Object Shadowing - a Key Concept for a Modern Programming Language

Marc Conrad[1], Tim French[1], and Carsten Maple[1]

University of Luton, LU1 3JU, UK
marc.conrad@luton.ac.uk, tim.french@luton.ac.uk,
carsten.maple@luton.ac.uk

Abstract. Object shadowing is an established programming concept that is supported by languages such as LPC. Thus far, shadowing in LPC appears to have been driven mainly by pragmatics rather than having followed any particular formal evolutionary path. In this paper, we describe the concept of object shadowing and suggest a number of suitable application areas that would benefit from further development of the concept. We explain how the concept must be specifically tailored for these application areas.

1 Introduction

LPC is an interpreted language that was created by Lars Pensjö for his invention LPMUD (Lars Pensjö Multi User Dungeon). MUDs, which can stand for Multiple User Dimensions, Multiple User Dungeons, or Multiple User Dialogues, are computer programs in which users control a computerised persona or character (for further details see [14] for example). Pensjö developed his own flavour of C, Lars Pensjö C or LPC, in 1988 as a simple language for the development of MUDs. As the name suggests, the basic syntax of LPC owes much to the language C, but with the addition of an object-oriented structure. There are no classes in LPC. Objects are either instantiated at run time from a file, or otherwise cloned from other run-time objects.

The evolution of LPC has been highly pragmatic and driven by the demand from the various programmers of MUDs, rather than following any particular systematic evolutionary path or academic imperative. Thus, shadowing support in LPC must be seen in this context: it has proved to be useful "as is" but has not, thus far, been evaluated academically. We hope to encourage further systematic research into this relatively simple concept with a longer-term goal of direct implementation within a mainstream language. In order to initiate this objective we proceed to present a description of shadowing from first principles. We then go on to suggest a number of suitable application domains (indicating the potential benefits of shadowing for both the domain and for the evolution of shadowing itself as a concept). Finally we tentatively suggest how wider adoption of shadowing within these areas might enable us to establish a more conceptually unified approach to aspects of dynamic inheritance in general.

2 The functionality of a shadow

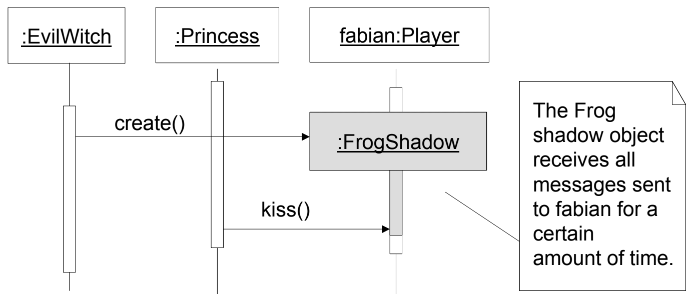
:EvilWitch
:Princess
fabian:Player
create()
:FrogShadow
kiss()
The Frog shadow object receives all messages sent to fabian for a certain amount of time.

Fig. 1. Sequence diagram for a typical situation in a computer game. An evil witch transforms a player (named fabian) into a frog. For that a suitable shadow object is created and added to the player. All messages sent to the player are then received by the shadow. The shadow basically "becomes" the player. In this example the shadow defines a method *kiss()* that destroys the shadow when it is sent by a princess.

The idea with the shadow functionality is to *mask* one or more methods in a target object (the "shadowed" object) for a certain amount of time. A shadow is applied at run-time rather than compile-time, in response to the dynamic need of programming applications (see figure 1). Every method that is sent to the shadowed object will then be processed first by the shadow (if the shadow defines it). If a method is not explicitly defined in the shadow it will be passed on to the shadowed object as if there were no shadow in between.

In the following we list some properties of shadows in LPC to give a better understanding of the concept. However, we should bear in mind that a shadow implementation in a future language may require "better" or more evaluated properties. It is partly the purpose of the paper to encourage a discussion on these possible variations of the shadow concept.

- A shadow cannot be shadowed, however an object can have more than one shadow.
- When an object returns "true" on a method call `query_prevent_shadow()` it cannot be shadowed. A special situation occurs, when a *shadow* defines query_prevent_shadow() returning "true": This shadow disallows any further shadows on the shadowed object.
- A method cannot be shadowed if it is declared `nomask`.
- Attributes cannot be shadowed.
- Only calls made from "external" objects can be shadowed. We illustrate this in an example: Let us consider a Java implementation of shadows, in which a method `bla()` of an object `fred` is shadowed. Then a method call `fred.bla()` will be received by the shadow while a call `bla()` in the object

`fred` itself will go to `fred` directly. We enable the possibility of shadowing an "internal" call by artificially making it external using the syntax `this.bla()`.

