# Chapter 1

# Calling Foreign Code with Spock

Interacting with the external world is a mandatory element of modern systems. Been able to call libraries is often a key success to business. In this chapter, we will explain you how to do that in Pharo using Foreign Function Interface.

Suppose you want to use OpenGL in Pharo. The solution is to make a *binding* of each C library function to a Smalltalk method, and then to call these methods from Smalltalk code. But we should answer several questions: how does a binding call a C function? How do you write such binding? How do we pass heap allocated objects? The libraries and mechanisms in charge of doing this kind of work are usually called FFI, for Foreign Function Interface. In this chapter we will help you to use this interface by exposing some practical examples of how to solve common problems faced when communicating with other languages.

Just to mention, there is another possible approach that would be to invoke C libraries by defining a specific plugin for the virtual machine, but it's more complex and it offers no important advantage compared to the results of using bindings in the same language and with the same constructions you are used to.

We will start to show you how to use Spock , a new foreign function library. Spock is based on NativeBoost and previous FFI libraries. `Luc` ▶*Alien and FFI. cite them here no?*◀

## 1.1   Getting Started

**Luc**  ▶ *Before the section "calling Ext func", I think we should first introduce the basics (part of the old "What is a FFI and how it works?" section). Because the reader will want to test while reading. And for that he probably will need a special VM (NativeBoost VM), an image with NativeBoost inside, ...* ◀

Right now to run Spock, the new FFI library described in this chapter, you will need a vm that has the NativeBoost plugin and you can find one at https://ci.lille.inria.fr/pharo/job/NB-Cog-Mac-Carbon/.

Then you should load in your image, the package NativeBoost−Installer that you can load as follows:

```
Gofer it
    squeaksource: 'NativeBoost';
    package: 'NativeBoost−Installer';
    load.
```

Once loaded you should execute the following expression: NBInstaller install.

## 1.2   Calling a Simple External Function

Suppose you want to know the amount of time the image has been running by calling the underlying OS function named clock. This function is part of the standard C library. Its C declaration is[1]:

```
uint clock (void)
```

To call clock from the image, you should write a binding: a normal Smalltalk method annotated with the <primitive:module:> pragma. This pragma specifies that a native call should be performed using the Native-Boost plugin. The external function call is described using the message nbCall:module: message[2]. We will explain the full mechanism that lies behind Spock in the last sections, after explaining how to use it. For now, here is the full definition of the binding. We explain the details below.

```
CExamples class>>ticksSinceStart
  <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

  ↑ self nbCall: #( uint clock () ) module: NativeBoost CLibrary
```

---

[1]According to man its return type is clock_t instead of uint, but we deliberately made it uint for the sake of simplicity. We will see how to deal with typedefs in the following sections

[2]Note, that this is not the only message which you can use for defining a callout. There are others, which we will discuss later, but for now use this form for introduction

In the above example, we defined a class named CExamples and defined the method on its class side. This way it is easy to access it by sending a message to this class without creating unnecessary instances. Of course, there is no limitations, about in what class (be it instance side or class side) where you can define an external function call.

Then, to know the time since the program started you can execute and print it. Note that these are just native clock ticks, which aren't of much use.

Script 1.1: *Invoking a C binding*

```
CExamples ticksSinceStart
```

## Analysis of an FFI Callout

Now that the call worked, let us look at the definition again:

```
CExamples class>>ticksSinceStart
  <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

  ↑ self nbCall: #( int clock () ) module: NativeBoost CLibrary
```

**Callout.** What we have just done is usually named an FFI *callout*. The #nbCall:module: is a method provided by NativeBoost package and defined for Object class (so you can use it anywhere). It is responsible for generating an FFI callout. Remember that calling a C function requires to marshall all arguments from Smalltalk world to C one, push them to the C stack, perform a call to the external function, and finally convert the return value from C to Smalltalk again. #nbCall:module: handles this using the description of the function you provide.

Note, that the given message #nbCall:module: should be sent only by methods that define the callout, because it does some context handling behind the scenes.

This message has two arguments: the *signature of the function* that will be invoked and the *module or library* where to look for it. In this example we look for clock() function in the standard C library.

The signature is described by an array: here #(int clock ()).

- Its first element is the C return type, here int.

