INFORMATICA, 1999, Vol. 10, No. 2, 203–218

© 1999 *Institute of Mathematics and Informatics, Vilnius*

Semantic Integrity of Switching Sections with Contracts: Discussion of a Case Study

Eivind J. NORDBY, Martin BLOM
Department of Computer Science, Karlstad University
651 88 Karlstad Sweden
e-mail: Eivind.Nordby@kau.se
Martin.Blom@kau.se

Received: January 1999

Abstract. We have studied the design documentation for two industrial software modules to see if they apply ideas corresponding to contracts, as introduced by Bertrand Meyer, either in an intuitive or in a formal way. They did not, and we identified this fact to be a potential risk factor. This paper presents one of the modules studied, consisting of a sequence of switching sections. Starting from this case study, the paper also discusses how switching sections in general can be designed using contracts in order to increase the semantic integrity of the module as a whole.

Key words: branch, contract, postcondition, precondition, selection, semantic integrity, semantic gap, software design, software quality, switch.

1. Introduction

At Karlstad University we appreciate the trends in modern software products towards more reactive and module based software components. Such software puts high requirements on the quality of the module design and on both the use and the implementation of such modules. The absence of a proper design will frequently lead to integration problems. We will use the term *semantic integrity* to describe the corresponding quality aspect. This is more fully explained in Section 2. We have a general impression that there is a gap in the documentation of the semantic aspects of much modern software.

To confirm or contradict this impression, we have started to study to which extent sound modern software design principles are applied in the software industry today. The objective for our study is to eventually propose methods, which promote the application of such principles in order to produce better quality software faster. To be useful these methods must be pragmatic enough to be applied by a great variety of software designers and developers with a varying degree of formal education. We also appreciate that new software functions are required at an increasing speed today, and any development method proposed must consider the shorter cycle times. The formalism should therefore be kept on a low to moderate level.

Within this research framework, in 1997 the authors of this paper studied the documentation for selected modules from two recent industrial software development projects.

Both resulted in high quality products sold and in production today. This work was part of a Master's thesis in Computer Science, presented in (Blom, 1997).

The objective of that study was to see if the design of the modules, as described in the related documentation, followed established design guidelines. The guidelines studied are extracted from current software engineering literature and documented in (Blom, 1997). In particular we wanted to see if the documentation reflected and rendered a clear understanding of the responsibilities assigned to the different modules studied. The guidelines for this study were taken from (Meyer and Bertrand, 1988) and the concept of "Programming by contract" introduced there, but was not limited to object oriented systems. We did not expect the contract concept to be applied explicitly, but wanted to see if the major ideas behind this concept were applied, either intuitively or explicitly.

One of the products from our study forms the basis for this paper. It was studied in more detail than the other one. The specific product area, as such, is not relevant to the presentation and discussion in this paper and will not be referenced. The module studied is presented shortly in Section 4 of this paper.

A full description and discussion of the module and its documentation can be found in (Blom, 1997). It shows that the specification of the module, as a whole, is well understood and well documented. Also, the implementation of the individual parts of the module seems to be well done, although that aspect was not a primary focus of the study. However, when it comes to the documentation of the module integration aspects, i.e., how the individual parts of the module cooperate to meet the overall module specification, the study found a lack of complete and consistent documentation.

In this paper, we evaluate this case with regard to its semantic integrity. We then extend the design to show how the module integration could be done and documented to achieve an even higher level of software quality. In particular, we look at how the concept of contracts can be used to manage the semantics of a branching construction and we then generalize these principles.

The rest of this paper is organized as follows. Sections 2 and 3 present the terms and concepts used and delimit the scope of this paper. The case study is presented in Section 4, and Section 5 discusses the areas where the design documentation does not support the module's semantic integrity, especially in relation to the switching sections used. Section 6 proposes a design methodology using contracts. It integrates the implementation of the switching sections in the overall module design and helps to assure the semantic integrity of the module as a whole. Section 7 then discusses the merits of this approach. Finally, Sections 8 and 9 conclude the paper and present some areas for further study.

2. The Terms and Concepts Used

This section defines most of the terms used in this paper. The first part describes some general terms and their use. The second part presents the term *semantic integrity* discussed in this paper and related concepts in some detail. The third part introduces the terms and problems directly related to the case study discussed in this paper.

