Attack of the Clones

James Noble **Brian Foote**

Computer Science
Victoria University of Wellington
New Zealand
kjx@mcs.vuw.ac.nz

209 W. Iowa
Urbana, IL 61801
U.S.A.
foote@laputan.org

Abstract

Self is a prototype-based programming language, often described as More Smalltalk Than Smalltalk. This paper presents implementations of a selection of the Design Patterns in Self, and investigates the differences between their implementations in Self compared with other object-oriented programming languages.

19. A language that doesn't affect
the way you think about programming,
is not worth knowing.
Alan Perlis, *Epigrams on Programming,* (Perlis 1982)

Introduction

…unfortunately, Self was not the last object-oriented programming language. Self does have some claim to be the last *interesting* object-oriented programming language, or at least the last interesting *modern* OO language — with most languages coming afterwards, C++, Java, C♯, C♭, being as distinguisable as Coke from Pepsi or Malboro from Rothmans: you don't drink the drink, you drink the advertising (Twitchell 2000). In Self, at least you can hear yourself scream.

Considered as a programming language, Self is quintessentially late-modern, in fact it is the minimalist late-modern analogue of Smalltalk, in much the same way that Scheme and Common Lisp are the minimal and maximal late-modern derivatives of the classic Lisp systems (Smith & Ungar 1997, Ungar & Smith 1991, Goldberg & Robson 1983, Moon, Stallman & Weinreb 1984). Self's implementation and programming environments are another matter: completely crazy compiler technology and a hyperreal user interface mark the Self system out as rather more modern than Smalltalk, Lisp, or almost anything else around.

This paper attempts to give some flavour of the language by presenting implemenations for a selection of the *Design Patterns* (Gamma, Helm, Johnson & Vlissides 1994) in Self. We concentrate on "implementations" (technically, solutions (Noble & Biddle 2002)) rather than the pattern's intent, simply because we are showing how particular preëxisting patterns can be implemented in Self. Generally, we've chosen patterns we like, we find interesting, or that have particularly "clever" Self implementations. We make no claims about to the actual use of these patterns (or implementations of patterns) by practicing Self programmers: this paper is more a hypothetical exploration of possible designs rather than a serious description of the style in which programmers actually write.

Form and Content

The bulk of this paper presents the pattern solutions we have implemented in Self. These patterns are organised in the same way and presented in the same order as the corresponding patterns in *Design Patterns*. We begin with the creational patterns Prototype and Singleton, then present the structural patterns Adaptor, Bridge, Decorator, and Proxy, and conclude with the Chain of Responsibility, State, and Strategy behavioural patterns. This paper contains a pattern catalogue (or at least a pattern solution catelogue), rather than a system or language of patterns: really, it is a translating dictionary, showing how the common vocabulary *Design Patterns* provides can be rendered into an uncommon language.

For space reasons, we have used a compressed form to present the patterns. Each pattern has a name, a statement of intent, and a problem statement, derived from the corresponding sections of *Design Patterns*. Aspects of the *solution*'s structure, participants, collaborations, and implementation then follow, based around a source code example. There are no discussions of known uses. For readers who are not yet part of the worldwide community of Self programmers (there must be at least ten of us still left) an introduction to Self has been added as an Appendix.

Acknowledgements

Many thanks to the members of the KoalaPLoP writers' workshop — Pronab Ganguly, Pradip Sarkar, Marie-Christine Fauvet — and, of course, to the Vast Self Conspiracy.

Envoi

We believe that many of the Design Patterns capture deep general properties of object-oriented software design. This paper is an experiment to determine whether (and how well) the patterns appear in Dave and Randy's excellent adventure: the slightly odd, wonderfully consise, pretty-damn always object-oriented programming language called Self.

Prototype

Intent

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Problem

You often need to be able to create the right kind of object for a particular task. How can you create new objects of the correct type?

Example

The following code presents a stand-alone point object that can be used as a prototype. The clone method is implemented in terms of a low-level primitive Clone operation. This is all that is needed to implement such an object; you can add messages (like equality shown here) as necessary.

```
    point = (|
        x.
        y.
        = aPoint = ((x = aPoint x) && (y = aPoint y)).
        clone = (_Clone).
|)
```

Implementation

In practice in Self, you wouldn't actually implement an object this way. Rather, you'd move the methods into a separate object (so they can be inherited rather than copied every time), and use inheritance from the system object traits clonable. The traits clonable object provides the exactly this definition of clone, plus a bunch of standard messages for printing, comparing, and so on.

```
point = (|
        parent* = traits point.
        x.
        y.
|)


traits point = (|
         parent* = traits clonable.
         = aPoint = ((x = aPoint x) && (y = aPoint y)).
|)


traits clonable = (|
        "lots of methods deleted"
         clone = (_Clone).
|)
```

Discussion

This pattern is built into Self — indeed, it is the only way a Self program can create new objects. Self is called a prototype-based language for just this reason. In Self, new objects are always created using the clone message to copy an existing object, and this message is eventually implemented by the system level primitive Clone.

