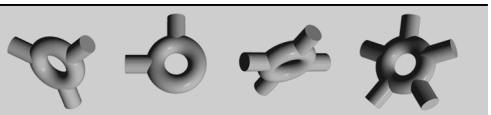
elements 

Modelling Standards

Doug Palmer[1]
TARMS Inc.

Danny Cron[2]
TARMS Inc.

September 8, 2000

Typeset in LaTeX.

[1] doug@tarms.com
[2] danny@tarms.com

Contents

1 Introduction **1**

2 Packages **1**

3 Use Cases **2**

4 Stereotypes **2**
4.1 Interfaces and Classes 4

5 Class Diagrams **5**

6 Dependencies **6**

7 Collection Classes **7**

8 Attributes **7**

9 Associations **7**
9.1 Compositions 8
9.2 Aggregations 8
9.3 Specialisations of Associations 8

10 Reference Data **9**

11 Components **10**

1 Introduction

UML[1] has been chosen as the object modelling language for the elements model. UML allows considerable latitude in terms of modelling approach. In order to ensure consistency, some standards need to be imposed.

This document outlines the modelling conventions used in the elements object model. As the model is developed, the need for new conventions will be exposed; this document, therefore, will change during the evolution of the project.

2 Packages

The elements object model is divided into a series of packages, which can be merged together to make a single, complete model. Separating the model into packages serves several purposes:

- It enables distinct areas of functionality to be isolated, and presented individually. This partioning of the logical elements of the system is expected to

facilitate a stronger and more expedient understanding of the specific area of the model of interest to the reader.

- Not all users of elements want to use every financial instrument and feature on offer. By selecting packages and merging the packages together, a model smaller than the complete model can be built.
- It provides greater control over the development of extensions to elements.
- The packages provide the basis for software packages or parcels in those languages that support such things.

The main part of a package is located in a uniquely named sub-package beneath the *Logical View* package in the Rose model. This package contains all the classes, use cases, class diagrams, etc. for the part being modelled. Any classes imported from other packages are placed in appropriately named sub-packages; these classes are *stubified* — stripped of documentation, attributes, associations and operations.

Extensions to existing classes in existing packages are made by adding new documentation, attributes, associations and operations to the stubified classes. These extensions are treated as special cases within the documentation generator and grouped with the main package. When several packages are merged, the extensions are added to the appropriate classes in the proper packages.

3 Use Cases

Use cases and diagrams are currently placed in the main package under the *Logical View* category. This is arguably wrong, but convenient.

Use cases can be used to describe examples of the kind of business elements being modelled. Example use cases should be grouped into use case diagrams that have “Example” as part of the diagram name.

4 Stereotypes

The following stereotypes — extensions of the vocabulary of the UML, describing further building blocks targeted at the model — are used.

Interface The standard UML stereotype used to specify a class that is fundamentally just a collection of behaviours, operation signatures and definitions, rather than classes which contain implementation details. See section 4.1 for more discussion about the relationship between classes and interfaces.

Interfaces are usually named for the behaviour or type of object that they specify, eg., Date, Holiday, RateConstructor.

Exception A class that is used to carry exception information. Exceptions are events that interrupt the normal flow of control, usually because some piece of data is missing. At this point, it is more convenient to deal with this atypical situation elsewhere rather than to interrupt the typical flow of execution. This is achieved by throwing an exception.

A thrown exception is caught by an appropriate exception handler, usually keyed to the exception class. The exception handler is then responsible for dealing with the exceptional condition and returning affairs to a more typical state.

Exceptions are *normal* occurrences, not events indicating some catastrophic error. As an example, an exception may be thrown by a position analytic calculating routine, indicating that the supplied data is insufficient and that the analytic cannot be calculated, at present. Clearly, this exception should not cause the process to halt. However, throwing an exception is, very often, cleaner than returning a series of "error" return values, that have to be specially handled by each calling operation.

Classes which are exceptions have "Exception" as the last part of the class name, eg., NotFoundException.

Service Interface A service, in this case, is not a UML component, as such, but an object that is part of the larger system than the object model. Services are not (necessarily) fully specified within the object model, but represent pluggable software objects that can be used to realise the function of a system that uses the object model.

A service interface is an interface specification for some service.[3]

An example service is a rate curve constructor. The exact process by which rate curves are constructed are a subject for the implementor of a system using elements, rather than elementsproper. However, elements objects must be able to interact with curve constructors so that curves, which are strictly part of elements, can be built.

