On the Relation between Design Contracts and Errors: A Software Development Strategy

Eivind J. Nordby, Martin Blom, Anna Brunstrom
Computer Science, Karlstad University
SE-651 88 Karlstad, Sweden
{Eivind.Nordby, Martin.Blom, Anna.Brunstrom}@kau.se

Abstract

When designing a software module or system, a systems engineer must consider and differentiate between how the system responds to external and internal errors. External errors cannot be eliminated and must be tolerated by the system, while the number of internal errors should be minimized and the faults they result in should be detected and removed. This paper presents a development strategy based on design contracts and a case study of an industrial project in which the strategy was successfully applied. The goal of the strategy is to minimize the number of internal errors during the development of a software system while accommodating external errors. A distinction is made between weak and strong contracts. These two types of contracts are applicable to external and internal errors respectively. According to the strategy, strong contracts should be applied initially to promote the correctness of the system. Before release, the contracts governing external interfaces should be weakened and error management of external errors enabled. This transformation of a strong contract to a weak one is harmless to client modules.

1 Introduction

In designing a software module or system, it is necessary for the systems engineer to consider and differentiate between the way in which the system responds to external and internal errors. Incorrect behavior by end users or by external systems is a typical example of external errors. Design and programming errors are typical examples of internal errors, resulting in faults in the system that is being built. External errors must be tolerated by the system, while the number of internal errors should be minimized and the faults they result in should be detected and removed.

This paper presents a software development strategy based on design contracts [11]. This strategy should support software developers in their decision as to what kind of error handling they should include in their software. It is based on three principles. The first principle is to make a distinction between weak and strong contracts. The second principle is to exploit the fact that external and internal errors correspond to weak and strong contracts respectively. The third principle is Liskov's substitution principle [8]. The strategy exploits the fact that a weaker contract defines a Liskov-subtype of a stronger one. It prescribes starting with strong contracts in order to minimize the number of internal errors and then to weaken selected contracts in order to tolerate external errors.

This paper also reports from a case study of an industrial development project in which the strategy described was applied. Parts of the product developed were designed using strong contracts. Towards the end of the project, some of these contracts were weakened in order to accommodate external errors in the user interfaces.

The remainder of the paper is organized in two major parts, a strategy part and a case study part, followed by conclusions. The strategy part consists of Sections 2 and 3. First, external and internal errors are related to weak and strong contracts. Then Section 3 demonstrates that the transformation of a strong contract to a weak one corresponds to Liskov subtyping, and introduces the development strategy based on this fact. The case study part consists of Sections 4 through 6. The project studied is presented in Section 4 and experience gained from applying the strategy in this project is given in Sections 5 and 6. They show that the application of the strategy contributed positively to a successful result. Finally, the conclusions are presented in Section 7.

2 Errors and design contracts

This section starts by reviewing the distinction between external and internal errors. It then summarizes the principles of design contracts and introduces two categories of contracts, in this paper called weak and strong contracts. Finally, a correspondence is established between these contract categories and external and internal errors respectively.

2.1 External and internal errors

Several definitions exist of the terms error, fault and failure [3, 7, 14]. This paper uses these terms according to Fenton and Pfleeger [3]. The term error is used for the dynamic property that something wrong is being done by someone or something, for instance a user misusing the system, an external system not responding correctly or a designer misunderstanding a specification. An error, for instance during design or implementation, may result in a fault, which is a static product property, a deviation from the correct implementation. A fault may cause a failure, which is a dynamic product property, implying that the system does not behave as intended. In brief, an actor may commit an error, a system may contain a fault and, as a consequence, the system may fail.

A systems engineer must consider and differentiate between external and internal errors in his or her development work. One distinction between the two kinds of errors is that external errors arise when the system is used while internal errors arise when it is created.

External errors are committed by actors external to the system. An end user may for instance enter some illegal or meaningless input, such as typing letters in a number field or entering a value out of the acceptable range. Similarly, an external system may malfunction, possibly because of a physical or logical fault. Examples include external storage device errors and networking problems. External errors affect the system from the outside without modifying the software itself. For the system to be robust and maintain both system integrity and user friendliness, it should be constructed to tolerate and deal with these errors.

