Reinforcing Fragile Base Classes

Kees Huizing and Ruurd Kuiper**

Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven,
The Netherlands,
keesh@win.tue.nl, wsinruur@win.tue.nl

Abstract. The Fragile Base Class problem is approached from the angle of a proof system for class invariants. It is shown that the source of the FBC itself can be understood as a problem of dynamic binding of predicates in the correctness proof of invariants. A solution is presented based on an extension of the concept of behavioural subtyping and the novel notion of cooperative contracts. Thus, flexible boundaries of reuse can be specified for each class.

1 Introduction and Position

When a developer believes in the assumption that changing the implementation of a method within the boundaries of the specification (contract) should leave the behaviour of other methods unchanged, he may be caught off guard by the Fragile Base Class problem (FBC).

Specifically: if a reuser subclasses a class C to a class D by inheritance, and later the provider of C revises C, keeping its contract, the reuser may find that D no longer satisfies its contract.

For this, inheritance is often blamed and therefore rejected as a vehicle for code reuse. This we regard as an unacceptable limitation on code reuse. Similarly, we regard disallowing the provider to change the base class as too restrictive.

In the literature, the solution to the FBC is mainly sought in syntactically restricting the allowed class dependencies during development. We propose a different approach that leads to a more fine-grained, or if you like, semantic, check of the dependency between classes and methods.

When we study the proof obligations in a formal proof system, we can pin down the problem to dynamic binding. Either the provider of the base class has to prove the invariant of a (future!) subclass, or the subclass developer needs access to the implementation of the base class to prove the invariant.

We solve this problem in two steps. First, we remove the dependency of the base class on the derived class by extending the notion of behavioural

** The authors are partially supported by ITEA DESS.

subtyping. This means that the developer of the subclass has to prove additional clauses for methods that are inherited.

The second step of the solution is a notion of cooperative contract. The developer of the base class can use a kind of parametrized postconditions that allow the reuser to derive more properties about a method than could be derived from an ordinary contract, without giving the reuser access to the implementation. Without cooperative contracts some code would not be amenable to reuse.

By choosing a level of cooperation in the contract, the provider can define the boundaries of reuse.

Workshop Position: Both providers and reusers of classes should be allowed to make adaptations to classes; the FBC should be understood and remedied in the context of a proof system:

1. user developments from the base classes should be in accordance with a new notion of behavioural subtyping;
2. provider's changes to a base class should respect a new notion of cooperative contract.

2 The Fragile Base Class

The following example is modified from an example in [8]. Consider a class `Set` that represents a set of integers with methods `add(int)` and `addSquared(int)`. The first method simply adds an integer to the set; the second one computes the square of an integer and then calls `add()` to add the result to the set. In Java this class could be defined as follows.

```
class Set {
    ⟨ I: true⟩
    ...

    public add(int n) {
        ...
        ⟨post: s = s~ ∪ {n} ⟩
    }

    public addSquared(int n) {
      add(n*n);
       ⟨post: s = s~ ∪ {n^2} ⟩
    }
}
```

We write assertions surrounded with angle brackets to avoid confusion with the Java curly brackets.

Here, s represents the contents of the set; a variable in a postcondition with a $\sim$-symbol attached represents the old value, i.e., the value at the start of the method. To keep the example simple, we used a trivial integrity check.

Now a subclass `CountingSet` of `Set` is made that adds an instance variable keeping track of the number of elements in the set. For this purpose, method `add()` is overridden and we get the following class.

```
class CountingSet extends Set {
    ⟨ I: |s| = size ⟩
    int size;
    ...

    public add(int n) {
        size++;
        ...
        ⟨ post: s = s~ ∪ {n} ∧ size = size~ + |{n} \ s~|⟩
    }
```

Here, I is the invariant of the class, which specifies the consistent states of all objects of this class. A user of an object of this type may expect the invariant to hold. As a consequence, all public methods should establish the invariant at method return.

Note that the postcondition of `add()` has been strengthened with information about `size`.

`CountingSet` has no need to override `addSquared()`, since this latter method calls `add()` and thanks to dynamic binding, the correct version of `add()` will be executed, depending on the type of the object underhand.

This is an example of the design pattern Template Method, where execution of a primitive operation (`add()` in this example) is delegated to a subclass [1].

Now suppose the class `Set` is revised. For some reason, maybe of efficiency, the method `addSquared` does not call `add()` anymore, but puts the element directly into the set. This seems harmless, since the contract of `addSquared` is maintained. For `Set`-objects everything works. For `CountingSet`-objects however, things go wrong. Since `add()` isn't called anymore, the variable `size` will not be updated when an element is added by means of `addSquared()`. As a consequence, the invariant **I** is not maintained.

