# Revisiting Schema Evolution in Object Databases in Support of Agile Development

Tilmann Zäschke and Moira C. Norrie

Institute for Information Systems, ETH Zurich
CH-8092 Zurich, Switzerland
{zaeschke,norrie}@inf.ethz.ch

**Abstract.** Based on a real-world case study in agile development, we examine issues of schema evolution in state-of-the-art object databases. In particular, we show how traditional problems and solutions discussed in the research literature do not match the requirements of modern agile development practices. To highlight these discrepancies, we present the approach to agile schema evolution taken in the case study and then focus on the aspects of backward/forward compatibility and object structures. In each case, we discuss the impact on managing software evolution and present approaches to dealing with these in practice.

## 1 Introduction

The introduction of agile software development methods impacts on schema evolution in at least two aspects. First, the agile preference for continuous evolution and refactoring of the data model results in more frequent schema evolution. Second, the shortened release cycles mean that not only the in-house data model has to be updated more often, but also the customers have to be more frequently provided with means to make their existing data usable with new software releases. Accordingly, and depending on the complexity of the data model and the environment, managing and implementing schema evolution can require significant effort in a software project.

The integration of traditional relational databases into object-oriented systems can restrict agility because of the need to maintain the object to relational mapping. The case has therefore been made that object databases are better suited to agile development methods [1]. Yet the support for schema evolution offered by the various object database products tends to be limited. As a result, application developers often have to produce significant custom code to manage evolution. At the same time, most of the solutions proposed in the research community precede modern software development practices and are based on invalid assumptions or address challenges that no longer represent the key issues.

To analyse the requirements and solutions for managing schema evolution, we examined the real-world case study of a software project that adopted agile development practices and uses an object database to store persistent data. The system was developed by the European Space Agency (ESA) to manage

A. Dearle and R.V. Zicari (Eds.): ICOODB 2010, LNCS 6348, pp. 10–24, 2010.

the scientific operation of their Herschel Space Observatory. It was found that little of the available research was applicable to the project, in part due to preconditions assumed (if not explicitly stated) by many researchers. While these preconditions are certainly valid for some projects, we believe that the features and requirements of the Herschel system are characteristic of many modern applications. Further, for some of the challenges faced by the developers of the Herschel system, we were unable to find any scientific references at all. The main contribution of this paper is to point out and give possible explanations for the differences between research and practice, thereby showing the demand and opportunities for further research.

We note that while our discussion is set in the Java/Versant world due to the case study considered, we believe that most of the lessons learned can be easily applied to other languages and object databases. Any parts of the discussion that are specific to the Java/Versant setting will be clearly indicated.

We begin in Sect. 2 with a discussion of the state of the art in terms of research and also support for schema evolution in object database products. An overview of the Herschel project is presented in Sect. 3 followed by a discussion of agile practices and their impact on requirements for managing schema evolution in Sect. 4. Sect. 5 discusses the issue of supporting backward and forward compatibility under schema evolution, while Sect. 6 examines the object structures supported in object database products and their effect on schema evolution. Sect. 7 provides a summary and discussion of our findings and concluding remarks are given in Sect. 8.

## 2 Background

In object databases, the schema evolution process consists of two main parts: evolution of the class schema followed by evolution of the object instances to reflect these changes. We will refer to these as *class evolution* and *data evolution*, respectively. Data evolution may require new values to be initialised or existing values to be restructured or transformed in some way and products typically offer an API which allows developers to write custom evolution code.

Schema evolution has been studied extensively over the years with many solutions proposed for both relational and object databases [14,13]. For object databases, two main categories of solutions have been proposed in research: a) methods that address the evolution of individual classes and b) methods to address the evolution of the entire database class schema.

In the first category, changes to a class include changes to attribute definitions, the creation or deletion of a class, and the creation or deletion of sublasses or superclasses. Such changes are no longer an issue in that most commercial object databases provide support for handling them, although the basic approach varies in terms of whether it is automated or under the control of the developer. For example, Objectivity[1] handles some changes automatically, while db4o[2] provides

[1] `http://www.objectivity.com`
[2] `http://www.db4o.com`

only an API that the developer can use to manage evolution. Versant[3] opts for semi-automatic support in that basic class changes can be handled automatically, whereas complex modification of the class hierarchy require the use of an API they provide.

