

# Towards Paradigm-Independent Software Assessment

Sérgio Bryton, Fernando Brito e Abreu  
FCT/UNL, CITI, QUASAR research group  
{bryton|fba}@di.fct.unl.pt

## Abstract

*The milestones of the history of software development are paradigm shifts. Each paradigm brought its own features and new ways of composing them to assemble software systems. Understanding the impact of paradigm shifts encompasses making comparisons among systems built with different paradigms. Performing this kind of assessments is a very difficult exercise since the characterization of software assets is generally performed using paradigm-specific quantifications.*

*In this paper we propose a metamodel for describing software products (either source code or design models) that is paradigm-independent. This metamodel, combined with a formal quantification approach, can help performing paradigm-independent software systems assessment. We illustrate the use of the proposed metamodel on a case study comparing functionally-equivalent systems produced with OOP and AOP.*

## 1. Introduction

New software development paradigms usually emerge and grow on a bottom-up fashion, following a sequence of overlapping concern phases. This evolution path can be observed through surveys on researchers' and leading practitioners' work evidences (conference communications, journal papers, books, etc). First, computer scientists come up with new programming metaphors usually bundled within programming languages; then commercial software-houses and open-source communities conceive development environments supporting those new languages; software engineering researchers then focus their attention on analysis and design concerns; methodologists start to be concerned on supporting new kinds of development and assessment processes; information systems people develop new arguments for business value of technology migration; and finally everybody gets worried on evolution and legacy systems. This was the case for the procedural and

object-oriented paradigms and the whole story will most probably repeat for the more recent one – the aspect-oriented paradigm. First of all notice that each paradigm “wave” is usually long. They often span over more than a decade. This long period of time corresponds to the *ad nauseam* exhaustion of each of the above mentioned research concerns in the context of the paradigm in hand.

If we take a specific concern from the above list, we can observe, throughout time, a strong evidence of periodic research output bursts, each one corresponding to the emergence of a new paradigm. For instance, regarding the concern of the quantitative assessment of software products we had a wave of interest centered on the procedural paradigm until the mid-eighties, then on the object-oriented paradigm until the late nineties and currently a wave is rising around the aspect-oriented paradigm. While the creation of new research niches may be interesting for PhD students searching for a potential new research topic, there are several perverse side-effects associated with this paradigm-shift phenomenon both on the research and practitioner's sides.

Let us look at the research side first. When performing a state of the art survey for finding who is doing what and where your bright new ideas can be published, you will often disregard or even avoid the exhausted old gold lode. In other words, either you ignore the work developed in the same topic, but in the context of a previous paradigm, more than a decade old as we saw, or you look at it with haughtiness to quickly conclude that the whole work performed in the past simply does not apply for the new paradigm. This is a very convenient conclusion for new researchers eager for publishing and creating their own space. Besides, comparing things across the paradigm border is a hard nut to crack, since new paradigm heralds tend to stress innovative paradigm characteristics and not the commonalities with the “old” paradigm. You have to be an expert on both (old and new) paradigms to understand the whole picture. The perverse effects of just looking at the new paradigm are that good reusable ideas of the past are forgotten and new

researchers often reinvent the wheel. Also reprehensible is to adapt good old ideas, presenting them as innovative (i.e. not crediting their origin).

Now let us look to the world of business. In smaller businesses where you can afford the risk of being adventurous, you are usually in the hands of hype marketers when it comes to paradigm shift. You are simply left in the dark because the quantitative studies bringing evidence on the technological and consequent managerial benefits of migration are very scarce [1, 2]. This unfortunate tradition for a lack of insufficient validation of claims has been reported for more than a decade [3, 4]. It has been pointed out that one of the strongest reasons for this scarcity is the lack of cooperation among academic researchers and practitioners facing the migration dilemma. As far as we know very few joint works of this sort were published [5].

On the other hand, although researchers gather to discuss migration concerns, their focus, although technical, is qualitative in nature (e.g. [6] reports on aspect oriented software development and software engineering researchers with a focus on reverse engineering, program comprehension, software evolution and software maintenance confronting their thoughts). However, for software managers, concerned with training, development and maintenance costs, ROI<sup>1</sup> and other economic figures, a pure technical and qualitative discussion is fruitless. A responsible CIO<sup>2</sup> or project manager will not take unplanned risks. He will let others prove that the technology supporting the new paradigm is mature enough before streamlining it in his development shop. In the end, strong players, such as big software vendors with their huge marketing sales force, will be the ones to impose the evolution track, after very careful technology maturation. By that time the careful CIO may have lost a competitive advantage.

Summing up, this *status quo* of long transition periods for paradigm shift (widespread adoption) will not change unless researchers do their share of the work: provide quantitative evidence that migration is cost-effective. This proof can be performed partly if researchers show, beyond doubt, that a system developed with a new paradigm is easier to understand and maintain than a functionally equivalent<sup>3</sup> one, built with the previous paradigm (see for instance [7]). To be adequate and fair, this kind of quantitative comparisons require that you do not compare apples with oranges. If, for instance, you are interested in

comparing the object-oriented programming (OOP) and the aspect-oriented programming (AOP) paradigms, then you should use the same ruler for measuring (quantifying) the characteristics of functionally-equivalent OOP and AOP artifacts. However, for reasons presented above in this paper, academic researchers are too much often tempted to do the opposite, that is, suggesting new rulers [8-12]. The justification for proposing specific metrics is that each paradigm is supported in formalisms that offer different constructs, introducing new sorts of containment and dependency relations.

