From Objects to Aspects: Assessing Modularity Evolution

Sérgio Bryton, Fernando Brito e Abreu

February 2008

Unlimited distribution subject to the copyright.

Technical Report
FCT/QUASAR-2008-TR-108
This technical report has been developed under the SOFTAS (POSC/EIA/60189/2004) project.
# Conteúdo

1 PIMETA: Paradigm Independent Meta-model  
   1.1 Introduction ................................................. 2  
   1.2 Paradigm Independent Meta-model ............................ 2  
       1.2.1 PIMETA description ..................................... 2  
       1.2.2 PIMETA well-formedness rules .......................... 4  
   1.3 PIMETA instantiation with OOP and AOP ....................... 6  
       1.3.1 PIMETA instantiation with OOP/Java .................... 6  
       1.3.2 PIMETA instantiation with AOP/AspectJ ................. 7  
   1.4 The Observer Design Pattern in Java and AspectJ .......... 7  
   1.5 PIMETA instantiation with the Observer versions ............ 8  
       1.5.1 PIMETA instantiation with the Java version of the Observer . . . . 8  
       1.5.2 PIMETA instantiation with the AspectJ version of the Observer 12  

A Java Feature and Dependency Types .............................. 13  

B AspectJ Feature and Dependency Types ............................ 17
[This page was intentionally left blank]
<table>
<thead>
<tr>
<th>Figura</th>
<th>Descripción</th>
<th>Página</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>PIMETA Meta-class diagram</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>PIMETA well-formedness rules in OCL</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>PIMETA well-formedness rules auxiliary operations in OCL</td>
<td>5</td>
</tr>
<tr>
<td>1.4</td>
<td>PIMETA instantiation extract for Java</td>
<td>6</td>
</tr>
<tr>
<td>1.5</td>
<td>PIMETA instantiation extract for AspectJ</td>
<td>7</td>
</tr>
<tr>
<td>1.6</td>
<td>Observer - Java version class diagram</td>
<td>8</td>
</tr>
<tr>
<td>1.7</td>
<td>Observer - AspectJ version class diagram</td>
<td>9</td>
</tr>
<tr>
<td>1.8</td>
<td>Screen class source code from the Observer pattern in Java</td>
<td>10</td>
</tr>
<tr>
<td>1.9</td>
<td>PIMETA instantiation with the Screen class of the Observer in Java</td>
<td>11</td>
</tr>
<tr>
<td>1.10</td>
<td>Screen class source code from the Observer pattern in AspectJ</td>
<td>12</td>
</tr>
<tr>
<td>1.11</td>
<td>PIMETA instantiation with the Screen class of the Observer in AspectJ</td>
<td>12</td>
</tr>
</tbody>
</table>
[This page was intentionally left blank]
**Lista de Tabelas**

<table>
<thead>
<tr>
<th>A.1</th>
<th>Java Feature Types</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.2</td>
<td>Java Dependency Types</td>
<td>15</td>
</tr>
<tr>
<td>B.1</td>
<td>AspectJ Feature Types</td>
<td>18</td>
</tr>
<tr>
<td>B.2</td>
<td>AspectJ Features (cont.)</td>
<td>19</td>
</tr>
<tr>
<td>B.3</td>
<td>AspectJ Dependency Types</td>
<td>20</td>
</tr>
<tr>
<td>B.4</td>
<td>AspectJ Dependency Types (cont.)</td>
<td>21</td>
</tr>
</tbody>
</table>
This chapter presents the PIMETA and its instantiation with OOP and AOP. It also presents its instantiation with the functional equivalent implementations of the Observer Design Pattern in Java and AspectJ, to show how PIMETA can be instantiated with any of these languages and paradigms, and how effectively their modularity differences are evidenced.
1.1 Introduction

In this chapter, a meta-model for describing the structural relationships of software systems, required to perform modularity assessments, across paradigm borders, will be proposed. The constructs of a given development language, either textual or graphical, implementing a paradigm, are mapped to elementary paradigm-independent concepts. Those concepts (e.g. features and dependencies) are represented within a single layer of abstraction, instead of being scattered throughout several layers, like for instance in a system modeled with several UML diagrams.

