To the FAQ/Table of Contents

Newspeak on Squeak

A Guide for the Perplexed

December 2014 Update
The Spur Release

Gilad Bracha, Peter Ahe, Vassili Bykov and Ryan Macnak

This document is a guide to using Newspeak in the prototype release running on top of Squeak. It acts as a tutorial for the IDE, and to a limited extent, for the Newspeak language as well.

You can read this document sequentially, from start to finish, as a structured tutorial; or, you can just use the FAQ to quickly lead you to specific task oriented sections. If you want to read things in sequence, just begin at the beginning.

Table of Contents/FAQ

I just want Hello World!

How do I install Newspeak?

How do I open the Newspeak IDE?

How do I open a Newspeak browser?

How do I browse an existing class?

How do I browse senders/implementors?

How do I delete a method?

How do I change the category of a method?

How do I navigate in the browser?

How do I erase my history?

How do I manage a Newspeak window?

To the FAQ/Table of Contents

How do I inspect the GUI?

How do I inspect an object?

How do I evaluate code?

What if evaluation raises an exception?

What if I want to get at the platform or IDE namespace from an inspector?

How do I run the Hopscotch debugger?

How do I open a workspace?

How do I edit the class header?

How do I edit the class comment?

How do I create a new class?

How do I convert Smalltalk code to Newspeak?

How do I add a method?

How do I delete a class?

How do I add a slot?

How do delete a slot?

How do I find a class by browsing the IDE namespace?

How does source control work?

How do I use the Native GUI?

How do I run unit tests?

Why is stuff red?

How does the Syntax differ from the Specification, and Why?

What is the configuration link on the home page for?

How do I deploy an application?

To the FAQ/Table of Contents

How do test my deployment configuration?

Begin at the Beginning

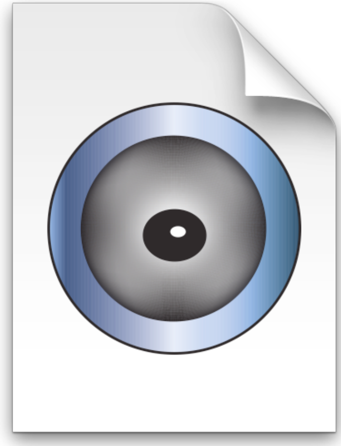
Let's start by opening the Newspeak IDE. Make sure you've installed Newspeak first!

How do I install Newspeak?

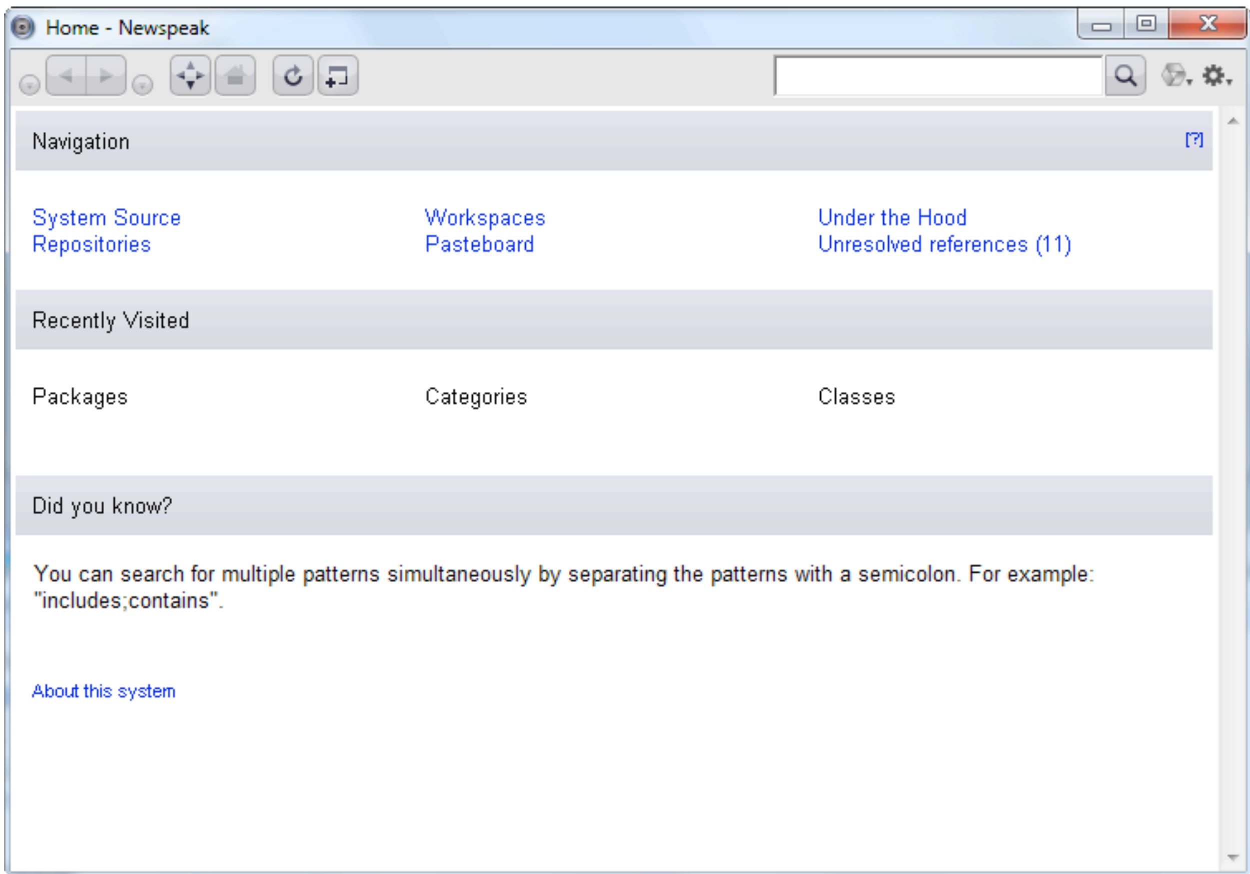
On Windows, use the installer. If you have a previous installation, it's best to uninstall the old one and reinstall. On a Mac, open Newspeak Spur Virtual Machine.dmg, and drag the VM to the Applications folder. Make sure that ns101.image is associated with the Newspeak Virtual Machine. This can be an issue, especially if you have ordinary Squeak installed on your machine as well. On Linux, follow the instructions given in linux-advice.txt.

How do I open the Newspeak IDE?

The directory where you found this document probably includes a file called **ns101.image**. This is a Newspeak image, and it should be identifiable by its icon, the Newspeak eye (well, it's big brother's eye actually):

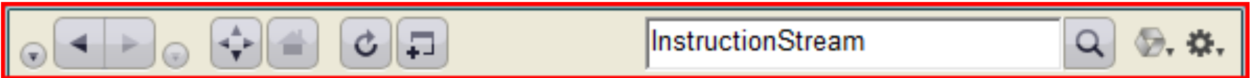


Click (or double click, as the case may be) on the image file. This will open the IDE; you should see something like this:



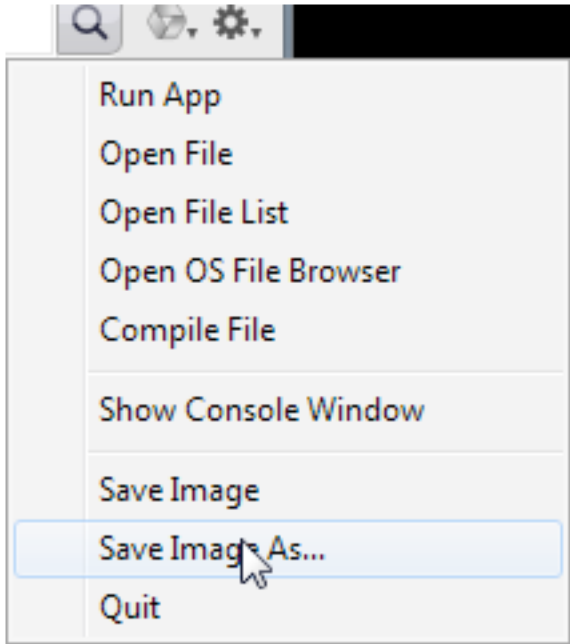
The very first thing you should do is save a copy of your image, so your experiments don't trash the release image you just opened.

At the top of the window, you'll see a tool bar that looks like this:



The rightmost icon denotes a drop down menu known as the *operate menu*.

Choose the option *Save Image As ...* and give the image some other name.



Now we can proceed.

Newspeak currently runs on top of Squeak. If you're running on Mac OS X or Linux, all IDE windows open up within the main Squeak window. On Windows we use a native GUI binding. Most screen shots in this document show the Windows native binding running on Windows 7.

The screen shot above shows the home page. You can always return to the home page of a Newspeak browser by clicking on the home icon at the top of the browser.

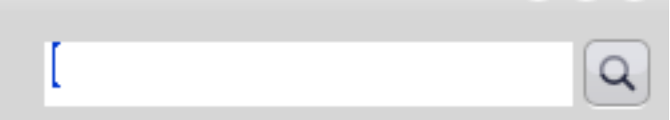
The home page includes links to a variety of useful places, like recently visited classes and packages, the source control page and more. If you look at the list of classes, you'll see each class has a round icon next to it. The icon tells you which language the class is written in; the current system mixes Smalltalk code with Newspeak. A gray icon

represents Smalltalk. Icons representing Newspeak are labeled with a version number. Golden icons represent Newspeak3, which is the currently operational dialect of Newspeak.

Now that we've got ourselves a browser, let's browse some existing classes.

How do I browse an existing class?

There are several ways to browse an existing class. If you know the name of the class (or some approximation thereof), you can search for it using the search pane in the upper right hand side of the Newspeak browser's tool bar:



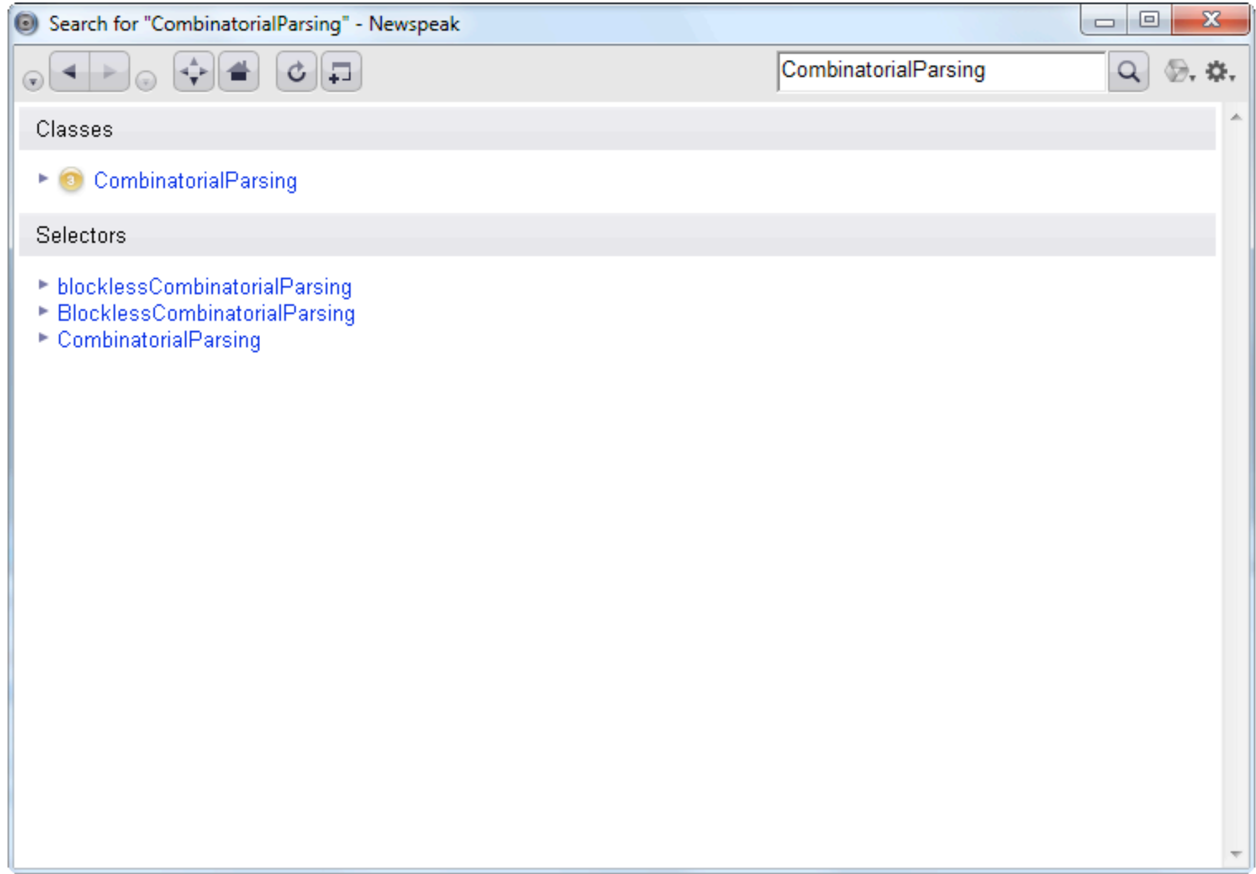
One can also go to the system source page and look for the class there. We'll examine both options below.

Searching

You can type multiple search terms, separated by semicolons. The search is case-insensitive, and will find anything that includes the searched strings. You can use * as a wildcard character. If you enclose a search term in double quotes, you'll only get exact matches.

We'll search for the class **CombinatorialParsing**. Type the name of the class into the search pane. This yields a list of classes and a list of method names matching that string.

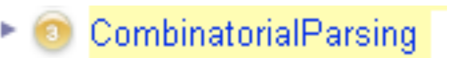
[To the FAQ/Table of Contents]

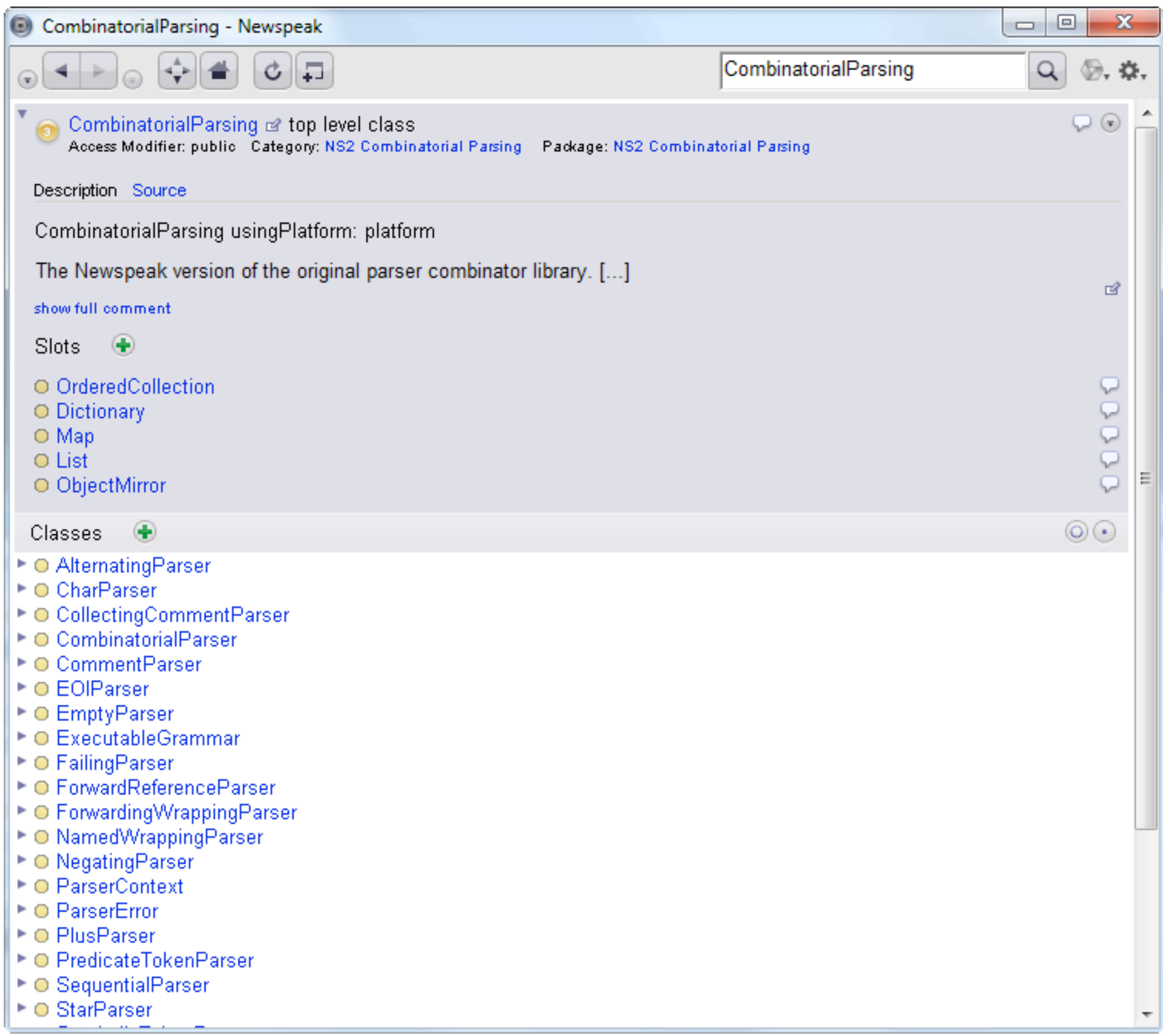


Every element in the lists is a link, and clicking on it will take us to the corresponding class or method. We can also view one or more of the classes/methods in place, by clicking on the arrow icon

to the left of the link. The behavior of the arrow icon is similar to its behavior in the mac finder (or if you use a PC, the plus/minus signs in Windows explorer).

Notice how the links are underlined as you hover over them, just like in a web browser. Click on the link for **CombinatorialParsing**



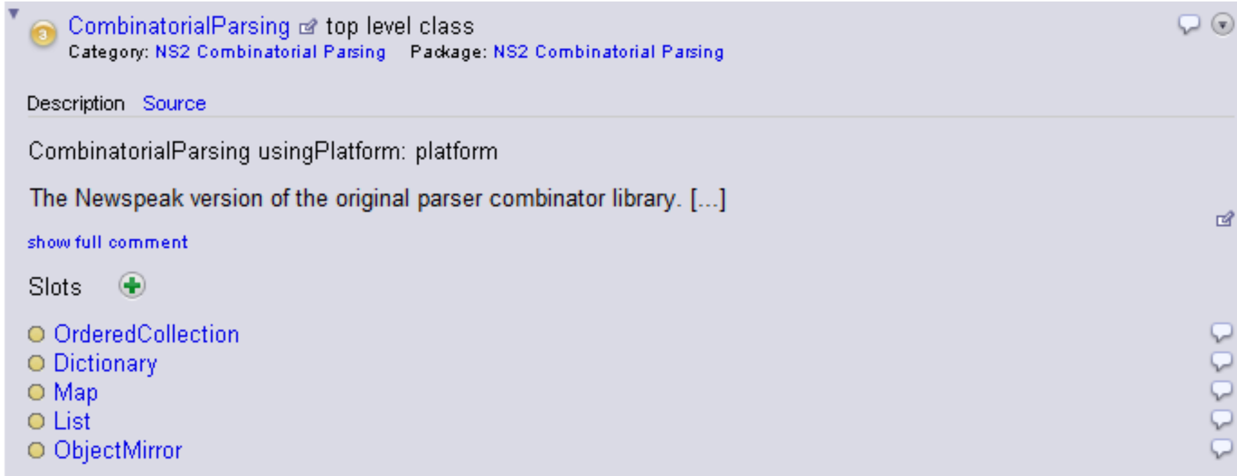


The browser now displays the **CombinatorialParsing** class.

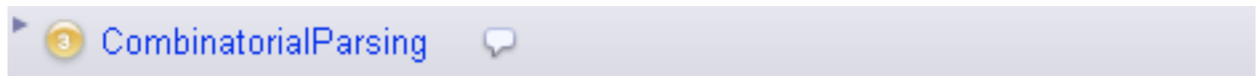
The class presenter has distinct sections for class header information, nested classes, methods and class methods.

Notice the yellow circles in front of each member. These indicate access rights to the member (yellow is the default, for protected; green for public and red for private). Access control is not yet enforced by the Squeak implementation, and we will largely ignore it in this tutorial.

The header information appears at the top with a gray background:



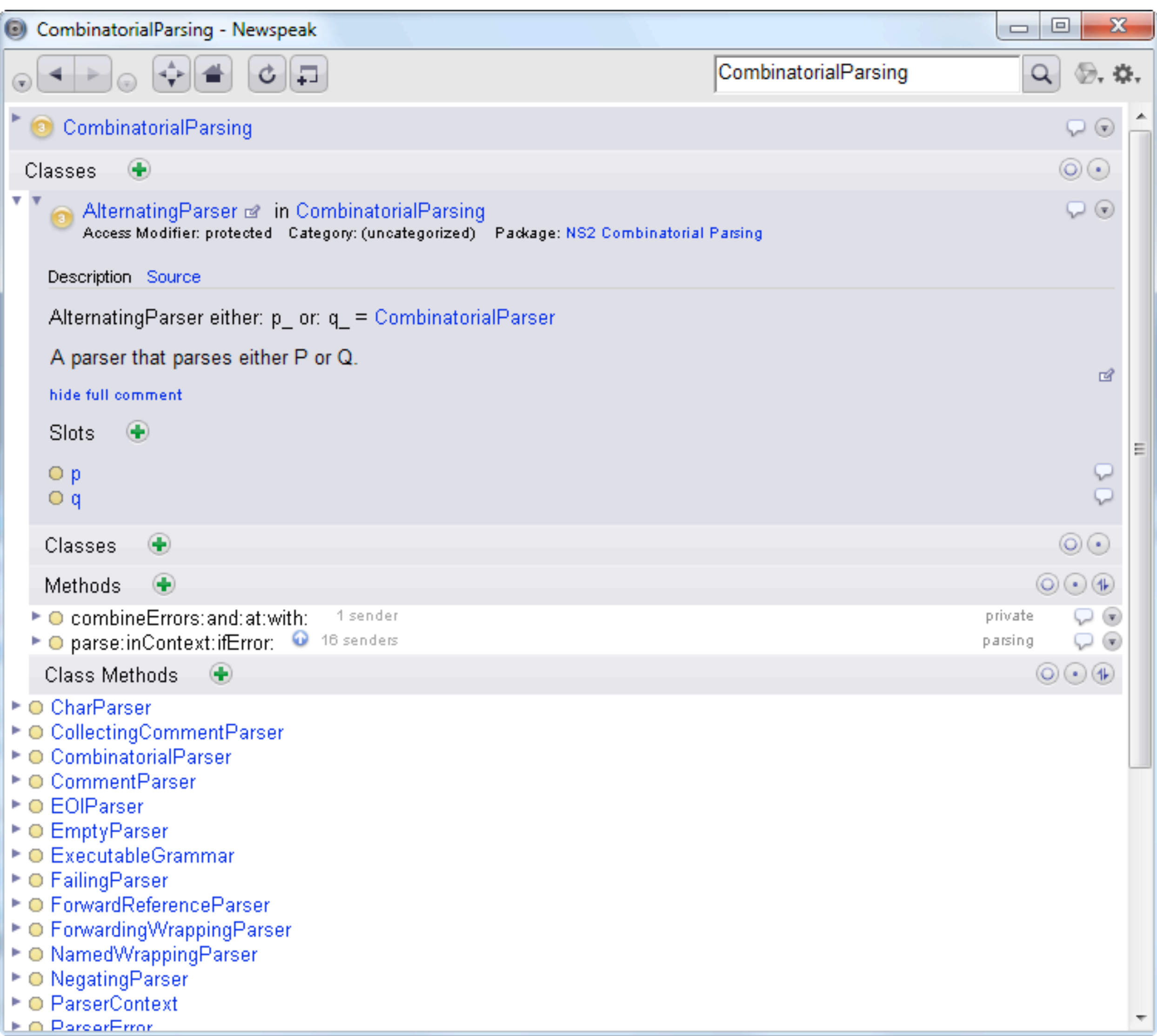
Notice the arrow icon in the top left corner. You can use it to collapse the entire header section:



For now, let's focus on the nested classes section; this is where most of the content is in this particular class. This is characteristic of module definitions. The slots usually represent the "imports" of the module, and there are typically relatively few module methods.

Each of the nested classes can be either expanded in place using the arrow, or linked to by clicking on the class name. Click on the arrow next to **AlternatingParser** to see how a nested class can be displayed in context. This is usually convenient with small nested classes.

You should now see something like this:



Now click on the name of the class **AlternatingParser**. That will take you to a full page display of the class. This is best when dealing with larger classes.

AlternatingParser has no nested classes, but it does have some instance methods. Each of the method names displayed is a link. However, when you click on it, the method opens in place, rather than on a new page.

```
parse: input <ReadStream> inContext: context <ParserContext> ifError: blk <[:String :Integer]> = (
    | rewindPosition |
    rewindPosition:: input position.
    ^p parse: input inContext: context ifError:
        [:msg1 :pos1 |
        input position: rewindPosition.
        context recordFailure: {msg1. pos1}.
        ^q parse: input inContext: context ifError:
            [:msg2 :pos2 |
            context recordFailure: {msg2. pos2}.
            pos1 > pos2 ifTrue: [^blk value: msg1 value: pos1].
            pos2 > pos1 ifTrue: [^blk value: msg2 value: pos2].
            ^combineErrors: msg1 and: msg2 at: pos1 with: blk]]
)
```

Clicking on the link again collapses the method display. This shows that link behavior in Hopscotch can be customized. It isn't very useful to show a method on a separate page. Newspeak methods are usually short, and should never be very long. In any case methods need to be understood in context.

Looking at the method **parse:inContext:ifError:**, you can see the syntax coloring scheme is deliberately low key. The selector is bold, parameters and local slots are bold gray. The message pattern is followed by an equal sign, and the method body appears between parentheses. This will change in later versions.

Looking more closely at the method link, we see that it is followed by various bits of information:

▾ ○ parse:inContext:ifError: 16 senders parsing

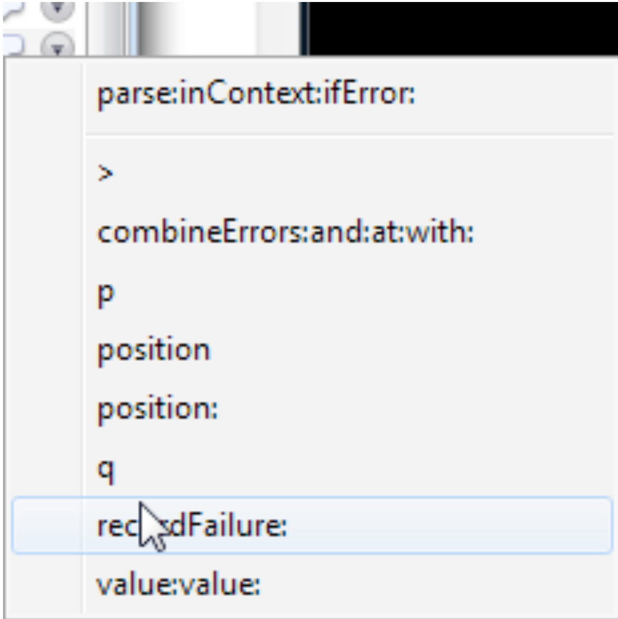
These elements help answer questions like:

How do I browse senders/implementors?

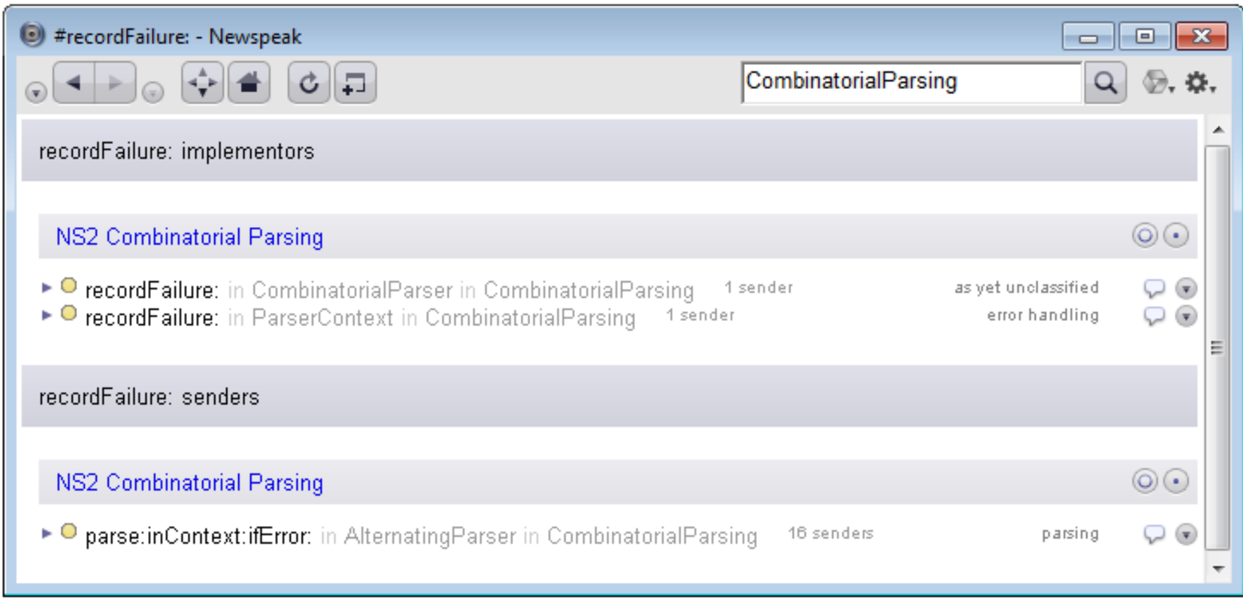
The number of senders of this message may appear immediately to the right of a method name link. You may see this information being filled in when a browser opens a class it has not displayed before. You do not need to wait for this process to complete. Because gathering this information takes significant time, it is done in the background. The sender information is actually a link.

Another way to get at the sender/implementor information is the speech bubble

which generally denotes references to an entity (*who's talking about me*). In the case of a method, clicking on it brings up a pop-up menu with a list of messages, starting with the method's selector, and including all messages used inside the method. Choose an item from the menu to get the list of implementors and senders for that message.



Of course, you can also enter a message selector into the search pane. In all these cases, the result is a list like this



How do I delete a method?

At the very right of a method link, you'll see an icon denoting another drop down menu:

The menu has an option to delete the method.

How do I change the category of a method?

On the right hand side of a method link, to the left of the speech bubble, is the category of the method. Clicking it brings up a menu that lets you choose from all existing categories in the class, or enter a new one.

At the top right of the method section, you'll notice a set of icons:

The open circle opens up all of the methods listed below. The dot icon does the opposite - it will close all open methods. The rightmost icon (with the number sign) controls ordering. By default, Newspeak methods are ordered by their names. Clicking on this icon will toggle the ordering of the methods between name-based and category based. The circle and dot icons are also used in other presenters (such as the class list) with the same semantics.

Why is some code highlighted in red?

There are two possible reasons: suspect implicit messages or syntax errors.

Suspect implicit messages

The browsers underline suspect identifiers and highlight them in red. For example, if an implicit message is not defined in the surrounding lexical scope, it may be indicate a problem - it may refer to an undefined message. On the other hand, it may be an inherited method. Unlike most other languages, in Newspeak one doesn't statically know what methods are inherited, because the superclass isn't statically known - it is determined dynamically by a message send.

The system uses heuristics to try and guess what methods are likely to be inherited. Currently, if any class has been computed based on the class declaration of the method, its superclass is checked for potentially inherited methods. These methods aren't highlighted.

In the case of a new class declaration, no such run time data is available, and so you may see spurious highlighting. Once you've run some code, it will go away. On the other hand, the highlights may indicate typos, or truly undefined methods, such as missing imports.

Syntax Errors

The syntax colorizer analyzes your code as you type. If, at any point, the syntax is malformed, it will mark the downstream code in red.

Debugging

The debugger will mark the site of the active message send in blue.

This concludes our review of browsing methods.

Let's navigate back to the original class. It's time we looked more carefully at navigation in the browser.

