Efficient Object-Oriented Programming in Prolog

Peter Schachte*
pets@cs.mu.OZ.AU
Department of Computer Science
The University of Melbourne
Parkville, VIC 3052
Australia

Georges Saab
gs@quintus.com
Quintus Corporation
2100 Geng Road
Palo Alto, California, 94301
USA

June 9, 1994

Abstract

We discuss `classes`, a package for efficient object-oriented programming in Prolog. Successive versions are described, along with a discussion of the problems encountered when using the initial version to build application products, and the solutions devised to correct these problems. Two examples which use the package are included.

1 Introduction

Object oriented programming offers many acknowledged benefits for application developers. Encapsulation aids in providing structural clarity, inheritance provides reusibility, and class modelling can help the aspiring developer analyze the components that make up the application.[1] These were exactly the benefits that Quintus desired when we began work on WorkPro, a Client/Server application which provides graphical interfaces to relational databases combined with rules written in Prolog. Many of the existing object systems available to Quintus had severe performance penalties[2], so we decided to build a new system designed with deliverable-application performance in mind. In this paper, we present the results of this effort.

2 The `Objects` Package– A precursor to `Classes`

When we began work on our Quintus Workpro application, we knew that a full-featured graphical user interface (GUI) would be essential. We also believed that such an interface would be easiest to build and maintain using an object-oriented programming style.

Unfortunately, all the object oriented Prolog extensions we knew of at the time either maintained state using Prolog's database, or didn't allow objects to be destructively modified at all.[3] These approaches would be too costly for the heavy use we anticipated for the GUI. We wanted

*The work herein was done while this author was at Quintus Corporation.

[1] For more information on the benefits of object-oriented programming, please see a dedicated text such as Rumbaugh[11] or Booch[2]

[2] The packages that we had access to when we began this project in 1990 were Quintus ProWindows[6], ProTalk[8], and Prolog++[13]. Since this time several new packages have become available, including L&O[4], Sicstus Objects[1], Quintec Objects[10] and OL(P)[3]. It has not yet been possible to compare these systems with the `classes` package.

[3] Examples of of these sorts of systems include ProTalk[8] and Prolog++[13].

efficiency approaching that of a C++ program, and were willing to declare the types of the data stored in each object's slots.

Quintus has a package, the "structs" package[9], which allows one to declare C language data structures, and to access and set members of these data structures. Quintus Prolog has special low-level support for accessing and setting memory, which allows the structs package to inspect and modify data structures fairly quickly.[5] What we really wanted for building our GUI was an object oriented version of the structs package. So we built one, and called it the `objects` package.

2.1 Implementation Strategy

The approach we took in building the `objects` package was to transform class definitions into structs package data structure definitions, using Prolog's *term expansion* feature. This allowed us to use the structs package directly to access and modify objects.

2.1.1 Message Dispatch

We used a simple dispatch mechanism to implement message sending. This mechanism takes advantage of the fact that the structs package, and therefore the `objects` package, represent a pointer to a data structure (objects) as a term whose functor is its type (class) and whose sole argument is the address of the actual data. When a message M, with name N and arity A is sent to object O, we execute a goal `send(`O, M`)`. Each class C has a clause for the `send`/2 multifile predicate which takes advantage of first argument indexing to dispatch on the class of the object to a predicate which handles all messages for that class. This predicate again uses first argument indexing to dispatch on the message to another predicate, which implements the actual method. That is, for each class C, there's a clause

```
send(C(Addr), Msg) :-
      send_C(Msg, C(Addr)).
```

and for each message named N of arity A defined for class C, there is a clause

```
send_C(N(X1,...,XA), Obj) :-
      send_C_N(X1, ..., XA, Obj).
```

as well as clauses for `send_`C`_`$N/A+1$ implementing the method.

All of these clauses are generated from the users method definition using term expansion.

2.1.2 Inheritance

Inheritance is implemented quite simply. Since we require that a superclass be compiled before compiling its subclasses, we can keep track of the methods defined by the superclass and the methods defined by the subclasses. For each method defined by an ancestor class S and not defined (shadowed) by the class C itself, we generate an inheritance clause that dispatches to S's code for that message:

```
send_C(N(X1,...,XA), Obj) :-
      send_S_N(X1, ..., XA, Obj).
```

2.1.3 Send Super

We also implement *send super*, which allows a method to invoke the method it shadows explicitly. This permits the user to implement a method which must do just a little more work than the superclass's method by invoking that method from the body of new class's method, so the user need only write the code that is different.

2.1.4 Object Constructors and Destructors

An object is created by calling the `new_instance`/2 procedure, passing a term describing the object to create. The principle functor of this term must be the class of the object to be created, and the arguments (if any) are passed as parameters to the creation method for the class. This method may initialize any slots, create other objects, or do whatever other initialization is necessary. The initialization method may be omitted if no initialization and no arguments are needed. Similarly, if a destruction method is defined, it is automatically called when the user calls `dispose_instance`/1 to destroy an object.

2.2 Experience

This system was implemented quite quickly—the bulk of it was completed in a few days. This time constraint severely limited what features could be supported by this package.

Nevertheless, the package proved quite useful. Many thousand lines of class definitions were written in it, and Quintus Workpro was released to many customers using this technology. The performance was very good—in fact the `objects` package was not thought to be a bottleneck for Quintus Workpro at all.

2.2.1 Code Expansion

After using the `objects` package for a while, we found that we generally liked the object oriented style for developing GUIs, but we found several flaws in the `objects` package. One serious problem was the code expansion we observed when defining many classes. This was due to the fact that the `objects` package was built on the structs package. The structs package generates 3 clauses for each structure slot, and 3 more for each structure. Since the `objects` package generates one structure for each class, and since each object slot, inherited or not, is a slot in the generated structure, we found that the `objects` package generated larger programs than we would have liked.

2.2.2 Module Objects

But this was not the only problem. Parts of the Quintus Workpro system other than the GUI used the `objects` package. Some of these uses required the ability to store terms in object slots, which the `objects` package couldn't support, built as it was on C language data structures. So fairly early on in the development of Quintus Workpro, we put a hook into the `objects` package to allow messages to be sent to other kinds of objects. These objects were represented as terms of the form `module(`*Desc, Term*`)`, where *Desc* is a descriminator which determines what code handles the message, and *Term* is some other term, usually an integer ID. The `objects` package handles a message M sent to such an object O, called a module object, by calling `send_to_module(`*Desc, M, O, Term*`)`. This predicate is multifile, so other packages could register their own sorts of objects.

This didn't prove to be a very satisfactory solution. It didn't allow methods for these module objects to be defined in the same way as for ordinary objects. Methods, and even the method dispatch, for module objects had to be coded as ordinary Prolog code. This is not difficult, but is a nuisance to write and maintain. The nuisance becomes a major impediment where there is a need to subclass such classes, since there is no mechanism for inheritance for module objects. Perhaps most significant, this seriously weakens encapsulation.

2.2.3 Term Slots

The impossibility of storing a Prolog term in a (non-module) object was also occasionally an irritation in other parts of the system. It could always be worked around using a dynamic predicate to store the terms, but this once again damages encapsulation, and makes the code uglier.

