Environments
(a namespace implementation for Smalltalk)

Germán Leiva (bio)
E-mail: leivagerman@gmail.com

Abstract

This presentation will focus in the current state of *Environments*, a lightweight namespace implementation for Pharo. I'll introduce some basic ideas related with namespaces doing a live tutorial (see the document belove). The main idea of the project is to create a collaborative space for the community to build a progressive implementation that meets most of the problems caused by the lack of namespaces. You can see the presentation of *Environments* with a simple example (less than 10 minutes) do it in the ESUG 2010 following this link.

Introduction

This document shows the basic ideas behind the Cross-Platform Namespaces project of the Google Summer of Code 2010.

What's a namespace?

[...] In real life, names are used for symbolic identification of real things which are known under this particular symbol in the particular context. Such things known by name are *notions*. [...][1]

One problem is that two different notions can have the same name in different contexts. In natural language this is solved by the context where the conversation happens.

According this document a namespace is only a classification of names, a collection of symbols that forms a family of related notions.

In Smalltalk

Why?

Traditionally, all globals in Smalltalk are visible to all code in the image. While this works fine on a small scale, it becomes a problem when code originates from many different individuals or teams (which is particularly common in an open source community such as Squeak or Pharo).

Each module that is developed will naturally have class names that reflect the domain being modeled. In any large system there is likely to be overlap among the names chosen (e.g., Person, User, Preferences, Server, Client, etc.) and the likelihood of global name collision (including class names) grows as the system grows.

The typical approach is to preface class names with an abbreviation of the module name. While this works, it is cumbersome and tends to make names either cryptic or cumbersome. What is needed is a system in which internal class names (and other "globals") are visible only to the internals of a module that defines them.

Main ideas begin the language

[...] In Smalltalk, expressions start with a variable or a literal. Literals are "simple data" which are instantiated in compile-time. Variables are place-holders for objects which receive the subsequent message in run-time. The variables are assigned in compile-time, i.e. they are **not** message sends but assignments. Hence, from this point of view, early binding is used for the identification of the primary receiver of the first "normal" message in an expression (this is true for all kinds of variables - temporary, instance, class up to pool dictionaries and global variables, although the last ones are accessed not directly, but rather via a "remote" object - an association). [...] [1]

To a common namespace implementation

Some Smalltalks like VisualWorks [2], GNU [3] , GemStone [4] and VA [5] have their own implementation of Namespaces.

For a general overview we can categorize the different approaches in

- Globals must have unique names (like Squeak or Pharo - with or without tooling help[6])
- Duplicate names are hidden from the compiler by manipulation of the traditional look-up scopes (*)

Who has the responsibility of knowing the namespace that must be used by some object in a specific moment? Different paths can be taking here (some of them more dynamic and other more static).

One of our goals is proposing a solution that doesn’t change radically the way we use Smalltalk today and that keep unchanged the current run time behaviour of the system.

When a class is used in some method the corresponding object of that literal must be found, commonly with a #bindindOf: or #associationAt: message to an object that responds to the dictionary interface.

In traditional Smalltalks this dictionary is composed by associations that have a key that is the name (a Symbol) of the global and a value that is the instance that represents that global.

The main idea is having different objects per namespace where each of them have that interface, relate them so they can build more complex namespaces and provide a standardized way to define and access his elements.

In Smalltalk, methods are recompiled when an instance variable is added or removed from a class because the scope for a variable binding can change.

(*) Note: In the second category we can do a look-up of the namespace itself in a possible collection of namespaces but this approach is not taken in the current reference implementation

Ideas

Global Namespace

The relations between namespaces are related to their categorization.

For compatibility purposes we need to maintain the current “Smalltalk dictionary”. A variety of classes are defined there like Collections, Numbers, Dates, etc. and having a special name for it seems to be reasonable. For this document “Smalltalk dictionary” will be the Global Namespace and his name will be ***Smalltalk***.

Sharing

We think that is needed some kind of specification about what classes need to be used in a namespace but are defined in some other place.

We are not encouraging the idea of imports but in the current design this was unavoidable.

In a default way and for a practical use all namespaces have a shared relationship with ***Smalltalk***. If for some reason is not needed this can be modified (this will allow some kind of **name shadowing** in the future)

Sharing with alias

When a direct sharing allow to use a class “like” that class is defined in the current namespace the use of alias explicitly shows the use of another namespace. This has a degree of subjectivity but we think that the explicitation is a good practice.