2.1. *General Terms*

This subsection defines the meaning of some terms used in the context of this paper. Since the paper is limited in scope, some terms will be used in a narrower, more specific way than what may be the case in other contexts. The scope of the paper is discussed in Section 3.

Our study involves *algorithms* and their properties. An algorithm describes the steps of a process and shall meet some requirements expressed in a specification. An algorithm describes an implementation and it may be expressed as code.

We use the term *routine* in the same sense as Meyer and Bertrand (1988). A routine may be is called a subroutine, a function, a procedure or a method in some programming languages.

The term *assertion* is used in its usual way. An assertion is a logical statement about the state of a process at a specific point of an algorithm. This specific point will always be between two steps of the algorithm. Frequently, an assertion expresses a *condition*. The assertion is said to hold, or the condition to be satisfied, if it evaluates to true at the given point.

Preconditions and *postconditions* play a central role in assuring semantic integrity. These terms are defined by Hoare (1972). They are expressed as assertions. They are frequently referred to by research papers and in other academic contexts but are not so frequently used in industry. An algorithm or a part of an algorithm, often implemented as a routine, may be enclosed by a matching pair consisting of a precondition and a postcondition. The precondition expresses a condition that is known to hold before the algorithm is executed. The postcondition expresses a condition that shall hold after the execution of the algorithm. Invariants, although representing another important aspect related to semantic integrity, are not discussed in this paper.

The concept of *contracts* was introduced by Bertrand Meyer (1988). A contract is expressed using preconditions and postconditions. A contract relates to a specific *service* and is an agreement between the *supplier* of the service and its *clients*. Normally, the supplier is implemented as a routine. The clients are then the calling routines.

A contract works as follows. A client requesting a service from a supplier should assure that the corresponding preconditions hold before the request is issued. If, and only if, this is the case, the supplier guarantees that the corresponding postconditions hold after the execution of that service. The contract explicitly avoids describing the situation at the end of the execution of the service in the eventuality that the requested precondition should not hold when the service is requested.

A *code section*, or just *section* for short, is some contiguous part of a computer program. Sometimes a complete software routine can be substituted by a section of code. In this paper, we will allow the use of contracts for code sections as well as for complete routines. In such a case, the "client" of the section will be the surrounding code. In this case, the client and the supplier are part of the same routine.

2.2. *Semantic Integrity*

We use the term *semantic integrity* to denote, in addition to the complete definition of the properties of a service, also the mutual respect for these properties from both the supplier itself and from its environment. If the properties of a service are defined using a contract, then semantic integrity is conserved if the contract is consistent and violated by neither clients nor the supplier. To verify this it is necessary to be able to match both invocations and the implementation of the service against the conditions of the contract.

As it turns out, to verify this implies a study of the implementation of each of the clients. The parts of the client algorithms are connected through assertions. A given pair of assertions constitutes a contract for the enclosed code section. For the semantic integrity of the total system to be maintained, this contract must be compatible with the contracts of the routines called by this code section, as discussed below. A software system where each module can be shown to maintain its semantic integrity has reached a high degree of quality in its design and implementation.

There are two typical and frequent ways to violate the semantic integrity of a piece of software. One is that the implementation of the supplier does not follow its own specifications. The other is that a client uses the service in a way, which is not consistent with these specifications. In some cases, the reason for these discrepancies is that the specifications themselves are incomplete or unclear. The concept of semantic integrity includes a requirement that there be a valid specification for the service, and that this specification is given independently of the implementation of that service. In the case study explored in this paper, we found that such a definition did not always exist or was not always well understood. This fact represents a threat to the quality of the software being designed.

A *semantic gap* appears when the relationships between different parts of a system are not defined or documented. An example of a semantic gap appeared in the case study. This is presented during the discussion in Section 5 below. Semantic gaps are another source of violation of a software system's semantic integrity.

The term *quality* in this paper is used in a limited sense to only include aspects of design and implementation of modules and their relation to semantic integrity. A high level of quality in the context of this paper means that the specifications of a module or code section are correct and complete, and that the use and implementation of this module or section preserve the semantic integrity.

