Traceability Across Refinement Steps in UML Modeling

C. Pons[1 ,2] and R-D Kutsche [3]

[1] LIFIA – Laboratorio de Investigación y Formación en Informática Avanzada, Universidad Nacional de La Plata, calle 50 y 115 - 1900 Buenos Aires, Argentina

[2] UAI – Universidad Abierta Interamericana, Facultad de Tecnología Informática Chacabuco 90 - 1° Piso, Ciudad de Buenos Aires, Argentina
cpons@info.unlp.edu.ar

[3]*CIS Computation and Information Structures CIS,* Technical University of Berlin, Faculty IV, Berlin, Germany
rkutsche@cs.tu-berlin.de

ABSTRACT. Documenting the refinement relationship between 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 and B and in different contexts, but there is still a lack of formal definitions of refinement in semi-formal languages, such as the UML. The contribution of this article is to clarify the abstraction/refinement relationship between UML models, providing basis for tools supporting the refinement driven modeling process. We formally describe a number of refinement patterns and present PAMPERO, a tool integrated in the Eclipse environment, based on the formal definition of model refinement.

1. Introduction

The refinement relationship [Dijkstra, , 1976] 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.

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 [Derrick and Boiten, 2001] and B[Lano, 1996] 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 modeling language UML [OMG, 2001] 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.

Although the Abstraction artifact allows for the explicit documentation of the abstraction/refinement relationship in UML models, an important amount of variations of abstraction/refinement remains unspecified, in general hidden under other notations. For example UML artifacts such as generalization, composite association, use case inclusion, among others, implicitly define abstraction/refinement relationship. The starting point to enable traceability of requirements across refinement steps is to discover and precisely capture the various forms of the abstraction/refinement relationship, in particular those forms which are hidden in the model.

To experiment, we created a tool integrated in the Eclipse environment [IBM, 2003], called PAMPERO (**P**recise **A**ssistant for the **M**odeling **P**rocess in an **E**nvironment with **R**efinement **O**rientation), based on the formal definition of refinement. 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 the remainder of this article we will describe a number of undercover refinements in UML modeling. Refinement can be established between model elements of either the same kind (e.g. between two classes) or different kind (e.g. between a use case model and a collaboration model) [Pons et al. 2000; Giandini and Pons, 2002; Pons et al.,2003]. In this article we restrict our attention to relationships between model elements of the same kind, in particular we focus on two UML artifacts: Classes, which are described in section 2 and Use Cases which are analyzed in section 3. For each one of these artifacts the discussion comprises two dimensions: intension and extension, where the intension of a modeling artifact is equated with its definition or specification, while its extension refers to the set of elements that fall under that definition.

2. Class Refinement

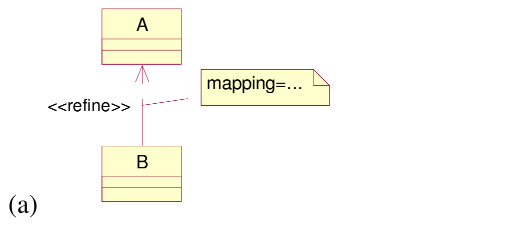
Classes serve as specifications for the properties of sets of objects that can be treated alike. The intension of a Class is defined as a pair (Attr, Ops) where Attr is a mapping from Attribute's name to Attribute's description and Ops is a mapping from Operation's name to Operation's description.

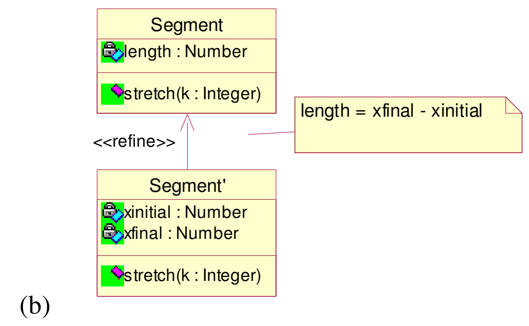
On the semantics side, in the view of Objects as labeled records [Abadi and Cardelli, 1996], the extension of a Class is the set of records meeting its definition: *extension*: Class $\rightarrow$ Set (Object) .

