PRIMA - Perl toolkit for X, win32 & OS/2 PM.

Anton Berezin <tobez@plab.ku.dk> Dmitry Karasik <dk@plab.ku.dk>
Vadim Belman <voland@plab.ku.dk> Vladimir Berezin <berezin@plab.ku.dk>
Elisabeth Bock <bock@plab.ku.dk>

July 13, 1999

Abstract

PRIMA is a general purpose extensible graphical user interface toolkit with a rich set of standard widgets and an emphasis on 2D image processing tasks. A Perl program using PRIMA looks and behaves identically on X, Win32 and OS/2 PM.

All widgets and supplementary objects in PRIMA have a common root class `Prima::Object`. The explicit goal was to be able to freely intermix methods of PRIMA classes written in C with methods written in Perl. Due to this, PRIMA does not use the **h2xs** preprocessor. Instead, a special Perl script named **gencls** produces a C code and headers for thunks. Any method can be written in C and then overridden in Perl in a derived class, and vice versa.

A specification of a self-sufficient C-level API for graphics and windowing was developed and implemented to facilitate the development of the part of PRIMA code written in C in a platform-independent manner. This API is more than just the common subset of supported platforms APIs.

There is no concept of graphics contexts or presentation spaces in PRIMA: graphics is drawn directly on PRIMA widgets. In a way, any PRIMA widget by itself represents a graphics context, with attributes like a current drawing color and a logical raster operation attached.

Only a few basic classes have parts written in C. Standard widgets: buttons, input lines, checkboxes etc. are currently written exclusively in Perl.

PRIMA toolkit is under the BSD-style license (without clauses 3 and 4). It has been used extensively for implementing image processing applications at the Protein Laboratory, University of Copenhagen.

1 Introduction.

PRIMA is a general purpose extensible graphical user interface toolkit with a rich set of standard widgets and an emphasis on 2D image processing tasks. A Perl program using PRIMA looks and behaves identically on X, Win32 and OS/2 PM.

There are several toolkits for Perl: Perl/Tk, gtk, and X11-Motif to name a few. Why create yet another one? First, it was a great fun to develop it. Second, to rephrase TMTOWTDI, there should be more than one way to build GUI programs in Perl. Last, but not least, we believe that PRIMA provides some really convenient means to extend the set of widgets and widgets' functionality using pure Perl, without resorting to XS programming in most cases. In fact, every standard PRIMA widget, with a notable exception of generic `Prima::Widget` class and top-level `Prima::Window` class, is written exclusively in Perl.

2 Code examples & demonstrations.

2.1 A "Hello world!" program.

One of the simplest PRIMA programs looks like this:

```
#! /usr/bin/perl -w

```

```
3   use Prima qw(Buttons);
4   use Prima::Application name => 'Hello1';
5
6   my $w = Prima::Window-> create(
       text => 'Hello, world!',
7      onClose => sub { $application-> destroy },
8   );
9
10  $w-> insert( Button =>
11     text => 'E~xit',
12     onClick => sub {
13        $w-> close;
14     },
15     growMode => gm::Center,
16  );
17
18  run Prima;
```

While running, it creates a top-level window of an unspecified size, carrying "Hello, world!" on its title bar. In the center of the window there is a button titled "Exit" which does exactly that when pushed. The window can be resized, if your WM supports this. The button still remains in the window center.

The program is mostly self-explanatory. The statement in line 3 is mandatory: PRIMA core initializes itself during "`use Prima`". The basic classes are always loaded by this statement. One should ask `Prima` module to load the rest, like buttons support in this example. This is just a convenient shortcut to the explicit "`use Prima::Buttons;`" statement.

The next statement creates the parent object of all PRIMA widgets, and assigns a name 'Hello1' to it. The lowercased version of this name is used as an instance name for the X resource manager. The application instance itself is stored in the global variable `$::application`. This is the only global variable used by PRIMA core.

The lines from 6 to 8 create a top-level window. The constructor method for all PRIMA objects is called `create()`. It takes a list of property name/property value pairs as its parameters. In this case only two properties are set. The second one is an event handler for "`close`" notification, which is sent to the top-level window by a WM when the "`close`" button on the window frame is pushed.

There are more than 90 properties which can be set for a generic widget, and every class derived from a `Prima::Widget` adds more properties of its own. Normally it is not necessary to specify values for most of them, since they all have acceptable defaults.

The lines 10–16 add one button to the window. The `insert()` method is merely a wrapper for one or more `create()` constructors. In this case it is roughly equivalent to

```
Prima::Button-> create(
   owner => $w,
   text => 'E~xit', ...
);
```

Not only a top-level window can be an owner (parent) of a widget. In fact, any widget can own almost any other widget, allowing an arbitrary nesting level.

We'd like to draw your attention to the property `growMode`. Its concept was borrowed from Inprise's (former Borland) text-mode toolkit TurboVision, and roughly corresponds to X's gravity. Here the value `gm::Center` tells PRIMA to maintain the button centered relatively to its owner. `gm::Center` is a constant in a "`gm`" namespace. There are lots of similarly named constants in PRIMA, for example `cl::Red` and `kb::Escape`. Such naming convention contradicts *perlstyle(1)* on two points, but nevertheless is very convenient.

The class method `Prima::run` enters an event loop.

