Disclosing the Secrets of Visual Smalltalk

By Leandro Caniglia

The original Visual Smalltalk *dialect* by Digitalk *has survived ten years of proclaimed discontinuity. Paradoxically, the languishing commercial interest in this legitimate successor of the Smalltalk lore had a positive consequence: it protected the product from sophistication. After stripping out the over dimensioned Envy-like* Enterprise *development environment, and forgetting the PARTS that nobody seemed to use in real projects, we are left with a simple, powerful, compact and fast Smalltalk implementation. Even though* Visual Smalltalk *does certainly have a prominent history, it is neither perfect nor finished. As it is usually the case with every corpus of fruitful intellectual work, there are still many interesting details waiting to be completed. To begin with, there is one crucial problem in* Visual Smalltalk*: it has a bunch of hidden classes and methods that nobody has disclosed so far. This document is an attempt to repair that historical mistake. It explains how to recuperate all hidden classes and how to retrieve all missing Smalltalk source code.*

Bytecodes and mnemonics

It is broadly accepted that Smalltalk is much more than a programming language. However, when seen as a programming language Smalltalk shares many of the nice properties of the low level ones. More precisely, the transformation that maps the source code of any method to its compiled form can be easily inverted. In fact most Smalltalk dialects provide compiler and decompiler tools as Smalltalk objects programmed in themselves.

Squeak programmers are already familiar with the mnemonic representation of compiled methods. The system has a *show bytecodes* command that prints any compiled method in a human readable format. While less documented, *Visual Smalltalk* also provides such a command. The expression

```
CompiledMethod byteCodeReaderClass displayMethod: aCompiledMethod on: aStream
```

prints out a mnemonic representation of aCompiledMethod on aStream. Let's see an example from the *SUnit* framework. The evaluation of the expression above on the TestCase>>#assert: method outputs the following piece of code:

```
TestCase>>#assert:
        type: 0
        blocks: 0
        args: 1
        stk temps: 0
        literals:
                'Assertion failed'
                signalFailure:
  1  <50>       LoadArgument1
  2  <1B>       TestJumpTrue   8
  5  <A3>       PushLiteral1
```

```
6  <0A>       LoadSelf
7  <E2>       SendSelector1    #signalFailure:
8  <49>       ReturnSelf
```

The code above is a textual representation of the compiled method TestCase>>#assert:. The header accounts for additional information as the number of temporaries and the contents of the literal frame. Below there is a mnemonic representation of the bytecodes that make up the compiled method.

The stack-based interpreter of the virtual-machine code follows some few simple rules. For instance, the fragment

```
5  <A3>       PushLiteral1
6  <0A>       LoadSelf
7  <E2>       SendSelector1    #signalFailure:
```

stands for the Smalltalk message:

```
self signalFailure: 'Assertion failed'
```

As this fragment reveals, the bytecode interpreter expects the receiver of the message in the topmost position of the stack. All arguments, if any, are also searched in the stack, from the receiver and down.

Code optimization is also easily revealed. Branching messages like ifTrue: and its variants are not compiled as message sends but *inlined* into the bytecodes. Inlining avoids the use of superfluous syntactic blocks; it also works around the message send mechanism for the sake of performance. For example, the fragment

```
1  <50>       LoadArgument1
2  <1B>       TestJumpTrue   8
```

stands for the Smalltalk expression:

```
<argument> ifFalse:
```

Note that the names of the formal arguments are not stored in the compiled method. Similarly, temporary names are also forgotten and can only be found in the Smalltalk source code.

Now, if after this short and straightforward analysis we put all together, we obtain:

```
TestCase>>assert: argument
    argument ifFalse: [self signalFailure: 'Assertion failed'].
    ^self
```

Eliminating the superfluous return of the receiver:

```
TestCase>>assert: argument
    argument ifFalse: [self signalFailure: 'Assertion failed'].
```

Finally, an elementary *type inference* analysis reveals that argument must be kind of Boolean (search for implementors of ifFalse:). Consequently we can rewrite the source code above as:

```
TestCase>>assert: aBoolean
    aBoolean ifFalse: [self signalFailure: 'Assertion failed']
```

This simple example shows two things. First it shows how any compiled method can be printed out in human readable form regardless the availability of its Smalltalk source code. Second, it shows how simple and straightforward is the translation of the mnemonic language to the Smalltalk language. The only non-trivial part of the overall process of rewriting the Smalltalk source code is that of type inference. It is by inferring types that we can name variables meaningfully. Otherwise, the expressiveness of aBoolean would be lost behind the rather impersonal argument.

