Converting Smalltalk to Newspeak

Gilad Bracha

This guide offers advice on how to convert existing Smalltalk code to Newspeak; specifically, the Newspeak2 dialect current as of March 2010. This document assumes you have a basic familiarity with both Smalltalk and Newspeak. Otherwise, you're not ready to do the sort of port/conversion we envisage here.

You will have to decide whether you want to convert the code into a single monolithic Newspeak class, or represent it via several top level classes that you link together. If your code base is very large, the latter is better. If it is of moderate size (say, less than 5 KLoC, certainly no more than 10 KLoC) you may prefer to keep it in one top level class.

If at all possible, get your code into a Newspeak image. Newspeak images are currently variants of Squeak, and can safely file in Smalltalk code in file out format.
Ensure that your code goes into its own distinct categories.

Of course, if your code includes any class whose name, **N**, is identical to that of a class already in the Squeak environment, you may run into problems because your file-in may modify **N** and break it. This is exactly the sort of weakness of Smalltalk that Newspeak excels at solving.

If this the case, rename your class N temporarily to avoid conflicts. You need not rename uses of the class **N** in Smalltalk code. We only need to rename those places in the file-in that would modify the existing class **N** (such as, 'N methodsFor: ..., and 'N class methodsFor: ....' etc.).

Now that you have your Smalltalk code loaded, you can start the conversion process. There are two possible approaches. One is to print out your classes in an actual syntax that is close to Newspeak syntax rather than file out format, and continue the conversion in a text editor.

The other option is to use tools in the programming environment to do some of the work for you. To be honest, these tools are immature and need some work, so the choice is not clear cut.

We'll describe both paths in this document.

Manual Conversion

Fully manual conversion is probably best for relatively small bodies of code.
View each class you wish to convert in the Hopscotch browser. Choose the option Print class from the class presenter's drop down menu. If your Smalltalk class is named Foo, you'll find a file named Foo.st in your working directory. Open it in your favorite text editor. You'll see that this isn't a file-out format file. Rather, it uses a real syntax for classes, not dissimilar to Newspeak syntax. We refer to this as NS0 (Newspeak-0) syntax - essentially Smalltalk with a top level class syntax.
There are still some adjustments to be made, but much of the syntactic conversion has already been done. However, you still need to deal with some syntactic issues.

Assignments

Convert ':=' (or, if you're code originated in Squeak or some ancient Smalltalk-80, '_') to '::', eliminating any leading spaces before the assignment. You will need to be careful if you have code that is chaining assignments, for example, x := y := z. The conversion to x:: y:: z won't be legal Newspeak code. Since it won't parse, you'll eventually find out and have the chance to fix these things manually.

Class headers

You need to add the keyword **class** in front of the NS0 class declaration. Also, NS0 has no separate instance initializer. Just add ')(' after the instance variables (or after the opening parentheses of the class declaration if there are no instance variables). Unless you expect this class to be a top level class in NS2 (highly unlikely), remove the first two lines of the file - the class category and language (Smalltalk) as well. In the very rare eventuality that this is going to become a top level class, just replace the first line with the word *Newsqueak2*.

Now you can save the file as Foo.txt. You'll need to repeat this exercise for each of your Smalltalk classes.

Primitives

Primitives are of course syntactically invalid in Newspeak. They should become calls to **VMMirror**.

Newspeak has no notion of primitives. Operations which may not be plausibly implemented directly in Newspeak are performed by sending a request to the Newspeak VM. Concretely, this means obtaining an instance of **VMMirror** and sending it the appropriate message.

The interface of **VMMirror** needs to be fixed - essentially determining a standard set of primitives. This has not yet been done. If you find you have code that uses a primitive that is not implemented by **VMMIrror** or by a library class, please post to the Newspeak forum.

If you are converting from Squeak, it may be that you have code that is invoking a primitive defined by a plugin. Such code should be treated as a foreign function call as described in the following section.

Foreign Function Calls

Many Smalltalk dialects have FFI calls that use a special syntax based on the primitive syntax. In contrast, Newspeak has no special syntax for foreign calls. Such calls should be converted to use Aliens.

