Subtyping Best Practices

Tips and Traps for Data Modelers

David A. Ruble

Principal Senior Partner

Olympic Consulting Group
Software Architecture and Development

P.O. Box 4008 – Federal Way, WA 98063
(253) 946-2690
www.ocgworld.com

Subtyping is a powerful technique used by data modelers and class modelers to portray generalization / specialization relationships. It is also a technique that can be highly abused. If employed improperly, it can twist a good data model into an unusable mess.

In this white paper, OCG's David Ruble offers some tips for the proper use of subtyping that will improve your model, and help you avoid some of the most common subtyping traps.

## Subtyping Best Practices – Tips and Traps for Data Modelers

Purpose

One of the most powerful techniques available for data modelers is the supertype/subtype relationship, yet is one of the more oft-abused practices. The purpose of this paper is to highlight several instances of improper use of subtyping, and to suggest alternate patterns that are more robust.

Overview of Supertyping / Subtyping

The Supertype/Subtype relationship is an important tool in software engineering. It distills common characteristics and behavior that pertain to all members of a group, and isolates characteristics and behavior that are particular to members of a group's subgroups. At design time in object-oriented systems, this will lead very directly - with few modifications - to an object-oriented class-inheritance hierarchy with high stability and reusability across various applications. In relational database designs, it gives the designer choices to retain the subtypes and supertype, collapse them all into the supertype, or implement only subtypes.
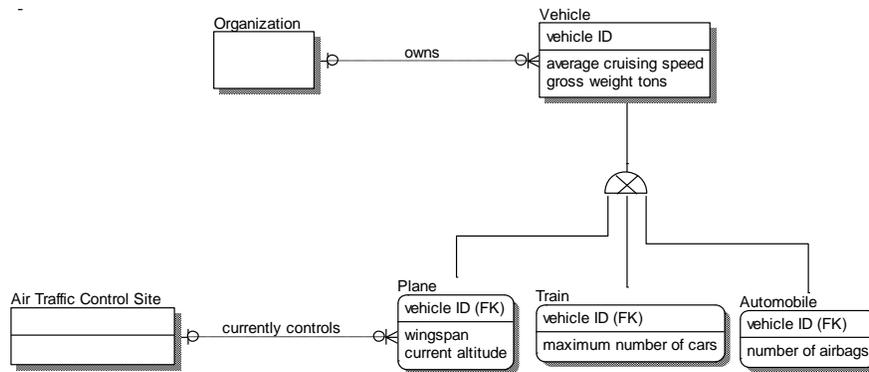
Supertype / Subtype example

In the real world, many objects belong to a similar class, but themselves have divergent characteristics and/or behavior. *Planes*, *Trains* and *Automobiles* are all examples of the class, *Vehicle*. One may have a valid business reason to refer to the collective fleet of a company's vehicles, and therefore it is useful to create a supertype entity: *Vehicle*, which can hold the attributes common to all members of its class, and also to participate in relationships valid for all members of its class. In the following figure, we see that the entity *Vehicle* has a set of attributes common to all vehicles, and that all vehicles of interest to the business can participate in relationship that defines the owning *Organization*.
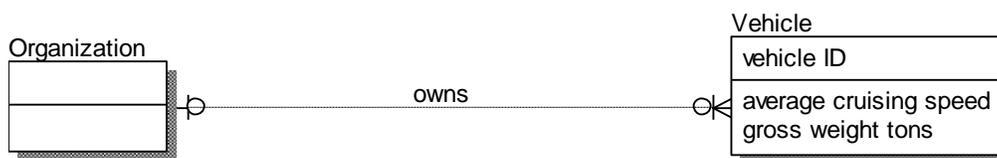
The next diagram reveals the power of the subtype relationship. In this diagram the subtypes of *Vehicle*, *Plane*, *Train* and *Automobile*; have been broken out as separate entities, attached to their supertype. The "X" in the subtyping symbol denoting that the relationship is exclusive – meaning any instance of vehicle can be only a plane or a train or an automobile.[1]

---

[1] Obviously, *Chitty Chitty Bang Bang* would violate this rule, but one could remove the exclusivity if modeling for Disney.