2.2.4 Get and Put Methods

Another reason for discontent with this package lay in accessing and modifying object slots. Since slots were accessed and modified through the structs package, any predicate could directly manipulate an object's slots. We generally resisted the temptation to do this, and instead adopted a standard message, `get`/2 for accessing slots (or finding other information) of an object, and `put`/2 for modifying slots (or other information). For example, we might send the message `get(height, H)` to find the height of some object, and `put(height, H)` to set the height. Since our implementation indexes on the first argument of a message, this was reasonably efficient.

The discipline of not directly accessing slots from outside of a class's methods gave us the encapsulation we wanted, and we generally found the duality between `get`/2 and `put`/2 quite comfortable. But `get`/2 and `put`/2 methods have a serious problem: `get`/2 and `put`/2 shouldn't really be methods, they should be thought of as families of methods, or perhaps *meta*-methods. The individual procedures to access or modify a slot should be separate methods. This became clear as soon as we defined a subclass that added slots to its parents: defining a `get`/2 method to access a slot introduced by the child class immediately shadowed the `get`/2 method of the parent class, shadowing all the accessors for inherited slots.

We were able to work around this flaw by using *send super*, forwarding to the superclass `get`/2 or `put`/2 messages not understood by the child class. Effectively, we implemented inheritance explicitly for each `get`/2 or `put`/2 method we wrote. This was a nuisance, but more importantly, it meant that we were using runtime inheritance for these methods. Since `get`/2 and `put`/2 were two of the most prevalent messages in our code, this was a significant performance penalty.

2.2.5 Descendent Slot Types

Our approach of building the `objects` package on our existing structs package created one subtle but significant restriction. When an object has a slot which is declared to point to an object of a certain class C, one would like to allow an object of any descendent class D to be stored in that slot. But the structs package will insist that the object to be stored in that slot must be an instance of *exactly* class C, not a descendent class. This is quite proper for a package implementing simple data structures, but not for an object oriented programming package.

Fixing this problem is not as simple as casting the object to the appropriate class when storing it: when fetching the object, we must be able to find its class. If we just assume that an object pointed to by a slot declared to point to objects of type C, then when we send a message to this object, we will inappropriately use C's method, rather than the method defined for that object's class, D. To fix this problem, each object must "know" its class.

2.2.6 Handling Undefined Methods

After developing a little code using the `objects` package, we discovered that our efficient message dispatch mechanism had one serious drawback: a message sent to an object that doesn't define or inherit a method for that message simply fails. This may have some theoretical attraction, but for people trying to debug a large system with many methods, it is distinctly unattractive.

2.2.7 Single Inheritance

Finally, we found that despite its theoretical and implementation problems, multiple inheritance does have some uses. In any case, it currently seems to be an expected feature for any good object oriented programming system.

3 The Second Version: The `Classes` Package

Our experience with the `objects` package was positive enough to spur us to set about fixing its flaws.

Since the code expansion problem required us to modify the package not to use the structs package, this had to be a significant rewrite. Some of the other flaws required other major changes. We therefore decided to rewrite the package from scratch; while we were at it, we decided to try to improve the package's efficiency, as well. For this version, we devoted much more time to design and development.

To avoid confusion with the `objects` package, we named this second implementation the `classes` package. This new package is not backward compatible with the `objects` package, because we decided that we wanted to be free to make significant changes.

While reading this section, you may find it helpful to refer to the small example given in section 5.1.

3.1 Syntax

For one thing, we created a class declaration syntax. The `objects` package used the structs package syntax; for the `classes` package, we tried to choose a syntax that better fit the object oriented paradigm. This syntax allows multiple inheritance, initial slot values, and public and private slots[4].

3.1.1 Get and Put Methods

We also generalized the syntax for sending a message. In view of the problems mentioned above with `get`/2 and `put`/2 messages, we decided to promote the operations of getting and putting from the level of messages to the level of message operators. That is, the `classes` package has three different kinds of messages: *get*, *put*, and *send* messages; where the `objects` package used only the operator `<-` to send messages, the `classes` package adds `>>` and `<<` for getting and putting, respectively[5]. This means that while for the `objects` package, a name and arity were sufficient to specify a message, for the `classes` package, one also needs a message operator to fully specify a message.

[4] Our private slots are equivalent to C++'s *protected* members; the `classes` package doesn't support what C++ calls *private* members. That is, a class can always access its ancestor classes' slots.

[5] The `classes` package also allows users to define other message operators, if they wish. We have never found the need to use this feature, though.

This has been a controversial change. Some may find it confusing to have different *kinds* of messages. Others have commented that there should only be two kinds of messages: send and get. Still, this approach is somewhat reminiscent of Common Lisp's `setf` macro[12], which has proved to be very popular and convenient. Furthermore, the `classes` package itself does not force one to use all of these kinds of messages, but it does encourage it by generating a get and a put method for every slot that is not declared private.

These methods need not be restricted to slot access. For example, it would be sensible for a window class to have a `managed`/1 get and put method, whose argument would be `true` or `false`, to inquire whether or not the window is managed, and to manage or unmanage it. This is more symmetrical, more flexible, and, once this style of slot access is learned, easier to remember, than having a method to inquire whether or not the window is managed, one to manage it, and one to unmanage it. It also generalizes better when, for example, an `iconified` state is added. The expectation is that there will usually be a get method with the same name and arity as each put method (though not necessarily the converse).

Another benefit of writing the `classes` package from scratch, instead of building on the structs package, is that we now have control over slot access. We have taken advantage of this to prevent direct access to slots except from a class's methods[6].

3.2 Compilation

Much of the work on the implementation of the `classes` package went into the compilation of methods. This was done, in part, because we knew that fixing some of the flaws of the `objects` package would reduce the runtime performance of the package, and in part because we simply wanted the package to be more efficient.

The most significant change we made in message handling was to provide for expansion of code that sends messages, as well as the method code itself. This allowed us to unfold much of the code for sending most messages, significantly reducing the overhead of sending messages for the common case.

3.2.1 Message Dispatch and Inheritance

Typically when sending a message, the message operator and the message's name and arity are known at compile-time. Therefore, we can do all the message dispatch, except for dispatching on the class of the actual object, at compile-time[7]. To do this, though, we had to rearrange the dispatch mechanism so that we first dispatch on the message (name and arity), and then on the class of the object receiving the message.

Of course, we also allow the message or even the message operator to be unbound at compile-time, as long as they are bound at runtime. If the message operator is unbound at compile time, then we essentially have an ordinary metacall, in which case the predicate `<-`/2 is called at runtime. The same applies for the get and put methods `>>`/2 and `<<`/2. These predicates do some runtime checking, and then call the multifile predicate with the same name and arity (but arguments swapped) in a separate module. This predicate is generated by the `classes` package, one clause for each distinct message, to dispatch on the message name and arity, and

[6] The `classes` package does carefully maintain enough information for a program to circumvent this protection and directly access any slot of any object (though not without some pain). This flexibility is important for debugging purposes, and it is hoped that it will not be abused by applications to thwart encapsulation.

[7] The `classes` package supports only late binding of methods (in C++ parlance, virtual member functions), because we feel this is semantically much cleaner. We do not try to determine the class of the object receiving a message at compile-time, because even if we could, at runtime it could always be an instance of a descendent class that received the message.

call a *message* procedure. The message procedure dispatches on the class (that is, the principal functor) of the object receiving the message, and calls the *method* procedure, which actually implements the method.

