First-Class Extensibility for UML —
Packaging of Profiles, Stereotypes, Patterns

Desmond D'Souza, Aamod Sane, Alan Birchenough
{ desmond.dsouza | aamod.sane | alan.birchenough } @platinum.com

Abstract. We discuss a first-class extensibility mechanism for the UML based on Catalysis packages and frameworks [3]. Packages define and structure meta-model extensions for different modeling language "profiles". Package frameworks support lightweight extensions like stereotypes as well as heavyweight extensions. OCL can be used to define constraints and rules for profiles and frameworks. Our approach rationalizes and consolidates some core concepts within the UML standard, uses a simple general mechanism for layering facilities onto that core in a precise and well-defined way, and offers a way to simplify and re-factor the UML specification.

1 Introduction

The UML has become a family of modeling languages, each with its own specialized stereotypes and other extensions. This is understandable, since rule-based systems, compilers, signal-processors, control-systems, etc. all want the end-user modeling language to simplify the expression of their own problems and solutions. The OMG "green" paper document ad/99-04-07 titled "White Paper on the Profile Mechanism" [1], proposes a mechanism named "Profile" to add to UML 1.3 to structure UML extensions and provide:

- Specialization, visibility, and selection of meta-models
- Context for stereotypes, tagged values, and constraints
- Heavy-weight extensions to the meta-model
- Miscellaneous extensions, such as rules and notations.

In this paper we show how to consolidate current UML facilities to improve model structuring and management, satisfying the requirements for meta-model "profiles" while providing first-class extensibility, with a simple application of:

- the existing UML package and package-import/generalization mechanism
- frameworks - to describe recurring patterns in models and meta-models
- OCL - to express constraints on the meta-model

The argument is quite simple:

- Use packages to define and structure both the meta-model and its extensions: this is how the meta-model is already structured. Packages and import (or its variant, package *generalization*) provide all that is required to control granularity of element grouping and visibility; profiles are just packages.

- Utilize package imports (or package *generalization*) so an importing package can extend the definition of an imported element: there are good reasons why different aspects of a meta-element should be defined in different packages e.g. Profiles. UML 1.3 ostensibly supports this for package generalization.
- Define the "join" rules by which such extended definitions are combined: if a given meta-element is extended in two different profile packages, then we need clear rules for combining these two extensions.
- Use meta-model level frameworks to express common structural patterns in models: For example, a framework can express structural patterns like JavaBeans, lightweight extensions like stereotypes, relationships such as "instantiation" and its implied constraints on the graphs of model elements, as well as heavyweight meta-model extensions.
- Use OCL at the meta-model level to describe constraints: For instance, to describe a constraint that the "Java" profile does not allow multiple inheritance, add OCL expressions to the Generalization meta-model element to forbid multiple inheritance.

2 Why use UML Packages as the Base?

All UML modeling is done in some package. Package import (via generalization) define visibility of elements, so elements from another package are only visible if that package is transitively imported (or at least transitively used via package generalization relationship). The granularity of a package determines the granularity of visibility control. The UML meta-model is already defined in packages, so we already have a mechanism for controlled granularity and visibility of meta-models. If the granularity is not adequate it simply means that the UML meta-model should be re-factored into smaller packages.

3 Packages for Structured Meta-Model Extension

A profile allows an element of the UML meta-model to be selected and specialized. Figure 1 shows how a *Real Time* profile may add new properties to the model element named *Activation*, such as an attribute *duration*; and a *Load Balancing* profile may associate a *Node* with an activation. These profiles define extensions to the same meta-element, *Activation*, rather than defining sub-classes. In particular, if you work with a *Load Balancing* + *Real-Time* profile, every Activation would have both a *duration* as well as a *Node*.

