Prototype-based programming

Marcus Arnström, Mikael Christiansen, Daniel Sehlberg

May 29, 2003

Abstract

The basic principle of prototype-based programming languages is to remove all abstraction and focus on creating concrete objects. New objects are created by copying exsisting ones. This is a quite different view of object-oriented programming, totally omitting classes. Proponents of this model claims that it is easier to think in prototypes than in classes. Some examples of prototype-based languages are Self, Kevo, NewtonScript and JavaScript. The general principles are the same for all these languages, but the implementation can vary greatly. Prototype-based languages implement inheritance in a more flexible way, which can be applied and removed during runtime.

Chapter 1

History

This chapter is meant to be an introduction to the historical background of classification, prototypes, and the prototype theory.

1.1 Early history of classes

Over two thousand years ago, a distinction between forms ("ideal" descriptions of things) and instances of these forms were made by the Greek philosopher Plato[Tai99]. Plato found the world of ideas much more important than the world of concrete objects.

This early research of (biological) classification was continued by Aristotle, Plato's student. Aristotle looked at all things in nature when performing his classifications: minerals, plants, animals and so on. This process was carried out using very much the same principles used in object-oriented programming today. Objects with similar features and properties belongs to the same category (class). New categories can be created from existing ones (subclassing), provided that the new ones contain at least the properties of the defining categories.

Aristotle's work has had an important side effect - In many of today's cultures, there is a common idea that there is one correct taxonomy of natural things, even though categorizations vary heavily depending on who is doing the categorization and for what purposes.

The way of categorizing objects by looking at common properties has been used by numerous scientists since, including Swedish natural scientist Carl von Linné.

1.2 Following criticisms

In the 19th century, criticisms of the "classical" Aristotelian view appeared [Tai99]. British philosophers W. Whewell and W.S. Jevons pointed out that there are no general rules to use when choosing properties for a classification. Also, the process of classification requires a creative and inventive mind. This led to their conclusion that there were no classifications that could be objectively defined as "correct" classifications.

An important criticism was presented in the 20th century by Ludwig Wittgenstein. There are some things that are very difficult to find common properties for, even though they may belong to the same category. For instance, the concept of a work of art is something that is extremely difficult to define as a class using common properties. There are really no limits for what art can be. Therefore finding properties that define art is an impossible task. Another example is the concept of a game. A game can be many things with completely different properties. All games still belong to the same category.

After pointing this out, Wittgenstein took perhaps the first step towards prototype-based programming. He defined the concept of a *family resemblance*. Games do not have shared properties, but they have a form of family resemblance. Chess resembles the family of activities called games, hence it is a game.

1.3 The prototype theory

Several people were inspired by Wittgenstein, and continued his research. In the 1970s, Eleanor Rosch first laid out the prototype theory. It stated that most categories have examples that best represent the category, and that example is called the prototype. Also, she stated that human senses is an important factor when categorizing. One of the important critisisms she had of the classical approach was:

If things are categorized after properties they all share, then there can be no such thing as a "better" example of a category. However, by for instance asking people for an example of a number, One can easily prove that small integers are "better" examples of numbers. Several other significant points were made in the prototype-theory, for a detailed discussion, see "Women, fire, and dangerous things" by George Lakoff [Lak87].

1.4 Classification in programming languages

Nowadays, most object-oriented programming languages are similar to the Aristotelian classic model. All objects of a certain class share exactly the same properties (data members and methods). Also, the way of making new classes by inheriting from others is very similar to the way Aristotele created new objects from existing ones. The similarities between OOP-laguages and the Aristotelian model also includes the limitations when it comes to modeling the real world. Most of todays applications do not suffer from that fact though, since most problems can be modeled fairly well with the classic model. The fact remains, current object-oriented languages are based on a model that philosophers and cognitive psychiatrics have abandoned a long time ago, and there are concepts which you simply cannot define in terms of shared properties, like traffic jams or the greenhouse effect.

So, are prototype-based languages any better in terms of benefitting from the prototype-theory? Unfortunately, as Taivalsaari states, "current prototype-based object-oriented languages are poorly developed when it comes to taking

into account the conceptual and philosophical benefits of the prototype-based approach". Current prototype-based languages seem to focus on other benefits, like the increased flexibility in manipulating objects, so there´s still a lot of research to be made to break free of the obsolete Aristotelian "classic" theory.

