Interpretation of Contracts and the Fragile Base Class Problem

Sonal Bhagat, Rushikesh K. Joshi *

Department of Computer Science and Engineering

Indian Institute of Technology, Bombay

Powai, Mumbai - 400 076, India.

Abstract

The fragile base class problem arises when the contract between the base class and the derived class is violated by the revision class. This paper focuses on the role of contracts in the fragile base class problem. The fragile base class problem plays a very important role in component systems. The user, to whom a component system is delivered, makes extensions to the given component system using inheritance in order to suit the requirements. In order to make meaningful extenions, the user makes use of certain implementation details that we call *Hidden Contracts* along with the conventional interface contracts. Also, the developer revises the base class implementation keeping the interface unchanged. We emphasize that in order to avoid the fragile base class problem, the developer should abide by both, i.e. convensional contracts and *Hidden Contracts* between the original component and the extended component. In this paper, we have taken examples from literature to show the cases of fragile base class problem due to violation of *Hidden Contracts*. Secondly, fragile base class problem may occur because of expansion of the convensional contract by the developer without involving the user. Finally, the problem may also occur because of the exclusion of a part of the convensional contract. Specifying and abiding to *Hidden Contracts* and convensional contracts avoids the fragile base class problem and the same time provides flexibility for evolution.

1 Introduction

The fragile base class problem plays a very important role in component systems from the point of view of system developers since a system that worked with original base classes may not work with their refinements. Mauth [Mau96] classifies the fragile base class problem into syntactic and semantic categories. The syntactic problem occurs whenever a new method or a new instance variable is added to a base class and due to which, all the classes referring to that class need to be recompiled. For example, in C++ the problem exists, whereas, in Smalltalk,

* *Contact Author Email: rkj@cse.iitb.ac.in*

it is nonexistent since the types are resolved at runtime. In JAVA, the problem does not exist [JG96], but it still persists in the JIT compiler as it too performs some static binding similar to a C++ compiler. The semantic fragile base class problem on the other hand concerns with semantics of classes, specifically consistency between refinements and extensions. The paper deals with this category of the fragile base class problems.

A component system may undergo evolution in two directions of extension and refinement. The *users* extend the functionality provided by a base class system by means of specialization, whereas, the *system developers* refine the implementation behind the agreed interfaces. The system developers are unaware of the extensions developed by the users and try to improve the functionality of the system by producing certain revision classes for the corresponding base classes. These revision classes are semantically and independently (without the consideration of derived classes) compatible with the corresponding base classes. But these revision classes may break in presence of the extensions to their earlier versions. This problem is known as the fragile base class problem [LM98].

Throughout the document we will use the term *developer* to denote the system developer who refine the base classes, and the term *user* to denote the programmers who extend the base classes to produce extensions. We investigate into the psychology of users and developers by virtue of which both independently believe in their work but when combined, the system fails. The reason can be traced to interpretation of *contracts*. While extending or refining a base class system, the user and the developer must abide by the contracts. With the help of examples of fragile base class problem in the literature, we analyze the problem to show that the contract is violated due to its varied interpretations, thereby causing a mismatch between the refinements and the extensions. The contract violations are classified in the categories of hidden contract voilation, interface contract expansion and interface contract exclusion as described below.

Hidden Contract Violation: The user may indirectly make use of certain implementation details of the base class while producing an extension. *Hidden Contracts* are these implementation details since they are not apparent from the interface abstractions. The developer may break the hidden contract by changing the implementation details. We argue that in order to eliminate the fragile base class problem, the developer must abide to these *hidden contracts* apart from the conventional interface contracts such as basic, behavioral, synchronization and QoS contracts [AB99], [CD99], [Mey97]. Hook exclusion and hook inclusion are two examples of hidden contracts, which are discussed in Section 4.

Interface Contract Expansion: The developer may independently add new features to the interface contract which interact with existing features. This can make the existing derived classes inconsistent with the base class refinements by the developer. Section 5 provides an example of this category.

Interface Contract Exclusion: It is possible that the users or the developers do not consider some of the features of the interface contracts and cause an inconsistency. Section 6 provides an example of this category.