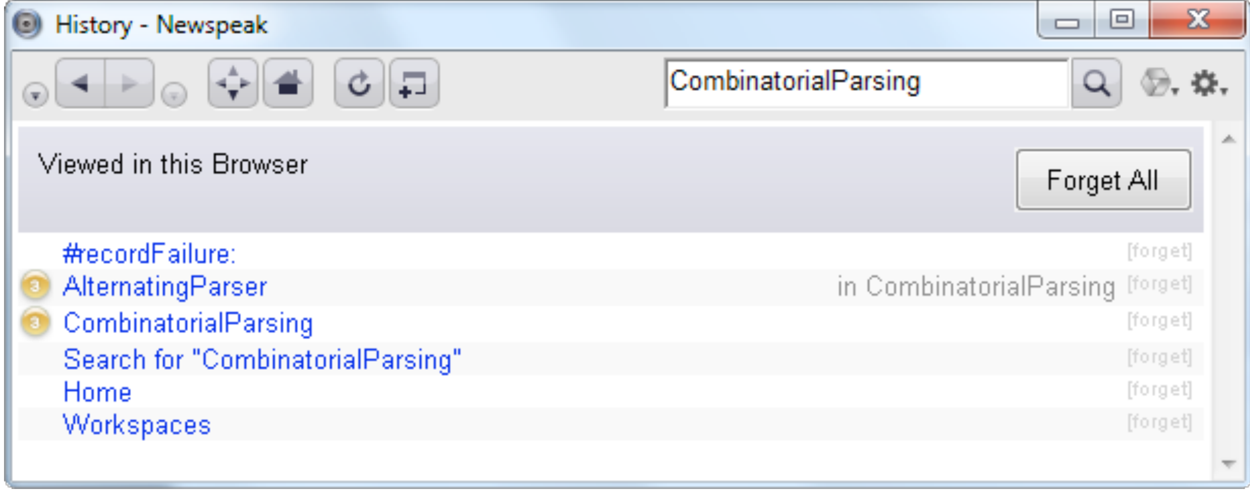
How do I navigate in the browser?

The top of the browser window shows a number of controls, some of which are reminiscent of a web browser. The back and front buttons

behave just as you'd expect. The two little drop down menu icons surrounding these

give you a menu of all the places on the browser stack, below or above the current location. You've already seen the home button. Perhaps the most useful feature is the history button:

It acts much like the history feature in a web browser. It takes you to a page that holds your browsing history for this particular browser, latest location on top:



Click on any link on the page to go to that location. You'll find it just as you left it - with subpanes open, unaccepted edits still preserved etc. This means you can leave any view to go to another, regardless of its state. Interaction with the browser is modeless. It also means that you are never more than two clicks away from any place you've browsed - one click to the history, and one click from there to any prior destination.

In the context of an IDE, this feature is actually much more useful than in typical web browsing, because you tend to build up a small working set of places you visit while working on code - a few classes, methods etc. that you keep bouncing between.

[To the FAQ/Table of Contents](#)

How do I erase my history?

After completing a task, you may want to eliminate it from your history to prevent clutter building up. To erase the history, click the *Forget All* button on the history page.

Forget All

Or just close the browser and open a new one. You can also clean out individual history entries using the [forget] link at the right of each entry.

One more thing while we're on the topic of managing the browser

How do I manage a Newspeak window?

Same as any other window. In Squeak, you'll find these icons on the right hand side of the window title bar:

The green one maximizes, the orange one minimizes, and the blue one provides a drop down menu. Most of the menu entries are standard, but the browser provides unusual options for inspecting the GUI:

How do I inspect the GUI?

Use the browser's meta menu, which is marked with this icon:

The meta menu has two options for inspecting the GUI: *Inspect Window* and *Inspect Application*. The latter opens a new browser showing an inspector on the original browser application. Usually this option is more interesting, as that is where the application logic resides. The former is similar, but the inspector is opened on the *Window* object representing the browser window. You can drill down and find what code implements a given GUI element. In addition, many presenters have an *Inspect Presenter* option in their drop down menu, which opens up an inspector on the presenter object.

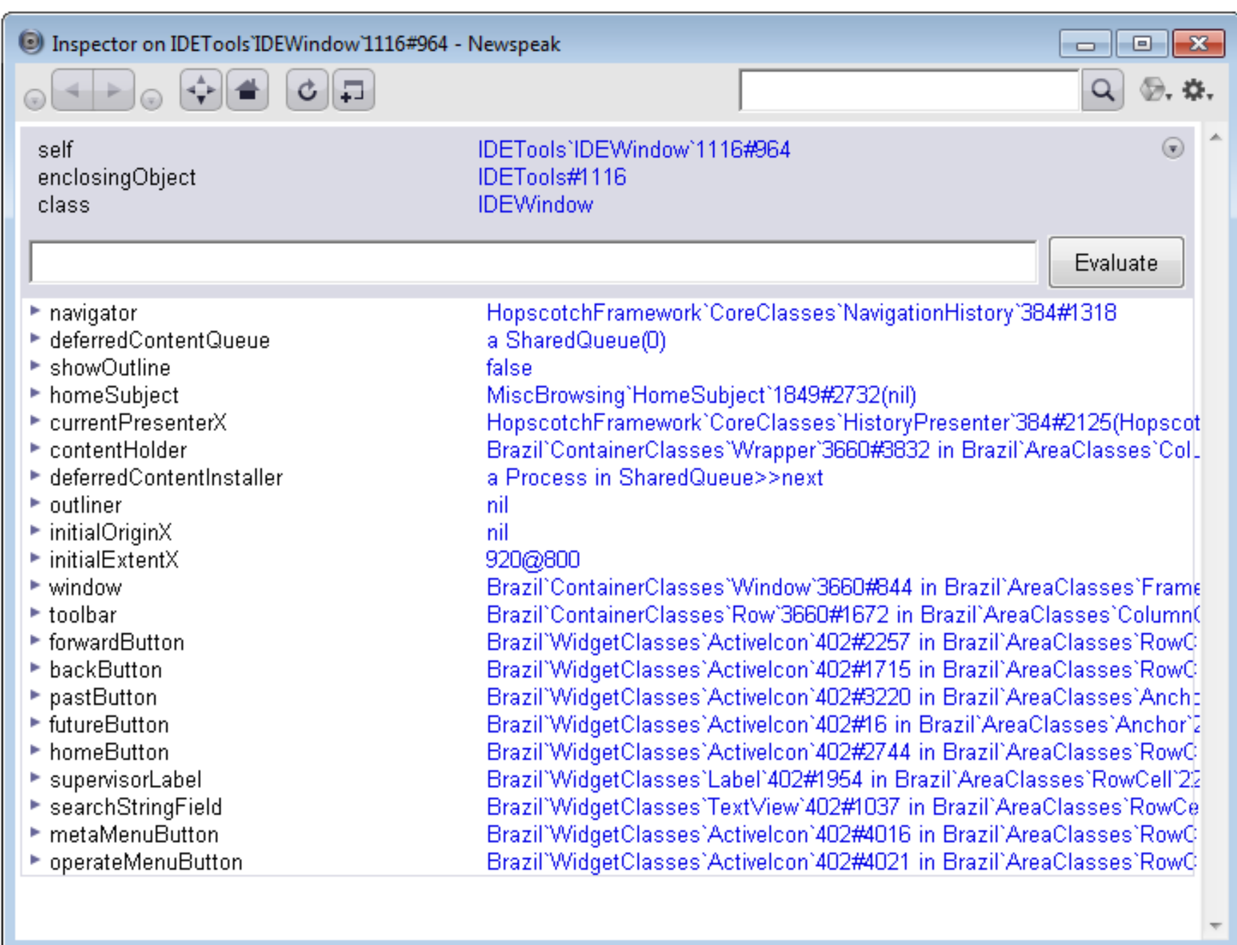
[To the FAQ/Table of Contents](#)

Of course, to utilize these, we need to understand how to inspect an object.

How do I inspect an object?

There are several actions that will open up an inspector. Besides the menu options described above, the *Inspect Mixin* menu item available in [the class header presenter](#) will open up an inspector on the corresponding mixin object. However, the most general way to get an inspector on an object is to [evaluate an expression](#) in a [workspace](#).

However you open an inspector, you're likely to see a view such as the following:



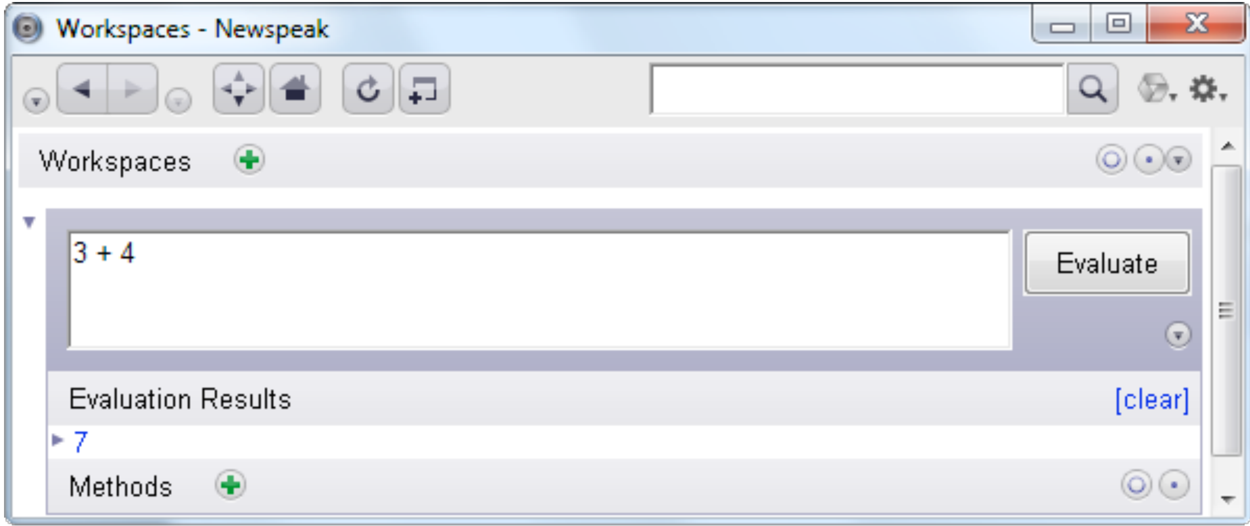
Each slot of the inspected object is listed in the inspector. It is prefixed by an arrow icon that allows you to view an inspector on the value of the slot in context . Following the

slot name is a description of the slot value, which is itself a link to an inspector on the slot value. The gray area at the top of the inspector gives a description of the object being inspected, and a link to its class and its enclosing object. Clicking on the class brings up a class presenter on the class, not an inspector on the class object. If you need to inspect the class object, you can do that by choosing *Inspect Class* from the drop down menu, whose icon you can see at at top right:

How do I evaluate code?

The inspector has an interaction pane in which you can evaluate expressions in the context of the object. Type in an expression (*3 + 4* in the example below) and select it. Then hit Ctrl-S (or cmd-S on a mac). A link to the result of the expression is added. You can expand it in place with the arrow icon, or link to it. You can also use the *Evaluate* button on the right instead of Ctrl-S/cmd-S.

If you don't make a selection explicitly, the current line will be evaluated. This is useful for short expressions. This design allows you to have multiple code snippets in an interaction pane and evaluate them as needed. After evaluation, the selection is highlighted so you can see what was evaluated



To the FAQ/Table of Contents

Note: If you split your expression among several lines, make sure you select it in its entirety.

Of course, you can always evaluate code in an ordinary Squeak workspace - but that would be Smalltalk code, not Newspeak code. However, Newspeak module definitions can be instantiated from Smalltalk; they are available as Smalltalk globals. Since they are stateless, nothing ungood can happen.

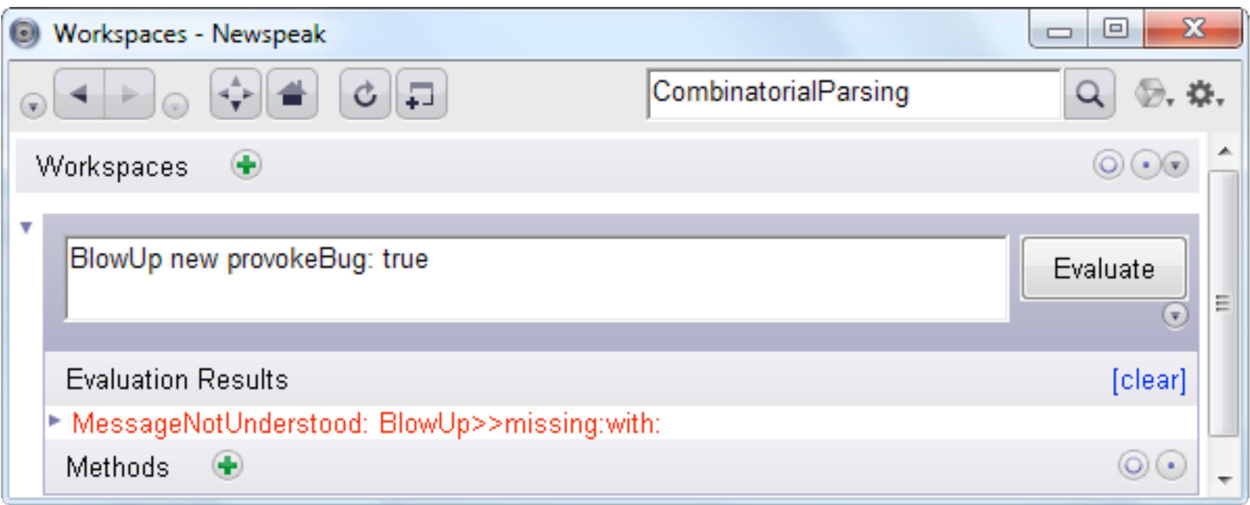
Actually, there is something ungood that can happen - our evaluation may raise an exception.

What if evaluation raises an exception?

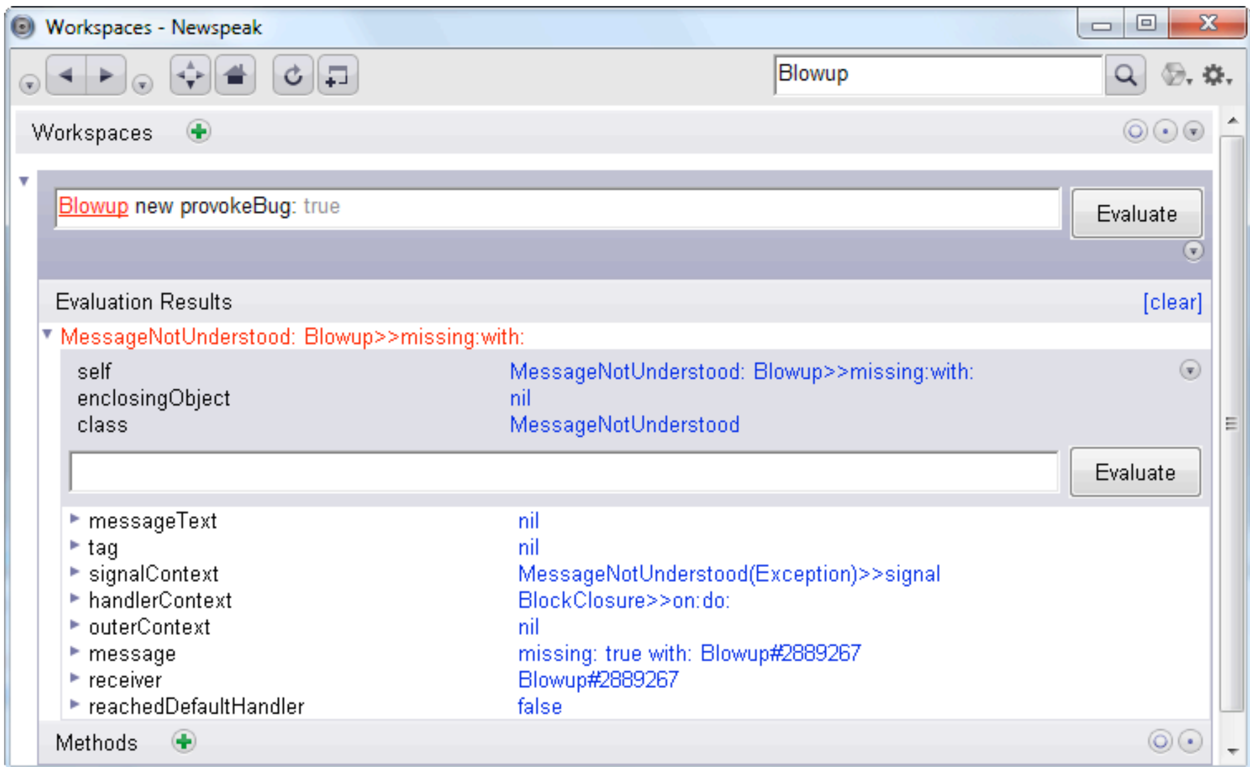
In that case, rather than a link to the result, you get a link to the debugger. Assume that we have a class **BlowUp**, with the following method

```
provokeBug: b <Boolean> = (
  ^missing: b with: self
)
```

Calling *BlowUp>>provokeBug:* from a workspace gives us the following display:



We can look at the exception in more detail by expanding the item MessageNotUnderstood BlowUp>>missing:with: as shown below:



If we want to debug this computation, we need to open a debugger on it.

What if I want to get at the platform or IDE namespace from an inspector?

Sometimes you need to evaluate an expression that involves an object that isn't accessible from the current scope. Perhaps you need to get at a module you didn't import.

In this situation, you can use a *workspace*. A workspace is an object in the IDE that gives access to the IDE's top level namespace, as well as to the platform object and its members.

In inspectors, the backtick (`) is allowed in expressions, and evaluates to a workspace. So you can send messages like

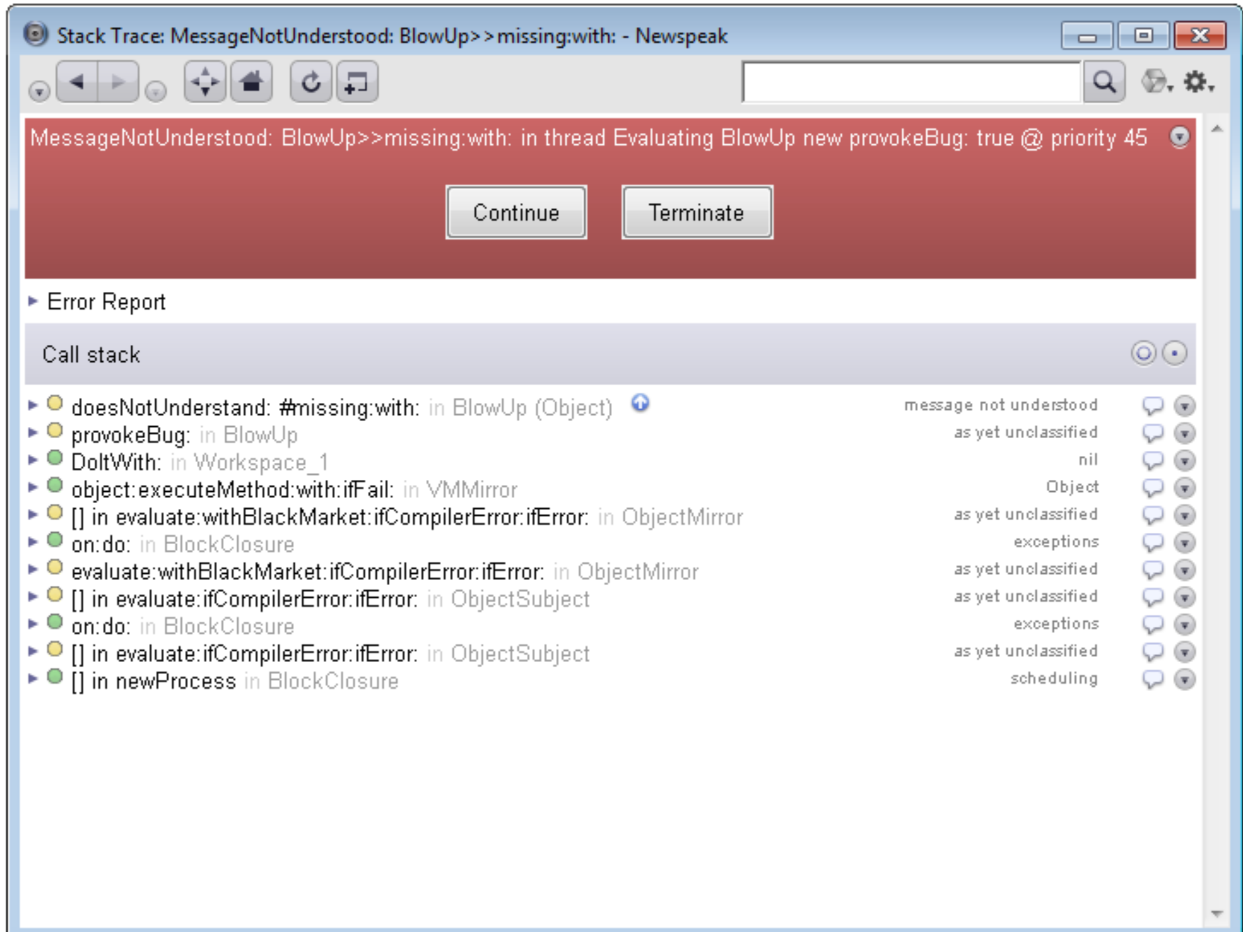
` *collections (* note the space after the backtick - it has to be there *)*

and get at the collections module of the platform, even if you forgot to import it. This won't work in regular code of course - the backtick was chosen precisely because it is not legal Newspeak syntax. And we don't want to undermine Newspeak's modularity by

providing such a back door. However, you the programmer are all-powerful during development. You can get anything you need through the workspace.

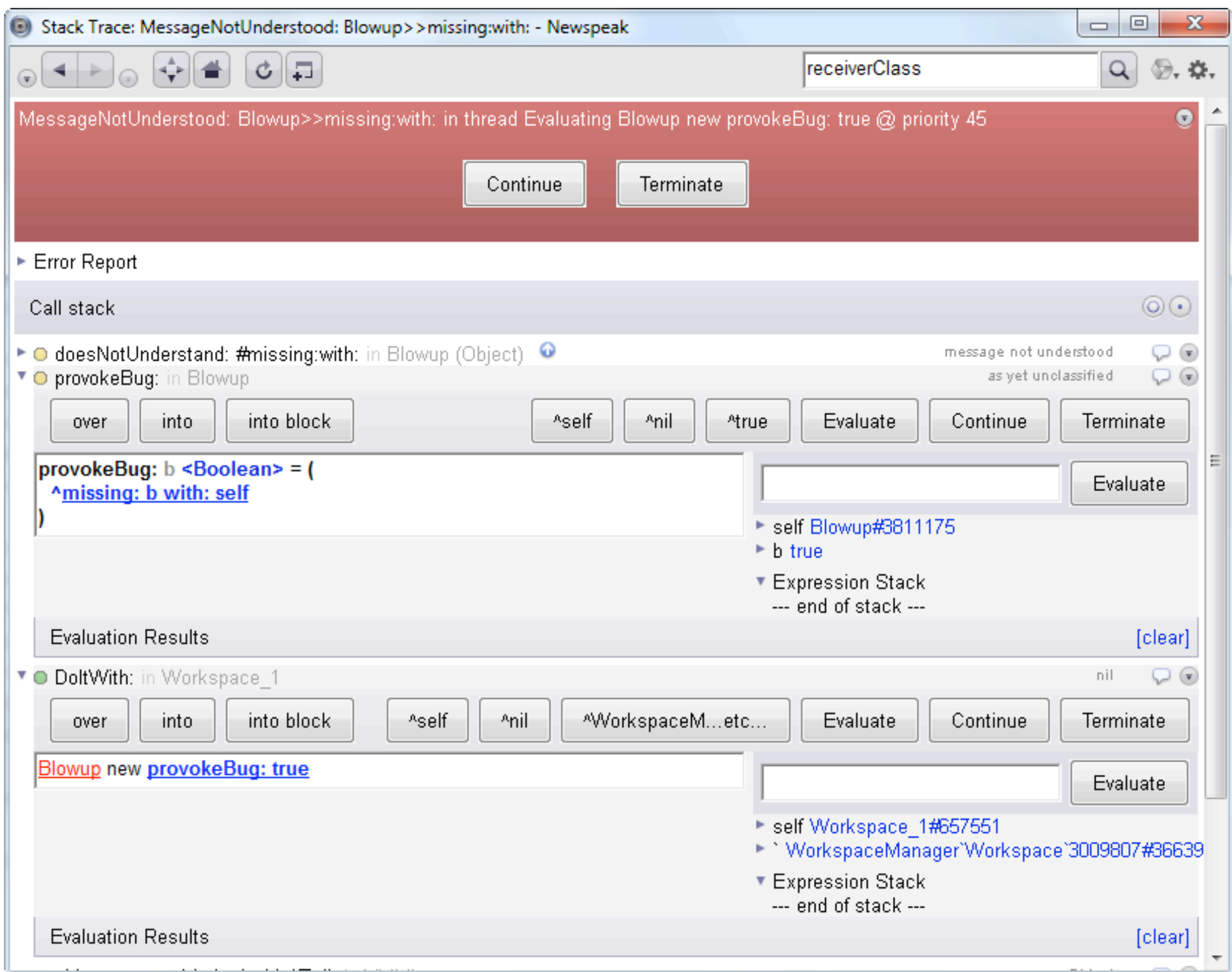
How do I run the Newspeak Debugger?

You will encounter the debugger if an uncaught exception occurs during execution. If you evaluate an expression that results in an uncaught exception in a Hopscotch workspace or object inspector, the inspector will catch the exception, as described above. The text describing the exception also serves as a link that opens up a debugger page on the stack of the failed computation. Click on the link, to get a debugger in a **new window**. The debugger displays the call stack.



Each activation frame is expandable to an activation presenter which allows you to see the method in question and the state of the frame.

The method is shown on the left, and above it buttons that control stepping etc.



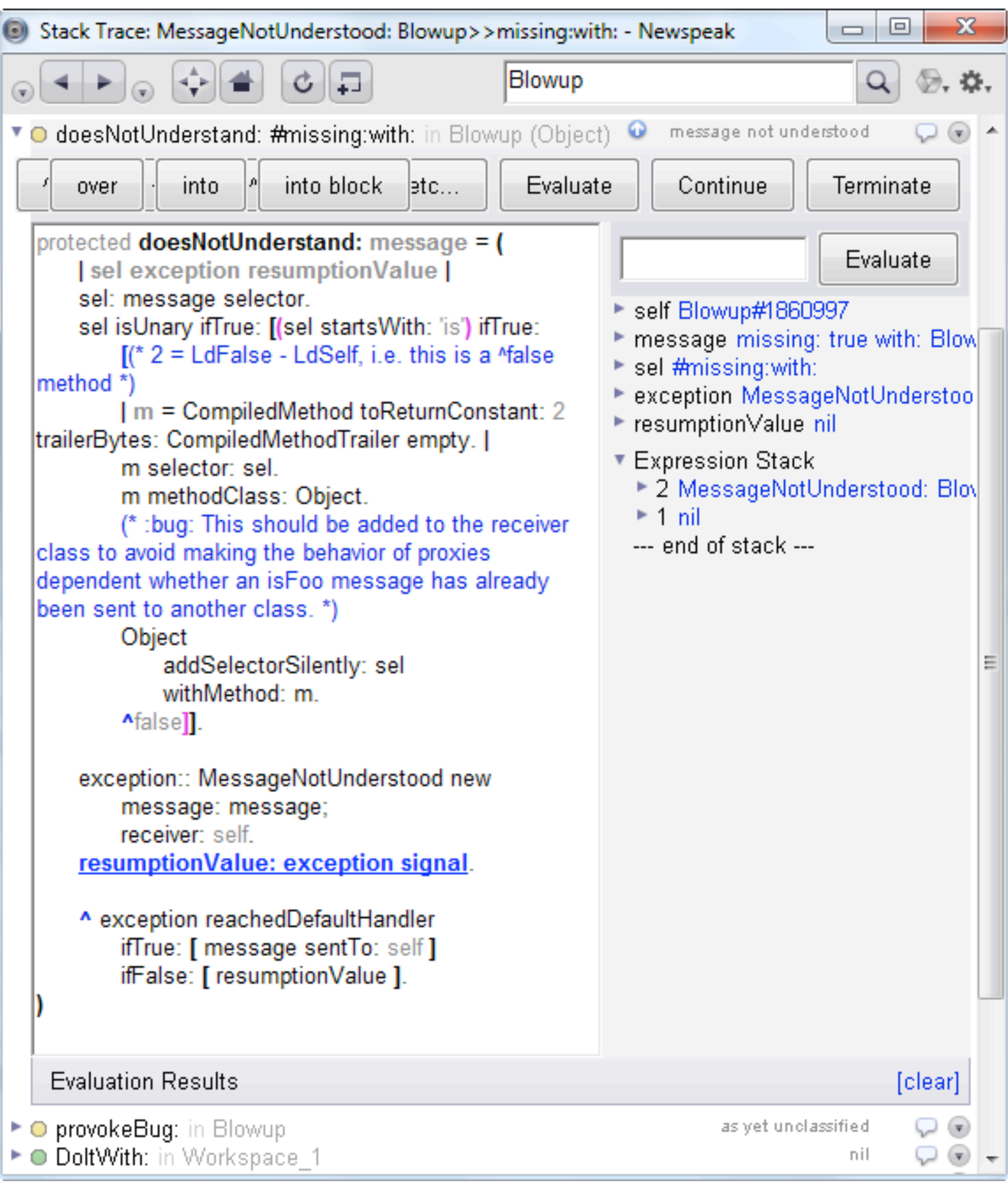
The frame state is shown on the right, and includes the receiver, any parameters and local slots, and the contents of the expression stack. Each of these is identified by name, and has a link to an inspector on the value next to it. These links are expandable in place via arrow icons.

Near the top right hand corner of the activation presenter is an evaluator, where we can evaluate expressions in the context of the activation, just like in an object inspector.

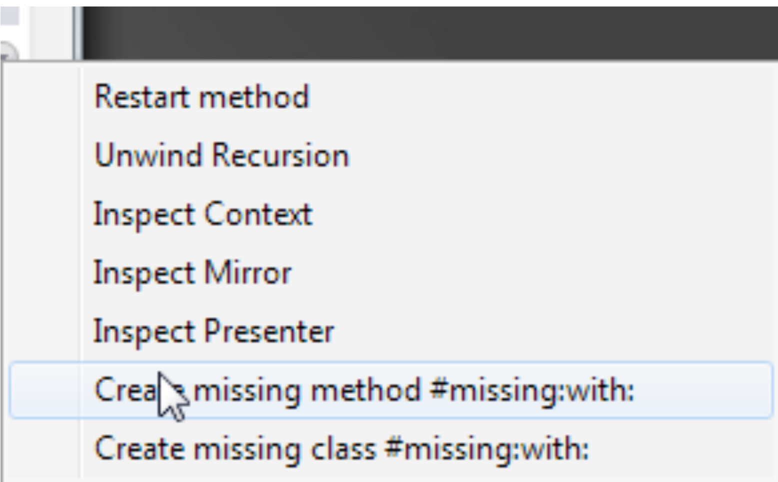
Evaluating an expression produces a link to the result. To the right of that link will be another link, return it. If you click on it, the object in question will be returned as the result of the method, irrespective of its normal course of computation.

You may use the method presenter on the left as you usually would; you may edit the method and save the changes for example. This will discard all activations above the edited method, and restart the method at its beginning.

