How to Build Articulate Class Models and get Real Benefits from UML

Leon Starr

Software Model Engineer
Model Integration, LLC.
www.modelint.com

San Francisco, California, USA

leon_starr@modelint.com

Abstract

The typical UML® class model is a nebulous representation of the reality it aspires to formalize. This, at least, has been my experience as a longtime executable UML modeler and project consultant. What I am defining as an "articulate" class model is one that expresses critical system rules with transparent, unambiguous precision. The contrast is best demonstrated with a good vs. bad model example. This won't be a contrived model comparison, but one representative of the sort of thing seen all the time on real projects. I will also itemize the negative consequences of an imprecise class model on a software system. And, of course, I will point out the practical benefits of doing things the right way. Finally, I will describe some simple techniques you can use to create more articulate, rule-expressive class models.

A UML class model[1] should do more than just give you a pretty picture of your data structures. An articulate class model will nail down subtle, yet critical, constraints in your application. It will expose hidden rules and assumptions that, when overlooked, lead to the nastiest of bugs down the road. A good model will head off key design miscalculations and save you and your team a lot of time and pain. If you aren't getting these kinds of results from your class models, you probably shouldn't bother with them at all.

Common UML Class models, both on projects and in industry literature, barely scratch the surface of their true potential. Key application constraints are often ignored. Models are open to multiple interpretations. This is often justified under the lazy mantra of "Sort it out in design". Well, if all the hard problems are going to be sorted out in design, why not just start designing now?

[1] The official term is "class diagram", but it is often useful to distinguish between a model which has no particular visual representation and a diagram which implies a 2D visual representation of the model with a particular notation. Both a statechart and a state table may be used to visualize the same underlying state model, for example. I only use the term "diagram" when I am referring to the graphical notation.

THE CLASS MODEL IS MISUNDERSTOOD

This trouble starts with a key misunderstanding. Class models are often described in the UML literature as "representations of static structure". In other words, it is understood and oft repeated that a class model does not reflect dynamic behavior. The fun stuff, after all, is in the statecharts, sequence diagrams, action language, collaborations and activity diagrams. And since the dynamic properties of your software constitute its observable features, it is easy to conclude that class models are the obligatory vegetable dish in the all-you-can-eat meat buffet of the UML. You are supposed to model the classes and relationships first, you are not quite sure why, and you want to get it over with as soon as possible.

STATIC MODELS CAN EXPRESS DYNAMIC RULES

Yes, class models *are* static. Then again, so are *rules*. Consider a rule like "Only one aircraft may take off from a runway at a time." If all goes well, this rule shouldn't be changing during runtime. An air traffic control system comprises thousands of such rules, some intuitive and obvious, others devilishly subtle. The behavior, intelligence, resilience, and reliability, in short, the dynamic personality of your software is defined and constrained by an application rule base. Every application has one and it is nice to crystallize the rule base in a platform independent manner. Few developers appreciate the degree to which class diagram notation can mold and constrain complex system behavior. But to get this result, you have to stop thinking about class models as mere repositories of data.

SO WHAT'S WRONG WITH AN INARTICULATE MODEL?

If the rules aren't expressed in your class model, they have to go somewhere, don't they? Any rules not captured in your class model are deferred to the statecharts. If not captured there, the rules are deferred to the state actions. If you are writing the code by hand instead of using a model compiler, you have yet another opportunity to catch the neglected rules in your code. Beyond that, let the users find them for you. If you are building avionics, anti-lock brakes or medical systems, that last option is considered bad form[2].

BENEFITS OF ARTICULATE, RULE-EXPRESSIVE CLASS MODELS

Why put rules and constraints in your class models? 1) Many rules are easily and more efficiently expressed in the class model once you know how. 2) Rules in a class model are highly visible to users and application experts who can provide critical guidance before the design solidifies. You can step an expert through an articulate class model and get quality input faster than having them absorb pages of sequence diagrams. 3) Increased visibility tends to force key application decisions, which are easily swept under the table until they emerge, when it is too late or expensive to make the necessary changes. 4) States and actions involve sequence, synchronization and timing which is generally harder to test and prove correct than "timeless" static structure. 5) Fewer rules in the actions means less code, more data perhaps, but it is easier to optimize a design to handle data e.g. choosing storage and access for read-only specifications vs. runtime values. 6) Rules expressed in data can be tuned and updated without having to change, recompile and retest the code. 7) By focusing on the data early on, you are more likely to identify and abstract configurable parameters than hard wire them into a procedure. With rules modeled in data you get the best of both worlds - configurability and hard enforcement of core principles. 8) The precision questions that must be asked and answered to build an articulate model will make you smarter - or at

[2] I just downloaded the latest beta into my iPacemaker! It's so kewl.... wait a minute, that's odd....

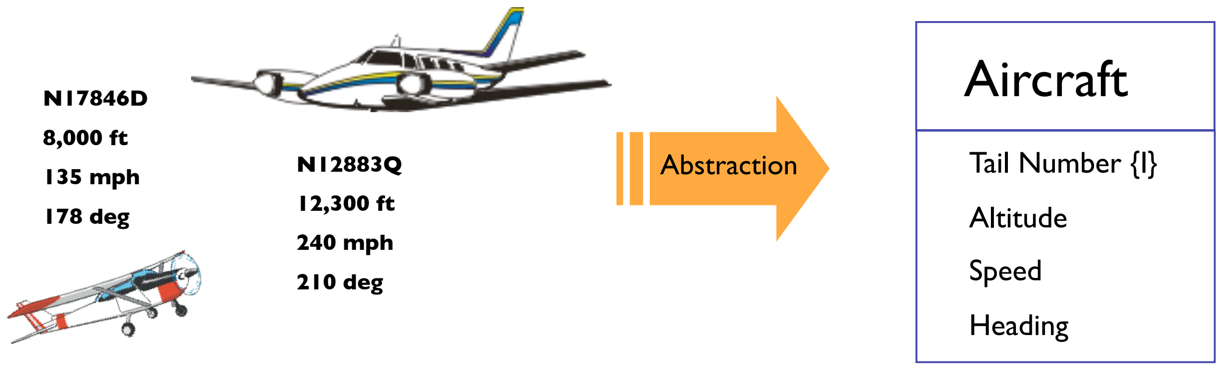
least create the appearance. (I have seen this last feature turn newbies into experts so fast that the newbies end up taking control of the project from the resident software cowboys who assume that no-one is smart enough to learn their stuff). Articulate class modeling is not just a software development technology, it's a brain development technology!

I could keep going, but let's get to those examples!

Capturing multiple rules with a single class

Let's start out simple. We're going to lay out a lot of groundwork with just one single class. So bear with me while we delve into the nuts and bolts. Once we get through this, we can start scaling up to something more interesting. You may be surprised, though, at just how many rules can be expressed in a single modeled class.

Say that we are tracking multiple aircraft flying around an airport. We want to avoid collisions and know where all the aircraft under our control happen to be at any given moment. So our software must maintain an internal representation of each aircraft instance flying around.



Abstracting a class from data in the real world

We've abstracted a class called "Aircraft". But what does this abstraction mean exactly? Is it a Java class? Is it an Objective-C class? A Python class? Is it a database table? Is it a section of an XML file? Much of my work is in embedded systems so the answer is often some tight C or Assembler structure. Each implementation technology carries a different set of assumptions and limitations. But our abstraction is none of the above, it's a UML class! But what does *that* mean? This is THE big problem with UML and modeled abstractions in general.

By contrast, when you look at good hard code it is easy to roll the mechanics of a problem around in your head and visualize "what happens if", "what can't happen", and so forth. A serious modeling language requires more than just graphical notation. There must be unambiguous semantics (meaning) underneath the notation so that, like code, a model means one thing and only works one way. It shouldn't be open to wishful interpretation.

DEFINING A PLATFORM INDEPENDENT CLASS

Unlike code, we would like a platform independent specification (as much as possible) so that we can separate the real world (application) rules from the rules introduced by a particular implementation. What we care about are the aircraft management rules that must be enforced in *every* implementation.

Rules about minimum vertical separation distance between two flying aircraft don't change just because the tracking system happens to be running on Linux.

Even though we are using UML, we must think about a class as an actual data structure. A simple rectangle on a sheet of paper has no interesting mechanics. A whole bunch of rectangles on a sheet of paper is a colossal waste of time. But once we agree on a UML class data structure we can evaluate the operations that can be applied to the data in that structure. That's when real thinking can start. But we need a platform independent data structure - is this possible? In fact, thanks to mathematics, set theory, and the relational model of data, there is. A platform neutral data structure for a class is simply a relation — less formally, a table. Not a relational database table (that would be one possible implementation).

