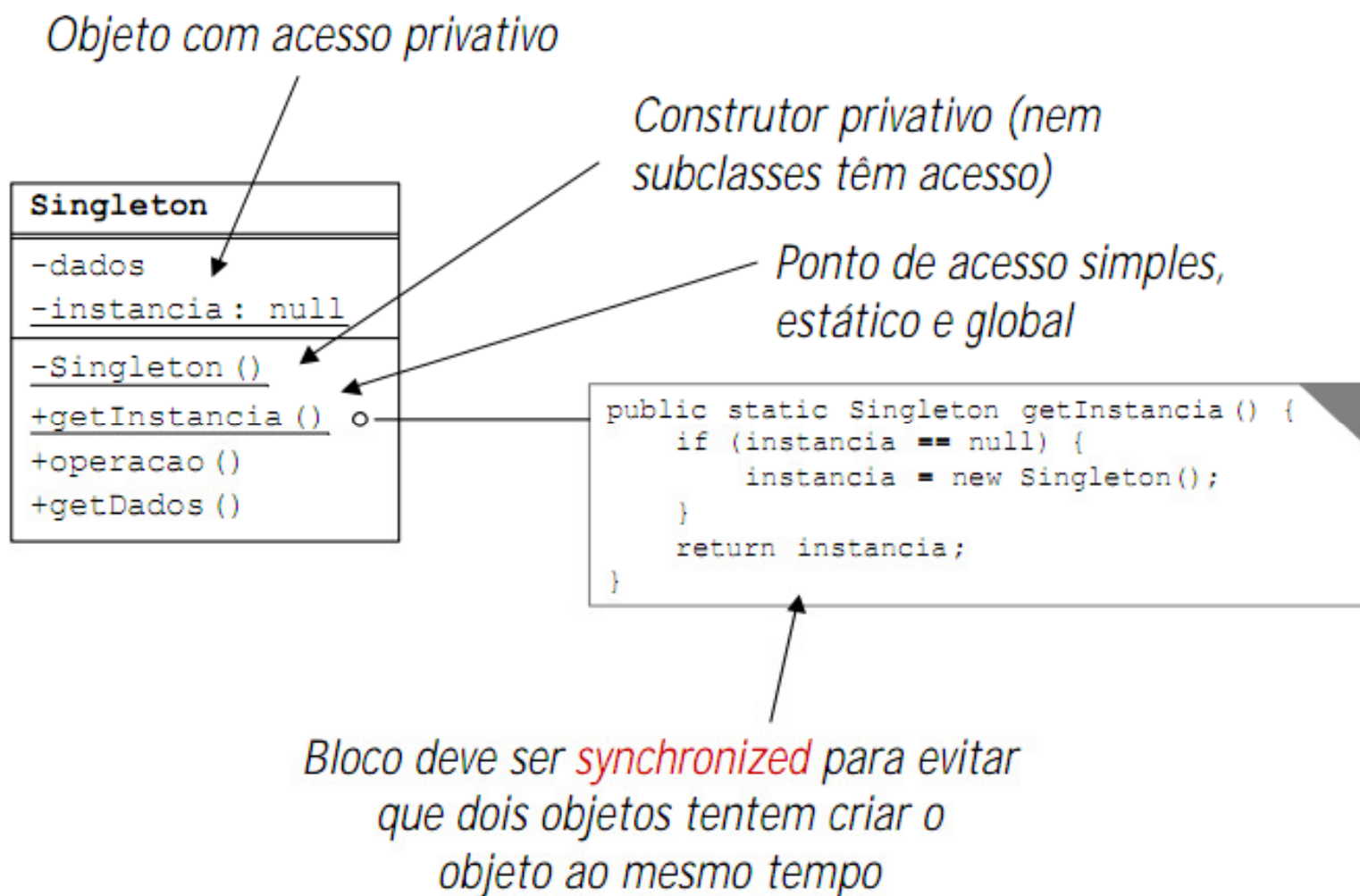
```
<terminated> MP3Player (2) [AspectJ/Java Application] C:\Arquivos de programas\Java\jdk1.6.0_16\bin\javaw.exe (08/08/2011 16:01:51)
foi criado um sujeito MP3Player
foi criado um observador Slider
foi criado um observador TextLevel

foi executado o método update() do observador Slider
::: Slider foi informado :::

foi executado o método update() do observador TextLevel
::: TextLevel foi informado :::
```


