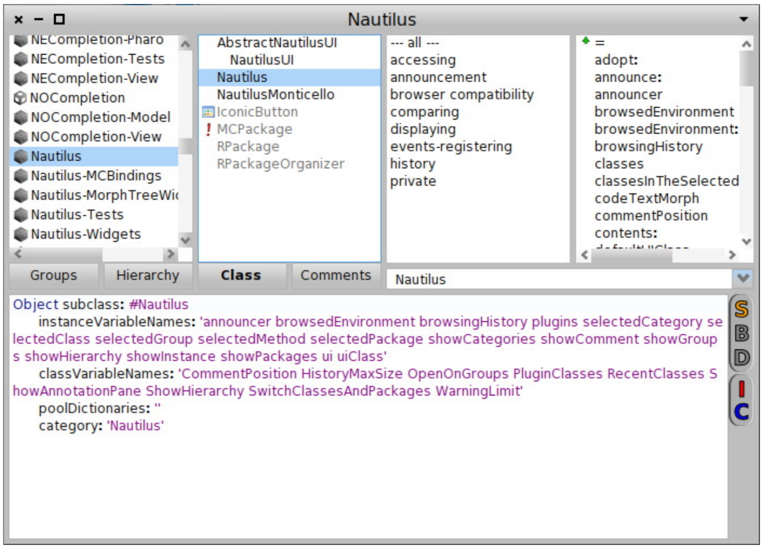
# Chapter 1

# Nautilus

*Nautilus* is a new browser made on top of Morphic. It was designed to browse *RPackage*, to be compatible with the RB refactoring engine, to be *environment aware* and to work with Announcements.

Quite a while ago it became obvious that a lot of needed features were missing in the current browsers. So we decided to make the new browser as simple as possible to extend and at the same time complete enough to become a really helpful environment.



```
Object subclass: #Nautilus
	instanceVariableNames: 'announcer browsedEnvironment browsingHistory plugins selectedCategory selectedClass selectedGroup selectedMethod selectedPackage showCategories showComment showGroups showHierarchy showInstance showPackages ui uiClass'
	classVariableNames: 'CommentPosition HistoryMaxSize OpenOnGroups PluginClasses RecentClasses ShowAnnotationPane ShowHierarchy SwitchClassesAndPackages WarningLimit'
	poolDictionaries: ''
	category: 'Nautilus'
```

## 1.1 Main Features

In this section the main features of *Nautilus* will be explained but keep in mind that there are a lot of little tricks which make *Nautilus* nice to work with.

### RPackage

*RPackage* is a model for packages. So instead of having to parse Strings and so on, you can manipulate real objects and query them about the system. *Nautilus* is the default browser to navigate through *RPackages*.

### Environment aware

We wanted *Nautilus* to be able to be easily connected with the refactoring engine. This way we made *Nautilus* totally dependent of a unique entry point which is an environment. So the whole system builds queries through this entry point. It enables browsing specific environments as remote systems or different name spaces.

### Ring

Using the same logic we decided to make *Nautilus Ring compliant*. This way we ensure that you can browse every piece of code using the tool. In addition to the sources in the image *Nautilus* together with *Ring* can browse change sets, slice, method references etc.

### Announcements

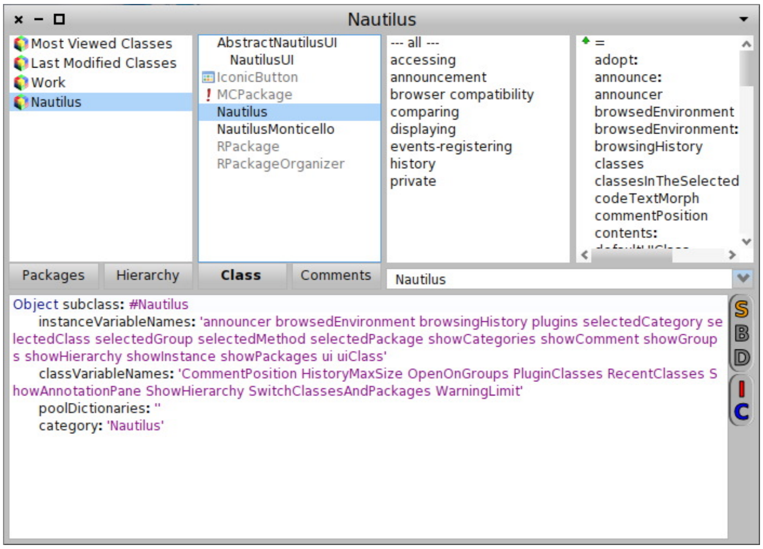
Since *Announcements* are used by *RPackage* and should replace Events soon, we have choose to only use *Announcements* in *Nautilus*. Currently the following *Announcements* are generated:

- when the system is modified and notify the changes
- when *Nautilus* emits its own changes (for plugins by example)

### Groups

We have noticed that most of the time when you are working your modifications are focussed on 2 or 3 packages. However in the default browser these packages are hidden in the list of all packages. Hence we have created groups to

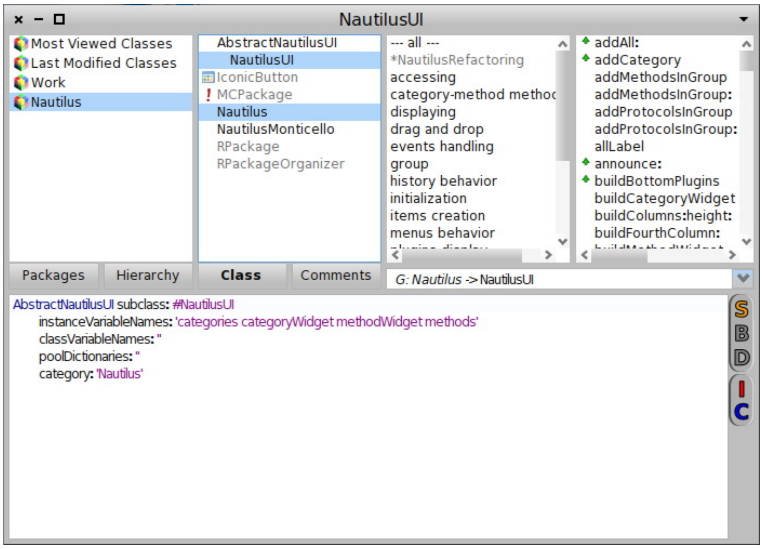
- select only the packages you are interested in
- join classes from different packages in single place
- be able to hide the information irrelevant to your task.



```
Object subclass: #Nautilus
	instanceVariableNames: 'announcer browsedEnvironment browsingHistory plugins selectedCategory se
lectedClass selectedGroup selectedMethod selectedPackage showCategories showComment showGroup
s showHierarchy showInstance showPackages ui uiClass'
	classVariableNames: 'CommentPosition HistoryMaxSize OpenOnGroups PluginClasses RecentClasses S
howAnnotationPane ShowHierarchy SwitchClassesAndPackages WarningLimit'
	poolDictionaries: ''
	category: 'Nautilus'
```

## Multi-Selections

We found out that most of the time the actions you are doing when manipulating the infrastructure are the same for a set of objects. By example when you refactor a class, you move a set of methods from a protocol to another. And this kind of example is relevant at different levels of the hierarchy. Hence we implemented multi-selection lists to be able to do such manipulations at once.



```
AbstractNautilusUI subclass: #NautilusUI
    instanceVariableNames: 'categories categoryWidget methodWidget methods'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Nautilus'
```