So far, so good. Let's extend our example with some nifty graphics:

```
#! /usr/bin/perl -w

use Prima qw(Buttons);
use constant PI => 4.0 * atan2 1, 1;
use constant D2R => PI / 180;
use constant Cos_120 => cos(D2R*(-120));
use constant Sin_120 => sin(D2R*(-120));

use Prima::Application name => 'Hello2';

my $w = Prima::Window-> create(
   text => 'Hello, world!',
   onClose => sub { $application-> destroy },
   onPaint => \&painter,
   lineWidth => 3,
   color => cl::Red,
);

$w-> insert( Button =>
   text => 'E~xit',
   onClick => sub {
      $w-> close;
```

```
22    },
23    growMode => gm::Center,
24 );
25
26 run Prima;
27
28 sub painter
29 {
30    my ($me,$canvas) = @_;
31    $me-> {angle} = 0
         unless exists $me-> {angle};
32    my @l = ();
33    my ($w, $h) = $canvas-> size;
34    my $sz = ($w > $h ? $h : $w) * 0.8 * 0.5;
35    my ($x, $y) = (25*$sz/34,$sz);
36    my ($fx, $fy) = ($x,$y);
37    ($w,$h) = ($w/2,$h/2);
38    my $angle = $me-> {angle};
39    for (1..3)
40    {
41       my $cos = cos(D2R*$angle);
42       my $sin = sin(D2R*$angle);
43       my @lines;
44       ($x,$y) = ($fx, $fy);
45       push @lines, $cos*$x-$sin*$y+$w,
                      $sin*$x+$cos*$y+$h;
46       $y = -$y;
47       push @lines, $cos*$x-$sin*$y+$w,
                      $sin*$x+$cos*$y+$h;
48       $x = Cos_120*$fx - Sin_120*$fy;
49       $y = Sin_120*$fx + Cos_120*$fy;
50       push @lines, $cos*$x-$sin*$y+$w,
                      $sin*$x+$cos*$y+$h;
51       $y = -$y/2;
52       push @lines, $cos*$x-$sin*$y+$w,
                      $sin*$x+$cos*$y+$h;
53       ($x, $y) = ($x-sqrt(3)*$y,0);
54       push @lines, $cos*$x-$sin*$y+$w,
                      $sin*$x+$cos*$y+$h;
55       push @l, \@lines;
56       $angle += 120;
57    }
58    my $fore = $me-> color;
59    $canvas-> color( $me-> backColor);
60    $canvas-> bar( 0, 0, $me-> get_size);
61    $canvas-> color( $fore);
62    $canvas-> polyline( shift @l);
63    $canvas-> polyline( shift @l);
64    $canvas-> polyline( shift @l);
65 }
```

This program draws the famous M. C. Escher's impossible triangle. Comparing to the "Hello, world!" program, the lines 4–8, 13–15 and 27–65 were added, and the line 9 was modified. There are few things to note in this program:

- there are no graphics contexts or canvas in PRIMA. Instead, one can paint on an object of any class derived from `Prima::Drawable`, including, of course, any widget object.
- the parameter `$canvas` (line 30) is the very same object as the parameter `$me` in a vast majority of cases. The distinction is only useful in the situations when the screenshot of some kind is required, or when trying to fool PRIMA in another non-trivial way.
- most of the properties which can be set as a name-value pair in a call to `create()`, can be also set individually by calling a method with the same name as the property (cp. lines 15 and 59).
- the paint-related properties set outside (lines 14 and 15) of an actual paint are persistent between the paints. The properties set inside the paint are not; if, for instance, we add a line `$canvas-> lineWidth(1)` after the line 64, it will not change anything at all.

All right. Let's make our little program a bit more dynamic.

```
 1 #! /usr/bin/perl -w
 2
 3 use Prima qw(Buttons);
 4 use constant PI => 4.0 * atan2 1, 1;
 5 use constant D2R => PI / 180;
 6 use constant Cos_120 => cos(D2R*(-120));
 7 use constant Sin_120 => sin(D2R*(-120));
 8
 9  use Prima::Application name => 'Hello3';
10
11 my $w = Prima::Window-> create(
      text => 'Hello, world!',
12    onClose => sub { $application-> destroy },
13    onPaint => \&painter,
14    lineWidth => 3,
15    color => cl::Red,
16  );
17
18 $w-> insert( Button =>
19     text => 'Exit',
20    onClick => sub {
21       $w-> close;
22    },
23    growMode => gm::Center,
```

```
24 );
25
26  $w-> {angle} = 0;
27
28  $w-> insert( Timer => timeout => 50,
29               onTick => sub {
30                             $w-> {angle} -= 5;
31                             $w-> repaint;
32                          },
33             )-> start;
34
35 run Prima;
36
37 sub painter
38 {
39    my ($me,$canvas) = @_;
40    $me-> {angle} = 0
          unless exists $me-> {angle};
41    my @l = ();
42    my ($w, $h) = $canvas-> size;
43    my $sz = ($w > $h ? $h : $w) * 0.8 * 0.5;
44    my ($x, $y) = (25*$sz/34,$sz);
45    my ($fx, $fy) = ($x,$y);
46    ($w,$h) = ($w/2,$h/2);
47    my $angle = $me-> {angle};
48    for (1..3)
49    {
50       my $cos = cos(D2R*$angle);
51       my $sin = sin(D2R*$angle);
52       my @lines;
53       ($x,$y) = ($fx, $fy);
54       push @lines, $cos*$x-$sin*$y+$w,
                      $sin*$x+$cos*$y+$h;
55       $y = -$y;
56       push @lines, $cos*$x-$sin*$y+$w,
                      $sin*$x+$cos*$y+$h;
57       $x = Cos_120*$fx - Sin_120*$fy;
58       $y = Sin_120*$fx + Cos_120*$fy;
59       push @lines, $cos*$x-$sin*$y+$w,
                      $sin*$x+$cos*$y+$h;
60       $y = -$y/2;
61       push @lines, $cos*$x-$sin*$y+$w,
                      $sin*$x+$cos*$y+$h;
62       ($x, $y) = ($x-sqrt(3)*$y,0);
63       push @lines, $cos*$x-$sin*$y+$w,
                      $sin*$x+$cos*$y+$h;
64       push @l, \@lines;
65       $angle += 120;
66    }
67    my $fore = $me-> color;
68    $canvas-> color( $me-> backColor);
69    $canvas-> bar( 0, 0, $me-> get_size);
70    $canvas-> color( $fore);
71    $canvas-> polyline( shift @l);
72    $canvas-> polyline( shift @l);
73    $canvas-> polyline( shift @l);
74 }
```