Atividade 3

Implementar o padrão *Singleton* usando aspectos, nos mesmos moldes que o exemplo anterior.



Atividade 3 - Aspecto usando Design Patterns

```
Singleton.java X
1 package pattern.java;
2
3 public class Singleton
4 {
5     private String dados ;
6
7     private static Singleton instancia = null ;
8
9     private Singleton( )
10    {
11    }
12
13    public static synchronized Singleton getInstancia( )
14    {
15        if ( instancia == null )
16        {
17            instancia = new Singleton( ) ;
18        }
19        return instancia ;
20    }
21
22    public void setDados( String dados )
23    {
24        this.dados = dados ;
25    }
26
27    public String getDados( )
28    {
29        return this.dados ;
30    }
31
32    public static void main( String[] args )
33    {
34        Singleton x = Singleton.getInstancia( ) ;
35        Singleton y = Singleton.getInstancia( ) ;
36        if ( x == y )
37        {
38            System.out.println( " sao iguais os objetos " );
39        }
40    }
41 }
```

Atividade 3 - Aspectos usando Design Patterns

```
Singleton.aj X
1 import java.util.HashMap;
2
3 public aspect Singleton
4 {
5     HashMap<Class<?>, Object> singletons = new HashMap<Class<?>, Object>();
6
7     pointcut tratarObjetosSingleton( ) : call( pattern.java.*.new(..) ) ;
8
9     Object around( ) : tratarObjetosSingleton( )
10    {
11        Class<?> singletonClass = thisJoinPoint.getSignature( ).getDeclaringType( ) ;
12        Object singletonObject = this.singletons.get( singletonClass ) ;
13
14        if ( singletonObject == null )
15        {
16            System.out.println( " uma nova instancia será criada e armazenada em cache " + singletonClass.getName( ) ) .
17            singletonObject = proceed( ) ;
18            this.singletons.put( singletonClass, singletonObject ) ;
19        }
20        else
21        {
22            System.out.println( " um objeto singleton foi recuperado -> " + singletonClass.getName( ) ) ;
23        }
24        return singletonObject;
25    }
26 }
27
```

Atividade 3 - Aspectos usando Design Patterns

```
1 package pattern.java;
2
3 public class Cliente
4 {
5     private String nome ;
6
7     public Cliente()
8     {
9     }
10
11     public void setNome( String nome )
12     {
13         this.nome = nome ;
14     }
15
16     public String getNome( )
17     {
18         return nome ;
19     }
20
21 }
22
```

```
1 package pattern.java;
2
3 public class Fornecedor
4 {
5     private String nome ;
6
7     public Fornecedor()
8     {
9     }
10
11     public void setNome( String nome )
12     {
13         this.nome = nome ;
14     }
15
16     public String getNome( )
17     {
18         return nome ;
19     }
20
21 }
22
```