2 Earlier Work on the Fragile Base Class Problem

The fragile base class problem (FBP) has been explored in the literature. Inheritance is seen as a cause of the semantic fragile base class problem as it breaks encapsulation. Leonid Mikhaljov and Emil Sekerinski [LM98] have stated the fragile base class problem in terms of a flexibility property. Realizing the shortcomings of inheritance, they have proposed certain requirements for disciplining inheritance by imposing certain restrictions on it to permit the safe substitution of a base class by the revision class in presence of extension classes. In [LM97], the authors have given three checklists for component framework designers, component framework developers and its users to avoid the fragile base class problem. While disciplining inheritance would solve the problem, our focus is on contract agreements rather than disciplining, since the latter may hamper flexibility. If it has to be argued that FBP occurs due to contract violations and a contract agreement would solve the problem, it is important to investigate the occurrences of the FBP from the point of view of contracts.

It is found that if the contracts are agreed upon and are followed by the users and the developers, FBP can be is eliminated. At the same time, certain refinements or extensions can be made possible which otherwise would not have been, had the inheritance been disciplined according to strict disciplining guidelines. Thus disciplining can be seen as the only solution in absence of clearly defined contract agreements, whereas if contracts are affordable, the system may obtain additional benefits from the point of evolvability.

3 Primitives and Diagrammatic Representation

In the subsequent sections, the examples from [LM98] have been analyzed in detail. The following primitives and notations have been used in the analysis.

1. **Hooks** : When the derived class method, say $n0$ overrides the base class method n, n acts as a hook. Thus n is replaced by $n0$. This is represented as: $n \xrightarrow{r} n0$. In this case the properties of n are replaced by $n0$. This is represented as: $\mathrm{p}[n] \xrightarrow{r} \mathrm{p}[n0]$. The code of execution realizes the properties of a method. In C++, a hook is represented as a virtual method.

2. **Method Calls** : When a method, say n, calls another method say m, it is represented as: $n \Longrightarrow m$. In this case the properties of n are the superset of the properties of m. This is represented as: $\mathrm{p}[n] \supset \mathrm{p}[m]$.

3. **Delegation** : When a method, say n delegates to another method say m, it is represented as: $n \xrightarrow{d} m$. We can say that delegation is a special case of method call where no other computation is carried out in the calling method and the processing is delegated to the callee method.

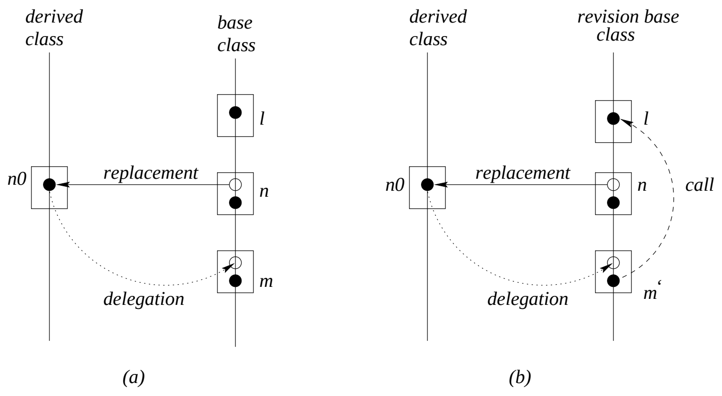

Figure 1: Diagrammatic Representation of Primitives

A diagrammatic representation of these primitives is shown in Figure 1. A vertical line represents a class. In Figure 1(a) the left line represents the derived class and the right line represents the superclass. Figure 1(a) shows the original scenario with the superclass and the derived class. Figure 1(b) shows the scenario with the revision class and the derived class. A box on the line represents an exported method of the class. The filled circle represents the concrete part that is the implementation and the hollow circle indicates that the method is a hook method.

A derived class method may override a superclass method. Here *n0* overrides n. Also, a superclass method may be modified by the revision class. Here m is modified to $m\prime$. A method call is represented by a dashed arc, with the pointer pointing to the called method. Also, a delegation is represented by a dotted arc, with the pointer pointing to the delegated method.

4 Hidden Contract Violation

A derived class may base its implementation on a specific call to a hook method from within a method in the base class. Although the call may appear to be an implementation aspect to the developer, should have been a part of the contract. A call is thus a part of the hidden contract, since it is not apparent from the interface abstractions. If a call is a part of the contract, the developer should be prohibited from dropping it. Similarly, an FBP may arise due to inclusion of a call to a hook method. In this section, two examples of hidden contract violations are analyzed.

