Editor: Bertrand Meyer, Interactive Software Engineering, 270 Storke Rd., Ste. 7, Goleta, CA 93117; voice (805) 685-1006; fax (805) 685-6869; ot-column@eiffel.com.

Schema evolution: Concepts, terminology, and solutions

Bertrand Meyer
ISE, Inc.

Most applications must keep objects from one session to the next. This is known as persistence. But objects are not raw data: They are instances of classes. What happens if an object's class (its *generator*) changes from one session to the next? This problem is known as *schema evolution* (the term *schema* is borrowed from relational databases). This column defines a framework for addressing schema evolution in object technology.

Forms of persistence

There are several ways to make objects persistent. First, the environment may offer a mechanism for storing objects in a file, as Figure 1 shows. Such a mechanism should satisfy the Persistence Closure Principle:

> When storing an object, it should also store any other to which it refers directly or indirectly.

This is what happens with the Eiffel *STORABLE* class: The operation `x.store ("some_file")` will store the entire object structure. The `x ?= retrieved` operation will recreate the structure.

Second, the persistence mechanism may use a relational-object interface to store objects in a relational database.

Finally, the mechanism may use an object-oriented database, whose concepts most closely reflect those of OO development.

A special case of the *STORABLE* mechanism actually transfers objects rather than storing them. The EiffelNet library, for example, uses *STORABLE* to exchange object structures between a client and a server. Even though no file or database is involved, the concepts discussed here directly apply to such a case.

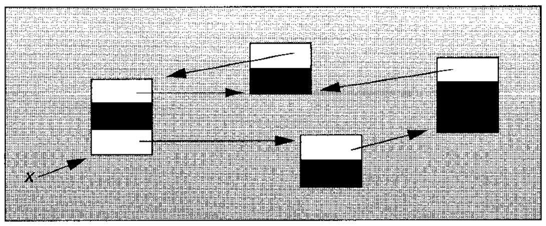


Figure 1. An object and its dependents.

Basic terminology

Schema evolution occurs if at least one class used by a *retrieving system* (the system that attempts to retrieve some objects) differs from its counterpart in the *storing system* (the system that stored these objects). In the case of object transmission through a network, the storing and retrieving systems would be better called the sending and receiving systems.

Object retrieval mismatch (*object mismatch* for short) occurs when the retrieving system attempts to retrieve an object whose generating class in the storing system was different. Object mismatch affects an individual object; schema evolution affects one or more classes.

Schema evolution occurs if at least one class used by a *retrieving system* differs from its counterpart in the *storing system.*

Possible solutions

Two extreme approaches to schema evolution are not appropriate:

- You could forsake previously stored objects (schema *revolution*!). The developers of the new application will like this idea because it makes their lives so much easier. But users will not be amused.
- You could offer a migration path from old format to new, converting the old objects en masse. Although this solution may be acceptable in some cases, it will not do for a large persistent store or one that must be available continuously.

What we really need is a way to convert old objects on the fly, as they are retrieved or updated. As a bonus, an on-the-fly mechanism will let you do en-masse conversion if you need it: You simply write a small system that retrieves the existing objects using the new classes, applies on-the-fly conversion as needed, and stores everything.

The mechanics of an on-the-fly conversion are tricky. We must get the details right, lest we end up with corrupted objects and corrupted databases. The most challenging problem is how each application will deal with an obsolete object.

Schema evolution involves three separate issues:

- *Detection* is the task of catching object mismatches (cases in which a retrieved object is obsolete) at retrieval time.
- *Notification* is the task of making the retrieving system aware of an object mismatch so that it can react appropriately. Continuing with an inconsistent object would likely cause major trouble later on!
- *Correction*, the task of the retrieving system, brings the mismatched object to a consistent state. In other words, it will be a correct instance of the new version of its class—a citizen, or at least a permanent resident, of its adopted system.

All three issues present delicate problems. Fortunately, they can be addressed separately.

DETECTION. There are two general categories of detection policy: nominal and *structural*. In both cases the goal is to detect a mismatch between the version of an object's generating class used by the storing system and the version used by the retrieving system.