Hidden Globals

Assume you have a global variable; say a *Class* or a *Pool Dictionary*. If you compile a method that uses that global and afterwards you remove that global from the Smalltalk dictionary, then the compiled method will still work the same as before

```
Smalltalk at: #Foo put: 'foo'.

Object>>#foo
    ^Foo

testFoo
    self assert: Object new foo = 'foo'.
    Smalltalk removeKey: #Foo.
    self assert: Object new foo = 'foo'
```

The compiled method remains the same

```
Object>>foo
        type: 0
        blocks: 0
        args: 0
        stk temps: 0
        literals:
                Foo ==> 'foo'
  1  <02>       NoFrameProlog
  2  <5A>       LoadAssoc1
  3  <48>       Return
```

What changes after removing the global is that if we now try to recompile the original source code, the compilation will fail because this time there is no known binding for the name 'Foo'. In other words, before recovering the source code from the compiled method, we must define the missing global. Note that we don't need to figure out its value, 'foo' in the example, as the global's value is saved in the literal frame of any compiled method that uses the global.

For instance, if the missing global is a complex *Pool Dictionary*, then there is no need to rebuild its many keys and values. That very same dictionary is stored in the literal frame of the compiled method. We are done by including the name of the global in the Smalltalk dictionary.

In *Visual Smalltalk* there is just one hidden *Pool Dictionary*. Actually there are two of them, but just one is used and the other one is ignored. To find them out we can go a step further with the interesting expression we used to print compiled methods in human readable format. As we have seen above, the object that displays those pretty assembly-like scripts is

```
CompiledMethod byteCodeReaderClass.
```

By inspecting that expression we discover some more interesting things. First, as the selector suggest, the expression retrieves a class. Second, the class name is blank. Third, the class definition refers to two *Pool Dictionaries* XCByteCodes and XCMaxLast that are not declared as globals. Fourth, even when the superclass is Object, the class is not included in the subclasses instance variable of the Object class.

There are many useful things we could do here. We can recover the first *Pool Dictionary* from the literal frame of any of the compiled method that uses it. For instance, the method shortSendFor:on: stores XCByteCodes in the first slot of its literal frame. Thus, we can define the global with the following script

```
| class method |
method := class compiledMethodAt: #shortSendFor:on:.
Smalltalk at: #XCByteCodes put: (method at: 1) value
```

We can now include the hidden class into the Smalltalk dictionary. From the message we have been taking advantage of we can deduce even one more thing. An appropriate name for the hidden class would be ByteCodeReader. Had we considered 'bytecode' as a noun and not as a composite word, we would have named the class BytecodeReader instead. However, at this stage we are not trying to improve or change the original programming style, but to recover it from whatever is available. Think of yourself as an archeologist.

To name the class we can proceed as follows

```
| class |
class := CompiledMethod byteCodeReaderClass.
class instVarAt: 4 put: #ByteCodeReader.
Smalltalk at: class symbol put: class
```

To complete this step we need to link the class in the hierarchy. As we observed above, its superclass does not include the class in its subclasses array. Hence

```
| class |
class := ByteCodeReader.
class superclass subclasses: (class superclass subclasses copyWith: class)
```

Note that there is no need to repeat the code above at the MetaClass level as the implementation of MetaClass>>#subclasses: is reportedly obsolete.

Constant Pool Names

At this point ByteCodeReader behaves as any regular class. You can now browse it and start recovering the source code of its methods as explained above. As we will see, this is just the beginning and there are quite a few other hidden classes that require a similar treatment.