To make a prototype of an object accessible to programs in general, you store one object of that type into a global namespace object, where programs can retrieve it and clone it as necessary. In Self, these global namespaces are just standard objects that are inherited by any object which wishes to access the objects defined therein: the prototype is found by inheritance whenever it's name is used.

Singleton

Object Creational

Intent

Ensure a class only has one instance, and provide a global point of access to it.

Problem

Some classes should always have exactly one instance. For example, objects representing real hardware resources, like disks, printers, or networks, or special system objects like trash cans, should only appear once.

Intent

Solution

Self objects are only ever created by the clone method. So you can make any object a singleton by redefining clone to do nothing, that is, to return the object itself.

Example

Redefining clone is simple in Self. For example, you could make a special point object to represent the origin: Sending clone to this object will return itself.

```
origin = (|
        clone = (self).
        x = 0.
        y = 0.
|)
```

Implementation

Self provides a traits oddballs object (similar to traits clonable) from which you can inherit to get a singleton. Also, traits oddballs has a lot of other methods for comparison, equality, and so on that do the right thing for a singleton.

```
origin = (|
        parent∗ = traits oddballs
        x = 0.
        y = 0.
|)


traits oddballs = (|
       "lots of methods deleted"
        clone = (self).
|)
```

Discussion

Design Patterns describes how Singleton can be generalised to control the number of objects created: you can do just the same in Self, quite easily. For example, to limit a given prototype to having only six examples, just maintain a cloneCount variable. This needs to be accessible to every clone, so the best place to put it will be in a traits object that the prototype inherits.

```
sillyPoint = (|
        parent∗ = traits sillyPoint.
        x ← 0.
        y ← 0.
|)

traits sillyPoint = (|
        parent∗ = traits clonable.   "we can clone these!"
        cloneCount ← 0.
        clone = ((cloneCount < 6)
                    ifTrue: [cloneCount: cloneCount succ.
                          resend.clone]
                    False: [error: 'Attack of the Clones!'])
|)
```

Adaptor

Intent

Convert the interface of a class into another interface users expect. Adaptor lets classes work together that couldn't otherwise because of incompatible interfaces.

Problem

Ignoring the class-based language of the pattern's intent statement, objects can be are worlds unto themselves, each with their own culture, language, and style. Systems often need to bridge several worlds, where objects' interfaces (the messages they understand) can be very different.

Solution

Make an adaptor object which converts the *Adaptee* object's interface to match the *target* interface.

Example

For example, you could adapt the point object (from the Singleton pattern) to provide polar point coordinates something like this:

```
polarAdaptor = (|
        adapteePoint.
        theta = ((adapteePoint y / adapteePoint x) arctan).
                "needs to be corrected. we don't do maths."
        r = ((adapteePoint x squared + adapteePoint y squared) sqrt).
|)
```

Implementation

This adaptor could be split into a prototype and a traits object in the usual way. You probably need to take care that messages such as clone do the right thing (e.g. cloning the adaptee and returning a new adaptor around that object).

Discussion

There are a couple of alternative options that do more things with inheritance. For example, we can inherit from the adaptee:

```
polarAdaptor = (|
        adapteePoint*. "inherit from adaptee"
        theta = ((y / x) arctan).
        r = ((x squared + y squared) sqrt).
|)
```

giving a design similar to a class adaptor, but with single inheritance from the Adaptee. Because Self (like Smalltalk) is dynamically typed, there's no need for the adapter to inherit from some target interface (Alpert, Brown & Woolf 1988).

The *Smalltalk Companion* also lists various other twisted kinds of parameterisable adaptors useful in Smalltalk, with the key advantage begin that they allow flexible adaption without havign to create new Smalltalk classes (Alpert et al. 1988). Since objects are self-defining in Self, we don't have to worry about lots of single-use classes: single-use objects are just fine.

Bridge

Intent

Decouple an abstraction from its implementation so that the two can vary independently.