Atividade 3 - Aspectos usando Design Patterns

```
Tester.java x
1 package pattern.java;
2
3 public class Tester
4 {
5     public static void main(String[] args)
6     {
7         Fornecedor marcos = new Fornecedor( );
8         marcos.setNome( " Marcos " );
9         System.out.println( marcos.getNome( ) );
10
11        Fornecedor roberto = new Fornecedor( );
12        roberto.setNome( " Roberto " );
13        System.out.println( roberto.getNome( ) );
14
15        System.out.println( "Novamente objeto::marcos " + marcos.getNome( ) );
16        System.out.println( "Novamente objeto::roberto " + roberto.getNome( ) );
17
18        Cliente macedo1 = new Cliente( );
19        Cliente macedo2 = new Cliente( );
20        Cliente macedo3 = new Cliente( );
21        Cliente macedo4 = new Cliente( );
22        Cliente macedo5 = new Cliente( );
23    }
24 }
```

```
Console x
<terminated> Tester (3) [Java Application] C:\Arquivos de programas\Java\jdk1.6.0_16\bin\javaw.exe (22/08/2011 16:22:44)
uma nova instancia será criada e armazenada em cache pattern.java.Fornecedor
Marcos
um objeto singleton foi recuperado -> pattern.java.Fornecedor
Roberto
Novamente objeto::marcos Roberto
Novamente objeto::roberto Roberto
uma nova instancia será criada e armazenada em cache pattern.java.Cliente
um objeto singleton foi recuperado -> pattern.java.Cliente
um objeto singleton foi recuperado -> pattern.java.Cliente
um objeto singleton foi recuperado -> pattern.java.Cliente
um objeto singleton foi recuperado -> pattern.java.Cliente
```

Implemente uma nova solução para o padrão *Singleton* usando Aspecto

Atividade 4

Atividade 4 - Analisar as classes Java e Aspectos

```
I.java X
1 public interface I
2 {
3     public void m( ) ;
4 }
5
```

```
B.java X
1 public class B implements I
2 {
3     public void m( )
4     {
5         System.out.println("Executando o método m de B");
6     }
7 }
```

```
A.java X
1 public class A implements I
2 {
3     private I i;
4
5     public A( I i )
6     {
7         this.i = i;
8     }
9
10    public void m( )
11    {
12        System.out.println("Executando o método m de A");
13        i.m( ) ;
14    }
15 }
```

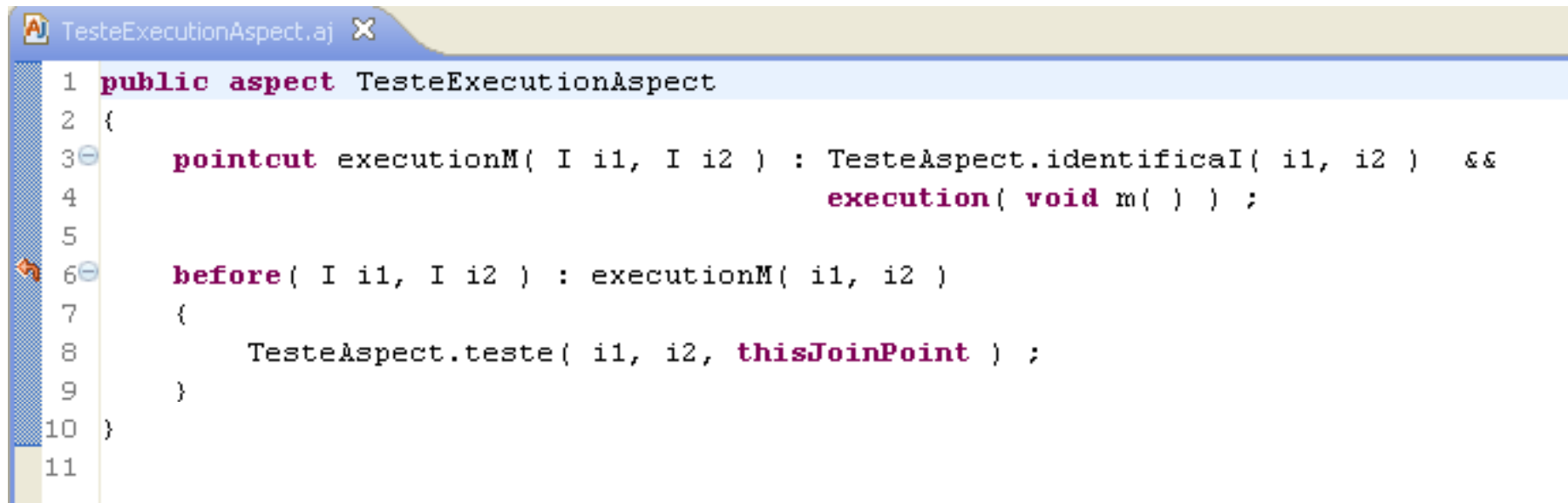
Atividade 4 - Analisar as classes Java e Aspectos

```
TesteAspect.aj x
1 public aspect TesteAspect
2 {
3     public void I.o( )
4     {
5         System.out.println("Método o de I");
6     }
7
8     pointcut identificaI( I i1, I i2 ): this( i1 ) && target( i2 );
9
10    static void teste( I i1, I i2, Object o )
11    {
12        System.out.println("-----");
13        System.out.println( o );
14        System.out.println( i1 );
15        System.out.println( i2 );
16        System.out.println("-----");
17    }
18
19    public static void main( String[ ] args )
20    {
21        I i = new A( new B( ) );
22        i.m( ) ;
23        i.o( ) ;
24    }
25 }
```

