Object Evolution by Model Evolution

Roland T. Mittermeir, Helfried Pirker, Dominik Rauner-Reithmayer*
Universität Klagenfurt
Institut für Informatik-Systeme
Universitätsstraße 65-67, 9020 Klagenfurt, Austria
{mittermeir, helfried, dominik}@ifi.uni-klu.ac.at

Abstract

Claims concerning the maintainability of object oriented software usually refer to encapsulation and inheritance mechanisms. However, if objects are perceived only from the code level, potentials for higher level maintenance operations are missed. Instead, classical maintenance destroys the relationship that once existed between specification and implementation.
We present an approach supporting object evolution by specification evolution such that for a substantial class of changes, the ensuing changes in the implementation can either be automatically performed or the complience with the new overall specification can be assured by automatically generated, well focussed test suits.

1. Motivation

To keep evolving software from structural deterioration, special effort is needed [6]. If evolution is not enforced by a very rigid maintenance process, it results in a divergence of the product from the requirements- and design documents used for its original development. Parnas [12] refers to this phenomenon as "software aging". Albeit encapsulation and other beneficial features, "objects" too are not immune against aging. This paper suggests service channels as an approach to software evolution that will keep the aging process relatively benign. The substance of this claim is due to keeping a firm relationship between the model (specification) of an object (resp. class) and its realization in some programming language.

The motivation behind our approach is, that maintenance of complex (and/or large) objects [1] is far from trivial. However, since objects not only encapsulate their state with respect to their environment but have their methods built around this state, following some common and coherent concept, maintenance can be supported to a higher degree than in plain procedural code.

This concept a software object represents is concisely represented in the object's specification. There, the state space and the methods reporting and manipulating it are given in an implementation independent form. As each method is coded as implementation of its respective specification, parts of the common concepts will become implicit in the implementation. Nevertheless they remain in the form of the specific realization of the state space or in the form of other interdependencies of methods. Thus, constraints that have been obvious to the initial implementor might need to be painfully uncovered by the maintenance programmer.

Thus, to alleviate the maintenance programmer from much of the burden to uncover implicit implementation constraints,

- code and specification of objects should co-evolve;
- maintenance activities are to be supported by relating the specification (*model level*) to the respective representation on the *implementation level*;
- for a class of maintenance operations the relationship between a modified specification to the respective code can be automatically maintained via *service channels*.

Thus, for large existing object-oriented software systems maintenance will become easier and certain reverse engineering operations will be even unnecessary or could be performed automatically.

The consequence of the approach is, though, that objects are formally specified and that maintenance activities should be planned and performed first on the specification- or model level. From there, one can identify, whether the implementation needs to be modified "by hand" or whether a change on the model level can be automatically propagated to the implementation level.

*This work was partially funded by the Austrian Research Promotion Fund (FWF), grant Nr. P 11.340-ÖMA.

[1] When using the term "object" in this paper, we generally refer to the class description.

In the rest of this paper we first discuss conventional object evolution and present then object evolution based on model evolution. To benefit from model evolution, service channels are proposed. Section 4 discusses the basic alternatives of how to implement them.

2. Object evolution caused by maintenance

Taking a look at object evolution, two communities can be distinguished. The database community, investigating the evolution of object-oriented database schemas (e.g.[1, 11, 2]), and the software engineering community investigating the evolution of object-(class-) hierarchies during the development process (e.g.[7, 8]). In software engineering, the end of the development process marks also the end of the examination of the objects evolution (or history). However this point is not the end of object evolution, it just marks the transition from development to maintenance.

