A Method for Modifying Dynamically Classes in the Object-Oriented Dynamic Programming Environment

Lu Pei **Yu Dachuan** **Lu Jian**
Institute of Computer Software, Nanjing University
State Key Laboratory for Novel Software Technology
Nanjing 210093, P.R.China
lj@nju.edu.cn

Abstract In the object-oriented dynamic programming environment, dynamic modification of a class, which permits change of it at run-time and without recompilation, is the key point to exploit flexibility and support rapid prototyping. However, it causes a problem that the existing objects of the modified class are difficult to handle. In this paper, the concept of "cloned class" is introduced, and a method based on it for modifying dynamically classes is proposed.

Keywords Rapid prototyping, object-oriented dynamic programming environment, dynamic modification, cloned class, source class.

1. Introduction

An object-oriented dynamic programming environment is a software developing environment which supports dynamic programming based on an object-oriented language. In this kind of environments, programmers can interactively input and execute some testing cases for certain classes, such as creating instances of them, and sending messages to objects of them. Meanwhile, if certain problems are found, programmers can modify any class and existing object as they will. After the modification, programmers can continue to execute programs interactively. Smalltalk-80 programming environment [Goldberg 83] is a classic object-oriented dynamic programming environment.

A significant advantage of the object-oriented dynamic programming environment is its flexibility, which is very useful for rapid prototyping. Obviously, dynamic modification of classes is the key point of this kind of programming environments. By dynamic modification, we mean the modification of classes at run-time, when programmers are testing these classes interactively . Since dynamic modification in a sense permits rapid change of classes, programmers can use it in prototyping classes [Krief 96] as they will rapidly and flexibly.

However, a big problem occurs when a class is dynamically modified. If the class to be modified has already had some objects, then after it is changed, how to deal with those objects? Two major parts of a class can be modified: member variables and member functions(methods). Modifications of member functions change the behavior of objects pertaining to that class while do not affect the storage structures of those objects in memory directly. But modifications of member variables will cause the storage structures of those objects in memory to be inconsistent with the modified class immediately. Therefore, The programs which are executed after the modification and involve the existing objects of the modified class, will be logically unreasonable in a sense.

There lie two major solutions to this problem. One is to discard all objects whose class's description of member variables is modified. Obviously it is a conservative approach. Although in this way, each existing object is confirmed to be consistent with the class of it, there is some serious loss of dynamicity. In fact, this kind of modification is not "dynamic" at all, because it is similar to clearing up all objects and restarting a new program after modification. The other method is to convert all those objects to make their storage structures in accordance with the description of the modified class. It is an optimistic approach. VisualAge [OTU 95], a dynamic programming environment based on Smalltalk language, uses this method. Although for Smalltalk, a highly dynamic programming language without type, it is not difficult to implement object conversion, the method still slows down the speed of dynamic modification greatly. The situation will be worse if the programming language is a typed object-oriented one. Since we can not only add or remove member variables, but also change their types when modifying classes, the conversion of objects is much more complex and time-consuming than that of Smalltalk.

According to above analysis, a new method of using *cloned class* to dynamically modify classes is proposed in this paper. It is based on MagicFrame, an object-oriented software developing environment still under construction of Motorola Inc. and us. The main idea is that, once a class needs to be modified dynamically, a cloned class of it is created automatically. Then, it is the cloned class rather than the original one that is actually modified, such that the original class is only modified logically. By this way, the problem mentioned above could be solved more reasonably and clearly.

2. The cloned-class method

2.1 Source class & cloned class

To introduce our method, first we present two new concepts: *cloned class* and *source class*.

A *cloned class* is a duplicate of a class for direct modification. The class which is duplicated is called a *source class*. The duplication process from source class to cloned class is called *cloning*.

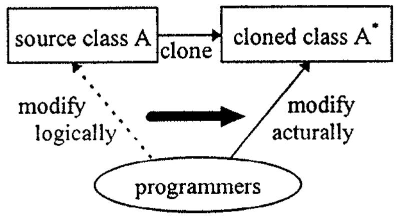

Figure 1 Modifying cloned classes

In an object-oriented dynamic programming environment, once a class A needs to be modified dynamically, a cloned class A^* of it is automatically created. Then, A^* is ready for direct modification, while A does not change at all. This process is shown as Figure 1. When modification is finished, programs can run with all existing objects of class A that already exist still pertaining to A, since there is no change on A. But, if statements of "creating A's objects" are encountered, A^*'s new objects are really created rather that A's (note that class A^* is the logically modified version of A). To sum up, it is the system that automatically and implicitly operates two kinds of objects differently: objects of the source class, which already existed before dynamic modification, and objects of the cloned class, which are created after dynamic modification.