Note: Some calls require parameters of non-trivial C datatypes. Aliens for these need to be constructed. The **Alien** documentation recommends subclassing **Alien** for this purpose. However, it is preferable to use composition rather than inheritance for several reasons. One is simply that NS2 classes cannot inherit from variable-sized classes such as **Alien**. It is also sometimes useful to interpret the same alien in different ways, which is easier if different objects share a reference to the same alien.

Class Methods

Usually, we'd worry about class methods later in the process, but there are some issues that make it advisable to attend to them early.

An explicit **new** method causes the NS2 system to go bonkers, because the classes don't have a primary factory, so a synthetic **new** method is expected, but the explicit one interferes with that. Best to get rid of **new** before compiling anything.

For any class that has a class method **new**, either:
(1) introduce an explicit primary factory
(2) eliminate **new**
(3) rename **new**

Often, **new** just calls **initialize**. In most cases, **initialize** just sets some slots. So I try and use the slot initialization syntax and get rid of **initialize** and **new** altogether. If

initialize makes any calls, they can usually go after the slots in the instance initializer as well.

Another issue is that class methods are not inherited. If these are factory methods, one may have to replicate them in subclasses. Other methods can often be moved out of the class into the enclosing class.

Accessors

Smalltalk code may have manually defined accessors, whose names are derived from the instance variable names being accessed exactly as in Newspeak. These conflict with the automatically generated ones, and confuse the system badly. Like **new** above, they should be eliminated before submitting code to the Newspeak compiler.

Module Boundaries

At this point, you should be over the syntactic issues, and those special concerns that might effect Newspeak compilation. Now you need to think (ouch!).

As discussed above, you have to decide on the modular organization of your application. Is it going to be one top level class, or several classes that interoperate? This is a serious design exercise. As part of that, you will need to decide on the name(s) of your top level Newspeak class(es) and which classes nest where.

The simplest tack is just to define a single top level class. It also reduces the chances of problems during the port. The safest path is to get things working in one monolithic class, and think about refactoring things later. But sometimes it is clear that there are several distinct parts that should be kept separate.

In any case, the first organizational scheme you come up with may not be idiomatic Newspeak. Even within a single class, a large library may be subdivided among several nested classes purely for structuring purposes - expressing the architecture, managing the namespace etc. But these are things you can do later as well, and experience shows it is easier to work in stages. That said, you should certainly plan on taking the time to refactor after you get the first version working

For simplicity, we will indeed assume that there is a single class **Bar** that represents your application. Create the file Bar.ns2 with an empty Newspeak2 class. Inside the body of that class, you will paste the contents of Foo.txt, for every Smalltalk class **Foo** you converted.

You should now have a syntactically valid Newspeak2 class, or something very close to it. The only remaining issue is one we alluded to above - there may be multiple assignments that got converted automatically, that need to be broken into separate parts. And of course, errors may have occurred in the above steps as well.

The only way to find out is to try and compile Bar.ns2. Once you get past the parser, you'll have **Bar** in the IDE and can continue working on it there.

Tests

The convention of including tests for a class in the class itself is strongly frowned upon in Newspeak. Tests should be factored out into a separate class in *X-tests* category, where *X* is the package for the module. It's best if they are converted to use the **NSUnit** testing facility.

Global, Pool and Class Variables

For each class **Foo**, you also want to check if it defined any class variables. Each such variable should be declared as a slot of the top level class **Bar**. Any globals or pool variables defined by your application should also become slots of the top level class.

It's rare, but possible, for two or more of these classes to define class variables with the same name. Even rarer would be conflicting pool variables, or globals that conflict with pool or class variables.

You can check this once you introduce the top level slot - search for senders of it. If it was originally a class variable but appears in two distinct nested classes, there is a conflict. If it was a pool variable, make sure all the classes it appears in used to access that pool - otherwise it is a conflict.

If that is the case, you'll need to rename at least one of the slots. To be safe, rename both - this will ensure that if you didn't rename a usage, the problem will manifest itself as a message-not-understood error.

The initialization of these variables likely occurred in some class initialization method, or relied on some other mechanism (like VisualWorks parcels). You should ensure that the module level instance initializer takes over these responsibilities - either by directly initializing the slots, or by calling the requisite methods. As with instance level initialization, the former is preferred, unless the initializing methods also make other calls etc.