## Shortcuts

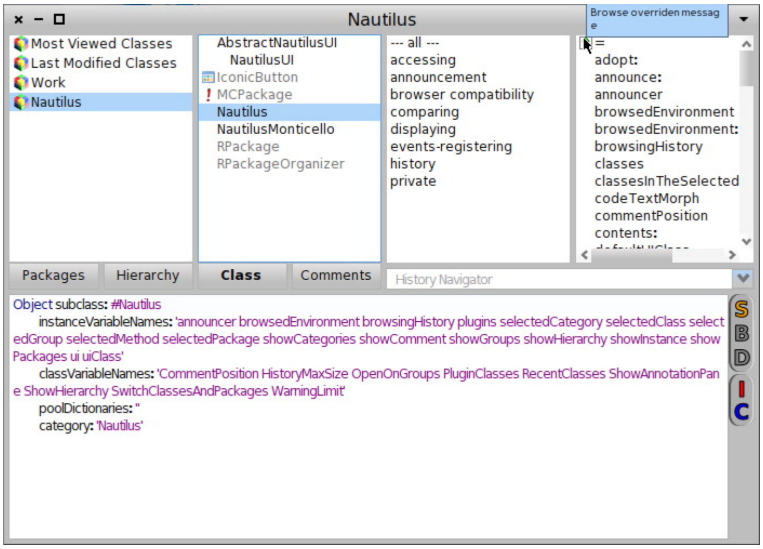
Because we do not want to be forced to use the mouse to navigate through the browser while coding and because it breaks the working flow, a lot of shortcuts have been implemented to increase the efficiency of the developer while using *Nautilus*.

The full list of shortcuts is available in the section 1.2.

## Interactive Icons

We were looking for a way to provide more information about the different structures. Then we realized that icons on list elements could provide those information at a glance. Thanks to that, you can instantly know that the method you are reading comes from a trait. Or that it is overriding a method from a superclass. And then we wanted to go further by providing an interaction with the icons, therefore most of the icons can be clicked.

See the section 1.3 for the complete list of icons.



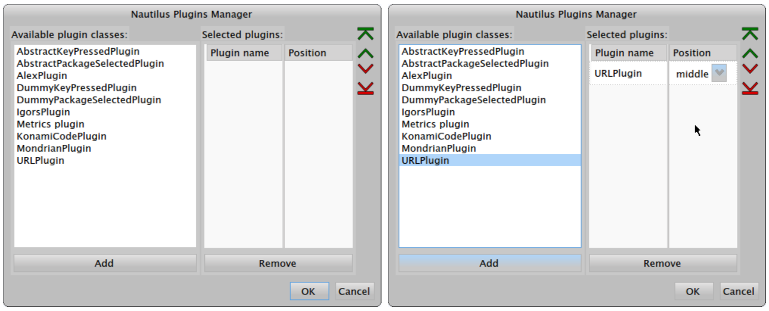
```
Object subclass: #Nautilus
	instanceVariableNames: 'announcer browsedEnvironment browsingHistory plugins selectedCategory selectedClass selectedGroup selectedMethod selectedPackage showCategories showComment showGroups showHierarchy showInstance showPackages ui uiClass'
	classVariableNames: 'CommentPosition HistoryMaxSize OpenOnGroups PluginClasses RecentClasses ShowAnnotationPane ShowHierarchy SwitchClassesAndPackages WarningLimit'
	poolDictionaries: ''
	category: 'Nautilus'
```

## Plugins

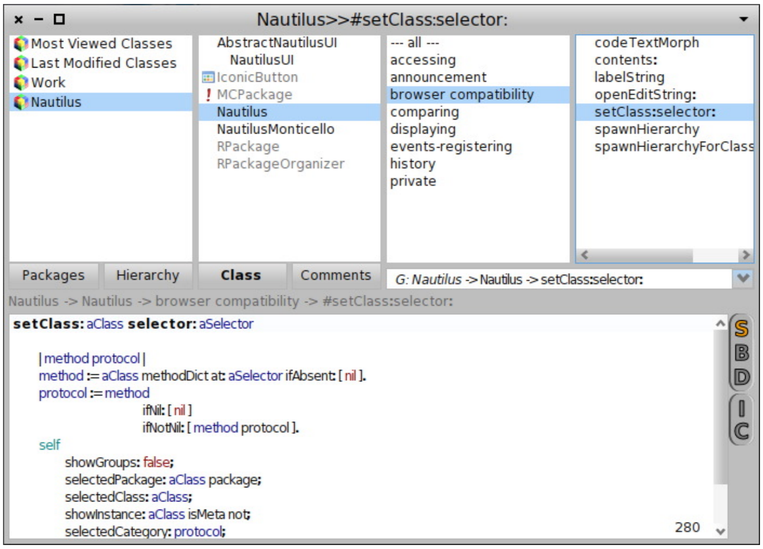
One of our goals was to make *Nautilus* easily extendable. To reach this goal we implemented a plugin mechanism. Every external project can create a *Nautilus-Plugin* and then the user can plug it into his browser, or the project can register the plugin to be loaded by default.