Problem

Sometimes you need a library of classes that provide a set of abstractions with a set of different implementations. For example, a collection library could provide a number of different kinds of collection abstractions (like sets, bags, dictionaries) that can each have a number of different implementations (in arrays, linked lists, hash tables).

Solution

Make two hierarchies of objects — one for the abstraction and one the implementation. The abstraction objects delegate some or all of the behaviour to one implemenation object.

Example

In Self, the abstraction object can inherit from their implementations, using a variable parent slot so that the implementation can be changed, if necessary. Considering the collections example, we could have some abstractions:

```
bagAbstraction =(|
        myImplementation∗. "variable parent slot"
        add: element = (append: element). "inherit append"
|)

setAbstraction = (|
        myImplementation∗. "variable parent slot"
        add: element = ((contains: element) ifFalse: [append: element]).
                "inherit both contains and append"
|)
```

and them the underlying implementations that use them:

```
listImplementation = (|
        myList.
        append: element = (myList insertAtEnd: element).
        contains: element = ( ... ).
        ...
|)

vectorImplementation = (|
       myVector. myIndex.
       append: element = (vector at: myIndex Put: element.
                      myIndex: myIndex succ).
       contains: element = ( ... ).
|)
```

The abstractions provide definitions of user-level methods like `add`, which is written in terms of implementation-level methods (like `append` and `contains`) which are inherited via the `myImplement ation*` variable parent slot.

Implementation

A serious implementation would factor both abstraction and implementation objects into separate hierarchies, using multiple inheritance so that an abstraction object inherits both from its abstraction traits and also its implementation object. This is weirdly reminiscient of the C++ multiply inheriting adapter described in *Design Patterns*, but, of course, done with objects rather than classes.

Discussion

This is a very nice example of the use of dynamic inhertiance in Self: should an abstraction need to change its implementation it can do so by cloning a new implementation, initialising it from the old one, and then replacing it (in much less code than it takes to write in English).

```
setAbstraction = (|
       useListImplementation = (
           myImplementation: listImplementation clone
                   initialiseFrom: myImplementation).
|)
```

Decorator

Object Structural

Intent

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Problem

Some objects are boring and irresponsible, painted in just the wrong shade of beige, but otherwise OK.

Solution

Use a decorator to redecorate your objects! Make a special object that supplies just the fresh new behaviour you need, and wrap it around the dowdy old object.

Example

So we have our tired room object:

```
room = (|
    colour = 'brown'.
    carpet = 'beige'.
    printString = ('A ', colour, ' room with ', carpet, ' carpet.').
|)
```

we can spruce it up with a decorator

```
puceColourDecorator = (|
    parent∗ ← room.
    colour = 'puce'.
|)
```

or two

```
lighteningCarpetDecorator = (|
    parent∗ ← puceCarpetDecorator.
    carpet = ('light ', resend.carpet).
|)
```

so the wall colour is now puce and the carpet light beige. A great improvement.

Implementation

Self will usually “do the right thing” regarding the “self problem” (Lieberman 1986); so for example calling printString on the fully decorated room would print *“A puce room with light beige carpet.”*. This will fail with primitives which bind to object identity, i.e. you will be able to tell the plain room and the decorated room apart. You can confuse the issue more by redefining the identity comparison message inside the decorators:

```
== x = (parent == x). “compare with my parent, not with me”
```

(but if you managed to parse that piece of code then you already understand everything going on in this paper).

If you had lots of decorators with very simple code, you could refactor them into an inheritance hierarchy with separate traits objects as we described for Adaptor. But, like many American presidents, Decorators are defined by what they are *not* — due to inheritance, they only need to describe their own responsibilities — so it is usually not worth the effort.

Discussion

Another lovely, natural example of dynamic inheritance in Self. You could do something almost as nice Smalltalk using doesNotUnderstand — but the *Smalltalk Companion* doesn’t talk about it because, well, they’re wimps (Alpert et al. 1988). You can also do exactly the same thing in Smalltalk using Rappers (Brant, Foote, Johnson & Roberts 1998).

Proxy

Object Structural

Intent

Provide a surrogate or placeholder for another object to provide access to it.

Problem

Some objects are never there when you want them — perhaps they're on disk, perhaps they're on another machine... what a hassle!

Solution

Make a proxy object to stand in for remote or expensive objects. Put the proxy where you'd like to put the original object, but can't, and make the proxy behave as if it were the original object.

