Revealing Undercover Refinement in UML Modeling

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

[1] LIFIA – Laboratorio de Investigación y Formación en Informática Avanzada, University of La Plata, 1900 Buenos Aires, Argentina
cpons@info.unlp.edu.ar
[2]*CIS Computation and Information Structures CIS,* Technical University of Berlin, Faculty IV, Berlin, Germany

ABSTRACT. 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 unspecified, in general hidden under other notations. 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. In this article we formally describe a number of undercover refinements and present PAMPERO, a tool integrated in the Eclipse environment, based on the formal definition of refinement. The tool supports the documentation of explicit refinements and the semi-automatic discovering and documentation of hidden refinements.

Keywords: UML, refinement, abstraction, traceability, CASE tool.

1. Introduction

Abstraction [Dijkstra, 76] 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. 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.

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.

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 **R**efinement **O**riented), 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 three UML artifacts: Classes, which are described in section 2; Associations presented in section 3 and Use Cases which are analyzed in section 4. 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=$\{a_1,...a_n\}$ is a set of Attribute's description and Ops= $\{op_1,...op_m\}$ is a set of Operation's description.

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*: The refined Class, B, is obtained by adding a new attribute, $attr_k$, to the abstract Class A. That is to say, B = A + ($\{attr_k\}$, {})[1]. Other case takes place when the Class B is obtained from Class A=($\{a_1, ..., a_k,..., a_n\}$, Ops) by replacing an attribute a_k by its refinement, that can be one or more new attributes, $a_{k1}, ..., a_{kl}$. That is to say, B= A[a_k | $a_{k1}, ..., a_{kl}$][2]. For example, figure 1b shows that the attribute *length* in Class *Segment* is refined by the attributes *xinitial* and *xfinal* through the mapping <length = xfinal – xinitial>.

b) *Operation Refinement*: The refined Class, B, is obtained by adding a new operation, op_k, to the abstract Class A. That is to say, B = A + ({}, $\{op_k\}$). On the other hand the Class B can be obtained from Class A=(Attr, $\{op_1, ..., op_k,..., op_n\}$) by replacing an operation op_k by its refinement, that can be one or more new operations, $op_{k1}, ..., op_{kl}$. That is to say, B= A[op_k | $op_{k1}, ..., op_{kl}$]. For example, figure 1c shows that the operation *stretch* in Class *Segment* is refined by the operations *moveXini* and *moveXfin* through the mapping <stretch(w) = moveXini(-w/2) ; moveXfin(w/2)>.

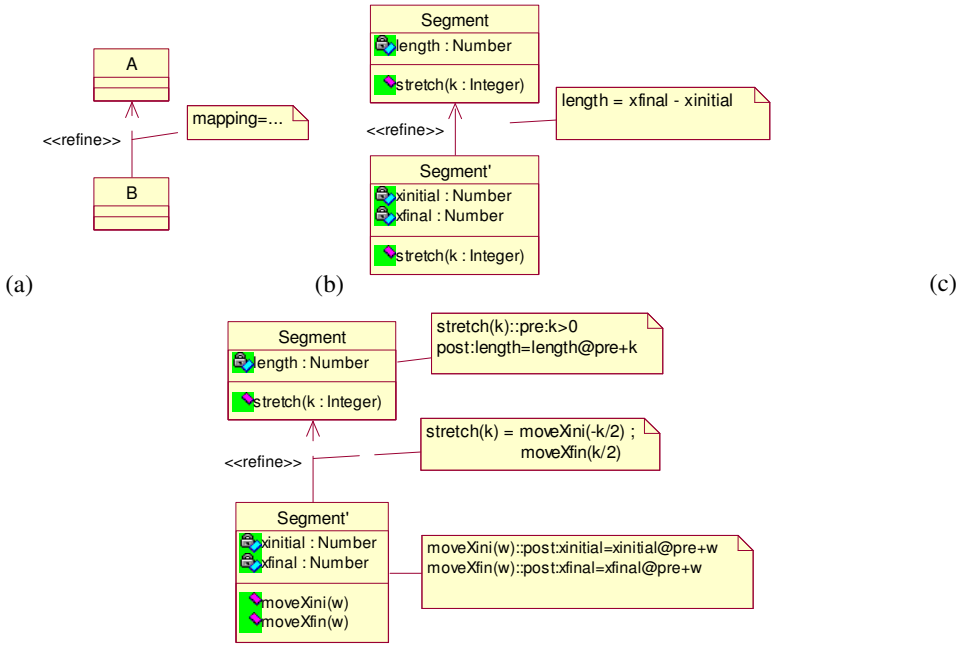


Figure 1: Abstraction/Refinement relationship between Classes. (a) UML notation. (b) Attribute refinement. (c) Operation refinement.

