An IDE-based solution for schema evolution of object-oriented software

An IDE-based Solution for Schema Evolution of Object-Oriented Software

Marco Piccioni
ETH Zurich, Switzerland
marco.piccioni@inf.ethz.ch

Manuel Oriol
University of York, United Kingdom
manuel@cs.york.ac.uk

Bertrand Meyer
Teseo Schneider
ETH Zurich, Switzerland
bertrand.meyer@inf.ethz.ch
teseos@student.ethz.ch

Abstract

Most large software systems rely on extensive amounts of persistent data — objects. Most large software systems last a long time, over which they need to change program elements — classes. Inevitably, these two characteristics clash: how do we retrieve previously "persisted" objects when the classes that describe them have changed? Naive solutions, such as initializing new fields to default values, are dangerous, since they risk invalidating the consistency of objects. For example it would be wrong for a bank account class to initialize a newly added "balance" field, in every retrieved object, to zero. Practical considerations also apply: we cannot force the code for each class to retain all its previous versions.

The ESCHER framework addresses these theoretical and practical issues through an IDE-based approach, guiding the developers interactively in the process of defining object adaptation on a class-by-class basis, respecting consistency requirements expressed by class invariants. Based on the conversions specified in this process, an algorithm will transform objects as needed, enforcing class invariants to prevent the creation of any corrupt objects. The presentation describes the principles behind invariant-safe schema evolution, the design and implementation of the ESCHER system, and the experience so far.

Categories and Subject Descriptors D.2.6 [*Programming environments*]: Integrated environments; D.2.3 [*Coding tools and techniques*]: Object-Oriented programming

General Terms Algorithms, Languages, Management, Reliability

Keywords Versioning, Refactoring, Persistence, Serialization, Object-Oriented Schema Evolution, IDE integration

1. Introduction

While coding a class in an object-oriented language, programmers usually provide a partial implementation of a certain semantic model. The implementation can be considered "partial" because, most of the time, writing complete formal specifications is impossible. In addition, the underlying assumptions about the outer world are poorly, if at all, specified. As time passes, the initial assumptions may change, and the software may also change to adapt to new requirements. Both issues are possible causes for software aging [1].

The most widespread approach to handling object-oriented schema evolution relies heavily on class developers: they have to provide conversion code to import older object schema versions into the new ones. To make an informed decision, developers must have the previous versions of the class available and prepare conversion code for all versions of the class. Designing such code is not trivial. For example, the conversion code might be cluttered in the class whose objects need to be stored, or alternatively scattered in different, unrelated classes.

Even when the code is available, issues may still arise: many currently available persistence solutions are coupled with tolerant retrieval algorithms, automatically making questionable decisions about the initialization of the retrieved objects. As a consequence, and in the name of a claimed "transparency" of the process, they enforce silent acceptance of possibly inconsistent objects into the system. This is a recipe for disaster, as class invariants may be silently violated. Following Bloch [19], 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.

To tackle the aforementioned issues, we first study how serializable classes from the widely used `java.util` package evolved from Java 1.2.2 to Java 6.0. It appears that

17.8% of the changes have impacted the persistence of the serializable classes of the package.

This allows to identify the most frequently occurring refactorings and to propose a formal model that represents them and the corresponding transformation functions generated from the suggested heuristics.

We then propose to shift the focus of developers from runtime to development time by introducing ESCHER (Eiffel SCHema Evolution suppoRt), a modified EiffelStudio IDE.

The tool aims at guiding and supporting developers through the schema evolution process at class release time, and includes version and release handling, code template generating features for the transformation functions, and support for custom serialization.

In addition we provide, as an external deserialization library complementary to the tool, a robust retrieval algorithm to prevent acceptance of inconsistent objects into the system.

Section 2 presents the analysis of some newly collected data about actually performed refactorings. Section 3 describes the model for software updates. Section 4 details the tool implementation with respect to both IDE integration and the retrieval algorithm. Section 5 analyzes contributions and limitations of the current approach. Section 6 summarizes the previous approaches, from both the authors and others. Finally, section 7 describes our conclusions and future work.

2. Schema Evolution in Practice

Advani et al. [14, 15] already made studies about refactorings[1] by evaluating refactorings made in fifteen open source Java systems. These studies show that refactorings like "rename field" and "move field", together with "rename method" and "move method", account for approximately 66% of the total refactorings identified. We can also observe that the "rename field" and "move field" refactorings alone account for 32% of the total refactorings identified by the authors. While these results are calculated on all classes of the considered systems, there is no evidence that classes meant to be persistent would exhibit the same characteristics.

To verify that persistent classes evolve in a similar manner, we consider the serializable classes from the the Java package `java.util`. Analyzing Java code was a conscious choice as classes that might have their instances serialized implement the `Serializable` interface. It would have been much more difficult to run a similar study on Eiffel code because it does not include such an easy marker. The package `java.util` was chosen because contains classes whose instances are likely to be serialized. In fact the package contains classes that model collections, dates, currencies, and locales. We considered the twenty-two classes in the package directly implementing the *Serializable* interface — whose instances programmers are encouraged to serialize — and analyze them manually across five major language versions: 1.2.2, 1.3.1, 1.4.2, 5.0, 6.0. Furthermore, we take into consideration 22 refactoring types, ten of which directly relevant to the serialization process. These 22 refactorings are shown in Table 1 and were derived by using a systematic approach that considered possible changes (addition, removal, modification) over possible targets (attribute, visibility, methods etc). While the complete raw data are available for download [16], we present here the most relevant aggregated results.

The list of refactorings we considered as relevant for Java persistence is the following:

- Attribute added
- Attribute removed
- Attribute renamed
- Attribute type changed
- Attribute initialization value changed
- Attribute becoming a constant
- Constant becoming an attribute

These refactorings, actually a superset of the the usually considered ones [36], aim at expressing more precisely the semantics of evolving classes whose objects are intended to be serialized. The "Attribute initialization value changed" refactoring is considered because it may influence the class invariant. The "Attribute becoming a constant" and its counterpart "Constant becoming an attribute" are relevant because constants are typically not serialized. In line with what discovered earlier, Table 1 shows that the persistence-related refactorings consist of 17.8% of the total number of refactorings detected.

Table 2 shows how refactorings are distributed across different classes and across all five language versions analyzed. The data suggest that persistence-related refactorings are sufficiently widespread among the classes, and confirm the supposition that persisted data might actually change significantly over time.

Table 3 shows the number of refactorings per kind, considered across all versions. Note that "Attribute added" and "Attribute removed" together constitute 74% of all persistence-related refactorings.

What this short study shows is that classes whom instances might be serialized change over time. Moreover 17.8% of these changes directly impact the capability of classes to deserialize the instances of their previous versions. This is aggravated by the fact that there may be a high number of stored objects that need to be updated. This does not directly imply that deserializing will be performed in a semantically inconsistent way, but it is likely that it will create issues at some point.

For example, consider a class *BANK_ACCOUNT* that computes the balance on demand by subtracting the total

[1] In this article, consistently to the work from Advani et al. [14, 15] we use the term *refactoring* for any minor modifications of the code rather than semantics preserving modification-only — which is the meaning that popular IDEs encourage.