- Its second is the name of the called function, here clock.

- Its third element is also an array which describes the function parameters if there's any.

Basically, if you strip down outer #(), what is inside is a C function proto-type, which is very close to normal C syntax. This is intentionally done so, that in most cases you can actually copy-and-paste a complete C function declaration, taken from header file or from documentation, and it is ready for use.

The second argument, #module:, is where you should specify a module name or its handle, where to look up for given function. NativeBoost provides a convenience method named CLibrary to obtain a handle of the standard C library. In other cases (suppose you want to use some standalone library), you must specify either a library name, or full path to it, or handle to already loaded library. You can use NativeBoost methods to load an external library (and hence obtain handle of it) before making any calls to it. For example: NativeBoost forCurrentPlatform loadModule: 'nameOfModule' loads the module 'nameOfModule'[3].

**About marshaling.**  The return type is int and not SmallInteger. This is the same as with function arguments. They are all described in terms of C language. In general you don't have to worry too much about this, because the Spock library knows how to map standard C values to Smalltalk objects and viceversa. So, when this message is executed, the return value will be turned into a SmallInteger. Also, you can define new mappings for C types defined elsewhere, for example when wrapping libraries that define new C types, we will show that later. This conversion process for types from different languages is called *marshalling*. We will see more examples of automatic conversions in this Chapter.

## Presentation Conventions

There are always three things to take into account when doing callouts: a C function, a specification of the binding between C and Smalltalk, and location for such declaration at the Smalltalk level.

1. First, of course, is the signature of the C function we will call, this definition is the definition of the function from a C point of view. Through this book we describe it in a C header box as follows:

   **C header.**

   ```
   int clock (void)
   ```

---

[3]Module names are platform dependent, so for example to load OpenGL library you may want to use 'OpenGL32' on windows and 'libGL.so' or even '/usr/lib/libGL.so' on linux

2. Second, how do we communicate between the C and Smalltalk worlds. That's the *Primitive pragma*, which instructs a VM to actually "do the thing", when you run the corresponding smalltalk method. Also, note, that there is no hidden complexity behind this primitive. It does only one thing: runs a generated machine code, while the code generation logic are found in image and ready for study and modification.

**Pragma declaration.**

```
<primitive: #primitiveNativeCall module: #NativeBoostPlugin>
```

3. Third, we have to see how we organize the C calls from the Smalltalk side. Often we create a class that groups callout definitions. Such a class can then be used as an entry point to perform callouts.

**Callout grouping.**

*CExamples ticksSinceStart*

# 1.3   Passing arguments to a function

The previous example was the simplest we could find: we asked for the execution of a function without parameters and got the results. Now let's look how we can execute functions that take arguments.

## Passing an integer

Let's try with a really simple function, abs(), which takes an integer and returns its absolute value.

**C header.**

```
int abs ( int n );
```

**Smalltalk binding.**

```
CExamples class>>abs: anInteger
    <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

    ↑ self nbCall: #( int abs (int anInteger) ) module: NativeBoost CLibrary
```

Compared to the previous example, we changed the name of the function and added an argument. When functions have arguments you have to specify two things for each of them: their type and the object that you want to be sent to the C function. That is, in the arguments array, we put the type of the argument, and after that, the *name of the Smalltalk variable* we pass as argument.

Here anInteger in #(int abs (int anInteger) means that the variable is bound to the abs: method parameter and will be converted to C int, when executing a call.

This type-and-name pairs will be repeated, separated by comma for each argument, as we will show in the next examples.

Now you can try printing this:

🌐  *CExamples abs: -20.*

**About arguments**

In the callout code/declaration, we are expressing not one but *two* different aspects: the obvious one is the C function signature, the other is which objects we are passing as arguments to the C function when the method is invoked. In this second aspect there are many possibilities. In our example the argument of the C function is the method argument: anInteger. But it is not always necessary the case.

You can also use some constants, and in that case it's not always necessary to specify the type of the argument. This is because NativeBoost automatically uses them as C 'int's: nil and false are converted to 0, and true to 1. Numbers are converted to their respective value, and can be positive and negative. As the type for these constants is int, there is no need to use: int abs (int −45) when you can use int abs (−45).