In object oriented systems, changes on classes lead not just to different variants but also to different versions of the maintained classes. This can also be viewed as a kind of object evolution producing an object history beyond the scope of development. But in contrast to the object evolution during development, object evolution in the maintenance phase mostly happens just on the implementation level. Even process models assuming some object specification (i.e. description on the model level) at the outset, [4], consider this specification just as starting point in development, but not as co-evolving during generalization and/or specialization of the respective object. Thus, an object life-cycle as illustrated in Figure 1 results and the high level information originally contained in the object model becomes progressively obsolete. We claim that this unnecessarily spoils information that has been already available and hence will lead in the end to costly reverse engineering efforts. Figure 1 illustrates, that the more the object evolves, the further it diverges from its initial object model OM. Hence, the initial object model is becoming less and less a documentation or specification of the system. So its value decreases with each evolutionary step.

3. Object model evolution

Departing from the assumption that maintenance on the model (specification) level is less costly than maintenance on the implementation level and certainly less costly then any reverse engineering activity, the process described in Figure 1 is wasteful in the long run.

Therefore we propose an approach for software maintenance on the model level as illustrated in Figure 2. Here *maintenance activities* are done on the *object model* OM. The resulting object model OM' is now the basis for the

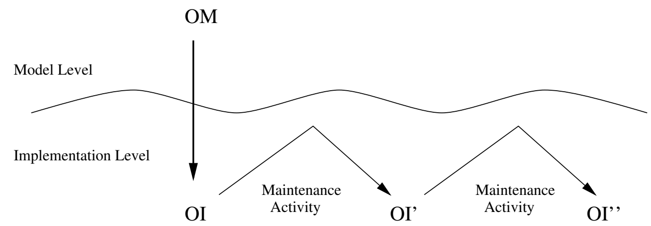

Figure 1. Traditional object evolution

derivation of a corresponding object implementation OI'. In accordance to the object evolution step (from OI to OI') we call the step from OM to OM' as *object model evolution*. The benefits of model evolution can be seen on the model level itself as well as between model and implementation level.

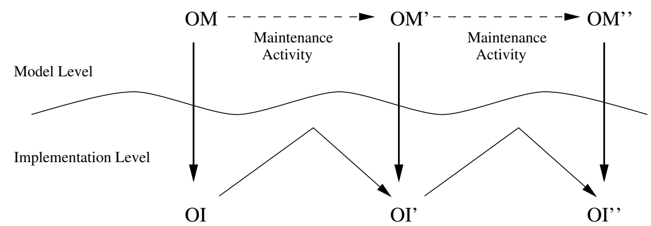

Figure 2. Object model evolution

Co-evolution of model and implementation will yield a set of "benefits of discipline". They fall in one of the following categories:

- Model evolution provides a network of related object models that define a road-map of object evolution and thus also a possibility for *specification based software retrieval* [9] and other forms of focussed software reuse.

- This provides *guidance* concerning the sequence of maintenance steps necessary to consistently build OM' out of OM. By maintenance activity steps we refer to changes such as described in [10, 5]. Taking the hierarchical structure of Kungs classification [5], it is possible to describe the distance between OM and OM' on different levels of granularity.

- *Making explicit constraints* that are (implicitly) assumed as given in (parts of) the implementation. Thus, one can relate those constraints and reason about them and it will become possible to decide whether a maintenance step violates a constraint in the object model that is only hard to find or even not explicitly represented at the implementation level. Constraints of this kind are a main source of the difficulty of program comprehension and hence a recurring source of maintenance and testing problems.

With further instrumentation of the relationships between model- and implementation level, one can obtain additionally:

1. Guided object evolution via *service channels*. This is the most powerful version of object evolution by model evolution.

2. Protected object evolution via *test support*.

3. Object evolution by *plain programming*.

A *service channel* is a mechanism to relate a sequence of transformations on the model level to the code level. In its most powerful version, as *adaptive* service channel, model level changes are propagated automatically to the implementation level. Such propagations are safe against introducing inconsistencies or violations of constraints expressed in the object's model. Thus, a safe transformation from OI to OI' can be guaranteed. These automatic code adaptions are only possible, if the service channel can be sure that there are not any hidden implicit constraints remaining. When this is not the case, service channels can still assume an observing role as *verificative* service channels. Based on the difference between OM and OM' they can be used to generate test-cases [14] for checking OI' against the changes in the specification. The specific benefits from focussed testing in class structures can be seen from [3]. Certainly, one can express modifications on the model level that are beyond the provisions foreseen by any service channel. This applies notably when OM and OM' seem to be unrelated from a tool's perspective. Then, the respective modifications to OI' have to be performed unsupported and no safe transition from OI to OI' can be guaranteed. The maintainer has, however, still the benefit to work in a forward looking manner and does not need to start the task with a reverse engineering activity.