Sooner or later, however, you will discover that some few methods cannot be recovered exactly. In other words, for some few exceptional cases, no source code will reproduce the original compiled method. The reason is that in *Visual Smalltalk* bindings to pool dictionary entries can be compiled in two different ways. If the *Pool Dictionary* is declared as 'constant,' then the compiler does not store the associations used in the method, but the values.

To see how this works let's take a look at the following method

```
CompilerInterface>>#bindingClassForPoolNamed: aString
    "return the class of binding used to represent elements
    of the pool named by the argument"
    |cp|
    cp := Smalltalk at: #ConstantPoolNames ifAbsent: [IdentityDictionary new].
    (cp includes: aString asSymbol)
        ifTrue: [^self compiler constantAssociationBindingClass]
        ifFalse: [^self compiler associationBindingClass]
```

Depending on the membership to the ConstantPoolNames dictionary, a different kind of *binding* is assigned to the pool's associations. From our reconstruction it turns out that the XCByteCodes pool dictionary had been declared as 'constant' when the missing source code was originally compiled. That declaration was subsequently revoked, but the methods compiled that way still have the original bytecodes.

In consequence, in order to recover the original source code we need to include that pool name into the ConstantPoolNames global. Here is how

```
    Smalltalk
        at: #ConstantPoolNames
        put: (ConstantPoolNames copyWith: #XCByteCodes)
```

Other hidden classes

The discussion about constant pools has other nice consequences. It shows us the way to reach the compiler plus two binding classes. These objects can be examined by inspecting the following expressions

```
| interface compiler constant assoc |
interface := CompilerInterface new.
compiler := interface compiler.
constant := compiler constantAssociationBindingClass.
assoc := compiler associationBindingClass
```

Once again, we get hidden classes. They all have blank names, they are not declared as globals and they are not included in the subclasses array of their superclasses. The compiler class is also referenced from the method

```
CompilerInterface>>#defaultCompilerClass
```

Consequently, we can use all this information to name the hidden classes as: DefaultCompiler, ConstantAssociationBinding and AssociationBinding. This time, however, we will take the more systematic approach described below.

Disclosing all hidden classes

At this point one starts wondering how many hidden classes are there in the image. There is a simple way to find that out

```
| hidden |
hidden := (MetaClass allInstances select: [:m | m name first = $ ])
    collect: [:m | m instanceClass]
```

The answer is that there are 53 hidden classes in *Visual Smalltalk*. So far, we have revealed four of them: ByteCodeReader, DefaultCompiler, ConstantAssociationBinding and AssociationBinding. Just 4 out of 53 might be a bit disappointing. However, the information we now have is more fruitful than that. For instance, among the hidden classes it should be one *Byte Code Writer* or Encoder. Also, the DefaultCompiler is not a subclass of Object but a subclass of an abstract class that we could rename as SmalltalkCompiler (we have chosen this name and not Compiler because there is already a global named that way in the image and we must avoid colliding with any existing name). Similarly, the superclass of AssociationBinding is an abstract class that could be renamed to Binding. Other kinds of bindings should correspond to other of its subclasses, as InstanceVarBinding, TemporaryBinding, and the like.

If there is a Compiler, it should be a Parser. If there is a Parser it should be a Scanner. Also, other hidden classes should correspond to all possible parse nodes. Given the number of different kinds of parse nodes and the variety of bindings used in Smalltalk, there should be no much classes left in the hidden collection.

Now that we have an approximate idea of the classes we are looking for we can take the more systematic approach claimed above. First let's observe that all hidden classes have blank names. Hence, we must distinguish them from the number of blank spaces that make up their names. As we discover their original names we can replace the blanks with meaningful strings. We can use the collection of hidden classes computed before to find out the minimum and maximum number of spaces a blank class name can have.

```
| hidden min max |
hidden := (MetaClass allInstances select: [:m | m name first = $ ])
    collect: [:m | m instanceClass].
min := hidden first symbol size.
min := hidden inject: min into: [:r :class | r min: class symbol size].
max := hidden inject: 0 into: [:r :class | r max: class symbol size].
^min to: max
```

The evaluation of the script shows that blank class names can have between 1 and 56 spaces. Since there are 53 hidden classes we deduce that, except in 3 cases, all blank symbols made up of 1 to 56 spaces correspond to the 53 hidden classes.