To compare apples with oranges, we must be able to abstract from their differences and concentrate on their commonalities. After all, apples and oranges have previously been shown to be remarkably similar [13]. In our case this means that we should concentrate on the empiric concepts that explain the software properties under measurement (see the representational theory of measurement in [14]). For instance, if we are interested in evaluating modularity, then we should consider modules, items belonging to them and their internal and external dependencies. Each paradigm (and in fact each reification language), will have its own instantiation of those concepts.

The objective of this paper is to propose a metamodel for describing the structural relationships of software systems that are required to perform modularity assessments across paradigm borders. The constructs of a given development language (either textual or graphical) implementing a paradigm are mapped to basic paradigm-independent concepts. Those concepts (e.g. modules 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 metamodel 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 metamodel only includes generic concepts, which are omnipresent in all paradigms. Since the metamodel is expressed as a UML class diagram, modularity metrics can then be defined over the metamodel using a formal language as the Object Constraint Language (OCL) [15], like proposed in [16]. OCL is also used for expressing well-formedness rules for the metamodel itself. To consider a new formalism or paradigm we only require a mapping of its specific constructs to the common concepts represented in the metamodel.

This paper is organized as follows. In the next section we present the **paradigm-independent metamodel** (PIMETA). In section 2, PIMETA is

---

<sup>1</sup> - Return On Investment

<sup>2</sup> - Chief Information Officer

<sup>3</sup> - Two systems are functionally equivalent when their features produce the same results in spite of having a different design.

instantiated with *Java* and *AspectJ*. In section 3 we present a case study: a functionally-equivalent program implemented both with *Java* and *AspectJ*, which will be the ground for section 4 where PIMETA is instantiated with both versions. In section 5 we discuss some related work and, in section 6, we sum up with some conclusions and the identification of future work.

## 2. Paradigm-independent metamodel

### 2.1. PIMETA description

Modularity is an architectural property that has an impact on software maintenance and reusability. From a modularity perspective, a system is composed of features, which interact among each other, originating dependencies.

Features can be 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 metamodel represented in Figure 1 can be logically divided into two different parts, and is inspired by the powertypes concept [17]. The upper part allows representing paradigms and formalisms, the kind of features and dependencies they offer (traduced by the four *XptoType* powertype meta-classes) and how they can be organized. The lower part allows representing the system under analysis, its concrete features, dependencies and organization.

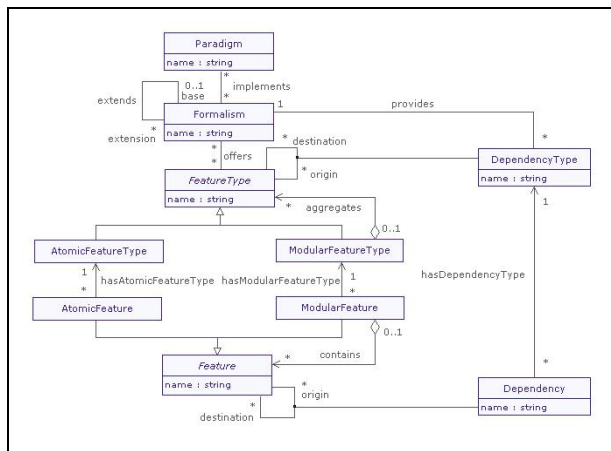


Figure 1: PIMETA Meta-class diagram

The first part is composed by the *Paradigm*, *Formalism*, *FeatureType*, *ModularFeatureType* and *DependencyType* meta-classes. The *Paradigm* meta-class may represent

paradigms like “OOP” or “AOP”. The *Formalism* meta-class may represent formalisms like “*Java*” or “*AspectJ*”. Each formalism implements a specific paradigm, and one formalism can extend another (e.g. “*AspectJ*” extends “*Java*”), thus enabling the reuse of the base formalism definitions. The abstract meta-class *FeatureType* represents the types of features that formalisms offer. *ModularFeatureType* represents the different modules offered by formalisms (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 formalisms (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 formalisms (e.g. an operation “calls” another operation in *Java*).

The second part is composed by the *Feature*, *ModularFeature*, *AtomicFeature* and *Dependency* meta-classes. 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 formalism). 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 remaining features from the system under analysis, like an inheritance declaration if *Java* is, again, the formalism 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 have always a type given from the correspondent meta-classes from the first part (the powertypes part) of the metamodel.

### 2.2. PIMETA well-formedness rules

To ensure metamodel consistency, we need to express several well-formedness rules. These were formalized as OCL invariants. This is the same technique as that used in the UML series of standards for defining the corresponding metamodel [18, 19].

