A Decade of Modeling Financial Vehicles

William F. Frank
Chief Scientist, Financial Systems Architects
wff@fsarch.com

Anil Karunaratne
Lead Architect, Markets Technology, J.P. Morgan
anilk_bgold@hotmail.com

1

Anil is not here. Too bad to loose the contrast. My brother calls him a "tea ceremony of the mind". I am more like a food fight of the mind. You can just hope they aren't serving lasagna today.

What is a Financial Vehicle?

Consider this Yacht.

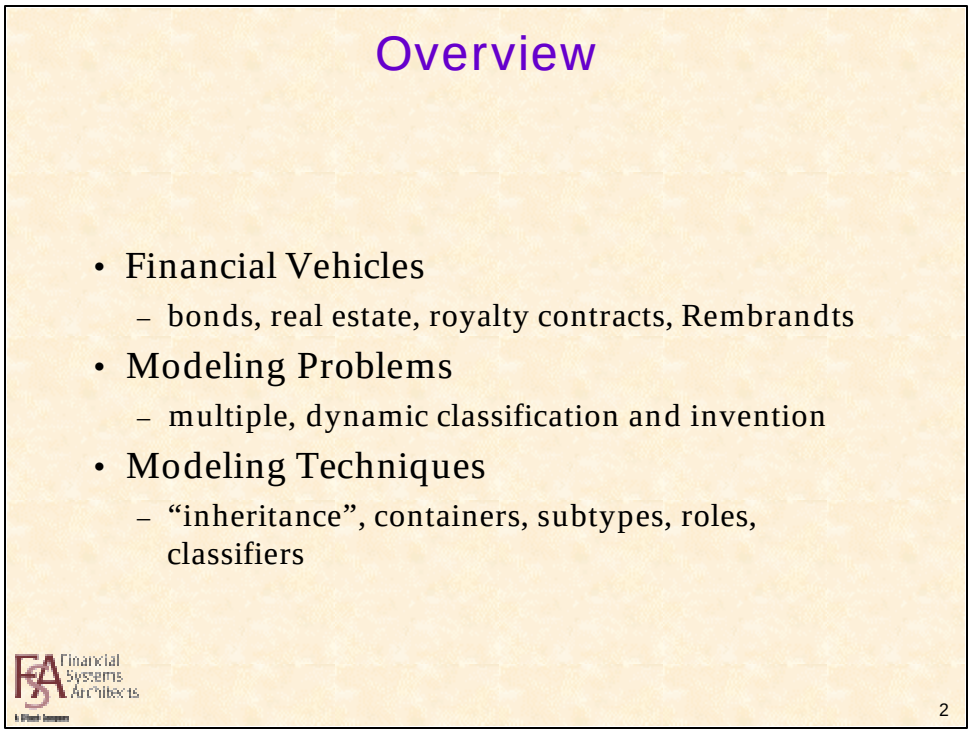
As a financial vehicle, it is in the same category as:

A Professional Basketball Contract
The World Trade Center in the distance
IBM common Stock

It is probably NOT in the same category as

A rowboat
An aircraft carrier

But, if you were to model it in the obvious way, you would more likely create a model in which your yacht object was an instance of some subclass of boat.



Our problem is that Financial Vehicle is such an

ABSTRACT CONCEPT,

And therefore

SO PLASTIC

It does not fit easy solutions, especially solutions that have to span multiple applications, business units, and endure over time.

We have used a variety of techniques, rejected some, but mostly simply, as each new technique proved useful, ADDED it to the ones already used in the model.

Some of The Projects

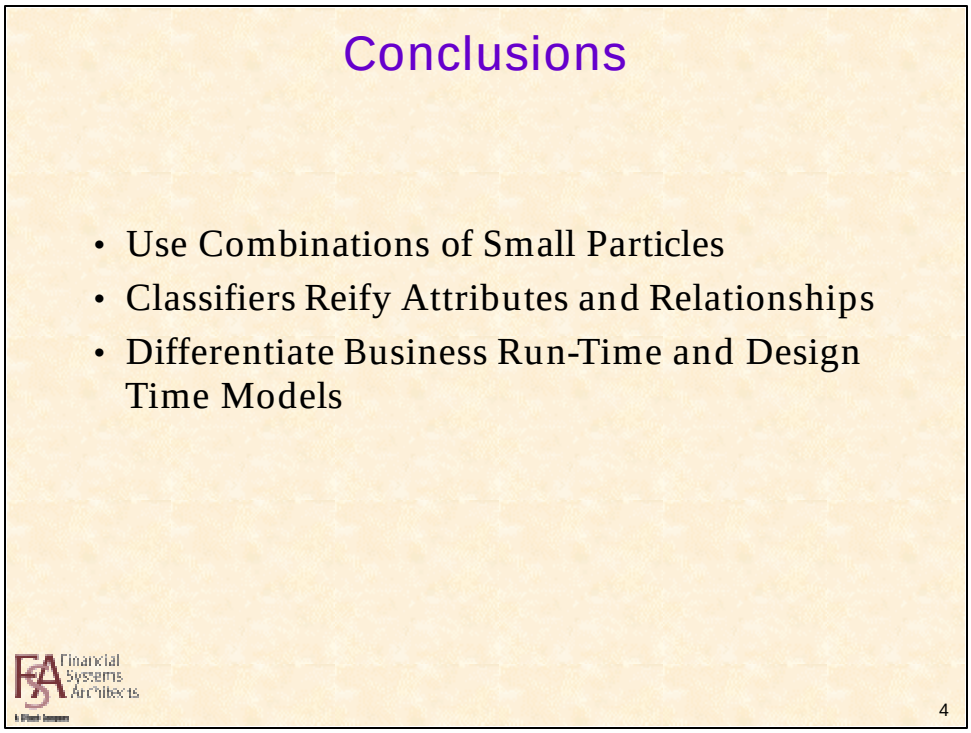
- Electronic Joint Venture – Bond Evaluator
- Citibank Enterprise Business Model, Foreign Exchange
- Fidelity Enterprise Architecture, TradeOrder Management
- CAD Portfolio Recordkeeper

Financial Systems Architects

3

These projects cover more than 10 years of work, including

narrowly focused implementations,
enterprise models,
implementations based on the enterprise models
domain models of economic sectors
products based on the domain models



We modeled the frequent creation of new vehicle types using recombinant particles

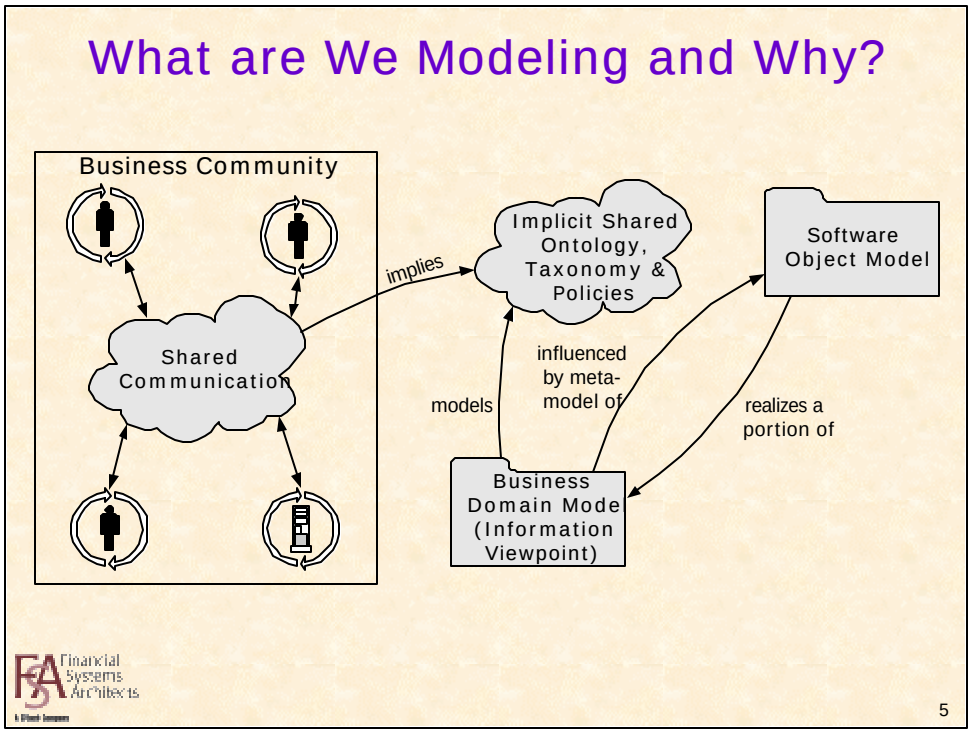
We had to treat the attributes and relationships between objects as objects in their own right

This finally let us to multiple business domains, and multiple software systems to support them, one for designing financial vehicles, a separate one for using them.

