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 of a class that changed, it may have to cope with modifications of the semantics. While there are numerous ways to handle the resulting mismatch at runtime, developers are typically required to provide some code to reestablish the intended semantics of the new class. We show here how to instruct an IDE with class version information, in a way that it can provide help and guidance for a semantically correct schema evolution.

Key words: object-oriented, schema evolution, type converter.

1 Introduction

In object-oriented applications, serializing objects (encoding them in binary or in some other format) is widely used to store data. Once an object has been serialized, it can be stored on disk for later deserialization or sent remotely to other applications. Serialization is more lightweight then either object-oriented and relational full-fledged database solutions. At the same time, it lacks important services like transaction handling and object querying. With respect to a relational database, both serializing objects and using an object-oriented database imply that programmers can remove the object-relational mapping layer from their application. Eliminating the well known object-relational impedance mismatch [2] has a price: the stored objects are more tightly coupled to the application, because an additional layer of indirection is missing. This can be an issue when the corresponding class structure evolves over time. The system is then typically not able to read the previously stored objects of a class anymore because a new version of the class itself is being used. Developers have 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 proposed approach to the schema evolution problem is two-fold. On the one hand we suggest a robust retrieving algorithm that, after trying different techniques to convert the old objects into the new ones, forbids the application to access potentially semantically inconsistent objects. On the other hand we propose to help the developer that works with different versions of a class and provide a solution integrated to an existing IDE.

Section 2 analyzes four approaches to object serialization, namely Java serialization, .NET serialization, the db4o object-oriented database system and Eiffel serialization. Section 3 presents the algorithm that performs the updates. Eventually Section 4 describes the proposed IDE integration and a first proof of concept implementation using the Eiffel language.

2 State of the art

This section describes four existing different approaches to object serialization: Java standard serialization mechanism, the Version Tolerant Serialization within the .Net framework, the db4o object-oriented database solution and the Eiffel serialization framework.

2.1 Java 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 [5]. A class can enable future serialization of its instances by implementing the `Serializable` interface. As suggested by Bloch [1], it is worth noticing that this deceivingly simple addition brings important constraints. In fact, every `Serializable` class has an associated unique version identification number, known as serial version UID. If one does not specify it by declaring a field named *serialVersionUID* and by explicitly giving it a value, the system automatically generates it using an algorithm that closely couples it with the class structure. The default serialized form of an object is therefore an encoding of the physical representation of the object graph rooted at the object itself. Considering that some class details may even vary depending on compiler implementations, unexpected exceptions during deserialization may happen at runtime.

More generally, by implementing the `Serializable` interface the flexibility to change the class implementation in the future significantly decreases, because all its internal representation becomes part of its exported API, thus invalidating encapsulation, one of the key principles of object-orientation. In addition to this, deserialization can be considered an extra-linguistic mechanism for creating objects, and so it should be responsible for establishing the class invariant and for ensuring that no illegal access to the object is possible.

While using the default serialized form can sometimes be appropriate, the ideal serialized form of an object should contain only the logical data represented by the object, and should be independent of the physical representation. This can be achieved by implementing a custom serialized form via methods `writeObject` and readObject, which are reflectively invoked and typically used to establish the class invariant. A developer can also implement readResolve, which creates a new object and then delegates to the constructors the task to reestablish the class invariant.

2.2 .NET serialization

The .NET framework, starting from version 2.0, provides a set of features, called Version Tolerant Serialization (VTS), which makes it easier to handle serializable types across different versions [11]. VTS comes in two flavors: binary serialization and XML (Soap) serialization. The binary serialization uses a BinaryFormatter to provide a compact and efficient byte stream for usage within the .NET framework. When an object is serialized, the name of the class, the assembly, and all the data members are serialized. A requirement placed on the serialized object and all the referenced objects in the object graph is that the corresponding classes have to be tagged with the Serializable attribute. As the Serializable attribute is not inherited, the serializable status for a class that inherits from an existing serializable class must be stated explicitly. If a data member should not be serialized, it is also possible to tag it using the NonSerializedAttribute attribute. A significant advantage to using attributes for events as opposed to using interfaces is that the event mechanism is decoupled from the class hierarchy.

If portability across different platforms is needed, XmlSerializer can be used instead. This only serializes public properties and fields.

The .NET framework handles schema evolution issues as follows: when an object of an old version of a class retrieves an object of a newer version with an added data member, the mechanism ignores the latter. When an object of a new version of a class with an added data member retrieves an object of an older version of the class, it does not throw an exception if the new data member is tagged as “optional” with the OptionalFieldAttribute attribute.

It’s worth noticing that the constructors of an object are not automatically called during the deserialization process, so implicit class invariant violations may happen if the developer does not take appropriate actions. In these situations the developer may adopt a custom serialization by implementing reflectively invoked methods that provide hooks into the serialization/deserialization process. For example, to provide an ad hoc initialization to a newly added field, one must create a method that accepts as an argument a StreamingContext and apply to it the OnDeserializingAttribute attribute.

