Helfried Pirker

Specification Based Software Maintenance
(a Motivation for Service Channels)

Dissertation

zur Erlangung des adademischen Grades
Doktor der technischen Wissenschaften

Studium: Angewandte Informatik

Universität Klagenfurt
Fakultät für Wirtschaftswissenschaften und Informatik

1. Begutachter: o.Univ.-Prof. Mag. Dr. Roland T. Mittermeir
2. Begutachter: o.Univ.-Prof. Dipl.-Ing. Dr. Johann Eder

Klagenfurt, September 2001

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich, daß ich die vorliegende Schrift verfaßt und alle ihr vorausgehenden oder sie begleitenden Arbeiten durchgeführt habe. Die in der Schrift verwendete Literatur sowie das Ausmaß der mir im gesamten Arbeitsvorgang gewährten Unterstützung sind ausnahmslos angegeben. Die Schrift ist noch keiner anderen Prüfungsbehörde vorgelegt worden.

Klagenfurt, 12. September 2001

Abstract

Software aging and structural deterioration are phenomena which increase the software maintenance effort. One of the main goals in software maintenance research is to keep the aging process of the system benign. But for achieving that, special effort and tool support is needed.

Object-orientation, often claimed to be a pradigm to facilitate software maintenance and reuse, is not the silver bullet to solve the above problem. Object-orientation itself and concepts like polymorphism, late binding and code tangling cause severe problems in maintenance. Objects are not immune against aging, too.

One path to reach the goal of decreasing the maintenance effort and to avoid structural deterioration is to establish a rigid maintenance process enforcing model evolution before the evolution of the implementation.

By *model evolution* we refer to maintenance activities on a representation of the system on a higher level of abstraction as the source code, i.e. the systems specification or design. We refer to such approaches, applying model evolution before the evolution of the implementation, as *model based maintenance.*

We argue, that the proper level of abstraction for doing model evolution is the systems specification. *Service Channels* as an instrumentation of *specification based maintenance* are introduced.

One key issue in all model based (specification based) maintenance approaches is that a firm relationship between the systems model and the systems implementation has to be established and maintained. Establishing such relationships means identifying and documenting them (in the best case during the systems development).

We argue, that service channels are a proper representation to capture the relationships between the systems model and implementation, and furthermore service channels provide active maintenance support on the model level for anticipated changes.

Most arguments against specification (model) based maintenance approaches state the additional documentation effort as the main disadvantage.

In this work we try to invalidate these arguments by showing, that most of the additional documentation can be acquired by consequently performing requirements traceability throughout the whole development process and by using static and dynamic analysis tools.

Zusammenfassung

”Software Alterung” (software aging) und der ”strukturelle Verfall von Software” (structural deterioration) sind Eigenschaften, die Software, die sich in Wartung befindet aufweisen kann und die den notwendigen Aufwand für Wartungsaktivitäten erhöhen. Eines der Ziele der Forschung im Gebiet der Software Wartung ist es daher, den Alterungsprozess von Software zu unterdrücken. Um dieses Ziel zu erreichen benötigen die Verantwortlichen für die Wartungsaktivitäten eine spezielle Werkzeugunterstützung.

Das objekt-orientierte Paradigma, dem zwar nachgesagt wird Software Wartung und Wiederverwendung zu erleichtern, kann die Software Alterung auch nur bedingt aufhalten. Objekt-orientierte Konzepte wie z.B. Polymorphismus, Late Binding, und Code Tangling erleichtern das Ausführen von Wartungsaktivitäten keineswegs. Die Folge ist, das auch Objekte ”altern”.

Ein Weg um den Aufwand für Wartungsaktivitäen zu senken und den ”strukturellen Verfall von Software” zu verhindern, ist die Einführung eines Wartungsprozesses, der eine Evolution auf Modellebene der Software (model evolution) vorsieht, bevor auch die dazugehörende Implementierung (Source Code) geändert wird.

Evolution auf Modellebene bedeutet, daß die vorhandenen Beschreibungen des Systems wie Entwurfsmodelle oder formale Spezifikationen, entsprechend den Wartungsanforderungen geändert werden. Ansätze, die eine Modell Evolution vorshen, werden auch als *modellbasierte Wartung* bezeichnet.

Wir sind überzeugt, daß formale Spezifikationen die richtige Abstraktionsebene ist, um Modell Evolution durchzuführen. Um eine solche *spezifikationsbasierte Wartung* zu unterstützen werden in dieser Arbeit *Service Channels* eingeführt.

Eine Kernfrage in allen modellbasierten und spezifikationsbasierten Wartungsansätzen ist es, Beziehungen zwischen ”zusammengehörenden” Elementen auf der Spezifations- und Implementierungsebene zu identifizieren, zu dokumentieren und zu warten. Die Identifikation und Dokumentation sollte möglichst schon bei der Entwicklung des zu wartenden Systems geschehen.

Wir sind überzeugt davon, daß Service Channels die richtigen Mechanismen sind, um solche Beziehungen zwischen Spezifikation und Implementierung zu fassen und zu dokumentieren. Darüberhinaus stellen Service Channels auch aktive Wartungsunterstüzung für zu erwartenden Änderungen im System zur Verfügung.

Gerade die Dokumentation dieser Beziehungen im System und der dafür notwendige Aufwand werden oft als Argumente gegen spezifikationsbasierte Wartungsansätze gennant. In dieser Arbeit soll auch aufgezeigt werden, daß der Großteil der notwendigen Dokumentation damit erreicht werden kann, bekannte Techniken wie Requirements Tracing konsequent während der Entwicklung einzusetzen. Ebenso erleichtern statische und dynamische Analyse Werkzeuge das Auffinden solcher Beziehungen.

Contents

1 Introduction

This section gives an overview of factors influencing software maintenance and of its state of the art. A motivation for this work and a road map through the work will be given.

1.1 Why focus on software maintenance?

Without performing an empirical study one can easily observe, that society, business and administration is becoming more and more dependent on software (information) systems. Investments in development and acquisition of such systems are considerable.

Much work in academia and industry has been spent on making the development of software systems faster and cheaper and on making the developed systems reliable and responsible to the initial requirements.

Shipping, installing the new system and training the users mark the end of development activities and put the system into work, which will possibly generate errors and/or new requirements. As the average lifetime of software systems is about 10 years [TT92], it is not surprising, that 40-90 % of the total life-cycle costs spent on a software system are spent for post-development (i.e. maintenance) activities [Ben97].

Reasons for the amount of effort spent on post-development activities can be found in [Par94, Leh80, Leh98].

Lehman's *Laws of Program Evolution* [Leh80] describe characteristics of software systems in use. The first law, the *law of continuing change* says, that a software system in use has to undergo continual change or will become less useful. As the system implements the users requirements it also has to reflect the changes in these requirements. If it does not reflect these changes, it loses its value to the user or to the organization using it. Parnas refers to that fact as *software aging* [Par94]. One cause for software aging is that the system does not reflect the above changes.

Lehman's second law, the law of *increasing complexity* says that the complexity of a system undergoing the above mentioned continual changes increases. This is due to the fact, that the personnel responsible for the initial development of the system is not maintaining it. Assumptions and rationales made by the requirements analysts, designers and programmers may not be known to the staff (the maintainers) responsible for post development activities. By making a change to the system, some of these assumptions or rationales may be violated, thus making the system less understandable. Parnas states this observation as the second cause for software aging. Blum refers to that phenomenon as the *software maintenance paradox* [Blu95], stating that the more experience one gains with a system, the more difficult is the maintenance of that system. He also states that "...the older the system is, the more costly it is to maintain."

Hence, the brief considerations mentioned above motivate to focus on software maintenance. Lehman sees in software evolution (and maintenance) "... the key to our survival in this computer age." [Leh98].

1.2 Software Maintenance

This section should briefly sketch the field of software maintenance and provides some definitions [Ben97, KH98].

1.2.1 Some Definitions

Software maintenance focuses on post-development activities on software systems already delivered to the customer. By following the IEEE definition, maintenance is a sub-area of software engineering. It defines *software engineering* [IEE91] as:

> the application of the systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is the application of engineering to software.

Software maintenance itself is defined as [IEE91]:

> the process of modifying the software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a change in environment.

The above definition of software maintenance already indicates different types of software maintenance. Bennett states four different categories of software maintenance [Ben97]:

- *Perfective maintenance* enforces changes to the system as a result of user requests for improved functionality of the system.
- *Adaptive maintenance* enforces changes due to changes in the systems environment.
- *Corrective maintenance* is the identification and elimination of errors and faults in the system.
- *Preventive maintenance* covers activities and changes in order to make the system more maintainable.

When looking at the above definitions, the main focus is on *change*. Performing changes (whatever category of maintenance they are belonging to) is not a trivial task (cf section 1.3). Change requests are usually expressed in changes in the users needs, in changes of the systems behavior. These change requests have to be transformed into changes in the source code of the system. Consultation of documents (if available) like the requirements specification, design documents, user and technical documentation facilitate maintenance. These documents have also to be kept up to date, i.e. must also reflect the performed change.

Other maintenance tasks are change *impact analysis* (determining the side effects of a change) or the management of different versions of the system (*configuration management*).

Maintaining a system means keeping the system up to date with the users requirements, in other words the aging process of the system is delayed. At some point in the life cycle of the system, a decision has to be made, whether the system should be replaced or not [Sak94].

The term *software evolution* is not commonly defined. Some authors refer to software evolution when focusing on *planned changes to the system*, and to maintenance in general when focusing on unplanned changes to the system. Lehman on the other hand sees software evolution as the *unification of the software development and maintenance processes* [Leh98].

1.2.2 Factors Affecting Software Maintenance

In order to make it easier to compare different software maintenance approaches, tools and empirical studies, some authors are trying to define an ontology for software maintenance. The first step was to define the factors influencing software maintenance. Figure 1.1 [KvN+99] is presenting the five key issues of software maintenance: the *maintenance process* itself, the *service type*, the *people* involved within software maintenance, the *(software) product* itself, and the *software (development) process* which lead to the the product to be maintained.

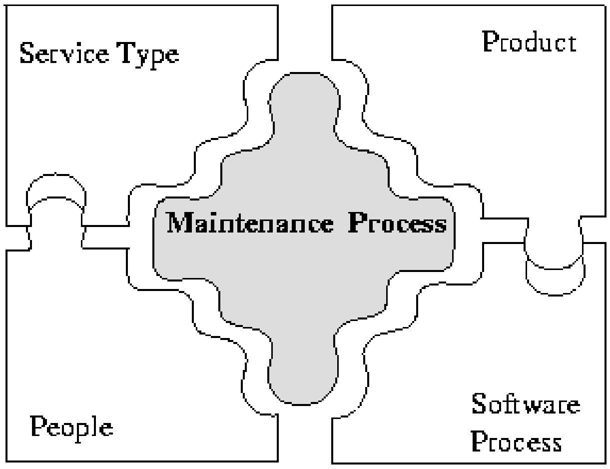

Figure 1.1: Factors Affecting Software Maintenance [KvN+99]

The Maintenance Process

The maintenance process describes how maintenance activities are organized. A maintenance process description is necessary, as maintenance activities are to be planned and should not be undertaken in a chaotic trial and error manner.

Maintenance process models are presented in section 3.1.

The Service Type

The service type refers to the different types of maintenance activities as defined in the previous section. The different service types are adaptive maintenance, corrective maintenance, perfective maintenance and preventive maintenance.

The Product

Another important factor influencing software maintenance is the product itself. Product relevant issues might be the size of the system, its age (in the sense of time as well as in the sense of David Parnas [Par94], cf section 1.1) or the products composition.

Composition [KvN+99] refers to which artifacts on different levels of abstraction are available to the maintenance team. The question, whether they have access to requirements documents, design models, specifications and additional documentation influences the productivity and quality of the maintainers work.

The Software Process

The software development process itself influences the maintenance process. The basic questions here are, whether development was aiming at developing a maintainable system or even more pragmatic whether the development environments or tools are available to the maintainers.

The People Involved

The last, but not the least factor influencing software maintenance are the people doing it. Questions here are the skills of the team members, how is the group organized and how are they managed. On the other hand, it is also important to know the type of users or customers working with the system to be maintained.

1.3 State of the Art of Software Maintenance

This section aims at providing insight into the state of the art of software maintenance by describing current practices in software maintenance and by taking a closer look on how object-orientation influences the software maintenance tasks.

1.3.1 Practices in Maintenance

This section states known problems and maintenance practices in software maintenance. The main goal is to identify problem areas in the software maintenance and to put a mirror to the idealistic introduction of software maintenance in section 1.2.

Documentation issues

The main cause for problems in software maintenance is the fact, that the maintenance personnel is not the same personnel who developed the system to be maintained. So maintainers heavily rely on the available documentation.

In many cases, the only document available is the source code of the old system to be maintained. Hence reverse engineering [CC90] activities have to be performed to get a better understanding of the system. In Browns study [BCD95] one of the major findings is, that *reverse- and re-engineering tools are seen as the most required tools within maintenance projects* due to the lack of documentation.

In other cases documentation is available, but it is not up to date with the system which is actually in use. This is due to the fact, that changes are made in the source code and these changes are not manifested in the documentation. Parnas refers to that phenomenon as *structural deterioration* [Par94]. Singer reports, that the biggest problem with the systems documentation was its maintenance [Sin98]. So there is a need for maintenance environments and rigorous maintenance processes enforcing the update of the systems documentation in order to complete the maintenance task. It has to be noted here, that the requirements specification or design documents have also to be kept up to date.

Another interesting finding in Singers study [Sin98] is, that even if documentation is available and is up to date with the system in use, it is *not consulted by the maintenance personnel*.

The primary source of information is the source code itself. Second, the maintenance personnel would consult the author of the source code and then other persons, who already worked with the systems source code. Jørgensens study [Jøg95] provides empirical data about the use of the different information available to maintenance personnel. For solving only 2 % of the examined maintenance tasks, the maintenance personnel consulted the user documentation or user manuals, for solving of 6 % of the tasks the application system documentation (e.g. design documents) was consulted, for 8 % of the tasks language or tool documentation was used as a source of information about the system.

On the other hand, users of the system (19 % of the tasks), system personnel like database administrators (23 % of the tasks) or other maintenance personnel of that system (29 % of the tasks) were consulted to gain more information about the system.

Another issue is, that *documentation is considered un-trustworthy* by maintainers. They state that documentation is often inconsistent and out of date or too vast and complex to be of use [Sin98, Yip95].

Maintenance personnel trusted more in documentation describing the architecture or design of the system. Hence, a more abstract description of the system is considered more useful and is believed to be more correct.

To summarize, one of the important problem areas in software maintenance is the existence and the quality of the documentation of the system.

Testing

Browns study [BCD95] compared several maintenance projects within the same organization. An interesting result was, that there is no common approach to systematic testing of the changed code. The maintenance personnel was either not up to date with current practices and approaches in testing or there was no time to introduce or experiment with these current practices.

It is not possible to generalize the result of one study, but testing seems not to be considered that important in software maintenance as it should be.

Configuration Management

The findings about configuration management are similar to those about testing. Brown stated, that all of the examined projects saw configuration management as a crucial element in support of their maintenance task, but there was no common understanding of it. Each project had its own configuration management system and policy. The only fact in common was the need of better configuration management support in the sense of more automated support [BCD95].

Focus of maintenance tasks is the source code

In the previous sections it was stated that keeping the documentation (design documents, requirements specification) up to date was one problem area and that structural deterioration is one of the consequences.

This is due to the fact, that the main focus of software maintenance is the source code [Sin98]. Changes are done on the source code first, the documentation is (frequently) updated afterwards.

To avoid this, a maintenance process has to be implemented, enforcing the evolution of the specification and design before the evolution of the source code. The problem here is, that e.g. design models (object models) are too general and too abstract. Most of the necessary changes in source code are not visible in the object model.

Lindvalls study [LR98] examined the visibility of changes in the object model (design document). In the system examined the object model is a design model provided by the Objectory CASE tool and the source code is the corresponding C++ implementation. Only 40 % of the changes in the source code classes are visible in the classes according the object model.

This is due to the fact that object models are too abstract to represent details of their implementation. Information about method bodies is usually not included in object models. In Lindvalls study, 40 % of the changes were changes of elements in the bodies of methods without changing the methods interface. So these changes were not visible in the object model.

1.3.2 Maintenance and Object-Orientation

Object-Orientation is, beside its benefits for requirements analysis and the development of software systems, also seen as a paradigm that facilitates software reuse and maintenance. Taking a closer look on object-oriented systems, one observes that object-orientation is not the solution to all problems of software maintenance and that OO can cause problems too.

In the advent of the object-oriented paradigm, *object-orientation* was claimed to be the "silver bullet" in software development. Arguments like

- structural clarity / well understood, clean design
- encapsulation / information hiding

were stated to claim that object-orientation not only facilitates the initial development of software systems, but also enforces *extensive reuse* and *easier maintenance*. These claims seemed to be fair, but there was a lack of empirical evidence [Jon94].

Several studies showed [BBDD97, KHJ97, DBM+95, HGK+95, CvM93, WH92, LMR92], that object-orientation decreases maintenance effort and that in object oriented systems the localization of changes is easier.

On the other hand, complex inheritance trees, late binding, dynamic creation of object instances, overloaded operations, polymorphism, code tangling and code scattering cause substantial problems for the maintenance of object-oriented systems.

1.4 Motivation for this work

As mentioned in the above sections, software aging and structural deterioration are phenomena which increase the maintenance effort. So one of the main goals in software maintenance research is to keep the aging process of the system benign. But for achieving that, special effort is needed [Leh80].

Object-orientation is not the silver bullet to solve the above problem. As stated above, object-orientation cause severe problems in maintenance and objects are not immune against aging, too.

In [BR00] a life cycle model for software maintenance (cf section 3.1) is presented. This model introduces three phases for systems under maintenance. The goal is to keep the system under maintenance as long as possible in the evolution stage. The evolution stage is characterized by the fact, that the system shows architectural integrity (all models of the system are available, up to date and corresponding).

One path to reach this goal is establishing a rigid maintenance process enforcing model evolution before the evolution of the implementation [MPRR98]. By *model evolution* we refer to maintenance activities on a representation of the system on a higher level of abstraction as the source code, i.e. the systems specification or design. We refer to such approaches, applying model evolution before the evolution of the implementation, as *model based maintenance*.

Several approaches for model based maintenance have been presented in the literature. One goal of this work is to present an overview of the diversity of model based maintenance approaches.

Our view on model based maintenance is also presented. We argue, that the proper level of abstraction for doing model evolution is the systems specification. *Service Channels* as an instrumentation of *specification based maintenance* are introduced.

One key issue in all model based (specification based) maintenance approaches is that a firm relationship between the systems model and the systems implementation has to be established and maintained. Establishing such relationships means identifying and documenting them (in the best case during the systems development). A proper representation has to be introduced and the relationships have to be stored in a proper way (repository of a maintenance environment).

Most arguments against model based maintenance approaches state the additional documentation effort as the main disadvantage of these approaches.

A goal of this work is to invalidate these arguments by showing, that most of the additional documentation can be acquired by consequently performing requirements traceability throughout the whole development process and by using static and dynamic analysis tools (e.g. slicers).

1.4.1 Organization of this work

The work is organized as follows (figure 1.2 should serve as a road map through this work):

- Chapter 2 introduces different contexts, where maintenance requests may occur.
- Specification based maintenance is introduced in chapter 3.
- Chapter 4 focuses on the documentation of the relationships within a system. How this additional documentation influences the development process is described in chapter 5.
- Chapter 6 focuses on the key maintenance activities of model based maintenance.
- In chapter 7 a prototype specification for a service channel is presented and a coarse architecture of a maintenance environment for model based maintenance is given.
- The work closes with conclusions and an outlook for further work..
- The appendices present two case studies, a temperature display and a team calendar.

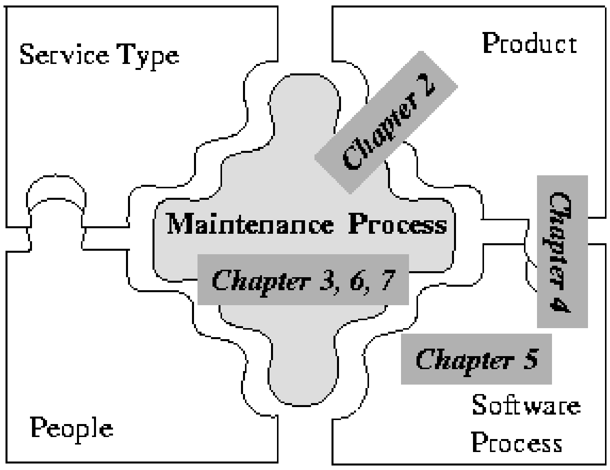

Figure 1.2: Road Map through this work

2

Different Maintenance Contexts

This chapter presents different contexts where maintenance activities are considered/performed throughout the systems life cycle. By context we refer to the maintenance process and to the products composition (in terms of the maintenance ontology).

As sketched in chapter 1, classical software maintenance happens on the source code level using reverse engineering support to gain a better understanding of the code. This chapter gives a more detailed overview where maintenance effort usually happens when initial development of the system has terminated.

One strategy to decrease the effort spent on post development maintenance is to increase the systems maintainability during its initial development. Some approaches following this strategy are presented later on in this chapter.

2.1 What is a Maintenance Context?

A maintenance context describes the situation, in which maintenance activities take place. To describe a maintenance context, the factors defined in the maintenance ontology have to be considered. We want to focus only on the technical issues, i.e. the maintenance process and the products composition.

Our focus is centered on the products composition. The questions here are:

- Which artifacts (components) of the system are available to the maintenance staff?
- On which level of abstraction are those available artifacts (components)?

Figure 2.1 summarizes different maintenance contexts (product compositions). It states, which artifacts are available and indicates the level of abstraction of the artifact. The triangle on the requirements level represents the maintenance requests, which triggers the maintenance activities.

In context A only binaries of the system to be maintained are available. This context is described in section 2.2.1. We refer to this context as *pathological maintenance*.

In context B the systems binaries and the source code are available. This situation is regarded as *classical, source code based maintenance* (cf. section 2.2.2).

Context C represents *model based maintenance*. In addition to the source code, the systems design and/or specification models are available to the maintainance personel (cf. section 2.4).

Context D represents the case, where the system contains additional artifacts *to increase the systems maintainability* (cf. section 2.3.1).

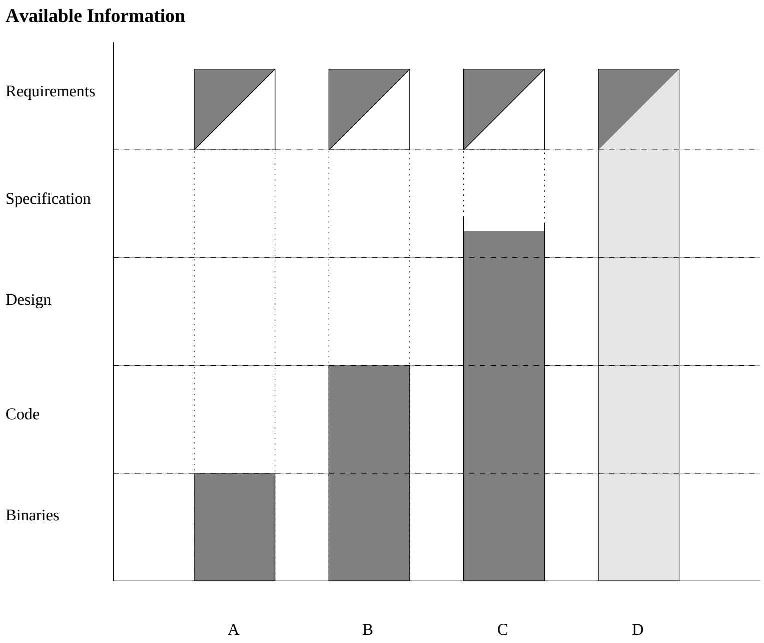

Figure 2.1: Possible Contexts For Software Maintenance

2.2 Classical Software Maintenance

By classical software maintenance we refer to situations and practices as sketched in the previous chapter. Maintenance requests are given to the maintenance team, who should perform the necessary activities on the running system. In most cases the initial developers are not available, as they are already involved in other development projects. The system is in most cases poorly documented and in some cases even the source code is not available. Additionally, in most cases the systems maintainability is not the best.

Doing maintenance under these conditions will be briefly sketched in the following sections on *source code based maintenance* and on *pathological software maintenance*.

2.2.1 Pathological Software Maintenance

In the context of pathological software maintenance, only the binaries or the object code of the system to be maintained exist. No source code, design model, requirements documents or any other system documentation are available. The only way to get information about the system on a higher level of abstraction is to analyze the object code. Decompilation and the application of slicing techniques are two attempts to this problem.