Chapter 2

Classes vs. prototypes

2.1 What is prototype-based programming?

Just because a programming language is object-oriented does not mean that it uses classes. Another model of object-oriented programming is based on prototypes, in which every object is concrete and manipulable. These objects, often referred to as prototypes, resemble an instance of a class in class-based programming. They are not as tightly bound to other objects as classes are bound to their superclasses. With an object from a prototype-based language, you can add or remove variables and methods at the level of the single object, and you do not have to worry about if the object still fits in the system. When you are creating new objects in prototype-based programming, you copy an existing object. This is called *cloning.* Inheritance is replaced by some other less class-centered mechanism [Tai99]. The most common substitute for inheritance is something called *delegation* which you can read more about in section 4.1. There are many prototype-based languages and they differ in implementation, and also a little in ways of thinking. Chapter 3 and 4 will describe the differences more in depth.

2.2 Abstract vs. concrete

You can hardly describe the prototype-based paradigm without comparing it to the class-based, because they are actually pretty similar. The main difference between class-based languages and prototype-based languages is that you do not have any abstract definitions in the latter. Instead everything is a concrete object. That is why the term 'prototype-based' sometimes is replaced by 'object-based'. Sophia Drossopoulou compares the differences between classes and prototypes to building a house [Dro]. Building an object in a class-based language is like building a house from a plan, while building an object in a prototype-based language is like building a house like the neighbors.

As described by Dony, Malenfant and Cointe [DMC92], the prototype paradigm is justified in two fundamental ways. First, people generally grasp concepts by creating concrete examples rather than abstract descriptions. Class-based languages force people to work in the opposite direction, from abstract

to concrete. Second, class-based languages seem to unnecessarily constrain objects. Both by disallowing distinctive behavior for individual objects among their instances and by forbidding inheritance between objects to share values of instance variables.

While class-based languages describe groups of objects with abstract classes, prototype-based languages do not rely so much on categorization and classification, but rather try to make the concepts in the problem domain as tangible and intuitive as possible [Nob97]. According to James Noble [Nob97], prototype-based thinking is not restricted to programming. When one designs a system, does one think in prototypes or classes? In some design methods, people think in objects even if they speak of them as classes.

Two very famous languages that you might have thought were related are actually pretty good examples of differences between class-based and prototype-based languages. Table 2.1 below illustrates differences between Java and JavaScript [Jav00].

Class-based (Java)	Prototype-based (JavaScript)
Class and instance are distinct entities.	All objects are instances.
Define a class with a class definition; instantiate a class with constructor methods.	Define and create a set of objects with constructor functions.
Create a single object with the `new` operator.	Same.
Construct an object hierarchy by using class definitions to define subclasses of existing classes.	Construct an object hierarchy by assigning an object as the prototype associated with a constructor function.
Inherit properties by following the class chain.	Inherit properties by following the prototype chain.
Class definition specifies all properties of all instances of a class. Cannot add properties dynamically at run time.	Constructor function or prototype specifies an initial set of properties. Can add or remove properties dynamically to individual objects or to the entire set of objects.

Table 2.1: Comparison of class-based (Java) and prototype-based (JavaScript) object systems

There is a family of languages called *frame languages* which are used very much within A.I. to represent human knowledge in knowledge-based systems. They use the functionality to add and remove slots dynamically, which indicates that they are prototype-based, alternatively closely related to prototype-based languages.

Mikel Evins [Evi93] writes that prototypes eliminate the potentially complicated meta-object problem of defining the class of a class. If a class is an object, then of what class is it an instance? Different languages have different solutions to this problem. In C++, classes are simply compile-time conventions and have no semantic meaning during runtime. They solve the meta-object problem by omitting it. Smalltalk uses a pretty big inheritance-tree to describe the meta-classes. Each object is the instance of a class, and each class is an instance of a meta-class. The meta-classes are subclasses of the class Class, which is an instance of itself. In prototype-based languages an object is just an object, or a copy of another object.

