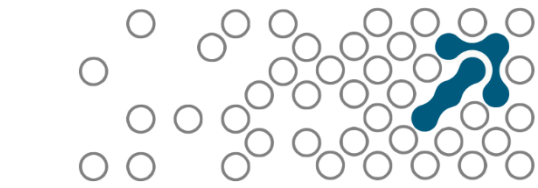
THE SEVEN FACES OF DR. "CLASS"

By: Dave McComb



semantic arts

The Seven Faces of Dr. "Class": Part 1

"Class" is a heavily overloaded term in computer science. Many technologies have implemented the concept slightly differently. In this paper we look at the sum total of concepts that might be implemented under the banner of "class" and then later we'll look at how different technologies have implemented subsets

The seven facets are:

- Template
- Set
- Query
- Type
- Constraint
- Inclusion
- Elaboration

Template

One aspect of a class is to act as a "template" or "cookie cutter" for creating new instances. This is also called a "frame" based system, where the template sets up the frame in which the slots (properties) are defined. In the simplest case, say in relational where we define a table with DDL (Data Definition Language) we are essentially saying ahead of time what attributes a new instance (tuple) of this class (table) can have. Object Oriented has this same concept, each instance of a class can have the attributes as defined in the class and its superclasses.

Set

A class can be seen as a collection of all the instances that belong to the set. Membership could be extensional (that is instances are just asserted to be members of the class) or intensional (see below under the discussion about the inclusional aspect). In the template aspect, it's almost like a caste system, instances are born into their class and stay there for their lifetime. With set-like classes an instance can be simultaneously members of many sets. One of the things that is interesting is what we don't say about class membership. With sets, we have the possibility that an instance is either provably in the set, provably not in the set, or satisfiably either.

Query

Classes create an implied query mechanism. When we create instances of the Person class, it is our expectation that we can later query this class and get a list of the currently know members of the class. In Cyc classes are called "collections" which reflect this idea that a class is, among other things, a collection of its members. A system would be pretty useless if we couldn't query the members of a class. We separate the query facet out here to shine a light on the case where we want to execute the query without previously having defined the class. For instance if we tag photos in Flickr with a folksonmy, and someone later wants to find a photo that had a combination of tags, a class, in the traditional sense was not created, unless you consider that the act of writing the query is the act of creating the class, and in which case that is the type of class we're talking about here. This is primarily the way concept like taxonomies such as SKOS operate: tags are proxies for future classes.

Type

Classes are often described as being types. But the concept of "type" despite being bandied about a lot is rarely well defined. The distinction we're going to use here is one of behavior. That is, it is the type aspect that sets up the allowable behavior. This is a little clearer in implementations that have type and little else, like xsd. It is the xsd type "date" that sets up the behavior for evaluating before or after or concurrent. And it is the polymorphism of types in object oriented that sets up the various behaviors (methods) that an object can respond to. It is the "typeness" of a geographicalRegion instance that allows us to calculate things like its centroid and where the overlap or boundary is with another geographicalRegion. We rarely refer to the class of all items that have xsd:date as if it were a collection, but we do expect them to all behave the same.

Constraint

Constraints are generally implemented as "guards" and prevent non-compliant instances from being persisted. There is no reason that the constraints need to be associated with the classes, they could easily be written separately and applied to instances, but many implementations do package constraints with the class definition, for two reasons: one the constraints are naturally written in and lexically tied to the class definition and the other is just for packaging around the concept of cohesion. The constraint can be a separate language (as with OCL the

Object Constraint Language) or may be an extension to the class definition (as ranges and foreign key constraints are in relational).

Inclusion

That is, inclusion criteria. This is for classes that support inference, and are the rules that determine whether an instance is a member of the class, or whether all members of a class are necessarily members of another class. It also includes exclusion criteria, as they are just inferred membership in the complement. While it is conceivable to think of the "open world" without inclusion criteria, it really comes to the fore when we consider inclusion criteria. Once we have rules of inclusion and exclusion from a set, we have set up the likelihood that we will have many instances that are neither provably members or provably not members, hence "satisfiability."

Elaboration

Elaboration is what else can be known about an item once one knows its class membership. In Object Oriented you may know things about an instance because of the superclasses it is also a member of, but this is a very limited case: all of this elaboration was known at the time the instance was created. With more flexible systems, as an instance creates new membership, we know more about it. For instance, let's say we use a passport number as evidence of inclusion in the class of Citizens, and therefore the class of People, we can know via elaboration that the passport holder has a birthday (without knowing what their birthday is).