A refined Class is specified in a language that is richer than the language of its abstraction. However, there exists always a mapping from the abstract to the refined language (figure 2a), usually called "implementation mapping" [Cardelli and Wegner, 1985]. 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).

---

[1] Addition of Class intension is defined in the usual way: (A,O) + (A',O') = (A∪A', O∪O').

[2] B[y | $x_1,...,x_n$] denotes the replacement of element y in B by elements $x_1,.., x_n$.

Figure 2: Mapping domains. (a) Implementation mapping. (b) Abstraction mapping .

On the semantics side, the extension of a Class is the set of objects meeting its definition:

extension: Class → Set (Object)

The semantics counterpart of the "implementation mapping" is the "abstraction mapping" that transforms each refined object to its abstract representation (figure 2b). For example, in the view of Objects as labeled records [Abadi and Cardelli, 1996], the mapping attached to the Abstraction artifact in figure 1b, can be defined in the following way: Φ : Record → Record,

Φ ([xinitial=x, xfinal=y, stretch=m]) = [length=y-x, stretch=m]

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.*includes*(d.client.extension.collect(d.abstractionMapping))

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 [Aristotle], 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 3 shows the syntactical connection between Generalization and Abstraction. The UML Generalization artifact (Figure 3a) 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 3b) 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.

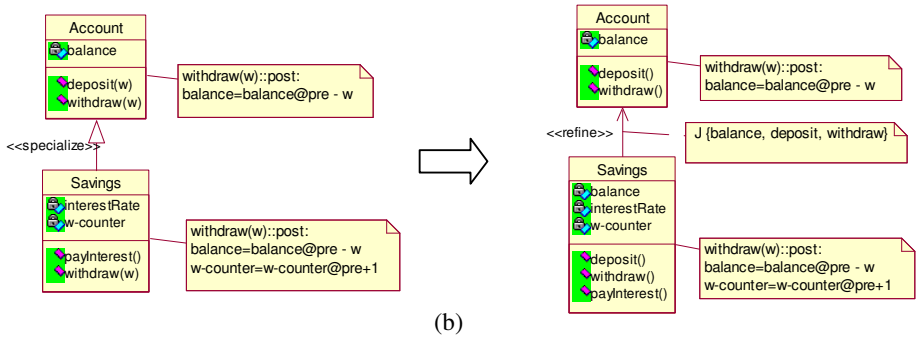
We define the mapping, reveal: Generalization → Abstraction, with the goal of making up the Abstraction artifact which is hidden under each Generalization artifact. In both cases the refinement is obtained by combining the parent and the child intension (considering method overriding if it is present).

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)[3] and

[3] Addition of class intension with method overriding is defined in the following way,

result.implementationMapping= id
result.abstractionMapping(r) = r⌡$_{\Sigma}$, [4]where Σ= self.parent.allFeatures

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 forgetting the rest.



(a) (b)

Figure 3: Refinement hidden under Specialization: (a) Generalization/Specialization relationship. (b) Abstraction/Refinement relationship derived from the Generalization.

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(d.abstractionMapping).
union(Checking.extension.collect(d.abstractionMapping))

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 4a 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 4a, 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); instances of Account are composed by instances of Movement.

Let A=(Attr, Ops) and A'=(Attr', Ops') and Base= Ops.reject(op| Ops'.collect(name).includes(op.name)),
A ⊕ A' = (Attr ∪ Attr' , Base ∪ Ops')

[4] The restriction functor ⌡$_{\Sigma}$ takes a labeled record and returns a restricted version of the record containing only those labels that are defined in Σ. For example: [balance=b, interesRate=r, w-counter=c, deposit=S1, withdraw=S2, payInterest=S3] ⌡$_{\{balance,\ deposit,\ withdraw\}}$ = [balance=b, deposit=S1, withdraw=S2].