Example

Self proxies (for local objects) can be done in just the same way as decorators:

```
doNothingProxy = (|
        parent∗ ← realSubject.
|)
```

It's more fun to implement proxies that do something, like a virtual proxy which loads an object from disk when it receives a message:

```
virtualProxy = (|
        parent∗ ← (|
             "placeholder"
             loadObjectFrom: filename = ( ... ).
             someMessage = (
                (parent: loadObjectFrom: myFilename).
                resend.someMessage).
        |)
        myFilename ← 'object.self'.
|)
```

in this case we've used a placeholder parent object (kind of like a specialised Null Object (Woolf 1998)) that arranges to load the object when it receives a message — Self's inheritance rules will sort out the various names correctly, so that parent replaces itself with the results of the loadObjectFrom message. Alternatively you could use the undefinedSelector trap to catch all the messages in one place, without having to define them:

```
virtualProxy = (|
        parent∗ ← (| |)  "empty object, not even nil"
        myFilename = 'object.self'.
        undefinedSelector: msg = (
           parent: loadObjectFrom: myFilename.
           parent perform: msg).
|)
```

(note that a serious implementation needs to handle message arguments — this example abbreviates the undefinedSelectord message).

Implementation

The inheriting implementation is fine when proxying local objects. For remote objects, you may have to do something more difficult, like trapping the `undefinedSelector` exception (like Smalltalk's `doesNotUnderstand`) because the object you need to delegate to isn't their yet. Generally, there are three strategies you can use to implement these kinds of things: manually delegating messages, using (dynamic) inheritance, and trapping exceptions — from the most sensible to the most hairy.

Discussion

How hardcore do you want to be?
How many lifetimes to you want to waste?

Chain of Responsibility

Intent

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Problem

Sometimes one object doesn't know enough to be able to respond to a message correctly. Perhaps responsibility is shared between several cooperating objects, or perhaps the objects are organised into a structure, and the structure dictates how the message should be handled, rather than any individual object. For example, a request for context sensitive help in a graphical user interface may not be able to be answered by any single object, rather it may need to be passed up from a button to a panel to the containing window.

Solution

Incorporate an extra inheritance link which models the chain of responsibly. Then, ensure some requests are passed along that link, rather than being handled normally.

Example

We can arrange a separate inheritance chain along the container link that points from a window to its structural container.

```
buttonWindow = (|
        parent∗ = traits buttonWindow. "normal inheritance"
        container∗ ← panel. "structural containment"
        windowTitle ← "Push Me".
|)

traits buttonWindow = (|
        display = ( ... ).
        "don't handle help message here"
|)

panel = (|
        parent∗ = traits panel. "normal inheritance"
        container∗ ← topLevelWindow. "structural containment"
        help = "Cutesy help message".
        display = ( ... ).
|)
```

If the buttonWindow gets a help message, it will be handled by its structurally containing panel.

Implementation

This design looks very nice in theory, but unfortunately will have problems in practice. The problem is that there can be many messages that will be ambiguous between the structural chain and normal taxonomic inheritance — sending display to the buttonWindow will raise an error because Self can't decide between the display method in panel or in traits buttonWindow.

To make this work at all, you have to choose which messages can be found on which links, and then somehow assure there are no ambiguous definitions, removing or renaming methods from objects as necessary. If a message called help is supposed to be answered via the structural container link, then ensure that help is never implemented via the taxonomic parent link: but perhaps make the structural root object resend help renamed as defaultHelp — where defaultHelp can be found on the taxonomic chain.

Discussion

Wow! Something that actually uses dynamic multiple inheritance. Pity is doesn't work very well: ambiguity errors will kill it in practice. The old version, Self 2.0, with priorities inheritance did this just fine (just make the structural parent a lower priority than the inheritance parent), till Randy made Dave wimp out and remove priorities (Smith & Ungar 1997). I guess hardcore Self programmers would consider using delegated performs on undefined selector exceptions (or, even better, on ambiguous selector exceptions) to do the right thing.

The only other languages that can do this kind of thing have been build specially to do it (like NewtonScript (Smith 1995) or Dewan's bidirectional inheritance scheme (Dewan 1991)). Self does this for free.

Observer

Intent

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Problem

This pattern raises two problems: how can you detect when an object has changed, and how to inform other objects interested in that change. Self provides no special support for this second problem, but does make it easier to determine when an object has changed.

Solution

Rename an object's data slot and replace it with accessor methods that access the renamed slot. Make the writer method automatically signal the observer.