The KEY CONFUSION that held us back is implicit in most O-O thought and in UML: the overloading of the term "class" to mean indiscriminantly:

1. a **template** from which a running program can create a software object
2. a **type**, used by people to classify things for some purpose.
3. the **set** of instances that *satisfy* a type**, or** are *created from* a template.
4. a **fundamental type** we use to organize our knowledge (specializing 3).

These differences may seem subtle, but then understanding and modeling *requires* some subtlety. And I can attest that slipping from one to the other without realizing it, while trying to construct a model, makes you very confused. I believe this confusion has cost the world billions of dollars.



This figure is a RATIONAL RECONSTRUCTION. We did not understand our work this way until very recently. (It is also incomplete, since in the end, the software itself is or becomes part of the business community.)

A model must be a model OF something. Weather forcasters have weather models, models OF the weather. Physicists have atomic models.

The expressions

"analysis model" and
"design model"

used by most O-O "methodologies", can, I believe, only be used by people comfortable talking about the emperor's new clothes. (For almost all these 10 years, I pretended to myself that I understood this talk. Anil never pretended.)

If, instead, you consider a model in terms of WHAT you are trying to model,

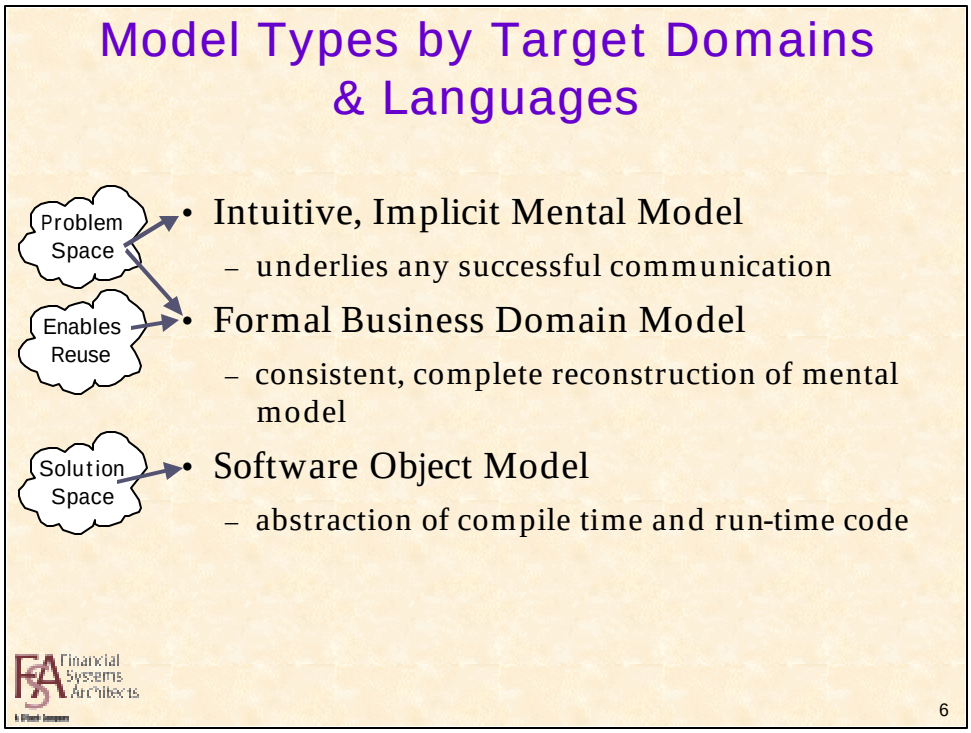
You might have models like the following:

A model of the software you are building

A model of the business your system is supporting, including the role of the system (hardware AND software) in the business

A model of the business domain in which the business operates

A model of the concepts implicit in the discourse of your users



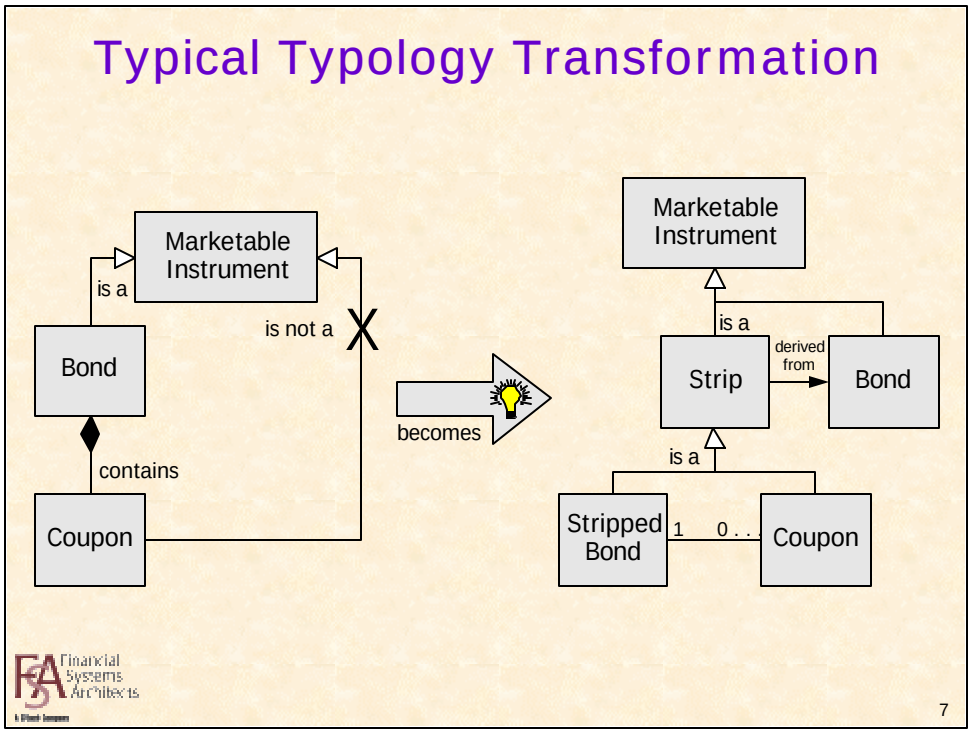
Of course, all these models are *related* to each other. Because they are models of different things that are also related to each other.

The job of the engineer is to create a software model that *corresponds* to some other related model in a useful way. There is no crank you can turn (no "methodology") to go from one model to another. People know and accept this about other fields. There is no "methodology" for going from a traffic flow model to the design for a bridge. It takes a great engineer. Even though the bridge must satisfy the traffic flow requirements.

The MYTH OF METHODOLOGY is the belief that there is some set of rules for looking at a business and deciding what your software objects have to be. *Especially* the rule, "find a category name, probably define a software class".

The need for multiple, independent but related models also means that software models do not really get built from business analysis by "refinement". They get built by transformation and creation. And if you don't think deeply enough, you get a software model that breaks easily.