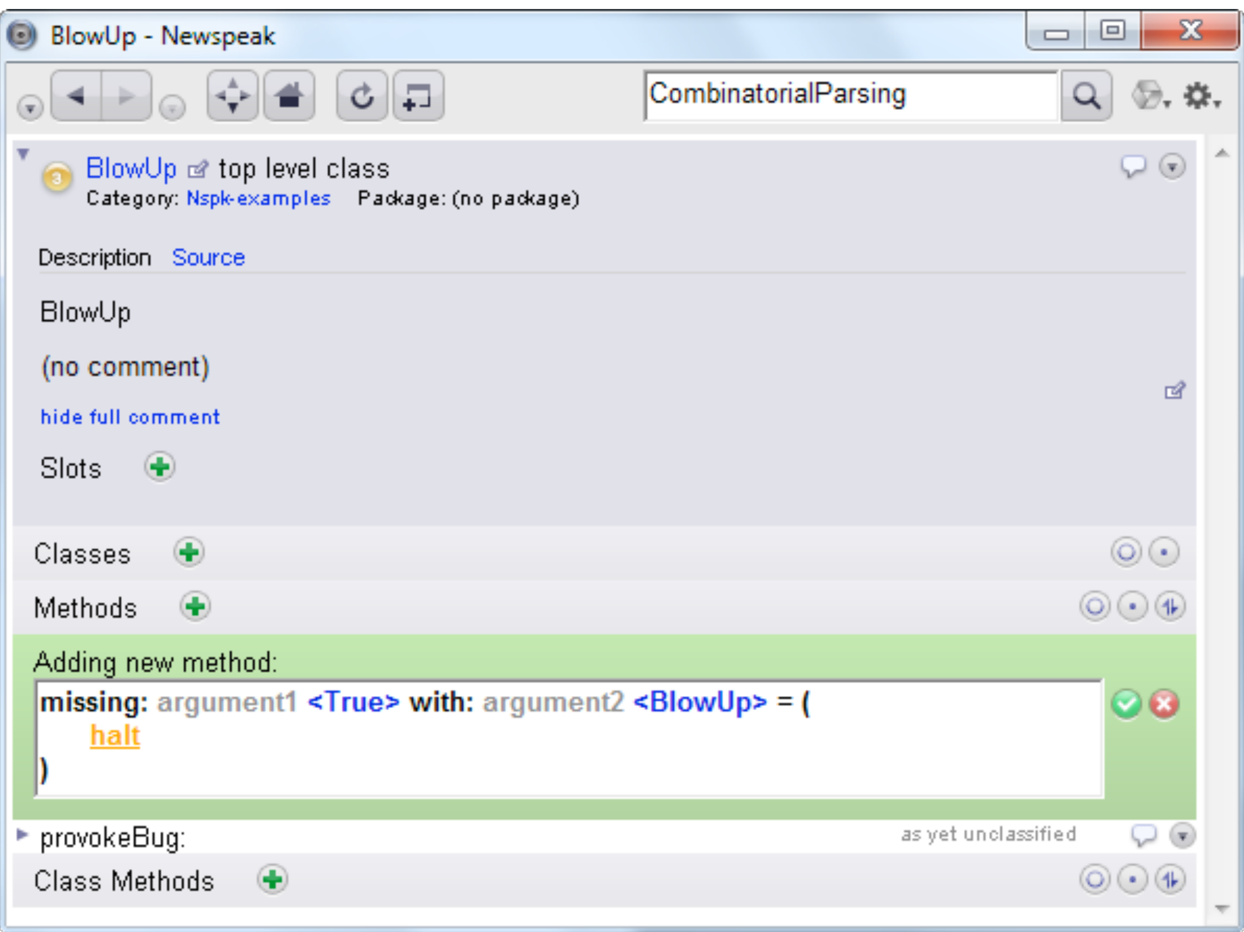
In our case, we can see that the problem is that **missing:with:** is not defined. Expand the top frame on the stack, to see the **doesNotUnderstand**: method.



If the activation involves the method **doesNotUnderstand**:, you will have an option to declare the missing method semi-automatically.



You will be transported to a class browser page for the class of the receiver, with an editor open on a new method with the appropriate name. The system will also guess types for the parameters based on the actual arguments that had been passed.

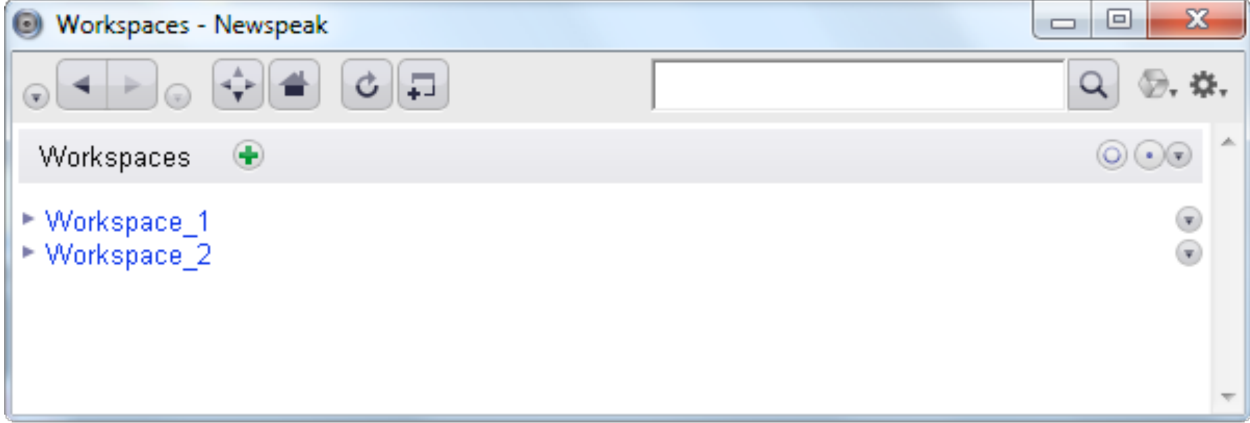


Once you've edited the method to your satisfaction and saved it, you can return to the debugger and/or press the *Continue* button at the top to proceed with the computation..

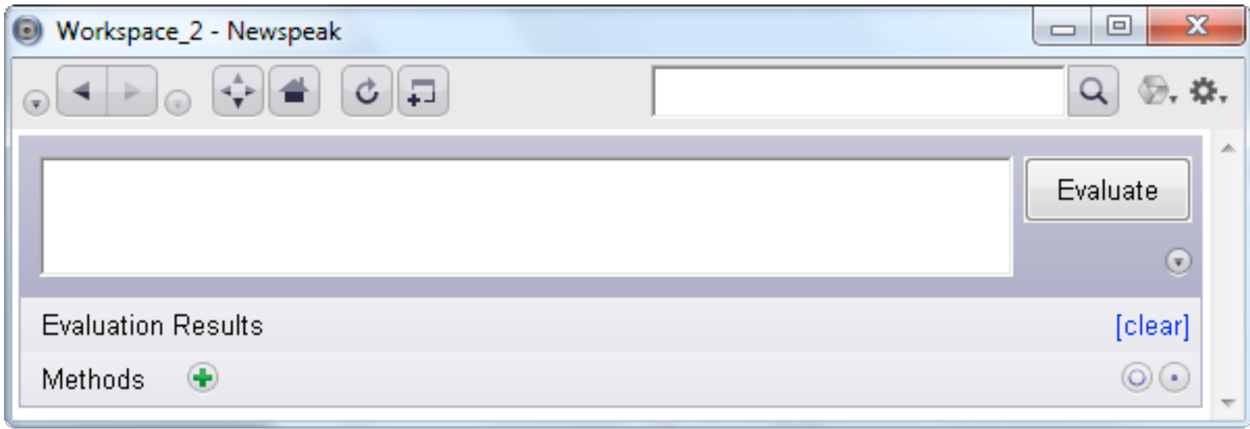
You may also restart the method using the activation's menu. This menu also provides the option to unwind recursive calls of a method. This is useful when you have stopped in an infinite recursion; you want to pop all recursive activations off the stack and get back to the very first activation of the method, where you can correct the method and then proceed with execution using the *Continue* button.

How do I open a workspace?

On the home page, you can click on the link labeled Workspaces. This will take you to the workspaces page. If there are no workspaces yet, one will be created and opened in place. If there is exactly one workspace it will be opened in the same way. Otherwise, you will be presented with a list of all existing workspaces to choose from.



You can create new workspaces by clicking the plus button. Workspaces are identified by cheerful names, such as *Workspace_1* etc. You can then navigate to a given workspace by clicking on its link in the workspaces page



A workspace allows you to evaluate code and provides you with access to the IDE's top level namespace. The IDE namespace gives you access to all top level classes loaded into the IDE (both Newspeak classes and Squeak classes). It also supports access to the Newspeak platform object via the name **platform**. In addition, you can use the name **ide** to gain access to the IDE namespace itself.

Finally, workspaces provide direct access to all the modules the Newspeak platform provides; e.g., the collections module is available via the name **collections**, the streams module via the name **streams** etc.

Each workspace has its own unique class. You can access it via the drop down menu located just under the *Evaluate* button. This allows you to define slots (sometimes known as workspace variables in other systems) for the workspace. Likewise, you can define workspace specific methods or even nested classes. All of these are treated exactly like members of any other class.

You can also define methods for the workspace class directly in the workspace: the **Methods** section at the bottom of the workspace works exactly like it does in a class browser. More on this below.

Back to CombinatorialParsing

Let's navigate back to the **CombinatorialParsing** class. We've seen how methods and nested classes work, but we haven't really investigated the class header. The header includes the class comment and slot definitions.

How do I edit the class header?

At the left hand side of the class header section is a link that is labeled as Source. Clicking it transforms the presentation of the header, thus:

One can edit the header source and so add, remove or rename slots, change the primary factory, edit the class comment, edit any initialization code that follows the slot declaration section etc.

You can switch back to the default view of the header by clicking on the Description link.

This concludes the sections dealing with browsing existing code. Now we consider creating new code.

How do I create a new class?

Currently, classes are created within class categories, which themselves are usually created in packages. Class categories are a Smalltalk legacy that we hope to drop in due course. Likewise, as our source management facilities mature, packages are likely to go extinct as well.

How do I create a package?

Packages are created on the source control page. It is also possible to create a category that is not part of a package.

How do I create a class category?

Go to the System Source page (linked to from the home page) , and find the package to which you wish to add the category. The display shows a list of packages on the left, and for each package, a list of class categories it contains on the right. Just to the left of the category list is a link marked add. Click it and fill in the name of the desired category.

Since we won't be adding our category in a package, we will add to the set of unpackaged categories listed at the top:

[To the FAQ/Table of Contents](#)

System Source - Newspeak

NSStreams

Packages and Categories

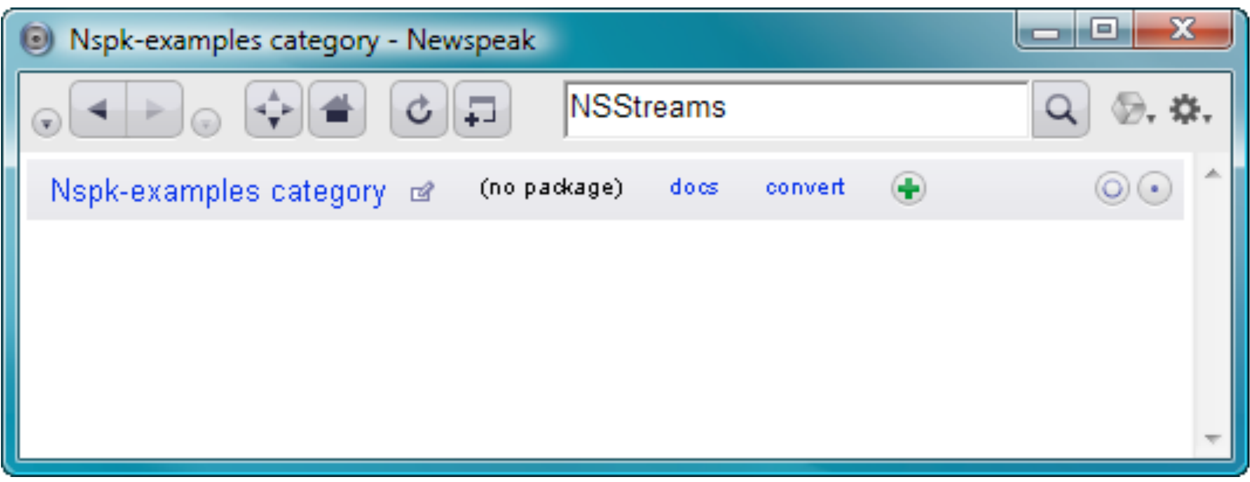
(unpackaged)	add	New category: Nspk-examples
38Deprecated	add	
Balloon	add	Balloon-Collections Balloon-Engine Balloon-Fills Balloon-Geometry Balloon-Simulation
BalloonMMFlash	add	
Brazil	add	Brazil-Plumbing Brazil-Visuals-Abstract Brazil-Visuals-Containers Brazil-Visuals-Widgets Brazil-Menu Brazil-Areas Brazil-Mapping Brazil-Manipulation Brazil-Manipulation-nested Brazil-Manipulation-nested-nested Brazil-Plumbing-nested Brazil-Plumbing-nested-nested Brazil-Plumbing-nested-nested-nested Brazil Brazil-nested Brazil-nested-nested Brazil-nested-nested-nested
BrazilForMorphic	add	BrazilForMorphic-Mapping BrazilForMorphic-CustomMorphs
BrazilForWindows	add	BrazilForWindows-Mapping BrazilForWindows-Mapping-nested BrazilForWindows-Mapping-nested-nested

Accept the category name by clicking on the green icon:

You can also just type Ctrl-S (or Cmd-S on a mac) in the text pane. If you wish to cancel the operation, click the red icon:

You will be asked to confirm the cancelation

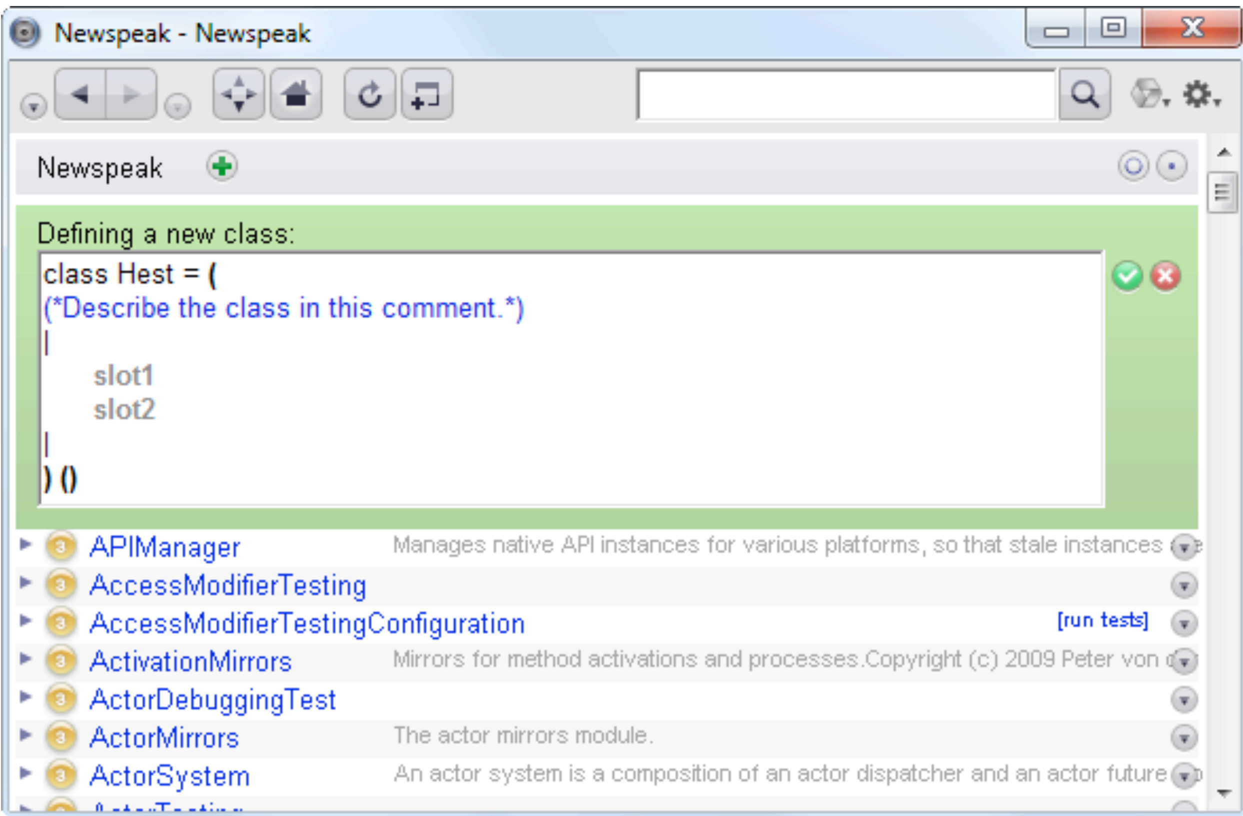
Now we can go look at our category



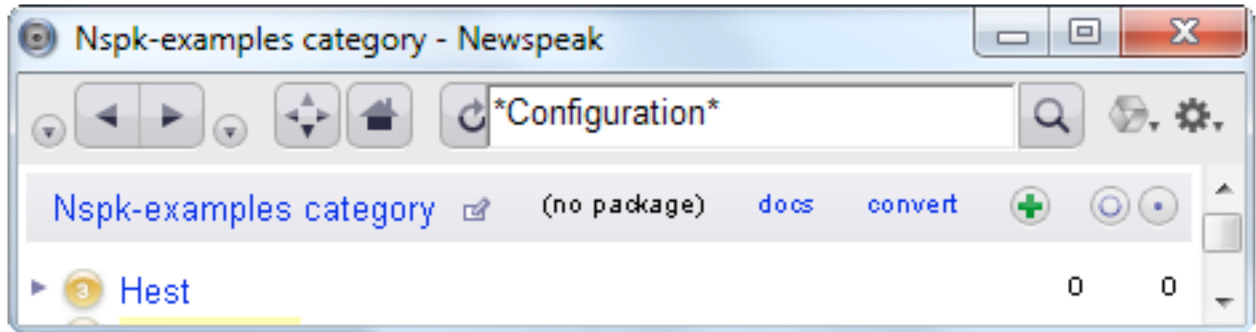
At the moment, there are no classes in the category. Let's add one, by clicking on the plus icon

You'll see an editable pane with a template of a class header.

To the FAQ/Table of Contents

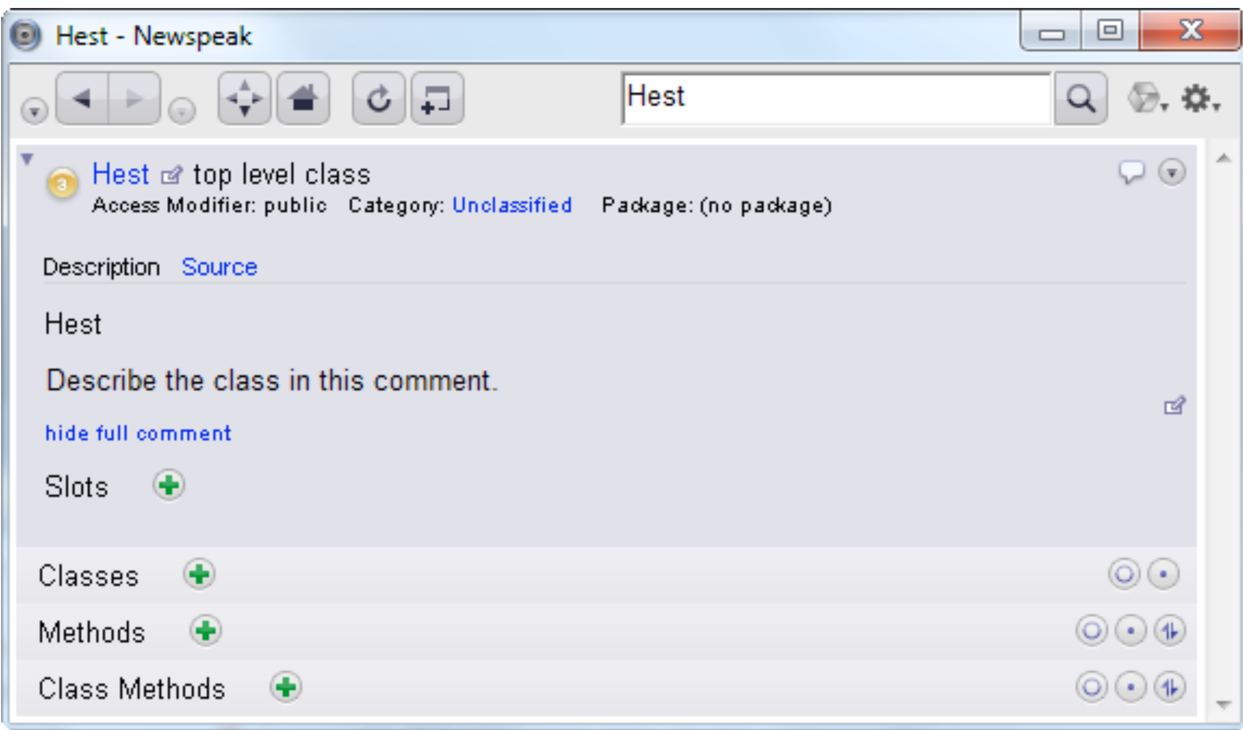


We've named our class **Hest**, which is Danish for horse. Don't ask why. We will erase the slots from the template, and accept, as usual, by typing ctri-s (or cmd-s on macs) or by clicking the green icon. Now our category is populated with a single class:



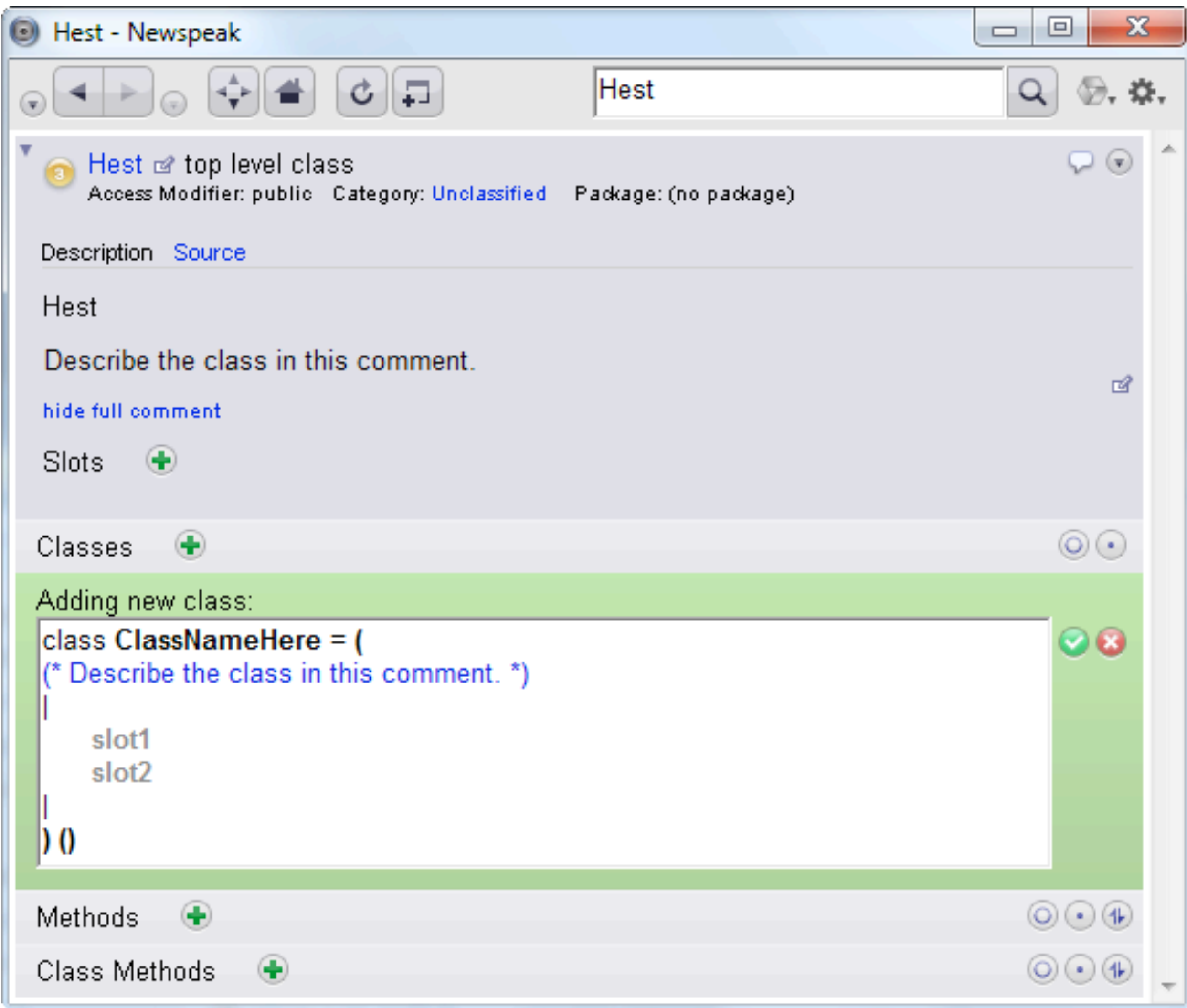
Click on the link to **Hest**. The view you see should look like this:

[To the FAQ/Table of Contents](#)



Now let's add a nested class, by clicking the plus icon next to the word **Classes.**

[To the FAQ/Table of Contents]



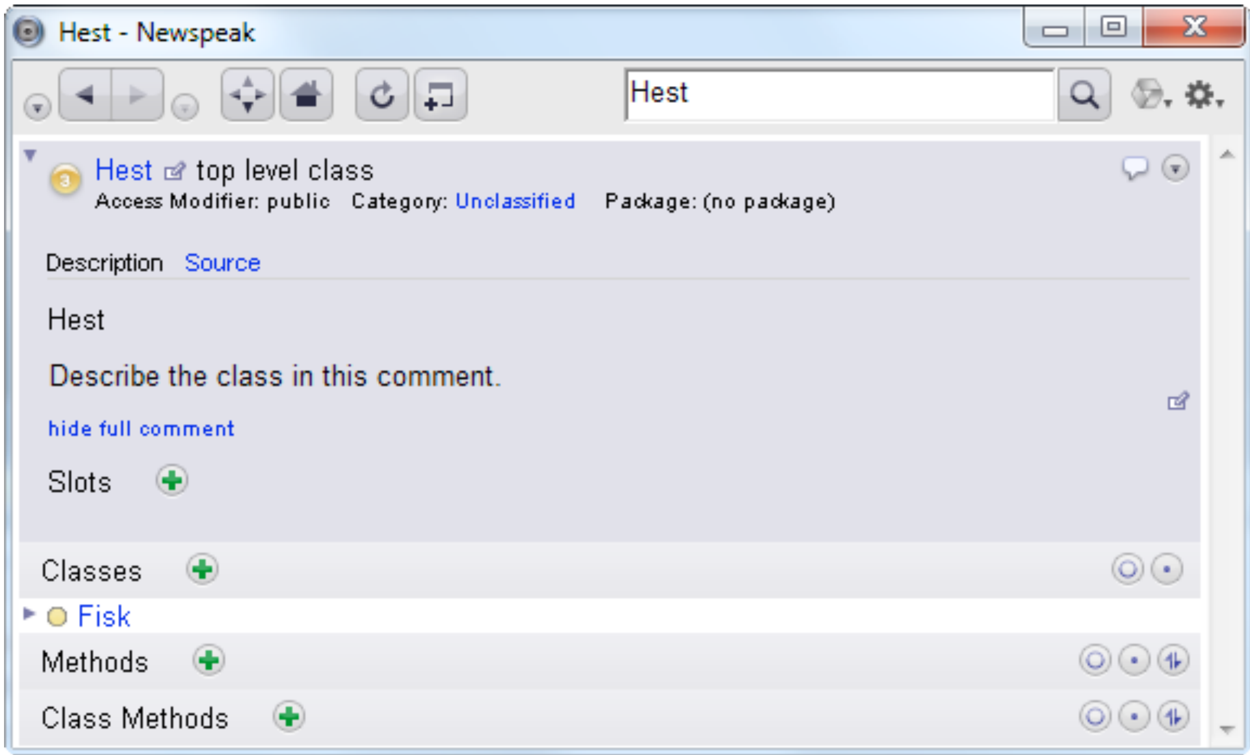
Edit the template to represent the desired nested class Our nested class will be called **Fisk** (Fish in Danish) in honor of the Scandinavian school of object orientation, which invented nested classes, virtual classes and classes in general.

Adding new class:

```
class Fisk = (
(* Describe the class in this comment. *)
) ()
```

Save the code. Your display now looks something like this:

To the FAQ/Table of Contents



Now we'll add some methods.

How do I add a method?

Click the plus icon in the **Methods** section (or **Class Methods** section, if you want to add a class method). Then edit the displayed method template.

Methods

Adding new method:

```
messageSelector = (

)
```

We'll call our method bar. It will return the number 91.

To the FAQ/Table of Contents

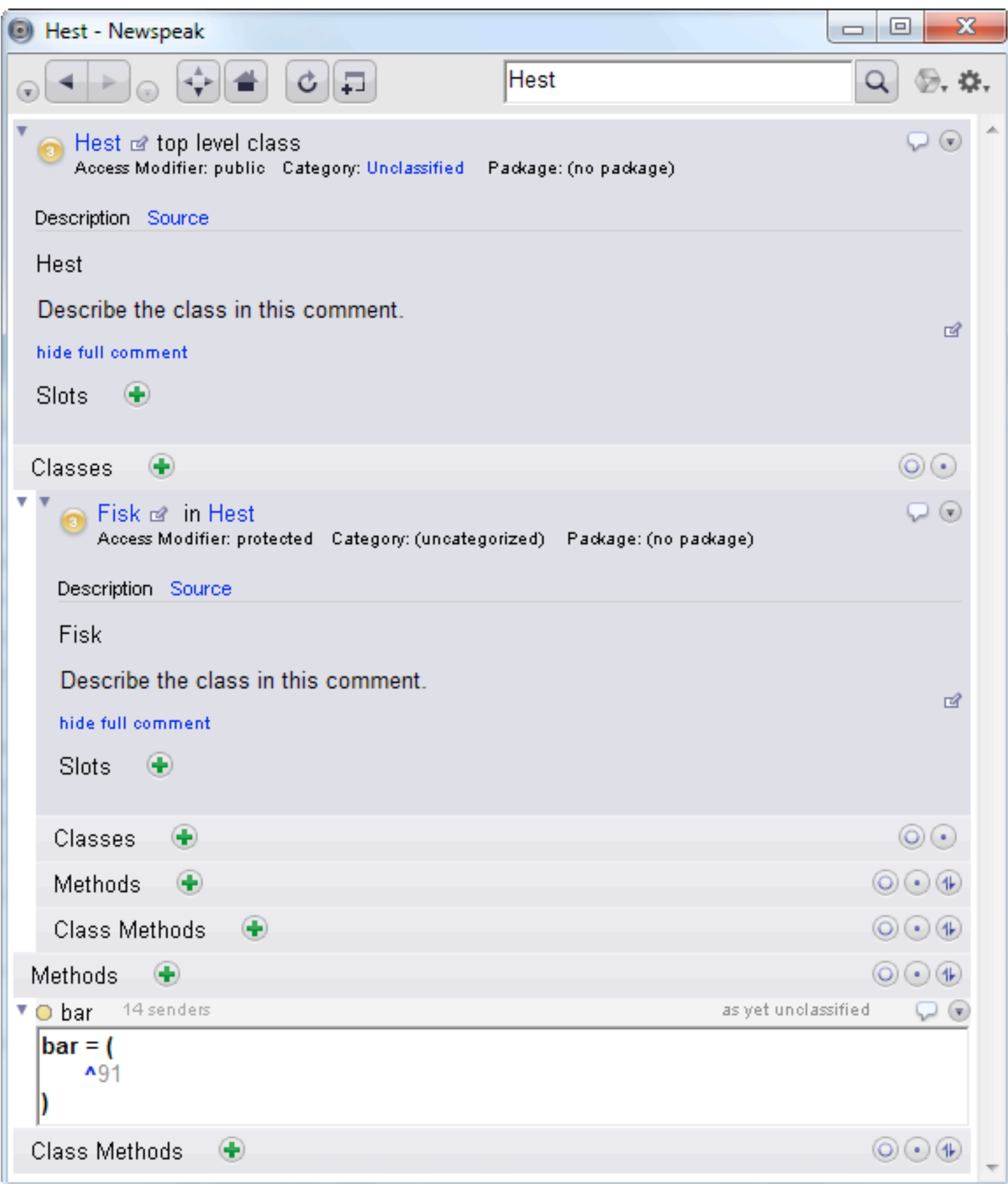
Methods

Adding new method:

```
bar = (
    ↑91
)
```

Accept the changes.

