Responsible Correctness – an Approach to Deal with the Indirect Invariant Effect

Isabel Nunes

Lisbon University, Bloco C5, Piso 1, Campo Grande,
1749-016 Lisboa, Portugal
in@di.fc.ul.pt

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. Class correctness, as defined in [13], does not cope with the Indirect Invariant Effect (IIE), that is, the violation of an object's invariant by an instance of another class. Moreover, due to the IIE, most assertion languages check invariant violation in a way that is unsatisfactory wrt several criteria. In this paper we present the notion of Responsible Correctness with which we define class correctness wrt to a specification, and show that it deals correctly with the indirect invariant effect. The notion of responsible correctness gives us an insight on how to monitor invariant violations in a way that is most satisfactory in what concerns the above criteria.

1 Introduction

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

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.

There is also a third kind of assertion - the class invariant - that tells what must be true of all the class instances throughout their entire lives. It is the responsibility of the supplier class to establish the invariant whenever a new instance is created, and not to let its clients falsify it. This is usually done by not letting attributes to be directly modified by clients but, instead, through specific set methods (attributes are tipically private entities accessible only through public methods).

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 [13], are included in the programming language itself; others, like iContract [9], 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 [2], ContractJava [8], JML [12, 4], jContractor [11], Handshake [7] for Java only, and [6,18] for C++, [3] for Smalltalk, [1] for .NET, among them. Some of these – Jass, iContract, Eiffel – as well as others, allow the monitoring of contracts at runtime.

1.1 The problem

In [13] these assertions constitute a basis for defining correctness of a class: a class is correct if and only if its implementation, as given by the methods bodies, is consistent with the invariants, pre- and post-conditions. That is, if all constructor methods, when executed in a state that satisfy their pre-conditions, establish the invariant and their post-conditions, and if all other visible methods, when executed in a state that satisfy their pre-conditions, establish their post-conditions and preserve the invariant.

This notion of correctness does not cope with the problem of Representation Exposure, where objects inside the representation of an object *o* may be referenced, and thus modified, by objects outside of *o*'s representation. The reason is that the above correctness definition only imposes the preservation of a class's invariant on methods of that class. But, because an object internal representation can be modified by means other than its class interface (in spite all its attributes being private), the Indirect Invariant Effect, as described in [13], may arise. This means that an object's invariant may be violated through indirect access to its representation.

1.2 Existing Approaches to the Problem

Class correctness is run-time checked in [13] by way of checking pre-conditions on method entry, post-conditions on method exit and invariants on method entry and exit. Method entry invariant checking, although seemingly redundant, is the way chosen by Eiffel and the above languages Jass, JML [4] and iContract, to name a few, to deal with the problem of Indirect Invariant Effect.

In this way, any violation of an object's *o* invariant is caught sooner or later: even if violation is due to other object indirect effect, it will be caught on entry of the next method of *o*'s class that is called for *o*.

There are several proposals to prevent representation exposure drawbacks at design level [5,14,15], by way of narrowing the universe of use of a given object representation. The notion that representation should be protected from external access is called Representation Containment. This allows stronger guarantees about an object's invariant, insofar as the only external means for changing its representation is through the object itself.

In this way, the notion of class correctness as defined in [13] would be enough: since all Indirect Invariant Violation is forbidden, it suffices to verify whether the methods of a class preserve its own invariant. At runtime, invariants would only need to be checked at method exit. There is not yet (to our knowledge), any assertion checking mechanism that implements this view.

1.3 Our Proposal

In this paper we claim that the approach followed in [13] and in the above mentioned assertion checking tools is unsatisfactory in what concerns criteria like blame identification and performance, and we show that the mirror invariant clause approach suggested in [13] as another possible approach to the problem, does not solve it.

We also claim that the notion of representation containment is inadequate in some cases, where we cannot establish that an object *o* is part of the representation of another object *p* in the sense that *o* depends entirely on what *p* allows it to do.