Here is what the Aircraft class might look like as a platform independent table:

Aircraft

Tail Number {I}	Altitude	Speed	Heading
N17846D	8,000 ft	135 mph	178 deg
N12883Q	12,300 ft	240 mph	210 deg

A populated Aircraft table

The {I} on the Tail Number attribute is a UML tag used as shorthand for the identity constraint [1]. It signifies that there can be no two duplicate values of Tail Number in the table. So at this point we see that the rectangle class symbol in our abstraction can be understood as a tangible data structure. Operations for manipulating data in a table[3] are well defined in relational algebra and supportable by UML actions. We will use some of these actions to access the Aircraft table data shortly.

Just to be clear, we are not specifying an implementation. The table above could be realized by a variety of implementation structures. The choice will depend on the target platform, read vs. write access optimization, and other performance and programming language characteristics. Our goal is to specify as many platform invariant application rules as possible without demanding anything in particular of the implementation. So the programmer (or model compiler) is free to package this stuff up as he, she or it sees fit as long as all application rules are perfectly preserved. In practice, I have seen tables transformed into everything from C++/Java classes to arrays and lists of structures in C and even simple bitmapped fields in assembly language[4].

EXPRESSING APPLICATION RULES WITH AN EMPTY TABLE (PLATFORM INDEPENDENT CLASS)

Now that we have our platform independent class data structure, we can get back to expressing what's really important - our aircraft rules!

[3] Those of you familiar with relational theory will see that a class is represented as a relation in third normal form (3NF). There is a nice Wikipedia entry on the topic.

[4] A few people are tinkering with translation directly to Verilog (hardware).

Since an application's rules are the same regardless of what instances happen to exist at a given slice of time, let's remove the data and focus on the structure itself, an empty table.

Aircraft

Tail Number {I}	Altitude	Speed	Heading

Empty table is a platform independent data structure

The distinction between structure and content (population) is key. While we often examine the data (instance populations) to abstract general principles and rules, it is the abstraction (structure) that we use for generating code. We throw the data away once we derive the abstractions. But on the way to our abstractions, especially in a complex system, we may plow through a LOT of data searching for patterns and subtle distinctions among instances.

Without worrying about what data happens to be in the table, let's take an inventory of rules expressed by this simple structure.

AIRCRAFT CLASS RULES

1) Each aircraft is identified by a unique tail number.
2) No two aircraft may have the same tail number.
3) Every aircraft has an altitude.
4) Every aircraft has a heading.
5) Every aircraft has some airspeed.

Rules 1 and 2 are established by the identity constraint described earlier. We'll talk more about how the constraint is actually enforced in an implementation later. For now all we know is that it is unambiguously declared on the class model.

Rule 3-5 are consequences of a relational rule which states that every cell must contain a non-null value. It's a nice rule that leads to sharper, less wishy-washy class abstractions. In this case, you have to ensure that all flying things covered by the Aircraft definition do in fact always have legitimate values per attribute at all times.

WHAT QUESTIONS CAN THIS CLASS ANSWER?

The power of an underlying data structure is best demonstrated by asking the model some questions and seeing if it can answer them. For each question, ask yourself if it can be answered by the populated table.

1) What is the heading of N12883Q?
2) Which aircraft are below 10,000 ft?
3) How many aircraft are in our Control Zone?
4) How fast is N17846D going?
5) Is a collision likely in 5 minutes?

Question assessment

(1) Yes. We can scan the Tail Number column, pick out "N12883Q" and following the row to the value in the Heading column. If we don't find the Tail Number, we conclude that there is no such Aircraft in our area.

(2) Yes. We scan down the Altitude column selecting each row with a value < 10,000 ft and then jump across the row and report the Tail Number.

(3) No. The model does not say anything about a Control Zone. (And neither did I until I listed the question!) You might expect to find another class titled Control Zone and some attribute reference in the Aircraft table to a Control Zone ID of some sort. But there is none in this example, so the answer is "No".

(4) Yes. Find the Tail Number as in (1) and scan across to the Speed column to get the value.

(5) No! We have insufficient information to compute collision possibilities to any useful degree of certainty. That's because we don't have enough spatial coordinates for each Aircraft. We know where an Aircraft is pointing, how fast it is going, but we don't really know where it is. Easily fixed, though:

Aircraft

Tail Number {I}	Altitude	Speed	Heading	Lat	Long

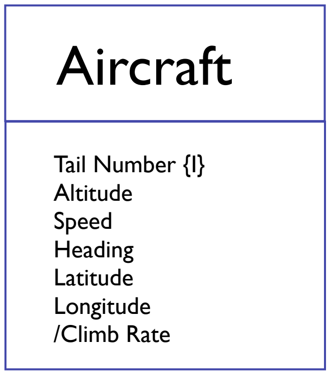
A smarter table that can answer more questions

Now we can answer question (5).

Notice that we really didn't need the populated table to evaluate our questions. We could use the empty Aircraft table and just rephrase the questions as follows:

1) What is the heading of a specified aircraft?
2) Which aircraft are below a specified altitude?
3) How fast is a specified aircraft going?
4) Is a collision likely in a specified time period?

All of these questions can be answered by the improved empty table, though you might insist on adding "Climb Rate". It's an interesting piece of data since it is computed from what we already have. Consequently, we can argue that the table is smart enough to answer the collision question with or without this derived attribute. It's reasonable to add it though, so let's go back to our UML Class notation and put it in.



Tail Number {I}	Altitude	Speed	Heading	Lat	Long	/Climb Rate

The / in front of Climb Rate indicates that the value is derived computationally. The judgement of whether or not to add computationally derived attributes is the subject of a whole other article, so I'm going to just say "Put them in as needed." for now.

UML AND UNDERLYING SEMANTICS

To recap, we've looked at a trivial example, a single class, and observed that several rules are already captured. The table representation gives us a hard data structure so that we can think about the mechanics of data access without regard to any particular implementation. In the process we found that we could make definitive statements about what we know, what we can compute and what we cannot compute. This all comes without knowing *anything* about the statecharts, actions or algorithms. Just the drawing of a single class puts us in a position to answer questions and to evaluate conformance with the world we plan to control. Just imagine how many rules can be captured[5] when we add relationships and hundreds of classes!

[5] There is a direct and fascinating correspondence between relational elements and logic theory described in [4]. The gist of it is that a relational model is essentially a set of logical predicates.

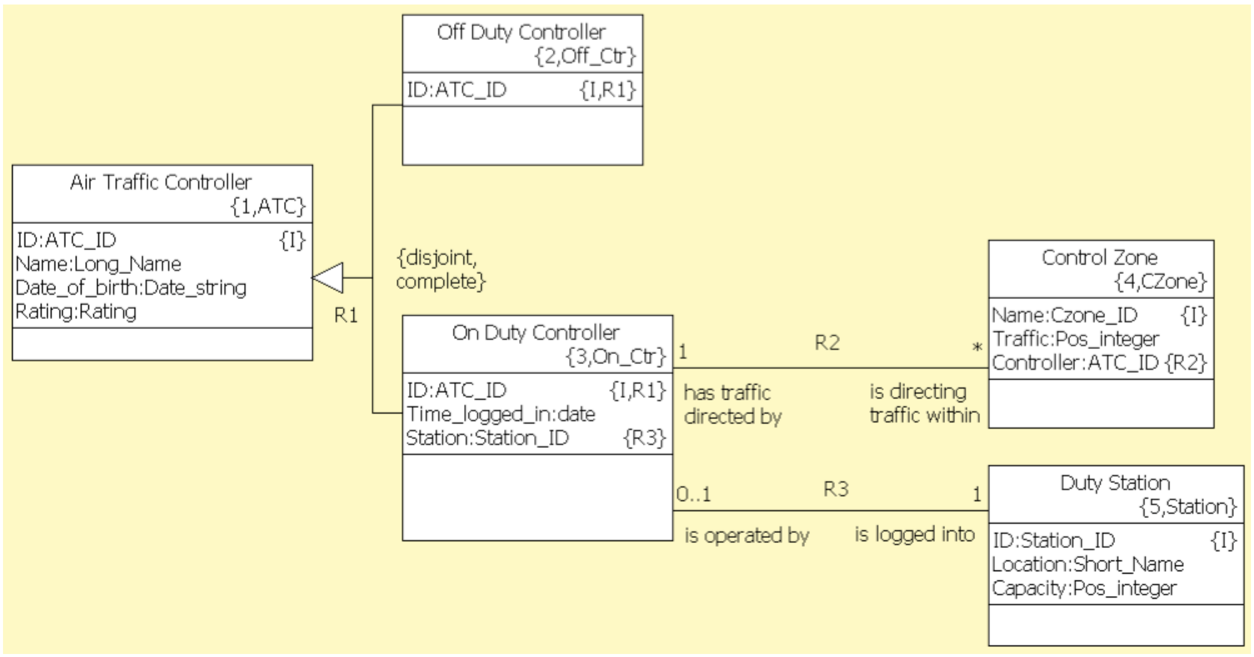
ENFORCING THE IDENTITY CONSTRAINT

You might wonder where the {I} constraint is actually enforced. The class model simply states the identity requirement. When code is generated, by a programmer or model compiler, there should be a systematic mechanism for handling identity. An embedded architecture might make use of a hash table whereas an enterprise architecture might rely on available database mechanisms for detecting and rejecting duplicate entries.