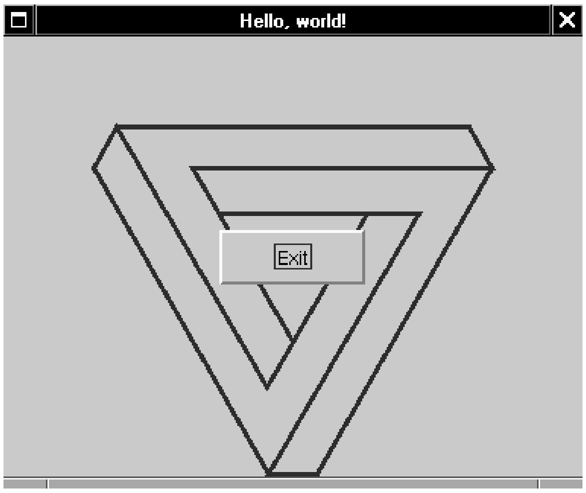

Figure 1: Screenshot of rotating Escher's triangle.

Comparing with the previous variant of the program, the line 9 was modified, and lines 25–33 were added.

This example shows that PRIMA has more than pure interface classes. We use here an instance of a `Prima::Timer` class to animate Escher's triangle. The timeout property specifies the timeout in milliseconds. The `insert()` method returns the reference to an object created, so we use that to start the timer. The `Prima::Widget`'s `repaint()` method forces a widget to update itself. One of the stages in triangle's rotation is shown in fig. 1.

2.2 The visual builder prototype.

To ease the development of the user interface of applications, the sort of a visual builder has been developed. The source code is too large for this paper, so we limited ourselves with a screenshot here (fig. 2). It is not very different from many applications with the same purpose, with the exception that it is written purely in Perl.

3 System description.

The main goal of PRIMA is to provide basic windowing and graphics services in a framework of Perl classes. Naturally, some parts of PRIMA are written in Perl, and other parts are written in C us-