Figure 1a shows the UML artifact specifying abstraction/refinement relationship between Classes. The Catalysis methodology (d'Souza and Wills, 1998) mentions that refinement between Classes (or Types) can be realized in two different ways:

a) *Attribute (or model) Refinement*: it can be obtained in two ways, on one hand the refined Class, B, is obtained by adding a new attribute, $attr_k$, to the abstract Class A. Other case takes place when the Class B is obtained from Class A by replacing an attribute a_k by its refinement, that can be one or more new attributes, a_{k1}, ..., a_{kl}.. For example, figure 1b shows that the attribute *length* in Class *Segment* is refined by the attributes *xinitial* and *xfinal*.

b) *Operation Refinement*: it can be obtained in two ways, on one hand the refined Class, B, is obtained by adding a new operation, op_k, to the abstract Class A. On the other hand the Class B can be obtained from Class A by replacing an operation op_k by its refinement, that can be one or more new operations, op_{k1}, ..., op_{kl}. For example, figure 1c shows that the operation *stretch* in Class *Segment* is refined by the operations *moveXini* and *moveXfin*.





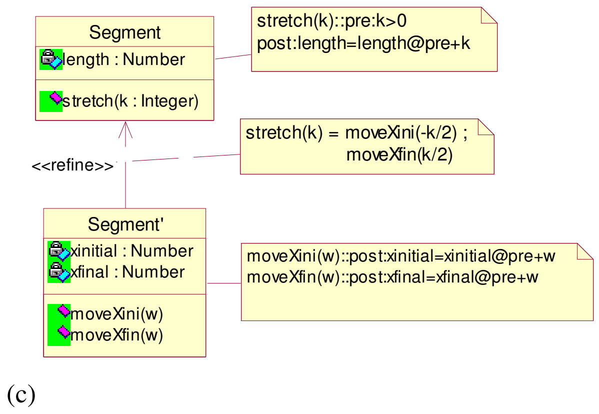


Figure 1: Abstraction/Refinement relationship between Classes. (a) UML notation. (b) Attribute refinement. (c) Operation refinement.

A refined Class is specified in a language that is richer than the language of its abstraction. The relation between the abstract and the refined specifications is usually called "implementation mapping" [Cardelli and Wegner, 1985] or "retrieve relation" [Derrick and Boiten, 2001]. This mapping makes it possible to translate OCL expressions written in the abstract language to the refined language, for example, the OCL expression (**Context** Segment **inv** self.length > 0) can be translated to the expression (**Context** Segment' **inv** |self.xfinal – self.xinitial| > 0).

The counterpart of the "implementation mapping" is the "abstraction mapping" that transforms each refined object to its abstract representation.

The retrieve relation R between Segment and Segment' is given by

∀s,s'•(s,s')∈R ↔ s.length=|xfinal-xinitial|

For example, the abstract object given by ⟨length==5⟩ is related to the concrete object ⟨xinitial==2, xfinal==7⟩ among others.

In all cases, the refinement is a more constrained specification, meaning that all properties specified for an abstraction when translated to its refined language also hold for its refinements, while more properties may hold for the refinement. Therefore the refinement is satisfied by a reduced number of objects. The meaning of Class refinement is that the extension of an abstraction (the supplier) includes the abstract representation of the extensions of all its refinements (the clients):

Context d:Abstraction **inv**:
d.supplier.extension –> *includesAll* (d.client.extension
–>collect(e| d.mapping(e)))