Decompilation in context of reverse engineering for software maintenance has been part of the research in the REDO project [van93]. The interested reader is refered to [BB92a, BB92b]

Static Slicing Techniques applied to binaries [CF97] is also seen as a way to get information out of the systems object code.

2.2.2 Source Code Based Maintenance

The basic activities in software maintenance are localizing where a requested change is to be applied, analyzing its impact on the system, performing the change, doing regression testing and documenting the change. But if only the source code of the system is available, the system has to be prepared to be maintained.

For localizing a change and for doing change impact analysis, the system has to be sufficiently documented. Hence the system has to be redocumented. This *redocumentation* can be done by using tools like static or dynamic analysis tools. With static analysis tools control flow diagrams (CFG) or data flow diagrams (DFD) can be generated out of the source code. Dynamic analysis tools follow and document module calls or the usage of data definitions. Tools and techniques used for reverse engineering [CC90] can also be of use for redocumenting the source code.

Having redocumented the system, *change impact analysis* can be performed on both, a local scope (e.g. within a module) and on a global scope.

After performing the change, regression *testing* has to be performed.

The above activities seem to be trivial, but on the source code level all activities are on the maintainers discretion. The system might be redocumented, but the generated documentation only contains different, more abstract views of the source code. The quality of the change impact analysis, and hence the risk, whether ripple effects are detected or not, depends heavily on the quality and power of the tools used for redocumenting the system. Guidance for performing the change is not provided.

In [Sne91, Leh91, Par86, Ben97, CYL96] more details on source code based maintenance can be found.

2.3 Maintainable Systems

Regarding the empirical results observing software maintenance practices (presented in chapter 1) and reflecting classical maintenance (presented in section 2.2), developing systems with increased maintainability seems to be the key issue for decreasing post development maintenance effort.

Following the maintenance ontology [KvN+99], the focus here is on the products composition and on the software development process.

Software Maintainability is defined as [IEE90]:

> The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.

The goal of this section is to present several different approaches considering maintenance during initial system development in order to increase the systems maintainability. The approaches presented here can be grouped into 2 categories:

- approaches aiming at documenting future maintenance requirements, but providing no operational support for maintenance activities
- and approaches additionally supporting post delivery maintenance activities.

The considerations on software maintainability presented here are of a technical/qualitative nature. Most work in the literature like [LPR98] is on quantitavie aspects of maintainability. These quantitative aspects are not subject of the following sections.

2.3.1 Increasing Maintainability

When trying to avoid the problems caused by classical software maintenance, one strategy is developing more adaptive systems and/or documenting future maintenance requirements already anticipated during the initial development of the system.

Enhancing documentation will decrease the effort needed for understanding the system.

- Hooks into frameworks [FHLS97] are one possibilty for framework developers to document possible future changes and adaptions to an application framework. They can be usefull when the framework has to be adapted.
- Modeling of future maintenance requests as non-functional requirements [FB97] is a more intuitive way to consider possible future changes of a system during development.

On the other hand, one can find in the literature several software development approaches aiming at supporting decentralization of software development or supporting prototyping by offering characteristics like separation of concerns, support in merging several orthogonal views of the system or supporting evolutionary prototyping. Originally these approaches (see list below) were developed for supporting the initial development of software systems, but their underlying concepts and methodologies could also be included in software maintenance environments.

- Design Patterns like the Visitor Pattern [GHJV95, PJ98]
- Adaptive Programming with Propagation Patterns [Lie95]
- Aspect Oriented Programming (AOP) [KLM+97]
- Subject Oriented Programming [HKOS96, HO93]
- Generative Programming [TB99a, TB99b].

The techniques and approaches listed above should lead to more adaptive software, but are not providing (operational) support for maintenance activities.

These approaches also have in common, that when they are used for supporting maintenance activities, they will mostly support code based maintenance.

There is no support for maintenance on a higher level of abstraction.

A summary of these approaches can be found in section 2.5. *Hooks into frameworks* (cf. 2.3.2) and *subject oriented programming* (cf. section 2.3.3) are explained in more detail.

2.3.2 Hooks into Frameworks

An object-oriented application framework represents a generic, design (within a given domain) for a larger-grained problem and a suitable, reusable implementation (a set of abstract classes). Applications are built by customizing or extending the framework.

Although frameworks offer a great potential for reuse, application developpers need to understand the framework and the usage of the framework to effectively build applications from frameworks. Hence, a good documentation of the framework and of its intented usage is needed.

Hooks [FHLS97] focus on documenting and providing guidance on the indented usage of an object-oriented application framework.

Hooks describe

- how a framework is intended to be used
- how a framework is to be changed to meet the requirements of the application
- where (within the framework) changes have to be made

Different Parts of Hooks

Hooks consist of the following parts to describe the intended usage of a framework:

- *Name*: within the context of the framework, a hook needs a unique name
- *Requirement*: the requirements part describes the problem to be solved, hence possible requirments for an application to be built from the framework.
- *Type*: this part describes, what has to be done (method of adaption) and what amount of support will be provided.
- *Area*: describes the affected parts of the framework
- *Uses*: states other hooks required to use this hook
- *Participants*: which existing components (of the framework) or new components participate in the hook

- *Changes*: this part of the hook sketches the changes to be made to interfaces, associations, control flow and synchronization amongst the components listed in the participants part. Also the order of the changes is given.
- *Constraints*: states limits in the usage of the hook
- *Comments*: any further, additional description

Characteristics of Hooks

The type of a hook is described by two main characteristics of hooks: the *method of adaption* and the *provided support*. Both characteristics quickly describe what the hook does and how difficult it may be to use.

- *Method of adaption*
  - *Enabling a feature*, which exists in the framework. Such a hook needs to describe all changes structural and behavioral changes required to enable the feature. Also constraints, like the required exclusion of other features when using the new one, have to be described.
  - *Disabling a feature* having some unwanted properties. In this case, configuration constraints have to be stated.
  - *Replacing an existing feature* is similar to disabling a feature and adding a new one. Here interface and behavioral constraints have to be described.
  - *Adding a feature* is a common way of adaption. Adding new features mostly means extending exsisting classes. The hook needs to describe, where what new classes and operations are needed and where they should be integrated in the framework.
  - *Augmenting a feature or existing behavior* means changing the control flow. The hook should describe, where in the control flow a change has to be made to fullfill the requirements and needs to point out any dependencies between existing and new behavior.
- The *level of support* describes, which support is provided within the framework. The support types for hooks are:
  - *Option Hooks*: a set of pre-built components, existing within the framework, can be chosen and enabled by the application developper.
  - *Supported Patterns Hooks*: here, no complete pre-built components exist. The framework offers a pre-defined method of fullfilling a requirement, application specific details have to be filled in by the application developper.

- *Open-Ended Hooks*: here, the framework does not provide direct support or fullfilling the requirement. It points to places, where changes need to be made and states the known contraints.

Benefits of using Hooks

The benefits of using hooks for describing the intended usage of frameworks can be summarized in the following points:

- A *better documentation* of the intended use of the framework is provided to the application developper.
- Hooks support the knowledge transfer from the framework developper to the application developper.
- Hooks make sure, that adaptions to and of the framework fits into its overall architecture and behavior.

Using Hooks in maintenance

Outside of the application framwork context, hooks can be used to describe future changes to a system to be maintained.

They may be used as

- a description of *future maintenance requirements*
- a description or *specification of service channels*
- a better *documentation of adaptive software*

2.3.3 Subject Oriented Programming / Design

Subject Oriented Programming [HKOS96, HO93] is focusing on those issues of object-orientation, which are caused by the fact that object-oriented systems are decomposed by classes. The consequences for source code based software maintenance are the following: lack of traceability within the source code, the application of design patterns cause invasive changes in the classes structure, and there may be difficulties when multiple teams are working on the same class.

The *lack of traceability within the source code of object oriented systems* is due to the fact, that system decomposition is by object (or by class), but requirements are expressed by features. Although within analysis or design models, this traceability can be kept (e.g. relations between static object models and use cases in UML [RJB99]), this structural mismatch occurs within the source code and leads to *code scattering* and *code tangling*.

Code scattering refers to the situation where the source code for one feature spreads over the whole system or the whole class hierarchy. Hence it is implemented within the attributes and methods of different classes. E.g. in the team calender case study, the feature "Add a new Meeting and the assisting team-members" will be implemented in methods in the classes `Date`, `Meeting` and `TeamMember`.

Code tangling refers to the situation, when one method is providing code for different features.

The use of design patterns [GHJV95] improves the structure of a software system, the reuse of components and the extensibility of the system. But on the other hand, if classes are playing different roles in different patterns and if patterns include multiple classes, this will lead to the fact, that the code of the pattern will spread over the classes they are applied to and that the code for patterns tangles with other code. There is no possibility to encapsulate the code for design patterns.

From the classes view, the class contains not only the code for different features, but also the (partial) code for the design patterns they are involved in.

Another aspect is, that the object oriented paradigm provides no support for the work of multiple teams, which work indepentendly on central, shared classes.

Subject-Oriented Programming is focusing on these "problems" by decomposing a system by subjects, not by classes/objects and by providing rules for the composition of subjects.

A *subject* is

> a collection of classes, defining a particular view of a domain or a coherent set of functionality. A subject may be a class, a fragment of a class, a pattern, a feature or a subsystem.

It is the task of *subject composition*

> to integrate the classes from separate subjects, to reconcile the differences between different views and to combine the different functionality.

Details of the composition [OKK+96] are described by *composition rules*. These composition rules define how components of the subjects are to be combined, e.g. how methods from different subjects for the same operation and class are to be combined, or whether nested compositions are allowed.

Decentralized development [OHBS94] is supported in two ways. *Decentralization in space* is possible by having multiple teams working on their subjects independently and by composing the subjects after completion of their development. *Decentralization in time* is also possible by developing subject after subject and by composing them in the end. If each feature in the requirements is implemented as one subject, this leads to *feature based development*.

This also reduces the code tangling and code scattering phenomenon as the code for each feature is implemented in one subject and composition is done by the development environment or by a special tool, not by the developer. The developer just has to specify the composition rules. Design patterns can also be encapsuled within their own subjects.

More recent work on subject-oriented software development is trying to elevate the approach to the design level. In [CHOT99, Vli98] these issues are discussed in more detail.

Raising subjects to the design level also enables better traceability. As requirements are expressed as a collection of features, and each feature can be represented as a subject in the design model and also be implemented as a subject, there is no structural mismatch to be considered when traceability links are established.

Subject-Orientation and Maintenance

Subject-orientation influences software maintenance as code tangling and code scattering is reduced and that due to the subjects nature traceability is facilitated.

Reducing code tangling and scattering eases the identification of change impacts as the source code for one feature (and so for a possible change) does not spread over the whole system. It is localized within the subject, hence change impact identification can be focused to the subject.

As mentioned above, if the possibilities for traceability provided by subjects are used, it is easier to follow a change request form the affected requirements feature via design subjects to code subjects.

Subjects also facilitate the extension of the system. The maintenance team develops the new (desired) feature as a new subject and composes it into the existing system.

Due to the decentralized nature of subjects, the maintenance team can be regarded as an additional development team working on existing subjects and developing new ones.

On the other hand, subject-orientation does not provide any assistance in change impact identification (besides delimiting the scope to the subject) and analysis. This means, tracing a change request from the requirements to code and within the code is still the task of the maintainer. There is also no support in assuring that a performed change does not introduce any unwanted side effects.

2.4 Model Based Maintenance

Model based maintenance approaches aim at elevating maintenance activities from the source code to the systems design or specification, or more general to a model of the system on a higher level of abstraction than the source code. They usually establish a more rigid maintenance process by enforcing the systems model to be updated first before updating the source code.

They also provide support in propagating the maintenance changes performed on the systems model to the source code.

- Two approaches using (formal) specifications of the system and an appropriate model to perform maintenance activities are *specification based maintenance* [LH91] and *Object Model evolution* [MPRR98]. Both enforce to change the systems specification first. While the first is reusing information on the refinement steps of the initial system to refine the new (updated) specification, the latter one is relating the changes on the specification to according changes on the source code by using service channels. An advantage of formal specifications is also, that tools like special theorem provers can be used to perform ripple effect analysis.
- Design maintenance systems [BP97, Bax92] are using design models of the system to perform maintenance activities first. When initialy developping the system, the so called design history is recorded. It contains all the tranformation steps departing from the systems design to the source code. Maintaining the system means maintaining the design history.

2.5 Summary of related work

To summarize the approaches mentioned within this chapter, brief descriptions of the approaches are given.

As sketched in the previous sections, the approaches can be summarized in the following groups:

- approaches providing additional documentation of the system;
- approaches facilitating software maintenance by structural mechanisms, but providing no active maintenance support;
- approaches providing active maintenance support.

Reverse engineering methods and techniques would fall into the second and third group, but are not covered here.

2.5.1 Additional System Documentation

Approach	Brief Description	Operational Support	Reference
Hooks into frameworks	Documentation of anticipated future changes within application frameworks	none	[FHLS97]
Non functional requirements	Modeling of future maintenance requirements as non functional requirements	none	[FB97]

2.5.2 Approaches facilitating software maintenance

Approach	Brief Description	Operational Support	Reference
Usage of Design Patters	Patterns like the Visitor Pattern provide design guidelines for more adaptive software.	none	[GHJV95, PJ98]
Adaptive Programming	Class dictionary graphs (CDG) to describe class structure, propagation patterns to describe the systems behavior. Source code is derived from the CDG and the propagation patterns	If CDG evolves, less effort needed in evolving propagation patterns.	[Lie95]
Aspect-Oriented Programming	Separation of concerns (aspects). The different aspects are woven to the resulting system.	By possibly defining maintenance aspects	[Kic96, KLM+97]
Subject-Oriented Programming	Decentralization of class definitions, automatic composition of several subjects	Maintenance changes could be made by producing a variant of the affected classes (= a new subject) and then be composed into the exesiting system.	[HKOS96, HO93]
Design Maintenance Systems	Evolution of systems design	none	[Bax92, BP97]

2.5.3 Active maintenance support

Approach	Brief Description	Operational Support	Reference
Specification-Based Maintenance	Evolution of systems specification	Semi-automatic refinement of changed specification	[LH91]
Service Channels	Evolution of systems specification	semi-automatic co-evolution of specification and implementation	[MPRR98, PMRR98]
Reuse Contracts	Implicit dependencies between parts of the system are made explicit.	Eases change impact identification	[LSM97]
EVA	Formal programming technique based on specification changes and program derivation.	If maintenance changes are expressed in specification changes, a new version of the program can be derived.	[MKH97]

3

Object Model Evolution and Service Channels

While chapter 2 introduced different maintenance contexts depending on the information available when performing maintenance changes to a system, this chapter presents different maintenance models from the methodological perspective.

3.1 Maintenance Process Models

Maintenance (process) models describe - like their counterparts in software development - the phases of software maintenance. One such model is the *staged software life cycle model for software maintenance* [BR00]. It describes the software life cycle after initial development.

In [Bas90] one can find three basic, but different models of software maintenance, the *quick fix model*, the *iterative enhancement model* and the *full reuse model*. These three basic models are compared to the *specification based maintenance model*.

3.1.1 A Software Life Cycle Model for Maintenance

In [BR00] a staged model for the software life cycle is presented (cf. figure 3.1). The focus of the model is to describe the phases a system runs through after its initial development. It represents the systems lifecycle as a sequence of stages, separating maintenance and evolution into different stages.

The first stage of the model is the *Initial Development* stage. This stage represents the development of the first version of the system following some development methodology. This first, running version of the system is the starting point for the maintenance life cycle model.

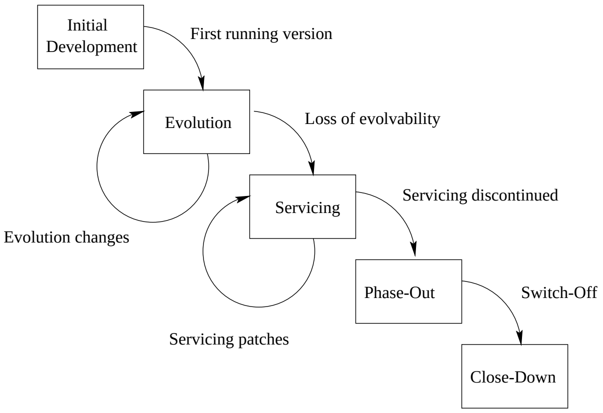

Figure 3.1: Software Maintenance Life Cycle Model

The maintenance phase of other process models (cf. chapter 5) is divided into three stages: *Evolution*, *Servicing* and *Phase-Out*.

In the *evolution stage*, the system is running and shows *no structural deterioration*. In the words of the authors of [BR00], the system shows *architectural integrity*; following the terminology of [TN97] the system is in *internal equilibrium*.

A second prerequisite for evolution is the availability of the (development) team knowledge. This knowledge about the systems architecture and internal invariants can assure, that changes made to the system will not lead to structural deterioration.

The system stays in the evolution stage, as long as evolution changes are not destroying the architectural integrity and as long as the team knowledge is available and updated.

If one of the two prerequisites for evolution is lost, the system moves to the *servicing stage*. Loss of knowledge about the system leads to a faster structural deterioration, which again requires more knowledge to perform a change.

The model does not foresee a reverse step back from the servicing to the evolution stage. The authors of [BR00] are sceptical, that for a system in the servicing stage it is possible to re-establish the architectural integrity and the lost knowledge about the system.

In the *phase-out stage*, the system is still in use, but no more servicing is done. During the *shut-down stage* the system is disconnected from its users.

In the setting of this model, models like the quick-fix model are models for performing maintenance activities for a system in the servicing stage. Models like specification based maintenance, iterative enhancement or the full reuse model may also be applied for a system in the evolution stage. These models are presented in the following sections.

3.1.2 Quick Fix Model

The Quick Fix Model (cf. figure 3.2) can be regarded as the classical maintenance model.

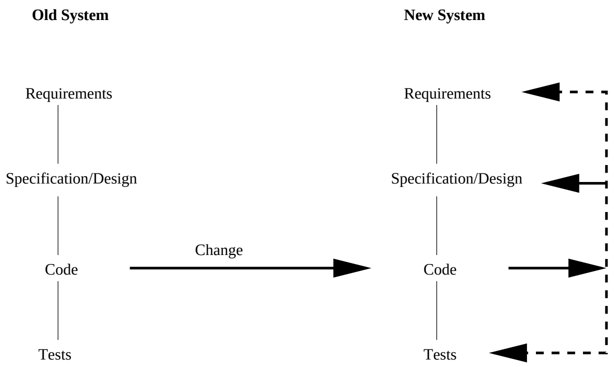

Figure 3.2: Quick Fix Model

The basic idea behind this model is to change the old system to produce a new one reflecting the maintenance requirements. Usually the source code is changed, then compiled and tested until the desired new functionality has been reached.

Afterwards the design documents, the systems specification and requirements documents are updated. The dotted arrows in figure 3.2 indicate, that these updates are not always done. This is due to the fact, that these documents are either not available, that the maintenance process is not that rigorous to enforce these necessary updates, or that lack of time and budget forces the maintenance personal to postpone or even cancel these necessary updates to upstream documents.

The consequence is structural deterioration (as pointed out in chapter 1) and increasing reverse engineering effort, needed to perform further changes to the source code. This is also described by Lehmans *law of increasing* *complexity*, the second law of Lehmans laws of progam evolution [Leh80], and by Blum in [Blu95] describing the *software maintenance paradox*.

Hence, the quick-fix model describes the way, maintenance activities are performed for a system in *servicing stage* in Bennett and Rajlichs staged software life cycle model for maintenance (cf. section 3.1.1).

3.1.3 Iterative Enhancement

The Iterative Enhancement Model (cf. figure 3.3) is very close to prototyping or evolutionary development approaches [Fug00, HSE90].

It starts with an analysis of the old systems usage which leads to a new set of requirements. With these new (changed) requirements the whole development life cycle is repeated again to produce a new system reflecting the new (changed) requirements.

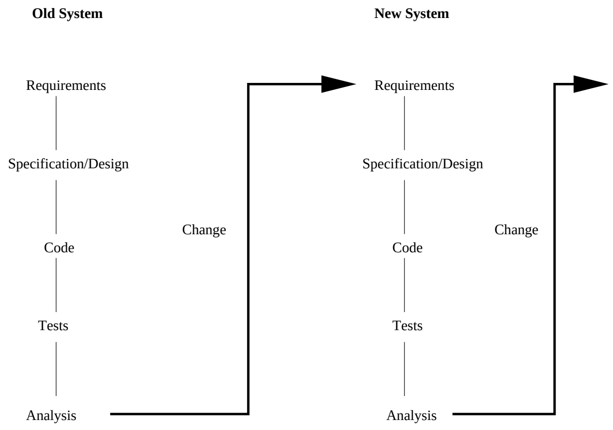

Figure 3.3: Iterative Enhancement Model

The iterative enhancement model wipes out the main drawback of the quick fix model - it avoids structural deterioration. It does so by repeating the whole system development life cycle when developing a new system based on the changed requirements of the old one. Hence, new specifications, design documents and additional documentation are produced.

If the newly developed system shows architectural integrity, the iterative enhancement model is suitable for systems in the evolution stage.

On the other hand, this model is not suitable for "small" changes in the systems requirements as it would be too expensive to build a completely new system if only small modifications to the old system are necessary.

3.1.4 Full Reuse Model

The Full Reuse Model (cf. figure 3.4) for software maintenance is based on the idea of reusing parts of the old system and adding new functionality to it. The focus is again not on changing the old system. The new system is developed according to a new (changed) set of requirements reusing components from the old system.

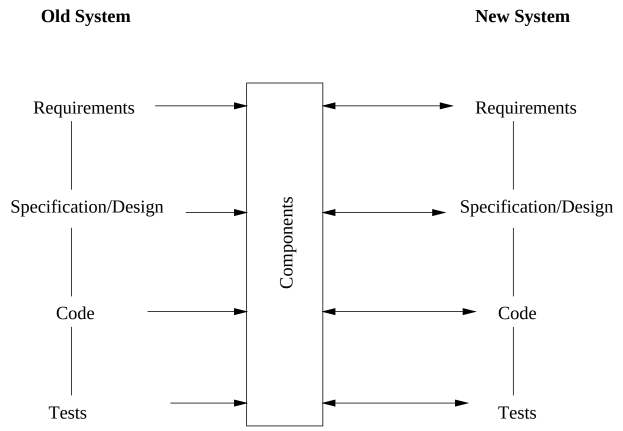

Figure 3.4: Full Reuse Model

Compared to the iterative enhancement model, the full reuse model has the advantage, that only new (added) components have to be developed and that existing reusable components are reused as such. But additional effort has to be spent in deciding which components are to be reused or not. Another prerequisite is, that the applied development process has to be "reuse oriented".

Like the iterative enhancement model, the full reuse model is also suitable for changing systems in the evolution stage.

Another important issue is, how to treat components identified to be reusable with modifications. Here the maintenance personal has to decide, either to redevelop this component (not to reuse it) or to modify it. But if this modification is done by applying a quick fix approach, the benefits of applying the full reuse model are lost.

3.1.5 Specification Based Maintenance

As outlined in the previous sections, the way to obtain a co-evolution of the systems source code and related design documents and specifications is to initiate a new development of a new system reflecting the changes in the requirements.

Specification Based Maintenance approaches (or more general Model Based Maintenance, cf. figure 3.5) on the other hand focus on changing the systems specification or design and then deriving the source code for the changed specification. So it can be compared to the quick fix model, but on a higher level of abstraction.

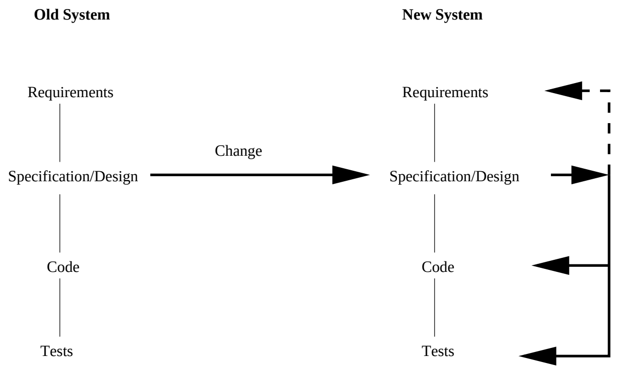

Figure 3.5: Specification Based Maintenance

One advantage is that structural deterioration is avoided without having to run through the whole development life cycle. Hence, the model is suitable for maintaining systems in the evolution stage.

Another issue is the very nature of formal specifications. When modifying or changing a specification, tools like theorem provers can be used to show whether there are side effects or ripples of the performed change. Refining a specification to source code is a formal, structured, reproducible way to derive source code. Hence after having changed the specification, the recorded refinement history can be reused to refine the new specification.