4.1 Exclusion of Call to a Hook Method

The code below shows a class *Bag*, implementing a collection class. Method *add* adds an element into the collection and method *addAll* adds more than one elements to the collection. Method *cardinality* gives the number of elements in the collection. Method *addAll* is implemented in terms of method *add*.

```
class Bag {
  add() {
   ...
  }

  addAll(){
    ...
    add();
    ...
  }
  cardinality(){
    ...
  }
}
```

```
class CountingBag{
  add(){
         count++ ;
       super.add();
    }
cardinality(){..}
}


class Bag'{
    add()      {...}
    addAll()   {...}
 cardinality{...}
}
```

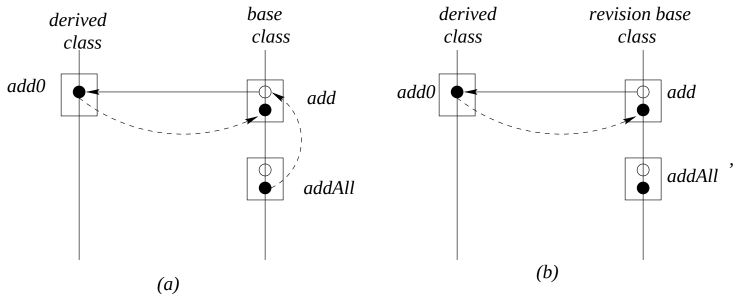

Figure 2: Bag Example : Hook Exclusion

Figure 2(a) depicts the situation of the original base class with the derived class. The user specializes *Bag* to implement a class *CountingBag*, that keeps a count of elements by introducing a new variable *count*. The user overrides *add* to increment the value of *count* to maintain the cardinality of the bag and calls *super.add* to add the element in the bag. The user is able to provide this extension because it is known that *addAll* calls *add*, so that *count* gets updated appropriately even when more than one elements are added to the bag. Thus, in this case, the call to hook method *add*, in *addAll* becomes the *hidden contract* between the base class and the derived class. We analyze this further as follows.

As *addAll* calls *add*.

addAll $\Longrightarrow$ *add*. So, p[*addAll*]$\supset$p[*add*]. -1

Also, *add* is a hook method.

add $\xrightarrow{r}$ *add0*. So p[*add*] $\xrightarrow{r}$ p[*add0*] -2

From (1) and (2), we can say that,

p[*addAll*] $\supset$ p[*add0*]. -3

Thus, (3) becomes the contract between the base class and the derived class.

Now Figure 2(b) depicts the situation after the replacement of base class by the revision class. The revision class developer improves the performance by eliminating an extra call overhead in *addAll*. Now *addAll* no more calls *add*.

So, p[*addAll*] $\not\supset$ p[*add*]

Eventually, p[*addAll*] $\not\supset$ p[*add0*] which violates (3). As *addAll* doesn't call *add* of the derived class, the value of *count* will not be incremented appropriately on addition of a group of elements, giving an incorrect cardinality of the bag. Since *add* is a hook method called in *addAll*, behavior of *addAll* in presence of the revision class relies on the extended implementation of *add*. Excluding a call to hook method *add* in the refinement of *addAll* excludes extended *add* from *addAll* in presence of the derived class. Thus the call to the hook method *add* should be considered a part of the *hidden contract*, which the revision class developer should not violate. Had *add* been a hidden concrete method (as in C++), it would not have been overridden by any derived class method and this problem would not have occurred.

4.2 Inclusion of Call to a Hook Method

The following example shows how inclusion of a hook method in a revision class method that was not included in the corresponding base class method causes a violation in the *hidden contract* between the base class and the derived class. This may bring out unexpected results like mutual recursion. The code below shows class *C*, which exports methods *m* and *n*, both incrementing a variable *x*.

```
C = class {                M = modifier{          C’ = class {
  x : int = 0                                        x : int = 0
  m() { x = x + 1}                                   m() { self.n() }
  n() { x = x + 1}          n() { self.m() }         n() { x = x + 1
}                          }                        }
```

It is clear from the description of the interfaces of *m* and *n* that they are interchangeable. Hence, as shown in the derived class below, the user overrides *n* to *n0* and delegates the call to *m*.

