Author Todor Todorov

Who's smalltalking to me?

Tracing Message Sends to an Object

Introduction

Many Smalltalk systems have existed for years, and in the case of VSE, the youngest systems are probably at least 10 years old. Code evolves over the years and can become complex and interconnected. Documentation gets lost or out-dated and out of sync. People that had the knowledge about particular part of a system may not be working for the company anymore. The system however is still running, but once in a while may need a change or two – and the trouble starts.

I had the task to implement some changes in a VSE system designed by others; Complex and interconnected system with many components, and documentation only at the reference level. I needed to understand how a certain component works, but where do I start? The usual method of taking senders of every single method would kill me. And even if I survived, it would still not give me a picture of the lifetime of the object I was interested in. Obviously, to understand more about the life and purpose of the object, I needed to know, *who is talking to that object, when and why*.

The solution I chose was to implement a mechanism to trace message sends to the object in interest.

This document describes the workings of a message send and a mechanism for tracing message sends to an object.

Contents

Introduction 1
Messages and Message Sends 3
  Definitions 3
  VSE Objects 3
  Object’s Class 4
  Sending Messages 5
  Super Sends 5
  Instance Specific Behaviour 9
Generic Message Send Tracing 12
  Message Send Tracing 12
  Instance Specific Tracing 12
  Hacking Instance Specific Super Sends 13
  Trace Methods 14
  Activating Tracing 16
  Internals 17
  Known Limitations 17
Conclusion 18
  Mixins 18
  Break Points 19
  Proxy Objects / Generic Tracing 19

Messages and Message Sends

This chapter is a general description of the Smalltalk and VSE concepts of object, message and message send.

Definitions

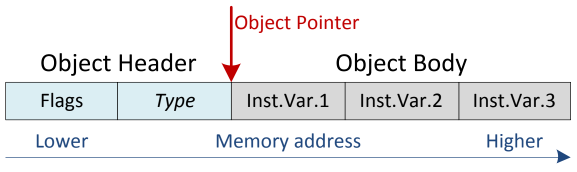
I don't need to write much here, people already know what they are, but anyway.

- ***Object***: language mechanism for binding data with methods that operate on that data.
- ***Method***: concrete implementation of some logic. VSE uses instances of the class **CompiledMethod** to store this logic.
- ***Message***: an object, method name (selector) and optionally, arguments passed to the method together with the object.
- ***Message Send***: an invocation (perform) of a message.

VSE Objects

This section contains the description of a VSE object. Other dialects may have chosen another approach to implement an object.

An object is a data structure in the computer memory, which the virtual machine (VM) treats in a special way. Typical VSE object could have the following memory representation:



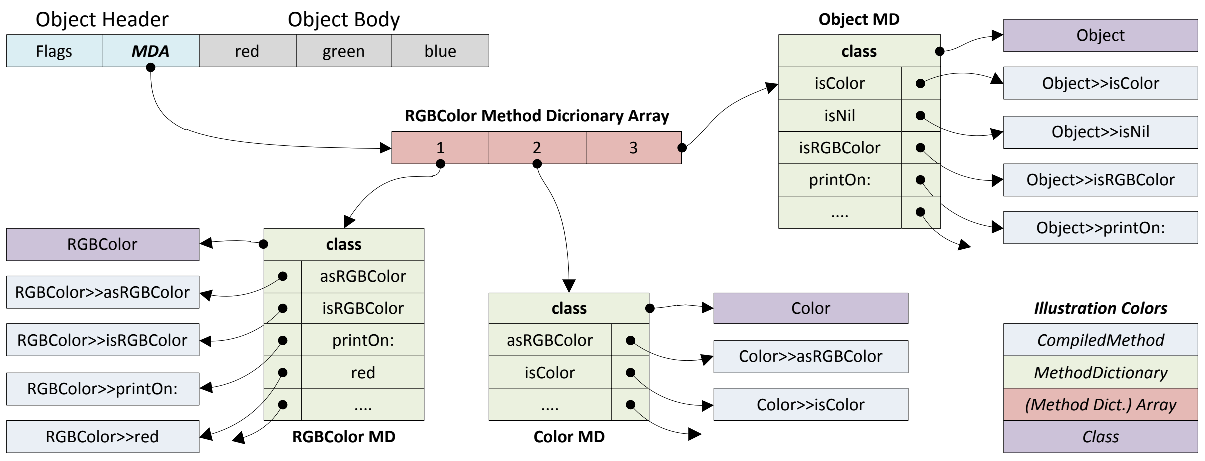
The object consists of two portions, the ***object header*** and the ***object body***. The object header is only visible to the VM, and invisible to Smalltalk. There are however ways to manipulate it. It may be compact, as shown above or extended, for large objects. The flags field contains the size of the object, bit-flags about the type of the object and the hash of the object. The header also contains information about the type of the object, i.e. about the object class and methods available on this object; more on this in a moment. The object body is optional, and for objects that contain information, it has references to other objects (the illustration has 3 references) or for binary objects, it directly contains the data.