In the nominal approach, each class version is identified by a version name. This approach assumes some kind of registration mechanism. Two variants are possible:

- You can use a configuration-management system to register each new version of the class, either by supplying a version name or letting the system supply one.
- You can use an automatic identification facility in the style of mechanisms used by Microsoft's OLE 2, which assigns random numbers so large that the likelihood of a clash is infinitesimal.

Efficiency dictates that you not waste too much storage space or time comparing descriptors.

Either solution requires some kind of central registry.

If you want to avoid this hassle, you will have to rely on the structural approach. The idea here is to associate a *class descriptor* with each class version. Descriptors are deduced from the actual structure of the class, as defined by the class declaration. Here you must ensure that the persistent mechanism stores both the object and its associated class descriptors. (Of course, you may store many instances of a class and only one copy of the class descriptor.) To address the detection problem, the retrieved mechanism will compare the class descriptor of each retrieved object with the new descriptor. If they are different, you have an object mismatch.

What goes into a class descriptor? Here you have some flexibility. The answer will be a trade-off between efficiency and reliability. Efficiency dictates that you not waste too much storage space or time comparing descriptors. Reliability dictates that you minimize the risk of missing an object mismatch. Here are four possible levels:

1. At one extreme, you could simply use the class name as the class descriptor. In most cases this is insufficient because the retrieving system would accept totally incompatible classes as the same class.
2. At the other extreme, you could use the entire class text as the class descriptor (stored perhaps not as a string but in an internal form such as an abstract syntax tree). This is clearly inefficient, and it may not even be reliable because some class changes are harmless. For example, if the new class text adds a routine but does not change any attribute or invariant clause, nothing bad can happen—yet an object mismatch will be detected, causing unwarranted trouble (such as an exception) in the retrieving system.
3. A more realistic approach is to include in the class descriptor the class name and the list of attributes, each characterized by its name and type. There is still the risk that two different classes might have both the same name and the same attributes, but this is unlikely.
4. A variation on technique 3 includes not just the attribute but also the entire class invariant. Including the invariant ensures that the addition or removal of a routine is harmless, because if the routine changes the semantics of the class it will affect the invariant.

Technique 3 is the minimum reasonable, and in usual cases seems a good trade-off, at least to start.

NOTIFICATION. A library mechanism can let the retrieving system know of an object mismatch so that it can take corrective action. In Eiffel, class *GENERAL* (ancestor of all classes in Eiffel, similar to `object` in Smalltalk) should include a procedure

```
correct_mismatch is
    do
       ...See full version below ...
    end
```

with the rule that any detection of an object mismatch will cause a call to `correct_mismatch` on the temporarily retrieved version of the object.

Any class can redefine the default version of `correct_mismatch`, and—like any redefinition of the default exception handling procedure `default_rescue`—any redefinition of `correct_mismatch` must ensure the invariant of the class.

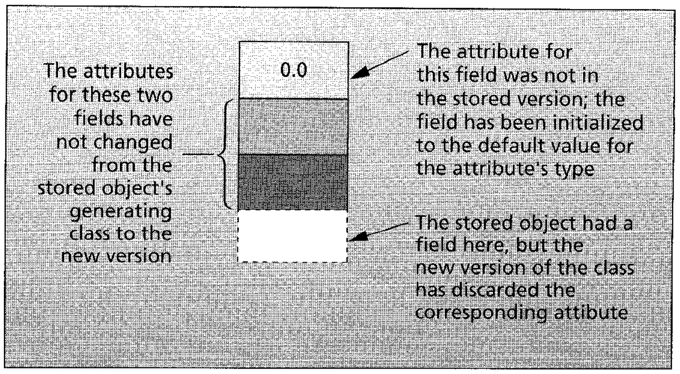
What should the default version of `correct_mismatch` do? It may be tempting, in the name of unobtrusiveness, to give it an empty body. But this is not appropriate because, by default, object retrieval mismatches will be ignored, leading to all kinds of possible abnormal behavior. The better global default is to raise an exception:

```
correct_mismatch is
       — Handle object retrieval mismatch.
    do
       raise_mismatch_exception
    end
```

where the procedure called in the body does what its name suggests. This might cause some unexpected exceptions, but it is better than letting mismatches go through undetected. A project that wants to override this default behavior can always redefine `correct_mismatch` at its own risk. If you do expect object mismatches for a certain class you can redefine `correct_mismatch` to update the retrieved object.

