

MBA - Desenvolvimento
de Soluções Corporativas
Java (SOA)

AOP
Programação
Orientada a Aspectos

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

Junho/2013

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. **Aplicação Completa: Regras de Negócios + Interface Gráfica + Banco de Dados**
 6. **Desenvolver um AspectJ para os Design Patterns: Adapter, Factory, Decorator e Facade**
-

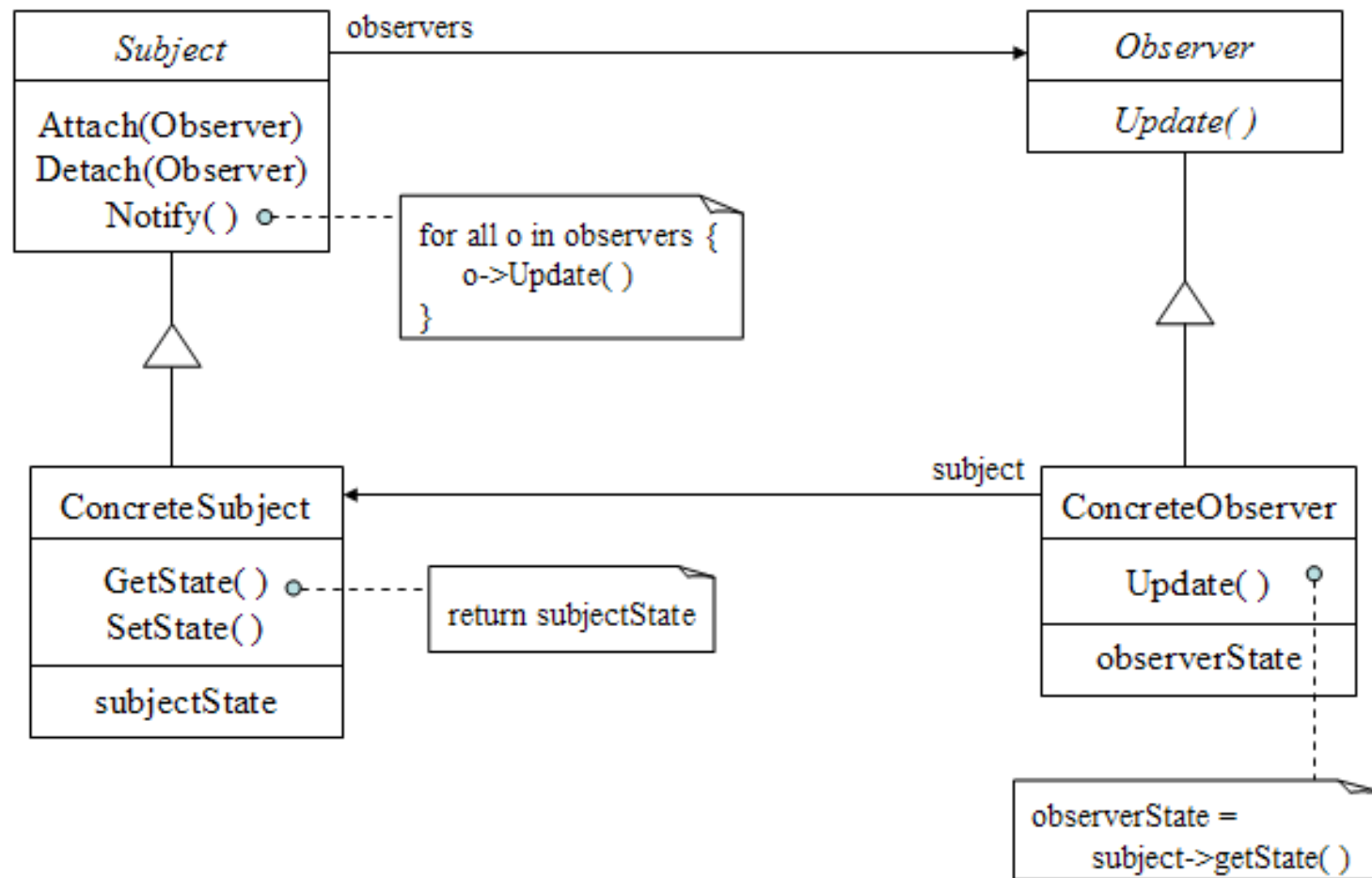