For example, approaches like the one described in [LH91] are using

- *invariants* within the specification to check the specifications consistency and to detect ripple effects after performing a change on the specification
- *information about the refinement steps* recorded when developing the source code from the original specification to develop the new source code out of

the changed specification

- *additional documentation* representing relationships between requirements and specification.

3.2 Object Model Evolution

This section introduces *object model evolution* [MPRR98]. First the context is set up and some methodological issues are shown. Service Channels are presented as an instrumentation of object model evolution.

The context of object model evolution is the setting of model based maintenance (cf. chapter 2). We depart from the assumption, that in addition to the source code to be maintained also models of the system on different levels of abstraction exist. These models can be analysis models, design models or formal specifications. We refer to these models on the model level as the *object model (OM)*, to the source code on the implementation level as the *object implementation (OI)*.

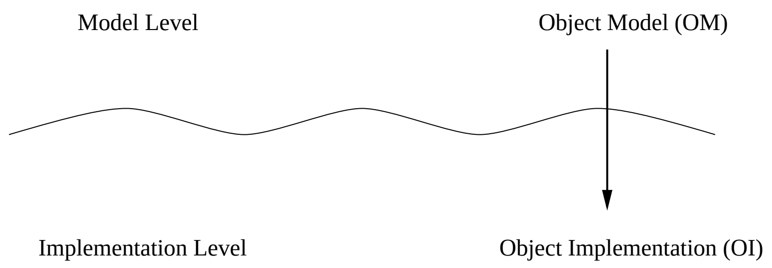

Figure 3.6: Context for Object Model Evolution

Figure 3.6 summarizes the context for object model evolution. We are assuming, that the implementation was built according to the existing model of the system, hence the implementation is consistent with the existing model. This is indicated by the arrow between the object model and the object implementation. The wavy line indicates, that there may be some levels of abstraction between the object model and the object implementation.

Figure 3.7 sketches classical software maintenance, if it is performed in a source code based maintenance context, following the quick fix model for maintenance.

The initial system to be maintained is described by *OI* and *OM*. Maintenance activities (changes) are done on the implementation yielding *OI'*. To do so,

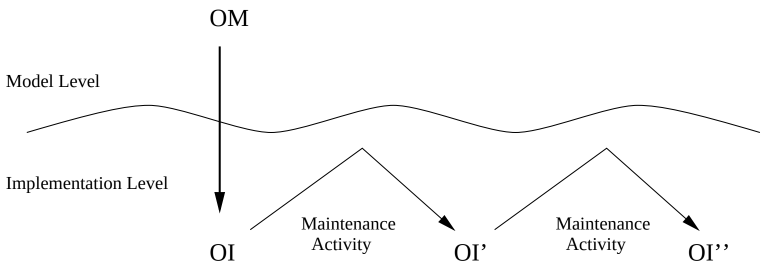

Figure 3.7: Classical Source Code Based Maintenance

reverse engineering tools and methods are applied to extract information on a higher level of abstraction out of the source code. The initial object models are rarely consulted (cf. chapter 1).

We refer to this evolution of the source code (implementation, *OI*) as *object evolution*.

Further maintenance requests produce *OI"* out of *OI'* without updating the object model. Hence *structural deterioration* and increasing alienation between *OM* and *OI"* may be observed. For each further maintenance activity, more reverse engineering effort is needed to understand actual source code. Hence maintenance activities become more and more costly.

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, we propose an approach for software maintenance on the model level as illustrated by the solid line in Figure 3.8.

Here *maintenance activities* are done on the *object model OM*. By maintenance activity steps we refer to changes such as described in [MK93, KGH+94, KGH+96].

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*.

As figure 3.8 shows, the main goal of the approach is to obtain a *co-evolution of the object model and the object implementation*.

To reach this goal, maintenance activities are to be supported by relating the specification (model level) to the respective representation on the implementation level. Hence, additional documentation of the system is needed within

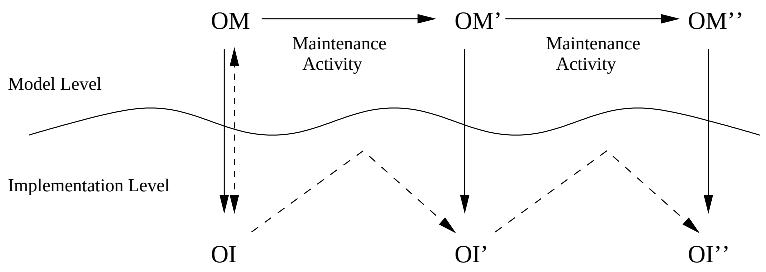

Figure 3.8: Object Model Evolution

the model and the implementation level and between the different levels. The first kind of documentation is necessary to ease change impact analysis, the second one to relate changes on the model level to corresponding changes on the implementation.

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.
- Model evolution provides *guidance* concerning the sequence of maintenance steps necessary to consistently build OM' out of OM.
  By maintenance steps we refer to changes to the object model. For those changes, the preconditions, their wanted effects and side effects are known (their postconditions). A fixed set of such changes can be the toolbox for defining a sequence of changes necessary to build a new object model reflecting new requirments. Stating the changed postcondition for the new, desired object model OM', the new model can be built by trying to derive the new postcondition out of the postcondition of the old object model OM.
  An example for such a fixed set of changes is the classification of changes described in [KGH$^+$96].
  Taking the hierarchical structure of Kungs classification of changes [KGH$^+$96], 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.
- The additional documentation *makes explicit hidden 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.

Object model evolution focuses on two aspects of software maintenance. It enforces a rigid maintenance process (performing model evolution before object evolution) and requires system documentation (on various levels of abstraction).

As the name also indicates, object model evolution is suitable for maintaining systems in the evolution stage (cf. section 3.1.1), as co-evolution of specification and implementation assures architectural integrity.

3.3 Service channels to support model evolution

In the previous section, object model evolution was outlined. The goal of object model evolution is to obtain a co-evolution of the systems implementation and the according models. It was also pointed out, that additional documentation is needed.

Service channels are now introduced, to support object model evolution in a planned and semi-automatic way.

Refering back, figure 3.8 could be interpreted just as a methodological advice. As such, it might already help in many situations and be in line with what is currently seen as “best practice”.

With (ad hoc) 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 mechanisms on which the above mentioned machine support can be based upon.

3.3.1 Basic Idea

The basic idea of service channels is to provide support for object model evolution for an anticipated class of changes following the object model evolution approach.

Service Channels represent maintenance functionality within the system or within a maintenance environment. They are responsible for the localization of changes and change impact analysis on the model level, for the evolution of the object model, for localizing the performed change on the implementation level and possibly propagating the change to the implementation and the provision of test cases.

A *service channel*

> is a mechanism relating a sequence of transformations on the model level to the code level for one anticipated class of changes.

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 [SC96] for checking OI' against the changes in the specification. The specific benefits from focused testing in class structures can be seen from [HMF92].

A third role service channels may play is the role of a *diagnostic service channel*. Diagnostic service channels are adaptive or verificative service channels failing to perform their full mission (e.g. an adaptive service channel, which can not perform a safe transformation). In this case, service channels (regardless of their type) should clearly point out, why they failed (e.g. which invariants are violated).

Diagnostic service channels are regarded as a mode of operation, which both other types of service channel incorporate.

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 tools 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.

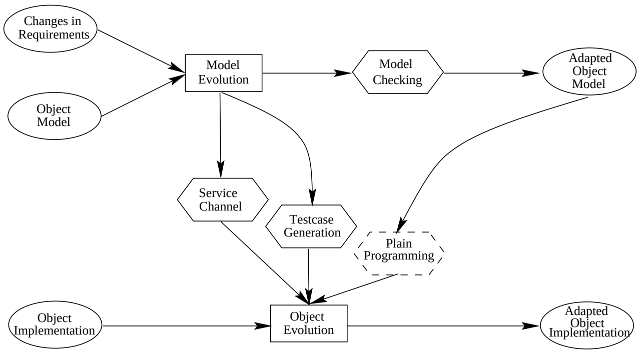

Figure 3.9: Maintenance using model evolution

Figure 3.9 summarizes the idea of object evolution by means of 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.

3.3.2 Maintenance Support by Service Channels

To position the functionality of service channels with respect to other research supporting software evolution, we relate them to the *evolution support space* [PK98] defined by Balzer in summarizing the results of the fist International Workshop on the Principles of Software Evolution (IWPSE'98) [Bal98].

The evolution support space spans two dimensions: the type of support and the scope of support.

The *type of support* distinguishes whether support is provided by *transforming* the system (performing an evolutionary change automatically), by *guiding* (sketching the necessary actions to perform the change by hand) or by *checking* the impact of a requested change.

The *scope of support* can be *local* (e.g. within an object), *distributed* or *compound*. For our considerations we consider two kinds of changes: *Intra-Object changes* (local scope) and to *Inter-Object changes* (compound scope).

Service channels provide maintenance support in two ways:

- As *verificative service channels* they provide guided maintenance by easing ripple effect analysis, change propagation and by generating test data for the respective change. Hence they support evolution by *guiding* and *checking*.
- As *adaptive service channels* they support evolution by *transforming* the system according the requested evolutionary change. They can do so for the class of changes that causes no ripple effects to other classes of the system to be maintained. They should propagate the change automatically from the specification to the implementation and update the relationships between specification and implementation.
- As *diagnostic service channels* they point out the reason, why the service channel (either an adaptive one or a verificative one) failed. Hence they support evolution by *guiding* and *checking*.

When determining the scope of evolution support provided by service channels, the class of changes they are built for has to be investigated.

- As one can imagine, the class of changes *adaptive service channel*s can be provided for is limited to those change categories anticipated during development. They can be captured as change cases [EDF96]. Adaptive service channels have to consider all the relevant relationships and invariants concerned by the respective change. The changes will be *intra-object changes* (changes on the objects structure causing no changes on the objects interface), hence the scope of support is *local*.
  According to Kungs classification [KGH$^+$94] these changes will be data changes, method components changes, method control structure changes and class component changes.
- *Verificative service channels* will cover a broader range of changes than adaptive service channels. They should be able to provide support to *inter-object changes*. These are changes on the objects interface and/or the objects relations (inheritance, aggregation) to other objects. Hence the scope of such service channels is *compound*. They can even perform local changes by invoking adaptive service channels.
  A verificative service channel identifies the directly related objects inflicted by a change request. Far-reaching changes are propagated further by the verificative service channels of the objects thus identified.

To summarize, table 3.1 places service channels within the evolution support space.

		Type of support		
		Trans-formation	Guidance	Checking
Scope of support	Intra-object changes	*Adaptive Service Channels*	*Verificative Service Channels*	*Verificative Service Channels*
	Inter-object changes		*Verificative Service Channels*	*Verificative Service Channels*

Table 3.1: Service Channels and the Evolution Support Space

3.4 Some Examples

The following examples are taken from the case studies which can be found in the appendix. The first one shows the basic types of maintenance support service channels provide within the thermometer case study. The second one scales these considerations up to a larger example like the team calendar case study.

3.4.1 Thermometer: Changing the temperature range

The example of the thermometer case study (cf. appendix A) consists of two classes, *Thermometer* (representing the front end for reading temperature values from a physical thermometer) and *TempDisplay* (for displaying the temperature measured by the thermometer on a two-digit display). The corresponding C++ - like implementation and the implementation rationale can also be found in appendix A.

For this example, one possible maintenance request could be the following:

The system, as it is actually implemented, is able to measure and display temperatures within a range between -20 and 40 centigrades (refered to as range(x,y), x = -20, y = 40)

This temperature range should be extended to cover the following temperatures:

$$A : x = -45;\ y = 45$$

$$B : x = -80;\ y = 80$$

$$C : x = -112;\ y = 176(degreesFahrenheit^1;\ equivalentto - 80to80centigrades).$$

Source code based maintenance

The following steps will be done when following a source code based maintenance approach.

Having the object model and the implementation rationale in mind, one will easily find, that the only place in the implementation, where this temperature range is defined and checked are the *Thermometer:GetTemp()* and the *Thermometer::Thermometer()* methods.

```
Thermometer::GetTemp() {
int newtemp;
...
if (range_lb <= newtemp <= range_ub)
   temp = newtemp
}

Thermometer::Thermometer() {
temp = 0;
range_lb = -20;
range_ub = 40
}
```

Hence, it is the initialization method *Thermometer::Thermometer()*, which has to be changed to define the new temperature range. The rest of the implementation has not to be changed. The new initialization method is now the following:

Changing the method as requested in case A, yields the following new initialization method:

```
Thermometer::ThermometerA() {
```

```
temp = 0;
range_lb = -45;
range_ub = 45
}
```

Compiling and testing the changed code will show, that this change introduces no side effects. The new code is still meeting its specification. The invariants within the two classes are not violated.

Changing the method as requested in case B, yields the following new initialization method:

```
Thermometer::ThermometerB() {
temp = 0;
range_lb = -80;
range_ub = 80
}
```

Compiling and testing will again show, that this change is safe and the system works as expected.

But in this case, when reviewing the specification, one can observe, that the code does not meet its specification. The new method *Thermometer::ThermometerV2()* is violating the invariant

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

within the *Thermometer* class, which restricts the temperature measured by the physical thermometer to be between -50 and 50 centigrades.

In this case, even if the system works, one can observe structural deterioration. The new system in use, measuring temperatures between -80 and 80 centigrades, drifts away from its specification. If mentioned within section 1.3, the systems specification or other documents are not updated, this structural deterioration will not be detected.

Changing the method as requested in case C, that the system is able to display degrees Fahrenheit as well, yields the following new initialization method:

```
Thermometer::ThermometerC() {
temp = 0;
range_lb = -112;
range_ub = 172
}
```

With naive trials, this change might even seem to work.

Methodological testing should show, that the temperature display is not able to display the three digits of the measured temperature. Hence in the cases, where temperatures above 99 degrees Fahrenheit and below -99 degrees Fahrenheit will be measured, the display won't be able to display the third digit necessary.

If testing is not done effectively and this error will not be detected, a system not meeting its expected functionality will be in use.

Here, not only local invariants within the *Thermometer* class are violated, also invariants within the *TempDisplay* class are violated. But the violations of these invariants are hard to detect within the source code, as they are not explicitly represented there (cf. section A.1.2).

Maintenance Support by Service Channels

The problems arising when performing the above changes with a source code based approach, originate from the fact, that invariants and dependencies expressed within the specification are not expressed within the source code.

When providing a service channel for the above change, not only the necessary changes on the specification and the source code are expressed in the service channel. The service channel also contains information expressed in the specification which is not included or expressed in the implementation. Additionally, the service channel knows the information expressed in the implementation rationale (cf. A.1.2).

Hence the service channel has to know the chain of implication, relating the invariants of the specification and it has to know, that the temperature range expressed in the specification is defined within the initialization method of the implemented *Thermometer* class.

As service channels implement a specification based maintenance approach, service channels first perform the change on the specification and check, whether the change is safe or not. They do so by proving that none of the invariants of the affected classes and that no maintenance invariants are violated.

The chain of implications (cf. A.1.2)

$$(D \Rightarrow C) \wedge (C \Rightarrow A) \wedge (A \Rightarrow B)$$

acts as such a maintenance invariant for the service channel responsible for changing the temperature range. As long as for the changed specification this chain of implications holds, the change is safe and can be propagated to the source code.

Hence, for case A the service channel provided for changing temperature ranges will act as an adaptive service channel, for case B and C it will act as a verificative service channel, pointing out the constraints violated by the change request.

3.4.2 Team Calendar: Changing the number of participants of a meeting

Within the team calendar case study, one possible change would be to change the number of participants allowed for a meeting. Before investigating the change, a possible implementation rationale is sketched. The complete specification of the team calendar can be found in appendix B.

Implementation Rationale for Meeting

Information about the number of participants allowed to attend a meeting can be found within the *Meeting* and *Rooms* classes.

Let's consider the invariants of the class *Meeting*:

$$A : maxparticipants = 10$$

$$B : \#participants \leq maxparticipants$$

$$C : \#participants \leq location.capacity$$

A is taken from the constant definition within the class *Meeting*. B and C are predicates from the state schema of the class *Meeting*. B and C also appear as predicates in Meetings method *InputMeeting*.

The invariant A defines the maximum number of participants to attend a meeting. B checks the actual number of participants, while C is assuring, that the actual number of participants is below the maximum capacity of the meeting room, where the meeting takes place.

The invariants in class *Rooms* define the maximum capacity of the rooms. In our case study, the largest room has the maximum capacity of 20 persons.

$$D : maxcapacity = 20$$

$$E : \forall\, r : existingrooms \bullet r.capacity \leq maxcapacity$$

D is taken from the constant definition within the class *Rooms*. E is the predicate within the state schema of class *Rooms*.

The invariant D defines the maximum capacity and E assures, that the capacity of all meeting rooms is below this maximum number.

Hence, in our case, as the number of participants allowed for a meeting is below the maximum capacity of all available rooms (maxcapacity > maxparticipants),

$$A \wedge B \Rightarrow C \wedge (D \wedge E)$$

holds. An efficient implementation would only check for B within *InputMeeting*. Within the implementation, the constants and invariants defining and checking the capacity of rooms will not appear.

One possible maintenance request could be the following:

Due to the increasing number of customers and employees, it is necessary to have meetings with up to the following numbers of participants. The new upper limits for the number of participants are the following:

$$A : 15 participants$$

$$B : 25 participants$$

Maintenance Support by Service Channels

A service channel for the above change, will focus on the definition of the maximum number of participants within the *Meeting* class.

$$A' : maxparticipants = 15$$

$$A'' : maxparticipants = 25$$

The service channel performs the changes on the model level, analyzing whether $A \wedge B \Rightarrow C \wedge (D \wedge E)$ holds after the change on the model level.

- $A' \wedge B \Rightarrow C \wedge (D \wedge E)$ holds.
  In this case, the service channel performs the change automatically on the implementation and works as an *adaptive service channel*.
- $A'' \wedge B \Rightarrow C \wedge (D \wedge E)$ does not hold
  In this case, the service channel performs change on the model level, identifies the affected parts of the implementation and will point out, that the chain of implications is violated. It will act as a *verificative service channel*

3.5 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).

3.5.1 Built-in service channels

A *built-in service channel* (BI-SC) or *internal service channel*, as shown in Figure 3.10, 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 to 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 constraints 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. The system can be seen as a set of virtual system instances, with each instance specified by appropriately invoking the service channel. Conceptually, the invocation of a built-in service channel with a certain change request determines the suitable solution in the set of possible solutions.

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

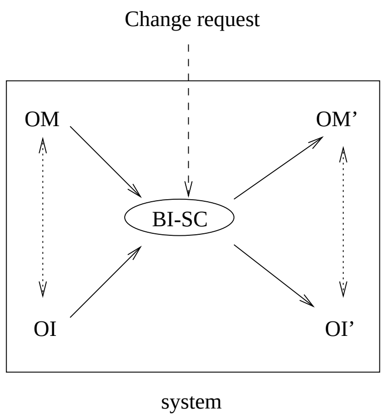

Figure 3.10: Built-in service channel

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. by 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.

Built-In Service Channels built into an object for specific anticipated changes to that object, will be *adaptive service channels* by nature. Their scope is local and if the requested change is admissible, they transform the object they are built into to a new instance of its relatively generic specification.

They will act as a *diagnostic service channel* in the case, the requested change can not be performed by the internal service channel. In this case, they will point out the side effects caused by the requested change, which cause failing of the service channel.

3.5.2 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, the objects 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 3.11. 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.

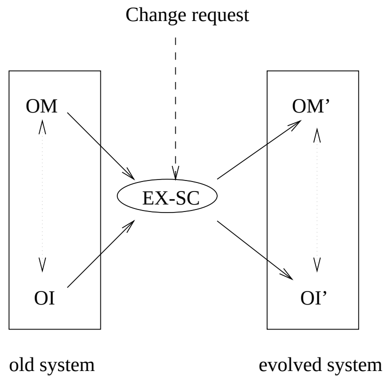

Figure 3.11: External service channel

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. [LH96]), ER- or structure charts generators [Vrb97] 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 pre-established 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.

External service channels will be *verificative* and *diagnostic* ones. They support inter-object changes by guiding and checking.

3.6 Related Work

Alternative approaches:

- Design Maintenance Systems [BP97, Bax92]
  Here, the systems design is seen as the central focus for development and maintenance. The according source code is seen as a byproduct. A transformational scheme is used for getting the source code.
- Configuration Management [CW98, CM94]
  Software Configuration Management (SCM) approaches incorporate version models to capture deltas between versions of (components of) the system. Mechanisms for compiling the deltas in the new build are provided. Usually SCM is focussed on source code, hence they offer poor support for model based maintenance.
- Specification based maintenance [LH91]
  This approach uses the object-oriented specification language Z++ to describe a formal specification, and a development record to store relationships between the specification and the source code. Guidance for performing changes is provided.
- Reuse-Contracts [LSM97]
  Here, different parts of the system (and their relationships) are described by relational operators. Reuse contracts and the according operators facilitate maintenance by pointing out how much work is needed to update the system, by pointing out what problems may occure and which parts of the system have to be tested.

- EVA Method [MKH97] EVA (EVolution mechanism for flexible Agent) is a formal programming technique base on specification changes and on derivation of the program code. If maintenance changes are expressed in the specification changes, a new version of the program can be derived.

The above listed approaches offer either *support and guidance* for maintenance (like reuse contracts or SCM) or they are based on *deriving or transforming* the design or specification into source code (hence each time, maintenance activities are performed, a new source code for the whole system is created).

Service channels offer support for maintaining a systems specification and implementation without having to newly recreate the whole system.

4

Documentation of Relationships in the System

An important prerequisite for effective software maintenance activities is to have an in depth-understanding of the system. This chapter outlines the information necessary to gain such an understanding. Intra-Object Schemas are introduced to store such information in our specification based maintenance setting using service channels.

4.1 Required Traceability

To gain an understanding of the software system to be maintained, not only source code and documentation is needed. The maintainer also needs abstract representations of the system like specifications or design models and he/she needs to be able to establish or to identify *dependencies* between various artifacts of the different system models. He/she should also be able to trace these dependencies in order to assess the impact of a change. Hence a certain degree of *traceability* is required.

Following the IEEE definition [IEE91], *traceability* is

> a degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor/successor or master/subordinate relationship to each other.

Traceability should provide the maintainer with semantic links between various artifacts of the system he/she can use to perform change impact analysis.

Such dependencies should be established between artifacts belonging to parts of the system at the same level of abstraction. Dependencies among source code elements, like chains of method calls would fall into this category of dependencies. But also dependencies between artifacts belonging to different levels of abstraction should be traceable, like an "implemented_by" relationship between elements of the design model and their corresponding counterparts in the source code.

By *dependencies (traces)* we refer to

> a direct relationship between two artifacts in the system $X \longrightarrow Y$ such that a (maintenance) programmer modifying X must be concerned about possible side effects in Y. [WH91, WH92]

Representing both kinds of dependencies graphically, yields the *tracing graph* as outlined in [Boh91] and sketched in figure 4.1. It is a directed graph, having named nodes representing different artifacts of the system.

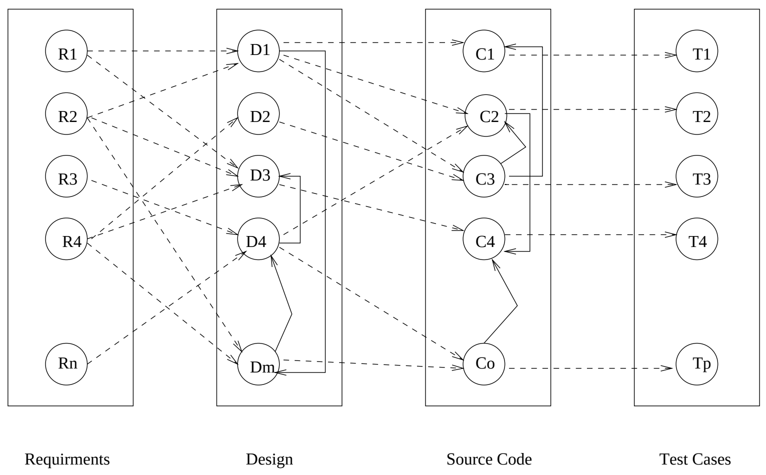

Figure 4.1: Tracing Graph for Software Maintenance

The dotted arrows represent dependencies like "is implemented by" or "is covered by". For example, requirement *R1* is covered by *D1* and *D3* in the design model. We refer to traces of that kind as *inter-level traces*, as they span among different levels of abstraction.

The solid arrows between artifacts belonging to the same level of abstraction denote dependencies among artifacts of that level. In our example of figure 4.1 the design elements *D4* and *D3* are related in some kind. We refer to traces of that kind as *intra-level traces*.