2.3. *Terms Related to the Case Study*

The following definitions introduce the particular focus for this paper. We will use the term *switching section* for a special kind of code section. The first part of a switching section is called a *switch*. It performs a test, and according to the result of the test, one of the following parts, called *branches*, is executed. All the branches converge to a common continuation point, the *convergence point*. A switching section is commonly implemented as a possibly nested if-then-else statement, a switch or a case statement. It can be illustrated by Fig. 1.

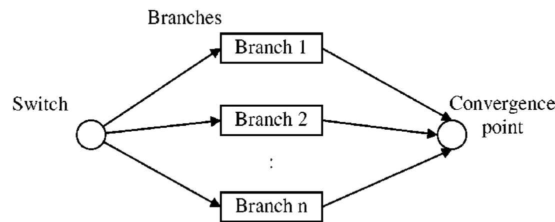

Fig. 1. The structure of a switching section.

The focus on switching sections for this paper was chosen because the case study described consists of a sequence of such sections. In the study, we found a lack of semantic specification, called semantic gaps, both between and inside these sections. These semantic gaps represent a potential source of semantic errors when the system continues to evolve.

3. The Scope of this Paper

This paper focuses on software quality in general and the design aspects for software modules in particular. It concentrates on those aspects, which have a bearing on a module's semantic integrity, in the sense defined above. In general, for a product to be usable, it must both "do the right things" and "do things right", i.e., it must be both useful and correct. The former aspect has to do with the validation of the product and the latter with its verification.

"Doing the right things" concern the specifications for a system and is an issue for system requirements engineers. It involves usability, fitness for use, user friendliness and other aspects related to user requirements. These issues are very important in software engineering, but lie outside the scope of this paper. The same applies to aspects like iterative specifications, prototyping or stepwise refinements, related to the software development process.

This paper is limited to studying software design aspects and to seeing how to "do things right". We assume that there is a valid, although possibly volatile, system specification and study the semantic integrity of the system in relation to this specification. One could say that semantic integrity is primarily related to software verification. It is far less related to software validation. It is worth emphasizing here that semantic integrity is only one of many quality criteria for a software system.

Our study relates to the correctness of individual software modules. We study how to define individual modules and how to make their implementation conformant with their definition. In particular, this paper focuses on switching sections. We discuss how the requirements for the individual branches of a switching section can be defined in such a way that they support the semantic requirements from the context.

This paper is based on an industrial case study. The objective of the case study was to investigate the description of a module presented by an industry partner. We wanted to

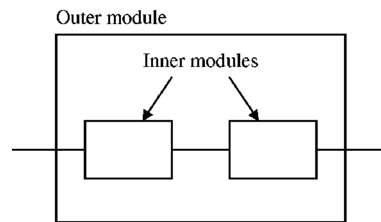

Fig. 2. The relationship between outer and inner modules.

see if the description constituted a complete base for both the use and the implementation of the module. To use the module one should not need to go to the implementation to see what was actually done. The module specification should tell the whole story. Similarly, to implement the module, one should not need to go to its client modules to investigate which assumptions are made about it. Again, the module specification should give the complete description against which the implementation should be checked. We did not expect the interface to be described as a contract, but we wanted to see if many of the ideas from the contracts concept were used.

We intended to limit our study to the module specification and to compare this specification to a number of design criteria extracted from Meyer (1988). The module studied was selected by the industry partner. As it turned out, for the case referred to in this paper, in addition to the design documentation of the main module, we also received a description of the module's implementation. This includes the specifications of the next lower level of modules used to implement the main module. We thus had two levels of module description, one used by the implementation of the other. Actually, this allowed us to draw more conclusions than we had planned to do.

4. Presentation of the Case Study

This section presents the structure and the documentation of the case study. In the next section, we discuss the system solution, and in Section 6 we propose some additions to it. A full description of the system and the evaluation is given in (Blom, 1997).

In the case study there are two levels of abstraction. The study involved one software module which was implemented using several other, smaller modules. We will use the general attributes *outer* and *inner* to distinguish them from each other. Fig. 2 shows the relationship between the outer and inner modules.

The implementation structure of the outer module is illustrated in Fig. 3. It consists of a number of chained switching sections and a data structure. The convergence point of one switching section is the switch of the following one. Each one of the branches is a separately developed module and represents a certain service. In a certain sense, the solution is dynamic. It is possible for the customer to add new branches for a switch later, after the system is set in operation. This allows new services to be added dynamically. The outer module also contains a local data structure. This data structure is directly accessible to all the inner modules and acts as a common global data structure to them.

