Role of the Hidden Contracts in Fragile Base Class Problem

Sonal Bhagat, R. K. Joshi *
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Powai, Mumbai - 400 076, India.

Abstract

This paper focuses on the role of contracts in the fragile base class problem. The fragile base class problem arises when a base class developer modifies the base class and makes it compatible with the old base class, but the modified base class doesn't work with the existing derived classes. Both the user and the superclass developer should abide by the contracts in order to prevent an occurrence of this problem. While it is obvious to include the interface descriptions in the contract, more interestingly, it is observed that some of the implementation features should also be included in the contract between the superclass and the subclass. We call these implementation contracts as *hidden contracts*. We have taken two examples of fragile base classes from the literature and analyzed them from the point of view of contracts to bring out the importance of the role that the hidden contracts play in avoiding the fragile base class problem.

1 Introduction

One of the mechanisms of reuse that the component technology facilitates is inheritance. The users extend the functionality provided by a base class by specializing it with a derived class. This extension adheres to the inheritance contract between the base class and the derived class, in other words, between the base class developer and the developer of the derived class. This contract mainly includes the interface abstractions on the basis of which the users produce the extensions. The contract should be adhered to by the user while producing an extension.

Similarly, the system developers refine their superclasses for the reasons such as performance improvement. System developers who are unaware of the extensions developed by the users tend to refine the implementation of the system by producing revision classes for the corresponding existing base classes. These revision classes are independently (without the consideration of derived classes) interchangeable with the corresponding base classes. But surprisingly, these revision classes may break in presence of the user extensions to their earlier

* *Contact Email: rkj@cse.iitb.ernet.in*

versions. This problem is known as the fragile base class problem and has been analyzed extensively in the literature [LM98]. The fragile base class problem plays a very important role in component systems from the point of view of system developers who want to refine the implementations of the base classes without affecting the existing usage of their older versions in direct or extended forms. Mikhaljov and Sekerinski [LM98] have provided certain requirements for disciplining inheritance to eliminate the fragile base class problem. In [LM97], the authors have given three checklists for component framework designers, component framework developers and its users to prevent the fragile base class problem.

We have viewed this problem from the point of view of contracts in order to understand the psychology of the base class developer and the user. Since they both provide refinements and extensions independently but arrive at an unexpected result when they combine, it is safe to assume that something goes wrong in understanding the inheritance contract. While the restrictions on the inheritance to avoid this problem are known, it is now important to analyze the problem from the point of view of contracts. The restrictions may eliminate the problem, but at the same time hamper certain meaningful development. Hence, a contract based approach to inheritance would focus on exposition of certain features as contracts keeping in mind the code semantics. These features may be exploited during development cycle by both the user and the developer, while they abide by the contracts. The question now boils down to *whether or not a contract violation can be pointed out on the instances of the fragile base class problem that have been discussed, and if the answer is yes, which are those features that need to be included in the contract?*. We take two examples from Mikhaljov and Sekerinski [LM98] for our analysis and try to provide an answer.

2 Hidden Contracts

While refining the base class, although the developer adheres to the interface contract, there are certain implementation features which are not considered during refinement as part of the contract. We observe that the user requires some of the implementation details in order to produce correct extensions. We call these implementation details as *hidden contracts*. 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 [AB99], [CD99], [Mey97]. If the hidden contracts are not provided to the user, specific extensions cannot be produced, or certain undesired extensions may be produced. Thus, it is very important for the the developer and the user to abide by the contracts while taking their maximum benefit in producing extensions or the refinements to allow their code to work correctly when combined through inheritance.