**About constants**

Often some functions in C libraries taking an options or flags, and have a number of them declared using #define in C header. You can, of course take these constant values from header and put then into your callout. But it is preferable to use a symbolic names for constants, which much less confusing than just bare numbers. For using symbolic constants you can create a class variable or variable in shared pool, and then use the constant name as argument in your callout.

For example, imagine that we always pass a MyConstant value to our function. For this we can define it as a class variable in our class:

```
Object subclass: #MyClass
  ...
  classVariables: 'MyConstant'
..
```

Then don't forget to initialize it properly:

```
MyClass>>initialize
  MyConstant := −45.
```

And so, in callout code, we can use it like following:

```
MyClass class>>absMinusFortyFive
  <primitive: #primitiveNativeCall module: #NativeBoostPlugin>
  ↑ NBFFICallout cdecl: #( int abs (MyConstant) ) module: NativeBoost CLibrary
```

You can also put constants, self or any instance variable as arguments to a C call. Suppose you want to add this abs wrapper (let us call it absoluteValue) to the class SmallInteger, so that we can execute −50 absoluteValue.

In that case we simply add the absoluteValue method to SmallInteger, and we directly pass self as illustrated below.

```
SmallInteger>>absoluteValue
  <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

  ↑ NBFFICallout cdecl: #( int abs (int self) ) module: NativeBoost CLibrary
```

Imagine that we want to have a wrapper that always calls the absolute function with the number -45. Then we directly define it as follows:

```
CExamples class>>absMinusFortyFive
  <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

  ↑ NBFFICallout cdecl: #( int abs (−45) ) module: NativeBoost CLibrary
```

Note that this time we don't put the type of the argument. It is is also possible to pass an instance variable, but we let you do it as an exercise.

## Passing strings

As you may know strings in C are sequences of characters terminated with a special character \0. It is then interesting to see how Spock deals with them since they are an important data structure in C. For this, we will call the very well known strlen function. This function requires a string as argument and returns its number of characters.

**C header.**

```
int strlen ( const char * str );
```

**Pragma declaration.**

```
CExamples class>>stringLength: aString
    <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

    ↑ NBFFICallout cdecl: #( int strlen (String aString) ) module: NativeBoost
        CLibrary
```

**Callout grouping.**

> CExamples stringLength: 'awesome!'

**Example analysis.**   You may have noticed that the callout description is not exactly the same as the C function header.

In the signature #( int strlen (String aString) ) there are two differences with the C signature.

- The first difference is the const keyword of the argument. For those not used to C, that's only a modifier keyword that the compiler takes into account to make some static validations at compile time. It's has no value when describing the signature for calling a function at runtime.

- The second difference, an important one, is that specify that the argument is String aString instead of char * aString. With String aString, Spock will automatic do the arguments conversion from Smalltalk strings to C strings (null terminated). Therefore it is important to use String and not char *.

  In the example, the string passed will be put in an external C char array and a null termination character will be added to it. Also, this array will be automatically released after the call ends. Using (String aString) is equivalent to (someString copyWith: (Character value:0) as in CExamples stringLength: (someString copyWith: (Character value:0). Conversely, Spock will take the C result value of calling the C function and convert it to a proper Smalltalk object, an Integer in this particular case.

**Passing two strings**

To continue with the examples we will call strcmp, which takes two arguments and returns -1, 0 or 1 depending on the relationship between both strings.

**C header.**

```
int strcmp ( const char * str1, const char * str2 );
```

**Pragma declaration.**

```
CExamples class>>stringCompare: aString with: anotherString
    <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

    ↑ NBFFICallout cdecl: #( int strcmp (String aString, String anotherString) )
        module: NativeBoost CLibrary
```

**Callout grouping.**

> *CExamples stringCompare: 'awesome!' with: 'awesome'*

Notice that you can add arguments by appending them to the arguments array, using a comma to separate them. Also notice that you have to explicitly tell which object is going to be sent for each argument, as already told. In this case, aString is the first one and anotherString is the second.

## Handling Heap-based Objects: Passing an Array

So far we only used objects that were taken from the Smalltalk heap and pushed into the C stack (doing the corresponding marshalling). That means that until now we only passed copies of the arguments, or that we passed them *by-value*. With the next example we will show how Spock supports to pass objects *by-reference*, so that they can be modified by the external world. `Stéf` ▶*here*◀ The next example adds a bit more complexity: we are going to send a ByteArray to the memset function. memset fills an array with one value (literally man defines it as: "fill a byte string with a byte value").

