Service Channels - Purpose and Tradeoffs

Helfried Pirker, Roland T. Mittermeir, Dominik Rauner-Reithmayer*
Universität Klagenfurt
Institut für Informatik-Systeme
Universitätsstraße 65-67, 9020 Klagenfurt, Austria
{helfried, mittermeir, 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. Moreover, classical maintenance focussing just on code destroys the relationship that once existed between specification and implementation.

We present service channels as mechanisms to support maintenance requests on the specification and, for a substantial class of requests, the propagation to the implementation keeping the relationships between specification and implementation consistent.

Keywords: *Software Re-engineering, Maintenance and Customization*

1. Motivation

To keep evolving software from structural deterioration, special effort is needed [10]. 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 [19] refers to this phenomenon as "software aging". Albeit encapsulation and other beneficial features, "objects" too are not immune against aging. Thus maintenance support for object-oriented systems appears desirable. In [17] service channels are suggested 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.

The 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, part 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 (cf. assumptions in [11]).

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*.

The principles mentioned above are uncontested, but nevertheless unresolved. So we propose that,

- 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

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

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

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.

In the rest of this paper we first present our considerations on performing maintenance operations on objects by object model evolution. Service channels are proposed as a mechanism to support these maintenance operations. Then possible realizations of service channels and their purpose are discussed using an illustrative example.

2. Maintenance operations on objects

2.1. Object model evolution

As outlined in [17], object evolution [1, 18, 4, 12, 13] is usually investigatied during the development process. But in contrast to object evolution during development, object evolution in the maintenance phase mostly happens just on the implementation level. Hence, the initial object model is becoming less and less a documentation or specification of the system. So its value decreases with each evolutory step.

Assuming 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 by the dashed line in Figure 1 is wasteful in the long run.

Therefore, we propose an approach for software maintenance on the model level as illustrated by the solid line in Figure 1. Here *maintenance activities* are done on the *object model OM*. The resulting object model *OM′* is then the basis for the 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′* *object model evolution*.

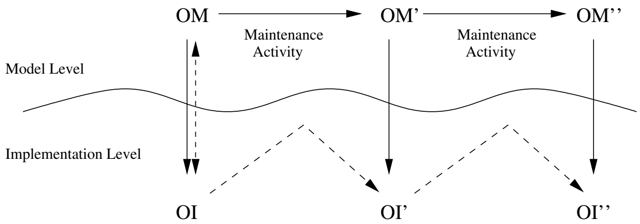

Figure 1. Object model evolution

The benefits of model evolution can be seen on the model level itself as well as between model and implementation level.

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* [2, 14] and other forms of focussed software reuse.

- Model evolution 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 [15, 8, 7]. Taking the hierarchical structure of Kungs classification [7], it is possible to describe the distance between *OM* and *OM′* on different levels of granularity. Kung et al. use three levels of granularity in their classification. E.g. when considering changes on classes, they distinguish between component changes, like add (delete) a defined data attribute, and relationship changes like add (delete) a subclass.

- *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.

2.2. Service channels to support model evolution

A *service channel* is a mechanism relating a sequence of transformations on the model level to the code level. To do so, they instrument the relationships between model- and implementation 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 possible, 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 [20] for checking *OI′* against the changes in the specification. The specific benefits from focussed testing in class structures can be seen from [6].

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 1 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". With rush-jobs, it seems counterintuitive to do stressful maintenance on both, code and specification. But neglecting specification level maintenance is quite often coupled with negligence to clean up later what has been postponed initially. 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.

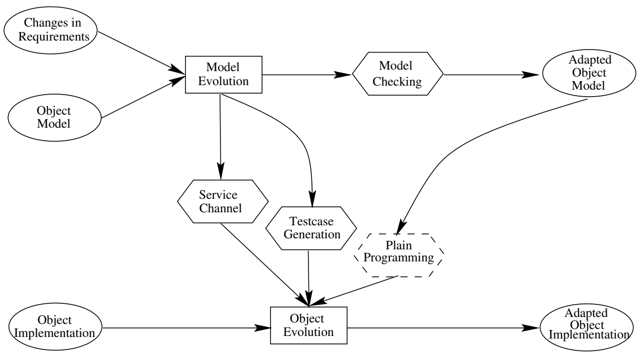

Figure 2. Maintenance using model evolution

Figure 2 summarizes the idea of object evolution through model evolution using service channels. Whether object evolution is fully supported by adaptive service channels, only ex-post supported by a test data generator (verificative service channel), or even basically not 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.

