A UML Profile to Model Classifier Refinements

Natalia Correa, Roxana Giandini and **Claudia Pons**

Universidad Nacional de La Plata, Facultad de Informática
LIFIA - Laboratorio de Investigación y Formación en Informática Avanzada
La Plata, Argentina, 1900
[nataliac, giandini, cpons]@sol.info.unlp.edu.ar
Workshop de Ingeniería de Software y Bases de Datos (WISBD)

Abstract

The refinement technique allows us to capture the relationship between specification and implementation in software developments. The precise documentation of the refinement relationship makes it possible the traceability of the requirements through the refinement steps. Unfortunately, the standard modelling language UML suffers from a lack of notation to specify complex cases of refinement, which hinders the traceability activities. In this article we present a profile of UML to model Classifier refinements as a composition of elementary refinements, allowing for a more precise syntactical specification of the refinement relationship.

Keywords: Software Engineering, Formal Methods.

1 Introduction

Abstraction [4] is the key to mastering complexity. Abstraction facilitates the understanding of complex systems by dealing with the major issues before getting involved in the detail. Apart from enabling for complexity management, the inverse of abstraction, refinement, captures the essential relationship between specification and implementation. Abstraction/Refinement relationship makes possible to understand how each business goal relates to each system requirement and how each requirement relates to each facet of the design and ultimately to each line of the code. Documenting the refinement relationship between these layers allows developers to verify whether the code meets its specification or not, trace the impact of changes in the business goals and execute test assertions written in terms of abstract model's vocabulary by translating them to the concrete model's vocabulary.

Refinement has been studied in many formal notations such as Z [2] and B [11] and in different contexts, but there is still a lack of formal definitions of refinement in semi-formal languages, such as the UML.

The standard modelling language UML [14] provides an artifact named *Abstraction* (a kind of Dependency) to explicitly specify abstraction/refinement relationship between UML model elements. In the UML metamodel an Abstraction is a directed relationship from a *client* (or clients) to a *supplier* (or suppliers) stating that the client (the refinement) is dependent on the supplier (the abstraction). The Abstraction artifact has a meta attribute called *mapping* designated to record the abstraction/implementation mappings, that is an explicit documentation of how the properties of an abstract element are mapped to its refined versions, and on the opposite direction, how concrete elements can be simplified to fit an abstract definition. The more formal the mapping is formulated, the more traceable across refinement steps the requirements are.

Some recently published works analyze the basis of the refinement relationship concepts in UML. In [1], the authors define a formal semantics for the refinement relationship and another subset of

UML elements; in [16] the refinement relationship is deeply analysed and compared with the specialisation relationship; Hnatkowska et al. in [7] and Liu et al. in [12] present two different approaches for the definition and use of the refinement relationship, whereas in [9] some formal methods for the automatic exploration of the Dependency concept in the context of UML specification, are described.

Although the Abstraction artifact allows for the explicit documentation of the Abstraction/Refinement relationship in UML models, an important amount of variations of this relationship remains underspecified by different causes; for example:

- refinement is hidden under other notations
- some refinements are composed by others more elementary ones that cannot be represented in UML, since this language lacks notation for it.

The former problem was analysed in [19], where the authors study some UML artifacts such as generalization, composite association and use case inclusion; which implicitly define Abstraction/Refinement relationship.

In this article, we restrict our attention to study the second problem in a syntactic way: UML models representing refinements that are composed by more elementary ones lack precision due to the fact that the refinement mapping cannot capture the elementary refinement documentation. Compound refinements appear even in the simplest case of refinement, such as the one of classes; composition takes place in class refinement due to the fact that a class refinement generally consists of attribute and operation refinements.

The organization of the article is the following one: in section 2 we discuss refinement relations between Classifiers such as Use Cases and Classes and their variants, whereas in section 3 we express these variants as composition of more elementary refinements. In section 4 we introduce a profile of UML based on the definition of new stereotypes, formalizing the proposal presented in section 3. Finally we present conclusions and lines of future work.

2 Classifier Refinements

In [3], authors mention that refinement between Classifiers (or Types) can be realized in two different ways: Attribute refinements (or model refinements) and operation refinements. These forms of refinements specify that new attributes/operations can be added to a Classifier or that existing attributes/ operations can be replaced for new ones, in order to obtain a refinement from an abstract Classifier. In this work, we consider these basic refinements and propose other possible variants, as follows:

- **Attribute refinement:** a Classifier can be obtained by replacing one of its attributes by its refinement (one o more attributes). For example, figure 1 a) shows that currentPosition attribute from Player class is refined by the attributes called previousPosition and stepsToMove through the mapping {currentPosition = previousPosition + stepsToMove}.