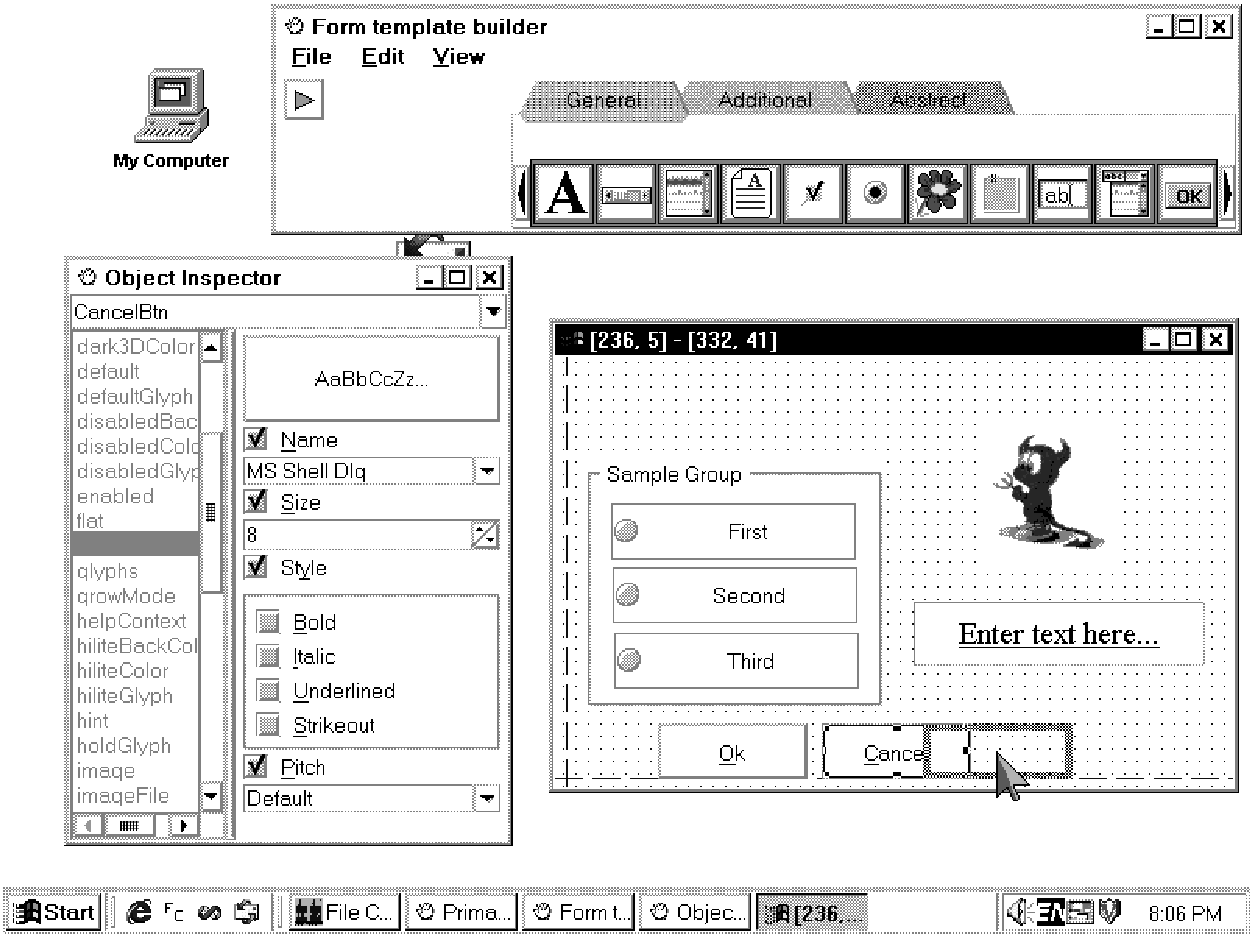

Figure 2: Screenshot of Visual Builder.

ing *perlguts(1)* API. Since PRIMA was planned to run on multiple platforms, the division of C code to the system-independent and system-dependent parts takes place.

PRIMA tries to make the fact that some of its classes' methods are written in Perl, and others in C as invisible as possible. For instance, when C method `init()` of a class `Prima::Widget` invokes method `set_text()`, there is no easy way to determine whether it is calling the original C `set_text()` method, or is overridden in the C method of a subclass `Prima::Window`, or is overridden in the Perl method of a subclass `Prima::Button`.

The first thing to achieve this fluency in the overriding of virtual methods in either language was to associate Perl objects with a C structure. There is nothing new here, many XS extensions do the same. Originally, a reserved key was used in a hash representing an object to hold a stringified version of a C pointer. Currently, the user type '~' magic is used instead. C-part knows its Perl "mate" by storing an SV pointer directly.

The second thing was to pretend that C is an object-oriented language. As a marginal note, other candidates for the low-level implementation language where not considered because of our personal preferences. Every PRIMA class is represented as a C structure that contains data fields and several auxiliary fields. The most important auxiliary field is a pointer to a virtual methods table (*VMT* for short) structure, which holds method pointers and some run-time typing information. Of course, the declaration of the VMT is specific for every class that has implemented in C non-overridden methods. The `gencls` script, described below, generates the VMT for classes which have C methods (core classes for short). Thus, the invocation of a method from C normally looks like

```
CWidget( self)-> set_text( self, "E~xit");
```

where `CWidget` is a convenience macro. Without it, the real call is more like

```
((struct _Widget_vmt*)
   ((struct _Widget*)self)->self)->
      set_text( self, "E~xit");
```

Most commonly, every C core class method is implemented as three C functions:

- `Classname_methodname()` does the real job and is suitable for calling from C.
- `Classname_methodname_FROMPERL()` is a generated by `gencls` XS function, which verifies the number and the type of Perl parameters passed to it, converts these parameters into appropriate C values, calls its `Classname_methodname()` C counterpart and finally converts and puts on the Perl stack the result. It is customary call such stub routines "thunks".
- `Classname_methodname_REDEFINED()` is also a generated thunk with the same parameters and return value as the original `Classname_methodname()`. This is used for calling in C methods overridden in Perl.

At the PRIMA startup all thunks of C methods of core classes are registered by calling `newXS`, so that C methods become available to Perl. This registration is also partly automated with `gencls` script.

Things become interesting when we define a pure Perl class inherited directly or indirectly from one of the core classes. From Perl's side, this is not a problem. It is a bit more complicated in C. Using our `set_text()` example, we need a way to correctly call `Prima::Button::set_text()` from C dealing with an instance of a button. To cut the long story short, this is achieved as follows (fig. 3).

Every preinitialized (generated with `gencls`) VMT structure has a pointer to initialized array (patch table) called `ClassnameVmtPatch`. When the very first instance of such a pure Perl class is being created, the internal function `gimme_the_vmt()` is called. The function maintains a hash of known VMTs and is able to determine that it has to create a new VMT. By traversing `@ISA` it finds a predecessor with an already existing VMT. `gimme_the_vmt()` then allocates space for a new VMT, copies the predecessor's VMT to it, and scans the patch table, finding methods which were overridden in Perl and replacing corresponding entries in the new VMT with an appropriate `Classname_methodname_REDEFINED()` address.

The approach described above has its limitations. First, it is not easy to call from C a completely new (not overridden) method of a pure Perl class. Second, it is not possible to define a class which has C methods and is inherited from purely Perl-implemented superclass. It is possible, though, to circumvent the latter by creating a *Classname.cls* file for the superclass (see `gencls` description below) and make this class known to the core by registering it.

3.1 `gencls` script.

Because of the complexity involved in managing PRIMA classes, as well as great uniformity of PRIMA methods declarations, the use of conventional `xsubpp` seems inappropriate. The `gencls` script has been written to automate thunks and VMTs generation.

Every PRIMA core class has a file with a *cls* extension, which describes the C fields and methods of the class. For instance:

```
object Menu( AbstractMenu)
{
   method Bool  get_selected();
   method void  set_selected( Bool selected);
   c_only void  update_sys_handle( HV * profile);
}
```

Running `gencls` script on this file will produce two other files, *Menu.h* and *Menu.inc*, containing the definitions of VMT and `VmtPatch` for the `Prima::Menu` class, the declarations of `Menu_methodname` functions and declarations and definitions of `Menu_methodname_FROMPERL` and `Menu_methodname_REDEFINED` functions, together with a couple of auxiliary functions for registering the `Prima::Menu` class with the PRIMA core.

