DEVELOPING HIGH CLASS UML CLASS MODELS

Jeff Jacobs, jmjacobs@jeffreyjacobs.com

DISCLAIMER

The views presented here are those of the presenter and do not represent those of any other person, organization, author of UML books, standards group, internationally recognized or self proclaimed UML guru.

OVERVIEW

I've been using various forms of modeling notations and techniques since my college days, spanning two centuries at this point in my career. As a consultant, architect and instructor, I've worked with and reviewed countless models and diagrams, and, sadly, find that UML Class Models tend to rank among the lowest in terms of clarity, precision, correctness and completeness.

There are many reasons of this, but the primary reason is the lack of prescriptive discipline, quality standards or even a general theory of good practices by UML creators and practitioners. Class Models are a notation that can be used in as many ways as Visio. There are almost no underlying semantics; a Class can be almost anything from a code construct to a data entity to a requirement to an architectural construct. Sadly, in the hands of most UML modelers, there is constant confusion between *true* modeling and implementation. Class modelers also fall into *big toolbox* syndrome, which is the opposite of "I have a hammer, so everything looks like a nail" syndrome. The big toolbox syndrome leads to "UML has all of these tools; I must demonstrate my ability to use them all".

This paper and its corresponding presentation will provide a set of rules for developing high quality UML Class Models and diagrams. These rules are based on well known techniques used in Entity Relationship Diagramming/Modeling.

This paper deals only with UML Class Models used for *forward engineering*, e.g., the model is constructed as part of the development process prior to implementation and focuses on the *information* aspects of UML Class Modeling. Operations and attributes are beyond the scope of this program I do not consider reverse engineering of a code base into a UML Class Model to be *modeling*.

Please note that some of the rules recommended in this paper do not correspond to common practices in the UML modeling world.

WHY DO WE MODEL

Modeling serves several purposes:

- Modeling helps us understand the real world, or the problem domain. When performed well, *it is an iterative process of discovery and refinement!* Such an approach is critical to developing high quality models.
- A good model is a representation of "reality", either of something that exists or something we plan on creating.
- Modeling is critical to understanding and expressing requirements.
- Well constructed models can be used to generate high quality code.
- Well constructed models can be used to generate databases; yes, Virginia, UML Class notation can be used for logical data modeling!
- Modeling should be a fundamental tool when designing code.

WHAT IS QUALITY?

There are 5 basic measures of the quality of a UML Class Model:

- Understandable - The model should be understandable to *all* interested parties, including the non-technical people. This is primarily accomplished by the appropriate use of terminology.
- Unambiguous - The model should be unambiguous, that is, if a particular construct is unclear, then it should be make clear. There should never be any extended discussion about the *intent* of the model. The most common cause of this is failure to appropriately name Classes and Associations. Even worse is the common failure to name Associations; I've watched in horror as countless hours have been wasted by people arguing over what an unlabeled Association meant!!!
- Complete - The model should be complete, with all necessary Classes, Associations and attributes identified. In particular, there should be no many to many (M:M) Associations left in the model!!! The elimination of M:M Associations as described in this paper is probably the single most important contributor to high quality models.
- Correct - The model should be correct, that is all interested parties should be largely in agreement on the correctness.
- Appropriate level of abstraction - The model should be at an appropriate level of abstraction.

UML Class Basics

The fundamental constructs in UML Class Models are *Classes* and *Associations*, which are very similar to *entities* and *relationships* in ER modeling. A third construct, *Association Classes*, may also be considered a fundamental construct; however, I strongly recommend against the use of Association Classes.

Classes

A Class is drawn as a box and should have a name. A Class may be thought of as *a thing of significance about which information must be kept and maintained, and the behavior of the Class*. Classes have *attributes*, which are the information about the Class' instances, and *operations*, which capture the available "services" or behavior of the Class. There are various properties associated with attributes and operations which are beyond the scope of this paper.