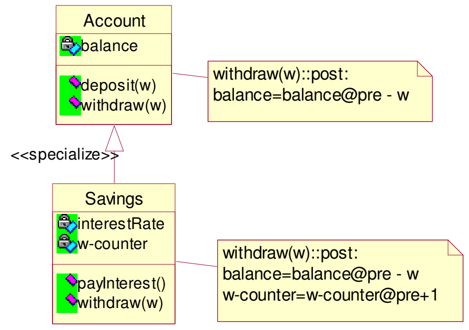
So far we have described the ways to explicitly define Abstraction/Refinement relationship between Classes, however there are other cases that remains hidden and should be discovered in order to allow us to formally check the refinement relationship and to trace properties from the abstract to the concrete models and backwards. In the remainder of this section we describe a number of forms of undercover refinement.

2.1 Refining by Specialization

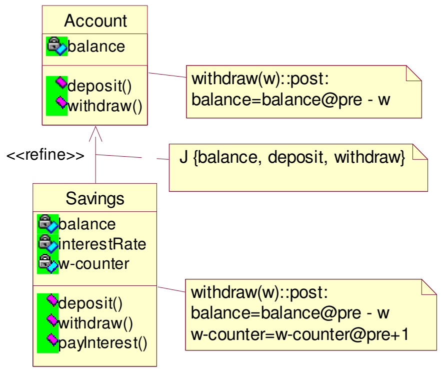
The technique of generalization/specialization which goes hand in hand with Inheritance [Booch, 91] [Wegner and Zdonik, 88], is a central issue in the object oriented paradigm. It is applied to enable reuse, so that less effort is spent when we re-specify things that have already been specified in a more abstract or more general way. In the object oriented paradigm a Class describes the structure and behavior of a set of objects, however it does so incrementally by describing extensions (increments) to previously defined classes (its parents or superclasses).

Figure 2 shows the syntactical connection between Generalization and Abstraction. The UML Generalization artifact (Figure 2a) relates two classes: the parent and the child. The child is not a self contained model, it is just an increment. While, on the other hand the Abstraction relationship (Figure 2b) relates self contained models that are obtained by combining the superclass with the increment.

Two cases of specialization can be distinguished: *Specialization without overriding* (the subclass adds new features without intersection with features in the superclass) and *Specialization with method overriding* (the subclass refines a method of the superclass). We do not consider the case of arbitrary method redefinition, only method refinement where the abstract version is replaced by the refined version of the method.



(a)



(b)

Figure 2: Refinement hidden under Specialization: (a) Generalization/Specialization relationship. (b) Abstraction/Refinement relationship derived from the Generalization.

We define the mapping, reveal: Generalization → Abstraction, with the goal of making up the Abstraction artifact which is hidden under each Generalization artifact. In both cases the refinement is obtained by combining the parent and the child intension (considering method overriding if it is present).

Context Generalization **def**: reveal(): Abstraction
pre: self.parent.oclIsKindOf(Class) and self.child.oclIsKindOf(Class)
post: result.stereotype=<<refine>> and
result.supplier = self.parent and

result.client = (self.parent ⊕ self.child)[1] and
result.mapping=∀o,o'•(o,o')∈R ↔
∀f∈Sig• o.f=o'.f, where Sig=self.parent.allAttributes

The implementation mapping is the identity function (even in the presence of overriding, abstract expressions are mapped to themselves because overriding preserves signatures). The abstraction mapping returns an abstract version of the refined object by keeping only the features defined in the parent Class while ignoring the rest. For example, the abstract object given by ⟨balance==100⟩ is related to the concrete object ⟨balance==100, interestRate==0.5,w-counter==3⟩among others.