Atividade 1

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

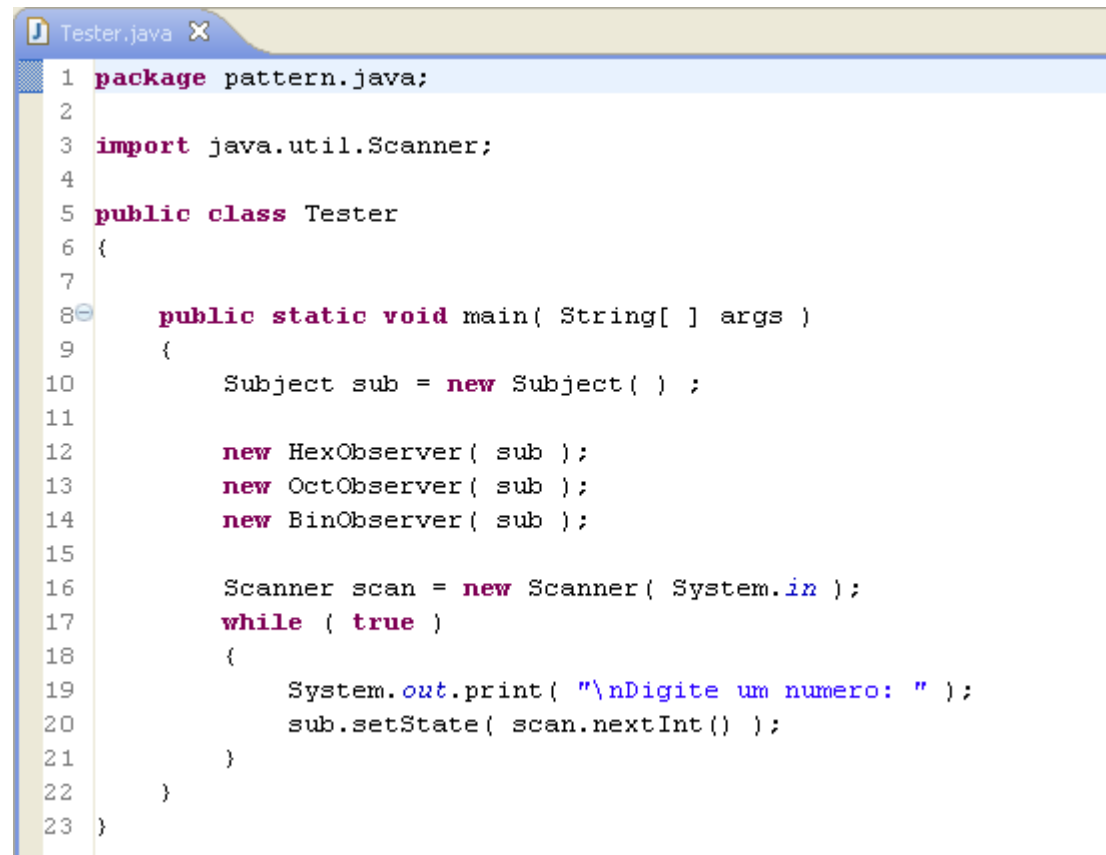
```
HexObserver.java X
1 package pattern.java;
2
3 class HexObserver extends Observer {
4
5     public HexObserver( Subject s )
6     {
7         subj = s;
8         subj.attach( this );
9     }
10
11     public void update( )
12     {
13         System.out.println( "Numero Hexadecimal: " + Integer.toHexString(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