The second category of solutions focus more on the evolution process and how to manage the evolution of the entire database class schema. For instance, in OTGen [8], the output of a schema change is a copy of the whole database for a new schema. In [5], Clamen proposes a functional mechanism to support schema evolution for centralised and distributed object databases that maintains compatibility for old applications. Other researchers have addressed the issue of backward compatibility of schema changes, e.g. CLOSQL [9]. However, as we will show later, this adds considerable complexity, especially in the case of agile development where evolution is frequent. Another major distinction between approaches is whether they version classes under evolution, see for example [15,3].

Although cycles in object graphs were used in the Herschel project, many research works assume directed acyclic graph (DAG) structures for both the class hierarchy and object graphs. Only a few publications discuss data structures including graphs with cycles. For example, [2] describes object structures with cycles but does not discuss the implementation of a schema evolution solution.

Similarly, the actual object structures supported in object database products cause some issues which have not been addressed by the research community. For example, several products introduce some concept of embedded or dependent objects to meet different requirements with respect to persistence. Specifically, many vendors introduce two categories of objects, first class objects (FCO) and second class objects (SCO). One of the key properties of SCOs that impacts on schema evolution is the fact that they do not have a class schema.

In summary, research has tended to focus mainly on primitives, rules, invariants and semantics of schema changes and less on managing the data evolution process. Further, much of this research was carried out before the adoption of agile development methods where evolution is much more frequent and one cannot assume that it is a case of evolving one complete and correct system into another. In the remainder of the paper, we will show how data evolution often proves to be the most complicated part, especially in very large systems under agile development.

## 3 Herschel System

The Herschel Common Science System (HCSS) is a project of the European Space Agency to support the scientific operation of the Herschel Space Observatory[4], an Infrared Observatory Satellite that was launched in May 2009. On the pre-observation side, the HCSS software provides the whole chain of submitting, evaluating and scheduling proposals for observation, instrument programming, editing of calibration tables and finally the generation of control commands for

[3] `http://www.versant.com`

[4] `http://www.esa.int/science/herschel`

the satellite. On the post-observation side, it is used for storing, extracting, calibrating, post-processing and distributing observational data. Excluding the post-processing, virtually all data is stored in object databases (Figure 1).

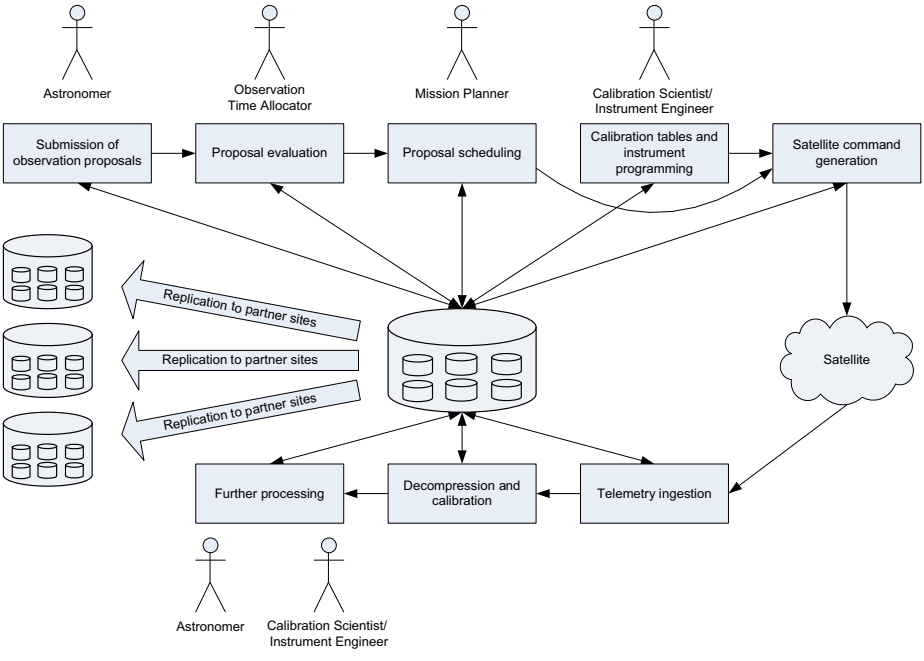


**Fig. 1.** Data flow between the subsystems of the HCSS software. All data (except for the last process) is stored in a central database system with multiple nodes. The database system is replicated to associated research institutions.

The operational time of the satellite is limited to 3-4 years, during which the expected data is in the order of $2*10^9$ objects which will amount to 10-15TB[5]. The data is partially replicated to other sites, so in total there are expected to be over 100 interconnected database nodes in the system. The software is implemented in Java 6, and the database used is the Versant Object Database accessed via Versant's Java interface (Versant JVI).