Now let’s turn our attention to *Fisk*. Add a nested class named **Hest** to **Fisk**. So we now have classes **Hest**, **Hest`Fisk**, and **Hest`Fisk`Hest**. We use the backquote character as a separator between the names of classes and their nested classes.


Hest - Newspeak
Hest
Hest top level class
Access Modifier: public Category: Unclassified Package: (no package)
Description Source
Hest
Describe the class in this comment.
hide full comment
Slots
Classes
Fisk in Hest
Access Modifier: protected Category: (uncategorized) Package: (no package)
Description Source
Fisk
Describe the class in this comment.
hide full comment
Slots
Classes
Methods
Class Methods
Methods
bar 14 senders as yet unclassified
bar = (
^91
)
Class Methods

To the FAQ/Table of Contents

How do I delete a class?

Use the drop down menu at the right of the class header.

How do I add a slot?

To add a slot, edit the class header.

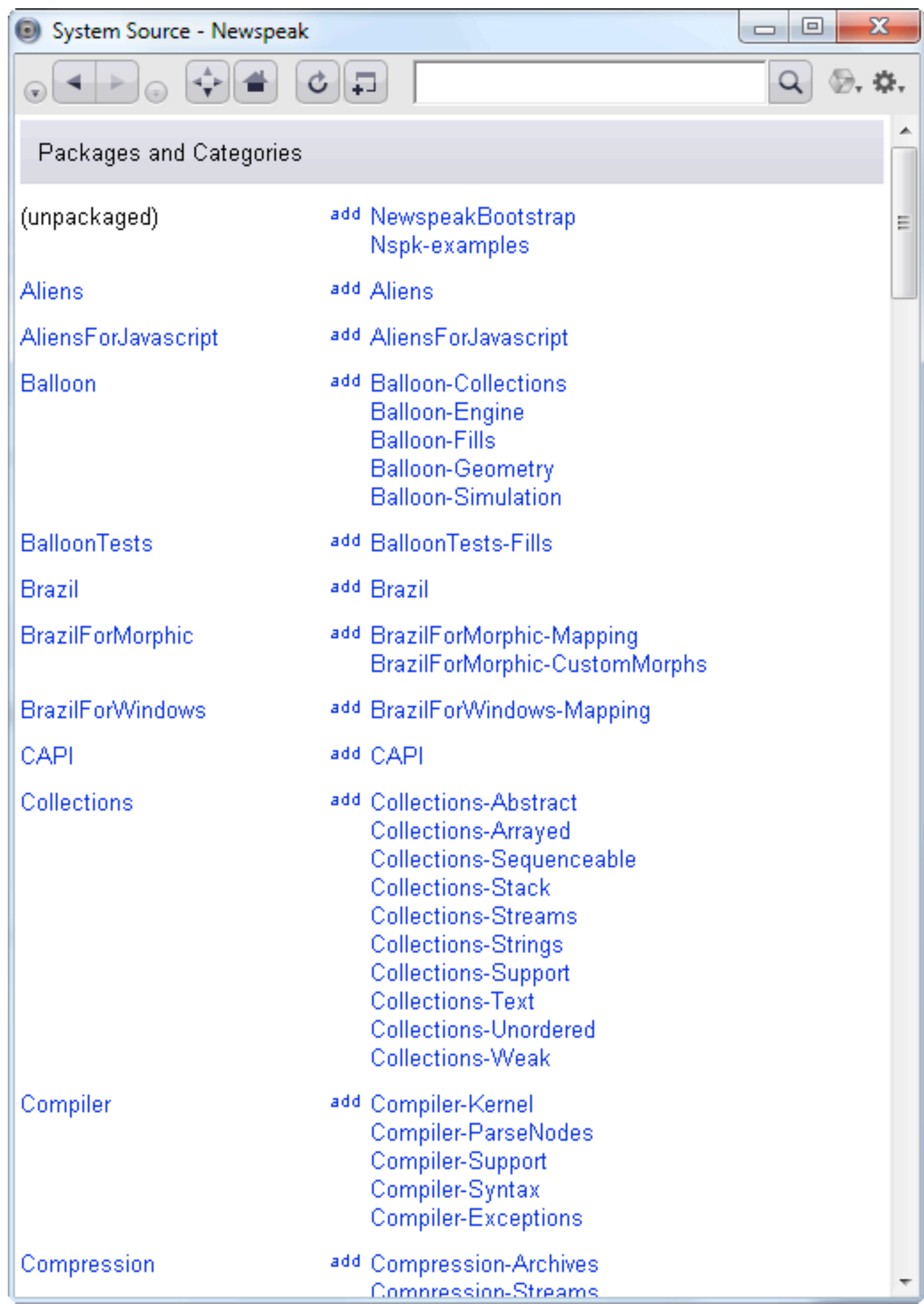
How do I delete a slot?

Edit the class header.

How do I find a class by browsing the IDE namespace?

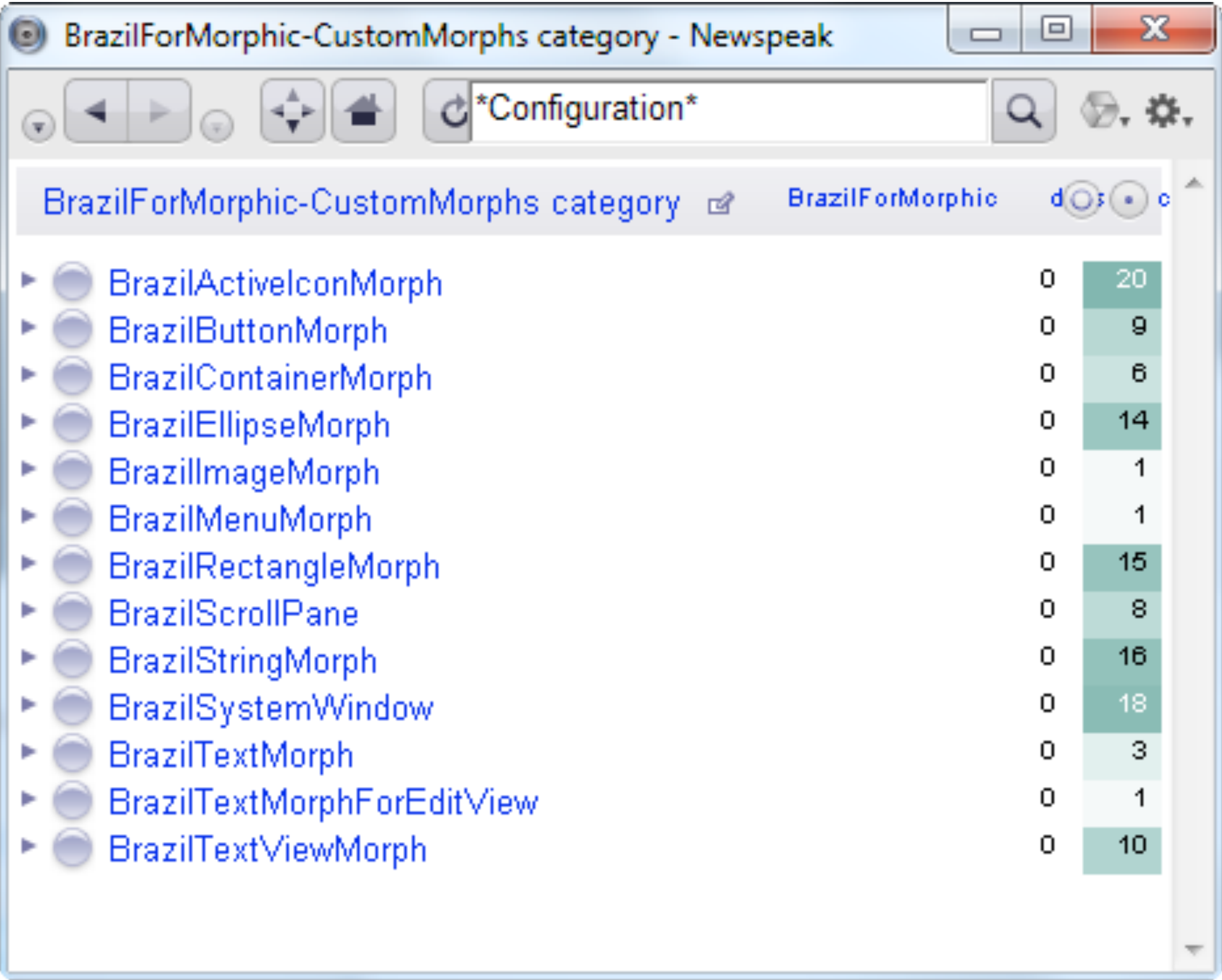
From the home page, you can click the link marked System Source. You'll see a listing of all the packages loaded in the image on the left. For each package, the right hand

To the FAQ/Table of Contents



column shows the list of class categories within. All the listed items are links. You can navigate to any category in this way, and from there to any specific class in the category.

Choose the category *BrazilForMorphic-CustomMorphs*:



This is an interesting choice, as it consists of Smalltalk classes. You'll note that next to each class, we find, on the right, two columns. The rightmost column tells us how many methods the class has; the darker the green background behind that number, the more methods are in the class - so you can tell at a glance what the big classes are.

The remaining column gives you an estimate of the number of subclasses the class has. This is just an estimate, as it can be pretty hard to tell this in Newspeak. These numbers have a progressively darker blue background the more subclasses there are - so you can quickly get a sense of which classes are important.

To the FAQ/Table of Contents

How do I add a Category

Accept the category name by clicking on the green icon:

You can also just type Ctrl-S (or cmd-s on a mac) in the text pane. If you wish to cancel the operation, click the red icon:

You will be asked to confirm the cancelation

Now we can go look at our category

How does Source Control Work?

The Hopscotch IDE features an integrated source code management system called MemoryHole, that currently runs on top of Mercurial (**hg**) or Git.

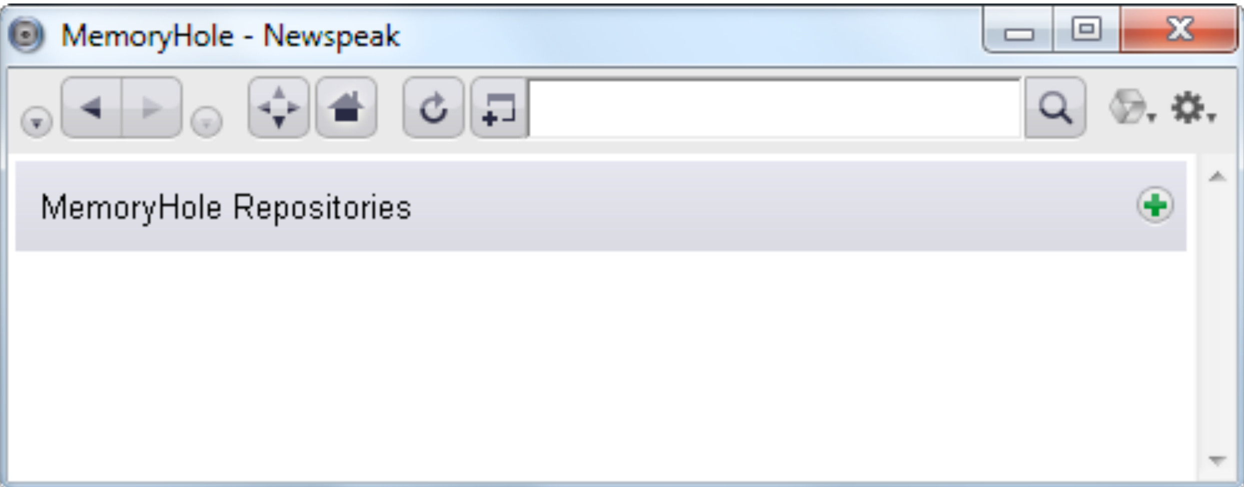
Of course, if MemoryHole doesn't suit you, you may always choose to save and load classes to/from files, using the version control software of your choice. To be honest, you may well be the first to go that route.

To the FAQ/Table of Contents

The MemoryHole Page

You get to source control from the link marked Repositories on the home page. You'll need to have Mercurial or Git installed for this to work.

MemoryHole shows you a list of repositories that are available to you. The first time you go to the source control page, it will be likely be empty.



However, if there are already some Git or Mercurial repositories in the directory where you are running or in its parent directories, the IDE will find and list them for you.

To add a new repository, click on the plus icon on the right hand side. This will initiate a dialog that will ask you for information needed to establish a connection to a Mercurial or Git repository.

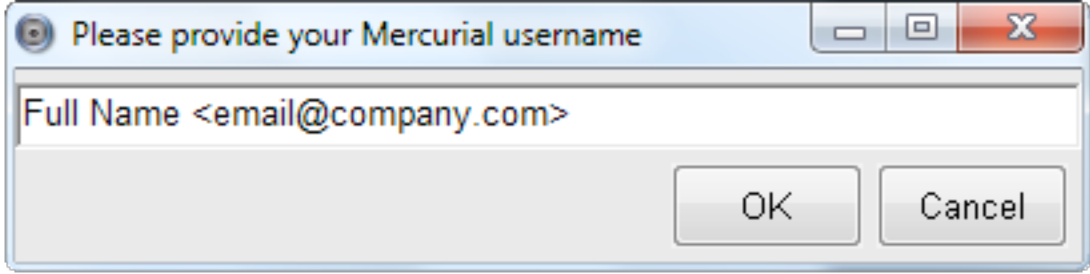
You have three choices. You can connect to a remote repository such as the public Newspeak repository (recommended); you can connect to an existing local Mercurial or

Git repository or you can create a new Mercurial repository. Note that local repository paths are relative to the current image directory! Choose the first option; type in

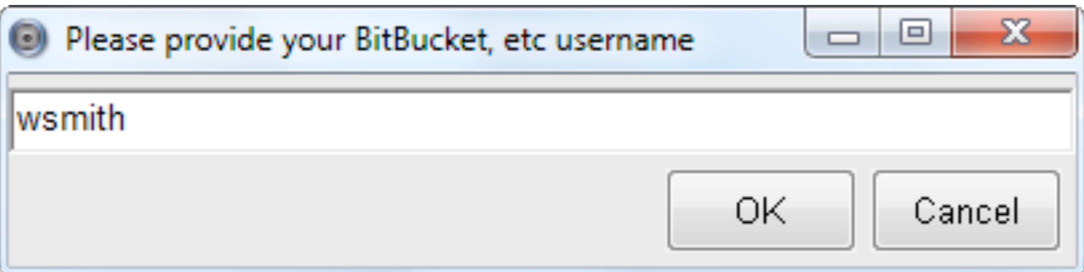
https://bitbucket.org/newspeaklanguage/newspeak

and accept your change by click the green icon.

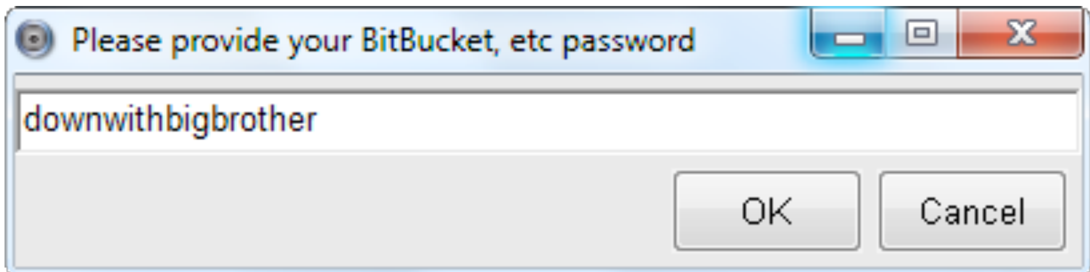
Once you choose one of these, more information is required. First, your name and e-mail address.



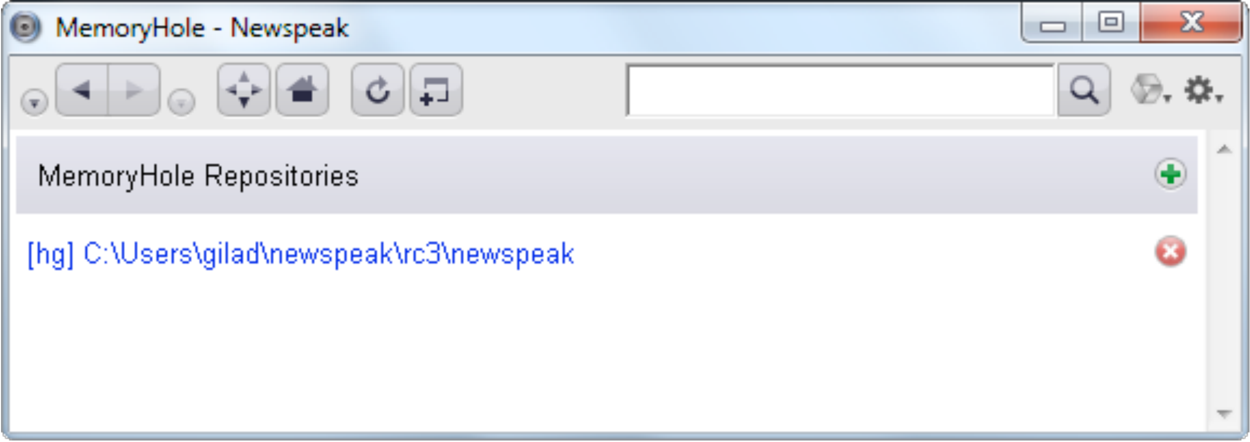
Next, your username at your hg provider. For the public Newspeak repository, this is BitBucket.



We also need your password at bitbucket:



Newspeak will set up a dedicated mercurial repository in the same directory as the image. This takes a short while. Once that is done, the new repository appears in the repository list on the MemoryHole page



Each entry in the list is a link to the page for that repository. Click on the new link. The very first time you do this will take a very long time, as we diff the live image against the repository. Take a walk.

At the end of this process, you will have a local repository that is a clone of the public newspeak repository. MemoryHole will track changes between the image and this repository. Moreover, MemoryHole will also notify us if changes occur in the public repository, and enable us to sync the local repository with the public one.

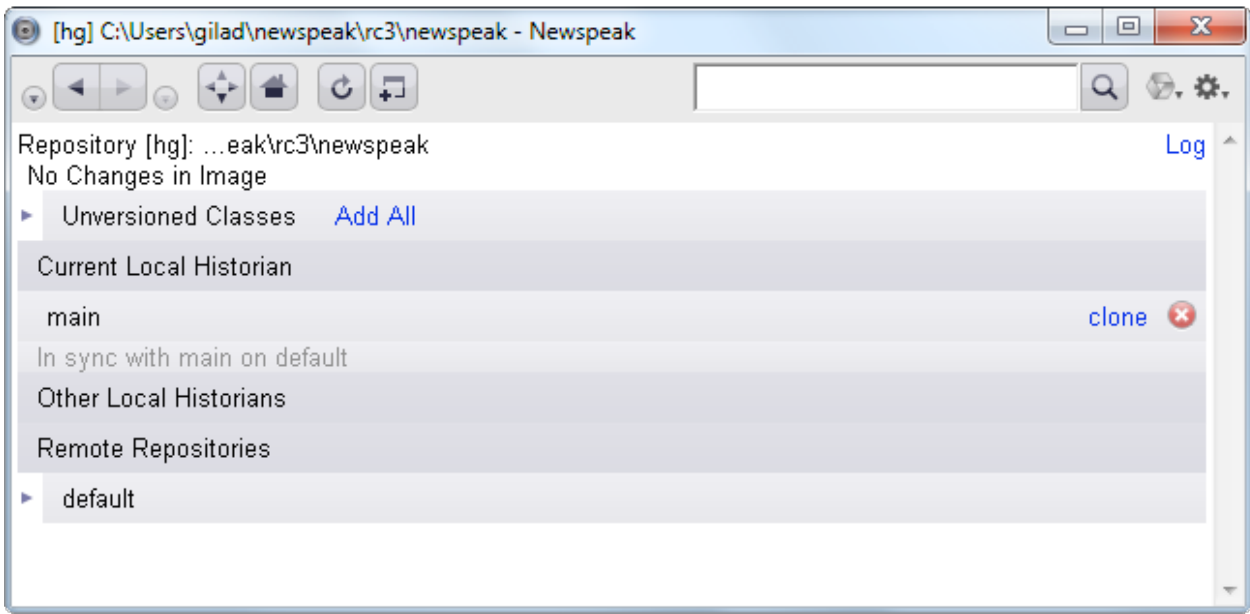
You can have several repositories open in an image. For example, you might want both the normal public repository

https://bitbucket.org/newspeaklanguage/newspeak/

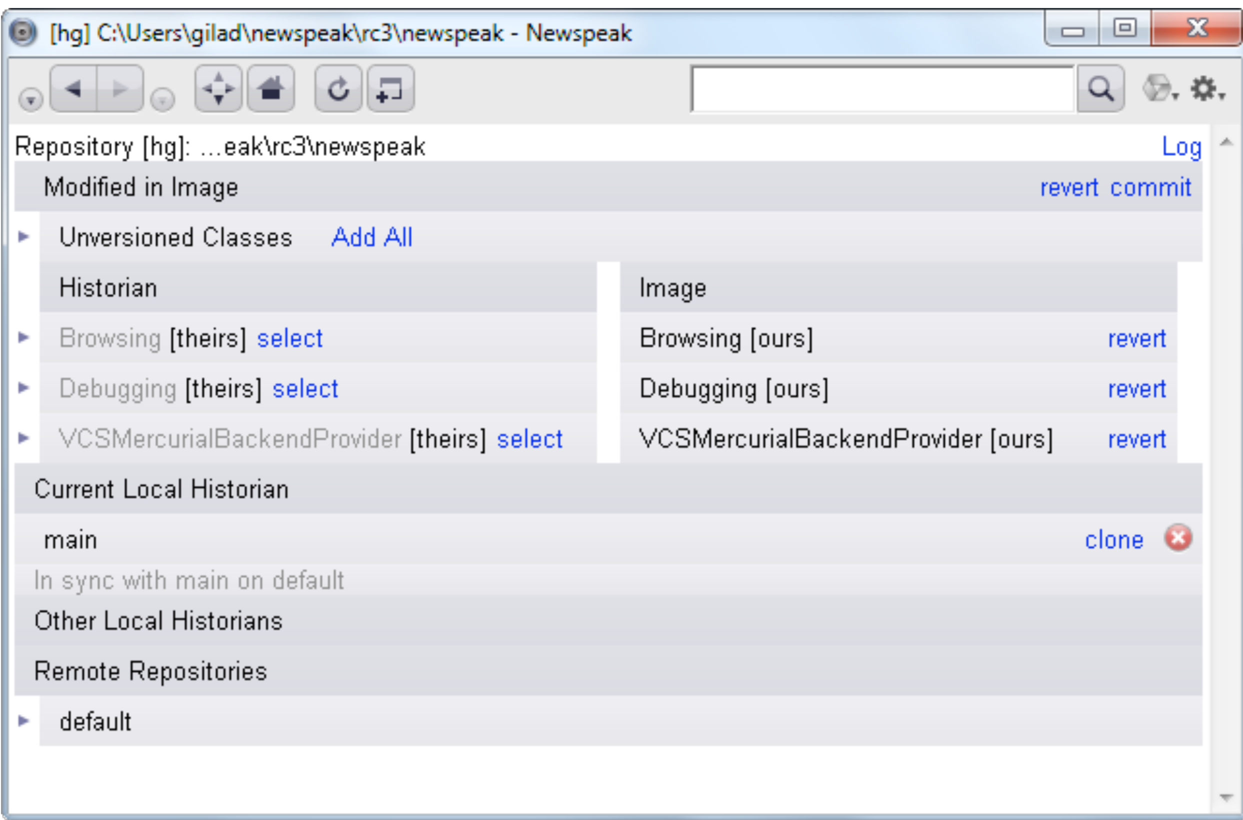
and the development repository

https://bitbucket.org/newspeaklanguage/newspeak_bleeding_edge/.

Once the initial set up is done, you may see a display like:



The above is what you'll see if your image and the repository are perfectly in sync, with no differences. In practice, that's not very likely, as the download image won't be updated nearly as frequently as the repository. So you're more likely to see a screen like the following:



The top most deep grey banner says: *Modified In Image*. Underneath, there is a collapsed pane that lists any classes that are *unversioned* - that is, are not under source control at all. There's also a link marked Log in the upper right hand corner that will take you to list of source control log entries.

Below are two banners side by side - one marked *Historian* the other *Image*. This allows us to compare the current local repository to the image. Beneath it is a list of classes that differ between the image and the local repository. They might differ because your image is out of date with respect to the repository (say, because the repository has moved forward since the download image was created) , or because you've made changes in the image (the common scenario in day-to-day work).

If you want to see the differences between the image and the repository, you can expand them. We can drill down from the changed top level class into methods and, recursively, into nested classes.

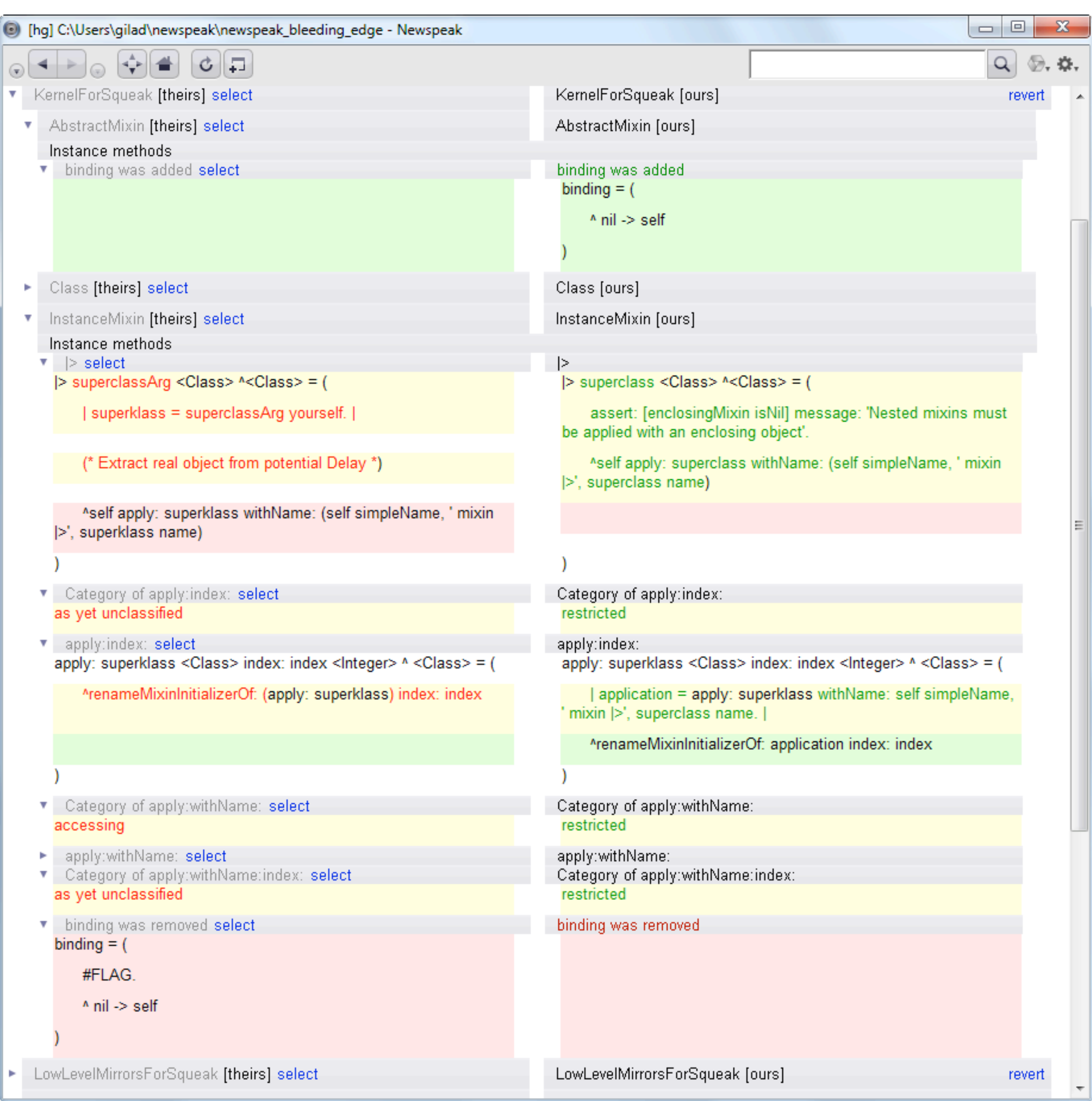
New classes and methods are shown on a green background.

Removed ones with a pink one.

[To the FAQ/Table of Contents](#)

Changed code is shown with yellow background.

Additions are highlighted in green and deletions in red. For example, in the screenshot below you can see that in class **KernelForSqueak**, the nested class **AbstractMixin** has had the method **binding** added. There have been changes made to the nested class **KernelForSqueak `Class** which are not shown. And in we see that in class **KernelForSqueak `InstanceMixin**, the method **binding** was removed and several methods were modified.



If we click on the revert link, the code in the image will be mutated to match the local repository. If you're starting up, as in this case, this ensures that we have the latest and greatest from the repository. If you've had the MemoryHole page open for while, it's a good idea to use the refresh button of the Hopscotch window to force it to update the display so it accurately reflects the differences with respect to the repository.

More on Historians

Before we go on, we should discuss MemoryHole's model of the world.

You probably noticed the term *historian* used in the above screenshots. A historian is a keeper of history (similar to a Git branch or Mercurial bookmark). What's a history? A *history* is a complete representation of the state of a branch in a repository at some point in time. A history includes the state of the source, all known prior states and how they relate to each other. A historian is a mutable pointer to a history. Usually a historian points to the latest history in a branch.

We can maintain multiple historians, pointing at different branches and repositories. One of these will be the *current historian*, which in our case points at the tip of a local **hg** repository. The *Modified In Image* panel shows us how the image and the current historian differ. Additional historians are listed under the banner *Other Local Historians*.

Above, we arranged for the local repository to be a clone of the public newspeak repository at a certain moment. MemoryHole is aware of the relationship between our clone and the original (in our case, the public newspeak repository). As the original evolves, MemoryHole will notify us and offer to sync the clone to it. Likewise, if the clone evolves - most likely because we publish some changes to it from the image - MemoryHole will offer to sync with the original. What is the mechanism which enables us to track changes to the original repository?

A local historian may *track* a remote one. If a historian is being tracked, MemoryHole will notify us when changes to the remote historian occur. When MemoryHole sets up a connection to a remote repository it will set the current historian to a clone of that repository, and have the local historian track the (historian of the) trunk of the remote one.

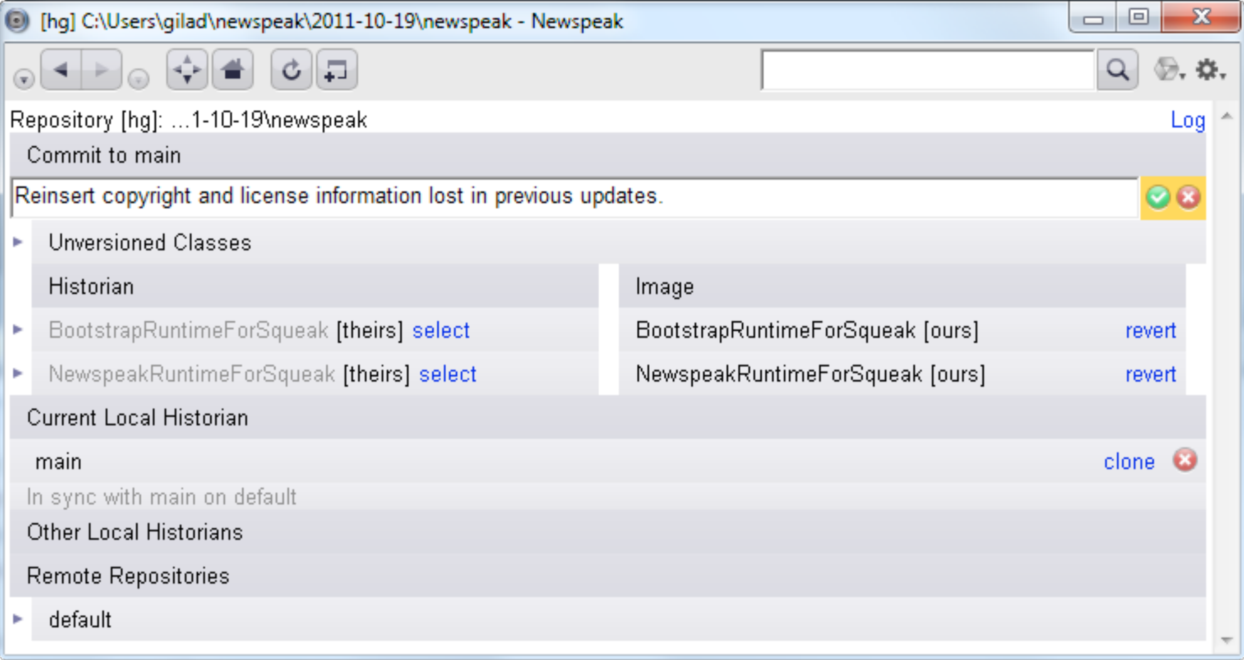
The original repository is listed under *Remote Repositories*. We can expand it and see all historians associated with the remote repository, and whether they are tracked by the current historian.