**C header.**

```
void * memset ( void * ptr, int value, int num );
```

Sets the first num bytes of the block of memory pointed by ptr to the specified value

**Pragma declaration.**

```
CExamples class>>memorySet: aByteArray with: anInteger for: aByteCount
    <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

    ↑ NBFFICallout cdecl: #( void* memset (char* aByteArray, int anInteger, int
        aByteCount) ) module: NativeBoost CLibrary
```

**Callout grouping.**



*array := ByteArray new: 10.*
*CExamples memorySet: array with: 100 for: 5.*
*array*

As you should notice, this time Spock takes care of finding out the memory address where aByteArray lives and pushes a pointer to it as the first argument of memset. After executing the example you can see that the byte array elements were actually changed.

Then, there is an important difference between passing arrays and passing strings. The expert user would notice that string functions in C language expects to receive a pointer to contiguous memory bytes ending with a null character to delimit its length. Spock is responsible for copying the string to a bigger buffer and inserting this special end character. This means that strings are passed by-copy, while on the other hand byte arrays are passed by reference!.

Also for expertise users, you may notice that in this last example, as sending a reference, we are sending to a native library a pointer to an object that lives in Smalltalk's memory. We already know that Smalltalk's memory is auto cleaned by a garbage collector that potentially move the objects to other places, so we have to be careful about this special behavior. This is really important in the case of asynchronous calls, so in that case special measures may need to be taken to avoid strange behavior. We have to be very careful when facing with C pointer arguments to Smalltalk's memory. We will talk about asynchronous FFI on the latest sections.

## Default primitives types

Spock automatically converts the object passed as argument based on the signature declaration. For example if function takes float/double argument you specify it in signature:

```
CExamples class>>mymethod: anArg
   <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

   ↑ NBFFICallout cdecl: #( int foo ( double anArg) )
```

and then you can use this method as follows: self mymethod: 4.5

how to access "objects" in external memory: for example you call from smalltalk malloc for that you get an external pointer.

## 1.4   Todo

- instance variables as argument in C-function signature declaration including self.

- default options

- returned null value as nil

- external object (automatically has a handle) CairoSurface inherits from NBExternalObject (which manages the opaque handle to the CairoSurface C structure then

```
create
   <>
   call: CairoSurface  fct
```

- NBExternalObjectType is responsible for the generation of the code

## 1.5   Using C structures

## 1.6   Loading code from other libraries

## 1.7   Asynchronous FFI

## 1.8   Behind the scenes: NativeBoost naked!

## 1.9   NativeBoost Executive Summary

Here we say everything again but in a concise way. Javier ►*Ha!*◄

## 1.10   Wrapping Cairo library

Cairo is a drawing library written in C. Here we present the basis of writing a wrapper for it, which should be complete enough to write any wrapper for any C library.

**Small example**

Whenever you want to start writing a wrapper, you should do it this way: find the smallest piece of code you can find that uses that library, and wrap everything you need to make it work from Smalltalk. Then you can go on adding more and more to the wrapped functionality.

In this example we have a minimal program that uses Cairo library, taken from Cairo'sFAQhttp://cairographics.org/FAQ/ It creates a Cairo surface, puts some text on it and renders everything to a PNG.

```c
#include <cairo.h>

int
main (int argc, char *argv[])
{
    cairo_surface_t *surface =
      cairo_image_surface_create (CAIRO_FORMAT_ARGB32, 240, 80);
    cairo_t *cr =
      cairo_create (surface);

    cairo_select_font_face (cr, "serif", CAIRO_FONT_SLANT_NORMAL,
     CAIRO_FONT_WEIGHT_BOLD);
    cairo_set_font_size (cr, 32.0);
    cairo_set_source_rgb (cr, 0.0, 0.0, 1.0);
    cairo_move_to (cr, 10.0, 50.0);
    cairo_show_text (cr, "Hello, world");

    cairo_destroy (cr);
    cairo_surface_write_to_png (surface, "hello.png");
    cairo_surface_destroy (surface);
    return 0;
}
```

The way we will work then is to wrap everything we find in our way from the first to the last line. In order to wrap a library, you should get its reference documentation, so that you know exactly which are its structures, and functions. We start by creating some classes for all Cairo structures that are going to be used here and there in the wrapper.