#### Formalism

// a formalism cannot extend itself either directly or indirectly

inv: not self.circularBaseFormalisms()

#### FeatureType

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

### DependencyType

// a formalism must provide the feature types involved in a given dependency

inv: formalism.featureType->includesAll(Bag(self.origin, self.destination))

We reproduce below the operations used above in invariants' specification:

```
// auxiliary operation to support implementing circularBaseFormalisms()
Formalism::circularBaseFormalismsAux(setF:Set(Formalism)):
    Bag(Formalism) =
    if (setF->includes(self.base)) then
        setF->asBag()->including(self.base)
    else
        self.base.circularBaseFormalismsAux(setF->including(self.base))
    endif

// predicate that detects if a circular base reference exists
Formalism::circularBaseFormalisms(): Boolean =
    circularBaseFormalismsAux(oclEmpty(Set(Formalism)))->size() <>
    circularBaseFormalismsAux(oclEmpty(Set(Formalism)))->asSet()->size()

// aux. operation to support implementing circularModularFeatureTypes()
FeatureType::circularModularFeatureTypesAux
    (setF: Set(ModularFeatureType)) : Bag(ModularFeatureType) =
    if (setF->includes(self.modularFeatureType)) then
        setF->asBag()->including(self.modularFeatureType)
    else
        self.modularFeatureType.circularModularFeatureTypesAux
            (setF->including(self.modularFeatureType))
    endif

// predicate that detects if a circular aggregation exists
FeatureType::circularModularFeatureTypes(): Boolean =
    circularModularFeatureTypesAux
        (oclEmpty(Set(ModularFeatureType)))->size() <>
    circularModularFeatureTypesAux
        (oclEmpty(Set(ModularFeatureType)))->asSet()->size()
```

### 2.3. PIMETA instantiation with OOP / Java

To demonstrate how PIMETA can be used for representing software systems designed (input is a model) or written (input is source code) in formalisms representing different paradigms, we have performed a full instantiation exercise for the *Java* and *AspectJ* languages, perhaps the best surrogates of the OOP and AOP paradigms, respectively. In the following subsections, we present fragments of that instantiation exercise, represented as UML meta-object diagrams. The complete instantiation cannot be shown due to space limitation, since the corresponding diagrams are very large. Nevertheless, we believe the represented extracts are sufficient for a good understanding of the proposal. In the text explaining the diagrams we use inverted commas for identifying meta-objects represented graphically.

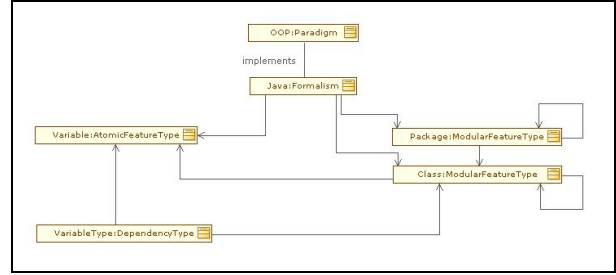


Figure 2: PIMETA instantiation extract for *Java*

In Figure 2, PIMETA is instantiated with the OOP paradigm, the *Java* formalism which implements it, and some *Java* features, namely the “Package” which aggregates other packages and the “Class” which, in turn, aggregates other classes. Finally, the fact that a “Variable” has a type, which is a “Class”, is represented by the “VariableType” meta-object.

### 2.4. PIMETA instantiation with AOP / AspectJ

In this section we demonstrate how PIMETA can be instantiated with *AspectJ* and AOP concepts. A fragment of the resulting meta-object diagram is shown in Figure 3.

In the history of programming languages and also of modeling languages, many proposals were made to extend existing formalisms while guarantying backward compatibility. Well-known cases are *C++* (compatible with *C*) and *AspectJ* (compatible with *Java*). PIMETA provides a formalism extension mechanism (reflexive meta-dependency “extends” in Figure 1). This mechanism allows reusing the powertype 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.

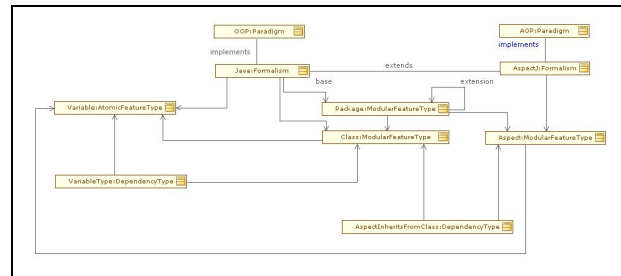


Figure 3: PIMETA instantiation extract for *AspectJ*

### 3. Case Study

A simple example of logging, implemented both with *Java* and *AspectJ*, developed by Laddad [20], was chosen to demonstrate PIMETA instantiation with the *Java* and *AspectJ* languages and to demonstrate how their modularity differences can effectively be put in evidence by PIMETA. This case study has the following advantages: (i) it is easily understandable, (ii) it complements the PIMETA instantiation exercise started in previous sub-sections, (iii) the instantiated systems are functionally equivalent and (iv) logging is one of the most used features to explain AOP and its modularity benefits over OOP [21]. However, due to space constraints, this example has the disadvantage of being small, thus only considering a part of the features offered by from both formalisms.

The chosen example, presented in Figure 4, is a shopping-cart model, in which logging is implemented in two different ways, the conventional way and the *AspectJ* way, both using the standard *Java* Logging API [22].

The shopping-cart system is composed by 4 classes: the *Item* class which models a shopping item that can be purchased; the *ShoppingCart* class which models a shopping cart; the *Inventory* class which models the shop inventory, and the *ShoppingCartOperator* class which manages the shopping cart.