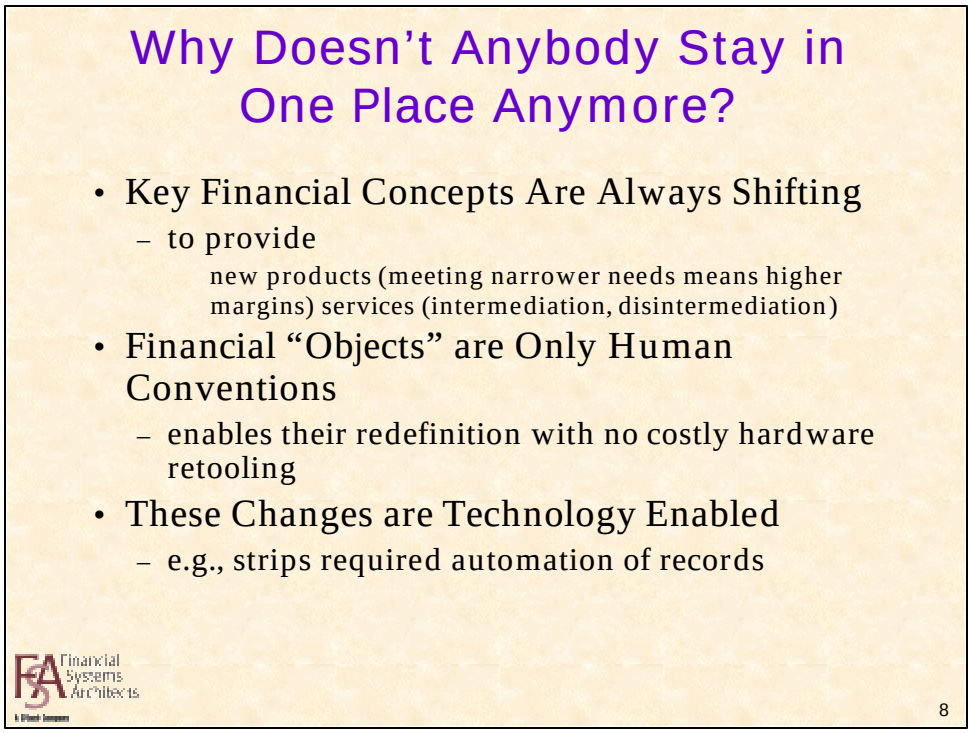
The basic ideas of model theory are applied to system modeling in RM-ODP (references later). Discussed in Jackson & Zave, "Where Do Operations Come From" (IEEE Transactions on Software Engineering 1996 p 508).



Look, for example, at this figure as a software model that is very close to what you might be told by bond traders, as the types of things they were concerned with.

Pre 1980, they would have told you a story that looked like the structure on the left. But as soon as Merrill Lynch got the bright idea of cutting the coupons off the bonds and selling them separately, you might have drawn a model like the one on the right, and your pre-1980 software would not work for the new instruments.

The main reason that the concept of a strip became practical was an increase in the level of automation for tracking holdings in custody.



Financial "products" are only contracts between people, easier to change than physical things,

And more products means meeting the more precise needs of clients, for more profitable deals.

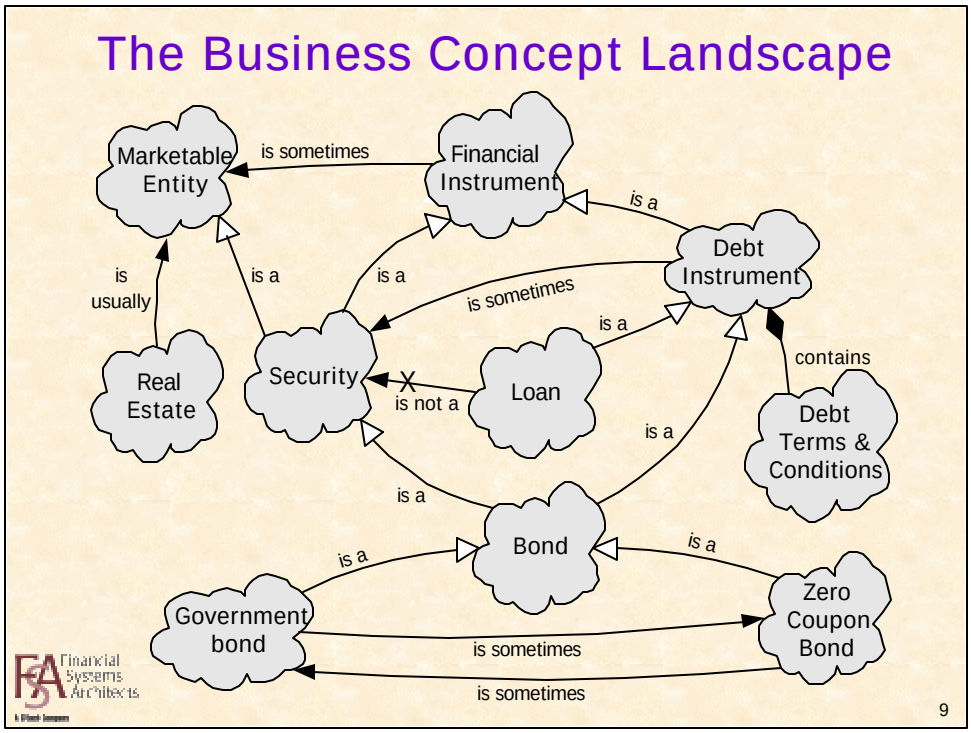
So,

Faster and FASTER

because of technology

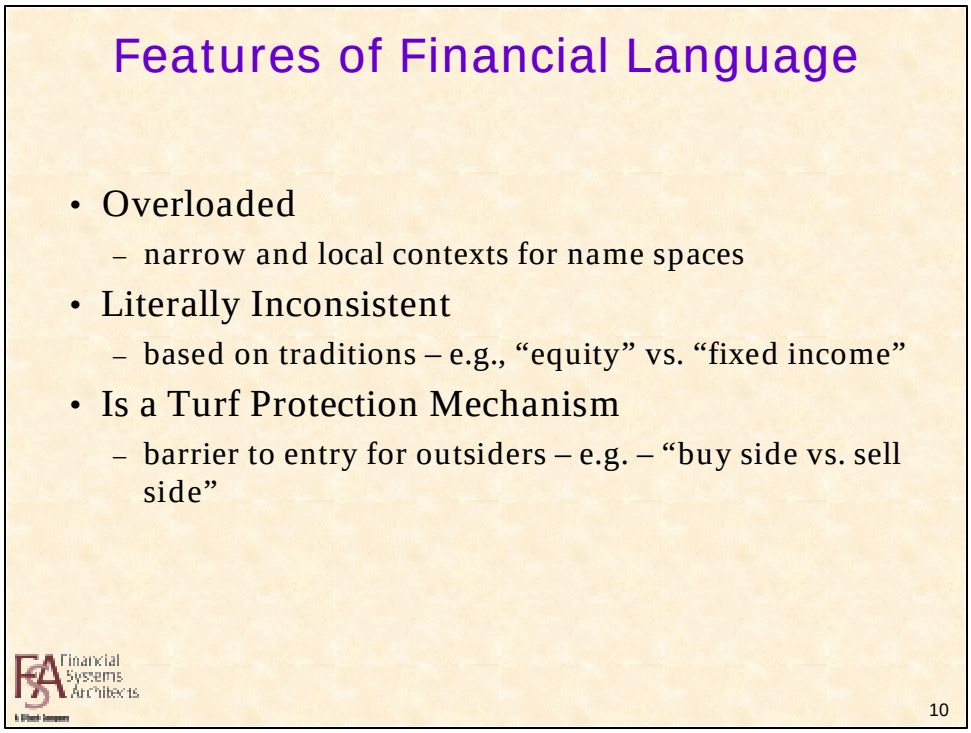
the close-to-the-surface mental models of financial people are changing.

What is required is what linguists call a "DEEP STRUCTURE", that provides some stable foundation underlying all this change. (Bach and Harms, Universals in Linguistic Theory). Jim Caldarella, a senior Citigroup technology officer, proposed to us the idea of creating a model of the "conceptual building blocks" for a bank in 1989, which he called a "business elements analysis", and later called these things "business objects". (Of course, that term has taken on many other meanings since.)



Let us go back and look at the INTUITIVE model as directly expressed in the language of financial people.