The meta-model is instantiated either from source code or from design models using appropriate loaders (transformers). The resulting model can be assessed on a paradigm independent fashion, since the meta-model only includes generic concepts, which are omnipresent in all paradigms. Since the meta-model is expressed as an UML meta-class diagram, modularity metrics can then be defined over the meta-model using a formal language, the Object Constraint Language (OCL) [OMG03b], like proposed in [Abr01]. OCL is also used for expressing well-formedness rules for the meta-model itself. To add support for a new language, or paradigm, a mapping of its specific constructs to the core concepts represented in the meta-model can be proposed. This possibility makes the meta-model based approach to metrics definition flexible.

In the next section the paradigm-independent meta-model(PIMETA) [BA07] is presented.

In section 1.3 the PIMETA will be instantiated with Java and AspectJ, to demonstrate how the specific constructs of different languages and paradigms can be mapped to its elementary modularity concepts.

In section 1.4 the two functional equivalent implementations, in Java and AspectJ, of the Observer Design pattern, are presented. These two languages are good representatives of the object-oriented and of the aspect-oriented paradigms, respectively.

These implementations will be the ground for section 1.5, where PIMETA is instantiated with both versions, to demonstrate how it can be instantiated by any system implemented with either Java or AspectJ, and its effectiveness in evidencing the modularity differences between these languages and the paradigms they represent.

1.2 Paradigm Independent Meta-model

1.2.1 PIMETA description

Modularity is an architectural property with impact on software maintenance and reusability [Pre00]. From a modularity perspective, a system is composed of features, which interact among each other, originating dependencies. Features are modular, and consequently called modules, when they have the capability to aggregate other features...
to logically organize them, like packages, classes or operations. Features are atomic when they do not allow the aggregation of other features (e.g. a parameter or a class attribute).

The meta-model depicted in figure 1.1 can be logically divided into two different parts. This division is represented by the double line and is inspired by the *powertypes* concept [Hal04].

The upper part, composed by the *Paradigm*, *Language*, *FeatureType*, *ModularFeatureType*, *AtomicFeatureType* and *DependencyType* meta-classes, allows representing paradigms and languages, the kind of features and dependencies they offer, and how they can be organized. The *Paradigm* meta-class is meant to represent paradigms like OOP or AOP. The *Language* meta-class represents languages like Java or AspectJ. Each language implements a specific paradigm, and one language can extend another (e.g. AspectJ extends Java), thus enabling the reuse of the base language definitions. The abstract meta-class *FeatureType* represents the types of features that languages offer. *ModularFeatureType* represents the different modules offered by languages (e.g. the class and package modules are offered by Java). A *ModularFeatureType* may contain any other *FeatureType* (e.g. a class may contain an attribute or another class in Java). *AtomicFeatureType* represents all non-modular features offered by languages (e.g. interface declaration or inheritance implementation are offered by Java). Finally, the *DependencyType* meta-class represents the different dependency types that can exist among features, provided by languages (e.g. an operation calls another operation in Java).

![PIMETA Meta-class diagram](image-url)

**Figura 1.1: PIMETA Meta-class diagram**
The lower part, composed by the Feature, ModularFeature, AtomicFeature and Dependency meta-classes, allows the representation of the system to be analyzed, its concrete features, dependencies and organization. The abstract meta-class Feature represents the features from the system under analysis. The ModularFeature meta-class is meant to be instantiated with the modules from the system under analysis (e.g. system classes or packages if Java is the language). A ModularFeature may contain any Feature, like a class may contain an attribute or another class in Java. The AtomicFeature meta-class should be instantiated by the features which are not modular, from the system under analysis, like an instance variable if Java is, again, the language used. The Dependency meta-class should be instantiated with the dependencies that exist in the system under analysis, like an operation calling another operation in a Java system. ModularFeature, AtomicFeature and Dependency instances always have a type from the corresponding meta-classes of the upper part of the meta-model.