The project followed the agile manifest[6] quite closely, emphasizing continuous user involvement. In particular, the following agile practices used in the project were relevant to schema evolution: Continuous integration with builds being used by interested users, frequent user releases (6-8 weeks), partly test-driven development, continuous evolution and re-evaluation of design and requirements

[5] This depends on the usage of the different observation modes of the satellite, each of which produces data at a different rate.

[6] `http://agilemanifesto.org`

with close interaction between users and developers. A full description of all agile aspects of the project is outside the scope of this paper.

Agile development was the method of choice for this project for several reasons. Scientific satellites do not follow an off-the-shelf design but feature unique capabilities, requirements and operational concepts. Furthermore, HCSS is the first ever project in ESA using such a complete end-to-end software suite to operate a satellite. Therefore, there was little experience with similar software.

Starting with a general architecture and an initial set of user requirements, the software architecture evolved continuously as users gained experience with operating the satellite hardware which was still in the laboratories. Similarly, the capabilities and requirements of the satellite hardware continuously evolved, leading to more adjustments to the initial design and requirements of the software. The continuous use of early software releases by users (scientists) with their instrument hardware ensured that problems and any misunderstandings common to multi-party development[7] had been cleared and that users were familiar and satisfied with the resulting software. The fact that most parts of the software would only be used by few expert users made the project ideal for close interaction between these users and developers.

In summary, one major achievement of using agile concepts was that, at the time of launch, not only were the users highly experienced with the software, but also the software itself was really mature and could be trusted with the operation of a multi billion Euro space observatory. This reliability was especially critical because the lifetime of the satellite is strictly limited to 3-4 years[8], meaning that every lost day would have cost over one million Euro.

According to plan, development is now frozen for most parts of the system since launch, except for critical bug fixes. The only part still under development is the data processing module. Similar to the other parts of the software, which continuously evolved with the hardware and users gaining experience in hardware operation, the data processing module now evolves as users gain experience in processing the satellite imagery. Data processing was already under heavy development before launch, as it was also required for satellite hardware development and because it had to be sufficient to process the first incoming data after launch. However, since launch, the software has been developed further due, not only experience gained with real satellite imagery, but feedback from a much larger user community of astronomers who previously had little involvement in the project.

The use of agile development practices was not the only property of the project that was relevant to schema evolution. The size and complexity of the data structure with about 250 persistent classes, including cycles and redundancies, some of which have around $2*10^9$ instances, played a major role. Further, some large database systems consisted of up to 30-40 database nodes which were also replicated to other sites (shared administration). Other features of note are that

[7] USA and many ESA member states contributed software or hardware via national institutes.

[8] Operation is limited by the cooling liquid which is required for the infrared detectors.

the DBAs were mostly non-professional and regular down-times of databases and the system were acceptable.

In the following sections, we will first examine the impact of agility on schema evolution strategies and then discuss the issues of backward/forward compatibility and support for complex object structures in detail.

## 4 Agile Schema Evolution

In the HCSS project, the use of agile development methodologies had considerable impact on schema evolution. For schema versioning, the project chose a concept where only one version of a class schema was accessible in any database. To ensure schema compatibility, each database contained a *database schema ID* which defined the schema versions in the database. The database schema ID allowed connection only from a limited range of software versions with a correlating *software schema ID*. The compatible software typically comprised only one user release and correlating nightly builds. The mechanism with the single ID can also be used to prevent accidental connection by indicating an ongoing or failed schema update by setting the ID to a defined invalid schema version.

The schema evolution procedure was split into two parts, each performed by a dedicated application. The first application required database downtime to perform evolution of the class schema and critical data. The application also incremented the database schema ID. Where possible, the evolution of non-critical data was performed later by a second application which did not require downtime as it performed data evolution on a live database. Consequently, this requires clients to tolerate data that has not yet evolved.

The two applications were maintained by one dedicated developer who collected schema change requests and draft code for data evolution from other developers. According to a schedule, the developer would then implement and release the schema changes and the two schema evolution applications. The task of implementing schema evolution was simplified by the fact that all persistent classes were under the custodianship of the database developer.

In order to evolve a database, the DBAs had to execute the evolution applications on their databases. The challenges of implementing schema evolution applications included usability for clients with little database expertise, evolution of large databases, evolution of a database system comprising multiple dependent databases and the evolution of data. In particular, implementing the data evolution could be difficult. First, it had to account for sometimes very complex calculations of initialisation values for new fields based on data from many other objects. Second, it also had to deal with variably inconsistent databases caused by occasional bugs in the frequent user releases, occasionally used less stable nightly builds and even custom applications.