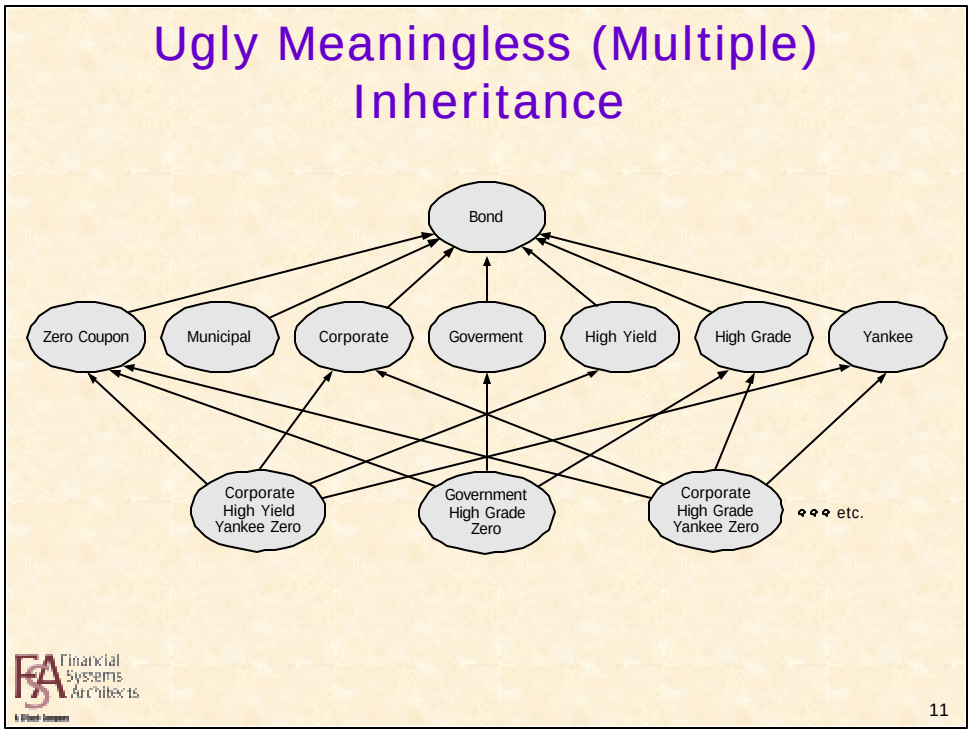
The main thing very prominent above but that we never see in an object model is this “is sometimes” relationship. But this is the “data” you will get from a business person, with respect to classification. And when “is sometimes” obtains in two directions between types of things, then you can anticipate a serious modeling problem.



The “Intuitive Semantics” of a business language is inadequate as a springboard for a software design for many reasons.

Traditions give rise to inconsistencies. For example, “equity” classifies instruments based on the kinds of ownership rights conveyed, while “fixed income” classifies them based on expected cash flows. The two categories both overlap, (e.g., preferred stock) and leave gaps in between (e.g., collateralized mortgage obligations). Yet many businesses divide all their work into these two areas.

Also, for example, “buy side” and “sell side” are overloaded based on context (of a trade, as a role in the financial marketplace), and the latter usage is deliberately obscure. Not understanding this term makes you an “outsider”.



This slide does NOT say that *MULTIPLE* inheritance is ugly and meaningless, (a currently very popular and misguided notion).

It says that *ANY* inheritance is ugly and meaningless as a RELATION BETWEEN TYPES OF THINGS. (which you want to represent in any kind of business model). Inheritance is a relation between *software classes*. More about this later.

The above is the kind of model we came up with the very first time we built an object oriented model in this field. Hot off a few C++ and Objective C programming projects. You find some classifier used by the business, turn it into a class, and go. NOT!!

Most of the classifications represented above are not even based on any attributes and relationships, or even values of same, of the objects classified. For example, what makes a "corporate" bond a corporate is the type of the legally responsible entity standing behind the issuer of the bond. So, it is a `corporate` based on the type of something related to something else that the bond is related to. Good software class, right? The *best* it gets is when the classification is based on the *range of values* of some attribute of the bond itself – for example, high yield bonds. The problem is, these classifications cannot be dismissed, either, all kinds of rules and processes depend on them.

Problems with Inheritance for Domain Modeling

- Multiple Inheritance Results in Spaghetti
  - combinatorial explosion
- Inheritance (specialization) vs. Typing (abstraction)
  - with specialization, classes come first,
  - in typing (abstraction) instances exist first
- Inheritance is Design Mechanism, not Semantic Relation
  - not about relationships between concepts; about machine tools building (software class + compiler&OS = object instance)

Financial Systems Architects

12

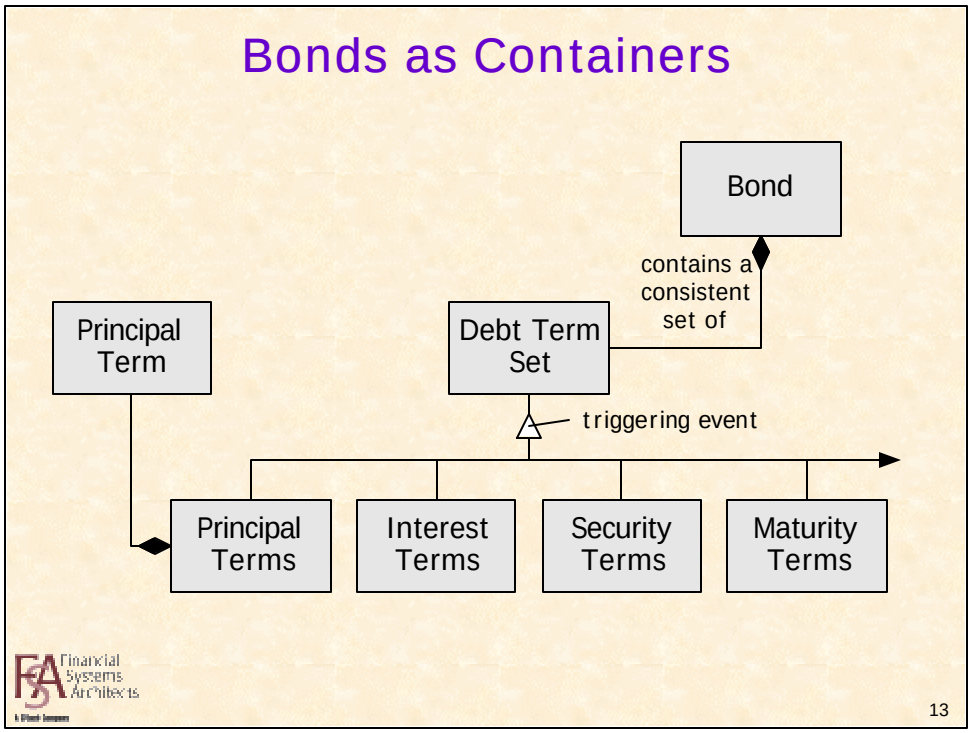
Let's just face the simple truth:

In the real world, types of things do not EVER, no NEVER, inherit from each other. Only INSTANCES do that, like the baby BEAR INSTANCE and the momma BEAR INSTANCE above. Not a single bear has ever "inherited" from the "bear type".

Nor did the polar bear type ever inherit from the bear type. Types just don't have that ability, to inherit. In fact, as creations of the human mind, the concept of a polar bear is historically prior to the general concept of a bear. Of course, when a new concept is a generalization or abstraction of an earlier one, then everything that the old one applies to, so does the new one. When it comes to concepts, we usually go from narrower ones to broader ones. But this does not mean that any particular relation exists between the attributes and behaviors of instances of the first and the second.

Just the opposite with a software design. In order to achieve reuse, we construct partial templates that can be specialized in a variety of different ways to create instantiable templates. Or even take one template and jigger it so it can do something different. Our great but obvious discovery:

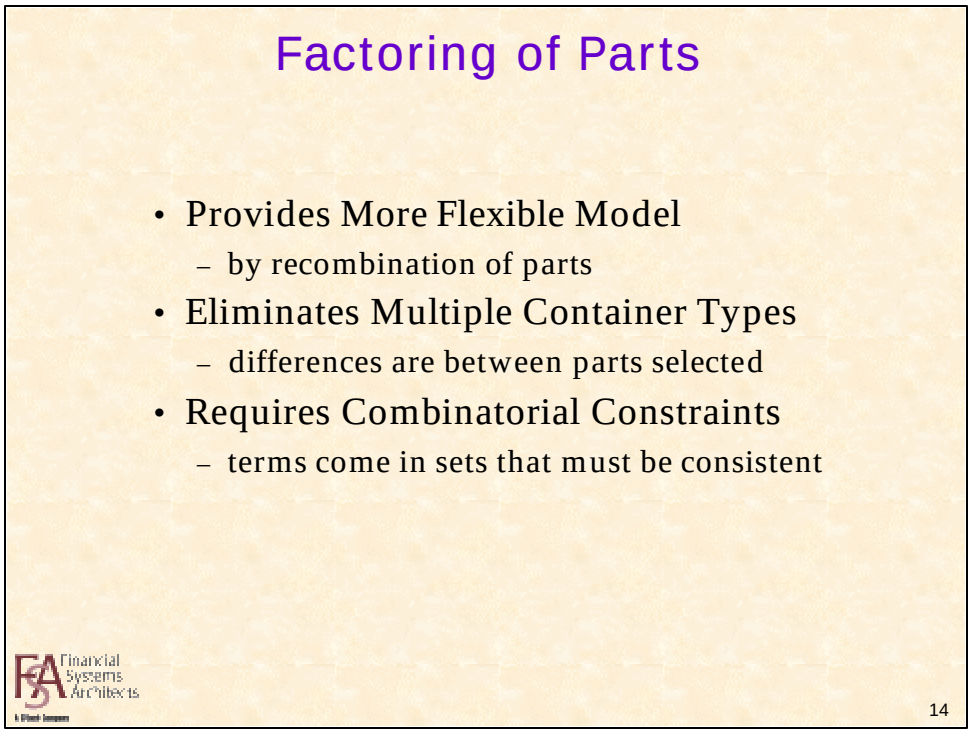
INHERITANCE AND GENERALIZATION ARE VERY DIFFERENT



Why is this so important? Because if you don't *explicitly* make clear to yourself that you are building *one* model that shows the relationships between types of things and relationships that just ARE THERE, not because you designed them that way, then you don't have the *freedom* to consciously DECIDE what, in your software design, you are going to LET inherit from what. Or, instead, you could equally make *clear to yourself* that you are **not** going to build a model of the world, just of the software, and stop pretending that you are. But we did not know this at the time. We only knew that multiple inheritance didn't work nicely.

