Dynamic and Multiple Classification

January 1992

Object-oriented analysis should *not* model reality—rather it should model the way reality is understood by *people*. The understanding and knowledge of people is the essential component in developing systems. Therefore, OO analysis should not be based on any implementation technology—including OO software implementation. As Brad Cox [Cox, 1990] so eloquently expressed it, "'object-oriented' refers to the war, not the weapons...." When practiced in this way, OO analysis allows us to analyze *any* area of human reality—not *just* that of data processing. In addition, this allows us to implement our conceptual model using technologies other than OO programming languages (OOPLs).

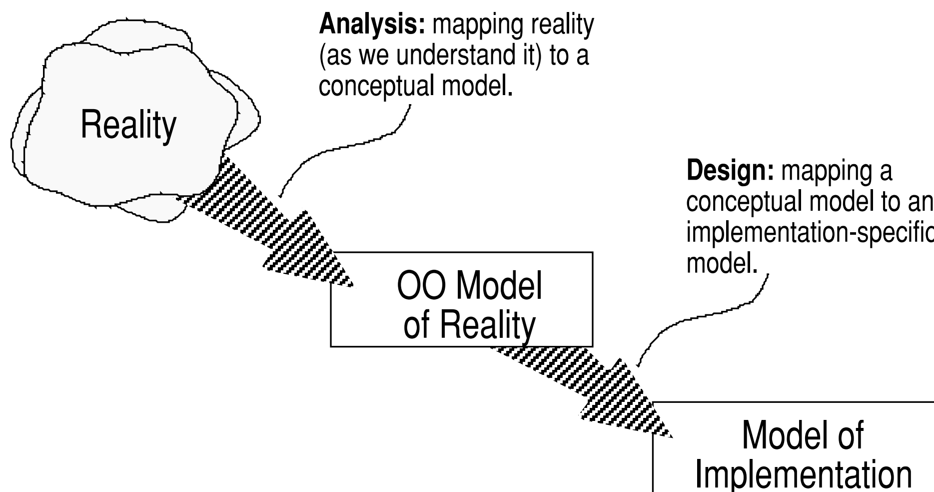

Figure 7.1. Analysis and design as different mappings.

The approach to design, however, is different. As illustrated in Fig. 7.1, design is defined as a process that specifies how the conceptual model of analysis will be implemented. A shift in thinking must take place from defining *what* is needed to *how* it can be provided. One of the reasons why the OO approach has been so successful is because the shift from concept to implementation is smaller than with conventional approaches. For instance, the *objects* we perceive can be implemented as objects in OO systems. The *types of objects* we define can be implemented as classes in OO systems.

However, the shift from an OO conceptual model to an OO implementation model is not always so smooth. This month's column explores two such areas: dynamic classification and multiple classification.

DYNAMIC CLASSIFICATION

Dynamic classification (also called dynamic typing) refers to the ability to change the classification of an object. For example in Fig. 7.2, the "Alice" object changes from being an instance of Employee, Manager, and Salesperson to being a Unemployed Person. However, she still remains an instance of the object type Person. In other words, with dynamic classification, an object can be an instance of different object types from moment to moment.

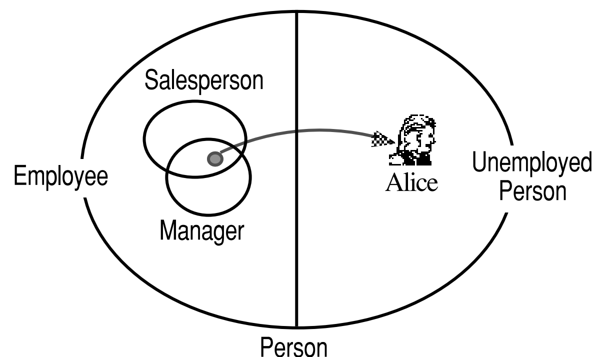

Figure 7.2. Dynamic classification refers to the ability to change an object's type.

