For a UML "Core" – Overview

Desmond DSouza, Kinetium, desmond@kinetium.com

1 Introduction

These are some issues for the UML 2.0 "core" i.e. some part of the UML infrastructure RFP, and potentially part of the MOF. The approach here is motivated by some simple observations:

1. Every time we unify two separate concepts in the current UML into one in the 2.0 "core", we significantly reduce the combinations that higher level constructs and profiles have to consider. Consider if the 2.0 core could fully unify objects and values, attributes and associations, states and attributes, patterns and template classes/ collaborations, activities and … .

2. One good approach would be an architecture in which new constructs can be defined as translations into core constructs. A good pattern capability would provide this: the abstract description of the pattern such as subject-observer is the higher-level construct (optionally with its own concrete or abstract syntax); the mechanism for realizing it with some structure of inheritance, adaptors, interactions, … is the translation.

3. The basic modeling constructs can then be consolidated to a bare minimum (e.g. the static part could become just objects, attributes, and static invariants), with other constructs defined as patterns that translate to this core (state, association, …)

2 Basic Packages and Import

The package is one of the most important constructs. Some core clean-up of packages and imports:

- No hiding of package elements in the core i.e. effectively, all elements are public.

  K-1. Different kinds of hiding can be built atop the core by factoring models across packages (Section 3.6), or by renaming (Section 3.3).

- Package import should be transitive.

  K-2. Consider a simple re-factoring operation on a package P1, where some of its elements are moved to a new common shared package P0, just as we might factor out parts of a class into a superclass. Packages that import P1 should not be affected by such a change.

- Every package should be self-contained i.e. every name used in that package should be adequately defined either within that package, or via one of the packages that it imports.

3 Patterns in the "Core"

3.1 Examples of Patterns

We start with some examples of things that could be described as patterns. Below we use *italics* to indicate a variable part of a pattern, and do not show the formal OCL.

1. Association
   - Association with roles *r1* and *r2* (and multiplicities *m1* and *m2*) between *C1* and *C2*
   - This translates into an attribute *r1* on *C1*, and attribute *r2* on *C2*, and OCL constraints on those two attributes for their multiplicities and their 'inverse' nature.
2. Aggregation
   - An aggregation *r1* between *C1* and *C2* [with your favorite syntax]
   - This translates to an association *r1* from *C1* to *C2*, constrained to a tree structure, and with [your favorite variation of] some lifetime or delete dependency.
3. Namespace
   - A relationship between a *NamingContext* and some *NamedElements* in which the *Context* associates a *Name* with each *NamedElement*, with a uniqueness rule.
   - Note: UML and MOF both mix namespaces with issues of containment and ownership. This makes it very hard to define some kinds of scope and renamings.
4. Unique
   - Attribute *a1* of *C1* is unique in the context of some association *r1* from *C0* to *C1*

   - This translates into a simple OCL constraint.

5. Template classes

   - Collection *C* with elements of type *T* has attributes and operations that refer to *T*.

6. State

   - A state *S* translates into an attribute *S:*Boolean

   - The structure of state chart translates into a (static) invariant on those attributes e.g. isOn = not isOff.

7. Java Bean Property

   - An attribute *A* of type *T* on classifier *C* can be marked as a <<property>>

   - This translates to a `get`*`A`*`():` *`T`* and `set`*`A`*`(t: T)` pair of methods

8. Stereotypes

3.2 Package = Pattern Definition

Patterns, in general, involve relationships between model elements (classes, objects, OCL constraints, interactions, …). The UML package is a container for interrelated model elements.

K-3. A pattern is any set of interrelated model elements defined in a package, some of whose elements will be substituted when that package is imported into another. In other words, no new construct is required to *define* a basic pattern. At the most we could add an indication for elements earmarked for substitution, but this is not required in the core; any element is substitutable.

3.3 Import + Renaming = Pattern Application

To apply pattern P to some target elements in package P1 means that P1 imports P, and renames some of the elements of P to correspond to the target elements. Renamings are associated with each import. The result in package P1 is that all elements with the same name have their definitions merged. [Note – the UML really should have dealt with this merging issue already. A class can be incrementally defined across multiple diagrams. These definitions have to be merged to produce the resultant definition of that class].

K-4. To apply a pattern to some model elements by substituting those elements for elements from the pattern definition, the meta-model must properly support namespaces and renaming. It must also define merging of model fragments.

K-5. Issues like "what is a name" are abstracted by having a Name type with a sameAs(Name): Boolean operation.