The effects, both negative and positive, of using agile development practices included:

**Constantly evolving design.** Constantly evolving design results in continually incoming requests for schema evolution. The HCSS project decided not

to implement forward or backward compatibility between schema versions for reasons discussed in the next section, instead using the concept of schema IDs introduced earlier to enforce compatibility between a database and any application accessing it. Based on the schema change requests collected together with draft code for the data evolution, the developer responsible for schema evolution implemented the schema evolution tool and scheduled a release date.

**Fast release cycle.** To avoid holding back new features and fixes, schema evolution had to be performed at least once per user release. To accommodate the fast release cycle of only 6-8 weeks (Figure 2) and give developers time of uninterrupted development between releases, the schedule usually placed schema evolution in the week before the following code freeze. This left only one week for beta-testing the changes.

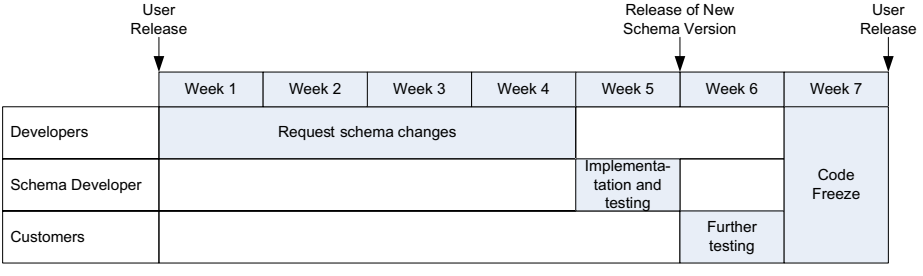


**Fig. 2.** Software release cycle with respect to the release of new schema versions.

**Close interaction with users.** During the one week of beta-testing, the close interaction with the users allowed them to run the new schema evolution code on copies of their most critical databases. The close interaction also made it possible to fix any issues that arose which would have not been possible otherwise.

**Evolving incomplete or incorrect data.** The beta-testing by users was essential because users often used nightly builds, patched builds and even custom software, which resulted in sometimes rare and unpredictable inconsistencies in their databases. Some inconsistencies would only occur in one or two of the more than 100 databases across the whole project, were virtually impossible to foresee and would not occur during previous internal tests by the schema evolution developer.

**Flexibility.** Another aspect of the close interaction with the users was the flexibility of the schedule, which was every time agreed with all developers and users, and had at times to be changed to avoid impact on other tasks.

**Stacked evolution.** The frequency of schema evolution and the fact that users sometimes skipped a release also meant that any schema evolution process had to support evolution of databases over several schema versions at a time (stacked evolution), without requiring the user to install any intermediate

releases. At the same time, the software should not accumulate schema evolution code for every schema evolution, so it was decided to implement a central repository from which schema evolution applications would download evolution code for older databases, if required. Database servers that did not have internet connections were accommodated by making the repository portable to custom locations.

**Maturity and training.** By the time the project switched from the development phase to the critical operational phase, the procedure and user interface for schema evolution were matured and users were routinely applying the tools to their databases.

## 5 Forward and Backward Compatibility

Forward and backward compatibility for schema versions is a frequently discussed topic in research related to schema evolution. Forward compatibility allows old software to access a newer database, while backward compatibility allows new software to access an older database. Although compatibility and the resulting flexibility is an obvious advantage, the HCSS project chose not to support forward and backward compatibility for several reasons:

**Old software accessing new data.** Allowing old software to access a new database also implies that old known problems in that software continue to affect data in the database. In other words, locking out old software implicitly enforces a minimum patch level for all accessing applications.

**New software accessing old data.** Allowing a new software version to access old data means that new software always has to be prepared to encounter old inconsistent data that would have otherwise been fixed by schema evolution.

**Forward compatibility.** This requires additional implementation effort, because it is the inverse implementation of normal schema evolution.

**Code cluttering.** The code allowing backward compatibility is similar to the schema evolution code. However, having this code in the client has the disadvantage that there have to be as many versions of the code as there are schema versions. In the HCSS project, there were over 35 schema versions in 5 years. This can significantly affect performance if old objects have to be evolved over multiple versions.

**Errors during data conversion.** Failures in the forward or backward conversion code confront the end-user with unrecoverable application errors which can only be fixed by developers or DBAs.