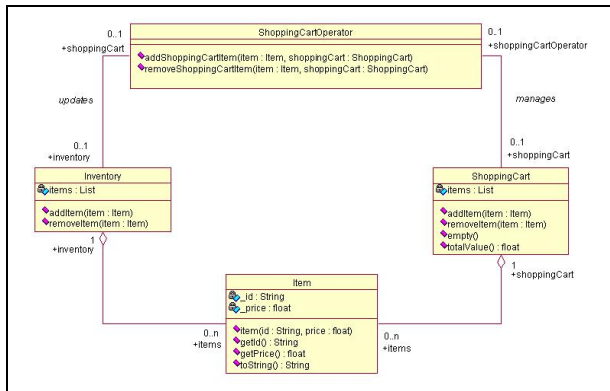


Figure 4: Shopping cart case-study

The conventional implementation of logging embeds the logging invocations in each module, thus originating code tangling.

The *AspectJ* implementation of logging adds the aspect *TraceAspect* to the system, which modularizes the logging concern by implementing the logging functionality, thus eliminating the need for core modules to carry any logging-related code.

### 4. Systems representation

#### 4.1. PIMETA instantiation with a OOP system

To demonstrate how PIMETA can be used for representing a *Java* system, PIMETA was instantiated with the “conventional” logging version of our case study. An extract of that instantiation is shown on Appendix A where only the *Item* class, presented in Figure 5 and its required external features are represented. Some instantiated *Java* concepts (powertype meta-objects) are represented such as the *ModularFeatureType* meta-objects required by the represented features. The profuse *DependencyType* meta-objects corresponding to the *Dependency* meta-objects represented are not shown.

```

package traditionalLogging;

import java.util.logging.*;

public class Item {
    private String _id;
    private float _price;
    static Logger _logger = Logger.getLogger("trace");

    public Item(String id, float price) {
        _id = id;
        _price = price;
    }

    public String getID() {
        _logger.logp(Level.INFO, "Item", "getID", "Entering");
        return _id;
    }

    public float getPrice() {
        _logger.logp(Level.INFO, "Item", "getPrice", "Entering");
        return _price;
    }

    public String toString() {
        _logger.logp(Level.INFO, "Item", "toString", "Entering");
        return "Item: " + _id;
    }
}
  
```

Figure 5: Traditional logging: *Item* class

For better apprehension, the meta-object diagram into Appendix A was split into several fragments.

#### 4.2. PIMETA instantiation with a AOP system

To demonstrate how PIMETA can be used for representing an *AspectJ* system, PIMETA was instantiated with the “aspectized” logging version of our case study, of which an extract is presented in

Appendix B. Therefore, only the *Item* class and the *TraceAspect* aspect, described in Figure 6 and Figure 7, respectively, and their required external features, are presented. The PIMETA instantiation with *AspectJ* is not represented, except for the *ModularFeatureType* classes required by the represented features. For sake of understandability, the meta-object diagram was split into several fragments, as represented in Appendix B.

```

package aspectizedLogging;

public class Item {
    private String _id;
    private float _price;

    public Item(String id, float price) {
        _id = id;
        _price = price;
    }

    public String getID() {
        return _id;
    }

    public float getPrice() {
        return _price;
    }

    public String toString() {
        return "Item: " + _id;
    }
}

```

Figure 6: Aspectized logging: *Item* class

```

package aspectizedLogging;
import java.util.logging.*;
import org.aspectj.lang.*;

public aspect TraceAspect {
    private Logger _logger = Logger.getLogger("trace");

    pointcut traceMethods()
        : execution(* *.*(..)) && !within(TraceAspect);

    before() : traceMethods() {
        Signature sig = thisJoinPointStaticPart.getSignature();
        _logger.logp(Level.INFO, sig.getDeclaringType().getName(),
            sig.getName(), "Entering");
    }
}

```

Figure 7: Aspectized logging: *TraceAspect* aspect

As we can see from the *Appendix B* (aspectized logging), the external and internal dependencies implemented by the *Item* class and the *TraceAspect* aspect are completely different from those used by the *Item* class in *Appendix A* (traditional logging) to accomplish exactly the same functionality with a

different paradigm and formalism. This perception is easily achieved because the original system was mapped into PIMETA, which shows us the system from a modularity perspective within a single layer of abstraction. This perception would not be so easy by only looking at the source code.

## 5. Related work

Garcia et al. presented an interesting comparison between two functionally-equivalent systems, implemented with OOP and AOP [23]. This comparison is focused, among other software architecture properties, on coupling and cohesion.

To quantify coupling and cohesion on both systems, object-oriented specific metrics like CBC<sup>4</sup> and LCOO<sup>5</sup> were tailored to AOP [24]. This is a good example of how researchers on metrication are dealing with paradigm shifts. They often adapt “old” metrics to cope with the innovative features proposed in formalisms supporting the new paradigm. However, the basic architectural concepts such as coupling, cohesion and modularization have not changed over the last decades. If we base our metrics on those basic concepts, instead of proposing tailored-based metrics, we can compare systems built in different languages and paradigms. Furthermore, also for comparability and experiment replication sake, we must guarantee precision on metrics definition, which can only be achieved through formality.