Object life cycles

When we determine that an object is of a specific type, the object is *classified* as an instance of that object type. When an object is no longer a particular

type of object, the object is *declassified* and removed as an instance of that object type. Figure 7.3 portrays the "Alice" object being classified and declassified in terms of the object type Employee. At some point in her life, Alice is first classified as an Employee. Later, through some process, Alice is declassified as an Employee: she becomes unemployed. At another point, Alice may become reemployed, followed again by a period of unemployment. This behavior may continue until retirement is reached or the process of death takes place.

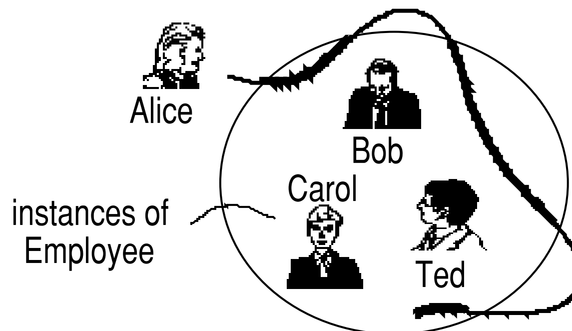

Figure 7.3. The object "Alice" moving in and out of being an Employee over time.

Figure 7.3 illustrates the lifecycle of an "Alice" object in terms of one object type. (Figures 7.2, 7.4, and 7.5 indicate portions of the object's lifecycle in terms of several object types.) Without dynamic classification, the classification of the "Alice" object cannot change. Therefore, every time she is removed from the Employee class Alice would cease to exist. To make things worse, when she became reemployed she would be created as a new object—devoid of any already-existing attributes and past history. Dynamic classification, then, permits objects to exist independently of the object types by which they are classified—and allows their termination.*

Implementing Dynamic Classification

Specifying classification changes is an important aspect of OO analysis. However, while Smalltalk, CLOS, and Iris support dynamic classification to a

---

* Note: An object cannot exist without any classification. Therefore, as long as an object is classified by at least one object type, the object still exists. In the case of Alice, Fig. 7.2 indicates that when she is declassified as an Employee, Alice would still be an instance of Person. In other words, Alice survives the employment termination process. When an object is finally terminated, it is declassified of all of its object types.

limited extent, the remaining OO programming languages offer no direct support. To implement dynamic classification requires the skill of an OO designer to develop a "work-around" solution. One solution is that whenever an object changes classes, a new object is created. The appropriate properties from the old object are copied to the new object, and the old object is finally terminated.

Another solution is to define a status flag that indicates the classification. For example, the Person class could contain an employment_status field indicating whether or not a Person object is employed or not. In this scenario, the programmer would need to write extra code to override method selection. For instance, even though the retire_Employee method would be associated with the Person class, it may not be invoked for unemployed persons. Therefore, the employment_status field would have to be checked first to determine whether or not the person is employed. In other words, a person's employment state would have to be evaluated by customized code instead of the method-selection mechanism inherent in OO software.

The point is no matter which implementation is chosen, the notion that an object can be classified and declassified is foreign to most OO languages.

MULTIPLE CLASSIFICATION

Figure 7.3 depicts an object in terms of only one object type. However, the "Alice" object may be classified and declassified as various object types over time. At the age of eighteen, she will change from the object type Girl to the object type Woman. At some point, she may get married and become an instance of the object types Married Person and Wife. Quite independently, she may be confirmed as a Supreme Court Justice or buy a pet and become a Pet Owner. While she may be a Supreme Court Justice for life, she may later give the pet away and be removed from the Pet Owner set.

In her lifetime, Alice may be an instance of many object types. This means, first, that the object types that apply to an object can change over time (dynamic classification). Second, it means that an object can have multiple object types that apply to it at any one moment. When an object is an instance of more than one object type, this is called multiple classification (not to be confused with multiple inheritance).