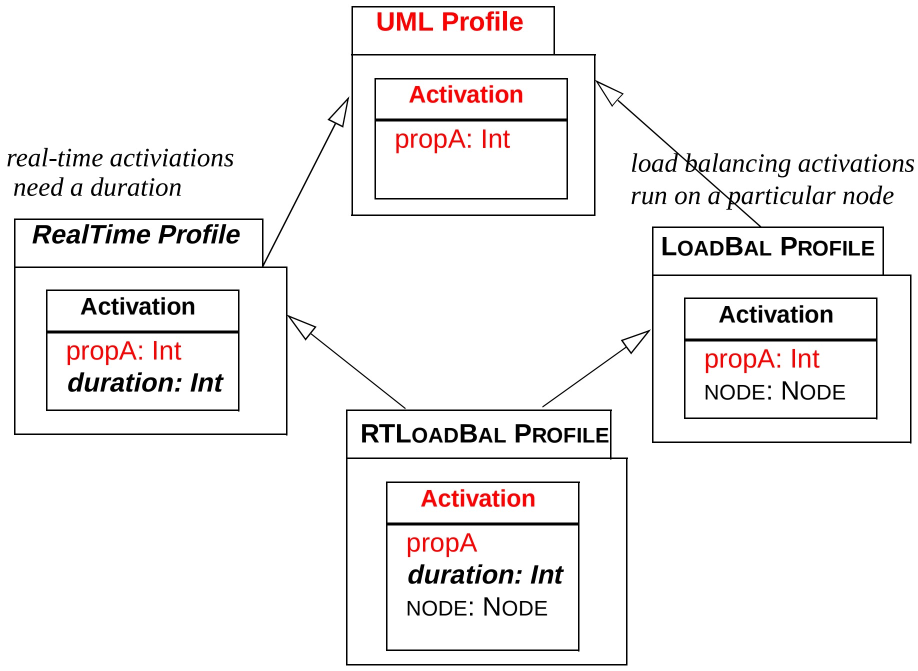

Figure 1: Package extension to define multiple "profiles"

A given (meta) model element — *Activation* in this case — may be introduced in one package, and then extended in other packages that have a generalization relation to it. As an analogy, note that *class* extension lets you *extend* a method (the mechanics might require "calling" the superclass method, or may offer an automatic mechanism such as before:after: declarations); common sub-typing specification practices let you *extend* an operation spec by adding more pre/post conditions. We want, at the package level, to *extend* any model element: same class with new attributes, operations, or invariants; same operation with new pre/post specs; etc.

When this feature is used, there is a difference between these two queries:

(a) "What are the attributes of type Activation?"

and:

(b) "What are the attributes of Activation <u>in package RealTime</u> (or LoadBal)?".

Naturally, (b) would return { *propA*, *duration* } for RealTime; and { *propA*, *NODE* } for LoadBal. We might choose for (a) to return { *propA* }; or it could be an undefined query. Thus, for a user, the result would depend upon which profile package was visible to him at that time.

3.1 UML 1.3 Compatibility

This approach is compatible with UML 1.3, and has no impact unless used. For example, UML 1.3 (Section 2.14.4) says:

> *"A package can have generalizations to other packages. This means that the public and protected elements owned or referenced by a package are also available to its heirs,* ***and can be used in the same way as any element owned*** *or referenced by the heirs themselves. This visibility is transitive."*

If the element can be used in the same way as any owned element, then the extending package can define additional properties e.g. new attributes, operations, associations, new specifications for existing operations, etc. This would suggest that the properties of any model element (e.g. the attributes of a type, the supertypes of a type, the operations of a type, the specification of an operation, the transitions of a state, etc.) could all be extended in another package exactly as though they were owned by that other package itself. If this interpretation is correct, it already provides the extension facility we recommend.

UML 1.3 (Section 2.14.3) also says:

> *"A Package may only own or reference Packages, Classifiers, Associations,* ***Generalizations****, Dependencies, Constraints, Collaborations, Signals, and Stereotypes."*

If a package owns generalizations, then package P3 can import package P2 and own a generalization between classes P2::A and P2::B; hence some features become associated with class P2::A only in package P3, by inheritance from P2::B via this generalization. This implies the UML must already deal with package-scoped properties of a class. UML 1.3 also already permits the same model element to be defined incrementally (with overlaps) across multiple diagrams in a given package, and so already has to compose incremental definitions from multiple places.