2.3 Flexibility

Prototype-based languages are generally more flexible than class-based since an object does not have to look like its parent, addition and deletion of slots are dynamic as well as cloning and delegation. One effect of dynamic delegation is that prototype-based languages can change their parents during runtime. A good example of this, as explained by David Ungar [SLSTU], is that if an object represented a running program, it could have a parent that describes a window. If the window is minimized, the object could switch parent to an icon. When the icon is double-clicked, the object switches back to the window-parent and so on. Of course, since it is so flexible, a user can make ill-planned changes that will make the program behave unpredictible. This is a typical tradeoff for the flexibility.

Perhaps the largest tradeoff is the runtime support needed since the relationships between objects can be pretty complex. That support costs both space and speed. Although Self has shown that prototype-based languages can be fast, and NewtonScript has shown that they can be small, the question is how to make them both small and fast at the same time [Evi93].

2.4 A simple prototype-based language

Alan Borning [Bor99] describes the simplest possible prototype-based language in the book 'Prototype-Based Programming' as follows: Suppose that every object is self-contained, so that an object consists of *state* and *behavior*[1]. The object can be asked for information, to change its state, or to change its behavior via messages. When a new object is created it will be a complete copy of some existing object, inheriting both state and behavior. Once the copy is made there are no relations between the new object and the original.

After describing this very simple language, he goes on by asking if something is missing, and states that there should definitely be some classification of kinds of objects, either by message protocol or by representation [Bor99]. Also, there should be some way of updating groups of objects at one time. Since both of these features are available in object-based languages, we can say that this model needs to borrow some ideas from them.

2.5 Hybrid languages

Class-based languages have often been criticised for their rigidity. Prototype-based languages, on the other hand, are usually considered too flexible [SDM95]. According to 'The Treaty of Orlando' [SLU89], much can be gained by combining the strength of the two paradigms. In 1995, when Stayaert and De Meuter wrote their article 'A Marriage of Class- and Object-Based Inheritance Without Unwanted Children' [SDM95], both communities (class and object) had researched some compromise, but without entirely successful results.

[1] The state is represented by instance-variables and the behavior by methods.

Jaques Malenfant have written an article called 'Object-centered programming' in which he critizises the prototype advocates [Mal]. He writes that unfortunately rejections of the abstraction hides the greatest benefit of prototype-based programming, namely the focus on direct manipulation of concrete entities. Therefore he proposes a new paradigm entitled *object-centered programming.* He goes on by stating that the rejection is jeopardizing the chances of survival of prototype-based languages. The future of prototypes depends on their ability to include some form of abstractions.

Chapter 3

Differences within prototype-based languages

All prototype-based languages differ from each other, just like all class-based languages. Everything that is described in this report is not necessarily true for every prototype-based language, but for most of them. This chapter will compare some variations within the paradigm.

3.1 Slots or methods and variables

Some prototype-based languages distincts variables from methods while other treats them the same way, in which case they are all called slots. Both alternatives have advantages and disadvantages. Self advocates slots with the motivation that they are more flexible. They allows users to access both methods and variables in the same way, and they permit overriding of an attribute with a method and vice versa [DMB99].

3.2 Inheritance and sharing

There are two ways of cloning objects. Languages that does not support delegation[1] creates something that is called augmented clones. These clones will be created from a prototype, but after creation they stand on their own feet. If the prototype's[2] behaviour or state is changed, the clone will not be affected in any way. This is called *creation-time sharing,* and prevents objects from being unexpectedly changed through their clones or prototypes. The opposite is languages that supports delegation. For them, there are two alternatives of creating new objects, either by cloning or by extension.

[1] Kevo is one example, and when it does not support delegation, it does not support creating objects by extension either

[2] An object's prototype is the same as its parent, just another word for the same thing

3.2.1 Property-share

Extension creates what is called *life-time sharing*. To explain that, assume the object *Rectangle* which have two attributes (width=4, height=4) and one method (getArea). We create a new object *3D_Box* as an **extension** of Rectangle. When 3D_Box is created, we gives it a new attribute (depth=4). We now have a 3D_Box with one prototype-pointer to Rectangle and one attribute of its own. Rectangle still have width, height and getArea (Figure 3.1). Whenever a user asks 3D_Box for its width, height or getArea those questions are forwarded (delegated) to Rectangle. In the other direction, if a new attribute (color) is added to Rectangle, it will also be available for 3D_Box. If Rectangle's width is changed to 6, then so will the width for 3D_Box since it is the same attribute. When an object shares all or some of its properties with another object, it is called *property-share*.