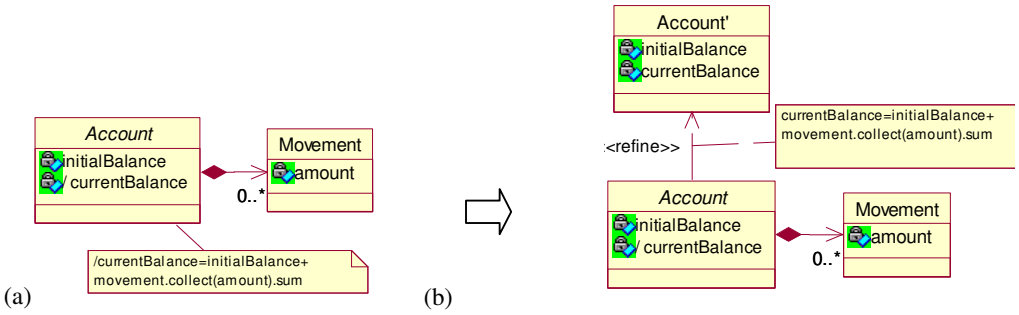


Figure 4: Refinement hidden under decomposition: (a) Composite Association relationship. (b) Abstraction/Refinement relationship derived from the Composite.

However from a composite associations we can derive a model abstraction relationship called abstractions by composition (see figure 5). In this case the relationship is established between models instead of being established between instances; for example, the Class Account' in figure 4b 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).

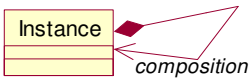


Figure 4: Composite at extension level.

Figure 5: Composite at intension level.

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(el e.aggregation=#composite).size=1--the association is a composite-- and
self.connection.forAll (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).type

context Classifier **def**: hideParts(): Classifier -- returns an abstraction of the classifier by hiding its parts--

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 4 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).

Regarding the extension sets, if d is an Abstraction resulting from a composite Association, then
d.supplier.extension = d.client.extension.collect(d.abstractionMapping), where the abstraction mapping transforms each composite object to its abstract representation. For example, the abstraction mapping in figure 4b is defined in the following way:

d.abstractionMapping ([initialBalance=b, currentBalance=f, movements=s]) =
[initialbalance= b, currentBalance= b + s.collect(amount).sum]

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. Association Refinement

The UML provides an artifact named Association to specify relationships between instances of Classifiers. The association's intension contains the declaration of the types involved in the association, as well as other properties such as navigability, multiplicity, etc.

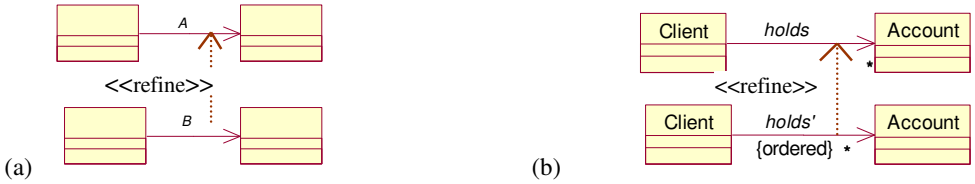


Figure 5: UML Abstraction/Refinement relationship between Associations. (a) UML notation. (b) Association refinement by constraining properties.

During the development process both classes and its relationships are gradually refined. It is usual to zoom in or out in both dimensions at the same time. Figure 5a shows the UML artifact to specify Abstraction/Refinement relationship between Associations. A form of explicit refinement takes place when an abstract association is constrained in some way, for example disallowing navigability, adding the constraint that the elements should be ordered, etc. Conversely, the abstraction is obtained by relaxing properties from the refinement. Figure 5b displays an example, where holds' is a refinement of holds.

Concerning the extension set, Associations are interpreted as relations in the mathematical sense, i.e., as subsets of the Cartesian products of the extensions of the involved types. The extensions of an association are sets of tuples called links:

extension: Association -> Set (AssociationInstance)
AssociationInstance = Set(Link)

The refinement is a more constrained specification, therefore it is satisfied by a reduced number of instances:

Context d:Abstraction **inv**:
d.supplier.extension.*includes*(d.client.extension.collect(d.abstractionMapping))

Refinement between Association can be explicitly specified, as described above; however there are other cases that remains hidden and should be discovered and made explicit. The different forms of undercover Abstraction/Refinement relationship between Associations are analyzed in the remainder of this section.

