Prototype based languages

Author

Tomas Billborn & Mallela Srinivasa Rao

Abstract
When object oriented languages are brought up as subject most of us think of languages that support data abstraction by providing data templates called classes. This since class-based languages are more common and familiar to most programmers than prototype-based ones. It may seem a bit peculiar to talk about objects without classes, but that is the subject of the paper. The paper will give an overview description of how prototype based languages function and examine some of their advantages and disadvantages.

Content

Introduction....4
The object orientated hierarchy....5
Prototype based languages....5
  Normal objects....5
  Prototypes....5
  Traits....5
Creation of objects.... 5
  Ex-nihilo....5
  Cloning....5
    Shallow cloning....5
    Deep cloning....6
  Extension.... 6
Inheritance....6
  Delegation-based inheritance....6
  Embedded--based inheritance....6
Sharing achieved by shallow cloning....6
Sharing achieved by delegation....6
  Implicit delegation....6
  Explicit delegation....6
Class-like sharing in prototype based languages.... 6
Dynamic inheritance....7
Advantages of prototype based inheritance.... 7
The need for abstraction....7
The origin of prototype based languages....7
Conclusion....8
References....9

Introduction
Since the late 1980s prototype based languages and programming has been proposed as an alternative to the class-based approach of object orientation programming. Prototype based languages are mainly characterized by the fact they have rejected the notation of classes. Instead each object holds its own description and the lack of bound to a class allows each object is allowed to evolve on its own. The most important idea of the prototype-based is that concrete objects are only mean to model applications. Prototypes are not meant to be abstractions like classes and are not linked in any other way to other objects that would describe them as in the class-based approach.

The object orientated hierarchy

The prototype based languages are a sub-sub branch of the object orientated hierarchy.

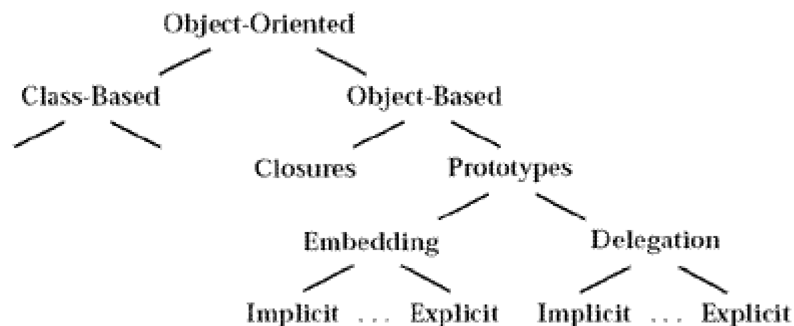

Figure 1: A Hierarchy of Object Oriented Features.

[1]

Prototype based languages

In a prototype based language there are no distinction between class and instance, there are simply objects. There are three types of objects, normal objects, prototypes and traits. Although the last two are not "true" objects.

- **Normal objects**: intended only to be used and to carry local state and should rely on traits for their methods [1].
- **Prototypes**: a special kind of object that are intended to be blueprints from which normal objects (and prototypes) are generated via cloned. Prototypes act like classes, but they are fully functional objects [1,2,3].
- **Traits**: abstract descriptions of objects. They are similar to classes, but are not allowed to possess state. A trait provides a set of methods that implements behaviour. Traits are intended only as the shared parents of normal objects: and should not be used directly or cloned [1,3,4].

Creation of objects

There are three primitive ways to create objects in prototype based languages:

- **Ex-nihilo:** creation of objects with no parents. If ex-nihilo creation is allowed, they can come in two forms. As empty new objects, or new objects with initial behaviour [2, 11].
- **Cloning:** new objects are created by making copies of objects that already exists. There are two types of cloning.
    - **Shallow cloning:** "takes a copy of the state of the object and shares the behaviour with the object from which it was cloned by means of delegation" [5].