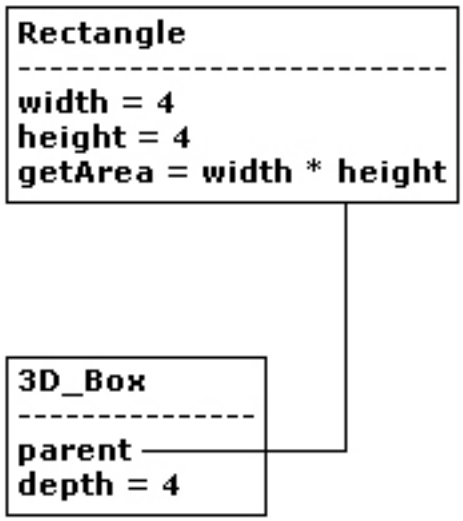

Figure 3.1: 3D_Box is an extension of Rectangle. Width, height and getArea are shared

3.2.2 Value-share

If you create a **clone** instead of an extension, the clone will have all attributes for itself and will not be affected by value changes in its prototype. The same rules applies in the other direction (i.e. the clone cannot change the values in the prototype). If we use the example with Rectangle and 3D_Box above, the values of width and height is set as default values to 3D_Box, but those can later be changed. If Rectangle's width again is changed to 6 after creation, the width of 3D_Box will still be 4. On the contrary, if 3D_Box's width is changed to 6, the width of Rectangle will still be 4. Alternatively, the clone can use its parent's getArea to save space (Figure 3.2). When you have this relation, which is called *value-share,* the two objects can evolve independently [DMB99]. There can also be special cases where a clone have some attributes local, and shares some attributes with its prototype.

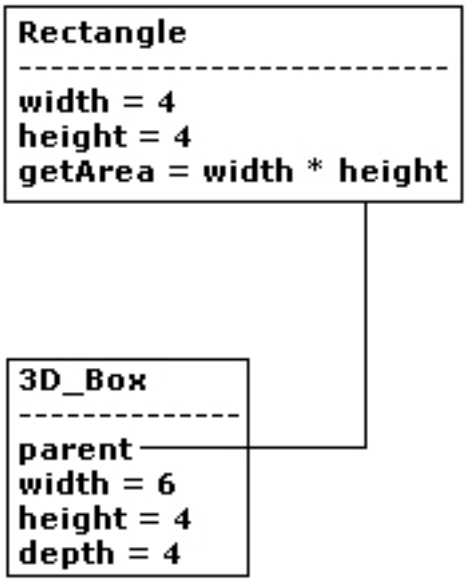

Figure 3.2: 3D_Box is a clone of Rectangle, and the only thing shared is getArea

Chapter 4

Prototype-based languages

This part of the report will mostly cover some of the prototype-based languages currently being used. It will just be a brief introduction to each. There will also be a short presentation of the various mechanisms that are individual for each language. There are plenty of prototype-based languages, but few have made their way to the general public. A language many people use, perhaps without realizing that it is partly prototype-based, is JavaScript.

4.1 Self

Self is an object oriented programming language. It is one of the most widely supported prototype-based languages. Self is similar to Smalltalk in syntax and semantics, but where Smalltalk uses the class based paradigm, Self uses prototypes. According to its creator, David Ungar [Ung87], it was designed by stripping Smalltalk of classes and variables, and seeing whether it was still programmable or not. [SLSTU] [ST]

4.1.1 Mechanisms

Objects

In the Self language, objects are comprised of slots. A slot has a name, and a reference to an object. The slots of an object contains its behaviour and state. Interaction with an object is done with message sending. The name of the message, called *selector*, is matched against the name of the slot. How the slot responds depends on what kind it is. There are two kinds of slots, method slots, and data slots. The method slot contains a method (i.e executable code). When a message is received, the result from executing the method is returned. Data slots are slots containing any kind of object that is not executable, much like class variables in class based programming. When a message is received to a data slot it simply returns the contents of its object.

New objects are created by simply copying existing ones, and then adding slots to get the desired behaviour.