Multiple classification is alien to most data-processing implementations. Typically, a data record can only be obtained via its creating record type. Most OO programming languages are similarly restrictive, requiring an object to be an instance of one OOPL class for life—not counting the object's superclasses. However, in OO analysis, we are not modeling how computers languages and databases work, we are analyzing our enterprise world as *people* would see it.

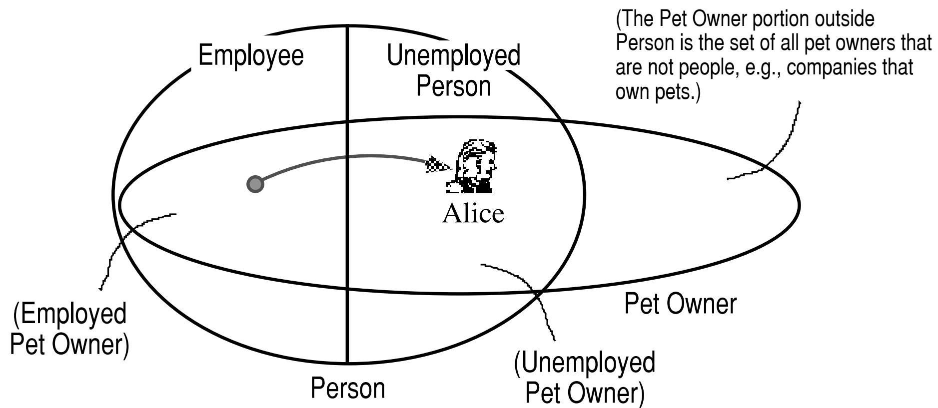

Figure 7.4. The object "Alice" involved in both dynamic classification and multiple classification.

The class/subclass hierarchy is one way of classifying an object in multiple ways. For example, if Alice is an instance of the Employee class, she is also an instance of Person (Fig. 7.4). In other words, when Alice is classified as an Employee, the Person classification is implied by the superclass hierarchy.

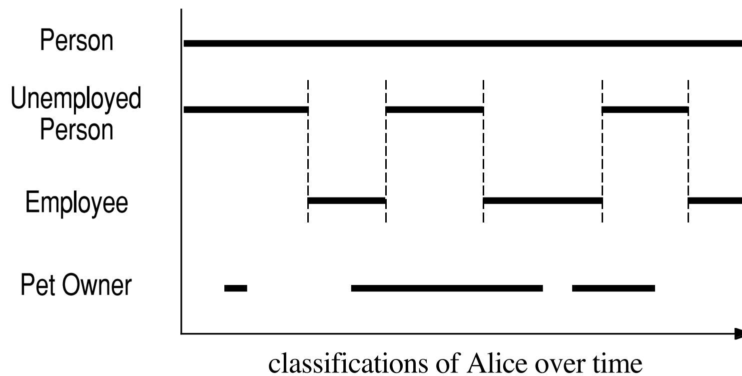

Figure 7.5. In the "real world," not only can the classifications of an object change, the object can be an instance of several object types at the same moment.

However, an object can also be an instance of multiple classes that are not implied from a superclass hierarchy. For example, multiple classification makes it possible for the "Alice" object to be an instance of both the Employee and Pet Owner classes. Since the Pet Owner class is not a superclass of Employee, being an Employee does not automatically imply being a Pet Owner. Therefore, if Alice is an instance of Employee, she is not necessarily an instance of Pet Owner—and vice versa. In order for Alice to be both a Pet Owner and an Employee, Alice must be explicitly classified as both. Over time, this change in classification could look something like the graph in Fig. 7.5.