[1] All "functions" mentioned below may not exist in all prototype based languages, but at least one from each category do.

- **Deep cloning:** "takes a copy of both the state and the behaviour of the object" [5].

- **Extension:** "to create an object that shares knowledge with a prototype, you construct an extension object, which has a list containing its prototypes, which may be shared by other objects, and personal behaviour idiosyncratic to the object itself" [2].

Inheritance
In prototype based languages there are four categories of inheritance.

- **Delegation-based inheritance:** reusing methods by sharing them with other objects. This can be done implicitly or explicitly.

- **Embedded-based inheritance:** reusing methods by extracting them from other objects. This can be done implicitly or explicitly.

Sharing achieved by shallow cloning
When an object is cloned by using shallow cloning the newly cloned object receives that objects state and behaviour [2].

Sharing achieved by delegation
The basic idea of delegation is to forward requests that cannot be handled by an object to the prototype which it is based on. As long as an object and its parent exist delegation achieves life-time sharing. There are two alternatives of delegation, implicit and explicit [2].

- **Implicit delegation:** when an object cannot answer a message, the interpreter automatically delegates it to another object; objects have a parent link to indicate to the interpreter to which objects messages should be delegated [2].

- **Explicit delegation:** the delegation of messages is done explicitly for each message to be delegated; the delegating object names the object to which the message has to be delegated [2].

Class-like sharing in prototype based languages
As shown cloning and delegation achieve different kinds of useful sharing, but is it the same kind of sharing found in traditional class-based object orientated languages? The answer to this is no, since reuse from prototypical instances using both cloning and delegation cause semantic problems.

The problem with cloning is that when modifying the prototypical instance, there is no way to insure that the family of objects cloned from this prototypical instance will be homogeneous; objects cloned before the modification will not be affected, thus they will diverge from the new concept implemented by the updated prototypical instance. With delegation, the problem is seen the other way around. When the prototypical instance is modified, all objects delegating to it are affected by the modification; if the prototypical instance looses its status of being prototypical and instead becomes an exceptional instance of the concept, the original concept is lost.

In both cases, a solution may be to make prototypical instances immutable, but this solution is contrived in two ways. First, it introduces a distinct
kind of objects in the prototype-based model, which then looses part of its elegance and simplicity. Second, by making prototypical instances immutable,
we loose the capability of normally speaking about these objects. We conclude that prototypical instances are not the right device to implement [2].

Dynamic inheritance
Some prototype based languages like SELF also supports dynamic inheritance, the ability of an object to change the code that it inherits at run time [6].

Advantages of prototype based inheritance
In prototype-based inheritance there are two big advantages, simplicity and correctness. Simple since the concept of class is eliminated and rather then having two inheritance relations as in class-based languages there is just one. This reduces the cognitive load of the programmer. Its more concrete since programming one modifies the objects directly and not class indirectly to achieve the effect on an object [7].

The need for abstraction
Despite all the advantages of the prototype based languages they still are faced with two major problems. One is related to the lack of strong identity for prototypes and the other one to organisation of programs. The programming model of prototype based languages are simpler and more flexible, but when faced with larger programs the lack of satisfactory way to define concepts makes it self visible.
Our other problem is related to how several prototype based system try to tackle the problem of organisation of programs with class like constructs. Some of these have been integrated to the system kernel and are not accessible to the programmer and therefor not much helpful to explicitly define abstractions [8].

The origin of prototype based languages
The definite origin of prototype based languages is hard to determine, but the creators may been inspired by Wittgenstein's criticism of classification [3,9] or by the prototype theory [10] by Eleanor Rosch from the mid seventies. Either way the papers referred to above might not hold the answer to what exactly that is, but will present the reader with some interesting philosophical background on prototypes on less technical grounds.

