An IDE-based, integrated solution to Schema Evolution of Object-Oriented Software

Marco Piccioni*, Manuel Oriol†, Bertrand Meyer* and Teseo Schneider*
*Chair of Software Engineering, ETH Zurich, Switzerland
Email: {marco.piccioni,bertrand.meyer}@inf.ethz.ch, teseos@student.ethz.ch
†University of York, United Kingdom
Email: manuel@cs.york.ac.uk

***Abstract*—With the wide support for serialization in object-oriented programming languages, persistent objects have become common place. Retrieving previously "persisted" objects from classes whose schema changed is however difficult, and may lead to invalidating the consistency of the application.**

The ESCHER framework addresses this issues through an IDE-based approach that handles schema evolution by managing versions of the code and generating transformation functions automatically. The infrastructure also enforces class invariants to prevent the introduction of any corrupt objects. This article describes the principles behind invariant-safe schema evolution, and the design and implementation of the ESCHER system.

***Index Terms*—versioning; refactoring; persistence; serialization; object-oriented schema evolution; IDE integration**

I. Introduction

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. Many currently available persistence solutions are coupled with tolerant retrieval algorithms, automatically making questionable decisions about the initialization of the retrieved objects. This enforces silent acceptance of possibly inconsistent objects into the system. This is a recipe for disaster, as class invariants may be silently violated.

To tackle these issues, we first study how serializable classes from the `java.util` package evolved from Java 1.2.2 to Java 6.0. It appears that 17.5% of the changes have impacted the persistence of the serializable classes of the package. This allows the identification of the most frequently occurring refactorings. We secondly propose a formal model that represents the refactorings as well as the corresponding transformation functions.

We then suggest 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, including version handling and code template generating features for the transformation functions. In addition we provide a robust retrieval algorithm to prevent the acceptance of inconsistent objects into the system.

Section II presents the analysis of some newly collected data about the refactorings of `java.util`. Section III describes the model for software updates. Section IV details the implementation of both the IDE integration and the retrieval algorithm. Section V analyzes contributions and limitations of the current approach. Section VI summarizes the previous approaches, from both the authors and others. Finally, section VII describes our conclusions and future work.

II. Schema Evolution in Practice

Advani et al. [1] already evaluated refactorings[1] in fifteen open source Java systems and showed that refactorings like "rename field", "move field", "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. 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.

Because instances of `Serializable` classes might persist, we study the serializable classes from the Java package `java.util` to check if persistent classes evolve in a similar manner. The package `java.util` itself is particularly interesting because it contains classes whose instances are likely to be serialized either directly or by transitive closure while serializing instances of client classes. The package contains classes that model collections, dates, currencies, and locales. We considered the 22 classes in the package directly implementing the *Serializable* interface and analyzed them manually across five major versions of the language: 1.2.2, 1.3.1, 1.4.2, 5.0, 6.0. We also took into consideration 22 refactoring types, ten of which are directly relevant to the serialization process. These 22 refactorings are shown in Table I and were created by using a systematic approach that considered all possible changes (addition, removal, modification) over all possible targets (attribute, visibility, methods etc). Note that the complete

[1]In this article, consistently to the work from Advani et al. [1] we use the term *refactoring* for any minor modifications of the code rather than semantics preserving modification-only.

raw data are available for download [2].

In line with what was discovered previously, Table I shows that the persistence-related refactorings consist of 17.5% of the total number of refactorings.

Class	Refact.	Persistence-related	%
ArrayList	18	2	11.1
BitSet	42	6	14.3
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 II
REFACTORINGS FOUND ACROSS 5 VERSIONS OF THE JDK, BY CLASS AND KIND.

Table II shows the distribution of refactorings across different classes and across all five versions of the Java Development Kit (JDK). The data suggests that persistence-related refactorings are sufficiently widespread among classes, and confirm that persisted data might actually change significantly over time.

Table III 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.