n $\xrightarrow{r}$ *n0*. -1

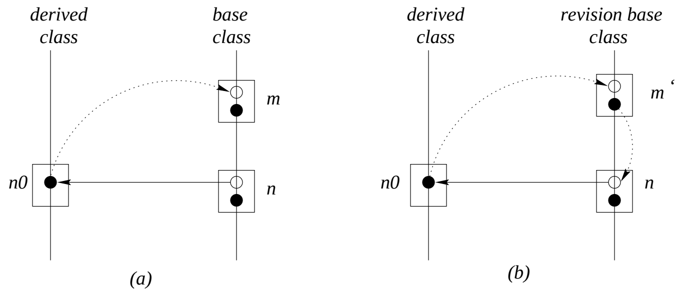

Figure 3: Mutual Recursion Example : Hook Inclusion

$n0 \xrightarrow{d} m$ -2

We argue that in order to make such an extension safely, the user knows that m in the base class is not implemented in terms of n, so that there is no possibility of mutual recursion. This fact that m does not call n is thus a part of the *hidden contract*, the contract beyond interface description. The developer should abide by this *hidden contract* to avoid the fragile base class problem.

So the *hidden contract* is :

$m \not\xrightarrow{d} n$ -3

The revision class developer maintains the interface contract while refining the base class by delegating m to n. From Figure 3(b), the following observations can be made : m has been modified to m' in the revision class.

$m \xrightarrow{r} m'$ -4

So the hidden contract in (3) now becomes

$m' \not\xrightarrow{d} n$

Now, the revision class method m' delegates to n that is a hook method, that is, $m' \xrightarrow{d} n$ which is clearly the violation of the hidden contract. Thus on calling a hook, that was not called earlier, the *hidden contract* is broken as shown and this results in mutual recursion.

5 Expansion of Contract by the Developer

In the code below, the revision class developer introduces a new variable y, and modifies the value of x through y.

```
C = class {                          M = modifier {
  x : int = 0
  m() { x = x + 1 }
  n() { x = x + 2 }                    n() { x = x + 2 }
}                                    }
```

```
C' = class {
  x : int = 0 ; y : int = 0
  m() { y = y + 1 ; x = y }
  n() { y = y + 2 ; x = y }
```

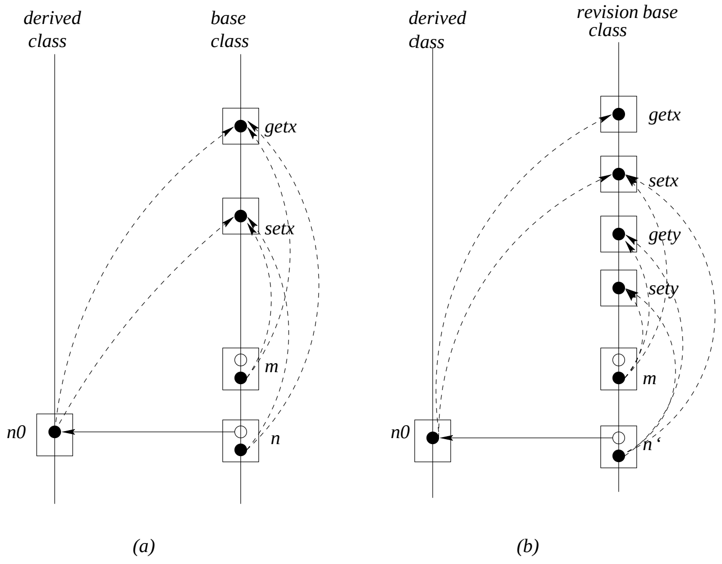

Figure 4: Contract Expansion Example

To analyze the problem, let *getx*, *gety* and *setx*, *sety* represent methods for retrieving and assigning values of x and y respectively. According to the contract between the base class and the derived class, *getx* gives the latest value of x at boundary points. The revision class developer adds to the contract by introducing *gety* and now *gety* gives the latest value of x at boundary points. But the user is not aware of this. If the user would have known this, he would have added y = x after updating the value of x, i.e. *setx* would follow *sety*(*getx*) in *n*. So the developer should not expand the contract as the user makes extensions being unaware of the expansion in contract.

6 Contract Exclusion by the Developer

The following code shows how fragile base class problem results if the semantics specified by the base class are not properly followed.

C = class {

m(val x : real, res r: real){

pre ≥ 0

post $r^2 = x$

}

n(val x : real, res r: real){

pre ≥ 0

post $r^4 = x$

}

}

M = Modifier

m(val x : real, res r : real) {

{ r = $-\sqrt{x}$ }

}

C$\prime$ = class {

m(val x = real, res r: real) { r = $\sqrt{x}$ }

n(val x = real, res r: real) {

self.m(x,r) ; self.m(r,r)

}

}

The specification of the base class says that given a positive number x, methods m and n will find r such that r's power of 2 and 4 in m and n respectively, equals x. It does not rule out the condition that r may be negative. The derived class method m reduces non-determinism by returning a negative number which it requires for its own application. The revision class developer simply passes the value of r in the second invocation of m in revised n, thus violating the contract regarding the consideration of dealing with a negative value of r. Fragile base class problem would not have occured if the revision class developer would have sent the mod of r as the parameter to m.