A Java package that implements shadows can be found on the web site [5] that also contains a number of examples. It extends the idea of shadowing objects in LPC to the concept shadowing of classes: A shadow of a class means that a shadow is thrown over every object that is instantiated from this class.

3 Application areas for Shadows

The first two examples in this section are in the context of software development. The third and fourth example show how shadows can be used to implement "non-standard" inheritance. Subsection 3.5 finally gives a provocative view on the relationship between shadows and inheritance.

3.1 Deprecated methods

Software Libraries are under constant evolution and it is a fact that sometimes methods are replaced by other methods that may be more consistent with, for example, naming conventions. The problem is that legacy code usually continues to use deprecated methods. Hence it is impossible for the provider to remove them completely from the library.

A shadow system could help the provider of the software to separate an object into two parts. The actual, official version of the object where the deprecated methods have been removed and a collection of shadows that implement deprecated methods. In cases that the deprecated method is necessary for an application the object would then be shadowed. This reduces the overhead of having the additional, deprecated, method as part of a library to applications where these methods are used.

Syntactically this could be provided similar to the Java properties mechanism: The required shadows for each class are specified in "environment variables". The class loader then can automatically decide when to add a shadow depending on the values defined by the environment.

3.2 Prototyping

Following the suggestion in the previous section that shadows could be used for "fading out" deprecated methods, in a similar manner, shadows could also be employed in prototyping for software development. This is especially the case when a development process starts from an existing library and it is vital that the library is not changed (or that it is not *possible* to change the library because, for example, it is bought from an external supplier or because of copyright issues).

A shadow then could be used to change the behavior of a class or object temporarily or for a well defined situation during development. An evaluation with extended experiments using the shadow may then be used to collect arguments for or against the adoption of the proposed change of the class.

Shadows can also be used (automatically) in a concurrent version system (CVS) to implement branching in software development.

3.3 Reclassification and Dynamic Inheritance

Reclassification and a special case of it, dynamic inheritance, is the process of changing the class of an object at run-time. These are extremely useful concepts and as such there have been ideas developed to overcome the limitations of C++ in this area: dynamic inheritance in C++ is discussed in [6]. Automatic reclassification based on the value of predicates is implemented as *predicate classes* [2] in Cecil [3]. In [11] a Java extension featuring dynamic inheritance is proposed. Dynamic inheritance is further supported in Lava as part of the Darwin project [10]. However, the most significant approach for reclassification can be found in *Fickle* (e.g. [9], [1], [8]).

It is not a coincidence that the *Fickle* example in [8] is in the context of a computer game. A *Player* that is (an instance of) a *Frog* is reclassified to a *Prince* after being kissed. It is exactly this kind of problem that is, in practice (namely using LPC), pragmatically solved with shadows. The main goal in reclassification is the change of the behavior of an object: The behavior of a *Frog* is different from the behavior of a *Prince*. This is achieved by the application of a shadow to a player object. Different shadows may change the default behavior of the player. For instance, a magic spell might add a "Frog shadow" to the prince, thus enabling "Frog"-like behavior. A kiss of a princess may then replace the "Frog shadow" by a "Prince shadow" (see also figure 1).

3.4 Interclassing

For a motivation of Interclassing we refer the reader to [13], [7] (in a general context), or [4] (in a mathematical context). The basic underlying principle in interclassing is the insertion of a new class in an existing inheritance hierarchy. Here the assumption is that the inheritance hierarchy to be modified is in the context of an existing library that cannot be changed (for instance because of a copyright or that it has to be left unchanged for existing applications etc).

The shadowing concept can be useful in such situations. We illustrate this using an example presented in [7]. An existing hierarchy with *Parallelogram* as a parent of a class *Square* should be extended by a class *Rectangle*. In [7] the proposed solution is the introduction of a "reverse inheritance" (specialization) relationship established from the *Rectangle* to the *Square*. A shadow – in contrast to that – could change the methods in the *Square* directly. A shadow class inherits from the Rectangle and shadows the *Square* (see figure 2).