Refactoring type	Refact.	% persist.-related
Attribute added	57	50.0%
Attribute removed	28	24.8%
Attribute renamed	3	2.7%
Attribute type changed	15	13.3%
Attribute value changed	3	2.7%
Attribute to constant	6	5.4%
Constant to attribute	1	0.9%
Total	113	100.0

TABLE III
PERSISTENCE-RELATED REFACTORINGS, ACROSS ALL VERSIONS, BY REFACTORING KIND.

What this short study shows is that classes whom instances might be serialized change over time. Moreover 17.5% of these changes directly impact the capability of classes to deserialize instances of their previous versions. 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.

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

III. MODEL

This part presents a model of updates that particularly emphasizes the generation of conversion functions — contrary to other previous models of refactoring [2], [3]. For the sake of brevity, we are only describing the main ideas. All the details about our model can be downloaded separately [4].

We present a simplified syntax of class definitions in Eiffel programs. We omit the declaration both of routines and constraints on generic parameters as they are not included in the serialized form. We also do not explicitly consider inheritance 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.

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

$$R : class \mapsto class$$

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. The three missing refactorings are treated as follows: “Attribute value changed” is taken into account when evaluating the new class invariant clause, so no special action is needed. “Attribute to constant” is considered as “Attribute removed”, as constants are not serialized. “Constant to attribute” is 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 — for example by deleting all attributes whose names are not present in the class anymore and adding the new ones. Thus, class transformations are complete with respect to attribute modifications.

While there is always a decomposition, using a straightforward algorithm might not produce the best results and devising

[2]http://se.inf.ethz.ch/people/piccioni/ASE2009/SchemaEvolutionData.xls (last visited: 31/8/09)

	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	2	1	0	2	0	1	81	27	13	3	11	5	11	0	0	0	0	1	1	0
All	57	28	3	15	3	6	1	8	182	40	166	6	24	9	39	6	0	8	9	1	3	33
Category	113 (17.5%)							534 (82.5%)														
Total	647																					

TABLE I
REFACTORINGS FOUND ACROSS 5 VERSIONS OF JAVA.UTIL PACKAGE.

heuristics is essential to use this model as the basis of an implementation.

The current solution relies on statically comparing the abstract syntax trees (AST) of both classes to detect refactorings. The extraction relies on a set of heuristics successively applied:

- An attribute that does not change name and declared type generates an “Attribute not changed” refactoring. The code generator assigns a release number to every class and creates a system release. No other action is necessary.

- An attribute that does not change name but changes type between two versions generates an “Attribute type changed” refactoring. Here the code generator provides the best results, being always able to both detect the refactoring and provide a complete code generation.

- 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. By only comparing the two classes’ ASTs it is not possible to determine what happened in this specific case. The code generator, after detecting this refactoring, will generate a conversion function template warning about the possibility of a rename. A better way of analyzing the code to find out about that point 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.

- 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. The framework always detects this refactoring, but the only reasonable suggestion that can be offered is to initialize the attribute with the default value and to give the hint to check the new class invariant with respect to the old one. Here we do not see space for a real improvement, because the initial value of a newly added attribute is largely determined by a semantics known to developers only.

- An attribute of the old version that cannot find a counterpart in the new version generates an “Attribute removed” refactoring. 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. In fact this could be a possible case of renaming.

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

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

IV. IMPLEMENTATION

Though our approach can be applied to any object-oriented programming language providing support for storing and retrieving objects, we use Eiffel (and its IDE EiffelStudio) 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 [5].

The versioning mechanism relies on the notion of *release*. A 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, 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 in need of retrieving objects stored by a newer system release.

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. At runtime, ESCHER relies on an ad hoc library that automatically performs the conversions across two different versions. 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, or (3) every specific field converter exists.

The algorithm is part of a serialization/deserialization library which is decoupled from the IDE, and therefore separately usable. The detailed documentation about the mechanism, a step-by-step tutorial, the source code and the executables for the tool are available for download on the ESCHER project page.[3]

V. Evaluation

This work proposes 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 more on possible issues that may arise from previous versions of the newly released code.