3.2 Contrast with Subclassing Approach

UML recommends subclassing as the default mechanism for extensibility. Contrast these two ways of accomplishing this in Figure 2: the one on the left is based on our

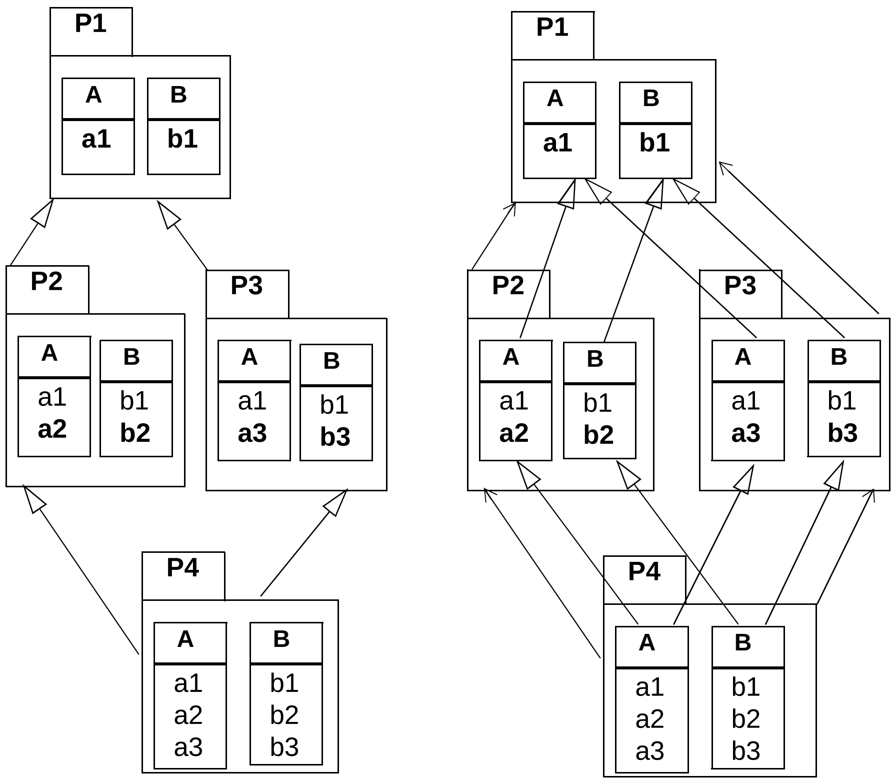

Figure 2: Package extension vs. Subclassing approaches

approach, where each package describes a separate perspective on the (meta) classes; the one on the right uses subclasses with explicit multiple inheritance, and becomes unmaintainable very quickly. In fact, the subclassing approach becomes far worse if you consider extensions not just of (meta) classes, but of attributes, associations, operations, etc. due to the combinatorial explosion of artificial sub-classes and explicit multiple-inheritance. The example in Figure 1 would not work correctly if two separate subclasses were created for the RealTime and Load Balancing properties, unless you used explicit multiple-inheritance. Moreover, our approach supports an explicit subclass if needed.

3.3 But <my favorite language> Works Differently

Let's start off by addressing a typical question:

> *"C++, Java, IDL don't let you define a class/interface in more than 1 place; why do this with UML?".*

Implementation languages impose different limitations on physical structure of code. Source-file based systems might require a complete class implementation to be in a contiguous section of a single file; modern-day IDEs offer direct manipulation of distinct program entities such as data members, methods signatures, and method bodies, with no underlying flat-file structure. Models and specifications, by their very nature, separate out different requirements or aspects of the same implementation unit. e.g. *"all remote calls must be synchronous*" and *"database calls must be remote"* and *"the place_order method must update the database"* are three distinct specification fragments that all influence the *place_order* method body. While certain UML packages might correspond to implementation units like files, namespaces, or modules, it is too restrictive to equate every model package with a particular language unit since you then lose much of the flexibility of factoring and re-using models and specifications.

One common question about Figure 1 is: *"What happens if I have an implementation of RealTime::Activation? Can someone else working with the model in the RTLOADBAL package expect to use that implementation?"* The answer comes in several parts:

- Firstly, package extension does not permit removing any statement about an imported model element i.e. there are no "overrides" at the specification level. Package extension facility is first and foremost a way to structure models, not implementations. This form of structuring has been used for a long time in formal methods [4]. When defining the Integer type you will never state all the properties of Integers in one "package" — just the basic ones like +, -, etc. that you believe are always needed. In some other package you may define additional properties of integers, such as *statistics* or the *fibonacci* function. Even Internet standards like XML [5] allow different properties of the same object to be defined in different documents.
- Secondly, when it comes to implementation, traditional implementation approaches require that you provide *in a single place* all the implementation code for a class. However, even here there are many approaches being used to defer this restriction until absolutely necessary.
    - ENVY/Developer — an environment for managing configurations of Smalltalk code — defines its equivalent of a "package" to contain some subset of the methods and instance variables from multiple classes; a different "package" may define additional methods for those same classes; or newer versions of those same methods that should be loaded as a patch to replace existing ones. Envy recognizes very clearly that implementing some subset of the behavior of some (group of) classes, or patching some subset of their behaviors, is a semantically very different operation than creating new subclasses.
    - *Aspect-oriented programming* is based on the assumption that you want to implement different *aspects* of a class (or a method, a collection of classes, an object, etc.) in different places to improve separate evolution and maintainability of those aspects. There is a stage where these different aspects are

    "woven" together into the combined implementation — in our terms, a package where the different implementations aspects are "joined".

  - The usual (IDL and C/C++) facility of *#include* is one implementation counterpart of package structure and imports. It is actually a very simple *"join"* based on textual concatenation. There is no reason to restrict ourselves to this, except at those selected package boundaries which must necessarily correspond to language boundaries (e.g. IDL modules).

- Thirdly, if you do have an implementation of *RealTime::Activation* and a client wanted to use *RTLoadBal::Activation* with that existing implementation, then you might use a delegation / wrapper around that implementation. Remember that the full name for any type is *PackageName::TypeName*.

3.4 Define "Join" Rules to Combine Element Definitions

Since 2 profiles may each "say more" about the same meta-model element, *X*, and we need to be able to combine profiles, when combined, you want meta-model element *X* to be automatically "joined". These join rules must be defined carefully. For example, when we are joining the two definitions of *Activation* into the RTLOADBAL package, the rule may include:

Join two type specs, the resulting attribute set is the union of the two attribute sets.

Join two operation specs, the resulting spec is pre1 & pre2 with post1 & post2.

Note the following points regarding "join":

- The stereotype / sub-classing approach will not support this automatic "join" without combinatorial multiple inheritance; our "extension" approach will automatically perform the "join" (see Figure 2).
- The UML approach to multiple diagrams already requires "join". UML 1.3 permits a given model element to be defined across multiple appearances on multiple diagrams; hence the UML 1.3 already needs to combine these appearances (the 1.3 spec unfortunately omits these rules). We would utilize this behavior for definitions that were introduced across more than 1 package.
- A "join" facility is already needed for UML patterns, supported notationally in UML 1.3. Since several patterns can be applied to a given model element, substituting for types, attributes, etc, you end up with parts of multiple patterns defining a type, attribute, operation etc. These must be "joined" in the resulting model element.
- It is not possible to insist on global consistency across all profiles ever defined, or to require that every pair of extensions be disjoint. Hence it is possible that the "joined" versions across two profiles will result in an inconsistent meta-model. There are many different strategies to dealing with this situations, and we will not explore them here.

3.5 What can this be used for?

This facility can be used in many ways (subject to “join” rules, in Section 3.4).

- Tags: Separate tags into an importing package so basic model elements remain independent of those tags.
- Notations: Separate the notational aspects (e.g. stereotypes, specialized graphical or textual syntax) from the core model constructs they represent.
- Views: Support end-user structuring of models into separate views or perspectives, without having to create combinatorial (and often semantically confusing) subclasses and multiple-inheritance.
- OCL extension: those using OCL today quickly reach its limits in readability. For example, if you only have a fixed set of operations on *Sequence* it becomes very awkward to define something like *shortestInitialSubsequence* repeatedly in terms of the predefined set. However, in our approach all of the OCL is defined in a package (using packages as frameworks we routinely define generic types such as *Seq(X)* and instantiate them with framework application — Section 4). You simply define your own *OCL Extension* package and extend the definition of *Sequence*; and you can continue to re-use packages that were developed in terms of the original OCL package. Subclassing would not work correctly here either: since there are existing packages which may use the original *Sequence* type, creating a separate subclass *MySequence* will not help with using those existing packages.