**Performance and quality of service.** Loading objects from an incompatible schema version can impact performance in the case of complex evolution algorithms or algorithms that require loading of additional objects. The latter can even cause the data evolution code to hang or cause *other* applications to fail, because the additionally required objects may cause unexpected locking problems. These issues are aggravated by the possibility that some objects may need to be evolved over many schema versions. An additional problem is that these issues only occur for objects of particular schema versions, meaning that application performance can become unpredictable.

**Forensics.** Determining the cause of database inconsistencies gets considerably more complicated through the increased number of software versions accessing the database and through the different paths of data access, namely multi-level forward or backward conversion, or direct access.

**Separation of concerns.** From a design point of view, schema evolution code does not belong in the domain of user applications. The design in the HCSS project kept normal applications free from such code.

A direct consequence of separating evolution code from clients is the need for *standalone tools*, i.e. applications, for schema evolution. Besides avoiding the above problems, some advantages of standalone evolution tools are:

- By the time the tool is finished, no more failures can occur, because all objects are evolved.
- If the tool requires a downtime, it can take advantage of being the only application accessing the database. For example, disabling locking improves performance and simplifies the evolution code.

The HCSS project implemented two distinct schema evolution tools. The first requires database downtime and evolves the class schema and some of the data. The second tool runs later in parallel to other client applications and evolves remaining data.

To minimize the required downtime, the first tool uses a kind of *partially delayed class evolution*, known as lazy evolution in Versant terminology. This means that the tool can update a class that requires only a simple change such as adding a primitive attribute within less than a second, independent of the number of instances of this class. The instances are later evolved transparently when the objects are accessed by clients. This functionality allows only simple updates such as removing or adding an attribute initialised to '0'. Apart from the minimised downtime, there is virtually no impact on client applications, because they need no extra code, the initialisation to '0' is very fast, and it is unlikely to fail.

The second tool runs later in *parallel to other applications* and evolves non-critical data. Non-critical data refers to data where clients are either prepared to encounter non-evolved data, or where the client itself is not critical and can afford an extended downtime until data evolution is finished. In the HCSS project, this delayed data evolution was only used for data with large cardinality that would have otherwise increased the required downtime by several hours.

We conclude that including schema evolution code in client applications needs careful evaluation. Especially projects that can afford database downtime are likely to fare better with simpler solutions. We can see advantages of forward and backward compatibility for short-term use to support smooth transition between schema changes when the additional effort is justified by the possibility of avoiding database downtime altogether. However, our experience with the HCSS project showed that its simple approach can be superior given the right circumstances. In the HCSS project, even for larger databases, using the above concept with two distinct tools resulted in downtimes rarely exceeding a few

minutes. So far, we could not find any research paper that discusses all of the above issues or gives recommendations for cases in which forward or backward compatibility should be implemented.

## 6 Object Structures

While most research tends to assume a relatively simple and pure object model, in practice, many object databases have constructs with special semantics, such as second class objects (SCOs) or non-DAG structures, that require special handling in schema evolution. We will first look SCOs and their counterpart first class objects (FCOs). The JDO specification [7] defines them as follows:

> "A First Class Object (FCO) is an instance of a persistence-capable class that has a JDO Identity, can be stored in a datastore, and can be independently deleted and queried. A Second Class Object (SCO) has no JDO Identity of its own and is stored in the datastore only as part of a First Class Object."

Versant uses an equivalent definition [16] in their (non-JDO) Java API, where *JDO Identity* is substituted by *Object Identity*. Objectivity supports a similar concept to SCOs for both Java [10] and C++ [11] called *embedded objects*. A common feature of SCOs is that they do not have an identity in the database, and their existence depends on an FCO or another SCO that references them. Note that primitive types, arrays and `java.lang.String` (in Java) are generally supported and not considered as SCOs.