Organization — owns — Vehicle

Vehicle
vehicle ID
average cruising speed
gross weight tons

Air Traffic Control Site — currently controls —

Plane
vehicle ID (FK)
wingspan
current altitude

Train
vehicle ID (FK)
maximum number of cars
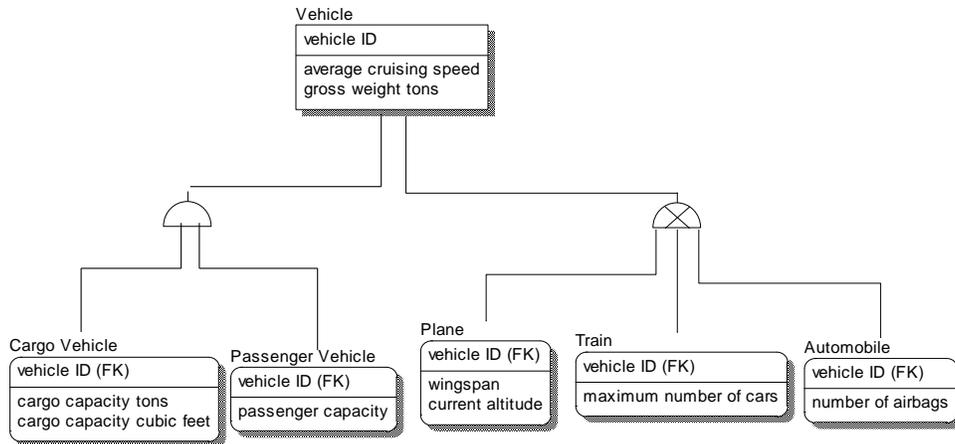
Automobile
vehicle ID (FK)
number of airbags

The subtyping on the diagram exposes several valuable business rules which would otherwise be buried in some dark corner of a dense written specification. First, we can see that each of the subtypes have attributes which describe characteristics unique to the subtype. For example, it is only desirable that planes have a "current altitude" above the ground. Similarly, it is interesting for the business to know how many airbags are in a given automobile. (I refer to the automobile's equipment, not its current population of consultants.) While subtyping to distinguish characteristics is a fine and illuminating modeling practice, it is even more revealing when subtypes are discovered to participate in relationships that are exclusive to the subtype. In the prior example, only planes can be under the current control of an *Air Traffic Control Site*.

If the reader will indulge me just a bit further, let's complicate this seemingly simplistic example before we move into the labyrinth of subtyping more esoteric entities such as *Party*.
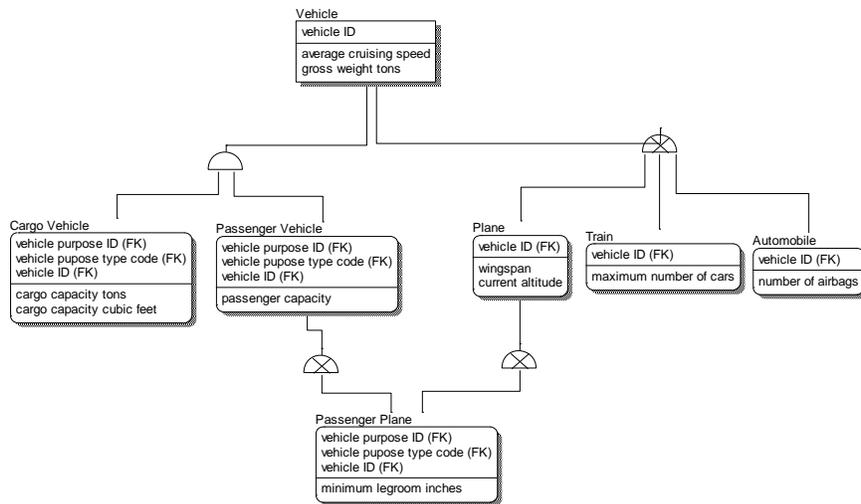
Getting ugly with multiple subtypes

It's fairly common that one finds *multiple* ways to dice and slice the supertype. What's an analyst to do? Some shops allow their analysts to adorn their models with any number of competing subtype discriminators at the same level – leaving it for the design team to sort out later. Some analysts, (including present company), recall being roughed up in the parking garage after hours by an unruly mob of programmers, and therefore try to resolve competing subtyping strategies before the project starts to run out of time and money.