Internal errors are errors committed by designers or programmers in the development team. A designer may for instance misunderstand some detail of the requirements for a certain module or a coworker may be ambiguous in specifying the need for some functionality. Similarly, a programmer may commit a programming error, possibly because he or she is tired or because a detail was overlooked. Internal errors are committed by the system designers and programmers while the system is developed and introduce static faults into the system software. An incorrect algorithm in a module, an ambiguous function specification and an incorrect instruction in a source code are examples of such faults. Once introduced, these faults remain in the system until they are detected and removed. They represent a potential cause of system failure even when the system is used correctly. While internal errors can never be totally eliminated, their number should be kept to a minimum and the faults they result in should be detected and removed.

```
Precondition:  true
Postcondition: if not empty@pre
               then result = top element
               else EmptyException thrown
```

Figure 1. A weak contract for `top`

2.2 Weak and strong contracts

Design contracts are used to define the semantics of operations and to specify the responsibilities of both clients and suppliers of the operation. A client of an operation is a software part using it and a supplier is one implementing it. The contract of an operation consists of a precondition and a postcondition [11]. Correctness with respect to the operation is achieved when clients satisfy the precondition before calling the operation and suppliers satisfy the postcondition when terminating. In the case in which a precondition is not initially satisfied by a client, the supplier does not have to satisfy the postcondition. In such a situation, the outcome of the operation is explicitly left undefined [11, 13]. This leaves the supplier free to produce any result, to not terminate or to abort the execution, to name but a few examples. The actual choice is a matter of convenience. It is not a correctness issue but is considered part of the robustness. The classic example, which will be used in this paper, is the stack and the operation `top`, returning the topmost element of the stack. This operation may be successfully completed only if the stack actually has a top element.

Two major categories of contracts are identified, called weak and strong contracts in this paper. Weak contracts typically have the precondition `true`, implying that the client has no obligations whatsoever. Instead, the supplier must be prepared to handle even meaningless calls, such as `top` being called when the stack is empty, and the definition of the operation must prescribe the outcome in such cases. Meyer [11] refers to this as the tolerant approach to contract design. The outcome will typically be some kind of error indication, such as a status value being defined or an exception being thrown. This approach is illustrated in Figure 1. The notations `empty@pre` and `result = top element` used in the figure are OCL-notations[1] for the value of the property `empty` at the start of the execution of the operation and the value returned from the operation respectively [15].

Strong contracts require clients to satisfy a specific precondition, as shown in Figure 2. The postcondition only states the outcome in the legal situations, that is the situations in which this precondition was true. Meyer [11] refers to this as the demanding approach to contract design or the

[1]OCL, the Object Constraint Language, defines a syntax to express preconditions, postconditions and other assertions. It was initially defined by IBM and is now included in the family of standards managed by the Object Management Group (OMG).

```
Precondition:  not empty
Postcondition: result = top element
```

Figure 2. A strong contract for `top`

”tough love” approach. Whenever applicable, he recommends this alternative.

2.3 Relation between contracts and errors

An interface exposed to an external system or an end user will be called an external interface, and one exposed to another part of the same system will be called an internal interface. A weak contract corresponds to accepting input errors and is suitable for external interfaces, which are exposed to external errors. A strong contract assumes that it is possible for the operation to be called correctly and is suitable for internal interfaces exposed to internal errors.

A weak contract allows meaningless calls to a supplier operation. Defining weak contracts on internal interfaces, like in Figure 1, would imply that meaningless calls could be given a meaningful treatment. However, these meaningless calls result from faults in the client part of the system. If the operation is defined with a weak contract, the supplier will have to detect the meaningless call. It will then have to return the responsibility to handle the situation back to the client software, for instance through an error status or an exception. However, handling this error indication requires as much effort from the client as assuring a correct call in the first place.

3 Transforming strong contracts to weak

One powerful property of contracts is that they may be used to define subtyping under the control of logical statements. The basic subtyping principle is stated by Barbara Liskov in her substitution principle [8]. One common application of this principle for assuring subtyping in the context of subclassing is Meyer’s assertion redeclaration rule [11]. On the basis of this rule, this section defines weak and strong contracts in a way satisfying the substitution principle. The definition gives practical guidelines for when the contract for an operation can be modified without affecting the clients of the operation. This definition is at the base of the development strategy, stated last in this section.