7 Contract Approach Leads to Flexibility

Disciplining inheritance [LM98] and following the checklists mensioned in [LM97] avoid the fragile base class problem, but at the same time make the development of a system of classes restrictive. Instead, it is found that specifying contracts and abiding by them makes the system more flexible to evolution. Following are the examples illustrating this approach. According to the checklist for users, a method of an extension class may call only those methods that are self-called by the counterpart in the base class. Hence, in the *mutual recursion* example, the user will not call *self.m* in n as it is not called by the counterpart in the base class, in order to confirm to the checklist for users. Also, according to the checklist for revision class developers, a method of the revision class may self-call only those methods that are self-called by the counterpart in the base class. So, confirming to the checklist for developers, the revision class developer too won't call *self.n* in m as *self.n* is not called in m of the base class. Following these restrictions specified in the checklist avoids the fragile base class problem in which mutual recursion results. However, it can be seen that the checklists restrict both the user and the developer to make extensions and refinements which could be meaningful extensions in the system.

Alternatively, if the fact that *m does not call n* would have been specified as a hidden contract by the base class developer, the user could have continued with calling *self.m* in n safely, making the system flexible enough to accept this modification.

Also, according to the user checklist, the user should not always assume that in a particular layout, base class self-calls are guaranteed to get directed to itself, since the revision class may modify the self-calling structure. Because of this, the user in the *counting bag* example will not be able to make the meaningful extension provided by the *CountingBag*. Hence, this method makes the system restrictive to changes. Instead, if the fact that *addAll calls add* is specified as a hidden contract, the user can overide *add* to improve the functionality of the base class, while the developer abides by this contract.

8 Conclusion

The semantic fragile base class problem is studied from the point of view of contract violations. We have analyzed examples to show how fragile base class problem occurs because on violation of *hidden contracts*, on expansion of contracts and on improper consideration of semantics of contract by the revision class developer. The following table gives the summary of our work.

Example	Section	Contract	Contract Violation
Bag	4.1	*addAll* calls *add*	*addAll* doesn't call *add*, hook exclusion
Mutual Recursion	4.2	$m' \not\xrightarrow{d} \mathrm{n}$	$m' \xrightarrow{d} \mathrm{n}$, hook inclusion
Contract Expansion	5	*gety* not in contract	*gety* in the contract
Root	6	r may be positive or negative	Negative r not considered

Fragile base class problem will not arise if a revision class method drops a call to a concrete method that was called in the corresponding base class method or includes a call to a concrete method that was not called in the corresponding base class method. The derived class cannot override the effects of a concrete method, so if the revision class developer takes enough care to maintain semantics of code of the base class, it is sufficient. Problem arises when a revised class method includes or excludes call to a hook method. In this case, the revision class developer cannot completely rely on the semantics of the called base class method because the user may provide a different implementation to the base class method as it is a hook. If the revision class violates the *hidden contract* which is the very basis of the user extension, fragile base class problem occurs. Following the contract approach serves the dual purpose of avoiding the fragile base class problem and permitting flexibility for evolution.

References

[AB99] Noel Plouzeau Damien Watkins Antoine Beugnard, Jean Mark. Making Components Contract Aware. *IEEE Computer*, 32(7), 1999.

[CD99] Shmuel Rotenstreich Cynthia Della. Behavioral Specification of Distributed Software Component Interfaces. *IEEE Computer*, 32(7), 1999.

[JG96] Henry McGilton James Gosling. The Java Language Environment A White Paper, 1996.

[LM97] Emil Sekerinski Leonid Mikhailov. The Fragile Base Class Problem and its Impact on Components. In *WCOP97 at ECOOP97*, 1997.

[LM98] Emil Sekerinski Leonid Mikhailov. A Study of The Fragile Base Class Problem. In *ECOOP98*, 1998.

[Mau96] Rainer Mauth. Development frameworks let you build an application with reusable objects, 1996.

[Mey97] Bertrand Meyer. Design by Contract: The Lessons of Ariane. *IEEE Computer*, 30(2), January 1997.