We create a class for all the library's constants (we will fill them later):

Class 1.2:

```
SharedPool subclass: #NBCairoConstants
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'NBCairo−Core'
```

and a class for all the library's typedefs and enums (we will also fill them later):

Class 1.3:

```
SharedPool subclass: #NBCairoTypes
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'NBCairo-Core'
```

Notice that both of them are Shared Pools. We now create a base class for structures used in the wrapper, putting the shared pools, so that we can use all of them in the derived classes.

Class 1.4:

```
NBExternalObject subclass: #NBCairoObject
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: 'NBCairoConstants NBCairoTypes'
   category: 'NBCairo-Core'
```

Finally, to wrap the first line of main function we will create a class to represent a Cairo surface.

Class 1.5:

```
NBCairoObject subclass: #NBCairoSurface
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'NBCairo-Core'
```

With most well written libraries, the contents of the structures they declare is private to their internals. They are opaque, in the sense that you may not know nor care what bits they have inside, because you only change them through calls to the library. For that kind of structures, Spock has the NBExternalObject. Our wrapping classes inherit from NBExternalObject, which is made for this situation, where we don't need to fill the fields of the structs.

Now, NBCairoSurface is described as an abstract structure, there are different concrete ones, like the image surface, so we add it as a subclass.

Class 1.6:

```
NBCairoSurface subclass: #NBCairoImageSurface
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'NBCairo-Core'
```

Then we can finally create a factory method, to wrap the `cairo_image_surface_create` function. We go to the class side and we enter:

```
NBCairoImageSurface class>>create: aFormat width: aWidth height: aHeight

    <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

    ↑ self nbCall: #(NBCairoImageSurface cairo_image_surface_create (int aFormat,
                                        int aWidth,
                                        int aHeight) )
```

and of course, we shouldn't forget to wrap the code that destroys the instance on the C side, but this time on the instance side of the NBCairoSurface class:

```
NBCairoSurface>>destroy
    <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

    ↑ self nbCall: #(void cairo_surface_destroy (NBCairoSurface self) )
```

In the creation method, we declare the function as returning a NBCairoImageSurface, but actually it is returning a cairo_surface_t pointer. The same happens on the destroy method which is described as an NBCairoSurface, but actually is a cairo_surface_t pointer. In both cases, spock takes care of the conversion.

These methods are almost working. To finish, we add some helper functions to the NBCairoObject class, the one which was going to contain some useful configuration:

```
NBCairoObject class>>nbLibraryNameOrHandle
    ↑NativeBoost forCurrentPlatform loadModule: '/usr/lib/libcairo.so.2'

NBCairoObject>>nbLibraryNameOrHandle
    ↑self class nbLibraryNameOrHandle
```

Now to be able to test a bit, we have to add some more things. We add cairo_status_t (which is an enum) and cairo_surface_t to the NBCairoTypes.

<div align="center">Class 1.7:</div>

```
SharedPool subclass: #NBCairoTypes
    instanceVariableNames: ''
    classVariableNames: 'cairo_status_t cairo_surface_t 'poolDictionaries: ''category:
        'NBCairo-Core'
```

```
NBCairoTypes class>>initialize

    "self initialize"
```

```
cairo_status_t := #int
cairo_surface_t  := #NBCairoSurface.
```

and we doit:

*NBCairoTypes initialize*

so that cairo_status_t and cairo_surface_t get initialized. Then, we add a method to the NBCairoSurface class:

```
NBCairoSurface>>status
  <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

  ↑ self nbCall: #(cairo_status_t cairo_surface_status (cairo_surface_t self) )
```

Note what we just did: we added two types mappings, cairo_status_t that maps to int, and cairo_surface_t that maps to NBCairoSurface. That way we can now use this mappings in the function specifications, getting closer to copy-pasting the functions definitions into the binding.

We are now ready to test just a bit. Take a second and save the image now. Remember that if you wrongly specify the parameters of an FFI call, you could crash the image.