A Good Model

Moving along to our model comparison, we start with the "good" model then see what's missing in a "bad" model example. I apologize for any narcissism in presenting *my* models as "Good Models" and everyone else's as "Bad Models". As a default tendency it's not a constructive (and is an often wrong) practice! Naturally, we need to focus on tangible practical differences that can help us all improve analysis and modeling technique and generate better code.

Here is the class diagram of the purportedly Good Model.



A Good Model

Some of the notation above is familiar UML, but there are a few twists. Let's just go class by class and see what's going on underneath all this notation. We'll zoom in to the underlying table data structure and then pull back and evaluate big picture again. After that, we will finally be in a position to contrast this model with a Bad Model.

AIR TRAFFIC CONTROLLER CLASSES

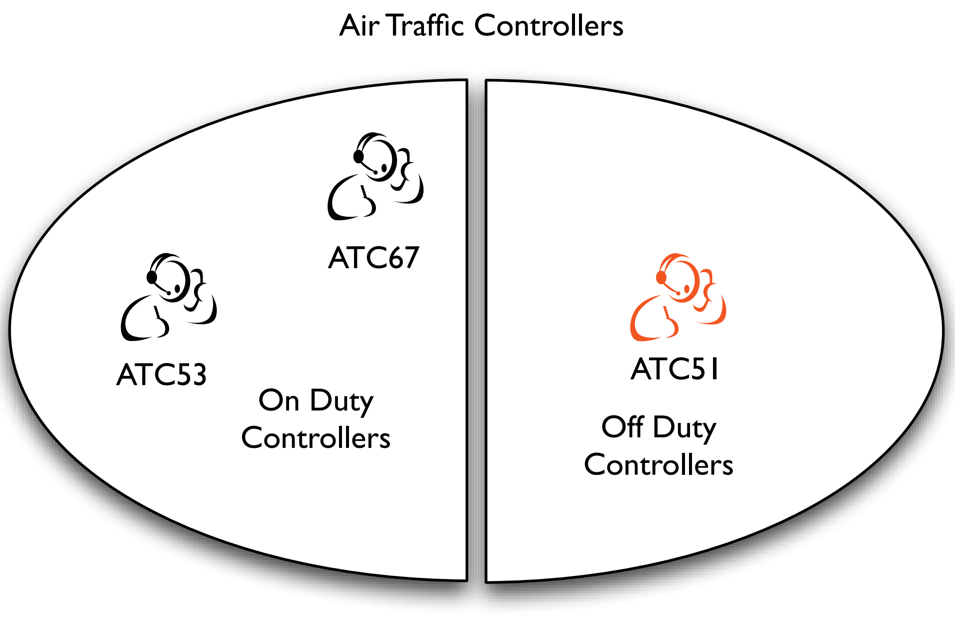
The left half of the model features three classes: Air Traffic Controller, Off Duty and On Duty Controller. The superclass table, populated with some sample data, might look like this:

Air Traffic Controller

ID {I}	Name	Date_of_birth	Rating
ATC53	Toshiko	Jun 12, 1975	A
ATC67	Gwen	Mar 28, 1981	B
ATC51	Owen	Dec 23, 1974	C

We have three Air Traffic Controllers and we know the name, age and overall skill (rating) of each. We also see that ID values must be unique since the {I} tag is present. This table contains data relevant to an ATC regardless of on or off duty status.

The {disjoint, complete} text next to the generalization relationship R1 is a pair of standard UML tags. It indicates that every Air Traffic Controller object is either an On Duty or an Off Duty object. The "disjoint" part means that it is not possible to be both on and off duty at the same time. The "complete" part means that every Air Traffic Controller is definitely on or off duty. Work status is defined at all times for each ATC. And it is certainly not possible for an On or Off Duty object to not be an Air Traffic Controller object! So R1 does not represent inheritance style generalization. It is more akin to relational set subtyping. It is the XOR of class modeling - an incredibly useful tool for incorporating logic into data structure.



Disjoint - Complete Generalization Relationship

On Duty Controller

ID {I, R1}	Time_logged_in	Station {R3}
ATC53	9/27/08 3pm	S2
ATC67	9/27/08 11am	S1

When an ATC is on duty, he or she must be logged into a Duty Station. Both the Station ID and the login time is kept for each of these ATC's. This data is relevant only while on duty.

Note the {R3} tag on the Station referential attribute. This tells us that the "Station" corresponds to the identifier on the other side of R3. Duty_Station.ID, in this case. This is another example of platform independent data structure mechanics. You can ask the question, "What station is ATC53 logged into?" It is clear from the table structure that you can just find the row containing ATC53 in the ID column and scan across to the Station column and return the value, "S2". You can go even further and retrieve data from the Station table using S2 as a key. So you can answer a more complex question like "What is the location of the Station logged into by ATC53?". You can query in both directions on any class model relationship, so you can also answer the question, "Who is logged into Station S2?" Just scan through the On Duty Controller table in the Station column, find the row and return the ID value, ATC53.

The point of all this row/column scanning is simply to show that the data is connected somehow in any implementation. It doesn't mean that we need to implement our actions in SQL or anything like that. The systems I work with usually strip out the referential attributes and replace them with pointers, handles or indices of some sort. Access directions on each association are typically optimized or disabled[6] as necessary. Model compilers exist which will do this automatically yielding nice C, Java or C++ and, believe it or not, even Assembler! Generating some kind of SQL is certainly an option for database-y types of systems, though.

By the way, note that there is an R1 tag on the ID attribute indicating that it is simultaneously an identifying attribute of On Duty Controller which matches its ATC superclass ID. All relationships in the Good Model are glued together with referential attributes.

Off Duty Controller

ID {I, R1}
ATC51

Oddly, there is no data kept for Off Duty Controllers. None that we know of yet, anyway. The main value of this class is that it shows that login times and station IDs are NOT maintained for Off Duty Controllers. This is a trick that keeps us from needing "not applicable" values in our tables.

[6] Some model compilers will, for example, scan the action language, see that no write accesses are made in a particular association direction and omit the code for the relevant accessor.

What's wrong with "not applicable"? If you permit these non-values, you are asking for if-then logic in your code to treat the special cases. You are also demanding storage space that the application doesn't really need.

Also note that when an ATC goes off duty, the Station and Time_logged_in data will be discarded (though possibly archived). Since an ATC can't be both on and off duty at the same time, when an ATC object migrates, it acquires or loses data as evident in the populated tables below.

Air Traffic Controllers

ID {I}	Name	Date_of_birth	Rating
ATC53	Toshiko	Jun 12, 1975	A
ATC67	Gwen	Mar 28, 1981	B
ATC51	Owen	Dec 23, 1974	C

This data stays the same on or off duty

Off Duty Controllers

ID {I, R1}
ATC51

Roles can migrate back and forth, adding and dropping data

On Duty Controllers

ID {I, R1}	Time_logged_in	Station {R3}
ATC53	9/27/08 3pm	S2
ATC67	9/27/08 11am	S1

Role migration

A distinction is thus drawn between an ATC object which is the sum of its general and specific data, represented at any one time by two class instances.

Control Zone

Name {I}	Traffic	Controller {R2}
CZ1	12	ATC53
CZ2	4	ATC53
CZ3	8	ATC67

A Control Zone is a region of air space managed by a single ATC. The rule we want to capture is that each Control Zone must, at all times, be handled by an ATC.

For each Control Zone we have a unique name, a quantity of traffic and an assigned ATC. You can probably imagine other useful attributes like location, volume, etc, but I am trying to keep this teaching example simple!

So now we can answer questions like "What is the quantity of traffic managed by ATC53?"