- **Operation refinement:** we consider two variants of this refinement:
  - Atomic operation refinement: it express that an operation refines its postcondition.
  The operation signature does not change; only its postcondition changes. The refined postcondition must imply the original one. For example, figure 1 b) shows that the move operation in the Player class is refined in an atomic way, modifying its postcondition. We can observe that when the mapping does not appear in the diagram, the operation maps to itself (i.e. mapping contains the identity function).
  - Non-Atomic operation refinement: A Classifier can be obtained by replacing one of its operations by its refinement (one o more operations). For example, figure 1 c) shows that

the move operation in the Player class is refined in a non-atomic way by the moveForwards and moveBackwards operations through the mapping <move=moveForwards; moveBackwards >.

- **Invariant refinement**: a Classifier can be obtained by replacing its invariant by its refinement (An invariant is an assertion that must stay true for all class instances). The refined class has a new invariant that implies the original one. For example, figure 1 d) shows the refinement of the invariant in the Player class.

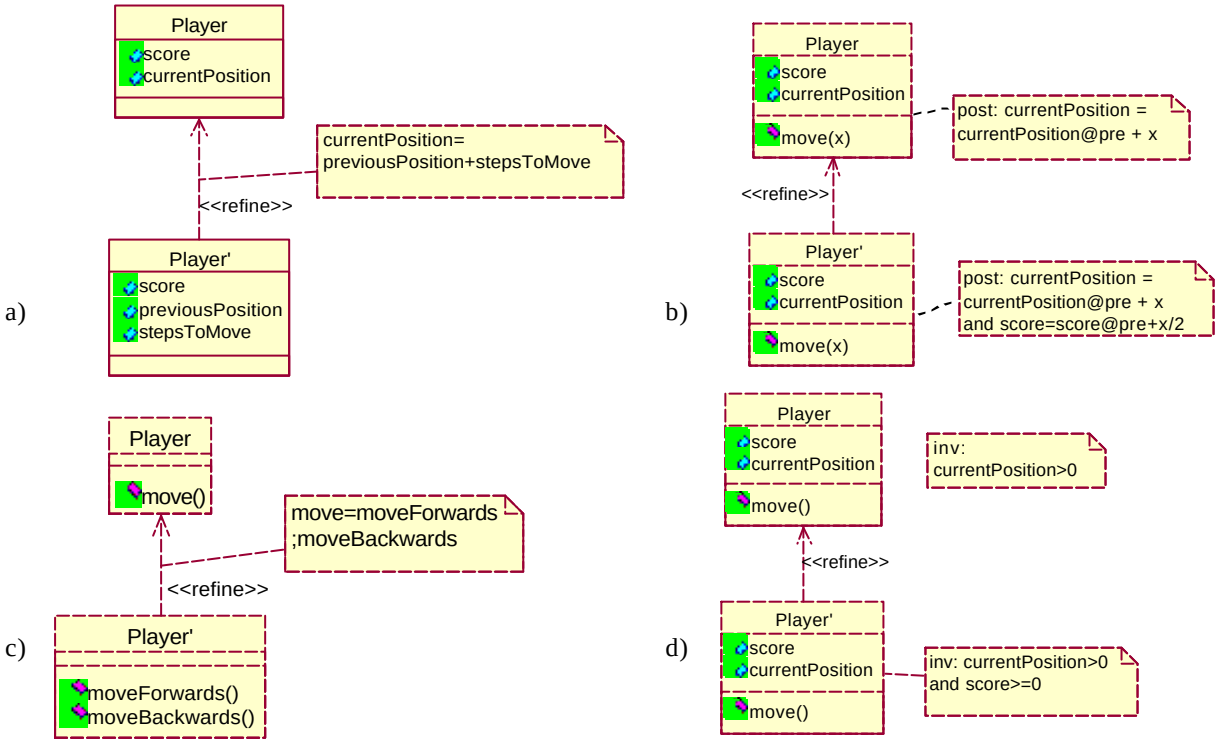

Figure 1: Cases of class refinement: a) attribute refinement, b) atomic operation refinement, c) non-atomic operation refinement, d) invariant refinement.

3 Classifier refinements expressed as composition of elementary refinements

In the UML, the specification of the Abstraction Dependency is vague. Even Abstraction stereotyped with <<refine>>, for example in the context of class refinement, cannot express which elements (attributes, operations, invariant, or a composition of them) are being refined. Figure 2 shows a UML metamodel instance which corresponds to the diagram in figure 1 c), where the refinement is specified between classes (although the actual refinement takes place on the move operation). The meta-attribute *mapping* attached to the relationship contains an expression specifying the connection between the abstract and the concrete operations; however it is not clear enough that it is an operation which is being refined. In this sense, UML metamodel lacks precision, hindering the traceability process.