	Persistence-related							Non-persistence-related														
	attributes added	attributes removed	attributes renamed	attributes type changed	attributes values changed	attributes becoming constants	constant becoming an attribute	attributes visibility changed	methods added	methods removed	method definition changed	interfaces added or replaced	inner classes added	inner classes removed	inner classes modified	volatile marker added	volatile marker removed	synchronized clause added	synchronized clause removed	static initializer added	static initializer removed	generics clauses added
1.2.2 - 1.3.1	4	0	0	0	1	0	0	1	6	0	1	0	3	0	0	0	0	0	0	0	0	0
1.3.1 - 1.4.2	10	13	1	1	0	0	1	5	53	5	1	2	9	4	0	5	0	6	7	0	2	0
1.4.2 - 5.0	29	6	0	13	2	4	0	1	42	8	151	1	1	0	28	1	0	2	2	0	0	33
5.0 - 6.0	14	9	4	1	0	2	0	1	81	27	13	3	11	5	11	0	0	0	0	1	1	0
All versions	57	28	5	15	3	6	1	8	182	40	166	6	24	9	39	6	0	8	9	1	3	33
By category	115 (17.8%)							534 (82.2%)														
Total	649																					

Table 1. Refactorings found across 5 versions of java.util package.

Class	Refact.	Persistence-related	%
ArrayList	18	2	11.1
BitSet	42	8	19.0
Calendar	52	24	46.2
Currency	3	2	66.7
Date	22	9	40.9
EnumMap	1	0	0.0
EnumSet	1	0	0.0
EventObject	1	1	100.0
HashMap	101	11	10.9
HashSet	9	2	22.2
HashTable	41	5	12.2
IdentityHashMap	20	2	10.0
LinkedHashSet	5	1	20.0
LinkedList	50	1	2.0
Locale	34	16	47.1
PriorityQueue	17	1	5.9
Random	13	7	53.8
TimeZone	28	7	25.0
TreeMap	122	13	10.7
TreeSet	39	3	7.7
UUID	0	0	0.0
Vector	30	0	0.0

Table 2. Refactorings found across 5 versions, divided by class and kind.

amount of withdrawals from the total amount of deposits. In a later version there could be the decision to store the current balance in an attribute instead. All the previously stored objects have to be adapted at retrieval time in a semantically consistent manner. This means that the new class invariant *balance = total_deposits - total_withdrawals* should hold immediately after retrieval. Also, initializing the balance attribute to zero for all retrieved objects is probably wrong.

Refactoring type	Refact.	% of persistence-related
Attribute added	57	49.7%
Attribute removed	28	24.3%
Attribute renamed	5	4.3%
Attribute type changed	15	13.0%
Attribute value changed	3	2.6%
Attribute to constant	6	5.2%
Constant to attribute	1	0.9%
Total	115	100.0

Table 3. Persistence-related refactorings, across all versions, divided by refactoring kind.

In the next section, we show a model based on this analysis that allows to represent refactorings and to generate converters.

3. Model

Following the work of Boyapati *et al.* [2] as well as Franconi *et al.* [3] we develop here a model of updates. While modeling changes, we especially emphasize the generation of conversion functions.

Figure 1 presents a simplified syntax of class definitions in Eiffel programs. In particular, we omit the declaration both of routines and constraints on generic parameters as they are not included in the serialized form. Another point of interest is that we do not consider inheritance at all because we have access to the flattened version of the class. This is a valid assumption as serialized objects are *de facto* flattened as well.

$$
\begin{array}{lcll}
C, D \in Names & & & class\ names \\
N, M \in GNames & & & names\ for\ generic\ parameters \\
name \in names & & & names\ for\ attributes \\
att & ::= & name : type & attribute\ definition \\
 & | & name : N & attribute\ of\ generic\ type \\
type & ::= & C & class\ type \\
 & | & type[type] & generic\ derivation \\
class & ::= & \textbf{Class}\ C\ \textbf{feature}\ att_1, \ldots, att_n\ \textbf{end} & class \\
 & | & \textbf{Class}\ C[N_1, \ldots, N_i]\ \textbf{feature}\ att_1, \ldots, att_n\ \textbf{end} & generic\ class
\end{array}
$$

Figure 1. Syntax of data parts of classes and types.

The following subsections define refactorings and class transformations, how to generate a type transformer from a class transformation, and possible strategies to extract type transformers.

3.1 Classifying software evolution: refactoring units

We call *refactoring* a basic transformation applied to a class. A refactoring is a function modifying at most one attribute:

$$R : class \mapsto class$$

Figure 2 shows the definition of the five kinds of refactoring.

Taking into account the data analysis in the previous section, we define five standard refactorings that our system recognizes:

- Attribute not changed
- Attribute added
- Attribute renamed
- Attribute type changed
- Attribute removed

They are sufficient to semantically include all the refactorings listed in the previous section because of the three missing refactorings will be treated as follows:

- "Attribute value changed" will be taken into account when evaluating the new class invariant clause, so no special action is needed.
- "Attribute to constant" will be considered as "Attribute removed", as constants are not serialized.
- "Constant to attribute" will be considered as "Attribute added" for the same reason as above.

A *class transformation* $T_{R_1,\ldots,R_N}$ going from one version of a class to another can then be described by a list of refactorings $R_1, \ldots, R_N$ ($N \geq 1$):

$$T_{R_1,\ldots,R_N} : class \mapsto class$$

such that

$$T_{R_1,\ldots,R_N}(class_0) = (R_n \circ \ldots \circ R_1)(class_0)$$

Note that any modifications to the attributes of a class can be described by some class transformation. In other words, class transformations are complete with respect to attribute modifications.

This can be seen easily: suppose we have two classes C1 and C2 with n attributes and m attributes respectively, and assume the generic parameter list is preserved between C1 and C2. Then there always exists a class transformation from C1 to C2 that first deletes all the n attributes from C1 and then adds all the m attributes to C2.

While the decomposition just used is valid, it is not always useful to capture the entire semantics of successive refactorings on attributes. This is why we need to devise some heuristics to complement it and make it useful for being used in an implementation.

3.2 Object transformers

While refactorings allow an easy representation of the static transformations of a class, this is not sufficient to generate the object transformers that create an instance of the new class from a serialized instance of the old class. The main reason is that there is a need for an explicit default initialization of new attributes. To define such initializations, the programmer actually needs to input default initialization values. To represent that, we use the token $\mathcal{I}[\![\cdot]\!]$ (for input), evaluating into the next value in a list of inputs. We also add an output generator $\mathcal{W}[\![\cdot]\!]$ (for warning), warning programmers of the impossibility of generating a translation or a removal operation and evaluates into **noop**.

Figure 3 presents a succint syntax of object transformers. The generation of object transformers from a class transformation can be expressed as transformation functions as shown in figure 4. The transformation function $\mathcal{G}[\![\cdot]\!]$ takes a class transformation and generates the associated object transformer.