Consider once again, our *Vehicle* model. In the following example, the modeler has discovered that the business has a need to distinguish between passenger vehicles and those which haul cargo – and that some vehicles are licensed to do both.

Vehicle
vehicle ID
average cruising speed
gross weight tons

Cargo Vehicle
vehicle ID (FK)
cargo capacity tons
cargo capacity cubic feet

Passenger Vehicle
vehicle ID (FK)
passenger capacity

Plane
vehicle ID (FK)
wingspan
current altitude

Train
vehicle ID (FK)
maximum number of cars

Automobile
vehicle ID (FK)
number of airbags

Now let's see how much of a mess we can make of this model. Suppose we uncover the requirement to record the minimum legroom available on passenger planes. The following model will make most data modelers and even the most strident object-oriented class modelers reach for their revolvers:

Vehicle
vehicle ID
average cruising speed
gross weight tons

Cargo Vehicle
vehicle purpose ID (FK)
vehicle pupose type code (FK)
vehicle ID (FK)
cargo capacity tons
cargo capacity cubic feet

Passenger Vehicle
vehicle purpose ID (FK)
vehicle pupose type code (FK)
vehicle ID (FK)
passenger capacity

Plane
vehicle ID (FK)
wingspan
current altitude

Train
vehicle ID (FK)
maximum number of cars

Automobile
vehicle ID (FK)
number of airbags

Passenger Plane
vehicle purpose ID (FK)
vehicle pupose type code (FK)
vehicle ID (FK)
minimum legroom inches

Here, we've run smack-dab into the specter of multiple inheritance. Multiple inheritance occurs when a subtype has more than one possible supertype. In essence, the model says that a *Passenger Plane* inherits all of the characteristics of both the *Plane* and the *Passenger Vehicle*. As clever humans, we are still able to wrap our minds around this concept, but most programming languages or database management systems simply throw up their electronic hands when this situation is encountered.[2]

In this example, the subtyping is analytically accurate, but it will require rework at design time to implement a solution. It would be preferable to find an equally analytically accurate expression of the business rules that requires less work at design time.

---

[2] The C++ language allows for multiple inheritance but it is often considered dangerous territory by some programming standards.

Subtype discriminators

The answer to cleaning up our subtyping example lies in the subtype discriminator. The subtype discriminator is the reason or discriminating factor that causes an instance of the supertype to be a member of one subtype group versus another.

In our *Vehicle* example, our first instinct was to declare subtypes that were "a kind of" vehicle. Subtyping by intrinsic property is a particularly strong practice because a member of one subtype is unlikely to migrate to become a member of another. A *Vehicle* that starts life as an *Automobile*, will go to the wrecking yard as an *Automobile*.

In our second attempt to subtype *Vehicle*, the subtyping discriminator was not an intrinsic property of the *Vehicle*; rather it was the "purpose" for which the vehicle is used. Essentially, it is the "role" the vehicle plays in the company. Additionally, a vehicle can be repurposed from one to the other throughout its useful life, and could actually be licensed to haul passengers and cargo at the same time. If our objective is to avoid multiple subtype discriminators at the same level, then we need to look for a different strategy for modeling the vehicle's various purposes.

