Error Management with Design Contracts

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

Abstract

When designing a software module or system, a software engineer needs to consider and differentiate between how the system handles external and internal errors. External errors must be tolerated by the system, while internal errors should be discovered and eliminated. This paper presents a development strategy based on design contracts to minimize the amount of internal errors in a software system while accommodating external errors. A distinction is made between weak and strong contracts that corresponds to the distinction between external and internal errors. 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. In addition to presenting the strategy, the paper also presents a case study of an industrial project where this strategy was successfully applied.

1 Introduction

When designing a software module or system, a software engineer needs to consider and differentiate between how the system handles external and internal errors. Incorrect behaviour by end users and by external systems are typical examples of external errors. Design and programming errors are typical examples of internal errors. Such errors result in faults in the system being built. External errors have to be be tolerated by the system, while internal errors should be minimized and the faults they result in should be discovered and removed.

This paper presents a development strategy based on design contracts for error management in software development. The strategy is based on three principles. One is to make a distinction between weak and strong contracts. Another principle is the correspondance between external and internal errors and weak and strong contracts respectively. The third one is Liskov's principle of substitutability [2], which implies that a strong contract may be replaced by a weaker one without harm. This is exploited by the strategy. It prescribes to first use strong contracts to minimize internal errors and then weaken selected contracts to tolerate external errors.

During the spring of 1999, a case study was conducted of an industrial development project where the strategy described was applied. Some software modules 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. This paper also reports on the experiences from this industrial project.

The remainder of the paper is organized in two major parts, a strategy part and a case study part, followed by a conlusion. In the strategy part, the two different kinds of errors that a software engineer has to face are first presented and related to strong and weak contracts. Then, Liskov's substitution principle and Meyer's assertion redeclaration rule [3] are presented and combined, showing that the transformation of a strong contract to a weak one is a harmless operation, confirming with Liskov subtyping. The development strategy, which is to start out with strong contracts and then weakening selected contracts, is then deduced from these principles. In the case study part, the project studied is presented and the experiences from applying the strategy in this project are summarized. The conclusion from this case study is that the application of the strategy gave a positive contribution to a successful result.

2 Errors and design contracts

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

2.1 External and internal errors

This paper uses the terms error, fault and failure according to Fenton and Pfleeger [1]. An error is a dynamic prop-

erty of an actor, something wrong 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 software engineer has to face external and internal errors in development work. External errors are errors committed by actors external to the system. An end user, for instance, may enter some illegal or meaningless input, like typing letters in a number field or entering a value out of range. Similarly, an external system may malfunction, possibly because of a physical or logical fault. Examples include external storage device errors and networking problems.

Internal errors are errors committed by designers or programmers in the development team. They result in faults being built into the system itself. These faults should never have been introduced, and the ones that are should be discovered and removed as soon as possible.

One important distinction between external and internal errors is that external errors arise when the system is used while internal errors arise when it is created. External errors affect the system dynamically from the outside without modifying the software itself. To maintain system integrity and user friendliness, these errors should be tolerated and dealt with by the system. The internal errors, on the other hand, are committed by the system designers and programmers while the system is being developed. They introduce static faults into the system software. Once introduced, these faults remain in the system until they are detected and removed, potentially causing the system to fail even when used correctly. Even if internal errors can never be totally eliminated, their number should be kept low and the faults they result in should be detected and removed.

2.2 Weak and strong contracts

Design contracts are used to define the semantics of operations and to specify the responsibilities of both the client and the supplier of the operation. A contract consists of a precondition and a postcondition [3]. Correctness is achieved when the client software satisfies the precondition before calling the operation and the supplier implementation satisfies the postcondition when terminating. In the case when the precondition is not initially satisfied by the client, the supplier is not bound to satisfying the postcondition. In such a situation, the outcome of the operation is explicitly left undefined [3], [5]. This leaves the supplier the freedom 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 also 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 according to this description only if the stack actually has a top element.

Two major categories of contracts are identified, called weak and strong contracts respectively in this paper. The weak contracts typically have the precondition `true`, implying that the client does not have any obligations whatsoever. Instead, the supplier must be prepared to handle even meaningless calls, like `top` being called when the stack is empty, and the definition of the operation must prescribe the outcome in such cases. Meyer [3] refers to this as the tolerant approach to contract design. The outcome will typically be some kind of error indication, like a status value being defined or an exception being thrown. This approach is illustrated in Figure 1. The notations `some property@pre` and `result = some expression` used in the figure are OCL-notations[1] for the value of the property `some property` at the start of the execution of the operation and the value returned from the operation respectively [4].

```
Precondition:  true
Postcondition: if empty@pre
               then EmptyException thrown
               else result = top element
```

Figure 1: A weak contract for `top`

Strong contracts require that the client satisfies a specific precondition, as shown in Figure 2. The postcondition only states the outcome in the legal situations, that is the situations where this precondition was true. Meyer [3] refers to this as the demanding approach to contract design or the "tough love" approach.

