Method Redefinition - Weakening the Rules for Contract Specification

ABSTRACT

The discipline of design by contract encourages to build correct classes wrt a specification right from the beginning. Pre- and post-conditions, and invariants are used to give semantics to methods, to document a class - insofar as they define the contracts that client classes must engage in when they use it - and to check classes at runtime in order to test and debug them.

The several existing languages for contract specification and associated monitoring tools, impose certain rules in the redefinition of methods in subclasses. This is to avoid wild changes to the semantics defined in parent classes, in order to cope with the substitution principle - in which a son object can always be substituted by a parent one.

The *golden* rule proposed in [Meyer] to ensure the substitution principle - "don't ask for more, don't offer less" - has been widely adopted by the community. The way this rule is applied in the several existing monitoring tools does not differ much: the implications between pre and post-conditions of the original method and its redefinition, are enforced in order to ensure that, in the redefinition process, neither the pre-condition gets stronger nor the post-condition gets weaker.

In this paper we show that this way of enforcing the "don't ask for more, don't offer less" rule forbids, in the sense of hierarchy correctness, many perfectly reasonable and obvious contracts. Moreover, we claim that the ways we have to surpass this limitation, while maintaining the existing control mechanisms, put an extra burden on the method developer insofar as they ask for more intricate, and yet redundant, assertions.

We propose to weaken the above rules in a way that turns legal, in the sense of hierarchy correctness, all the above forbidden contracts while ensuring the substitution principle.

Keywords

Design by contract, inheritance, method redefinition, substitution principle.

1. INTRODUCTION

The discipline of design by contract defines rules and mechanisms that help us to build correct classes with respect to a specification right from the beginning.

This discipline relies on assertions that specify the rules of the contract that is established between supplier and client classes. Each time a client class calls a method of a supplier class, a contract is implicit: if the client class gives the supplier class what she asks for - the method's pre-condition - the supplier class guarantees what the client is expecting for - the method's post-condition. In this way, one may blame the right entity for a method's failure: if there is a pre-condition violation, we blame the client; if there is a post-condition violation we blame the supplier.

Both pre- and post-conditions as well as invariants are a precious tool for building reliable software. They provide a powerful means for specifying classes allowing the definition of the semantics of methods. If written in a friendly language of assertions, they provide very useful and unambiguous documentation for client classes implementors. Assertions can also be used for software test, debugging and quality assurance. Whenever they are monitored during method execution, (contract) illegal calls can be detected, and wrong implementations of post-conditions and invariant violations can be discovered.

There are several assertion languages that allow the specification of assertions in different, but yet similar, ways. Some, like Eiffel [11], are included in the programming language itself; others, like iContract [6], have Java boolean expressions as its base, and allow to write assertions for Java programs that are preprocessed into Java commands. There are many others that promote assertion mechanisms in several programming languages, like, e.g., Jass [1], ContractJava [5], JML [9, 2], jContractor [8], Handshake [4] for Java only, and [3,13] for C++, among them. Some of these – Jass, iContract, Eiffel – as well as others, allow the monitoring of contracts at runtime.

1.1 The Problem

The redefinition of a method in sub-classes can completely change the semantics of the parent class's method unless there are effective mechanisms that prevent it. The "is a" relation, that is implicit in the OO approach, between a subclass and the class from which it inherits, partly justifies the need to preserve the semantics - our intuition tells us that if a B "is an" A then it should behave like an A, or, at least, in a similar way (although *similar* is hard to define).

But, what really asks for effective measures that prevent wild changes in the semantics of redefined operations, is the existence of polymorphic entities together with the mechanism of dynamic binding. It may be the case, as presented in [11], that some client class C deals with a B object while thinking it is dealing with an A object. This is only natural in OO systems, and it reveals much of the power of OO. But we must be careful not to change behaviours in some way that goes against the client's expectations: if the client honours the contract defined in the method it invokes, the object that serves it (whatever it is) should also honour that same contract. The idea put forward in [11] - "don't ask for more, don't offer less" - as a means of granting the substitution principle (in which every B object should be able to be substituted for an object A) is widely accepted in the OO community, and several approaches exist that implement it [11, 6, 5, 1, 10], to name a few.