3. Realizations of service channels

So far, the purpose of service channels has been presented. Now we introduce two quite different options to realize them. Our considerations how to realize service channels are driven by the question, how much of the systems ability to evolve should be represented inside the system (built-in service channel) or outside the system within a maintenance environment (external service channel).

3.1. Built-in service channels

A *built-in service channel* (BI-SC), as shown in Figure 3, is a modification mechanism inside the system developed to cover a fixed set of changes. These changes are formulated as change requests on the model level and are propagated automatically on both the model and implementation level, by the built-in service channel.

To do so, the built-in service channel has to know about all relationships between *OM* and *OI* and of all implicit contraints hidden in the implementation, which are necessary to perform the change automatically.

Hence, the object model, its corresponding implementation and the built-in service channel together represent a set of possible solutions within the application domain. Conceptually, the invocation of a built-in service channel with a certain change request determines the suitable solution in the set of possible solutions.

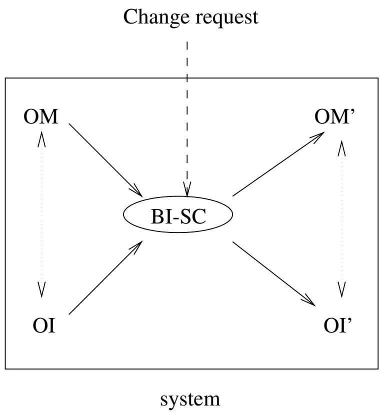

Figure 3. Built-in service channel

For example, built-in service channels can be realized as 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 "personal 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. Of course, each such service channel is limited to its spectrum of changes. Hence, it needs to know the relationships between *OM* and *OI* that are relevant for change requests falling into this spectrum.

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 check for ripple effects, perform this change on the source code level and after recompilation, the object’s implementation will be consistent again.

3.2. External service channels

The example given above demonstrated that normal operations 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* (EX-SC), whose relation to the system to be maintained is shown in Figure 4. 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.

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 (e.g. [9]), ER- or structure charts generators [21] etc. With them, support can be given for change categories not anticipated and therefore

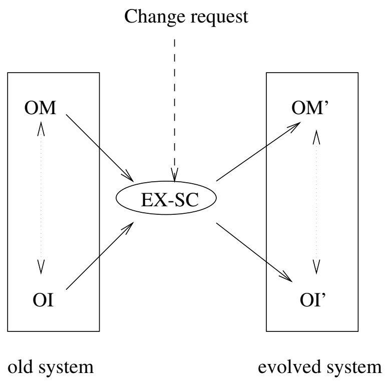

Figure 4. External service channel

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.

4. An illustrative example

The following example demonstrates the usefulness of object model evolution even in the seemingly trivial case of modifying constant values. We use (a part of) a system to display the temperature measured by a thermometer to sketch the basic ideas of the concept.

The thermometer, as originally designed, measures and supplies the temperature in centigrades. It covers a range between -50 and 50 centigrades. The system then displays the temperature using a two-digit display and a sign indicating wether the temperature is below 0 or not.

4.1. The object model

As basis for the discussion, the *object specification* in our *object model* is represented using *Object-Z* [3, 5]. It consists of two objects, *TempDisplay* and *Thermometer*.

The *Thermometer*-object is the interface to the physical thermometer. The history invariant represents the temperature range of the physical thermometer. *GetTemp* obtains the temperature from the physical thermometer and stores it in the state variable *temp*, *YieldTemp* presents this value on request to the caller of this method. *TempDisplay*, for the sake of presentation modelled as a distinct object, requests the current temperature from *Thermometer* and prepares it for output on a two-digit display.