3.2 System-dependent part.

From the very beginning we decided to make the implementation of the core classes completely platform-independent. To facilitate this, an additional layer of C code was created to hide the specificities of a particular windowing system. This layer implements the gory details of creating windows, requesting device or graphics contexts, allocating colors and managing pens, brushes, and the like.

Ideally, the system-dependent API should be completely unaware of the fact that PRIMA itself is just a collection of Perl modules. This requirement was mostly met. However, the existing system-dependent API closely resembles the organization of PRIMA core classes. This is neccesary partly because, in all supported platforms, the top-level windows are very different from normal widgets: they are created in a different way, they have external decorations which are under control of a window manager rather than an application program, etc. Hence the separate low-level functions for creating generic widgets and top-level windows.

All platform-dependent functions have `apc_` prefix in their names. *APC* stands for *apricot*, the name of the master include file in PRIMA. It is difficult to say now what "apricot" itself stands for, but "pri' definitely stands for PRIMA.

The vast majority of *apc* functions take as a first parameter the handle (in reality just a pointer to) of a PRIMA object. *Apc* functions are allowed to use C-level fields of objects and even call methods, though this practice is frowned upon because of the possible side effects and a deep recursion problem.

A great effort was made to achieve the unification of the API. On the whole the result differs from every supported platform, with different parts of APC resembling different systems. For example, APC borrowed bottom-left origin from OS/2, bitmapped masks to control shapes of widgets came from X, and colorful pointers are Win32 heritage.

To control the size of the system-dependent parts of PRIMA, the standard widgets provided with some of the supported systems were not used. With the exception of the general window management and graphics, only a few other features are implemented at the low level. They are: clipboard/cut buffers support, pull-down and popup menus, timers support and printing support (not available in X version at the moment).

3.3 Images support in core.

PRIMA has a rich set of operations and transformations applicable to 2D images. A `Prima::Image` class has the same parent class, `Prima::Drawable`, as a generic widget, allowing the same set of drawing operations. This is supported on images of arbitrary depth regardless of the current screen depth. On the other hand, PRIMA does not implement any drawing operations by itself, using the underlying system on-pixmap drawing mechanisms instead. This means that images are first transformed into a form most appropriate for drawing, and then, transparently for the user, transformed back. Thus, a certain reduction of the image quality is possible.

PRIMA can load and save images using many file formats. The exact list PRIMA core supports is defined at compile time, though a dynamic linking of the image file format drivers is supported.

Images can also be scaled and resampled.

3.4 A set of standard widgets: hierarchy of classes.

The hierarchy of PRIMA core classes is presented on fig. 4 (`Prima::` prefix is removed from all classes names to save space). The root class, `Prima::Object`, provides some basic construction, initialization, cleanup and destruction methods for all PRIMA classes. It has no other function by itself. `Prima::Component` adds event and notification processing, as well as the ability to own other objects and be owned by them. All other PRIMA classes are inherited, directly or not, from `Prima::Component`. Then there are supporting non-widgets classes: `Prima::Clipboard`, `Prima::Timer` and `Prima::AbstractMenu`. `Prima::AccelTable`, `Prima::Menu` and `Prima::Popup` are inherited from `Prima::AbstractMenu`. `Prima::Drawable` is an abstract class with a complete set of drawing methods. `Prima::Image` and `Prima::Widget` are both drawables. `Prima::Widget` is the most important class in PRIMA hierarchy; all standard widgets are its descendants. The only core descendants of `Prima::Widget` are `Prima::Application` and `Prima::Window`, the latter implements top-level windows.

PRIMA standard distribution includes many Perl classes which implement a set of widgets expected to be found in any windowing toolkit: buttons, checkboxes and radio buttons, input fields, list boxes etc. The complete list of currently implemented widgets (see fig. 5) is too large to describe in this text; this set is a moving target, and new widgets are routinely added to PRIMA.

3.5 A set of standard dialogs.

The important part of any user interface toolkit is a mechanism that makes it easy to create such common interface facilities as message boxes, file open/save dialogs, choose directory dialogs, choose color dialogs etc. PRIMA provides several of those, though we would like to extend the list in the future.

A screenshot of a standard "Choose Color" dialog is on the fig. 6.

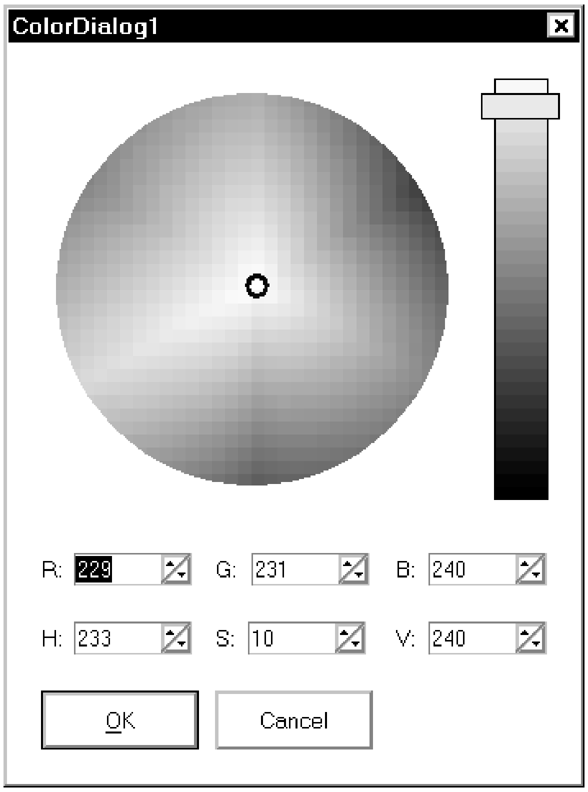

Figure 6: Standard color dialog.