We propose a new definition of class correctness that depends on a new concept here defined - responsible correctness. This new approach deals with the Indirect Invariant Effect without the drawbacks of the other approaches. The notion of responsible correctness gives us an insight on how to monitor invariant violations in a way that is most satisfactory in what concerns the above criteria

1.4 Outline of the paper

This paper has 5 sections. In section 2 we describe the Indirect Invariant Effect and the way most assertion languages deal with it, and we argue why this is not a satisfactory approach. We also describe the mirror invariant approach suggested in [13] as an alternative to method entry invariant checking, and show that it does not work properly. In section 3 we propose the concept of Responsible Correctness, and a way of writing responsibly correct classes. In section 4 we define class correctness using the concepts introduced in the previous section. We further exemplify the need for this kind of correctness. Finally, in section 5 we present the conclusions.

2 Invariant Monitoring

In [13] the correctness of a class with respect to its assertions – pre- and post-conditions, and invariants - is defined using Hoare triples:

Let C be a class, INV its invariant and pre_r and $post_r$ the pre- and post-conditions of some method r of C. Let $Default_C$ be the assertion expressing that the attributes of the class have the default values of their types. A class is correct with respect to its assertions if and only if:

C1- For any set of valid arguments x_p to a creation procedure p:
{$Default_C$ and pre_p (x_p)} $Body_p$ {$post_p$(x_p) and INV}

C2- For every exported routine r and any set of valid arguments x_r:
{pre_r (x_r) and INV} $Body_r$ {$post_r$(x_r) and INV}

From here onwards we will refer to the two clauses just defined by C1 and C2. The above definition of class correctness guarantees that a correct class never violates its invariant. Rule C1 says that all newborn instances of a correct class satisfy the invariant. Rule C2 says that all methods of a correct class preserve the invariant. A class that follows rules C1 and C2, and for which the state of its instances can only be modified through the exported class methods (that is, attributes are private) can be proven, by induction, never to violate its invariant through its methods.

When we use assertions for software test, debugging and quality assurance, we expect to be warned, during program execution, whenever an assertion is violated (through exception raising for example). The code that is generated for assertion monitoring of iContract, Jass, JML and Eiffel assertions, to name a few, evaluate pre-conditions on entry of methods, evaluate post-conditions on exit of methods; invariants are monitored on entry and exit of methods.

But, if we recall what was said above - the invariant is preserved if all constructor methods establish the invariant (on exit) and all exported methods preserve it - we would expect that invariants be monitored only on method exit.

2.1 The Indirect Invariant Effect

The reason for the need to check that invariants hold on method entry is explained in [13] as being due to the Indirect Invariant Effect. The problem is that, even declaring the attributes private, it can be the case that the invariant of an object be modified by an operation on another object. Let us see the example presented in [13] written in Java and iContract [9].

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.

```
/**
```

```
*   @invariant residence()!= null implies
*               residence().landLord().equals(this)
*/
public class Person {
    ...
    private House residence;
    ...
    public House residence() { //the person's residence
          return residence;
    }

    public void setResidence (House newHouse) {
          residence = newHouse;
          if (newHouse != null)
                residence.setLandLord(this);
    } // end of method setResidence
    ...
} // end of class Person

/**
*   @ invariant
*/
public class House {
    ...
    private Person landLord;
    ...
    public Person landLord() { //the house's landLord
          return landLord;
    }

    public void setLandLord (Person newLandLord) {
          landLord = newLandLord;
    } // end of method setLandLord
    ...
} // end of class House
```

Both classes appear to be correct. Class `Person` preserves its invariant - in method `changeResidence(...)` a call to the method `changeLandLord(...)` for its `residence` attribute is made in order to re-establish the invariant. Class House is also (vacuosly) correct in what respects its invariant. But consider the following statements:

```
Person p1 = new Person (...);
House h1 = new House (...);
...
```

```
p1.setResidence(h1);
h1.setLandLord(null);
...
```