Figure 2 could be interpreted just as a methodological advice. As such, it might already help in lots of situations and be in line with what is currently seen as "best practice". The argument against this disciplined approach is usually stressful maintenance and neglicence to clean up later, what has been missed in an initial rush-job. The argument of doing the same work twice (on the model level and on the implementation level) is raised as an excuse. This excuse – it might never have been valid – is rendered invalid if the sum of work on the horizontal and the downward pointing arrow is less than the work one would have to do on the bent arrow representing implementation level maintenance. Machine support for the vertical arrow will help to turn the economics to where the technical perfection rests. Service channels are proposed as adequate mechanisms to achieve this.

Figure 3 summarizes the idea of object evolution by model evolution using service channels. Whether object evolution is fully supported by service channels, only expost supported by a test data generator, or even basically not

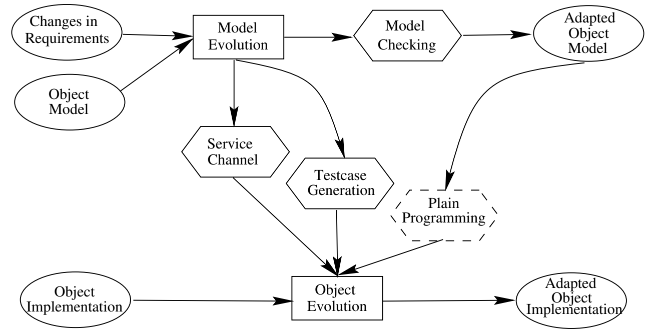

Figure 3. Maintenance with service channels

supported at all so that just the benefits of model checking remain, depends on a classification of the changes on the model level resulting from respective requirements changes.

4. Realizations of service channels

So far, service channels have just been presented as a concept. Now we introduce two quite different options to realize them. A detailed discussion of these variations backed up by example modifications is given in [13].

4.1 Built-in service channels

Built-in service channels are special methods written specifically for the object under consideration, implemented as "service methods", not accessible to the regular "clients" of this object. Such service devices are not a new concept in conventional engineering. We find them as extra functionality due to the engineering knowledge of the developer, built beyond any users request. Examples one might think of range from water pipes built into buildings for use by the fire brigade via staircases or elevators in hotels marked by "personell only" to plugs in cars, where special diagnostic equipment can be connected and test-busses on highly complex VLSI-chips. These examples show already a breadth of purpose as well as the fact, that there is an engineering decision as to how much one builds into the specific object (pipes etc.) and how much one leaves outside for instrumentation on demand (service–plug).

As can be seen from the engineering examples, built-in service channels are designed with full knowledge of the design of the artefact they are built into. This applies to software service channels too: They "know" the object's model, and for the spectrum of changes they are to support also the relationship between model and implementation.

For an obvious example we refer to the relationship between the state space, its implementation and its realization in various methods. Assume a requirements change leads to an extension of the state space. The service channel supporting this change will identify all those parts in the implementation that need to be changed, in case the change can be performed in a simple way (e.g. changing a constant) it will perform this change on the source code level and after recompilation, the object's implementation will be consistent again.

Under certain conditions, one could also conceive of such modifications on the fly. Its discussion is beyond the scope of this paper though.

4.2 External service channels

The example given above demonstrated that normal operation of an object and operating its service channel are quite different operations. While during normal operation, its state will be changed, operating its service channel changes its state space. Hence, it is not an operation on the instance level, but – to borrow data base terminology – on the schema level. Since we are dealing with software, recompilation is the normal consequence.