Duty Station

ID {I}	Location	Capacity
S1	Front	20
S2	Front	45
S3	Center	30

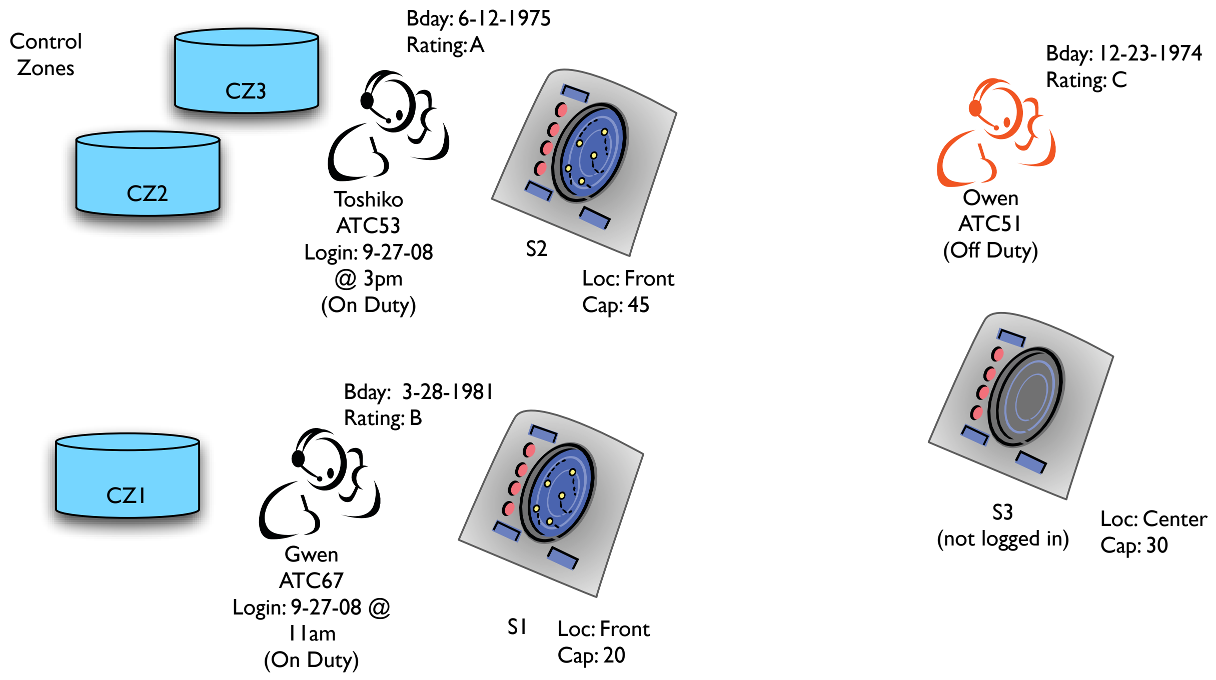
Each Duty Station has a unique identifier, a general location in the facility and a maximum amount of traffic (Capacity) that it is certified to manage. There are no referential attributes here since we already have one in the On Duty Controller class. So if we ask a question like "How old is the person logged into station S2?" we can handle it. Just take the value "S2", scan through the On Duty Controller table to locate the row. If it exists, go into the ATC table, find the row containing "S2", and get the value out of the Date_of_birth column. Compare to the current date (should be available as a core system service), do the math and you have the age value.

As mentioned earlier, these access steps are only relevant when reading or executing[7] the model. It is up to the programmer or model compiler to devise an efficient way to glue the pieces together. This sets up a nice division of responsibility with the modeler/analyst saying what data must be connected on *any* platform and the programmer/model compiler finding clever ways to implement the connections for specific platforms. This is not as complex as it sounds since a single systematic translation pattern may be applied (along with a few configuration parameters for local customization) across numerous model elements. A handful of such patterns will suffice to handle most, if not all of the code. So there is no need to lovingly handcraft the code for each individual class.

[7] When you add state and action models, you can execute the models in a simulator that manages state transitions, event queuing, collaboration and data access/computation. Once you get everything working, you can generate or write the code.

An illustrated scenario

The tables are a nice way of formalizing the abstract structure of data. But an illustration can be helpful in testing to see if those tables can accommodate a specific real world scenario.



Illustrated ATC Scenario

The illustrated scenario shows three ATCs, two on duty and one off duty. Each On Duty Controller is logged into a single dedicated Duty Station. We have recorded the login time for each of these ATCs. Each Control Zone is being managed by a single On Duty Controller. The Off Duty Controller is not logged in and has no login time recorded. One Duty Station is sitting inactive. Neither the Off Duty Controller nor the inactive Duty Station has any association with Control Zones. It seems our Good Model can, in fact, accommodate a realistic scenario.

This style of diagram, which purposely avoids any UML notation, is one useful tool for avoiding the dreaded analysis-paralysis. When you limit yourself to a bland palette of boxes, lines and stickmen, it is easy to get absorbed in your abstractions and lose sight of interesting situations that break the rules. Exclusive focus on your model elements, ironically, makes it easy to miss patterns in the real world data that may yield clever abstractions.

So to keep from getting caught up in model hacking (a constant struggle!), I alternately draw a freeform scenario diagram, then build a UML model, then compare them. Validation works in both directions. Can I populate my tables with the data in the scenario illustration? Can I generate a convincing scenario illustration using only the data in my tables? If I discover a new scenario, I check to see if the data will fit the model. If I extend the model, I may draw an updated scenario to see if it makes sense. Going back and forth in this manner will repeatedly reveal flaws in my thinking and ultimately lead to a solid, articulate model. As a nice side effect, I leave a trail of useful documentation! Another great benefit is that users and application experts will give me much more detailed and useful feedback on a scenario illustration than on a model or sequence diagram. Unfortunately, existing UML tools support only the line-box-stickman aspect of modeling and analysis. A huge swath of analysis tooling is yet untapped!

So invent your own notations and icons, steal clipart, draw freely and use the UML notation to crystallize your abstractions.

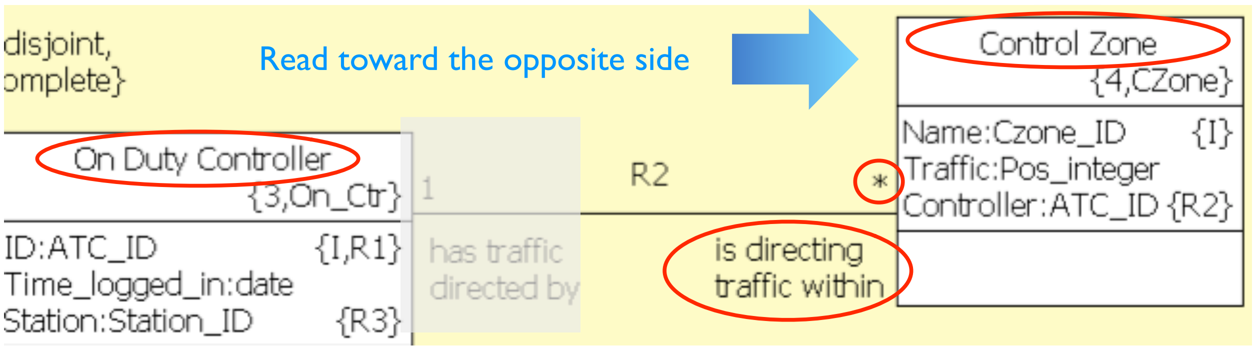
Air traffic control relationships

So far, our focus has been on the classes but there remain two critical associations in the Good Model.

R2 - is directing traffic within

If you look at R2 on the diagram you may notice that the style of naming the roles is a bit different than what you may typically see on common UML class diagrams. This is intentional, and you'll see that I use the common, and less descriptive style, on the Bad Model example.

To read an association named using verb phrases, start on one side of the association and read the phrase, multiplicity and class name on the opposite side as shown.



On Duty Controller is directing traffic within zero or many Control Zones

In plain english, we have "an On Duty Controller is directing traffic within zero or many Control Zones". So we see that it is possible for an ATC to be on duty, but not handling any Control Zones at a given time.

The other way around we have "a Control Zone has traffic directed by exactly one On Duty Controller". So at all times, a given Control Zone is being handled by some On Duty Controller. And only one at a time!

R3 - is logged into

Here we see that an On Duty Controller is logged into exactly one Duty Station. So it is clear that you can't be On Duty unless there is an available Duty Station and you are successfully logged in.

From the other perspective, a Duty Station may or may not be operated by an On Duty Controller at any given time. You could also say "is operated by zero or one On Duty Controller" — same thing.