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

Figure 2: A strong contract for `top`

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

[1] OCL, the Object Contraint 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 by the Object Management Group (OMG).

exposed to external errors. A strong contract assumes that it is possible for the operation to be called correctly. That is possible only for operations in internal interfaces, but even clients to such operations will contain faults resulting from internal errors.

Applying weak contracts, as illustrated in Figure 1, for internal errors would correspond to accepting unacceptable situations that ultimately result from program faults. A weak contract allows meaningless calls to a supplier operation. The supplier is required to detect the error and will return the responsibility back to the client software, expecting it to take care of the returned error indication. 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 modified under the control of logical statements. The basic principle is stated by Barbara Liskov in her principle of substitutability [2]. This principle is refined by Bertrand Meyer for class inheritance in his Assertion Redeclaration rule [3]. The same logic can be applied to the contract of an operation to predict whether the modification of the operation will affect the clients or be unnoticeable to them.

3.1 Liskov's substitution principle

Barbara Liskov stated her principle of substitutability while relating the usefulness of inheritance hierarchies in program development to data abstraction [2]:

> 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, it states that the type of an object is a subtype of the type of another object if it is impossible to observe any difference in behavior when the latter object is substituted by the former. This property is also wanted when a contract is replaced by another in a module, since it allows the client environment of the module to remain unchanged across the modification. We therefore need a principle of substitutability for contracts, answering the question when a contract defines a module to be a subtype of another.

3.2 Transparent transformations

If a contract defines one module to be a subtype of another, replacing the latter by the former is a transparent operation as seen from the clients' point of view. This corresponds to Meyer's Assertion Redeclaration rule for classes [3]. It expresses when an object of a subclass can replace an object of its superclass without affecting the clients of the class.

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

3.3 Definition of strong and weak contracts

Up till now, the terms strong and weak contract have been defined intuitively only. Now, they can be defined in a somewhat more precise way. A contract is strong or weak relative to another one. That means that a contract can be stronger than or weaker than another one but no absolute measure of "strongness" is defined. Our definition of when a contract is stronger than another is given below:

> If two contracts obey the Assertion Redeclaration rule of Section 3.2, then the original contract is said to be stronger than the redefined one.

This definition automatically implies that the transformation of a contract to a weaker one follows the Assertion Redeclaration rule. Such a transformation is transparent to clients of the operation since it does not affect their behavior, as paralleled by Liskov's principle of substitutability.

3.4 Development strategy and expected effects

The main principles discussed so far are summarized below.

- External errors should be managed by the system.
- Internal errors should be minimized and the faults introduced identified and removed.
- Weak contracts are useful for tolerating external errors.
- Strong contracts are useful for detecting and removing faults introduced by internal errors.
- Strong contracts can be weakened without affecting their clients.

Combining these observations, we propose the following development strategy.

> When developing a system with external interfaces, start out with strong contracts for all operations and equip the operations with a contract violation detection mechanism. Then weaken selectively 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 outside the scope of this paper, but inspections and runtime monitoring are two relevant alternatives. They may be used alone or in combination. Similarly, a discussion of alternative techniques for contract weakening is also outside 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 is a decrease in the number of faults. It also focuses on a consistent use of strong contracts. An expected effect of this is a lower product complexity, which in turn is 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 extepted 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.