The most obvious would be for the object to have reference (stored in the header) to the class object it is an instance of. The class has a list of methods available to that object. For performance, this is not the case. Most often, the object behaviour is more interesting than its class. In other words, the list of methods available for an object is more important. This is because every message sends needs to resolve to actual method implementations. The illustration on the next page shows the memory layout for an **RGBColor** object. The key elements are:

- ***Method dictionary array***. This array describes the behaviour of the object. It is referenced directly from the object's header. It is not a special object, but just a regular **Array** object. It contains **MethodDictionary** objects. The size of the method dictionary array corresponds to the depth of the object's class hierarchy, i.e. 3 levels for **RGBColor**. The method dictionary array is shared between objects of a same class.

- ***Method dictionary***. Method dictionaries are instances of **MethodDictionary**. Those are identity dictionaries, accepting only symbols as keys and compiled methods as values. Except this, they also have a special instance variable named **class**. This is instrumental for finding the class of an object.
- ***Compiled methods***. Those contain the logic.
- ***Class***. Describes the object, but also holds a reference to the method dictionary array, which is used when new objects are instantiated. The methods implemented by the class, as we see them in the development tools are hold in the **first** method dictionary in the method dictionary array.

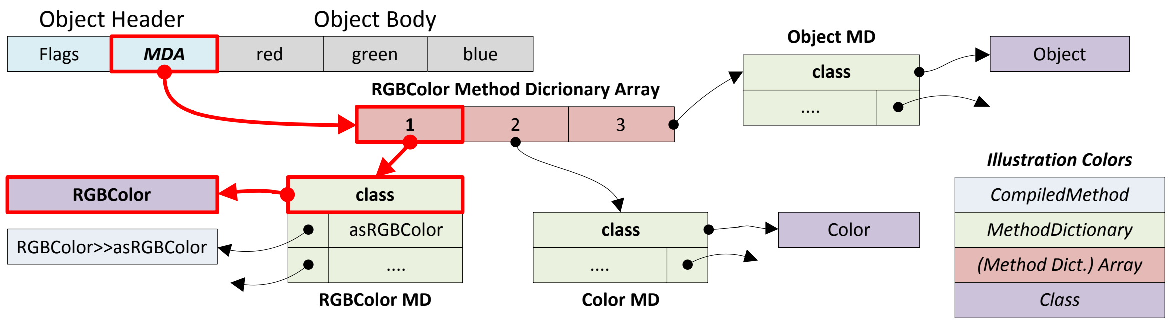
Below is a diagram of a VSE **RGBColor** object and the objects referenced by it. The diagram is slightly simplified for clarity.



More about the details of a VSE object's memory layout can be found in `\SAMPLE\USERPRIM\` examples in the VSE installation directory.