An alternative strategy for modeling role

One strategy for avoiding multiple subtyping is to create an entity that represents the real-world item's unique characteristics that are pertinent to the *roles* it plays. In the following model, an entity called *Vehicle Purpose* has been created, and it has been subtyped to hold the facts gathered when a vehicle plays the role of *Passenger Vehicle* or *Cargo Vehicle*.
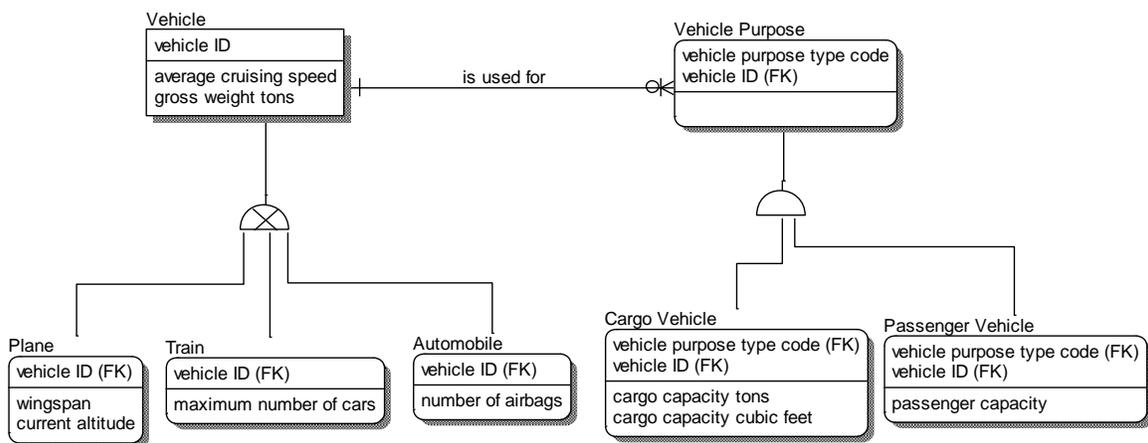
Notice that this expression is as analytically correct as the previous example. A *Vehicle* can be used for one or more *Vehicle Purposes* – and a *Vehicle Purpose* can be *Cargo Vehicle*, *Passenger Vehicle*. The lack of an "x" in the subtyping symbol tells us that a vehicle's purpose can be both cargo and passenger at the same time. A single instance of *Vehicle* can therefore have up to two instances of *Vehicle Purpose*, but only one instance per *Vehicle Type*. By subtyping *Vehicle Purpose*, we can easily demonstrate that the additional attributes of a *Cargo Vehicle* are different than those of a *Passenger Vehicle*. Furthermore, the subtypes may participate in relationships that are unique to passenger or cargo vehicles. Designers will note that this expression of the

business requirements can be directly implemented with little or no rework, and does not suffer from the perils of multiple inheritance. Database designers still are allowed the choice of whether to implement *Vehicle Purpose* as one, two or three tables, or even roll it back into Vehicle as a set of optional fields.

Party vs. Party Role

Now let's try out these concepts with something more complex – the *Party* model.

Party is at the core of enterprise data models that have the need to recognize that the people and organizations of interest to the business play multiple roles – but remain the same individuals and organizations. The *Party* concept has been growing in popularity as businesses attempt to get a complete view of their customers, and fully understand their total involvement with the enterprise.

In most traditional systems, people and organizations were modeled separately, according to the role they played. Customers where physically represented in the order entry, billing and A/R *Customer* tables. *Vendors* took up residence in the A/P system, *Employees* in the HR systems, and so on. Many of the data integration challenges of the last decade were to consolidate competing Customer, HR and Accounting systems within an enterprise and at minimum, try to whittle the redundant representations of these roles down one system of record. In many companies, scant attention has been paid to consolidating a total view of all parties of interest to the enterprise.

There are some industries and areas of public service that have had pressing needs to understand a party's total involvement, and have thus pioneered the concept of *Party* modeling. For example, public health systems for epidemiology and contagious disease investigation have the need to know that a patient on one case is a family member or contact person on other cases. Court systems have had a need to know that a defendant in one case was a co-defendant or witness in another. Even in business systems, there have been examples of companies paying large sums of money to their vendors, unaware that the same vendor was delinquent on their payments as a customer. The inability to cross check A/P with A/R has been costly for many firms.

The impulse to subtype *Party* by the roles they play is very strong. We are so engrained to think of parties in terms of *Customer*, *Vendor*, *Employee, and Agent* that our first instinct may be to use *role* as the subtyping discriminator for *Party*.