After the execution of `p1.setResidence(h1)`, `p1` satisfies its invariant because the landlord of its own residence is `p1` itself.

	landLord	*residence*
p1		h1
h1	p1	

But, after the execution of `c1.setLandLord(null)`, `p1`'s invariant is violated: the landlord of its residence is no longer `p1`.

	landLord	*residence*
p1		h1
h1	null	

Let us suppose that the invariant is only monitored on method exit. The next call to a method of class `Person` over `p1` - say `p1.residence()` - would manipulate an object that violates its invariant, and an invariant violation would occur on `residence()` exit. Apart from the dangers that manipulating an inconsistent object can bring, this invariant violation warning would blame the `residence()` method; but the guilt is another class's method (namely the `House` class `setLandLord` method).

The idea that class `Person` is correct, due to the fact that all its methods preserve the invariant, is not enough since the invariant of a person may involve instances of other classes, and this way of proving a class correctness does not deal with the effect that these other classes' features may have on the invariant of `Person` - the Indirect Invariant Effect. This problem applies to all classes C that have an attribute which type is a class D that declares an attribute of type C, and which invariant involves a reference to an object of class D (doubly linked list are one example).

The way Eiffel deals with this problem, as well as Jass, iContract and JML [4], is by monitoring the invariant on method entry also. In this way, any invariant violation would be trapped as soon as possible, minimizing inconsistencies. Furthermore, because the violation would be caught at method entry, that method could not be erroneously blamed for that violation. We can nevertheless point out two inconveniences to this approach: i) who's to blame; ii) performance.

The blame for turning an object inconsistent cannot be decided with this approach, because the invariant violation warning is only issued after the real violation. Only when one of the methods of the class which invariant was violated is called, is a warning issued. Unless we know the exact trace of execution, we cannot assign the blame to anyone, that is, we cannot know where the bug is.

Generalizing invariant monitoring on method entry to all and every method, brings a performance penalty that could be avoided through another approach.

2.2 Mirror Invariant Clause

Meyer suggests that "*A more satisfactory solution would be to obtain a statically enforceable validity rule, which would guarantee that whenever the invariant of a class A involves references to instances of a class B, the invariant of B includes a mirror clause*". We think this approach is intrusive, insofar as it imposes an invariant into a class which specification did not asked for it. Furthermore we will show that this does not solve the problem completely.

In the example above, the application of the enforceable validity rule means that class `House` would have the invariant:

```
/**
*   @ invariant landLord()!= null implies
*               landLord().residence().equals(this)
*/
public class House {
```

The method `setLandLord()` would have to be modified in order to preserve the invariant:

```
public void setLandLord (Person newLandLord) {
      landLord = newLandLord;
      if (newLandLord!= null
                  && newLandLord.residence()!=this)
            landLord.setResidence(this);
} // end of method setLandLord
```

The condition `newLandLord.residence()!=this` has to be verified also, to avoid circular calls. The method `setResidence()` in class Person is modified accordingly:

```
public void setResidence (House newHouse) {
      residence = newHouse;
      if (newHouse != null
                  && newHouse.landLord()!=this)
            residence.setLandLord(this);
} // end of method setResidence
```

But this does not solve the problem. Let us see why. Consider the following statements:

```
Person p1 = new Person (...);
Person p2 = new Person (...);
House h1 = new House (...);
House h2 = new House (...);
p1.setResidence(h1);
p2.setResidence(h2);

...
h1.setLandLord(p2);
...
```

After the statements `p1.setResidence(h1);p2.setResidence(h2)` we would have:

	landLord	*residence*
p1		h1
p2		h2
h1	p1	
h2	p2	

After the statement `h1.setLandLord(p2)` we would have:

	landLord	*residence*
p1		h1
p2		***h1***
h1	***p2***	
h2	p2	

Both objects h1 and p2 would be consistent with their invariants: the residence of h1's landlord is h1 itself and the landlord of p2's residence is p2 itself.