We can now create the NameDiscloser class as follows:

```
Object subclass: #NameDiscloser
  instanceVariableNames:
    ' hidden '
  classVariableNames: ''
  poolDictionaries: ''
```

In the class side the #nameArray method answers the mapping between the number of white spaces of the hidden class and the original name of the class. We initialize that mapping with blank symbols of the appropriate lengths. At index i we put the symbol

```
((String new: i) atAllPut: $ ; yourself) asSymbol.
```

Afterwards, as our investigation goes on and we happen to discover more class names, we can modify the entries of the array accordingly.

In the instance side of the class we provide four main services for the sake of convenience.

```
NameDiscloser>>#initialize
    hidden := (MetaClass allInstances select: [:m | m name first = $ ])
        collect: [:m | m instanceClass]
```

This method initializes the hidden instance variable as shown above.

```
NameDiscloser>>#nameHidden
    | names n name |
    names := self class nameArray.
    hidden do: [:class |
        n := class name size.
        name := names at: n.
        Smalltalk privateRemoveKey: class symbol ifAbsent: [].
        class instVarAt: 4 put: name asSymbol.
        Smalltalk at: class symbol put: class]
```

The method above uses the symbols in the nameArray to rename the hidden classes. Note that our low level implementation breaks the encapsulation of the classes being renamed and that of the SystemDictionary. The removal of the association at the old class symbol avoids the side effect of having two globals pointing to the same class. That allows us to send the nameHidden messages as many times as required, forgetting at the same time obsolete global names.

As discussed before, we can also declare #XCByteCodes as a global name, and include it in the ConstantPoolNames array.

```
NameDiscloser>>#markConstantXCByteCodes
    | class method |
    class := CompiledMethod byteCodeReaderClass.
    method := class compiledMethodAt: #shortSendFor:on:.
    Smalltalk at: #XCByteCodes ifAbsentPut: [(method at: 1) value].
    (ConstantPoolNames includes: #XCByteCodes) ifFalse: [
        Smalltalk
            at: #ConstantPoolNames
            put: (ConstantPoolNames copyWith: #XCByteCodes)]
```

Finally we fix the hierarchy of classes by adding the hidden classes to the subclasses list of their superclasses:

```
NameDiscloser>>#linkClassHierarchy
    | sbclasses orphans1 orphans2 |
    hidden do: [:cls |
        sbclasses := hidden select: [:c | c superclass == cls].
        cls subclasses isNil ifTrue: [cls subclasses: sbclasses]].
    orphans1 := hidden
        select: [:cls | cls superclass == Object
            and: [(Object subclasses includes: cls) not]].
    Object subclasses: Object subclasses , orphans1.
    orphans2 := hidden
        select: [:cls | (cls superclass subclasses includes: cls) not].
    orphans2
        do: [:cls | cls superclass
            subclasses: (cls superclass subclasses copyWith: cls)]
```

At this point one would expect to see a bunch of blank named classes in the class hierarchy browser. However, there is still one more switch that prevents that from happening. The method

```
Behavior>>#withAllSubclasses
        "Answer an OrderedCollection of the receiver and
        all of its subclasses in hierarchical order."
    ^self withAllSubclasses: false
```

filters out all classes whose names start with a white space. Hence, it must be changed to:

```
Behavior>>#withAllSubclasses
        "Answer an OrderedCollection of the receiver and
        all of its subclasses in hierarchical order."
    ^self withAllSubclasses: true
```

otherwise classes with blank names will not be included in browsers and will not be searched for class references, senders or implementors which are crucial for the type inference we are interested in.

Recovering more hidden class names

As we have seen above, many of the blank named classes can be meaningfully renamed based on the messages that retrieve them. Some others however, have to be renamed after a behavioral analysis. It is always a good idea to resolve first the simplest classes. Now that senders and implementors do work, we could write a simple script to recover all selectors of all hidden classes that end with the string 'Class'. If we include that as a service in our NameDiscloser class we could program it as

```
NameDiscloser>>#classRevealingMethods
    answer := OrderedCollection new.
    hidden do: [:class |
        class selectors do: [:each |
            ('*Class' sunitMatch: each)
                ifTrue: [answer add: (class compiledMethod at: each)]].
    ^answer
```

This technique allows deducing 25 original names from the hidden classes. As showed before, the class name is the capitalization of the main part of the selector. After obtaining that information, we can improve our NameDiscloser>>#nameArray method by replacing the 25 blank names with the original ones.