While one quadrant of the analyst's brain is busily subtyping *Party* by role, another section of gray matter immediately notices that parties darken our door in the form of *Persons* and *Organizations*. In the following diagram, we see that *Party* has been subtyped into *Person* and *Organization*, and in a separate structure, into *Customer*, *Vendor* and *Employee*. The notation indicates that a Party is either a *Person* or *Organization*, but not both. Conversely, a *Party* can be *Customer* and also a *Vendor* and also an *Employee*. As Oliver Hardy was fond of saying, "Another fine mess you've got us into this time, Stanley." We are once again confronted with multiple subtyping at the same level.
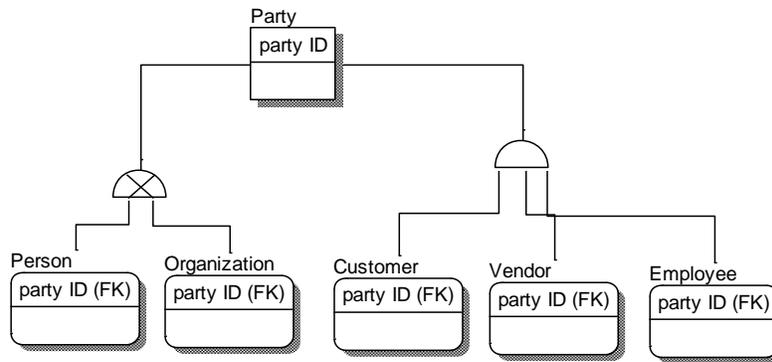
If we hearken back a few pages to our *Vehicle* example, we can employ the same guiding principles to help us avoid multiple subtype discriminators for *Party*. Which of these subtyping discriminators is the stronger case for subtyping *Party*? Our heuristic for subtyping a real world thing by its intrinsic properties leads to select Person/Organization as the stronger candidate.

A *Person* is born a person and dies a person. Likewise, *Organizations* do not morph into becoming people at any point in their existence. *Customer*, *Vendor* and *Employee*, on the other hand, are roles played by parties under specific circumstances.

Just like we did with the purpose of the vehicle, we can promote *Party Role* to become an entity in its own right, and separate it from the intrinsic properties of the parties themselves.
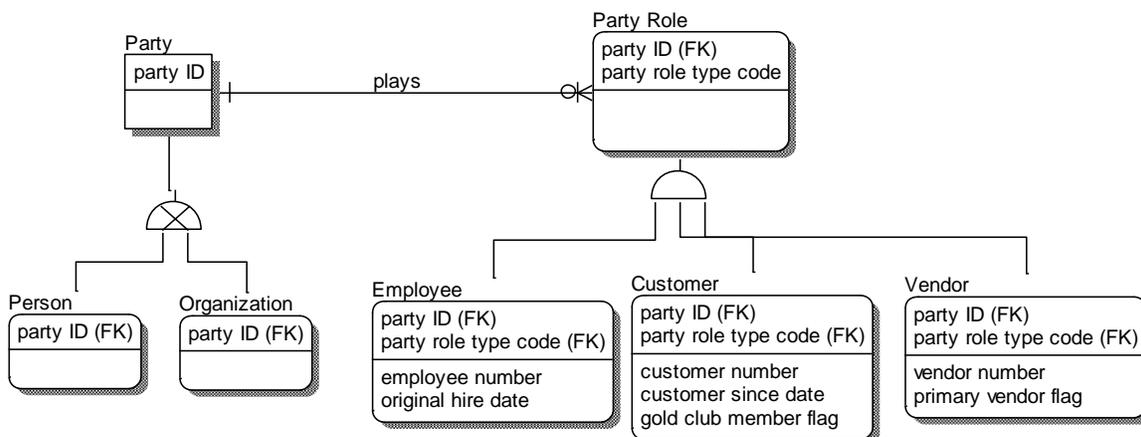
The way we read the modified diagram (above), is that a *Party* can play many roles in the organization, and that the roles one plays may be any combination of *Employee*, *Customer* and/or *Vendor* (by virtue of the inclusive symbol). We can now add the attributes that are particular to *Employee*, *Customer* and *Vendor* to their respective entities, and the subtypes are free to participate in relationships that are exclusive to instances of their respective entity types. Like our solution for *Vehicle*, this model can be implemented with little design rework.