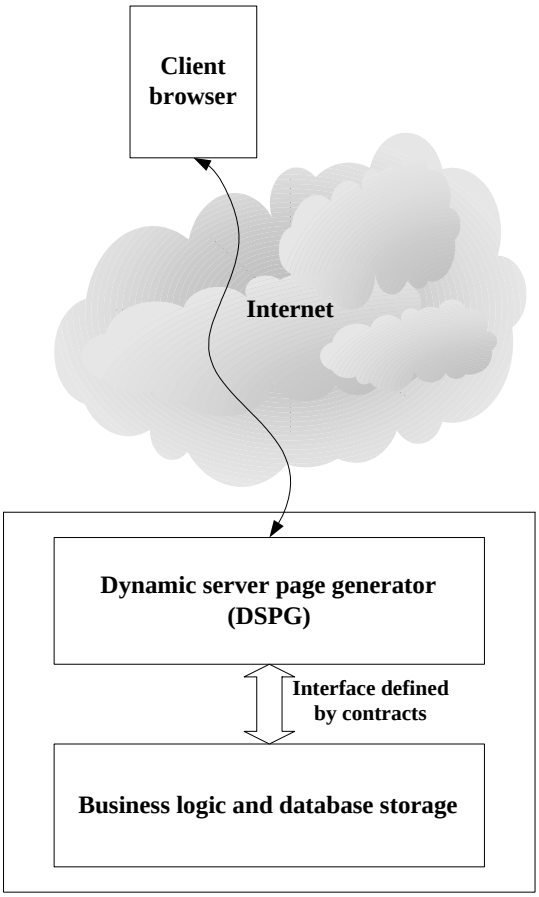

Figure 3: Architectural overview of the system

4 Presentation of the case study

As mentioned initially, a case study of an industrial project where the strategy presented above was applied has been conducted. The remainder of this paper presents the case study and the experiences gained from it. This section starts by 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 system produced by the project studied is a wap[2] server that also includes a web interface. It uses dynamic server pages technology to allow the end users to access and modify user defined menu structures in a data base hosted by the server. Access to the system is through wap enabled telephones or through 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.

The project was of medium size. It involved about 10 persons, most of them full time, for a period of 6 months. The size of the resulting software produced during the project is 15,800 new lines of code, including comments and empty lines.

The whole system consists of a client browser and the server system, the latter being divided into the dynamic server page generator (DSPG) and the business logic and database storage. The user interacts with the system by selecting a menu alternative or by clicking on a button in the wap or web pages displayed in the browser. A user command is transformed into a URL with parameters and transmitted across Internet. On the server side, a dispatcher transforms it 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 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.

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

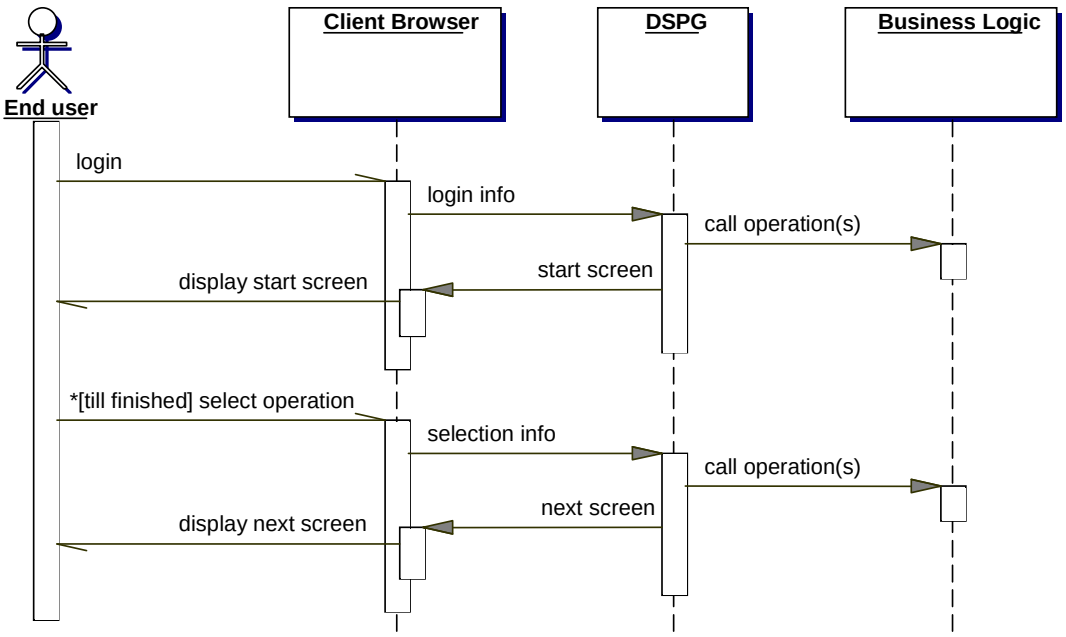

Figure 4: Browser/server interaction

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 one is the server interface, managed by the DSPG. Finally, there is the business logic interface. Of these, the first one is an external interface and the last two are logically internal interfaces, since they are under the direct control of the software. In this setup, the user may only call the operations and supply the arguments defined by the current browser page, which in turn is defined by the DSPG. These interactions 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 contained 17 classes with a total of about 70 public operations defined using 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.