Thus for each message named N of arity A for message operator `<-` defined for any class, there is a clause that looks like[8]:

```
<-(N(X1,...,XA), Obj) :-
        '<-N'(Obj, Obj, X1, ..., XA).
```

(Note that the `Obj` argument is passed twice: once just for indexing, and once for any actual use of that object in the method. This saves us from reconstructing the object term on the Prolog heap.) For each class C that defines a method for this message, there is a clause:

```
'<-N'(C(_), Obj, X1, ..., XA) :-
        'C<-N'(X1, Obj, 0, X2, ..., XA).
```

For each descendent class D that inherits this method from C, there is a clause:

```
'<-N'(D(_), Obj, X1, ..., XA) :-
        'C<-N'(X1, Obj, Offset, X2, ..., XA).
```

The *Offset* argument is a number determined at compile time; its purpose will be described below in the context of multiple inheritance. Note that the rather odd argument ordering is used to minimize argument shuffling, reducing runtime dispatch overhead, while still providing indexing on the first argument of the method. Method clauses for message N of arity A for class C will be compiled into clauses for the predicate 'C<-N'/A+2. The same naming pattern is used for message operators other than `<-`.

3.2.2 Unfolding Message Dispatch

The `classes` package uses Prolog's term expansion feature to transform every clause in the program. Where it finds a message being sent for which the message name and arity are known at compile-time, it unfolds as much of the message dispatch code as possible, generating code to call the message procedure directly. That is, when it sees a goal

```
Obj <- N(X1, ..., XA)
```

it turns it into the goal

```
'<-N'(Obj, Obj, X1, ..., XA)
```

Since this procedure directly calls the method procedure, the only runtime overhead of sending a message, over and above the cost of a predicate call, is the cost of the message procedure: just one procedure call and indexing on the class (principle functor) of the object.

3.2.3 Inline Messages

For many messages this is quite an acceptable overhead. However, for some messages which do very little (for example, slot accessor messages), even this overhead is too much. We therefore extend the compilation scheme to generate better code for simple methods, where "simple" is defined by a heuristic. This optimization, though, can't be done one clause at a time, since the compiler must see all the clauses for a method to decide whether or not it is simple. We therefore take advantage of the fact that Quintus' term expansion facility allows a list of clauses

[8] To simplify our exposition, we omit the module qualifiers in these descriptions. We actually put all of these clauses in modules separate from user code to avoid predicate name collisions.

to be returned as the result of an expansion: every method clause is expanded to the empty list of clauses. When we've seen the end of the class declaration, we analyze all the clauses for all the methods of that class, finding the simple methods, and generate the code for all of that class's methods at once. For simple methods, we unfold the call from the message procedure to the (simple) method clause, effectively inlining the method in the message dispatch. This means that the overhead of sending a simple message, over and above the overhead of an ordinary Prolog procedure call, is only the cost of indexing on the object receiving the message, which in practice is negligible.

For example, if the body of a `rectangle` class's `area`/1 method is:

```
fetch_slot(width,W), fetch_slot(height,H), A is W*H.
```

assuming this is considered simple (the current heuristic would consider this simple), the message clause generated for this message would be[9]:

```
'<-area'(rectangle(Addr), Self, A) :-
        W is integer_at(Addr),
        H is integer_at(Addr+4),
        A is W*H.
```

No code would be generated for a `'rectangle<-area'`/3 procedure, as would be expected for a non-inline method. Subclasses of `rectangle` that don't define their own `area`/1 method would get the same code, except that the `rectangle`/1 functor in the head of the clause would be replaced with the subclass' names.

Furthermore, when an inline method (say for `a`/1) sends another inline message (say `b`/1) to the same object, we can compile the code for `b`/1 inline in the body of `a`/1. In this case, the overhead of the `b`/1 message is literally zero. This optimization can be performed because when a descendent of this class inherits `a`/1, we will duplicate all the code for `a`/1, so if this descendent class should redefine `b`/1, we will generate different code for `a`/1. Unfortunately, we cannot make this optimization if `a`/1 is not simple, because the code for a non-simple method is shared by all classes that inherit it. This means that if `b`/1 were redefined by a subclass that inherited `a`/1, the wrong method for `b`/1 would be executed.

3.2.4 Multiple Inheritance

The `classes` package implements multiple inheritance. Since all inheritance is computed at compile time, this does not add much runtime overhead. It does create two problems, however: how to resolve the collision that occurs when one class inherits the same message from two (unrelated) ancestors; and how a method can find the right slots.

We chose to handle the problem of inheritance collisions in a very simple way: force the user to choose which ancestor to inherit from. The `classes` package issues a warning whenever a class inherits the same message from more than one ancestor class without a directive specifying which one to inherit.

The problem of finding an object's slots is a bit tricky. With single inheritance, we can easily implement slot inheritance by adding a subclass's slots after the superclass's. Thus the superclass's methods can directly access a given slot at the same offset for any descendent of the class that defined that slot. With multiple inheritance, this is no longer possible.

The solution we adopted for this problem is to pass to each method the offset at which it

---

[9] `integer_at` is an expression supported by Quintus Prolog that evaluates to the 32 bit integer stored at the specified address. Quintus Prolog supports a similar approach to memory modification.[5]

In this example, I assume the width and height of the rectangle are 32 bit integers stored as the first two words of the rectangle object. Other configurations would work equally efficiently, with slightly different code.

can find the slots of the class that defines that method. This is what the *Offset* argument in the linkage clause above is for. If all of the classes that inherit that message have only one superclass, that offset will always be zero. However, when a class C with multiple superclasses inherits a method from ancestor class A, the offset will be the byte offset of the beginning of A's slots in C.

Note that for inline methods, this offset computation is combined into the address calculation for slot access and modification. This means that the code shown above for the inline `area/1` method for the rectangle class would be just as simple, but with different slot offsets, for a subclass of `rectangle` that had multiple superclasses.

3.3 New Features

We tried to fix all the flaws we discovered in the `objects` package. Following are some features we introduced in the `classes` package to correct these flaws.

3.3.1 Term Slots

The `objects` package introduced the concept of "module objects" to work around the fact that it had no way to store general Prolog terms in object slots. The `classes` package addresses this straight on by supporting term slots. The implementation is quite simple: for each term slot there is a binary predicate named for the slot whose first argument is the address of an object, and whose second argument is the contents of that slot of the object at that address. The `classes` package allows you to access and set these slots in exactly the same way as any other slot.

3.3.2 Term Objects

To some extent, term slots obviate the need for "module objects," but not entirely. Module objects allow users complete control over the storage of their objects, which term slots don't. We also thought it would be useful to be able to treat ordinary compound Prolog terms as objects, sending them messages and accessing their slots (though of course not modifying their slots) using the same mechanisms used for other objects. For classes that don't need to have their slots modified, this would be an efficient way to represent them. Since ordinary objects are `malloc`'d, there is a significant time penalty in creating one, and some wasted space, For example, it might be better to use a compound Prolog term, rather than an object, to represent a two-dimensional X-Y position.

Storing such an object in a slot of an ordinary object, however, is a bit trickier. Since ordinary objects are not reclaimed by backtracking, and Prolog terms are, objects must not hold pointers to terms, otherwise the object will have a dangling pointer after backtracking. Our solution to this problem is to allow users to define the *essence* of the term, the parts of it which define it. In most cases, this will be all the arguments of the term, but it need not be. The user specifies the types of each part of the essence of the term, and the `classes` package stores all these values wherever it needs to store one of these objects. For example, if we defined `point(`X, Y`)` to be a term class called `point`, and defined X and Y, both integers, to be its essence, then any object which had a slot of type `point` would store the two integers to store the point. Accessing that slot would create a `point/2` term, and storing into it would take apart one.