3.3 Heuristics for schema evolution refactorings

While we are currently considering the extraction of transformations based on actual changes made by programmers using the IDE, the current solution relies on statically comparing the abstract syntax trees (AST) of both classes to de-

$$att_i \in class_0;\ R_{noChange(att_i)}(class_0) = class_0$$

$$att \notin \{att_1, \ldots, att_n\};\ R_{new(att)}(\textbf{Class}\ C \ldots \textbf{feature}\ att_1, \ldots, att_n\ \textbf{end}\ ) = \textbf{Class}\ C \ldots \textbf{feature}\ att_1, \ldots, att_n, att\ \textbf{end}$$

$$\frac{\begin{array}{c}(att_i = name : N \wedge att_i' = name' : N) \vee (att_i = name : type \wedge att_i' = name' : type) \\ att_i \in class_0;\ class_0 = \textbf{Class}\ C \ldots \textbf{feature}\ att_0, \ldots, att_n\ \textbf{end} \\ class_1 = \textbf{Class}\ C \ldots \textbf{feature}\ att_0, \ldots, att_{i-1}, att_i', att_{i+1}, \ldots, att_n\ \textbf{end}\end{array}}{R_{nameChange(name,name')}(class_0) = class_1}$$

$$\frac{\begin{array}{c}type \neq type';\ att_i = name : type \wedge att_i' = name : type' \\ att_i \in class_0;\ class_0 = \textbf{Class}\ C \ldots \textbf{feature}\ att_0, \ldots, att_n\ \textbf{end} \\ class_1 = \textbf{Class}\ C \ldots \textbf{feature}\ att_0, \ldots, att_{i-1}, att_i', att_{i+1}, \ldots, att_n\ \textbf{end}\end{array}}{R_{typeChange(name:type,name:type')}(class_0) = class_1}$$

$$\frac{\begin{array}{c}att_i = name : type \\ att_i \in class_0;\ class_0 = \textbf{Class}\ C \ldots \textbf{feature}\ att_0, \ldots, att_n\ \textbf{end} \\ class_1 = \textbf{Class}\ C \ldots \textbf{feature}\ att_0, \ldots, att_{i-1}, att_{i+1}, \ldots, att_n\ \textbf{end}\end{array}}{R_{removedAttribute(name)}(class_0) = class_1}$$

Figure 2. Five kinds of refactorings.

$otname \in OTNames$			*object transformers names*
e	::=	$e.e \mid$ **oldc** $\mid name$	*expressions*
$instruction$	::=	**Result.**$name$**:=**$e \mid$ **noop** $\mid otname(e, e)$	*instructions*
$instructions$	::=	$instruction \mid instruction; instructions$	*list of instructions*
ot	::=	$otname(oldc : oldC) : C\ instructions$	*object transformer*

Figure 3. Syntax of object transformers.

$$\mathcal{G}[\![\cdot]\!] : (class \mapsto class) \mapsto ot$$
$$\mathcal{G}[\![T_{R_1,\ldots,R_n}]\!] = otname(oldc : oldC) : C\ \mathcal{S}[\![T_{R_1,\ldots,R_N}]\!]$$

$$\mathcal{S}[\![\cdot]\!] : (class \mapsto class) \mapsto instructions$$

$\mathcal{S}[\![R_{noChange(att_i)}]\!]$	=	**Result.**$name_i$**:=oldc.**$name_i$
$\mathcal{S}[\![R_{new(att)}]\!]$	=	**Result.**$name$**:=**$\mathcal{I}[\![\cdot]\!]$
		$where\ att = name : \ldots$
$\mathcal{S}[\![R_{nameChange(name:type,name':type)}]\!]$	=	**Result.**$name'$**:=oldc.**$name$
$\mathcal{S}[\![R_{typeChange(name:type,name:type')}]\!]$	=	**Result.**$name$**:=oldc.**$name$
		$if\ type'\ is\ assignable\ to\ type$
	=	**Result.**$name$**:=**$otname$(**oldc.**$name$)
		$if\ there\ is\ an\ object\ transformer\ otname$
		$transforming\ an\ object\ of\ type\ type$
		$into\ an\ object\ of\ type\ type'$
	=	$\mathcal{W}[\![name_0]\!]\ otherwise$
$\mathcal{S}[\![R_{removeAttribute(name_0)}]\!]$	=	$\mathcal{W}[\![name_0]\!]$
$\mathcal{S}[\![T_{R_1,\ldots,R_n}]\!]$	=	$\mathcal{S}[\![R_1]\!]; \mathcal{S}[\![T_{R_2,\ldots,R_N}]\!]$

Figure 4. Object transformers generation.

tect refactorings. This implies that in certain situations, like the "Attribute renamed" refactoring below, the outcome of the tool can suggest a possibility, not the certainty, that the specific refactoring has happened. The extraction relies on a set of heuristics successively applied:

1. An attribute that does not change name and declared type generates an "Attribute not changed" refactoring.
2. An attribute of the new version for which a counterpart with the same name cannot be found in the old version generates an "Attribute added" refactoring.
3. An attribute *att* in the old version that does not have a counterpart with the same name in the new version while having at least a counterpart *att'* with the same type in the new version is a candidate for generating an "Attribute renamed" refactoring. Notice that in general it is not possible, by only comparing the two classes AST's, to determine what happened in this specific case.
4. An attribute that does not change name but changes type between two versions generates an "Attribute type changed" refactoring.
5. An attribute of the old version that cannot find a counterpart in the new version generates an "Attribute removed" refactoring.

To detect the refactorings we iterate trough the new class attributes, search for a correspondence with attributes in the old class and create a corresponding heuristic. We then repeat the process staring from the old class, in order to gather more information, for example to find all the attributes that were removed.

Here is an example that can help clarifying the code generation step. For the class *BANK_ACCOUNT* in Figure 5, we only show the code that is useful for our purpose. Notice that the balance is not stored as such but computed on-demand.

We will suppose now that the following happens:

- For efficiency reasons we decide to change the implementation of the query *balance* to use memory instead of computation. So a new attribute *balance* is created. It will be updated every time a deposit or a withdraw will take place. Clients are not affected in this case, because in Eiffel both attributes and functions can be accessed in the same way from outside the class, providing uniform access to the class itself.
- We also decide to store the attribute *info* in a more appropriate numeric field.

The resulting class is illustrated in Figure 6: The conversion code will be generated in an ad-hoc generated class (see Figure 7), possibly containing all the conversion functions necessary for converting objects of different versions into one another.

Note that:

```
class
  BANK_ACCOUNT
inherit
  VERSIONED_CLASS

create make

feature -- Version implementation

  version: INTEGER
      -- Class version number.
    do
      Result := 1
    end

feature -- Initialization

  make
      -- Creation feature.
    do
      tot_deposits := 1
    end

feature -- Status report

  balance: INTEGER
      -- Account balance.
    do
      Result := tot_deposits - tot_withdrawals
    end

  info: STRING
      -- Some numeric information.

feature -- Basic operations

      -- omitted

feature {NONE} -- implementation

  tot_deposits : INTEGER
      -- Total amount deposited.

  tot_withdrawals : INTEGER
      -- Total amount withdrawn.

invariant
  valid_account: tot_deposits > tot_withdrawals
end
```

Figure 5. *BANK_ACCOUNT*, version 1.

- The call *set_conversion_function_(1, 2, v1_to_v2)* is necessary to save the information that a conversion function between versions one and two now exists.
- The code generation for the "Attribute type changed" refactoring does not require further human intervention.
- The initialization value for *balance* cannot be guessed by the tool because the class semantics is involved. In this case even if a lazy developer does not fix the wrong default provided, it will trigger an invariant failure at runtime.

```
class
  BANK_ACCOUNT
inherit
  VERSIONED_CLASS

create make

feature -- Version implementation

  version: INTEGER
      -- Class version number.
    do
      Result := 2
    end

feature -- Initialization
  make
      -- Creation feature.
    do
      tot_deposits := 1
    end

feature -- Status report

  balance: INTEGER
      -- Account balance.

  info: INTEGER
      -- Some numeric information.

  feature -- Basic operations

      -- omitted

feature {NONE} -- implementation

  tot_deposits : INTEGER
      -- Total amount deposited.

  tot_withdrawals : INTEGER
      -- Total amount withdrawn.

invariant
  valid_account: tot_deposits > tot_withdrawals
end
```

Figure 6. *BANK_ACCOUNT*, version 2.