1.2.2 PIMETA well-formedness rules

To ensure the meta-model’s consistency, several well-formedness rules need to be expressed. These were formalized as OCL invariants and can be found at figure 1.2. This is the same technique as that used in the UML series of standards for defining the corresponding meta-model [OMG03c] [OMG03a].

```
Language
1  // a language cannot extend itself either directly or indirectly
2  inv: not self.circularBaseLanguages()

FeatureType
3  // there are no circular aggregations (either direct or indirect)
4  inv: not self.circularModularFeatureTypes()

DependencyType
5  // a language must provide the feature types involved in a given dependency
6  inv: language.featureType->includesAllBag(self.origin, self.destination)
```

Figura 1.2: PIMETA well-formedness rules in OCL

The OCL invariants make use of several auxiliary operations, also defined in OCL, which can be found at figure 1.3.
Figura 1.3: PIMETA well-formedness rules auxiliary operations in OCL
1.3 PIMETA instantiation with OOP and AOP

To demonstrate how PIMETA can be used for representing software systems designed (input is a model) or written (input is source code) in different languages and paradigms, a full instantiation has been performed, for the Java and AspectJ languages.

In the following subsections, fragments of that instantiation are presented, represented as UML meta-object diagrams. The complete instantiation cannot be shown due to space limitation, since the corresponding diagrams are very large. Nevertheless, it is believed that the represented extracts are sufficient for a good understanding of the instantiation process.

1.3.1 PIMETA instantiation with OOP/Java

Figure 1.4 exhibits a partial representation of the PIMETA instantiation with the OOP paradigm, the Java language which implements it, as well as some of its features, namely the Package which aggregates other packages, and the Class which, in turn, aggregates other classes and variables, and the type of a variable, represented by the variable type dependency type binding the variable and the class which provides its type.

Further details on how and which Features and Dependencies, from the Java language, have been instantiated at PIMETA, can be found at appendix A.
1.3.2 PIMETA instantiation with AOP/AspectJ

Figure 1.5 shows a fragment of the resulting meta-object diagram from the PIMETA instantiation with AOP and AspectJ. In the history of programming languages and also of modeling languages, many proposals were made to extend existing languages while guaranteeing backward compatibility. Well-known cases are C++ (compatible with C) and AspectJ (compatible with Java). PIMETA provides a language extension mechanism (reflexive meta-dependency extends in figure 1.1). This mechanism allows reusing the meta-objects created for Java in the context of AspectJ. The specific features of AspectJ are the only ones that must then be added, such as the fact that a package may contain aspects and that an aspect may contain variables. The dependency corresponding to an aspect inheriting from a class is also represented, through the AspectInheritsFromClass meta-object.

Further details on how and which Features and Dependencies, from the AspectJ language, have been instantiated at PIMETA, can be found at appendix B.

1.4 The Observer Design Pattern in Java and AspectJ

The Observer Design Pattern [GHJV95], implemented both with Java and AspectJ by Hannemann and Kiczales [HK02], was chosen to show how PIMETA can be instantiated with systems from any of these two different languages and paradigms, and how their modularity differences can effectively be put in evidence by PIMETA. This choice has the following advantages:

i) its small size facilitates understanding;

ii) it complements the PIMETA instantiation examples started in previous subsections;
iii) the instantiated systems are functionally equivalent;
iv) the Observer pattern is a paradigmatic example of a way to avoid high coupling;
v) it was developed by experts with the purpose of putting in evidence the differences between both paradigms (OOP and AOP).

However, this example only encompasses a small subset of the features offered by both languages. Another disadvantage is the fact of not being a real world example.

The UML class diagram of the Observer Design Pattern implementation in Java is presented in figure 1.6. The UML class diagram of the Observer Design Pattern implementation in AspectJ is presented in figure 1.7.

1.5 PIMETA instantiation with the Observer versions

1.5.1 PIMETA instantiation with the Java version of the Observer

To demonstrate how PIMETA can be instantiated by a Java system, PIMETA was instantiated with the Java version of the Observer design pattern. An extract of that instantiation is shown at figure 1.9 where only the Screen class is represented, also due to space constraints. For simplicity and understandability reasons, we chose not to represent this instantiation as an object diagram but, instead, to represent it as a graph, based on the PIMETA meta-classes. For the same reasons, several objects are
PIMETA: PARADIGM INDEPENDENT META-MODEL

1.5. PIMETA instantiation with the Observer versions

Figura 1.7: Observer - AspectJ version class diagram

not shown, namely the Java objects needed by this instantiation, the profuse DependencyType meta-objects corresponding to the Dependency meta-objects represented, and the dependencies’ names.