Relationship

A relationship is a "connection" between two Classes, and is drawn as line between two Classes.

There are several flavors of relationship:

- Association – a "structural" relationship. The semantics are defined by the name/role (see below).
- Generalization – this is the "inheritance" structure, represented by a triangle. This paper does not address generalization/inheritance.
- Aggregation – this is a weak construct that indicates some "grouping"; note this has no clear definition or semantics.
- Composition - this construct indicates a parent/child Association where the children cannot exist without a parent. Note that this is UML's equivalent of a relational dependent foreign key constraint, e.g., it is both mandatory and non-updateable.

Associations have "adornments", which include:

- Name of the Association.
- Role(s) – these are additional wording that describe the "role" that each Class plays in the Association. Typically, an Association will have either roles or a name, but not both.
- Multiplicity – multiplicity determines how many instance of each Class may be involved in Association, typically, 0, 1, or many.
- Navigation – represented by a simple open arrowhead. The Association can only be navigated in one direction, e.g., starting with one instance of Class A, you can find the instances of Class B that are party to the Association, but, given an instance of Class B, you cannot find the instances of Class A. Note that this is a constraint which should be handled in implementation, but is seldom actually enforced.
- Aggregation – represented by an empty diamond (see Aggregation above).
- Composition – represented by a solid diamond (see Composition above).

THE PROBLEM

The following "diagram" summarizes all of the common problems found in UML Class Models.

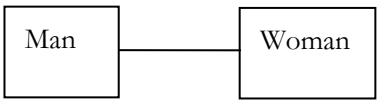

Figure 1

There is simply no way to determine what is intended here. The Association could be anything. It could be a dating relationship, birth mother/son, marriage, who knows? And if it were marriage, what are the constraints on the marriage? Monogamy? Polygamy? Polyandry?

But the following model is clear:

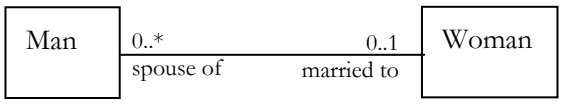

Figure 2

The simple addition of appropriate multiplicity and wording makes it perfectly clear that this is a model of monogamous marriage between men and women.

RULE 1 – EXPLICIT MULTIPLICITY

Rule 1 is simple; *always* state *explicit* multiplicity, both the *minimum* and *maximum*. In particular, avoid using "1" in lieu of 0..1 (or is it 1..1?). This makes it clear to everybody, including the casual diagram reader, exactly what is meant. (See Rule 10 below for more on this issue).

RULE 2 – NAME THAT ASSOCIATION

Associations may have one "Association name or a role name for each end of the Association (or both)". Unnamed lines are useless and lead to endless, futile speculation about the intent. In general, you should standardize on using either Association names, roles or both, but having and following a standard is most important.

RULE 2.1A – USE ► FOR ASSOCIATION NAMES

When using Association names, the ► indicates which direction the Association should be read as shown in figure 3 below.

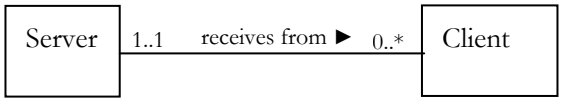

Figure 3

As shown in Figure 3, this eliminates any confusion as to which direction the "information" flows between the Server and the Client. Without the ►, ambiguity rules the day.

I'm partial to roles, as they both reduce ambiguity and make it easier to read the Association in both directions, e.g.,:

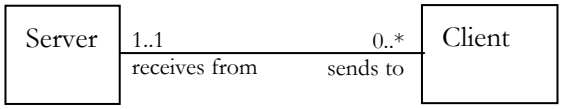

Figure 4

Using roles, nobody has to remember (or argue about) the "opposite" of "receives from".

RULE 3 – USE "GOOD" NAMES/ROLES

Coming up with meaningful, understandable names/roles is what distinguishes the top modelers from the mediocre. All too often, the names in UML Class Models are virtually meaningless. Figure 5 is taken from a so-called "industry standard" model.