The next section describes how we implemented this model and how it is integrated in the EiffelStudio IDE.

4. Implementation

Implementing a tool that uses the algorithm and model presented in the previous section makes it usable in practice. Starting from the assumption that a completely automated schema evolution implementation of the tool would be impossible, we choose to help developers in generating complete transformers when information is missing. The ESCHER tool has the following characteristics:

- It is integrated to the EiffelStudio IDE.
- It is cohesive with respect to schema evolution code.
- It is flexible with respect to storage.

```
class
  BANK_ACCOUNT_SCHEMA_EVOLUTION_HANDLER
inherit
  SCHEMA_EVOLUTION_HANDLER

redefine make end

create make

feature -- Initialization
  make
    -- Creation feature.
  do
    Precursor {SCHEMA_EVOLUTION_HANDLER}
     set_conversion_function_ (1, 2, v1_to_v2)
  end

feature {NONE} -- implementation

  v1_to_v2: DS_HASH_TABLE[TUPLE[LIST[STRING],
    FUNCTION[ANY, TUPLE[LIST[ANY]],ANY]], STRING]
      -- Conversion table from version 1 to 2
    local
      tmp:
          SCHEMA_EVOLUTION_DEFAULT_CONVERSION_FUNCTIONS

    do
      -- Auto-generated code.
      -- Attribute type change detected.
      create tmp
      create Result.make_default
      Result.force (tmp.variable_changed_type ("info", agent tmp.
          to_integer (?)), "info")
      -- Attribute type change performed.
      -- Attribute added detected.
      -- Please check the class invariant: it may be important to
          assign a meaningful value to the attribute
      Result.force (tmp.variable_constant (0, "balance"))
      -- Attribute added performed.
      -- Auto-generated code.
    end
end
```

Figure 7. *BANK_ACCOUNT_SCHEMA_EVOLUTION_HANDLER*.

The IDE integration stresses the importance we give to schema evolution handling, an activity we suggest being more effectively performed at development time.

ESCHER has high cohesion with respect to schema evolution code because all the conversion code necessary for migrating instances between different versions of the a given class is located in only one ad-hoc handler class, different from the class itself.

Flexibility with respect to storage is implemented by allowing developers to choose between the "physical" and a "logical" representation of an object. The "physical" representation encodes every object detail appearing in the corresponding class structure. This is clearly more sensitive to the smallest modifications of the structure itself. However, there may be information that is not useful, safe, or efficient to store. As suggested by Bloch [19], the ideal serialized form of an object contains only the logical data represented by

the object. It is independent of the physical representation. Storing all the details of a class structure should always be possible and it may be an acceptable default, but it should also be possible to define a customized serialized form. The proposed implementation allows programmers to choose a customized serialized form by selecting the appropriate attributes.

Though our approach can be applied to any object-oriented programming language providing support for storing and retrieving objects, we use Eiffel, and therefore the EiffelStudio IDE, mainly because of its integrated support for Design by Contract, and in particular for class invariants. Class invariants occupy a very important role in complementing the work done by the code generator. Other language characteristics have proved useful, though not essential: agents (closure-like operation wrappers), as seen in the example from the same section, have allowed passing around transformation functions as arguments of other functions. Finally, multiple inheritance has helped after we decided that using inheritance was a reasonable choice for both implementing versions and filters (custom serialization). Because of this choice we did not have to worry about having only one free slot for inheritance and consequently having to look for alternative solutions [6, 7, 8, 9].

With respect to the three principles of orthogonal persistence [22, 24], our approach, when applied to the Eiffel language, scores as follows:

- Type Orthogonality: in Eiffel every type is based on a class, so there is no special treatment for the so-called "primitive types" when it comes to persistence handling. Therefore every object has the same right to persist.
- Persistence by Reachability: the whole object graph starting from the chosen root object is always stored and retrieved, so this is achieved.
- Persistence Independence: by introducing versioning for all classes, all code is potentially considered as if operating on long-lived data. The objects, though, are not stored automatically and transparently but on demand. Complete persistence independence is not achievable because there can be semantic changes across different versions of the same class.

The detailed documentation about the mechanism, the source code and the executables for the tool are available for download on the ESCHER project page [11].

4.1 Version handling

A first step is to make the tool able to manage different class versions. To achieve that, we require that the class implements an ad-hoc query returning the version number. This is consistent with the Java standard way of indicating the version of a serializable class, but only if java developers provide their own serial version unique identifier, not relying on the automatically generated one which, as already seen, closely mimics the class structure. In the case of our Eiffel implementation, the class whose objects need to be persisted is required to inherit from *VERSIONED_CLASS*, which provides one deferred (abstract) query whose implementation, in the descendants, will in turn return the version number. In Java, this corresponds exactly to having to implement an interface having a method that takes no argument and returns the version number.

Other approaches would have been possible: for example appending the version number to the class name or adding a version number as meta-information to the class. These are however not as flexible to access the version as the one that was chosen because they require to use reflection or post-processing, which might impact performances negatively. By using inheritance we suggest two things:

- The version information should become an essential, structural part of the class itself.
- The design will probably not change over time.

Because Eiffel fully supports multiple inheritance (as well as Java for interfaces), it does not prevent the class from further inheriting from other classes. In addition, to keep the burden low on developers ESCHER instruments the code automatically.

A second step is to detect existing previous versions of the same class. To address this point we need to consider two different timings: development time and runtime.

At development time, we rely on the notion of release. A system release is a versioned set of classes compiled and thus released together. While each different class has a different class version identifier, different versions of the same class can only be part of different releases. At release time, ESCHER automatically increases the class version numbers of the modified classes.

At runtime, objects of two different versions of the same class will never coexist. Provided the right conversion functions are there though, it will always be possible to retrieve any object of a certain class and version into an object of another version of the same class. This applies to both forwards and backwards updates as customers might be running an old system (specified by an old release number) and need to retrieve objects stored by a newer system release.

4.2 Code generation at development time

While modest, the support for generating code provided by ESCHER relieves developers from writing boiler plate code to transform instances stored in another version to the current one. It lets them focus on the actual specification of the transformer rather than on the details of the code. In particular it provides the following facilities:

- Instrumentation of a persistent class with versioning code: it includes adding an inheritance clause from *VERSIONED_CLASS*, and a query *version* that returns the current version number.

- Creation of a *SCHEMA_EVOLUTION_HANDLER* class associated to the persistent class and containing all the conversion functions from and to any version of the class itself.
- Creation of a *SCHEMA_EVOLUTION_PROJECT_MANAGER* class intended to contain all the associations between handlers and classes for a given project.
- Creation and management of all the separate directories that contain the releases, and inside them, the handlers and the managers.
- Instrumentation of the persistent class with a *FILTER_CLASS* to handle custom serialization.
- Generation of conversion functions, a step detailed below.

The code generation algorithm for the conversion functions works as follows:

1. It compares the two abstract syntax trees of two class versions statically, looking for known refactorings.
2. If a known refactoring is detected, it generates a conversion function and code according to the respective heuristic rule. The conversion function is then placed into the corresponding schema evolution handler class.
3. If more than one refactoring could have taken place, in addition to a skeleton ESCHER generates some comments to make developers aware of the possibility.
4. Comments, hints and console informational messages are always generated to guide developers through the process.

An example of code generation has been already shown in Section 3. An assessment of the code generation process will be presented in Section 5.