But p1 and h2 are in a state that violates their invariant: the residence of h2's landlord (p2) is not h2 itself and the landlord of p1's residence (h1) is not p1 itself. Mirror clauses in invariants do not solve the problem completely: although they succeed in keeping both the current instance and their new components consistent, they do not care for keeping *old* components consistent. These old components are not necessarily garbagge, so they must be kept consistent with their invariant. Any call to a method of class `Person` over `p1` would manipulate an inconsistent object. The same apllies to `h2`.

3 Responsible Correctness

The concern should be the following: whenever a method modifies an object, it should be responsible for preserving the invariant of all the objects that are directly affected

by that modification. These are, tipically, the new and old versions of the modified reference. Following this idea we proppose the concept of Responsible Correctness wrt a Specification (invariants of the class, pre- and post-conditions of methods).

Definition (Invariant Responsible) We say that a class A is *invariant responsible* for its supplier class B if, whenever an instance of A modifies one of its B components, the invariant of B is preserved for both old and new versions of that component.

We take as supplier classes of a class A the classes to which belong all objects that are manipulated by instances of A.

Definition (Responsible Correctness wrt a specification) A class that is both correct in the sense of [13], that is, it satisfies correctness statements C1 and C2 above, and is invariant responsible for all its suppliers is a Responsibly Correct class.

We also define the notion of Responsible Correctness in what concerns a given class.

Definition (Responsible Correctness in what concerns) A class that is both correct in the sense of [13], that is, it satisfies correctness statements C1 and C2 above, and is invariant responsible for its supplier class B is a Responsibly Correct class in what concerns B.

Obviously, these definitions take as granted that class attributes are private, that is, they can only be modified through the class's exported methods.

We have already seen that mirror invariant clauses proposed in [13] are intrusive and insufficient. Our way to enforce responsible correctness does not include those invariant mirror clauses. We work at the level of post-conditions instead.

A way to enforce a class A to be responsibly correct in what concerns its supplier class B, is to have, in all A methods that modify its B component, a post-condition that verifies the invariant of that B component for both its old and new versions. This can only be verified if we have a way to know which objects are modified by a given method.

Let us extend the assertion language in use with a clause similar to the *changeonly* clause of Jass [2] or the *modifies/assignable* clause of JML [12], in order to be able to declare which are the entities that can be modified by a method. The frame condition given by the *assignable* clause of JML, for example, in a method, only allows that method to assign values to a location *loc* if:

- loc is mentioned in an assignable clause;
- loc was not assigned any value at that method entry;
- loc is local to that method;

There is one more case that has to do with JML *depends* clause which is irrelevant here because we only need to adopt a clause like the *assignable* clause for our immediate purposes.

We will consider that, in order to allow client classes to understand the *modifies* clauses, the modifiable entities that appear in a *modifies* clause are either public functions that access object attributes or parameters of the method.

Let us now recall the original Person/House problem – the one without the mirror invariant in class `House` – and change the classes in order to obtain responsibly correct classes.

```
/* @invariant residence()!= null implies
*               residence().landLord().equals(this)
*/
public class Person {
    ...
    private House residence;
    ...
    public House residence() { //the person's residence
          return residence;
    }

    /*
    *     @modifies residence();
    *     @post residence().equals(newHouse)
    */
    public void setResidence (House newHouse) {
          residence = newHouse;
          if (newHouse != null
                      && newHouse.landLord()!=this)
                residence.setLandLord(this);
    } // end of method setResidence
    ...
} // end of class Person

/* @invariant
*/
public class House {
    ...
    private Person landLord;
    ...
    public Person landLord() { //the house's landLord
          return landLord;
    }

    /*
    *  @modifies landLord();
    *  @post landLord().equals(newLandLord)
    *     && landLord()@pre.residence()!=null implies
    *           landLord()@pre.residence().landLord().
    *                                 equals(landLord()@pre)
    *     && landLord().residence()!=null implies
    *           landLord().residence().landLord().
    *                                 equals(landLord())
    */
    public void setLandLord (Person newLandLord) {
          if (newLandLord!= null)
```

```
            landLord.setResidence(null);
        landLord = newLandLord;
        if (newLandLord!= null
                    && newLandLord.residence()!=this)
            landLord.setResidence(this);

    } // end of method setLandLord
    ...
} // end of class House
```