1.2 Existing Approaches to the Problem

Maintaining or weakening pre-conditions and maintaining or enforcing post-conditions and invariants when redefining methods in extended classes is the solution that is adopted in the Eiffel language as well as iContract [6], and ContractJava [5] for example.

Let preA and postA be the pre and post-conditions of some method m in class A. Let preB and postB be the pre and post-conditions that specify the behaviour of that same method m when redefined in subclass B. Ensuring that preA implies preB ensures that preB is not stronger than preA. Ensuring that postB implies postA ensures that postB is not weaker than postA.

ContractJava, for example, verifies these implications at runtime, reporting hierarchy errors whenever they are falsified.

Eiffel and iContract establish that contracts are inherited, and that redefined methods only define the parts that are of their direct contribution. They borrow the facts that

i) (P and Q) imply P, and

ii) P implies (P or Q),

to establish that the complete pre and post-conditions of m in class B are given by (preA *or* preB), and (postA *and* postB) where preB and postB are the assertions that are written in class B's method m. Eiffel distinguishes between a simple pre-condition and a pre-condition that extends some other: the former follows the word requires, and the latter follows the words requires else. It also distinguishes between a simple post-condition and a post-condition that extends some other: the former follows the word ensures, and the latter follows the words ensures then. The automatic or-ing and and-ing of assertions makes hierarchy errors absent from executions.

All these approaches ensure the substitution principle; all son objects behave at least like their parents in all situations where a parent is expected.

1.3 Our Approach

In this paper we claim that the above approaches are too restrictive. Let us explain why. In sub-classes we may weaken or maintain a redefined method pre-condition. This means that the method may be invoked over a son object in the same, or more, states.

In order to grant the substitution principle, the son must ensure, at least, what the parent would, whenever the method is invoked in states that satisfy the parent's pre-condition. This allows for the son to be substituted for the father; it also seems reasonable whenever the son is acting as itself - because every B is an A, it can be expected that a B behaves like an A at the same situations.

But, as will be shown, the above approaches also restrict the behaviour of son objects in the extra states that result from the weakening of the pre-condition. This does not seem reasonable to us.

When we weaken the pre-condition of a method m, we are saying that son objects can execute that operation in states where parent objects cannot. We are saying that son objects know how to deal with situations that parent objects do not. Then, it is only reasonable that we let son objects decide what has to be done in those new situations, which, in what parent classes are concerned, are novel, undefined situations.

We claim that the obligation to maintain or strengthen the post-condition, that is, to ensure what the parent does plus, eventually, some things more, is much too restrictive.

1.4 Outline of the Paper

The paper has five sections. The next section shows how existing assertion languages and monitoring tools restrict the accepted post-conditions for redefined methods in a way that, we argue, is unnecessary, and which discourages developers in the task of building contracts. Section 3 uses the notion of subtype conformance to stress the claims that are made in section 2. Section 4 talks about one approach to this problem proposed in the literature, and argues why it is not satisfactory. In that same section we propose our approach which implies weakening the rules that are used by most assertion monitoring tools. Finally, section 5 concludes.

2. Redefinition - Contracts

In this section, we will present an example that shows how restrictive existing approaches to redefinition control are. We will use the Java language and the iContract assertion language.

The iContract assertion language is based on Java boolean expressions with some extensions: pre-, post-conditions and invariants follow the words @pre, @post, and @invariant respectively. In post-conditions we may refer to the old value of a reference *f* by *f@pre*; we can access the result of a function through the identifier *return*. It also allows to quantify over some collection with *forall* and *exist* constructs and allows the writing of implications through the *implies* operator.

Let a class Account declare, among other features of a typical bank account, its balance and the person who owns it. It also defines methods that allow the deposit and withdrawal of money to and from the account. We are going to focus on the withDraw method.