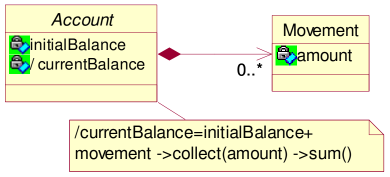
On the semantics side, the subclass introduces a differentiation between the objects specified by the superclass. That is to say, as a consequence of adding more and more detail to the description of a class, a new characteristic that it is not present in all the individuals described by the class, is revealed. Different subsets have different characteristics. Consequently, the specialization technique allows modelers to implicitly specify refinement relationship that generates a partition of the class's extension into two or more subsets. For example, let d be the Abstraction artifact derived from the specialization of Class Account into two subclasses, Savings and Checking:

Account.extension = Savings.extension –> collect (s| d.mapping(s)) –>
union (Checking.extension –> collect (c| d.mapping(c)))

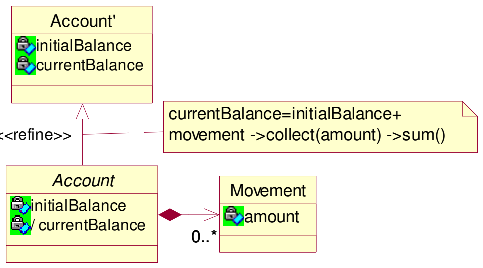
2.2 Refining by Decomposition

Generally, things are composed by smaller things, and this recursively. Composition is a form of abstraction: the composite represents its components in sufficient detail in all contexts in which the fact of being composed is not relevant. However, UML (and o-o modeling in general) has no notion of composition as a form of model abstraction. Consequently, the abstraction relationship remains hidden under composite association relationship. Figure 3a shows an example, where Account is a composite object holding a history of Movements. Additionaly the class Account has a basic attribute called initialBalance storing the value of the balance at the beginning of a period, and a derived attribute recording the current balance. Finally, there is an OCL constraint specifying how the current balance is calculated.

In [Steimann et al., 2003] it is observed that composite association, such as the one in figure 3a, is not a model abstraction relationship, which is reflected in the fact that Movement.extension is not included into Account.extension (In fact, there is a type mismatch between these two extension sets). Composite association is a relationship at the instance level (figure 4(a)); instances of Account are composed by instances of Movement.


Account
initialBalance
/ currentBalance
Movement
amount
0..*
/currentBalance=initialBalance+
movement ->collect(amount) ->sum()

(a)



(b)

Figure 3: Refinement hidden under decomposition: (a) Composite Association relationship. (b) Abstraction/Refinement relationship derived from the Composite.

However from a composite associations we can derive a model abstraction relationship called abstractions by composition (see figure 4(b)). In this case the relationship is established between models instead of being established between instances; for example, the Class Account' in figure 3b is an abstraction of the Class Account. Conversely, Account is a refined version of Account', showing more details (i.e. the fact of being a composite).

We define a mapping, reveal: Association → Abstraction, that returns the Abstraction artifact which is derived from each Composite Association artifact:

context Association **def**: reveal() : Abstraction
pre: self.connection–>select(e|
e.aggregation=#composite).size=1--the association is a composite-- and
self.connection–>forAll

[1] Addition of class intension with method overriding is defined in the following way: A ⊕ A' = (A.Attr ⊕ A'Attr , A.Ops ⊕ A'.Ops). Where ⊕ stands for relational overriding.

(ele.type.oclIsKindOf(Class)) -- the association connects classes --
post: result.stereotype=<<refine>> and
result.supplier = self.compound.hideParts and
result.client = self.compound
context Association **def**: compound (): Classifier -- returns the composite participant of the association--
post: result =self.connection–>detect(el e.aggregation=#composite).participant

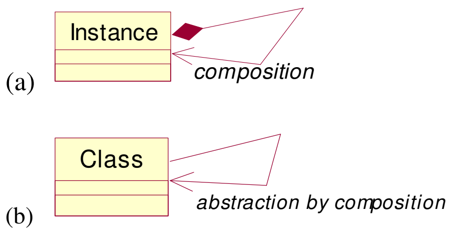


Figure 4: (a)Composite at extension level. (b)Composite at intension level.