The code corresponding to the Screen class from the Java implementation of the Observer design pattern is fully presented at figure 1.8. At this example, depicted in figure 1.9 there are atomic features, represented by a circle (i.e. Screen.name), modular features represented by a square (i.e. Screen.display) and features with an undetermined\(^1\) type are represented by a polygon (i.e. Iterator).

Each solid black line represents a containment relation, that is, each feature that contains another feature has a solid black line with an arrow pointing towards it (i.e. Screen contains Screen.display). Each dashed line represents a dependency, that is, when one feature dependends on another, a dashed line is drawn from the dependent towards the depended with an arrow pointing towards it (i.e. Screen.display calls System.out.println).

The big grey box represents the border of the Screen class. All the features outside this box do not belong to the Screen class, while the opposite is also true. In this example, all the dependencies that leave the box are responsible for the coupling of the Screen class, while those that remain inside the box are responsible for its cohesion.

For clarification, all features have their names fully qualified according to their container (i.e. Screen.Screen is the constructor of the Screen class). Also should be noticed that the Screen class is represented inside its own border for two reasons. One is for coherence on the graphical representation of the containment relations, and the other is because the class itself can be part of dependencies that may contribute either for coupling or cohesion, and having the class inside its own border makes it easy to see

\(^1\)Undetermined due to limitations of the automatic process which generated this graph.
1.5. PIMETA instantiation with the Observer versions

those that are directed towards the outside (coupling) and those that remain in the inside (cohesion).

This notation for the representation of features and dependencies from a system as a graph will be used at similar examples throughout this dissertation.

Each solid black line represents a containment relation, that is, each feature that contains another feature has a solid black line with an arrow pointing towards it. Each dashed line represents a dependency from the dependent towards the depended.

The big gray box represents the border of the Screen class. None of the features outside this box belong to the Screen class, while all the ones inside it do. In this example, all the dependencies that leave the box are responsible for the coupling of the Screen class, while those that remain inside the box are responsible for its cohesion.

Figura 1.8: Screen class source code from the Observer pattern in Java
Figura 1.9: PIMETA instantiation with the *Screen* class of the Observer in Java
1.5.2 PIMETA instantiation with the AspectJ version of the Observer

To demonstrate how PIMETA can be used for representing an AspectJ system, PIMETA was instantiated with the AspectJ version of the Observer design pattern. An extract of that instantiation is shown at figure 1.11 where only the Screen class is represented.²

Several objects are also not shown, namely, the AspectJ objects needed by this instantiation, the profuse DependencyType meta-objects corresponding to the Dependency meta-objects represented, and the dependencies’ names.

The code corresponding to the Screen class from the AspectJ implementation of the Observer design pattern is fully depicted at figure 1.10.

By looking at figures 1.9 and 1.11, the modularity differences introduced by AOP at this concrete pattern implementation are perceivable, being this capability one of the benefits of PIMETA.

²The notation used is the same described at section 1.5.1
This appendix presents the Java Feature Types, their mutual aggregation possibilities and their Dependency Types with which PIMETA is instantiated.
### ModularFeatureTypes
- Package
- Class
- Interface
- Method
- Constructor

### AtomicFeatureTypes
- Field
- LocalVariable
- Enumeration
- Parameter
- ReturnValue
- Exception

### Aggregations
- (Package, Package)
- (Package, Class)
- (Package, Interface)
- (Class, Class)
- (Class, Interface)
- (Class, Method)
- (Class, Constructor)
- (Class, Field)
- (Class, Enumeration)
- (Method, LocalVariable)
- (Method, Class)
- (Method, Parameter)
- (Method, ReturnValue)
- (Method, Enumeration)
- (Constructor, LocalVariable)
- (Constructor, Class)
- (Constructor, Parameter)
- (Constructor, Enumeration)
- (Interface, Method)
- (Interface, Constructor)
- (Interface, Enumeration)
- (Interface, Field)
- (Interface, Class)
- (Interface, Interface)