Such a graph can be the base for performing *change impact analysis*. Having determined the requirements statements affected by a change request, a "projection"[1] on the graph yields the affected parts of the system on the other levels of abstraction.

Figure 4.2 outlines the effect of changing the requirement statement *R3*. The dark arrows span a subgraph of the tracing graph, following all the dependencies starting at *R3*. The nodes within this subgraph represent those artifacts of the system affected by changing *R3*.

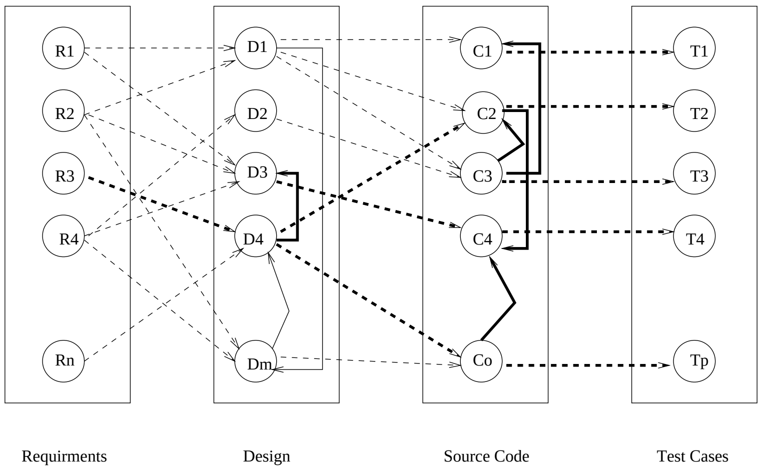

Figure 4.2: Change Impact Analysis using the Tracing Graph

Additionally, this graph can be used to evaluate the quality of the systems design or maintainability.

Regarding the intra-level traces for one, say, design element, the number of traces directed into that element indicates the quality of the design. Keeping this number low indicates, that the number of other elements directly affecting this element is low. Hence this number (the in-degree) can be seen like a coupling measure similar to the one known from structured design [SMC74].

In the following, intra-level traces and inter-level traces are described in more detail.

[1]The term projection is used by analogy to projection in the field of relational database theory.

4.1.1 Intra-Level Traces

Intra-level traces describe dependencies between artifacts in the system, belonging to the same level of abstraction, like dependencies between entities of the source code or entities of the specification. As the focus of this work is on specification based maintenance, this section sketches classes of (intra-level) dependencies on the code level and on the specification level which are of interest for maintaining an object-oriented system.

Code-level Entities and Dependencies

In order to define classes of dependencies between entities on the source code level, one has to define the *classes of entities* to focus on. Within the field of program comprehension and program understanding, a lot of work has been done to identify such classes of entities and dependencies necessary to understand a certain piece of code.

In the setting of (source code based) software maintenance, it is necessary to know the structure of the source code, to be able to analyze the effects of a change to a piece of code.

In [LMR91, LMR92] the authors describe a system, where semantic relations within the source code are stored in a relational database. These semantic relations are language independent and also cover object oriented language features and relations (overloading, etc.).

Here, the entity classes and dependency classes described in [WCMH91, WH91, WH92] are presented. They also cover object oriented language features.

The limits of both approaches are the issues introduced by polymorphism, overloading and "high level constructs". To fully identify these relations, dynamic analysis of the source code is necessary.

Table 4.1 summarizes the classes of code entities [WCMH91]. Here, alongside with entities describing structured, procedural programming language features, object-oriented language features are considered as follows: classes are seen as composite types, methods as special subprograms and messages as special names.

Table 4.2 and table 4.3 show the dependency classes between the entities defined previously.

These dependencies can be gathered by static analysis tools and can be used to identify possibly effected entities in the code when modifying another one.

Entity Class	**Description**
data (D)	Any entity holding data values
- constant (CO)	
- variable (V)	
– simple variable (SV)	eg. an integer or float, a component of a struct or class
– composite variable (CV)	an entity, containing other data; eg. a struct or a class
– pointer variable (PV)	points to a simple or composite variable
subprogram (S)	eg. a procedure, function, subroutine, etc.
- function (FS)	eg. a C language function
- method (ME)	eg. a C++ member function or a Smalltalk method
type (T)	describes the structure or behavior of data
- atomicType (AT)	eg. int, float, char, etc.
- compositeType (CT)	eg. struct, union, etc.
– class (CL)	C++ or Smalltalk class
name (N)	a name used within the software to refer to one or more program entities; also several names may refer to the same entity.
- message (MG)	eg. a Smalltalk message selector or the name of a C++ member function. Overloading allows the message to refer to one of several methods.

Table 4.1: Code level entity classes [WCMH91]

Dependency Class	Description
Data $\Longrightarrow$ Data	
isPartOf (V,CV)	V is directly a part of CV - not recursive
dataAffects (D1,D2)	the value of D1 affects the value in D2
- usedToCompute (D1,D2)	D1 is used in a computation of D2
– movedTo (D1,D2)	the value of D1 is moved into D2
— matchesInParam (D1,D2)	actual parameter D1 matches formal parameter D2 in a subprogram call
— matchesOutParam (D1,D2)	actual parameter D1 matches formal parameter D2 in a subprogram call; the parameter passing scheme must allow outward flow of data from D1 to D2; eg. call by reference
- sharesMemory (D1,D2)	D1 and D2 may share some memory
– contains (D1,D2)	D2 is contained within D1
– equivalencedTo (D1,D2)	D1 defined at D2, eg. matching union elements
– pointerAlias (D1,D2)	created by pointer assignment
- subscriptOf (D1,D2)	D1 is a subscript in n assignment to array D2
- controlsValueOf (D1,D2)	D1 controls a computation of the value of D2
Data $\Longrightarrow$ Subprogram	
- isParameterOf (V,S)	V is a parameter of subprogram S
- isLocalOf (V,S)	V is a local variable in S
- isImportedBy (V,S)	non-local V is referenced in S
Data $\Longrightarrow$ Type	
- determinesSizeOf (D,T)	data D determines size of T (an array)
- isOfType (D,T)	data V is of type of class T
Subprogram $\Longrightarrow$ Subprogram	
- calls (S1,S2)	subprogram S1 calls subprogram S2
Subprogram $\Longrightarrow$ Type	
- subprogramReturnsKindOf (S,T)	type of the return value of S
- methodImplementedIn (ME,CT)	method ME is implemented in composite type CT

Table 4.2: Code level dependency classes [WCMH91]

Dependency Class	Description / Comments
Class $\Longrightarrow$ Class	
- isDirectSuperClass (CL1,CL2)	CL1 is a direct superclass of CL2
- isDirectSubClass (CL1, CL2)	CL1 is a direct subclass of CL2
- inheritsFrom (CL1,CL2)	CL1 inherits from CL2
- uses (CL1,CL2)	CL1 uses CL2; may be subclassed as "uses for interface", and "uses for implementation"
Class $\Longrightarrow$ Method	
- methodReturnsObjectOfClass (ME,CL)	method ME returns object of class CL
- implementsMethod (CL,ME)	class CL implements method ME
- inherits (CL,ME)	class CL inherits method ME
Class $\Longrightarrow$ Message	
- understands (CL,ME)	class CL understands method ME
Class $\Longrightarrow$ Variable	
- isInstanceOfClass (V,CL)	V is an instance of class CL
- isClassVariableOf (V,CL)	V is a class variable of class CL
- isInstanceVariableOf (V,CL)	V is an instance variable of class CL
- isDefinedBy (V,CL)	V is defined by class CL
Method $\Longrightarrow$ Variable	
- isParamForMethod (V,ME)	V is a parameter for method ME
- isLocalInMethod (V,ME)	V is a local variable in method ME
- isImportedByMethod (V,ME)	V is a non-local variable used in method ME
- isDefinedByMethod (V,ME)	V is defined by method ME
- refersTo (ME,V)	method ME refers to variable V
Method $\Longrightarrow$ Message	
- isNameOfMethod (MG,ME)	message MG is name of method ME
- sendsMessage (ME,MG)	method ME sends message MG
Method $\Longrightarrow$ Method	
-invokes (ME1,ME2)	method ME1 invokes method ME2
-overrides (ME1,ME2)	method ME1 overrides ME2

Table 4.3: Code level dependency classes for OOP [WH92]

Specification-level Entities and Dependencies

To establish intra-level dependencies on the specification level, one has also to define specification entities and dependencies among them. Taking model based specification languages like Object-Z [CDD+90, DKRS91, DRS94], the structure of the language is pretty similar to those of object-oriented programming languages.

An *Object-Z specification* consists of *class definitions*, related by inheritance and instantiation. A class consists of a *state schema* and *operations* (methods). *Objects* are instances of a class, which acts as a template for objects.

The syntax of a class is the following (described as an Object-Z class).

ClassName[*generic parameters*]

visibility list
inherited classes
type definitions
constant definitions
state schema
initial state schema
operation schemas

history invariant

The *visibility list* shows those operations and parts of the state schema, which are visible from outside the class. The *history invariant* is a predicate over histories of objects of the class and is typically expressed in temporal logic.

Hence, one can easily imagine, that specification level entities and dependencies can similarly be established as on the source code level.

4.1.2 Inter-Level Traces

Inter-level traces cover dependencies like "is implemented by", "is covered by" or design rationales. Entities to be related are those from the code and specification level, extended by entities for requirements and design rationales.

4.2 Classes of dependencies

The next question which comes up is how to identify those dependencies described above. How are those dependencies manifested within the system?

We found three possible ways, how dependencies may be manifested within a software system and grouped them into three different classes of dependencies. They may be directly stated in the system (explicite dependencies), not directly stated within the system (hidden dependencies) or they may be manifested as "additional systems documentation" within special maintenance tools like service channels (materialized dependencies).

4.2.1 Explicit Dependencies

Explicit dependencies are those, that are explicitly expressed or documented within the system. They are either expressed in the systems programming or specification language.

Most of the inter-level traces are explicit dependencies. For example, considering the history invariant $\Box(-50 \leq temp \leq 50)$[2] of the class *Thermometer*. Within the C++ class `Thermometer`, the two constants `range_lb` and `range_ub` contain the boundary values for `temp`. Hence, this fact has to be explicitly expressed as an inter-level trace.

4.2.2 Hidden Dependencies

Hidden dependencies are dependencies not explicitly expressed within the system. Such dependencies are provided by the developer or can be identified using tools like dependency analysis tools or static or dynamic analysis tools like slicers or theorem provers.

Hidden dependencies occur as intra-level traces and as inter-level traces as shown by the following two examples out of the thermometer case study (cf. appendix A).

First, a *hidden intra-level trace on the specification* is the following focusing on the *temp* variable local in the *ShowActTemp* method of class *TempDisplay*. It is used for calculating the two digits of the display and is restricted to $-99 \leq temp \leq 99$. Let's consider the following trace:

1 $temp = thermometer.YieldTemp$
2 *YieldTemp* reads the *temp* variable of *Thermometers* state scheme.

[2] $\Box$ denotes the temporal logic operator *always*.

3 *temp* within the *Thermometer* class is restricted by *Thermometers* history invariant $\Box(-50 \leq temp \leq 50)$.

Hence, the value of *TempDisplay.temp* is always between -50 and 50 centigrades. Following the above trace, one can identify the fact, that there is a dependency between *TempDisplay.temp* and the history invariant of the *Thermometer* class. Hence,

$$\Box(-50 \leq TempDisplay.temp \leq 50).$$

Second, a *hidden inter-level trace* can be the following: let's consider the local variable *temp* within the *ShowActTemp* method of the class *TempDisplay*.

On the specification, *temp* is restricted by $-99 \leq temp \leq 99$. Within the C++ class `TempDisplay::ShowActTemp()` the corresponding variable is also called `temp`, but the restriction expressed in the specification is not checked and expressed within the source code. Hence the fact, that the `temp` variable in the source code is also restricted to be between -99 and 99 centigrade is a hidden inter-level trace within the code.

How can these traces be gathered?

Hidden intra-level traces can be gathered by static and dynamic analysis tools. Such dependency analysis tools are presented in [WH91, WH92] and [LMR91, LMR92].

Another technique for gathering such hidden traces is *(program) slicing*. Slicing was introduced by [Wei84].

Informally,

> a *program slice* consists of the parts of a program that (potentially) affect the values computed at some point of interest, referred to as a *slicing criterion*. The task of computing program slices is called *program slicing*. [Tip94]

Slicing is applied in software maintenance in order to decompose the system to be maintained [HTS96, GL91]. To do so, the part of the system to be changed act as the slicing criterion. Forward and/or backward slices are computed. All parts of the system not in the slices need not be considered for performing the change.

The application of slicing for performing ripple effect analysis is described in [WTCR96]. There exist algorithms and techniques for slicing procedural and object oriented software [LH98, LH96], for slicing formal specifications [OA93] and even for binaries [CF97]. Hence, slicing is a promising approach for supporting a specification based maintenance approach.

For surveys on slicing describing various slicing techniques and applications the reader is refered to [Tip94, Kam95, BG96].

Tool support for identifying *hidden inter-level traces* can be provided by dependency analysis tools under the assumption that inter-level traces have been previously established by applying requirements tracing or similar techniques. If inter-level traces are poorly established or not established at all, hidden inter-level traces have to be identified by the maintenance staff.

4.2.3 Materialized Dependencies

Materialized dependencies are dependencies expressed in mechanisms like internal service channels. Such dependencies are based on hidden and explicit dependencies and are used for performing a special task.

One example for a materialized dependency is the chain of implications used by the service channel sketched in section 3.4.1. This maintenance invariant is used by the service channel to decide, whether the requested change of the temperature range is a safe one or not.

$$(D \Rightarrow C) \wedge (C \Rightarrow A) \wedge (A \Rightarrow B)$$

is established by the developer of the service channel and is representing the implementation rationale for this part of the system (cf. section A.1.2).

4.3 Intra-Object Schemas

Within our specification based setting we are aiming at a co-evolution of the systems implementation and specification during adaptive maintenance requests. Service channels should be provided for anticipated changes and service channels should therefore know the dependencies within the system necessary to decide whether to perform the change or not. Hence, those dependencies have to be expressed and stored either within the system or within the service channel. Here, intra-object schemas are presented as one approach to store such dependencies and to make them accessible for maintenance tools like service channels.

4.3.1 Basic Idea

The basic idea behind *intra-object schemas* [MK93] is to establish relationships between the systems specification and implementation and to highlight information, that is somewhere hidden in the system.

Intra-object schemas serve, like conceptual schemas in data bases, as a level of indirection between specification and implementation. Focusing on the specification, it contains information of the kind "what did you want", focusing on the implementation on information like "where can you find it and how is it related". Hence, such intra-object schemas contain intra-level traces and inter-level traces.

Intra-object schemas are provided from the developer to support anticipated changes to the system. Hence, they are holding the information and dependencies necessary to support the change.

The class of changes to be supported by intra-object schemas covers changes of the structure of the object (like modification of constants declared on class level, or changes in the declarations of variables containing the "user-state").

An intra-object schema, as shown in figure 4.3, consists of three parts: the *specification*, the *schema* and the *implementation*.

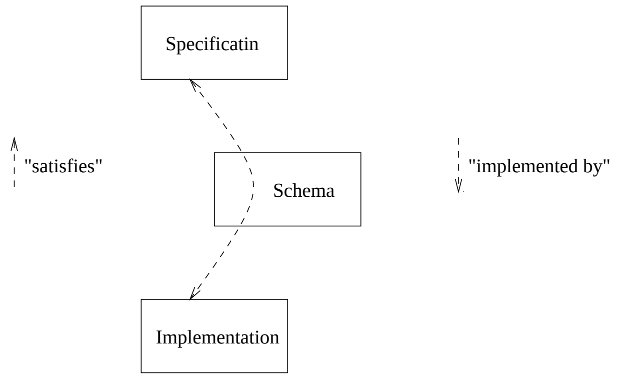

Figure 4.3: Intra-Object Schema

The specification part

The specification part of an intra-object schema contains those parts of the specification, which are possibly affected by the change it supports. The specification can be expressed in a specification language like Object-Z, a relational specification approach [Mil83] or others.

The Implementation part

The implementation part contains the implementation of the system, expressed in the chosen programming language.

The Schema

The schema finally specifies the dependencies between the various parts of the implementation and the specification under concern.

To do so, they contain

- *virtual methods*
  acting as guardians of integrity constraints between various attributes of the object. Virtual methods can be regarded like views in a data base system, but in our setting they are not providing derived functionality, they are used for guarding constraints when a change is to be performed.
- *constraints on attributes of methods*
  Such constraints should offer more possibilities as the type system does, e.g. avoiding a division through zero, if a variable x is divided through a substraction of two variables, which are some disjunct integer subranges, hence it is not possible, that x will be divided by zero. But changing one integer subrange type to include a value of the other subrange, a division by zero is possible. The type system will not identify this. This should be avoided by specifying such constraints.
- *usage constraints*
  to support safeness of the used methods. They should avoid unexpected side effects.
- *inter-object constraints*
  to hold between the component objects of complex objects.

4.3.2 Intra-Object Schemas and Service Channels

In our approach, intra-object schemas are related to service channels. As mentioned earlier, a service channel is provided for a specific, anticipated class of changes. Each service channel should have access to an intra-object schema, containing the necessary explicit and hidden dependencies between the parts of the specification and implementation affected by the change the service channel is provided for.

An intra-object schema is a part of the system, while the service channel need not (internal service channels are part of the system, external service channels are part of a maintenance environment outside the system). But both types of service channels use intra-object schemas to store and guard dependencies.

The service channels contain the materialized dependencies and maintenance methods providing the service channel functionality.

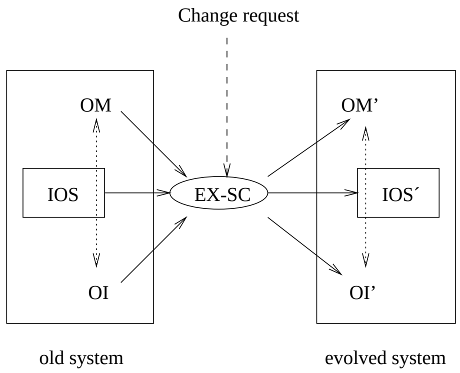

Figure 4.4: External Service Channel

Figure 4.4 sketches an external service channel and figure 4.5 an internal service channel, both containing intra-object schemas.

4.3.3 The system development perspective

If, during development of the system, some parts of the system are identified of possibly being subject to a future change, these parts should be described by means of an intra-object schema to support the anticipated change. This intra-object schema will contain explicit dependencies and hidden dependencies made explicit by an analysis tool or by the developer.

Analyzing the anticipated changes and the underlying dependencies will yield to a decision, what kind of maintenance support should be provided.

If the decision is made, to provide an internal service channel, it will use the intra-object schema. The internal service channel and the intra-object schema will be part of the system.

If the kind of anticipated changes can be taken care of by an external service channel (as part of a maintenance environment) the system will only contain

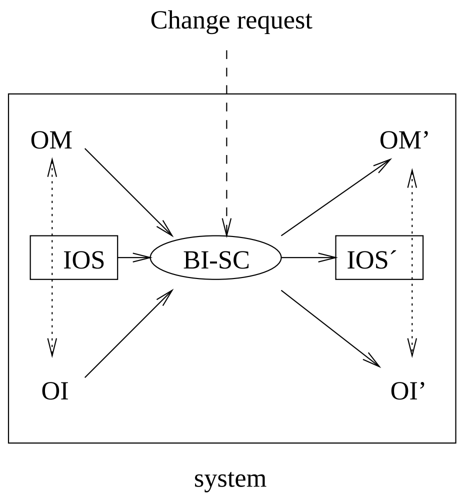

Figure 4.5: Internal Service Channel

the intra-object schema.

The connection between the external service channel and the intra-object schema will be made by plugging the system to be maintained into the maintenance environment. This will enable the external service channel to access the intra-object schema.

4.4 Summary

As mentioned above and in chapter 3, service channels heavily rely (like a human maintainer) on information about the structure of the affected parts of the system and about dependencies among the various artifacts within these parts of the system.

Within this chapter, intra-level traces and inter-level traces were presented as kinds of dependencies, which are of interest for the maintainer (from the maintenance tools perspective, as well as from the human perspective). Such dependencies may be identified as hidden, explicit or materialized dependencies. Intra-object schemas were presented as an approach to store such dependencies. Service channels use intra-object schemas to perform their functionality.

A prototype of a service channel is presented in chapter 7.

The discussion about the effort necessary to provide intra-object schemas has the same arguments as the discussion about providing service channels.

Identifying hidden dependencies can be done by using slicing and dependency analysis tools.

Documenting and storing inter-level dependencies can be done using requirements tracing tools. This is even done in projects, were it is not indented to provide maintenance support.

5

Effects on the Development process

While in the previous chapters the specification (model) based maintenance approach was presented, this chapter is focusing on the different phases of the development process. It outlines, which phases are influenced, when developing a system to be maintained by using service channels.

5.1 The Software Development Process

Development of software is a rather complex process. *Software life cycle models* and *software process models* are used to structure a software project and to control the progress of the project by defining milestones between the start and the end of the project.

A *software life cycle* defines and describes the different stages (or phases) in the lifetime of a software product [Fug00]. Additionally, principles and guidelines how to carry out the different stages and how to complete one stage and move to the next one are described.

A basic, but well known life cycle model is the *waterfall model* [Roy70, Boe76] (cf. figure 5.1). It divides the lifetime of a software product into five phases and provides a sequential ordering of the phases. Completing one phase enforces doing verification and validation activities against the previous phase. Iterations between preceeding phases are foreseen.

The goal of the *analysis* phase is to capture, describe and specify the requirements for the problem to be solved. During the *design* phase, a technical specification of the software system (the solution) is produced and implemented within the *implementation* phase. Although quality assurance is done within all previous phases, a *testing* phase is forseen and necessary.

The *maintenance* phase in figure 5.1 incorporates installing and maintaining (adapting) the system.

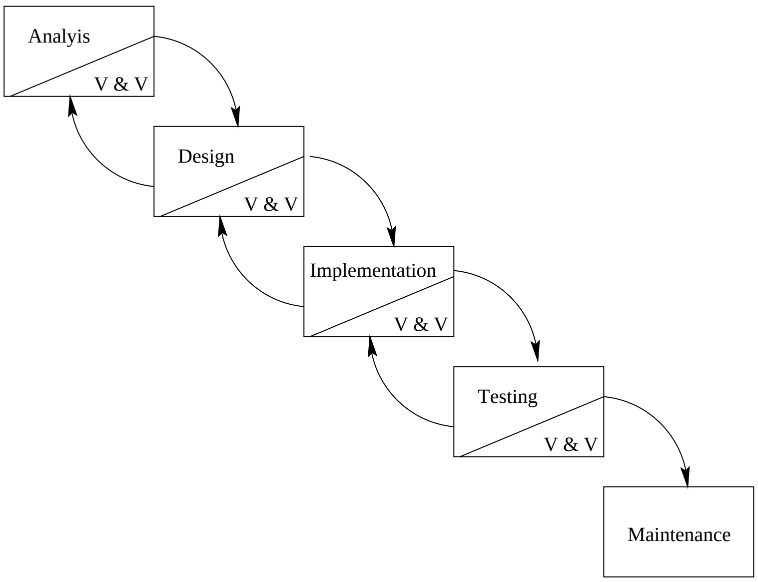

Figure 5.1: The Waterfall Model for Software Development

The *spiral model* [Boe88] breaks with the sequential nature of models like the waterall model and describes software development as a systematic iteration of activities driven by risk analysis and prototyping.

The *fountain model* [HSE90] also focuses on iterations and cycles between the phases. As object oriented systems consist of a (possibly large) number of classes, the fountain model provides not only a life cycle for the whole system, but also allows life cycles for individual classes.

Although all of the above software process models claim to be life cycle models, they are focussing on describing the development effort for the software product. All of them forsee a transition to the deployment and maintenance of the software product, but they poorly define post development stages of the software product. A *phased maintenance model* like the one presented in [BR00] and sketched in section 3.1 describes those post development stages.

Software life cycle models are a starting point for how to develop software. But they are not describing a precise set of actions, tools, methods or organizational structures needed for software development, but software processes do.

A *software process* is a coherent set of policies, organizational structures, technologies, procedures, and artifacts needed to conceive, develop, deploy and maintain a software product [Fug00].

Based on a software life cycle model, a software process describes the needed software technology (e.g. tools, infrastructures or environments), software development methods and techniques (to use the available technology), the organizational behavior of the teams and people involved, and economic aspects (e.g. the software must meet the customers requirements in a specific market setting).

