Inheritance vs. delegation: Is one better than the other?

Peter Bosch
University of Twente
Enschede
The Netherlands
Peter.Bosch@cs.utwente.nl

April 6, 1993

Abstract

In this paper I will discuss the differences between inheritance based systems and delegation based systems. Specifically, I would like to discuss the way objects are organized in both systems, the way inheritance is implemented, how the *self* variable is treated and I would like to discuss the advantages and disadvantages of both language types.

1 Introduction

Inheritance based systems split the objects up into *classes*. Classes describe the shared behaviour of a set of objects. A new object is instantiated from the class and has no correspondance to other objects of the same class except for the class object. If some class shares behaviour with another class but has some peculiarities, a *subclass* is created which inherits the shared behaviour from the *superclass*. An object in a prototype based system represents the *default* behaviour of an object. If a new object has a different behaviour, a new object is created which describes its differences from the *prototype* object [3].

In this paper I would like to take a closer look at the differences between class and prototype based systems. Specifically, after reading [5] and [3] it seems that prototype based system are a superset of class based systems.

This paper is organized as follows. Chapter 2 will describe class based systems, chapter 3 will describe prototype based system, and chapter 4 will describe the differences between both systems. Conclusions can be found in chapter 5.

2 Class Based Systems

In a class based system, two object types exist: class objects which describe the common behaviour of objects and instance objects which hold the instance variables for an object [3]

[1]. The class objects contain methods to *instantiate* an instance object. To construct a class object, the common behaviour of a set of objects is identified and stored in the class object. The idiosyncracies of the objects, however, are stored in the instances of an object. If, for example, you are building a class based, object oriented, windowing system, you would put the common behaviour of windows in the class *windows* (such as the fact that every window has a position on the screen), while the idiosyncracies of a specific window are put in an instance object (such as the current location of the window on a screen).

If an object wants to share behaviour with a class, but the object itself has some peculiarities, the added behaviour can be captured in a subclass. The common functions can be accessed through *inheritance* [4]. The subclass can specify which common behavior it wants to use of the superclass and which behaviour it doesn't want to use.

Several techniques exist to invoke an operation on an object. For example, in *C++* [1] an operation on an object is performed by executing one of the methods of the object. The methods can be found through a pointer in the instance object to a method table containing the addresses of the method functions. Inheritance is implemented such that the instance variables are merged into the instance object, and the methods are merged into the method table. All the references are resolved at compile time. When executing a function, the pseudo variable *this* is bound to the current object. The variable *this* can be used to refer to the object's instance variables and methods.

In *Smalltalk* [2] an operation on an instance object is performed by sending the object a message describing which operation needs to be executed. A method is made up of a message pattern and a set of expressions. The correct method is selected by comparing the message and the method description. Inheritance is implemented through an *inheritance chain*. This chain describes from which class the current class is created. By following the inheritance chain, the system is able to find the instance variables and method definitions of superclasses. In *Smalltalk* two pseudo variables exist which point either to the object itself (*self*) or to the super class object in which the method or instance variable is defined (*super*). *Self* is the object which receives the method request message. When referencing *self*, method determination starts at bottom of the inheritance chain, when referencing to *super* method determination starts in the object referred at by *super*.

The major difference between *Smalltalk* and *C++* is that *Smalltalk* is an interpreted language and *C++* is a compiled language. In *C++* the method table and used instance variables are determined statically (i.e. when a the program is compiled). In *Smalltalk*, you send a message to an object which will determine if it has implemented the operation. If not, the message is forwarded through the inheritance chain.

3 Prototype Based Languages

In a prototype based language only one object type exists. This object type contains the instance variables of the objects and the behaviour of the object. If another object wants to share some behaviour of the former object, a new object is created in which only the idiosyncracies of the new object are stored. If the new object receives a message informing it

[1] Although I state this explicitly, some object oriented system don not even have seperate class objects. *C++*, for example, only supports instance objects

to do something, and the object hasn't implemented the operation, the operation is forwarded or *delegated* to the prototype object [3].

If, for example, you are implementing a prototype based, object oriented, windowing system, you would implement a prototype object which implements the common behaviour of windows (such as dragging the window across the screen), while the derived object may contain the physical location of the window. In a prototype based system it is possible to use both the shared behaviour of objects while defining "extra" behaviour for an object. If, for example, some object has special behaviour when a window is dragged across the screen (such as creating a control window), the window can still use the common behaviour by first executing the extra behaviour before delegating a *drag-window* message to the prototype object.

Self [5] implements a prototype based, object oriented system. In this system every object contains a list of slots which describe the instance variables or object methods. To retrieve the value of an instance variable or to execute a method, a message is sent to the object. If the object does not have a slot for the request, the message is delegated to the prototype object (all the way to the top object, if no object has a slot for the request). In Self the pseudo variable *self* refers to the object which receives the first (original) message. When the request is delegated to a prototype, *self* still refers to the original object. This way, methods defined in a prototype are still able to refer to the original object.

4 Prototypes vs. Classes

At a first glance, prototype based systems work much like class based systems. However, if you take a closer look at the differences between both systems, you find that there are some interesting issues.

In class based systems, you always end up creating at least two objects to represent some object. You first program the class object before you instantiate the "real" object. In prototype based systems, you simply program the object. Prototype based systems, therefore, "feel" better when you are writing small applications or try to prototype some problem. If you have programmed your object, and you want to model a new object which shares state and behaviour with the prototype object, you can simply make the new object and delegate requests for the shared behaviour to the prototype. Any object in the system can be made a prototype object. This provides a very flexible way of programming new objects. However, this can also lead to "spaghetti" object systems where objects are dependening on prototypes, while the prototypes are, again, depending on other prototypes.