Inheritance

Inheritance in Self is done by making data slots into parent slots. Parent slots *delegates* messages to the objects they are referring to if the selector of a message is not matched to any slot in the object. This means that a method only has to be written once. When contained in a special object, called *trait*, that method can be used by any other object.

4.2 JavaScript

JavaScript is a Object Oriented Language mostly used on websites. It is not a full programming language, but it has some interesting features, such as object based inheritance (i.e prototype-based). Many people are using this language, often without realizing exactly what they are using. [JOO]

4.2.1 Mechanisms

Objects

An object in JavaScript is really similar to objects in class-based languages. They are comprised of *properties*, which basically consists of a data type, a method, or another object. Objects are created through *constructor functions*, which creates an object and sets the necessary properties.

A small example:

```
function Book(title){
  this.title = title;
  this.currentPage = 0;
  this.nrOfPages = 300;
  this.back = "hardback";
  this.turnPage = function (){
    this.currentPage += 1;
  }
}
```

This object describes a book with title, current page, number of pages and type of back. its page turning function increases the current page property.

Inheritance

In JavaScript there are two ways to give new objects similar functionality as an existing one. One is to rewrite the properties every time, which can be tedious and unnecessary. The other way is to use prototypes. The idea is to reduce the code by reusing the properties of similar objects, and then adding new, or changing existing properties. This can be compared to subclassing.

The following example expands on the Objects example:

```
function pocketBook(title){
this.title = title;
this.nrOfpages = 450;
this.back = "paperback";
}
pocketBook.prototype = new Book;

var book = new Book(Robinson Crusoe);
var pocket = new pocketBook(Treasure Island);
```

Here the prototype approach is used to create a new type of book, the pocketBook. The advantage is that the currentPage, and turnPage properties do not need to be defined again. The prototype function acts even before the constructor function. That means that the constructor for Book sets all the properties for pocketBook, and then pocketBook's constructor is run.

4.3 Kevo

Kevo is a small language developed by Antero Taivalsaari [Tai92]. It is based on the idea that a prototype-based programming language does not have to support delegation, like Self. Instead, Kevo uses a mechanism called *concatenation*, which will be described later. [SLSTU]

4.3.1 Mechanisms

Objects

In Kevo, objects are represented as independent units which contains all the properties of a desired effect. This means that, unlike Self, objects does not share methods or data, at least not logically. But, in order to make things easier on programmers, Kevo supports similar objects being grouped together. It also supports groupwise modifications of these objects. This means that the programmer only needs to alter one object, and then the change is propagated to all the other objects in the group.

Inheritance

As mentioned earlier, Kevo does not support delegation. The effect of this is that some other kind of mechanism is needed in order to get some kind of inheritance. Kevo solves this by *concatenation*. Concatenation provides a couple of copying functions, and several modification functions. So, in order to create a new kind of object, the programmer needs to copy an existing object, and then apply the necessary modification operations to get the desired properties. Concatenation operations also support grupwise modification.

4.4 NewtonScript

Being a language almost exlusively used in the Newton PDA's[1], it might be considered less useful. However, it has proved that prototype-based languages can be put to practical use. The two greatest influences for NewtonScript are Self and LISP, a functional programming language. [SLSTU] [NS]

4.4.1 Mechanisms

Objects

The object model i NewtonScript is quite similar to the one in Self. There is a special kind of object, called *frame*, which has named slots. The frame is the only kind of object that can receive messages. Each slot in the frame is a reference to another object. Like Self, when a message is received the name contained within it is matched against the slots of the object.

Inheritance

NewtonScript implements a kind of double inheritance. This scheme is well suited for the main use of NewtonScript, GUI[2] applications. In a frame, there is two special slots, called *_proto*, and *_parent*. _proto contains a reference to the frame's prototype (the object used to make the frame). If a message cannot match its selector with any of the slots, it is forwarded to the _proto object. The _parent reference basically the same. When a function cannot be found in a frame or its prototype, the message is sent to the _parent object. _parent is used for the GUI part of NewtonScript's inheritance scheme, to maintain a relationship between a window and its child-windows.