K-6. An element that is not named by a package, or that has been renamed to "", is not accessible via that package. This provides a simple way to define different kinds of hiding rules on top of a simple core.

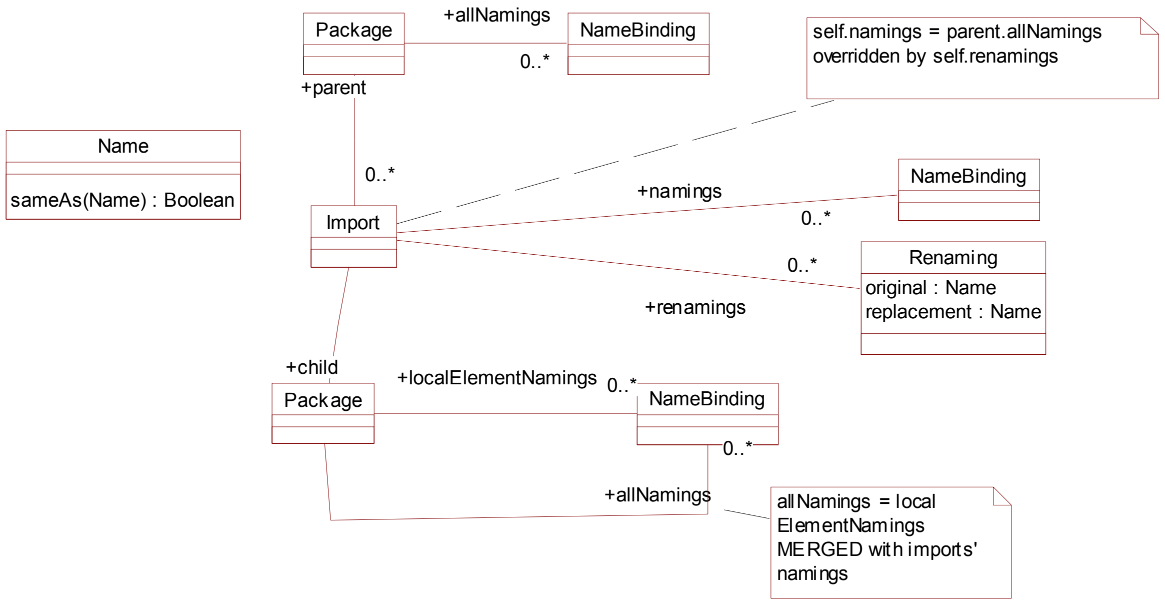


Figure 1 Basic meta-model for patterns

So an import can refer to *names of elements* in the importing and imported package. In addition, it is very useful to extend the idea of import substitutions to not just explicitly named elements, but also elements that can be accessed from a named element via some traversal of the known meta-model e.g. the *multiplicity* of an association, or the *entry-point* of a method.

3.4 Patterns have "Pre-conditions"

Most patterns are only applicable if the elements that are substituted satisfy certain properties. For example, the *CorbaAttribute* pattern may be applicable only to attributes of *CorbaInterfaces*; or, the *OrderedCollection* pattern may only be applicable if the *ElementType* have an appropriate definition of ordering. Even the C++ standard template library (STL) is full of such cases.

K-7. The pattern meta-model should permit definition of constraints or requirements on the substituted elements.

3.5 Identifiers are Parameterized

Patterns are not just about structures of model elements in the UML sense, but also about the names used to refer to those elements. Names indicate roles in the pattern, hence names themselves need to be structured terms. This leads to parameterized identifiers e.g. *get_<attr>* and *set_<attr>* could be the structured names of two methods, parameterized by the name of some attribute *<attr>*. Hence patterns, when applied, can generate meaningful interrelated names (modulo internationalization). Name equality needs minor re-definition to handle such structured names.

K-8. Identifiers in patterns can be parameterized by other identifiers.

3.6 MOF Package Generalization as Special Case of Patterns

MOF has a concept of package generalization; however, it is not uniform and makes unnecessary special restrictions. For example, it allows for package P2 to import P1, and introduce a new association to P1::A, and to add new OCL constraints. However, it does not allow P2 to add new attributes, operations, or superclasses to P1::A.

Unfortunately, this is inconsistent. There could be a common association, *r*, that is needed by P1::A and P1::B; this commonality should be handled by introducing a superclass of A and B *within package P2*. Similarly, adding a new association is no different from adding attributes (the MOFs current distinction of attribute, association, and reference appears to be motivated by implementation issues); in fact, a new association may directly introduce derived attributes, such as a summary value from a 1-N association.