Mappings vary according to which element is being refined (attributes, operations or invariants), consequently suppliers and clients of the Dependency relationship should vary accordingly (they are attributes, operations or invariants respectively). In order to specify these variations we need a more expressive notation to represent Classifier refinement. This notation would allow us to look at a

Classifier refinement no just as an atomic piece of documentation, but as a composition of more elementary pieces. That is to say, we would be able to express that a Classifier refinement is composed by zero or more attribute refinements, zero or more operation refinements and zero or more invariant refinements. In the following subsections we will analyze some cases of elementary refinements which can take place within a compound Classifier refinement.

In the first place, we need to specify that a Refinement is composed by other Refinements, that is to say we need to declare an Association between Refinements (notice that the composition is a particular case of Association). This is not possible in UML 1.x neither in UML 2.0 because Associations can be only established between Classifiers (notice that Dependencies are not Classifiers). A solution for this drawback is, on one hand, to define a new stereotype <<refinementComposition>> based on the DirectedRelationship metaclass allowing us to express composition between Refinements. On the other hand, we define a hierarchy of new stereotypes based on the Abstraction metaclass, representing the different kinds of Refinements (i.e. compound refinements and elementary refinements) which will be connected by the stereotyped relationship.

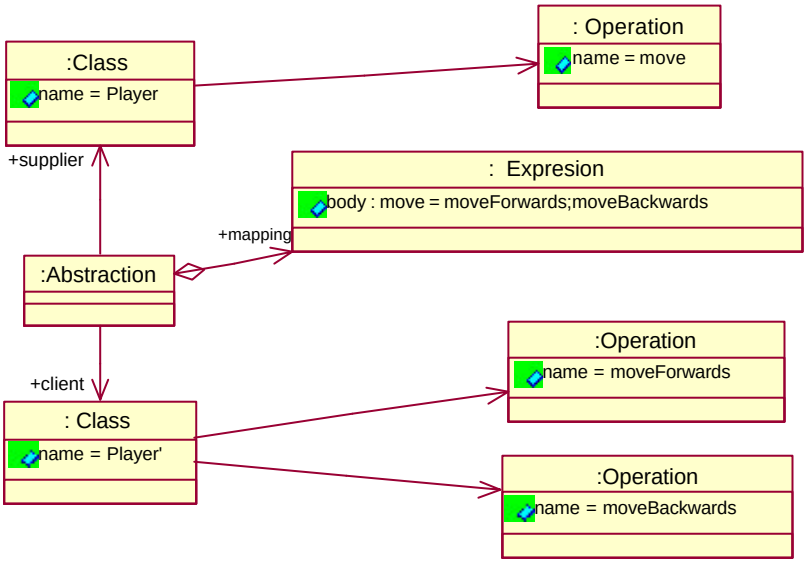

Figure 2: metamodel instance corresponding to diagram of figure 1 c)

3.1 Compound Refinement in Class Diagrams

Figure 3 shows the class refinement corresponding to figure 1c). It explicitly shows that the refinement is composed by an operation refinement. Then, Figure 4 shows the diagram in Figure 3 expressed as an instance of the proposed metamodel. The composition relationship is an instance of DirectedRelationship metaclass. Its *source* is the class refinement and its *target* is the non-atomic operation refinement. *Clients* of the non-atomic operation refinement form a set specifying the refining operations while *supplier* specifies the refined one. In [6] we have analyzed in detail other elementary Classifier refinements (attribute and invariant).

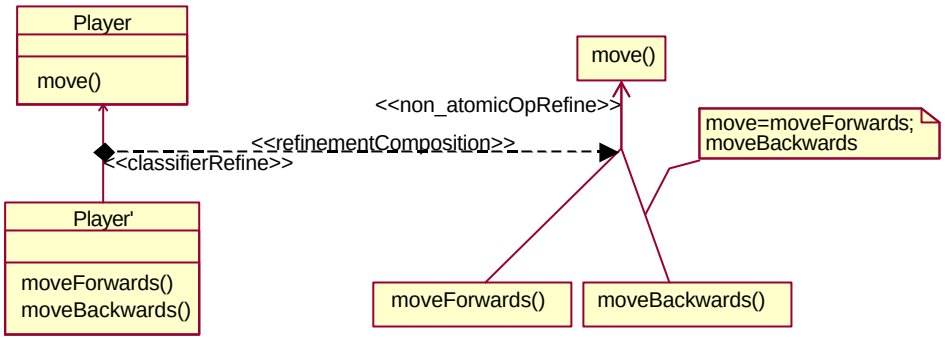

Figure 3: Class refinement composed by non-atomic operation refinement.