Conclusion
We have looked at a general description of the prototype based languages and come to the conclusion that: The main characteristic of prototype based languages is the fact that they are classless. There are several different prototyped languages and all do not posses all the same functionality, but they do posses at least one of the different types of descriptions of type of objects, creation of objects, inheritance that has been shown.

It has also been shown that prototype based languages possess several advantages over class-based ones.

- Prototype based languages are conceptually easier to understand then class-based languages, since there is just one type of object.

- There is only one kind of relationship, inheritance from parent, not as in for example class-based languages instance and subclass.

- Prototype based languages are more concrete since copying and modification of objects are more direct.

- Prototype based languages are more flexible since it can hold behaviour or, through copy act as a source for new objects.

With the advantages also comes some disadvantages.

- Prototypes can be inadvertently modified which might affect future clones. This can lead to subtle bugs. This is referred to as the prototype corruption problem.

- The construction of certain objects requires a certain construction plan to be executed. In Class Based Languages, it is possible to formalise this plan as a class constructor. In Prototype based languages there is no such thing: objects are created by cloning other objects and it requires a lot of discipline from programmers to make sure certain procedures are followed.

- Prototype based languages suffer from a re-entrancy problem. A problem with ”naive prototype-based thinking” is that, when cloning objects not only their state but also their behaviour gets copied. This means that an object with 10 methods will give rise to 50 methods when cloned 5 times, or at least 50 pointers all pointing to the same methods. The problem can be circumvented using the traits technique, but this is sometimes problematic as the traits technique heavily interferes with the inheritance hierarchy.

References

1. [Cardelli96] Luca Cardelli, Object-based vs. Class-based Languages
http://research.microsoft.com/Users/luca/Slides/1996-05%20Class-based%20vs%20Object-based%20Languages%20(PLDI%20Tutorial).pdf

2. [DMC92] Christophe Deny, Jacques Malenfant and Pierre Cointe, Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation
http://portal.acm.org/citation.cfm?id=141954&coll=GUIDE&dl=GUIDE&CFID=44159798&CFTOKEN=15280960&ret=1#Fulltext

3. [Pesonen01] Juha Petteri Pesonen, Psychological criticism of the prototype-based object-orientated languages
http://www.helsinki.fi/~jppesone/papers/kandi.html

4. [Merizzi05] Nicholas Merizzi, Object Oriented Classes, Objects, Inheritance and Typing
http://www.cas.mcmaster.ca/~carette/CAS706/2005/OOpresentation.pdf

5. [DM02] Jessie Dedecker and Wolfgang De Meuter, Using the Prototype-based Programming Paradigm for Structuring Mobile Applications
http://prog.vub.ac.be/Publications/2002/vub-prog-tr-02-28.pdf

6. [BAB04] Andi Bejleri, Jonathan Aldrich, and Kevin Bierhoff, A type checked prototype-based model with linearity
http://reports-archive.adm.cs.cmu.edu/anon/isri2004/CMU-ISRI-04-142.pdf

7. [Smith95] Walter R. Smith, Using a Prototype-based Language for User Interface: The Newton Project's Experience
http://wsmith.best.vwh.net/OOPSLA95.pdf

8. [Bardou00] Daniel Bardou, Inheritance Hierarchy Automatic (Re)organization and Prototype-Based Languages
ftp://ftp.inrialpes.fr/pub/romans/publications/bardou00a.pdf

9. [Aerts01] Kris Aerts, Visto: A Declarative Methodology for Graphical User Interfaces, based on Haskell
http://www.cs.kuleuven.ac.be/publicaties/doctoraten/cw/CW2001_1.pdf

10. [Pesonen02] Juha Petteri Pesonen, Concepts and Object-Oriented Knowledge Representation
http://ethesis.helsinki.fi/julkaisut/hum/psyko/pg/pesonen/concepts.pdf

11. [Wuyts05] Roel Wuyts, Programming Languages
http://www.ulb.ac.be/di/rwuyts/INFO020_2004/09-PrototypesAndSelf.pdf