The next object concept we learned was containers, so we eliminated the combinatorial explosion and enabled easier changes by modeling a bond, or any other instrument, as a container that could contain different term instances. For example, a British Consul is a rare kind of bond with no maturity terms. We do not need a new class to model it, we simply do not select a maturity term set when defining the terms of any British Consul.

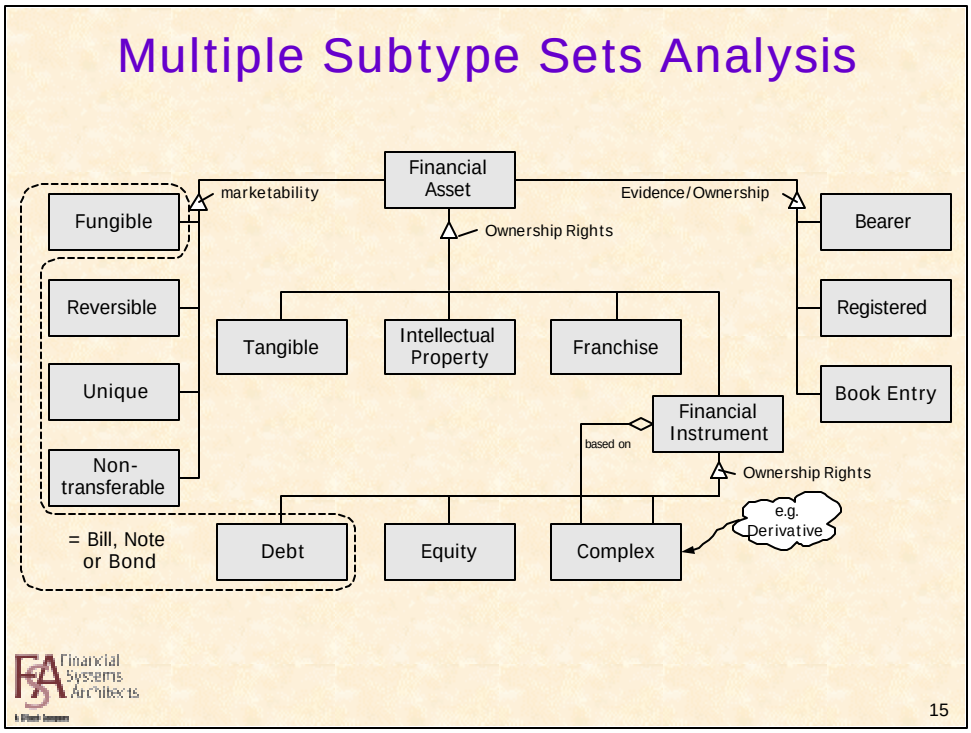
The discriminator for the different types of term sets is the "joint action" they describe. A joint action (what use cases SHOULD be) is a combined behavior of a number of participants that results in some related set of changes of states. Joint action is introduced Wirfs-Brock (slide 19) and RM-ODP (slide 17) and exploited in D'Souza and Wills Object, Components, and Frameworks. The paying of interest in one type of joint action. (in some contexts, a use case).



This means that we use the concept of "British Consul" explicitly as a template for designing a specific instance, but not as a software class. As a set of rules expressed in text (and put in database) for HOW to create an instance that will conform to the type.

This model is a lit-tle more complicated, in that the bond is a container of a bunch of containers. And the contents of each container must be consistent. We had no good way to express these consistency rules. Later, we began to write text to go with object models, to capture all the constraints that did not fit in the graphical representation. But an Object Constraint Language would have been useful.

Of course, the same "business classifications" of bonds must still be accounted for, that are found when listening to bond traders. But now, it was clear that those classifications were NOT software classes, but rather something else. [At this stage, the "templates" for construction discussed above.] Note that the kind of construction of instances we are talking about required the intervention of a person; it was not the simple instantiation of a software class, because it required defining first instances of parts, and then including consistent sets of those parts in the container. *This* concept of template we learned from Pat Tsien, now the head of financial services consulting for Andersen.

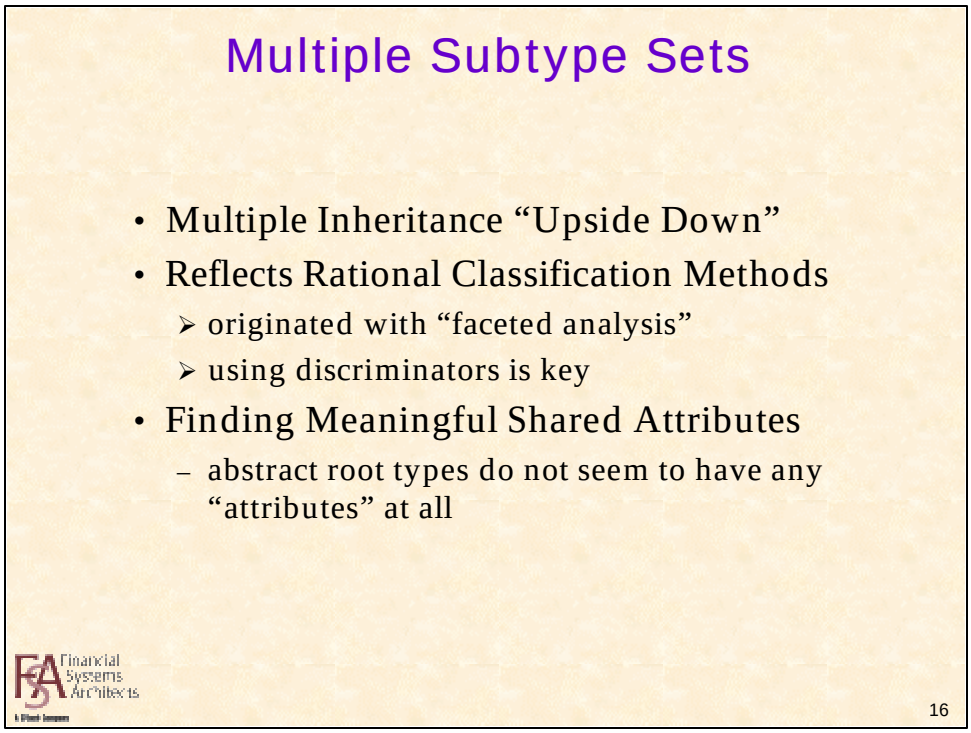


While we found the container and a set of templates a good model for handling the subtypes of each fairly general type of instrument, such as a bond or a stock, we still had the problem of organizing *all* instruments, and relating them to other kinds of financial vehicles.

The solution was is in a little Bellcore monograph of Haim Kilov's, later turned into his first book, Information Modeling: Adapt the IDEF-1 data modeling concept of multiple subtype sets to objects, and turn "multiple inheritance" on its head, allowing each type of thing to be subtyped in multiple ways, always using a consistent discriminator. We related this idea to Ruben Prieto-Diaz's use of multi-faceted classification for information retrieval.

Once we applied the principle of strict subtyping, according to a discriminator, we found that a "bond" itself was not a fundamental type, but rather an object that satisfied a particular combination of such types, namely, being a debt instrument that was ALSO fungible (indistinguishable among units – e.g., one 10,000 GMAC bond is entirely indistinguishable and exchangeable for any other) and unrestrictedly marketable, as well as satisfying other restrictions on the values of some of its terms.

The figure ignores topic of where *money* fits in this scheme. (A solution to *that* problem being a chapter in itself.)