Each plugin can have an optional graphical part included into *Nautilus* and listen to *Announcements* emitted by *Nautilus*.

Read the section 1.4 to learn how to create your own plugin.



| Plugin name | Position |
|---|---|
| URLPlugin | middle |



```
setClass: aClass selector: aSelector

	| method protocol |
	method := aClass methodDict at: aSelector ifAbsent: [ nil ].
	protocol := method
				ifNil: [ nil ]
				ifNotNil: [ method protocol ].
	self
		showGroups: false;
		selectedPackage: aClass package;
		selectedClass: aClass;
		showInstance: aClass isMeta not;
		selectedCategory: protocol;
```

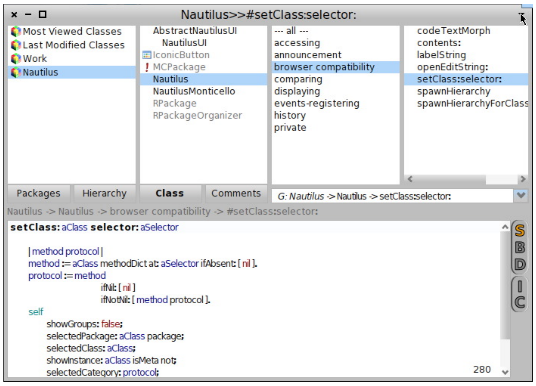
### Extendable menus

Always with the idea of being easily extendable, all the menus of *Nautilus* are defined using pragmas. This allows every external package to create an extension possibly adding entries to menus as long as it links the method to the correct pragma(s).

## 1.2 Shortcuts

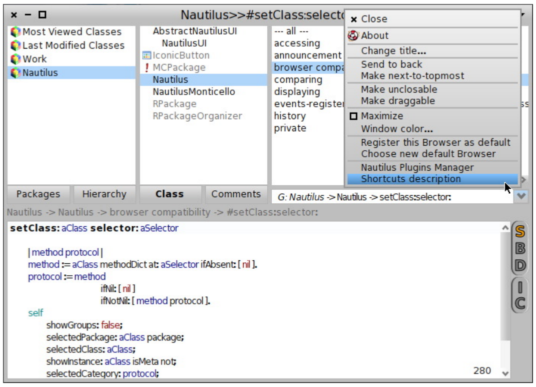
Thanks to Guillermo Polito, shortcuts are defined using Keymapping. This structure allows to be able to easily defined new shortcuts.

To retrieve the description of shortcuts, just follow the few steps below:



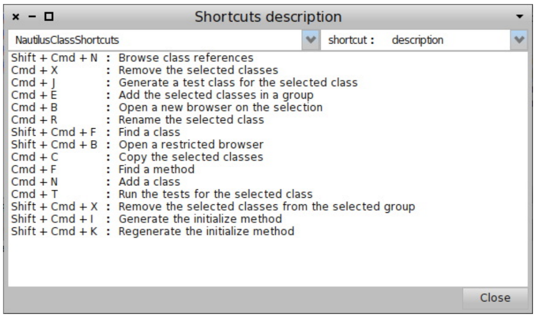
```
setClass: aClass selector: aSelector

	| method protocol |
	method := aClass methodDict at: aSelector ifAbsent: [ nil ].
	protocol := method
				ifNil: [ nil ]
				ifNotNil: [ method protocol ].
	self
		showGroups: false;
		selectedPackage: aClass package;
		selectedClass: aClass;
		showInstance: aClass isMeta not;
		selectedCategory: protocol;
```



```
setClass: aClass selector: aSelector

	| method protocol |
	method := aClass methodDict at: aSelector ifAbsent: [ nil ].
	protocol := method
				ifNil: [ nil ]
				ifNotNil: [ method protocol ].
	self
		showGroups: false;
		selectedPackage: aClass package;
		selectedClass: aClass;
		showInstance: aClass isMeta not;
		selectedCategory: protocol;
```



```
Shift + Cmd + N  :  Browse class references
Cmd + X          :  Remove the selected classes
Cmd + J          :  Generate a test class for the selected class
Cmd + E          :  Add the selected classes in a group
Cmd + B          :  Open a new browser on the selection
Cmd + R          :  Rename the selected class
Shift + Cmd + F  :  Find a class
Shift + Cmd + B  :  Open a restricted browser
Cmd + C          :  Copy the selected classes
Cmd + F          :  Find a method
Cmd + N          :  Add a class
Cmd + T          :  Run the tests for the selected class
Shift + Cmd + X  :  Remove the selected classes from the selected group
Shift + Cmd + I  :  Generate the initialize method
Shift + Cmd + K  :  Regenerate the initialize method
```

## 1.3 Icons

### Packages

| Icon | Meaning | If clicked |
|---|---|---|
| ■ | The default package icon | Can't be clicked |
| 🟥 | The corresponding MCPackage is dirty | Open the MonticelloBrowser on this package |
| □ | The package is empty | Can't be clicked |

### Groups

| Icon | Meaning | If clicked |
|---|---|---|
| ▦ | The default group icon | Open a new browser browsing an environment based on the selected classes |

### Classes

| Icon | Meaning | If clicked |
|---|---|---|
| ! | The class is not commented | Pop up a comment editor |
| ⚫ | The tests haven't been run | Run the tests |
| 🟢 | The tests passed | Run the tests |
| 🟡 | Some of the tests failed | Run the tests |
| 🔴 | Some of the tests produced an error | Run the tests |

### Protocols

For the moment, there is no icons on protocols.

### Methods

| Icon | Meaning | If clicked |
|---|---|---|
| ⬆ | The method overrides another method | Jump to the overridden method |
| ⬇ | The method is overridden | Browse the methods |
| $\mathcal{T}$ | The method comes from a trait | Browse the Trait class |
| ⚫ | The test has not been run | Run the test |
| 🟢 | The test passed | Run the test |
| 🟡 | The test failed | Run the test |
| 🔴 | The test produced an error | Run the test |

## 1.4 How to create Nautilus-Plugins

Here we will give some brief explanations on how to create your own plugin. By the way there are only two requirements to create a *Nautilus-Plugin*:

- the class should inherit from `AbstractNautilusPlugin`
- it should implement `#registerTo: aModel`

### Announcement subscription

The method `#registerTo:` is used by the plugin to register itself to `aModel` announcements.

```
MyPlugin>>#registerTo: aModel

   aModel announcer
      on: NautilusKeyPressed send: #keyPressed: to: self
```

In this example, the instance of `MyPlugin` subscribe themselves to `NautilusKeyPressed`, and tell *aModel's announcer* to send: `#keyPressed` to the instance.

So each time a key will be pressed in a Nautilus window the method `keyPressed:` will be called.

## Display

If you want your plugin to add a graphical widget to *Nautilus* you should override the `display` method. This method should return the *Morphic* element you want *Nautilus* to display. By default the method returns nil to notify *Nautilus* not to display anything.

```
MyPlugin>>#display

   morph :=  LabelMorph new
              contents: '';
              enabled: false;
              vResizing: #shrinkWrap;
              hResizing: #spaceFill;
              yourself.
   ↑ morph
```

You can also redefine the following methods on the class side:

- `#defaultPosition` : defines the default position of the morph (possible values are {#top, #middle, #bottom, #none}). The default value is #none.
- `#possiblePositions` : answers a collection of the possible positions the widget could adopt.

And finally you could redefine the `name` method to change the name displayed in the *Nautilus Plugin Manager*.