3.1 Liskov’s substitution principle

Barbara Liskov has shown the usefulness of type hierarchies for several purposes, for instance in software design. She stated the substitution principle while relating the usefulness of type hierarchies in program development to data abstraction [8]:

> If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T.

In summary, this principle states that the type S of an object is a subtype of the type T of another object if it is impossible to observe any difference in behavior when the S object is substituted for the T object. Liskov notes in particular that the subtype must have all the operations of its supertype and the operations must do the same things.

Subtyping is a desirable property when the contract of an operation in a module is replaced by another contract. If the new contract is such that the modified module is a subtype of the same module with the original contract, we will say for simplicity that the new contract defines a subtype of the original contract. In that case, the client environment of the module may remain unchanged across the modification. We therefore need a substitution rule for contracts, answering the question of when one contract for an operation defines a subtype of another.

3.2 Definition of strong and weak contracts

Up to this point, the terms strong and weak contract have been defined intuitively. We will now propose a more precise definition that allows us to compare the strongness of two contracts for the same operation and that defines the module of a weak contract to be a subtype of a module with a stronger contract for the same operation. Replacing the latter module by the former will therefore be a transparent operation, as seen from the point of view of the clients of the module. The definition will for instance make it possible to compare the contracts of Figure 1 and Figure 2 and determine that the first one defines a subtype of the second. In the following, saying that condition A is stronger than condition B is equivalent to saying that A logically implies B, or that B is true whenever A is. Weaker, of course, has the opposite meaning.

Bertrand Meyer’s assertion redeclaration rule for classes [11] may be used as a starting point for defining a rule for subtyping in terms of the design contracts of the module operations. It says:

> A routine redeclaration may only replace the original precondition by one equal or weaker and the original postcondition by one equal or stronger.

This rule defines routine redeclaration as a behavioral subtyping [4, 9, 11] satisfying Liskov’s substitution principle [8]. For the purposes of supporting the development strategy presented here, however, it is too restrictive. In particular, it does not make Figure 1 define a subtype of Figure 2. Certainly, the precondition of Figure 1 is weaker than that of Figure 2. The postcondition of Figure 1, however, is not stronger than that of Figure 2, since a thrown exception satisfies the former but not the latter.

We therefore propose the following definition of weak and strong contracts, which achieves our goal of matching external and internal errors, still satisfying Liskov's substitution principle:

> A redefined contract is weaker than the original one if its precondition is equal to or weaker than the original precondition and its postcondition is equal to or stronger than the original postcondition in the domain of the original contract.

This definition determines the cases in which one contract is stronger or weaker than another one, but no absolute measure of "strongness" is defined[2]. Of course, for a contract to be redefined, the precondition and postcondition cannot both remain the same.

The clause "in the domain of the original contract" means that the redefined postcondition is considered only for those cases for which the original precondition is satisfied. This clause is needed to support the development strategy proposed in this paper and sufficient to conform to the substitution principle. The definition corresponds to the methods rule of the pre-behavioral subtyping defined in [1] and is motivated by the following informal reasoning.

A weak contract will typically have the precondition true where the corresponding, stronger contract has a specific precondition, as illustrated in Figures 1 and 2. As shown in Figure 1, the postcondition of the weak contract will consist of two parts. One part will be the same as the postcondition of the corresponding strong contract conditioned by the precondition of the strong contract and one part will be new compared to the corresponding strong contract. The substitution principle states that the redefined module should behave in the same way as the original module for all programs defined in terms of the original module. Such programs will assure the strong precondition and will then find the original postcondition satisfied after the call. The result in the case that the original precondition is not assured is explicitly left undefined by the contract. Thus, in that case, any postcondition will do for the redefined contract. It is therefore sufficient for the redefined contract to specify a postcondition at least as strong as the original one in the domain of the original precondition. Formal proof that this definition conforms to the substitution principle is beyond the scope of this paper.

[2]Truly, the strongest and weakest possible contracts can be defined. The strongest possible contract has the precondition false and the postcondition true. The weakest possible contract has the precondition true and the postcondition false. None of these are, however, particularly useful.

3.3 Development strategy and expected effects

The main principles discussed thus far are summarized below.

- External errors should be managed by the system.
- Weak contracts are useful for managing external errors.
- Internal errors should be minimized and the faults introduced detected and removed.
- Strong contracts are useful for disclosing and removing faults introduced by internal errors.
- Strong contracts can be weakened without affecting the clients of the operations.