4.3 The Runtime Mechanism: a robust algorithm to perform updates

At runtime, ESCHER relies on an ad hoc written library which automatically performs the conversions across two different versions. It is important to notice that the algorithm will raise an exception when one of the following conditions is not met:

1. The specific schema evolution handler exists.
2. The specific conversion function between the two versions exists.
3. Every specific field converter exists.

Figure 8 presents a simplified description of the algorithm expressed in pseudo-code.

It may be interesting to compare the different situations that may arise when dealing with class schema evolution:

```
if (obj_old_version.is_equal (obj_new_version))
 perform standard_retrieval
else if (not class_version_handler_exists)
 raise exception ("Conversion impossible for this class")
else if (not version_handler_exists (v1, v2))
 raise exception ("Incompatible versions")
else if (not attribute.is_convertible)
 raise exception ("A specific attribute cannot be converted")
else perform cross_version_retrieval
```

Figure 8. Algorithm that performs the updates.

1. No schema evolution handling: old objects will typically not be retrievable if the corresponding class has evolved in the meantime.
2. Minimal schema evolution handling: retrieval problems will be detected at runtime, if and when they will happen. If some "transparent schema evolution" is being used, there is a concrete risk of having inconsistent objects being granted access to the system.
3. Schema evolution aware development: developers are aware of schema evolution issues, and take full responsibility for writing transformation functions by themselves. No support is provided by existing tools with this respect.
4. ESCHER invariant-safe schema evolution: developers are guided through the whole process while using the IDE. At runtime there are two checks on retrieved objects: firstly by the algorithm, and secondly by the retrieving class invariant.

Notice that developers have to try hard to bypass the checks mentioned in the last item. They should write wrong conversion functions first, then write wrong class invariants or disable runtime checks for them altogether.

The algorithm is part of a serialization/deserialization library which is decoupled from the IDE, and therefore usable separately. The library, including the serializer, the deserializer and various helper classes, was developed within the scope of the current research. The library can be downloaded separately [12].

4.4 IDE Integration

At the basis of the decision to integrate ESCHER into an IDE lies the assumption that the best time to provide support for schema evolution is when a new version of a class is released.

We have therefore integrated the following additional functionalities into the EiffelStudio IDE:

- Management of system releases.
- Management of class versions.
- Detection, at release time, of already existing previous versions of the same class in the repository.

- Notification, at release time, of already existing previous versions of the same class.
- Correction support, in the form of code templates generation.
- Custom serialization support.

Figure 9 shows a screen capture of the ESCHER tool, please particularly note:

1. The buttons to trigger a new release, create and release a schema evolution handler class and create a filter class to customize serialization.
2. The different release folders and the schema evolution handlers folder on the right.
3. The serializer and serializer code generator libraries on the right.

A system release is seen as a set of classes compiling successfully and constituting a semantically consistent unit meant to be executed at runtime.

Class versions are handled independently from system releases. The same class (having the same version number) can be found in different releases. If in the current release at least one class changes, a new release will be created at the right time. In fact release creation does not happen automatically. The decision on when to release a system is left to developers, who can expressly release a system by pressing the "Release" button in the IDE.

It is at this point that the whole mechanism springs into action. If previous versions of the same class exist in the repository, the tool sends a notification. Developers will then learn, for each class in the current release, exactly which previous versions do exist.

At this point it is again the developer's choice to decide whether it is needed to write a conversion function between two versions. If this is the case, developers will press the "Create Handler" button to trigger a dialog as in Figure 10. The dialog is dynamic with respect to the number of versions present in the repository. Once decided between which versions we need a conversion function, the tool generates and presents the schema evolution handler class. Developers can then check and possibly integrate the generated code.

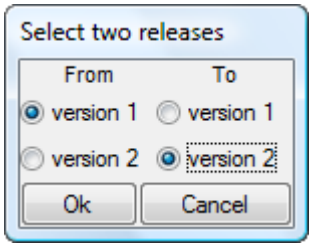

Figure 10. The version selection dialog.

When they think the handler is ready to be released, they trigger the handler release by pressing the "Release Handler" button.

The ESCHER tool provides support for generating a customized serializable form as well. This is triggered by pressing the "Create Filter" button. First developers are prompted to choose a release and a class for which they want to create a custom serialized form (see Figure 11). Developers are then asked to specify which attributes they do not want to store and their default value, necessary for a semantically correct future retrieval (see Figure 12). In fact it may well be the case that the attributes that we are no interested in —with respect to serialization — may have initialization values that differ from the default ones.

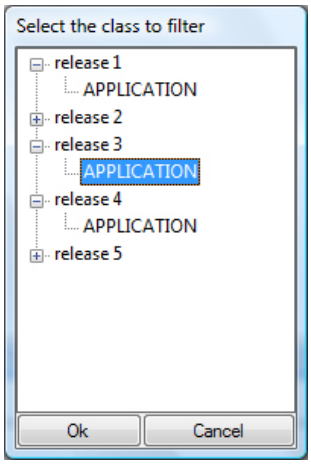

Figure 11. The filter selection dialog.

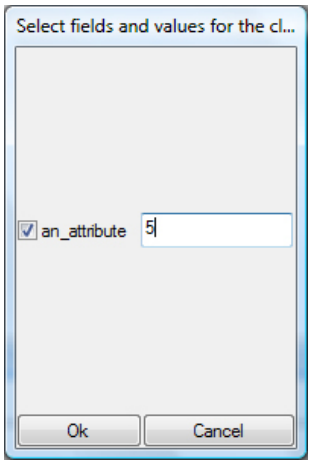

Figure 12. Choice of initialization for non-serilaized attributes.

An introductory tutorial for the tool is available on the ESCHER project pages as well [13].

5. Evaluation

By analyzing some data about widely used object-oriented applications and libraries, we have shown that schema evolution happens also for classes that allow instances to be serialized. A significant part of it involves attribute transfor-