Changing the namespace not the code

With the sharing mechanism we can change the one namespace for another that defined the same classes in a polymorphic way (with is particular implementation) but the code that use that classes must maintain unchanged.

An example for this will be in the testing area. When you have to test some piece of code that have side-effects (a change of state in the screen, the internal state of some object, a persistence mechanism, etc.) is common to use **mocks**.

The idea is that the mocking process is implicit in the change of namespace and the code will be more readable and more important the true “business code” must be tested.

Look-up

What is a class is defined in the namespace and in someone of the shared ones? The natural solution is that the current namespace have priority over the others.

Then some kind of order must be set up in the related namespaces or explicitly (with tooling support) avoid name collisions between them.

Accessing the classes in a Namespace

Now that we have an idea of what is a namespace and what features can have, we need to answer some others questions related to the usability.

We can tread a namespace like a dictionary and because of that we can use the same protocol. e.g.

```
 India at: #Elephant
```

With this approach if we want to instantiate an Indian elephant we need to write

```
(India at: #Elephant) new
```

The drawback of using exactly the same protocol of dictionaries is that we need to put parenthesis enclosing the message send.

We think that this reduce the expressiveness and is unnecessary.

This problem can be solved using a message without parameters, something like

```
India Elephant new
```

In this case Elephant is a unary message, not a class.

But two well known conventions in Smalltalk are

- All message selectors starts with lowercase (included the class method selectors)
- All class names starts with uppercase

This means that we are not following the conventions because Elephant is a message selector that starts with uppercase. In spite of this we think that using the name of the class as a message (with uppercase) encourage the notion of using a class and benefits the readability.

This “convention exception” only is used by the namespace mechanism but in any other place

can’t be restricted directly by the system.

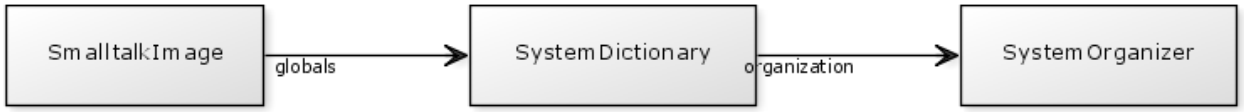
In Pharo

Environments

The first step to have a cross-platform unification need to start at community level, we want to prepare the grown for a community accepted way of using namespaces

For this "reference implementation" we choose Pharo (a fork from the Squeak open-source Smalltalk).

Pharo has a modified implementation of the “Smalltalk dictionary”



Smalltalk *is a global variable that points to the sole instance of SmalltalkImage*
SystemOrganization *is a global variable that points to the sole instance of SystemOrganizer*

In most cases SmalltalkImage and SystemDictionary are polymorphic but the idea is to separate his responsibilities.