As we find out more original names, we rerun the nameHidden method of the NameDiscloser. Iterating this way, the intention of the remaining hidden classes is increasingly made more apparent. Finally we end up with the complete mapping of names.

Recovering the Smalltalk Source Code

Once all hidden classes are renamed, the work that remains is the translation of the source code from the mnemonic form to the Smalltalk language. A simple inspection shows how many methods we are talking about:

```
CompiledMethod allInstances
    inject: 0
    into: [:r :cm | r + (cm source isNil ifTrue: [1] ifFalse: [0])]
```

A better estimation should take into account only current versions. Actually the result depends on the particular image were it is evaluated. In general, however, the number will be something about 1720 methods. Even though there is a considerable amount of work to do, it is achievable as a pastime. One could think of two strategies. The first one is to write a decompiler. The second is to translate each of the methods one by one. The problem with the first approach is that the decompiler would depend on code that is in mnemonic format. The other drawback is that automatically decompiled code uses generic arguments and temporary names of the form arg1, arg2, t1, t2, etc.; which are not in the spirit of Smalltalk. There is finally a legal issue. The use of a decompiler tool could be interpreted as a *reverse engineering* practice. Unlike reverse engineering, *manual translation* from one human readable language into another one, enhanced with meaningful names derived from labor-intensive type inference cannot be qualified neither as *reverse* (it is a simple translation) nor as *engineering* (it is the work of an artisan); mostly, when both languages are provided in the base code of a discontinued product whose license has been legitimately acquired.

The approach followed in the present work was the *labor-intensive* one. This was possible because most translations were straightforward, and just a very small quantity out of the 1700 or so methods was a bit tedious to work with.

SUnit Tests

After recompiling the new methods, the old ones are not referenced anymore. However, since the old compiled methods belong in a Smalltalk link library, they are not garbage collected.

Having unreferenced methods in the image is indeed a side effect that we should overcome. However, before getting rid of the duplications we can take advantage of them for testing purposes.

For each new method, we can recover its original counterpart and then check that they are equivalent. The only differences between the two versions should be in the names of the globals. If an original method referenced a hidden class or pool dictionary, then there will be a difference in the literal frame with the new method. In fact the difference should be only in the key part of the literal association, as the key holds the name. Note that no differences should be observed in the byteCodeArray instance variables.

Original compiled methods can be recovered with

```
| originalMethods |
originalMethods := CompiledMethod allInstances select: [:cm | cm source isNil]
```

The main fragment of the test is in the following method

```
assertOn: aCompiledMethod
    | current |
    current := self currentVersionFor: aCompiledMethod.
    self assert: current source notNil.
    self assert: current byteCodeArray = aCompiledMethod byteCodeArray
```

Note that the current version for a compiled method is

```
    aCompiledMethod classField compiledMethodAt: aCompiledMethod selector
```

As stated above, we could also compare the literal frames provided we take into account the renaming of the blank global names.

```
    self assert: current literals size = aCompiledMethod literals size.
    current literals with: aCompiledMethod literals do: [:a :b |
        self assert: a class = b class.
        a isAssociation
            ifTrue: [self assert: a value = b value]
            ifFalse: [self assert: a = b]].
```

In doing this, we discovered that ByteCodeReader>>#sendPseudoFor:on: has a typo. The author of the original method wrote botAnd: instead of bitAnd:.