[1] Personal Digital Assistant
[2] Graphical User Interface

Chapter 5

Examples in JavaScript

Following are prototype-based programming concepts exemplified in JavaScript. Examples are from "The Core JavaScript Guide" [Jav00].

5.1 Creating Objects

There are two ways of creating new objects in JavaScript. One is to use an *Object Initializer*. This is the most direct approach, where the object is created and initialized in one command.

The second way of creating new objects involves making a constructor function. Any JavaScript function can be used as a constructor. A constructor function is made just like any function, and the 'this'-keyword is used when initializing the properties. To create an object with a constructor function, the "new" command is used, very much like in Java or C++.

With Object Initializer

Notice that objects can be created in a nested way, by creating objects as properties in an object.

```
myObject = {property1:value, property2:value2...
myCar = {color:"red",wheels:4,engine:{cylinders:4,size:2.2}}
```

With constructor function

The following code defines a constructor function for a car with three properties; make, model and year. An object, myCar, is created with the constructor.

```
function car(make, model, year) {
   this.make = make;
   this.model = model;
   this.year = year;
}
myCar = new car("Nissan", "300ZX", 1992);
```

5.2 Adding and removing properties to objects

Properties can be added to objects at runtime in a simple fashion. When a property is added using the pre-defined property 'prototype', that property is added to all objects that share the same prototype.

```
car1.stereo="Yamaha CDX200";  // Add a propery to a single object

car.prototype.color=null;  // Add a property to all objects share
                           // the same prototype as car

object.methodname = function_name; // Add a method to a object

delete myobj.a; // Delete a property from myobj
```

5.3 Inheritance

JavaScript has, just as other prototype-based languages, a different approach to inheritance. In this example a WorkerBee-object inherits from an Employee-object. First, constructors are created for the two objects. Then a `new` Employee object is assigned to the prototype of the WorkerBee-object. In this way, when creating a WorkingBee-object, it will also have the properties of an Employee.

```
function Employee () {    // constructor function for Employee
   this.name = "";
   this.dept = "general";
}

function WorkerBee () {   // constructor function for WorkerBee
   this.projects = [];
}
// WorkerBee gets properties of Emloyee.
// The assignment can be placed anywhere in code
WorkerBee.prototype = new Employee;
```

This may look quite similar to the class-based approach, but there is an important differance. In JavaScript, the effect of inheritance is achieved by a simple assignment. This means that you can combine objects at runtime. In class-based programming, this must be declared from the beginning.

Chapter 6

Related works

There is a book called 'Prototype-based programming' [NTM99] in which every chapter has got a unique author. The book collects almost everything within the area, except for some concrete examples. A good thing about there being many different authors is that even if the chapters does not deal with the same issues, you will hear some different views on prototype-based programming. The beginning of the book concerns history of prototypes, and also describes what a prototype is. Later on it goes deeper into some of the biggest languages, Self, NewtonScript etc. A chapter on how to classify prototype-based languages is also included.

The same writers that wrote the classification-chapter in the book have also written an article on the same subject, which is a good complement to the chapter in the book [DMC92]. The article was written before the book.

Steyaert and De Meuter have written an article about hybrid languages, where they propose that both the class-based and the prototype-based communities can learn much from each other [SDM95]. Another article whose purpose is quite similar is 'Object-centered programming', which criticizes the advocates of prototype-based langugages for rejecting classes [Mal].

'Object lessons from Class-Free Programming' is a good introduction to the prototype-based paradigm [SLSTU]. It also features sections from four experts. Mark Lentczner[1] from Glyphic Technology, Walter R. Smith[2] from Apple Computer Inc., Antero Taivalsaari[3] from Nokia Research Center, and David Ungar[4] from Sun Microsystems Laboratories. This report is built on a disussion panel of these four experts and a moderator, where each of the experts presents his language and ideas.

[1] Mark Lentczner has been a principal designer of Glyphic Script
[2] Walter R. Smith is the author of NewtonScript
[3] Antero Taivalsaari is the designer of Kevo
[4] David Ungar co-leads the Self project

Chapter 7

Conclusions and discussion

7.1 Conclusions

The prototype concept is that you should use concrete objects and never abstractions. An object in prototype-based programming can delegate messages to its prototype or some other object. The object can change its delegates during runtime. Furthermore, it can add or delete attributes and methods dynamically.