For more details on MemoryHole, see Matthias Kleine's thesis at

http://www.hpi.uni-potsdam.de/hirschfeld/publications/media/KleineHirschfeldBracha_2012_AnAbstractionForVersionControlSystems_HPI54.pdf.

Publishing Code

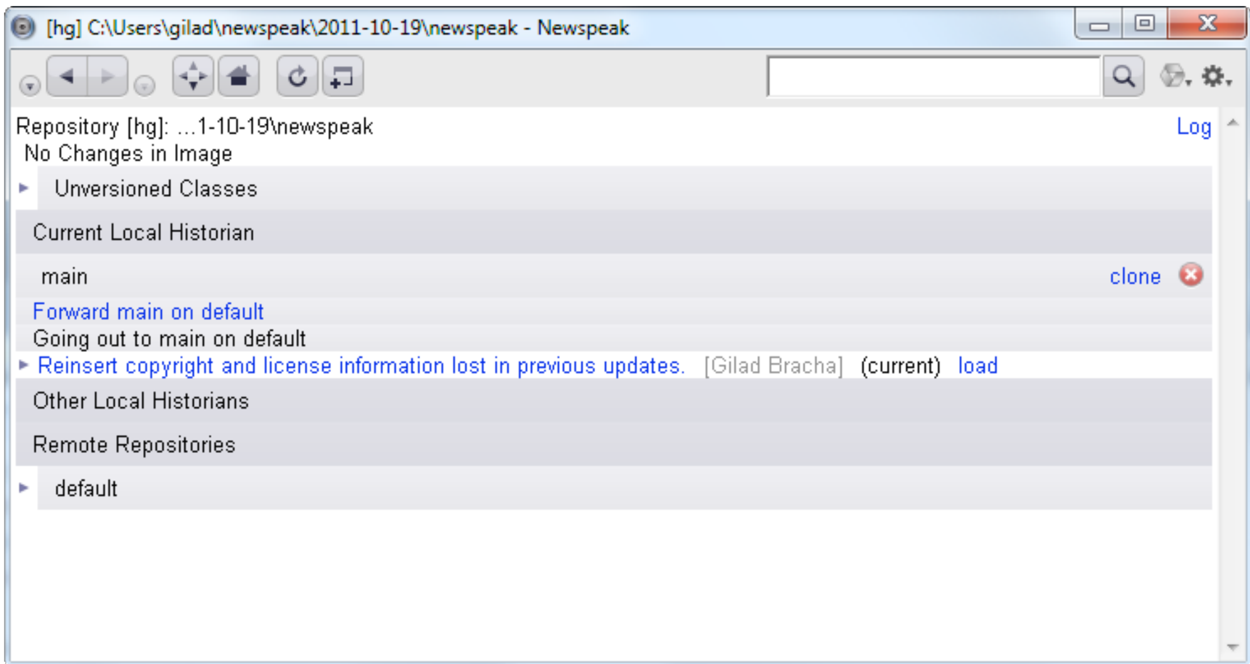
If you edit a class that is under source control, the changes between the image and the repository will be displayed in the same way as we've seen above. You can then use the commit link to publish them to the local repository. You will be asked to provide a commit message describing your changes.



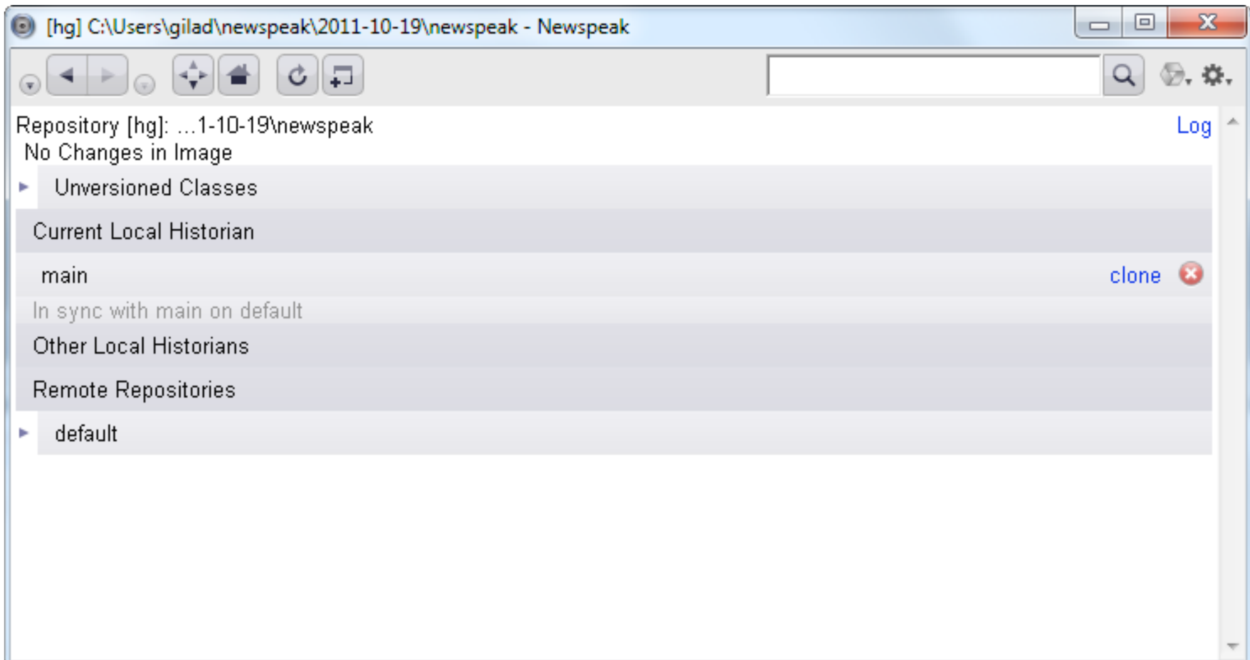
At this point, the current historian is no longer in sync with the remote one it is tracking. Under the heading *Going out to main on default*, you will find a list of commits that have not been sync'ed to the remote repository. The commit message for each such commit is listed. In our case, there is just one - the commit we just made.

You can expand each commit and see what it consists of.

To the FAQ/Table of Contents



Assuming you have the rights to publish to the remote repository, use the Forward to main on default link to push the updates from your local repository to the remote one. Once that's done, the local and remote repositories are in sync, and all is well.



[To the FAQ/Table of Contents](#)

Getting Updates

As indicated above, MemoryHole will notify you when your remote repository changes. The changes will be listed under the heading *Coming in from main on default* as shown above. Again, this is a list of commits that have not been sync'ed, but in this case, these are commits to the remote repository.

A typical situation is where you have made changes in your image on the one hand, while the remote repository has been updated on the other, as illustrated above.

In this case, clicking the Forward to main on default link will bring the local repository up to date with the remote one.

How do I use the Native GUI?

The native GUI binding currently only exists for Windows (XP, Vista, Windows 7, Windows 8). If you're running on Windows, the native binding is on by default; beware that on Windows 8, you have to use a muse or trackpad, since touch events trigger errors. Native windows currently co-exist with the main Squeak window where the Morphic binding is used. Eventually, the system will operate exclusively with the native binding, but this transition will take time.

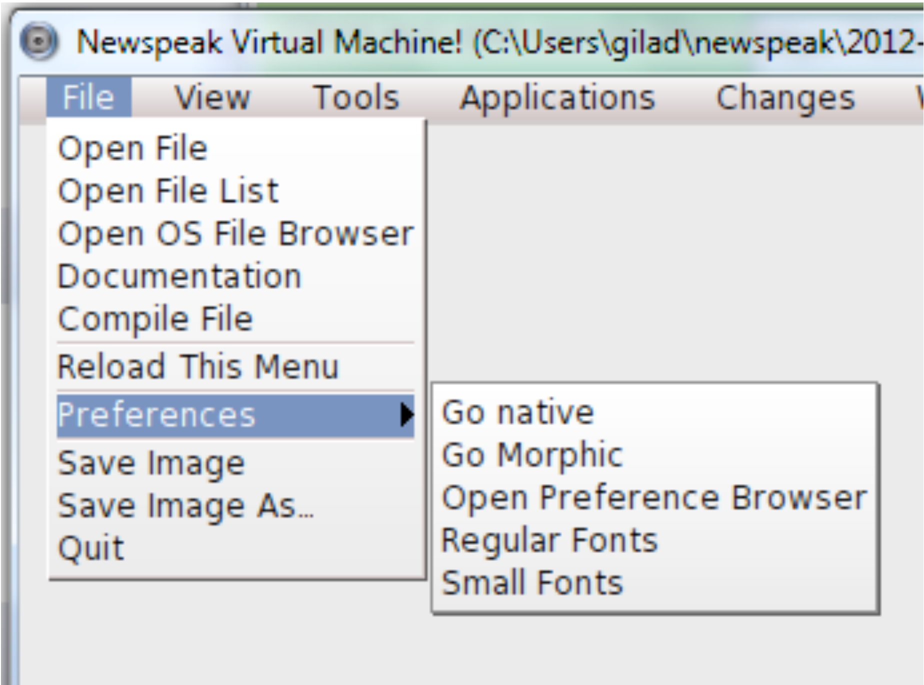
The native binding still suffers from some bugs/limitations. These bugs are detailed below.

If you find these problems are too acute for you to bear, and want to retreat to the safety of Morphic, you can do so. First, you need to display the morphic window (known as the console window).

How do I open a console Window?

Choose *Show Console Window* from the operate menu. If you are running on Mac or Linux, you are always running within the console window, which is simply the top level Squeak window, aka the morphic window,

Then choose the *File>>Preferences>>goMorphic* option. All Hopscotch windows will revert back to their Morphic form, and from then on any new ones will open as Morphic windows as well.



You can always change your mind again. Choose the *File>>Preferences>>goNative* option, which is the inverse of the above. At this point, all existing Hopscotch windows will become native, any new ones you create will be native as well.

You can go back and forth among these choices as many times as you like.

Windows binding bugs and limitations:

Various keyboard shortcuts specific to Squeak also don't work in the native version. You can of course use Ctrl-S to save, and Ctrl-C, Ctrl-X and Ctrl-V to copy, cut and paste.

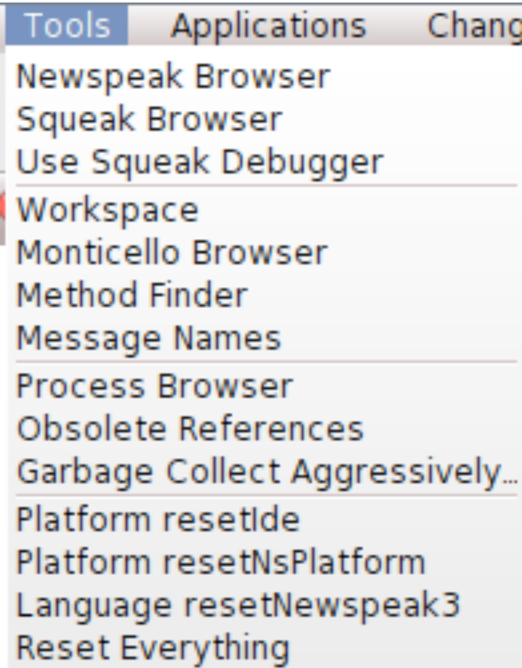
There are doubtless others.

How do I open a Newspeak browser from the Console?

At the top of the console window, you'll see a tool bar that looks like this:

To the FAQ/Table of Contents

The first item of the Tools menu below opens a fresh Newspeak browser, displaying the home page.



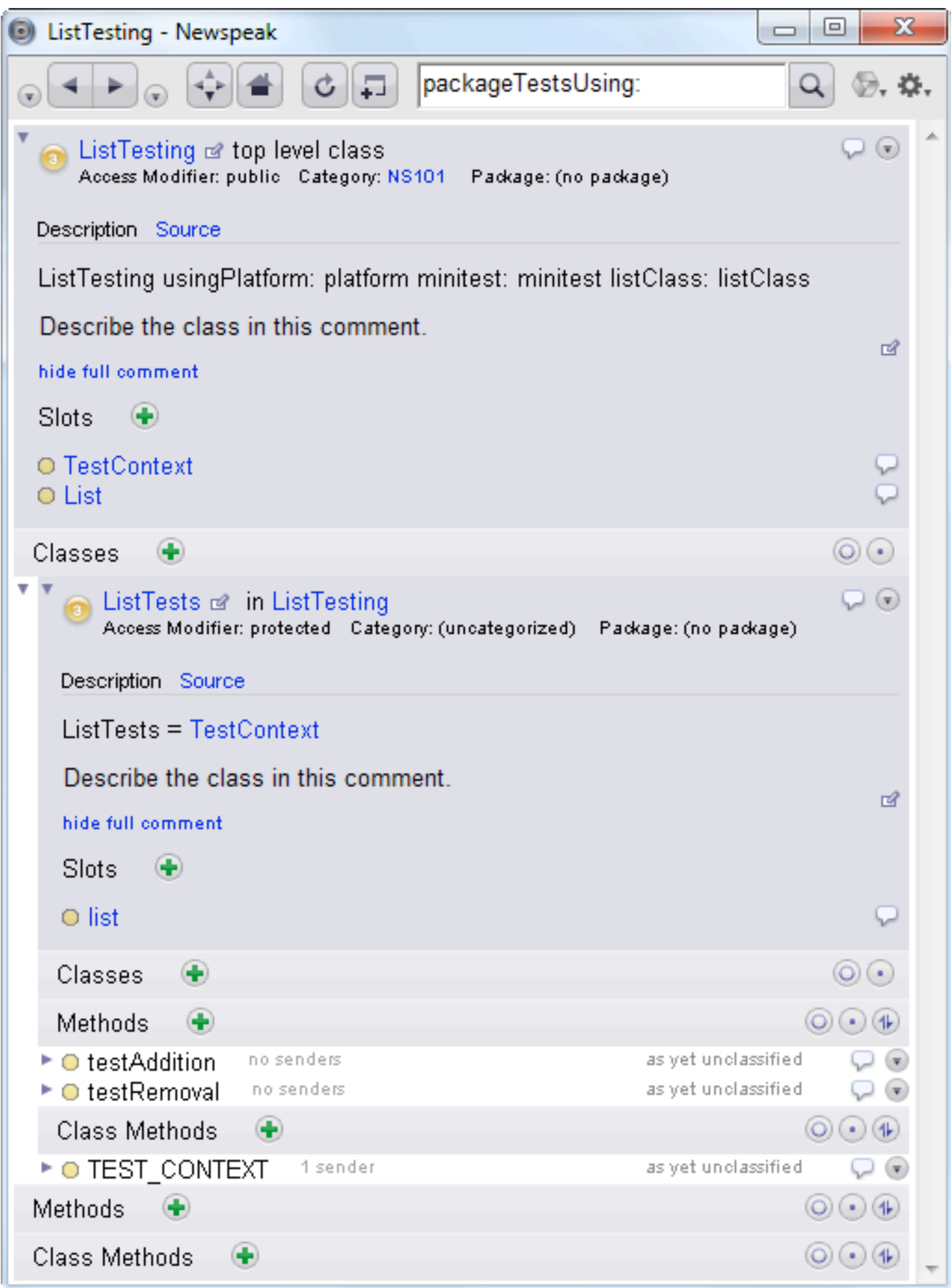
How do I run Unit Tests?

Newspeak's unit testing framework is called Minitest. Support for Minitest is integrated into the Hopscotch IDE.

In Minitest, you define a *testing module*, which is designed to test a particular interface (not a particular implementation). To run tests, one needs to feed the testing module with the particular implementation(s) that one wishes to test. A *test configuration* module does just that. Newspeak naturally enforces this separation of interface and implementation.

Here is a testing module **ListTesting**. It is a very simplistic set of tests for lists. **ListTesting**'s factory method takes 3 arguments: ***platform*** (the Newspeak platform, from which all kinds of generally useful libraries might be obtained), ***minitest*** (an instance of Minitest, naturally) and ***listClass***, a factory that will produce lists for us to test. This is typical: the first two arguments to a test module factory are almost always a platform object and an instance of Minitest, while the third is the object under test.

To the FAQ/Table of Contents



Nested within the module is the class **ListTests**, which includes the actual tests. Test methods are identified by the convention that their names begin with **test**. Each test will be executed in a test context; that is, for each test method being run, Minitest will instantiate a fresh ***ListTests*** object. That is why ***ListTests*** is called a *test context* - it provides a context for a single test.

It is common to define test context classes like ***ListTests*** as subclasses of the class **TestContext** defined by the Minitest framework. One reason why having a Minitest factory argument is useful is so we can import **TestContext**. **TestContext** provides useful methods like ***deny:***, so it is convenient to use it. However, inheriting from **TestContext** is **not** essential. What identifies **ListTests** as a test context is the marker class method ***TEST_CONTEXT***, not inheriting from **TestContext**.

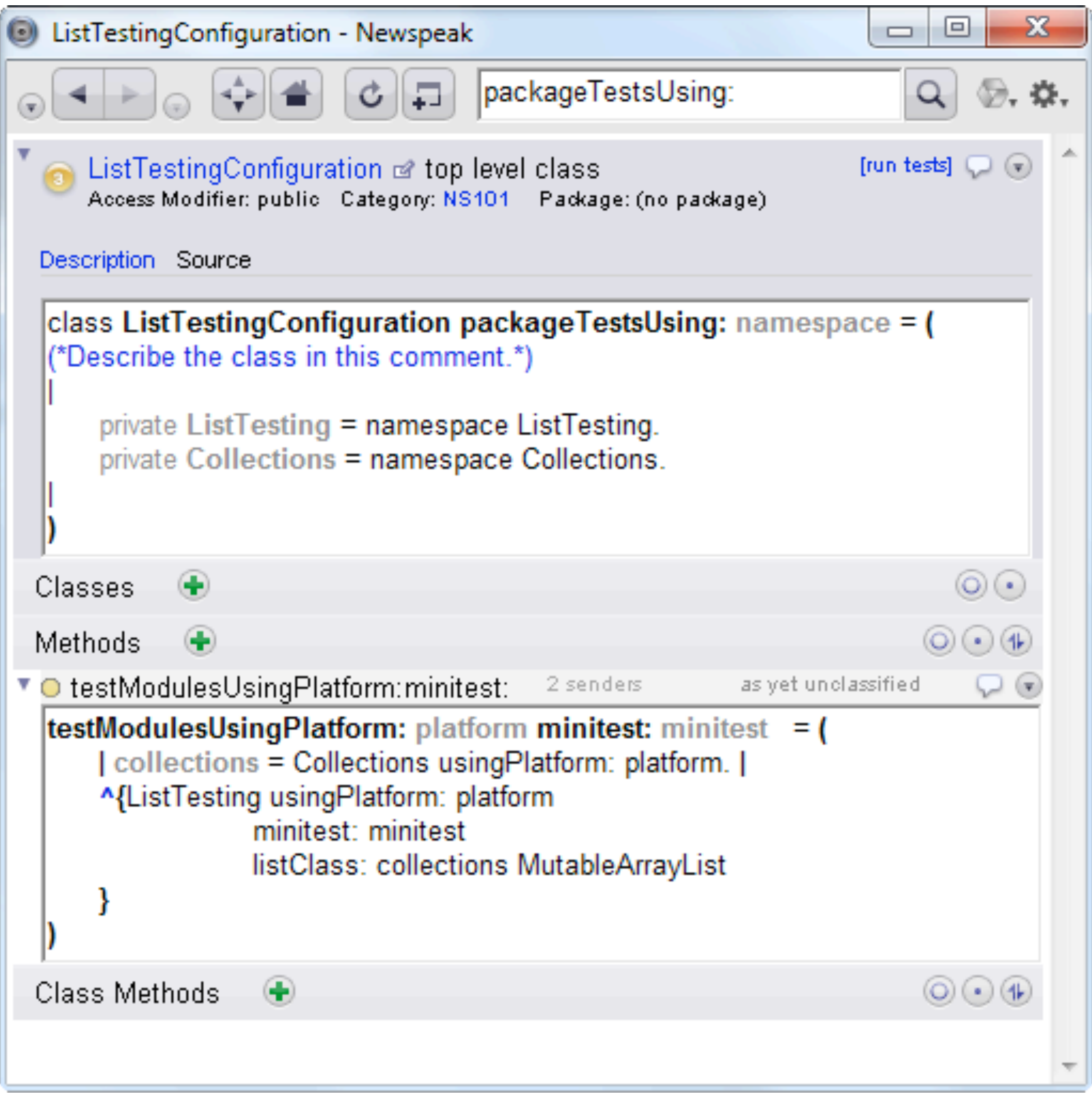
Minitest will do its work by examining the nested classes of the test module and seeing which are test contexts (that is, which have a class method named ***TEST_CONTEXT***). For each test context ***tc***, Minitest will list all its test methods (the ones with names beginning with **test**) and for each of those, it will instantiate ***tc*** and call the selected method on it, gathering data on success or failure.

We need a test configuration to run the tests, as the test module definition is always parametric with respect to any implementation that we would actually test.

A test configuration module is defined by a top level class with the factory method

packageTestsUsing: namespace

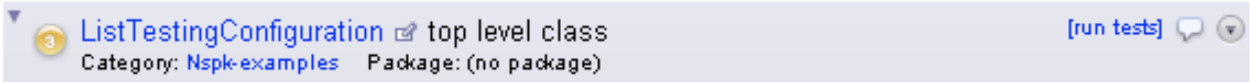
The factory takes a namespace object that should provide access to the testing module declaration and to any concrete classes or objects we want to test. This arrangement is very similar to how we package applications from within the IDE.



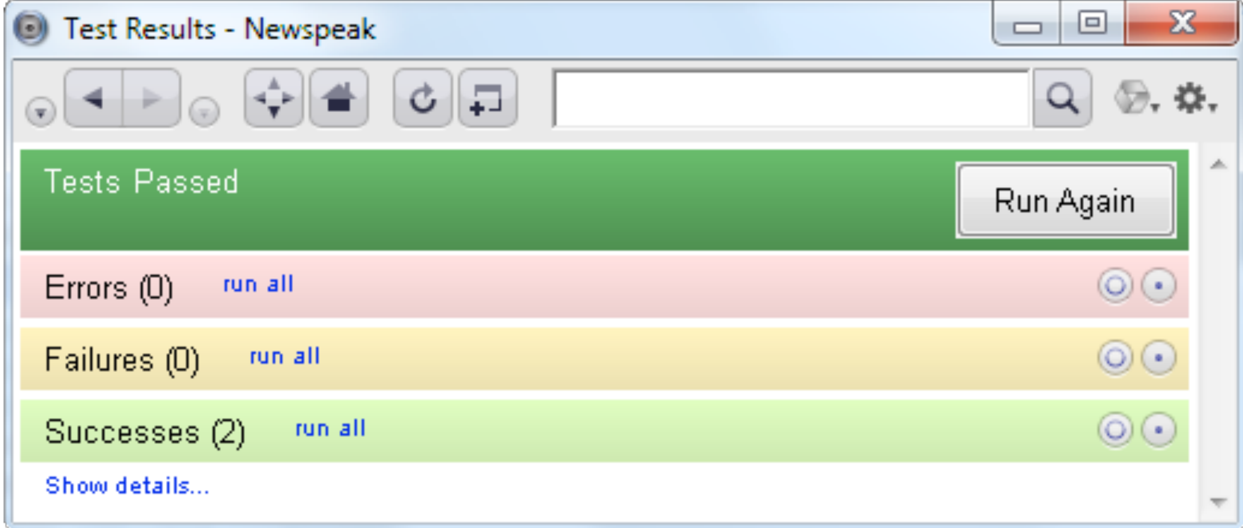
We show a single test configuration **ListTestingConfiguration**, but you can define as many you like.

The method ***testModulesUsingPlatform:minitest:*** must be provided by the configuration. It will be called by Minitest to produce a set of testing modules, each of which will be processed by the framework as outlined above (i.e., searched for test contexts to be run).

The IDE recognizes test configurations based on the name of the factory method - that is, a class with a class method ***packageTestsUsing:*** is considered a test configuration, and the IDE will provide a **run tests** link in the class browser's upper right hand corner.

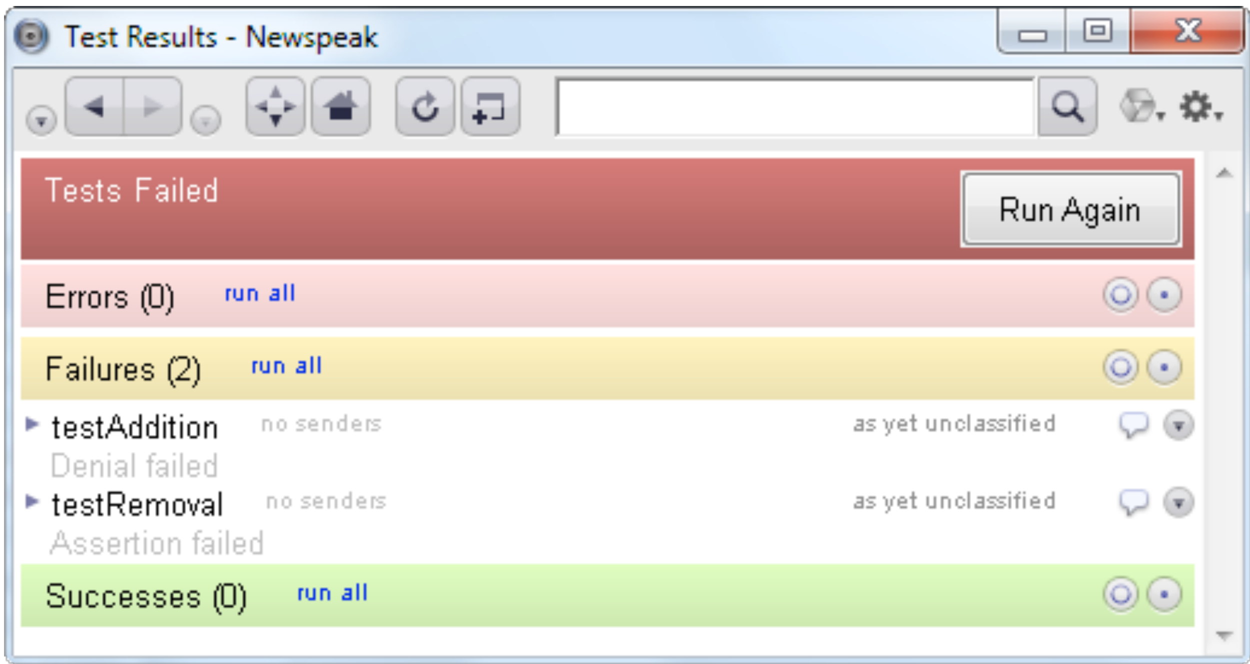


Running the tests will display a progress bar, and once they have run you will see a test results page:



Things are more interesting if some tests fail (did not produce expected results) or cause errors (raised unexpected exceptions):

[To the FAQ/Table of Contents](#)



If any of tests have failed in any way (i.e., resulted in failures or errors) the banner at the head of the page will say so on a red background. If all tests succeed the banner will be green. A gray banner indicates that even though no test has failed, not all tests have been run.

Note that successes are hidden by default, as no one cares about your successes - only your errors and failures. There is a link that allows you to bask in their glory if you need to.

You can click on each test method just as you would in a class browser to see the failing test code. Beneath the method is a link to the exception; click on the link to see a stack trace.

More about Minitest

If you are used to SUnit (or any of the many unit testing frameworks it has inspired, liked JUnit etc.), it may be worth noting some of the differences.

Minitest does away with concepts like ***TestResource*** that are typically used to hold data for tests.

In the simple case above, the data for the test gets created by the instance initializer of **ListTests** . However, what if the data for the test needs to be shared among multiple tests (say, because it is expensive to create)?

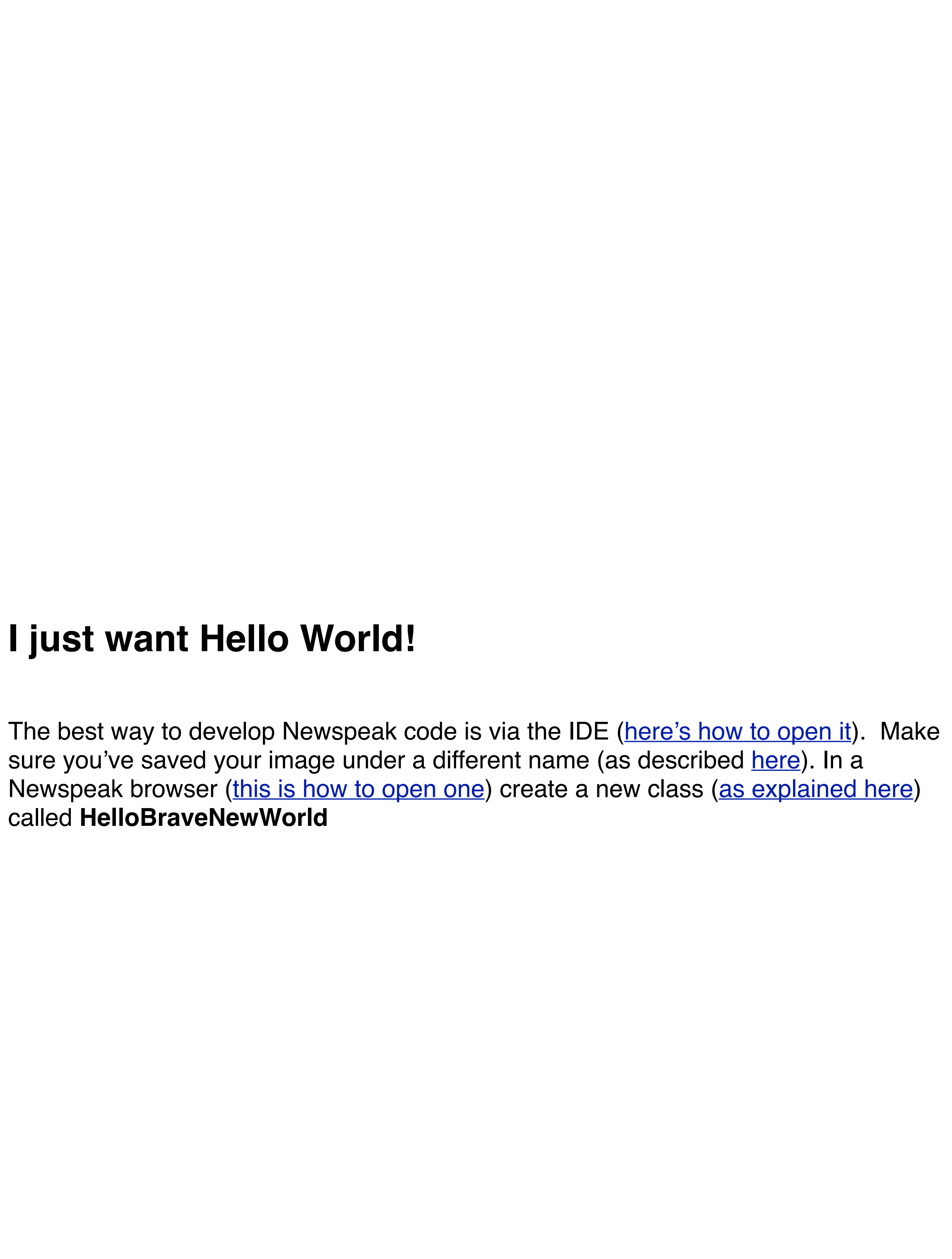
As an example, suppose we want to test a compiler, and setting up the compiler is relatively costly.

```
class CompilerTesting usingPlatform: platform
                minitest: mintest
                compilerClass: compilerClass = (
| Compiler = compilerClass. |  )
(
   class CompilerHolder = (
     | compiler = Compiler configuredInAParticularWay. |
 )(
   class StatementsTests ( ...) (....): ( TEST_CONTEXT = ())
  )
)
```

Minitest leverages Newspeak's nested structure in these cases. A test context (**StatementTests** above) does not have to be a direct nested class of the test module. Instead, we can nest it more deeply inside another nested class (**CompilerHolder**). That nested class will serve to hold any state that we want to share among multiple tests - in our case, an instance of the compiler, which it will create and store as part of its initialization.