While more expressive, this model isn't perfect. The party role type code is necessary at the supertype level to indicate that although a *Party* can have multiple *Party Roles*, they can only have one per subtype. There is no need for the *party role type code* at the subtype level since the existence of the subtype entity is sufficient; however the CASE tool includes it as part of the

---

key. It is also desirable to mark *employee number*, *customer number* and *vendor number* as alternate primary keys of their respective subtypes. In all likelihood, the database designers will implement these subtype entities as separate tables, and carry the *party ID* as a unique foreign key.

The perils of subtyping by state

Another subtyping practice that comes to grief at design time is subtyping by the status of the supertype. The status of an entity tracks its progress as an instance of the entity migrates through the various phases of its lifecycle.

Status is particularly important entities such as *Customer Order* and *Customer Account*. The information collected about an *Order* or *Account* can vary tremendously depending on the Account's state – making it very tempting to trot out our old friend, subtyping, however, modeling state using subtyping can be tricky.

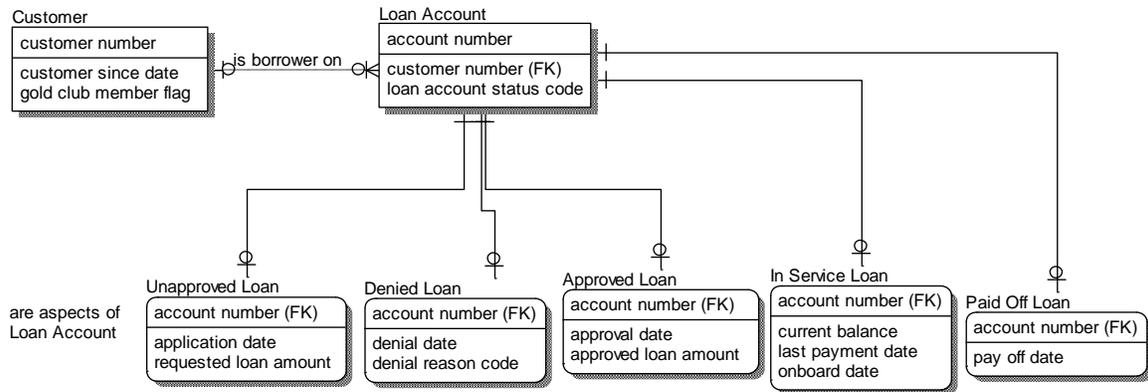Consider the following model of a *Loan Account*:
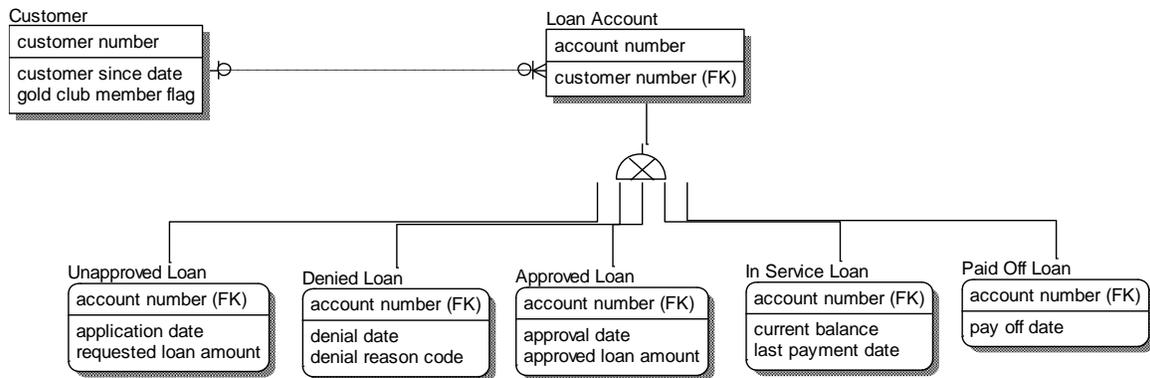
In this model, the analyst has subtyped *Loan Account* into the various states that the *Loan Account* passes during its lifetime. The analyst is attempting to show that some attributes are collected during specific states, and not during others. The analyst's instinct is right on, but the execution of the model has a problem.