To test that everything is working do print this in a workspace:

*surface := NBCairoImageSurface new. surface status*

If everything is fine, status should be 0. Then don't forget to destroy the surface:

*surface destroy*

To make the example complete, let's add the final pieces needed.

Here is how the original example will look after translating it to use the wrapper:

```
CairoExamples>>minimalProgram
  "
  self new minimalProgram
  "

  | surface context fileName |
  surface := NBCairoImageSurface create: CAIRO_FORMAT_ARGB32 width: 240
    height: 80.
  context := surface createContext.
  context selectFont: 'serif' slant: CAIRO_FONT_SLANT_NORMAL weight:
    CAIRO_FONT_WEIGHT_BOLD;
      setFontSize: 32.0;
```

```
        setSourceR: 0.0 G: 0.0 B: 1.0;
        moveToX: 10.0 Y: 50.0;
        showText: 'Hello, world';
        destroy.

    fileName := (FileDirectory default / 'Hello.png') fullName.
    surface writeToPng: fileName.
    surface destroy.

    Form openImageInWindow: fileName
```

but before it works we have to add some constants and types. `CAIRO_FORMAT_ARGB32` is a constant defined on `cairo_format_t`, and we are using constants for font slants and font weights too. For each enum we add a type on NBCairoTypes and its constants to NBCairoConstants:

```
    instanceVariableNames: ''
    classVariableNames: 'CAIRO_FONT_SLANT_ITALIC
        CAIRO_FONT_SLANT_NORMAL CAIRO_FONT_SLANT_OBLIQUE
        CAIRO_FONT_WEIGHT_BOLD CAIRO_FONT_WEIGHT_NORMAL
        CAIRO_FORMAT_A1 CAIRO_FORMAT_A8 CAIRO_FORMAT_ARGB32
        CAIRO_FORMAT_INVALID CAIRO_FORMAT_RGB16_565
        CAIRO_FORMAT_RGB24'
    poolDictionaries: ''
    category: 'NBCairo-Core'

SharedPool subclass: #NBCairoTypes
    instanceVariableNames: ''
    classVariableNames: 'cairo_font_slant_t cairo_font_weight_t cairo_status_t
        cairo_surface_t cairo_t'
    poolDictionaries: ''
    category: 'NBCairo-Core'
```

and then we add the code to initialize these variables:

```
NBCairoTypes class>>initialize

    "self initialize"

    cairo_status_t := cairo_font_slant_t := cairo_font_weight_t := #int
    cairo_t := #NBCairoContext.
    cairo_surface_t  := #NBCairoSurface.


NBCairoConstants class>>initialize

    "enum cairo_format_t"
    CAIRO_FORMAT_INVALID := -1.
    CAIRO_FORMAT_ARGB32 := 0.
```

```
CAIRO_FORMAT_RGB24 := 1.
CAIRO_FORMAT_A8 := 2.
CAIRO_FORMAT_A1 := 3.
CAIRO_FORMAT_RGB16_565 := 4

"enum cairo_font_slant_t"
CAIRO_FONT_SLANT_NORMAL := 0.
CAIRO_FONT_SLANT_ITALIC := 1.
CAIRO_FONT_SLANT_OBLIQUE := 2

"enum _cairo_font_weight_t"
CAIRO_FONT_WEIGHT_NORMAL := 0.
CAIRO_FONT_WEIGHT_BOLD := 1.
```

don't forget to doit:

   *NBCairoConstants initialize*

and

   *NBCairoTypes initialize*

As a last step, you need to wrap the remaining functions. We leave here a last example and the rest of them you can take the code from the NBCairo repository on squeaksource.

<div align="center">Class 1.8:</div>

```
NBCairoSurface subclass: #NBCairoImageSurface
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'NBCairo-Core'
```

```
CairoContext>>selectFont: aFamily slant: slant weight: aWeight
  <primitive: #primitiveNativeCall module: #NativeBoostPlugin>

  ↑ self nbCall: #(void cairo_select_font_face (cairo_t self,
                                    String aFamily,
                                    cairo_font_slant_t slant,
                                    cairo_font_weight_t aWeight) )
```

Now as a general principle it is better not to manipulate all the variables in your program. Why first because the call out will be slowler and second because it is better to encapsulate all these constants. `Javier` ▶*Explain more!*◀