Imports

In Newspeak, one has to identify all external dependencies. It makes sense to do this once you've defined the slots corresponding to your Smalltalk code's pool, class and global variables. Go through the code, checking out anything highlighted in red,

especially upper case identifiers, which are likely imports (unless you've overlooked a global, pool or class variable, or forgotten to paste in a class).

Of course, inherited methods may be highlighted red as well, but they won't begin with an upper case letter.

Each import will need to be declared and initialized. Which means you will have to define the primary factory for your top level class, if you haven't done so already. Even if you have, you may realize things have to change because of imports you overlooked.

Refinements

Your original Smalltalk code base likely consists of a number of distinct class categories. These categories can serve as a guide to structuring the Newspeak version of the code, especially if the category based organization was a sensible one.

If you decide to keep your code in one top level class, you may want to replace the initial, flat internal structure with nested classes corresponding to each category. This imparts a clean architectural structure to your code. It does, however, introduce an additional level of enclosing object between the code that does the actual work and the module level. It will also tend to break things.

Consider the **HopscotchFramework** class. It contains several nested classes, such as **CoreClasses**. Each of these consists of numerous nested classes, and is analogous to a class category. Corresponding to **CoreClasses**, there is a module variable **core** which holds an instance of **CoreClasses**. Any code within the module (but outside of **CoreClasses**) that needs to access classes defined in **CoreClasses** does so via the module variable **core**.

The structure used in **HopscotchFramework** is a general pattern. Once you have a working NS2 module with many nested classes, you may refactor it in a similar style. For each of your original categories "Some Category", create a nested class **SomeCategory**. Define a module variable

someCategory = SomeCategory new.

Move the classes that originated in the "Some Category" category so they nest inside **SomeCategory**. For each class **N** in **SomeCategory**, you need to track down all sends of *N* in the module that are outside of **SomeCategory**, and replace them with *someCategory N*.

Of course, you may want to refine things further, moving other module variables into nested classes etc. If your original category structure was deficient, you may want to introduce a different organization.

Semi-automatic Conversion

Newspeak offers some basic tools to partially automate the conversion process. The idea is to convert a single class category into a top level Newspeak class. We will want to keep all the classes that will end up being nested in an enclosing top level Newspeak class in the same category, and all classes that belong in a different top level Newspeak class in a different category.

If you plan to produce a single top level class for your application, converting a class category is all you need to do. Otherwise, my need to consider how to tie the various top level classes you produce into a single application or library. However, that process is unrelated to code conversion; it is part of the design of any Newspeak library or application that spans multiple module declarations.

Therefore, we will focus on the conversion of a single category. Throughout this section, the words **manual** or **manually** will be highlighted whenever human intervention may be required in the conversion process.

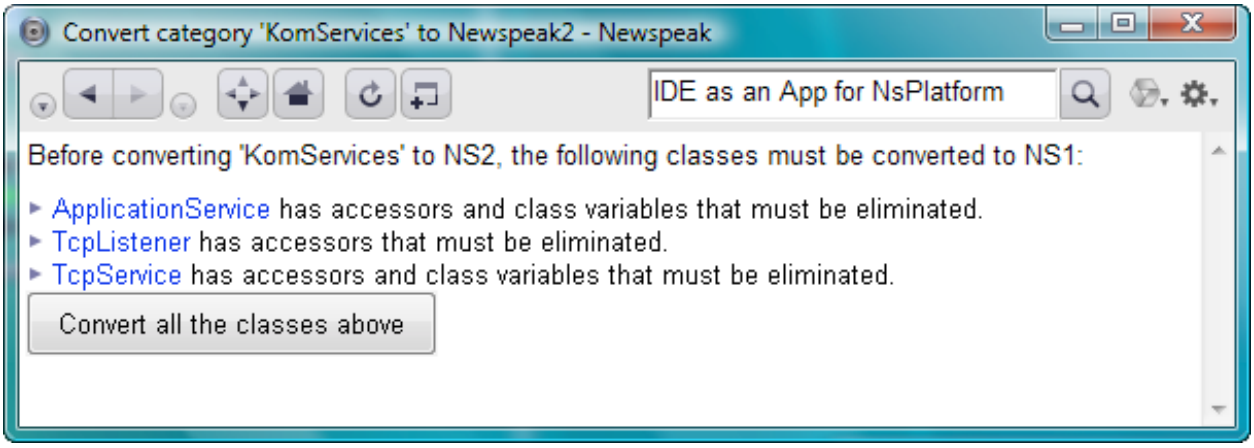
Converting a Class category

Open the category in a Hopscotch browser.