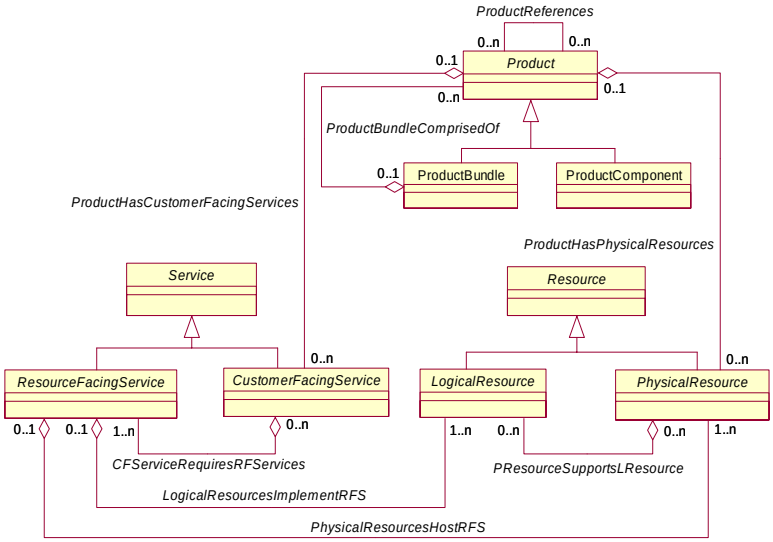

Figure 5

Note that the names effectively contribute nothing to understanding the model, but merely restate the same information contained in the artifacts themselves, e.g., *ProductReferences* provides no additional useful information; it just restates the existence of the association. *ProductHasPhysicalResources*; duh!

Further the inclusion of the Class name in the Association names makes the model even more difficult to read and layout. Avoid this type of redundancy!

RULE 3.1 – USE SPACES OR UNDERSCORES…

because *ReallyLongNamesWithCapitalizationAreHardtoReadforMereMortals*

This is of course totally counter to "popular wisdom". Isn't

Really Long Names with Capitalization Are Hard to Read for Mere Mortals

easier to read? Or even *Long_Names_with_Capitalization_Are_Hard_to_Read_for_Mere_Mortals?*

Of course, this assume that the modeler actually wants the model understood by the non-techie!

RULE 3.2 INCLUDE ADDITIONAL INFORMATION

Every tool supports additional free form text for Definitions/Description/Comments on Classes, Associations, attributes, etc. *The diagram is not the model!* Names/roles that appear on a diagram are seldom sufficiently descriptive or detailed; capture the additional knowledge in your tool!

RULE 4 – USE E/R READING CONVENTIONS

Over the years, I've found that the use of "may be" (or "may") for optional relationships and "must be" (or "must") for mandatory relationships enforces a discipline and consistency that greatly improves the quality of models and substantially improves communication with the non-techie. It is easy to train the non-techie to read models using this technique, and modelers make fewer mistakes when they adopt this approach.

The use of "zero, one or more" and "one or more" does not convey the clarity of purpose of optionality. Using "regular" English also make the non-techie more accepting of the modeler as being someone other than a priest of geekdom.

But whichever reading convention you adopt, *be consistent!*

RULE 5 - RESOLVE MANY TO MANY (M:M) ASSOCIATIONS

The failure to resolve M:M Associations is probably the single biggest problem in most UML Class Models. Resolving M:M Associations should be considered one of the most important jobs for the modeler. M:M resolution is the heart and soul of the iterative refinement of the model. M:M's are always hiding important information, requirements and constraints.

Allowing M:M's to remain results in very weak Object/Relational mappings. The resulting database will be difficult to modify and tune, and will be overly dependent on the Object/Relational mapping tool.

Direct implementation of M:M Associations in procedural code also leads to very brittle applications, which require constant maintenance as missed requirements surface.