3.3.3 Descendent Slot Types

One flaw of the `objects` package that we especially wanted to fix was its inability to store in a slot a pointer to an instance of a descendent class of the declared class for that slot. The crux of the problem is that the `objects` package only knows the class of an object by the functor of the Prolog term that wraps its address. Once the address of the object is stored in a slot in another object, there is no way to determine its class.

The `classes` package fixes this problem by storing the class of an object before its first word of data. So when the `classes` package fetches the contents of an object pointer slot, it reads the first word of memory before the object's data and uses it and the object's address to compose the term representing the object. This is a significant performance penalty, and one of the few aspect of the `classes` package that is slower than the `objects` package.

3.3.4 Handling Undefined Methods

As mentioned earlier, the dispatch mechanism for the `objects` package precludes catching undefined messages without an unacceptable performance penalty. The `classes` package improves on this in two ways. Firstly, since we analyze messages at compile time, we can detect at compile time an attempt to send a message that is not defined for any class, and issue a warning. The second improvement is a pragmatic one: we provide a directive to tell the `classes` package to generate code to check each message for validity before sending it. We also generate code to check that an object being stored in an object pointer slot is a descendent of the declared class of that slot. This directive remains in effect until the compiler sees a directive to turn it off. These tests slow message sending and storing into class pointer slots significantly, but it is intended that these directives be removed from a piece of code once it's been debugged. Certainly this is not an ideal solution, but it does aid debugging without crippling performance when not debugging.

3.3.5 Instance Methods

Another feature we introduced in the `classes` package is *instance methods*, methods which may differ for each instance of a class. This brings the power of Prolog's feature of dynamic predicates to object oriented programming. Users must explicitly declare any methods which they wish to make specific for individual instances in the class definition. They may then optionally provide code for that method, which will serve as a default method for each instance, but which will be superseded for an individual object if another method is specified for that object.

One would want to use this feature when different instances of a class need to behave differently. For example, each instance of a pushbutton class needs to perform a different operation when the button is pressed. One sensible way to implement a pushbutton class is for it to send itself a `pressed` message when the button is pressed. If this is an instance method, it will be easy to make different buttons behave differently when pressed.

3.3.6 Other New Features

The `classes` package has a few other new features which our experience with the `objects` package suggested would be useful. The `classes` package allows one to define a class of which no instances can be created. Such a class, called an *abstract class*, may be useful as a superclass to other classes which can be instantiated. Also, as well as allowing (actually forcing) one to declare which superclass a class is to inherit a method from, the `classes` package allows one to declare that it *shouldn't* inherit a specific method at all. This can be used to make an abstract class be a descendent of a concrete (non-abstract) class by not inheriting any `create` methods, or simply when a superclass's method doesn't make sense for the subclass.

3.4 Results

The `classes` package was completed recently, and a version of Quintus WorkPro has been successfully built with it. It is rather early yet to report results, but the first indications are positive.

Certainly the performance improvement is significant. Preliminary benchmarking on a Sun SparcStation 2 shows that the best case for get and put messages with the `objects` package is over 20 microseconds; with the `classes` package, the worst case is under 6 microseconds. Note that this is for a class with no superclass; an inherited get or put method would be much worse for the `objects` package, using runtime inheritance as mentioned above, while the `classes` package would experience no degradation. An ordinary inherited method takes almost 13 microseconds under the `objects` package, compared to about 3.5 microseconds under the `classes` package (both of these numbers also count the time for 3 trivial Prolog predicate calls). Method inlining seems to bring a substantial speedup for the `classes` package, reducing the benchmark to 0.2 microseconds![10]

3.5 Important Omissions

There are still a few features we wish the `classes` package could support.

3.5.1 Garbage Collection

Garbage collection is one of the most significant omissions of the `classes` package. Prolog programmers are accustomed to having storage managed for them, and an object oriented programming package that forces the programmer to manage their own storage is definitely a step backward in this respect.

Unfortunately, garbage collecting objects would require the cooperation of the Prolog system, as there is no way for the `classes` package to know whether an object is referred to from a Prolog term. This would surely require a big change in the Prolog system, and probably in the `classes` package, as well.

3.5.2 Object/Term Integration

A subtler problem is the weakness of the integration of Prolog terms and objects. As was pointed out earlier, we can't store a pointer from an object to a Prolog term, because the Prolog term will be reclaimed on failure, and the object won't, leaving the object with a dangling pointer. Also, we can't use unification to create and analyze objects, as we can do with terms. This, for example, makes it less intuitive for an object to contain a list of something. Certainly it is easy to create a list class, but writing code to traverse such a list is much less convenient than for Prolog lists.

[10] In fact, the benchmark had to be changed slightly to be able to measure this. The measured time for the `objects` package on this benchmark was 13.6 microseconds.

4 Application Experience: Quintus WorkPro

Quintus WorkPro is a Client/Server application which provides a graphical interface to relational databases, and a set of tools which allow the WorkPro administrator to build and customize the database schema and the graphical user interface of the WorkPro application.

WorkPro has been installed in over 125 sites worldwide at such large companies as Exxon, Intel, Hewlett-Packard, and Mead Data Central. It is now well into its second release cycle and will soon be available on Microsoft Windows NT and Windows 3.1 on PCs as well as on Unix systems like Sun, HP, and IBM RS6000.[11]

Of WorkPro's 50000 lines of Prolog code, about half uses the `objects` package either to define classes or to send messages to instances. The main areas which use objects extensively are the graphical user interface, as well as the 'destination' classes, which encapsulate routing textual output from the application to scrolling windows, files, email and faxes. As far as was possible, the graphical classes were designed to be independent of windowing system, so that they could be retargeted to other windowing systems (like MS Windows) as easily as possible.

This approach proved very useful at an early stage, as customers soon demanded a TTY interface as well as a Motif GUI. Quintus was able to respond by developing a TTY version of the GUI classes which supported the same classes and methods as the Motif version using an interface from Prolog to the Unix curses library. Isolating the programming of the user interface to `objects` classes made it possible to build the TTY version by only redefining the these classes. The rest of the WorkPro code remained the same. A recent project whose goal was to port the WorkPro client GUI from Motif to Microsoft Windows was extremely successful, in large part because the classes and methods which were used encapsulated all the interaction between the GUI and the application, so that only the GUI classes needed to be ported; the rest of the application remained the same.

There are about 75 classes in WorkPro, and about 600 methods (including overridden, access, and private methods). In addition to the classes which define dynamically created instances, there are a set of classes which handle persistently stored instances which remain in the relational database and are loaded by the WorkPro client.

The main problems with the `objects` system that were encountered when implementing WorkPro were the code-expansion problem and the lack of slots which could handle instances of descendants. These have been fixed in the `classes` package and the development version of WorkPro which is based on `classes` has already taken advantage of these.[12]

WorkPro is one of the largest and most successful Prolog applications commercially available today. It has has benefitted tremendously from using the `objects` package both during the implementation phase and the support and maintenance phase.

5 Examples

In order to demonstrate the use of the `classes` package, we have prepared two examples. The first is a simple example which implements some standard graphical support classes using the `classes` package. The second is a more detailed example which builds a color_chooser application with a Motif GUI.