Combining these observations, we propose the following development strategy [12].

> When developing a system with external interfaces, start by applying strong contracts for all operations and use a contract violation detection mechanism to identify and remove faults. Then selectively weaken the contracts of the external interfaces to tolerate external errors and add robustness in the external interface.

A discussion of contract violation detection mechanisms is beyond the scope of this paper, but inspections and runtime monitoring are relevant static and dynamic alternatives respectively [2, 5, 6, 10]. These may be used alone or in combination. Similarly, a discussion of alternative techniques for contract weakening is also not in the scope of this paper. The alternatives include modification of the interface, inheritance and wrapping.

The proposed strategy focuses on correctness and contract conformity, the primary expected effect of this being a decrease in the number of faults. It also focuses on a consistent use of strong contracts. An expected effect of this is lower product complexity, which is in turn expected to reduce both development time and the number of faults. As a result of the reduced number of faults, the time spent on testing and fault correction is also expected to be reduced. The final weakening of the contracts will probably contribute to some increase in development time, but planning for this weakening should minimize the extra effort and time needed. This increase in time should also be compensated for by the savings mentioned. The case study reported in the rest of this paper supports these expectations, but more research is needed to draw decisive conclusions.

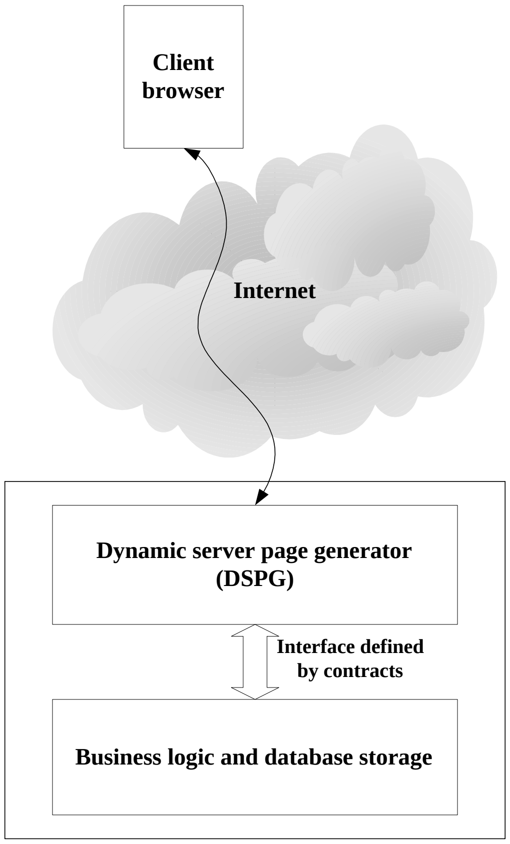
Client browser
Internet
Dynamic server page generator (DSPG)
Interface defined by contracts
Business logic and database storage

Figure 3. Architectural overview of the system

4 Presentation of the case study

A case study was made during the spring of 1999 of an industrial project that applied the strategy presented above. The project was of medium size, with about ten persons, most of them full time, for a period of six months. The remainder of this paper presents the case study and the experience gained from it. This section starts with an overall presentation of the nature and architecture of the software system produced in the industrial project. It then identifies the nature of the interfaces in the system and presents the contracts used initially by one of the system modules.

4.1 Overall system description

The product of the industrial project is an Internet server with both a wap[3] and a web interface. The server uses dynamic server pages technology to allow end users to access and modify user defined menu structures in a database hosted by the server. Access to the system is through wap enabled telephones or standard web browsers, at the user's discretion. The parts of the overall system architecture of relevance to this paper are shown in Figure 3.

[3]Wireless Application Protocol, a standard for providing Internet communications on digital mobile phones and other wireless terminals.

The system consists of the server system, divided into the dynamic server page generator module (DSPG) and the business logic and database storage module. The user interacts with the system through a wap or web browser, selecting a menu alternative or clicking on a button in the wap or web pages displayed in the browser. The browser transforms the user command into a URL string with parameters that it transmits across the Internet. On the server side, a dispatcher transforms this URL string into a call with parameters to an operation in the DSPG module. The business logic part supports this module with tailored operations on the data structure, which is stored in the database.