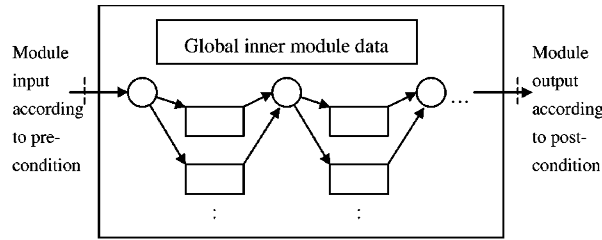

Fig. 3. The outer module and its overall implementation structure.

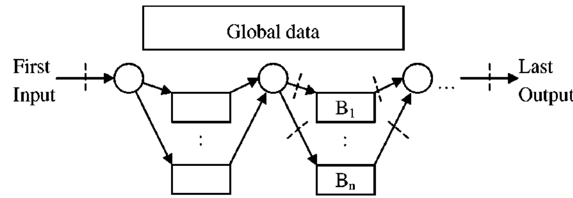

Fig. 4. The inner modules and their interrelationship.

The outer module as a whole and its functionality are well described in a system requirements document. The input to the module and the output from it are well specified, as well as the conditions governing when it can be called from its environment. These requirements are symbolized in Fig. 3 by the dashed lines, labeled precondition and postcondition respectively, crossing the arrows going into and out of the outer module.

Fig. 4 shows the chain of switching sections, which constitute the inner parts of the outer module. The input requirements for the outer module also serve as input requirements for the first switching section. Similarly, the output requirements for the outer module serve as output requirements for the last switching section.

In the case study, the individual inner modules were documented independently of each other. Their functionality was described in terms of their preconditions and postconditions. This is symbolized in Fig. 4 by dashed lines crossing the arrows entering and leaving the inner modules, where the modules labeled B_1 and B_n serve as examples.

However, when it comes to the interrelationship between the inner modules, we found a gap in the documentation. All the switching sections, seen as a whole, take First Input as input and produce Last Output as output. Each one of the individual inner modules participate in this overall operation. However, there was no apparent connection between the task performed by the individual inner modules and this overall role. This aspect will be further discussed in the next section.

In the rest of this paper, rather than considering the outer module as such, we will discuss its implementation, which is the sequence of switching sections, and the branches of the switching sections. The switches and branches are the inner modules. We thus have a sequential implementation with a global data structure, which will be involved in the pre- and postconditions of the inner modules. We also have the specifications for the input of the first part and for the output of the last part, as illustrated in Fig. 4.

5. Some Comments on the Solution from the Case Study in Relation to its Semantic Integrity

Assume that the individual modules, such as B_1 through B_n in Fig. 4, are themselves well specified. That is not enough to show how they are related to the First Input condition or how they contribute to meeting the Last Output condition. This is because there are several semantic aspects involved.

The outer module has both a specification aspect and an implementation aspect. Similarly, each of the inner modules also has a specification aspect and an implementation aspect. The relationship between these four aspects is illustrated in Fig. 5. In the case study, the specification of the outer module was well documented but we could not find any documentation, such as for example A, B and C in Fig. 5, for its implementation. This is an example of a semantic gap as mentioned in Subsection 2.2.

For the inner modules, the specification was documented using preconditions and postconditions for each individual module, such as X and Y, but these were not explicitly related to the outer module. The dependency between the inner modules and the surroundings was only implicitly present in the head of the designer and will be lost when a new person takes charge of developing the system further. In this context, it is not important if the inner modules are implemented as inline code sections or as separate modules. In the case study, they were implemented as separate modules. The fourth aspect, the implementation of the inner modules, was not studied by us.

There are two kinds of dependency between the inner modules and their environment. On the one hand, the switching sections to which the inner modules belong are chained together, each one depending on the result of the previous one in the chain. This defines the pre- and postconditions, such as A and C of Fig.5, for each switching section and introduces a *progress dependency* on the inner modules. On the other hand, each branch of a switching section also depends on the outcome of the switch condition. This introduces a *branch dependency* on the inner modules, such as B and C of Fig. 5. Obviously, the pre- and postconditions, such as X and Y, of the inner modules must satisfy the requirements B and C from the surrounding branch. In many cases X will be the same as B and Y the same as C, but, especially when modules are reused, that does not need to be the case.