So the solution is to do something like @@@to be fixed@@

```
AthensCairoSurface>>primWidth: aWidth height: aHeight
  <primitive: #primitiveNativeCall module: #NativeBoostPlugin>
```

```
↑ self nbCall: #(AthensCairoSurface cairo_image_surface_create(
      CAIRO_FORMAT_ARGB32,
         int aWidth,
         int aHeight) )
```

## Original FFI stuff

Thanks to a Foreign Function Interface (FFI), it is possible to achieve this and interact with third-party libraries. Multiple FFI libraries are available in Pharo such as FFI (http://wiki.squeak.org/squeak/1414) and Alien (http://www.squeaksource.com/Alien). As an example the dynamic `OpenGL` library (.dll, .so or .dylib depending on the operating system) is used in Croquet through the FFI library and AlienOpenGL (http://www.squeaksource.com/AlienOpenGL) use it through the Alien library.

In this chapter, we will describe what is a FFI library and how it works with the virtual machine. We will then dive into the Alien library and learn how it can be used thanks to the AlienOpenGL example.

## 1.11   What is a FFI and how it works?

The Pharo image is loaded and executed by a virtual machine (VM) which is itself executed on the top of the operating system. Two different virtual machines can be used for Pharo images: the squeak VM and the Cog VM. Regarding the general FFI mechanism, it is not really relevant to make a distinction[4] Both are based on a plugin architecture. A plugin enable the Smalltalk code (in the image) to access third-party functionalities (outside the image and the VM).

The Alien library uses the *IA32ABI* plugin which ....

## 1.12   Using the Alien Library

## Installation

Alien is a Squeaksource project http://www.squeaksource.com/Alien. Use the following code to install Alien in a Pharo 1.1 image:

```
Gofer new
   url: 'http://www.squeaksource.com/Alien' ;
   package: 'ConfigurationOfAlien';
   load.

(Smalltalk at: #ConfigurationOfAlien) loadCore ;
   loadTests ;
   loadLibC.
```

---

[4]It worth noting that the Cog VM is younger and really faster than the Squeak VM but it still presents some small limitations and bugs at the time writing this book.
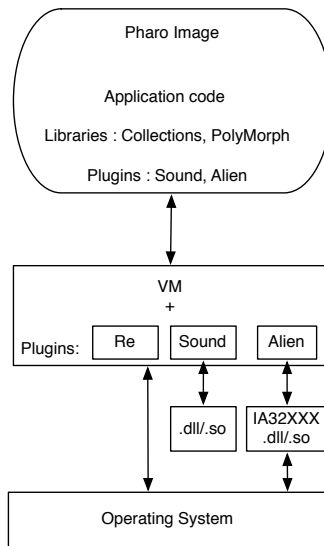
Figure 1.1: Pharo Architecture

Run the tests to ensure that Alien is well installed and that the VM plugin is correctly working.

### First example

### Core Classes

Figure 1.2 shows an UML diagram of the Alien-Core package that contains the main classes of this library.

### Memory management

### Calling an external C library from Smalltalk

### Callbacks from C to Smalltalk code

## 1.13   Dissection of Alien
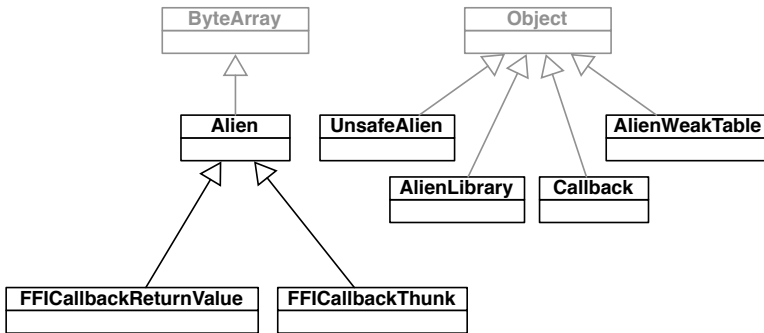
### Alien

### FFICallbackReturnValue

Figure 1.2: Core classes of the Alien library

**FFICallbackThunk**

**UnsafeAlien**

**AlienLibrary**

**Callback**

**AlienWeakTable**

## 1.14   Study Case: the AlienOpenGL package