The "essential" types in this kind of a model are different from classifications mainly in that provide a neat and reusable organization for the things in the business.

As ***types***, they represent predicates that *may apply* to instances of things

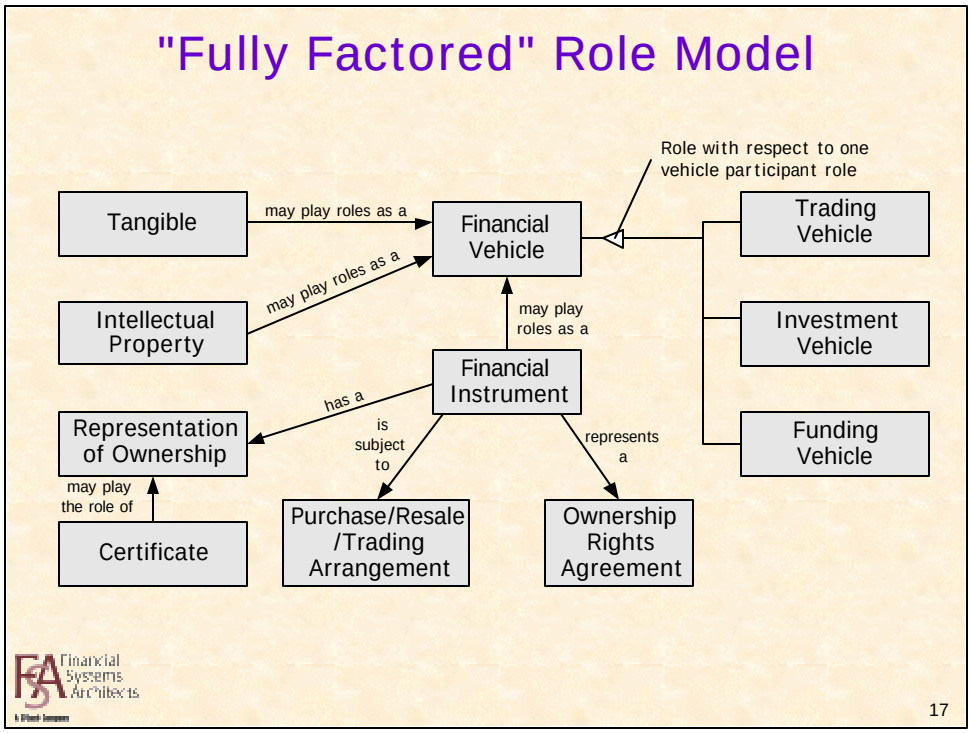
as opposed to

software classes, which are templates *used to generate* object instances that may be models of aspects of those things.

The question was: how to map from one to the other?

This is not a matter of "refinement", but of *mappings* between distinct entities that exist in distinct models. Many possibilities exist. For example, since any real thing will simultaneously satisfy a single type from each subtype set, an object that models a thing can be a container which contains one object from each of the subtype sets.

We found no simple, generally applicable, rules. An unsatisfactory result, especially since the root type in this model seemed to have no attributes or behaviors at all. And the model above is not even sufficiently abstract, dealing only with the investor's perspective on financial vehicles.

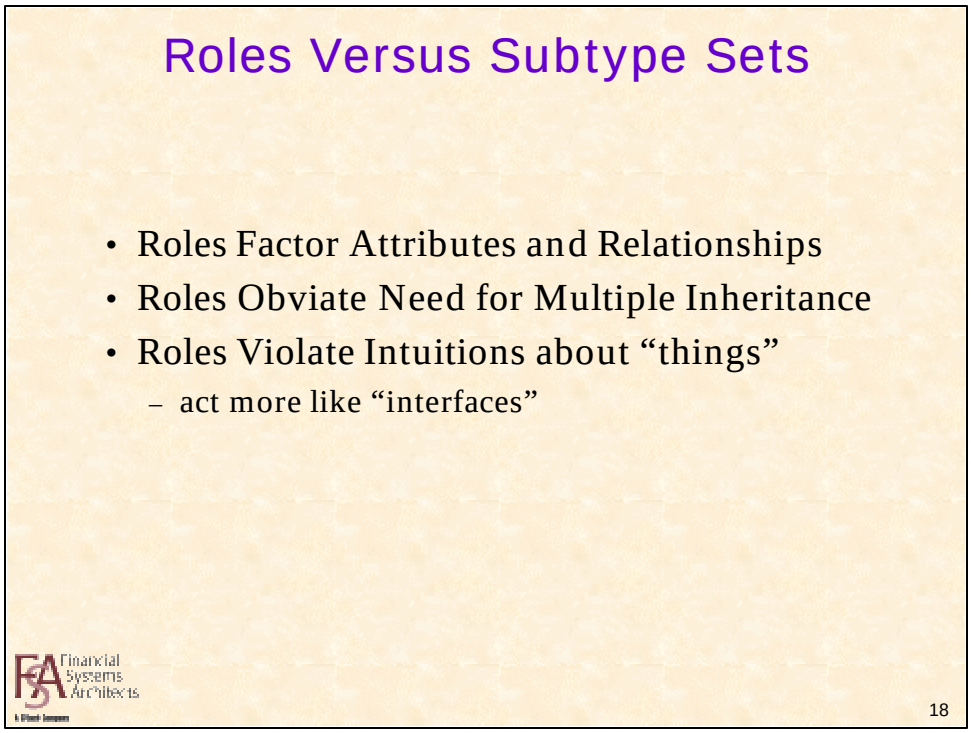


So, abstracting from *investment vehicle*, we wound up with a questionable incredibly abstract root type, *financial vehicle* – questionable because it might suggest that it could be possibly reasonable to make it into a very abstract software class – a purpose for which it has not proven to be useful.

Under the influence of the Foundations section of the Reference Model for Open Distributed Processing, ISO standard 10746-2, (RM-ODP) and learning more about how to *use* roles from Trygve Regenskaug's Working with Objects, we decided to model the behaviors and attributes of a thing, when used for a particular kind of financial purpose, as a *role played by the thing*, rather than as a type of thing.

We break the Gordian knot created by classifying yachts and patents for AIDS medicine as the same type of thing by NOT classifying them at all, rather letting them play the same role sometimes. For example, both the yacht and the patent might have a bid and ask price. But this is not an attribute of the things themselves, it is rather an attribute of them in the role of *trading vehicle.*

Do not think that a trading vehicle, for instance, is "really" a role, rather than a type. You *can* create a type for anything you want. Even things that might be better modeled as roles.

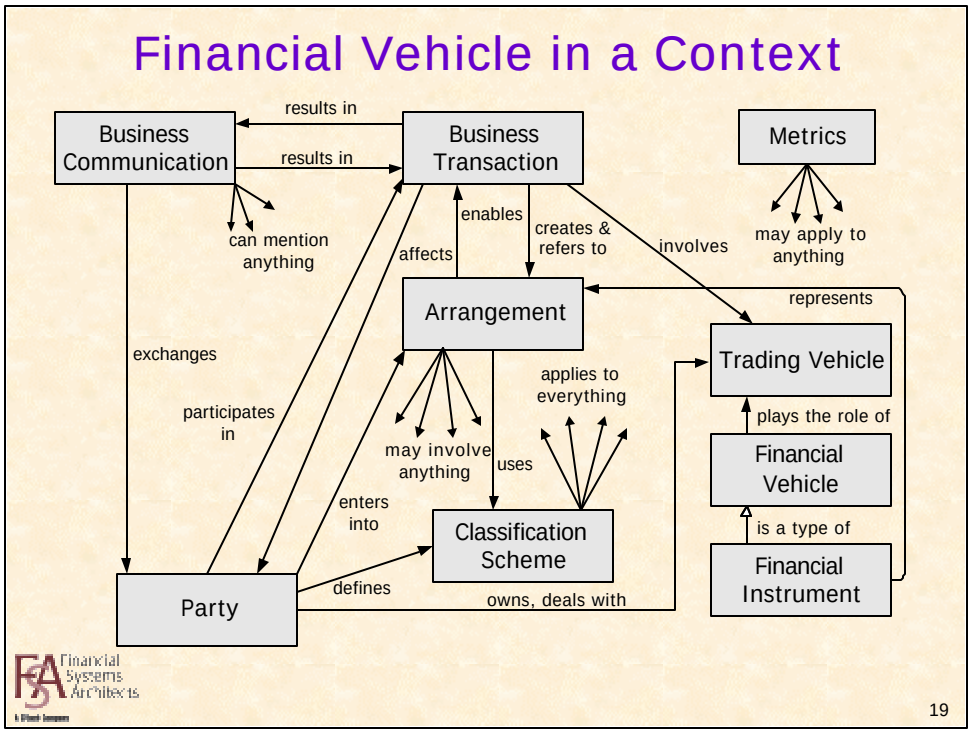


Similarly, a software designer might or might not *decide* to model roles as objects. But here, even though a mapping is required, we have found that there are clear "Best Ways" to map from roles to classes, depending on the programming language. In most cases, the software designer who uses roles moves further away from a lot of inheritance (a direction begun when containers took on some of work), multiple or otherwise.