RULES EXPRESSED BY THE GOOD MODEL

Now that we've walked through the model elements, the important thing is to take stock of the rules and constraints expressed, not just by individual elements, but by all the elements taken in combination!

1) An Air Traffic Controller is either an On or Off Duty Controller at any given moment. {R1}
2) An On Duty Controller must be logged into a single Duty Station. {R3}
3) At any given time, a Duty Station may or may not be operated by (logged into by) a single On Duty Controller {R3}
4) A Control Zone must have its traffic directed by exactly one On Duty Controller at all times. {R2}
5) An On Duty Controller may or may not be directing the traffic in one or more Control Zone at all times. {R2}

WHAT BEHAVIORAL CONSTRAINTS ARE EXPRESSED IN THE GOOD MODEL?

As you can see, the multiplicity and phrase on each side of an association is critical to establishing precise rules. Now let's ask some behavioral questions. To get the full benefit, put the Good Model in front of you and work out the answer to each question on your own. No cheating!

1) What must happen when an Off Duty Controller becomes On Duty?
2) If there is only one On Duty Controller left, can he or she go off duty?
3) If every Control Zone is being directed, can another Off Duty Controller log in?
4) If there are 5 On Duty Controllers and 5 Duty Stations, what must be done to take a Duty Station off line for maintenance?
5) If there are 3 Control Zones and 3 On Duty Controllers what is the maximum number of Control Zones handled by the same On Duty Controller?
6) Assuming there is at least one instance of Control Zone and only one On Duty Controller, can that On Duty Controller go off duty?

THE ANSWERS

1) To become On Duty, it is necessary to log into an available Duty Station. The Off Duty Controller instance will be deleted and replaced[8] by a new instance of On Duty Controller referring to the same ATC instance.

2) It depends. Every Control Zone must be directed. Assuming there are one or more instances of Control Zone, the answer is "No". It won't be possible to migrate the On Duty Controller instance without breaking the left side of the R2 association, unless there are zero instances of Control Zone

[8] In embedded systems, creating and deleting in the model is sometimes implemented by preallocating a region of memory and flipping a bit to indicate whether an instance is really there or not. So there are other ways to implement creation/deletion other than constructors and destructors.

in which case the answer would be "Yes". If there were just one other On Duty Controller, the Control Zones could be handed off first, but there isn't in this case, so it's not an option.

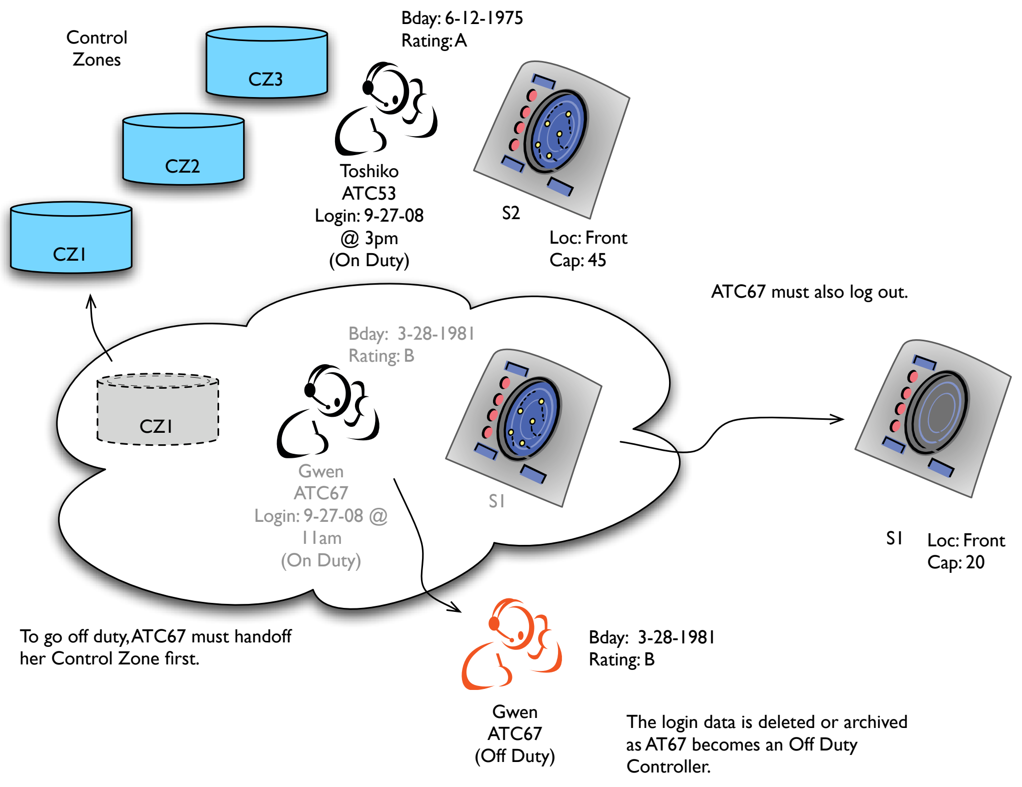
3) Yes. An On Duty Controller is not required to direct any Control Zones. Consequently, the number of On Duty Controllers is only limited by the availability of Duty Stations.

4) Since there is an equal number of Duty Stations and On Duty Controllers we can assume that each Duty Station is in use. So we must log out an On Duty Controller, but to do that, we must make take the ATC off duty. And to do that, we must first ensure that all Control Zones directed by that On Duty Controller are first handed off to some other On Duty Controller.

5) Three. It is okay for an On Duty Controller to have zero Control Zones, so one of them could be directing all three. The other two On Duty Controllers would not have any Control Zones assigned. We just need to ensure that each Control Zone is being directed.

6) No. (Some Off Duty Controller must first become on duty, then all Control Zones must be handed off).

Here is an illustration of some behavior required by the Good Model when an On Duty Controller goes off duty.



ATC67 migrates from on to off duty

The populated ATC tables are updated as shown:

Air Traffic Controllers

ID {I}	Name	Date_of_birth	Rating
ATC53	Toshiko	Jun 12, 1975	A
ATC67	Gwen	Mar 28, 1981	B
ATC51	Owen	Dec 23, 1974	C

Off Duty Controllers

ID {I, R1}
ATC51
ATC67

On Duty Controllers

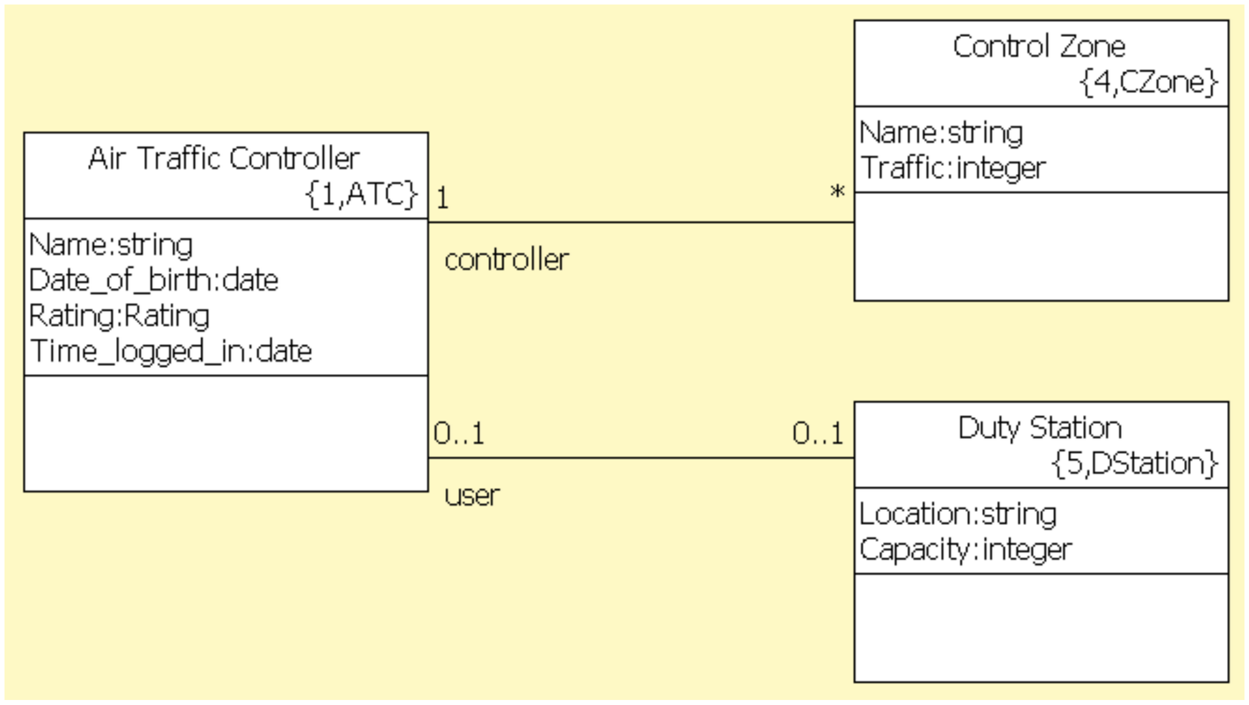
ID {I, R1}	Time_logged_in	Station {R3}
ATC53	9/27/08 3pm	S2

ATC67 migrates from on to off duty

The goal is to create a model that allows legal data sets only. After all, if only legal data can be entered into the data structures, there is no need for code to check for the consequences of bad data! More restrictive class models mean less code, less testing, and higher quality from the get-go. In practice, this can be an elusive goal, but the pursuit causes difficult questions to be asked and answered. As we will see, the Bad Model fails in this regard. On to the Bad Model!