As you can see there is no need for a special ***setUp*** method or a test resource class. Newspeak's nesting structure and built-in instance initializers take care of all that. If the shared resource is just an object in memory, then it will also be disposed of via garbage collection after the test is run. Of course, some resources cannot be just garbage collected. In that case, one should define a method named ***cleanUp*** in the test context class.

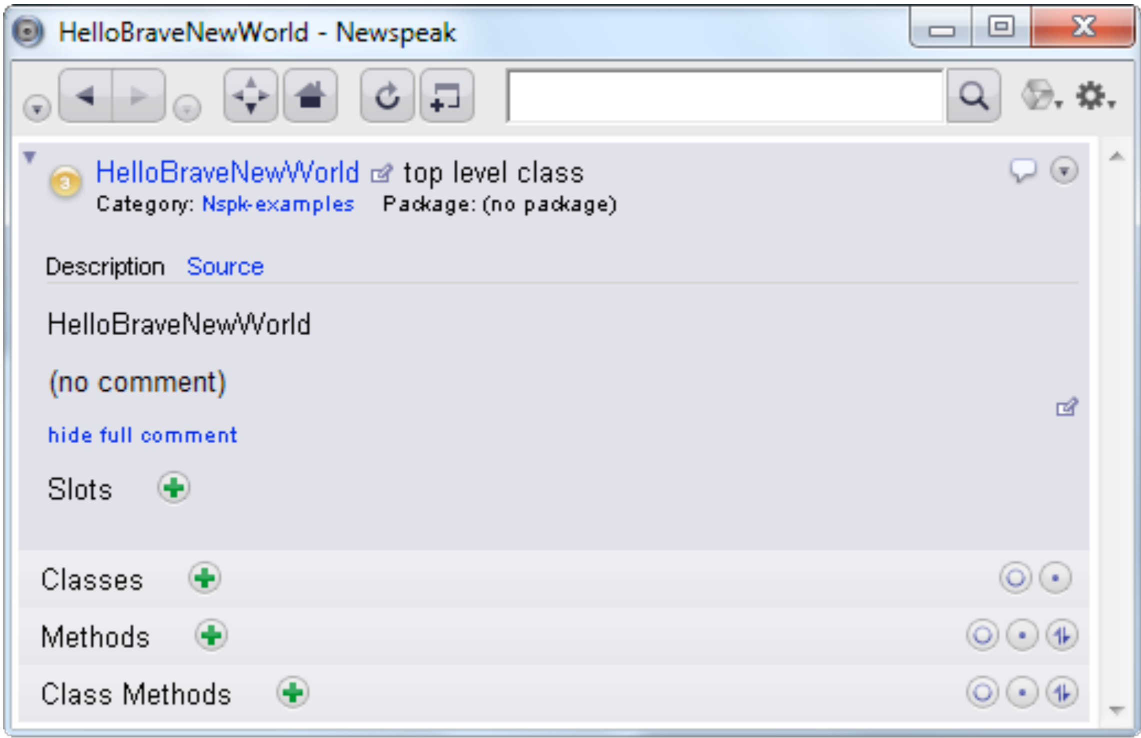
Minitest cleanly breaks down the multiple roles an SUnit **TestCase** has. The definition of a set of tests is done by a test context. The actual configuration is done a test configuration. And the actual command to run a specific test (the thing that should be called **TestCase**) is not the user's concern anymore - the test framework handles it but need not expose it.



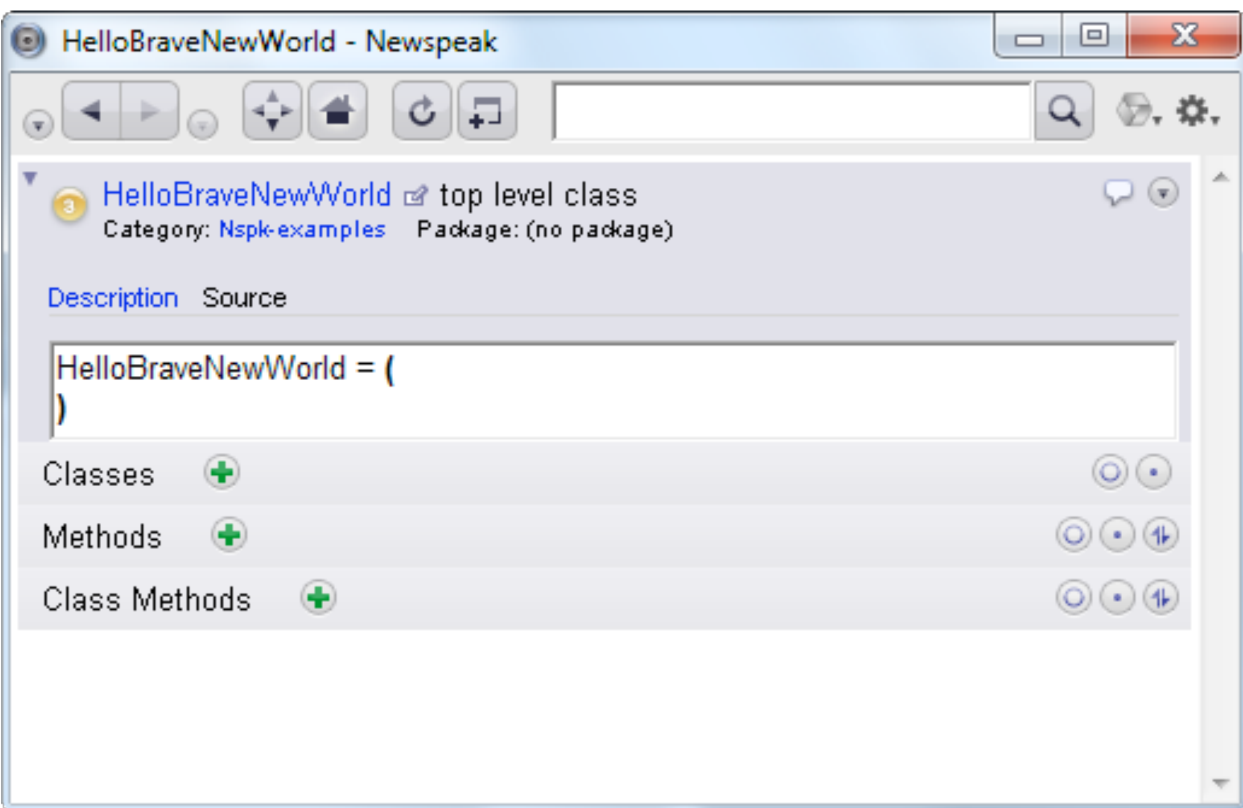
I just want Hello World!

The best way to develop Newspeak code is via the IDE (here’s how to open it). Make sure you’ve saved your image under a different name (as described here). In a Newspeak browser (this is how to open one) create a new class (as explained here) called **HelloBraveNewWorld**

[To the FAQ/Table of Contents]



Click on the link that says Source. You should see something like the following:



What we see is the actual syntax for a Newspeak class header. In this case, we see that the class name is **HelloBraveNewWorld**. The keyword **class** is omitted in this view - we know it's a class. Its superclass is listed after the **=** sign (unless it is **Object**). Disregard the rest - we'll explain it as we go along.

Now select the entire text, and replace it by typing in the following code:

```
class HelloBraveNewWorld usingPlatform: platform = (
        platform squeak Transcript open show: 'Hello, Oh Brave new world'.
)
```

and accepting it (Ctrl-S or Cmd-S). What have we done? We've changed the stuff between the parentheses a good deal. The parentheses delimit the ***instance initializer***. The instance initializer contains all the ***slot*** (aka field/instance variable) declarations of the class, and any initialization code for them. In this case, there are no slots; the initialization code is all there is - the line that says

platform squeak Transcript new open show: 'Hello, Oh Brave new world'.

If you know Smalltalk or Self, you'll recognize the syntax. Otherwise, stay with us as we take this expression apart:

platform is a parameter to the initializer. It's declared in the line above:

HelloBraveNewWorld usingPlatform: platform

more on that in a bit. The parameter is an object that will represent the underlying platform we are running on. It is our link to the outside world.

platform squeak sends a message to *platform*. The message has no arguments - it consists solely of the identifier **squeak**. In mainstream syntax, this might have been written as

platform.squeak()

In Newspeak, you don't need the dot - just use whitespace. Likewise, you don't need the empty parameter list in parentheses - if there are no parameters, you don't write any.

The message "squeak" is a way of getting at things you are specific to the Squeak-based implementation. It returns an object that represent's the namespace of the Squeak system. In our case, we are using it to get at Squeak's "console", Transcript.

platform squeak Transcript in turn sends the message **Transcript** to *platform squeak*. The transcript is the standard output stream in most Smalltalk systems, including Squeak. So now we have an output stream to write to.

platform squeak Transcript open show: 'Hello, Oh Brave new world'

We're sending the message *open* to the output stream we got from *platform Transcript*. This will open a window to display the transcript stream, and return the stream.

Finally, we send the stream the message *show: 'Hello, Oh Brave new world'*. This message includes the argument *'Hello, Oh Brave new world'*, which is a string literal. The name (often called the *message selector*) of the message is **show:**. So we're asking the output stream to show a string - which was the entire purpose of the exercise. Because this code appears in the instance initializer, it will get executed whenever we create an instance of the class.

Again, it may help to see this in a more traditional syntax:

platform.squeak().Transcript().open().show("Hello, Oh Brave new world");

To create an instance of a class, we must send it a message. Sending a message is the only operation in Newspeak. What message shall we send? Well, the class declaration specifies that message immediately after the class name

HelloBraveNewWorld usingPlatform: platform

The latter part of the line above tells us (and the compiler) that the message named **usingPlatform:** will be used to create instances of this class. The message takes a single parameter named *platform*. When the class receives such a message, the actual parameter is made available to the instance initializer under the name *platform*.

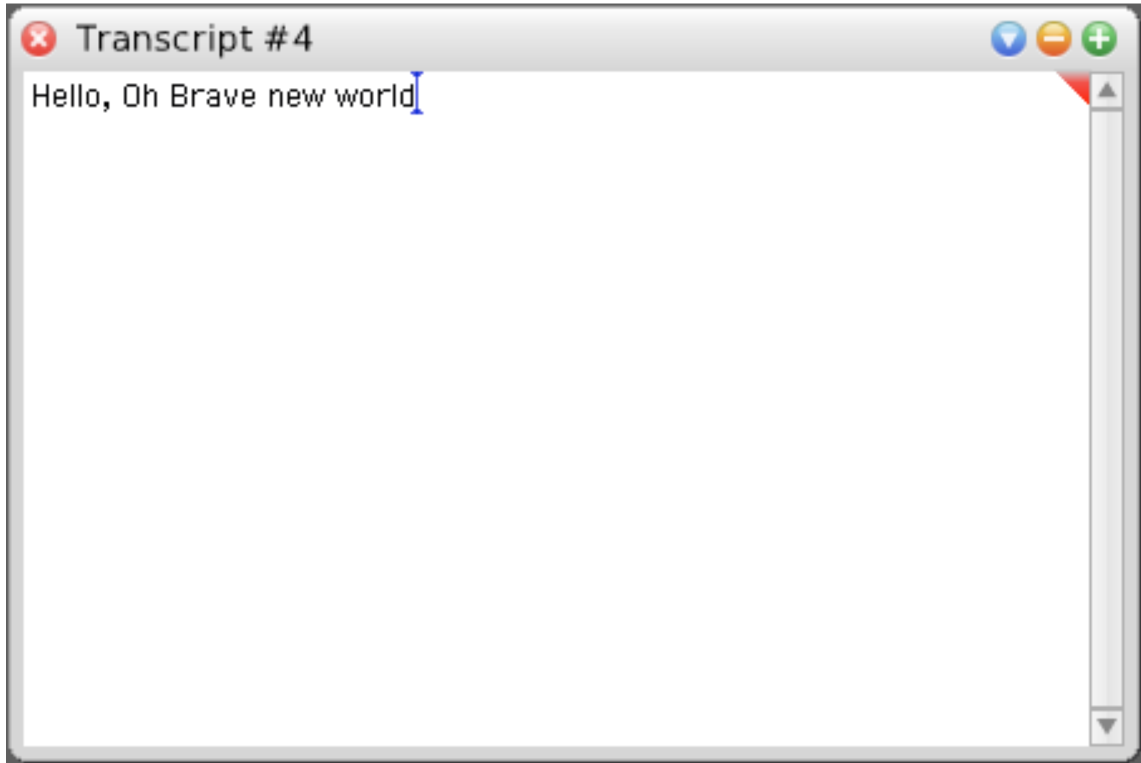
So now you know that we must send **HelloBraveNewWorld** the message **usingPlatform:** with an argument representing the underlying platform - or at least an object that responds to the message **Transcript** with a valid output stream. This will create an instance of the class, causing its initializer to run, and write to the output as we desire.

Where shall we get such an object? And how shall we send this message?
Open a workspace (this way) and type in:

HelloBraveNewWorld usingPlatform: platform

Note: If you are running on Windows, choose the open console option from the operate menu, so you can see the transcript window when it opens.

Select the expression and hit Ctrl-S (Cmd-s on a Mac). You should see some results:



Now, let's discuss some high principles. Good Newspeak style requires that a module declaration list all its external dependencies clearly and explicitly. What does this have to do with **HelloBraveNewWorld**? Well, we have, perhaps unknowingly, created a module.

In Newspeak, a top level class declaration is always a module declaration. A module declaration has no access to any surrounding scope; any names used inside the declaration must be declared within it, or inherited from another module declaration. Module declarations are of course instantiable like any other class; their instances are called ***modules***.

This is why we had to declare a parameter for our initializer. If we had written

```
class HelloBraveNewWorld  = (
        Transcript open show:  'Hello, Oh Brave new world'.
)
```

and then created an instance via *HelloBraveNewWorld new*
(if a class doesn't specify a message for creating instances, **new** is the default), we would get a **doesNotUnderstand:** error, because **HelloBraveNewWorld** does not understand the message **Transcript**. There simply is no way to access the standard

output stream, or any other system state, without having it passed in via a parameter when a module is instantiated.

To comply with the style guidelines, we'll change the code to:

```
class HelloBraveNewWorld usingPlatform: platform = (

| Transcript  = platform squeak Transcript. |
Transcript open show:  'Hello, Oh Brave new world'.
)
```

What we've done is declared a slot (aka field/instance variable) named **Transcript**. The slot declaration includes an initializer that initializes it to hold the object returned by *platform squeak Transcript*. Slots are declared between a pair of vertical bars, much like Smalltalk local variables. We can use **Transcript** to access the output stream in the rest of code.

The nice thing about this is that our dependence on **Transcript** is localized to one point - the declaration of the slot **Transcript**. The slot declaration plays a role similar to an import. It may not be a big deal in this tiny example, but in real code this localization is very valuable. You can see all the external dependencies of a module in one place, by going through it's slot declarations.

The use of slots as imports also allows us to rename imported elements if that makes sense. Usually it doesn't, but we could just as easily have written:

```
HelloBraveNewWorld usingPlatform: platform = (
| stdout  = platform squeak Transcript. |
stdout open show:  'Hello, Oh Brave new world'.
)
```

Of course, printing text isn't as interesting nowadays as it was when the original *Hello World* example was written over 30 years ago. The world expects much more today. The next step is to extend **HelloBraveNewWorld** to do a modern GUI.

... O brave new world

That has such people in't!

- William Shakespeare, The Tempest

Brave new GUI

Until now, we've used the Hopscotch IDE - but the IDE is only one possible application of the Hopscotch library. Hopscotch is a complete GUI application framework that can be used for all sorts of applications. We'll now use it to create a very simple application; we'll build a fancier one later.

A Hopscotch application consists of three parts: a ***presenter***, a ***subject*** and a ***model***. The presenter, as you'd expect, manages presentation. The subject of the presentation provides application logic. The model is the object that we want our application to interact with. The model has no knowledge of the GUI; therefore, it is not bound by any specific protocol or interface. It is the subject's role to provide logic to bridge between it and the presentation.

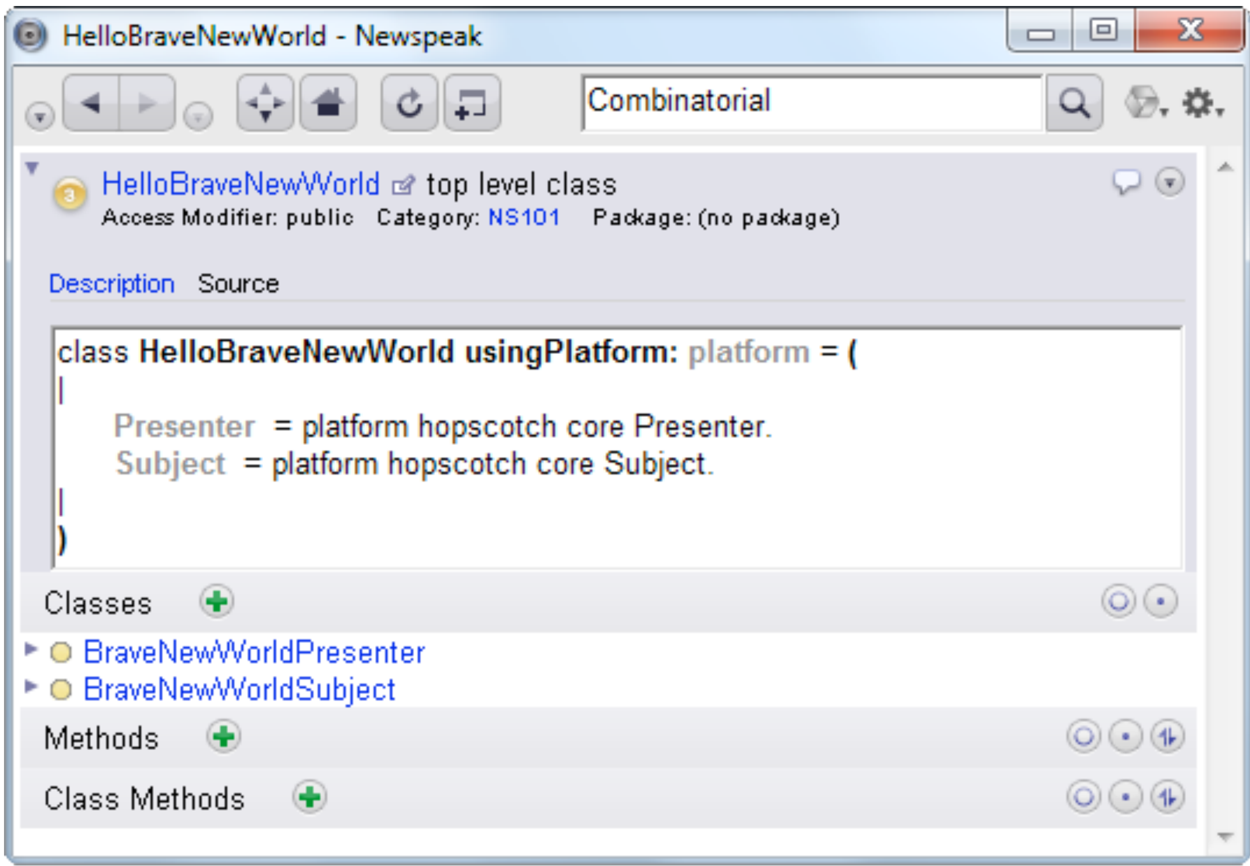
It follows that to build a GUI, we need to define a presenter class and a subject class. Presenter classes extend the **Presenter** class of **HopscotchFramework**, and subject classes extend its **Subject** class.

Let's change the header so that it imports these two classes

```
class HelloBraveNewWorld usingPlatform: platform = (
|
        Presenter  = platform hopscotch core Presenter.
        Subject  = platform hopscotch core Subject.
|
)
```

The message *platform hopscotch* provides us with the platform's built-in instance of **HopscotchFramework**. Sending it the message *core* gets us a namespace object that holds core classes of the framework, such as **Subject** and **Presenter**.

Next, we'll define presenter and subject classes - **BraveNewWorldPresenter** and **BraveNewWordSubject**, respectively, nested within the **HelloBraveNewWorld** class (here's how to declare a nested class).



The definition of **BraveNewWorldSubject** should read:

class BraveNewWorldSubject **onModel:** m **<String>** = Subject onModel: m **()()**

A subject class typically has a factory method **onModel:**, which takes a model as an argument.

Each class declaration determines what arguments are to be passed to the factory method of its superclass. Since *Subject* requires the model object (indeed it stores it in a slot named *model*) we pass it up, using an extended form of the superclass clause *Subject onModel: m* that specifies the factory method and arguments to be used. Prior to running the subclass' instance initializer, the named superclass factory will be invoked, causing its instance initializer to run (and those of its superclasses, recursively).

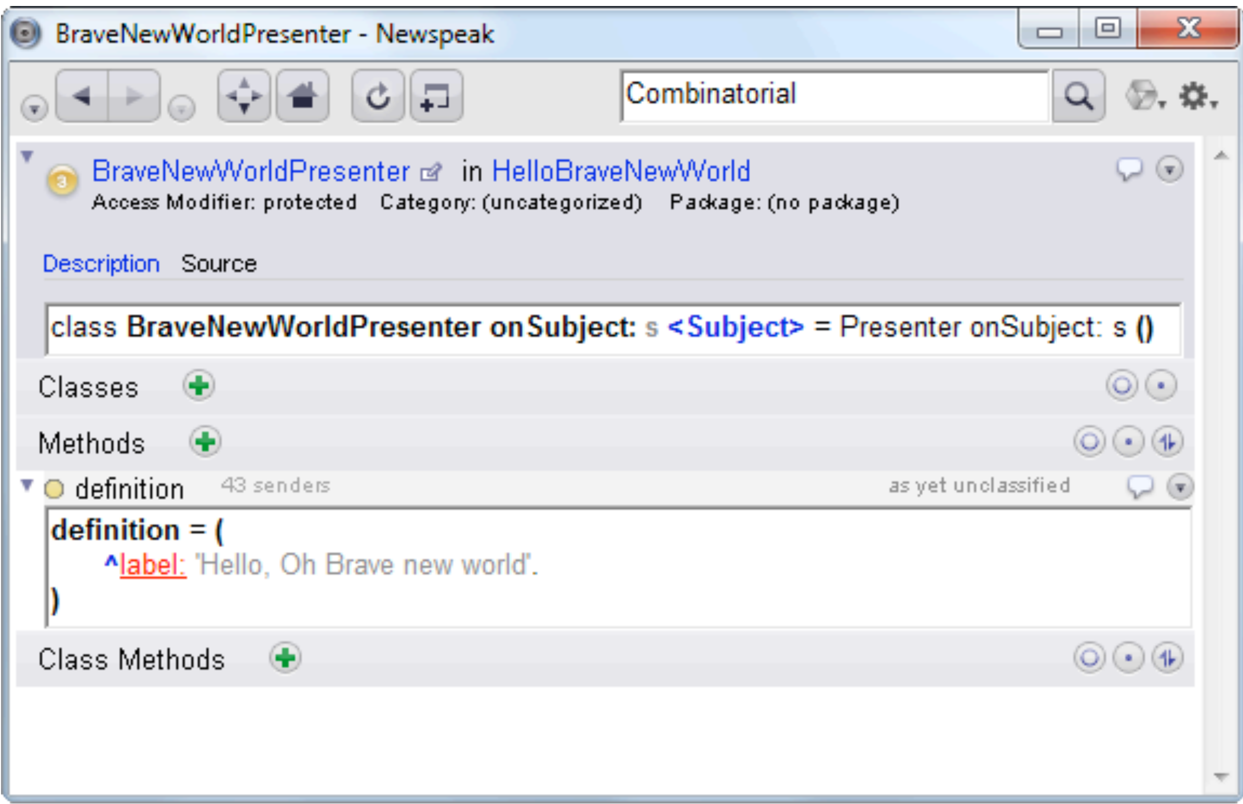
In this simple example, the model won't matter because we won't actually use it - but that is highly unusual.

Having established a subject, we can move on to the presenter. Define **BraveNewWorldPresenter**'s thus:

class BraveNewWorldPresenter **onSubject:** s **<Subject>** = Presenter onSubject: s **()()**

[To the FAQ/Table of Contents]

Now we can add some functionality to our presenter. The most important method (here's how to add a method) in a presenter is **definition**, which defines what gets displayed. In our case, we will simply present the string *'Hello, Oh Brave new world'*. The Hopscotch library provides a set of basic operators, called combinators, that display information on the screen. One of the simplest combinators is **label:**, which allows us to display a string.



The definition method declares its return type to be **Fragment**, Type annotations are displayed in **blue**, and types are delimited with angle brackets. If you dislike types, relax; they are strictly optional. Fragments are the basic elements of display. The **label:** combinator constructs a fragment from a string.

The **^** sign in front of **label:** indicates that this is a return statement. This is the traditional Smalltalk syntax; we might change it to **return:** in a future version of the language, to make the syntax accessible to a larger population of programmers.

Why is *label:* red?

To the FAQ/Table of Contents

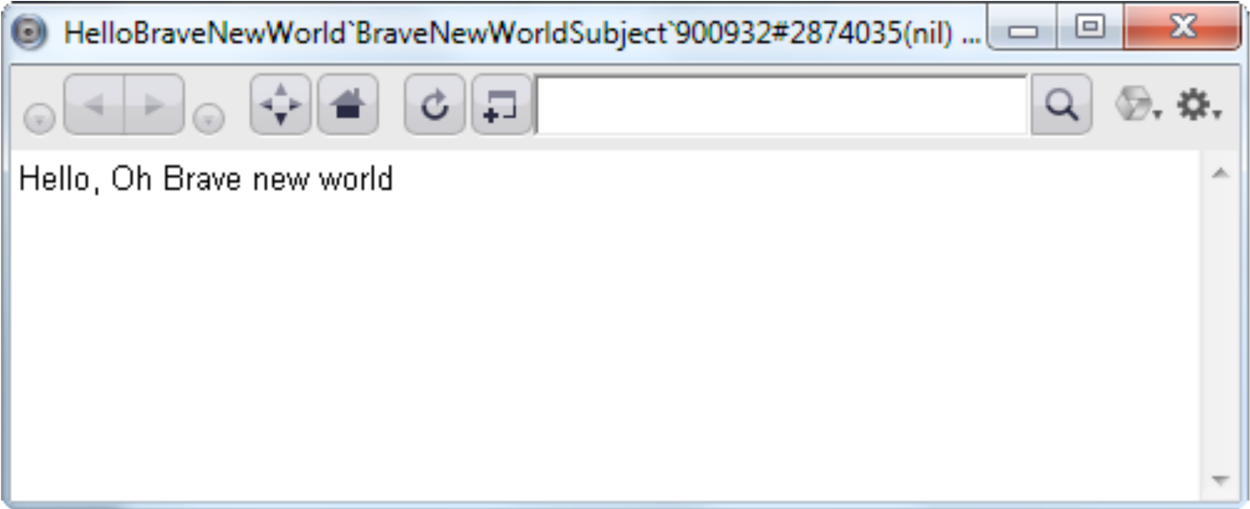
At this stage, things are so simple that our subject class does almost nothing. It's only function is to determine what kind of presenter to use, by means of its **createPresenter** method. We don't even need a model. So add this method to **BraveNewWorldSubject**:

```
createPresenter ^ <Presenter> = (
	^BraveNewWorldPresenter onSubject: self
)
```

This is enough to actually open a window. In a workspace, evaluate

ide IDEWindow openSubject: ((HelloBraveNewWorld usingPlatform: platform) BraveNewWorldSubject onModel: nil)

You should see something like this:



Still not terribly interesting, but we now have a Hopscotch window running our application. This the modern day equivalent of *Hello World*.

This is a good time to take a break. Get yourself a glass of *Chateau Margaux* or whatever your favorite beverage is, and come back when you're ready.

To the FAQ/Table of Contents

Intermission

Our next, and most substantial, example will require a few more language constructs. It's best to introduce them now via very simple examples so that you are up to speed when the real fun starts.

Keyword Messages

Until now, we've only seen messages that take zero or one arguments. This helped keep the presentation simple. However, the keyword syntax generalizes to multiple parameters. Consider the mathematical function *max(x,y)*. One could declare a method

```
max: x <Number> and: y <Number> = (
        (* some code *)
)
```

and invoke it like this:

```
max: 3 and: 4 (* just like max(3, 4) *)
```

The idea is that the colons mark the positions of the arguments - sort of the way ***%*** marks data positions in a **printf** string. The method's name is **max:and:**, but when invoked, the arguments are interspersed within the name. The order of the keywords matters, so **and:max**: is a different method altogether.

In terms of precedence, binary expressions have higher precedence than keyword expressions, and unary expressions have higher precedence than binary ones:

```
min: 3 + 4 and: 3 factorial + 2 (* evaluates to 7, just like min(3 + 4, 3.factorial() + 2) *)
```

Some more examples

```
Address number: 1600 street: 'Pennsylvania Avenue' city: 'Washington' state: 'DC'
country: 'USA'
(* Roughly like Address.new(1600, 'Pennsylvania Avenue', 'Washington', 'DC', 'USA') *)
```

min: (max: 3 and: 4) and: 3 (* evaluates to 3, just like min(max(3,4), 3) *)

parser parse: 'printf("max %d and %d", ++p*, --q**[3]);' inContext: getParserContext
(* parser.parse('printf("max %d and %d", ++p*, --q**[3]);', getParserContext()) *)

This syntax may require some getting used to, but it grows on you. Trust me, you've dealt with much weirder syntax. The keyword syntax lends itself to defining internal DSLs. Code like the *Address number: ...* example above is much more readable this way. Another advantage is that you can't get the arity wrong.

Tuples

Here is a literal tuple: **{**'six'. 3 + 4 min: 6. 3 factorial**}**. It denotes an array with 3 elements - the string 'six', the number 6, and the number 6 again. Array indexing begins at 1. Maybe we'll change this someday.

Closures

A **closure** is is a block of code representing an action you want done. Closures are delimited by square brackets. A very simple closure would be:

[3+ 4**]**

The expression inside the closure, 3 + 4, is not evaluated until the closure is invoked. To invoke it, send the closure the message **value**, as in

[3+ 4**]** value (* evaluates to 7 *)

The value returned by a closure is the value of its last statement.

[3+ 4. 42**]** value (* evaluates to 42 *)

A closure is of course, an object, like everything else in Newspeak.

Closures may have parameters. This example has two: **x** and **y**.

[:**x** :**y** **|** x + y**]** value: 3 value: 4 (* evaluates to 7 *)

The parameters are identified by prefixing them with a colon. A vertical bar marks the end of the parameter list. The closure is invoked using a **value:** message whose arity matches that of the closure.

Closures can contain return statements. This is useful to manage control flow.

[*To the FAQ/Table of Contents*]

Control flow

Newspeak has no built in control constructs - all operations are method invocations/ message sends, without exception.

The most important example of this is the **ifTrue:ifFalse:** method defined on boolean objects.

```
x > y ifTrue: [x] ifFalse: [y] (* evaluates to x if x > y; y otherwise *)
```

As you can see, it takes two closures as arguments - one for the true branch and one for the false branch. If the receiver is true, it will invoke its first argument and return the result. If it is false, it will act similarly, but invoke the second argument.

```
max: x <Number> and: y <Number>  = (
	x > y ifTrue: [^x] ifFalse: [^y]
)
```

The method above implements the mathematical function max(x,y). It's important to understand that a return statement always returns from the nearest enclosing method - **max:and:** in this case - not the enclosing closure. The above behaves the same as

```
max: x <Number> and: y <Number>  = (
	^x > y ifTrue: [x] ifFalse: [y]
)
```

This latter version is much better style of course.