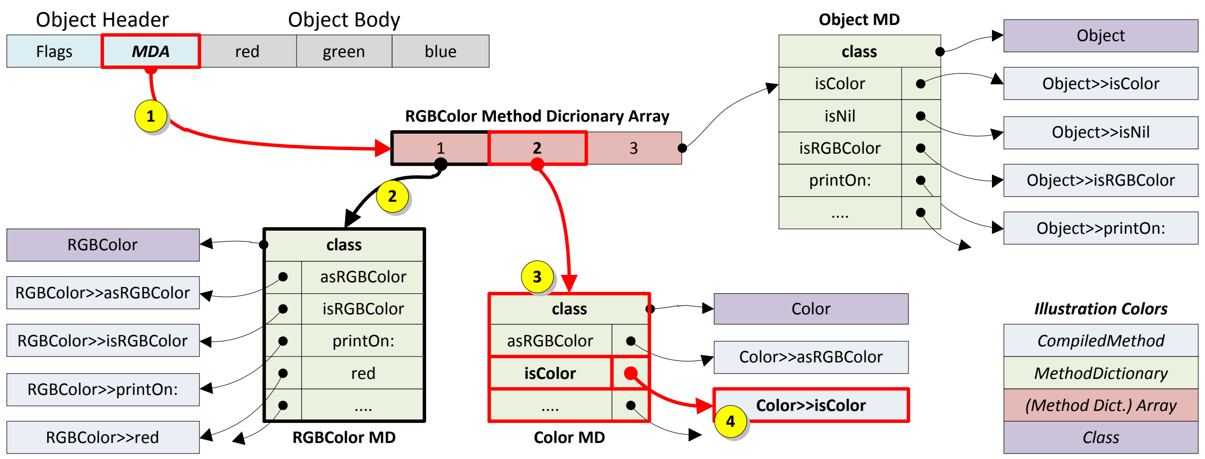
Object's Class

Sends to `#class`, for example `self class` end in a primitive in the VM. Since the object has no direct reference to its class, the VM has to find the class trough the method dictionary array and the **first** method dictionary in the array. The illustration below shows how the class for an object is retrieved.



Sending Messages

When a message is sent, the message only contains the selector for the method to be executed. The VM needs to find the compiled method that implements the logic for the requested selector for the given receiver. Let's say we have an **RGBColor**, and we send it `#isColor`. The illustration below shows the search process.



1. Use the **RGBColor** object's header to find the method dictionary array for the **RGBColor** object.
2. Look in the first method dictionary in the array. This dictionary contains all the methods defined directly on the **RGBColor** class. In our example, no implementation is found for the `#isColor` method in this dictionary.
3. Look in the second dictionary in the array. This dictionary contains all the methods defined directly on the **Color** class. In the example, we find `#isColor` method in this dictionary.
4. Get the compiled method object for `#isColor` defined on the **Color** class.

Once found, it can be executed with the **RGBColor** object as `self`. Also, the compiled method is cached in an inline cache, so subsequent message sends will not have to look for it.

If the VM can't find implementation for the `#isColor` method in **any** of the method dictionaries in the method dictionary array, it creates a Message object with `self` as the receiver, `#isColor` as the selector, and in this case no arguments. It then performs the same lookup for the `#doesNotUnderstand:` method and executes the method with the message object as parameter. If it can't find the `#doesNotUnderstand:` method in any of the method dictionaries, the VM displays a message box to the user and crashes.

The `#vmInterrupt:` method is another key method that must be present. The VM crashes in similar way, if it is missing.