[11] These machine and windowing system names are trademarks of their respective owners.

[12] Measuring the difference in code-expansion has proved difficult because other factors have changed as well. Storing instances of decendants of a slot's type has been used in many places, and has proved very useful.

5.1 Simple Example: Positions and Sizes

We begin with a few simple example classes to introduce the `classes` syntax.

5.1.1 A `position` Class

One class you may find useful when working with graphics is a `position` class which encodes an (X,Y) position pair, where X and Y are integers. Here's how you might implement it. (The following text is in file `position_class.pl`.)

First, you must load the standard `classes` files.

```
:- module(position_class, []).
:- load_files(library(class_decl), [when(compile_time), if(changed)]).
:- use_module(library(classes)).
```

Next we declare the position class:

```
:- class position = [x:integer, y:integer].
```

We want to be able to create a position without specifying any parameters, in which case `x` and `y` will be initialized to 0. It would also be convenient to be able to specify the initial values for `x` and `y` at creation time. These create methods (constructors) allow this.

```
Self <- create.
Self <- create(X, Y) :-
        store_slot(x, X),
        store_slot(y, Y).
```

Since we did not declare `x` and `y` to be private, they may be publicly accessed, which is what we want. We have noticed, however, that usually when we want to set or get one, we also want the other. Therefore, we also define get and put methods that work with both `x` and `y` at once:

```
Self >> position(X, Y) :-
        fetch_slot(x, X),
        fetch_slot(y, Y).
Self << position(X, Y) :-
        store_slot(x, X),
        store_slot(y, Y).
```

That's all we need for our `position` class. Here's how we might use it at the Prolog prompt:

```
| ?- [library(classes), position_class].
```

Prolog prints file loading messages.

```
| ?- create(position(20,30), P),
     P >> position(X,Y).

P = position(1395228),
X = 20,
Y = 30
| ?- create(position, P),
     P >> position(X1,Y1),
     P << y(10),
     P >> position(X2,Y2),
     P << position(-50,50),
     P >> position(X3,Y3).

P = position(1395236),
X1 = 0,
```

```
Y1 = 0,
X2 = 0,
Y2 = 10,
X3 = -50,
Y3 = 50
```

5.1.2 A `size` Class

Another useful class represents a geometrical size. This file is `size_class.pl`.

```
:- module(size_class, []).
:- load_files(library(class_decl), [when(compile_time), if(changed)]).
:- use_module(library(classes)).

:- class size = [width:integer, height:integer].

Self <- create.
Self <- create(Width, Height) :-
        store_slot(width, Width),
        store_slot(height, Height).

Self >> size(Width, Height) :-
        fetch_slot(width, Width),
        fetch_slot(height, Height).
Self << size(Width, Height) :-
        store_slot(width, Width),
        store_slot(height, Height).
```

We also might find it useful to get the area of a `size`.

```
Self <- area(A) :-
        Self >> size(Width, Height),
        A is Width*Height.
```

After loading this file, we test out our new class.

```
| ?- create(size(20,30), S),
     S <- area(A).

S = size(1395242),
A = 600
```

5.1.3 A `region` Class

Finally we illustrate inheritance, in this case multiple inheritance, with a `region` class which is rectangular region determined by the position of its upper left corner and its size. This is file `region_class.pl`.

```
:- module(region_class, []).
:- load_files(library(class_decl), [when(compile_time), if(changed)]).
:- use_module(library(classes)).
```

For the `region` class, since it has superclasses, we must make sure that the code for the superclasses are available when we use this class. Furthermore, we must also ensure that these files are loaded when we *compile* this file, so that the class declarations of the superclasses are known to the `classes` compiler. The following two lines ensure that both of these requirements are met.

```
:- load_files(position_class, [when(both), if(changed)]).
:- load_files(size_class, [when(both), if(changed)]).
```

```
:- class region = position+size.

Self <- create.
Self <- create(X, Y, Width, Height) :-
        Self << region(X, Y, Width, Height).
```

We don't want to allow a `region` to be created specifying two parameters, as would happen if we inherited the `<-create`/2 method from `position` or `size`. In fact, this would be an inheritance collision, as we would inherit that method from both superclasses. What we really want is not to inherit either `<-create`/2 method.

```
:- uninherit _ <- create/2.
```

And we again define a get and put method to work with all the data at once. For most classes you would write, this is not a sensible thing to do, but for positions, sizes, and regions, it is convenient.

```
Self >> region(X, Y, Width, Height) :-
        Self >> position(X,Y),
        Self >> size(Width, Height).
Self << region(X, Y, Width, Height) :-
        Self << position(X,Y),
        Self << size(Width, Height).
```

A quick test:

```
| ?- create(region(10,20,30,40), R),
     R >> position(X, Y),
     R <- area(A).

R = region(1395256),
X = 10,
Y = 20
A = 1200,
```

5.2 Detailed Example: Color Chooser

An application area which lends itself very naturally to using object-oriented style is the graphical user interface (GUI). This example builds the GUI for an application which allows us to change its color scheme by popping up a color chooser. In order to create and manipulate the interface this example uses `ProXT`[7], an interface to OSF Motif and the X toolkit (Xt).

5.2.1 Application Description

The color chooser is a non-modal dialog which has a frame displaying the currently selected color, three `sliders` which control the red, green, and blue amounts in the selected color, and an OK button which allows us to accept the currently selected color. Accepting the currently selected color in the color chooser should set the background of the application to that color.

Already from this description we can see that there are several pieces of GUI that we will need to create, and several pieces of control. There appear to be places where we will be able to reuse components within the application, most notably the three sliders, and possibly the color chooser unit itself, if we choose to allow multiple color choosers to come up at once.

We start by choosing the classes that we will need for the `color_chooser`. We know that we need three `sliders` to control the Red, Green, and Blue settings. So we probably want to have a `slider` class, and we can make each of these sliders an instance of that class. Through `ProXT` we have access to the Motif `XmScale` widget, so we only need to encapsulate the functionality that the `XmScale` widget offers to have the `slider` class.

The next thing we need is a box which will display the current color. Once again, we can use a Motif widget accessible through `ProXT`: the `XmFrame` widget. The final thing that we we will need for the basic `color_chooser` is some kind of container which will hold the `sliders` and the `frame`. The `XmRowColumn` widget is a good candidate for this, because it will display the `sliders` and `frame` in an attractive manner while requiring a minimal amount of 'layout' work on our part.

Since we will now be creating at least three types of Motif widgets, and the interface for creating and manipulating these is very similar, we can start to abstract a bit, by adding the class `graphical_object`, which will define the behavior for classes which are wrappers around Motif classes. The basic operations that we need for `graphical_objects` are `create`, `destroy`, `add_callback`, get and put methods for `managed` and a put method for `color`.