3.6 Properties and default values.

Although PRIMA classes have large number of properties, it is not necessary to specify values of most of them — the defaults will do. But sometimes an inherited class needs to override the default values of its superclass. To further complicate things, there is a certain overlapping in the meaning of properties. For example, a size and a position of a widget can be specified in several ways:

```
rect => [ 10, 10, 100, 100]  or
size => [ 90, 90], pos => [ 10, 10] or
left => 10, right => 100, bottom => 100, top = 100
```

PRIMA manages these details by providing two methods with most of the classes: `profile_default` and `profile_check_in`. `profile_default` should return an anonymous hash with default values for <u>all</u> properties of a given class. When the class adds some new properties and/or wishes to supply different default values for some other properties of its parent class, it overrides `profile_default` method. During object creation, the programmer-supplied properties are merged with properties returned by `profile_default`. But before this step, `profile_check_in` is invoked. Its job is to remove any ambiguity resulting from overlapping in the meaning of properties mentioned above, and make other transformation of properties values as necessary. This mechanism is flexible enough even to handle *system-defined* default values, like those queried from the X resource manager.

Properties of objects can also be set (and queried) during object's lifetime. PRIMA supports three ways of setting properties after object creation. First, properties can be bulk-set by calling method `set`. Just like a constructor `create`, `set` takes property name/property value pairs. Second, every property has a correspondent method `set_propertyname`. Third, as a convenient shortcut, a method with the same name as a property can be used. Such methods, when called without any parameters, are also good for *getting* properties values. Another way of getting a value of a property is to call a get_propertyname method. There is no bulk-get method.

The most complex property PRIMA classes deal with is a *font*. It is a hash (more precisely, a hash reference) with several allowed key names like name, size and height. During object creation, a Prima::Drawable::profile_check_in takes care of merging the default font hash with specified one. Convenience method font returns an instance of stand-alone class Prima::Font, so that things like

```
$btn-> set_font(
   { %{ $btn-> get_font }, size => 18}
);
```

become just

```
$btn-> font-> size( 18);
```

The name property of any Prima::Component serves two purposes. First, it determines an instance name of an object to do resource manager database lookups. Second, it provides an easy way to refer to child components: if an unknown method of a component is called, Prima::Component::AUTOLOAD tries to find a child component with the name of a method. This makes possible constructions like this:

```
$w-> OKButton-> enabled( 0);
```

Of course, it is also possible to get a list of all children.

3.7 Construction/initialization and destruction/cleaning up.

PRIMA makes a distinction between construction and initialization. The only constructor in standard PRIMA class hierarchy is Prima::Object::create. It calls several auxiliary XS functions which set the links between Perl and C parts of an object properly. At the end, the initialization method init is called. Normally init is overridden in every PRIMA class. For a normal PRIMA operation it is absolutely necessary for an object's init to call its SUPER::init, preferably near init's beginning. init takes a list of property name/property value pairs as parameters, and has to return this *profile* to a caller after possible modifications. The properties in a profile are already merged with the default profile and validated by profile_check_in at the time init is called.

The distinction between destruction and cleaning up is more important, because the destruction can be initiated from three places: in Perl, in C, and when a platform specific window object corresponding to the PRIMA widget ceases to exist. The clean-up method is called done. PRIMA guarantees that it will be invoked exactly once, whatever "destruction path" an object will follow. Under normal conditions, a PRIMA object is not destroyed when Perl variable holding a reference to it goes out of scope or gets assigned to. This happens because an owner of the objects holds references to all its children. On the other hand, if an object is "half-dead", i.e. its method done was already called, it is still possible that the object is referenced by one or more Perl variables. Although the object cannot do much being in this state, PRIMA respects Perl rules and DESTROY will not be called prematurely.

3.8 Extensibility.

The system is obviously extensible. First, since all standard widgets are written in Perl, nothing prevents a programmer from adding new widgets with arbitrarily complex behavior. Second, PRIMA quite naturally supports extension modules written in C/XS, because Perl5 itself supports them. However, it is not easy to write a C extension module which plugs itself into PRIMA class hierarchy, with all the fluency of PRIMA object manipulating and the convenience of the method overriding.

To facilitate these tasks, gencls script can generate different class registration routines for add-on modules. It is a responsibility of the module's author to call the registration routine during normal dynamic XS module initialization. gencls also supports generation of thunks for normal functions outside PRIMA class framework, much like SWIG does, though in more limited way. We exploited these possibilities to create a set of image processing modules, which are used now at the Protein Laboratory.

4 Current state of the project.

PRIMA is under copyright of the Protein Laboratory, University of Copenhagen, and has a BSD-style license with clauses 3 and 4 removed.

Although OS/2 was the platform on which PRIMA was born, currently the most completed code exists for Win32. The X system-specific code is under very active development at the moment, and we hope to make it as good as the Win32 version before Fall 1999. OS/2 Presentation Manager-specific code is usable but outdated since much of the APC API has been changed during the last half year.

You should be able to download the latest version of PRIMA from **http://www.prima.eu.org/**. Currently PRIMA is not on the CPAN, but we hope to put it there soon.

4.1 Code sizes and memory consumption.

PRIMA is quite big, so you'd better have a machine with enough memory. 32 MB of RAM is highly recommended for a single-user workstation running FreeBSD, Linux or Windows NT.