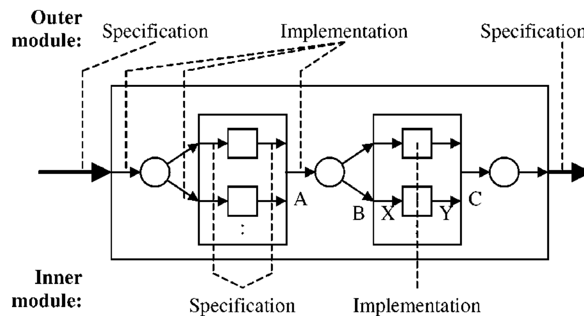

Fig. 5. The specification and implementation aspects of the modules.

Actually, A module, which is reused, already has its pre- and postconditions defined. A sufficient condition then, is that the reused, inner module is "strong enough", as described in Section 6 below. With reference to Fig. 5, the pre- and postconditions X and Y must at least satisfy the requirements defined by B and C. These, in turn, are derived from the position of the switching section in the outer module.

One reason for the semantic gap, which we identified between the specifications of the outer and inner modules, may be that the two-layer architecture of the outer module does not appear clearly. The individual inner modules are one semantic level removed from the outer module and this may not be obvious to the designer.

The rest of this paper will focus on how to identify and document the semantic requirements on the inner modules and to fill the semantic gap. The objective is to achieve semantic integrity for the outer module.

6. A Proposed Design Structure and Development Method for a Module Containing a Switching Section

In the present section we try to generalize the lessons learned from the previous study. We look at a general situation similar to the one studied, e.g., an outer module whose implementation contains at least one switching section. We propose a small, pragmatic set of documentation rules to support the semantic integrity of the outer module, the switching section and the branches for the switch. We also propose a three-step method to help to identify the necessary semantic information.

The scope of the proposed method is limited to the overall view and one of the switching sections. The questions to which we propose an answer are the following:

- which documentation is necessary in the case of a switching section in order to assure the semantic integrity of both the switching section and of the surrounding module?
- when a switching section appears as part of a module's implementation, how should one proceed to maintain a semantic chain from the module being implemented down to the individual branches of the switching section, in order to maintain the semantic integrity of the whole module?

The steps to a design and implementation, which we propose, shall promote the semantic integrity of the outer module and its constituents. In this case we propose a top down procedure, going through the following three steps:

1. Section contracts. Identify the major implementation sections of the outer module and express the assertions between the sections. Each switching section in particular will be identified as a separate implementation section. The assertions immediately surrounding the switching section will define the contract it has to satisfy, corresponding to the conditions at A and C in Fig. 5.
2. Branch contracts. For each branch of the switching section, identify the branch dependency. It is based on the precondition for the switching section and the

switch condition for that branch. Then extract the contract for this branch, corresponding to the conditions at B and C in Fig. 5.

3. Branch specification and implementation. Specify and implement each branch so that it satisfies the corresponding contract. The pre- and postconditions for the branch must be at least as strong the branch contract, as developed in Subsection 6.3. This corresponds to the conditions at X and Y in Fig. 5.

Step 1 relates to the *implementation* of the outer module and accounts for the progress dependency mentioned in Section 5. Step 2 relates to the *implementation* of the switching section and accounts for the branch dependency, also mentioned in Section 5. Step 3 relates to the *specification* and *implementation* of the branch sections as shown in Fig. 5. In this way, the implementations of the individual inner modules are tied to the requirements stemming from the implementation of the outer module. More often than not, the specification of the branch in Step 3 will be the same as the branch contract from Step 2, but they have different semantic bearing. They may also differ, for instance if the branch implementation is based on reusable components, which already have their pre- and postconditions defined. The semantic gaps mentioned earlier are filled by the Steps 1 and 2.

The point here is that we are deducing the requirements of what each one of the branches of a switching section should do (Step 2) from the switching section's position in the implementation of the outer module (Step 1). This is a way to bridge the implementation of the outer module to the specification of the inner one. This sets the smaller building sections, which are the inner modules, in the context of the surrounding module and supplies a tool for verifying the consistency in the requirements. Therefore, it helps in achieving the semantic integrity of the outer module by assuring that it implements what it has promised in its own external contract.