To the best of our knowledge, there is no well supported language and or environment that supports all these facets well. As a practical consequence designers select a language implement the aspects that are native and figure out other strategies for the remaining facets. In the next installment of this series, we will examine how popular environments satisfy these aspects, and what we need to do to shore up each.

The Seven Faces of Dr. "Class": Part 2

Now that we have teased apart the seven aspects of "class-ness", let's see what we can do with this. Let's first look at "class" as it has been implemented in existing systems.

Class	Relational	Object Oriented	OWL	Rules
Template	Yes, DDL is a template for each tuple	Yes, new()		Possible
Query	Yes, via SQL	Not native, must be implemented usually in the persistence layer	Not native, can be done with SPARQL + rdf:type	Depends on underlying persistence
Set			Yes	Not native
Type	Only of attribute ranges	Best practice polymorphic types	Because there isn't a default implementation there aren't types	Not native
Constraint	FK + other constraints often implemented at DB level	Typically coded in the "setter" methods	No, OWL doesn't have a constraint mechanism	Constraints could be written in rules
Inclusion			Yes	Interference can be coded in rules
Elaboration			Yes	

So what do we make of this? The first thing is that none of the environments we currently have available covers all the aspects of class out-of-the-box. Some of the aspects end up being patterns or add-ons.

Armed with this we can begin to start thinking of new configurations of language plus pattern that will give us all the capabilities we need, and in many cases do a better job (sometimes when we accept the

default implementation of one of these aspects we short change what we could have had.

The rest of this paper outlines a hybrid set of languages and patterns that takes maximal advantage of these aspects.

Let's start with OWL, and use OWL for what it is good for:" inclusion" and "elaboration." The "set" aspect of OWL is a bit problematic because of the two faced nature of the open world (it's good for some things and gets in the way in other cases.). Let's propose the usage and extension of SPARQL to cover the "query" aspect and the "set" aspect: by using the NOT operator in SPARQL (and mostly abandoning it in OWL) and by adding a SATISFIABLE clause to SPARQL. What SATISFIABLE gives us is access to the open world sets (individuals that might be terrorists for example) but closing the world with a negation as failure style NOT operator (essentially give me all the passengers who are NOT [provably] terrorists).

The "template" and "constraint" facility may end up being closely related. There have been some attempts to add a constraint feature to OWL, but maybe we should disconnect it a bit more, and probably it should be implemented in rules or a DSL (Domain Specific Language) that implements rule like behavior. The simple case of template creates instances that are relatively flat and look like the class they were cookie-cuttered out of. Correspondingly a constraint language would only need to concern itself with validity rules for the handful of attributes or relations that were attached directly to the instance.

But imagine a template language that could create a large constellation of objects and relationships. In some ways it would be a bit like the factory pattern in Object Oriented Design Patterns. Sometimes templates make instances in their own image, but that is just the simplest case. The language of the template would mostly be: "make," "find" and "findOrMake" (each would have a set of parameters which would be the clues or finding and the minimal values for making), and "assert" (which would have a propertyName and an object which could be another "findOrMake."

The constraint language would have to support: exists (against a query or enum), doesn't exist query or enum) format (type or regular expression), range, cross fied (<, <=,=,>,>=) and timeNow. The constraint language is a guard, and acts on the transactional level.

The type aspect brings up the concept: what can I do "to" or "with" instances of this type. With the simple types the behavior is pretty much built in, but implied: with dates you can calculate durations, and overlaps. With geographical primitives you can calculate distances, and areas and overlaps. Documents can be printed and edited. Things or messages can be sent to addresses (depending on the type of address, determines the type of thing or message that can be sent). Textual content can be translated from language to language. Magnitudes can be compared and converted to different units of measure. Sensors/Monitors can be "read" and actuators can started, stopped or moved. Programs can be run. Obligations can be discharged or violated. The question is: is something similar to gist, sufficient to define the behavioral primitives or is a language of behavior needed? Content can be rendered to appropriate media (music to speakers, text and graphics to printers or displays. Communiques can be sent to people or organizations.

Summary

All seven aspects of class are necessary and useful. No language gives us a native way to easily express all seven aspects easily and unambiguously. Until such as language exists, we can best serve our needs by adopting a set of patterns and techniques to supply the missing capability from whatever environment we're in.

11 Old Town Square
Suite 200
Fort Collins, CO 80524

970-490-2224
305-425-2224
info@semanticarts.com

© Semantic Arts, Inc.