The goal of this chapter is to show how development is influenced, if the system is to be maintained by a specification based maintenance approach. We assume, that the underlying software process model is incorporating a waterfall life cycle model.

If the software product under development is to be maintained following a specification based maintenance approach as described in chapter 3, which requires additional documentation of relationships in the system (cf. chapter 4) and the provision of service channels for anticipated changes, the following technologies and methods should be applied in the specific development stages:

Analysis As service channels are provided for anticipated changes, those changes have to be identified and described within the analysis phase. For describing such anticipated changes, *change cases* are proposed (cf. section 5.2).

Design Inter-level traces have been identified to be necessary for being able to provide service channels. *Design rationales* play an important role. Hence they have to be recorded (cf. section 5.3).

Requirements Tracing An important prerequisite for being able to provide service channels, is the recording of inter-level and intra-level traces. Consequently requirements tracing offers a framework for doing so (cf. section 5.4).

Provision of Service Channels At the end of development of the initial system, service channels are provided for designated change cases (cf. section 5.5).

5.2 Anticipating Potential Changes

Anticipating potential future changes during the requirements analysis phase is an essential prerequisite for being able to provide service channels for some changes. This section focuses on the sources for discovering potential changes and presents an approach for describing such potential changes.

5.2.1 Sources for Potential Changes

As mentioned in chapter 1, software systems in use have to be adapted and changed to reflect the changing requirements they implement or they will become less useful. Lehman summarized this fact in his *law of continuing change* [Leh80].

Brooks also observed that adapting software systems to changing requirements in inevitable. But he also gives hints about the sources of such changes. He states that "the software product is embedded in a cultural matrix of applications, users, laws, and machine vehicles. These all change continually, and their changes inexorably force change upon the software product." [Bro87]

Such changes may be the following [EDF96]:

Market Changes Success in business causes in most cases an increasing number of clients. New clients may come up with additional or changed requirements.

Business Requirements Changes The organization may change its policies and procedures to cope with changes in its environment.

Operational Process Changes Changes in the operational processes of the organization due to internal or economic reason may raise new requirements.

Legislative or Regulatory Changes New laws or changed laws may force a change in the operational processes of the organization.

Imaginative Users Users familiar with the software system may propose enhancements or changes to the system.

Beside those external changes (changes in the users requirements), also internal changes (changes in the systems requirements) have to be considered.

Technical Changes or System Driven Changes Such changes may be triggered by events like the upgrade of the operating system or database manangement system the system is operating on.

Sources for information about such changes may be

- comments of users during requirements analysis sessions
- review of regulatory and legal environment
- drafts of pending legislation and regulations
- review of the organizations technology and/or platform strategy
- planned or scheduled changes to the product (service offerings)

Having identified such potential future changes, they now should be described in a way, that this description may be the base for developing a service channel.

5.2.2 Change Cases

Change cases [EDF96] are constructs to identify and incorporate potential (expected) future changes into an analysis and design model.

Change cases are expressed using the notation of *use cases* [RJB99] and describe new or revised scenarios. Figure 5.2 sketches the relationships of changes, change cases and use cases. The goal of change cases is to capture potential change and describe potential functionality of the system.

Change cases can be used, if an object-oriented design methodology is applied and use cases are used to capture initial requirements. If the methodology also foresees traceability links, these links can also be used to express the relationships between changes, change cases and use cases.

Change cases are not intented to be implemented. But if ever a change (covered by a change case) is intented to be implemented, the links from the change case to other use cases will help to determine the scope of the change on a use case level.

In the setting of change cases,

> a *change* is a requirements-level expression of an intended revision.

A *change case* consists of

1 a use case containing a new or revised scenario, derived from a change to requirements

2 a set of existing use cases, that have to be changed to be consistent with the changed requirements

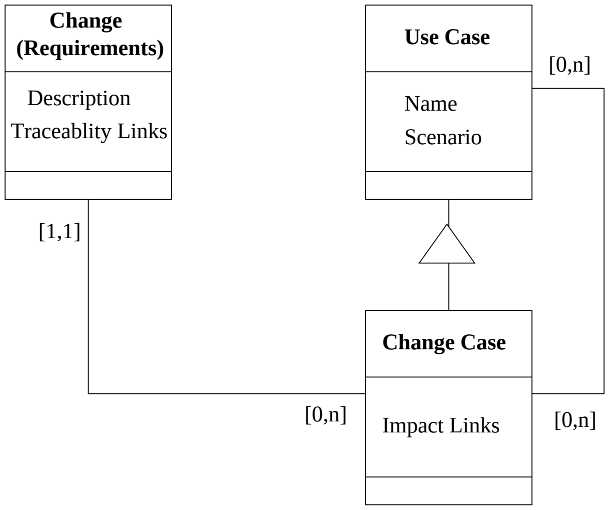

Figure 5.2: Change, Change Cases and Use Cases

A change case is a specialized use case, which inherits the ability to describe scenarios from use cases. It is linked to the change triggering the change case. It is also linked to those use cases, which are affected by the intended change.

Like change cases, which describe changes in the users requirements, changes in the system requirements (internal changes) can be described by linking a change trigger to the affecting system feature.

The prerequisite for profiting from change cases is the usage of an object-oriented development methodology supporting *traceability* (cf. chapter 4 and section 5.4). This supports the identification of all affected parts of the system if a change is to be performed, by following the traces starting from the proper change case.

In our setting, change cases are seen as a way to describe anticipated changes in requirements to serve as a *base for constructing service channels*.

Another way to describe anticipated changes to the systems requirements or potential variants of the system may be the use of *viewpoints* [ACNL97]. Here different viewpoints are defined to represent different variants of the system. This approach is appropriate, if a development methodology like subject oriented design / programming is used. A subject may be represented within a viewpoint.

Describing potential changes or variants of the system as *non-functional requirements* as proposed in [FB97] does not provide an active maintenance support, but expresses such changes in terms of requirements.

5.3 Design Rationale

During the software design process usually a number of possible design choices or design models are developed. The design team then has to make a decision, which of these alternatives will become the final design model. The focus of design rationales is to record the different alternatives and the respective decisions.

Almost all of the works in the field of design rationales use structures similar to the following to describe a design rationale [Han97], [PB88] and [Lee91]:

- a *design issue* or *goal* which is raised concerning a design artifact
- a *design deliberation process*; determines what artifacts to derive and why
- a resulting *design decision* which may effect a number of artifacts.

Surveys on different design rationale approaches can be found in [Lee97] and [MSPD95].

One prerequisite for applying design rationales is the definition of the artifacts to be linked to design rationales. Those artifacts are not limited to design artifacts, as advocated by early approaches like [PB88]. Like artifacts on all levels of abstraction (analysis, design, code) can be linked together via traceability links, those artifacts can also be linked to "design rationales" [CLV92].

In a maintenance setting, like the one described in [LV95] or [ASW93], design rationales are an important source of information for maintenance personal. If unused artifacts are also recorded, design rationales facilitate the reuse or reactivation of such artifacts.

Design rationales together with traceability offer a powerful approach to document a software system during development and thus providing information for maintenance (either for tools like service channels or for maintenance personal doing maintenance by hand). But keeping the design rationales up to date has to be enforced by the development and maintenance process.

In our setting design rationales should be linked to the traceability links

- to provide information to the service channel to perform its task
- and to provide information for the service channel developer.

5.4 Tracing Dependencies

An important prerequisite for providing service channels for anticipated future changes is the capturing of dependencies between different artifacts of the system. In chapter 4 such dependencies were presented. Capturing such dependencies (or traces) has to be done throughout the development of the system. Requirements tracing is an approach, which aims in the same direction.

5.4.1 Requirements Tracing

> *Requirements traceability* refers to the ability to describe and follow the life of a requirement, in both a forward and backward direction (i.e. from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases) [GF94].

Requirements traceability focuses on two areas:

- *Pre-traceability* documents the rationale and sociopolitical context from which the requirements emerge [Jar98], or in other words, it refers to those aspects of a requirement's life prior to the inclusion in the requirements specification [GF94].
- *Post-traceability* links requirements to design and implementation, documenting responsibility assignment, compliance verification, or impact analysis of a requirement [Jar98]. It describes the aspects of a requirement that result from the inclusion in the requirement specification.

Pre-traceability supports the elicitation, documentation and specification of the customers requirements by establishing links between customers and requirements and between derived requirements. Post-traceability links parts (or concepts) of the systems implementation, design or specification to requirements they originate from.

Four kinds of *traceability links* are classified in [Jar98]:

- *Forward from requirements*: links requirements to systems design and implementation components, documenting responsibility assignment or the impact of a requirement.
- *Backward to requirements*: links system design and implementation components to requirement for compliance verification issues.
- *Forward to requirements*: links contributions structures [Got93] and stakeholders etc. to requirements
- *Backward from requirements*: links requirements to contribution structures.

The first two links enable post-traceability, while the latter two focus on pre-traceability.

In [DP98] different *types of traceability data* are summarized:

- Bi-directional links between customer requirements, derived requirements and software components are needed to document the impact of requirements within the system.
- Design rationale, design decisions, alternatives, underlying assumptions are additional kinds of information represented implicitly within the system. Documenting this information and linking it to the according system components (by using the bi-directional links stated above) makes it explicitly available.
- Information on contribution structures, stakeholders, etc. should improve communication and cooperation among teams.
- Documenting process data, performed tasks, etc. supports project planning and control.

In [DP98, GF94] also a survey on tool support for requirements tracing is given.

5.4.2 Requirements Tracing and Service Channels

As mentioned in chapter 4, service channels heavily depend on different kinds of information about the system. Two kinds of dependencies (inter-level and intra-level dependencies) and several ways how such dependencies might be expressed in the system, were presented.

Requirements tracing now offers an approach to capture and store such dependencies during development. Post-traceability links are suited for storing such dependencies.

Applying development environments already incorporating requirements tracing possibilities (cf. tool survey in [DP98, GF94]) facilitate the capturing, storing and management of traceability links. These environments must provide the possibility to adapt and define project specific traceability links in order to support different tracing or maintenance strategies.

Bi-directional links between different artifacts of the system and design rationales are traceability data types suited for being captured and recorded within intra-object schemas.

Even if no service channels are developed, change impact analysis in a maintenance setting can be supported by following post-traceability links.

5.5 Development for Service Channels

The main question of this section is how the strategy to provide and to use service channels for software maintenance influences the development of the initial system.

Figure 5.3 gives again a waterfall model for software development and points out, which phases are influenced by capturing information needed for the provision of service channels.

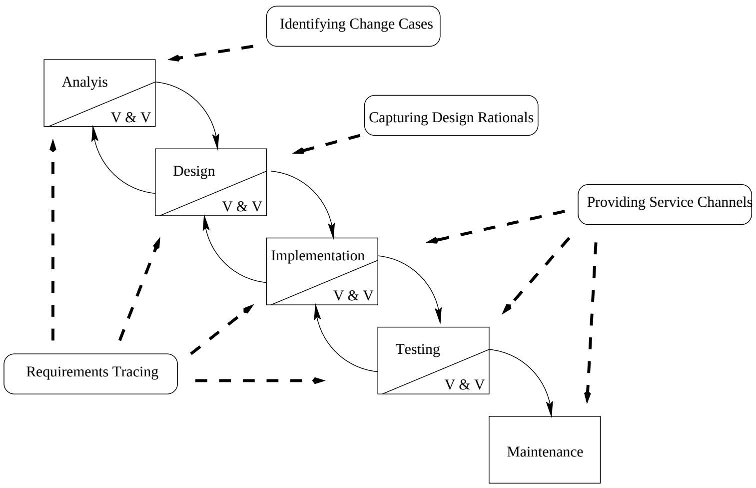

Figure 5.3: Development for Service Channels

If a system is to be maintained via a specification based approach by using service channels, this maintenance strategy has to be known and considered from the beginning of the development. Hence, a commitment from the management and project management is needed, as additional effort has to be spent on documenting the necessary information and dependencies.

5.5.1 Requirements Analysis

The requirements analysis phase is influenced by the fact, that not only functional and non-functional requirements have to be identified and described or modeled. *Potential future changes to the requirements*, which may also cause changes to the system, have to be captured too.

The kind of changes, which can be identified in the analysis phase and hints for possible source of information about such changes is given in section 5.2.1.

Another important aspect is the fact, that those changes have to be described in a proper way. In a proper way means, that the description should not only be another piece of documentation. It should semantically fit to the models provided for functional and/or non-functional requirements and it should also fit into the traceability scheme enforced by the development methodology.

Change cases have been proposed as such constructs for modeling potential future changes in the requirements (cf. section 5.2.2). They are closely related to use cases and link potential changes to the use cases describing those requirements which are affected by the change. They are expressed as use cases and are linked to system components via traceability links.

In a nutshell, identifying and modeling future changes is an additional activitiy to be done within the analysis phase. Descriptions or models of such changes are additional deliverables of that phase.

5.5.2 Design

During the design phase, when a technical specification of the system is built, one additional task comes up. *Design decisions* or *design rationales* have to be captured.

Design rationales are an important kind of inter-level dependencies for developers of service channels or maintenance staff, as they contain information which cannot be automatically or dynamically gathered out of the existing system. They are not hidden within the system and so they have to be documented explicitly.

As the design rationales have to be documented manually, this will cause additional effort to be spent during the design phase. Developers are usually not very keen to produce documentations and have to be motivated and forced to capture design rationales.

5.5.3 Requirements Tracing in all Phases

One activity occurring in all phases of the development process is *requirements tracing* (cf. section 5.4). Within all phases, inter-level and intra-level dependencies have to be captured.

In the words of requirements tracing post-traceability links in both directions (forward from requirements and backward to requirements) are to be established.

During analysis, intra-level dependencies are established to link change cases to the requirements affected by the change.

During design, inter-level dependencies are captured relating artifacts of the design model to the requirements they cover. Design rationales are also captured as well as intra-level dependencies.

Within the implementation phase, also intra-level and inter-level dependencies are captured. Most intra-level dependencies on the code level can be gathered using static and dynamic analysis tools.

Hence, as requirements tracing occurs throughout the whole development process, it is best integrated within a development environment supporting traceability. This would also decrease the effort necessary for capturing the traces.

5.5.4 Providing Service Channels

The development of the software product and of the service channels supporting the maintenance of the software cannot be separated.

During the analysis phase change cases are described. In the design phase a design model for each change case is developed. This design model (including variants of the system) is now the base for the decision to build service channels.

Based on the change cases and on the design model, a decision has to be made which of the potential future changes will be supported by a service channel. This decision has two steps.

First, the changes to be supported by service channels, have to be identified. Factors influencing this decision might be the following:

- How important is the change case for the business of the systems user?
- How likely is the change to occur?
- How much effort will it take, to provide a service channel.

- How complex is the change (with or without maintenance support)?

The *importance* and *likelihood* of a change can be explored during the analysis phase. Hence, when describing a potential change, also the importance and the likelyhood of the change should be documented.

The *necessary effort* for developing a service channel and the *complexity* of the change can be estimated by following the traceability links in the system. Following these links yields the parts of the system affected by the change.

Is the change a rather complex one (without maintenance support) and is the likelyhood and importance of the change high, then the effort of developing a service channel should be spent.

If not, a motivation to develop service channels would be to develop a system as maintainable as possible.

The second decision to be made, is to chose the right type of service channel for the respective change. The question is, *"is the change suitable for being supported by an internal or by an external service channel?"*.

Here again, to decide whether a change can be supported by an internal service channel (local change causing no side effects), the traceability data can be used for a ripple effect analysis.

One important issue is, that quality assurance activities and testing have to be performed after developing a service channel. Hence there has to be an iteration to the testing phase.

Even if no service channels will be developed, the stored traceability data will facilitate software maintenance by supporting the systems (program) understanding task.

In [TN97] the authors report about two studies concerning the efforts spent on software maintenance tasks. In [DBSB91] the authors state, that 30 - 60 % of the maintenance costs is spent for system understanding, following [FH76] system understanding consumes 47 % of the total maintenance effort.

Hence, having available traceability links will help to decrease the effort necessary for system understanding.

6

(Model Based) Maintenance Activities

This chapter sketches the varieties activities, methods and techniques for implementing changes into an existing system. The focus will be on change impact analysis, change propagation and quality assurance issues.

6.1 The Software Change Process

The software change process usually consists of the following steps:

1. Understand software change and determine impact
2. Specify and design software change
3. Implement software change
4. (Re)Test affected software

Unterstanding the required change and determining its possible impact is subject to *change impact analysis* (cf. section 6.2).

Specification, design and implementation of the change includes the location of the change and the implementation of possible additional changes due to ripple effects. *Change Propagation* (cf. section 6.3) covers these issues.

Finally, *regression testing* (cf. section 6.4) is neccessary to test whether the modified system is still satifying its specification.

All the methods and techniques presented below rely on their proper model of the system to change. Most of these models are based on data and/or control flow dependencies.

6.2 Change Impact Analysis

When changes to existing systems have to be made, the issue of estimating the scope of the change (complexity, size etc.) comes up. Side Effects have to be determined. The implementation of the change has to be planned.

For small systems (or programs), the programmer has this scoping and planning in mind (possibly after browsing the source code). But for large systems, this does not work.

Change impact analysis aims at facilitating the understanding and implementation of a change by providing a detailed examination of the consequencies of the requested change.

The basic concept behind change impact analysis approaches is to establish and to analyze relationships among various types of software artifacts (specifications, design, code, tests,..). Analysis of these relationships identifies those work products, which are affected by a requested change. Such relationships among software artifacts have been introduced in chapter 4.

A comprehensive overview of change impact analysis methods and techniques can be found in [Boh96, BA96a, BA96b].

Related terms to impact analysis, sometimes used synonymously, are ripple effect and side effect.

A *side effect* is an

> "error or other undesirable behavior that occurs as a result of a modification." [FW81]

A *ripple effect* is the

> "effect caused by making a small change to a system which affects many other parts of a system." [SMC74]

Two basic technologies are used for change impact analysis: traceability analysis and dependency analysis. Both analysis techniques are sketched in the following sections and program slicing as a method supporting dependency analysis is sketched.

- *Traceability analysis*
  focuses on relationships among all types of software artifacts. It covers relationships between artifacts like documentation, requirements specifications, source code and test cases.
  Traceability is defined as

> "ability to trace between software artifacts generated and modified during the software product life cycle" [BA96a].

Definitions of traceability and traceability relationships have been introduced and discussed in chapter 4 of this work. Traceability analysis focuses on *inter level traces* introduced in chapter 4.

- Dependency Analysis
  Dependency Analysis has a much narrower scope than traceablility analysis. Here, the aim is examining detailed relationships among program (source code) entities like variables, modules and the like.
  The main focus of dependency analysis lies on data flow and control flow analysis.
  In context of the terms introduced in chapter 4, dependency analysis focuses on *intra level traces*.
  Dependency analysis tools are static and dynamic analysis tools. Such tools are presented in [WH91, WH92] and [LMR91, LMR92]. A popular technique for gathering such dependency relationships is *(program) slicing* [Wei84].

In [BA96b] a wide variety of of tools and techniques supporting change impact analysis are presented.

Slicing is presented here as a popular technique supporting dependency analysis in more detail.

6.2.1 Slicing

Program slicing is a technique supporting the restriction of the behaviour of a program to some subset of interest [GL91]. Slicing supports dependency analysis in the way, that it produces subsets of a program, containing statements that are related by certain data and control flow dependencies.

Informally,

> a *program slice* consists of the parts of a program that (potentially) affect the values computed at some point of interest, referred to as a *slicing criterion*. The task of computing program slices is called *program slicing*. [Tip94]

Program slicing was introduced in [Wei84]. Different kinds of slices, ways to construct slices, and applications of slicing has been identified and defined. Surveys of slicing applications and techniques are [BG96, Kam95, Tip94].

A slice (of programm P) is usually denoted as *S(v,n)*, where v stands for a varable (or a set of variables) and n refers to a specific statement in P (usually a statement number). The slice *S(v,n)* contains those parts of the program, that contributed to (influenced) the value of variable v just before the execution of statement n. *S(v,n)* is also called the *slicing criterion*.

Slices are executable programs (in the definition of [Wei84]), which can be computed automatically. Most techniques and algorithms for constructing slices are based on *dependence graphs* containing *data and control flow information* of the program to analyse.

Slices as defined in [Wei84] are called *backward slices*. During the construction of such slices the dependence graph is traversed backwards, starting from statement n.

Forward slices where introduced in [HRB90]. Informally, a forward slice yields those parts of the programm P, that are affected by the value of v at statement n.

Slices are also classified as static or dynamic slices. *Static slices* are not considering the program's input, while *dynamic slices* are computed for a particular fixed input for program P.

Program slicing is applied for program differencing (finding differences between programs), program integration, testing, debugging, quality assurance, software maintenance, reverse engineering and so forth. For more details on the application of slicing, the reader is refered to [BG96].

There exist algorithms and techniques for slicing procedural and object oriented software [LH98, LH96] and for slicing formal specifications [OA93]. An approach for slicing binaries, machine code and assembly code is introduced in [CF97].

In the following sections, the application of slicing in the context of *software maintenance* is sketched. *Decomposition slices* are presented for partitioning the program to change into independent parts to narrow the scope for a required change. An approach for applying *slicing for ripple effect analysis* shows how forward and backward slices combined can be used to localize possible ripple effects. Finally, *specification slicing* is sketched. This variety of slicing approaches shows, that slicing is a well suited mechanism for dependency analysis in model based maintenance setting.

Decomposition Slices

The focus when using *decomposition slices* [GL91] is on partioning the program into (possibly) independent parts. If changes are made in one independent part, just this part needs to be understood and tested after making the change.

A decomposition slice has the same characteristics as ”normal slices”, except that it includes all computations of a variable and is independent of a program location, whereas ”normal slices” capture the value of a variable at a particular program location (statement).

Having now computed a decomposition slice with respect to a variable v (the reader is refered to [GL91] for details on the construction of decompositon slices), the program can now be partioned in the following parts:

- The *independent part* containing statements of the decomposition slice that are not in any other decomposition slice.
- The *dependent part* containing statements of the decomposition slice, that are in another decompostions slice.
- The *compliment* containing statement, that are in some other decomposition slice, but not in the one computed with respect to v.

Using decompositons slices has the following advantages, if a maintainer has to change the variable v:

- Only the independent and dependent part of the decomposition slice created with respect to v have to be understood.
- If just modifications to the indpependent parts are made, there are no side effects in the compliment part.
- If changes are only made in the independent part, just the independent part has to be tested.

Finally, combining the different parts of the programs leads to a modified and tested program [HPR89, GL91].

Slicing for Ripple Effect Analysis

While the decomposition slicing approach is focusing on partitioning the program before making a change, a combination of forward and backward slices supports ripple effect analysis after making a change [HTS96, WTCR96].

The generic model for ripple effect analysis [WTCR96] is the following:

1 Make initial change
2 Identify potentially affected areas due to that change
3 Determine, which of these areas need further changes to remain consistent
4 For each additional change: determine how to make this change
Until no more additional changes are to be made

The slices used are *generalized program slices* [HTS96]. Those kind of slices may be computed in both directions (forward and backward slices) and may be restricted by a set of constraints.

In [WTCR96], three kinds of changes are identified and ripple effect analysis processes are stated. These processes are sketched in the following. The used symbols are described below [1].

- *Define change*
  A define change is a change on the left-hand side of an assignment statement. The right-hand side of the assignment statement remains unchanged.
  Before change: $S : D = f(U_i);\ i > 0$
  After change: $S : D' = f(U_i);\ i > 0$
  The REA process for the define change is the following:
  (1) Forward slicing: $< S, D >$ in P' and $< S, D' >$ in P'
  (2) Backward slicing: $< S, U_i >$ in P', for each i
  The forwad slicing highlights the statements in the initial program P and the resulting program P', which may be affected by ripple effect due to the change of D. The backward slice highlights the statements influencing the right-hand side of the changed statement. The three slices together show all the possible side effects of a define change.
- *Use change*
  A use change is a change on the right-hand side of an assignment statement. The left-hand side remains unchanged.
  Before change: $S : D = f(U_i);\ i > 0$
  After change: $S : D = f(U'_j);\ j > 0$ and $U_i \neq U'_j$
  REA process:
  Forward slicing: $< S, D >$ in P'
  Backward slicing: $< S, U'_j >$ in P', for each j
  A forward slice for the define variable, and a backward slice for each changed use variable highlight the possible side effects of a use change.
- *Control change*

[1] S: Statement in program P; D: Define variable; U_i: Use variable; f: Computation function

A control change is a change that affects (or might affect) the execution sequence of a program.
Before change: $S : D = f(U_1, U_2, .., U_i);\ i > 0$
After change: $S : D = f(U'_1, U'_2, .., U'_j);\ j > 0$ and $U_i \neq U'_j$
Here, the computation function f denotes the predicate expression in conditional and loop constructs. In a control change one or more use variables in the predicate expression are changed.
REA process:
Forward slicing: $< S, U'_1, U'_2, .., U'_j >$; for $j > 0$
Backward slicing: $< S, U'_j - U_i >$
The forward slice is used for identifying side effects down the execution stream introduced by the use variables. For the newly introduces use variables, the backward slice is used to check, whether the usage of the new use variables is valid.