CORRECTION. Suppose the retrieval mechanism (through feature *retrieved* of class *STORABLE*, a database operation) has created a new object deduced from a stored object with the same generating class, but it has also detected a mismatch. The new object is in a temporary state and may be inconsistent. For example, it may have lost a field that was present in the stored object or gained a field not present in the original. Think of it as a foreigner without a visa.

Such a state is similar to the intermediate state of an object being created—apart from any persistence consideration—by a creation instruction `!! x.make ()` just after the object's memory cell has been allocated and initialized to default values, but just before `make` has been called. At that stage the object has all the required components but is not yet ready for acceptance by the community. It is the official purpose of the creation procedure *make* to override default initializations as needed to ensure the invariant.



The attributes for these two fields have not changed from the stored object's generating class to the new version
0.0
The attribute for this field was not in the stored version; the field has been initialized to the default value for the attribute's type
The stored object had a field here, but the new version of the class has discarded the corresponding attibute

Figure 2. Object mismatch.

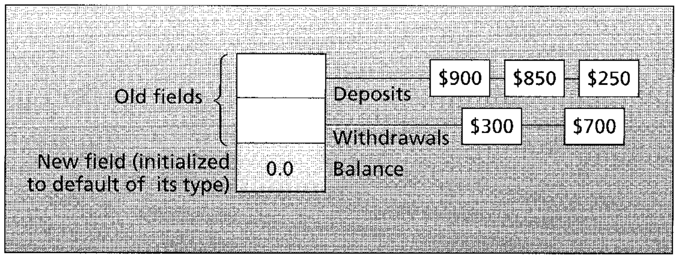


Figure 3. Retrieving an account object.

Let us assume for simplicity that the detection technique is structural and based on attributes (technique 3), although this discussion applies to other solutions, nominal or structural. The mismatch is a consequence of a change in the attribute properties of the class. We can assume it is a combination of attribute additions and removal (treating attribute replacement as a removal followed by an addition.)

Attribute removal does not raise any obvious difficulty: If the new class does not include a certain attribute, the corresponding object fields may simply be discarded. In fact, `correct_mismatch` does not need to do anything because the retrieval mechanism will have discarded them when it created a tentative instance of the new class. If you do worry that the attributes were needed, you need a more elaborate detection policy, such as technique 4.

Attribute addition is more delicate. Suppose the situation is as in Figure 2, and the new class has added an attribute that yields a new field (the top field in the object in Figure 2). We must initialize this field somehow. In the systems I have seen, the solution is to use a conventional default (usually zero for numbers, empty for strings) as an initialization value. But this may be very wrong!

Figure 3 illustrates why. Class *ACCOUNT* has the attributes *deposits_list* and *withdrawals_list*. Assume that a new version adds the attribute `balance` and that a system using this new version attempts to retrieve an instance created from the previous version. The purpose of adding the `balance` attribute is clear: Instead of recomputing an account's balance on demand, we keep it in the object and update it when necessary.

The new class invariant reflects this through a clause of the form

```
balance = deposits_list.total
 - withdrawals_list.total
```

But if we initialize `balance` as zero, it will not agree with the record of deposits and withdrawals: It should be $1,000.

Hence the importance of the `correct_mismatch` mechanism. In this case the class will simply redefine the procedure as

```
correct_mismatch is
      — Handle object retrieval mismatch by
      — correctly setting up balance
   do
      balance := deposits_list.total -
      withdrawals_list.total
   end
```

Note that if the author of the new class has not planned for this case, the default version of `correct_mismatch` will, if not redefined in *ANY* or another ancestor, raise an exception. This will cause the application to terminate abnormally unless a `retry` (another recovery possibility) handles it. This is the right outcome, because continuing execution could lead to the ultimate crime: Destroying the consistency of some objects and, worse yet, the integrity of the persistent object structure.