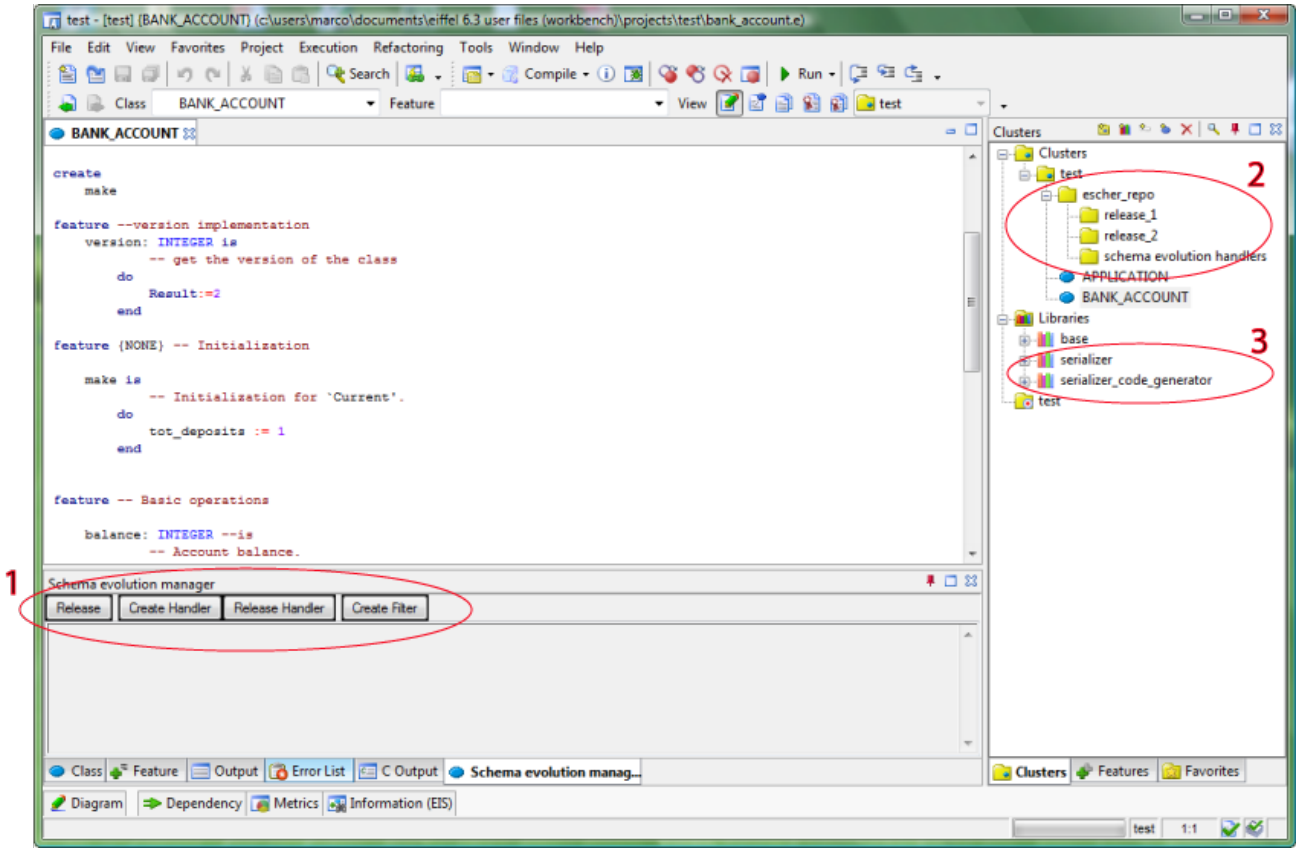

Figure 9. The ESCHER schema evolution tool seamlessly integrates into the EiffelStudio IDE. Note on the following areas: 1. buttons to trigger the release mechanism, 2. release folders and handlers folders, 3. serializer and serializer code generator libraries.

mations. ESCHER is able to discover all the types of transformations that might happen. It also generates object transformers from the transformations it found.

The simple model we devised is enough to model realistically the refactorings. We analyze here informally how each refactoring is recognized:

Attribute not changed: the code generator is active in assigning a release number to every class and creating a system release. Nothing else needs to be done.

Attribute added: the code generator always detects this refactoring, but the quality of its suggestion is low. In fact by only comparing the AST's the only reasonable suggestion that can be offered is to initialize the attribute with the default value. Fortunately more can be done, thanks to the class invariants embedded in the Eiffel language. If the class invariants are written correctly, they will be checked at runtime and enable a truly invariant-safe schema evolution.

Attribute renamed: the code generator cannot succeed in always detecting this refactoring. The best it can do is to generate a conversion function template and warn developers of the possibility of a rename. Figures 13 and 14 illustrate the possible dilemma that may arise: was *att_2* removed and *att_2* added or *att_2* was simply renamed? A better way of analyzing the code to find out about that would be to check whether the variables are used by the same clients in the same context. As it requires a global analysis of the code we considered it was too time consuming and could actually be captured in a better way by tracking user direct use of a renaming function.

Attribute type changed: here the code generator provides the best results, being always able to both detect the refactoring and provide a complete code generation, as shown in the example in Section 3.

Attribute removed: the code generator always detects this refactoring, and issues a warning when in the new version there is a new attribute having the same type as the removed attribute. This would be a possible case of renaming.

In short, even if we have acceptable results in the majority of the cases, more can be done in the case of attribute renaming (less than 5% of persistence-related changes). These limitations are here because of the way we compare class versions, directly both classes' AST.

```
class
  SAMPLE_CLASS

feature -- Status report

  att_1 : INTEGER

  att_2 : STRING

feature -- Other features omitted

end
```

Figure 13. *SAMPLE_CLASS*, version 1.

```
class
  SAMPLE_CLASS

feature -- Status report

  att_1 : INTEGER

  att_3 : STRING

feature -- Other features omitted

end
```

Figure 14. *SAMPLE_CLASS*, version 2.

This work suggests a shift of attitude on how developers presently cope with class schema evolution. More precisely, by integrating support into an IDE, the approach elevates class schema evolution to the status of first-class citizen of the software development process rather than undesirable side effect of the software production activities. It also proposes a significant time shift for any schema evolution effort, moving it from runtime to release time. A system release becomes then an event triggering a whole set of tool activities intended to help developers focus on solving possible issues that may arise from previous versions of the newly released code.

A question often asked is how programmers can be sure that an object of a previous class version and violating the new class invariant will not be accepted into the new system. To address this point, we have devised a robust, version-aware algorithm for object retrieval, which guarantees that only objects for which conversion functions have been previously written, and for which class invariants hold, will be accepted into the new system.

5.1 ESCHER in practice

As we have seen, analyzing the history of classes along different versions can provide useful insights over the schema evolution process. To provide a better evaluation it may also be interesting to analyze what the effect of using the tool would have been in the case of a widely used library class. We now choose a well known Eiffel library class, *HASH_TABLE*, because as we have already seen for its Java counterpart, its objects are typically serialized a lot. The class is analyzed along a period going from 1999 to 2008. Three major revisions are considered. The results are the following:

- From revision 13752 to revision 47039: three attributes changed type.
- From revision 47039 to revision 62327: one attribute added, two constant attributes changed type, one constant attribute removed.
- From revision 62327 to revision 76420: two constant attributes removed.

The first thing to note is that there are a significant number of refactorings affecting constant attributes, which typically are not serialized. We will therefore focus on the other refactorings: three attributes that changed type and then one attribute added. By reading the comments in the code an interesting story comes out: the three attributes mentioned in the first time interval changed type because of efficiency reasons. Basically the implementation switched from an array to a more efficient container. Then in a later version a transformation function was provided, to help converting old versions objects into the new ones. This function used an ad-hoc created attribute, *hash_table_version_57*, specifically created to help importing the old objects into the new schema. This is the added attribute mentioned earlier.

What would have happened if the class developers would have had the possibility to use the ESCHER tool? The process would have certainly been smoother, because the transformation code would have been shipped together with the library. In addition, the conversion code would not have cluttered the class itself. For what concerns the possibility to accept corrupt old objects into the system, this would have not happened anyway because the Eiffel serialization library was already robust against this possibility at that time thanks to the class invariants written for it.

All in all, we think that the proposed approach has some inherent limitations, due to the fact that human intervention cannot be completely ruled out, and some specific limitations, object of future work. However, thanks to the use of class invariants and seamless integration into the development life cycle, also takes a step forward towards a more aware object-oriented schema evolution handling, and is a valuable addition to the current scenario of software development.

5.2 Threats to validity

By analyzing data from persistent classes in `java.util` it is clear that serializable classes evolve over time. The connection with persistent classes is however not that obvious as one could argue that serialization could also be used simply to send data over the network or for storing temporarily instances. Because the serialization facility can be used to

make objects persistent, it deserves a mechanism to handle schema evolution. In the case of Java, the standard mechanism is to reference the version of the class using a simple identification number and rejecting data from older versions of the code. It is however a threat to the validity of our data analysis.