Much of the functionality in the system consists of routines to allow the user to manipulate the menus to be used from the wap telephone. A user will define menus containing his or her most common telecom services or links to frequently visited wap or web pages. These menus are presented as a line oriented series of choices. The operations the user can use to configure the wap menus include operations to add a new menu selection, to move a selection within a menu or to another menu, to define the details of a selection and to remove a selection. The user can also define new menus, link menus to each other and delete menus.

4.2 Identification of interfaces

Three principal interfaces can be identified in this architecture. One is the user interface, represented by the wap and web pages in the client browser. The next is the server interface, managed by the DSPG. Finally, there is the business logic interface. Of these, the first is an external interface and the last two are logically internal interfaces, since they are under the direct control of the software. The current browser page, which is defined by the DSPG, determines the operations that may be called and the arguments that may be supplied by the user. The interactions between the user, the browser and the system are illustrated in Figure 4.

4.3 The initial choice of contracts

Consistent with the strategy proposed in this paper, the interface to the business logic module was defined with strong contracts. The module contains 17 classes with a total of about 70 public operations, all initially defined with strong contracts. Including support operations, this accounts for about 6,000 lines of code, including comments and empty lines, or about 40% of the total code size. The contract for the operation to retrieve the details of a menu selection can be taken as an example. It is shown in Figure 5. The corresponding implementation is illustrated by the pseudocode in Figure 6. This implementation is con-

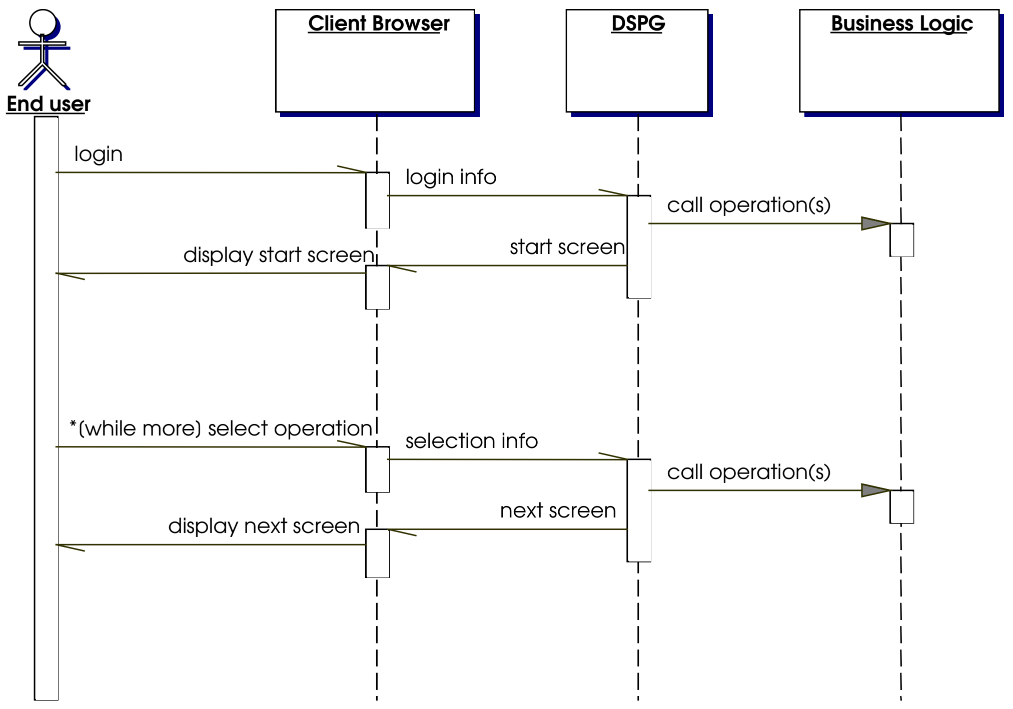

Figure 4. Browser/server interaction

sistent with the precondition, which states that the item searched exists in the menu. That implies for instance that the menu contains at least one element, so the loop will run at least once. It also implies that there is no need to check for the end of the list, since the element searched will always be found before the end of the list is reached. Also consistent with the contract principle is the fact that there is no strategy to recover in the case that the precondition is not satisfied. It is assumed to be satisfied.