3.1 Refining by Specialization

Consider that a specialization is applied to the class Account generating two classes: Savings and Checking. At the same time, a specialization is applied to Client and a new class comes into existence: YoungClient. In this domain YoungClients are allowed to open only Savings while the remaining Clients may hold both Checking and Savings accounts (see figure 6a).

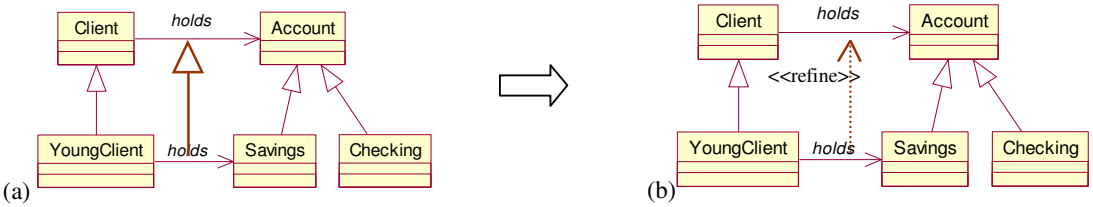


Figure 6: Association refinement hidden under a sub classification: (a) UML Generalization/Specialization relationship. (b) UML Abstraction/Refinement relationship.

Generalization between Associations is subtly different from Generalization between other UML artifacts. While generalization hierarchies in general reflect incremental development, generalization hierarchy of association is not incremental, but restrictive. The child Association frequently is a self contained model not just an increment, adding some constraints (e.g. restricting AssociationEnd´s type, disallowing navigability, adding the constraint that the elements should be ordered)

Figure 6 shows the syntactical connection between Generalization and Abstraction. The UML Generalization artifact in figure 6a relates two associations: the parent and the child. In this case both are self contained models. Therefore, the Generalization artifact can be simply replaced for an Abstraction artifact, see figure 6b, without altering the model's meaning.

The mapping reveal: Generalization → Abstraction, returns the Abstraction hidden under the Generalization.

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

Semantically, the generalization/specialization relationship between associations denotes an inclusion relation between the corresponding extension sets. Additionally, this form of refinement splits the set of abstract links into two or more subsets. For example, the association Client-*holds*-Account is partitioned into two refined sub-associations, as follows: Client-*holds*-Account.extension = YoungClient-*holds*-Savings.extension.union(ElderClient-*holds*-Account.extension), where ElderClient denotes the set of Clients who are not Young clients. Account abstractly denotes the union of both Savings and Checking.

3.2 Refining by Link Decomposition

A group of semantically related associations can be subsumed by an abstract association. For example Figure 7 shows two association between Client and Bank: *holdsAccount* and *hasCredit*. These two associations can be abstracted in a single association, called *worksWith*.

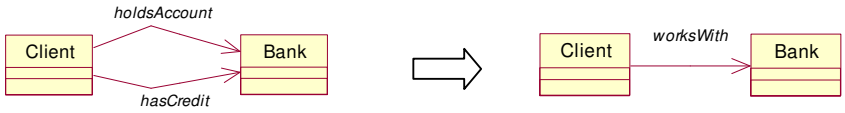


Figure 7: association abstraction.

In the UML Associations cannot be compound (only Classifiers can be compound). However, other languages, such as the one proposed by Catalysis [D'Souza and Wills, 1998], enable us to specify composition of Associations; see figure 8a.

Composition of association is neither a form of model abstraction in the UML. You could feel tempted to use the Abstraction artifact to specify composition of Associations, but to declare that *hasCredit* is a refinement of *worksWith* is as wrong as saying that Movement is a refinement of Account, in model in figure 3, despite the fact that here the type mismatch is not so evident.

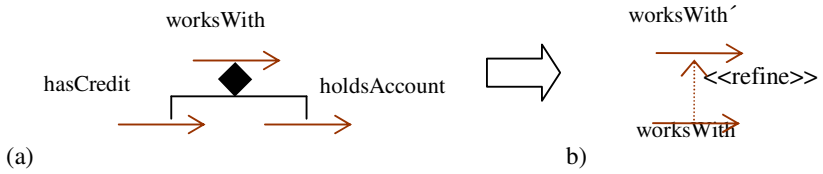


Figure 8: Association refinement hidden under link decomposition: (a) Composite/Component relationship in Catalysis. (b) Abstraction/Refinement relationship derived from the Composite.