Super Sends

Slightly more complicated is when a method is sent to `super` instead of `self`. Let's look at an example. We'll define 3 subclasses of **Object**. Some "pseudo code" below:

```
Object subclass: #Subclass1.
Subclass1 subclass: #Subclass2.
Subclass2 subclass: #Subclass3.
```

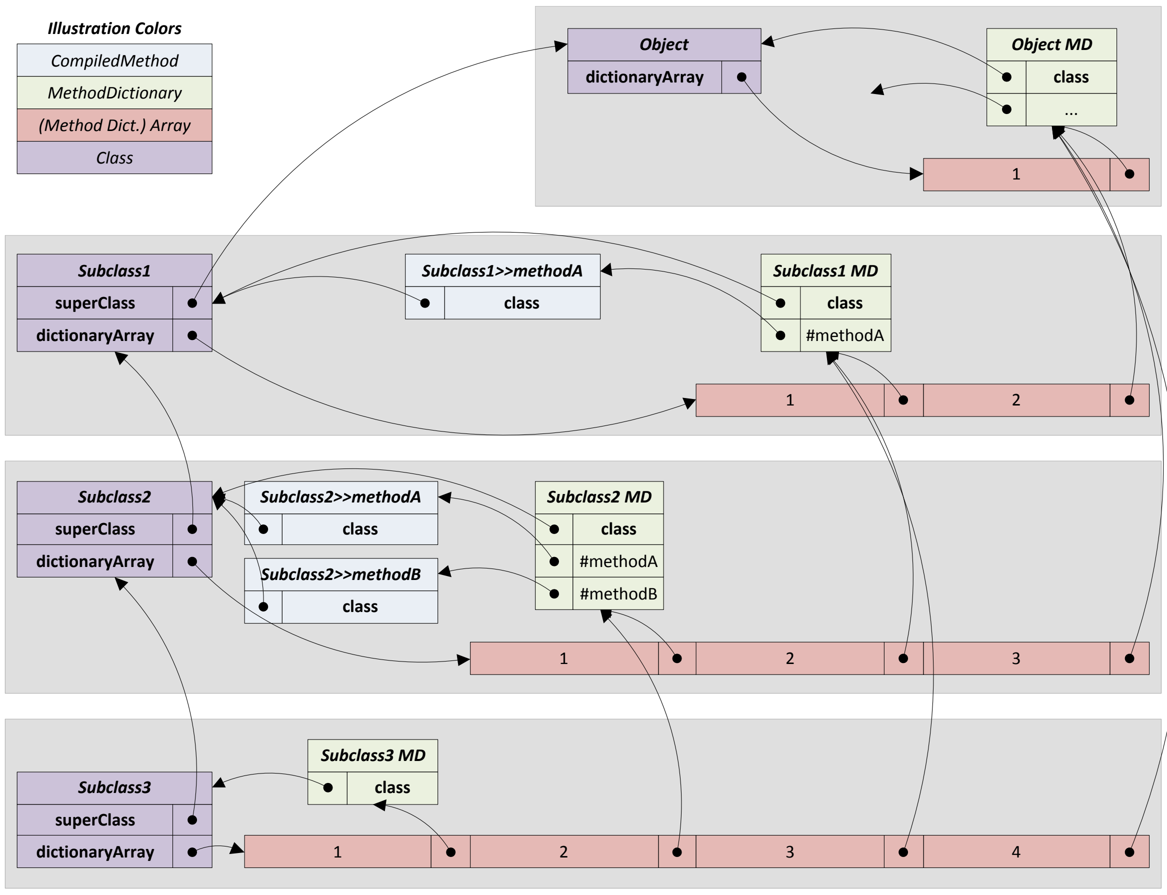
Let’s add some methods to the classes:

```
Subclass1>>methodA
      Transcript show: 'Subclass 1, Method A'; cr.

Subclass2>>methodA
      Transcript show: 'Subclass 2, Method A'; cr.

Subclass2>>methodB
      Transcript show: 'Subclass 2, Method B'; cr.
      super methodA.
```

Now, we have the following class hierarchy:

Object	*[many methods]*
Subclass1	`#methodA`
Subclass2	`#methodA`, `#methodB`
Subclass3	*[no method]*

Below is a simplified illustration of the object graph for the classes, compiled methods, method dictionaries and method dictionary array that are of particular interest.