This operation can only succeed if the balance of the account covers the amount that is to be withdrawn. If it succeeds, the account's balance is decreased by the given amount.

```
/**
*/
public class Account {
    ...
    public double balance() {...}
    public Person owner() {...}

    /**
    * @pre amount>0
    * @post balance() == balance()@pre +
    *                                amount
    public void deposit(double amount) {
           ...
    }

    /**
    * @pre balance() >= amount
    * @post balance() == balance()@pre -
    *                                amount
    public void withDraw(double amount){
           ...
    }
} // end of class Account
```

Another class follows that describes a special account in the sense that it is backed up by its owner's salary. That is, for some amount m to be withdrawn, it suffices that m does not exceed the balance plus the salary. The bank allows the balance to become negative down to some point, because the owner's

company deposits his salary in the account every month - this gives the bank some assurance that the balance will eventually return to a positive value. Moreover, whenever the balance becomes negative due to the withdrawal, an extra money is taken from the account: it corresponds to some kind of tax that is applied to the withdrawn amount.

Thus, the withDraw method has to be redefined in this SpecialAccount class.

```
/**
*/
public class SpecialAccount extends Account{
    ...
    public double tax() {...}

    /**
    * @pre balance() + owner().salary() >=
    *                              amount
    * @post balance()@pre >= amount implies
    * balance() == balance()@pre - amount
    * @post balance()@pre < amount implies
    * balance() == balance()@pre -
    *                      amount*(1+tax())
    public void withDraw(double amount){
           ...
    }
} end of class SpecialAccount
```

The SpecialAccount withDraw pre-condition - call it `preSP` - follows the "cannot be stronger" rule, because it is weaker than the one in class Account - call it `preAcc`. The set of states in which this operation can be invoked contains the set of states defined by `preAcc`.

In order to the post-condition to obey the "cannot be weaker" rule, it should imply the withDraw post-condition in class Account. Let us see that this is not the case.

Let X be the condition `balance()@pre >= amount`

Let Y be `balance() == balance()@pre - amount*(1+tax())`

Let Z be `balance() == balance()@pre - amount`

To prove that the SpecialAccount withDraw post-condition - call it `postSP` - implies the Account withDraw one - call it `postAcc` - we have to prove:

((*not* X *implies* Y) *and* (X *implies* Z)) *implies* Z

For this purpose, since it is impossible to have both Y and Z (unless tax()=0, but this is irrelevant here), we will consider that Y is (*not* Z), so we have to prove:

((*not* X *implies not* Z) *and* (X *implies* Z)) *implies* Z

But this does not hold. Thus, the condition `postSP` is not equal to nor stronger than `postAcc`, and it would be rejected according to the rules of contract construction of the mentioned existing approaches.

But we claim that this is a perfectly reasonable redefinition of the semantics of the withDraw method. At all states that verify `preAcc`, the invocation of the withDraw method over a SpecialAccount object results in a state that verifies `postAcc`. This ensures that, in what withDraw is concerned, any SpecialAccount object behaves exactly in the same way as Account objects do in all states that are defined in class Account as being admissible withDraw states. This implies that any SpecialAccount object can be substituted for an Account object.

The semantics of the withDraw operation only changes for invocations in states that do not satisfy `preAcc`. This seems reasonable.

These are states that all Account clients know to be forbidden to withdrawal operations - thus they have no expectations on what kind of result they would get. If they call the withDraw method over an Account reference *r* in one of these states, they should be aware that two things may happen: either *r* makes reference to an Account object and the result is undefined, or *r* makes reference to an object of one of the Account's subclasses and the result is as defined by the post-condition of that subclass withDraw operation (as long as that state verifies the corresponding pre-condition, as is the case with SpecialAccount). Thus, let the Account subclasses define what is to be done in these states.