2.3 Using an OODBMS for serialization: db4o

When using an object-oriented database like db4o to serialize objects [3, 10], we neither have to change the class schema by implementing an interface like Serializable nor have to tag the class with a Serializable attribute. The db4o container, ObjectContainer, takes care of providing all the needed persistence services. It receives each object as an argument and stores it as-is. The increased transparency, the possibility to have services like transaction handling, object browsing, native querying, and a very small memory footprint, suggest that this solution can be considered an overall better alternative to pure object serialization for both Java and .NET.

Regarding schema evolution handling, in case the developer needs a custom behavior to reestablish the invariant with respect to an older stored version of the object, there are two possibilities. He can either choose to use reflectively invoked methods like `objectOnActivate` in the object class or, even better, can register listeners to specific `ObjectContainer` events outside the object class. In the latter case, when the container "activates" an object after retrieval, it invokes the method `onEvent`, passing to it the newborn object as an argument, so that it can be properly initialized.

An alternative and interesting scenario occurs when the developer does not foresee the possible issues and "forgets" to code appropriate methods to handle the conversion. Unfortunately, in this case the newly added attributes are automatically initialized to their default values. This is an excessively confident level of transparency, because it may lead to introduce in the system objects whose class invariants may not be valid anymore.

2.4 Eiffel serialization

Eiffel serialization [4, 6, 7] presents a 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.
3. Safe conversion of the needed objects on demand.

The implementation of this solution is similar to the one previously reviewed for Java, except for the fact that Eiffel does not need interfaces, because it supports multiple inheritance. Custom behavior can therefore be provided by inheriting from class `MISMATCH_CORRECTOR` and redefining the reflectively invoked feature `correct_mismatch` to establish the class invariant. It is also worth mentioning that in Eiffel an invariant violation is very easy to detect, because the language provides embedded support for Design by Contract providing an explicit way to declare the class invariant itself in the class text via the `invariant` clause.

The Eiffel serialization mechanism cannot be defined as fully "tolerant" either. In fact an exception is raised as a default if a mismatch is detected at runtime and no redefinition of the feature `correct_mismatch` is found. This implementation choice makes therefore impossible for an inconsistent object to be accepted in the system after deserialization because a developer happened to forget to explicitly take care of the conversion.

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 numerous 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 consists in several steps, illustrated in the synthetic flow-chart in Fig. 1.

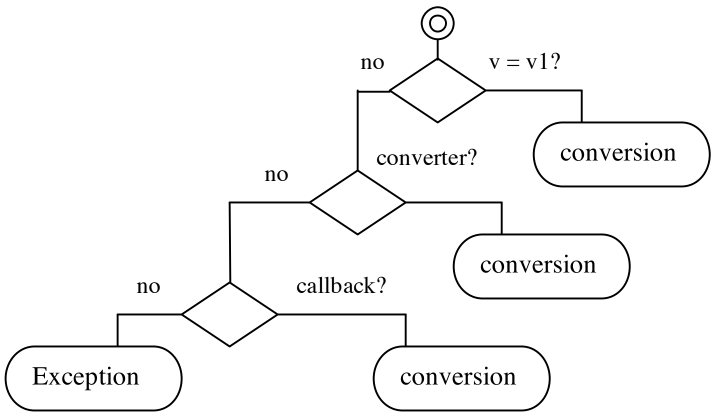

Fig. 1. Flow-chart of the proposed update algorithm from an old class version v1 to a new class version v.

The algorithm first checks 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 (see 4.1). If it is available, it will invoke and pass 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 raises an exception to state clearly that an inconsistency may happen and to stop the application before the inconsistent object can do any damage. Thus the algorithm takes into account several 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 type converters 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

Neamtiu et al., in their work on dynamic software evolution [9], proposed to use type converters to update data types at runtime. Their solution relies on heuristics that builds semi-automatically the converters from static analysis of the code and its instrumentation. The Eiffel language has a `convert` clause [4, 8] to specify conversions from a type to another. The mechanism is already used to provide a systematic way to handle conversions between basic (or primitive) types like `INTEGER` or `REAL`, or between `STRING` types as represented in Eiffel and .NET. Similar conversions are supported by most programming languages in an ad hoc fashion. While the basic idea to provide conversion functions that take care of the conversion details is well known, having the converters embedded in the language provides the advantage of a coherent framework that takes care of the conversions at runtime without additional instrumentation.

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. As a type obviously conforms to itself, it is not possible to convert a type to another type as-is. This happens because the two types retain the same name, even if they have a different schema, so the compiler would reject the conversion.

Our assumption is that two versions of the same class are different types. With this in mind, we propose a prototype implementation that is mostly transparent to the developer and generates different names for different versions of the class to use the converter mechanism.