A role type can be, and often should be, modeled as a software class, with instances of the role representing some other object playing the role. What is special about instantiation of a role class, then, is that the role instance must be associated with some other object that is playing the role. It cannot stand alone.

Some people object to this. Perhaps they are wedded to the idea that software objects must represent "things", and of course a role is not a thing. But you can use a software class and the objects it generates for anything that it is useful to use them for, and this includes representing roles.

The biggest weakness in this discussion of roles is that role modeling emphasizes the *dynamic* aspects of the modeled world (behaviors, in fact, joint behaviors). But all our diagrams and examples concern only *static* relationships.



Here is a business domain ontology for trading financial instruments in which roles are included among a variety of other kinds of objects of thought. (Some others of which, such as business transactions and metrics, are also not "things").

We have mapped structures like this (of course expanding all the categories above) to software classes with simply rules, although these still need to be designed to meet project needs. But the mappings resulted in *efficient* designs, because only the aspects of the things involved in a particular piece of work need be dragged into play. For example, when you are issuing a financial instrument, you need to handle its underlying terms and conditions (the arrangement it represents). But when you are trading it, this model lets you leave all that behind.

The only problem with roles is that they may become yet another "silver bullet". Some people assert that *only* roles count –and like all extremism, this is probably a dangerous view. But maybe they are really just trying to recapitulate the *responsibility* driven design of Rebecca Wirfs-Brock's Designing Object Oriented Software,

Using Financial Vehicle Model on a Project

- Part of The Complete Ontology
  - strict partition of object types
- Factoring Makes this Easier
  - relationships replace many hierarchies
- Implementing Roles with Java Interfaces
  - requires no common object type

Financial Systems Architects

20

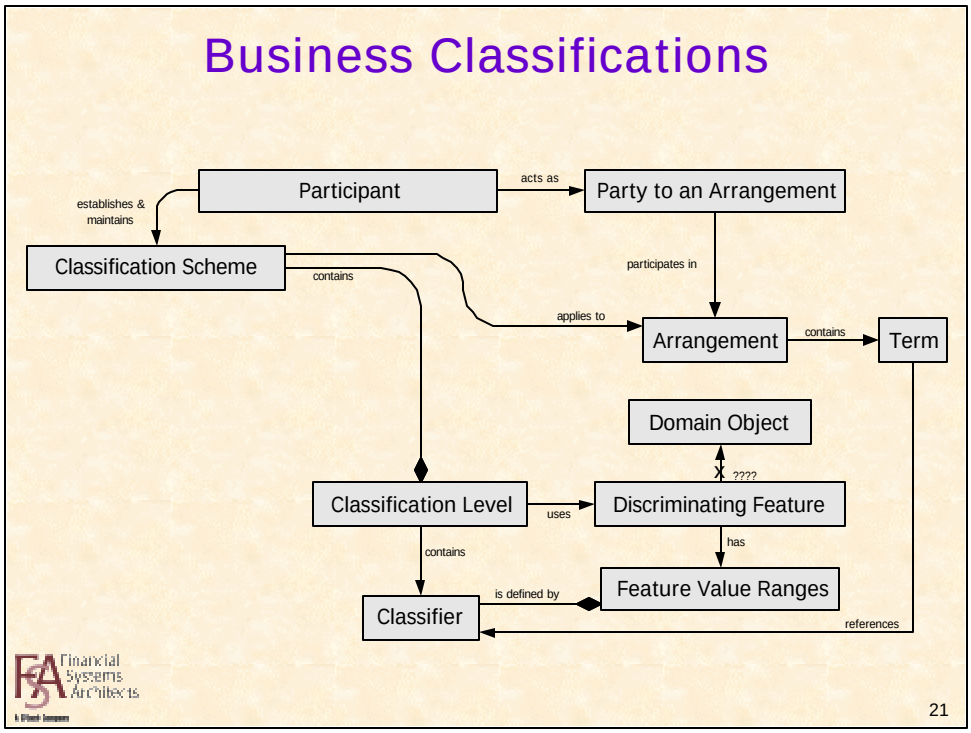
An ontology, for us as business modelers, is a *partition* of the universe of discourse into all the possible objects of thought or "categories of being". The taxonomies that explode each category rely not only on subtyping, but on any other definitional relations. For example, we divide *parties* into *party entities*, such as people, and *participants*, (roles played by parties, such as *customer*).

In Java, my firm FSA has been implementing roles as interfaces. This seems to work very well. Domain independent categories of being, such as roles, rules,and events, might all have specific best strategies for representing them as software objects, depending on the programming language. This seems to be the idea underlying Peter Coad's *colors*, as implemented in the Together tool, where different colors seem to classify objects into *things*, *roles*, and *facts* (although he gives these more complex names)

.

In any case, to understand roles better, we have been looking at the work of Wegmann & Genilloud, such as "The Role of Roles in Use Case Diagrams" and "A Foundation for the Concept of Role in Use Case Modeling" <guy.genilloud@ica.epfl.ch>.

But many other categories of being also figure into modeling financial vehicles.



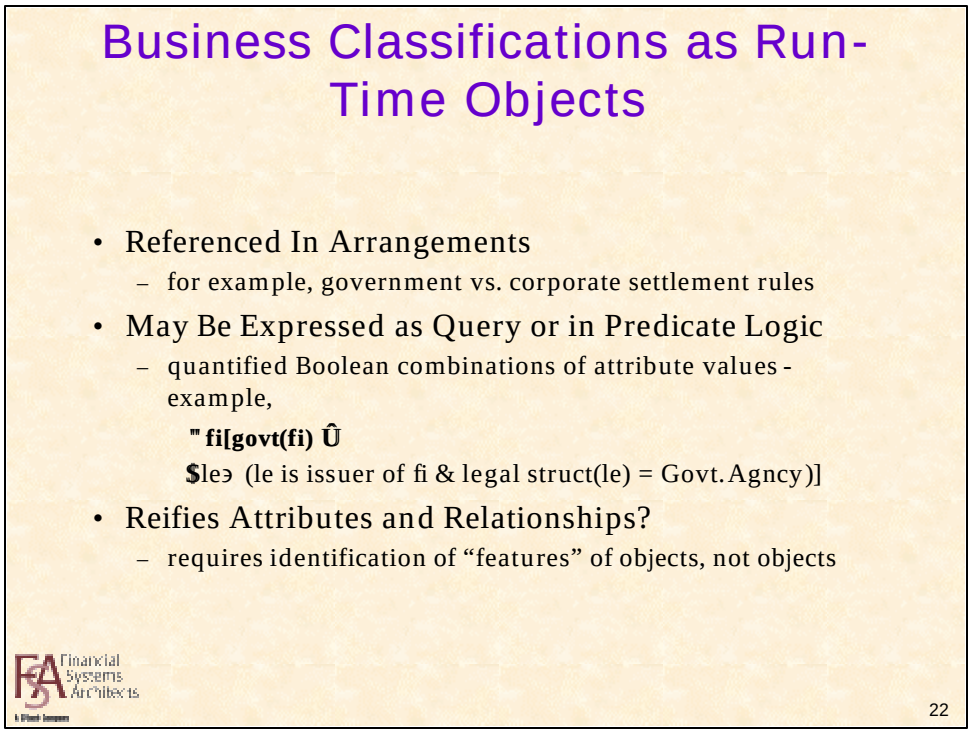
Particularly, the pesky

classifiers of vehicles

created and used by business people, that we mistook for fundamental types, and then further mistook for things that should be implemented as classes.