In parallel direction to the composition between classes, the composition between associations is a relation at the instance level, specifying that each link of the compound association is composed by links of the component associations (figure 9). For example, let (c1,b1) be a link belonging to the extension of worksWith association, meaning that a Client c1 works with a Bank b1. We may zoom into this link revealing that the working relationship between c1 and b1 actually embraces two relationships: c1 has a credit in b1 and c1 holds an account in b1. That is to say, the link (c1,b1) of the compound association contains links of the component associations.

Figure 9: composition at the extension level.

Figure 10: composition at the intension level.

Once again, from a composition we can derive a model abstraction relationship called abstractions by composition (see figure 10). In this case the relationship is established between models instead of being established between instances. For example, the Association worksWith´ is an abstraction of the Association worksWith (see figure 8b). Conversely, worksWith is a refined version of worksWith´, showing the fact of being composed by two sub-associations.

The mapping, reveal: Association → Abstraction, returns the Abstraction artifact which can be derived from each decomposition of Association. Assuming that the UML metamodel is modified in order to permit the definition of Associations between Associations, the mapping is formally defined in the following way:

context Association **def**: reveal() : Abstraction
pre: self.connection.select(e| e.aggregation=#composite).size=1--the association is a composite-- and
self.connection.forAll (e|e.type.oclIsKindOf(Association)) -- the association connects associations--
post: result.stereotype=<<refine>> and
result.supplier = self.compound.hideParts and result.client = self.compound

context Association **def**: hideParts(): Association -- returns an abstraction of the association by hiding its parts--

4. 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=(a_1,...a_n) is a set of Attribute's description and Ops= (op_1,...op_m) is a set of Operation's description. To simply 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 → Set (Scenario)
Scenario = Seq(Action)
uc.*extension* = {<a_1,…,a_n> | uc.*pre*{<a_1,…,a_n>}uc.*post* }[5]

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. The meaning of use case refinement is that the extension of an abstraction includes the abstract representation of the extensions of all its refinements:

Context d:Abstraction **inv**:
d.supplier.extension.*includes*(d.client.extension.collect(d.abstractionMapping))

In the following sections we analyze some forms of hidden Abstraction/Refinement relationship between Use Cases.

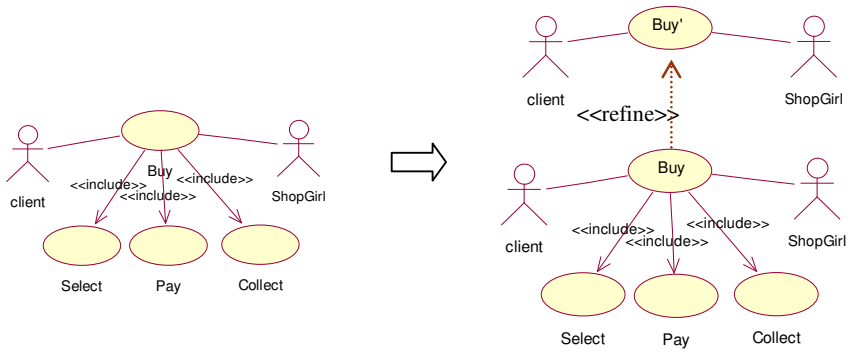
4.1 Refining by Action Decomposition

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 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 11a shows an example, where Buy is a composite action holding three constituent parts: Select, Pay and Collect .

[5] 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.

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 11a, is not a model abstraction relationship, but a relationship at the instance level (figure 12). 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>.



(a) (b)

Figure 11: Use Case refinement hidden under a decomposition: (a) Composite/Component relationship. (b) Abstraction/Refinement relationship.

Figure 12: UC inclusion at extension level.

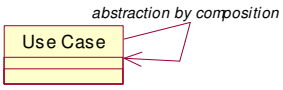


Figure 13: UC inclusion at intension level.