Thermometer

$\upharpoonright(\mathit{INIT}, \mathit{GetTemp}, \mathit{YieldTemp})$

$temp : \mathbb{Z}$

GetTemp

$\Delta(temp)$

$temp' = \ldots$

/* stores temperature supplied by physical thermometer */

YieldTemp

$temp\_out! : \mathbb{Z}$

$temp\_out! = temp$

INIT

$temp = 0$

$\square(-50 \leq temp \leq 50)$

/* temperature range of physical thermometer */

TempDisplay

$\upharpoonright(\mathit{INIT}, \mathit{ShowTemp})$

$SignDigit ::= \text{"}+\text{"} \mid \text{"}-\text{"}$

$digit1, digit2 : \mathbb{Z}$
$sign : SignDigit$

ShowActTemp

$\Delta(digit1, digit2, sign)$
$temp : \mathbb{Z}$

$-99 \leq temp \leq 99$

$temp = Thermometer.YieldTemp$

$temp \geq 0 \Rightarrow$
$\quad (digit1' = temp \bmod 10$
$\quad digit2' = temp \operatorname{div} 10$
$\quad sign' = \text{"}+\text{"})$

$temp < 0 \Rightarrow$
$\quad (digit1' = -temp \bmod 10$
$\quad digit2' = -temp \operatorname{div} 10$
$\quad sign' = \text{"}-\text{"})$

INIT

$digit1 = digit2 = 0 \wedge sign = \text{"}+\text{"}$

$\square(0 \leq digit1 \leq 9)$
$\square(0 \leq digit2 \leq 9)$

4.2. Implementation rationale

Due to the simplicity of this object model, the corresponding implementation is not given here. But the following points should be considered.

The behaviour of both objects is restricted by some invariants. The *TempDisplay*-objects invariants

$$A: \quad -99 \leq temp \leq 99$$

(in the sequel referred to as constraint *A*) and

$$B: \quad \square(0 \leq digit1 \leq 9) \wedge \square(0 \leq digit2 \leq 9)$$

(both summarized as constraint *B*) are driven by the requirement of a two-digit display and are interrelated in the following way. Knowing one of them and the way to calculate the values of the two digits (digit1, digit2), implies the other one

$$A \Rightarrow B \quad \text{and} \quad B \Rightarrow A.$$

The *Thermometer*-object has only one invariant

$$C: \quad \square(-50 \leq temp \leq 50)$$

(in the sequel referred to as constraint *C*) representing some physical limitations of the thermometer.

Implementing this specification, one notes that the value of the variable *temp* in the *Thermometer*-object (restricted by constraint *C*) determines the value of the variable *temp* in *TempDisplay* (restricted by constraint *A*). Thus, *Thermometer.temp* can be substituted for *temp* in *C* and because under this substitution *C* is stronger than *A*, the chain of implications

$$C \Rightarrow A \Rightarrow B$$

holds. Therefore, it is safe and efficient to check in the implementation only for *C*. There is no actual need for constraints *A* and *B* to appear within the implementation of the *TempDisplay*-object.

This reasoning will be done by any programmer looking for performance and for compact code. If, however, the algorithmic transformations would have been much more complex than those between *Thermometer.temp* and *digit*1 or *digit*2 it might have been hard for the maintenance programmer, to see, why this system functions correct even with out-of-range input. Here, an explicit link between the identifiers used in the implementation and those of the specification, that can be traced by the machine might already prove to be a valuable aid and constitute a minimal service channel.

We will now consider such a service channel when discussing two requirements changes.

4.3. Changing the temperature range

One possible change in the requirements could be the fact, that our thermometer should be able to measure temperatures in the range between -80 and 80 centigrade.

4.3.1 Considerations on the model level

This change influences the *Thermometer*-object, as the history invariant has to be changed as follows:

$$C' : \quad \Box\,(\,-80 \,\le\, temp \,\le\, 80\,)$$

(referenced as constraint C'). As C' remains stronger than A the change can be done locally in the *Thermometer*-object without any consequence for *TempDisplay*. The unique spot of modification can be identified safely, by proving that no other constraint on the model level is inflicted.

4.3.2 Considerations on the implementation level

To support the change of the temperature range on the implementation level, the service channel has to know about the implementation of the *Thermometer* object. The service channel has to take care of the following:

- the implementation of the variable *temp* and
- the checking of constraint C in the implementation.

Other objects are not to be considered, as the service channel has proved on the model level, that the requested change has no effect on the *TempDisplay* object.