This problem is known in the literature as the Fragile Base Class Problem ([5]). This version is called the semantic problem. There is also a *syntactic version*, which concerns the problems when the subclass is not recompiled after revision of the base class. Since Java solves the syntactic problem, the semantic one is even more important. In this paper, we will mean the semantic problem when we speak about the Fragile

Base Class problem. There are several related problems, such as infinite recursion as a consequence of incorrect overriding. As we only consider partial correctness here, we will not treat this problem. For an extensive list of issues related to the Fragile Base Class problem, see [8].

3 Pinning down the problem

When we use a formal verification system, it becomes clear where the intuition makes an unjustified assumption that causes the FBC problem. For this purpose, we use a proof system along the lines of the one presented in [2]; other systems such as [7] could do as well. The problem is centered around the concept of dynamic binding.

3.1 Dynamic binding

In Java, the method call `b.m()` means: choose the first definition of `m` that you find when going up in the type hierarchy, starting in the class to which object `b` belongs, and then execute this method in the context of object `b`. With "object `b`" we mean the object that results from evaluating the expression `b`. Note that this value depends on the state, and so does the class of which method `m()` is chosen. This state dependency of method choice is called *dynamic binding*.

We extend this notation to predicates. The expression $b.I$ means: interpret predicate I in the context of object b. As with method invocation, the choice of I depends on the class of b.

To simplify the discussion, without losing anything of the essence, we formulate a proof rule for a parameterless method in the absence of recursion.

$$\frac{\begin{array}{l}\text{for all classes } C \text{ containing } m\text{:}\\ \langle \texttt{this}.\mathsf{pre} \wedge \texttt{this}.I \wedge \ldots\rangle\ \mathsf{body}_{m_C}\ \langle \texttt{this}.\mathit{post} \wedge \texttt{this}.I \wedge \ldots\rangle\end{array}}{\langle b.\mathsf{pre}\rangle\ b.m()\ \langle b.\mathsf{post} \wedge b.I\rangle}$$

The premise requires proofs for the bodies of all the methods that could be executed as a result of the call $b.m()$. We keep on the safe side by taking every method called m. Since the correctness of all these bodies should be proven anyhow, albeit not for this specific call, we are not interested in removing this redundancy from the rule.

On the dots, more invariants may appear, depending on the proof system and the circumstances. This does not interfere with the exposition, however; details can be found in [2].

The object `this` in the premise refers to the object on which the method body is executed. We write `this.pre` etc. to stress that the predicate `pre`

has to be evaluated in the context of this object and likewise for post and I.

Now dynamic binding puts us a problem. What are the predicates pre, post, and I? Since we don't know the dynamic type of b, we don't know which version of $m()$ will be executed and hence, which precondition, postcondition, and invariant will apply.

3.2 The solution

In practice, the pre- and postconditions will not arbitrarily change in a subtype (inheritance) hierarchy. A method in a subclass should at least fulfill the contract of the method it overrides. This idea is called *behavioural subtyping* after [4] and it amounts to the requirement that, when $D <: C$ (D is a subtype of C), the following implications should hold:

- $\mathsf{pre}_C \Rightarrow \mathsf{pre}_D$
- $\mathsf{post}_D \Rightarrow \mathsf{post}_C$
- $I_D \Rightarrow I_C$

This is a principle of good OO design and it enables us to replace in many cases dynamic reference to predicates with static ones. When b is an object and P is a predicate (pre/postcondition or invariant), we use the notation $b{:}P$ for the *staticly bound* P, i.e., if C is the static type of expression b, then $b{:}P = (C)b.P$, where (C) is the type cast operator. We assume that the static type of an expression is always known. Surely it can be derived from the program context. Under assumption of behavioural subtyping, we can formulate a friendlier proof rule for method call:

$$\frac{\begin{array}{l}\text{for all classes } C \text{ containing } m\text{:}\\ \langle \texttt{this}{:}\mathsf{pre} \wedge \texttt{this}{:}I \wedge \ldots \rangle\ \mathsf{body}_{m_C}\ \langle \texttt{this}{:}\mathsf{post} \wedge \texttt{this}.I \wedge \ldots \rangle\end{array}}{\langle b{:}\mathsf{pre} \wedge \ldots \rangle\ b.m()\ \langle b{:}\mathsf{post} \wedge b{:}I \wedge \ldots \rangle}$$

In this rule, all occurrences of dynamic binding of predicates are removed, except one: this.I in the premise. In a minute we will see how to get rid of this dynamic binding also.