### Tabela A.1: Java Feature Types
DependencyTypes
ClassInheritance : (Class, Class)
ClassInterfaceImplementation : (Class, Interface)
ClassImportsPackage : (Class, Package)
ClassImportsClass : (Class, Class)
ClassCallsConstructor : (Class, Constructor)
InterfaceInheritance : (Interface, Interface)
InterfaceImportsPackage : (Interface, Package)
InterfaceImportsClass : (Interface, Class)
MethodUsesField : (Method, Field)
MethodUsesEnumeration : (Method, Enumeration)
MethodCallsMethod : (Method, Method)
MethodCallsConstructor : (Method, Constructor)
MethodRaisesException : (Method, Exception)
ConstructorUsesField : (Constructor, Field)
ConstructorUsesEnumeration : (Constructor, Enumeration)
ConstructorCallsMethod : (Constructor, Method)
ConstructorCallsConstructor : (Constructor, Constructor)
ConstructorRaisesException : (Constructor, Exception)
ReturnValueType : (ReturnValue, Class)
ParameterType : (Parameter, Class)
ExceptionType : (Exception, Class)
FieldType : (Field, Class)
LocalVariableType : (LocalVariable, Class)

Tabela A.2: Java Dependency Types
This appendix presents the AspectJ Feature Types, their mutual aggregation possibilities and their Dependency Types with which PIMETA is instantiated.
### B. AspectJ Feature and Dependency Types

**ModularFeatureTypes**
- Aspect
- AdviceBefore
- AdviceAround
- AdviceAfter
- InterTypeOperation
- PointcutDefinition

**AtomicFeatureTypes**
- InterTypeField
- DeclareParents
- DeclareWarning
- DeclareError
- DeclareSoft
- DeclarePrecedence
- DeclareAtType
- DeclareAtMethod
- DeclareAtConstructor
- DeclareAtField
- Pointcut

---

**Tabla B.1: AspectJ Feature Types**
**Aggregations**

(Package, Aspect)
(Class, Aspect)
(Aspect, Aspect)
(Aspect, Class)
(Aspect, Interface)
(Aspect, Method)
(Aspect, Field)
(Aspect, Enumeration)
(Aspect, AdviceBefore)
(Aspect, AdviceAround)
(Aspect, AdviceAfter)
(Aspect, InterTypeOperation)
(Aspect, PointcutDefinition)
(Aspect, InterTypeField)
(Aspect, DeclareParents)
(Aspect, DeclareWarning)
(Aspect, DeclareError)
(Aspect, DeclareSoft)
(Aspect, DeclarePrecedence)
(Aspect, DeclareAtType)
(Aspect, DeclareAtMethod)
(Aspect, DeclareAtConstructor)
(Aspect, DeclareAtField)
(PointcutDefinition, Pointcut)
(PointcutDefinition, Parameter)
(AdviceBefore, Pointcut)
(AdviceBefore, LocalVariable)
(AdviceBefore, Class)
(AdviceBefore, Parameter)
(AdviceBefore, ReturnValue)
(AdviceAround, Pointcut)
(AdviceAround, LocalVariable)
(AdviceAround, Parameter)
(AdviceAround, ReturnValue)
(AdviceAfter, Pointcut)
(AdviceAfter, LocalVariable)
(AdviceAfter, Class)
(AdviceAfter, Parameter)
(AdviceAfter, ReturnValue)
(InterTypeOperation, Pointcut)
(InterTypeOperation, LocalVariable)
(InterTypeOperation, Class)
(InterTypeOperation, Parameter)
(InterTypeOperation, ReturnValue)
Dependency Types