One can easily imagine several possible implementations of the variable *temp* (subrange, enumeration, integer, ...) within the Temperature object and their automatic adaption according to the requested change as long as basic restrictions (e.g. range of the implementation types) are not violated.

A more powerful service channel might even directly perform the modification as long as

$$C' \Rightarrow A \Rightarrow B$$

holds. One might conceive of many variations of this example. Among those, variations of the last implicant so that the whole chain of implications still holds.

Although this seems not very realistic in this example, it might lead to situations, where changing the respective spot on the model level and running an automatic proof would be all that is needed. To figure this out on the basis of an unsupported implementation might be quite hard.

4.4. Changing the thermometer

Obviously, not all changes are that benign that they can be (almost) fully taken care of on the model level and via service channels. Nevertheless, some help is provided.

As example of such a change, assume the physical thermometer supplying centigrades is exchanged to one supplying temperature measured in degrees Fahrenheit over the same temperature range. In this case the physical thermometer will provide values in the range between -112 and 176 degrees Fahrenheit (corresponding[2] to the range between -80 and 80 centigrades).

4.4.1 Considerations on the model level

This change on the model level again concerns only the invariant in the *Thermometer*-object. Thus, from a naive perspective it is of the same nature as the previous one, a simple change of constants such that the new constraint C'' is now

$$C'' : \quad \Box\,(\,-112 \,\le\, temp \,\le\, 176\,).$$

If this change is straight done on an implementation, developed according to the rationals mentioned in section 4.2, the object *TempDisplay* cannot display all values supplied by the *Thermometer*-object. As constraints A and B are not checked in the implementation it will show undefined behaviour at very low or relatively high values. Although this problem should at least be identified during boundary value testing, not all test strategies might catch it.

Performing this change on the model level though, one will easily see that constraint C'' is no longer stronger than A. This violates the assumption

$$C'' \Rightarrow A \Rightarrow B$$

and since the connection between the implementation and specification does not show any direct check for compliance with A, an open proof-obligation for A will figure and a ripple effect from the *Thermometer* object to the *TempDisplay* object will be diagnosed.

In this case a service channel will not automatically insert code for dealing with a third digit and modifying A and B appropriately. It will however clearly point to the fact that A is violated and that *TempDisplay* needs to be modified in consequence of the modification in *Thermometer*. Thus, focussed modification "by hand" is possible. Likewise, the set of test data can be automatically extended (or even modified).

[2] x centigrade = (x * 1.8) + 32 degrees Fahrenheit

5. Discussion

5.1. Discussion of example

The maintenance activities described above show that even simple requirements changes may lead to severe differences in implementations. Doing maintenance only on the implementation level might cause inappropriately high effort and allows the implementation to drift away from its specification. It can also cause undefined behaviour of the system. Therefore, we propose to perform object model evolution before stepping down to the implementation level.

Doing more elaborate changes than the simple change of constants sketched above can also lead to violations of system invariants possibly not mentioned in the implementation (e.g. a change of the calculation of the digits such that measurements obtained in centigrade are displayed in Fahrenheit can also violate constraint *B*).

Of course, given the simplicity of the example one might wonder, why we need service channels to deal with changes a smart programmer might have solved by appropriate parameterization. However, critique of this kind misses the point, that system generation and system execution are conceptually different activities. System modification has to do with system execution and with service channels, model consistency can be assured and more powerful generation/modification-operations can be performed than with simple parameterization. After all, even this simple example and its discussion might have shown the limits of parameterization, limits that cannot be fully covered even by very defensive programming.

Using a specialized theorem prover for the ripple effect analysis discussed seems appropriate. Such a theorem prover can be considered as an external service channel like the special analyzer needed to identify the links between predicates that will be submitted to this analysis.

Consider now extending the example by a *Temperature-tracker*. This tracker just consists of a *collector*, an array of frequency-counters over the full temperature-range, and a *reporter* showing a list of temperature-frequency pairs. Assume this array is, based on the original specification, implemented as *int collector [101]*. Apparently, there is a direct relation between the bounds of *temp* in the specification of *Thermometer*, the upper bound of the array *collector* and the temperature readings to be printed in *report*. However, since the array- (and loop-) boundaries are a function of *temp*'s domain boundaries, while the temperature readings are direct values of the domain at least two different constants, with one dependent on the other, are needed. An internal service channel will be a suitable mechanism to take care of this.