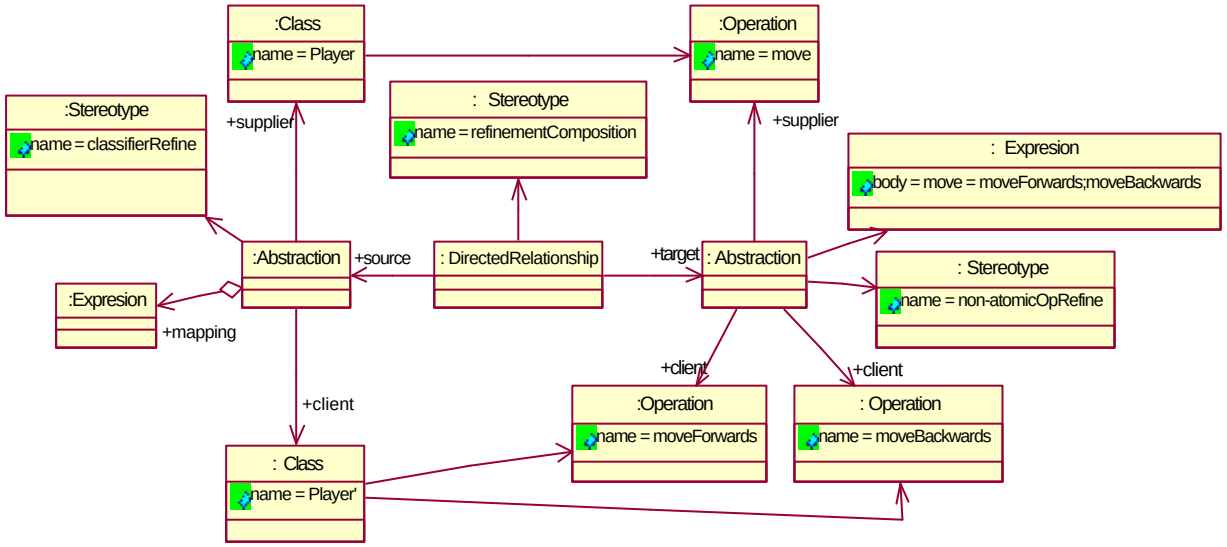

Figure 4: instance of the proposed metamodel corresponding to diagram in Figure 3.

3.2 Compound Refinement in Use Case Diagrams

Frequently, we specify refinements between use cases in UML in an implicit way. Those refinements remain hidden in the model. When modelling one or more use cases with an <<include>> Dependency between them, actually we are defining a use case operation refinement. The set of included use cases form the refinement itself. An example is shown in figure 5. Generalization between Use Cases is another way to model hidden refinements.

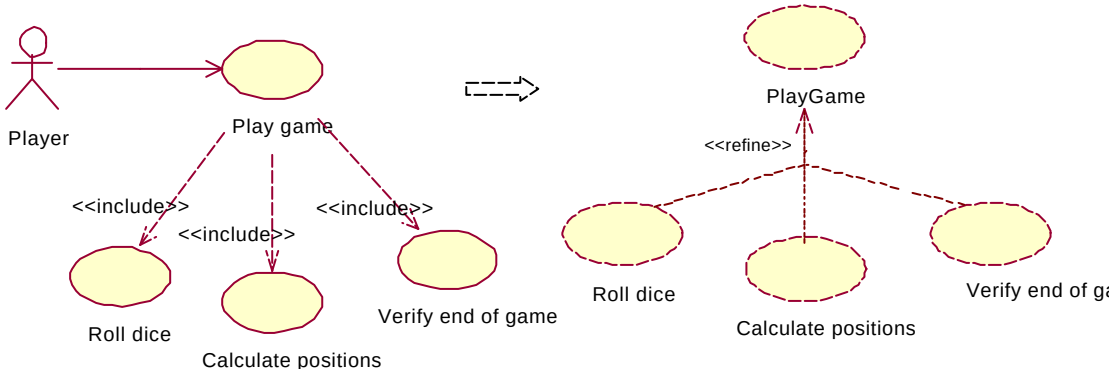

Figure 5: Implication between <<include>> Dependency and Classifier Refinement between use cases.

Since use cases are Classifiers, use case refinement can be expressed as Classifier refinement composed by a non-atomic operation refinement. Figure 6 shows an example, while Figure 7 shows its instantiation on the proposed metamodel. In this instantiation, we can see explicitly the operation refinement relationship between use cases.