5 Experience with strong contracts

This section summarizes experience gained from the application of strong contracts in the business logic layer of the product. It starts with a description of how the focus on correctness allowed a fast implementation of the business logic part. This is followed by a report on three effects of the strong contracts on the programming of the DSPG software. Thanks to the strong contracts, an error detection mechanism could pinpoint violations made by the DSPG programmers. They soon learned to respect the contracts, and this helped them to reduce the number of faults introduced in the system. Strong contracts in combination with some kind of violation detection mechanism proved to be a strong tool for correctness. During testing, the system also exhibited a more stable failure profile than an earlier, comparable project.

5.1 Focus on correctness

As usual, the project was under time pressure, and the business logic part was essential for the progress of the DSPG part. Two designers were assigned the responsibility of producing the business logic part. After an initial phase settling the design principles, the contracts for the operations were defined. After that, the operations could be rapidly implemented, focusing on correct functionality and avoiding error checking of input parameters. The implementation assumed that a precondition that was stated in a contract was always satisfied. This procedure allowed a fast and fault free implementation of the module. On only two occasions after the internal delivery to the project were minor adjustments needed in this part of the system.

5.2 Use of contract violation detection

With the finished business logic module, the DSPG programmers could start their progress. In this stage, the project took advantage of the error checking mechanisms in Java to detect and signal contract violations. The business logic module contained no error checking, but as soon as the precondition of an operation was not satisfied, the implementation code would perform some kind of illegal operation, for instance indexing an element out of bounds or attempting to reference an object through a null pointer. This would be caught by the built-in error control in Java.

```
Precondition:  the item exists in the menu
Postcondition: result = details for item
```

Figure 5. The contract for retrieving the details for a menu selection

Typically, the program would then crash and a Java system dump identifying the offending call would be displayed.

5.3 Client programmers conforming to the rules

To start with, the information given to the DSPG programmers about the strong contract definitions was insufficient. Being used to less strict function definitions, they did not pay a great deal of attention to the details in the calls and frequently made the error of violating the preconditions. Since violations of the preconditions caused the program to crash, they could not progress with their work until they had conformed to the contracts. This, of course, caused much frustration and acted as a very strong motivation to study and conform to the rules set up and to produce fault free calls to the operations in the business logic module.

5.4 Absence of faults in the client modules

All the frustration and system crashes were not in vain. Two facts could be noted. One, already mentioned, was that the business logic software produced the correct result. Only two faults were reported during testing and both were easily corrected. The other was that even the DSPG software was free from faults in its communication with the business logic part. The programmers made fewer and fewer errors and the faults produced during development were rapidly discovered and removed during testing.

5.5 Stability with respect to failures

Faults surviving module and integration testing were detected early during system testing. If for instance ten test cases were run without failure before system delivery, subsequent test cases run by the customer were also failure free. This is different from earlier projects not using contracts, where ten test cases could be run without failure but later test cases run by the customer after delivery could experience transient failures, revealing a fault. Transient failures were not a problem in the project reported here, and no new faults were normally discovered after delivery.

```
loop from first item
  compare current item with parameter
until parameter item found
return the details of the current item
```

Figure 6. The pseudocode for retrieving the details for a menu selection

6 Weakening the contracts

In the business logic layer, all the operations were defined with strong contracts. Some of these operations were exposed to external errors through an interface accessible to the end users. These operations therefore had to use weaker contracts in the finished product. The change from strong to weak contracts was made late in the project. The operations whose contracts needed to be weakened, and the weakening procedure chosen are presented in this section. Experience gained during this process is then described and shows that the weakening of the contracts did not cause any noticeable problems.

6.1 The contracts and the weakening procedure

The first step was to identify the contracts that had to be weakened. The end user interface, as it appears in the wap or web browsers, is under the control of the system. This forces correct use of most of the operations. However, the calls from the client browser to the server pass as standard URL strings that may be repeated or manipulated by the end user, potentially producing an illegal call to the server. This call would propagate down through the DSPG module to the business logic layer. These operations, accessible directly from the Internet interface, were thus the candidates for weaker contracts. The project identified 16 such operations.

Two alternative procedures were considered to make these operations tolerant to external errors. One was to implement explicit inquiry operations for the contract preconditions in the business logic module and implement a wrapper module with the weaker contracts. The other was to modify the business logic operations themselves. While the first procedure is the most modular one, the second one was chosen, mainly because it could be implemented faster.