The simplicity of the example did not show another important isssue of using service channels, namely the question, whether a change request causes *sequencing constraints* [16] or not. The example demonstrated the use of service channels to do ripple analysis to isolate the parts of the object model and of the corresponding implementation affected by the change request. The issue here is, whether the modifications can be done in an arbitrary order or not. If the modifications can not be done in an arbitrary order, there must be a mechanism to determine the possible order(s).

5.2. Disadvantages of object model evolution

Of course, doing maintenance by the two step approach described (model evolution before object evolution) might lead to a higher effort in certain situations. This effort can be divided into two main parts: The effort spent for adapting the object model can be balanced when regarding the advantages of our approach and the derivation of the implementation. The latter effort can be huge, as small changes in the model can lead to vast changes in the implementation.

We claim though, that proper definition of service channels will reduce the likelyhood of situations where the two-step approach leads to measurable higher effort. These cases might be justified by less effort in other situations or they might be written off on the ground of better product quality.

5.3. Advantages of object model evolution

Our approach offers advantages on both model level and implementation level. Describing maintenance requirements on the model level is easier and more precise. It is easier, because the level of abstraction of the object model is closer to the level of abstraction of the maintenance requirements. It is more precise, as the object model is represented by some formal model, e.g. Object-Z.

The ability to describe maintenance activities on the model level has two main advantages. As we have a sequence of maintenance activities on the model level we can give a sequence of changes on the implementation level, having the same semantics.

Having the information, that the activity violates none of the constraints of the model, we can conclude, that the corresponding changes on the implementation level will lead to no unwanted side-effects. Therefore, it is possible to distinguish two possible ways of object evolution: One supported by a service channel maintaining an already established level of consistency between implementation and specification; the second one is doing the changes by plain programming. Test data generation can support the second way, but might even have its merits in the first one.

6. 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 as mechanisms relating a sequence of transformations of the specification to the implementation by instrumenting 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.

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] N. Boudriga, A. Mili, and R. Mittermeir. Semantic-based software retrieval to support rapid prototyping. *Structured Progamming*, 13:109–127, 1992.

[3] D. A. Carrington, D. Duke, R. Duke, P. King, G. A. Rose, and G. Smith. Object-Z: An object-oriented extension to Z. In S. Vuong, editor, *Formal Description Techniques II, FORTE'89*, pages 281–296. North Holland, 1990.

[4] 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.

[5] R. Duke, R. King, G. Rose, and G. Smith. The Object-Z specification language. Technical Report 91-1, University of Queensland, Dept. of Computer Science, Software Verification Research Centre, May 1991.

[6] 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.

[7] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. On regression testing of object-oriented programs. *Journal of Sytems and Software*, 32(1):21–40, Jan. 1996.

[8] 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.

[9] L. Larsen and M. J. Harrold. Slicing object-oriented software. In *18th International Conference on Software Engineering*, pages 495 – 505, 1996.

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

[11] M. M. Lehman. Software's future: Managing evolution. *IEEE Software*, 15(1):40–44, Jan. 1998.

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

[13] 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.

[14] 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.

[15] R. Mittermeir and K. Kienzl. Intra-object schemas to enhance adaptive software maintenance. In *Austro-Hungarian Software Engineering Seminar*, 1993.

[16] R. T. Mittermeir and H. Pirker. Internal service channels - principles and limits. In *Proc. International Workshop on Principles of Software Evolution (IWPSE98)*, pages 63–67, 1998.

[17] R. T. Mittermeir, H. Pirker, and D. Rauner-Reithmayer. Object evolution by model evolution. In *Proc. 2nd EUROMICRO Conference on Software Maintenance and Reengineering (CSMR98)*, pages 216–219, 1998.

[18] S. Monk and I. Sommerville. Schema evolution in oodbs using class versioning. *SIGMOD RECORD*, 22(3):16–22, 1993.

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

[20] P. Stocks and D. Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, Nov. 1996.

[21] M. P. Vrbicky. Rekonstruktion von Structure Charts und intermodularem Datenfluß aus C Quellcode. Master's thesis, Universität Klagenfurt, 1997.