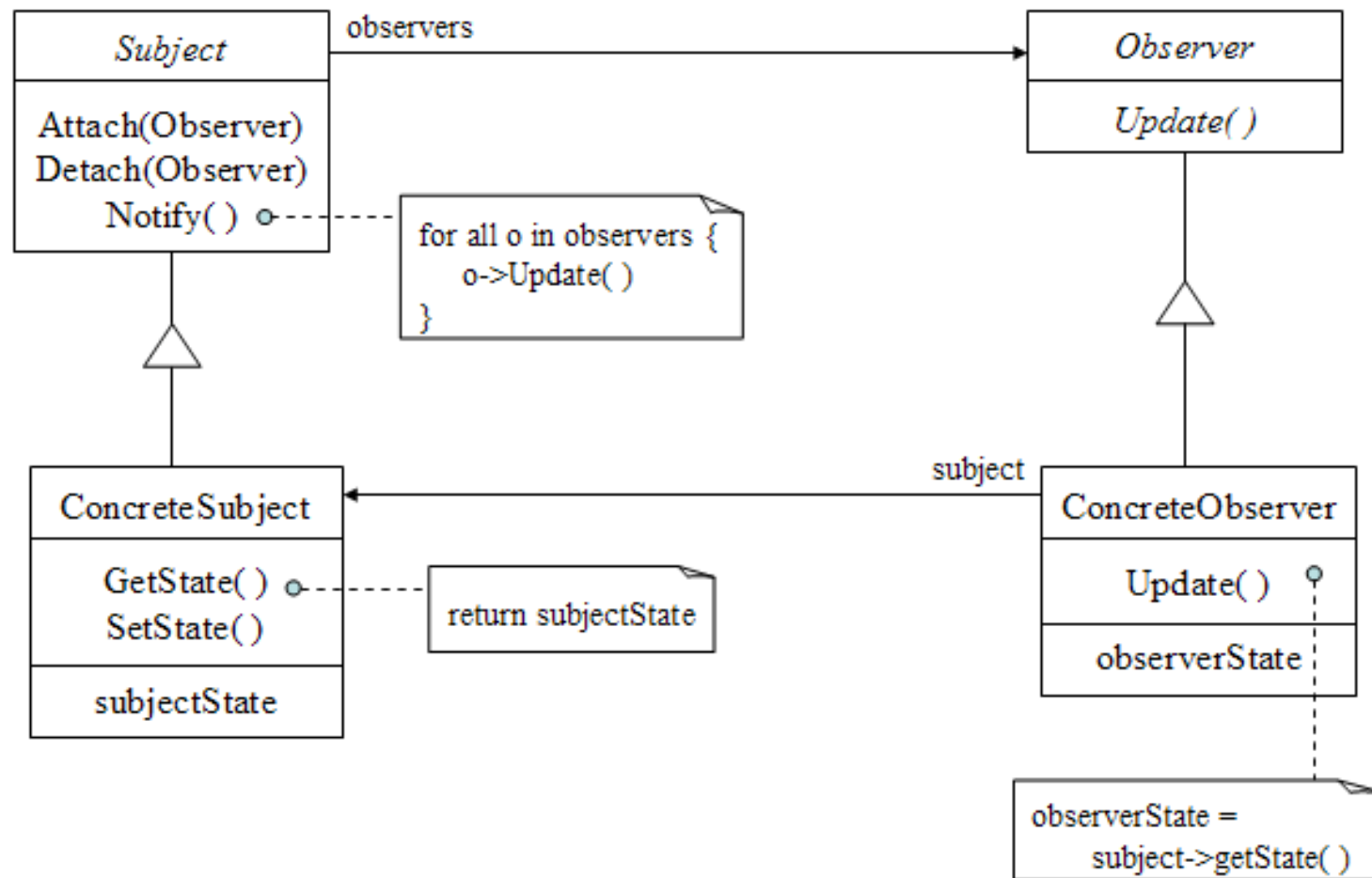
```
Subject.java X
1 package pattern.java;
2
3 class Subject
4 {
5     private Observer[ ] observers = new Observer[ 9 ] ;
6
7     private int totalObs = 0;
8
9     private int state;
10
11     public void attach( Observer o )
12     {
13         observers[ totalObs++ ] = o ;
14     }
15
16     public 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