4 Frameworks — Light and Heavyweight Extensions

Frameworks are packages that capture recurring patterns. We suggest that they should be used at modeling as well as meta-modeling levels. The following sections show how frameworks help realize JavaBeans in a Java profile, stereotypes, and meta-level patterns like instantiation.

4.1 JavaBeans for a Java Profile

JavaBean models are based on properties, methods, and event. For example, a Stock object might have a *price* property; this means that the Stock class should have two methods, *getPrice*() and *setPrice*(), and an instance variable, *price*. In the JavaBean profile, you don’t want to work at the level of individual getters and setters, but directly use the concept of property. This can be captured as the *Property* framework in Figure

3, in which we have used the UML pattern notation to show framework application.

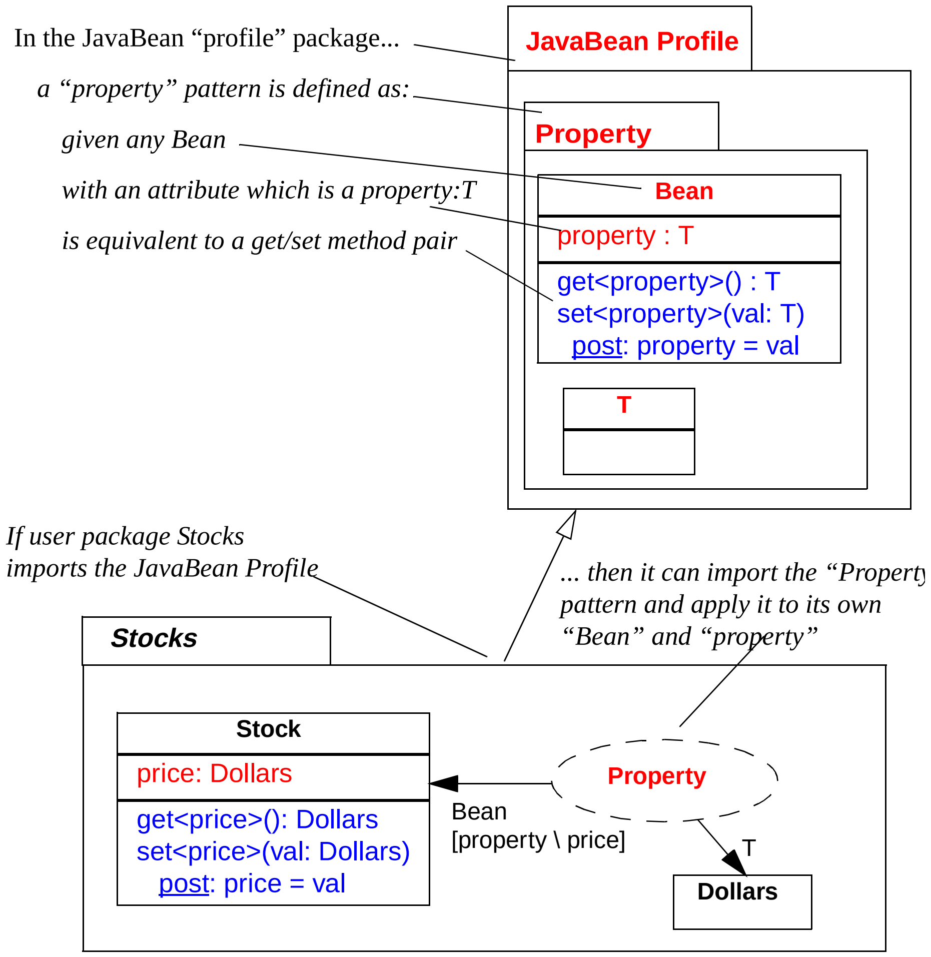

Figure 3: JavaBean profile can be defined with packages and frameworks

You import the *Property* subpackage of the *JavaBean* package (visible because you have imported *JavaBean*), and substitute Stock for Bean, Dollar for T, and Stock::price for Bean::property. The methods *get<price>* and *set<price>*, with their corresponding OCL specifications, are generated as a result of framework application. Note that the JavaBean Profile package provides the context for the Property pattern; the same approach can be used to define a profile for the Corba Component model, containing patterns such as *Ports*, *Facets*, *Receptacles*, and *Events*.