A subtype inherits the key of the supertype. In this case, the *account number* is the key of *Loan Account*, and it remains the key of a *Loan Account* throughout the life of the loan. The exclusive nature of the subtyping relationship informs us that if a *Loan Account* is a member of one subtype, it cannot be a member of any other subtype at the same time.
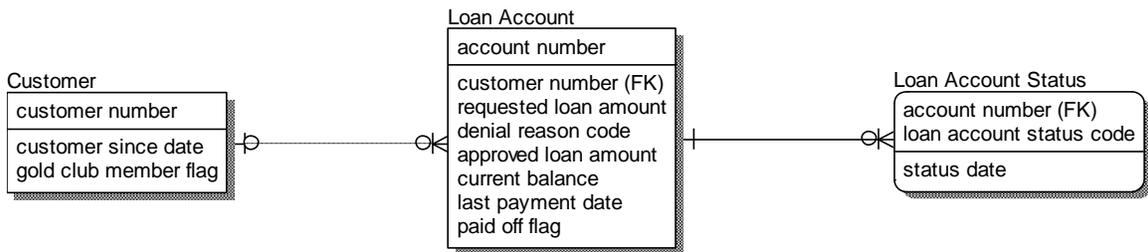
Imagine an instance of loan moving along the line as it changes from state to state. In a physical sense, when a loan goes from being an instance of an *Unapproved Loan* to an instance of an *Approved Loan*, the model implies it must re-instantiate itself as the new subtype, and at the same time kill itself off as an *Unapproved Loan*. This form of object-oriented hari-kari due to subclass migration is not catered to by most environments. Now, this wouldn't be a problem in the logical model if we only needed to keep a snapshot of the loan in its current state. However, due to the

fact that we need to keep an accumulated history of loan attributes, this becomes unacceptable. Subtyping is simply the wrong technique for this problem.

A solution to this modelling problem is to treat the various states of *Loan Account* as aspects of a *Loan Account* rather than as subtypes. This is handled somewhat well in UML by using the aggregation symbol, but Erwin offers no equivalent. In the diagram, below, we see that a Loan Account can have various aspects that are collected over time. The change in *loan account status code* causes the creation of a new aspect.

Customer
customer number

customer since date
gold club member flag

—is borrower on—

Loan Account
account number

customer number (FK)
loan account status code

are aspects of
Loan Account

Unapproved Loan
account number (FK)

application date
requested loan amount

Denied Loan
account number (FK)

denial date
denial reason code

Approved Loan
account number (FK)

approval date
approved loan amount

In Service Loan
account number (FK)

current balance
last payment date
onboard date

Paid Off Loan
account number (FK)

pay off date

It looks an awful lot like subtyping, but it's not. We seem to be stretching the limits of a static data modeling notation. Many modelers choose to collapse all of the attributes back into *Loan Account*, and employ a different model to convey which attributes are created or updated in various states of the loan. In the following diagram, all attributes are shoved back into *Loan Account*, and *Loan Account Status* has been created to hold the status value and the status date.

Customer
customer number

customer since date
gold club member flag

Loan Account
account number

customer number (FK)
requested loan amount
denial reason code
approved loan amount
current balance
last payment date
paid off flag

Loan Account Status
account number (FK)
loan account status code

status date

The snippet of a model, shown above, is highly simplified from a real *Loan Account*. On a real model, we will probably find that certain aspects of the loan, such as the *Loan Application* and perhaps *Loan Actions*, such as approval or denial, might warrant separate entities.

Entity Lifecycle Matrices

An entity lifecycle matrix is a more robust model for showing how attributes are added over time as the entity passes through its respective states. In the simple matrix, shown below, we can see the attributes listed on the *y* axis, and the states of the entity listed on the *x* axis. In each cell, one can denote whether the attribute's value is created or updated while in that state. Matrices such as this one can also be used to show changes in optionality.