While we think that ESCHER addresses most practical challenges, it still needs a broader validation than the limited testing that we were able to perform. In particular the ease of use of the interface and the actual refactorings detection and code generation should be further investigated through extensive acceptance testing and the actual development and maintenance of applications that use persistence extensively. The ideal case would be that independent developers who use persistence in Eiffel would pick ESCHER to handle it. Not having such a study or such acceptance threatens the validity of our approach.

6. Previous Approaches

While recognizing the importance of the schema evolution issues in relational databases [17], we focus here on object-oriented challenges. The issues arising from object-oriented schema evolution are widely acknowledged. They affect both the object-oriented databases and all the object oriented programming languages providing a serialization facility. We detail mainly two types of approaches: the *class descriptor* approach and the *versioning* approach.

6.1 The *class descriptor* approach

The most widespread approach to object-oriented schema evolution handling relies on class modification to accommodate the changes in the class schema. The idea is to devise a *class descriptor* representing the information to serialize. The class descriptor can therefore encode either a physical or a logical representation of the class itself. To convert objects from an older version into the current ones, developers typically implement conversion functions. As a conversion function only converts a specific stored version into the current one, it is difficult to handle all the different versions that may exist.

The Java language, for example, offers an object serialization API that uses a binary format. A class can enable future serialization of its instances by implementing the `Serializable` interface [18]. Bloch [19] notices that this simple addition brings important constraints. In fact, by using the automatically generated serial version unique identifier, the default serialized form of an object becomes 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 only 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. While using the default serialized form can sometimes be appropriate, a more flexible serialized form of an object should be independent of the physical representation. This can be achieved by explicitly setting the serial version unique identifier.

For a panorama of other approaches to persistence in Java see also [28, 29]. 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 [30]. In spite of the name, VTS does not implement true version support. 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. By using attributes instead of interfaces the mechanism is decoupled from the class hierarchy.

It seems that the serialization mechanisms of both Java and the .NET languages are slowly converging towards a full-fledged solution such as an object oriented database management system (OODBMS). To assess the benefits of such a solution we have found useful to examine the db4o object oriented database [31, 32]. One advantage of using it is that objects can be serialized as they are, without having their code polluted with persistence code.

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 developers need a custom behavior to reestablish the invariant with respect to an older stored version of the object, there are two possibilities. They can either choose to use reflectively invoked methods in the object class or can register listeners to specific `ObjectContainer` events outside the object class. An interesting scenario occurs when developers do not foresee the possible issues and "forget" to code appropriate methods to handle the conversion. Unfortunately, in this case the newly added attributes are automatically initialized to their default values. As already seen earlier, this appears to be an excessively optimistic level of transparency, because it allows into the system objects whose class invariants may not be valid anymore.

Eiffel's current serialization mechanism presents a solution in which all conflicts are resolved in one class. Custom serialization behavior can therefore be provided by inheriting from a `MISMATCH_CORRECTOR` class and redefining a callback feature *correct_mismatch* which at runtime will correct the mismatch to re-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. In fact, Eiffel programmers have an explicit way to declare the class invariant itself in the class text via the `invariant` clause. Like with Java and .NET, the Eiffel serialization mechanism is completely unaware of all the different object versions that may have been stored in the past. On the positive side, the retrieval algorithm will not accept objects for which a mismatch occurs and no conversion function has been previously programmed. This implementation choice makes therefore impossible for an object that does not satisfy the class invariant to be accepted in the system after deserialization because a developer happened to forget to explicitly take care of writing the conversion function.

With respect to schema evolution, the main differences between the implementations of the class descriptor approach just described and the one proposed by the authors are the following:

- We use versioning instead of a class descriptor, keeping the possibility to track back the history of changes occurred and being able to convert an object of any version in an object of any other version of the same class.
- The version number is not bound to the class structure, and versions are handled automatically.
- We use a tool seamlessly integrated into the IDE, instead of a language API.
- We use a robust retrieval algorithm that disallows object not satisfying the class invariant to access the retrieving system.

6.2 The *versioning* approach

This versioning approach solves some of the issues seen previously for the class descriptor, because it keeps the information about all the versions of each class. While providing a consistent view of the logical structure of the repository, it enables reliable handling of backward and forward evolution of class schemata. Also robustness is increased. A typical scenario this approach is able to handle is the one in which there is a need of undoing a certain conversion that raised an exception, for example because of a class invariant failure.

An example of a an object oriented database implementing the versioning approach was formerly known as Poet, now Versant Fastobjects [33]. Simplifying the architecture for our purpose, a Fastobjects database consists of two main storage areas: a dictionary, which carries the class definitions for objects in the database, and the actual database, containing the serialized objects forms. The dictionary is necessary to understand the structure of the objects stored in the database. The delicate issue here is modifying the dictionary after having stored objects. This is where versions come into play: in fact the dictionary keeps versions for every class. When a class schema modification is detected, objects stored according to a certain schema are "transparently" and lazily converted to another schema. As in the case of db4o, the term "transparent schema evolution" is a bit misleading here, because in general there are no guarantees of consistency for the transformation of the old schema into the new one.

CLOSQL constitutes another example of a system using the versioning approach. It uses LISP as implementation language. To convert a stored object of a certain version into an object of the current version, update or backdate routines are executed. A requirement is that a database administrator is needed every time a class is created, to specify which update or backdate routines have to be executed. As in our approach, the update and backdate routines for a certain version are kept all together, in this case in an "update method". As a limitation, only linear versioning is supported: a new version can only be generated from the latest version. Monk and Sommerville describe the issues and challenges that have been faced [34][35].

In contrast to CLOSQL, we provide object transformers from any version to any other.

6.3 Other approaches

Orthogonal Persistence Java (OPJ) is an extension of the Java language Specification (JLS) providing "orthogonal persistence" capabilities to the Java platform [21, 23]. The three principles defining orthogonal persistence are:

(1) Type Orthogonality: persistence is available for all data, irrespectively of type.

(2) Persistence by Reachability: the lifetime of all objects is determined by reachability from a designated set of root objects, the so-called persistent roots.

(3) Persistence Independence: it is indistinguishable whether code is operating on short-lived or long-lived data.

The well known prototype implementation for OPJ, PJama [25, 26], is an extension of the Java Virtual machine (JVM) together with a persistent store, in which the state of an executing application is kept. The system state is checkpointed atomically and periodically to be able to recover from exceptions and crashes. PJama provides an approach to schema evolution that is quite similar to the class descriptor approach seen in the previous section. The main difference is that it involves persisting both objects and classes. Following Dmitriev [27], this is the best way to preserve both structural consistency and behavioral consistency of the data. Structural consistency is defined as the correspondence between class definitions and real structure of the corresponding objects. Behavioral consistency is defined as formal correctness of the program: every method which is called is defined and every field which is read/written is defined. To effectively evolve the system, PJama provides a small API and a standalone, command-line utility used by developers to perform conversions between classes. In case

the changes are validated, the objects are typically converted eagerly.

When a certain class evolves over time, an interesting dilemma arises: should we be considering it a different type, and name it differently, or should we consider it the same type, leaving the same name but providing some other means of taking into account the different inner structure and semantics? Both options are possible and while the second is mainstream, the first is previous work by the authors [4]. It relies on the assumption that the evolved class might be considered a different class with respect to the un-evolved one, because its schema has changed. Therefore it makes sense to provide it with a different name, possibly including version information. The Eiffel programming language provides an embedded mechanism to cope with this kind of situation, namely the "converter semantics" [5]. The main idea is that two types can either conform, via inheritance, or convert to each other, but not conform and convert at the same time. This provides an answer to issues like the conversion of different string implementations across different systems, or conversions across different numeric types like integers and reals, which don't conform to each other. The mechanism takes care of automatically invoking appropriate conversion functions placed into the class itself. This will typically happen when operations like assignments or argument passing are performed. For the converter semantics to work, it is essential for two different versions of the same class to have different names. As the converter mechanism is integrated into the Eiffel compiler, versions can therefore be considered full citizens of the type system.