Example

Here's our point object hacked so that it will signal a change whenever its slots are assigned to.

```
    subjectPoint = (|
        privateX.
        privateY.
        x = (privateX).
        y = (privateY).
        x: newX = (signalChangeOf: 'x' to: newX. privateX: newX).
        x: newY = (signalChangeOf: 'y' to: newY. privateY: newY).
|)
```

of course it should to implement the "signalChangeOf:to:" message to actually handle the change.

Implementation

The key here is that a slot called foo is only read by the accessor message foo and written by the setter message foo:. An external object can't tell the difference between something implemented as a pair of getter and setter methods implementing those messages, or by a single data slot. The two methods (say foo and foo:) will appear to all clients to be exactly the same as a single variable called foo.

Discussion

It's interesting to think that C♯ introduces a whole lot of crud into the language (properties) to handle just this issue — whereas the clean, minimal design of Self gets it all for free.

State

Intent

Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.

Problem

You have an object whose behaviour depends upon its internal state in a complex way, with lots of conditional code.

Solution

The State pattern puts each branch of the conditional in a separate object. This lets you treat the object's state as an object in its own right that can vary independently from other objects.

Example

Some of Self's collection objects actually do this to provide different behaviour when the are empty to when they are nonempty.

```
list = (|
        parent∗ ← traits emptyList.
        ... “slots go here!”
|)

traits emptyList = (|
        parent∗ traits list.
        add: e = (parent: traits fullList.  ... “do whatever”)
        ...
|)


traits fullList = (|
        parent∗ = traits list.
        removeAll = (“do whatever”. parent: traits emptyList)
        ...
|)
```

The list object inherits from one of two state objects depending on whether it is empty or not: both of these then inherit from traits fullList which contains all the state independent behaviour.

Implementation

Here the context object (the list has only one state object, so we make the sole parent slot dynamic. We could alternatively have a standard static parent slot and use a dynamically inherited state object (as shown in the strategy pattern example).

Discussion

The solutions of state and strategy are pretty much the same (except that state objects are usually encapsulated in their contexts, and contexts can change them automatically, while strategies may be visible to clients and are not usually changed by their contexts). The difference between the patterns is primarily one of intent, not solution structure (Noble & Biddle 2002, Alpert et al. 1988).

Strategy

Intent

Define a family of algorithms, and make them interchangeable.

Problem

You have an object that needs to use several alternative algorithms (and perhaps switch between them) each with its own interface and customisable parameters.

Solution

Make a separate strategy object to represent each algorithm, and make the algorithm's parameters into the attributes of the strategy object. Make the context object which uses the algorithms inherit from the appropriate strategy object.

Example

Consider a graphical window that needs to switch between different rendering algorithms. We can make each algorithm a separate rendering strategy object and swap the objects whenever we need to change algorithms. The Self twist is that we can use multiple dynamic inheritance to link the context to the strategy objects.

```
contextWindow = (|
        parent* = traits contextWindow.  "standard inheritance hierarchy"
        open = ( ... ).
        close =( ... ).
        x. y. width. height.
        renderingStrategy* ← renderingStrategy. "inherit from strategy"
|)

traits contextWindow = (|
        display = (render).
|)

renderingStrategy = (|
        render = (drawRectangleX: x Y: y Width: w: Height: h. ...)
|)
```

Asking the contextWindow to display itself will invoke the renderingStrategy's render method via inheritance. We can change the strategies whenever we need to by assigning a new object to the renderingStrategy slot.

Implementation

Because of the way Self's inheritance rules work (very similar to Smalltalk or Java, but just between objects rather than classes) methods on the strategy can access all the slots and methods in the context by sending itself messages (the context behaves as a subclass of the strategy, so messages the strategy sends can be overridden by the context — here by the context's slots). Inheritance effectively incorporates all the standard ways of passing information between contexts and components (Noble 2002).

Discussion

Another example combining dynamic inheritance and multiple inheritance. Because the interface of the state objects is so limited, you can usually get away with this (even when a single object has multiple orthogonal strategies) without running into the ambiguity problems you could get when using multiple inheritance for Chains of Responsibilities.

Self

This section introduces the SELF programming language (The Self Group 1992, Ungar & Smith 1991) and is intended to provide enough background to allow a reader to understand the SELF examples presented in the rest of this paper. It begins with a description of objects and expressions, describes the organisation of a SELF program, and then presents an example of an object-oriented stack.