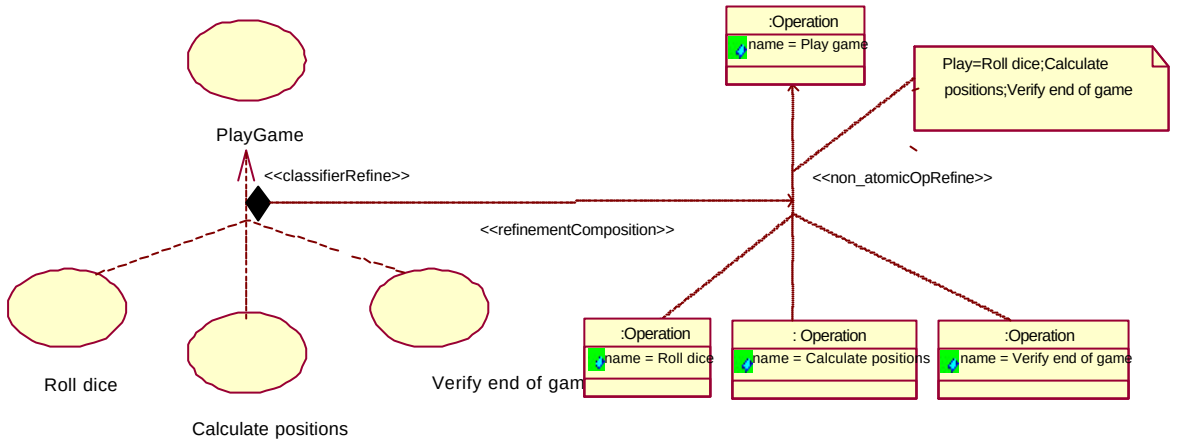

Figure 6: Use Case refinement composed by non-atomic operation refinement.

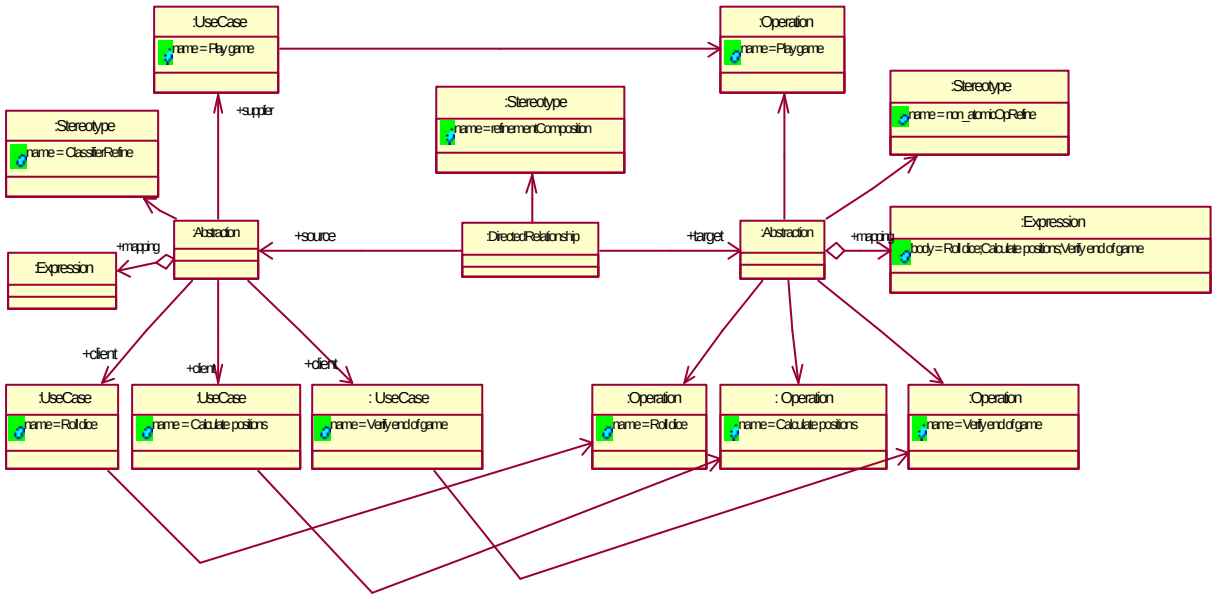

Figure 7: Instantiation of proposed metamodel for diagram in figure 6

4 A UML profile to express Classifier refinements

As we discussed in previous section, in order to be able to model elementary class refinements in UML, it is necessary to extend UML. On one hand, we introduce the extension to model refinement composition. This specific composition is expressed defining a stereotype extending the metaclass DirectedRelationship (Figure 8).

Figure 8: The stereotype << refinementComposition>> with base in DirectedRelationship.

On the other hand, we propose a stereotype hierarchy, depicted in figure 9. The hierarchy root is the stereotype <<refine>> which already exists in UML with base in Abstraction metaclass. The stereotype <<ClassifierRefine>> models compound refinements between Classifiers while <<elementaryRefine>> models elementary refinements; in particular, we consider elementary refinements of attribute, operation and invariant but the hierarchy is open to future extensions.

