IDE-integrated Support for Schema Evolution in Object-Oriented Applications

Position paper

Marco Piccioni[1], Manuel Oriol[1], Bertrand Meyer[1]

[1] Chair of Software Engineering, ETH Zurich,
Clausiusstrasse 59,
8092 Zurich, Switzerland
{marco.piccioni, manuel.oriol, bertrand.meyer@inf.ethz.ch}

Abstract. When an application retrieves serialized objects for which the class has changed, it may have to cope with modifications of the semantics. While there are numerous ways to handle the resulting mismatch at runtime, the developer is typically required to provide some code to reestablish the intended semantics of the new class. The present article shows how to instruct an IDE with class version information, in a way that it can provide the developer with help and guidance for a semantically correct schema evolution.

Keywords: object oriented, schema evolution, type converter.

1 Introduction

In object-oriented applications, serializing objects (encoding them in binary or in some other format) is a widely used way of storing data. Once an object has been serialized, it can be stored on disk for later deserialization or sent remotely to other applications. When compared to a full-fledged database solution, it offers the advantage of being more lightweight but it lacks important services like transaction handling or object querying. With respect to a relational database, serializing objects or using an object-oriented database both offer one clear advantage by eliminating the well known object-relational impedance mismatch [1]. One disadvantage of the direct mapping to the persistent store is that stored objects are more tightly coupled to the application. This can be an issue when the corresponding class structure evolves over time. The system may not be able to read the previously stored objects of a class anymore because a new version of the class itself is being used. The developer has therefore to gather information on the stored class version, understand how the values of the stored objects relate to the semantics of the new class and finally provide an appropriate conversion routine.
The main idea of our approach is to make use of the converter semantics provided by the Eiffel language to provide an integration to the EiffelStudio IDE in order to give more support to the developer who needs to update his classes. Section 2 analyses three approaches to object serialization, namely Java serialization, the db4o object-

oriented database system and Eiffel serialization. Section 3 presents the approach that will be adopted for performing the updates. Eventually Section 4 describes the proposed solution and a first prototype implementation using the Eiffel language.

2 State of the art

Hereafter follow three different approaches to object serialization that we find interesting to investigate.

2.1 Java standard serialization

The Java object serialization API, a framework for serializing and deserializing objects, provides the standard wire-level object representation for remote communication, and the standard persistent data format for the JavaBeans component architecture [2]. A class can be set up for future serialization of its instances by making it implement the `Serializable` interface. As suggested by Bloch [3], serialization can be considered an extra-linguistic mechanism for creating objects, and so it is responsible for establishing the class invariant and for ensuring that no illegal access to the object is possible. When the default serialized form is not appropriate for that purpose, a custom serialized form can be implemented via methods that are reflectively invoked when a mismatch occurs.

2.2 Using an OODBMS for serialization

When using an object-oriented database like db4o to serialize our objects [4, 5], we don’t need to change the class schema by implementing an interface like `Serializable`. The objects are therefore more transparent to the persistent services. In case a custom behavior is needed to reestablish the invariant with respect to an older stored version of the object, one can either choose to use reflectively invoked methods in the object class or register listeners to specific container events. However, as newly added attributes are automatically initialized to their default values, if the developer does not foresee the possible issues and does not provide the necessary method implementation, the class invariant may silently be invalidated.

2.3 Eiffel serialization

Eiffel serialization presents a comprehensive solution based on the identification of three steps: 1) detection of version mismatches for previously stored objects, 2) notification to the system of such mismatches, and 3) safe conversion of the needed objects on demand [6, 7, 8]. A remarkable difference with the previous approaches is that as a default an exception is raised if a mismatch is detected at runtime. Custom behavior can be provided by inheriting from class `MISMATCH_CORRECTOR` and implementing the feature `correct_mismatch`.

3 Performing the updates: general approach

Programmers that work on different versions of a class have typically very little help in managing these versions. To be aware of the consequences of deserializing objects of old versions of a class they have to run a number of test cases proportional to the product of the number of stored classes and the number of releases, which can be quite large. These tests cannot be constructed automatically because, in addition to binary compatibility, one must test for semantic compatibility. To allow a higher degree of control on the schema evolution and to keep compatibility with the already existing solutions, the proposed update algorithm will first check if the retrieved version is the same as the current version. If it is not, it will check if a *converter* is available in the current class for the stored version. If it is available, it will invoke it passing the retrieved type as an argument. If a converter is not available, it will further check if the class inherits from a specific class that can help in handling the mismatch (like `MISMATCH_CORRECTOR` in Eiffel) and invoke a redefined feature (like `correct_mismatch` in Eiffel). If both the last two checks fail, it will raise an exception to state clearly that an inconsistency may happen and to stop the application before it can do any damage. Thus the algorithm takes into account different mechanisms for handling schema evolution, and assigns to them different priorities.