Objects

SELF objects are collections of named slots, and are written as lists of slots enclosed by "(|" and "|)". Each slot is associated with a message selector and may hold either a constant, a variable, or a method. For example, the example below defines an object named "fred" which defines the constant slot "pi", the one-argument keyword message "circ:" and the data (or variable) slot "size".

```
fred = (|
        pi = 3.14159265.        "constant"
        circ: r = (2 ∗ pi ∗ r).  "method"
        size ← 3.               "variable"
     |)

"comments are enclosed in double quotes"
```

Object Creation

Within an executing program, new objects are created as *clones* (slot-by-slot copies) of other objects. Each object has a unique *identity*. Two objects may have the same slots, and the same values contained within those slots, but if the objects were created separately, they can be distinguished on the basis of their identities. In particular, if the value of an object's variable slot is changed, no other objects will be affected (Baker 1993).

Objects intended to be used as patterns for cloning are known as *prototypes*. Like Lisp or Smalltalk, a garbage collector is used so that objects do not have to be disposed of explicitly.

Some types of objects can be created using simple literals. These include numbers, strings, and blocks.

Expressions

SELF's expression syntax is directly derived from SMALLTALK. Since SELF is a pure object-oriented language, almost all expressions are either literals or describe message sends to a particular *receiver* object. There are three types of messages: unary, binary, and keyword; a message's type depends upon its selector and the number of arguments it takes.

Unary messages simply name an operation and provide no arguments other than the message receiver object. For example, "top", "isEmpty" and "isFull" are unary messages.

Binary messages provide one argument as well as the receiver. Their selectors must be composed of nonalphabetic characters. "+", "−" and "∗" are binary messages for addition, subtraction, and multiplication. Similarly, "@" creates a point from two numbers, and "##" creates a rectangle from two points.

Keyword messages are the oddest part of SMALLTALK and SELF syntax. They provide messages with one or more arguments. A keyword message has a particular arity, and the message selector is divided into that many parts. A keyword message is written as a sequence of keywords with argument values interspersed. For example, "at:Put:" is a two-argument keyword message for array assignment. The PASCAL code

```
a[i] := j;
```

is written in SELF as

```
a at: i Put: j.
```

Messages can be sent directly to the results returned by other messages. Parentheses can be used for grouping. For example:

```
draw = (style drawLineFrom: start negated To: finish negated).
c = ( (a squared + b squared) squareRoot ).
```

Implicit Self

The only expression that is not a literal object or message send is the keyword "self". This denotes the current object, that is, the receiver of the currently executing method. A message sent to "self" is known as a *self-send*. Because "self" is used pervasively in SELF (especially as messages are used to access variables, as described below), it is elided from the syntax wherever possible. The language is named "SELF" in honour of this omnipresent but mostly invisible keyword.

Accessing Variables

A variable or constant slot is read by sending a unary message corresponding to its name, and a variable slot is written by a one-argument keyword message, again corresponding to the variable slot's name. The SELF code

```
foo: 43.
bar: foo.
```

is roughly equivalent to the PASCAL

```
foo := 43;
bar := foo;
```

if "foo" and "bar" are variables, but equivalent to

```
foo(43);
bar(foo);
```

if "foo" and "bar" are methods.

The use of messages to access variables is one of the main differences between SELF and SMALLTALK.

Blocks

Blocks represent lambda expressions. For example, the expression $\lambda xy.x + y$ when written in SELF is:

```
[| :x. :y. | x + y].
```

A block optionally may have arguments and temporary variables: these are written at the start of the block surrounded by "|" symbols. Arguments are prefixed by colons: temporary variables are not. Blocks are used to implement control structures, in concert with keyword messages. For example:

```
n isEven ifTrue: ['n is even!' printLine]
      False: ['n is odd!' printLine].
```

There is nothing special about the "ifTrue:False:" keyword message: unlike LISP it is not a special form or macro. Its arguments must be enclosed in blocks to avoid premature evaluation: "ifTrue:False:" will evaluate the appropriate argument block.