These classifiers are types defined by any predicate that a thing can satisfy. Testing satisfaction usually requires looking at other things that are related to the things, the way we need to find the domicile of the obligor of the bond, and the domicile of the issuer, to determine whether a bond is a Sushi (or perhaps instead a Salsa bond or a Maple Leaf). They are used to organize information, (what is our exposure to Sushis?), apply rules (the tax ajusted return for Sushis is equal to…), and make and express decisions (let's get the hell out of Sushis!).

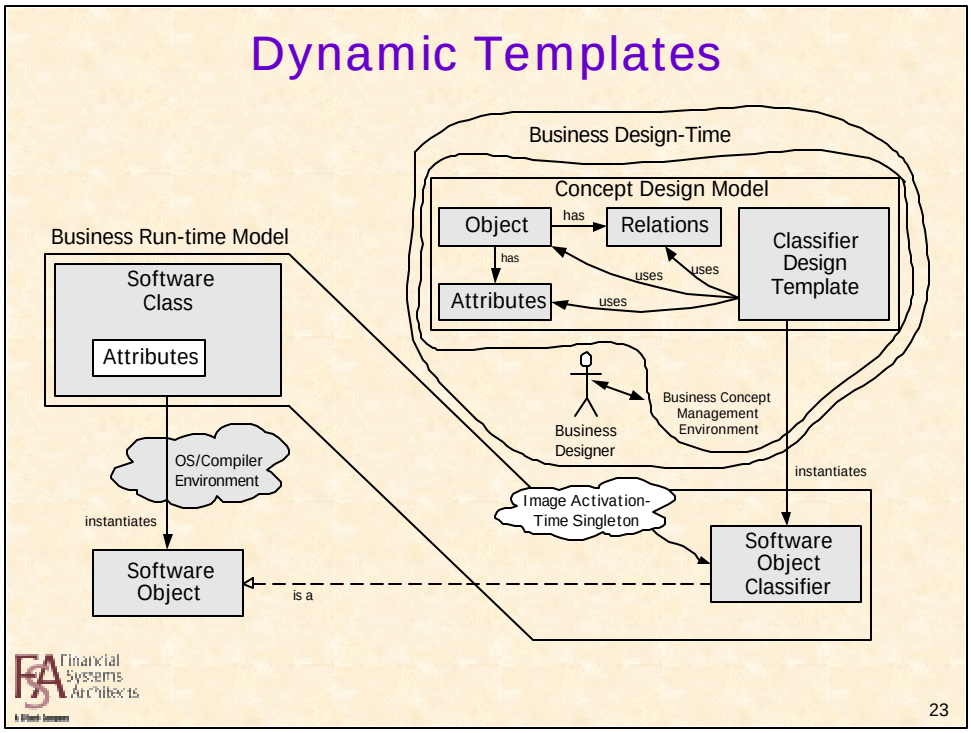
How should they be modeled? certainly not as software classes, or even categories for organizing a business model. They are not a modeler's tool, they are tools of the the business itself. Each classifier is an individual "thing" *in* the business. So, the modeling job is to account for them as categories of being, like any others, such as roles or parties.



Unlike most of the things people think of a "business objects", that represent things, such as IBM stock or gold bars, classifiers are tools of thought and expression. Our mistake was to think that these kinds of tools belonged only to modelers, instead of to the business itself, or at least, to *confuse* the ones that belonged to us modelers with the ones that belong to the business. And so to put them in the model as classes, or even in a "metalanguage". Business types are right there in the business universe, along with people and customers and gold bars. And, as the previous slide shows, not only do business people use classifiers, but they organize them into groups and hierarchies of groups.

What you find inside a classifier object, of course, is something that looks like an object SQL query. And this means that what the OO world deals with as "metalanguage" must come home and live with what they call an "object language". This is not hard, from a mathematical point of view. In fact, using higher order logics is a much neater way of modeling abstraction than all meta meta meta meta models that are currently in vogue. But it wrecks havoc with the simplicity of the models, because to express a classifier satisfaction criterion, you need to refer to the attributes and relationships of your objects as objects themselves. [Reification.]

Dealing effectively with reification is the big problem.

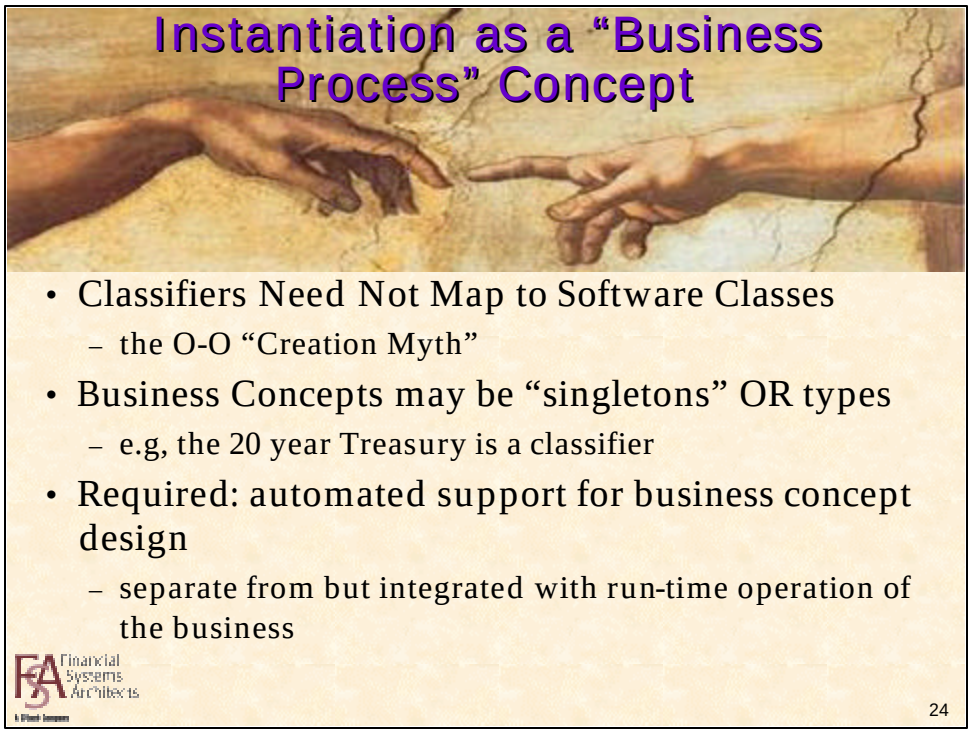


When you define new classifiers, you need consider all the attributes and relationships in the model, as well as all the domains of values that the attributes can have, as "fair game" to be referenced by another object. In other words, you have to treat it as an object.

This *also* applies to defining new kinds of financial instruments(as well as new kinds of "products" in other business domains). You take pieces of ideas from a variety of places and put them together or sell the pieces separately. Clock Radios, FM Tuners, mutual funds, the dividends only from a stock.

One way to deal with this is simply to bite the bullet and build systems that map to models where all the parts are exposed. For example, Nakamura & Johnson's "Adaptive Framework for the REA Accounting Model" (hnakamur @cs.uiuc.edu), and Riehle & Timal's "Dynamic Object Model" (www. riehle,org) Use reflection in CLOS. But this is only half the story.

Once you have defined a new category, or built the new product, you don't want to keep dealing with the parts. It is more efficient, more modular, to do what businesses always have done, and create TWO separate environments, one that supports product design, another that supports product delivery. NOW the problem is actually simpler: how to make the links between the two environments exactly as tight as they need to be, but not tighter.



No matter *how* short classification and product life cycles get, lets say, one hour between design and delivery, there is still a *big* difference between that and "instantaneous". And a federated pair of systems might be the right solution for that kind of time constraint.

Just as a software class is a template for stamping out software objects, product templates, can stamp out new bond instances in the "back room", and feed them to the bond trading systems, without the bond trading systems having a similar ability to define new types. Similarly, if a risk manager decides that she wants to aggregate positions according to some new classifier, you can define that classifier in one system, and use it in another.

The foundation for our thinking is now RM-ODP and logic & scientific method. ODP is unfortunately as dense as it is deep. Few accessible overviews exist. One is a brief chapter in Blair and Stefani's Open Distributed Processing and Multimedia. There are the proceedings of the nine consecutive OOPSLA workshops on Behavioral Semantics (kenb@ccs.neu.edu). For our specific subjects, Genillou & Wegmann, "On Types, Instances, and Classes in UML"(alain.wegmann@epfl.ch). A good logic book is Boolos and Jeffries, Logic and Computability.

I hope you have enjoyed this food fight, the weapon of choice being the "mystery meat" served by the methodology industry at our own OMG.