The dynamic references in the *conclusion* can be replaced by static ones because of behavioural subtyping. This is in fact an application of the rule of consequence: $\langle P \rangle\ S\ \langle Q \rangle$ implies $\langle P' \rangle\ S\ \langle Q' \rangle$ if $P' \Rightarrow P$ and $Q \Rightarrow Q'$.

Substituting this:pre and this:post in the *premise* is allowed because of the rules for method choice in Java and most other OO languages. If the dynamic type D of this differs from the static type C, there can be no method m defined in D, since otherwise method m from D was chosen instead of C.

For the invariant in the premise, however, this reasoning does not apply, as each type may have its own invariant, independent of whether m is overridden or not. The stronger obligation to prove `this`.I remains here. It is here that the intuition unjustifiedly assumes that proving the invariant of class C is enough.

In terms of the fragile base class example: The method `addSquared` has been defined in class `Set` and hence the proof of its correctness is performed in the context of `Set` (the static type of `this` is `Set`). Since `addSquared` is not redefined in `CountingSet`, exection of this method may well be in the context of an object of the class `CountingSet` and then the *dynamic* type of `this` is `CountingSet`.

In the most pregnant examples of the Fragile Base Class, the development of base class C is both in time and place remote from the development of derived class D. E.g., company X produces a class library of which C is a part. Then company Y uses this library and reuses C by inheritance. X cannot be held responsible for the invariants of subclasses yet to be made. On the other hand, passing the proof obligation to the developer of D would require X to give its users insight in its source code. This is unpractical and from a commercial point of view unwanted. Furthermore, it denies the concept of abstraction by contract.

So the only reasonable proof obligation in the premise is `this`:I.

This obligation is too weak, however, and would render the proof rule unsound. To solve this, we propose a strengthening of the notion of behavioural subtyping: When D has a stronger invariant than C, it should be proven from the contract (specification) provided by C.

Definition 1 (Reinforced Behavioural Subtyping). *Type D is a reinforced behavioural subtype of C if:*

1. $I_D \Rightarrow I_C$
2. *for every method m_D overriding m_C:*
   $\mathsf{pre}_{m_C} \Rightarrow \mathsf{pre}_{m_D}$
   $\mathsf{post}_{m_D} \Rightarrow \mathsf{post}_{m_C}$
3. *for every non-private method m not overridden in D and any object d of type D:*
   $d{:}(I \restriction \mathsf{inv}_m \wedge I^{\sim} \wedge \mathsf{post}_m \wedge I_C) \Rightarrow d{:}I$.

Here, the set inv_m consists of the variables that are not changed during execution of m. It is derived from the specification of m. (This set is specified explicitly or implicitly by specifying which variables are allowed to change. Here, we do not elaborate on how to specify inv precisely.)

The predicate $I^{\sim}$ is defined as the predicate I with all variables adorned with the symbol $\sim$. It simply says that I was true at the start of the execution of m.

Obligation 3 is new. It requires the developer of D to prove the possibly stronger invariant of the derived class after execution of m. For this he

may use the invariant of the base class I_C, the postcondition of m, the stronger invariant restricted to variables that do not change under m, and the knowledge that this stronger invariant held at the start of m.

Since this proof obligation is about a method that is not overridden in D, it is possible that the developer of D has no access to the contract of m, e.g., if m is a private method of C of a supertype thereof. This would make obligation 3 impossible to fulfill. We take the approach, however, that private methods need not establish the invariant, as in [3].

This proof obligation leaves not much room for the developer of the derived class to strengthen the invariant. In the Fragile Base Class example, e.g., the invariant of `CountingSet` cannot be proven. For situations like this we propose the concept of *cooperative contracts*. The postcondition of a method may contain references to postconditions of other methods, in particular methods to be overridden in subclasses. In proof obligation 3 of reinforced behavioural subtyping, the interpretation of these references depends on the type of d. This way, such a reference may result in a stronger predicate if the corresponding method is overridden in the subclass. To see this, suppose post_m refers to a post_n, the postcondition of another method n. When this method is overridden in D with a stronger postcondition, the conjunct post_m will be effectively stronger too, since it is evaluated in the context of an object of class D. If done right, this makes it possible to prove the stronger invariant of class D.