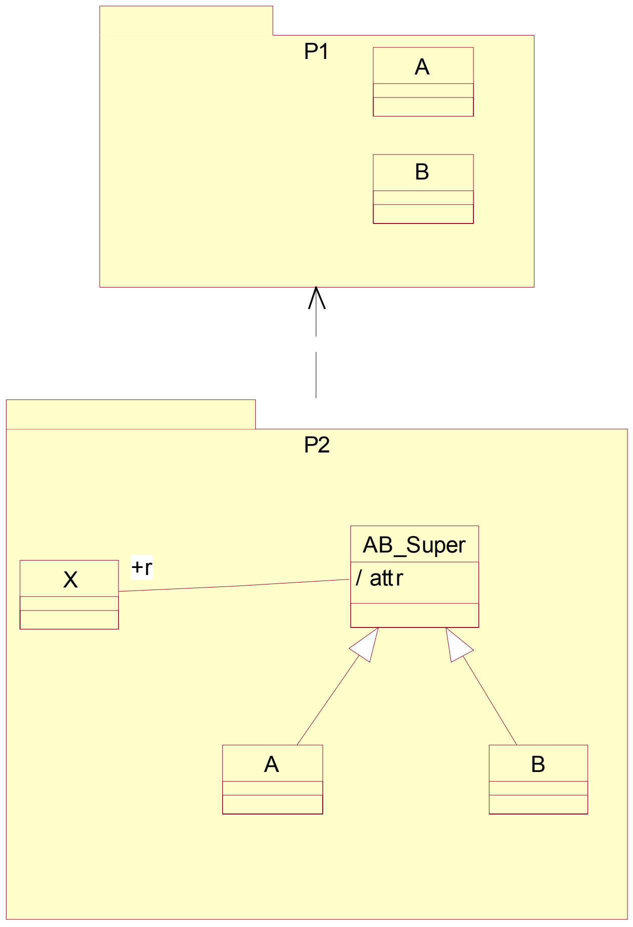


Figure 2 MOF package generalization limitations

Further, an added OCL constraint could well need a definition of a new attribute or operation. Moreover, constraints are just a textual form of models: if a child package can add new OCL for a class from a parent package, it can effectively define new attributes and operations on that class using "**def:**", so to be consistent we should permit this in non-text models as well.

This uniform version of package generalization can be handled by correctly doing patterns. All that package generalization does is implicitly rename each element of P1 to the same name in P2. The merging of elements from patterns provides package generalization.

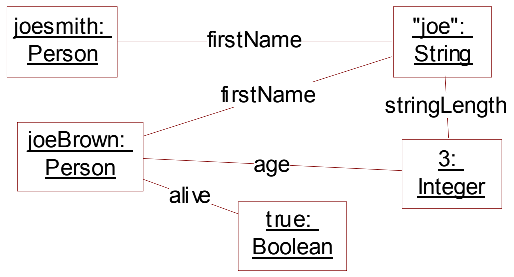
This form of package extension could be easily motivated with an example:

K-9. The OCL package provides a definition of collections and numbers. Suppose we need to extend these definitions e.g. add statistical properties to numbers, or add more convenient subsequence operators on collections. Creating and naming subclasses is not the right solution, since our intent is to define new properties of the *same* set of objects.

4 *Unify Objects and Values*

4.1 Objects and Named Links

Objects are distinguishable individuals in some world. Objects can include people, components, numbers, dates, classes, procedure activations, booleans. We consider objects to have labeled directed links to others, giving a uniform graph-like conceptual view of the state of a world that includes persons, dates, numbers, strings, and booleans. Some objects always existed, some come and go dynamically. Some links always existed, some are immutable, some change with time. Some links and objects would be highly optimized in an implementation e.g. a 2's complement encoding of links to integers enables very efficient traversal of the immutable links between integers, without ever explicitly creating an integer 'object'. Mutable vs. immutable can be problem dependent.



K-10. Products, numbers, classes, times, … all individuals are modeled uniformly as objects with named links to others in a platform independent way. The relevant properties and specification styles varies between immutable and mutable objects.