By analyzing some data about a widely used Java library, we have shown that schema evolution happens also for persistent classes.

To evaluate ESCHER we check how ESCHER scores with respect to the 113 persistence-related refactorings previously detected. This can be done because the considered refactorings are clearly language-independent and they can be easily recreated using ESCHER. The result obtained is that ESCHER is able to exactly identify the correct refactorings in **all cases** (100%) from the changes in `java.util`. Even if this is unlikely in the general case, especially for less stable code, this result suggest that the simple model we devised is enough to model a realistic set of refactorings. In Section III we have analyzed how each refactoring is recognized, and what can be done to further improve the framework support for code generation.

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

- No schema evolution handling: old objects will typically not be retrievable if the corresponding class has evolved in the meantime.
- 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.
- Schema evolution-aware development: developers are aware of schema evolution issues, and take full responsibility for writing transformation functions by themselves. No integrated support is provided by existing tools with this respect.
- 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.

Note that developers have to be particularly absent-minded to bypass all 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.

Unfortunately class invariants are not widely accepted and coded among developers not using the Eiffel language. An obvious question is then the following: what can a Java developer do if he wishes to emulate the ESCHER mechanism with respect to class invariants? While analyzing the Java libraries, we discovered something that may help. The class *BitSet*, during the transition from version 5.0 to 6.0, shows that the importance of class invariants is clearly recognized and can be put into effect. In fact a new method *checkInvariants()* is introduced, with the idea of enforcing some class-wide properties. This method is then invoked from every public method in the class, emulating the Eiffel invariant checking mechanism.

Threats to validity. 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.

VI. Previous Approaches

The most widespread approach 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. To convert objects from an older version into the current ones, developers typically implement conversion functions. The Java language and the .NET framework with Version Tolerant Serialization (VTS) [4] both follow this approach [6]. The db4o OODB [7] advertises a "transparency" which

[3]http://escher.origo.ethz.ch/wiki/escher

[4]http://msdn2.microsoft.com/en-us/library /ms229752.aspx (last visited: 31/8/09)

seems overly optimistic, considering that retrieved objects with default values for attributes can be automatically accepted into the system.

The versioning approach, for example implemented in the Versant Fastobjects OODB,[5] keeps the information about all the versions of each class. While providing a consistent view of the logical structure of the repository, it enables handling of backward and forward evolution of class schemata. Again, issues arise when dealing with a semantically consistent objects retrieval. To convert a stored object of a certain version into an object of the current version, CLOSQL [8] uses update or backdate routines. An unfortunate 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.

PJama [9], [10] is an extension of the Java Virtual machine 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 involves persisting both objects and classes, but again does not solve the fundamental consistency problem cited before. When a certain class evolves over time, it may be considered a different type, and named differently, or it may be considered the same type, leaving the same name but providing some other means of taking into account the different inner structure and semantics. While the second is mainstream, the first approach has been already explored [11] by the authors.

Another example of making different, parallel versions explicit can be found in UpgradeJ [12], though not focused neither on object persistence nor on enforcing semantical consistency. The automated detection of refactorings has been also extensively explored in literature, both with respect to existing libraries and software configuration systems, for example by Dig *et al.* [13]. A similar idea of using a transformational approach and a classification of modifications, but applied to assist metamodel evolution by stepwise adaptation instead, is presented in [14][15][16][17]. Analyzing the AST in metamodels and applying similarity metrics to detect changes has been done in [18]. All the cited works about metamodels are of interest because class schemas are a particular kind of metamodels. The automatic generation of converters through type transformers generation has been described by Neamtiu *et al.* [19]. It focuses on updating structs in C programs whose layout might evolve. However, it is not per se linked to object-orientation and it does not benefit from having a model or an integration into an IDE.