Besides the correctness of the new Smalltalk sources we can test the correctness of the classes whose names have been recovered.

```
testNoBlankNames
    allClasses do: [:class | self assert: class name trimBlanks size > 0]
```

The variable allClasses is initialized as:

```
    allClasses := MetaClass allInstances collect: [:meta | meta instanceClass]
```

We can also test the hierarchy links

```
testClassLinks
    allClasses
        do: [:class | class superclass isNil
            or: [class superclass subclasses includes: class]]
```

Auto Mark Dirty

Database applications in general and *GemStone* applications in particular usually require the markDirty message to be sent whenever an instance variable of a persistent object is assigned in the client image. The automatic inclusion of the markDirty message is a feature that can be also enabled in *Visual Smalltalk*. This undocumented option is made apparent when translating the source code of AssignmentNode>>#compute or AssignmentNode>>#evaluate. The activation of the automatic mark dirty option is achieved by sending the message

```
    addOption: #sendDirty with: <markDirty selector>
```

to the CompilerInterface. Both methods are similar; here is one of them:

```
AssignmentNode>>#evaluate
    | dirty |
    self setIcStart.
    dirty := false.
    expression evaluate.
    assignees do: [:v |
        dirty ifFalse: [dirty := v isInstanceVariable].
        v storeRetain].
    (dirty and: [self compiler environment includesOption: #sendDirty]) ifTrue: [
        self encoder pushResult.
        self encoder loadSelf.
        self encoder send: (self compiler environment optionValue: #sendDirty).
        self encoder popR].
    self setIcEnd
```

As shown above, the #sendDirty option automatically inserts in the compiled method the bytecodes that send the <markDirty selector> to self whenever any of its instance variables is assigned. Note also that the inserted message is sent with the object being assigned as the argument of the <markDirty selector>.

Rebuilding the Development Library

Once the tests have been written and run there is no reason to have duplicated compiled methods in the image. After recompiling the methods from their recovered Smalltalk code we should save the libraries where the original compiled methods belong in.

One of those libraries, unfortunately, cannot be rebuilt as easily as a regular one. As it turns out, the *development* library is the one that contains the 53 hidden classes. However, in order to rebuild that library we must first retrieve all its information. It is precisely here where we encounter the first obstacle. The expression

```
SmalltalkLibraryReporter for: SmalltalkLibraryBinder devLibName
```

signals a DNU error because the information available for the library is not complete.

Fortunately, now that we have all the Smalltalk sources it is easier to find another way to retrieve the information required to rebuild the library. After taking a look at SmalltalkLibraryBinder>>#bind, it becomes apparent that before rejecting binding the development library a second time, the binder reads from the file all the information we are looking for.

```
NameDiscloser class>>#devLibInfo
    | fname binder metaMeta space info |
    fname := SmalltalkLibraryBinder devLibName.
    binder := SmalltalkLibraryBinder currentClass new.
    binder skipPostBindCleanUp: true.
    binder file: fname.
    [
        binder open.
        binder checkVersion.
        metaMeta := binder getMetaMeta.
        space := binder readSpace.
        space := binder addObjects: space externals: metaMeta.
        space associationsDo: [:assoc | binder objectStore metaInfo add: assoc].
        binder resolveExternals.
        binder getObjects]
        ensure: [binder close].
    ^binder
```

Once the information of the development library is captured, the binder is closed without completing the binding.

The method above enables us to rebuild the library. Given that this time we are providing the complete Smalltalk sources and non-blank names for the hidden classes, the new library will have the .sml file we want. There is, however, one more obstacle to deal with. The information retrieved from the original file does not include the hidden classes. In consequence we must add the 53 hidden classes by hand, before building the new version of the library.

The remaining source code belongs principally to two additional libraries: VOSW31 and VSLB31. Since these two libraries can be easily rebuilt, after including all other recovered methods in the appropriate library, we are done.

Conclusions

The caducity of *Visual Smalltalk* was planned almost ten years ago. However, because living objects have an innate tendency to survive and change, Smalltalk systems keep subsisting and evolving as long as some smalltalker holds at least one *non weak* reference to them. *Squeak*, for instance, germinated inside an unreleased implementation of *Smalltalk-80* for the *Mac* which had been abandoned fifteen years before.

It is our hope that this work contributes to the rescue of a brilliant offspring of the Smalltalk wisdom. The reader should note that the inherent simplicity of its design made it feasible disclosing the veiled parts of *Visual Smalltalk*. Sophistication rather than simplicity is what condemns systems to their ruin; the future should not lead us beyond the limits of personal understanding.