4 IDE support for handling schema evolution in Eiffel

To ease the task of writing the transformers and the mismatch correctors we propose an integration in the EiffelStudio IDE to make it class-version-aware and therefore capable of providing support to the developer for taking the most appropriate action. This implies augmenting the stored objects with additional meta-information about versions, to be used at retrieval time.

4.1 Type converters

Type converters have been already explored in connection with dynamic software updating [8]. The Eiffel language has a `convert` clause [9] to specify conversions from a type to another. The mechanism is used to provide a systematic way to handle the conversions between basic (or primitive) types like `INTEGER` or `REAL`. Similar conversions are supported by most programming languages in an ad hoc fashion. An important semantic constraint of this approach is that a type is considered to either *conform* (in the sense of inheritance) or *convert* to another. It is not possible to convert a type to another type with the same name, even if it has a different schema, because a type conforms to itself and the compiler would reject the conversion. As it seems reasonable to think that two versions of the same class can be considered as different types, we propose a prototype implementation that codes the class names differently between versions while being mostly transparent to the developer.

4.2 An integrated EiffelStudio GUI

The GUI should be integrated in the EiffelStudio class browser and should perform, at minimum, the following tasks: 1) enable browsing of the previous class versions, 2) enable consolidation of the current version so that it is ready to be released (saved with the updated version information), 3) record user actions, specifically the different kind of refactorings that may take place, and 4) in case of consolidation, generate converters depending on the recorded refactorings.

4.3 Implementation details

The minimum support provided should be: 1) a skeleton implementation of the converter body, 2) a check of the older version's class invariant (that could be invalidated by the new version code), and 3) a proposed default initialization for possibly added fields, if such an initialization does not invalidate the invariant. It will be the developer's responsibility to check and complete the converter implementation as this operation cannot be fully automated.

4.4 A first proof of concept

To test the idea of using converters for schema evolution we have tagged class names with version numbers and have showed with a prototype that the approach is feasible. Assignments and argument passing from the old version to the new one are also tested: http://se.inf.ethz.ch/people/piccioni/software/prototype_code.zip. Changing explicitly class names is not desirable because the new version would break all the clients that are using the old class version, and in addition a separate concern like serialization should not be so tightly coupled to the class itself via its name. The class name should in fact ideally reflect the underlying abstraction only. As mentioned earlier, we propose to use the converters in a way that they accept the same type with a different version number. To give an idea of how the converters syntax looks like, here is an extract from the prototype implementation:

```
class
        MY_SAMPLE_CLASS
create
        make,
        from_my_sample_class_v1
convert
        from_my_sample_class_v1({MY_SAMPLE_CLASS_V1})

feature -- Access
        sample_integer: INTEGER
        sample_string: STRING
        added_attribute:STRING

feature -- Conversion
        from_my_sample_class_v1(a_v1:MY_SAMPLE_CLASS_V1)
                        --the ad hoc converter
        do
                sample_integer := a_v1.sample_integer
                sample_string := a_v1.sample_string
```

```
                        added_attribute:="This string has been added"
            end
end
```

5 Conclusions and Future Work

This article shows how to provide support to developers that have to handle schema evolution in object-oriented applications. This will be achieved by integrating a module in an IDE. The work that has to be done to release a fully integrated module can be divided into several steps. The preprocessor has to be modified in order to tag class names with a version number in a transparent way. An extension to the EiffelStudio GUI as described above has to be programmed. The automatic support will be extended including the ability to serialize objects of the current version into objects of previous versions and the ability to deserialize objects of more recent versions into objects of previous versions. The automatic support will also be extended to include different kind of refactorings, like: removing, renaming, changing type or visibility of an attribute; adding, removing, renaming, changing type, visibility, return type or arguments of a routine, renaming a class and adding or removing an inheritance relationship.

References

1. Date C.: Introduction to Database Systems 8th ed. Addison Wesley (2003)
2. Gosling J., Joy B., Steel G. and Bracha G.: The Java Language Specification. 3rd ed. Addison Wesley (2005)
3. Bloch, J.: Effective Java. Prentice Hall PTR. (2001)
4. Paterson J., Edlich S., Hörning H. and Hörning R.: The Definitive Guide to db4o. Apress (2006)
5. http://www.db4o.org/community/ontheroad/apidocumentation/index.html
6. Meyer B.: Object Oriented Software Construction. 2nd ed. Prentice Hall PTR (1997)
7. Meyer B.: Eiffel: The Language. Prentice Hall (1992)
8. Neamtiu I., Hicks M., Stoyle G. and Oriol M.: Practical Dynamic Software Updating for C. ACM Conference on Programming Language Design and Implementation (PLDI). Ottawa, Canada (2006)
9. ECMA committee TC39-TG4, ECMA International standard 367. Eiffel Analysis, Design and Programming Language (2005)