AspectInheritsFromClass : (Aspect, Class)
AspectInheritsFromAspect : (Aspect, Aspect)
AspectImplementsInterface : (Aspect, Interface)
AspectImportsPackage : (Aspect, Package)
AspectImportsClass : (Aspect, Class)
AspectCallsConstructor : (Aspect, Constructor)
MethodUsesInterTypeField : (Method, InterTypeField)
MethodCallsInterTypeOperation : (Method, InterTypeOperation)
ConstructorUsesInterTypeField : (Constructor, InterTypeField)
ConstructorCallsInterTypeOperation : (Constructor, InterTypeOperation)
AdviceBeforeUsesField : (AdviceBefore, Field)
AdviceBeforeUsesEnumeration : (AdviceBefore, Enumeration)
AdviceBeforeCallsMethod : (AdviceBefore, Method)
AdviceBeforeCallsConstructor : (AdviceBefore, Constructor)
AdviceBeforeRaisesException : (AdviceBefore, Class)
AdviceBeforeUsesPointcutDefinition : (AdviceBefore, PointcutDefinition)
AdviceBeforeUsesInterTypeField : (AdviceBefore, InterTypeField)
AdviceBeforeCallsInterTypeOperation : (AdviceBefore, InterTypeOperation)
AdviceAroundUsesField : (AdviceAround, Field)
AdviceAroundUsesEnumeration : (AdviceAround, Enumeration)
AdviceAroundCallsMethod : (AdviceAround, Method)
AdviceAroundCallsConstructor : (AdviceAround, Constructor)
AdviceAroundRaisesException : (AdviceAround, Class)
AdviceAroundUsesPointcutDefinition : (AdviceAround, PointcutDefinition)
AdviceAroundUsesInterTypeField : (AdviceAround, InterTypeField)
AdviceAroundCallsInterTypeOperation : (AdviceAround, InterTypeOperation)
AdviceAfterUsesField : (AdviceAfter, Field)
AdviceAfterUsesEnumeration : (AdviceAfter, Enumeration)
AdviceAfterCallsMethod : (AdviceAfter, Method)
AdviceAfterCallsConstructor : (AdviceAfter, Constructor)
AdviceAfterRaisesException : (AdviceAfter, Class)
AdviceAfterUsesPointcutDefinition : (AdviceAfter, PointcutDefinition)
AdviceAfterUsesInterTypeField : (AdviceAfter, InterTypeField)
AdviceAfterCallsInterTypeOperation : (AdviceAfter, InterTypeOperation)
InterTypeOperationUsesField : (InterTypeOperation, Field)
InterTypeOperationUsesEnumeration : (InterTypeOperation, Enumeration)
InterTypeOperationCallsMethod : (InterTypeOperation, Method)
InterTypeOperationCallsConstructor : (InterTypeOperation, Constructor)
InterTypeOperationRaisesException : (InterTypeOperation, Class)
InterTypeOperationDestinationClass : (InterTypeOperation, Class)
InterTypeOperationDestinationAspect : (InterTypeOperation, Aspect)
InterTypeOperationUsesInterTypeField : (InterTypeOperation, InterTypeField)
InterTypeOperationCallsInterTypeOperation : (InterTypeOperation, InterTypeOperation)
InterTypeFieldType : (InterTypeField, Class)
InterTypeFieldDestinationClass : (InterTypeField, Class)
InterTypeFieldDestinationAspect : (InterTypeField, Aspect)
**DependencyTypes**

DeclareParentsParentClass : (DeclareParents, Class)
DeclareParentsParentInterface : (DeclareParents, Interface)
DeclareParentsParentAspect : (DeclareParents, Aspect)
DeclareParentsChildClass : (DeclareParents, Class)
DeclareParentsChildInterface : (DeclareParents, Interface)
DeclareParentsChildAspect : (DeclareParents, Aspect)
DeclareParentsImplementedInterface : (DeclareParents, Interface)
DeclareParentsClassImplementingInterface : (DeclareParents, Class)
DeclareParentsAspectImplementingInterface : (DeclareParents, Class)
DeclareWarningUsesPointcutDefinition : (DeclareWarning, PointcutDefinition)
DeclareErrorUsesPointcutDefinition : (DeclareError, PointcutDefinition)
DeclareSoftUsesPointcutDefinition : (DeclareSoft, PointcutDefinition)
DeclarePrecedenceUsesAspect : (DeclarePrecedence, Aspect)
DeclareAtTypeUsesClass : (DeclareAtType, Class)
DeclareAtTypeUsesAspect : (DeclareAtType, Aspect)
DeclareAtMethodUsesPointcutDefinition : (DeclareAtMethod, PointcutDefinition)
DeclareAtConstructorUsesClass : (DeclareAtConstructor, Class)
DeclareAtFieldUsesClass : (DeclareAtField, Class)
PointcutUsesPointcutDefinition : (Pointcut, PointcutDefinition)
PointcutUsesPackage : (Pointcut, Package)
PointcutUsesClass : (Pointcut, Class)
PointcutUsesInterface : (Pointcut, Interface)
PointcutUsesAspect : (Pointcut, Aspect)
PointcutUsesMethod : (Pointcut, Method)
PointcutUsesConstructor : (Pointcut, Constructor)
PointcutUsesField : (Pointcut, Field)

**Tabela B.4: AspectJ Dependency Types (cont.)**
Bibliografia


[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