Iterators

Blocks can be used as mapping functions and iterators. For example, collections provide a "do:" message which applies a one-argument block to each element of the collection in turn. The total of the items in a collection can be computed by passing "do:" a block which accumulates each element:

```
total: 0.
collection do: [| :item. | total: total + item].
```

Returns

The "ˆ" prefix operator is used to return prematurely from a method. Its semantics are essentially the same as the C `return` statement. A linear search of a collection for the string "`'foo'`" can be performed by combining an iterator and a return operator.

```
collection do: [| :item. | item = 'foo' ifTrue: [ˆtrue]].
ˆfalse.
```

Inheritance

Since SELF does not have classes, objects can inherit directly from other objects by using parent slots. A parent slot's declaration is suffixed by an asterisk "*". When a message is sent to an object, a *message lookup* algorithm is used to find a method to execute or variable to access. SELF's message lookup algorithm searches the message's receiver, then recursively searches any of the receiver's parents. If an implementation of the message cannot be found, an *undefined selector* exception is raised.

```
foo = (|
        fred = ('Implemented in foo' printLine).
    |).

bar = (|
        parent* = foo.
        nigel = ('Implemented in bar' printLine).
    |).
```

For example, in the two objects above, sending "fred" or "nigel" messages to the "bar" object will execute successfully, but sending "nigel" to "foo" or "thomas" to either will result in an undefined selector exception.

Traits and Prototypes

Inheritance is often used to divide objects into two parts — a *prototype* and a *trait*. Typically the trait object contains method slots shared by all clones of the prototype, while the prototype contains the "per-instance" variables of each clone and a parent slot referring to the trait object. Prototypes roughly correspond to SMALLTALK's instances, and traits to SMALLTALK's classes (Ungar, Chambers, Chang & Hölzle 1991, Chambers, Ungar, Chang & Hölzle 1991).

Multiple and Dynamic Inheritance

An object may contain more than one parent slot to provide multiple inheritance: if the method lookup algorithm finds more than one matching method, an *ambiguous lookup* exception is signalled. Parent slots may be variable slots as well as constant slots: this provides dynamic inheritance, which allows the inheritance hierarchy to change at runtime. For example, a binary tree node can be implemented using one variable parent slot but two alternative trait objects. One parent is used by empty tree nodes, and the other by tree nodes containing values. A node is created empty, and uses the empty node trait object. When a node receives a value, it alters its parent slot so that it inherits from the non-empty node trait object.

Resends

A method can use the resend operator (prefixed to a message selector) to call an inherited version of the method. In the example below, both "`Implemented in bar`" and "`Implemented in foo`" will be printed if "fred" is sent to "bar".

```
foo = (|
        fred = ('Implemented in foo' printLine).
    |).

bar = (|
        parent∗ = foo.
        fred = ('Implemented in bar' printLine.
                resend.fred).
    |).
```

The SELF Library

SELF includes a library containing over two hundred prototype objects. These are divided into various categories:

Control Structure Objects such as blocks, true and false provide messages like "ifTrue:False:" which implement basic control structures.

Numbers SELF includes both integers and floating point numbers.

Collections The largest category of SELF objects, collections are containers that hold other objects. SELF's containers include vectors and byteVectors which are fixed size arrays; sequences and orderedCollections which are like arrays but can grow or shrink to accommodate a variable number of arguments; strings which are special collections of characters; sets and dictionarys which are implemented either as hash tables or trees; doubly-linked lists; and sharedQueues which can be used to synchronise multiple processes.

Geometry Objects such as points, extents and rectangles provide basic two-dimensional geometry.

Mirrors SELF is structurally reflexive. This is provided by mirror objects, which reflect upon other objects. Each mirror is associated with one other object, the mirror's *reflectee*. Mirrors understand messages such as names, localDataSlots, and localAssignmentSlots, which respectively return the names of all slots, all data slots, and all assignment slots in the mirror's *reflectee*.

Foreign Proxies Various proxy objects provide access to functions and objects written in C or C++. Tarraingím uses proxies to provide graphical output using the **X** window system.

Example SELF Programs

Figures 1 and 2 contain a SELF version of a stack.

Figure 1 contains the definition of the stack object. This is split into two objects, traits stack containing method declarations, and the stack prototype, which inherits from traits stack. The push and pop methods are publicly exported from traits stack, while in the stack prototype, all the slots are intended to be private. Because the prototype inherits from the traits object, the methods defined in traits stack are able to access the data slots defined in the prototype. The traits object similarly inherits extra behaviour from traits collection.

The traits stack object provides a definition of clone to create new stacks. This uses the resend operation to call the clone operation defined in traits collection, and then clones the contents vector, which should not be shared between different stacks. Figure 2 illustrates a SELF version of the stack main program. Blocks are used widely to implement the looping control structures.

Figure 3 shows a SELF implementation of Quicksort.