5.2.2 The `graphical_object` Class

This is the root of the hierarchy of graphical objects. In its most basic form, this class will act as a wrapper around a `ProXT` widget, handling creation, management, and destruction. Note that this is an abstract class, which is not intended to be instanced directly, and therefore it has no `<- create` method of any arity. In order to be instanceable, subclasses must define a `<- create/3` method, which will usually send a `<- create_graphical_object/4` message to itself to create the widget and store it in the object. There is one slot for this class, which is intended to store the widget that corresponds to the class instance.

```
:- class graphical_object = [widget:pointer(widget)].
```

`<- create_graphical_object`(+*WidgetClass*, +*Parent*, +*Name*, +*Attribs*)

Create the instance by calling the `ProXT` predicate `xtCreateWidget/5`. The list of widget attributes *Attribs* is passed through to the `xtCreateWidget/5` call. Note that by default the widget is not managed.

```
Self <- create_graphical_object(WidgetClass, Parent, Name, Attribs) :-
        Parent >> widget(ParentWidget),
        xtCreateWidget(Name, WidgetClass, ParentWidget, Attribs, Widget),
        Self << widget(Widget).
```

`<< managed`(+*Bool*)

Define the put method for `managed` to manage or unmanage the widget. This makes it 'visible' in its parent widget.

```
Self << managed(Bool) :-
        Self >> widget(Widget),
        ( Bool == true ->
          xtManageChild(Widget)
        ; Bool == false ->
          xtUnmanageChild(Widget)
        ; raise_exception(value_not_a_boolean(Bool))
        ).
```

`>> managed`(-*Bool*)

Define the get method for `managed` to succeed if it can unify *Bool* with the current management state of the widget. *Bool* is `true` if the widget is managed, `false` if it is unmanaged.

```
Self >> managed(Bool) :-
        Self >> widget(Widget),
        xtIsManaged(Widget, Bool).
```

`<- add_callback`(+*Callback*, +*Functor*, +*DestObject*)

Add a method for adding callbacks to the widget. *Functor* is name of the Prolog predicate (of arity 3) that will be called when a callback occurs. *DestObject* is the object to which a message should be sent saying that the callback has occurred. By convention, the callback predicate *Functor*/3 sends the message `<-Functor`/0 to *DestObject*. *DestObject* is passed to *Functor*/3 in the client data slot (the second argument of the `ProXT` callback predicate).

```
Self <- add_callback(Callback, Functor, DestObject) :-
        Self >> widget(Widget),
        xtAddCallback(Widget, Callback, Functor, DestObject).
```

`<< color`(+*FG*, +*BG*)

Add a put method for setting the widget's foreground and background color. This demonstrates using a put method which has more than one argument.

```
Self << color(FG,BG) :-
        Self >> widget(Box),
        xtSetValues(Box, [xmNbackground(BG),
                          xmNforeground(FG)]).
```

In a more general `color_chooser` class, we would probably define a `>> color` method as well, but in this example it is not needed.

`<- destroy`

Define a method for cleaning up when the instance is destroyed using `destroy`/1 from the `classes` package. For `graphical_object`, this method destroys the instance widget.

```
Self <- destroy :-
        Self >> widget(Widget),
        xtDestroyWidget(Widget).

:- end_class.
```

With this as a start, it is very simple for us to define the three subclasses of `graphical_object` (`slider`, `frame`, and `row_column`). We will inherit all of the methods from `graphical_object`, and define a `<- create`/3 message so that the class is instanceable. The `slider` class will also have additional methods added for getting the value of the `slider`.

5.2.3 The `frame` Class

This subclass of `graphical_object` encapsulates the `XmFrame` widget class.

```
:- class frame = graphical_object.
```

`<- create`(+*Parent*, +*Name*, +*Attribs*)

Create the graphical object using `xmFrameWidgetClass` as the widget class.

```
Self <- create(Parent, Name, Attribs) :-
        Self <- create_graphical_object(xmFrameWidgetClass,
                                        Parent, Name, Attribs).

:- end_class.
```

5.2.4 The `row_column` Class

This subclass of `graphical_object` encapsulates the `XmRowColumn` widget class.

```
:- class row_column = graphical_object.
```

`<-` `create`(+*Parent*, +*Name*, +*Attribs*)

Create the graphical object using `xmRowColumnWidgetClass` as the widget class.

```
Self <- create(Parent, Name, Attribs) :-
        Self <- create_graphical_object(xmRowColumnWidgetClass,
                                        Parent, Name, Attribs).

:- end_class.
```

5.2.5 The `slider` Class

This is a subclass of `graphical_object` which encapsulates the `XmScale` widget class. In addition to the methods inherited from `graphical_object`, this subclass adds the get method `value`/1, which returns the current integer value of the scale.

```
:- class slider = graphical_object.
```

`<-` `create`(+*Parent*, +*Name*, +*Attribs*)

Create the graphical object using `xmScaleWidgetClass` as the widget class.

```
Self <- create(Parent, Name, Attribs) :-
        Self <- create_graphical_object(xmScaleWidgetClass,
                                        Parent, Name, Attribs).
```

`>>` `value`(*?Value*)

Unify *Value* with the slider value of the instance widget.

```
Self >> value(Value) :-
        Self >> widget(Widget),
        xmScaleGetValue(Widget, Value).
```

`<<` `value`(+*Value*)

Set the slider value of the instance widget to *Value*.

```
Self << value(Value) :-
        Self >> widget(Widget),
        xmScaleSetValue(Widget, Value).
```

5.2.6 The `color_chooser` Class

This class defines the `color_chooser`, a complex class which contains three `slider`s and a `frame` contained in a `row_column`. The behavior is that as the `slider`s change, the color of the `frame` is modified according to the new values of the `slider`s. In the class definition we define the slots storing internal instances as private. The color chooser should be accessed as a black box, that is, no messages should be sent directly to internal instances from outside the class.

```
:- class color_chooser = row_column + [
                                       private red_slider:slider,
                                       private green_slider:slider,
                                       private blue_slider:slider,
                                       private colorbox:frame
                                      ].
```

`<- create(+Parent)`

Creates the `color_chooser` instance as a child of the instance *Parent*. The `color_chooser` consists of a `row_column` which contains three `sliders` (red, green, and blue), and a `frame` which changes color as the `sliders` are changed. Create the `color_chooser` by 'sending super' to create the `row_column`, and then creating the `slider` and `frame` instances. Store the instances in the appropriate slot.

```
Self <- create(Parent) :-
        Self <- create(Parent, color_chooser, [xmNorientation(xmHORIZONTAL)]),
   % Create and store the slider instances:
        create(slider(Self, red, []), Red),
        store_slot(red_slider,Red),
        create(slider(Self, green, []), Green),
        store_slot(green_slider,Green),
        create(slider(Self, blue, []), Blue),
        store_slot(blue_slider,Blue),
   % Create and store the frame instance:
        create(frame(Self, colorbox, [xmNwidth(200),
                                       xmNheight(200)]), Colorbox),
        store_slot(colorbox,Colorbox),
   % Add value changed callbacks to the sliders so that we will be sent
   % the message slider_changed/0 if the user moves one of the sliders.
        Red   <- add_callback(xmNvalueChangedCallback, slider_changed, Self),
        Green <- add_callback(xmNvalueChangedCallback, slider_changed, Self),
        Blue  <- add_callback(xmNvalueChangedCallback, slider_changed, Self),
   % Add drag callbacks to the sliders so that we will be sent
   % the message <-slider_changed as the user drags the slider.
        Red   <- add_callback(xmNdragCallback, slider_changed, Self),
        Green <- add_callback(xmNdragCallback, slider_changed, Self),
        Blue  <- add_callback(xmNdragCallback, slider_changed, Self),
   % Manage the components of the color_chooser
        Red      << managed(true),
        Green    << managed(true),
        Blue     << managed(true),
        Colorbox << managed(true),
        Self     << managed(true),
   % Send the callback message to set the Frame to the appropriate color
        Self  <- slider_changed.
```