Class based systems have a clear distinction between shared behaviour and instances. The shared behaviour is stored in the class objects while the instance variables are stored in the instance objects. If an object shares behaviour and data with some other class but has some peculiarities, you have to create a "subclass" to capture the special behaviour and use the shared part from the "superclass".

Instance objects "feel" more active than class objects, in the sense that by sending a message to an instance object, you can change the object. You don't send messages to a class object to change a class object, you take the class object into your editor and update the class object. It takes more effort to change a class object. In a prototype based system, many objects can depend on a prototype object. This means that if the prototype object changes, all the other objects also change. Although this might seem handy, it also means that you have to be very careful when you update a prototype object. You have to know the object dependency graph of the objects derived from the prototype object, in order to do a safe update. I.e. you don't want the objects which are depending on the prototype object to stop working if you update the prototype. In a class based system, instance objects are only depending on class objects. Class objects might be depending on other class objects. If you change the class object, you know that you are going to change the behaviour of the instance objects and/or subclasses, i.e. changing a class is more explicit. It seems that in a class based system, you are more aware of the fact that some objects are depending on the behaviour of other objects whereas in prototype based systems, it can be totally unclear which objects are depending on which other objects. In general, the class inheritance tree is more visible than the prototype dependency chains.

Class based systems can be implemented in a prototype based system. This is done by explicitly defining "class" objects which contain the shared behaviour for a class of objects. If some object wants to share some behaviour with a class object, but the new object has some peculiarities, a "subclass" object is created which holds the peculiarities. Next, the new object is instantiated from the subclass, by sending the subclass a *clone* message. This operation causes the creation of a new object in which the instance variables of the class object are cloned, and the methods are accessed by delegating a message to the class object. If a subclass receives a message which is implemented by the "superclass", the subclass simply delegates the message to the superclass. Recall that in a delegation based system the *self* variable is bound to the object which receives the message first. In order to access a method which is defined in the subclass, the superclass can simply delegate a message to the object *self*. However, this strategy doesn't prevent people from building "spaghetti" object systems. It is still possible to create objects which are solely depending on "instance" objects.

The reverse, however, is not true. It is not possible to build a prototype based system on top of a class based system. The reason is that the *self* variable in a class based system is bound to the object receiving the message. If a message is delegated from an instance object to another instance object, the *self* variable will refer to the object which receives the delegated message. For a delegation based system, the self variable needs to be referring to the object which delegated the message. This can be solved by passing a reference to the original object as a parameter, when a message is delegated. However: I consider this a hack. Another possibility would be to attach code blocks to class variables. When an object is instantiated, the code block is cloned in the new object. This I consider also a hack[2].

Prototype based systems always need to resolve method and instance variables starting from the object which originally received the message. If that object has implemented its own idiosyncratic method or instance variable, a new lookup has to be done along the delegation chain, starting from the *self* variable. If the prototype based system uses messages to communicate between objects, executing some expressions might end up in sending and receiving a lot of messages between objects. In class based systems, this is not the case. It is not possible to go back on the inheritance chain when resolving methods and/or instance variables. This also means that class based systems can be implemented without sending messages to objects, *C++* for example, has implemented the inheritance chain in a simple

[2] There exists a language which allows a hybrid form of class and prototype based objects. Whenever an object is instantiated in kok¨«˙ [6], an empty sub-class is created. Messages to this empty sub-class are delegated to the "real" class. If an object wants to implement special behaviour, you can install a new method (or instance variable) in the dynamic sub-class by sending the sub-class an *install-method* message.

array of functions and/or instance variables[3].

5 Conclusions

Although prototype systems and class based systems have many things in common, there are some subtle differences between the two approaches. When it comes to building large systems with many objects, using a class based system seems to make more sense. The reason for this is that class based systems force the programmer to think about the common behaviour of a set of objects, and to place this common behaviour in a seperate class object. In a prototype based system, the programmer can put common behaviour in any object, thereby making dependencies between live objects. Changing a prototype object, which is simplier than changing the class, might result in obscure errors in other objects which are depending on the prototype object. If, however, small systems are built with many single objects, the prototype based system might be more reasonable since the programmer doesn't need to create a class description first before objects can be created: the programmer simply programs the object.

References

[1] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1991.

[2] Adele Golberg and David Robson. *SmallTalk-80, The Language*. Addison-Wesley, 1989.

[3] H. Lieberman. Using prototypical object to implement shared behaviour in object-oriented systems. In *OOPSLA '86 (october). Object-Orientated Programming Systems, Languages and Applications. Conference Proceedings*, volume 21, pages 214–223, Portland, OR, USA, November 1986.

[4] Bjarne Stroustrup. What is "object-oriented programming"? In G. Goos and J. Hartmanis, editors, *Proc. European Conf. on Object-Oriented Programming*, Paris (France), June 1987. Springer-Verlag, Lecture Notes in Computer Science no. 276.

[5] David Ungar and Randall B. Smith. Self: The power of simplicity. In Norman Meyrowitz, editor, *Proceedings of the OOPSLA'87 conference*, pages 227–242, Orlando FL (USA), October 1987. ACM.

[6] Guido van Rossum and Jelke de Boer. Interactibely testing remote servers using the python programming language. *CWI Quarterly*, 4(4):283–303, December 1991.

[3] Note that I'm not trying to say that prototype based cannot be implemented efficiently but these systems always need to traverse the complete dependency chain from the base object to resolve a method or instance variable