There are many different prototype-based languages and they use different mechanisms. However, the main concept stays the same.

Prototype-based programming is much more flexible than class-based, but that also means that there is more that can go wrong. The cost in speed and space is a big tradeoff for the flexibility, and how to make the prototype-based languages faster and smaller is still a research area.

Proponents of prototype-based programming argues that the use of abstract classes uses a way of thinking that is unnatural. People thinks in concrete objects and not in abstractions, but if the prototype community does not start to borrow ideas from the class community, they will never grow.

7.2 Discussion

If you are an experienced class-based programmer, it will probably be hard to get used to, and even to like, prototype-based programming. Absorbing new concepts in programming is often time consuming, and people who knows some programming paradigm are less likely to learn a new one that goes against what they already have learned. Class-based programming have got a large group of programmers. Prototype-based programming have got a smaller group, and we believe that it will continue to be like that. Prototype-based languages will most likely never be larger than class-based languages, but it may well be an important complement. If the two paradigms continue to influence each other, maybe someday a hybrid combining the best of two worlds will surface.

Bibliography

[Bor99] Alan Borning: Classes versus Prototypes in Object-Oriented Languages *Chapter 4 in Prototype-Based Programming [NTM99]*

[DMB99] Christophe Dony, Jacques Malenfant, Daniel Bardou: Classifying Prototype-Based Programming Languages *Chapter 2 in Prototype-Based Programming [NTM99]*

[DMC92] Christophe Dony, Jacques Malenfant, Pierre Cointe: Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. *LITP (IBP) Université Paris-VI, 1992*

[Dro] Sophia Drossopoulou: Advanced Issues in Object Oriented Programming: Object Based Languages or Prototype Based Languages *Department of Computing, University of London*
http://www.doc.ic.ac.uk/ scd/Teaching/ObjectBased.pdf

[Evi93] Mikel Evins: Prototype-Based OOLs *1993*
http://www.mactech.com/articles/frameworks/7_6/Prototype-based_OOLs_Evins.html

[Jav00] The Core JavaScript Guide
http://devedge.netscape.com/library/manuals/2000/javascript/1.5/guide

[JOO] JavaScript Object-Oriented Programming
http://www.sitepoint.com/article/470
http://www.sitepoint.com/article/473

[Lak87] G. Lakoff: Women, Fire, And Dangerous Things: What Categories reveal about the mind *University of Chicago Press, 1987*

[Mal] Jaques Malenfant: Object-centered programming *Département d'informatique, Ecole des Mines de Nantes*

[Nob97] James Noble: Call For Participation to 2nd ECOOP Workshop on Prototype-Based Object-Oriented Programming *Jyvaskyla, Finland, 1997*

[NS] The NewtonScript Programming Language
http://www.cc.gatech.edu/ schoedl/projects/NewtonScript/

[NTM99] James Noble, Antero Taivalsaari, Ivan Moore: Prototype-Based Programming *Springer-Verlag Singapore Pte. Ltd. 1999*

[SDM95] Patrick Steyaert, Wolfgang De Meuter: A Marriage of Class- and Object-Based Inheritance Without Unwanted Children. *Programming Technology Lab, Vrije Universiteit, Brussel, 1995*

[SLSTU] Randall B. Smith, Mark Lentczner, Walter R. Smith, Antero Taivalsaari, David Ungar: Prototype-Based Languages: Object Lessons from Class-Free Programming

[SLU89] L.A. Stein, H. Lieberman, D. Ungar: A Shared View of Sharing: The Treaty of Orlando. *Object-Oriented Concepts, Databases, and Applications, W. Kim and F.H. Lochovsky (Eds.), pp. 31-48, ACM Press, 1989*

[ST] Self tutorial
http://research.sun.com/research/self/release/Self-4.0/Tutorial/index.html

[Tai92] A. Taivalsaari: Kevo - a prototype based object-oriented language based on concatenation and module operations. *University of Victoria, Victoria, B.C, Canada, June 1992*

[Tai99] Antero Taivalsaari: Classes vs. Prototypes *Chapter 1 in Prototype-Based Programming [NTM99]*

[Ung87] D. Ungar, R. B Smith: Self: The power of simplicity. *Orlando, Florida 1987*