`<- slider_changed`

When the slider changes, we should get the color as calculated from the slider values by the `>> color` method, and then set the internal colorbox to display that color:

```
Self <- slider_changed :-
        Self >> color(FG, BG),
        fetch_slot(colorbox,Colorbox),
        Colorbox << color(FG,BG).
```

`>> color(-FG,-BG)`

Returns the foreground color *FG* and background color *BG* indicated by the present settings of the Red, Green, and Blue `sliders`. The background color is calculated by calling the foreign routine `get_color/5`, and the foreground is calculated based on the background color using simple the heuristic "If the background color is greater than half its maximum value, the foreground is the 'minimum value', and if the background is less than half its maximum value then the foreground is the 'maximum value'.[13]

```
Self >> color(FG,BG) :-
        fetch_slot(red_slider,Red),
        fetch_slot(green_slider,Green),
        fetch_slot(blue_slider,Blue),
   % Get slider values:
        Red   >> value(R),
        Green >> value(G),
        Blue  >> value(B),
   % Get or allocate the color in the Colormap:
        fetch_slot(widget, W),
        get_color(W, R,G,B, BG),
   % Determine the foreground color by simple heuristic:
        Total is R + G + B,
        ( Total > 150 ->
          get_color(W,0,0,0,FG)
        ; get_color(W,100,100,100,FG)
        ).

:- end_class.
```

5.2.7 The `toplevel` Class

In order to use `ProXT` to create the interface, the application will have to do a little bit of work to initialize the Motif toolkit. For convenience and consistency, we will also wrap this in a class. While it would be possible to make this a subclass of `graphical_object`, the messages that it needs to be able to accept are different enough that this should stand on its own as a class.

Once again, the `widget` slot will store the widget created by `ProXT`.

```
:- class toplevel = [widget:pointer(widget)].
```

`<- create`

Create the toplevel XT widget by calling `xtInitialize/3` and then storing the widget returned in the widget slot.

```
Self <- create :-
        xtInitialize(demo, color, Top),
        Self << widget(Top).
```

`<- realize`

Realize the toplevel XT widget by looking up the widget from the widget slot and then calling `xtRealizeWidget/1`.

```
Self <- realize :-
        Self >> widget(Widget),
        xtRealizeWidget(Widget).
```

[13] The foreground is also computed using `get_color/5`. It is *not* possible to use the color returned by `get_color/5` for comparison since it is merely a colormap entry. Its value says nothing about how light or dark the color is.

`<- mainloop`

Start the main event processing loop of the application by calling `xtMainLoop/0`. This is wrapped in an exception handler which recursively sends the message `<- mainloop` to the toplevel instance.

```
Self <- mainloop :-
        on_exception(_E, xtMainLoop, (Self <- mainloop)).

:- end_class.
```

As mentioned earlier, it is necessary to define `ProXT` callback predicates which send a message to the (stored) destination instance.

This method has the disadvantage that because of the way that `ProXT` expands meta predicates, all such callback stubs must be added in this module (the module making the actual `xtAddCallbacks` call), rather than the module with the subclass calling `<- add_callback`.

A closer integration of `ProXT` and `classes` would provide a more seamless way of implementing callbacks as messages sent directly to the destination instance, but that is a task too involved for this example.

```
slider_changed(_,Self,_) :-
        Self <- slider_changed.

button_pressed(_,Self,_) :-
        Self <- button_pressed.
```

Now the basic `color_chooser` is complete. Loading `color_classes.pl` into Prolog, we can type:

```
| ?- use_module(library(classes)),
     create(toplevel, T),
     create(color_chooser(T), C),
     T <- realize,
     T <- mainloop.
```

A `color_chooser` comes up in a Motif toplevel shell. The `color_chooser` is internally consistent; when we move one of the `sliders`, the `frame` changes color appropriately. The `color_chooser` can be inserted into an application very easily, and can be used directly in a window such as a control panel, or put into a popup dialog as the programmer wishes.

Already several goals have been achieved. Several useful classes have been created, and more importantly, a model has been developed for building classes based on Motif widgets and `ProXT` which can be easily extended to additional widgets with minimal work thanks to inheritance. We have also found a way to call C code from our classes in a way which comes naturally in Prolog using the foreign interface.

This first foray has demonstrated the use of inheritance to abstract the operations which were common among `ProXT` widgets into the `graphical_object` class, providing a simple and consistent interface to these objects.

Even though we are starting to see some of the benefits of our approach, we still have more to do to complete the application! The next task is to build a simple application that uses this `color_chooser` to allow the user to change its background color. In order to keep things simple, the application will be merely a window containing a pushbutton that pops up the color chooser. Since the basic color chooser includes no means of accepting the color, we will add an OK button.

So the way we expect this application to be used is:

1. App window created
2. User presses button
3. Color Chooser comes up
4. User chooses color
5. User Presses OK
6. App button changes color

Steps 2-6 can be repeated, and it is not necessary for steps 4-5 to be performed before step 2 is repeated (there can be multiple color choosers up at once).

Now that we have an idea of what we want, we can look at the existing pieces to see what can be reused. The `color_chooser` already does everything it should except that it doesn't come up in its own window and have an 'OK' button. We can add these to the existing `color_chooser` by subclassing.

Clearly we will need a a pushbutton class (for the app button and the OK button) and a popup shell class (to put the `color_chooser` in). Both of these will work well as subclasses of `graphical_object`.

Finally, we might as well wrap the application in its own class, to make creation and startup as simple as possible.

5.2.8 The `pushbutton` Class

A subclass of `graphical_object` which wraps the `XmPushButton` widget class.

```
:- class pushbutton = graphical_object.
```

`<- create`(+*Parent*, +*Name*, +*Attribs*)

Create the `graphical\_object` using `xmPushButton` as the widget class

```
Self <- create(Parent, Name, Attribs) :-
        Self <- create_graphical_object(xmPushButtonWidgetClass,
                                        Parent, Name, Attribs).

:- end_class.
```

5.2.9 The `shell` Class

A subclass of `graphical_object` to wrap `XmShell` classes.

```
:- class shell = graphical_object.
```

`<- create`(+*Parent*, +*Name*, +*Attribs*)

Create the shell by sending message `<- create_shell`/4 with the `toplevelShellWidgetClass` shell class.

```
Self <- create(Parent, Name, Attribs) :-
        Self <- create_shell(topLevelShellWidgetClass, Parent, Name, Attribs).

:- uninherit graphical_object<-create_graphical_object/4.
```

Note that this subclass replaces `<- create_graphical_object` with its own creation method. First we use the `uninherit` directive to prevent inheriting the `<-create_graphical_object`/4 method. Then we define its replacement, `<- create_shell`/4.

```
:- uninherit graphical_object<-create_graphical_object/4.
```

`<- create_shell`(+*WidgetClass*, +*Parent*, +*Name*, +*Attribs*)

Create the shell using `xtCreatePopupShell`/5. Add a callback to handle the Window Manager close operation.

```
Self <- create_shell(ShellClass, Parent, Name, Attribs) :-
        Parent >> widget(ParentWidget),
        xtCreatePopupShell(Name, ShellClass, ParentWidget, Attribs, Widget),
        Self << widget(Widget),
        (   prolog_flag(version, '3.1') ->
                xtSetValues(Widget, [xmNdeleteResponse(2)]) % ProXT bug in 3.1
        ;   xtSetValues(Widget, [xmNdeleteResponse(xmDO_NOTHING)])
        ),
        xtDisplay(ParentWidget, Display),
        xmInternAtom(Display, 'WM_DELETE_WINDOW', false, Delete),
        xmAddWMProtocolCallback(Widget, Delete, wm_delete_window, Self).
```