The snapshot is translatable to equivalent logical statements, where the identifiers `joeSmith`, `joeBrown` are just variables (named links to objects) of type `Person`:

```
joeSmith, joeBrown: Person;
joeSmith.age = 3;    // 3 is a (constant) name for the object
joeBrown.alive = true;
```

The core has a built-in concept of object identity, without separate value types. *But aren't equality checks different from identity checks?* A `String` value type might have `equal` defined as the same sequence of characters. Modeled as an object, `String` simply has an invariant constraint saying `strings` can be `equal` only if they have the same identity (and mutative operations are replaced by corresponding non-mutative ones)[1]:

```
d1.equals(d2) implies d1 = d2
```

[1] This is *specification* object identity and can have different implementations. For example, Java == or equal are needed depending on whether specification objects are implemented as shared code objects or as duplicate copies. Java strings, implemented as duplicate copies compared with equal, could have been implemented with a smart constructor that ensured a single shared copy of any string.

Thus, value equality is replaced by object identity with an identity constraint; all individuals become objects. Thus, there is really no difference behind these three statements:

- No two people can have the same social security number
- No two drivers licenses can have the same state as well as the same serial number
- No two dates can have the same day, same month, and same year.

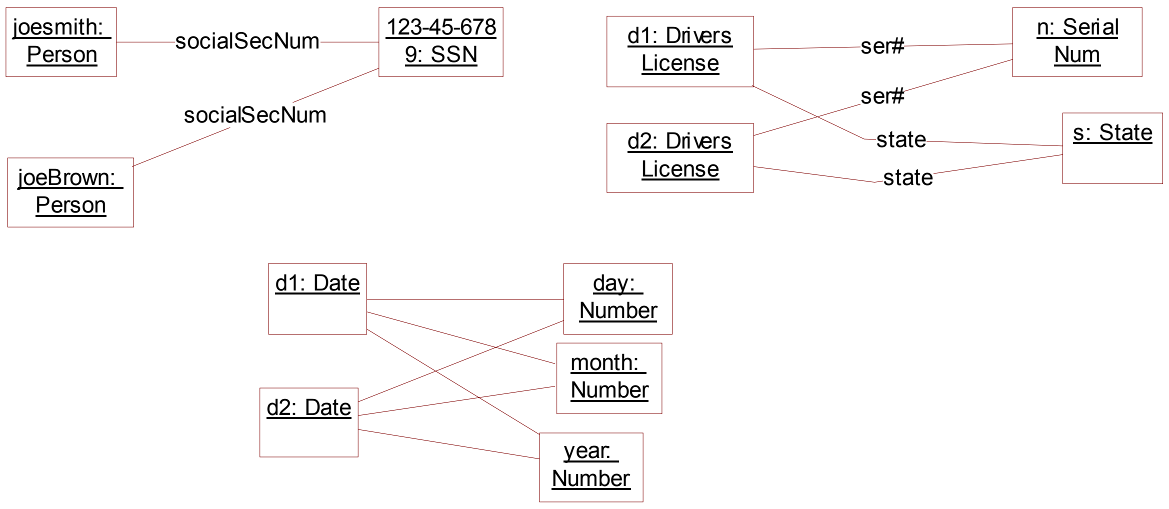
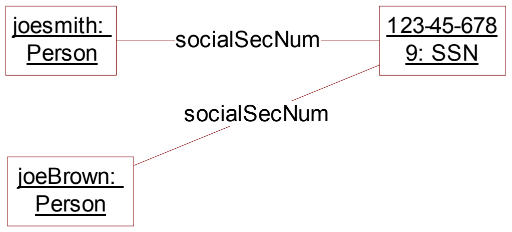


Figure 3 Uniform Treatment of Identity Constraints

Additional problem-specific *similarity* checks may still be needed; these are distinct from identity or equality checks.

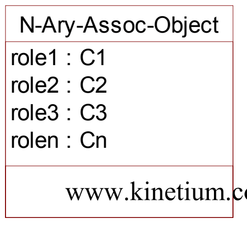


K-11. 'Value types' are modeled as objects with an inequality constraint on any two distinct objects.

K-12. There is no need for anything like a "composite identity" e.g. it is sometimes suggested that the identity of the association instance is the composition of the identities of the linked objects. If you need to refer to tuples as objects, go right ahead: a tuple has an identity just like any regular ol' object, and an additional constraint saying no two distinct tuples (different identities) can have the same linked objects.

This applies to the meta-models as well. e.g. UML defines an N-ary association as: "Each instance of the association is an n-tuple of values from the respective classifier. The multiplicity on a role represents the potential number of instance tuples in the association when the other N-1 values are fixed." The 'n-tuple of values' is a ordinary object type with an identity constraint:

```
context N-Ary-Assoc-Object
  N-Ary-Assoc-Object->forAll (t |
       t.r1 = self.r1 &  -- identity checks
```



```
        t.r2 = self.r2 &
        ...
        t.rn = self.rn
        implies t = self) – identity constraint
```

Platform issues like Corba call-by-value, besides being separable into a distinct meta-model, can be mapped directly to call-by-reference to a copy.