Class Classes (classes with no stereotype) represent implementations of interfaces. See section 4.1 for a discussion of the relationship between classes and interfaces.

Classes always have "Model" as the last part of the class name. If the class directly implements an interface, then it usually has the same name as the interface, suffixed with "Model", eg., DateModel or HolidayModel.

[3] Note that code generation for Java often depends on the *Interface* stereotype being a special case. Service interfaces are also interfaces, although the code generator may have some difficulty with this fact. It is the responsibility of whatever preparation scripts are used on the model to prepare it for code generation in a particular language to rename service interfaces (or any other interfaces) appropriately.

Service A service is a class that implements a service interface.

Services are included in elements wherever there is a sufficiently standard methodology for a system (eg. a yield to maturity analytic) or where an example is useful.

Architectural Service Interface A service interface which specifies the behaviour of some architectural component of a complete system. The elements model is not directly concerned with system architecture. However, there are some points where contact needs to be made and a minimal interface specified.

Architectural Service A service model that implements an architectural service interface.

Static Method Static methods are operations that apply to the class, rather than an instance of that class. In Smalltalk[5] terms, static methods are class methods.

4.1 Interfaces and Classes

First some terminology, mostly derived from Java:[2]

- A *type* is a group of related method signatures. For example, a Collection type may consist of the `add` `remove` and `do` methods.
- An *interface* is a formal description a type.
- A *class* defines the representation and methods for an instance of that class.

Classes implement interfaces, by supplying the behaviour for the methods in an interface. The type of an interface is the type derived from the list of method signatures in the interface. Classes may have multiple types; as a special case, the type of a class is the type that can be derived from collecting all the method signatures of the class.

Both classes and interfaces allow inheritance. Classes may have multiple inheritance (eg. C++[6] and Eiffel[3]) or single inheritance (eg. Smalltalk and Java). Interfaces generally have multiple inheritance. Classes that implement interfaces do not need to follow the inheritance pattern of their associated interfaces.

In strongly typed languages, such as Java, C++ or Eiffel, the arguments in messages must be typed. If classes are used as types, then there may be difficulties when the behaviour of one class needs to be extended to encompass another class somewhere else on the class hierarchy. For example, it may be desirable to treat users and groups of users identically for certain operations. However, for implementation purposes, groups of users may need to be placed in a different branch of the class hierarchy (eg. it should be a subclass of groupable reference data).

Requiring all things which act like a user to be a subclass of user would distort the class structure.

The usual approach within languages which allow multiple inheritance, such as C++ and Eiffel, is to use the features of multiple inheritance to allow type-equivalence across hierarchies. Using the example above, an abstract user class would be abstracted, and this class would be a superclass of both the user and user group classes.

Smalltalk is not strongly typed. Any class which implements a method may respond to a suitable message. Any type errors are detected at run-time and raise an exception. Although Smalltalk provides no support for types, there is an informal model of types as collections of methods which naturally group together. Making the implicit types of Smalltalk explicit strengthens an understanding of any model.

To allow ready extensibility, explanatory power and ease of implementation across languages, the following conventions are used:

- Interfaces provide the primary method of behaviour description; all methods are grouped together into interfaces and described in terms of their abstract behaviour.

- Classes only implement interfaces; from the point of view of the object model, classes have no methods that are not part of an interface (during implementation, private utility methods are likely). All method signatures and attributes (ie. argument types, return types, etc.) are defined purely in terms of interfaces, rather than classes.

Interfaces provide an abstract description of the object model's behaviour. The object model then needs to implement those interfaces which represent concrete objects as classes. A class hierarchy of classes, each class being a variation of implementation behaviours may be required (eg. a hierarchy of different rate quotation methods). Classes are single inheritance and need not follow the inheritance pattern of their associated interfaces; implementation inheritance and behaviour inheritance are two distinct concepts.

5 Class Diagrams

Class diagrams should contain a single related group of interfaces and classes. More than one class diagram should be used within a package to avoid over-complex diagrams.

Class diagrams should follow standard layout:

- The generalisation, inheritance and realisation hierarchy should run vertically. Base classes should, generally, be above subclasses. Associations should, where possible, run horizontally, with directional associations pointing to the right.