How does the Syntax Differ from the Specification, and Why?

Sigh. Newspeak has been evolving gradually from Smalltalk. The specification indicates the way things should be, but the implementation lags.

Expect the following changes, and possibly more:

Classes, methods and closures will use curly braces as delimiters instead of parentheses.

Tuples will use square brackets as delimiters instead of curly braces.

String literals will be delimited by double quotes, instead of, or in addition to, single quotes and denote interned strings (aka symbols).

Symbol literals (currently written as #sym) may be eliminated

Character literals will be merged into strings.

The return statement may be written **return: e** rather than **^e**.

Object literals will be supported.

Kernel`Object

The default superclass for Newspeak classes is **Kernel`Object**, which is distinct from Squeak's **Object** class. Squeak's **Object** class contains a lot of methods, almost none of which will be in Newspeak's library. You can browse **Kernel`Object** and get an idea of how small the API of **Object** should be.

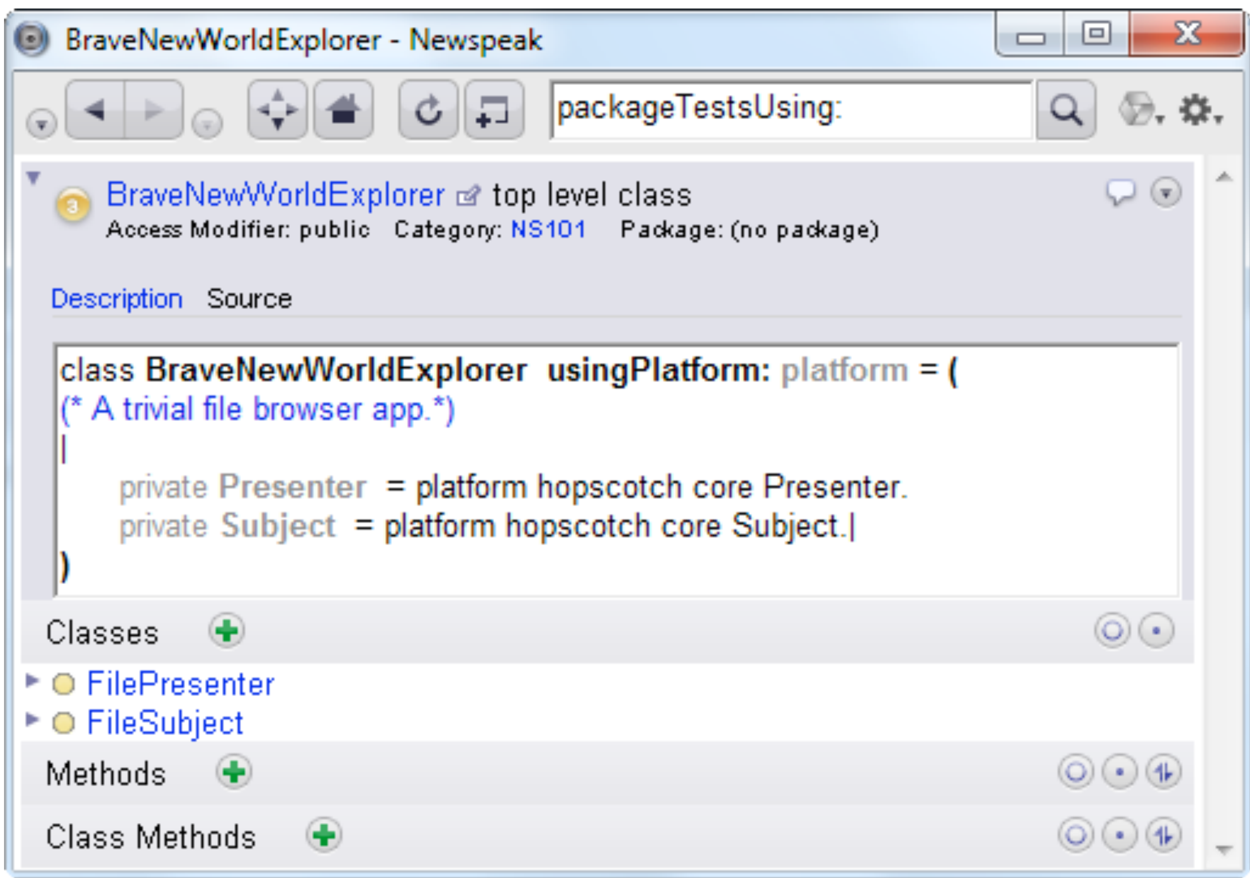
You now know all you need to know to move on to our next conquest.

Exploring the Brave New World

To make things more interesting, our next task will be to build an application that really explores the world around it - specifically, the file system.

We'll start by defining a new top level class called **BraveNewWorldExplorer**. Again, we'll import **Subject** and **Presenter**. We'll define two nested classes - **FileSubject** and **FilePresenter**.

To the FAQ/Table of Contents



This time, we will need to have a model for **FileSubject** It will be a file name - a string that gives the fully qualified pathname for the file. Make sure the definition of **FileSubject** is as follows:

class FileSubject **onModel:** m **<String>** = Subject onModel: m **()()**

and that **FilePresenter**'s definition is:

class FilePresenter **onSubject:** s **<Subject>** = Presenter onSubject: s **()()**

We'll add a very simple definition method to **FilePresenter**

```
definition ^ <Fragment> = (
	^label: subject model
)
```

Why are things red again?

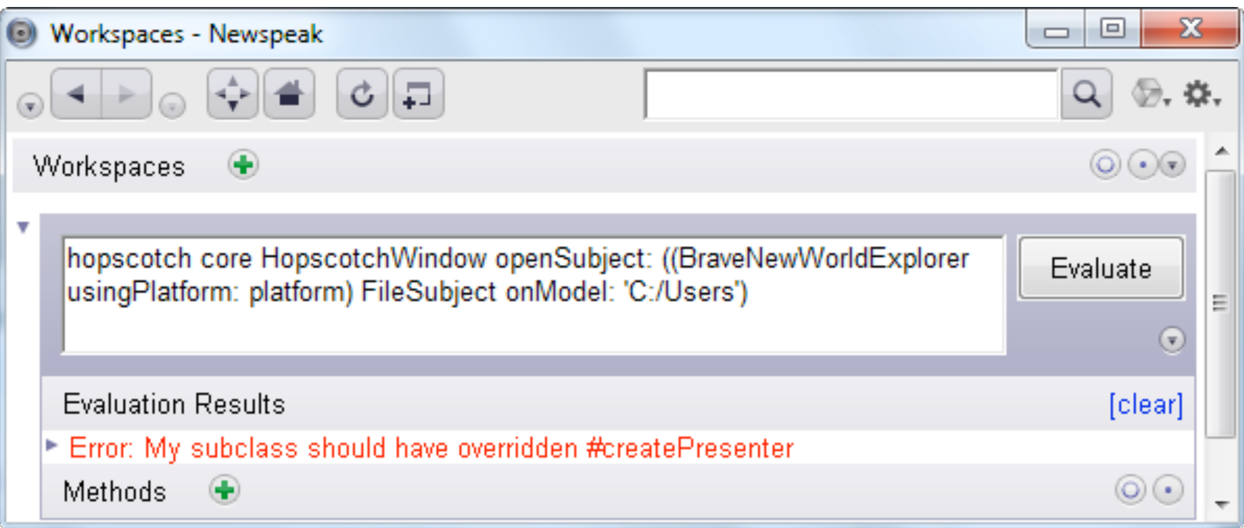
As in the previous example, we've used the **label:** method, which is inherited by all presenters.

To the FAQ/Table of Contents

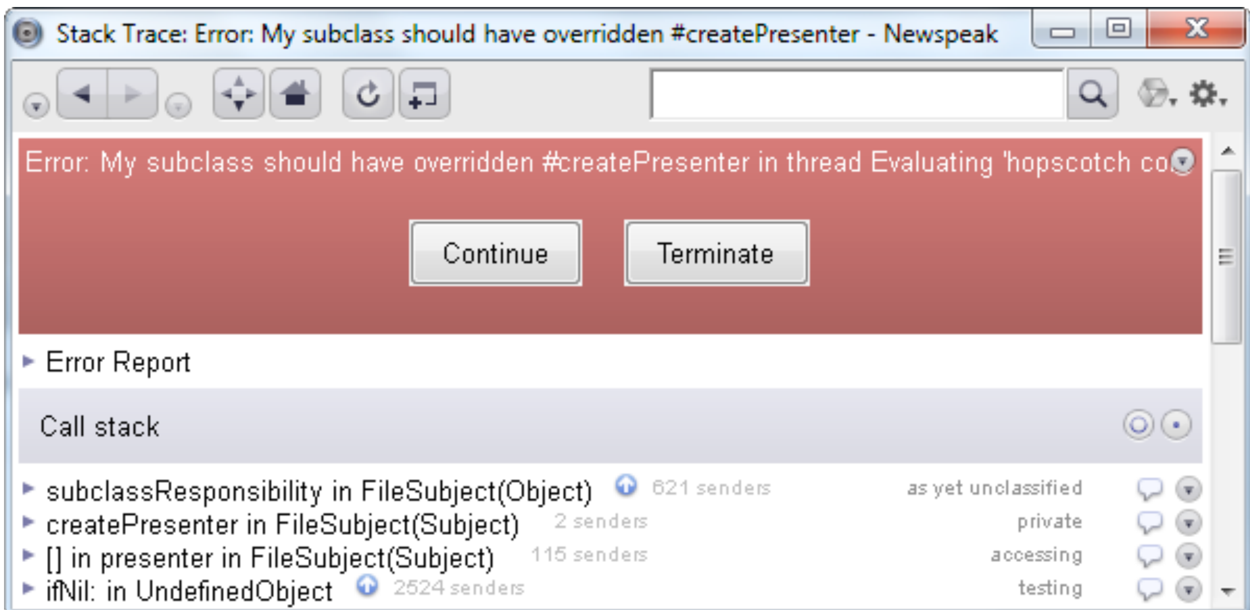
We can now open a workspace and type in

hopscotch core HopscotchWindow openSubject: ((BraveNewWorldExplorer usingPlatform: platform) FileSubject onModel: 'C:/Users')

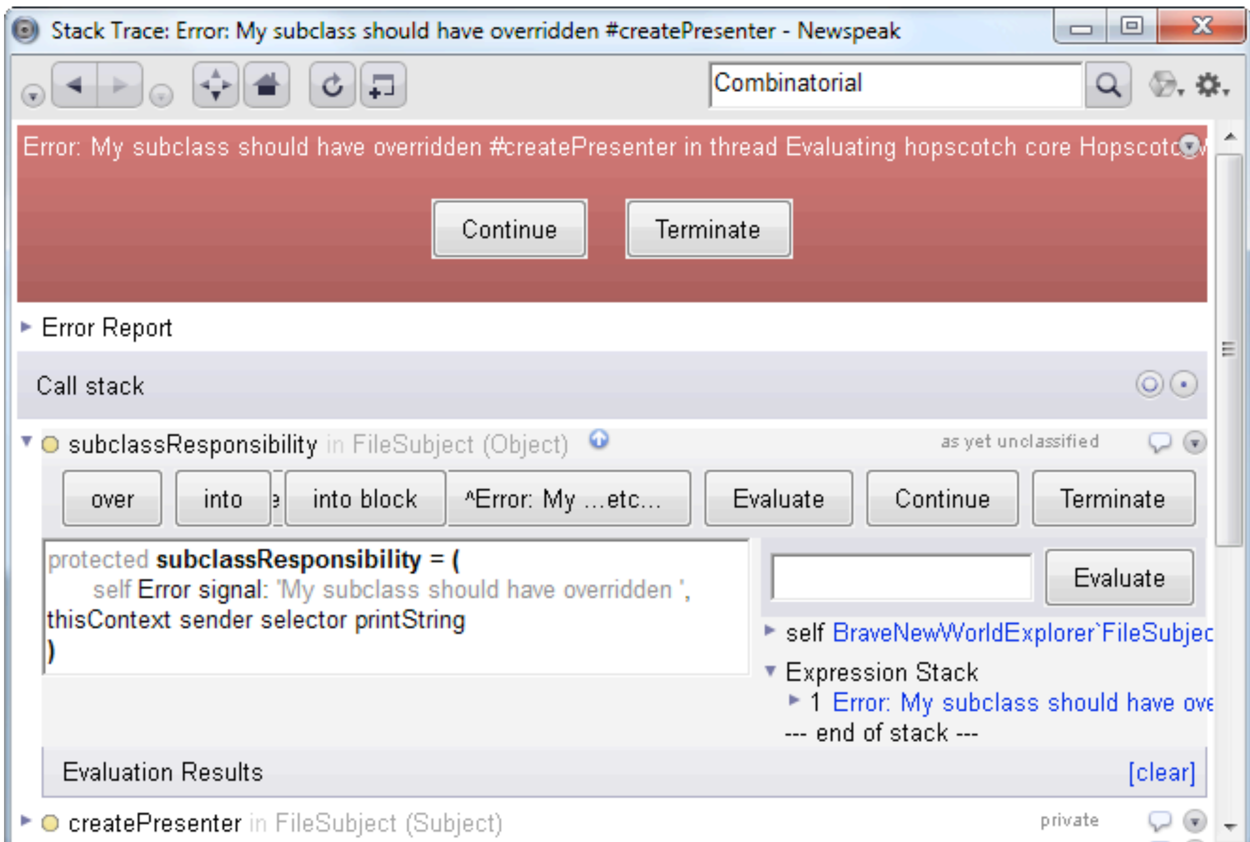
You may need to change the string *'C:/Users'* if you aren't running on a common Windows setup; replace it with a fully qualified path name that works on your system. Unfortunately, evaluating this fails:



You may recall that **FileSubject** should have a **createPresenter** method that determines what kind of presenter should present it by default. To correct this situation,



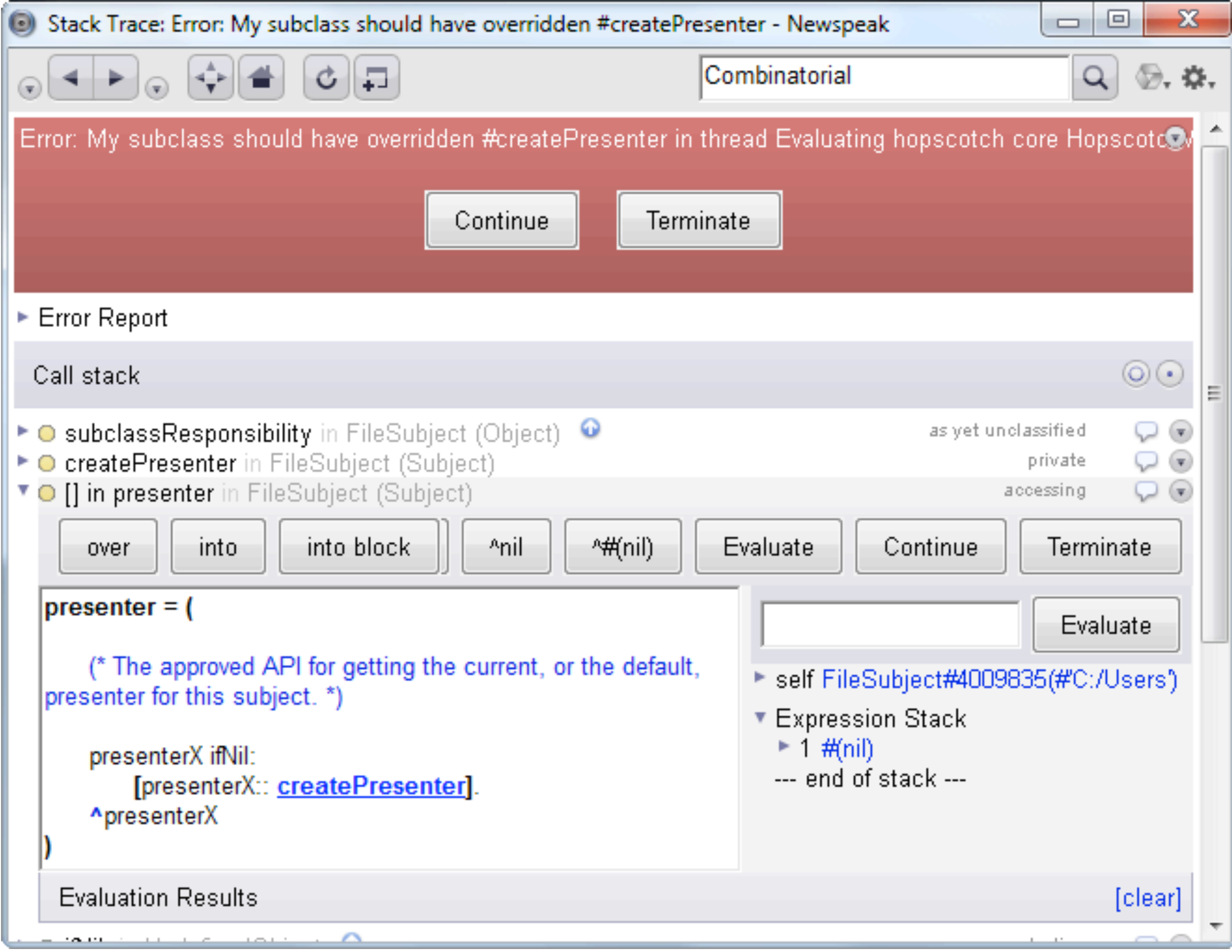
follow the Error: My subclass should have overridden #createPresenter link.
Next, expand the **subclassResponsibility** frame:



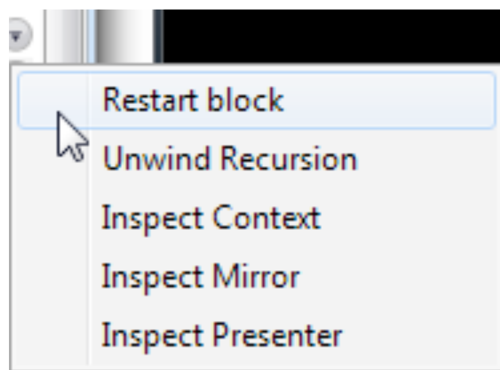
Clicking on FileSubject will take us to **BraveNewWorldExplorer`FileSubject**.
Add this method:

```
createPresenter ^ <Presenter> = (
        ^FilePresenter onSubject: self
)
```

Then go back to the debugger. We can see that **createPresenter** is called from the method **presenter**, so let's open the activation of **presenter**:



Now use choose *Restart block* from the menu



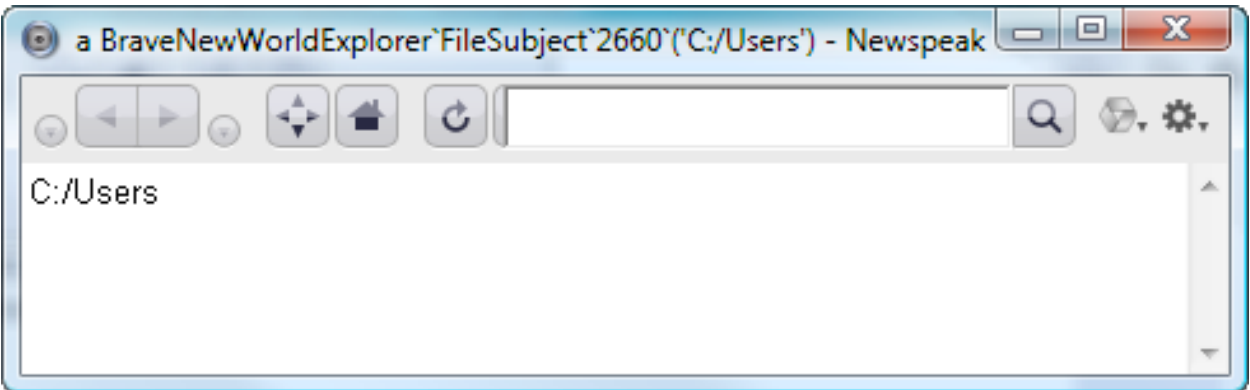
We should be able to proceed successfully from this point. Of course, if we had remembered to add a **createPresenter** method at the beginning, we wouldn't need to

do all this. However, this tutorial deliberately neglected to remind you of this point, so that we could demonstrate a common style of program development:

Rather than define everything up front, we sometimes choose to leave methods undefined and let the program fail on purpose. We can then add the missing method at the point of failure. Working this way, we often have a better sense of the sort of data that will be available to us in the method. Some developers really love working this way. On the other hand, there is some risk that your test paths will miss the method altogether, and so there is something to be said for defining the method in advance.

This concludes our digression into the debugger. Now click on the *Continue* button.

You should see a new Hopscotch window, like this:



It would be more useful if we could actually browse the structure of the */Users* directory. To do this, we'll refine both our subject and our presenter.

We'll need to be able to determine whether we're looking at a directory or a simple file. So let's add an **isDirectory** method to **FileSubject**.

```
isDirectory ^ <Boolean> = (
        ^(FilePath for: fullFilePath) isDirectory
)
```

The code above shows how this is done using the Newspeak libraries. A **FilePath** represents a path in file system in an abstract manner, independent of OS specific details like what separator character is used. We construct it based on from our model, a string representing the absolute file name. **FilePath** is marked as a suspect implicit send. Unlike previous cases, this is not a spurious warning.

We need to import **FilePath** into the **BraveNewWorldExplorer** class, like so:

```
FilePath = platform files FilePath.
```

This is an example of how the language forces you to keep all your external dependencies explicit.

The highlighting of **fullFilePath** isn't spurious either; this isn't an inherited method. We'll define **fullFilePath** as an alias of **model**.

```
fullFilePath ^ <String> = (
	^model
)
```

It's clearer, and our presenter doesn't need to know if we use the path name as a model, or something else (like a **FilePath** object).

We'd rather not display full path names all the time, so let's add

```
localFileName ^ <String> = (
	(* Answer only the file name portion of the path name *)
	^(FilePath for: fullFilePath) simpleName
)
```

Note that **FilePath** isn't red anymore.

We also need a way of getting the contents of a directory. The code below will do this.

```
contents ^ <Collection[FileSubject]> = (
	(* Answer a collection of subjects on the receiver elements *)
	^isDirectory
		ifTrue:
			[ | thisDirectory |
			thisDirectory: (FilePath for: fullFilePath).
			thisDirectory entries collect:
				[:each |
				FileSubject onModel: each name
				]
		]
		ifFalse: [MutableArrayList new]
)
```

You should import **MutableArrayList** from *platform collections*.
The method begins by testing if the current file is a directory. Obviously, if it isn't, it has no contents and we return an empty list. If it is a directory, we compute *thisDirectory*,

the path object for the directory and extract all its elements using the *entries* method. We can then collect subjects for the name of each entry in the directory.

Now let's refine our definition method as follows:

```
definition ^ <Fragment> = (
	^subject isDirectory
		ifTrue: [directoryPresentation]
		ifFalse: [label: subject localFileName]
)
```

We'll also need to define *directoryPresentation*.

```
directoryPresentation ^ <Fragment> = (
	^heading: (label: subject localFileName)
	 details: [column: directoryContentsPresenters]
)
```

What's going on here? The method **heading:details:** is, like **label:**, a fragment combinator inherited by all presenters. It creates a collapsible heading. The first argument defines the collapsed form, and the second determines the expanded one.

Drilling down, the first argument is the result of a familiar call to **label:**. In general, it could be anything that evaluates to a fragment. The second argument is a closure; this is required so that the expanded view can be computed later, at the time of expansion.

The closure must return a fragment. In our case, the body of the closure is an invocation of the **column:** combinator. As you'd expect, it constructs a vertical column, where the rows of the column are given by its argument, which is a list of fragments. The list in question is the result of **directoryContentsPresenters**, which we define as:

```
directoryContentsPresenters ^ <Collection[Presenter]> = (
	^(subject contents) collect: [:each | each presenter]
)
```

this computes a presenter corresponding to each file in the directory. Presenters are a kind of fragment, so they can be used with fragment combinators like **column:**.

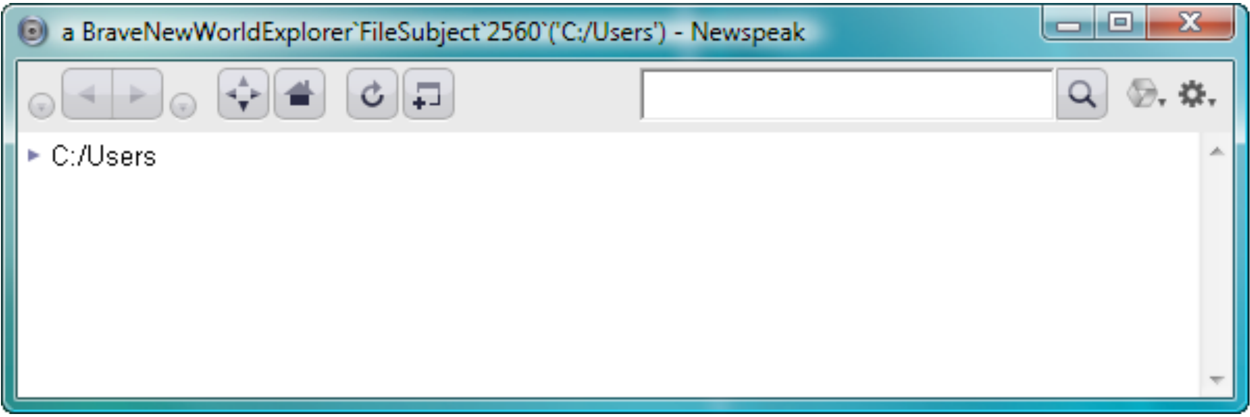
Finally, let's do a small refactoring and add a method **filePresentation** to **FilePresenter**, which we can then use in both **directoryPresentation** and *definition*.

```
filePresentation ^ <Fragment> = (
	^label: subject localFileName
)
```

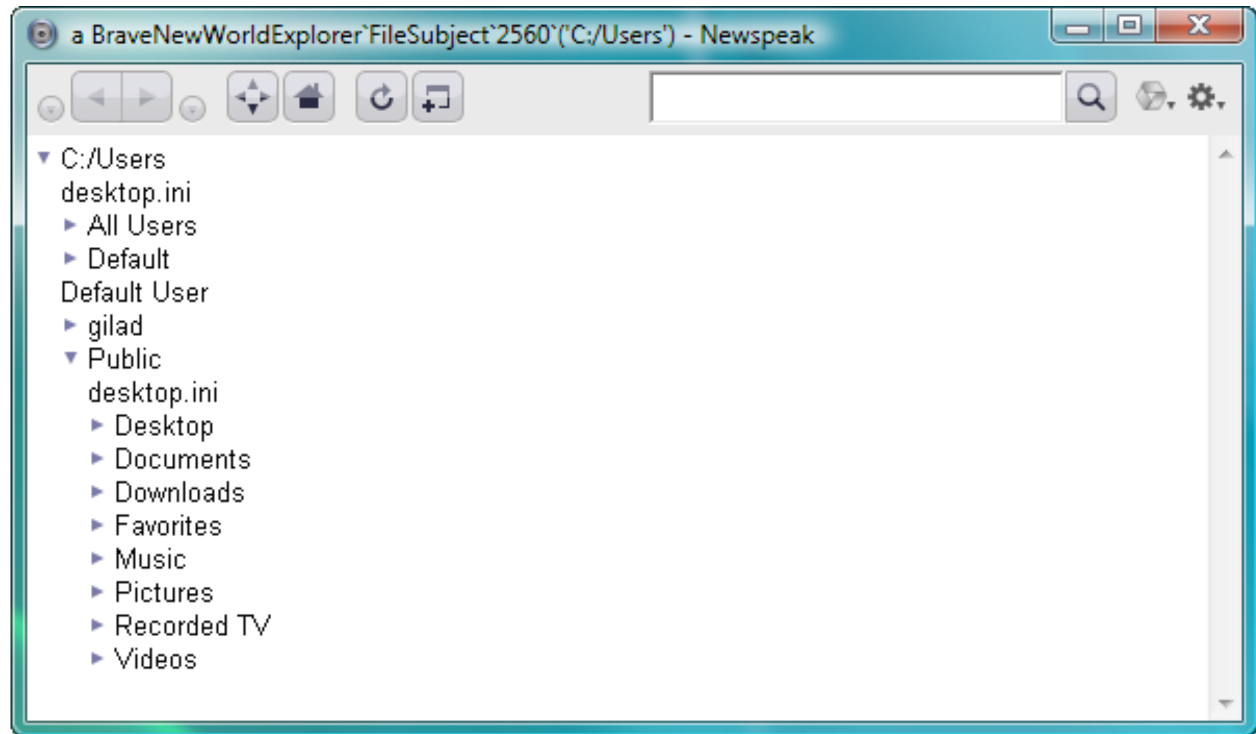
```
directoryPresentation ^ <Fragment> = (
        ^heading: filePresentation
         details: [column: directoryContentsPresenters]
)

definition ^ <Fragment> = (
        ^subject isDirectory
                ifTrue: [directoryPresentation]
                ifFalse: [filePresentation]
)
```

Go back to your workspace and reevaluate the code.



We're finally getting somewhere. If you click on the arrow, you can see the directory contents.



To pretty things up, add this method to **FilePresenter**.

```
bar: def <Fragment> ^ <Collection[Presenter]> = (
	^(column: {
		blank: 2.
		row: {
			blank: 4.
			elastic: def.
			blank: 4.
		}.
		blank: 2.
	}) color: (Color gray: 0.9)
)
```

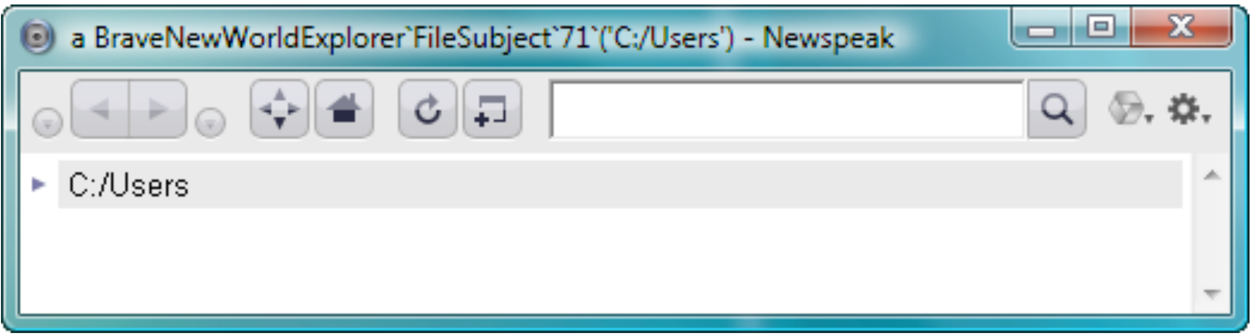
and change **directoryPresentation** as follows

```
directoryPresentation ^ <Fragment> = (
	^heading: (bar: filePresentation)
	 details: [column: directoryContentsPresenters]
)
```

You'll need to import **Color** thusly

    Color = platform graphics Color.

Now re-evaluate the code in the workspace.



What have we done? We defined a new fragment combinator, **bar:**, which displays its argument fragment in a grey bar. The inherited combinator **blank:** creates *n* pixels of blank space given an integer argument *n*; **row:** is analogous to **column:**; and **elastic:** makes its argument stretchable to fill up the available space. You can set the color of a fragment using **color:**.

By now you should have a sense of how you can build up display structures using fragment combinators. Notice how the resulting code looks like a purpose built domain specific language (DSL) for describing the GUI. You can think of Hopscotch as such a DSL embedded in a general purpose language (aka an internal DSL).

This sort of thing isn't specific to the GUI library, though it is an excellent example. Newspeak's features conspire to allow you to easily define such internal DSLs for all sorts of purposes.

Now let's make yet another change to **directoryPresentation**.

```
directoryPresentation ^ <Fragment> = (
	^heading: (bar: (link: subject localFileName action: [openOnNewPage]))
	details: [column: directoryContentsPresenters]
)
```

The **link:action:** combinator produces a hyperlink. The first argument determines how the link is displayed. The second is a closure that is invoked when the link is clicked on.