`<- wm_delete_window`

By default this merely unmanages the shell (this can be overridden in subclasses).

```
Self <- wm_delete_window :-
        Self << managed(false).

:- end_class.
```

5.2.10 The popup_shell Class

This subclass of shell wraps the Motif Shell class `transientShell`. Transient shells differ from toplevel shells in their client decorations and in that they stay in front the parent shell, and iconify along with the parent shell.

```
:- class popup_shell = shell.
```

`<- create`(+*Parent*, +*Name*, +*Attribs*)

Create the shell by sending the message `<- create_shell`/4 with the shell class `toplevelShell`.

```
Self <- create(Parent, Name, Attribs) :-
        Self <- create_shell(transientShellWidgetClass, Parent, Name, Attribs).

:- end_class.
```

5.2.11 The popup_color_chooser Class

This subclass of `color_chooser` lets the `color_chooser` come up in its own window. The `shell` slot stores the shell instance in which the `color_chooser` has been placed.

```
:- class popup_color_chooser = color_chooser +[private shell:shell].
```

<- create(+*Parent*)

To create a popup_color_chooser, create a popup_shell and 'send super' passing the newly created popup_shell instance as the parent.

Adds an OK button, and sets the button's callback to send the message button_pressed to Self. Note that even though the slot type for the shell slot is 'shell', we can store an instance of popup_shell because classes will allow us to store instances of descendant classes in slots.

```
Self <- create(Parent) :-
        create(popup_shell(Parent,'Color Chooser', []), Shell),
        store_slot(shell,Shell),
        super  <- create(Shell),
        create(pushbutton(Self, 'OK', []), Button),
        Button <- add_callback(xmNactivateCallback, button_pressed, Self),
        Button << managed(true),
        Shell  << managed(true).
```

<- button_pressed

By default, the popup_color_chooser merely unmanages its shell when the OK button is pressed.

```
Self <- button_pressed :-
        fetch_slot(shell,Shell),
        Shell << managed(false).

:- end_class.
```

5.2.12 The app Class

We define the application as a class. This is a subclass of toplevel so that we inherit all the creation and startup methods for the ProXT toplevel. The application itself is simple – just a pushbutton which creates a color chooser when pressed.

```
:- class app = toplevel + [ private pushbutton:pushbutton].
```

<- create

In order to create the application, we first 'send super' to initialize ProXT from the <- create/0 method inherited from toplevel, and then create a pushbutton. We add a callback to the pushbutton to send us the message <- button_pressed/0 when it is pressed.

```
Self <- create :-
        super <- create,
        create(pushbutton(Self, 'Press for Color Chooser',
                          [xmNwidth(200), xmNheight(100)]), Button),
        store_slot(pushbutton, Button),
        Button <- add_callback(xmNactivateCallback, button_pressed, Self),
        Button << managed(true),
        Self   <- realize,
        Self   <- mainloop.
```

<- button_pressed

When the button is pressed, we simply create a new color_chooser.

```
Self <- button_pressed :-
        create(app_color_chooser(Self), _ColorChooser).
```

<< color($+FG$, $+BG$)

In order to set the color, we set the color of the pushbutton instance.

```
Self << color(FG,BG) :-
        fetch_slot(pushbutton, Button),
        Button << color(FG,BG).

:- end_class.
```

5.2.13 The app_color_chooser Class

This is the application specific class of color_chooser. It is based off the more general popup_color_chooser above, but this time it stores its parent in a slot, which it then uses when OK is pressed to send a message to its parent telling it to set the color calculated by the color_chooser.

```
:- class app_color_chooser = popup_color_chooser + [private parent:app].
```

<- create($+Parent$)

First store the parent instance *Parent* in the parent slot, then 'send super' to create the new popup_color_chooser.

```
Self <- create(Parent) :-
        store_slot(parent,Parent),
        super <- create(Parent).
```

<- button_pressed

Override the popup_color_chooser's <- button_pressed/0 method to look up the Parent and tell it to set its color to the color calculated by the color_chooser's >> color/2 method.

```
Self <- button_pressed :-
        fetch_slot(parent,Parent),
        Self   >> color(FG, BG),
        Parent << color(FG, BG),
        super <- button_pressed.

:- end_class.
```

As before, we define the callback procedure to send a message with the same functor to the object stored in the client data slot (the second argument).

```
wm_delete_window(_,Self,_) :-
        Self <- wm_delete_window.
```

In order to begin the application, we merely create the instance of class app. Its creation method takes care of all the rest!

```
user:runtime_entry(start) :-
        create(app, _C).
```

This example shows how easy it is to begin writing object oriented programs in Prolog using the classes package. Writing in object-oriented style helped us make good decisions about how to divide responsibility and encapsulate functionality in a way which led to more reusable code. Writing in Prolog allowed us to take a high-level approach to the problem at hand and resulted in fewer lines of code.

6 Conclusions

The `classes` package provides a way to write efficient object-oriented programs in Prolog. It has proved an invaluable development tool in the implementation of the Quintus WorkPro application. The experience gained by using the first version, `objects`, suggested several areas for improvement which are addressed in `classes`. With these improvements, `classes` provides an excellent way to add the benefits of object-oriented programming to Prolog applications.

References

[1] J. Almgren, et al. *The Sicstus Prolog Library Manual.* Swedish Institute of Computer Science. Kista, Sweden, 1993.

[2] G. Booch. *Object Oriented Design with Applications.* The Benjamin/Cummings Publishing Company, Inc. Readwood City, CA, 1991.

[3] M. Fromhertz. *OL(P) version 1.1.* Available for anonymous ftp from parcftp@parc.xerox.com (13.1.64.94) in /pub/ol. Xerox PARC, Palo Alto, 1993.

[4] F. McCabe. *Logic and Objects.* Prentice Hall International(UK) Ltd. Hemel Hempstead, UK, 1992.

[5] Quintus. *The Quintus Prolog Reference Manual.* Quintus Corporation. Palo Alto, CA, 1991.

[6] Quintus. *The Quintus ProWindows Reference Manual.* Quintus Corporation. Palo Alto, CA, 1991.

[7] Quintus. *The ProXT Reference Manual.* Quintus Corporation. Palo Alto, CA, 1991.

[8] Quintus. *Quintus ProTALK 1.0 User's Guide.* Library documentation supplement to [5] delivered with Quintus Prolog. Quintus Corporation. Palo Alto, CA, 1989.

[9] Quintus. *The Structs Package.* Library documentation supplement to [5] delivered with Quintus Prolog. Quintus Corporation. Palo Alto, CA, 1991.

[10] B. Rivers. *Quintec Objects.* Quintec Software, Ltd. Oxford, 1990.

[11] J. Rumbaugh, et al. *Object-Oriented Modeling and Design.* Prentice Hall, Inc. New Jersey, 1992.

[12] G. Steele, Jr. *Common Lisp: The Language.* Digital Press, USA, 1984.

[13] P. Vasey, et al. *Prolog++ Programming Reference Manual.* Logic Programming Associates. London, UK, 1990.