However, from a Use Case inclusion we can derive a model abstraction relationship called abstractions by composition (figure 13). 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 11b. 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--

4.2 Refining by Specialization

Use Cases are GeneralizableElements, so a Use Case may specialize a more general one. Figure 14a 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.

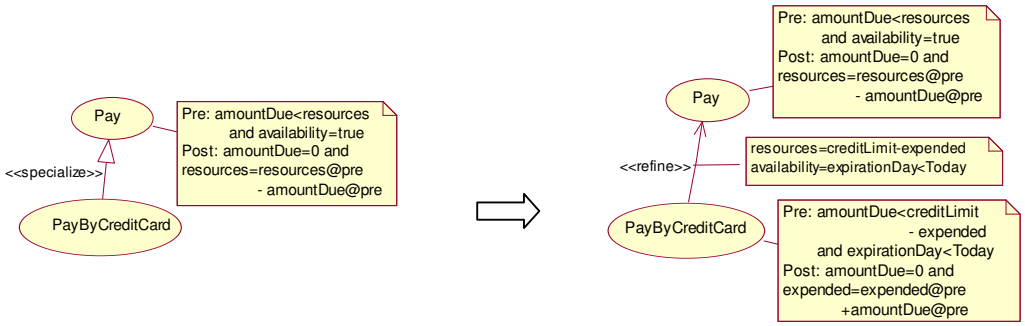
The inheritance mechanism allows modelers to describe Use Cases incrementally by describing extensions (increments) to previously defined Use Cases (its parents). Reusing in this way the more general specification.

The Figure 14 shows the connection between generalization/specialization relationship and abstraction/refinement relationship between Use Cases. The UML Generalization artifact in figure 14a 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 14b 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.

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)[6] and
result.abstractionMapping(r) = $r\downarrow_{\Sigma}$ [7] , where Σ= self.parent.allFeatures



(a) (b)

Figure 14: Use Case Refinement hidden under a Generalization: (a) Generalization/Specialization relationship. (b) Abstraction/Refinement relationship.

The abstraction mapping returns an abstract version of the object by keeping only the features defined in the parent Class while forgetting the rest. The implementation mapping displayed in figure 14b 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).

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(d.abstractionMapping)

5. 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:

---

[6] Addition of Use Case intension with operation overriding is defined in the following way,
Let U = (Attr, (pre_{op}, $post_{op}$)) and U'= (Attr', ($pre_{op'}$, $post_{op'}$))
U ⊕ U' = (Attr ∪ Attr' , (pre_{op} or $pre_{op'}$, $post_{op}$ and $post_{op'}$))

[7] The restriction functor $\downarrow_{\Sigma}$ takes an Scenario and returns a restricted version of the scenario containing only those actions that are defined in Σ.