2.2 The cloned-class tree

Since a class can be modified in many aspects, there could be more than one cloned class derived from a single original class. What's more, since further modification may be done to the modified classes , a cloned class could have its own cloned classes. So, modifying a class A may generate a tree, which is shown as Figure 2. The root of that tree is class A, and each of other nodes is a cloned class of its parent node. Each branch represents a single process of modification. We call this tree *the cloned-class tree* (or *dynamic modification tree*) of class A.

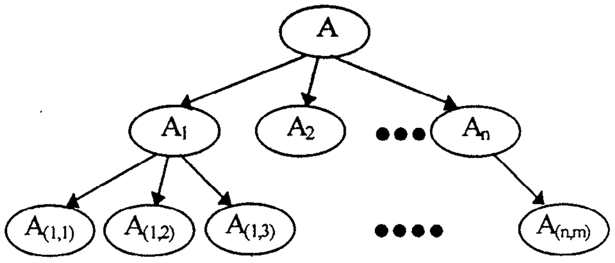

Figure 2 The cloned class tree of A

The modifying process of class A is in fact the expanding process of the cloned-class tree of class A. The lately created node A_c in the tree is called the *current cloned class*, which is the newly modified class. If the program running after modifation needs to create A's objects, the system will automatically create A_c's objects instead, and operate those objects as A_c's objects rather than A's. Creating an object of the current cloned class is called *validation* of it.

Before the current cloned class B is validated, all modification on B is directly done to B, and there is no need to create B's cloned class. The new modification and the old modification on B are incorporated into one as a single branch of the cloned-class tree. The reason to introduce this rule is obvious. Since class B is changed before validation, it is impossible for any objects of the old class B to be created, and there is no need to maintain it as a node of the cloned-class tree.

When programmers think some permanent change on class A is really needed, they can take an operation called *updating confirmation*, which updates the source class A with one of its cloned classes and releases the whole cloned-class tree of A. When updating confirmation finishes, the corresponding dynamic modification process finishes as well. Generally, updating confirmation will be done at the time that programmer stop inputting programs interactively to test classes.

It is necessary to point out that, If class A has subclasses, when A is dynamically modified, subclasses of A do not clone as A, and all objects of them do not change at all (that is, the semantics of modification of a class can not be passed to its subclasses immediately). Since in an object-oriented dynamic programming environment, most dynamic classes are classes still under construction and have no subclasses(i.e. they are leaf nodes in the class hierarchy tree), this rule affect little on the clone-class method. When programmers really want to change a class and all its subclasses simultaneously, they can modify it statically rather than dynamically.

2.3 Advantages

Compared with the two traditional methods mentioned above, the cloned-class method has the following advantages:

(1) Modifications of cloned classes do not affect source classes. So all objects of source classes are consistently in accordance with their classes, and there is no need to delete or convert the existing objects of source classes. The whole developing process of classes is logically clear and reasonable.

(2) In dynamic programming environments, modifying classes is often a tentative action. One modification of a class may be undone by the next one. If we use methods that directly modify classes, there will be a lot of undo processes in certain situations. But if we use the cloned-class method, there is no need to undo the modification that do not satisfy us. The only thing we should do is to restore the original class at the time of updating confirmation.

(3) Another important advantage of cloned-class method is that, since objects of the original class and objects of the cloned class can coexist, we can compare their behavior and observe the result of dynamic modification interactively and intuitively. For example, there is a window class W and one of its instance w_1. When we are not satisfied with one of W's display styles, we can modify class W dynamically even when w_1 is active in memory. Then, according to our cloned-class method, a cloned class W^* is created automatically. Programmers can create a new object w_2 of class W^*, compare these two windows w_1 and w_2 in a What-You-See-Is-What-You-Get manner, and decide whether the modification is really wanted. Obviously, this advantage can not be obtained by those two traditional methods mentioned above. It greatly exploits the flexibility of dynamic modification and facilitates the prototyping process of classes.

In fact, the cloned-class method can be considered an application of buffering technique on dynamic modification. And in a sense, it extends the traditional idea of buffering, because a class can have a "buffer tree" rather than a "buffer class". To sum up, the method not only improves the type safety of dynamic modification, but also makes it more flexible for prototyping.