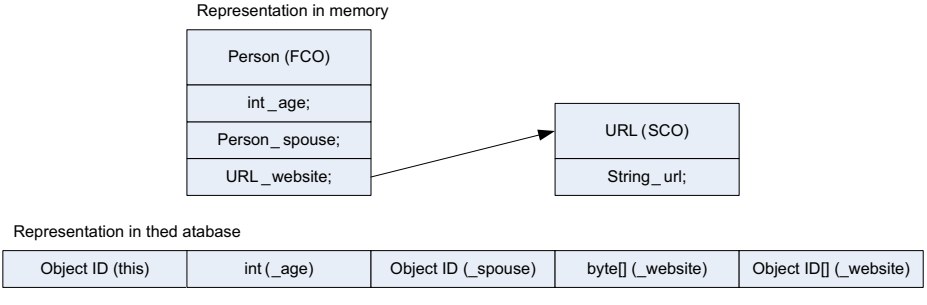


**Fig. 3.** FCO (Person) references SCO (URL)

Figure 3 shows an example of an FCO of class `Person` and an SCO of class `URL`. `Person` has a schema defined in the database, but `URL` has not. Looking at Versant, when a `URL` is stored in a database, it becomes part of the `Person` and is serialized into an array of bytes and an array of object IDs for storing possible references from the SCO to other FCOs. The relationship changes from aggregation in memory, where the `URL` is an independent object, to composition in the database which means that deleting the `Person` from the database will also delete the `URL`.

The properties of SCOs can be summarised as follows:

**No schema.** They do not have a schema defined in the database.

**References to FCOs.** When serialised, references from SCOs to FCOs are replaced by their respective object IDs. Referenced SCOs are serialised in-line.

**Not managed.** SCOs typically do not contain any database related code (inserted manually or automatically via byte-code enhancement (Java) or macros (C/C++)). This allows the storage of third party objects as SCOs in a database. One resulting issue is that they are stateless (see below).

**Dependent objects.** While stored in a database, they behave much like *dependent objects* [2] in that they can have only one owner, and their existence depends on that owner. Like dependent objects, they can reference other dependent or independent objects. Their owner can be an independent or dependent object. These restrictions do not apply to SCOs in memory.

Although common in practice, to the best of our knowledge, no research paper has yet fully discussed SCOs. In particular, we have not seen any mention of the implications of the above properties. Some implications and issues are:

**Stateless / Not managed.** SCOs do not have state flags and access is not monitored. For example, an SCO cannot be marked dirty, and modifying an SCO from unmanaged code will not flag its FCO as dirty either. Instead, an FCO has to manually be flagged as dirty whenever one of its SCOs is modified.

**Queries and indexing.** We do not know of any object database that supports indexes or queries on SCOs.

**SCO duplication.** SCOs are vulnerable to object duplication. Duplication occurs when an SCO is referenced by two FCOs. When the FCOs are stored, the FCO will be serialised twice in the database, resulting in two separate SCOs as shown in Fig. 4. This results in unwanted object duplication since when they are later loaded from the database, the two FCOs will no longer reference the same SCO but two distinct copies of the original SCO.

**Uncontrolled schema evolution.** One problem with SCOs that was encountered in the HCSS project occurred when a developer changed one of the serialised classes, but was not aware of the implications regarding schema evolution. The software continued to function correctly, because storing objects still worked and because the affected type of objects was rarely read. The problem was only discovered a few weeks later when someone tried to look at objects that were serialised before the class was changed and found that de-serialisation of these old objects failed. Versant provides no API dedicated to schema evolution of SCOs, but writing a custom de-serialisation class solved the problem.

A similar problem can occur when the SCOs stem from a third party library. Libraries usually only keep their API stable, not their internals. Therefore, any library update may contain changes in the fields of its classes. Furthermore, clients may have different versions of a library installed, with compatible APIs but possibly incompatible internal fields.