3 Violation of Hidden Contracts by Hook Exclusion

The code below shows a class *Bag* implementing a collection class. Method *add* adds an element in to 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 {
  private:
      Array bag;
   public:
     Bag() { // initialize the bag data structure
        ...
      }
     virtual add(char element) { // find an empty location and add the element
        ...
     }
     virtual addAll(char bs[],size) {
        while(size){
           self.add(bs[size])
           size--;
        }
     }
     virtual int cardinality(void) { // find and return the number of elements
         ...
     }
};
```

Conventionally, the base class developer provides the interface abstractions of class *Bag* to the user. It may not be apparent that the user who extends this class may require the information that method *addAll* calls method *add*.

In the code below, the user specializes the above class 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.

We observe that *addAll* is not overridden in the extension class, since the base class implements *addAll* in terms of *add*. We argue that to make such an extension, the user has to explicitly know whether or not *addAll* is implemented in terms of *add*. In absence of this aspect of implementation in the contract, the user will not be able to correctly implement class *CountingBag* without fully re-implementing both *add* and *addAll*. We call this fact that *addAll*

```
class CountingBag : public Bag {
  private:
        int count;
        ...
  public:
    CountingBag(){
        count=0;
        ...
    }
    add(char element) {
        count++;
        super.add(element)
    }
    int cardinality(void) {
       return count;
    }
};
```

calls *add* as the *hidden contract* since it is not apparent from the abstractions of interfaces. The revision class developer should abide by this hidden contract in order to prevent the occurrence of the fragile base class problem.

The refinement class given in the code below gives a revision class *Bag'* that is independently (in absence of extensions) interchangeable with the old base class. The revision class developer improves the performance by eliminating an extra call overhead in *addAll*.

We can see that the revision class has violated the *hidden contract* by dropping the call to *add* in *addAll*, which is the very basis of the extension class. As a result, the count of elements is not incremented appropriately in the derived class. This gives an incorrect value of cardinality in presence of the extension. 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*, in other terms, a contract beyond the interface abstraction, a contract which the revision class developer should not violate.

4 Violation of Hidden Contracts by Hook Inclusion

Calling a hook method in the revision class method that has not been earlier called in the base class may also cause problems if the user is aware of this fact. The code below shows class *Base*, which exports methods m and n, both incrementing a variable x.

```
class Bag' {
  private:
      Array b[];
   public:
     Bag'() { // initialize the bag data structure
       ...
     }
    virtual add(char element) { // find an empty location and add the element
       ...
    }
    virtual addAll(char bs[],size) {
       while(size){ // find an empty location and add the element
         ...
       }
    }
    virtual int cardinality(void) { // find and return the number of elements
       ...
    }
};


class Base {
 private:  int x;
 public :  base () { x  = 0;}
           virtual void m(void) { x = x + 1; }
           virtual void n(void) { 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 and delegates the call to m.

```
class derived: public base {
 public : void n(void) { self.m(); }
};
```

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.

In an example given below, the revision class developer maintains the interface contract while refining the base class by delegating m to n. We can see that the revision class is independently interchangeable with the original base class, but not so in presence of the derived class. However, the *hidden contract* is violated resulting in mutual recursion.

```
class revised {
 private:  int x;
 public :  revised () { x  = 0;}
           virtual void m(void) { self.n(); }
           virtual void n(void) { x = x + 1; }
};
```

Since the revision class includes a call to hook method n, in its refinement of method m, extended implementation of n is included in m in presence of the derived class. This turns out to be unexpected to the user.

5 Conclusion

Both the derived class and the revision class developers should abide by the contracts while producing extensions or revisions of the base classes to avoid incompatibilities between the extensions and revisions. The notion of *hidden contracts* was introduced. Beyond the interface contracts, the *hidden contracts* must be adhered to by the revision classes. We identified the *hidden contracts* in two examples of the fragile base class problem in the literature. The users make extensions on the basis of the *hidden contracts*. For example, a call to hook method, *add* in *addAll* was the very basis of the extension class by the user, without the knowledge which, the user could not have made the extension discussed. We discussed the aspect of hidden contract violation in two fragile base class examples from the literature. Thus, *Hidden contracts* are shown to be important for the user to make meaningful extensions and should be followed by both the user and the developer to avoid the fragile base class problem.

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.

[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.

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