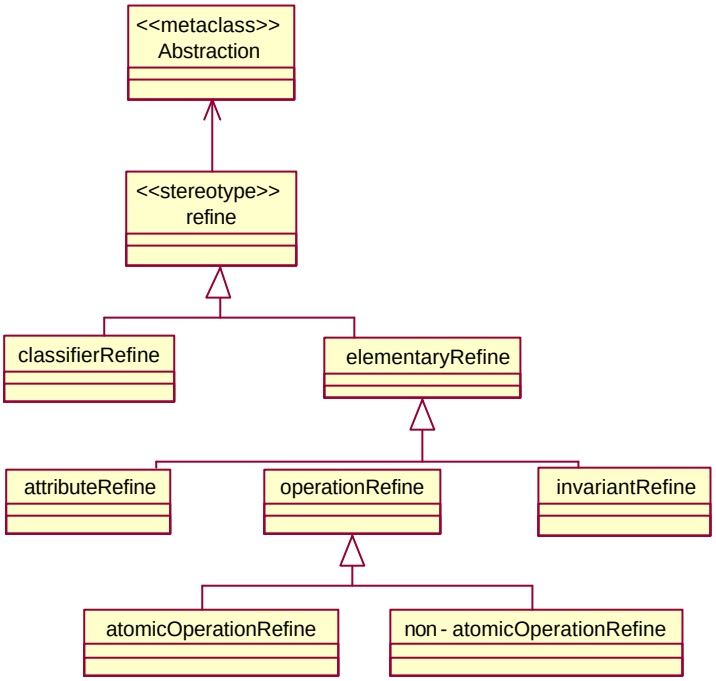

Figure 9: Stereotypes proposed as extension of UML metamodel.

We include well formedness rules (constraints) to express that a refinement with stereotype <<ClassifierRefine>> is composed by elementary refinements (i.e. refinements with <<elementaryRefine>> substereotypes, such as <<attributeRefine>>, <<operationRefine>> or <<invariantRefine>>).
As well, the stereotype <<operationRefine>> is sub classified with more specific stereotypes: <<atomicOperationRefine>> and <<non_atomicOperationRefine>>, allowing for the modeling of the different kinds of operation refinements that we introduced in section 2.

The formal definition of the new stereotypes, using OCL [13] constraints to express their well formedness rules are as follows:

- STEREOTYPE <<**refinementComposition**>>

Base: DirectedRelationship

Constraints:

1. A << refinementComposition>> relationship must be established between refinements (the ends must be stereotyped with stereotypes of the *refine* hierarchy)

```
self.source.oclIsKindOf(Abstraction)  and  self.target.oclIsKindOf(Abstraction)
and
self.source.stereotype.allParents -> includes (<<refine>>) and
self.target.stereotype.allParents -> includes (<<refine>>)
```

2. The relationship source must be a compound refinement (that is to say it cannot be an elementary refinement)

```
self.source.stereotype.allParents -> (excludes (<<elementaryRefine>>))
```

3. The target of the relationship must belong to the elementary refinements hierarchy and it must not belong to another composition between refinements (strong composition).

```
self.target.stereotype.allParents  ->  includes  (<<elementaryRefine>>)  and
(self.owningPackage.ownedMember  ->  select  (r:  DirectedRelationship  |
r.stereotype= << refinementComposition >> and r <> self ))-> forAll (r |
r.target <> self.target)
```

- STEREOTYPE **<<ClassifierRefine>>**

Base: Abstraction
Parent:: <<*refine*>> stereotype

*Constraint*s:
1. A <<*ClassifierRefine*>> Abstraction must be established between Classifiers.

```
self.supplier.oclIsKindOf(Classifier) and self.client.oclIsKindOf(Classifier)
```

2. The mapping between Classifiers is a correspondence relation between the features of the supplier Classifiers and the features of the client Classifiers

```
self.mapping.body = [σ(<p1, p2, ..,pn>) = < f(p1), f(p2), .. f(pn)>]
```

The Classifier specification can be defined by its feature list (its attributes, operations, etc.) that describes the Classifier objects [19]. In this case the feature list <p1, p2,..,pn> specifies the supplier Classifier (self.supplier); s is the function that applied to the supplier Classifier (its specification), returns its refinement (self.client); and f is the function describing the mapping for each property.

s : Classifier-> Classifier
f : Feature -> Feature

Note: We have used an abstract representation to express the function associated to the mapping, due to the fact that the mappings between features (for example operation composition) cannot be expressed in OCL.

- STEREOTYPE **<<elementaryRefine>>**.

Base: Abstraction
Parent:: <<*refine*>> stereotype. It is an abstract stereotype.