K-1. Platform specific distinctions of data type, parameter passing conventions, exceptions, etc. are mapped to the core in separate meta-models or profiles.

Common questions:

- *But you do not pass value types around by reference.* At the conceptual level you have to have a notion of "individuals", whether persons, numbers, or dates. All object modeling, and for that matter all predicate logic, is based on this. Thus, there is a *conceptual* notion of object identity – called identity by the object crowd, and equality by others – that is the basis of writing specifications that refers to individuals. Now, you have to map your conceptual notion of individuals and identity to your implementation. Sometimes it is appropriate to use *implementation references* (e.g. pointers) to indicate conceptual identity; at other times it is better to have a *"fat reference"* i.e. encoded into the *reference itself* is some information about the thing referred to.

  Thus, passing around a copy of [day, month, year] for dates is no more than passing around a fat encoding that refers to the conceptual date; determining if two of these refer to the same date object, of course, means comparing the relevant parts of that encoding.

  Similarly, a 2's complement encoding of a reference to the integer **5** provides very efficient traversal of the immutable *next* and *previous* links that every integer has. So, conceptually every integer has attributes *next* and *previous*, and 2's complement is just a very efficient and systematic encoding of those references. This is very similar to a common programming optimization of references to objects: if you know from domain knowledge that you have a small fixed numbers of objects, and that you need a *next* for each of those objects, you can represent those objects in an array, and encode the representation and traversal of *next* into array indexes / memory addresses.

- *But Corba does have value types.* Absolutely. And Corba's notion of values should be defined in the Corba profile, and not in the UML or MOF core. Also, such value types and call-by-value rules are can be modeled as call-by-reference to a copy. This also makes explicit the definition of *copy* that is used, where some platforms (C++) will truncate information when calling by value, and others (Java RMI) will pass through a complete copy, including polymorphic behaviors.

- *But the number 5 does not understand the "+" message.* The decision of how to model behaviors (only single-receiver localized operations, or more general joint-actions as may be more suited for use-cases and business actions and events) is separate from uniformly treating all individuals as objects with identity.

- *But this will complicate things like enumeration types.* Actually, it makes it more uniform. Most types define a set of objects that are members of that type by specifying their properties. An enumeration type simply lists, and names, its members directly. Thus an enumeration type is just a set of names for its member objects, with the constraint that each name denotes a distinct object. This also enables us to define richer sets of objects as enumerations.

4.2 Object / Reference / Reified Reference is Fractal

Just as actions get reified, so do object references. This includes parameters and stored references – a virtual reference becomes a particular new "name" value type with some name resolver. See ODP engineering viewpoints.

4.3 Attribute Types

Attribute types should include sets, sequences, and maps. To motivate this, consider why any of the following should be modeled as operations instead of as attributes:

- car.wheel1, … car.wheel4
- Cars have 4 ordered wheels: car.wheels.first
- Trucks have a whole bunch of ordered wheels: trailerTruck.wheel (n)
- The price of a product: product.price
- The base price and incremental unit price of a product: product.base_price, product.incremental_price
- The price of some quantity of a product: product.priceFor (quantity)

UML already permits specification of multiplicity and ordered constraints on attributes. Presumably the elements in an ordered attribute set can be referred to by their position in that order e.g. In other words, if we allow attributes to have any defined type, including sets, maps, etc., we get better abstractions of object state. Almost all specification languages provide similar facilities to define and structure object state.

Common questions:

- *But we already have operations for doing this.* If attributes are supposed to abstract the state of an object, there is no real reason for switching to operations just because we need a better abstraction of object state.

  This also simplifies the meta-model. If we separately specify the meta-model for statics from that for dynamics, then the static meta-model does not need operations at all.

- *Post-conditions vs. invariants*. Invariants hold at "all times". State relationships hold at all times.

- *Style of specifying queries*. Query functions become trivial; all complexity is in the state invariants. Operation specs become greatly simplified with convenience attributes, especially convenience parameterized Boolean queries.

5 Unify Activity and Action

State machines, operations, actions, and activity diagrams unified with **Action**.

6 Unify Causality

Key question:

7 Refinement

One of the most visible demands placed on the UML by the new MDA initiative is a proper treatment of refinement. Refinement is a relationship between two models, one of which is strictly more detailed than the other and which maintains the guarantees made by the other. The meta-model for refinement needs to be very close to the "core" of UML, since many other things will need to be related to refinement.