The resulting slices for each kind of change contain those parts of the program P, which is possibly affected by the respective change. Hence, side effects and possible additional changes due to the previous change will appear within those slices.

Specification Slicing

Slicing techniques have not only been developed for analyzing (object oriented) programs, also slicing techniques for formal specifications have been introduced. In [OA93] a slicing technique for the Z Notation [Spi89, Dil94] is presented.

Informally,

> a ”*specification slice* is a set of pieces of specifications that restricts values of a variable.” [OA93]

Specifications slices are, as their counterparts for programs, based on dependency relationships between parts of the specification.

The main goal of using specification slices in software maintenance is to reduce the size of the specification to analyze when changes are to be made to a specification. This is possible, because of the following characteristics of specification slices.

- In most cases, a specification slice is smaller than the original specification.
- A specification slice restricts the variable in focus in exactly the same way, as the original specification does.

Hence for a model based maintenance approach, specification slicing can be applied for impact analysis and for easing ripple effect analysis on a specification level, before changing the source code.

6.3 Change Propagation

While change impact analysis techniques and methods focus on the understanding of changes and their impact to the existing system, change propagation is a process of consitently implementing a change.

After a change has been introduced, the affected (changed) entity of the system may no longer fit to the rest of the system. This is due to the fact that, as mentioned earlier in this work, dependencies and consistency relationships may exist between several entities of the system. The changed entity may also require several services of other entities and provide services to others. Hence, changing the entity may introduce inconsistencies to the system.

Informally,

> *change propagation* keeps track of the inconsistencies introduced by a change and of the location where possible secondary changes are to be made, in order to *re-introduce consistency* to the changed system. [Raj97, Raj96]

Several models for change propagation have been proposed, like [YNTL88, Luq90, Raj97, Raj96]. Here, the model presented in [Raj96] is sketched.

It is based on the evolution of the dependency graph for a given program. Therefore, a model was defined for capturing the *entities* of a program, the *dependencies* and *inconsistencies*.

With respect to a given entity to be changed, top dependencies (entities not influenced by others but influencing others), bottom dependencies, incoming and outgoing dependencies are kept. The *program neighbourhood* for an entity contains all the top, bottom, incoming and outgoing dependencies for that entity.

A step in change propagation is the replacement of one entity by an updated one in the program neighbourhood.

A set containing *marked entities* (entities being marked, because of the inconsistencies due to a change) is also held.

As a change is usually composed of a sequence of different changes, in [Raj96] two change processes based on the above model are presented to describe and define the sequence of changes. The change processes are the *change and fix model* and the *top down change propagation model*, also called MSE (methodology for software evolution). Those models will be briefly sketched in the following.

Change and Fix Model

The change and fix model starts with the change of the entity. If this change is introducing any outgoing inconsistencies, the affected entities will be marked. The marked entities have to be inspected and possibly be changed. This process ends, if there are no more marked entities.

An informal description of the algorithm is given below. The formal definitions can be found in [Raj96].

1. Start with a consistent program P
2. Select the entity to change
3. Change the entity
4. Update the program neighbourhood and the marked entities for the changed entity
5. Check for marked entities
6. If there are marked entities, change them if necessary, until there are no more marked entities.

MSE: Top Down Change Propagation

In contrast to the change and fix model, the MSE model always starts on the top dependencies for an entity to be changed. Therefore, the set of scheduled entities is introduced. The set of scheduled entitites contains the "highest marked" entities in the dependency graph.

The algorithm assures, that changes always propagate top down within the dependency graph.

Again, an informal description of the algorithm is given below. The formal definitions can be found in [Raj96].

1. Start with a consistent program P
2. Select the entity to change from the set of top entities
3. Change the entity
4. Update the program neighbourhood and the marked and scheduled entities for the changed entity

5 Check for marked entities
6 If there are marked entities, change the according scheduled entity, until there are no more marked entities.

6.4 Regression Testing

After identifying a change, analyzing its impact, implementing the change, possibly propagating the change, regression testing is the next activity.

Regression testing involves the retesting of part of the system after it was modified. Only those parts should be retested, that are affected by the modification [KGH+96].

Regression testing has to adress the following questions:

1 How to (automatically) identify the affected parts of the system?
2 What retesting strategy should be used to retest those parts?
3 What coverage criteria are to be used for those retests?
4 How to select, reuse and/or modify existing test cases?

Approaches for the above raised questions can be found in [HMF92, SC96, GHS92, Pos94, JE94].

In [KGH+96, KGH+94, KGH+95] an approach for regression testing for object oriented software is presented. Also an algorithm for defining a test strategy, the so called test order, is presented.

The approach is based on a regression test model consisting of several diagrams. An *object relation diagram (ORD)* describing relations like inheritance, aggregation and associations and also capturing dependencies between clases.

A *block branch diagram (BBD)* captures the interface and internal structure of a class and its dependency to other classes. The BBD consist of a so called body containing the control flow of the member function, global and class data that are used by the member function, input and output parameters, global and class data that are defined by this member function and all other member functions called by this member function.

Out of the BBD the so called *class firewall* is constructed. Informally,

> a *class firewall* for a class C in an object oriented program or library is the set of classes that could be affected by changes to the class C. [KGH+96]

Hence, the class firewall represents the impact of the implemented change. The algorithm for constructing class firewalls is described in [KGH+96, KGH+95]. Also an algorithm for constructing the so called *test order* is presented in [KGH+96, KGH+95].

Informally,

> the *test order problem* for class firewalls can be stated as finding a desirable order for testing the classes that are affected by code changes to a set of classes. [KGH+96]

An effective test order implies reuse of existing test cases [HMF92, KGH+96].

7

Service Channels and Maintenance Environments

This chapter sketches a prototype for a service channel for a change out of the thermometer case study. The integration of service channels into maintenance environments is also outlined.

7.1 Issues for Building Service Channels

When building a service channel, the following questions have to be addressed:

1 What has a service channel to know about the system?
This question has already been adressed in chapter 4, where traceability was discussed.

2 What has a service channel to check when being invoked?
This questions covers mainly the issues of the *admissibility of the requested change* (is the change safe?) and of the *admissibility of the service channel* (does the service channel still fit into the system?).

7.1.1 Admissibility of the Requested Change

As introduced in section 3.3, service channels are maintenance mechanisms for guiding (and possibly performing) safe changes to the system they are built for. A safe change is a change, not violating its maintenance invariants (invariants to hold for this change) and avoiding structural deterioration.

Checking the admissibility of a requested change can be done within the boundaries of the class the change occurs or system wide.

In the first case, when the adminissibility should be checked locally, whithin a class, the local invariants of the changed class are checked. If these invariants hold, the change is *locally safe*. Client classes, using the changed class (server class), can still trust the interface of the server class.

To do so, the service channel must check the *inner condition*.

> The *inner condition* is defined as
>
> $$\forall_i \, C \longrightarrow IC_i.$$
>
> Here, C stands for the changed parts of the class and IC for the invariants of the class restricting C.
>
> The arrow has the meaning that C "satisfies" IC. This means, that the invariants are not violated by the changed parts of the class.
>
> This inner condition must hold for all invariants IC restricting C.

If, for a requested change, one or more of the invariants of the inner conditions do not hold, the service channel will fail and classify the change as unsafe, as clients of this class might not trust the interface of the changed server class.

In the second case, invariants in classes using changed parts of the changed class are also checked. If these also hold, the change is *safe*. In the case, that a client class is restricting the used part of the server class by its own, than those invariants have also to be checked.

To do so, the service channel must check the *call condition*.

> The *call condition* is defined as
>
> $$\forall_j (\forall_i \, C \longrightarrow IC_i) \longrightarrow CC_j.$$
>
> This means, that for a changed C, after the checking of the inner condition, all known call conditions CC restricting C when using it, have to be checked.

Whereas, if a service channel is only checking the inner conditions of a change can only identify local inconsistencies, a service channel checking all call conditions has more "diagnostical power" if a change fails, as it is able to identify violations of invariants in client classes.

In our thermometer case study, for changing the temperature range of the thermometer (cf. appendix A and section 3.4.1), the implementation rationale is yielding the following chain of implications,

$$\underbrace{(D \Rightarrow C)}_{Thermometer} \wedge (C \Rightarrow A) \wedge \underbrace{(A \Rightarrow B)}_{TempDisplay} .$$

Reviewing the chain of implications, the left part of the chain is a local invariant of the *Thermometer* class, whereas the right side are invariants outside the *Thermometer* class, using and restricting D.

Here

$$D \Rightarrow C$$

is the inner condition for this change. If the changed temperature range (D) is satisfying the invariant C, than the change is locally safe.

The call condition is the following chain of implications

$$(D \Rightarrow C) \wedge (D \Rightarrow A)$$

In this case, the invariant A in class *TempDisplay* is the only one restricting D outside the *Thermometer* class. Other invariants in other classes using and restricting D would also have to be included in the full call condition.

7.1.2 Admissibility of the Service Channel

Service channels can be used to perform or to guide changes, as they incorporate knowledge about the system (even implicit and materialized knowledge) and use it to "guarantee" safe changes. Hence, the service channel assumes, that those parts of the system covered by the service channel, have not been changed "unproperly" without using the service channel. The service channel has to check, whether these assumptions are still valid.

One assumption is, that the service channel knows all invariants in the inner and call conditions. Before performing a change, the service channel has to check, whether there are no additional invariants in the system not covered by the inner condition (and call condition) defined in the service channel.

7.2 Prototype of a Service Channel

This section sketches a prototype for a service channel supporting the change of the temperature range (cf. section 3.4.1) in the thermometer case study (cf. appendix A).

As outlined in section 3.4.1, the focus of that change are the two constants restricting the *temp* variable in the *GetTemp*() method of class *Thermometer*.

These constants now have been named *getTempUB* and *getTempLB* denoting the upper and lower bounds of the *temp* variable in the method *GetTemp*(). Also, the remaining constants in the *Thermometer* and *TempDisplay* classes have been named. This yields the following new specification.

The implementation and the implementation rationale as described in appendix A remains unchanged and are therefore not included in this section.

Thermometer

$\upharpoonright(\mathrm{INIT}, YieldTemp)$

$tempUB, tempLB : \mathbb{Z}$
$tempUB = 50$
$tempLB = -50$

$getTempUB, getTempLB : \mathbb{Z}$
$getTempUB = 40$
$getTempLB = -20$

$temp : \mathbb{Z}$

GetTemp
$\Delta(temp)$
$temp' = \ldots$
$getTempLB \leq temp' \leq getTempUB$
/* stores temperature supplied by physical thermometer */

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

INIT
$temp = 0$

$\Box(tempLB \leq temp \leq tempUB)$

$$
\begin{array}{l}
\textbf{TempDisplay} \\
\upharpoonright(\mathit{INIT}, \mathit{ShowActTemp}) \\
\mathit{SignDigit} ::= + \mid - \\
\begin{array}{l|l}
\mathit{digitUB}, \mathit{digitLB} : \mathbb{Z} & \mathit{actTempUB}, \mathit{actTempLB} : \mathbb{Z} \\
\hline
\mathit{digitUB} = 9 & \mathit{actTempUB} = 99 \\
\mathit{digitLB} = -9 & \mathit{actTempLB} = -99
\end{array} \\
\hline
\mathit{digit1}, \mathit{digit2} : \mathbb{Z} \\
\mathit{sign} : \mathit{SignDigit} \\
\mathit{themometer} : \mathit{Thermometer} \\
\hline
\textbf{ShowActTemp} \\
\Delta(\mathit{digit1}, \mathit{digit2}, \mathit{sign}) \\
\mathit{temp} : \mathbb{Z} \\
\hline
\mathit{actTempLB} \leq \mathit{temp} \leq \mathit{actTempUB} \\
\mathit{temp} = \mathit{thermometer}.\mathit{YieldTemp} \\
\mathit{temp} \geq 0 \Rightarrow \\
\quad (\mathit{digit1}' = \mathit{temp} \bmod 10 \\
\quad \mathit{digit2}' = \mathit{temp} \operatorname{div} 10 \\
\quad \mathit{sign}' = +) \\
\mathit{temp} < 0 \Rightarrow \\
\quad (\mathit{digit1}' = -\mathit{temp} \bmod 10 \\
\quad \mathit{digit2}' = -\mathit{temp} \operatorname{div} 10 \\
\quad \mathit{sign}' = -) \\
\hline
\textbf{INIT} \\
\mathit{digit1} = \mathit{digit2} = 0 \wedge \mathit{sign} = + \\
\mathit{thermometer}.\mathit{INIT} \\
\hline
\Box(\mathit{digitLB} \leq \mathit{digit1} \leq \mathit{digitUB}) \\
\Box(\mathit{digitLB} \leq \mathit{digit2} \leq \mathit{digitUB})
\end{array}
$$

To build a service channel for changing the temperature range measured by the thermometer, the following questions have to be considered.

1 When is the requested change safe? What parts of the specification and implementation have to be changed?

2 When is the change unsafe? Why is the change unsafe? Which invariants or definitions are violated by the requested change?

3 What are the assumptions the service channel is built upon? Under which conditions can the service channel be invoked? Can the service channel detect structural deterioration violating these assumptions?

In the following, these considerations are described for the above change and then a specification for a prototype for a service channel supporting this change is given.

7.2.1 Safe change of the temperature range

Reviewing the implementation rationale sketched in appendix A already gives a clue for changing the temperature range in a safe way.

As long as the chain of implications

$$(D \Rightarrow C) \wedge (C \Rightarrow A) \wedge (A \Rightarrow B)$$

holds, the temperatures measured by the thermometer stays consistent with the temperatures displayed by the temperature display.

The change affects the invariant D, which stands for the invariant

$$getTempLB \leq temp' \leq getTempUB$$

in the *thermometer* class. Changing the temperature range means changing the initial definitions of the constants $getTempUB$ and $getTempLB$.

C in the above chain of implications stands for

$$\Box(tempLB \leq temp \leq tempUB)$$

restricting the *temp* variable in the state space of the *thermometer* class.

A stands for the invariant

$$actTempLB \leq temp \leq actTempUB$$

in the *TempDisplay* class, whereas B denotes the invariants

$$\Box(digitLB \leq digit1 \leq digitUB)$$
$$\Box(digitLB \leq digit2 \leq digitUB)$$

restricting the digits displayed.

A and B are part of this chain of implications, as the temperatures measured by the thermometer (restricted by C and D) is moved to the temperature displayed by *TempDisplay* (restricted by A and B).

Hence, reviewing the chain of implications,

$$\underbrace{(D \Rightarrow C)}_{Thermometer} \wedge (C \Rightarrow A) \wedge \underbrace{(A \Rightarrow B)}_{TempDisplay}.$$

the first part is assigned to the *Thermometer* class stands for the inner condition of that change. The second part, assigned to the *TempDisplay* class, completes the call condition.

As the change affects D, a service channel has at least to check the inner condition $D \Rightarrow C$. Hence,

$$getTempUB \leq tempUB$$

and

$$getTempLB \geq tempLB$$

and

$$getTempUB > getTempLB.$$

If this is the case, the requested change is not violating the local invariants on the changed ranges of the *temp* variable in the *Thermometer* class. Other classes, like the *TempDisplay* class, using the value of the *temp* variable of *Thermometer* can still trust the interface of that class. In this case, the service channel could take the role of an *adaptive service channel* by changing the *getTempUB* and *getTempLB* within the *Thermometer* class.

If $D \Rightarrow C$ does not hold, the requested change is violating the history invariant in *Thermometer*. The consequent, besides this violation, is also that classes like *TempDisplay* using the the value of the *temp* variable of *Thermometer* can not trust the interface of *Thermometer* any more. In this case, the service channel could take the role of a *diagnostic service channel* pointing out which invariants are violated, but not performing the requested change.

If the service channel would also check the call condition, it would be able to show, whether the *TempDisplay* class is affected by this violation of the inner condition. Hence, the service channel would be able to show ripple effects outside the class the change takes place.

7.2.2 Unsafe change of the temperature range

The change of the temperature range is unsafe, if the above inner or call conditions do not hold. But different inconsistencies can be observed.

If

$$getTempUB > tempUB$$

or

$$getTempLB < tempLB,$$

then invariant C in the *Thermometer* class is violated. If A and B are not violated, then there are (local) unwanted side effects in the *Thermometer* class, but no unwanted side effects outside this class. The inner condition is violated, but the call condition still holds.

The service channel should point out, that $D \Rightarrow C$ is violated, and that $(C \Rightarrow A) \wedge (A \Rightarrow B)$ is not .

If

$$getTempUB > actTempUB$$

or

$$getTempLB < actTempLB,$$

then $(D \Rightarrow C) \wedge (C \Rightarrow A)$ is violated. Hence there are local (within the *Thermometer* class) and non local unwanted side effects, as the inner and the call conditions are violated. The service channel should point out, which invariants in both classes are affected.

7.2.3 Is the Service Channel up to date?

The third question to check is, whether the assumptions under which the service channel was built, are still valid when the service channel is invoked.

These assumptions may be violated by maintenance activities yielding structural deterioration or by maintenance activities concerning the change the service channels was provided for, put performed outside (without using the) service channel.

The assumptions for our change are the following:

1 The measured temperature is stored in the *temp* variable of class *Thermometer*

2 This *temp* variable is controlled (restricted) by the constants *getTempUB* and *getTempLB* in the *GetTemp()* method, and by the constants *tempUB*

and *tempLB* in the history invariant. There are no other constants or variables or invariants controlling this *temp* variable. Hence, there are no other invariants in the *Thermometer* class to be included in the inner conditions.

3 The *temp* variable in the *Thermometer* class is moved to the *temp* variable in the *TempDisplay* class. Here, the *temp* variable is controlled by the *actTempUB* and *actTempLB* constants. There are no other constants or variables or invariants in the *TempDisplay* class controlling this variable.

4 The usage of the *Thermometer.temp* variable in the *ThempDisplay* class is the only usage of this variable, which is controlled by some constants or variables of invariants. Hence, there are no other invariants to be included in the call conditions.

If these assumptions are still valid, the service channel for this change can be applied properly.

7.2.4 A service channel for changing temperature ranges

Building a service channel for changing the temperature range in our thermometer case study has now to consider the issues raised above. To do so, the intra-level dependencies on the specification level have to be analyzed.

Additionally, the service channel needs to know the inter-level dependencies relating the specification to the implementation.

In the rest of this section a service channel supporting the change of the temperature range in the thermometer case study is sketched and (partially) described using the *Z* notation [Spi89, Dil94].

The schema named *ServiceChannel* includes a schema named *IOS* standing for *Intra-Object Schema*. *IOS* is capturing the intra-level dependencies of the specification and implementation as well as the inter-level dependencies. The dependencies are named as sketched in the tables in chapter 4.

The service channel takes two new boundary values *newUB*? and *newLB*? for the temperature stored in the *Thermometer* class as input.

Checking the safeness of the requested change is done by checking the inner condition, by comparing the input values with *ValueOf("tempUB")* and *ValueOf("tempLB")*.

The function *ValueOf()* is defined in the *IOS* yielding the initial values of constants in the specification. As an Intra-Object Schema (cf section 4.3) captures relationships between various concepts of the specification and the according implementation, but not between the actual values of those concepts, a generic function like *ValueOf ()* is needed to get the value of those concepts.

The constants *tempUB* and *tempLB* are the two constants restricting the *temp* variable in the state space of the *Thermometer* class, described by invariant C in the specification.

As the input variables denote the new values of *getTempUB* and *getTempLB*, this service channel checks, whether $D \Rightarrow C$, the inner condition, holds.

Checking the admissibility of the service channel means checking the assumptions the service channel is built upon. First, the *temp* variable in the *Thermometer* class is only restricted by *getTempUB* and *getTempLB* (within the *GetTemp*() method) and by *tempUB* and *tempLB* in the state space via the history invariant. Second, this history invariant and the two constants in this invariants are not checked in the implementation.

This checking of the admissibility of the service channel is done by using the *ControlesValueOf()* function defined in *IOS*, standing for the dependency with the same name described in table 4.2.

The function *implementedBy*() stores the inter-level dependencies in the *IOS*. Here, the service channel checks that both constants of the history invariant are not implemented in the source code.

$$
\begin{array}{l}
\textbf{ServiceChannelSafe}_{TempRange} \\
\hline
IOS \\
newUB?, newLB? : \mathbb{Z} \\
evolItemsSpec! : \mathbb{P}\, SpecItems \\
evolItemsImpl! : \mathbb{P}\, ImplItems \\
\hline
\text{/* Check of inner conditions */} \\
newUB? > newLB? \\
newUB? \leq ValueOf(\text{"}tempUB\text{"}) \\
newLB? \geq ValueOf(\text{"}tempLB\text{"}) \\
\text{/* Admissibility of Service Channel */} \\
ControlesValueOF(\text{"}getTempUB\text{"}, \text{"}temp\text{"}) \\
ControlesValueOF(\text{"}getTempLB\text{"}, \text{"}temp\text{"}) \\
\nexists\, item : DataItem \mid \\
\qquad item \neq \text{"}getTempUB\text{"} \wedge \\
\qquad item \neq \text{"}getTempLB\text{"} \wedge \\
\qquad ControlesValueOF(item, \text{"}temp\text{"}) \\
ControlesValueOF(\text{"}tempUB\text{"}, \text{"}temp\text{"}) \\
ControlesValueOF(\text{"}tempLB\text{"}, \text{"}temp\text{"}) \\
\nexists\, item : DataItem \mid \\
\qquad item \neq \text{"}tempUB\text{"} \wedge \\
\qquad item \neq \text{"}tempLB\text{"} \wedge \\
\qquad ControlesValueOF(item, \text{"}temp\text{"}) \\
implementedBy(\text{"}tempUB\text{"}) = \emptyset \\
implementedBy(\text{"}tempLB\text{"}) = \emptyset \\
\text{/* Specification and Implementation Item to be changed */} \\
evolItemsSpec! = \text{"}getTempUB\text{"} \cup \text{"}getTempLB\text{"} \\
evolItemsImpl! = \\
\qquad implementedBy(\text{"}getTempUB\text{"}) \cup implementedBy(\text{"}getTempLB\text{"}) \\
\hline
\end{array}
$$

The output of this service channel are two sets *evolItemsSpec!* and *evolItemsImpl!* containing the specification and implementation parts (in this example range_lb and range_ub of the Thermometer class) to be changed.

DataItem is holding the actual values of the specification and implementation parts covered by this service channel.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad ServiceChannelUnSafe_{TempRange} \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
IOS \\
newUB?, newLB? : \mathbb{Z} \\
evolItemsSpec! : \mathbb{P}\, SpecItems \\
evolItemsImpl! : \mathbb{P}\, ImplItems \\
violatedItemsSpec! : \mathbb{P}\, SpecItems \\
violatedItemsImpl! : \mathbb{P}\, ImplItems \\
\hline
\text{/* Violation of inner conditions */} \\
(\\
newUB? \leq newLB? \wedge \\
violatedItemsSpec! = newUB? \cup newLB? \\
) \\
\vee \\
(\\
(newUB? > ValueOf(\text{"}tempUB\text{"}) \vee \\
newLB? < ValueOf(\text{"}tempLB\text{"})) \wedge \\
violatedItemsSpec! = \\
\quad\quad \text{"}getTempUB\text{"} \cup \text{"}getTempLB\text{"} \cup \text{"}tempUB\text{"} \cup \text{"}tempLB\text{"} \\
) \\
evolItemsSpec! = evolItemsImpl! = \varnothing \\
\hline
\end{array}
$$

$ServiceChannelUnSafe_{TempRange}$ covers the cases, where the requested change violates the inner condition $D \Rightarrow C$ and where $newUB? \leq newLB?$. Hints are also given, which parts of the specification are affected.

The consequence is, that the service channel returns an empty set of implementation and specification items to be changed. The specification items causing the failing of the service channel are returned. Hence the service channel acts a diagnostic service channel.