Neither implementation nor abstraction functions can be automatically derived from the composite, because more than one design decision can apply. However some hints are provided automatically. In the example in figure 3 the specification of the derived attribute currentBalance is suggested as implementation mapping making it possible to translate OCL invariants such as

(**Context** Account' **inv** currentBalance>0) to a refined version

(**Context** Account **inv**
initialBalance+ (movements–>collect(amount)->sum > 0).

The retrieve relation R between Account and Account' is given by

$\forall$a,a'•(a,a')$\in$R $\leftrightarrow$ a.initialBalance=a'.initialBalance

and a.currentBalance= a'.initialBalance + a'movements->collect(amount)->sum

For example, the abstract object given by ⟨initialBalance==100, currentBalance=500⟩ is related to the concrete object ⟨initialBalance==100, currentBalance=500,movements= <⟨amount=400⟩> ⟩ among others.

Regarding the extension sets, if d is an Abstraction resulting from a composite Association, then

d.supplier.extension = d.client.extension->collect(ol d.mapping(o)), where the abstraction mapping transforms each composite object to its abstract representation.

2.3 Other cases

Other UML constructs hiding Class refinement are the Interface dependency, the Instantiation relationship, the Parameterization construct, among others. Due to space limitation we cannot explain here all these cases, but the results are similar to the ones presented above

3. Use Case Refinement

The Use Case construct is used to define the behavior of a system or other entity without revealing the entity's internal state. The intension of a Use Case consists of a pair (Attr, Ops) where Attr is a mapping from Attribute's name to Attribute's description and Ops is is a mapping from Operation's name to Operation's description. To simplify we consider only one operation because that is the usual case. Operation is defined by pre and post conditions.

The extension of a Use Case is a sequence of actions that the entity can perform interacting with actors of the system, such that if the precondition holds, after executing the sequence of actions, the post condition is ensured:

extension: Use Case $\rightarrow$ Set (Scenario)
Scenario = Seq(Action)
uc.*extension* ={<a_1,…,a_n> | uc.*pre*{<a_1,…,a_n>}uc.*post* }[2]

Use Case refinement is obtained by refining attributes and/or by constraining the operation specification. Therefore the refinement is satisfied by a reduced number of scenarios. In the following sections we analyze some forms of hidden Abstraction/Refinement relationship between Use Cases.

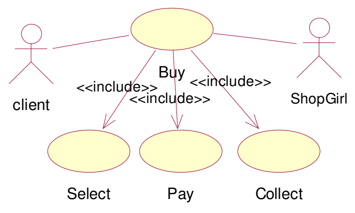
3.1 Refining by Action Decomposition

Action abstraction is the technique of treating an interaction between several participants as one single action. Then it is possible to zoom into, or refine, an action to see more detail. What was one single action is now seen to be composed of several actions. Each one

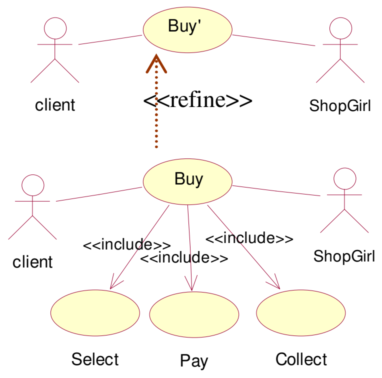
[2] Here < a_1,…,a_n > stands for a sequence of actions executable in some way. Using the formalism of Hoare´s logic we say that a sequence of actions S is correct with respect to precondition p and postcondition q (denoted p{S}q), when starting in any state that satisfies precondition p, the actions terminates in a state satisfying q. Function *pre* (respectively *post*) returns the precondition (respectively postcondition) of the (only) operation of the use case.

of these actions can be split again into smaller ones, into as much detail as required.

This form of abstraction remains hidden because of the fact that UML does not consider composition as a form of model abstraction. To specify composite actions UML provides a relationship between Use Cases called *Include*. Figure 5a shows an example, where Buy is a composite action holding three constituent parts: Select, Pay and Collect .