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

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

According to the contract theory, the implementation of this operation will assume that the item is actually present in the menu, so this condition will not be checked by the code. The resulting implementation is illustrated by the pseudo-code in Figure 6.

```
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

This implementation is consistent 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 case the precondition is not satisfied. It is assumed to be satisfied.

5 Experiences from strong contracts

This section summarizes some experiences from the application of strong contracts in the business layer of the project. 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 keep the number of faults introduced in the system down. Strong contracts in combination with some kind of violation detection mechanism showed 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 DSPG part to progress. Two designers were assigned the responsibility to produce 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 the contract was always satisfied. This procedure allowed a fast and fault free implementation of the module. Only on two occasions after the internal delivery to the project were minor adjustments needed in this part of the system.

5.2 Use of violation detection

With the finished business logic module, the DSPG programmers could progress. In this stage, the project took advantage of the potential in the contracts to detect and signal contract violations. The error checking mechanisms in Java were exploited to detect contract violations. The business logic module had no error checking in it, but as soon as a precondition of an operation was not satisfied, the operation 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. Typically, the program would then crash with a run-time exception.

5.3 Client programmers conforming to the rules

To start with, the information to the DSPG programmers about the strong contracts used was insufficient. Being used to less strict function definitions, they did not pay so much attention to the details in the calls and frequently made the error to violate the preconditions. Since violations of the preconditions caused the program to crash, they could not progress with their work until they conformed with the contracts. This, of course, caused a lot of frustration and was a very strong motivation to study and conform with the rules set up and to produce fault-free code.

5.4 Absence of errors 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 fact is 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 that existed during development were rapidly discovered and removed during module testing.

5.5 Stability with respect to failures

During system testing, if a module had a fault, it was discovered during the early test cases. If for instance ten test cases were run without failure before 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. For the project reported in this paper, transient failures were not a problem and new faults were normally not discovered after delivery.

6 Weakening the contracts

Some of the operations defined by strong contracts 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 done late in the project. The operations, whose contracts needed to be weakened, as well as the weakening strategy chosen are presented in this section. After that, the experiences gained during this process are presented, showing that the weakening of the contracts did not cause any noticeable problems.

6.1 Identification of contracts and strategy

The first step was to identify the contracts that needed to be weakened. The end user interface, as it appears in the wap or web browsers, are under the control of the system. This forces a 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 entered or repeated by the end user, potentially producing an illegal call to the server. Therefore, the operations accessible directly from the Internet interface were the candidates for weaker contracts. The project identified 16 such operations.

Two main strategies were considered to make these operations tolerant to external errors. One was to implement explicit inquiry operations for the contract preconditions and implement a wrapper module with the weaker contracts. The other was to modify the operations themselves. The first strategy was the most modular one, but the second one was chosen. The main reason for this choice, was that it could be implemented faster. The modification to weaker contracts had not been anticipated sufficiently

early, so there was no support for the first strategy in the module and there was no time to implement it.

6.2 Experience from contracts weakening

With a strong contract, all the input conditions that do not satisfy the precondition are invalid. When the contract is weakened, some or all of these conditions are defined to be valid and special cases are added to the postcondition to specify their result. The implementation of the operation must then be modified to take care of these extra cases according to the new postcondition. The project confirmed that this could be done and that the modified operations did not affect existing client code that already satisfied the stronger contract.

6.3 Adaptation of client modules

The contracts were weakened by specifying that some exceptions should be thrown in the new special cases now allowed in the preconditions. In order to accommodate external errors, some client code was modified to catch the new return situations defined by the new postconditions. This was easily done by adding code to catch the exceptions thrown and display a user message stating that the call was not valid. All these modifications were easy to control and did not cause problems or introduce new faults.

7 Conclusions and furher 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. As an extension to Liskov's principle of substitutability, rules were provided for the transformation between strong and weak contracts. A strong contract can be substituted by a weaker one without affecting the clients of the operation defined by the contract. This is based on the same reasoning as Meyer's Assertion Redeclaration rule for subclassing.

We have also presented a case study of an industrial project where 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 in both the business logic module itself and 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 did not introduce new faults.

The expected effects of the strategy is a total gain in time and quality, as presented in Section 3.4. Although the case study supports the positive effects of the strategy, more research is required to be conslusive.

References

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

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

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

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

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