## 6. Conclusions and future work

The proposed metamodel is not meant to be used for designing a system, since it lacks the adequate semantic expressiveness present in other metamodels like the UML2 one. Instead, the PIMETA metamodel is intended to be instantiated with data from source code or design models in no matter what formalism, as long as the containment and dependency relationships allowed in that formalism are expressed as powertype instances.

The proposed metamodel is compact and understandable, but representing its instantiation using an object diagram very easily becomes unfeasible even for small systems, as can be concluded by the partial examples presented in the annexes. This means that if we want to observe the architectural modularity of a given system, we should use a different visualization technique. We are currently working on a PIMETA-based 3D representation technique, allied with elision

<sup>4</sup> - Coupling between components

<sup>5</sup> - Lack of cohesion in operations

capabilities, that is expected to improve software architecture visualization, thus mitigating the previous problem.

Through an instantiation exercise based on a shopping-cart case-study written both in *Java* and *AspectJ*, we have shown how the PIMETA metamodel can be used for representing systems conceived with different paradigms, namely OOP and AOP. We intend to further demonstrate PIMETA paradigm and formalism independency with larger case studies and other paradigms and formalisms, like functional programming and logic programming. We are currently using PIMETA to instantiate UML2 models as well. We plan to show its usefulness for representing the modularity of other kinds of systems besides software ones, like computer infrastructures, for instance.

PIMETA offers provisions for performing a formal comparison of software systems using the Metamodel-Driven Measurement Approach (M2DM) originally proposed in [16]. Given the expressiveness and preciseness of OCL for expressing modularity metrics and the simplicity of the PIMETA semantics, combined with its intended multi-paradigm instantiation ability, PIMETA appears to be an adequate ground for basing paradigm and formalism-independent modularity assessments. The benefit of independence allows comparisons across paradigm boundaries.

Our initial motivation when we started defining PIMETA was to provide a ground for the quantification of the modularity benefits obtained with AOP versus OOP of which there is only anecdotal evidence [25]. We now tend to believe that PIMETA can facilitate a much larger plethora of other experimental software engineering studies in areas such as software evolution or change impact evaluation, besides modularity assessment.

## References

- [1] R. L. Glass, I. Vessey, and V. Ramesh, "Research in software engineering: an analysis of the literature," *Information and Software Technology*, vol. 44, pp. 491-506, 2002.
- [2] V. Ramesh, R. L. Glass, and I. Vessey, "Research in computer science: an empirical study," *Information and Software Technology*, vol. 70, pp. 165-176, 2004.
- [3] W. Tichy, P. Lukowicz, L. Prechelt, and E. A. Heinz, "Experimental Evaluation in Computer Science: A Quantitative Study," *Journal for Systems and Software*, vol. 28, pp. 9-18, 1995.
- [4] M. V. Zelkowitz and D. Wallace, "Experimental Validation in Software Engineering," *Journal of Information and Software Technology*, vol. 39, pp. 735-743, 1997.
- [5] D. J. Leigh, C. J. Theaker, N. Blackwood, and R. Mason, "Measuring the Impact of Migration to an Object Oriented Paradigm," presented at Workshop on Quantitative Approaches in Object Oriented Software Engineering (QAOOSE'1999), Lisboa, Portugal, 1999.
- [6] L. Moonen, G. Ramalingam, and S. Clarke, "Aspects For Legacy Applications," Schloss Dagstuhl, Germany 26th to July 29th 2006.
- [7] A. v. Deursen, M. Marin, and L. Moonen, "AJHotDraw: A showcase for refactoring to aspects," presented at Linking Aspect Technology and Evolution Workshop (LATE) at AOSD'2005, Chicago, USA, 2005.
- [8] J.-F. Gélinas, M. Badri, and L. Badri, "A Cohesion Measure for Aspects," *Journal of Object Technology*, vol. 5, pp. 97-114, 2006.
- [9] A. A. Zakaria and H. Hosny, "Metrics for Aspect-Oriented Software Design," presented at Workshop on Aspect-Oriented Modeling with UML at AOSD'2003, Boston, USA, 2003.
- [10] J. Zhao, "Measuring coupling in aspect-oriented systems," presented at 10th International Software Metrics Symposium (METRICS), Chicago, Illinois, USA, 2004.
- [11] J. Zhao and B. Xu, "Measuring Aspect Cohesion," presented at International Conference on Fundamental Approaches to Software Engineering (FASE'2004), Barcelona, Spain, 2004.
- [12] M. Ceccato and P. Tonella, "Measuring the Effects of Software Aspectization," presented at First Workshop on Aspect Reverse Engineering (WARE 2004) in conjunction with WCRE 2004, Delft, the Netherlands, 2004.
- [13] J. e. Barone, "Comparing apples and oranges: a randomised prospective study," *British Medical Journal*, pp. 1569-1570, 2000.
- [14] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous & Practical Approach*, 2nd ed. London, UK: International Thomson Computer Press, 1997.
- [15] OMG, "UML 2.0 OCL Final Adopted specification," Object Management Group Inc. ptc/03-10-14, October 2003.
- [16] F. B. Abreu, "Using OCL to formalize object oriented metrics definitions," INESC, Software Engineering Group, Technical Report ES007/2001, May 2001.