VII. Conclusions and Future Work

Classes which produce persistent instances evolve over time. In our study more than one change out of six impacted the compatibility of persistent instances. This high ratio implies that the basic mechanisms that support persistence — like serialization— need a way to handle the schema evolution of their classes.

This article presents the ESCHER platform to cope with this issue. It relies on an IDE which identifies refactorings and generates semi-automatically migration code.

Thanks to the use of class invariants and seamless integration into the development life cycle, ESCHER takes a step forward towards a more reliable schema evolution handling, and seems a valuable addition to the current scenario of software development.

In the near future we plan to improve the quality of the code generation and to run usability studies with practitioners from industry and academia to further validate our approach.

[5] www.versant.com (last visited: 31/8/09)

References

[1] D. Advani, Y. Hassoun, and S. Counsell, "Extracting refactoring trends from open-source software and a possible solution to the 'related refactoring' conundrum," in *SAC*, 2006, pp. 1713–1720.

[2] C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman, "Lazy modular upgrades in persistent object stores," in *OOPSLA*, 2003, pp. 403–417.

[3] E. Franconi, F. Grandi, and F. Mandreoli, "Schema evolution and versioning: A logical and computational characterisation," in *FMLDO*, 2000, pp. 85–99.

[4] M. Piccioni, M. Oriol, B. Meyer, and T. Schneider, "An ide-based, integrated solution to schema evolution of object-oriented software," ETH Technical Report, Tech. Rep. 638, 2009. [Online]. Available: ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/6xx/638.pdf

[5] *ECMA International standard: Eiffel Analysis, Design and Programming Language*, ECMA committee TC39-TG4 Std. 367, 2005.

[6] M. Jordan, "A comparative study of persistence mechanisms for the java platform," Sun Microsystems Laboratories Technical Report, Tech. Rep. TR-2004-136, 2004.

[7] J. Paterson, S. Edlich, H. Hörning, and R. Hörning, *The Definitive Guide to db4o*. Apress, 2006.

[8] S. R. Monk and I. Sommerville, "Schema evolution in oodbs using class versioning," *SIGMOD Record*, vol. 22, no. 3, pp. 16–22, 1993.

[9] M. P. Atkinson and M. Jordan, "A review of the rationale and architectures of pjama: a durable, flexible, evolvable and scalable orthogonally persistent programming platform," Sun Microsystems Laboratories Technical Report, Tech. Rep. SMLI TR-2000-90, 2000.

[10] M. Dmitriev, "Safe class and data evolution in long-lived java applications," Sun Microsystems Laboratories Technical Report, Tech. Rep. SMLI TR-2001-98, 2001.

[11] M. Piccioni, M. Oriol, and B. Meyer, "Ide-integrated support for schema evolution in object-oriented applications," in *RAM-SE*, 2007, pp. 27–36.

[12] G. M. Bierman, M. J. Parkinson, and J. Noble, "Upgradej: Incremental typechecking for class upgrades," in *ECOOP*, 2008, pp. 235–259.

[13] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen, "Refactoring-aware configuration management for object-oriented programs," in *ICSE*, 2007, pp. 427–436.

[14] G. Wachsmuth, "Metamodel adaptation and model co-adaptation," in *ECOOP*, 2007, pp. 600–624.

[15] B. Gruschko, D. S. Kolovos, and R. F. Paige, "Towards synchronizing models with evolving metamodels," in *MODSE*, 2007.

[16] M. Herrmannsdoerfer, S. Benz, and E. Jürgens, "Automatability of coupled evolution of metamodels and models in practice," in *MoDELS*, 2008, pp. 645–659.

[17] A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio, "Automating co-evolution in model-driven engineering," in *EDOC*, 2008, pp. 222–231.

[18] J.-R. Falleri, M. Huchard, M. Lafourcade, and C. Nebut, "Metamodel matching for automatic model transformation generation," in *MoDELS*, 2008, pp. 326–340.

[19] I. Neamtiu, M. W. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for c," in *PLDI*, 2006, pp. 72–83.