4.2 Stereotypes as Framework Application

A simple, novel, and semantically-rich way to define stereotypes is as a direct shorthand for framework application. Frameworks can simulate the default UML mechanism of implicit meta-model subclasses, or patterns on existing meta-model classes.

In UML, a stereotype is a meta-model element that is a subclass of a standard meta-model element. When you apply a stereotype <<property>> to an Attribute, that Attribute Instance is an instance of the *Property* subclass of Attribute. Frameworks already provide the capability to define a stereotype, and framework application lets us use a stereotype. We can make the *Property* stereotype definition in the traditional UML form explicit in a framework, as in Figure 4. Note that the JavaBean profile, imported by *Stocks*, provides the context for the definition of the Property stereotype.

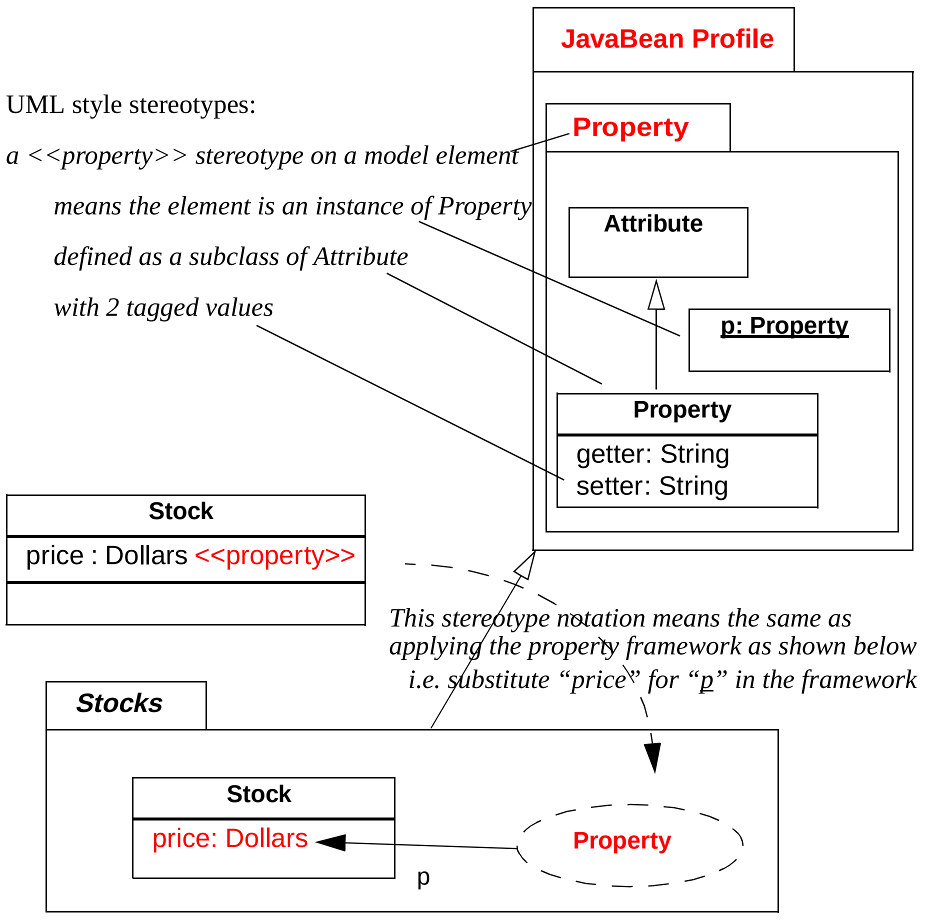
JavaBean Profile
UML style stereotypes:
a <<property>> stereotype on a model element
means the element is an instance of Property
defined as a subclass of Attribute
with 2 tagged values
Property
Attribute
p: Property
Property
getter: String
setter: String
Stock
price : Dollars <<property>>
This stereotype notation means the same as
applying the property framework as shown below
i.e. substitute "price" for "p" in the framework
Stocks
Stock
price: Dollars
p
Property

Figure 4: Frameworks can realize traditional UML stereotypes

Rather than implicit sub-classes and tagged-values, we can apply stereotypes as we would apply any modeling pattern: the stereotype serves as a short syntax for an expanded form of that pattern, and additionally defines its semantics as a translation. For example, suppose we define *Stock* with the stereotype *property* as in Figure 5. We can regard this as syntactic sugar for the framework application in Figure 3. This is our preferred means to give semantics to stereotype definition and stereotype application, as framework application is grounded in very clear semantics. Still, frameworks can also support the subclassing style.