- [17] T. A. Halpin, "Information Modeling and Higher-order Types," presented at EMMSAD'04 workshop in connection with the CAiSE'04 conference, Riga, Latvia, 2004.
- [18] OMG, "UML 2.0 Superstructure Final Adopted Specification," Object Management Group Inc. ptc/03-08-02, August 2003.
- [19] OMG, "UML 2.0 Infrastructure Final Adopted Specification," Object Management Group, Inc. ptc/03-09-15, September 2003.
- [20] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*: Manning Publications Co., 2003.
- [21] G. Kiczales and M. Mezini, "Aspect-Oriented Programming and Modular Reasoning," presented at ICSE'05, St. Louis, Missouri, USA, 2005.
- [22] D. Coward, "Logging API Specification," Sun Microsystems, Java Specification Request JSR 47, May 09 2002.
- [23] U. Kulesza, C. Sant'Anna, A. Garcia, R. Coelho, A. v. Staa, and C. Lucena, "Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study," 2006.
- [24] C. Sant'Anna, A. Garcia, C. Chavez, C. Lucena, and A. v. Staa, "On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework," presented at Brazilian Symposium on Software Engineering, 2003.
- [25] J. Benn, C. Constantinides, H. K. Padda, K. H. Pedersen, F. Rioux, and X. Ye, "Reasoning on Software Quality Improvement with Aspect-Oriented Refactoring: A Case Study," presented at Software Engineering and Applications (SEA 2005), Phoenix, AZ, USA, 2005.

**Acknowledgment.** This research work was developed with the partial support of the SOFTAS (POSC/EIA/60189/2004) project, financed by the Portuguese Foundation for Science and Technology.

## Appendix A: Extract of PIMETA instantiation for traditional logging

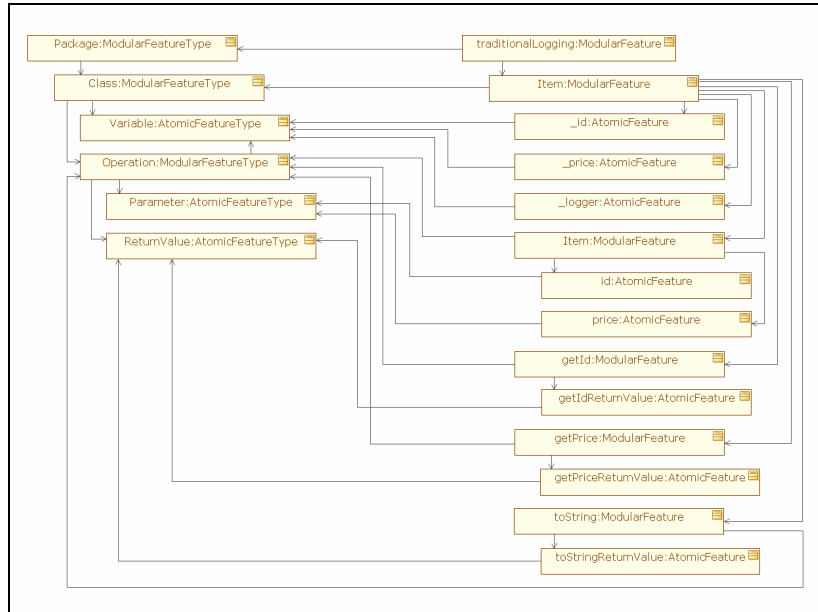


Figure 8: *Item* class structure

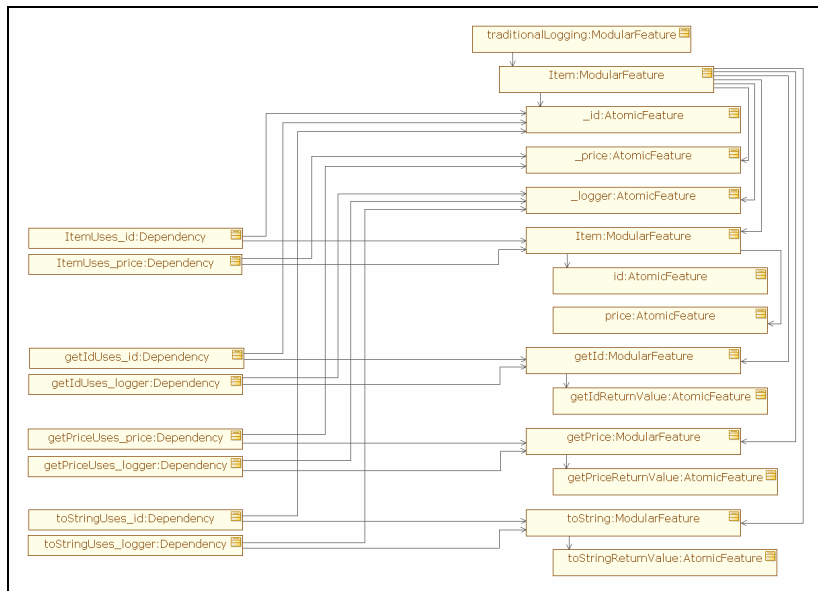


Figure 9: *Item* class internal dependencies (cohesion)