*Constraint*s:
1. An <<*elementaryRefine*>> Abstraction cannot exist by itself, it must be "part of" only one compound refinement (of Classifier):

```
self.composites -> size() = 1 and self.composite.stereotype = <<ClassifierRefine
>>
```

where the operation *composites* returns the set of compound refinements containing self, while the operation *composite* returns any element belonging to that set:

composites: Abstraction -> Set(Abstraction)

```
composites = (self.owningPackage.ownedMember -> select (r: DirectedRelationship
| r.stereotype= << refinementComposition >> and
r.target = self ))  -> collect (r: DirectedRelationship | r.source )
```

- STEREOTYPE <<**attributeRefine**>>

Base: Abstraction
Parent:: << *elementaryRefine* >> stereotype. It is an abstract stereotype

Constraints:
1. An <<*attributeRefine*>> Abstraction must be established between attributes.

```
self.supplier.oclIsKindOf(Attribute)              and              self.client->
forAll(a|a.oclIsKindOf(Attribute))
```

2. The attributes that form the refinement must be defined in the corresponding Classifiers, in particular:

2.1 The client attributes must belong to the client Classifier of the compound Abstraction which contains the elementary Abstraction:

```
self.client-> forAll(a |self.composite.client.ownedAttribute -> includes(a))
```

2.2 The supplier attribute must belong to the supplier Classifier of the compound Abstraction which contains the elementary Abstraction:

```
self.composite.supplier.ownedAttribute -> includes (self.supplier)
```

- STEREOTYPE <<**operationRefine**>>

Base: Abstraction
Parent:: << *elementaryRefine* >> stereotype. It is an abstract stereotype

Constraints:
1. An <<*operationRefine*>> Abstraction must be established between operations.

```
self.supplier.oclIsKindOf(Operation) and self.client->
forAll(o|o.oclIsKindOf(Operation))
```

2. The operations that form the refinement must be defined in the corresponding classes, in particular:

2.1 the client operations must belong to the client Classifier of the compound Abstraction which contains the elementary Abstraction:

```
self.client-> forAll(o |self.composite.client.ownedOperation -> includes(o))
```

2.2 the supplier operation must belong to the supplier Classifier of the compound Abstraction which contains the elementary Abstraction:

```
self.composite.supplier.ownedOperation -> includes (self.supplier)
```

- STEREOTYPE <<**atomicOperationRefine**>>

Base: Abstraction
Parent:: << *operationRefine* >> stereotype.

Constraints:
1. In each refinement relationship, an operation is refined with only one operation.

```
self.client -> size() = 1
```

2. The client operation postcondition implies the supplier operation postcondition.

```
(self.client.ownedRule->any (r| r.stereotype=<<postCondition>>)).specification
implies (self.supplier.ownedRule->any (r|
r.stereotype=<<postCondition>>)).specification
```

- STEREOTYPE <<**non-atomicOperationRefine**>>

Base: Abstraction
Parent:: << *operationRefine* >> stereotype.

Constraints:
1. An operation is refined with more than one operation.

```
self.client -> size() > 1
```

- STEREOTYPE << **invariantRefine**>>

Base: Abstraction
Parent:: << *elementaryRefine* >> stereotype.

Constraints:
1. It is established between Classifier invariants, that is to say client and supplier are constraints with stereotype <<invariant>>

```
self.supplier.oclIsKindOf(Constraint) and self.client.oclIsKindOf(Constraint)
and self.supplier.stereotype=<<invariant>> and self.client.stereotype =
<<invariant>>
```

2. The invariants that form the refinement must belong to the corresponding Classifiers, in particular:

2.1 The client invariant must be an invariant of the client Classifier of the compound Abstraction which contains the elementary Abstraction:

```
self.composite.client.ownedRule -> includes (self.client)
```

2.2 The supplier invariant must be an invariant of the supplier Classifier of the compound Abstraction which contains the elementary Abstraction:

```
self.composite.supplier.ownedRule -> includes (self.supplier)
```

3. In each refinement relationship , an invariant is refined with only one invariant.

```
self.client -> size() = 1
```

4. The client invariant must imply the supplier invariant.

```
self.client.specification implies self.supplier.specification
```

5 Conclusions and future work

Abstraction/Refinement relationship makes it possible to understand how each business goal relates to each system requirement and how each requirement relates to each facet of the design and ultimately to each line of the code. The more in detail the documentation of refinements (mapping) in the software development is formulated, the more traceable across refinement steps the requirements are.

The problem of lack of formality in the specification of refinements on UML models has been analysed by our research team; to experiment, a tool called ePLATERO [17], integrated in the Eclipse environment [8] and based on the formal definition of refinement was created. The tool supports the documentation of explicit refinements (i.e. Abstractions artifacts with their corresponding mapping expressions) and the semi-automatic discovering and documentation of hidden refinements. In addition, it was published a work [18] with the last advances of ePLATERO project.