RESOLVING M:M ASSOCIATIONS

To resolve a M:M Association:

1. Create a new Class (*not* an Association Class).
2. Create Associations with 1..1 and 1..* multiplicities back to the original Classes. While it is technically correct to use composition, I recommend against this, as composition tends to become cast in concrete and tends to inhibit re-examination.
3. Use meaningful names for new Class and Associations. *This is the most critical step*! Good names will lead to more refinement and discovery; poor names will block any further discovery. The name of the new Class should be derived from one the original Association role or name, not a combination of the original two Classes. The new names should be meaningful to the business.
4. Examine new Class for attributes and Associations.

As an example:

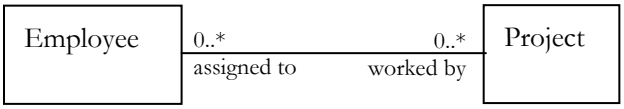

Figure 6

After step 3, the model becomes:

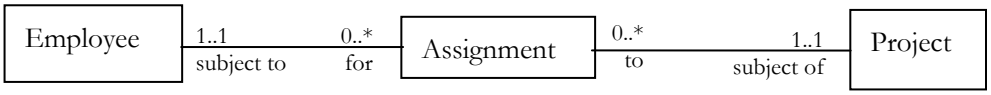

Figure 7

Per step 4, we take the revised model back to the business (which happens to be a consultancy), where we are informed that what they really want is the billable hours for each employee for each week worked on a project. The new "discovery results in:

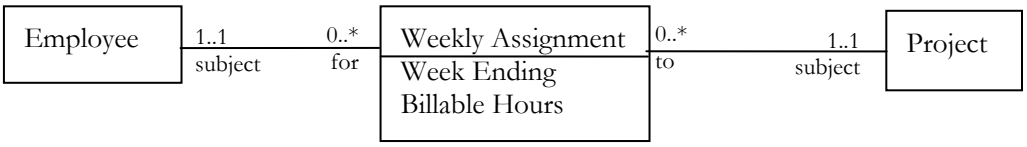

Figure 8

Note that this was made possibly by using "Assignment", which was meaningful to the business. Had we used "Proj/Emp" or "ProjectEmployee", as is so often recommended, the entire process would almost certainly have come to a premature end, as these terms have no meaning to the business.

I cannot stress the importance of good naming strongly enough; it is the foundation of high quality models!

Rule 5.1 – Eschew Association Classes

Don't use Association Classes for the following reasons:

1. They are confusing to end users.
2. There is generally no support in today's programming languages (nor should there be).
3. The only significant difference from a regular Class is in fact a severe limitation; *they cannot be attached to more than one Association!* This is a roadblock in the iterative refinement path.

Use regular Classes instead. They are less confusing and lead to better analysis. There is no need to "convert" them when a new, meaningful Association is discovered.

Rule 6 – Avoid Dependencies

A dependency is *"A semantic relationship between two things in which a change to one (independent thing) may affect the semantics of the other (dependent thing)" – The Unified Modeling Language User Guide, Booch, Rumbaugh and Jacobson.*

This is one of those tools that should never be used.

It is basically a *code* construct dealing with input parameters. Even *The Unified Modeling Language User Guide* states "If you provide the full signature, you don't normally need to show the dependency".

Just say "no" to dependencies!

Rule 6.1 Avoid Navigation

Navigation is seldom meaningful. Navigation is a "constraint" that limits the ability to navigate from one Class to another. It restricts the flexibility of the resulting application, (if it were actually implemented; most developers ignore them).

Further, they are generally incorrect. In 99% of the models I've reviewed that used navigation, I could easily construct a business case that violated the navigation.

Use Association roles/names instead.