6.2 Experience from contracts weakening

With the strong contracts, all input conditions that did not satisfy the precondition were invalid. To weaken the contracts, some or all of these conditions were defined to be valid and special cases were added to the postconditions to specify their result. In the project, an exception was specified to be thrown in these special cases. The changes were similar to the ones made from Figure 2 to Figure 1. The implementation of the operations in the business logic layer was then modified to take care of these extra cases, throwing exceptions according to the new postcondition. Similarly, some code in the DSPG module was modified to catch these exceptions, displaying a user message stating that the call was invalid.

The project confirmed that the changes to the business logic layer could be done and that the modified operations did not affect existing client code that already satisfied the initial, strong contracts. The modifications to the DSPG code were also easily made and did not cause problems or introduce new faults.

7 Conclusions and further study

We have presented a strategy based on design contracts for error management during software development. The strategy states that the development of a subsystem should be based on strong contracts in order to identify and eliminate internal errors. Before delivery, the contracts for subsystems with external interfaces should be weakened in order to tolerate external errors. The strategy is based on the mapping that exists between contracts and errors, where weak contracts are appropriate for interfaces exposed to external errors and strong contracts are appropriate for interfaces exposed to internal errors. A practical definition of weak and strong contracts was provided that conforms to Liskov's substitution principle. This definition assures that a strong contract can be substituted by a weaker one without affecting the clients of the operation defined by the contract.

We have also presented a case study of an industrial project in which the strategy was successfully applied. The interface to the business logic module was defined with strong contracts. This proved efficient in keeping down the number of faults both in the business logic module itself and in its clients. It also contributed to making the system stable with respect to failures, with few transient failures both before and after delivery. Late in the project, the contracts of the operations accessible to the external user interface were weakened to tolerate external end user errors. The adaptation of the implementation to these weakened contracts was easily made and did not introduce new faults.

The expected effect of the strategy is a total gain in time and quality. Although the positive effects of the strategy were confirmed in the case study, more research is required to provide general support for this conclusion.

Acknowledgements

The authors would like to thank Per Grundström, Helena Lindskog and the development team for welcoming us into the development project and supporting us during the case study. This work was partly funded by the Swedish National Board for Industrial and Technical Development (NUTEK).

References

[1] K. K. Dhara, Leavens, G. T. Forcing Behavioral Subtyping Through Specification Inheritance. In *Proceedings 18th International Conference on Software Engineering*, pages 258-267, IEEE Berlin, Germany, 1996.

[2] M. Dyer. *The Cleanroom Approach to Quality Software Development*. Wiley, 1992.

[3] N. E. Fenton, S. L. Pfleeger. *Software Metrics, A Rigorous & Practical Approach, second edition*. PWS Publishing Company, 1997.

[4] R. B. Findler, M. Felleisen. Contract Soundness for Object-Oriented Languages. In *OOPSLA '01 Conference Proceedings*, pages 1-15, ACM, 2001.

[5] T. Heyer. *Semantic Inspection of Software Artifacts: From Theory to Practice*. Ph.D. thesis, Department of Computer and Information Science, Linköping University, SE-581 83 Linköping, Sweden, 2001.

[6] R. Kramer. iContract — The Java Design by Contract Tool. In *Proceedings of the TOOLS'98 Conference*, Santa Barbara, USA, 1998.

[7] J. Laprie (ed.). *Dependability: Basic Concepts and Terminology*. Springer-Verlag New York, 1992.

[8] B. Liskov. Data Abstraction and Hierarchy. In *OOPSLA '87 Addendum to the Proceedings*, October 1987.

[9] B. Liskov, J. M. Wing. A Behavioral Notion of Subtyping. In *ACM Transactions on Programming Languages and Systems*, November 1994.

[10] B. Meyer. *Eiffel: The Language*. Object-Oriented Series, Prentice Hall, 1992.

[11] B. Meyer. *Object Oriented Software Construction, 2nd edition*. Prentice Hall, 1997

[12] B. Meyer. The Significance of dot-NET. In *Software Development Magazine*, November 2000.

[13] J. Rumbaugh et al. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.

[14] M. Shooman. *Software Engineering*. McGraw Hill, New York, 1983.

[15] J. Warmer, A. Kleppe. *The Object Constraint Language, Precise Modeling with UML*. Addison Wesley, 1999.