As a continuation of this previous work, in the present article, we focussed our attention to the study of composition of refinements. The standard modelling language UML suffers from a lack of notation to specify complex cases of refinement, due to the fact that the UML refinement mapping cannot capture the elementary refinements which make up a compound refinement. Consequently, compound refinements remain under specified in UML models, which obstruct the traceability process.

As a solution for this weakness we developed a profile for UML to model refinements, based in the definition of new stereotypes.

The refinement relation can be established between elements of the same type (for example between classes) or elements of different types (for example a use cases model and a collaboration model realising it) [15] [5]. In this work we have studied one form of refinement between elements of the same type (the refinement of Classifiers). As a line of future work, we will include the treatment of other refinements between elements of different type. The idea is to discriminate the hierarchy of the stereotype <<refine>> of Abstraction. One branch will be under the discriminant sameTypeElement and the other branch under the discriminant differentTypeElement. In the later case, the Client and the Supplier of the refinement relationship have different types.

This set of stereotypes will add new notations and restrictions which will contribute to the improvement of the UML language towards the precise specification of refinements, with the final purpose of increasing the accuracy of the traceability process.

References

[1] Davies, J., Crichton, C. Concurrency and refinement in the UML. Electronic Notes in Theoretical Computer Science, Volume 70, July 2002

[2] Derrick, J. and Boiten,E. Refinement in Z and Object-Z. Foundation and Advanced Applications. FACIT, Springer, 2001
[3] D´ Souza, Desmond and Wills, Alan. Objects, Components and Frameworks with UML.Addison-Wesley. 1998.
[4] Dijkstra, E.W., A Discipline of Programming. Prentice-Hall, 1976.
[5] Giandini, R., Pons, C., Pérez,G. Use Case Refinements in the Object Oriented Software Development Process. Proceedings of CLEI 2002, ISBN 9974-7704-1-6, Uruguay. 2002.
[6] Giandini, R., Pons, C. Una extensión de UML para modelar refinamientos. Proceedings of Workshop IDEAS´05, Editor: Hernán Astudillo. Valparaíso, Chile. May 2005.
[7] Hnatkowska B., Huzar Z., Tuzinkiewicz L On Understanding of Refinement Relationship. Workshop in Consistency Problems in UML-based Software Development III – Understanding and Usage of Dependency Relationships at the 7th International Conference on the UML. Lisbon, Portugal, October 11, 2004.
[8] IBM, The Eclipse Project. Home Page. Copyright IBM Corp. and others, 2000-2003. http://www.eclipse.org/.
[9] Kostrzewa M. Exploration and Refinement of the UML Dependency Concepts. Workshop in Consistency
[10] Problems in UML-based Software Development III at the 7th International Conference on the UML. Lisbon, Portugal, Oct 11, 2004
[11] Lano,K. The B Language and Method. FACIT. Springer, 1996.
[12] Liu Z., Jifeng H., Li X., Chen Y. Consistency and Refinement of UML Models. Workshop in Consistency Problems in UML-based Software Development III at the 7th International Conference on the UML. Lisbon, Portugal, Oct 11, 2004
[13] OMG (a). The Object Constrain Language Specification – Version 2.0, for UML 2.0, revised by the OMG, http://www.omg.org, April 2004.
[14] OMG (b). The Unified Modeling Language Specification – Version 2.0, UML Specification, revised by the OMG, http://www.omg.org, April 2004.
[15] Pons, C., Giandini R. and Baum G. Specifying Relationships between models though the software development process, Tenth International Workshop on Software specification and Design, San Diego, IEEE Computer Society Press. November 2000.
[16] Pons, C., Pérez,G., Giandini, R., Kutsche, R. Understanding Refinement and Specialization in the UML. and 2nd International Workshop on MAnaging SPEcialization/Generalization Hierarchies (MASPEGHI). In IEEE ASE 2003, Canada.
[17] Pons, C., Giandini, R, Pérez., G., Pesce, P., Becker, V., Longinotti,J., Cengia,J., Kutsche, R-D., The ePLATERO Project: “Formal Tool for the Evolutionary Software Development Process”. Home page: http://sol.info.unlp.edu.ar/eclipse., 2003-4
[18] Pons C., Giandini R., Pérez G., Pesce P., Becker V., Longinotti J., Cengia J., Correa N. and Labaronnie P. Pampero: Precise Assistant for the Modeling Process in an Environment with Refinement Orientation In "UML Modeling Languages and Applications: UML 2004 Satellite Activities, Revised Selected Papers". Lecture Notes in Computer Science number 3297. Springer, Oct., 2004
[19] Pons, C., Kutsche, R. Traceability Across Refinement Steps in UML Modeling. 3rd Workshop in Software Model Engineering WiSME at the 7th International Conference on the UML. October 11th 2004. Lisbon, Portugal .