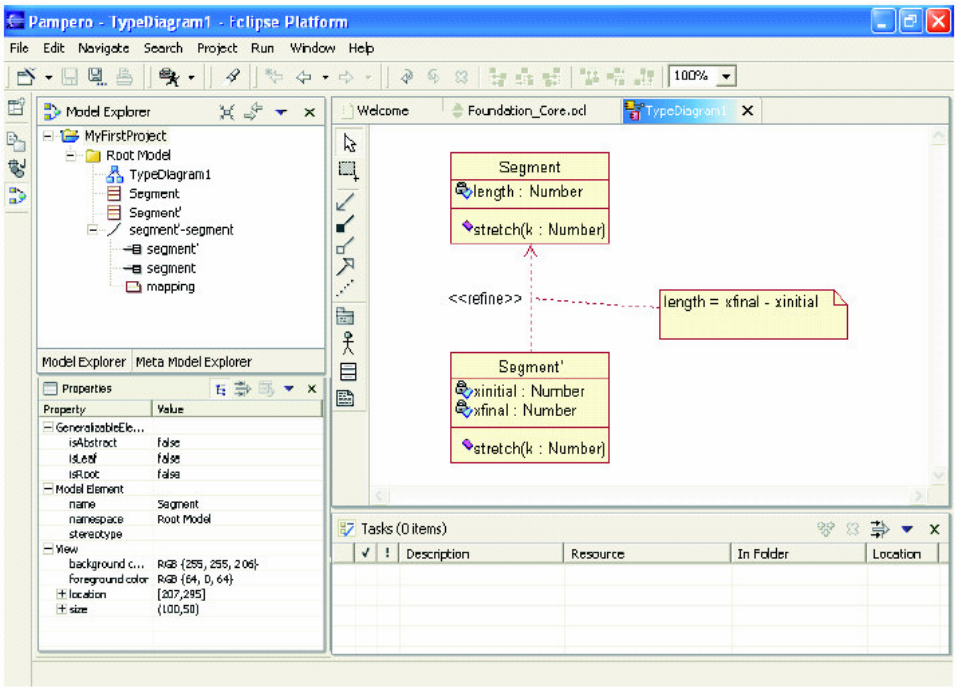


Figure 15. The PAMPERO tool: Edition of explicit refinement.

The Editor. The editor supports the creation of a number of UML artifacts, including Abstractions; see figure 15. 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.

The evaluator. The evaluator takes OCL expressions and evaluates them on a given model (see figure 16). Expressions might be either originally written in the model's vocabulary or translated by the translator from another abstraction level.

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.

Additional components, such as the *Refinement Checker*, are being currently designed to enhance the tool; the Refinement Checker will be able to prove the existence of a formal refinement relationship between model elements, for instance given an operation refinement, the checker will prove that the precondition of the abstract operation implies the precondition of its refinement and the postcondition of the abstract operation is implied by the postcondition of its refinement.

6. Conclusions

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, although its application to industrial cases is a pending task which is essential to determine its scalability and practical advantages.

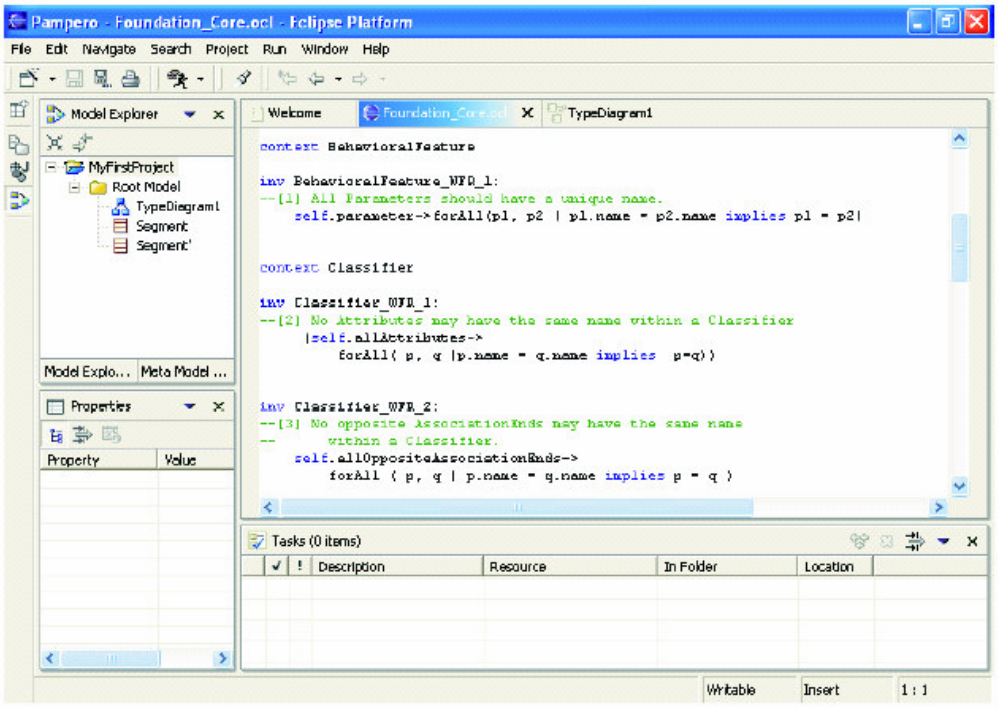


Figure 16. The PAMPERO tool: Evaluation of OCL constraints.

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.

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, Correctness. LNCS 430. 1989.

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.

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

OMG. The Unified Modeling Language Specification – Version 1.4, UML Specification, revised by the OMG, http://www.omg.org, September 2001

Pons, Claudia, Giandini Roxana and Baum Gabriel. Specifying Relationships between models through the software development process, Tenth International Workshop on Software specification and Design, San Diego., IEEE Computer Society Press. November 2000.

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

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

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.