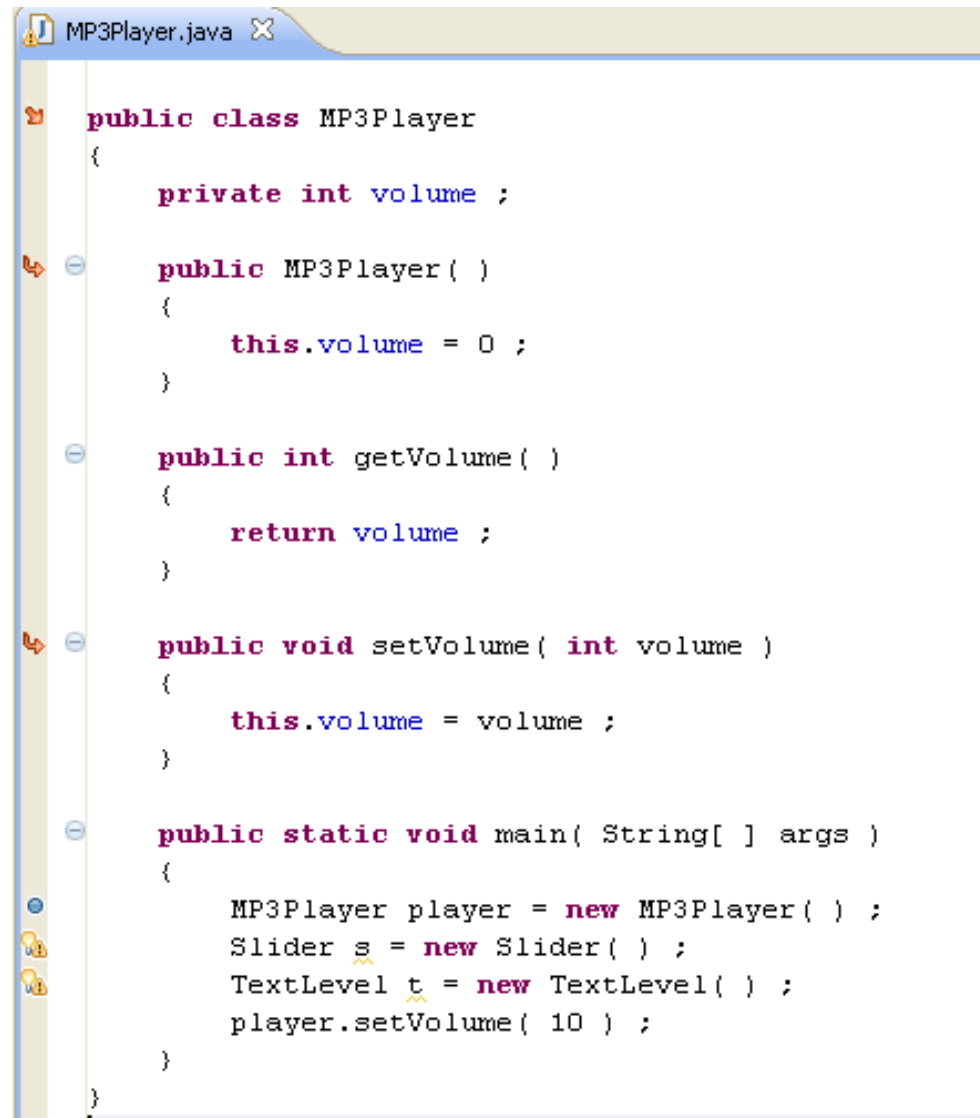
A screenshot of a Java IDE window titled 'Tester.java'. The code is as follows:

```
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>( );

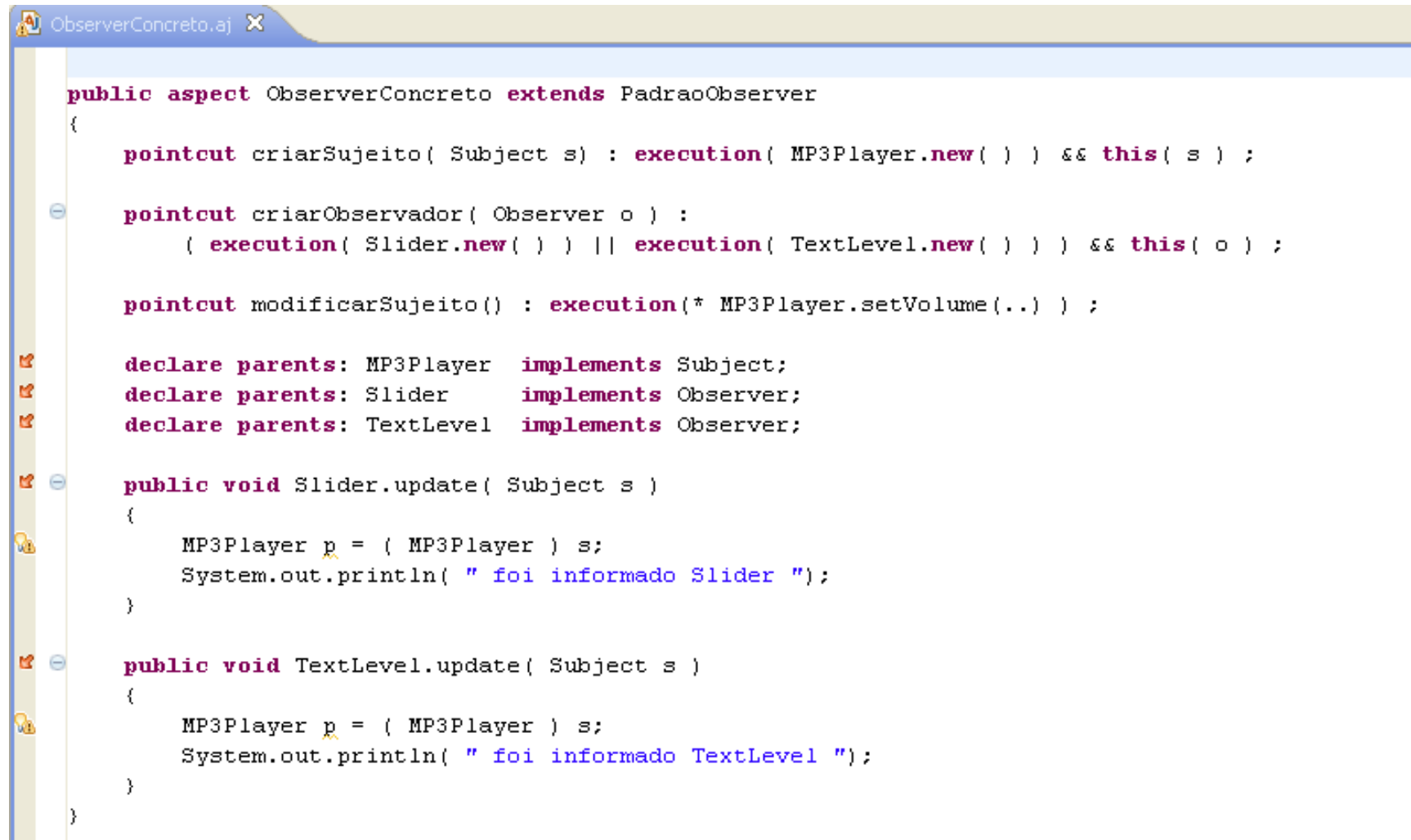
    abstract pointcut criarSujeito( Subject s ) ;