Presently, the system-independent portion of manually-written C code has a size of 356747 bytes (12695 lines). The size of Win32-specific code is 278239 bytes (9583 lines), the size of Unix-specific code is 191362 bytes (6910 lines). The total size of all manually written C headers is 137599 bytes (4866 lines). The size of the Perl code implementing PRIMA classes is 616204 bytes, or 19742 lines. Autogenerated code for thunks takes 581506 bytes, or 17676 lines.

The minimal "Hello, world!" program using a non-optimized debugging version of PRIMA core occupies 5576 kilobytes on Windows NT 4.0 (according to Task Manager), and 7076 kilobytes on FreeBSD 4.0 (according to `ps(1)`). Non-trivial programs, naturally, take more than that. The exact figures are probably of no interest since the sizes vary widely for different classes of applications. For image processing tasks involving operations with 768x576 8-bit grayscale images, it is not uncommon to reach 32MB and more.

It is difficult to measure a speed of a graphical toolkit objectively. PRIMA programs are usually slow at startup because of loading of large dynamically linked libraries and compilation of the large amounts of Perl code. Once started, however, it is quite fast. One of us runs PRIMA programs on his old 486 home machine with 32 MB of RAM, and has not noticed any unresponsiveness.

5 Conclusions.

We found PRIMA to be a convenient and easy to use tool for creating graphical applications for our users at the Protein Laboratory. We hope that other Perl hackers will find it useful, too.

PRIMA is still under development. There are many things which need improvements. The most important tasks are to complete the implementation of the X version, to document PRIMA, and to add more widgets.

The development of PRIMA is not closed. We need your ideas, your patches, and useful widgets to make it better. There is a mailing list dedicated to PRIMA, `prima@prima.eu.org`. To subscribe to the list, send mail to <`majordomo@prima.eu.org`> and include `subscribe prima <optional address>` in the body of your message.

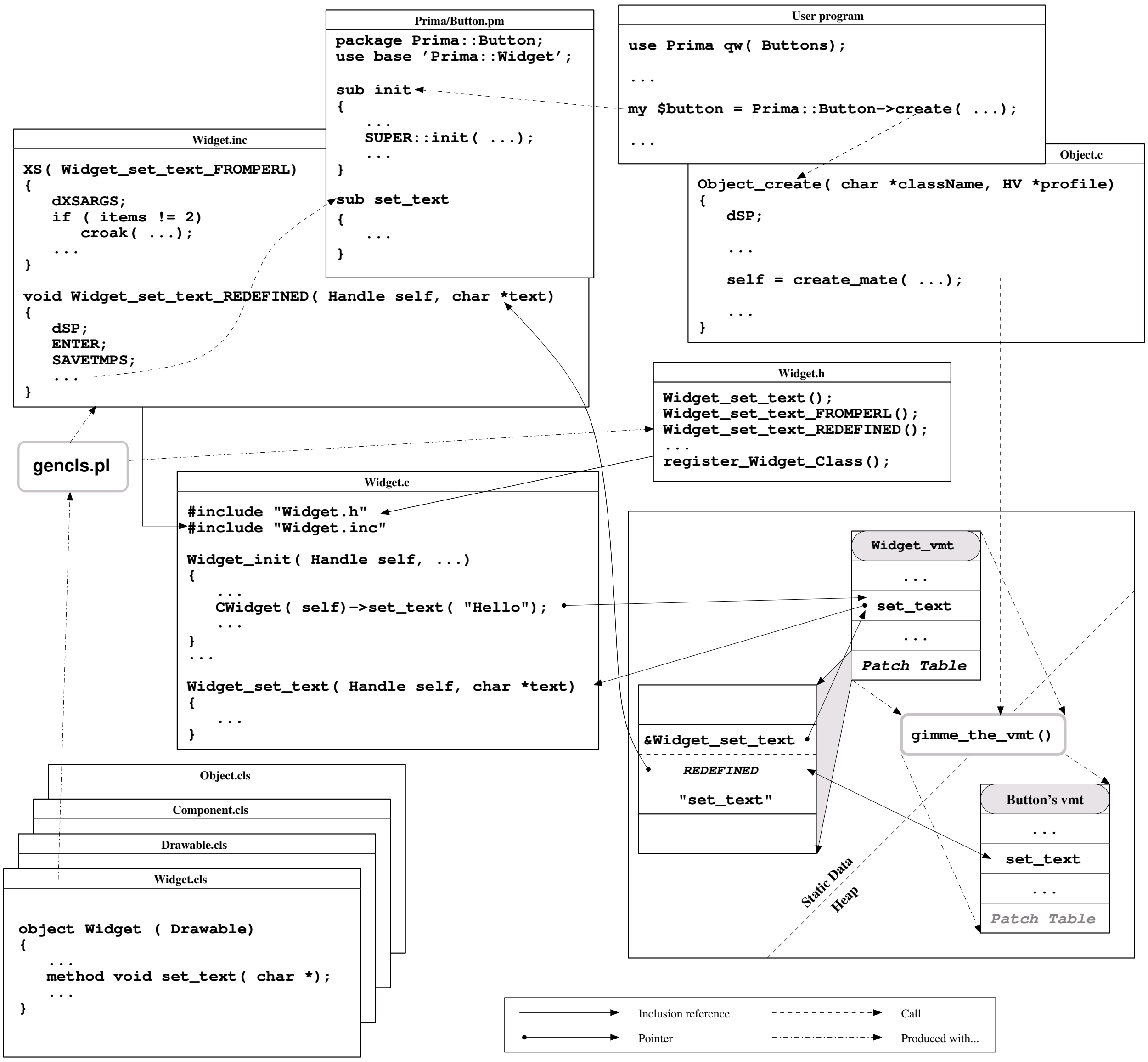

Figure 3: Virtual methods and VMT patching.