Let us now suppose that the reader agrees with the reasonableness of `postSP`. If we want to write these classes and respective assertions using the Eiffel language, for example, we have to be able to write the `postSP` condition in the form (`postAcc` *and* `S`) in order to find the assertion `S` that would appear after the `ensures then` clause of the Eiffel method declaration (other assertion languages, like iContract, also require this). But we cannot find any assertion `S` such that (`postAcc` *and* `S`) is equivalent to `postSP`! Thus, none of these languages would be able to cope with this kind of less restrictive post-conditions. Not even ContractJava and similar languages, for which `postSP` would fully constitute the post-condition for SpecialAccount withDraw operation, would accept `postSP`, because it simply does not entail `postAcc` - it would issue a hierarchy error.

3. Redefinition - Subtyping

The above claims can be justified in a more rigorous way if we use an algebraic approach to types and subtyping as, for example, in [12]. We will show that our claim is coherent with the notion of sub-type conformant behaviour. Let the following be (part of) a representation of the abstract data types Account and SpecialAccount. We omit several operations and axioms of these ADTs that are not needed for the purpose of this paper.

```
Specification Account
Operations
    ...
    balance: Account → num
    owner: Account → Person
    withDraw: Account num → Account
    ...
Axioms (forall a:Account; amm:num)
    balance(withDraw(a,amm))=balance(a)-amm
                          if balance(a)≥amm
    ...
End of spec


Specification SpecialAccount ≤ Account
Operations
    ...
    tax: SpecialAccount → num
```

```
    withDraw: SpecialAccount num →
                          SpecialAccount
    ...
Axioms (forall a:SpecialAccount; amm:num)
    balance(withDraw(a,amm))=
    ((balance(a)-amm  if balance(a)≥amm)
    and
    (balance(a)-amm*(1+tax(a)) otherwise))
    if balance(a)+salary(owner(a))≥amm
    ...
End of spec
```

The semantics of the operation that is of interest here - the withDraw operation - is given by showing the results of query functions applied to an withdrawn account of the type in question. The only query that is affected by the withDraw operation is the `balance()` one thus, the other queries do not appear in axioms of the type. In type Account, the balance of an account that results from withdrawing a quantity `amm` from an account `a`, is only defined if the balance of account `a` is not smaller than `amm`. It is undefined for all other cases. In type SpecialAccount, the balance of an account that results from withdrawing a quantity `amm` from an account `a`, is only defined if the balance of account `a` summed with the its owner's salary is not smaller than `amm`. If this is the case, the resulting balance depends on the relation between the balance of account `a` and `amm`.

The axiom refinement that happens in a subtype B (of a given type A) due to the redefinition of some operation m, must logically entail the original axiom, in order to maintain sub-type conformant behaviour. Let us prove that the refined axiom in SpecialAccount entails the one in Account.

For simplicity, let us rename several parts of the conditions above in the following way:

This letter	stands for	this term
A		balance(withDraw(a,amm))
X		balance(a)-amm
Y		balance(a)-amm*(1+tax(a))
b		balance(a)
so		salary(owner(a))

We prove that the axiom in class SpecialAccount, which is

b+so≥amm *implies* ((b≥amm *implies* A=X) *and*

(b<amm *implies* A=Y)) (1)

entails the one in class Account, which is

b≥amm *implies* A=X (2)

From (1) we have,

(b+so≥amm *implies* (b≥amm *implies* A=X)) *and*

(b+so≥amm *implies* (b<amm *implies* A=Y))

Thus, we also have,

(b+so≥amm *and* b≥amm) *implies* A=X) *and*

(b+so≥amm *and* b<amm) *implies* A=Y)) (3)

Because (b+so≥amm *and* b≥amm) is (b≥amm), from (3) we have,

(b≥amm *implies* A=X) *and*

(b+so≥amm *and* b<amm) *implies* A=Y))

Which clearly implies (2). Thus, we have proved that SpecialAccount is subtype conformant with Account in what concerns the withDraw operation.

4. Weakening the Rules for Contract Specification

To the best of our knowledge, the languages that allow the definition of assertions, be it the programming language itself as is the case of Eiffel, be it special-purpose languages based in some OO programming language with some added constructs as are the cases of iContract [6], JavaContract [5] and others that have Java as their base language [1,2,4,8,9], and [3,13] that have C++ as the base language, use Hoare style semantics for pre and post-conditions. This is to say, the semantics of the post-condition depends on the pre-condition in the sense that it denotes a condition that is only worthwhile evaluating after the execution of its associated method from a state that verifies the pre-condition.