2.4 Implementation

Advantages of the cloned-class method are obtained from its relatively complex implementation.

Since a cloned class is not a common class but a new class with little difference from the source class, its storage structure is definitely different from common classes. There are two kinds of members in its storage structure: (1) *Duplicated members*, which are members duplicated directly from the source class. (2) *Changed members*, which are originally in the source class and then modified (obviously, the new members added are also in this catalogue). All members in a cloned class are implemented by pointers. If a member is a duplicated one, it is exactly a pointer to that member of the source class in memory. If a member is a changed one, it is a pointer to the newly allocated memory area of that member.

When we begin to dynamically modify a class A in our program, the system creates a cloned-class tree T for A automatically. Each node in the tree is a cloned-class structure. At first, there is only one root node, which is exactly the source class A. When we modify class A for the first time, the system creates a cloned class A_1 of source class A , and remark A_1 as the *current cloned class*. Then, A_1 is modified directly by us. When we finish modifying A_1, The system attaches class A_1 to the cloned-class tree T. When the program resumes, all existing objects of A are still A's objects. The system interprets their operations according to A's behavior. But, if there is a statement of creating a new object a_l of class A in the following program, the system will create a instance a_l of the current cloned class A_l, which is the lately modified version of A. Then, a_l is remarked as A_l's objects, and all operations on a_l is interpreted according to class A_l.

When further modifications are needed, the system prompts the cloned-class tree T to programmers in a visual way. Programmers can select any class node of T to pursue further modification. If the selected class A' is not validated, then class A' is directly changed, and no new cloned classes are created from A'. If A' is validated, then a new class A'' is cloned from class A' and remarked as the new *current cloned class*. In this way, the cloned-class tree T expands as the program runs, and there are more and more objects of cloned classes in the cloned-class tree.

When the program finishes, the system prompts the cloned-class tree T of class A to the programmer in a visual way, and requires the programmer to select a class in that tree to substitute the original class A. This is the process of updating confirmation.

In practical situations, the depth of the cloned-class tree of a class is often less than 3, and the number of all nodes in the tree is less than 5. Usually, only a small part of a class is dynamically modified, and all duplicated members are shared by the source class and cloned classes. Therefore, it costs little time and memory to maintain a cloned-class tree. During the execution of a program, our system needs to do only two extra things: remarking objects as different classes' instances and remarking a class "validated" when its new instance is created. Thus, the cloned-class method have little side effect on performance of the whole system.

2.5 Dynamic modification of object structures

We can modify dynamically not only classes but also existing objects in our dynamic programming environment. There are two kinds of modifications to an object: (1) Modifications of its value, which changes values of data members in that object. (2) Modifications of its structure, which changes types of data members, including adding or deleting some data members in that object. Obviously, modifying the structure of an object is actually modifying the class of that object indirectly.

Also, we can use the cloned-class method to support dynamic modification on object structures. Once the structure of an object needs to be modified, a cloned class of that object's class is created implicitly for direct modification. The modified object is operated as an instance of the cloned class, and all other objects which belong to the original class are not affected at all. This process is shown as Figure 3.

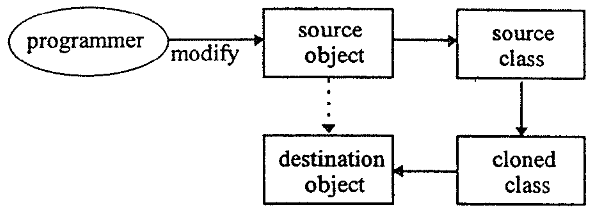

Figure 3 Dynamic modification of object structure

3. Conclusions

The method for dynamically modifying classes by cloned classes discussed above is proposed by us when we studied object-oriented dynamic programming environments. Compared with the two traditional methods, It is a fairly satisfactory solution to the problem that how to handle existing objects whose classes are dynamically modified.

Acknowledgment This research is supported by National Science Foundation for Excellent Young Scientists, Foundation of Trans-Century Talented Persons Fostering Program and PanDeng Program of China. The first author is the recipient of a Motorola Partnership in Research Grant. We also thank all the members in MagicFrame group, which is directed by Prof. Xu Yongsen.

References

[Goldberg 83] Adele Goldberg, SmallTalk-80 : The Interactive Programming Environment. Addison-esley , 1983.

[Krief 96] Philippe Krief, Prototyping with objects, Prentice-Hall, 1996.

[OTU 95] School One: Smalltalk Developer/Tester Training Student Notebook . IBM Corp, 1995.