The third part of this service channel specification is to cover the cases, where the service channel is admissible or not. This is described in the schema *ServiceChannelNotAdmissible*.

$$
\begin{array}{|l}
\hline
\textit{ServiceChannelNotAdmissible}_{TempRange} \\
IOS \\
\hline
\text{/* Service Channel not admissible */} \\
\exists\ item : DataItem\ | \\
\qquad ControlesValueOF(item, "temp") \wedge \\
\qquad item \notin \{"getTempUB", "getTempLB", "tempUB", "tempLB"\} \\
\vee \\
implementedBy("tempUB") \neq \varnothing \\
\vee \\
implementedBy("tempLB") \neq \varnothing \\
\hline
\end{array}
$$

The service channel is not admissible, if additional constraints exists on the *temp* variable or if in the implementation the history invariant is somehow implemented. These violations of the service channels assumptions about the system may be introduced by maintenance activities concerning the temperature range without using or updating the service channel. From the view of the service channel, structural deterioration occurred and the use of the service channel is not admissible.

To complete the specification of a service channel for changing the temperature range, the three parts of the specification are disjuncted:

$TempRangeServiceChannel \mathrel{\widehat{=}}$
$ServiceChannelSafe_{TempRange} \vee$
$ServiceChannelUnsafe_{TempRange} \vee$
$ServiceChannelNotAdmissible_{TempRange}$

This service channel may play different roles for the above change:

- In the case, the *change is safe* and does not violate the inner condition, the service channel may play an *adaptive* or *verificative* role.
  If the service channel plays an adaptive role, the returned sets *evolItemsSpec!* and *evolItemsImpl!* are not empty. If the service channel plays a verificative role, those sets are returned as empty sets.
  As the change is local and no call conditons are to be checked, the service channel may be realized as an internal service channel playing an adaptive role.
  As an external service channel (verificative role) it is able to point out the parts to be changed.
- In the case, the *change is unsafe* and violates the inner condition, it may

play a diagnostic or verificative role depending on its realization. A diagnostic service channel is an internal service channel pointing out the violated invariants if it fails to play an adaptive role.

- If the *service channel is not admissible*, than the service channel fails.

7.3 Implementation Strategies for Service Channels

How can this above specified service channel be implemented? In section 3.5 some hints are given. Internal and External Service Channels are sketched. This section now briefly sketches how the above service channels can be realized.

7.3.1 Internal Service Channels

Building an internal service channel means, building a modification mechanism inside the system.

A simple internal service channel for the above change would look like this:

```
Thermometer::ServChan_TempRange(int l,u){
if (l >= -50 && u <= 50 && l < u)
 { change_const(range_lb, l);
   change_const(range_ub, u) }
 else
 { // non admissible change request
 }
}
```

This internal service channel only changes the constants holding the temperature range if the change is safe. It offers no diagnostics. If the change is safe (the inner condition holds) it performs the change.

The decision to put diagnostical functionality into this service channel if the change request cannot be fullfilled in a safe way, it is on the descrition of the designer of the service channel. The above code chunk of such a service channel shows no diagnostic functionality.

An important issue is, that for some changes, *recompilation* of the system is neccessary.

7.3.2 External Service Channels

External Service Channels are modification mechanisms outside of the system to maintain. Their functionality will be verificative or diagnostic service channels.

How such service channels can be integrated in a maintenance environment is shown in figure 7.1, which sketches the architecture of such an environment [CvM93].

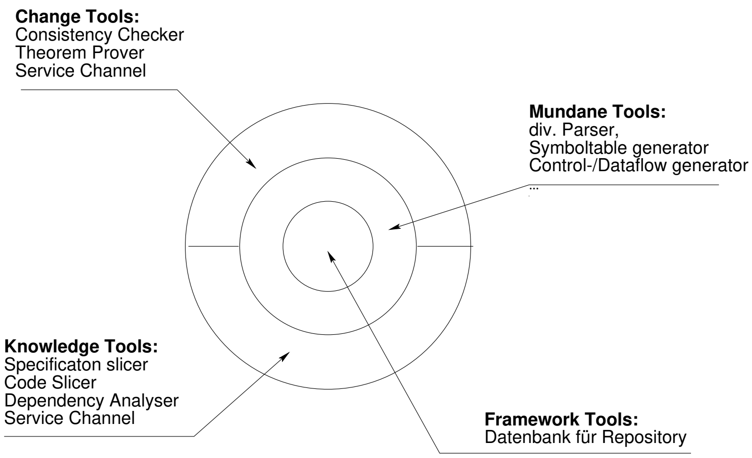

Figure 7.1: Architecture of a maintenance environment

The base of such a maintenance environment are the *framework and mundane tools*. They incorporate basic functionalities like the repository or a dictionary. Also several parsers and symbol table generators can be integrated. These two layers of the architecture offer basic functionality for software development or maintenance environments.

The repository is used for storing and managing the different types of dependencies and design rationales for the system to be maintained.

The *knowledge and change* tools layer incorporates tools needed for specification based maintenance using service channels, including the service channels themselves.

The service channel now uses the information stored in the repository for analyzing and possible supporting a requested change. It may also use the available knowledge tools like specification or code slicers and dependency analyzer and the available change tools for carrying out the requested change.

Such an environment should also offer the possibility to define ad-hoc service channels. Such service channel have no specific documentation in the repository. They can be defined if a requested change becomes known, they build the neccessary dependencies on the fly (if they do not exist) and then they analyze the change.

Such ad-hoc service channels are beyond the scope of this work.

- IOS generated by tools of maintenance environment
- SC described for the system in the maintenance environment

8

Conclusion and Further Work

In Bennett's and Rajlich's roadmap for software maintenance and evolution [BR00] a life cycle model for software maintenance (cf section 3.1) and future research directions are presented.

With respect to this life cycle model, the goal should be to keep the system under maintenance as long as possible in the evolution stage. This means, the system shows architectural integrity (all models of the system are available, up to date and corresponding).

The studies referenced and presented in section 1.3 show, that the focus of software maintenance is still the source code. Hence, most system under maintenance quickly leave the evolution stage.

One of the research directions identified in Bennett's and Rajlich's roadmap is to raise the level of abstraction of the language or model maintenance activities happens in order to preserve architectural integrity.

This work motivates, that *specification (model) based maintenance* (evolution of models of the system together with the source code) is an approach to preserve the systems architectural integrity.

Model based maintenance means establishing a rigorous maintenance process, that enforces change the requirements, specification and/or design models first and propagating the change to the source code afterwards.

In this work, *object model evolution* was presented as our view on *specification (model) based maintenance*. The model of the system we are focusing on is the (formal) specification of the system and enforces a *co-evolution of the systems specification and implementation*.

To achieve this goal, specification and implementation have to be closely related by documenting dependencies like *inter-level traces* (dependencies between specification and implementation) and *intra-level traces* (dependencies within the specification and within the source code). Various approaches for gathering and documenting these dependencies have been preseted.

Additionally, the concept of *service channels* has been introduced to instrument specification based maintenance. Service channels provide semi-automatic maintenance support for already anticipated changes. For such anticipated changes, they document the necessary dependencies and provide support in three ways:

- As *adaptive service channel* they transform the system according to the required change they implement. This automatic support is provided, if the required change is not introducing any unwanted side effects.
- As *verificative service channels* they support ripple effect analysis, change propagation and the generation of test cases.
- As *diagnostic service channels* (adaptive or verificative service channels that fail) they point out why a service channel failed.

A prototype for a service channel has been sketched in this work. Two case studies showing maintenance support by service channels have been described.

Providing service channels for an anticipated change, requires the analysis of the change, documentation of the required system dependencies and implementation of the service channel. The effects of these additional activities during system development have been shown in this work.

The *benefits* of using service channels (or a model based maintenance approach) are the following:

- When applying service channels (or a model based maintenance approach) the systems architectural integrity is preserved and *the system stays in the evolution stage*. This decreases the effort needed for system understanding.
- The higher effort in development is paid back by the decreased efforts for system understanding and maintenance activities. The same applies for higher efforts in system documentation.
- The higher efforts for system development due to additonal documentation of dependencies and the provision of service channels can be decreased by the usage of established, tool supported techniques like requirments tracing or program slicing.

Open Issues and further work:

In this work the concept of service channels and a specification of a service channel at the prototype level have been presented. Implementation strategies for external and internal service channels have to be refined and validated. The defintion of classes of changes and their supporting service channels also have to be refined.

An further issue is the one, to which extent existing service channels (or the documentations of system dependencies) may be re-used to support ad hoc maintenance activities for not anticipated, but at some time required changes.

A

The Thermometer Case Study

The Thermometer example [PM98] is (a part of) a system to display the temperature measured by a thermometer. The thermometer, as originally designed, measures and supplies the temperature in centigrades. It covers a range between -50 and 50 centigrades. The system can be customized to a narrower range of temperatures (here, it is customized to -20 .. 40). The system then displays the temperature using a two-digit display and a sign indicating wether the temperature is below 0 or not.

A.1 OO Analysis Model

The object-oriented analysis model is described using the OMT notation [RBP+91]. It consists of two classes.

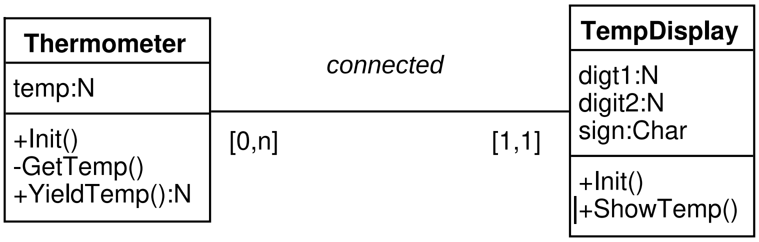

Figure A.1: Temperature Display Object Model

Class *Thermometer* represents the interface to the physical thermometer. The private method *GetTemp()* polls the physical thermometer and stores the measured temperature within the state space of the class. *YieldTemp()* supplies the recently measured temperature to its caller.

Class *TempDisplay* stores the values for the two digits to desplay and a sign indicating whether the temperature is below zero or not in its statspace and is connected to one thermometer. *ShowTemp()* updates the display by interrogating the connected thermometer.

A.1.1 Specification Level

The *object specification* in our *object model* is represented using *Object-Z* [CDD+90, DKRS91, DRS94]. It consists of two objects, *TempDisplay* and *Thermometer*.

Thermometer

$\upharpoonright(\textit{INIT}, \textit{YieldTemp})$

$$temp : \mathbb{Z}$$

GetTemp

$$\Delta(temp)$$

$$temp' = \ldots$$
$$-20 \leq temp' \leq 40$$

/* stores temperature supplied by physical thermometer */

YieldTemp

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

$$temp\_out! = temp$$

INIT

$$temp = 0$$

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

$$
\begin{array}{l}
\textbf{TempDisplay} \\
\upharpoonright(\mathit{INIT}, \mathit{ShowActTemp}) \\
\mathit{SignDigit} ::= + \mid - \\
\hline
\mathit{digit1}, \mathit{digit2} : \mathbb{Z} \\
\mathit{sign} : \mathit{SignDigit} \\
\mathit{themometer} : \mathit{Thermometer} \\
\hline
\textbf{ShowActTemp} \\
\Delta(\mathit{digit1}, \mathit{digit2}, \mathit{sign}) \\
\mathit{temp} : \mathbb{Z} \\
\hline
-99 \leq \mathit{temp} \leq 99 \\
\mathit{temp} = \mathit{thermometer}.\mathit{YieldTemp} \\
\mathit{temp} \geq 0 \Rightarrow \\
\quad (\mathit{digit1}' = \mathit{temp} \bmod 10 \\
\quad \mathit{digit2}' = \mathit{temp} \operatorname{div} 10 \\
\quad \mathit{sign}' = +) \\
\mathit{temp} < 0 \Rightarrow \\
\quad (\mathit{digit1}' = -\mathit{temp} \bmod 10 \\
\quad \mathit{digit2}' = -\mathit{temp} \operatorname{div} 10 \\
\quad \mathit{sign}' = -) \\
\hline
\textbf{INIT} \\
\mathit{digit1} = \mathit{digit2} = 0 \wedge \mathit{sign} = + \\
\mathit{thermometer}.\mathit{INIT} \\
\hline
\square(0 \leq \mathit{digit1} \leq 9) \\
\square(0 \leq \mathit{digit2} \leq 9)
\end{array}
$$

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.

A.1.2 Implementation Rationale

Implementing this specification, the following points might be considered.

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

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

and

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

are driven by the requirement of a two-digit display. They are interrelated such that knowing one of them and the way to calculate the values of the two digits (digit1, digit2), implies the other one, hence

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

The *Thermometer*-object has the class invariant

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

representing some physical limitations of the thermometer and the plausibility check

$$D: \quad -20 \leq temp \leq 40$$

Implementing this specification, one notes that the value of the variable *temp* in the *Thermometer*-object (restricted by constraint C and D) 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, and furthermore D is stronger than C the chain of implications

$$(D \Rightarrow C) \wedge (C \Rightarrow A) \wedge (A \Rightarrow B)$$

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

A.1.3 C++ like Implementation

```
class Thermometer {
int temp;
int range_ub, range_lb;
public:
```

```
  GetTemp();
  int YieldTemp();
  Thermometer();
};


int Thermometer::YieldTemp() {
  return temp;
}

Thermometer::GetTemp() {
int newtemp;
...
if (range_lb <= newtemp <= range_ub)
   temp = newtemp
}

Thermometer::Thermometer() {
temp = 0;
range_lb = -20;
range_ub = 40
}



class TempDisplay {
int digit1, digit2;
char sign;
Thermometer thermometer;
public:
  ShowActTemp();
  TempDisplay();
};

TempDisplay::ShowActTemp() {
int temp;
temp = thermometer.YieldTemp;
sign = ``+'';

if (temp < 0) {
   temp = - temp;
```

```
   sign = ‘‘-’’ }

digit1 = temp mod 10;
digit2 = temp div 10;
}

TempDisplay::TempDisplay() {
digit1 = digit2 = 0;
sign = ‘‘+’’;
thermometer = new Thermometer
}
```

B

The TeamCalendar Case Study

The TeamCalendar case study presents a system for organizing meetings and meeting rooms for a small organization. An object-oriented analysis model [Hus99] is presented as well as a formal specification.

B.1 OO Analysis Model

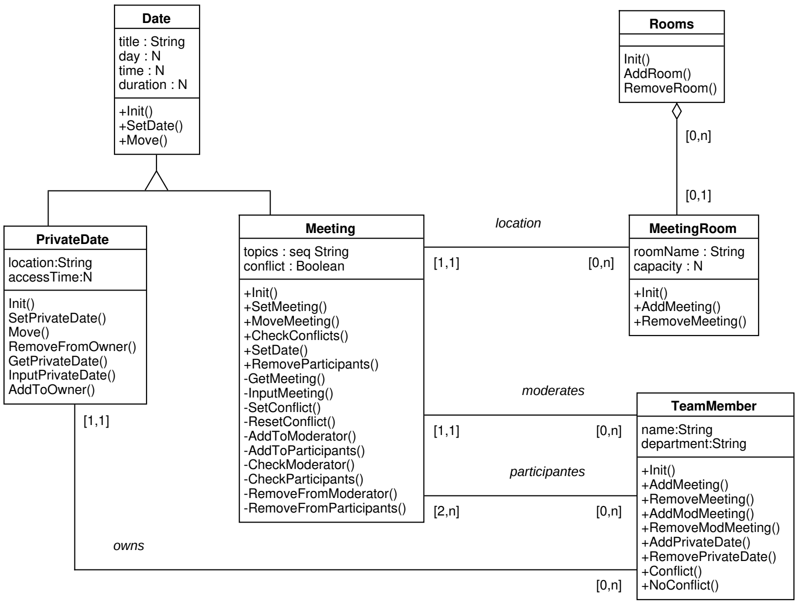

Figure B.1: TeamCalender - Objectmodel

B.2 Object-Z Specification

$$Char ::= a \mid b$$
$$String == \operatorname{seq} Char$$

MeetingRoom

$$\upharpoonright(\mathit{INIT}, AddMeeting, RemoveMeeting)$$

$$\begin{array}{l} roomName : String; \\ capacity : \mathbb{N}; \\ meetings : \mathbb{P}\, Meeting \\ \hline \forall\, m : meetings \bullet \#m.participants \leq capacity \end{array}$$

INIT

$$meetings = \varnothing$$

$$AddMeeting \mathrel{\widehat=} [\,\mathrm{true}\,]$$
$$RemoveMeeting \mathrel{\widehat=} [\,\mathrm{true}\,]$$

Rooms

$$\upharpoonright(\mathit{INIT}, AddRoom, RemoveRoom)$$

$$\begin{array}{l} maxcapacity : \mathbb{N} \\ \hline maxcapacity = 20 \end{array}$$

$$\begin{array}{l} existingrooms : \mathbb{P}\, MeetingRoom_{\copyright} \\ \hline \forall\, r : existingrooms \bullet r.capacity \leq maxcapacity \end{array}$$

INIT

$$\mathrm{true}$$

$$AddRoom \mathrel{\widehat=} [\,\mathrm{true}\,]$$
$$RemoveRoom \mathrel{\widehat=} [\,\mathrm{true}\,]$$

Date

$title : String;$
$day : \mathbb{N};$
$time : \mathbb{N};$
$duration : \mathbb{N}$

INIT

$title = \langle\,\rangle$
$day = 1011999$
$time = 0$
$duration = 0$

SetDate

$\Delta(title, day, time, duration)$
$newtitle? : String;$
$newday? : \mathbb{N};$
$newtime? : \mathbb{N};$
$newduration? : \mathbb{N}$

$title' = newtitle?$
$day' = newday?$
$time' = newtime?$
$duration' = newduration?$

Move

$\Delta(day, time)$
$newday? : \mathbb{N};$
$newtime? : \mathbb{N}$

$day' = newday?$
$time' = newtime?$

Meeting

$\upharpoonright(\textit{INIT}, \textit{SetMeeting}, \textit{MoveMeeting}, \textit{CheckConflicts}, \textit{SetDate}, \textit{RemoveParticipants})$

$\textit{Date}$

$$\begin{array}{l} maxparticipants : \mathbb{N} \\ \hline maxparticipants = 10 \end{array}$$

$$\begin{array}{l} topics : \operatorname{seq} String; \\ moderator : TeamMember; \\ participants : \mathbb{P}\, TeamMember; \\ location : MeetingRoom; \\ conflict : \mathbb{B} \\ \hline moderator \in participants \\ \#participants \leq location.capacity \\ \#participants \leq maxparticipants \\ \#participants > 2 \\ \#topics > 0 \end{array}$$

INIT

$$\begin{array}{l} topics = \langle\, \rangle \\ participants = \varnothing \\ \neg\ conflict \end{array}$$

GetMeeting

$$\begin{array}{l} m! : Meeting \\ \hline m! = self \end{array}$$

SetConflict

$$\begin{array}{l} \Delta(conflict) \\ \hline \neg\ conflict \\ conflict' \end{array}$$

ResetConflict

$$\begin{array}{l} \Delta(conflict) \\ \hline conflict \\ \neg\ conflict' \end{array}$$

InputMeeting

$$\begin{array}{l} \Delta(topics, moderator, participants, location) \\ newtopics? : \operatorname{seq} String; \\ newmoderator? : TeamMember; \\ newparticipants? : \mathbb{P}\, TeamMember; \\ newlocation? : MeetingRoom \\ \hline \#newtopics? > 0 \\ \#newparticipants? \leq newlocation?.capacity \\ \#newparticipants? \leq maxparticipants \\ newmoderator? \in newparticipants? \\ topics' = newtopics? \\ moderator' = newmoderator? \\ participants = newparticipants? \\ location' = newlocation? \end{array}$$

/* Class Meeting: to be cont'd */

Meeting

/* Class Meeting: cont'd */

$AddToModerator \mathrel{\widehat=} GetMeeting \mathbin{;} moderator.AddModMeeting[m?/newmeeting?]$

$AddToParticipants \mathrel{\widehat=} \bigwedge p : participants \bullet GetMeeting \mathbin{;} p.AddMeeting[m?/newmeeting?]$

$SetMeeting \mathrel{\widehat=} InputMeeting \wedge AddToModerator \wedge AddToParticipants$

$$\begin{aligned} CheckModerator \mathrel{\widehat=} & (GetMeeting \mathbin{;} moderator.Conflict[m?/meeting?]) \wedge SetConflict \\ & [] \\ & GetMeeting \mathbin{;} moderator.NoConflict[m?/meeting?] \end{aligned}$$

$$\begin{aligned} CheckParticipants \mathrel{\widehat=} & \bigwedge p : participants \bullet \\ & (GetMeeting \mathbin{;} p.Conflict[m?/meeting?]) \wedge SetConflict \\ & [] \\ & GetMeeting \mathbin{;} p.NoConflict[m?/meeting?] \end{aligned}$$

$CheckConflicts \mathrel{\widehat=} CheckModerator \wedge CheckParticipants$

$MoveMeeting \mathrel{\widehat=} Move \wedge ResetConflict$

$RemoveFromModerator \mathrel{\widehat=} GetMeeting \mathbin{;} moderator.RemoveModMeeting[m?/meeting?]$

$RemoveFromParticipants \mathrel{\widehat=} \bigwedge p : participants \bullet GetMeeting \mathbin{;} p.RemoveMeeting[m?/meeting?]$

$RemoveParticipants \mathrel{\widehat=} RemoveFromModerator \wedge RemoveFromParticipants$

PrivateDate

$\upharpoonright(\textit{INIT}, \textit{SetPrivateDate}, \textit{Move}, \textit{RemoveFromOwner})$

Date

$location : String;$
$accessTime : \mathbb{N};$
$owner : TeamMember$

INIT

true

GetPrivateDate

$pd! : PrivateDate$

$pd! = self$

InputPrivateDate

$\Delta(location, accessTime, owner)$
$newlocation? : String;$
$newowner? : TeamMember;$
$newaccessTime? : \mathbb{N}$

$location' = newlocation?$
$owner' = newowner?$
$accessTime' = newaccessTime?$

$AddToOwner \mathrel{\widehat=} GetPrivateDate \mathbin{⨾} owner.AddPrivateDate[pd?/newpd?]$

$SetPrivateDate \mathrel{\widehat=} InputPrivateDate \wedge AddToOwner$

$RemoveFromOwner \mathrel{\widehat=} GetPrivateDate \mathbin{⨾} owner.RemovePrivateDate$

TeamMember

$$
\begin{array}{l}
name : String; \\
department : String; \\
moderates : \mathbb{P}\, Meeting; \\
participates : \mathbb{P}\, Meeting; \\
privateDates : \mathbb{P}\, PrivateDate \\
\hline
moderates \subseteq participates
\end{array}
$$

AddMeeting

$$
\begin{array}{l}
\Delta(participates) \\
newmeeting? : Meeting \\
\hline
newmeeting? \notin participates \\
newmeeting? \in participates'
\end{array}
$$

AddModMeeting

$$
\begin{array}{l}
\Delta(moderates) \\
newmeeting? : Meeting \\
\hline
newmeeting? \notin moderates \\
newmeeting? \in moderates'
\end{array}
$$

RemoveMeeting

$$
\begin{array}{l}
\Delta(participates) \\
meeting? : Meeting \\
\hline
meeting? \in participates \\
meeting? \notin participates'
\end{array}
$$

RemoveModMeeting

$$
\begin{array}{l}
\Delta(moderates) \\
meeting? : Meeting \\
\hline
meeting? \in moderates \\
meeting? \notin moderates'
\end{array}
$$

AddPrivateDate

$$
\begin{array}{l}
\Delta(privateDates) \\
newpd? : PrivateDate \\
\hline
newpd? \notin privateDates \\
newpd? \in privateDates'
\end{array}
$$

RemovePrivateDate

$$
\begin{array}{l}
\Delta(privateDates) \\
pd? : PrivateDate \\
\hline
pd? \in privateDates \\
pd? \notin privateDates'
\end{array}
$$

Conflict

$$
\begin{array}{l}
meeting? : Meeting; \\
conflict! : \mathbb{B} \\
\hline
(\exists\, m : participates \bullet \\
\quad (m.time < meeting?.time \land \\
\quad (m.time + m.duration) > meeting?.time) \lor \\
\quad (m.time > meeting?.time \land \\
\quad (meeting?.time + meeting?.duration) > m.time)) \\
conflict!
\end{array}
$$