Entity Lifecycle Matrix: *Loan Account*

	Unapproved	Denied	Approved	In Service	Paid Off
Account number	C				
Requested loan amount	C				
Denial reason code		C			
Approved loan amount			C		
Current balance			C	U	U
Last payment date				U	U
Paid off flag					C

State-transition Models

In addition to this type of matrix, state transition models are another tool that the analyst can use to show which business events or processes kick the entity from one state to the next. By showing all of the legal transitions, the illegal transitions can be discerned by virtue of their omission.

Loan Account.Loan Account Status
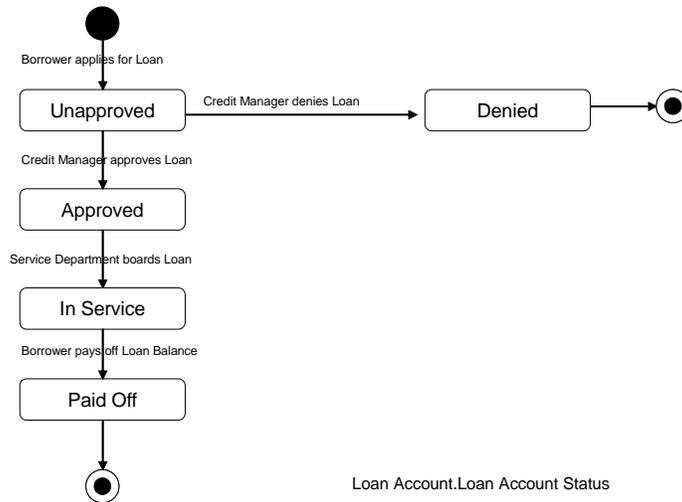
In the state-transition diagram, above, we can see the potential paths that a loan can take as it moves through its lifecycle. This type of model is very easy to construct and very easy to read. A bank employee looking at this model may immediately ask, "What happens if the borrower withdraws their application?" Hence, we have spotted a hole in our analysis that may not have been obvious by looking at an entity-relationship diagram.

Summary

In this paper, I have exposed two common misapplications of subtyping. The first deals with multiple subtype discriminators at the same level – which is analytically accurate, but causes nightmares for designers. An equally analytically accurate technique is to use regular

associations to separate roles an entity plays from the actual entity itself. This causes less grief downstream.

Another common misapplication of subtyping is to subtype by state. Subtyping by state is really only useful if the application has no responsibility of remembering anything about the entity aside from the characteristics of its current state. That is rarely the case in business systems. Therefore, subtyping is the wrong technique, rather the entity is an aggregation of the various aspects of the entity that are built up over time as it passes through its lifecycle. This dynamic accretion of information is difficult to depict on a static entity-relationship model, and the analyst may find it more illuminating to use an Entity-Lifecycle Matrix along with a State-Transition model.

About the author

David Ruble, a principal senior partner at Olympic Consulting Group (www.ocgworld.com), is a senior analyst, designer, author and educator. He is widely regarded as an expert in the field of business analysis, information modeling, GUI design and functional specification. He has been a principal analyst and designer of many mission-critical global corporate information systems – linking suppliers and customers worldwide. David also has significant experience designing applications in the transportation, health care and public safety sectors. His background in business, technology and art create a unique skill set that allows David to communicate with ease among business people, technologists and graphic designers.

As an educator, he has taught software engineering techniques to hundreds of students throughout the United States. He is the author of Practical Analysis & Design for Client/Server & GUI Systems, published by Prentice-Hall, 1997. His book has been used widely in colleges and universities throughout the United States and Thailand, Mexico and Argentina and is considered a timeless classic in the field of business analysis.

About Olympic Consulting Group

Olympic Consulting Group (OCG) is a full-service system architecture and development firm serving the Puget Sound region since 1997. The firm specializes in delivering high-performance consulting in the analysis, design, development and project management for complex business systems and government agencies. www.ocgworld.com

Selected Bibliography

Page-Jones, Meilir. *Role Modeling*. PowerPoint Presentation, Bellevue, WA: Wayland Systems, Inc., 2000

Ruble, David A. *Practical Analysis & Design for Client/Server & GUI Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1997