A Bad Class Model

Now let's take a look at the type of UML class model seen on numerous projects and unfortunately encouraged by many books on UML.



A Bad Model

It's the same application, but a very different model. A superficial comparison will reveal the following differences.

1) Fewer classes and relationships
2) Shorter and incomplete names on associations
3) No referential attribute or identifier tags
4) Less precise (implementation oriented) data types on attributes

A careful look, taking all the elements together, reveals that some application rules are missing and even stated incorrectly.

First let's take stock of the superficial differences and then dive into the deeper problems affecting behavioral constraints.

FEWER MODEL ELEMENTS

There are fewer classes in the Bad Model because on and off duty roles are not modeled. One consequence is that we will have a meaningless value for Time_logged_in for each off duty ATC. This will introduce some if-then logic to the actions and/or statecharts. Also this model is telling the programmer or model compiler to reserve space for an unnecessary attribute. Now this may not be a big deal with a handful of ATC's, but what if we had thousands? Still it might not matter much, but what if we scattered not-applicables across a multitude of classes? In some systems it is not unusual to have hundreds of thousands of instances. So this is just one small way in which a more compact model can lead to less efficient code. In practice, there are many times when a larger, more detailed class model carries the DNA necessary to spawn a tighter implementation.

Naturally, less is better, but only as long as it does not come at the cost of expressing application rules. Modeling is all about analysis which means "taking things apart". Design and implementation, on the other hand, focuses on synthesis, "packing things together". A modeler performing object oriented analysis strives to divine and expose all of the application rules using as many components as required. The programmer or model compiler maps and repackages the analysis elements as necessary to yield an efficient design on the target platform. This repackaging must account for every modeled rule, though it is perfectly acceptable to reduce the total number of elements in the process. That's what design is all about - clever packaging!

It is important to understand that the analyst's and implementer's goals are often contradictory. A model that attempts to satisfy both purposes simultaneously usually compromises each. You end up with cramped analysis and bloated implementation which is one reason why good programmers often avoid UML. Our goal in keeping design out of the models is to get the best of both worlds.

SHORTER NAMES

The common role naming style is used to label associations the Bad Model. Roles are okay if you are creating a class diagram as a more or less direct picture of your code implementation. For analysis purposes, however, verb phrases are much more effective.

In the Bad Model, R1 says that an ATC plays the role of "controller" with respect to zero or many Control Zones. Since it is hard to think of an opposite role "controllee?" it is just left out. The "think of a role name game" is often a waste of time that leads to some pretty stupid names. You don't have to play word puzzles with verb phrases, though you do need to think about what you really mean.

In fact, since the role name often matches the associated class name, you can usually drop them altogether without affecting a model's expressiveness. An Air Traffic Controller is a controller? Thanks, that clears things up!

In fact, it is difficult to tell from the Bad Model whether we mean that an ATC is assigned a Control Zone on an ongoing basis (like a reserved parking space or a favorite coffee cup) or if we are talking about the current moment. Note that the Good Model verb phrase pair "is directing traffic within" / "is having traffic directed by" clears up the temporal confusion in addition to explaining what "control" really means.

By placing a precise verb phrase on both sides of each association, the analyst is forced to consider the multiplicity and conditionality carefully on each side. Since this is where many of the rules are expressed, this is critically important. Generic, all-inclusive verb phrases such as "contains", "is a group of", "has" and my favorite "is associated with" are to be avoided. You need only google "composition vs. aggregation" to get a sense of the frivolity and futility in using generic terms to express precise associations. Which statement tells you more? Memory Block "is partitioned into mutually exclusive" 1..* Region or Memory Block "is a (pick one - aggregation / composition) of" Region?

When you say that an ATC “is directing traffic within” <how many?> Control Zone(s) you are forced to consider the application rules in detail. You should realize at this point that it is 0..* and that the zero case is due to being off duty. Rephrasing as class “On Duty Controller” “is directing traffic within” <how many?> Control Zone(s), you hit on a definite multiplicity of 1..* and nail down an important rule. Flipping the verb phrase from active to passive, you get Control Zone is having its traffic directed by exactly one On Duty Controller. There must always be one and he or she must be on duty according to the application rules. Verb phrases guide a model to increased precision.

Why all this fuss when the phrases themselves aren’t actually translated into code[9]? After all, it’s the multiplicity that directs the choice of data structure and access implementation. Ah, but how did you arrive at the correct multiplicity? Models with generic association names almost always belie the true nature of the association and obscure interesting boundary conditions leading to incorrect multiplicity which begets incorrect code.

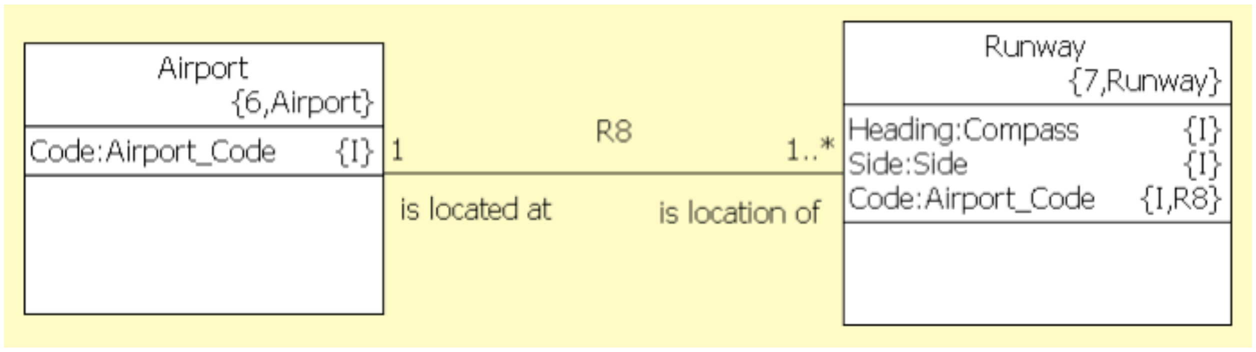
NO IDENTIFIER OR REFERENTIAL ATTRIBUTE TAGS

In this particular example, neither tag type plays an instrumental role, so I won’t dwell on them. The referential attributes simply serve to show that the data is truly connected in table form. Modern model compilers assume the existence of referential attributes and can manage them behind the scenes. That said, there are a few simple techniques where you can propagate and merge referential attributes across multiple relationships to express powerful constraints. I do this a lot, so by default, I retain the referential attributes.

We assume that each instance of a class is automatically unique. In other words, the code will be generated such that each row of a table corresponds to a uniquely selectable entity. Consequently, we don’t need to slap an {I} attribute on every class. Thus if you have a class called “Thing” you could give it an artificial attribute Thing.ID {I}, but it is not necessary. That’s true only for artificial identifiers.

On the other hand, real world uniqueness constraints should always be expressed. Consider the case where we model runways at multiple airports. You can’t have two runways with the same heading + side at the same airport. (Two 28Rs for example!) How do you express this constraint? Here’s a model that does it by incorporating a referential attribute as an identifier component.

[9] Actually, they may appear as concatenated names or appear in comments. But the model compiler is not smart enough to comprehend the text in the verb phrase and choose any kind of design based on it. Probably a good thing, too.



Incorporating referential attributes into an identifier to express a real world constraint

This model says, thanks to the {I} and {R} tags in the Runway class, that a unique instance of Runway can be selected by providing a Heading, Side and Airport Code, 28L at SFO, for example (Runway 28 Left at San Francisco International). So you can combine the identity and referential tags to express a fact about the real world. Namely: Multiple runways may have the same Heading + Side, but not at the same Airport.