4.2 An integrated EiffelStudio GUI

The main intent of our proposal is to guide the developer while delivering a new version of a class as version-aware as possible. This means that the class, once consolidated, knows how to handle all the possible type conversions that may be necessary in the future.

Both type converters and the update algorithm shown in figure 1 are the backbone of a semantically consistent framework for schema evolution.

The GUI is fully integrated in the EiffelStudio class browser. It is the presentation layer of the proposed framework, and fosters interaction between the developer and the underlying mechanism. It performs the following basic tasks:

1. Enables browsing of all the previous class versions.

2. Enables consolidation of the current version so that it is ready to be released (it saves it with the updated version information).
3. Proposes different code templates for the type converters bodies, depending on previously recorded refactorings.
4. Issues an appropriate warning, stating that conversions from older versions to newer ones will not be possible anymore at runtime, if the developer refuses to take appropriate action to handle the conversion.

4.3 Implementation details

Before realizing all the steps illustrated above, it is necessary to make a preprocessor in order to tag class names with a version number in a transparent way.

In addition, the GUI backend has to:

1. Record developer's actions, more specifically the different kind of refactorings that may take place.
2. Associate the different refactorings to the different versions.
3. Read the recorded refactorings for the current version in case of consolidation.
4. Generate different code templates for the type converters bodies, depending on the recorded refactorings.
5. Consolidate the converters depending on the developer's choice.

As the overall effort is non-trivial, it is necessary to separate the intended task into different steps, undertaken in an iterative fashion. As a minimum support, the framework does the following:

1. Provides a skeleton implementation of at least a converter body.
2. Performs a test of object creation by checking a possible violation of the current class schema with the invariant of the old version, looking for a possible violation. Issues a warning if a violation occurs.
3. Suggests an initialization that does not invalidate the new invariant for added fields.
4. Issues a warning to the developer, suggesting that it is his responsibility to check and complete the converter implementation, as this operation cannot be fully automated.

In addition to handling the inclusion of a new attribute in the class schema, an extension of the support can include different kinds of refactoring, like:

- Refactorings on data fields:
    - Removing an attribute.
    - Renaming an attribute.
    - Changing an attribute type.
    - Changing an attribute visibility.
- Refactorings on routines:

    - Adding a routine.
    - Removing a routine.
    - Renaming a routine.
    - Changing a routine return type.
    - Changing a routine visibility.
    - Changing the type of arguments of a routine.
    - Changing the order of arguments of a routine.
- Refactorings on classes
    - Renaming a class.
    - Adding an inheritance relationship.
    - Removing an inheritance relationship.
    - Changing the type of a generic parameter.
    - Changing the constraint of a generic parameter.

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.

We therefore propose to introduce version numbers to the serialized objects and then use the converters in a way that they accept the same type with a different version number.

To give an idea of the converters syntax, we hereafter show 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

We have shown how to provide better support for handling schema evolution in object-oriented applications. This can be achieved using a two-sided approach. On the one hand we instruct the system to record specific refactorings across different class versions so that it can propose reasonable templates for the converters. On the other hand we propose to improve the developer's interaction with the system by integrating a module in an existing IDE. To release a fully integrated module we need to:

- Adapt the embedded converter mechanism in the Eiffel language so that it can accept to convert two different versions of the same class.
- Program an extension to the EiffelStudio GUI that enables browsing of all the previous class versions, saves them with the updated version information, proposes different code templates for the type converters bodies (depending on previously recorded refactorings).
- Program an extension to the framework that can include different kinds of refactoring.

This is what we are currently implementing.
In the future, the automatic support can also be further extended including:

- The ability to serialize objects of the current version into objects of previous versions.
- The ability to deserialize objects of more recent versions into objects of previous versions.

References

1. Bloch, J.: Effective Java. Prentice Hall PTR. (2001)
2. Date C.: Introduction to Database Systems 8th ed. Addison Wesley (2003)
3. db4o object oriented database API documentation, http://www.db4o.org/community/ontheroad/apidocumentation/index.html
4. ECMA committee TC39-TG4, ECMA International standard 367. Eiffel Analysis, Design and Programming Language (2005)
5. Gosling J., Joy B., Steel G. and Bracha G.: The Java Language Specification. 3rd ed. Addison Wesley (2005)
6. Meyer B.: Object Oriented Software Construction. 2nd ed. Prentice Hall PTR (1997)
7. Meyer B.: Eiffel: The Language. Prentice Hall (1992)
8. Meyer B.: Conversions in an Object-Oriented Language with Inheritance, in JOOP (Journal of Object-Oriented Programming), vol. 13, no. 9, January 2001, pages 28-31.
9. 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)
10. Paterson J., Edlich S., Hörning H. and Hörning R.: The Definitive Guide to db4o. Apress (2006)
11. Version Tolerant Serialization, http://msdn2.microsoft.com/en-us/library/ms229752.aspx