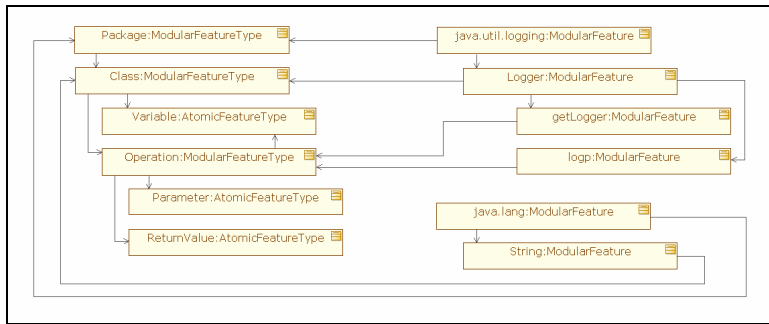


Figure 10: External required modules' structure

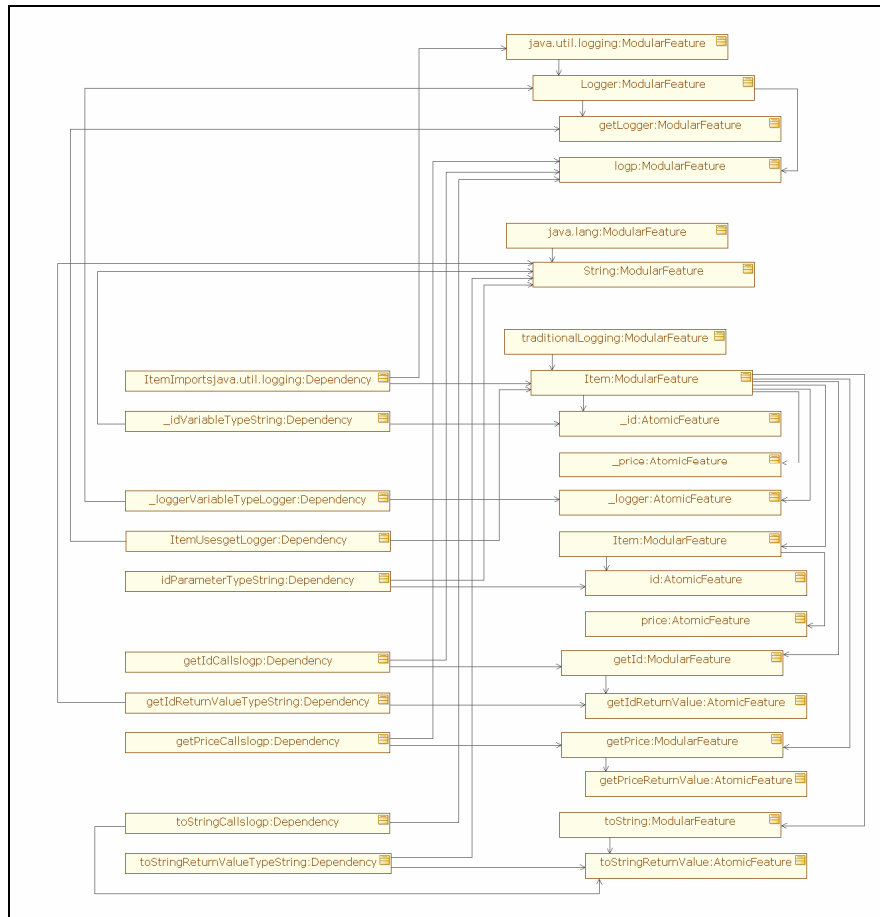


Figure 11: *Item* class external dependencies

## Appendix B: Extract of PIMETA instantiation for aspectized logging

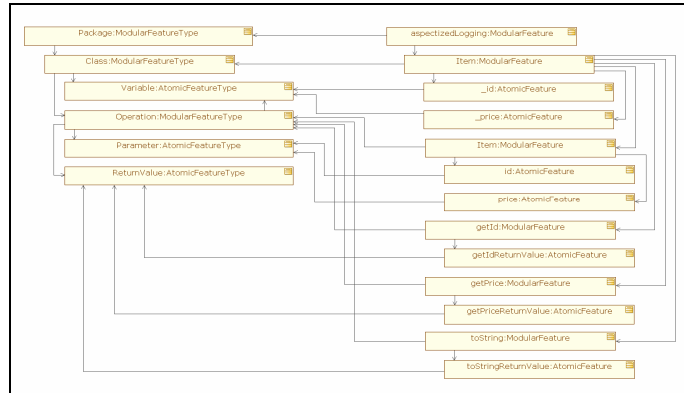


Figure 12: *Item* class structure

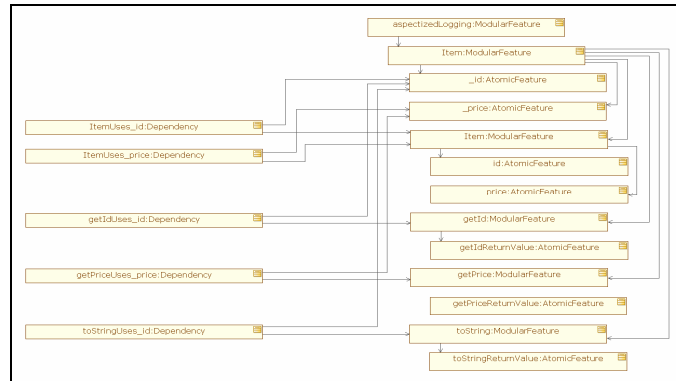


Figure 13: *Item* class internal dependencies (cohesion)