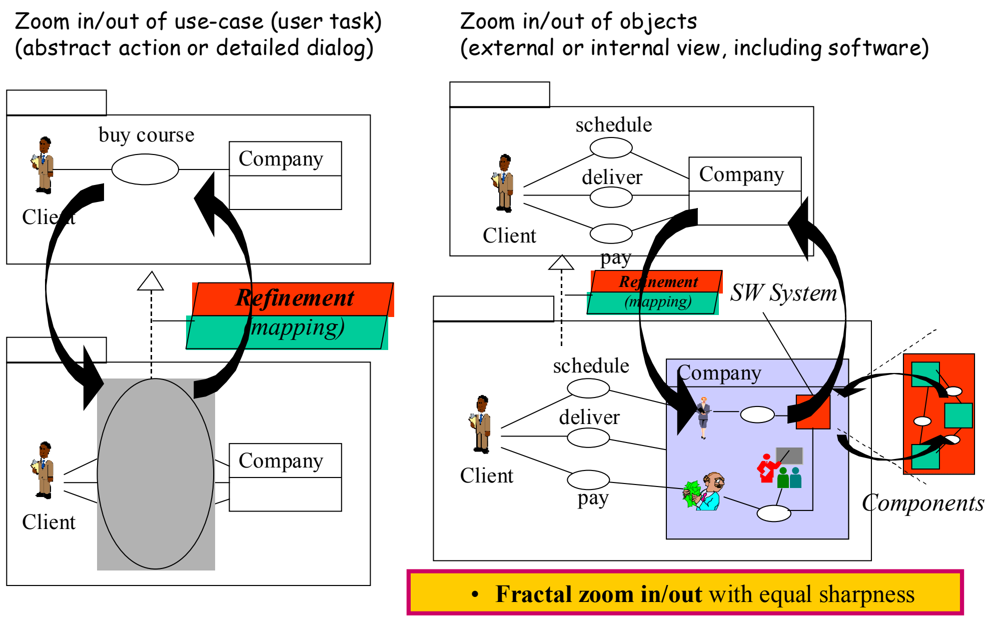
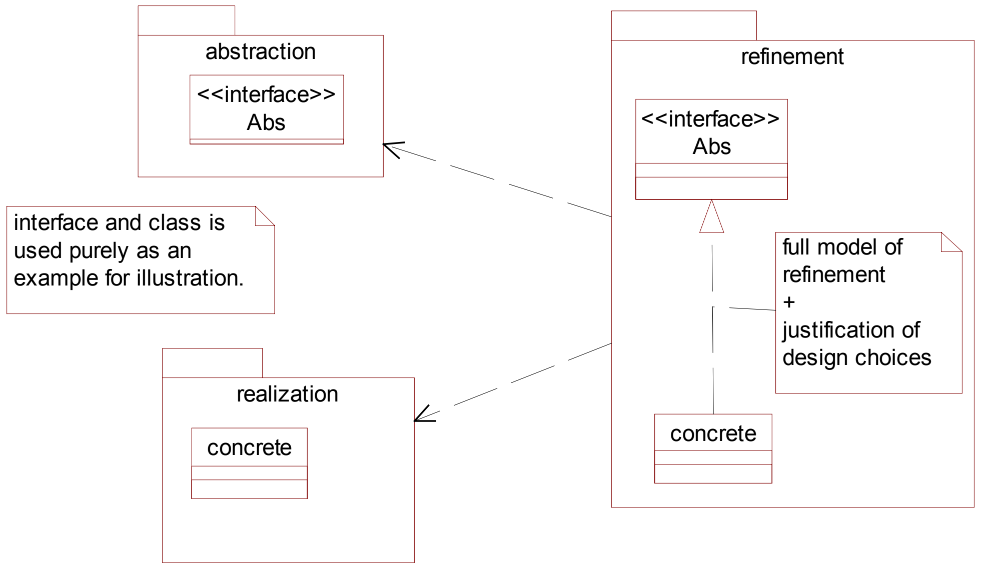


Figure 4 Illustration of refinement relations

Doing refinement properly will unify a core concept behind many disconnected constructs:

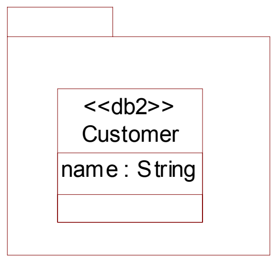
- System, subsystem, model – system and subsystem are just objects, possibly at different levels of abstraction.
- Use case, action, activity, … these are all different abstractions of behavior or interactions.
- Interface, type, and class – there is again a realization / abstraction relation between these.