The approach taken by [10], where the above kind of problems are solved by guarding post-conditions, is strange and somewhat redundant insofar as they propose to guard each post-condition with the corresponding pre-condition: "To support redefinition of features, guard each post-condition clause with its corresponding pre-condition. This allows unforeseen redefinitions by those developing subclasses". Following this approach, the post-condition of method withDraw for the above Account class would be the following,:

```
/**
* @pre balance() >= amount
* @post balance()@pre >= amount implies
*         balance() == balance()@pre - amount
public void withDraw(double amount){...
}
```

Likewise, the post-condition of withDraw in class SpecialAccount should be something like the following:

```
/**
* @pre balance() + owner().salary() >= amount
* @post balance()@pre+owner()@pre.salary()@pre
*              >= amount  implies
* (balance()@pre < amount implies
* balance() == balance()@pre-amount*(1+tax()))
public void withDraw(double amount){...
}
```

We consider here that the disjunction of pre-conditions and the conjunction of post-conditions is automatically done in subclasses.

Apart from bringing extra, and redundant work, it only succeeds in its goal - to allow unforeseen redefinitions - if all classes are specified with guarded post-conditions. If a developer trying to specify the contracts for the methods of some extended class, reaches a situation in which he cannot specify some post-condition because the parent class is not equipped with guarded post-conditions, most certainly he will give up from building contracts for his class. It seems obvious to us that, instead of adding difficulties to the task of building contract specifications, we should facilitate the most, while keeping adequate and powerfull contract semantics.

The approach we advocate in this paper goes towards the weakening of the rule that asks that the post-condition of the son's method implies the post-condition of the father's one. Our solution, described below, is secure insofar as it respects the substitution principle, and does not bring extra burden to developers.

Let `preA` and `postA` be the pre and post-conditions of `m` in class A, and `preB` and `postB` be the pre and post-conditions that specify the behaviour of that same method `m` when redefined in a subclass B of class A. We claim that the only restrictions that should be enforced when redefining operation *m* in subclass B, should be that:

1. `preA` implies `preB` - good old "don't ask for more";
2. if `preA` is true at invocation time, then `postA` must be true at exit time - good old "don't offer less" (no need to restrict more than this).

These two rules ensure the substitution principle: every B can be substituted for an A, because it does not refuse execution (the pre-condition is not stronger), and it does at least the things that A does in the situations where A is able to execute the method. All other situations - the ones not described by `preA`, but allowed by `preB` - are not subject to any hierarchy control at all, because they are novel situations in what concerns the parent A class.

Developers only have to ensure, when specifying contracts for redefined methods, that they do not strengthen the pre-condition and that, under the conditions of the parent pre-condition, the new semantics ensures, at least, the same behaviour.

We are talking here about the mechanisms tools should or should not have in order to prevent wild changes in the semantics of methods, and thus keeping hierarchy correctness in what concerns the substitution principle. It is obvious that the monitoring tools must always verify that pre-conditions of methods are verified when methods are called, and that post-conditions are verified after methods have been executed. So, when method m of the son class is invoked, its pre-condition `preB` is verified, as well as its post-condition `postB` at the time m finishes execution.

In order for the proposed approach to be feasible, the automatic or-ing of pre-conditions, and and-ing of post-conditions has to be abandoned. We have already shown that, in some cases, one cannot find the conditions which should be and-ed to the parent post-condition in order to obtain the ones we desire. The automatic replacement that tools like Eiffel, and others already referred to, make, prevents us from monitoring many correct contracts.

We capitalize on the arguments of the authors of ContractJava [5] when they argue that monitoring tools should check the developer's original contracts, because checking the ones that are obtained from or-ing and and-ing the father's contracts with the assertions the developers think to be the ones that give the aimed result, can mask developers and programmers errors.