To the FAQ/Table of Contents

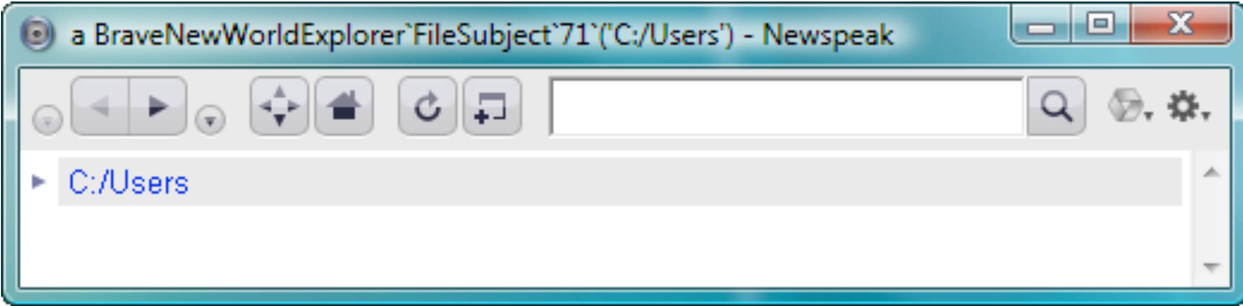
The action in question is defined as

```
openOnNewPage = (
	enterSubject:: subject class onModel: subject model
)
```

Look carefully at **enterSubject:**. You'll notice there is a double colon. This is not a typo. A single keyword message with an extra colon affixed to the selector has lower precedence, so we can avoid wrapping the argument in parentheses.

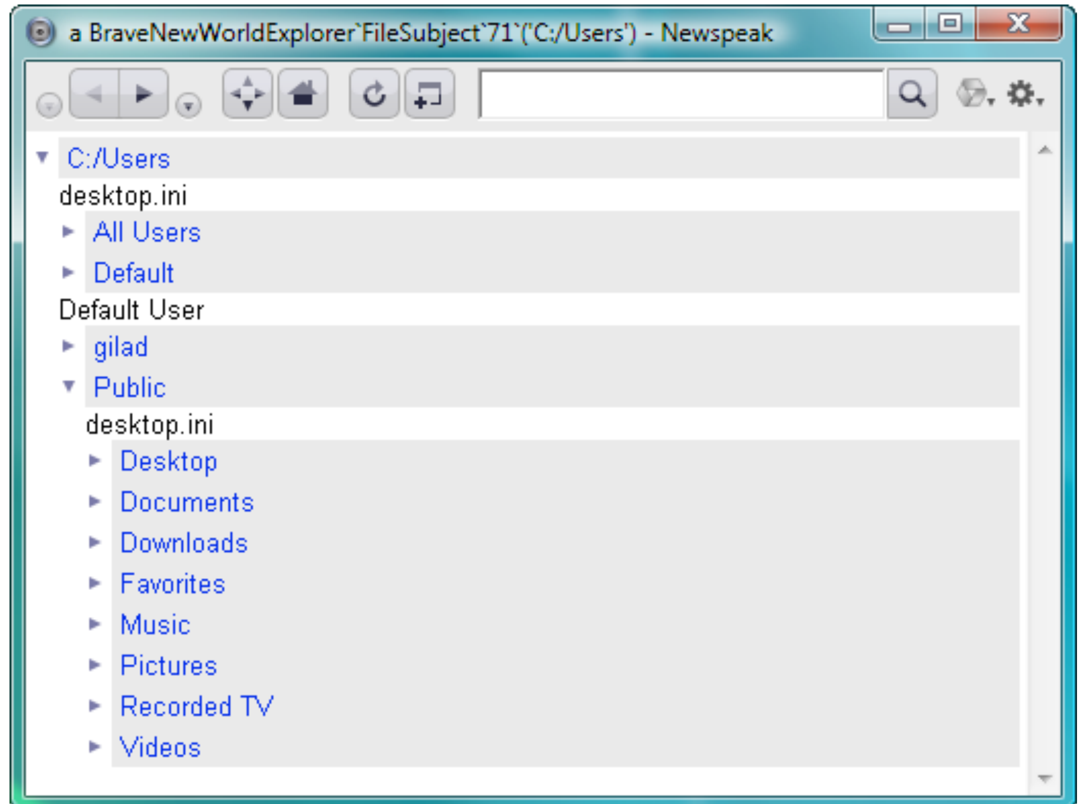
enterSubject: is inherited by all presenters. It takes a subject and causes the Hopscotch browser to make it the current subject of presentation - consequently displaying its presenter. Here, we provide a new subject on the directory as an argument.

Check it out. You just need to refresh the existing window in this case, since we have not added any state to the application, only modified its behavior.

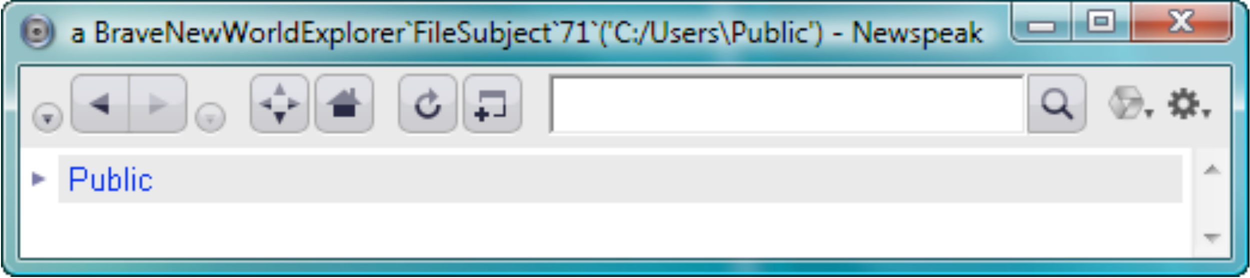


Not bad, but there are still some problems. Following the link seems to have no effect! To understand what's happening, let's first open up the directory hierarchy a few levels by clicking on the arrows.

To the FAQ/Table of Contents



Now follow one of the nested directory links, such as the one named *Public*.



It's not that the links don't work - it's just that the directory is collapsed when you follow the link. This is a nuisance, but we can fix that. Modify the definition of **FileSubject**

```
class FileSubject onModel: m <String> = Subject onModel: m (
|
	initiallyExpanded <Boolean>  ::= false.
|
)
```

To the FAQ/Table of Contents

This adds a *mutable slot* to **FileSubject**. Until now, we've only introduced immutable slots. These slots are set once, in the instance initializer, and are never mutated again. They can't be mutated except via reflection. In contrast, mutable slots can be changed at any time, by means of an automatically defined setter method. For example, to change the value of **initiallyExpanded** to true, write *initiallyExpanded: true.*

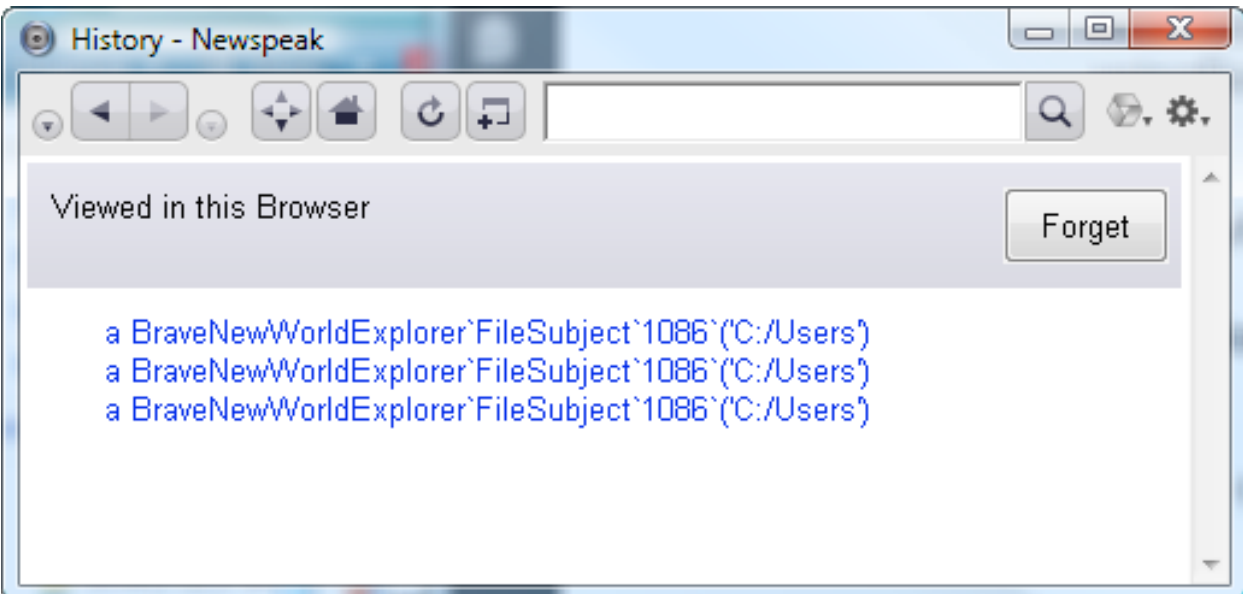
Now make these changes to **FilePresenter** :

```
openOnNewPage = (
	enterSubject:: (subject class onModel: subject model) initiallyExpanded: true
)


directoryPresentation ^ <Fragment> = (
	^heading: (bar: (link: subject localFileName action:[openOnNewPage]))
	details: [column: directoryContentsPresenters]
	initiallyExpanded: subject initiallyExpanded
)
```

Restart the app by re-evaluating the code in the workspace and verify that the links open when you click on them.

There are still some details we should attend to. If you look at the history, you may find that there are multiple entries for the same directory:



What you see will depend on what exact actions you took. The above might result from clicking repeatedly on the link to 'C/Users'. We know that all these are views of the same directory. However, the browser doesn't know this; each is a distinct object, with its own identity.

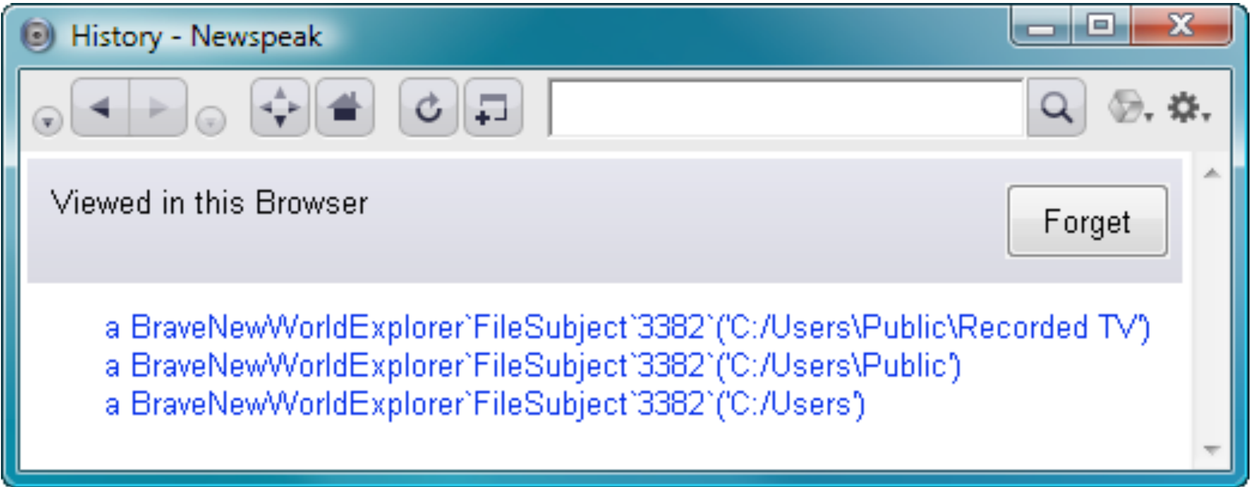
To address this, we can add an equality method to **FileSubject**.

```
= x <Object> ^ <Boolean> = (
        ^x class = class
                and:[x model = model
                and:[x initiallyExpanded = initiallyExpanded]]
)
```

Of course, if we define an equality method, we need to define a hash

```
hash ^ <Integer> = (
        ^class hash bitXor: model hash
)
```

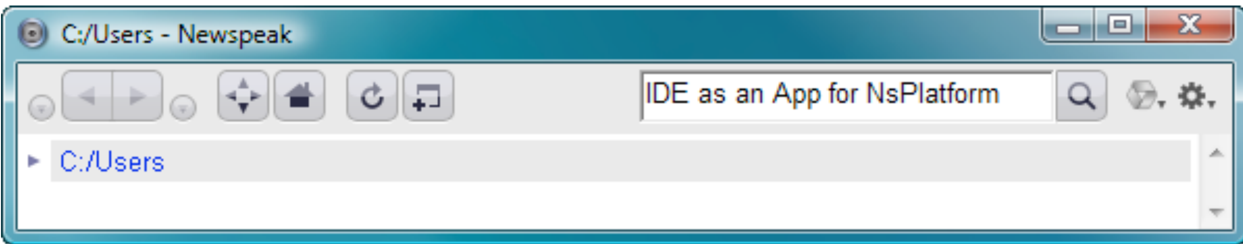
open a new copy of the application by re-evaluating the code in the workspace. Expand the *Users* directory, and follow one of the subdirectory links (here, we'll follow Public), and then follow a subdirectory link again (we'll choose Recorded TV). The history would look something like:



So now we have a reliable view of which directories we've visited, courtesy of the Hopscotch browser.

One more detail: we can control the name the browser displays when presenting a directory by defining a **title** method in **FileSubject**.

```
title ^ <String> = (
        ^fullFilePath
)
```



Having built an application, we should consider how to deploy it.

How do I deploy an application?

Having built an application, there is some question how to deploy it. You, can of course, save your image with your application in its initial state, and distribute that, with the IDE intact. We won't pursue that option further here. We are working toward more convenient options.

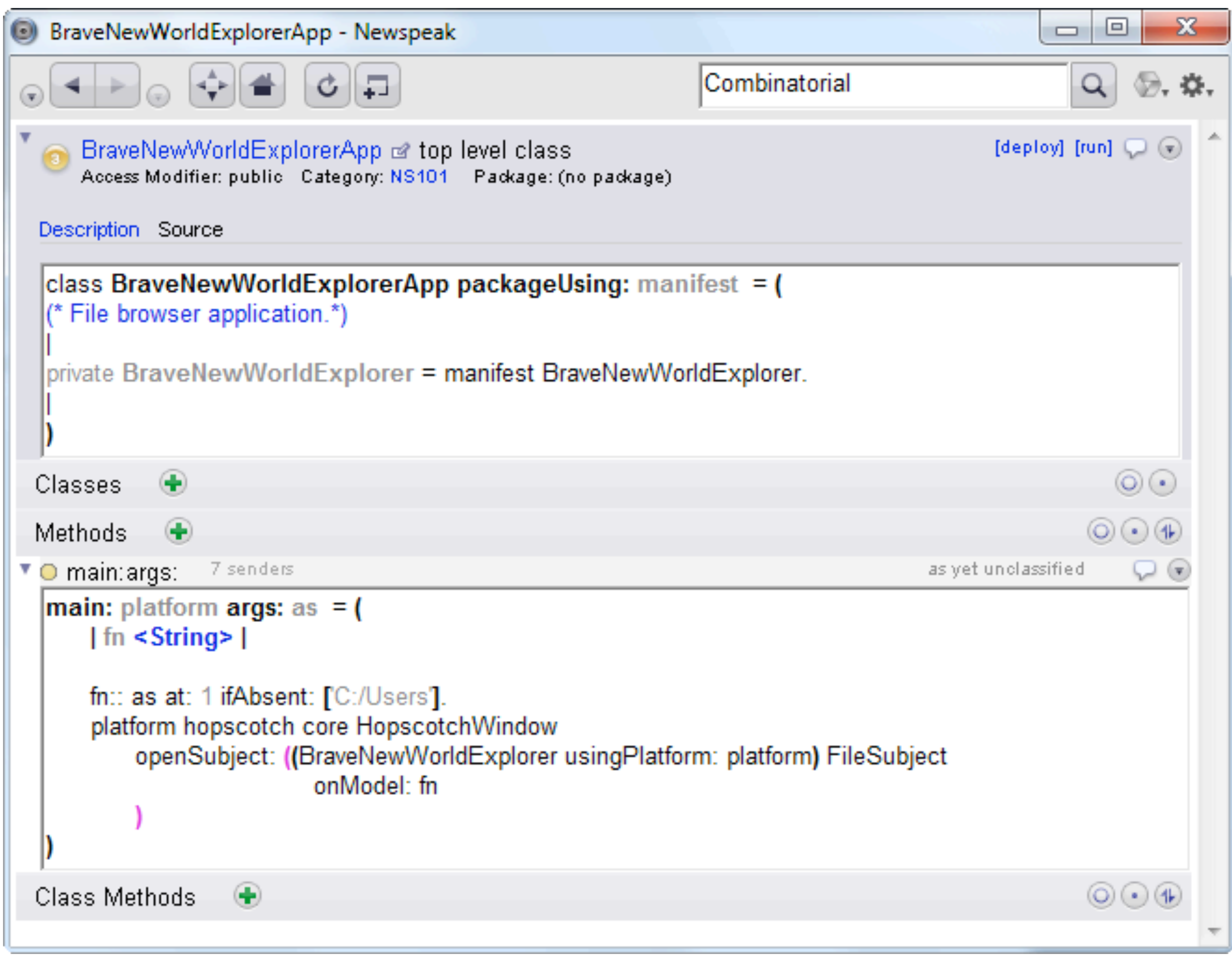
The first step is structuring your code as a stand-alone application.

How do I structure a Newspeak Application?

A Newspeak application is an *object* conforming to a standard API. This API consists of a single method, **main:args:**. In concept, it is similar to the *main()* method of a Java or C program.

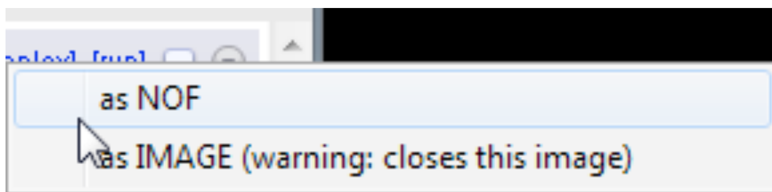
The main method's purpose is to instantiate the various module definitions that make up the application and start running the application code. To create an application object for **BraveNewWorldExplorer**, we'll define the following class:

To the FAQ/Table of Contents



Deploying a Newspeak Application as a NOF File

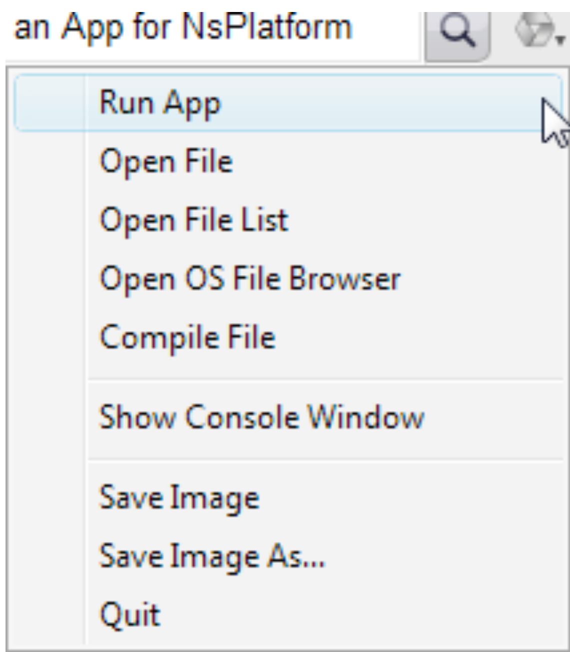
Notice the deploy link near the top right of the class presenter. This link appears whenever a class defines a **packageUsing:** class method. Click on it, and you get a pop-up menu; choose *as NOF*.



Check your working directory: there now two new files: *BraveNewWorldExplorerApp.nof* and its gzipped sibling. At this point, we're done, but it's always good to test your deployed application.

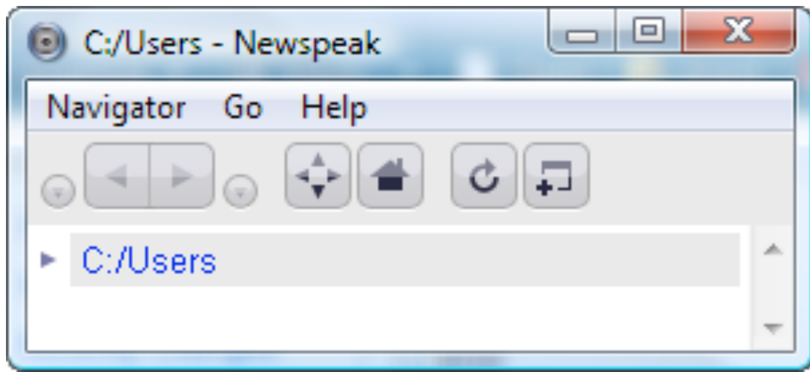
How do I test my deployment code?

You can test your deployable app in the IDE. Click on the *run* link in your application class. If you've actually saved an application in a NOF file, you can choose *Run App* from the operate menu.



This will open up a file chooser dialog, which will let you select a NOF file. Once you choose the file, the IDE will bring it for for you.

Note that if you actually mean to deploy code without exposing the IDE, NOF won't do that for you at this point. In the past, we've supported deploying as .exe, or as an image that disallows access to the IDE. Those options don't work in the current release. What they would give you would be an app like the following:



In this configuration is there is no way for the user of the application to get to the Newspeak IDE. Notice the absence of the operate and meta menu icons and the search pane. All these are features provided by Hopscotch's **IDEWindow**. We configured our app to use a **HopscotchWindow** rather than an **IDEWindow**. After all, we typically don't want the IDE as part of a deployed application.

You'll also note the presence of a menu bar. Again, this is due to the use of **HopscotchWindow**. We at the Ministry of Truth feel that menu bars are somewhat ungood, which is why the IDE doesn't use them. If you are building a real application, you'll have to decide whether to enable the menu bar, and what it should show. This particular menu bar is just a placeholder; you would never want to use it without customizing it.

Because the IDE isn't part of the default deployment set up, any attempt to use elements of it would fail, unless we *explicitly* included by extracting them from the **manifest** parameter in **packageUsing:** and inserting them into the application (as opposed to the Hopscotch framework, which we can obtain from the **platform** in **main:args:**)

Another consequence of this packaging is that if something goes wrong in our packaged application we won't get a nice debugger. Instead, a file named *error* will be deposited in the working directory, with a textual stack trace.

These options did not prove popular; they acted as a proof of concept for how one could deploy applications. To be genuinely useful, more engineering is needed: the produced executable should be much smaller (they hid the IDE rather than eliminating it) and should run natively on today's popular platforms (web, mobile). We hope to address these deficiencies and bring better deployment options in the future.

To the FAQ/Table of Contents

Finally, let's do something crazy. Change the superclass of **FilePresenter** to be **ProgrammingPresenter**.

Note: ProgrammingPresenter comes as part of the IDE itself, not part of the general Hopscotch framework that is part of the basic platform. This makes our application IDE-dependent! We can't deploy it unless we include the IDE explicitly in our application. We won't be doing that - the goal here is just to show off some cool functionality.

You'll need to import **ProgrammingPresenter** :

```
ProgrammingPresenter = ide tools ProgrammingPresenter.
```

You'll see that **ide** is highlighted in red. We need to get access to the IDE to get hold of **ProgrammingPresenter**. So we'll modify the factory of **BraveNewWorldExplorer** to take **ide** as a parameter.

```
BraveNewWorldExplorer usingPlatform: platform ide: ide
```

Now we'll add a menu. The menu won't do anything you'd do in a regular application. Rather, it will do something cool, that isn't easily done in a traditional IDE.

We'll define our menu as

```
filePresenterMenu = (
	^menuWithLabelsAndActions: {
		'Inspect Presenter' -> [respondToInspectPresenter].
		'Show Implementation' -> [respondToShowImplementation]
		}
)
```

The combinator **menuWithLabelsAndActions:** takes a list as an argument. Each list element describes a single entry in the menu. An entry is described via an association which maps a string (the label of the entry) to a closure that describes the action to be taken when the entry is selected. Add the two methods invoked from the menu:

```
respondToInspectPresenter = (
	inspect: self
)
```

```
respondToShowImplementation = (
	browseClass: class
)
```

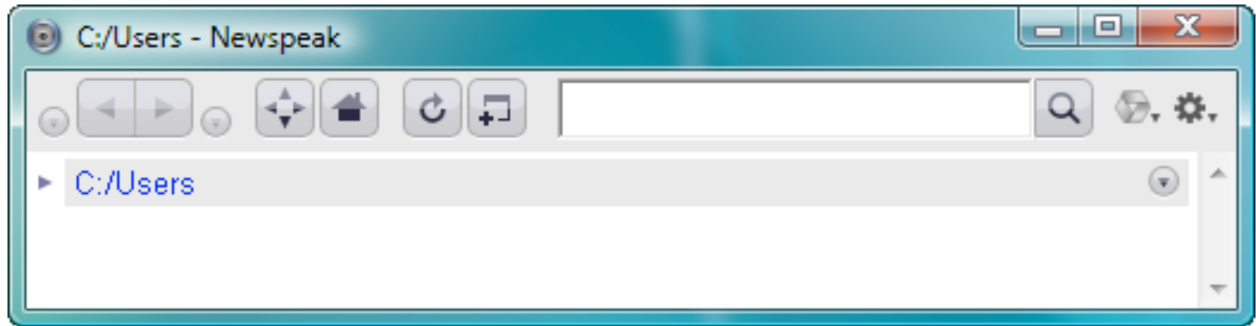
and update *directoryPresentation*

```
directoryPresentation ^ <Fragment> = (
        ^heading:
                (bar:
                        (row: {
                                link: subject localFileName action:[openOnNewPage].
                                filler.
                                dropDownMenu: [filePresenterMenu]
                                }
                        ))
        details: [column: directoryContentsPresenters]
        initiallyExpanded: subject initiallyExpanded
)
```

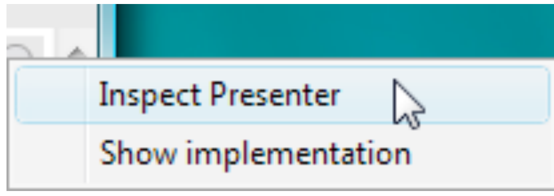
We need to modify the incantation to run our application, because we added an extra parameter. Evaluate:

ide IDEWindow openSubject: ((BraveNewWorldExplorer usingPlatform: platform ide: ide) FileSubject onModel: 'C:/Users')

This will produce a new window.

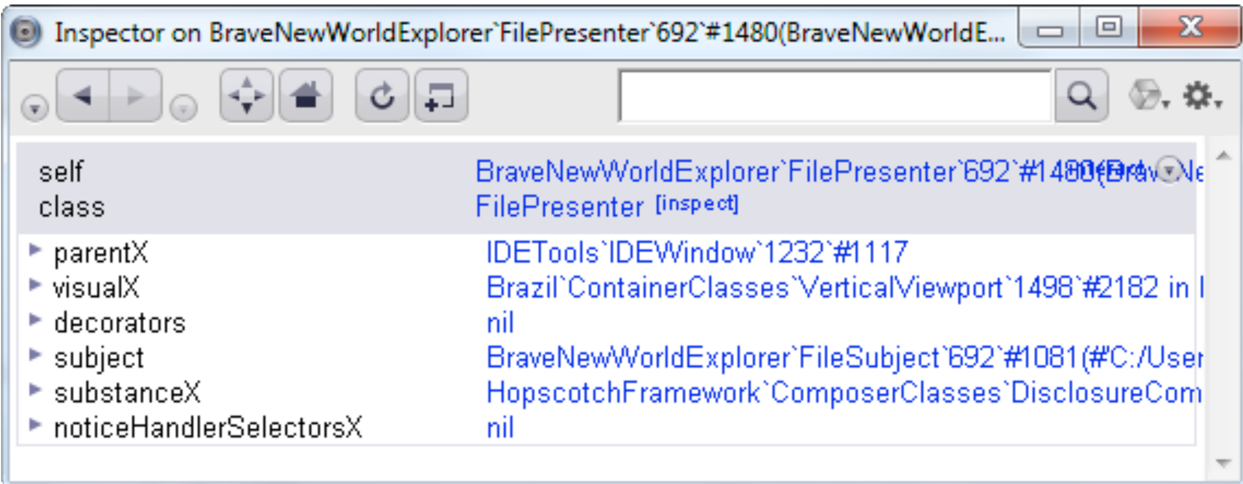


Next choose *Inspect Presenter* from newly added drop down the menu on the right.

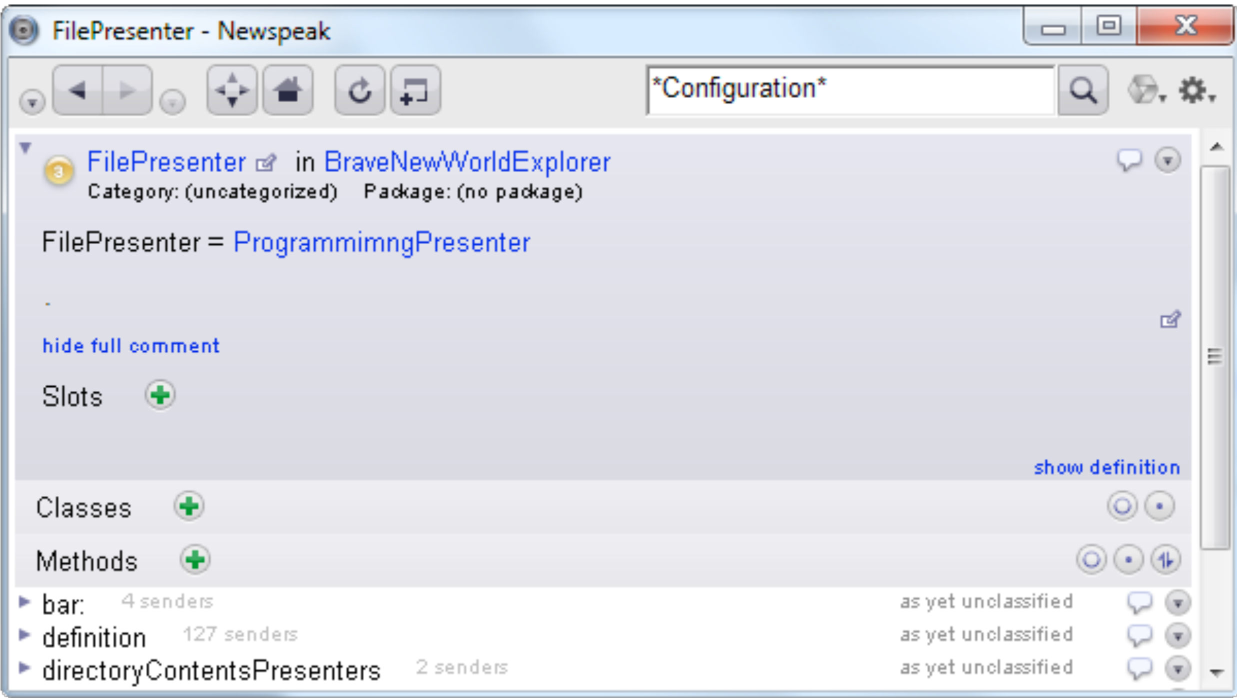


[To the FAQ/Table of Contents](#)

You'll see an object inspector on the live **FilePresenter** instance managing the presentation.



Now choose the *Show Implementation* menu option from the menu. This allows us to directly access a class browser on the **FilePresenter** class, so our application is directly metacircular.



In reality, these options are completely inappropriate for a file browser, or for any end-user facing application. Even if you don't care about exposing your IP in this way, they pose a security risk. They are only included here to show how you can easily extend and integrate with the Hopscotch IDE.

What's next

We hope you enjoyed this peek into the brave new world of Newspeak. Now it's up to you to help Newspeak mature into a platform that can survive in the cowardly old world into which it was born.

The Newspeak language home page has links to forums where you can ask questions and to our open source repositories, where you can get updates - either via MemoryHole or, occasionally, as new images.

Newspeak is an open source project. It is still far from finished, and needs a lot of work to realize the vision we have for it. If you appreciate the ideas and their potential, we hope you'll use it and contribute to it.