3.5 Specialization and Inheritance

The usual way to implement specialization in a class-based language is to derive a child class and make appropriate changes. However, alternatively we could also

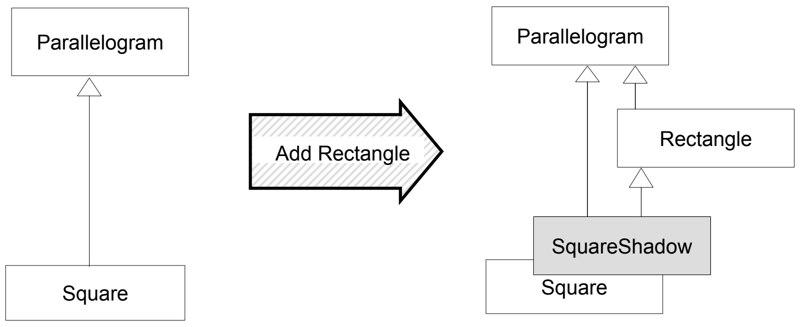

Fig. 2. The Rectangle is added into the *Square* / *Parrallelogram* hierarchy. However as we assume that *Square* cannot be recompiled it has to be shadowed to become a child class of *Rectangle*. Note that on the right hand side, because of the shadowing and because the *Rectangle* is also a child class of *Parallelogram*, the *Square* – *Parallelogram* generalization is irrelevant

think of instantiating an object and then shadowing the object where the shadow provides the specialized functionality. When using the well known Vehicle/Car inheritance example (The vehicle defines a method move() that is implemented in Car), we could equally think of an instantiation of a vehicle object that is then shadowed with a shadow that implements the move() method. This simple example illustrates the power of the shadow concept: It might even be able to *emulate* inheritance. That means we could think of a programming language that has shadows as a first class feature and derives inheritance as a special application of shadows.

4 Conclusion

We have discussed the concept of shadows, a technique already employed in programming applications in which there is a need for dynamic updating, most notably games programming. We have shown that shadows provide a useful tool in a number of other application areas. Moreover it would be a *unified* approach for a diversity of applications such as deprecating methods, prototyping, reclassification, interclassing, and specialization. Certainly a dynamic object oriented language would profit when shadows are embodied as a key concept.

References

1. D. Acnona, C. Anderson, F. Damiani, S. Drossopoulou, P. Giannini, E. Zucca. *A type preserving translation of Fickle into Java* Electronic Notes in Theoretical Computer Science 62 (2001). Available at: http://www.elsevier.nl/locate/entcs/volume62.html
2. C. Chambers. *Predicate classes*, in: *Proceedings of the ECOOP'93*, volume 707 of *Lecture Notes in Computer Science*, pages 268–296, Kaiserslautern, Germany, July 1993.

3. C. Chambers. *The Cecil Language: Specification & Rationale*, avialable at: http://www.cs.washington.edu/research/projects/cecil/www/pubs/cecil-spec.html.
4. M. Conrad, T. French, C. Maple, S. Pott: *Approaching Inheritance from a "Natural" Mathematical Perspective and from a Java driven viewpoint: a Comparative Review*, Preprint available from: http://ring.perisic.com.
5. Marc Conrad. *The com.perisic.shadow package*, http://perisic.com/shadow.
6. James Coplien. *Advanced C++ programming styles and idioms*, Addison-Wesley 1992.
7. Pierre Crescenzo, Philippe Lahire. *Using Both Specialisation and Generalisation in a Programming Language: Why and How?* Advances in Object-Oriented Information Sytems, OOIS 2002 Workshops, Montpellier, pages 64–73, September 2002.
8. F. Damiani, M. Dezani-Ciancaglini, P. Giavinni *Re-classification and Multi-thrading: Fickle$_{MT}$*, in: *SAC 2004*, to appear.
9. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini and P. Giannini. *Fickle: Dynamic object re-classification*, in: *ECOOP'01*, LNCS **2072** (2001), pp. 130–149.
10. *The Darwin Project*, http://javalab.iai.uni-bonn.de/research/darwin.
11. Günter Kniesel. *Darwin & Lava - Object-based Dynamic Inheritance ... in Java*, Poster presentation at ECOOP 2002.
12. Lars Pensjö. *LPC*. Documentation available at: http://www.lysator.liu.se/mud/lpc.html
13. P. Rapicault, A. Napoli. *Evolution d'une hirarchie de classes par interclassement.* In: LMO'2001, Hermes Sc. Pub. *"L'objet"*, vol. 7 - no. 1–2/2001.
14. Ronny Wikh, *LPC*, available at: http://genesis.cs.chalmers.se/coding/lpcdoc/lpc.html (last update 2003)