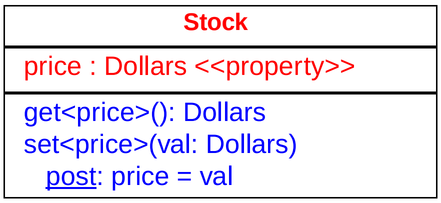

Figure 5: Stereotype as syntax for pattern application

4.3 Framework for Heavyweight Extensions: Instantiation

According to UML 1.3, heavyweight extensions explicitly define new meta-classes. The example in Figure 3 already shows the capability to support heavy-weight extension. An even richer example is described in the following. Consider making the idea of "instantiation" explicit in the meta-model, and even the meta-meta-model: some things (descriptors) declaring the kinds of properties that other things (occurrences) can possess. We can describe this in the *Instantiation* framework, capturing a (simplified) constraint on the graphs of the descriptors and their corresponding occurrences. This framework is used first to define the UML meta-type *Type*, its relationship to corresponding *TypeOccurrences* (such as *Person*); and then to define the instantiation relation between *Person* and *p1*.

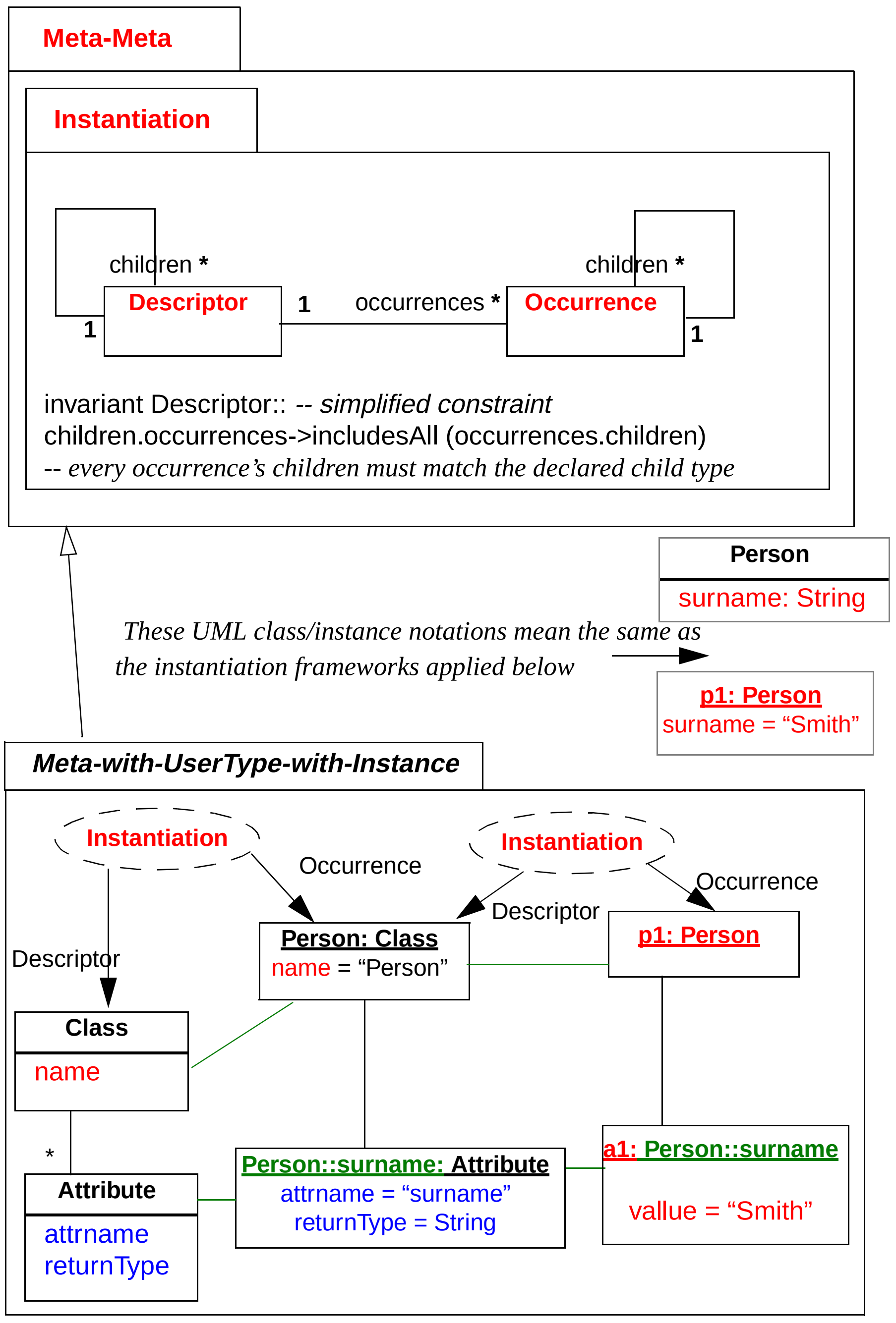

Figure 6: Some frameworks apply across user, meta, and meta-meta levels

4.4 Constraints on Framework Application

A framework can also include design-time "pre-conditions" that must be satisfied by the elements that are substituted in any framework application. For example, a JavaBeans *Property-Property* connector could be defined as an abstract way to connect two beans together keeping their properties continuously synchronized, but with the design-time condition that the corresponding attributes already have some specific *Property* frameworks applied to them e.g. one of them is writable, the other is readable and raises an event when it changes.

5 Other Extensibility Mechanisms

We argue that the remaining semantic extensibility mechanisms do not add fundamental new needs either.

- *Constraints*: We already showed an example (Figure 3) of using OCL in a framework application. At the meta-level, the profiles paper [1] describes a constraint that the Java profile does not allow multiple inheritance. For this, we can use OCL at the meta level. The Java profile imports the basic meta model and adds an invariant to the Generalization element, for instance "*generalizations->size = 1*".
- *Tagged values*: With frameworks, and the ability to substitute attributes and generate other model elements, tagged values are redundant. If desired they would be untyped string-value pairs in a framework (Figure 4).