Notice that the invariant of class House is empty as in the original version of the problem, that is, it is the assertion `true`.

In class `Person`, the post-condition of method `setResidence()` asserts that the new value for the residence is the parameter value. We do not have to assert in the post-condition that the Person's invariant is established at method exit, because we know that class invariants are checked at method exit. Because this method has the clause `@modifies residence()`, its post-condition should, accordingly, verify the invariant for its old and new residences: it does so in a vacuous way because this invariant is the assertion `true`. It does not matter whether the old residence has the current person as its landlord or not, because there is nothing in class House that forbids it. So, class `Person` is responsibly correct in what concerns class House.

The method `setLandLord()` of class `House` announces to the world that it only modifies the `landLord()` object: it does this through the `@modifies landLord()` clause. Accordingly, we have in its post-condition two clauses that verify the `Person`'s invariant both for the new and old values of `landLord()`

```
*   landLord()@pre.residence()==null ||
*         landLord()@pre.residence().landLord().
*                              equals(landLord()@pre)
*   && landLord().residence()==null ||
*         landLord().residence().landLord().
*                                   equals(landLord())
```

In this way, class `House` is made responsible for every modification of its landLord objects - the old and the new. It must modify the old landlord in such a way that it verifies its invariant, that is, it no longer owns the current house (it owns no house, in fact). It must modify its new landlord in such a way that it verifies its invariant, that is, it is either null, or the landlord of its own residence is himself. So, class `House` is responsibly correct in what concerns class `Person`.

These post-conditions are useful in at least three ways: i) they allow to verify responsible correctness of the class they belong to and, therefore, to verify correctness of supplier classes; ii) they help implementors of the corresponding methods in the task of code writing; iii) they allow run time checking of their classes' responsible behaviour.

The assertion in the post-condition of the method `setLandLord()` of class `House` is somewhat complicated because it has to assert that the invariant of class `Person` is true for both the old and new versions of the modifiable feature `landLord()`. If our aim is to assert in post-conditions the truth of other classes' invariants for specific instances, we should have an easy and secure way of doing this. The task of writing another class invariant applied to a certain feature may introduce errors. Moreover, when it applies to polymorphic references, it may be the case that the invariant that is being checked is weaker (the one in the parent class of the target object class) than it should be. We use meta-assertions [16,17] for this task, more specifically, meta-invariants. A meta-invariant is a (meta-)assertion that denotes another class invariant. A meta-invariant applied to a feature *f* denotes the invariant of *f*'s class applied to *f* itself.

In the example above we would write the post-condition of the `setLandLord()` method in the following way:

```
*  @post landLord().equals(newLandLord)
*     && (landLord()@pre)>>inv
*     && landLord()>>inv
```

the meta-assertion `landLord()>>inv`, for example, denotes the invariant of the dynamic type of the `landLord()` reference. This is exactly the same as the assertion seen above when the dynamic type of `landLord()` reference is `Person`:

```
*   landLord().residence()!=null implies
*         landLord().residence().landLord().
*                                  equals(landLord())
```

In [16,17] a semantics is given to meta-assertions and processes of generating simple checkable assertions from meta-assertions are shown and proved sound wrt the semantics.