Implementing Multiple Classification

Multiple classification, based on a class/subclass hierarchy, is supported and is one of the benefits of OOPLs. However, multiple classifications that are *not* part of an object's inheritance hierarchy are not supported by OOPLs. Typically, if a developer wishes to "get around" this limitation, additional classes are specified. These additional classes define the necessary multiple-classification combinations. For instance, to support the "Alice" object as a Pet Owner that can also be an Employee or an Unemployed Person, two additional subclasses (Fig. 7.6) are defined: Employed Pet Owner and Unemployed Pet Owner. The Unemployed Pet Owner class will be a subclass of both Unemployed Person and Pet Owner; the Employed Pet Owner class will be a subclass of both Employee and Pet Owner. In this way, multiple classification could be handled via the multiple inheritance mechanism supported by some OOPLs. This approach has two drawbacks. First, a class is needed for every multiple-classification combination that can occur within a system (where 2^n combinations are theoretically possible). Second, not all OOPLs support multiple inheritance.

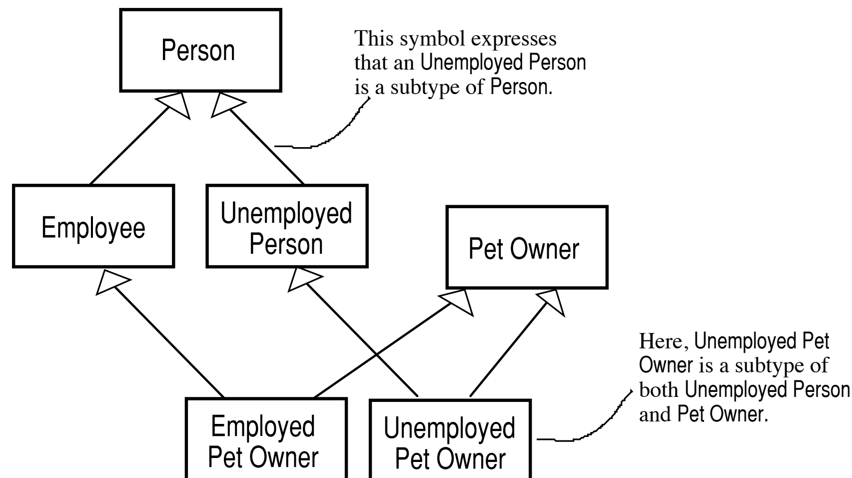

Figure 7.6. Multiple classification can be implemented via a multiple inheritance mechanism. In this diagram, Employee, Unemployed Person, and Pet Owner are classes whose combinations define the subclasses Employed Pet Owner and Unemployed Pet Owner.

Object Slicing

Another common technique that supports multiple (as well as dynamic) classification is called *object slicing*. In object slicing, an object with multiple classifications can be thought of as being "sliced" into multiple pieces. Each piece is then distributed to one of the object's various classes. For example in Fig. 7.5, the "Alice" object is depicted as an instance of the Employee and Pet Owner classes. To implement these two facts in an OOPL, one piece of the "Alice" object must become an instance of Employee and the other an instance of Pet Owner.

Obviously, objects cannot be "sliced" and made into instances of classes: it is a metaphor. These "slices," however, can be implemented by surrogate objects. In addition, an "unsliced" version of the object must also be recorded to serve—physically and conceptually—as a unification point for its surrogates. The slices, then, become the various recorded aspects of one unsliced object. One way to accomplish this is by adding two new classes: Implementation Object and Conceptual Object. The instances of Implementation Object are the object slices, where each is an instance of a different class. The instances of Conceptual Object are the unsliced objects, where each maintains pointers to its various slices.