At the top right, you'll see a link named convert. Press it.

It will tell you that your classes need to be converted to NS1 as a first step.

The system should be refined so it can do the entire conversion from Smalltalk in one step.



If your code has accessor methods (that is, methods whose names are the same as the names of instance variables, possibly with a colon at the end), you will be asked to eliminate them as shown above. In other words, **manual** intervention is required. The most common situation is probably simple getters and setters: instance variable i with methods

i

^i

i: v

i := v

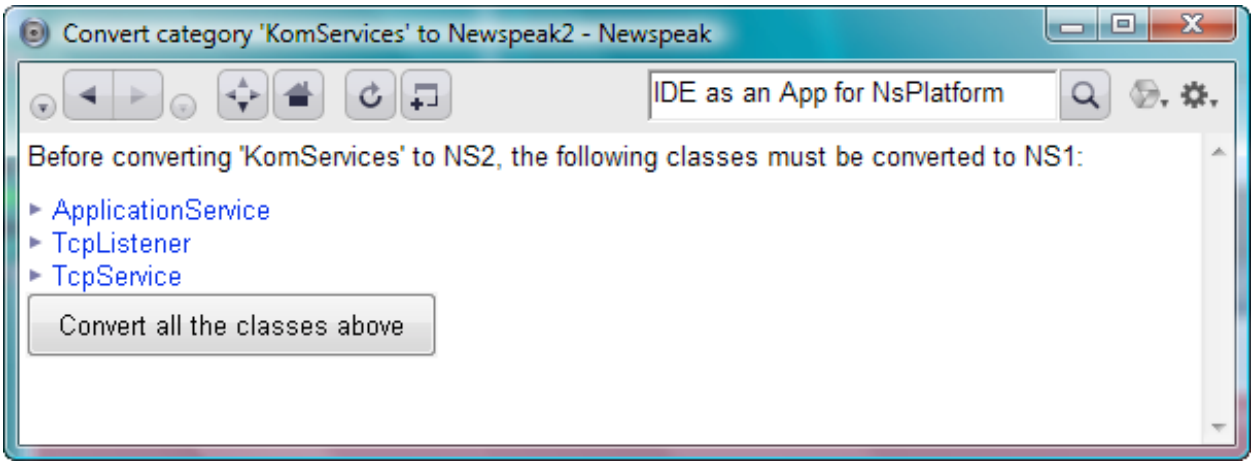
These methods should simply be deleted before the conversion begins. In any other case of accessor methods, we want to rename the instance variable **i** to some unused name such a s **i_0**. The methods will no longer be considered accessors, and conversion can proceed.

The system should be able to cope with accessors automatically, but it doesn't.

If your code defines global or pool variables, their declarations will be lost in the conversion and they will be converted into imports as described below. If your code defines class variables, you will have to eliminate them before proceeding as shown in the above screenshot. When you delete them, the system will warn you if they are actually used. In that case, they will be moved to the ***Undeclared*** variable list. This will effectively convert them into globals; they too will end up as module variables with imports. Make a note of any class variables you eliminate; you will need to deal with their initialization later.

The converter should be able to detect class variables defined by classes within the category and convert them into uninitialized module variables. Likewise, pools used exclusively by classes in ***CategoryToBeConverted****. However, the initialization of such slots would still require* ***manual*** *intervention.*

Once you have dealt with these obstacles, you should be able to move ahead with the conversion to NS1.