    abstract pointcut criarObservador( Observer o ) ;
    abstract pointcut modificarSujeito( ) ;

    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 ) ;

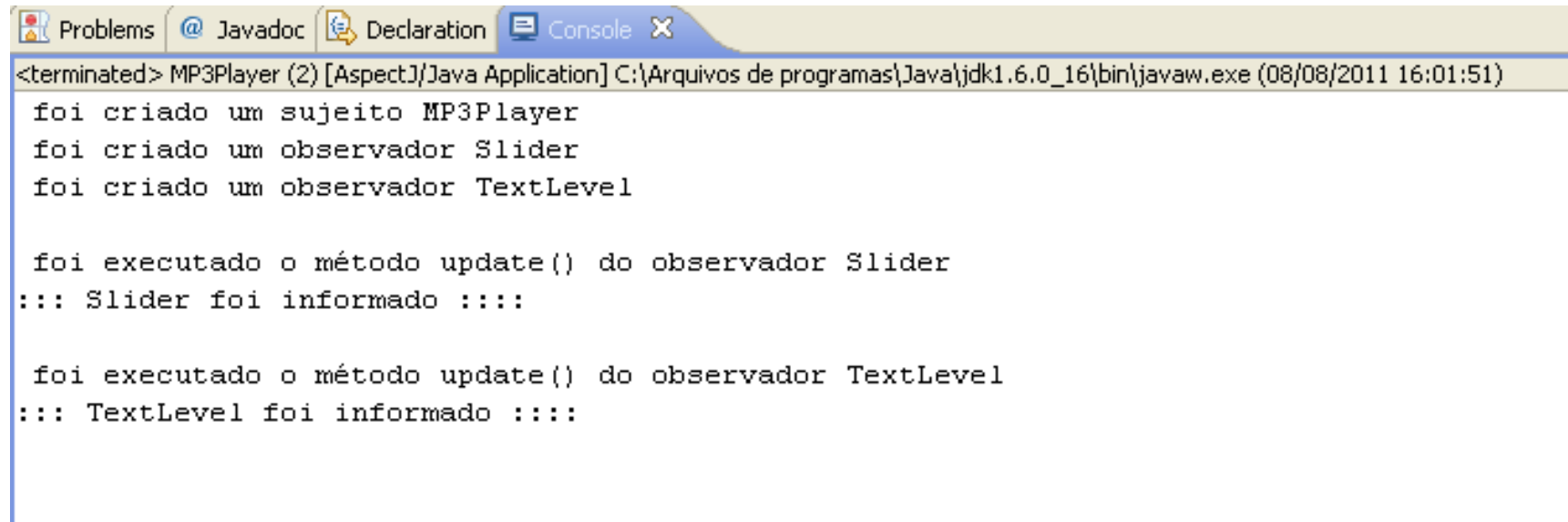
    pointcut modificarSujeito() : execution(* MP3Player.setVolume(..) ) ;