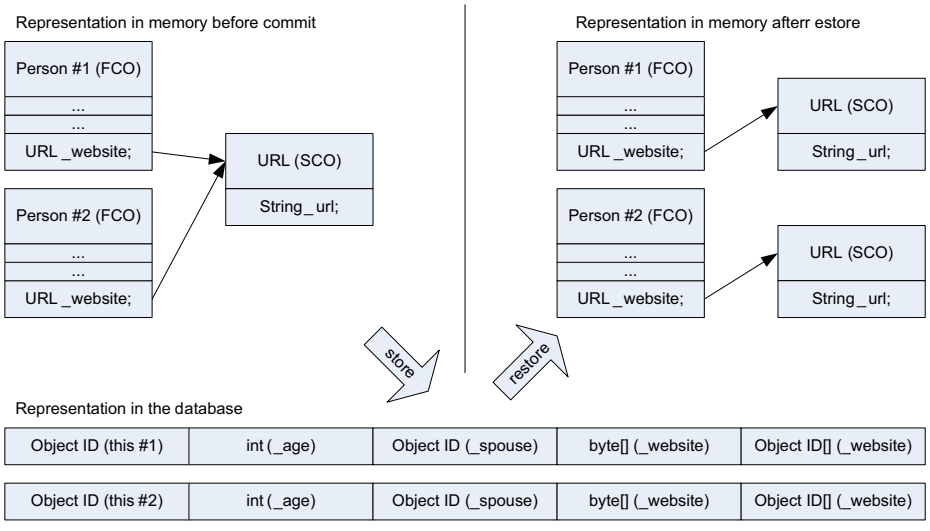


**Fig. 4.** SCO duplication during storage

**False schema evolution.** In Versant, only objects of classes that implement the `Serializable` interface can become SCOs. The catch here is that even changes that do not directly affect the schema, such as changing a field to `final`, can cause serialisation exceptions, because the `serialVersionUID` changes. This problem can be avoided by explicitly specifying this ID when the class is created, with the disadvantage that real schema changes may not be detected if the ID is not correctly updated.

**Varying support.** Support of SCOs varies between vendors such as Objectivity [10] (containers, `Date` and `Time`), Versant [16] (containers and classes that implement `java.io.Serializable` or `java.io.Externalizable`) and ObjectStore [12] (SCOs are completely prohibited).

Despite these problems, SCOs can be quite useful. There seem to be two main reasons for using SCOs.

The first reason is persistence of third party classes. Storing third party classes as SCOs directly in a database avoids having to map third party data to local classes. Often third party classes cannot be made persistent, mainly because the injection of management code (byte code enhancement in Java or macros in C/C++) is difficult or even impossible with third party libraries. Enhancing third party classes can also be problematic if the application requires these classes to have a certain structure, e.g. when serializing them into files or for network transport and communication with other applications. Using third party SCOs can also cause problems like *false schema evolution* described above.

The second reason for using SCOs concerns performance and storage efficiency. SCOs have no storage overhead and improve performance because they do not need unique object identifiers nor information about their location in

a database. In the case of small SCOs, performance is also improved because SCOs provide a very efficient way of implementing clustering and group-load operations without special API calls.

In the HCSS project, SCOs were only used for the second reason, improving performance of numerous small objects. One example was the logging of function calls, with the parameters being stored as an array of type-value pairs, each pair being an SCO. We conclude that SCOs are useful and safe when used carefully. Our recommendations regarding SCOs in first party classes are to use `serialVersionUID` (in Java) to avoid *false schema evolution* and to have a dedicated developer manage SCO classes along with FCOs, possibly in a separate package in Java, to avoid accidental modification by other developers. For third party SCOs, we recommend ensuring that third party libraries can be strongly linked to the product so that users cannot accidentally load a different version of the library. It can also not be emphasized enough that third party library updates need to be checked very carefully and that a plan should be prepared for performing library updates that cause SCO incompatibility.

Two other common features of object structures encountered in practice have received little or no attention by the research community. The first of these concerns the general assumption in most fundamental research on schema evolution (for example [17]) that the *class hierarchy* is a DAG structure with all of its complexities. While single inheritance is technically covered by this research, no research investigates how much schema evolution is simplified by languages like Java that do not allow multiple inheritance.

As mentioned previously, many researchers also assume that the *object graph* is a DAG structure. While DAG-like data structures are clean and may be used in some projects, cycles and other redundancies are often introduced for pragmatic reasons. The HCSS project frequently used cycles to provide short cuts for navigation for applications with differing access patterns and to simplify queries with joins. Similarly, the HCSS project sometimes stored the same data in fields of different classes to avoid navigation altogether, thereby allowing for better indexing or avoiding joins in queries.

## 7 Discussion

As we have shown, the use of agile development methods has a strong impact on schema evolution. The issues resulted from the increased frequency and amount of schema evolution requests, from reduced time for implementing and testing schema evolution code and from the widespread use of nightly builds and custom applications, which lead to unpredictable inconsistencies in the databases and further complicated the testing of schema evolution code.

These issues were mitigated by other aspects of agile development and by additional techniques to manage schema evolution, for example the close interaction with end users which allowed for regular and efficient beta-testing of schema evolution code. Also, the use of the two distinct schema evolution tools proved to be very helpful, especially because their simple handling posed

little problems to users. Finally, the policy of accumulating changes and releasing them at a regular, yet flexible, interval reduced the impact on other tasks.

Regarding the often proposed concept of forward and backward compatibility, we found that it can result in considerable problems when combined with agile techniques. The HCSS project did not implement forward and backward schema compatibility, instead using a simple yet powerful one-version concept, which was shown to be advantageous in all respects except the need for downtime. The advantages can be summarised as reduced development effort, better and more predictable application performance and the fact that only DBAs will be faced with schema evolution problems. Here we see an opportunity for further research that examines the impact of agile development methods on projects that use object databases. Moving away from backward and forward compatibility, this research could investigate alternative concepts and compare their fitness for different project settings.