(a)



(b)

Figure 5: Use Case refinement hidden under a decomposition: (a) Composite/Component relationship. (b) Abstraction/Refinement relationship.

In the abstract model the action Buy is treated as a single action whereas the refined model shows that the action Buy is composed by three sub actions. It should be observed that Use Case inclusion, such as the one in figure 5a, is not a model abstraction relationship, but a relationship at the instance level (figure 6a). It specifies that each UseCaseInstance of the compound Use Case is composed by UseCaseInstances of the component Use Cases For example, let <Ana_buys_a_dress> be a UseCaseInstance belonging to the extension of the Buy Use Case. We may refine this instance revealing that actually it includes three instances inside it: <Ana_selects_a_dress>, <Ana_pays_for_the_dress > and <Ana_collects_her_dress>.

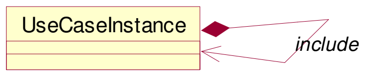
However, from a Use Case inclusion we can derive a model abstraction relationship called abstractions by composition (figure 6b). In this case the relationship is established between models instead of being established between instances. For example, the Use Case Buy' is an abstraction of the Use Case Buy (see figure 5b. Conversely, Buy is a refined version of Buy', showing the fact of being composed by three sub Use Cases.

The mapping, reveal: Include –> Abstraction, is defined to return the Abstraction artifact which can be derived from each Include artifact,

context Include **def**: reveal() : Abstraction
post: result.stereotype=<<refine>> and
    result.supplier = self.base.hideParts() and
    result.client = self.base

context UseCase **def**: hideParts(): UseCase -- returns an abstraction of the use case by hiding its parts--

(a)



(b)

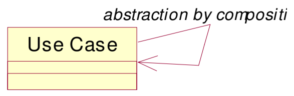


Figure 6: (a) UseCase inclusion at extension level. (b) UseCase inclusion at intension level.

3.2 Refining by Specialization

Use Cases are GeneralizableElements, so a Use Case may specialize a more general one. Figure 7a shows a general use case and its specialization. The general Use Case describes a Payment, while the specialization describes a particular kind of payment: PaybyCreditCard.

The inheritance mechanism allows modelers to describe Use Cases incrementally by describing extensions (increments) to previously defined Use Cases (its parents). Reusing in this way the more general specification.

The Figure 7 shows the connection between generalization/specialization relationship and abstraction/refinement relationship between Use Cases. The UML Generalization artifact in figure 7a relates two Use Cases: the parent, Pay, and the child, PayByCreditCard. The child is not a self contained model, it is just an increment of its parent. While, on the other hand the abstraction/refinement relationship in figure 7b relates self contained models that are obtained by combining the parent Use Case with the child Use Case. Attributes, preconditions as well as post conditions are combined.

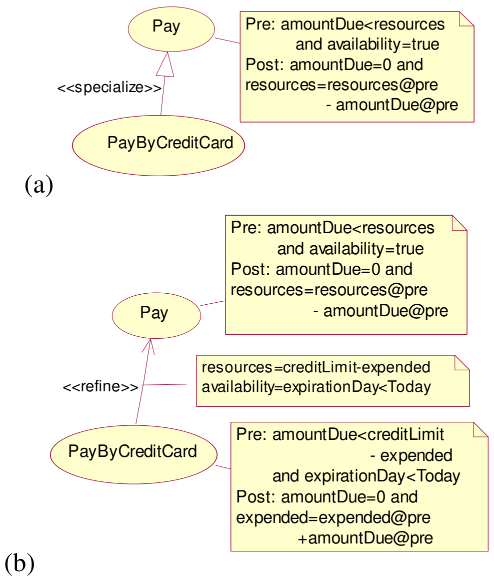


Figure 7: Use Case Refinement hidden under a Generalization: (a) Generalization/Specialization relationship. (b) Abstraction/Refinement relationship.