    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 the Eclipse IDE's Console window. The title bar includes tabs for 'Problems', 'Javadoc', 'Declaration', and 'Console'. The console output is as follows:

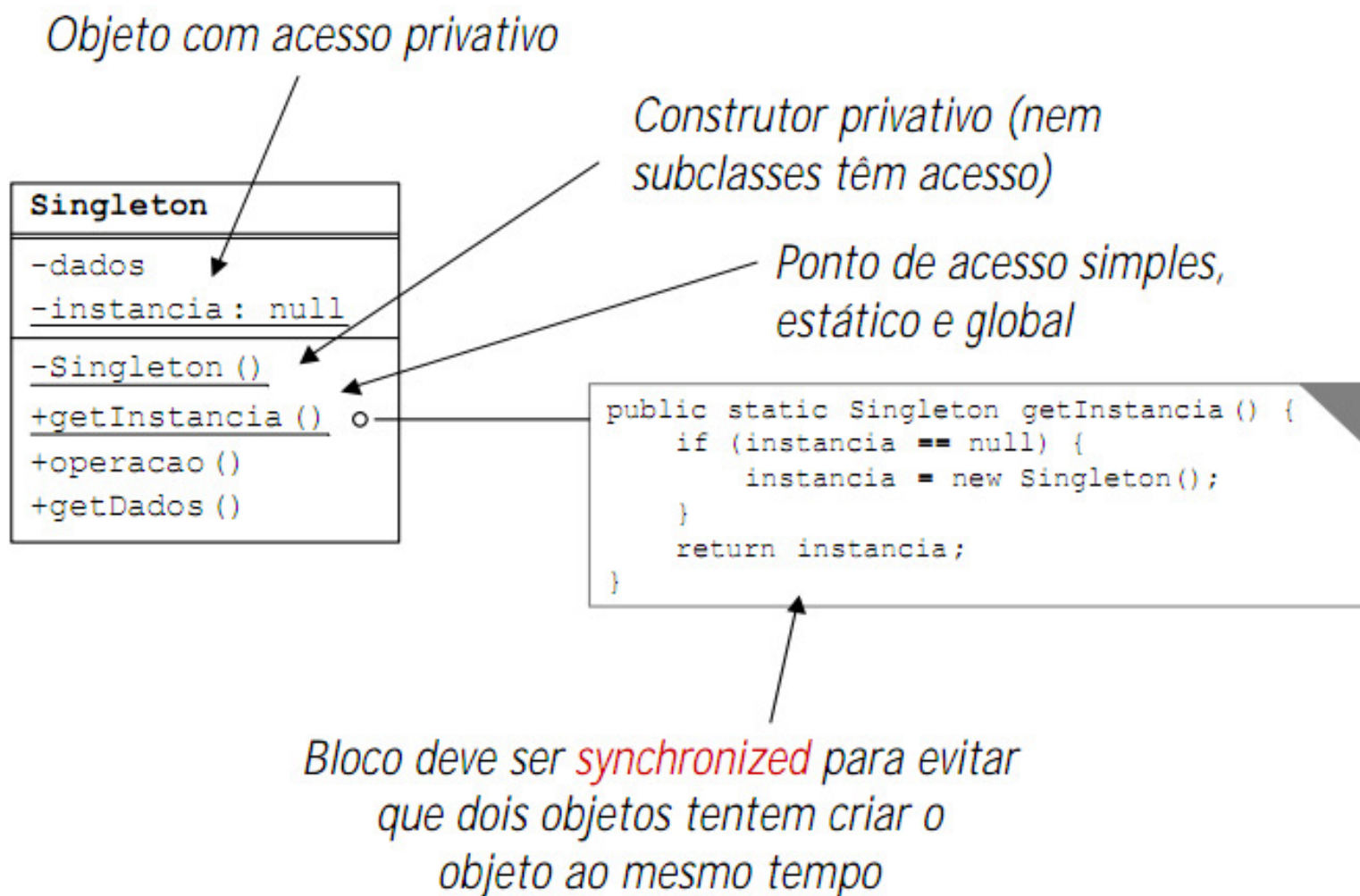
```
<terminated> MP3Player (2) [AspectJ/Java Application] C:\Arquivos de programas\Java\jdk1.6.0_16\bin\javaw.exe (08/08/2011 16:01:51)
foi criado um sujeito MP3Player
foi criado um observador Slider
foi criado um observador TextLevel