Due to the use of SCOs and non-DAG structures, many schema evolution solutions proposed in research were not applicable. While their use may appear unorthodox, we believe that their benefits cannot be ignored in real-life projects like the HCSS. SCOs need careful handling, but can improve performance significantly. Non-DAG structures could be exploited to greatly simplify access code and improve performance and did not cause notable complications during schema evolution. We believe that the use of both SCOs and non-DAG structures deserve and could benefit from further research.

Considering the implementation of schema evolution code, another opportunity for further research would be the implementation of tools and a standardised API[9] for schema evolution, possibly embedded in an IDE. Contrary to frameworks proposed in earlier research [4,9,17,6], it would focus on the challenges of agile development and provide a standardised solution for multiple major vendors. We are currently planning to design such a framework based on our experiences from the HCSS project. We also plan to investigate improved support for SCOs and similar concepts, multi-node schema evolution and, depending on requirements, a system for deployment of stand-alone schema evolution applications as used in HCSS.

Finally, we note that in HCSS data evolution tended to be considerably more complex than class evolution, but is an issue that has received much less attention in research.

## 8 Conclusion

The HCSS project shows that agile development methodologies have a strong impact on schema evolution. Some agile aspects clearly increase schema evolution effort, while other aspects and good management practices help mitigating the effect. Considering technical decisions like forward and backward compatibility or details of object structures, we found that many of the questions faced during

[9] The latest JDO draft 2.3 proposes such an API.

the HCSS project are at most only briefly discussed in existing research literature and provide opportunities for further research.

## Acknowledgements

We would like to thank Tilmann Zäschke's former employer VEGA[10], his former colleagues at ESA[11] who worked on the HCSS project and of course ESA itself, who own the HCSS project.

## References

1. Ambler, S.W.: Agile Techniques for Object Databases (September 2005), http://www.db4o.com/about/productinformation/whitepapers/
2. Banerjee, J., Chou, H.T., Garza, J.F., Kim, W., Woelk, D., Ballou, N., Kim, H.J.: Data Model Issues for Object-Oriented Applications. ACM Transactions on Information Systems 5(1), 26 (1987)
3. Bjornerstedt, A., Britts, S.: AVANCE: an Object Management System. ACM SIGPLAN Notices 23(11) (1988)
4. Breche, P., Ferrandina, F., Kuklok, M.: Simulation of Schema Change Using Views. In: Proc. 6th Int. Conf. on Databases and Expert Systems Applications (1995)
5. Clamen, S.M.: Schema Evolution and Integration. Distributed and Parallel Databases 2(1) (1994)
6. Claypool, K.T., Jin, J., Rundensteiner, E.A.: Serf: Schema evolution through an extensible, re-usable and flexible framework. In: Proc. of the 7th Int. Conf. on Information and knowledge management. ACM, New York (1998)
7. Java Data Objects Expert Group. Java Data Objects 2.2. Technical Report JSR 243, SUN Microsystems Inc. (2008)
8. Lerner, B.S., Habermann, A.N.: Beyond Schema Evolution to Database Reorganization. In: Proc. ECOOP (1990)
9. Monk, S., Sommerville, T.: Schema Evolution in OODBs Using Class Versioning. ACM SIGMOD Record (1993)
10. Objectivity, Inc. Objectivity for Java Programmerís Guide Release 9.4 (2007)
11. Objectivity, Inc. Objectivity/C++ Programmerís Guide Release 9.4 (2007)
12. Progress Software Corporation. PSE Pro for Java User Guide Release 7.1 (2008)
13. Ram, S., Shankaranarayanan, G.: Research Issues in Database Schema Evolution: The Road Not Taken. Technical report, Boston University School of Management, Department of Information Systems (2003)
14. Roddick, J.F.: A Survey of Schema Versioning Issues For Database Systems. Information and Software Technology 37(7) (1995)
15. Skarra, A.H., Zdonik, S.B.: The Management of Changing Types in an Object-Oriented Database. In: Proc. OOPSLA 1986 (1986)
16. Versant Corporation. Java Versant Interface Usage Manual Release 7.0.1.4 (2008)
17. Zicari, R.: Primitives for Schema Updates in an Object-Oriented Database System: A Proposal. Computer Standards & Interfaces 13(1-3), 271–284 (1991)

[10] http://www.vegaspace.eu
[11] http://www.esa.int