Proposition 1. A class that verifies the correctness clauses C1, C2, and C3 is a responsibly correct class, where C1 and C2 are as above, and C3 is the following correctness clause:

C3. For any feature (function or attribute) *f* that appears in a *modifies* clause on a method *m*, the assertions

`(f@pre)>>inv` and `f>>inv`

are part of *m*'s post-condition.

By definition, a responsibly correct class A must be Invariant Responsible for all its suppliers. This means that, whenever an instance of A modifies one of its B components, the invariant of B is preserved for both old and new versions of that component. Clause C3 above enforces invariant responsibility of a class A towards its suppliers, insofar as it asserts that all methods of A that modify some feature *f* must verify *f*'s class invariant for both old and new versions of *f*.

4 Class Correctness – a Responsible Approach

We have already seen that C1 and C2 only guarantee that the methods of a class A do not violate the invariant of A's own instances. But we have seen that it can be the case that instances of A's supplier classes B, C, ... can indirectly violate A's invariant. Thus, we can only guarantee that A's invariant is not violated if we can be sure that none of B, C, ... classes can modify A's instances while violating A's invariant.

The above definitions of responsible correctness allow us to define correctness wrt a specification.

Definition (Correctness wrt a specification) If a class A is responsibly correct wrt to a specification and all its supplier classes are responsibly correct in what concerns A, then A is correct.

With this notion of class correctness we do not need invariants to be checked at method entry anymore in order to cope with the Indirect Invariant Effect. We only need pre-conditions to be verified on method entry, and both invariants and post-conditions to be verified on method exit.

The performance penalty for runtime checking assertions will be less: instead of verifying a class C invariant in each and every C method entry and exit (even for non-problematic classes or methods), we only have to check C invariant at C methods exit and, for C methods that modify non-primitive entities, verify, at exit only, these entities invariant for their old and new versions.

The *who's to blame* problem is solved by this approach: every invariant violation will be detected exactly in the method where it occurs (in its post-condition) allowing to put the blame on the guilty method/class – the bug is framed and, thus, easier to discover.

We go back now to the notion of representation containment in 1.2 and its use as a way to manage representation exposure in order to avoid indirect invariant effect [5,14,15]. Following is an example where this concept is shown much too restrictive due to the inadequacy in considering one object as dominator and the other as dominee.

```
/* @invariant mate()!= null implies
*                    mate().livesWith(this)
*/
public class Person {
    ...
    private Person mate; //the person who this lives
                         // with
    ...
    public Person mate() { //the person's mate
          return mate;
    }

    /*
    *     @pre p!=null
    */
```

```
    public boolean livesWith(Person p) {
          return mate.equals(p);
    }

    /*
    * Changes the mate of the current person
    */
    public void setMate (Person newMate) {
          ...
    } // end of method setMate
    ...
} // end of class Person
```

A person does not have to be part of another person. Two persons *p1* and *p2* should be able, independently and by themselves, to choose who to live with. We cannot see as adequate, in these kind of cases, the imposition of a dominator and a dominee. So, it should be possible to execute the following piece of code:

```
Person p1,p2,p3;
...
p1.setMate(p2);
...
p2.setMate(p3);
...
```

Due to the fact that the invariant of class `Person` refers to properties of its `mate()`, the method `setMate()` applied to a person `p1` must modify `p1`'s mate in order to preserve `p1`'s invariant. However, as we saw in section 3, this is not enough if we want to make `p1` responsible by the modification it operates on its mate. Object `p1` should preserve the invariant of both its old and new mates:

```
/*
*     @modifies mate();
*     @post mate().equals(newmate)
*           && (mate()@pre)>>inv
*           && mate()>>inv
*/
public void setMate (Person newMate) {
      if (mate.livesWith(this))
                  mate.setMate(null);
      mate = newMate;
      if (newMate != null
                   && !newMate.livesWith(this)
            newMate.setMate(this);
} // end of method setMate
```

After the execution of `p1.setMate(p2)`, `p1` satisfies its invariant because the person with whom its mate lives is `p1` itself.

	mate	*lives with*
p1	p2	p2
p2	p1	p1

After the execution of `p2.setMate(p3)`, all is well again because this method preserves the invariants of `p1` (`p2`'s old mate) and of `p3` (`p2`'s new mate).

	mate	*lives with*
p1	null	-
p2	p3	p3
p3	p2	p2

Class `Person` is responsibly correct in what concerns its unique supplier: class `Person` itself.