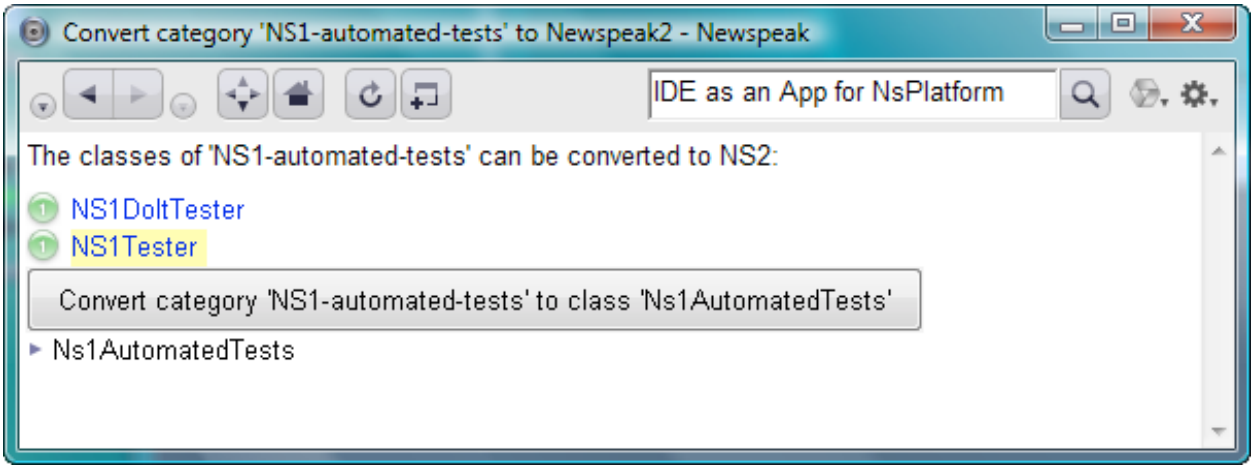
The NS1 conversion will automatically deal with the more common syntactic issues, such as assignments and array literals.

Note, however, that it currently does not deal with multiple assignments, though it could and should. You'll have to deal with those ***manually****, as discussed above.*

The conversion may fail because your code uses primitives or FFI calls, or because you overlooked a multiple assignment (or because it's buggy).

The converter should warn about primitives/FFI calls, but it doesn't.

If the conversion to NS1 has been successful, you're ready to go ahead and convert to NS2.



What the converter will do is produce a top level NS2 class representing the entire category. If your category is named "Category to be converted", the top level class will be named #**CategoryToBeConverted**. Nested inside it will be NS2 versions of each class within the category.

CategoryToBeConverted will have a a factory method named **#usingPlatform:** with **platform** as it argument. This is only a first approximation, and will likely need **manual** adjustment.

The converter will detect all references to global names that are not classes defined in the category, and produce rudimentary imports for them. For example, if you have code that references a class in another category, say **ClassInAnotherCategory**, their will be module variable defined as:

ClassInAnotherCategory = platform ClassInAnotherCategory.

The same holds for any global variables, regardless of whether they are defined by classes in the category or not. Remember also that by now, all class and pool variable declarations have been eliminated, so these are treated as globals as well. A module variable will be created, with an import. If such a variable is defined by your code, you'll need to **manually** change the declaration so that it initializes the module variable in a suitable way.

If your code uses class instance variables, you're on your own. The declarations of these variables will disappear. You need to **manually** track them and figure out how to convert them.

Class instance variables are uncommon, but the converter should probably demand they be removed, or at least issue a warning.

Inherited class methods will often need to be copied down **manually**. Newspeak metaclasses do not inherit from their superclass' metaclass.

The converter could do this for you. However, this is likely to lead to bloat. Many of these inherited methods are constructors for the superclass and not suitable for the subclass. Others belong in the surrounding module declaration, and yet others are simply not used. Heuristics might be useful for deciding the issue.

Class initialization methods will continue to work after the conversion (modulo any reliance on inherited class methods as discussed above). However, one cannot rely on them being called before the class is used. To deal with this, it is best to **manually** eliminate these methods, transferring their actions to the top level instance initializer.

The system could deal with this problem automatically by ensuring that the initializer of any nested class was called from the top level initializer. This is however, undesirable for several reasons:

a. It is bad style. In Newspeak, classes should not have initializers; a class has no state of its own, so there is nothing to initialize.
b. It causes eager creation of nested classes, slowing down module instantiation and possibly wasting space.
c. It is unreliable. If you have a non-standard initializer name, it won't work

Some of these methods should just be migrated to the surrounding class.

If you've gotten your code converted and working using the tools, it still makes sense to attempt to clean up the converted code to make it idiomatic, as suggested above.

Beyond this, you are on your own. Good luck!