In what concerns tools like ContractJava, where contracts are exactly the ones that are written in each method, and where the tool issues a hierarchy error every time the "don't ask for more, don't offer less" rule fails, it would suffice to change the implication rule that it uses for the post-conditions (`postB` implies `postA`); our second rule would be used instead.

5. Conclusions

The capacity of redefining methods in subclasses, combined with the use of polymorphic entities gives OO languages much of their strength. And because these are powerful mechanisms, they should be used with careful. Nothing in the most common OO languages prevents us from completely changing the semantics of some method when we redefine it in a subclass. Unless we have some way of enforcing the substitution principle, in which a son object can always be substituted by a parent one, this brings the possibility of building quite unreliable systems.

The use of contracts to specify, document, and verify our classes is growing within the OO community, and the ease with which one can specify contracts is a key issue to keep this growing tendency.

Contracts are also a great tool to control method redefinition, insofar as they define the semantics of methods. One can impose restrictions to pre and post-conditions of redefined methods in order to cope with the substitution principle. The *golden* rule proposed in [Meyer] to ensure the substitution principle - "don't ask for more, don't offer less" - has been widely adopted by the community, and many of the existing assertion monitoring tools have mechanisms that apply this rule to method redefinition.

In this paper we showed that these mechanisms are much too restrictive, insofar as they do not make possible the existence of certain method redefinitions which are perfectly correct and reasonable. The difficulties that arise if one wants their "illegal" contracts to be accepted by the tools, most certainly will cause one to give up from using contracts.

We proposed the weakening of one of the widely used rules in method redefinition control. As a consequence, we obtain the natural inclusion of those above forbidden contracts, while maintaining the substitution principle.

6. REFERENCES

[1] D.Bartezko, C.Fischer, M.Moller and H.Wehrheim: Jass-Java with assertions, in Workshop on RunTime Verification held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01, 2001.

[2] Y.Cheon and G.T.Leavens, A Runtime Assertion Checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun (eds.), International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada. CSREA Press, pp. 322-328, June 2002.

[3] C.K.Duby, S.Meyers and S.P.Reiss, CCEL: A Metalanguage for C++, In USENIX C++ Technical Conference Proceedings, pp.99-115, Portland, USA. Aug. 1992.

[4] A.Duncan and U.Holzle, Adding Contracts to Java with Handshake, Technical Report TRC98-32, Dept. of Computer Science, Univ. of California, Santa Barbara, CA, Dec. 1998

[5] R.B.Findler and M.Felleisen, Contract Soundness for Object-Oriented Languages, OOPSLA 2001.

[6] iContract HomePage. http://www.reliable-systems.com/tools/iContract/iContract.htm.

[7] H. B. M. Jonkers, Upgrading the Pre- and Postcondition Technique, VDM Europe (1), pp.428-456, 1991.

[8] M.Karaorman, U.Holzle and J.Bruno, jContractor: A Refelective Java Library to Support Design by Contract, In Pierre Cointe (ed), Meta-Level Architectures and Reflection, Second International Conference on Reflection'99. Saint-Malo, France, July 1999, Proceedings, vol 1616 LNCS, pp.175-196. Springer-Verlag, July 1999.

[9] G.T. Leavens, K.R.M. Leino, E. Poll, C. Ruby, and B. Jacobs, JML: notations and tools supporting detailed design in Java, OOPSLA 2000 Companion.

[10] R.Mitchell, J.McKim, Design by Contract, by Example, Addison-Wesley, ISBN 0-201-63460-0, 2002.

[11] B.Meyer, Object-Oriented Software Construction, 2nd edition, Prentice-Hall PTR, ISBN 0-13-629155-4, 1997.

[12] A.J.H.Simons, The Theory of Classification, Part 5: Axioms, Assertions and Subtypes, in Journal of Object Technology, vol. 2, n0 1, January-February 2003, pp. 13-21. http://www.jot.fm/issues/issue_2003_01/column2.

[13] D.Welch and S.Strong, An Exception-Based Assertion Mechanism for C++, Journal of Object-Oriented Programming, 11(4), pp.50-60, Jul/Aug 1998.