5 Conclusions and Further Work

The Indirect Invariant Effect arises whenever an object's invariant is violated due to the manipulation of its representation by some way, other than its interface. This effect is one of the problematic consequences of the representation exposure characteristic of object-oriented languages. In this paper we claimed that the way most assertion checkers deal with this problem at the level of class testing, is not satisfactory insofar as the identification of infractors is not possible and, thus, bugs are not framed. Moreover, the strategy they adopt to impose class correctness brings a heavy performance penalty.

We also claimed that the approaches that adopt representation containment, where the representation of an object can only be accessed through that owner object, are too restrictive for some cases, where it is unatural to establish that an object is part of the representation of another object in the sense that the former depends entirely on what the latter allows it to do.

In this paper we proposed the concept of Responsible Correctness as a means to define class correctness wrt a specification that correctly deals with the Indirect Invariant Effect. By making classes responsible for the modifications they operate on their suppliers, we can assert class correctness without the restrictions and drawbacks of other approaches. At the level of class specification, this concept is applicable in a natural way as presented in section 3. At the level of class design, it allows to avoid representation containment, enabling true object animation as exemplified in section 4. At the level of testing, it allows to surpass the inconveniences that the other approaches have, as seen in section 4.

References

1. K.Arnaut and R.Simon, The .NET Contract Wizard: Adding Design by Contract to Languages other than Eiffel, In Proceedings of TOOLS 39, 2001, California, pp.14-23. IEEE Computer Society, 2001.
2. 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.
3. M.Carrillo-Castellon, J.Garcia-Molina, E.Pimentel and I.Repiso, Design by Contract in Smalltalk, Journal of Object-Oriented Programming, 9(7), pp.23-28, Nov/Dec 1996.
4. 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.
5. D.G.Clarke, J.M.Potter, and J.Noble, Ownership Types for Flexible Alias Protection. In Proceedings of OOPSLA, vol 33(10), ACM SIGPLAN Notices, October 1998.
6. 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.
7. 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
8. R.B.Findler and M.Felleisen, Contract Soundness for Object-Oriented Languages, OOPSLA 2001.
9. iContract HomePage. http://www.reliable-systems.com/tools/iContract/iContract.htm.
10. H. B. M. Jonkers, Upgrading the Pre- and Postcondition Technique, VDM Europe (1), pp.428-456, 1991.
11. 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.
12. 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.
13. B.Meyer, Object-Oriented Software Construction, 2nd edition, Prentice-Hall PTR, ISBN 0-13-629155-4, 1997.
14. P.Muller and A.Poetzsch-Heffter, A Type System for Controlling Representation Exposure in Java, In S.Drossopoulou et al (eds) Formal Techniques for Java Programs, 2000. Technical Report 269, Feruniversitat Hagen.
15. P.Muller, A.Poetzsch-Heffter, and G.T.Leavens, Modular Specification of Frame Properties in JML, Technical Report 01-03 Dept. of Computer Science, Iowa State Univ, April 2001.
16. I.Nunes, Design by Contract Using Meta-Assertions, in Journal of Object Technology, vol. 1, no. 3, special issue: TOOLS USA 2002 proceedings, pages 37-56. http://www.jot.fm/issues/issue_2002_08/article3.
17. I.Nunes, Polimorphism in (Meta)-Contract Verification of Object-Oriented Systems, October 2002, submitted.
18. 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.