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

FIAP

```
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 }
```

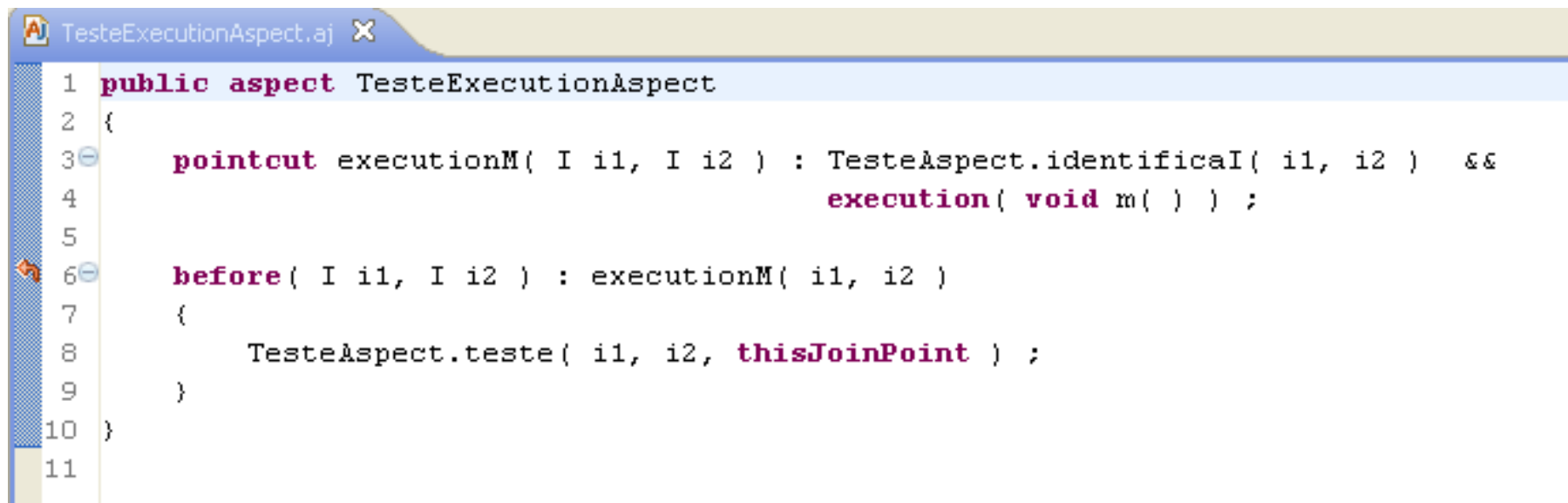
Atividade 4 - Analisar as classes Java e Aspectos

```
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 method calls on interfaces "I i1" and "I i2" where the method is identified by "TesteAspect.identificaI(i1, i2)". The "pointcut" is associated with the "execution" of a "void m()" method. A "before" advice is defined for the "executionM" pointcut, which calls "TesteAspect.teste(i1, i2, thisJoinPoint)" before the target method is executed.

```
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

**- Implementar usando
AspectJ e JBossAOP**

FIAP

Registro de Pedidos

- □ ×

Nro. Pedido

Data24/ 8 /2011 ▴ ▾

Tipo ClienteFísico ▾

ClienteMarcos R Macedo ▾

ProdutoProgramando em Aspecto usando Aspect J e Spring AOP ▾

Quantidade 10

Adicionar produto

TOTAL GERAL R\$ 10.000,00

Id Produto	Desc Produto	Vlr. Unitario	Qde	Desconto	Total

< ||| >

☒ Finalizado

Atualizar Total

-
1. Gerar log de auditoria antes e depois de cada regra de negócio executada.
 2. Gerar log de vendas diárias contendo a quantidade e valores totais dos pedidos gerados.
 3. Colocar as seguintes regras usando Aspectos
 - a) Toda venda maior que R\$ 1.000,00 terá desconto de 5% do valor em cada item;
 - b) Se o cliente comprar mais de 10 itens de pedidos terá automaticamente um desconto de 5% do valor total
 - c) Para pedidos emitidos entre os meses de Agosto e Setembro deverá ser adicionado um adicional de 10% do total do valor total do pedido
 - d) Para pedidos realizados aos domingos os mesmos estarão sob efeito de promoção, isto é, desconto de R\$ 100,00, isso somente para a compra do livro de “AspectJ – a arte de Pensar Diferente”

4. **Gerar um log de ERRO caso o banco de dados MySQL estiver indisponível nas ações de persistência.**

5. **Gravar em uma tabela de ERRO caso o usuário informe um data de pedido inferior à data atual do sistema.**



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

Campi:

Aclimação I

Aclimação II

Paulista

Alphaville

Copyright © 2013 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).