LESS PRECISE DATA TYPES

The Good Model uses tightly defined application data types whereas the Bad Model relies on loose implementation types. Consider the Control_Zone.Name attribute. In the Good Model it is defined as type "CZone_ID". You can see from the illustration that Control Zones are named with "CZ" followed by an integer value, thus "CZ1, CZ2, etc". Arguably, this can be accommodated by the loose string type. But when we say, in the model, that we require string, what are we asking of the implementation? If the programmer/model compiler works from the more specific application type, he/she/it can choose a tighter implementation type if necessary.

Going back to the single class model, we have a data type called "Altitude". This could be defined as the amount of meters in the range 70000..-400 with a precision of .01, let's say. That would be more precise with respect to the application reality. Naturally, a programmer may choose to implement this as a float in C or whatever type is relevant given the target platform.

Again, it comes down to the same principle. The model should express the application's true requirements, as minimally and precisely as possible without telling the programmer how to write code. You want the model to shrinkwrap the application reality as tightly as possible[10].

Even when I allow an attribute to be a loosely defined string such as a name, I might use the data types "Long Name" and "Short Name". The first might be defined as a string up to 80 characters while the other might be up to 10 characters. They could be targeted to the same or different implementation string types.

[10] It is also okay to expand your application reality to allow for future requirements, as prudent given your project reality! If the current application handles fixed wing aircraft only, but helicopters requiring helipads instead of runways will be added in the future, it might make sense to rethink the definition of runway and landing surface. Whether you fatten up the application rules or not, you still need to build a precise model of whatever reality you define. Leaving the model vague is never the right way to accommodate future requirements — it's just plain laziness. (Plane laziness? sorry...)

BEHAVIORAL CONSTRAINTS NOT EXPRESSED AND QUESTIONS UNANSWERED IN THE BAD MODEL

Now let's take a deeper look and consider the model as a whole. What does the Bad Model say about the ATC application behavior and is it correct?

In the Bad Model, the "controller" association is 1:*. The * side permits zero. This glosses over the subtle distinction between an on duty controller with no current Control Zone assignment and an ATC who is off duty.

Moving on to R2 in the Bad Model, we see that it is 0..1:0..1. This noncommittal association covers both off duty controllers who aren't logged in as well as Duty Stations not in use at the moment. So we've lost the constraint that while on duty, an ATC must be logged in.

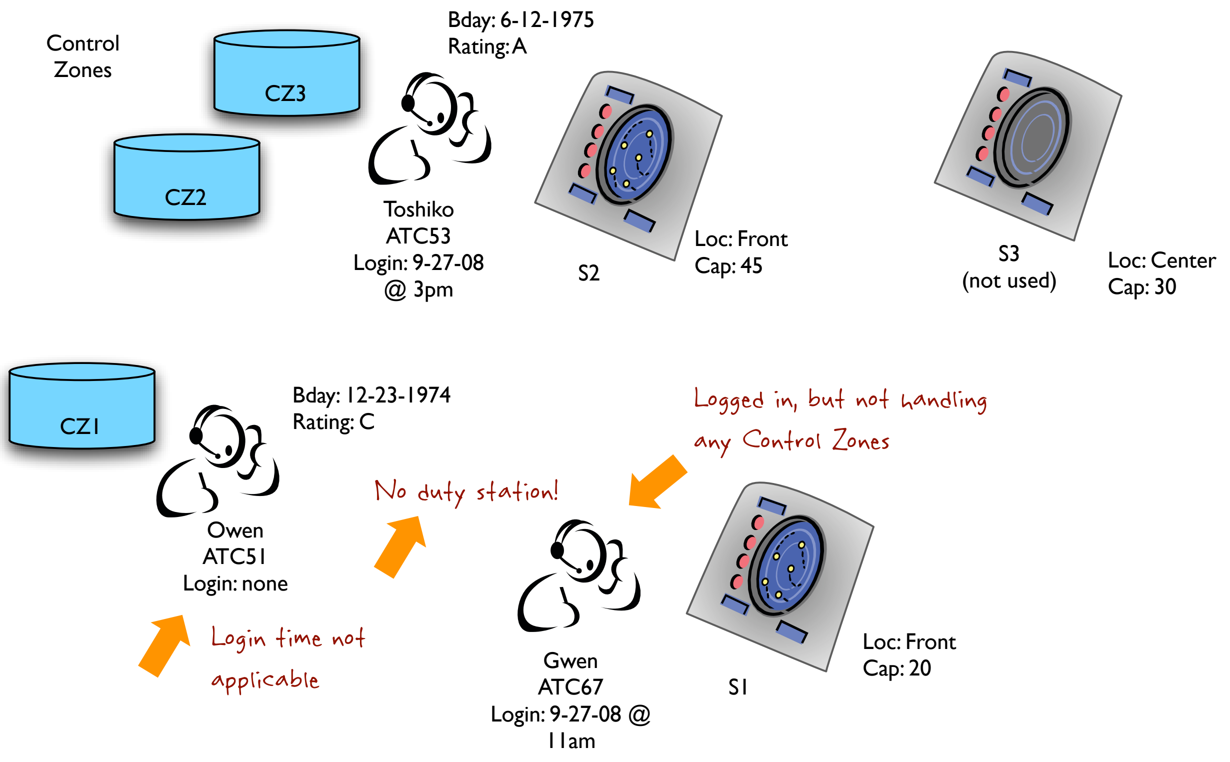
If an ATC is not using a Duty Station, can he or she still be controlling a Control Zone? The model says "yes" since the two associations on the Bad Model are completely independent of one another.

If an ATC is not controlling any Control Zones can he or she still be logged into a Duty Station? Both Good and Bad Models permit it. And, according to the application rules, this is okay.

With the single class "Aircraft" table example, we considered the questions that could be answered by a model. Try asking the Bad Model the question "Which Air Traffic Controllers are off duty right now?" The Bad Model cannot really answer this question. It can tell you which ATCs are not handling any Control Zones. But those ATCs may be on duty. Can you fix it by adding a status attribute to the ATC class in the Bad Model? Sure, but you still have no protection against an ATC with the value "off duty" handling a Control Zone. The status attribute solution also necessitates action language to manage the constraint. That's more code and more testing, all due to one status attribute[11].

[11] A single status attribute now and then is not so bad. But when abused, and it so often is, the resultant complexity in the state and action models can be dramatic.

Here is one example of an illegal scenario allowed in the Bad Model:



Bad Model allows an ATC to work without a Duty Station

The picture above looks okay with the exception of ATC51 (Owen) who is handling Control Zone CZ1, but is not logged into a Duty Station.

ADVANTAGES AND DISADVANTAGES OF THE BAD MODEL

[+] In defense of the Bad Model, it does not *forbid* legal data populations. Data representing an Air Traffic Controller logging in and out of a Duty Station is accommodated. But the same is true for data illegally showing that an on duty Air Traffic Controller is directing traffic in a Control Zone.

[-] The problem is that some illegal configurations are *also* possible. Unless action language and/or code is written to carefully enforce these constraints bugs, will emerge down the road.

[+] Another advantage of the Bad Model is it probably took less time to build.

[-] Sure, less thinking went into it. The time saved will be eaten up writing action language to define and handle the constraints or fixing it in the code. And if any of the constraints slip through those cracks, the time will be spent in testing and debugging.

The purpose of modeling and analysis is to expose and evaluate requirements. We analysts break concepts apart while programmers (or model compilers) compact them down into an efficient ball of code. The class model is a powerful tool when it comes to exposing rules. Even without knowing much about class models, it is not difficult to walk an application expert through the Good Model and get valuable feedback. Walk them through the non-committal Bad Model and you'll just get tepid nods of approval.

The Good Model, on the other hand, might raise questions as to whether or not a Duty Station might be shared and to the specific circumstances where this would occur.

The right tool for the job

To be an effective analyst you want to reach into the UML toolbox and retrieve the best tool for the job at hand. Class models are the most powerful tool for expressing rules and constraints directly in data structure. State models are great for formalizing sequence and synchronization. A state model, for example, is ideal for sequencing the actions required to migrate between on and off duty status[12]. Action language is useful for describing data flow, computation and data access.

By deferring rules and constraints to less effective tools (states and actions), you end up creating more work for yourself. Worse still, neglected rules and constraints tend to get buried in complex actions where they are not as easily seen by application experts, if not entirely forgotten. One of the main reasons to model is to resolve potential flaws early by tweaking a relationship or attribute rather than later by overhauling an entrenched implementation.

Why not just express constraints with OCL?

There is an object constraint language (OCL) included in the UML. Having a language dedicated to constraints is a really good thing, but there are a few downsides.