The mapping, reveal: Generalization → Abstraction, is defined to return the Abstraction artifact which is hidden under each Generalization artifact. The refinement is obtained by combining the parent and the child intension.

Context Generalization **def**: reveal(): Abstraction
pre: self.parent.oclIsKindOf(UseCase)
and self.child.oclIsKindOf(UseCase)
post: result.stereotype=<<refine>> and
result.supplier = self.parent and
result.client = (self.parent ⊕ self.child)

The abstraction mapping returns an abstract version of the UseCase by keeping only the features defined in the parent UseCase while forgetting the rest. On the other hand, the implementation mapping displayed in figure 7b is not automatically produced. The designer is asked to specify the relation between the attributes of the abstract use Case (i.e. amountDue, resources and availability) and the attributes of the refined Use Case (i.e. amountDue, creditLimit, expended and expirationDate), in the following way;

∀u,u'•(u,u')∈R ↔ (u.amountDue=u'.amountDue
and resources=creditLimit – expended
and availability= expirationDate<Today

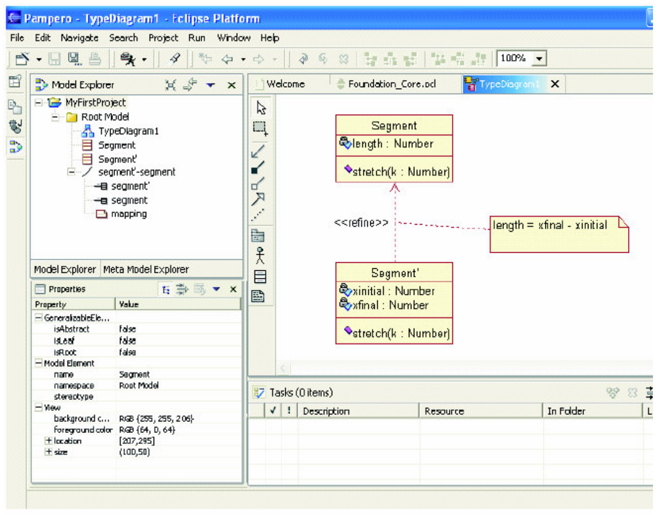
On the semantics side, as a consequence of the differentiation introduced by the specialization, the extension set of the abstract Use Case becomes partitioned in two or more sub sets, for example, let d be the abstraction artifact derived from the specialization of the use case Pay by the sub use cases PayByCheck and PayByCreditCard; Pay.extension is equal to (PayByCreditCard.extension–>union(PayByCheck.extension)) –>collect(e| d.mapping(e))

4. Tool support

The task of documenting refinement steps needs to be assisted by tools. We created PAMPERO [Pons et al., 2004] that is a plug-in to the Eclipse development environment [IBM, 2003]. It consists of four components: an UML editor, an abstraction/refinement translator, an evaluator, and a detective:

The Editor. The editor supports the creation of a number of UML artifacts, including Abstractions; see figure 8. Additionally, the editor allows developers to specify the abstraction mapping attached to Abstraction artifacts, using OCL expressions.

The abstraction/refinement Translator. The translator takes an OCL expression attached to a Class and translates it to concrete vocabularies, following the refinement steps. The translation of expressions attached to elements other than Class, is not supported yet.



Property	Value
GeneralizableEle...	
isAbstract	false
isLeaf	false
isRoot	false
Model Element	
name	Segment
namespace	Root Model
stereotype	
View	
background c...	RGB {255, 255, 206}
foreground color	RGB {64, 0, 64}
location	[207,295]
size	(100,50)

Figure 8. The PAMPERO tool: Edition of explicit refinement

The evaluator. The evaluator takes OCL expressions and evaluates them on a given model. Expressions might be either originally written in the model's vocabulary or translated by the translator from another abstraction level. The evaluator was implemented following the design of the USE evaluator [Richters and Gogolla, 2000].

The Detective. This component looks into the model to discover and reveal cases of hidden refinement. The abstraction mappings automatically generated by the detective are generally in an immature state and should be completed by the developer.