This method allows for a top down, as well as a bottom up, approach. The implementation may be done top down, using the outer contract as its requirement specification. Steps 1 and 2 define the requirements on each branch in the switching section. Step 3 implements these branches. The implementation of the branches may also be bottom up by reusing a module, which already meets the branch contract. The following subsections will develop these three steps further.

6.1. *Identify the Major Implementation Sections of the Outer Module and the Corresponding Section Contracts*

In this paper, we only study implementation sections, which correspond to complete switching section. Once the sections are identified, the assertions between each section should be described. The descriptions do not need to be very formal, but should be as complete as possible. The assertions before and after one section will constitute the contract for that section. This is illustrated in Fig. 6.

When the implementation of the outer module is seen as a sequence of sections, each section drives the state of the module towards meeting the outer module's postcondition. The assertions A_i, which separate the implementation sections, should evolve, starting with the outer module's precondition and ending with its postcondition. The contracts C_i

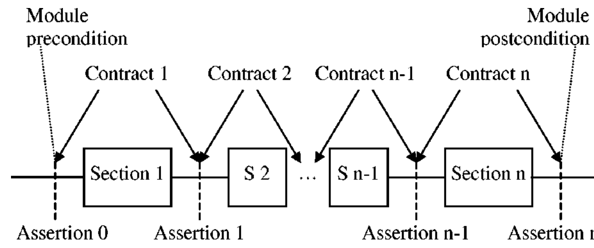

Fig. 6. The major sections and their contracts.

are defined by matching these assertions pairwise in such a way that the postcondition of one contract will be the precondition for the following one. Provided each section meets its contract, this will assure the semantic integrity of the outer module, since the sequence of implementation sections will implement the outer module correctly. The important point is that the correctness of the implementation can be based on the section contracts alone and not on the actual implementation of the sections.

6.2. *Identify the Switch Conditions and Each Branch Contract*

This is the step, which connects the requirements specification of the individual branches to the overall implementation of the outer module. It is the central step in assuring the semantic integrity of each switching section. This is illustrated in Fig. 7.

We have to make a simple assumption about the switch S itself. It should give control to one of the branches only, and not have any side effects observable to the rest of the switching section. If this is not the case, the switch will have to be split into smaller parts to meet these requirements.

The branch contract is deduced from the contract for the whole switching section in the following manner. Assume that the condition for the switch to select branch i is characterized by the assertion s_i. Then the precondition of the contract pre_i for branch i is the precondition Pre for the whole switching section strengthened with s_i. Mathematically this is expressed as

$$pre_i = Pre \wedge s_i.$$

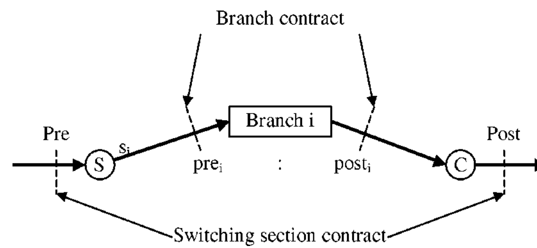

Fig. 7. The branch contracts.

The corresponding relation for the postconditions is simpler. Since there is no data manipulation in the convergence point C, the postconditions $post_i$ for the branches and $Post$ for the switching section are the same, so

$$post_i = Post.$$

It follows that each branch will have a separate contract, depending on the corresponding switch condition. If every branch is specified this way, the branching section as a whole will have its semantic integrity assured. This, in turn, then assures the semantic integrity of the outer module as a whole.

6.3. *Specify and Implement Each Branch so that it Satisfies its Contract*

Now that the contract for each individual branch is known, all that remains is to implement that branch. This can be done in a top down or bottom up fashion.

The top down approach is the most common one. It implies to the production of some code which transforms the outer module's state from the state specified by the precondition to the one specified by the postcondition. The bottom up approach corresponds to reusing an already existing module. How that can be done without the risk of violating the branch's semantic integrity in discussed below.