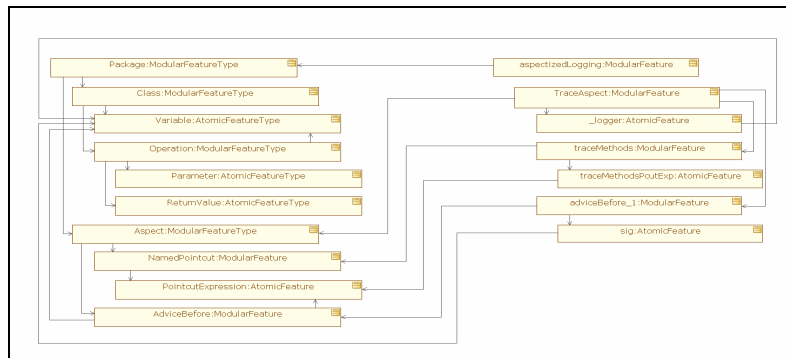


Figure 14: *TraceAspect* aspect structure

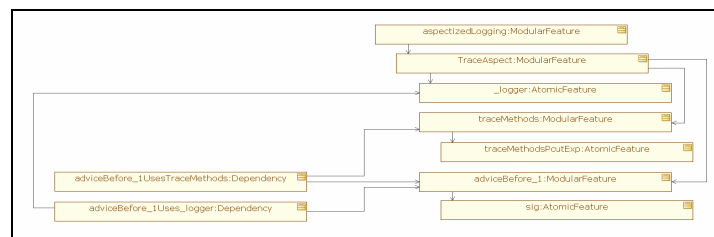


Figure 15: *TraceAspect* internal dependencies (cohesion)

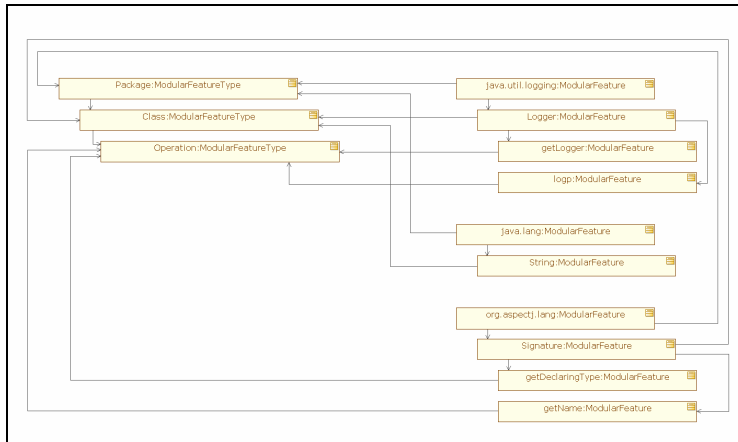


Figure 16: External required modules' structure

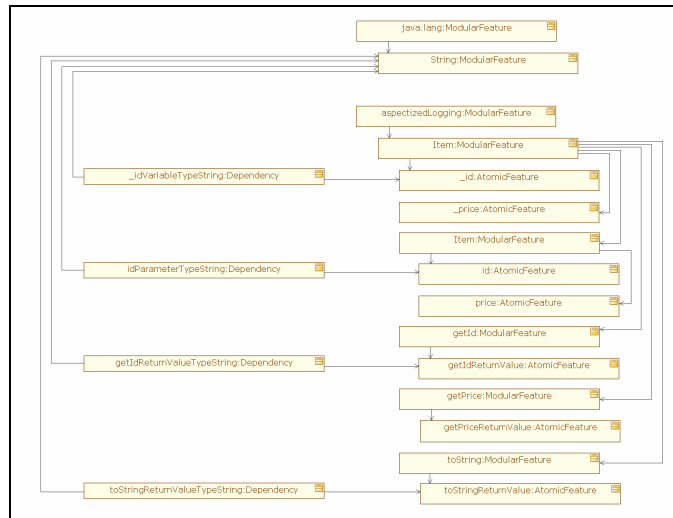


Figure 17: *Item* class external dependencies (coupling)

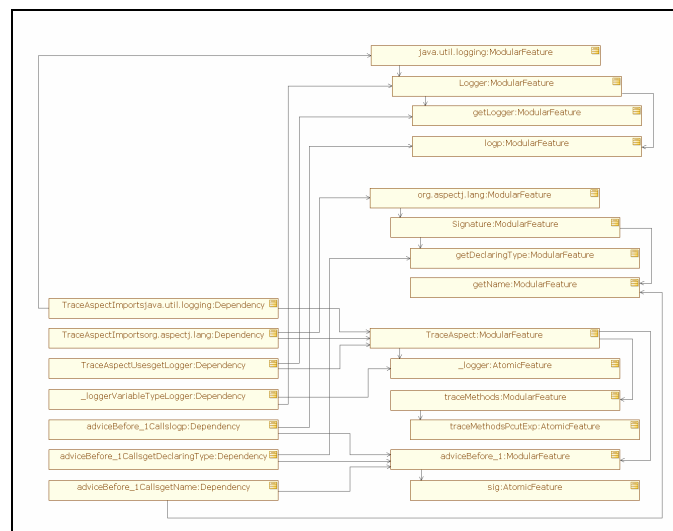


Figure 18: *TraceAspect* aspect external dependencies (coupling)