- Interfaces should appear *above* the concrete classes that realises the interface. This helps emphasise the "inheritance" nature of the interface. If a class both realises an interface and inherits from a subclass, then the "most important" relationship is the most vertical relationship on the class diagram.
- If a class has several subclasses, the inheritance arrows should be linked together to form a multi-tailed inheritance arrow, rather than making a separate inheritance arrow to the superclass.
- Classes that realise multiple interfaces should have a separate realisation arrow to each interface.
- Interfaces that inherit from multiple interfaces should have a separate inheritance arrow.
- Try not to cross lines. If lines are crossing, it may be that your diagram is too big.
- It would be nice to leave role names off directional associations in diagrams. Role names are needed by the model merger (see section 9), but clutter things up and confuse the issue. Sadly, there appears to be no way of controlling the view of an association.
- If a class is imported from another package or class diagram, then the attributes and operations of that class should be suppressed.
- If there are several independent relationships with classes imported from other packages or class diagrams, then clarity *may be* improved by having more than one copy of a class depicted on the diagram, rather than having a single copy with multiple arrows pointing to it.[4]

  > Break any of these rules sooner than say [draw] anything outright barbarous.[4]

Note that the Auto-Layout function in Rose does *not* follow these guidelines. In fact, the function tends to re-arrange the diagram in counter-intuitive manner — particularly with respect to the relationship between interfaces and classes — and it should be avoided.

6 Dependencies

Dependencies should be avoided until we can figure out a coherent use for them.

[4] This is a controversial point. One author (DP) prefers this approach, as he feels that it unclutters the diagram and emphasises the distinctness of the objects. The other author (DC) feels that this approach confuses classes and instances and that having all associations pointing to a single class improves understanding. Make up your own mind.

7 Collection Classes

Certain operations either return or take as parameters classes that have collection-like behaviour. These classes usually benefit from a parameterisation giving the type of collection element. Assumed collection classes are:

Collection<Type> A generic collection that can be iterated over.

OrderedCollection<Type> A collection that assumes that elements remain ordered in the order in which they were added and maintains the ordering of its elements.

Dictionary<KeyType, Type> A generic dictionary that allows the looking up of elements via some sort of comparable key. If no parameters are included, a mapping from anything Comparable onto any object is assumed.

8 Attributes

An *attribute* is an atomic object that is only referenced by the enclosing object. Atomic objects are those for which operations create new objects, as opposed to manipulating the internal structure of the object. Atomic objects are: Strings, Numbers, Dates and Timestamps, Symbols and Enumerations, Booleans, PrimitiveInstruments and the Null object. An example attribute is the name of a user.

The ValueSemantics interface, defined in the Utilities package, can be used to indicate that an object is a suitable candidate for being an attribute.

9 Associations

Associations are used to model the relationships between complex objects.

Generally, the cardinality of the ends of the association should be indicated; aggregates and compositions are assumed to be 1-many relationships and need not be so marked.

Associations may only leave non-interface classes, although directional associations should terminate, wherever possible, at interfaces. The reasoning here is that associations imply some sort of instance variable that holds the association; a foreign idea to our conception of interfaces.

Wherever possible, the direction of the association should be indicated. Very often, two-way associations can be modelled by using two directional associations running from the concrete class to the interface. Eg., if you have interfaces A and B and classes AModel and BModel, you may be tempted to run a two-way association between AModel and BModel; this can also be modelled as a directional association running from AModel to B and a directional association running from BModel to A. See figure 1 for a diagram of this example.[5]

[5] Note that it is almost impossible to elegantly avoid breaking the rule about crossing lines in

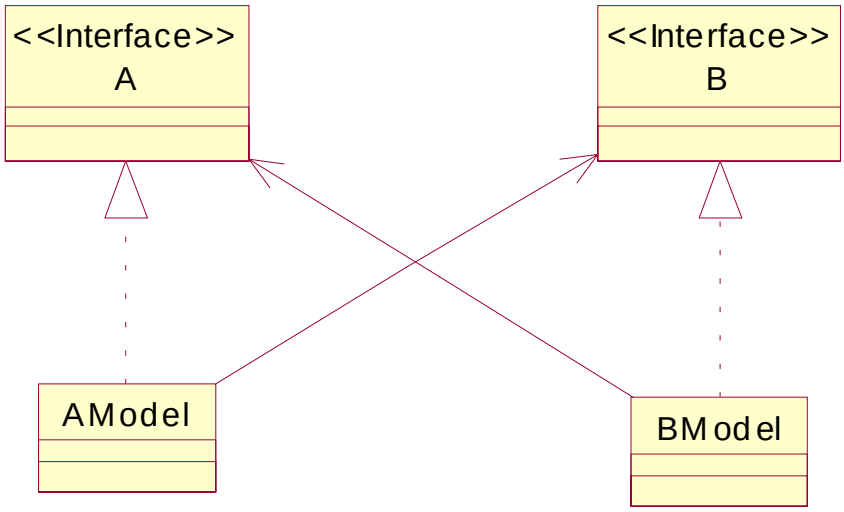


Figure 1: Example Two-Way Association

Associations should be named. Roles need only be named if there is some useful information to be imparted by naming the role; directional associations rarely need additional role names.