The general structure of the refinement relation is shown below. The note marked "full model of refinement" could be a complete model, including classes, attributes, associations, interactions, state charts, activity diagrams, … that establish that the concrete faithfully meets the guarantees of the abstract. The kinds of models most that are useful depend on the kind of refinement. The rigor is variable: the refinement model could stubbed out with just “*Because Joe says it will work*”.


abstraction
<<interface>>
Abs
refinement
<<interface>>
Abs
interface and class is used purely as an example for illustration.
full model of refinement + justification of design choices
realization
concrete
concrete

8 Meta-Model Architecture

8.1 Structure of Meta-Models



Given a package from your favorite tool, does the familiar box labeled Customer represent a UML class? A MOF class? Does the <<db2>> on the box mean IBM’s definition of the <<db2>> stereotype? Someone else’s <<db2>>? Does the <<…>> even mean UML stereotype?

It is nice to treat a package as a collection of model elements, provided every language construct used is unambiguously identified with a definition of that construct.

K-2. A language or meta-model is defined in a package that can include the rules of concrete syntax, corresponding abstract syntax, and semantics of each language construct. That package can use all the usual package structuring facilities. A model – i.e. an instance of that meta-model – must import the language definition package so every language construct used in the model is identified with its definition in the meta-model.

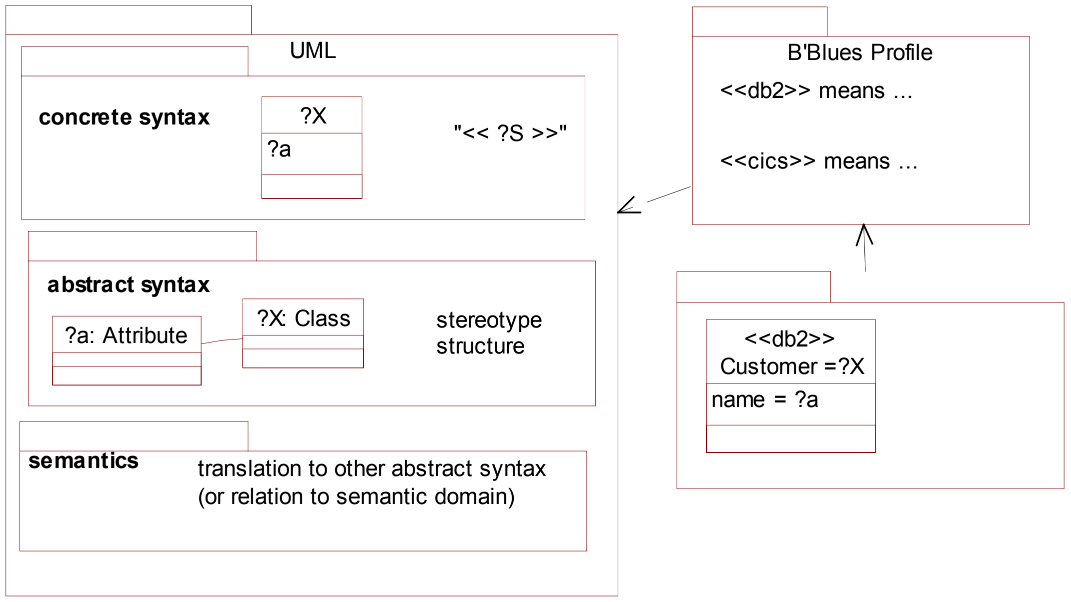
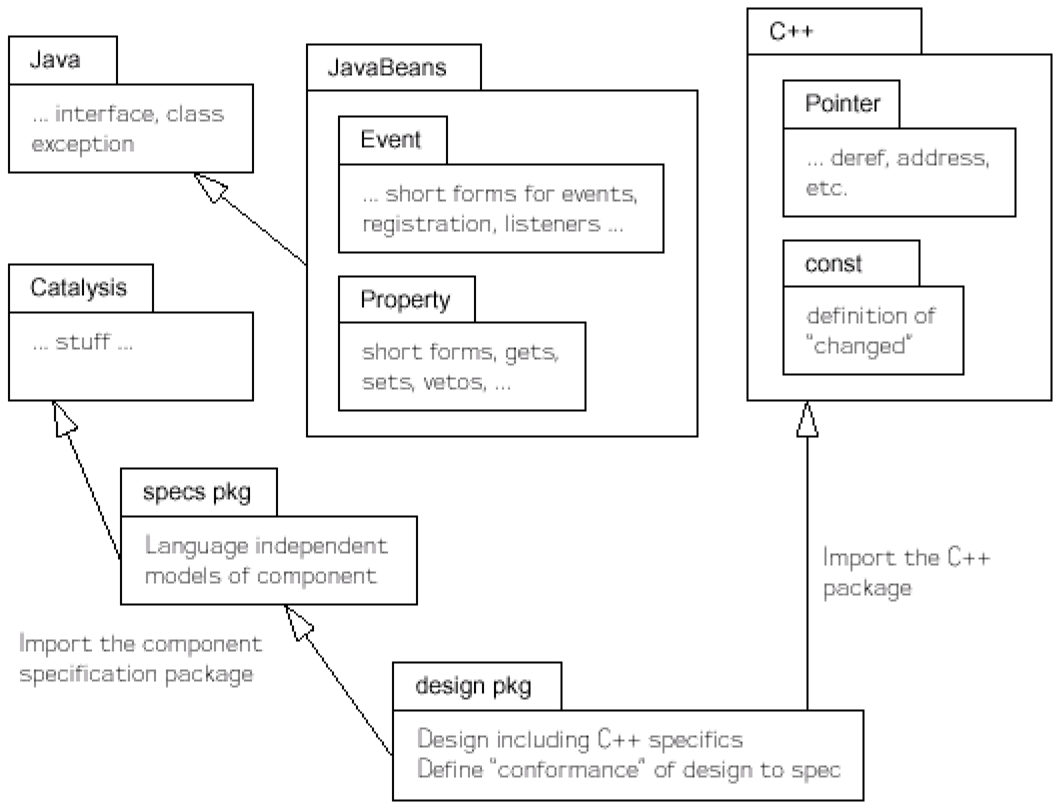