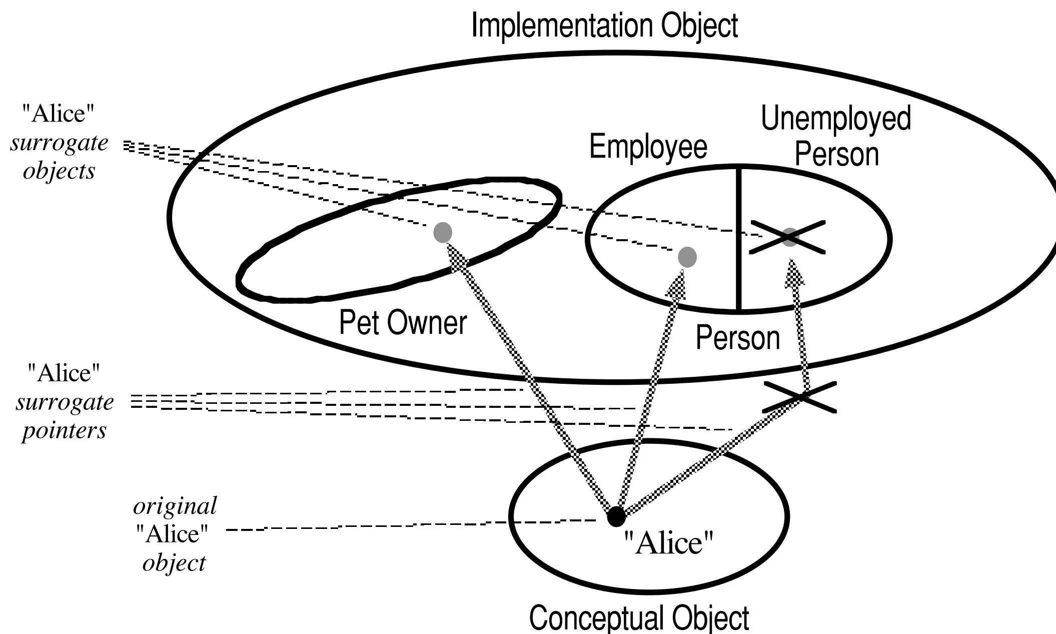

Figure 7.7. Object slicing supports dynamic and multiple classification.

An example of how object slicing can be applied is illustrated in Fig. 7.7. In this figure, the unsliced “Alice” object is represented as an instance of the Conceptual Object class. This one object representing “Alice” as a whole points to multiple “Alice” object slices. Since these slices represent different class implementations of the conceptual “Alice” object, they are instances of the Implementation Object class. The instances in the Pet Owner and Employee classes are slices of the “Alice” object. In other words, object slices of the whole “Alice” object are also “Alice” objects. However, the slices comply with the conventional OOPL requirement that each is an instance of only one class.

Additionally, changes in state can be accomplished by adding or removing the surrogates and the pointers to them. For instance, when Alice was classified as Unemployed Person, there was a pointer from the Conceptual Object “Alice” to the Unemployed Person “Alice” surrogate. When Alice became employed, the surrogate Unemployed Person object and its pointer were removed and replaced by a surrogate Employee “Alice” object and its pointer.

As each object is added or removed from the various classes, the construction and destruction operations of those classes would still apply. However, the object-slicing mechanism must add to these class-level operations by ensuring that objects do not have conflicting multiple states. For example, an object can simultaneously be an instance of the Pet Owner and Employee

classes. However, it cannot simultaneously be an instance of both the Unemployed Person and Employee classes. For an object to be classified as an Unemployed Person, the object must be removed as an instance of the Employee class.

Object slicing is a reasonably elegant solution to a problem not yet directly supported by OOPLs. However, in addition to the programming overhead mentioned above, object slicing also requires extra logic to support polymorphism and supplement the OOPL's method-selection mechanism.

CONCLUSION

This column examined two important modeling phenomena:

- dynamic classification, and
- multiple classification.

In analysis, they provide very useful notions that describe some of the ways we think about our world. The transition to OO implementation, however, is not a smooth one. While "work-around" solutions are possible, OO programming languages provide little or no *direct* support for these notions. If it were a voting issue, I would certainly cast mine in favor of enhancing OO programming languages so that these notions are directly supported. (In fact, I would like to hear how some of you might vote.) In the meantime, the OO designer will have more work to do.

REFERENCES

Cox, Brad J., "Planning the Software Industrial Revolution," *IEEE Software*, 7:6, November 1990, pp. 25-33.