5. Conclusions

The traditional purpose of refinement is to show that an implementation meets the requirements set out in an initial specification. Armed with such a methodology, program development can then begin with an abstract specification and proceed via a number of steps, each step producing a slightly more detailed design which is shown to be a refinement of the previous one.

Although the UML allows for the explicit documentation of the abstraction/refinement relationship, an important amount of variations of this relationship remains unspecified, in general hidden under other notations. To enable traceability of requirements the presence of "undercover refinement" should be discovered and precisely documented.

When the mapping between the abstract and the concrete models is explicitly (and formally) documented, assertions written in the abstract model's vocabulary can be translated, following the representation mapping, in order to analyze if they hold in the implementation. Alternatively, instances of concrete models can be abstracted according to the abstraction mapping so that abstract properties can be tested on them.

The contribution of this article is to clarify the abstraction/refinement relationship in UML models, providing basis for tools supporting the refinement driven modeling process. PAMPERO, the tool reported in this article, is an evidence of the feasibility of the proposal.

References

Abadi, Martin and Cardelli, Luca. A Theory of Objects, Monographs in Computer Science, Springer, 1996.

Booch, G.. Object Oriented Analysis and Design with Applications. Benjamin Cummings, 1991.

Cardelli, L.,Wegner P. On Understanding Types, Data Abstraction and Polymorphism. Computing Surveys, 17(4). 1985.

Derrick, J. and Boiten,E. Refinement in Z and Object-Z. Foundation and Advanced. Applications. FACIT, Springer, 2001

D´ Souza, Desmond and Wills, Alan. Objects, Components and Frameworks with UML.Addison-Wesley. 1998.

Dijkstra, E.W., A Discipline of Programming. Prentice-Hall, 1976.

Eric, H. and.Sernadas, A. Algebraic Implementation of Objects over Objects. In stepwise Refinement of Distributed. Systems, Models, Formalism. . LNCS 430. 1989.

Giandini, R., Pons, C., Pérez,G. Use Case Refinements in the OO Software Development Process. Proceedings of CLEI 2002, ISBN 9974-7704-1-6, Uruguay. 2002.

Richters Mark and Gogolla Martin. Validating UML Models and OCL Constraints. Springer-Verlag, 2000. http://www.db.informatik.uni-bremen.de/projects/USE.

IBM, The Eclipse Project. Home Page. Copyright IBM Corp. and others, 2000-2003. http://www.eclipse.org/.

Lano,K. The B Language and Method. FACIT. Springer, 1996.

OMG. The Unified Modeling Language Specification – Version 1.5, UML Specification, revised by the OMG, http://www.omg.org, March 2003

Pons, C., Giandini R. and Baum G.. Specifying Relationships between models through the SD process, Tenth International Workshop on Software specification and Design, IEEE Computer Society Press. Nov. 2000.

Pons, C., Pérez,G., Giandini, R., Kutsche, Ralf-D. Understanding Refinement and Specialization in the UML. In. 2nd Int. Workshop on MAnaging SPEcialization /Generalization Hierarchies. In IEEE ASE 2003, Canada.

Pons, C., Giandini, R, Pérez., G., Pesce, P., Becker, V., Longinotti,J., Cengia,J., Kutsche, R-D., C.Neil. The PAMPERO Project: "Formal Tool for the Evolutionary Software Development Process". Home page: http://sol.info.unlp.edu.ar/eclipse. 2004.

Steimann,F., Göβner,J, Mück,T. On the key rol of compositioning object oriented modelling. Proceedings of the 6th Int. Conference <<UML 2003>>. LNCS 2863. Springer. 2003.

Wegner, P and Zdonik, S, Inheritance as an Incremental Modification Mechanism or What like is an isn't like. in proceedings 3rd European Conference on Object-Oriented Programming (ECOOP'88), Springer, 1988.