This very different usage pattern motivates the question why such an operation has to be built-in and not kept separate from the object as an independent tool. Obviously, this is a valid alternative. We are referring to such special tools as *external service channels*. Their main difference to built-in service channels can be seen again from an analogy: Considering the fire brigade, a fire-man on a ladder sprinkling water out of a hose to a burning building would be the "external" alternative to the built-in pipes and sprinklers.

These are specific maintenance tools, designed independently of the specific object they are operating on.

External service channels are specific maintenance tools, designed independently of the specific object they are operating on. Their purpose is to identify change, change propagation and limits to change propagation. An external service channel consists of general tools for program understanding and reverse engineering such as slicers, ER- or structure charts generators etc. With them, support can be given for change categories not anticipated and therefore infeasible to deal with by built-in service channels. With the external service channel, the conceptional network that is preestablished in the internal service channel will be defined on the fly. A consequence is, that the maintenance support they provide will be reduced. To improve their performance, special *service plugs*, such as explicit links between identifiers used at the model level and identifiers used in the implementation can be provided.

5. Conclusion

This paper discusses concepts to improve maintenance of object oriented software beyond conventional code-level inheritance mechanisms. Based on formal specifications, service channels have been proposed to instrument links between the specification and implementation of objects. Thus, specification and implementation of objects evolve in a consistent manner and aging processes due to successive maintenance are blocked or reduced.

The power and generality of service channels depends on their specific architecture. Hints for the main architectural choices have been given. For a detailed discussion we refer to [13].

References

[1] J. Banerjee, H.-T. Chou, H. J. Kim, and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. *SIGMOD RECORD*, 16(3):311–322, 1987.

[2] E. Casais. Managing Class Evolution in Object-Oriented Systems. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 204–244. Prentice Hall International (UK) Ltd., 1995.

[3] M. J. Harrold, J. D. McGregor, and K. Fitzpatrick. Incremental testing of object-oriented class structures. In *Proc. 14th International Conference on Software Engineering (ICSE'92)*, pages 68 – 80, 1992.

[4] B. Henderson-Sellers and J. Edwards. The object-oriented systems life cycle. *Communications of the ACM*, 33(9):142 – 159, Sept. 1990.

[5] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. Change Impact Identification in Object-Oriented Software Maintenance. In *Proc. of the International Conference on Software Maintenance*, pages 202–211, 1994.

[6] M. M. Lehman. Programs, life cycles and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060 – 1076, Sept. 1980.

[7] K. J. Lieberherr and C. Xiao. Object-Oriented Software Evolution. *IEEE Transactions on Software Engineering*, 19(4):313–343, Apr. 1993.

[8] S. Matsuura, H. Kuruma, and S. Honiden. EVA: A Flexible Programming Method for Evolving Systems. *IEEE Transactions on Software Engineering*, 23(4):296–313, May 1997.

[9] R. Mili, A. Mili, and R. T. Mittermeir. Storing and Retrieving Software Components: A Refinement-Based System. *IEEE Transactions on Software Engineering*, 23(7):445–459, July 1997.

[10] R. Mittermeir and K. Kienzl. Intra-Object Schemas to Enhance Adaptive Software Maintenance. In *Austro-Hungarian Software Engineering Seminar*, 1993.

[11] S. Monk and I. Sommerville. Schema Evolution in OODBs Using Class Versioning. *SIGMOD RECORD*, 22(3):16–22, 1993.

[12] D. Parnas. Software aging. In *Proc. 16th Int. Conference on Software Engineering (ICSE'94)*, pages 279 – 287, 1994.

[13] H. Pirker and R. T. Mittermeir. Service Channels - Purpose and Realization. Technical report, Univ. Klagenfurt, 1997.

[14] P. Stocks and D. Carrington. A Framework for Specification-Based Testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, Nov. 1996.