Figure 5. Language Definition and Usage

Figure 5 sketches how UML concrete syntax (a class box, or a stereotype), its mapping to abstract syntax (a class and its attribute, or the inheritance structure for a stereotype), its semantics (semantics can be defined by translating, in the form of “patterns”, a higher-level construct into a lower-level constructs, provided the lower level one has its semantics defined in some form), and particular profiles with their stereotypes, are all used by a model. The basic idea is quite similar to programming. When you write a source file in C++, you must (a) indicate clearly that it is C++, and sometimes even indicate which version of the language or compiler can handle it ☺; and (b) #include any header files that define macros that you use. Facilities for packages and I mports leads to a structure of both incremental language definitions (meta-models) and language uses (models), as shown in the figure below.


Java
... interface, class exception
JavaBeans
Event
... short forms for events, registration, listeners ...
Property
short forms, gets, sets, vetos, ...
C++
Pointer
... deref, address, etc.
const
definition of "changed"
Catalysis
... stuff ...
specs pkg
Language independent models of component
Import the C++ package
Import the component specification package
design pkg
Design including C++ specifics
Define "conformance" of design to spec

9 Interoperability, Model Interchange, and Translation

Interoperability should not be reduced to a least-common-denominator. Instead, it should allow a model expressed in a given modeling language or dialect to be at least partially understood by a machine that did not understand that entire language or dialect. For example, a design-time search for components that meet a certain real-time spec, in a tool that did not fully understand the real-time profile, should still retrieve candidates that appear to meet all other search criteria. Similarly, a run-time agent that understands attributes but not state machines, when faced with a state-machine model, should be able to reason with the attribute-equivalents of those states. This requires that language definitions themselves be structured to share partial language definitions and extend or translate into other languages. So we could:

- Introduce state constructs and an access API for those that understood states
- Translate into attributes, offering access through the 'attribute' API for less evolved clients
- Include a generic reflective access, of course

10 Fractal Foundation

- Object is any granularity
    - Hence multi-threaded, multiple interfaces, etc.

    - Interface relationships within object
    - Object refinement
- Action is any granularity
    - Concurrency across actions
    - Long-lived actions
    - Action refinement

11 Composition Foundation

- Components x, y; do not have to be all of the same type (so can define use-case, record-type, …)
    - ports xp1, xp2, yp1, … abstract the internals of x, y
- Connectors C1, C2
    - Define the meaning of compositions
- Assembly X
    - Can itself be a different type
    - Can have an independent external specification;
        - in which case multiple compositions as alternative refinements