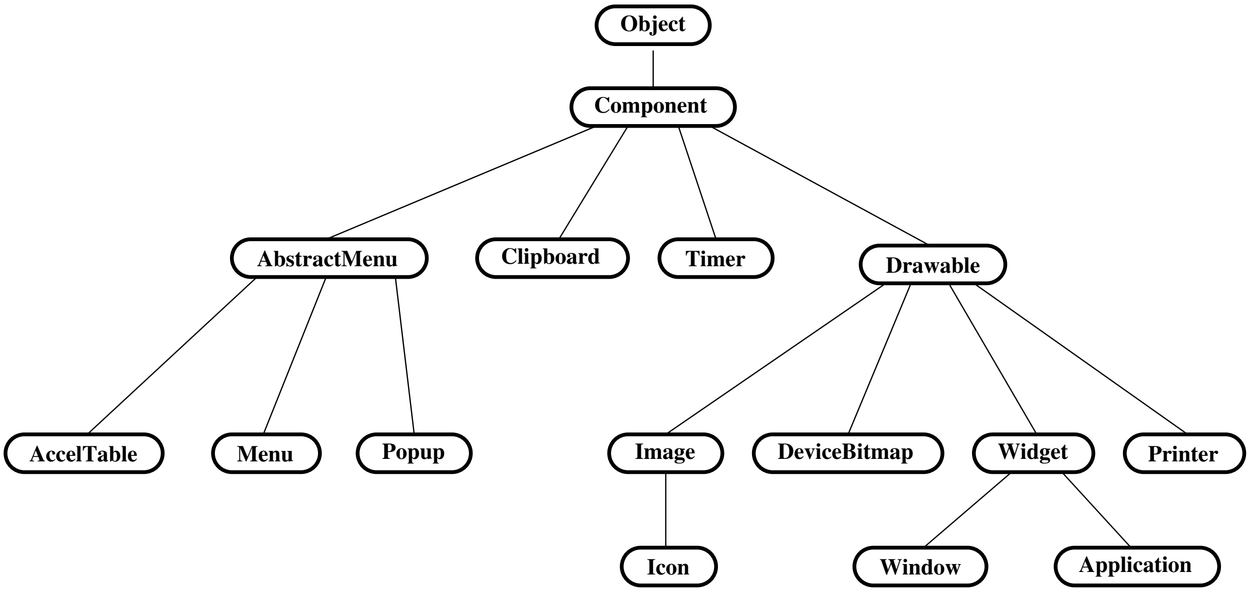

Figure 4: Prima core classes.

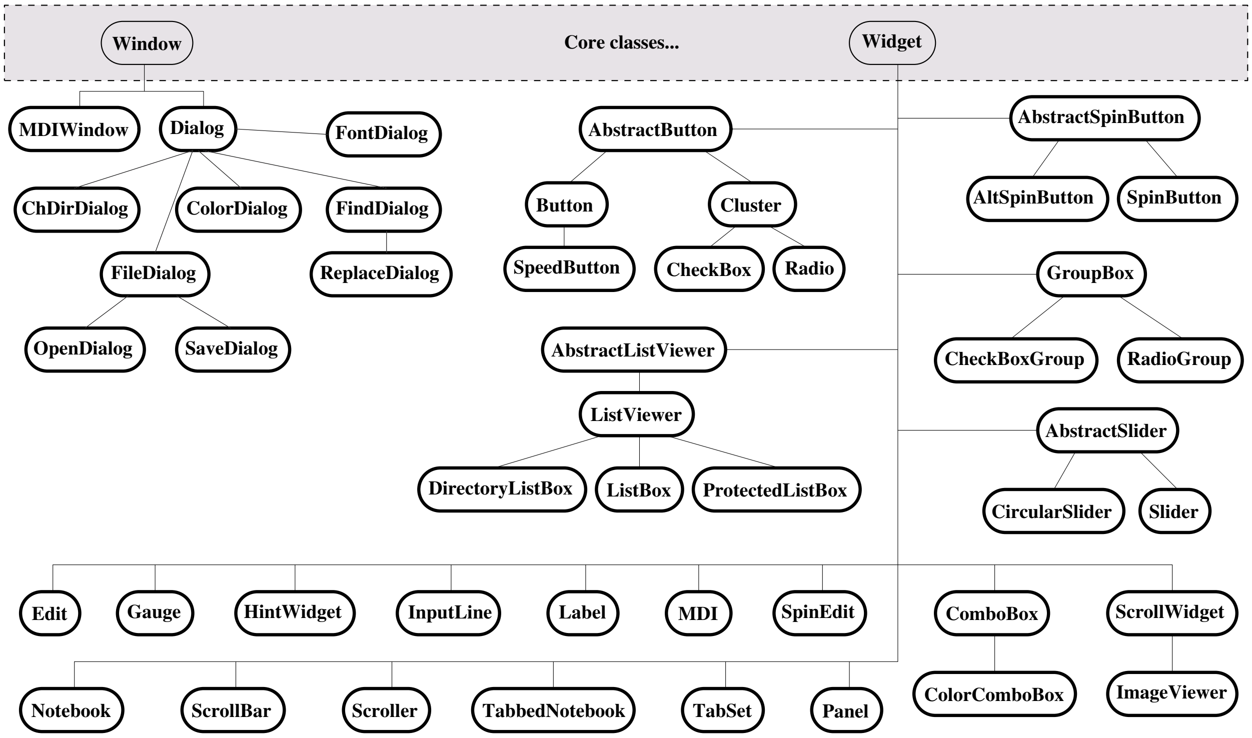

Figure 5: Prima widgets set.