Atividade 4 - Analisar as classes Java e Aspectos

```
TesteCallAspect.aj x
1 public aspect TesteCallAspect
2 {
3     pointcut callM( I i1, I i2 ) : TesteAspect.identificaI( i1, i2 ) &&
4         call( void m( ) ) ;
5
6     before( I i1, I i2 ) : callM( i1, i2 )
7     {
8         TesteAspect.teste( i1, i2, thisJoinPoint ) ;
9     }
10 }
11
```

Atividade 4 - Analisar as classes Java e Aspectos



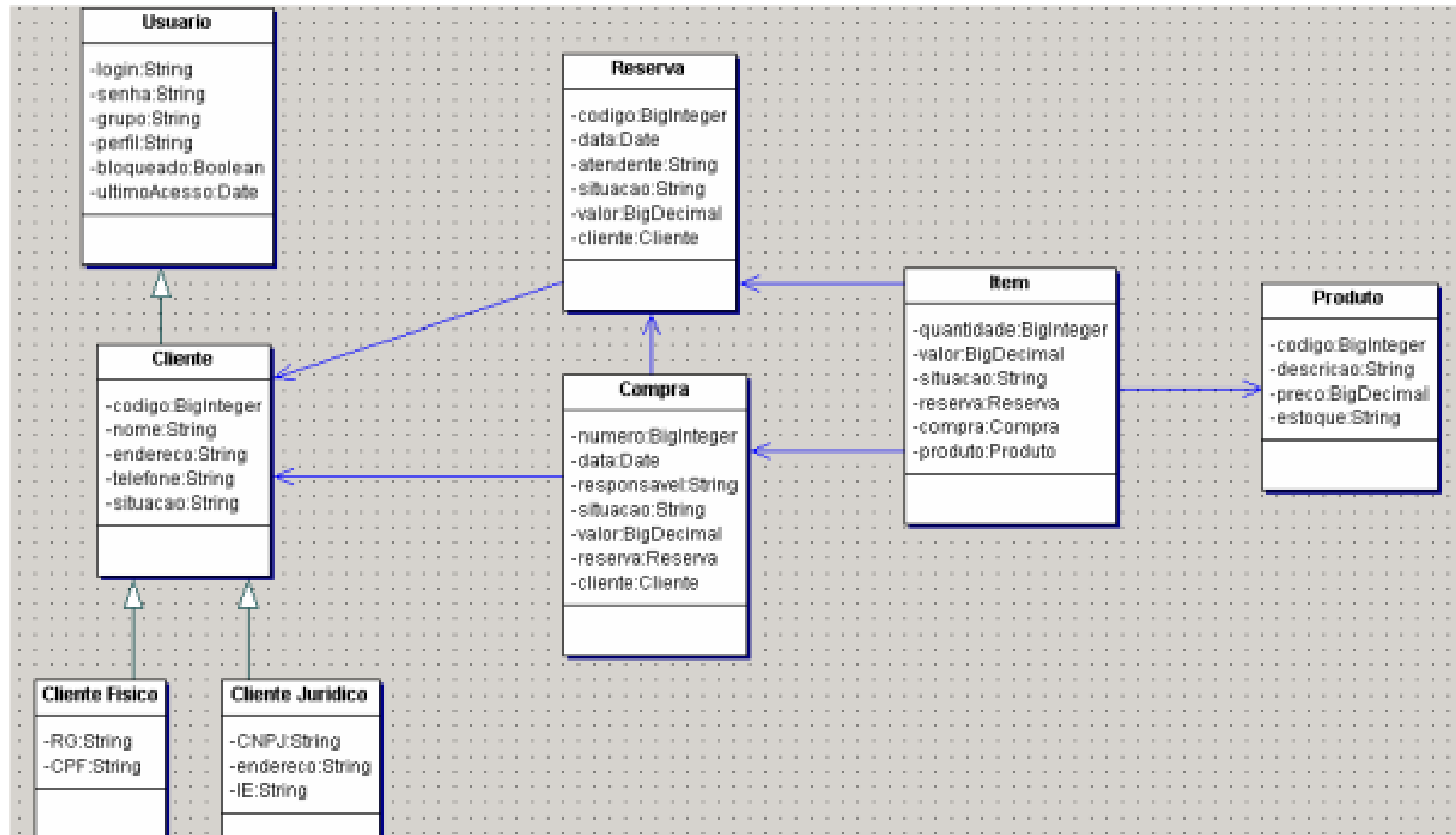
The screenshot shows a code editor window titled "TesteExecutionAspect.aj". The code is written in Java and defines an aspect named "TesteExecutionAspect". It includes a "pointcut" named "executionM" that matches the execution of methods "m" in classes "i1" and "i2" that are identified by "TesteAspect.identificaI". The "before" advice is applied to the "executionM" pointcut, and it calls "TesteAspect.teste" with the arguments "i1", "i2", and "thisJoinPoint".

```
1 public aspect TesteExecutionAspect
2 {
3     pointcut executionM( I i1, I i2 ) : TesteAspect.identificaI( i1, i2 ) &&
4         execution( void m( ) ) ;
5
6     before( I i1, I i2 ) : executionM( i1, i2 )
7     {
8         TesteAspect.teste( i1, i2, thisJoinPoint ) ;
9     }
10 }
11
```

Atividade 5

- Faça as alterações no diagrama para atender aos requisitos**

Atividade 5 - Diagrama de Classes



Atividade 5 - Requisitos Solicitados

FIAP

-
- 1. Imprimir na console apenas as Compras com valor maior que zero e menor que R\$ 500,00**
 - 2. Para os produtos que custam menos que R\$ 1.000,00, e que tenham pelo menos duas unidades em estoque, aplique um desconto nas compras no valor de R\$ 75,00**
 - 3. Para os Clientes que possuem pelo menos um compra realizada em Fevereiro de 2013, nas próximas compras a partir de hoje aplique um desconto de 10% em suas novas compras**
 - 4. Excluir todos os Clientes que realizaram apenas um compra nos últimos 5 anos**
 - 5. Imprime na console o percentual de Pedidos/Compras que foram geradas a partir de uma Reserva**



www.fiap.com.br – Central de Atendimento: (11) 3385-8000

Campi:

Aclimação I

Aclimação II

Paulista

Alphaville

Copyright © 2014 Prof. Ms. Marcos Macedo

Todos direitos reservados. Reprodução ou divulgação total ou parcial deste documento é expressamente proibido sem o consentimento formal, por escrito, do Professor (autor).