To be a correct implementation of a branch, the reusable module must satisfy the contract for that particular branch. This is determined by comparing the branch contract with that module's pre- and postconditions. The module's precondition should be satisfied by the branch precondition. Conversely, the branch postcondition should be satisfied by the module postcondition. Informally one may say the module should be "better" than – or at least "as good" as –what is required by the contract, e.g., produce more with less input. More formally this can be stated by saying that the module's precondition must be weaker than or equal to that of the contract and the module's postcondition stronger than or equal to that of the contract. Expressed mathematically, using the index m for the module and c for the contract:

$$pre_c \Rightarrow pre_m \wedge post_m \Rightarrow post_c.$$

It is interesting to note that this is the same condition as the one governing the semantic restrictions of a subclass routine, as discussed in (Meyer, 1988). In that case, c and m would correspond to the superclass and subclass, respectively. This analogy can be exploited in further studies.

In the case study, one of the requirements was that it should be possible to add more branches in the future. To do that in a safe manner, Steps 2 and 3 should be repeated for each new branch. This, of course, requires that Step 1 has already been done.

7. Discussion of the Case Study as Compared to the Proposed Method

In this section, we compare the design of the product studied with our proposed method for the case of switching sections. The questions we ask are these:

1. Were all or some of the three steps done, implicitly or explicitly, during the product development?
2. Were the results from these steps documented in the specification, in the design documentation or in the code itself?
3. What consequences can we draw from the answers to these questions for the present and future quality of the product implemented?

In the solution studied, there certainly was an understanding of the requirements for the individual inner modules, corresponding to Step 1, but it was not documented. According to what we could see in the documentation, only the Step 3 was done explicitly. The design leader followed a method, which we did not study in detail. The objective of our study was to see if the contract principles were applied, not why they were or were not applied. Therefore, we only studied the resulting documentation, not the development method applied to define this documentation. However, we may assume that it was a top down method, since the inner modules were actually tailored to the specification of the outer module, so the output from Step 2 and 3 are identical. We also learned from the design leader that the method specifies that Step 3 be done, and we actually found that the branch modules were specified with their pre- and postconditions.

However, both Step 1 and Step 2 were missing in the documentation of the design of the inner blocks. Therefore, we could not verify if the specification and implementation of the branches, as defined by their pre- and postconditions, satisfied the requirements of the outer module.

From the conversations we had with the designers of the system, however, we can assume that even Step 1 was done, but informally and intuitively. The design leader, who also worked with the implementation, informally understood the conditions at each step and designed the branch blocks according to her understanding. Therefore, it is reasonable to believe that the implemented solution was correct, although this knowledge was only in her mind. However, we saw no trace of Step 2, which is a conscious analysis of the consequences of the switching operation, in the documentation. In addition, the actual designer does not work with this product any more, and even if she did, she would have forgotten many details by now. Therefore, a problem may arise when the branch modules need to be changed or new branches be added.

In the current solution, part of the knowledge required for maintenance and further development will need to be extracted from the documentation of the pre- and postconditions for the existing branch modules. This approach has two problems.

Firstly, as it turned out, even these conditions were not complete and well understood, but were included because the method used said so, so it may even be necessary to go to the implementation of the branch modules for information. This, of course, raises a serious question for any method maker, including the authors of this paper. What is worse, a good method with a poor understanding or a good understanding with a poor method? This is a question for further study.

Secondly, the branch modules' pre- and postconditions only express the conditions actually met by the individual components according to Step 3. They do not express the needs set by the outer module as by Step 1 or the requirements stemming from the

switching logic as by Step 2. Equipped with individual pre- and postconditions only, the branch modules are like isolated islands. Some reverse requirements engineering is necessary in order to extract the conditions they need to meet.

In the design structure and method steps proposed in this paper, the needs for each part of the outer module are clearly documented in the form of a contract. Any new inner module, which follows the contract, is a valid module. The outer module does not rely on the knowledge of the implementation of a specific inner module for its correctness, only on the contract.

8. Conclusions

In this paper, we have studied the design and implementation of a software module containing a sequence of switching sections. We discovered "semantic gaps" in the case study, meaning that there was no documented continuous semantic line from the external requirements of the module down to the specification and implementation of the individual parts. The gaps appeared both in the specification of each switching section and in the documentation of the requirements for each switch branch. Therefore, the specification of the modules, which implemented the individual branches of the switching section, could not be backed up semantically. Rather, these specifications appeared in isolation and did not sufficiently support the continuous development and maintenance process. Nevertheless, at the time of our study, the project was successful. Now, two years later, is would be interesting to follow up its success record. Section 9 explains why that has not been done.

