Open Types and Bidirectional Relationships as an Alternative to Classes and Inheritance

Christian Heinlein, Abt. Rechnerstrukturen, Universität Ulm

Since the advent of procedural programming languages in the 1960s, data structures appearing in programs are typically modeled as *records* (or *structs*), and even the object-oriented notion of a *class* is basically a record equipped with a suite of procedures or methods. Even though data modeling based on records, e. g., modeling a person as a record possessing *fields* (or *members*) such as name, date of birth, address (which might itself be a record), etc., is rather natural and straightforward, a severe limitation of records is the fact that they are *fixed*: Once a record type has been defined, the set of its fields is invariably determined, and all its instances will possess exactly these fields.

Variant records in procedural languages and *subclasses* in object-oriented languages provide some more flexibility in this regard, but a particular type or class still remains fixed once it has been defined. So, even though it is possible, for instance, to "extend" a given class `Person` by defining a subclass `PersonWithSsno` in order to model persons with a social security number (ssno), the base class `Person` actually remains unchanged (so the term "extend" is quite misleading). This is particularly problematic when an existing program (e. g., a person management system) shall be extended later in an unanticipated way: If the original source code is not available or shall not be modified for reasons of modularity, introducing a subclass of `Person` does not help since the original code still creates instances of the original class.

To overcome these limitations of traditional record and class types, *open types* will be presented in this talk as an alternative data model for procedural and object-oriented programming languages. Its basic idea is to *separate* the definition of types from the definition of their constituents, i. e, their data fields, associations, and base types. Data fields are modeled as *attributes* defining unidirectional mappings from an open type to any other type, while associations are modeled as *relationships* defining bidirectional mappings between two open types (i. e., pairs of attributes constituting mutually inverse mappings). Finally, base types (or "supertypes" in object-oriented terminology) of an open type are declared by establishing one-to-one relationships between the type and its base types which are applied automatically on demand to map an object of the derived type ("subtype") to an associated object of a base type ("supertype"). By that means, attributes of the base types are apparently "inherited" by the derived type, i. e., objects of the derived type can be used whenever an object of a base type is required ("subtype polymorphism"). On the other hand, the model is more flexible than typical object-oriented approaches for the following reasons:

- Relationships to base types can be established retroactively, i. e., after a type has been defined. This allows in particular to introduce new "supertypes" later, which is usually impossible in object-oriented languages.
- Multiple and even repeated inheritance is possible without encountering any of the typical problems usually associated with these concepts. For example, name collisions between multiply "inherited" attributes are simply solved by explicitly moving from an object to the appropriate "supertype" object before applying the attribute, instead of relying on the automatic mapping described above. Similarly, the distinction between "virtual" and "non-virtual" inheritance (to use C++ terminology), i. e., whether a multiply inherited base type is shared or replicated in the derived type, is simply expressed by creating either a single or multiple objects of it. By that means it is even possible to model intermediate forms between virtual and non-virtual inheritance, i. e., to partially share several base type objects in an object of a derived type.
- An object of some type can dynamically "evolve" to an object of a derived type (e. g., a person can become a student) without changing its identity by simply creating an associated object of the derived type and establishing a relationship between the objects. Again, this kind of object evolution is usually impossible in object-oriented languages.