Rule 7 – Use Aggregation Sparingly

"*Simple aggregation is entirely conceptual and does nothing more than distinguish a 'whole' from a* 'part'" – *Unified Modeling Language User Guide*

Aggregation has no real semantics, and it is all too frequently confused with composition, which has very important and well defined semantics. Even the book constantly mixes them up and often uses them incorrectly in the same chapter!

They add nothing to the model. Clearly stated Associations are much better.

Rule 7.1 – Be Careful with Composition

Composition has well defined semantics, e.g., a child cannot exist without a parent. Only one parent is allowed, and the parent should not be changeable. This has obvious implications on the final implementation, so use it only when this type of Association is actually required.

Rule 8 - Avoid N-ary Associations

UML allows N-ary Association, e.g., an Association among 3 or more Classes. This approach was abandoned by the relational world in the last millennium, as the binary Association was clearly superior. (Someday the OO world *will* catch up).

N-ary Associations are extremely confusing and seldom informative. Any such Association will always require attributes to be added at some point, and the ultimate implementation will still be a table or OO class/object.

This is simply bad modeling, and should never be seen.

RULE 9 – BE STINGY WITH OBJECTS

Objects are *instances* of a Class. In general, they are not appropriate in Class models. They are too specific for anything other than a very concrete design model, and even then would be questionable. Objects are normally used in other types of UML models.

If you do use them, be sure to specify which Class to which the object belongs using the "object:class_name" syntax.

RULE 10 – GET THE OPTIONALITY CORRECT

This is the most common and egregious mistake to be found in UML Class Models. If these rules were actually implemented as specified, the resulting application would generally be non-functional!

It is particularly sad how pervasive this error is throughout the literature; *The Unified Modeling Language User Guide, Booch, Rumbaugh and Jacobson* is replete with faulty optionality, as shown in Figure 7 and describe below.

The rules for determining the correctness of optionality are simple:

If the multiplicity is *mandatory,* i.e. 1..0 or 1..*, then an instance *must always, with no exceptions, be present.* The modeler's job is to try to determine if there is any situation where this would not hold true. *Don't assume it doesn't matter!* While many procedural programmers will not enforce optionality, some will. But database designer and O/R tools *will* enforce this as a database constraint! If this is enforced and is incorrect, the resulting application will be full of exceptions, dummy values and in general of very low quality.

If in doubt, the Association should be optional, e.g., 0..1 or 0..*.

Let's examine the example from the book and identify some of the obvious business issues.

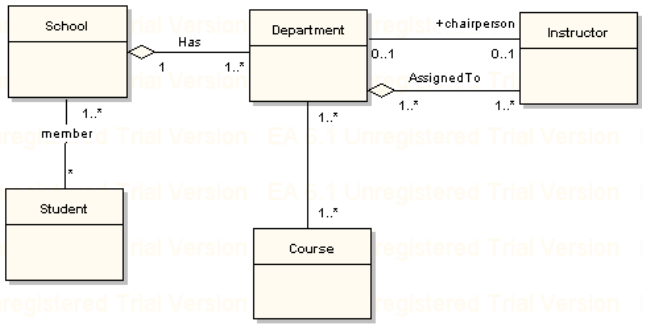

Figure 7

- We can't have a School without a department. How do we create a new school and add Departments later? Do we have a dummy Department?
- We cannot create a Department without an Instructor. Do we have a dummy instructor?
- An Instructor may only be chairperson of one Department. What is the business case for this? If we have a Department without a chairperson, wouldn't we want to assign an experienced chairperson as an "interim" chairperson?
- We can't create a Department without at least one course. Obviously, "UML Quality for Dummies" should be the name of the dummy course!

SUMMARY

While UML Class Models are all too often confusing, misleading and ultimately incorrect, they need not be so. The simple rules describe here, which are so familiar to the E/R community, can and should be applied to UML Class Models. These rules will result in:

- Better communication with the business.
- Less confusion and ambiguity.
- Better, less brittle applications and databases.
- More effective modelers and their resulting work products.