NoConflict

$$
\begin{array}{l}
meeting? : Meeting; \\
conflict! : \mathbb{B} \\
\hline
(\nexists m : participates \bullet \\
\quad (m.time < meeting?.time \land \\
\quad (m.time + m.duration) > meeting?.time) \lor \\
\quad (m.time > meeting?.time \land \\
\quad (meeting?.time + meeting?.duration) > m.time)) \\
\neg\, conflict!
\end{array}
$$

$$
\begin{array}{l}
\textbf{class } \mathit{TeamDateBook} \\
\upharpoonright(\mathrm{INIT}, \mathit{AddMeeting}, \mathit{AddPrivateDate}, \mathit{RemoveMeeting}, \\
\quad \mathit{RemovePrivateDate}, \mathit{AddTeamMember}, \mathit{RemoveTeamMember}, \mathit{MoveMeeting}) \\
\\
\quad \mathit{dates} : \mathbb{P} \downarrow \mathit{Date}_{\copyright}; \\
\quad \mathit{team} : \mathbb{P}\ \mathit{TeamMember} \\
\quad \hline \\
\quad \forall\, pd : \mathit{PrivateDate} \mid pd \in \mathit{dates} \bullet pd.\mathit{owner} \in \mathit{team} \\
\quad \forall\, m : \mathit{Meeting} \mid m \in \mathit{dates} \bullet m.\mathit{moderator} \in \mathit{team} \\
\quad \forall\, m : \mathit{Meeting} \mid m \in \mathit{dates} \bullet m.\mathit{participants} \subseteq \mathit{team} \\
\quad \forall\, tm : \mathit{team} \bullet (tm.\mathit{moderates} \cup tm.\mathit{participates}) \subseteq \mathit{dates} \\
\quad \forall\, tm : \mathit{team} \bullet tm.\mathit{privateDates} \subseteq \mathit{dates} \\
\\
\quad \mathrm{INIT} \\
\quad \mathit{dates} = \varnothing \\
\quad \mathit{team} = \varnothing \\
\\
\quad \mathit{AddDate} \\
\quad \Delta(\mathit{dates}) \\
\quad \mathit{newdate}? : \downarrow \mathit{Date} \\
\quad \hline \\
\quad \mathit{newdate}? \notin \mathit{dates} \\
\quad \mathit{dates}' = \mathit{dates} \cup \{\mathit{newdate}?\} \\
\\
\quad \mathit{RemoveDate} \\
\quad \Delta(\mathit{dates}) \\
\quad \mathit{date}? : \downarrow \mathit{Date} \\
\quad \hline \\
\quad \mathit{date}? \in \mathit{dates} \\
\quad \mathit{dates}' = \mathit{dates} \setminus \{\mathit{date}?\} \\
\\
\quad \mathit{NewMeeting} \mathrel{\widehat=} [\, m? : \mathit{Meeting} \mid m?.\mathrm{INIT} \,] \bullet \\
\qquad\qquad m?.\mathit{SetDate} \land m?.\mathit{SetMeeting} \land m?.\mathit{CheckConflicts} \\
\quad \mathit{AddMeeting} \mathrel{\widehat=} \mathit{NewMeeting} \land \mathit{AddDate}[m?/\mathit{newdate}?] \\
\quad \mathit{MoveMeeting} \mathrel{\widehat=} [\, m? : \mathit{Meeting} \mid m? \in \mathit{dates} \,] \bullet m?.\mathit{MoveMeeting} \land m?.\mathit{CheckConflicts} \\
\quad \mathit{NewPrivateDate} \mathrel{\widehat=} [\, pd? : \mathit{PrivateDate} \mid pd?.\mathrm{INIT} \,] \bullet pd?.\mathit{SetDate} \land pd?.\mathit{SetPrivateDate} \\
\quad \mathit{AddPrivateDate} \mathrel{\widehat=} \mathit{NewPrivateDate} \land \mathit{AddDate}[pd?/\mathit{newdate}?] \\
\quad \mathit{MovePrivateDate} \mathrel{\widehat=} [\, pd? : \mathit{PrivateDate} \mid pd? \in \mathit{dates} \,] \bullet pd?.\mathit{Move} \\
\quad \mathit{RemovePrivateDate} \mathrel{\widehat=} ([\, pd? : \mathit{PrivateDate} \mid pd? \in \mathit{dates} \,] \bullet pd?.\mathit{RemoveFromOwner}) \\
\qquad\qquad \land \mathit{RemoveDate}[pd?/\mathit{date}?] \\
\quad \mathit{RemoveMeeting} \mathrel{\widehat=} ([\, m? : \mathit{Meeting} \mid m? \in \mathit{dates} \,] \bullet m?.\mathit{RemoveParticipants}) \\
\qquad\qquad \land \mathit{RemoveDate}[m?/\mathit{date}?] \\
\quad \mathit{AddTeamMember} \mathrel{\widehat=} [\, \mathrm{true} \,] \\
\quad \mathit{RemoveTeamMember} \mathrel{\widehat=} [\, \mathrm{true} \,]
\end{array}
$$

Bibliography

[ACNL97] Paulo Alencar, Donald Cowan, Torsten Nelson, and Carlos J. Lucena. Viewpoints as an evolutionary approach to software system maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM'97)*, pages 260–267, 1997.

[ASW93] Robert S. Arnold, Malcolm Slovin, and Norman Wilde. Do design records really benefit software maintenance. In *Proceedings of the Conference on Software Maintenance (CSM'93)*, pages 234–243, 1993.

[BA96a] Shawn A. Bohner and Robert S. Arnold. An introduction to software change impact analysis. In *[BA96b]*, pages 1–26. 1996.

[BA96b] Shawn A. Bohner and Robert S. Arnold, editors. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[Bal98] R. Balzer et. al. International Workshop on the Principles of Software Evolution (IWPSE'98), 1998.

[Bas90] R. V. Basili. Viewing maintenance as reuse-oriented software development. *IEEE Software*, 7(1):19–25, 1990.

[Bax92] Ira D. Baxter. Design maintenance systems. *Communications of the ACM*, 35(4):73–89, 1992.

[BB92a] J. P. Bowen and P. T. Breuer. Decompilation. In *[van93]*, pages 131–138. 1992.

[BB92b] P. T. Breuer and J. P. Bowen. Decompilation: the enumeration of types and grammars. Technical Report PRG-TR-11-92, Oxford University Computing Laboratory, 11 Keble Road, Oxford, UK, 1992.

[BBDD97] L. Briand, C. Bunse, J. Daly, and C. Differding. An experimental comparison of the maintainability of object-oriented and structured design documents. In *Proceedings of the International Conference on Software Maintenance (ICSM'97)*, pages 130–138, 1997.

[BCD95] Alan W. Brown, Alan M. Christie, and Susan A. Dart. An examination of software maintenance practices in a u.s. government organization. *Journal of Software Maintenance: Research and Practice*, 7:223–238, 1995.

[Ben97] Keith H. Bennett. Software maintenance: A tutorial. In *[DT97]*, pages 289–303. 1997.

[BG96] David W. Binkley and Keith Brian Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[Blu95] Bruce I. Blum. Resolving the software maintenance paradox. *Journal of Software Maintenance: Research and Practice*, 7:3–26, 1995.

[Boe76] Barry W. Boehm. Software engineering. *IEEE Transactions on Computers*, C-25(12):1226–1241, 1976.

[Boe88] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Software*, 31(5), 1988.

[Boh91] Shawn A. Bohner. Software change impact analysis for design evolution. In *Proceedings of the Conference on Software Maintenance (CSM'91)*, pages 292–301, 1991.

[Boh96] Shawn A. Bohner. Impact analysis in the software change process: A year 2000 perspective. In *Proceedings of the International Conference on Software Maintenance (ICSM'96)*, pages 42–51, 1996.

[BP97] Ira D. Baxter and Christopher W. Pidgeon. Software change through design maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM'97)*, pages 250–259, 1997.

[BR00] Keith H. Bennett and Vaclav T. Rajlich. Software maintenance and evolution: A roadmap. In *[Fin00]*, pages 73–87. 2000.

[Bro87] Frederick Brooks. No silver bullet – essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[CC90] E. Chikofski and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):24–27, 1990.

[CDD+90] David A. Carrington, David Duke, Roger Duke, Paul King, Gordon A. Rose, and Graeme 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.

[CF97] Cristina Cifuentes and Antoine Fraboulet. Intraprocedural static slicing of binary executables. In *Proceedings of the International Conference on Software Maintenance (ICSM'97)*, pages 188–195, 1997.

[CHOT99] Siobhan Clarke, William Harrison, Harold Ossher, and Peri Tarr. Subject-oriented design: Towards improved alignment of requirements, design and code. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, 1999.

[CLV92] A. Cimitile, F. Lanubile, and G. Visaggio. Traceability based on design decisions. In *Proceedings of the Conference on Software Maintenance (CSM'92)*, pages 309–317, 1992.

[CM94] Miriam A. M. Capretz and Malcolm Munro. Software configuration management issues in the maintenance of existing systems. *Journal of Software Maintenance: Research and Practice*, 6:1–14, 1994.

[CvM93] Ronald T. Crocker and Anneliese von Mayrhauser. Maintenance support needs for object-oriented software. In *Proceedings of the 17th Annual International Computer Software & Applications Conference (COMPSAC'93)*, 1993.

[CW98] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.

[CYL96] William C. Chu, Hongji Yang, and Paul Luker. A formal method for software maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM'96)*, pages 206–216, 1996.

[DBM+95] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood. The effect on inheritance on the maintainability of object-oriented software: An empirical study. In *Proceedings of the International Conference on Software Maintenance (ICSM'95)*, pages 20–29, 1995.

[DBSB91] P. Devanbu, R. J. Brachman, P. G. Selfridge, and B. W. Ballard. LaSSIE: A knowledge-based software information system. *Communications of the ACM*, 34(5):34–49, 1991.

[Dil94] Antoni Diller. *Z - An Introduction to Formal Methods*. John Wiley & Sons Inc., 1994.

[DKRS91] 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.

[DP98] Ralf Doemges and Klaus Pohl. Adapting tracability environments to project-specific need. *Communications of the ACM*, 41(12):54–62, December 1998.

[DRS94] R. Duke, G. Rose, and G. Smith. Object-z: A specification language advocated for the description of standards. Technical report 94-45, Software Verification Research Centre, School of Information Technology, The University of Queensland, Brisbane 4072. Australia, December 1994.

[DT97] Merlin Dorfman and Richard H. Thayer, editors. *Software Engineering*. IEEE Computer Society Press, 1997.

[EDF96] Earl F. Ecklund, Lois M. L. Delcambre, and Michael J. Freiling. Change cases: Use cases that identify future requirements. In *Proceedings of the 11th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'96)*, pages 342–358, 1996.

[FB97] Xavier Franch and Pere Botella. Supporting software maintenance with non-functional information. In *Proceedings of the First EUROMICRO Conference on Software Maintenance and Reengineeering (CSMR'97)*, pages 10–16, 1997.

[FH76] R. K. Fjelstad and W. T. Hamlen. Application program maintenance study - report to our respondents. In *Proceedings of GUIDE 48, The Guide Corporation, Philadelphia*, 1976.

[FHLS97] Gary Froehlich, H. James Hoover, Ling Liu, and Paul Sorensen. Hooking into object-oriented application frameworks. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, pages 491–501, 1997.

[Fin00] Antony Finkelstein, editor. *The Future of Software Engineering.* ACM Press, 2000.

[Fug00] Alfonso Fuggetta. Software process: A roadmap. In *[Fin00]*, pages 27–34. 2000.

[FW81] D. P. Freedman and G. M. Weinberg. A checklist for potential side effects of a maintenance change. In G. Parikh, editor, *Techniques of Program and System Maintenance*, pages 93–100. 1981.

[GF94] Orlena C. Z. Gotel and Anthony C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the International Conference on Requirements Engineering (ICRE'94)*, pages 94–101, 1994.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GHS92] Rajiv Gupta, Mary Jean Harrold, and Mary Lou Soffa. An approach to regression testing using slicing. In *Proceedings of the Conference on Software Maintenance (CSM'92)*, pages 299–308, 1992.

[GL91] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[Got93] O. Gotel. Modelling the contribution structure underlying requirements. In *Workshop on Requirements Engineering: Foundations for Software Quality*, 1993.

[Han97] Jun Han. Designing for increased software maintainability. In *Proceedings of the International Conference on Software Maintenance (ICSM'97)*, pages 278–286, 1997.

[HGK+95] P. Hsia, A. Gupta, C. Kung, J. Peng, and S. Liu. A study on the effect of architecture on maintainability of object-oriented systems. In *Proceedings of the International Conference on Software Maintenance (ICSM'95)*, pages 4–11, 1995.

[HKOS96] W. H. Harrison, H. Kilov, H. L. Ossher, and I. Simmonds. From dynamic supertypes to subjects: A natural way to specify and develop systmes. *IBM Systems Journal*, 35(2):244–256, 1996.

[HMF92] Mary Jean Harrold, John D. McGregor, and Kevin Fitzpatrick. Incremental testing of object-oriented class structures. In *Proceedings of the 14th International Conference on Software Engineering (ICSE'92)*, pages 68 – 80, 1992.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the 8th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'93)*, 1993.

[HPR89] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, 1989.

[HRB90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependece graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):35–46, 1990.

[HSE90] B. Henderson-Sellers and J.M. Edwards. The object-oriented systems life cycle. *Communications of the ACM*, 33(9):142 – 159, September 1990.

[HTS96] H. Huang, Wei-Tek Tsai, and S. Subramanian. Generalized program slicing for software maintenance. In *Proceedings of the 8th International Conference on Software Engineering and Knowledge Engineering (SEKE'96)*, pages 261–268, 1996.

[Hus99] Heinrich Hussmann. Meetingorganizer: A comprehensive but small example for the introduction of oo concepts. In *OOPSLA'99 Workshop: Quest for Effective Classroom Examples*, 1999.

[IEE90] IEEE. Standard computer dictionary. The Institute of Electrical and Electronic Engineers, New York, 1990.

[IEE91] IEEE. IEEE standard glossary of software engineering technology. IEEE Std. 610.12-1990, 1991.

[Jar98] Mathias Jarke. Requirements tracing. *Communications of the ACM*, 41(12):32–36, December 1998.

[JE94] Paul C. Jorgensen and Carl Erickson. Object-oriented integration testing. *Communications of the ACM*, 37(9):30–38, September 1994.

[Jøg95] Magne Jøgensen. An empirical study of software maintenance tasks. *Journal of Software Maintenance: Research and Practice*, 7:27–48, 1995.

[Jon94] Capers Jones. Gaps in the object-oriented paradigm. *IEEE Computer*, 27(6):90–91, June 1994.

[Kam95] Mariam Kamkar. An overview and comparative classification of program slicing techniques. *Journal of Systems and Software*, 31:197–214, 1995.

[KGH+94] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. Change impact identification in object-oriented software maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM'94)*, pages 202–211, 1994.

[KGH+95] David Kung, Jerry Gao, Pei Hsia, Jeremy Lin, and Yasufumi Toyoshima. Class firewall, test order, and regression testing of object-oriented programs. *Journal of Object-Oriented Programming*, pages 51–65, May 1995.

[KGH+96] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. On regression testing of object-oriented programs. *Journal of Systems and Software*, 32(1):21–40, January 1996.

[KH98] Hsiang-Jui Kung and Cheng Hsu. Software maintenance life cycle. In *Proceedings of the International Conference on Software Maintenance (ICSM'98)*, 1998.

[KHJ97] G. Aditya Kiran, S. Haripriya, and Pankaj Jalote. Effect of object orientation on maintainability of software. In *Proceedings of the International Conference on Software Maintenance (ICSM'97)*, pages 114–121, 1997.

[Kic96] Gregor Kiczales. Beyond the black box: Open implementation. *IEEE Software*, 13(1):8–11, January 1996.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Lointier, and J. Irwin. Aspect-oriented programming. Parc technical report spl97-008 p97 10042, Xerox Palo Alot Research Center, February 1997.

[KvN+99] B. A. Kitchenham, A. von Mayrhauser, F. Niessink, N. Schneidewind, J. Singer, G. H. Travassos, S. Takada, R. Vehvilainen, and H. Yang. Towards and ontology of maintenance. Technical Report TR99-03, Department of Computer Science, University of Keele, Keele, Staffordshire, ST5 5BG, UK, 1999.

[Lee91] Jintae Lee. Extending the potts and bruns model for recording design rationale. In *Proceedings of the 13th International Conference on Software Engineering (ICSE'91)*, pages 114–125, 1991.

[Lee97] Jintae Lee. Design rationale systems: Unterstanding the issues. *IEEE Expert*, pages 78–85, May 1997.

[Leh80] Meir M. Lehman. Programs, life cycles and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060 – 1076, September 1980.

[Leh91] Franz Lehner. *Software Wartung*. Carl Hanser Verlag, M nchen, Wien, 1991.

[Leh98] Meir M. Lehman. Software's future: Managing evolution. *IEEE Software*, 15(1):40–44, January 1998.

[LH91] Kevin Lano and Howard Haughton. A specification-based approach to maintenance. *Journal of Software Maintenance: Research and Practice*, 3:193–213, 1991.

[LH96] Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *Proceedings of the 18th International Conference on Software Engineering (ICSE'96)*, pages 495 – 505, 1996.

[LH98] Donglin Liang and Marry Jean Harrold. Slicing objects using system dependence graphs. In *Proceedings of the International Conference on Software Maintenance (ICSM'98)*, 1998.

[Lie95] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, 1995.

[LMR91] Moises Lejter, Scott Meyers, and Steven P. Reiss. Support for maintaining object-oriented programs. In *Proceedings of the Conference on Software Maintenance (CSM'91)*, pages 171–178, 1991.

[LMR92] Moises Lejter, Scott Meyers, and Steven P. Reiss. Support for maintaining object-oriented programs. *IEEE Transactions on Software Engineering*, 18(12):1045–1052, December 1992.

[LPR98] Meir M. Lehman, D. E. Perry, and J. F. Ramil. Implications of evolutions metrics on software maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM'98)*, 1998.

[LR98] Mikael Lindvall and Magnus Runesson. The visibility of maintenance in object models: An empirical study. In *Proceedings of the International Conference on Software Maintenance (ICSM'98)*, 1998.

[LSM97] Carine Lucas, Patrick Steyaert, and Kim Mens. Managing software evolution through reuse contracts. In *Proceedings of the First EUROMICRO Conference on Software Maintenance and Reengineeering (CSMR'97)*, pages 165–168, 1997.

[Luq90] Luqi. A Graph Model for Software Evolution. *IEEE Transactions on Software Engineering*, pages 917–927, 1990.

[LV95] Filippo Lanubile and Giuseppe Visaggio. Decision-driven maintenance. *Journal of Software Maintenance: Research and Practice*, 7:91–115, 1995.

[Mil83] Ali Mili. A relational approach to the design of deterministic programs. *Acta Informatica*, 20(4):315–328, 1983.

[MK93] Roland Mittermeir and Klaus Kienzl. Intra-object schemas to enhance adaptive software maintenance. In *Austro-Hungarian Software Engineering Seminar*, 1993.

[MKH97] Saeko Matsuura, Hironobu Kuruma, and Shinichi Honiden. Eva: A flexible programming method for evolving systems.

IEEE Transactions on Software Engineering, 23(4):296–313, May 1997.

[MPRR98] Roland T. Mittermeir, Helfried Pirker, and Dominik Rauner-Reithmayer. Object evolution by model evolution. In *Proceedings of the Second EUROMICRO Conference on Software Maintenance and Reengineeering (CSMR'98)*, pages 216–219, 1998.

[MSPD95] Simon Monk, Ian Sommerville, Jean Michel Pendaries, and Bernard Durin. Supporting design rationale f r system evolution. In *Proceedings of the 5th European Software Engineering Conference (ESEC'95)*, pages 397–323, 1995.

[OA93] Tomohiro Oda and Keijiro Araki. Specification slicing in formal methods of software development. In *Proceedings of the 17th Annual International Computer Software & Applications Conference (COMPSAC'93)*, pages 313–319, 1993.

[OHBS94] Harold Ossher, William Harrison, Frank Budinsky, and Ian Simmonds. Subject-oriented programming: Supporting decentralized development of objects. In *Proceedings of the* 7^{th} *IBM Conference on Ojbect-Oriented Technology*, 1994.

[OKK+96] Harold Ossher, Mathew Kaplan, Alexander Katz, William Harrison, and Vincent Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems*, 2(3):179–202, 1996.

[Par86] Girish Parikh. *Handbook of Software Maintenance*. John Wiley & Sons, New York, 1986.

[Par94] D.L. Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering (ICSE'94)*, pages 279 – 287, 1994.

[PB88] Colin Potts and Glenn Bruns. Recording the reasons for design decisions. In *Proceedings of the 10th International Conference on Software Engineering (ICSE'88)*, pages 418–427, 1988.

[PJ98] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Proceedings of the 22nd Annual International Computer Software & Applications Conference (COMPSAC'98)*, pages 9–15, 1998.

[PK98] Dewayne E. Perry and Takuya Katayama. Panel: Critical issues in software evolution. In *Proc. 20th International Conference on Software Engineering, Vol. II*, page 12, 1998.

[PM98] Helfried Pirker and Roland T. Mittermeir. Internal service channels - principles and limits. In *Proceedings of the International Workshop on the Principles of Software Evolution (IWPSE'98)*, pages 63–67, 1998.

[PMRR98] Helfried Pirker, Roland T. Mittermeir, and Dominik Rauner-Reithmayer. Service channels - purpose and tradeoffs. In *Proceedings of the 22nd Annual International Computer Software & Applications Conference (COMPSAC'98)*, pages 204–211, 1998.

[Pos94] Robert M. Poston. Automated testing from object models. *Communications of the ACM*, 37(9):48–58, September 1994.

[Raj96] Vaclav Rajlich. MSE: A Methodology for Software Evolution. *Journal of Software Maintenance: Research and Practice*, 9:103–124, July 1996.

[Raj97] Vaclav Rajlich. A model for change propagation based on graph rewriting. In *Proceedings of the International Conference on Software Maintenance (ICSM'97)*, 1997.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modelling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[Roy70] W. W. Royce. Managing the develpment of large software systems: concepts and techniques. In *Proceedings IEEE WESCON*, pages 1–9, 1970.

[Sak94] Sachidanandam Sakthivel. A decision model to chose between software maintenance and software redevelopment. *Journal of Software Maintenance: Research and Practice*, 6:121–143, 1994.

[SC96] P. Stocks and D. Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.

[Sin98] Janice Singer. Practices of software maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM'98)*, 1998.

[SMC74] W. P. Stevens, G. J. Myers, and L. Constantine. Structured design. *IBM Systems Journal*, 13(2), 1974.

[Sne91] Harry Sneed. *Software Wartung*. Verlagsgesellschaft Rudolf M ller GmbH, K ln, 1991.

[Spi89] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, 1989.

[TB99a] Lance Tokuda and Don Batory. Automating three modes of evolution for object-oriented architechtures. In *Proc. 5th Conference on Object-Oriented Technologies (COOTS'99)*, 1999.

[TB99b] Lance Tokuda and Don Batory. Evolving Object-Oriented Architectures with Refactorings. *Automated Software Engineering*, 1999.

[Tip94] Frank Tip. A survey of program slicing techniques. Cs-r9438 1994, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1994.

[TN97] Eirik Tryggeseth and Oystein Nytro. Dynamic tracability links supported by a system architecture description. In *Proceedings of the International Conference on Software Maintenance (ICSM'97)*, pages 180–187, 1997.

[TT92] Tetsuo Tamai and Yohsuke Torimitsu. Software lifetime and its evolution process over generations. In *Proceedings of the Conference on Software Maintenance (CSM'92)*, pages 63–69, 1992.

[van93] H. van Zuylen. *The REDO Compendium: Reverse Engineering for Software Maintenance*. John Wiley, 1993.

[Vli98] John Vlissides. Subject-oriented design. *C++ Report*, 10(2), 1998.

[Vrb97] Michael P. Vrbicky. Rekonstruktion von Structure Charts und intermodularem Datenfluß aus C Quellcode. Master's thesis, Universität Klagenfurt, 1997.

[WCMH91] Norman Wilde, Allen Chapman, Paul Mathews, and Ross Huitt. Describing object-oriented software: What maintainers need to know. Technical Report SERC-TR-54-F, Software Engineering Research Center, University of Florida, 1991.

[Wei84] Mark Weiser. Programm slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[WH91] Norman Wilde and Ross Huitt. Maintenance support for object oriented programs. In *Proceedings of the Conference on Software Maintenance (CSM'91)*, pages 162–170, 1991.

[WH92] Norman Wilde and Ross Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, 18(12):1038–1044, December 1992.

[WTCR96] Yamin Wang, Wei-Tek Tsai, Xiaping Chen, and Sanjai Rayadurgam. The role of program slicing in ripple effect analysis. In *Proceedings of the 8th International Conference on Software Engineering and Knowledge Engineering (SEKE'96)*, pages 369–376, 1996.

[Yip95] Stephen W. L. Yip. Software maintenance in hong kong. In *Proceedings of the International Conference on Software Maintenance (ICSM'95)*, pages 88–97, 1995.

[YNTL88] S. S. Yau, R. A. Nicholl, J. J. Tsai, and S. Liu. An Integrated Life-Cycle Model for Software Maintenance. *IEEE Transactions on Software Engineering*, 15(7):58–95, 1988.