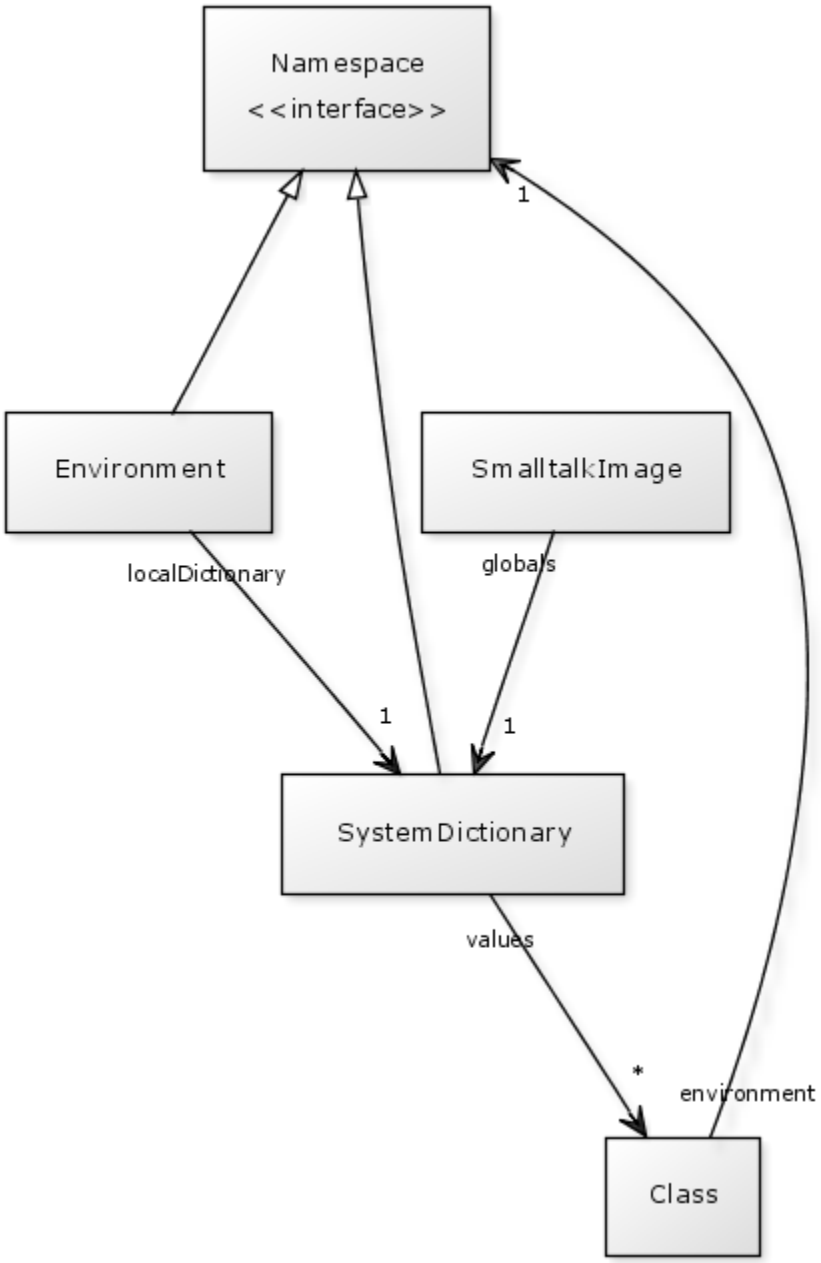
In the class Class is defined an **environment** instance variable that references the same object that ***Smalltalk*** variable.

Some refactors for future modifications that are encourage are

- the use of the message send Smalltalk globals instead of the global variable ***Smalltalk***
- and the use of the message **environment** when is possible

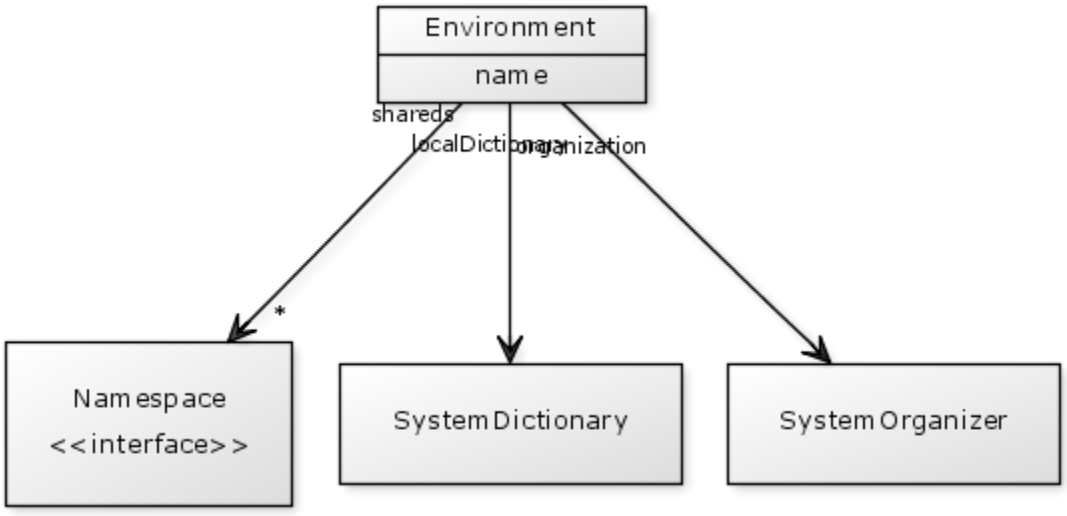
We see this like a nice hookup for a possible namespace implementation.

Current design



For this implementation we choose to have subclasses of Environment for every new environment, the reason begin this is that the messages for accesing the classes (described earlier in **Accesing the classes in a Namespace**) are not implemented redefining #doesNotUnderstand:, instead we compile an remove that methods in a controlled way for every subclass. The implementation can change this is only a reference.