- *Rules*: OCL can be used to specify validation rules, and a functional-language subset of OCL can be used to express transformation rules.
- *Notation and Presentation*: Notations and presentation rules can also be defined as frameworks if framework elements are allowed to range over sets of presentation or syntactic elements. In general, extensions should include the concrete syntax that defines each newly introduced modeling construct. However, some new machinery may still be required to deal with this style of incremental syntax definition and syntactic ambiguities.

6 Proof of Concept

The mechanisms described in this paper are well understood and used in the Catalysis approach to using the UML [3] and implemented successfully in modeling tool prototypes. In Catalysis, all modeling and meta-modeling notions, extensions, and specializations, are defined in a structure of packages and frameworks. For example, see Section 9.8.2, Section 9.9.3, or Section 9.9.4 of the Catalysis book.

7 Conclusions

We have shown how packages and frameworks can provide a first-class extensibility mechanism for UML, and to structure UML extensions into “profiles”. The UML 1.3 metamodel is already quite large and is known to contain some inconsistencies. Rather than add yet more facilities to it for extensibility or profiles, our approach is to rationalize and consolidate some core concepts within the standard, and then use a simple general mechanism for layering facilities onto that core in a precise and well-defined way. In fact, the UML itself could be simplified considerably if re-factored, and our approach makes it possible to re-factor the UML to a cleaner specification, while retaining the original semantics of the UML constructs where needed.

References

[1] “White Paper on the Profile mechanism”, Version 1.0, OMG Document ad/99-04-07. Also available at http://uml.shl.com/u2wg/default.htm

[2] “OOAD and Corba/IDL - A Common Base”, D. D’Souza and A. Wills, http://www.iconcomp.com/papers/omg-ooa-d/OMG-OOA-D-rfi.frm.html. This paper outlined a semantic scheme for extensibility and a layered meta-model to accomplish the goals that the Profiles paper raises.

[3] “Objects, Components, and Frameworks with UML - the Catalysis Approach”, D. D’Souza and A. Wills, Addison-Wesley, 1998.

[4] “Larch — Languages and Tools for Formal Specification”, http://www-theory.dcs.st-and.ac.uk/~mnd/larch.html