A downside of this approach is the lack of scalability. This can be observed when applied to many different class versions. Every type needing to handle versions should provide converters for each previous version, clogging the class code with a significant number of conversion functions.

The work probably most similar to the present one in terms of automatic generation of converters is the type transformers generation described by Neamtiu *et al.* [37]. It focuses on updating structs in C programs whose layout might evolve. While it is not per se linked to object-orientation, it presents a way of generating type transformers. However, it does not benefit from having a model, an empirical evaluation, or an IDE.

7. Conclusions and Future Work

Contrary to what one might expect, classes which produce persistent instances evolve over time. In our study more than one change out of six impacted the compatibility of persistent instances with the new version of serializable class. This high ratio implies that the basic mechanisms that support persistence — like serialization— need a better way to handle the schema evolution of their classes.

This paper presents a solution to this issue: the ESCHER platform. This solution mainly relies on an IDE that identifies refactorings (modifications in the code and data structures), potentially using developers' input, and generated the migration code to handle deserializing instances of an older version of the code. The handlers rely on a persistence library that supports the runtime aspects of the infrastructure. In all practical examples it worked in a satisfactory manner.

A future improvement of our work will provide better support for detecting refactorings by monitoring developers while they are using the IDE (what they add/remove/modify...). This would possibly lead to an even more accurate identification of the refactorings. We also plan to support a wider set of languages and study whether there are significant differences between them. Finally it might be interesting to study the migration of data from one language to another and see whether our infrastructure could act as a migration tool by analyzing differences across language boundaries.

References

[1] Parnas, D.L.: Software Aging, in Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, pp. 279-287 IEEE press (1994).

[2] Boyapaty C., Liskov B., Shrira L., Moh C.H., Richman S., Lazy Modular Upgrades in Persistent Object Stores, in Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) 2003.

[3] Franconi E., Grandi F., Mandreoli F.: Schema Evolution and Versioning: a Logical and Computational Characterization, in FMLDO, Springer LNCS vol. 2065, 2001, pp. 85-99.

[4] Piccioni M., M. Oriol M., Meyer B.: IDE-integrated Support for Schema Evolution in Object-Oriented Applications. In: Proceedings of 4th ECOOP' 2007 Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'07), Berlin 30 July - 3 August 2007, pp. 27-36. Ed. W Cazzola, S Chiba, Y Coady, S. Ducasse, G. Kniesel, M. Oriol, G. Saake.

[5] 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.

[6] Meyer B.: Eiffel: The Language. Prentice Hall (1992).

[7] ECMA committee TC39-TG4, ECMA International standard 367. Eiffel Analysis, Design and Programming Language (2005).

[8] Meyer B.: Object Oriented Software Construction. 2nd ed. Prentice Hall PTR (1997).

[9] Ribet P., Adrian C., Zendra O., Colnet D.: Conformance of agents in the Eiffel language, in JOT (Journal of Object Technology), vol. 3, no. 4, April 2004, Special issue: TOOLS USA 2003, pp. 125-143.

[10] EiffelStudio: http://www.eiffelsoftware.com/products/studio/index.html

[11] ESCHER project: http://escher.origo.ethz.ch/wiki/escher

[12] ESCHER serialization library: https://svn.origo.ethz.ch/escher/schema evolution tool/serial-

izer_lib

[13] ESCHER tutorial:
http://escher.origo.ethz.ch/wiki/escher_eiffelstudio_tool_tutorial

[14] Advani D., Hassoun Y., Counsell S.: Refactoring trends across N versions of N Java open source systems: an empirical study. Technical Report BBKCS-05-02, Birkbeck College, School of Computer Science and Information Systems, 2005.

[15] Advani D., Hassoun Y., Counsell S.: Extracting refactoring trends from open-source software and a possible solution to the 'related refactoring' conundrum. Proceedings of the 2006 ACM Symposium on Applied Computing (SAC).

[16] Schema evolution experimental data:
http://se.inf.ethz.ch/people/piccioni/papers/data/SchEvJavaData.xls

[17] Date C.: Introduction to Database Systems 8th ed. Addison Wesley (2003).

[18] Gosling J., Joy B., Steel G. and Bracha G.: The Java Language Specification. 3rd ed. Addison Wesley (2005).

[19] Bloch, J.: Effective Java. Prentice Hall PTR. (2001).

[20] Milne P.: Using XMLEncoder
http://java.sun.com/products/jfc/tsc/articles/persistence4/

[21] Atkinson M.P.: Programming languages and Databases. VLDB Journal, pp. 408-419 (1978).

[22] Atkinson, M.P., Bailey, P.J., Chisholm, K., Cockshott, W.P., Morrison, R.: An approach to persistent programming. Comput. J. 26 (1983) 360365

[23] Atkinson M.P., Morrison R.: Orthogonally Persistent Object Systems. VLDB Journal, 4(3), pp.319-401, 1995.

[24] Atkinson, M.P., Daynes, L., Jordan, M.J., Printezis, T., Spence, S.: An orthogonally persistent Java. SIGMOD Record 25 (1996) 6875

[25] Atkinson, M.P., Jordan M.: A Review of the Rationale and Architectures of PJama: a Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform. Sun Microsystems Laboratories Technical Report, SMLI TR-2000-90, (2000).

[26] Lewis B., Mathiske B. and Gafter N.: Architecture of the PEVM: A High-Performance Orthogonally Persistent Java Virtual machine. Sun Microsystems Laboratories Technical Report, SMLI TR-2000-93, (2000).

[27] Safe Class and data Evolution in Long-Lived Java Applications. Sun Microsystems Laboratories Technical Report, SMLI TR-2001-98, (2001).

[28] Atkinson M.P.: Persistence and Java - A Balancing Act, in Object and Databases. International Symposium, Sophia Antipolis, France. Revised Papers, LNCS vol. 1944, Springer Verlag (2000).

[29] Jordan M., A Comparative Study of Persistence Mechanisms for the Java Platform,
http://research.sun.com/techrep/2004 (2004).

[30] Version Tolerant Serialization:
http://msdn2.microsoft.com/en-us/library/ms229752.aspx

[31] Paterson J., Edlich S., Hörning H. and Hörning R.: The Definitive Guide to db4o. Apress (2006).

[32] db4o API documentation:
http://developer.db4o.com/resources/api/db4o-java

[33] Fastobjects object oriented database: www.versant.com

[34] Monk S., Sommerville I.: Schema Evolution in OODBs Using Class Versioning. In: ACM SIGMOD Record archive, Volume 22, Issue 3 (September 1993), pp. 16-22.

[35] Monk S., Sommerville I.: A Model for Versioning of Classes in Object-Oriented Databases. In: Proceedings of British National Conference on Databases (BNCOD), July 1992.

[36] Fowler M., Refactoring (Improving the Design of Existing Code). Addison Wesley, 1999.

[37] Neamtiu I., Hicks M., Stoyle G. and Oriol M.: Practical Dynamic Software Updating for C. In: ACM Conference on Programming Language Design and Implementation (PLDI), pp. 72-83. ACM Press, Ottawa, Canada (2006).