9.1 Compositions

A composition association is only used to indicate that the composite object is the only object which refers to the components. The composite object owns its components. An example of composition is that an instrument is composed of transactions; no other instrument uses those transactions.

9.2 Aggregations

An aggregation association is only used wherever the aggregate object's main purpose is to hold the aggregated objects. The aggregated objects are referred to by objects other than the aggregate object and the aggregate object does not own its components. An example aggregate object is a deal cache.

9.3 Specialisations of Associations

In some cases, it may be necessary to specialise an existing association when moving to a subclass. This is generally needed is when there are two related classes

diagrams when doing this.

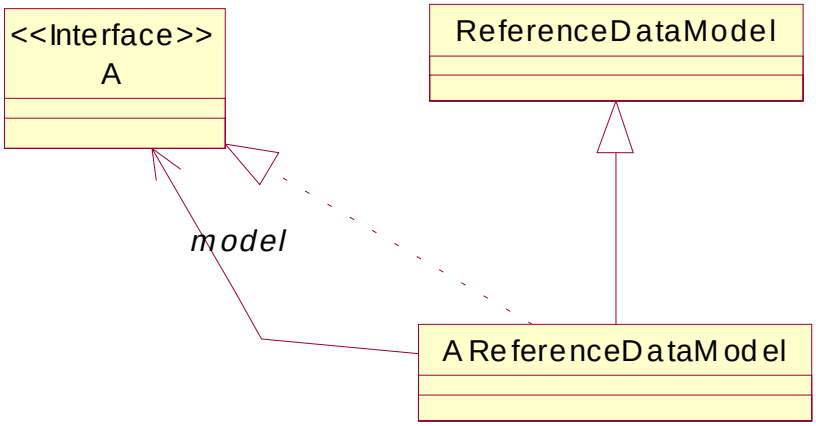


Figure 2: Example Reference Data Structure

and specialisations of the two classes. Generally, there will be an association between the related superclasses; the *same* association for the subclasses would, legitimately, want to specialise the association to just be between the subclasses.

At the moment, the suggested technique is to re-iterate the association with a note mentioning the specialisation. This is not exactly satisfactory, but it's the best that we can come up with.

10 Reference Data

The general approach to take when modelling reference data is to treat the model as something in its own right. A separate model, inheriting from the ReferenceDataModel class is then used to wrap this model and manage the object as reference data. The reference data class realises the same interface as the actual model and can be used in place of an unadorned model; the operations on the interface are simply implemented by passing through to the wrapped model.

As an example, a Language-type piece of reference data is implemented using three entities: the Language interface, the LanguageModel class and the LanguageReferenceDataModel class. The LanguageModel class realises the Language interface. LanguageReferenceDataModel inherits from ReferenceDataModel and also realises the Language interface by associating itself with an actual instance some object that implements the Language interface (usually LanguageModel) and delegating any Language operations to that instance. This process is shown in figure 2.

The advantage of the above approach is that it makes "roll-your-own" data or named reference data interchangeable, adding to flexibility and separating the fact that something is reference data from the behaviour that it is expected to imple-

ment.

The association between the reference data model and the wrapped model is, conventionally, called “model” with “reference data” and “model” roles.

The reference data model is usually named by adding “ReferenceDataModel” to the interface name.

11 Components

Classes and interfaces should *not* be assigned to any specific component, nor should the target language be any other language than Analysis. elements is supposed to be language-neutral and assigning things to components tends to introduce a certain amount of language-specific assumptions. Assigning a language, other than Analysis, to a class or component tends to introduce all kinds of oddities, as Rose tries to adapt to your target language. In particular, primitive types tend to be language-specific.

We are developing scripts that will automatically generate components, assign languages and adjust type declarations for a given target language. To use the scripts, a copy of the model is taken and converted to a language-specific copy for the purposes of code generation.

References

[1] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, 1999.

[2] *Javasoft Home Page*. http://www.javasoft.com.

[3] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

[4] George Orwell. *Politics and the English Language*, 1946. http://www.abattoir.com/ prime8/Orwell/patee.html.

[5] *Smalltalk Industry Council (STIC)*. http://stic.oti.com.

[6] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesely, second edition, 1991. (Corrections 1992).