An environment knows the rules of look-up for a class in that namespace, they have his own organization and his own system dictionary.



Example

If we have a namespace India and the only class defined there is Elephant then … what happens with this method?

```
India Elephant >> createFood
      | aLotOfPeanuts |
      aLotOfPeanuts := Bag new.
      100 timesRepeat: [ aLotOfPeanuts add: Peanut new ].
      ^aLotOfPeanuts
```

This code shows the use of classes defined in namespaces different than the one in witch code is being written.

Global Namespace

We are using 2 classes in this method but both of them are not defined in India.
Classes defined outside a namespace can be categorized in

- System classes (collections, dates, numbers, exceptions, etc.)
- Classes in other namespaces

For the consistency of the mechanism all classes are defined in one namespace.

Besides this, we need to explicitly create a relation between namespace India and the Global Namespace if we want to use his classes in India.

By default all namespaces has a relation with the Global Namespace.

Back in our example, in the method create food if we assume that Bag is in the Global Namespace then the Bag class will be available.

Other namespaces

What happens to the class Peanut? If that class is not defined in the India namespace and the Global Namespace then it won't be available.
We can define Peanut in other namespace, for example Food, and then create a relation between India and Food.

Monticello

The model of monticello doesn't have the idea of environment (the classes are searched in Smalltalk, the organization is always SystemOrganization, etc.) an implementation based in the current model was added but a new design is needed for future add-ons.

OmniBrowser

We added an EnvironmentsSystemBrowser (a rought GUI for environments) based in the SystemBrowser, an important number of the features work for environments but in some places a major refactoring is needed.

Implementation tips

Association trick

We will play with globals variables for a minute

If we do
```
Smalltalk globals at: #UltimateAnswer put: 'Forty Two'.
```

We create in the a system dictionary a new association **#UltimateAnswer -> 'Forty Two'**

And then we define
```
SomeClass >> surprise
  ^UltimateAnswer
```

This will reference the same association created before and if we execute

```
SomeClass new surprise. "This will return 'Forty Two' "
```

If we made a change
```
Smalltalk at: #UltimateAnswer put: 42.
```

This will change the **same association**, his identity is not lost.

And now we execute
```
SomeClass new surprise. "This will return 42"
```

Notice that the result changed but the method is not recompiled.

And finally, if we do
```
Smalltalk removeKey: #UltimateAnswer
```

And now we execute
```
SomeClass new surprise. "This will still return 42"
```

Again, the system dictionary lose the reference to that association but is still referenced in the method.

We can take advantage of this behavior.

Associations Cache

For improving the lookup when a class is searched in an environment, that has others shared namespaces, we can add the association founded in a cache for future lookups.

The main problem with this idea is related with the "association trick".
If we save in the cache a new association any changed do it in the shared namespace will not be seen in the current environment.

This idea was used in first place to do a replacement from one shared environment to other without recompiling the methods.

This problem can be solved using some kind of announcement to update the modification in the association placed in the cache, but this will introduce and overhead that it's needs to be measure with the benefits made by the cache.

Further work

Package and Namespaces

We need to take a deeper idea of the relation of packages and namespaces. With the introduction as packages as first class-object we think that a better relation with this two concepts can be made.

The need of backwards compatibility and the integration with Monticello made us take some assumptions

- We assume that the name of the category is the name of the environment
- One package is related with one environment

This limitations are only in this reference implementation and need to be solved, the design allows and independence between categories and environments and multiple environments in

one package.

Coverage of the namespace

Does a namespace apply to all methods in a class or can different methods in a class be associated with different namespaces?

The current implementation does not support method coverage at method level, but we think that this change is one that relate the namespace with a package implementation.

References

[1] http://wiki.squeak.org/squeak/734
[2] http://www.cincomsmalltalk.com/
[3] http://www.gnu.org/software/smalltalk/manual/html_node/Namespaces.html
[4] http://www.gemstone.com/docs/GemStoneS/GemStone64Bit/2.4/GS64-ProgGuide-2.4.pdf
[5] http://www2.instantiations.com/VAST/Docs/802/wwhelp/wwhimpl/js/html/wwhelp.htm#href=pr/stpr666.html
[6] http://swiki.krampe.se/gohu/32