1) OCL is not supported by many tools
2) OCL can be difficult to read and write
3) A constraint written on the side can be less reliable than one integrated into the data structure
4) Constraints written on the side can result in more code to write and test

Here is an example[1] of the OCL corresponding to the identifier {I} tag:

```
context <class> inv:
    <class>.allinstances()->forAll( p1,p2 |
        p1<> p2 implies
            p1.<identifier> <> p2.<identifier> )
```

It's nice to have both a tag for the class model as well as a detailed expression of the constraint.

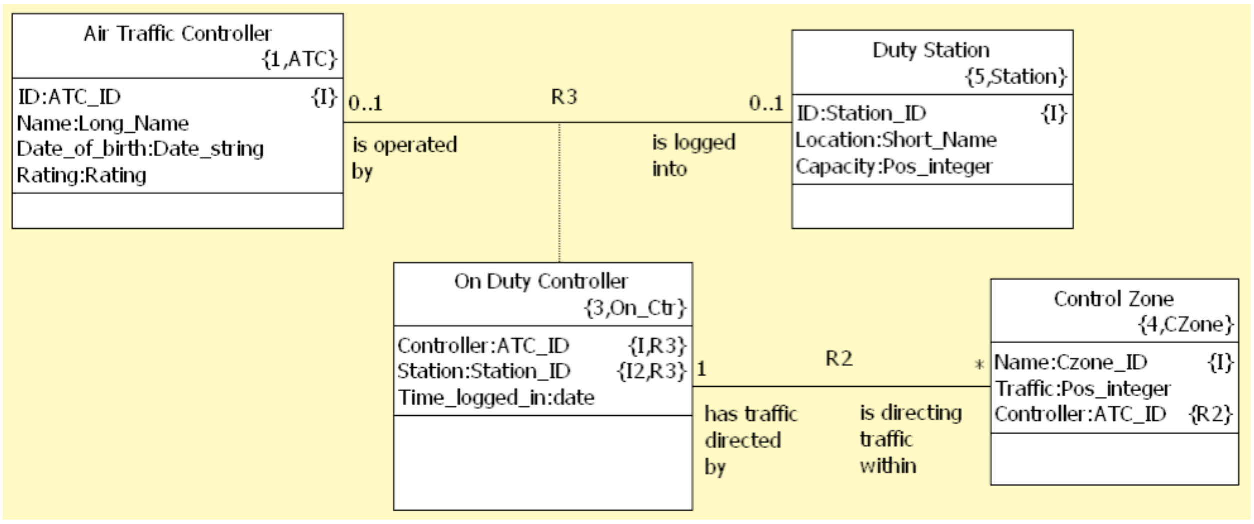
OCL can be useful for expressing constraints not easily shown on a class model, like ensuring that the value assigned to one attribute is less than a maximum value specified by another attribute, for example. So using OCL to fine tune the constraints on an articulate class model can result in a powerful combination.

But there is no better way to enforce constraints and prevent bugs than to define data structures that will not accept bad data in the first place. Building a Bad Model and expecting to express all your constraints in OCL is going to be rather difficult. This is mainly because a Bad Model won't offer the detailed vocabulary (classes, attributes and relationships) to which the OCL would refer!

[12] E-mail a request to me at leon_starr@modelint.com and I can send you the state model and action language for the Good Model. (This is my way of finding out if anyone has actually read this far!)

Making the Good Model more concise

As promised we can condense the Good Model a bit without sacrificing any application rules.



A more concise Good Model

All I've done here is replace the generalization relationship with an association class. Since there are no attributes or interesting behavior on the Off Duty Controller, we can eliminate the class. Now an instance of On Duty Controller is created as a consequence of linking an ATC object to a Duty Station object. Upon unlinking, an ATC from a Duty Station (logging off), the representative On Duty Controller instance disappears along with the Time_logged_in attribute value and any Control Zone links. Note that the multiplicity on R3 is 0..1 on both sides. So, at any given time, an ATC may or may not be logged in. From the perspective of a Duty Station, it may or may not be available.

So why didn't I just do it this way in the first place? There are a couple of potential benefits to the original Good Model. The Off Duty Controller subclass serves as a nice place holder in case we do discover attributes or behavior unique to that role. Also, if we discover an association that applies only to an Off Duty Controller, we would have an anchor for it. Also, the original example is a bit more intuitive since the subclasses mirror the primary states of an ATC. The limited scope of this teaching example favors the association class solution, but real world complexity may necessitate the generalization solution.

Some suggested techniques

Any time you see a relationship between two classes with * (zero or many) or 0..1, you have conditional associations. There's not necessarily anything wrong with that, but it should make you ask whether or not the conditionality is due to changing roles or circumstances. If so, consider specializing or using an association class. Look, especially, for associations and attributes that would be relevant only to certain roles or circumstances.

If you have attributes that can take on "not applicable" values, then you should probably refactor your analysis a bit. Perhaps a generalization or association class may help create a better home (class) for the conditional attribute. Think of a class as "The definition of a set where each member of the set exhibits the same behavior and has the same attributes and associations." So if the set you are defining has any "special" members, refactor! Strict adherence to this principle will often lead to more classes and relationships in your class model. But it stands to reason that a class model with an expanded vocabulary can be more intelligent. And you, yourself, will use this richer vocabulary to phrase more penetrating questions about the application requirements and make the model, and ultimately software, even more intelligent!

For every association use verb phrases instead of role names. Name both sides of every association as precisely as possible. This will force you to consider the multiplicity on each side more carefully and, again, lead you to a clearer understanding about the deeper truths in your application.

Tag all real world identifiers (tail numbers, airport codes, license plates, serial numbers, etc). Look, especially for multi-attribute identifiers (airport + heading + side), (file cabinet name + drawer number) and so forth. Incorporate referential attributes in these identifiers. You will be surprised how many important constraints can be discovered.

Sketch non-UML scenarios with specific instances and data values corresponding to the classes, attributes and relationships in your UML model. Use these to solicit feedback from the application experts and to explore boundary conditions.

Summary

The value of creating UML class models depends greatly on what kinds of models are being built. Imprecise class models that do not express key constraints are often the result of critical misconceptions about the purpose of class modeling. One of these is that class models are static and therefore do not play a role in defining overall system behavior. Another is that analysis, design and implementation goals can somehow be reconciled within the same model. You can, of course, just use UML to draw pictures of your code, but, if you're going to do that, why not save time and just write the code?

The good/bad model comparison was intended to illustrate some ways in which a good class model can express detailed application rules and shape and constrain overall system behavior. If-then logic and other behavioral switches can often be embodied directly in data structure. This reduces the complexity of the statecharts and actions and leads to less code and testing.

Rather than think of a UML class in implementation terms, like a Java or C++ class, a UML class is defined, for analysis purposes, as a set of things with the same characteristics, behavior and relationships formalized in a relational table structure. This table structure may be implemented in a wide range of platform data structures not necessarily resembling tables.

This frees the analysis from platform specific considerations. The analyst is free to focus on the application rules that must be enforced in any implementation. The result is an application rule base that remains stable as the underlying platform evolves. Another benefit is that the application models can be retargeted to migrating or spinoff products that deploy the same rules on different technologies.

There are a number of analysis techniques that lead to useful, rule expressive class models. One is to balance abstract modeling with concrete scenarios using detailed, non-UML illustrations. Another is to use verb phrases to name all association sides articulately as a way of obtaining precise multiplicities.

This approach should yield practical and compelling benefits from UML class modeling.

Resources

The ATC example, while only shown here as a class model, is available with states and actions in a fully executable form if you have either the Mentor Graphics BridgePoint or Kennedy Carter iUML tools. If you would like a copy, please send me an e-mail. If you don't have either of these tools available, I can send you PDFs and/or text files with the complete set. Feel free to use my ATC example for commercial or non-commercial purposes as long as you retain the copyright information and credit the author.

My latest writings, including book excerpts, can be found at my Google Knol page. If the link doesn't work, please go to knol.google.com and search on my name.

[1] Executable UML, A Foundation for Model Driven Architecture, Mellor-Balcer, Addison-Wesley, 2002, ISBN 0-201-74804-5

[2] Executable UML, How to Build Class Models, Leon Starr, Prentice-Hall, ISBN 0-13-067479-6

[3] Model Driven Architecture with Executable UML, Raistrick, et al, Cambridge University Press, ISBN 0-521-53771-1

[4] Databases, Types and the Relational Model, C.J. Date - Hugh Darwen, Addison-Wesley, ISBN 0-321-39942-0