So under the assumption of Reinforced Behavioural Subtyping we have the following **proof rule for non-recursive method call**:

$$\frac{\begin{array}{l}\text{for all classes } C \text{ containing } m\text{:}\\ \langle \texttt{this}{:}\mathsf{pre} \wedge \texttt{this}{:}I \wedge \ldots \rangle\ \mathsf{body}_{m_C}\ \langle \texttt{this}{:}\mathsf{post} \wedge \texttt{this}{:}I \wedge \ldots \rangle\end{array}}{\langle b{:}\mathsf{pre} \wedge \ldots \rangle\ b.m()\ \langle b{:}\mathsf{post} \wedge b{:}I \wedge \ldots \rangle}$$

Now we can go back to the example of the Fragile Base Class and change the postcondition of `addSquared()` to:

$$\mathsf{post}_{addSquared} : \mathsf{post}_{add}(n^2)$$

Proof obligation 3 now becomes (we assume there are no relevant invariance properties in the specification of `addSquared`):

$$d{:}(|s^{\sim}| = size^{\sim} \wedge \mathsf{post}_{add}(n^2)) \Rightarrow d{:}(|s| = size)$$

which is equivalent to

$$d{:}(|s^{\sim}| = size^{\sim} \wedge s = s^{\sim} \cup \{n^2\} \wedge size = size^{\sim} + |\{n^2\} \setminus s^{\sim}|) \Rightarrow d{:}(|s| = size)$$

which follows easily from the facts

$$|x \cup y| = |x| + |y| - |x \cap y|$$

and

$$|x \setminus y| = |x| - |x \cap y|$$

Of course, we could have strengthened the postcondition of `add` in the class `CountingSet` with the invariant. This would have trivialized the proof above, however. We believe that the current proof is more interesting because it shows the role for the predicate $I^{\sim}$.

This shows that the correctness proof of the subclass is based on the specification only of the base class, in contrast to the original situation where formal correctness needs the code of the base class. As a consequence, this allows for revising the base class, within the boundaries of the contract. E.g., in the method `addSquared` above, the implementation of the integrity check could be altered without fear of the Fragile Base Class problem. The provider of the base class can decide how cooperative the contract will be.

4 Conclusion

We have shown how to use formal methods to better understand the Fragile Base Class problem and how to solve the problem. Other approaches to this problem can be found in the literature. [3] has an extensive framework to specify properties of (Java) programs. It elaboreates on properties of the call graph, wehereas we use a more assertional approach, concentrating on the validity of invariants.

In [6], the notion of cooperation contracts is introduced as part of a solution of the Fragile Base Class problem. Their approach is purely syntactical and their notion is not the same as our cooperative contracts, which play a central role in proving the validity of assertions.

Mikhajlov and Sekerinski in [8] formulate several requirements that guarantee true refinement of superclasses. These requirements are more restrictive and do not allow the example of a subclass invariant that combines instance variables of the subclass with those of the superclass as in the `CountingSet` example.

We believe that we have clarified the Fragile Base Class problem, and, with it, similar problems connected with inheritance and dynamic binding in object-oriented languages. The novel approach of cooperative contracts allows for a fine grained semantic solution to the problem of safe code reuse in the subtle frameworks that can be found in many object oriented program designs.

References

1. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.
2. K. Huizing, R. Kuiper, and SOOP. Verification of Object Oriented Programs Using Class Invariants. In *Fundamental Approaches to Software Engineering (FASE 2000)* (Maibaum, Ed.), Berlin, 2000, Lecture

Notes in Computer Science, Vol. 1783, Springer-Verlag, Berlin, 2000, pp. 208–221.
3. Clyde Ruby and Gary T. Leavens. Safely Creating Correct Subclasses without Seeing Superclass Code. In *OOPSLA 2000 – Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Minneapolis, Minnesota. SIGPLAN Vol. 35(10), pp. 208–228, 2000.
4. B. Liskov and J. Wing, *A behavioral notion of subtyping*, ACM TOPLAS, 16:6, pp. 1811–1841, 1994.
5. C. Szyperski, *Component software: Beyond object-oriented programming*, Addison-Wesley, 1998.
6. M. Mezini. Maintaining the consistency and behavior of class libraries during their evolution. In *Conference Proceedings of OOPSLA '97*, ACM SIGPLAN Notices, Vol. 32(10), pp. 1–21, Oct. 1997.
7. A. Poetzsch-Heffter and P. Müller, *Logical foundations for typed object-oriented languages*, in D. Gries and W.P. de Roever, Eds., Programming Concepts and Methods (PROCOMET), 1998.
8. L. Mikhajlov and E. Sekerinski. A Study of the Fragile Base Class Problem. In *ECOOP'98 – Object-Oriented Programming 12th European Conference* (E. Jul, Ed.), Brussels, July 1998, pp. 355–382, Lecture Notes in Computer Science Vol. 1445, Springer-Verlag, 1998.