We have shown that contracts can be set up to specify the requirements for the switching section as a whole. Contracts can also be used to specify the pre- and postconditions for the branches of the switching section so that their correctness requirements can be established. With these requirements documented it is also safe to modify the branches or extend the number of branches at a later date. The kind of quality obtained was called semantic integrity. We have also proposed three steps to identify and document this semantic information and to fill in the semantic gaps. The steps are simple and pragmatic enough to be applied in practical software development work.

9. Further Studies

This paper results from a work which is part of our research in software quality based on semantic integrity. Part of this research focuses on how the use of preconditions, postconditions and invariants can be promoted in the industry in order to obtain better quality code. We believe that the contract concept is a useful vehicle for this.

It is interesting to note from this case study, how the branches of a selection follow similar semantic rules as subclasses in object oriented software construction. The contract defined for each branch corresponds to the contract of an abstract superclass or of an interface definition. The implementation of a branch corresponds to a concrete subclass. The

contract for the branch settles the requirements for the branch, but the implementation does not need to follow the contract exactly. It can do better than what is required by the contract without violating it. Actually, this is a general observation regarding contracts. The implementation of the service, which shall satisfy the contract, may do better without harm but may not do worse. This observation suggests that each branch could be defined as an abstract class, with possibly different implementations defined as subclasses.

The original study reported in (Blom, 1997) involved two different development projects. The case study referred to in this paper was the one studied the most thoroughly, but the other one led to similar conclusions. None of the projects was planned to be followed up regarding semantic errors, but the other project had an error tracking system. To be able to confirm our predictions of possible further development and maintenance problems, we therefore have followed up the other project. However, after collecting and studying the error report, it turned out that the kind of errors and corrections reported were too general and of no help to us. This explains why the quality of the project presented in this paper has not been followed up with respect to semantic errors, and that no such follow-up is scheduled.

Instead, in January 1999, we started a new three-year research program in cooperation with a software development company in Karlstad. The project is supported by NUTEK, the Swedish National Board for Industrial and Technical Development. The objective for the project is to study the software methods applied in industry, to try to identify potential for improvement in the area of module specifications and to develop methods to achieve such improvements. During this project, we will try to set the principles presented in this paper at work and to monitor the effects of this effort.

This paper has also presented the problem of the balance between the method and the understanding. This is an important challenge to all method developers and will be focused on in the NUTEK supported project.

The case of semantic integrity studied in this paper may be called a horizontal integrity, since the switching sections are peer with each other and so are the branches within one switching section. A completely different approach would be used in the case of multi-tier architectures. There, contracts may be used to support what might be called vertical semantic integrity, since different parts of the system support each other to perform different parts of a common task. This is the subject for a separate study.

References

Hoare, C.A.R. (1972). Proof of correctness of data representation. *Acta Informatica*, **1** , 271–281.

Meyer, Bertrand (1988). *Object Oriented Software Construction.* Prentice Hall.

Blom, M. (1997). Semantic integrity in program development. *Master's Dissertation*. Karlstad University.

Blom M., E.J. Nordby, D.F. Ross and E. Jonsson (1998). Semantic integrity in the programming industry: a case study, In *Proceedings European Software Day, Euromicro 98*. Västerås, Sweden.

E.J. Nordby has a M. Sc. in Computer Science, dating from 1979 from the University of Oslo, Norway. He has been working in research and development and is now teaching at Karlstad University in Sweden. His special interest is software quality in general and semantic specifications in particular. He is currently conducting a research project on semantic specifications together with a software development company, sponsored by NUTEK, the Swedish National Board for Industrial and Technical Development.

M. Blom holds a B. Sc. in Computer Science and is working with Eivind as a doctoral student in the same project. Martin also teaches undergraduate courses in Computer Science at Karlstad University.

Sandoriai kaip sekcijų perjungimo semantinės darnos užtikrinimo priemonė. Konkretaus atvejo analizė

Eivind J. NORDBY, Martin BLOM

Straipsnyje išanalizuota telefono stoties valdymo sistemos sekcijų perjungimo modulio projektinė dokumentacija ir aptarta, kaip, panaudojus Bertrand Meyer įvesta sandorio sąvoką, galima padidinti šio modulio semantinę darną.