As you can see from this example, SELF syntax is indeed a thing of beauty, but…

```
"definition of stack"
traits stack = (|
    parent∗ = traits collection.

    push: c = (contents at: index Put: c. index: index + 1).
    pop = (index: index − 1. contents at: index).
    isEmpty = (index = 0).

    clone = (resend.clone contents: contents clone).
|)

stack = (|
    parent∗ = traits stack.

    contents ← vector copySize: 80.
    index ← 0.
|)
```

Figure 1: SELF Definition of a Stack Object

```
main = (|
      lines ← 0.
      s ← stack.

      handleLine = (
          [eoln] whileFalse: [s push: read].
          [s isEmpty] whileFalse: [s pop write].
          lines: lines + 1).

      initialise = (s: stack clone).

      reverse = (
          initialise.
          [eof] whileFalse: [handleLine].
          ('Reversed: ',lines,' lines\n') printLine).
|)
```

Figure 2: SELF Program using a Stack Object

```
quick = (|
      quickSort =  (quickSortFrom: 0 To: size predecessor).
      quickSortFrom: l To: r = ( | i. j. x. |
          i: l.
          j: r.
          x: at: (l + r) / 2.
            [
              [(at: i) < x] whileTrue: [i: i successor].
              [x < (at: j)] whileTrue: [j: j predecessor].
              i ≤ j ifTrue: [
                i = j ifFalse: [swap: i And: j].
                i: i successor.
                j: j predecessor.
              ].
            ] untilFalse: [ i ≤ j ].

          l < j ifTrue: [quickSortFrom: l To: j].
          i < r ifTrue: [quickSortFrom: i To: r].
          self).
|)
```

Figure 3: Quicksort in SELF

References

Alpert, S. R., Brown, K. & Woolf, B. (1988), *The Design Patterns Smalltalk Companion*, Addison-Wesley.

Baker, H. G. (1993), 'Equal rights for functional objects or, the more things change, the more they are the same', *OOPS Messenger* **4**(4).

Brant, J., Foote, B., Johnson, R. E. & Roberts, D. (1998), Wrappers to the rescue, *in* 'ECOOP Proceedings', pp. 396–417.

Chambers, C., Ungar, D., Chang, B.-W. & Hölzle, U. (1991), 'Parents are shared parts of objects: inheritance and encapsulation in Self', *Lisp And Symbolic Computation* **4**(3).

Dewan, P. (1991), 'An inheritance model for supporting flexible displays of data structures', *Software—Practice and Experience* **21**(7), 719–738.

Gamma, E., Helm, R., Johnson, R. E. & Vlissides, J. (1994), *Design Patterns*, Addison-Wesley.

Goldberg, A. & Robson, D. (1983), *Smalltalk-80: The Language and its Implementation*, Addison-Wesley.

Lieberman, H. (1986), Using prototypical objects to implement shared behavior in object oriented systems, *in* 'OOPSLA Proceedings'.

Moon, D., Stallman, R. & Weinreb, D. (1984), *The Lisp Machine Manual*, M.I.T. A I Laboratory.

Noble, J. (2002), Need to know: Patterns for coupling contexts and components, *in* 'Proceedings of the Second Annual Asian Pacific Conference on Pattern Languages of Programs', pp. 196–206.

Noble, J. & Biddle, R. (2002), Patterns as signs, *in* 'ECOOP Proceedings'.

Perlis, A. (1982), 'Epigrams on programming', *ACM SIGPLAN Notices* **17**(9).

Smith, R. B. & Ungar, D. (1997), Programming as an experience: The inspiration for Self, *in* J. Noble, A. Taivalsaari & I. Moore, eds, 'Prototype-Based Programming: Conecepts, Languages, Applications', Springer-Verlag.

Smith, W. R. (1995), Using a prototype-based language for user interface: The Newton project's experience, *in* 'OOPSLA Proceedings'.

The Self Group (1992), *Self Programmer's Reference Manual*, 2.0alpha edn, Sun Microsystems and Stanford University.

Twitchell, J. B. (2000), *twenty ADS that shook the WORLD*, Three Rivers Press.

Ungar, D., Chambers, C., Chang, B.-W. & Hölzle, U. (1991), 'Organizing programs without classes', *Lisp And Symbolic Computation* **4**(3).

Ungar, D. & Smith, R. B. (1991), 'SELF: the Power of Simplicity', *Lisp And Symbolic Computation* **4**(3).

Woolf, B. (1998), Null object, *in* R. C. Martin, D. Riehle & F. Buschmann, eds, 'Pattern Languages of Program Design', Vol. 3, Addison-Wesley.