Let’s look at an example.

```
Subclass3 new methodB.
```

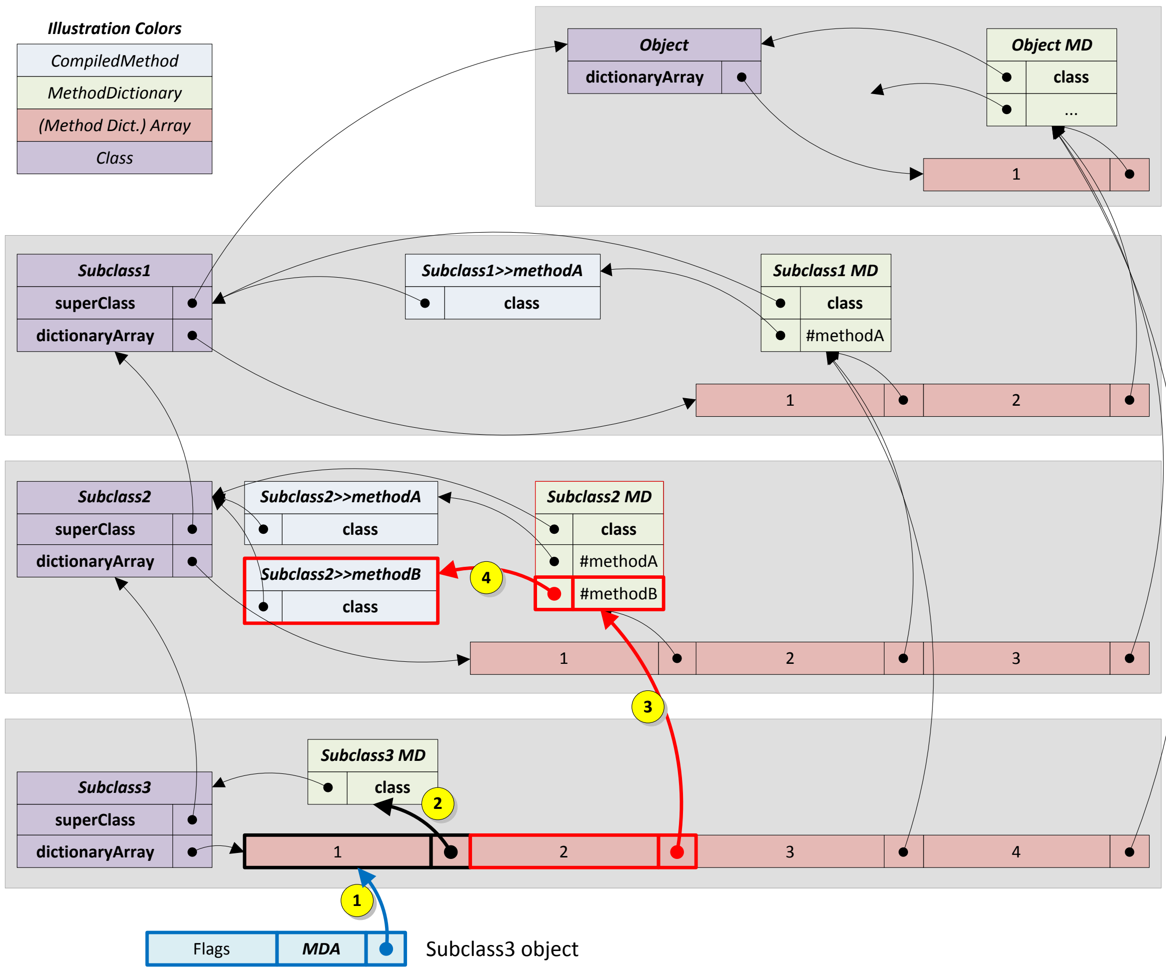
Executing the code above will give the following class stack:

```
Subclass3(Subclass1)>>methodA
Subclass3(Subclass2)>>methodB
UndefinedObject>>Doit
```

Transcript will show:

Subclass 2, Method B
Subclass 1, Method A

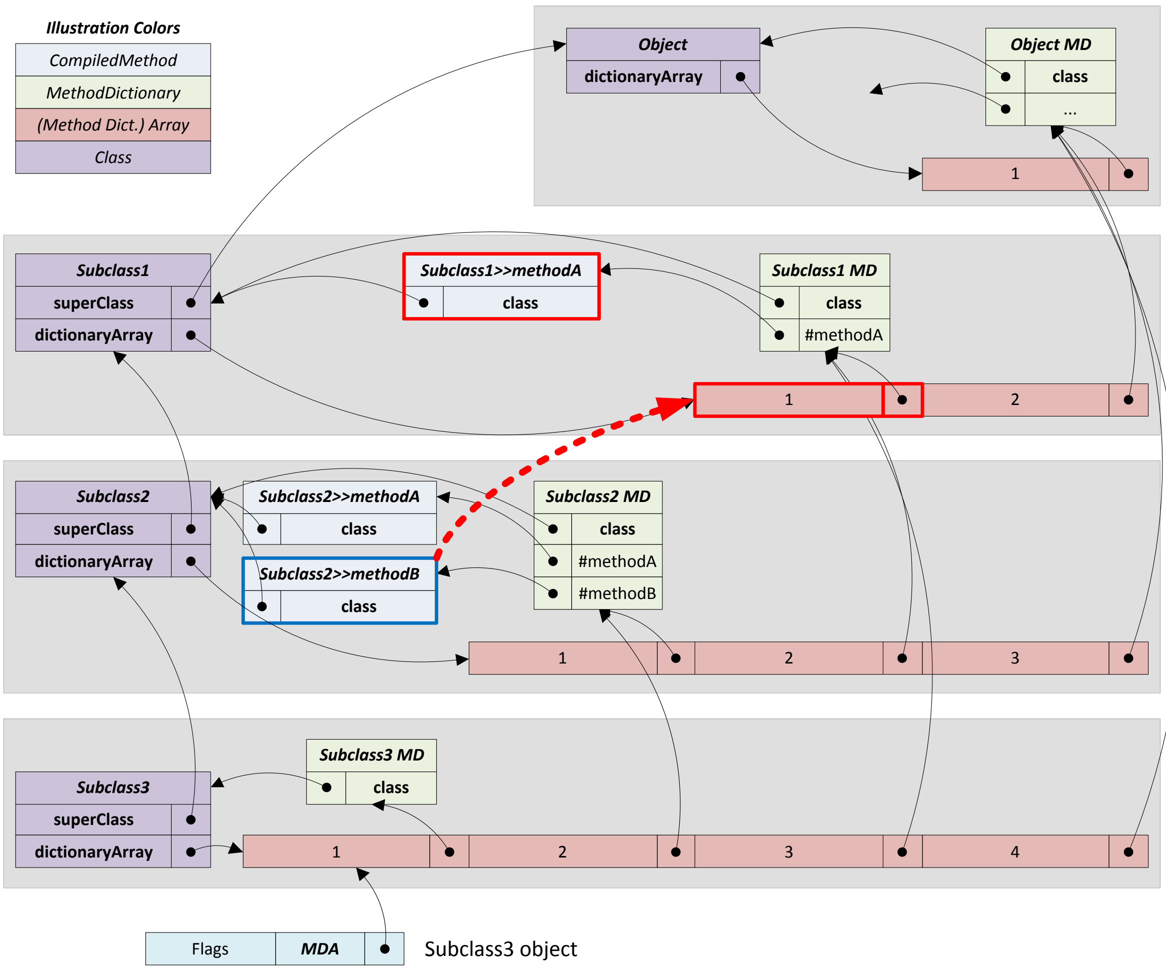
Interesting here is when `#methodB` sends `#methodA`, which version of it is getting executed. Of course, it should be the one defined on **Subclass1**, because this is the one that is *super* implementation relative to `#methodB`. The instance of the receiver, in this case an instance of **Subclass3** has no influence on the method lookup. The illustration below shows the lookup sequences.



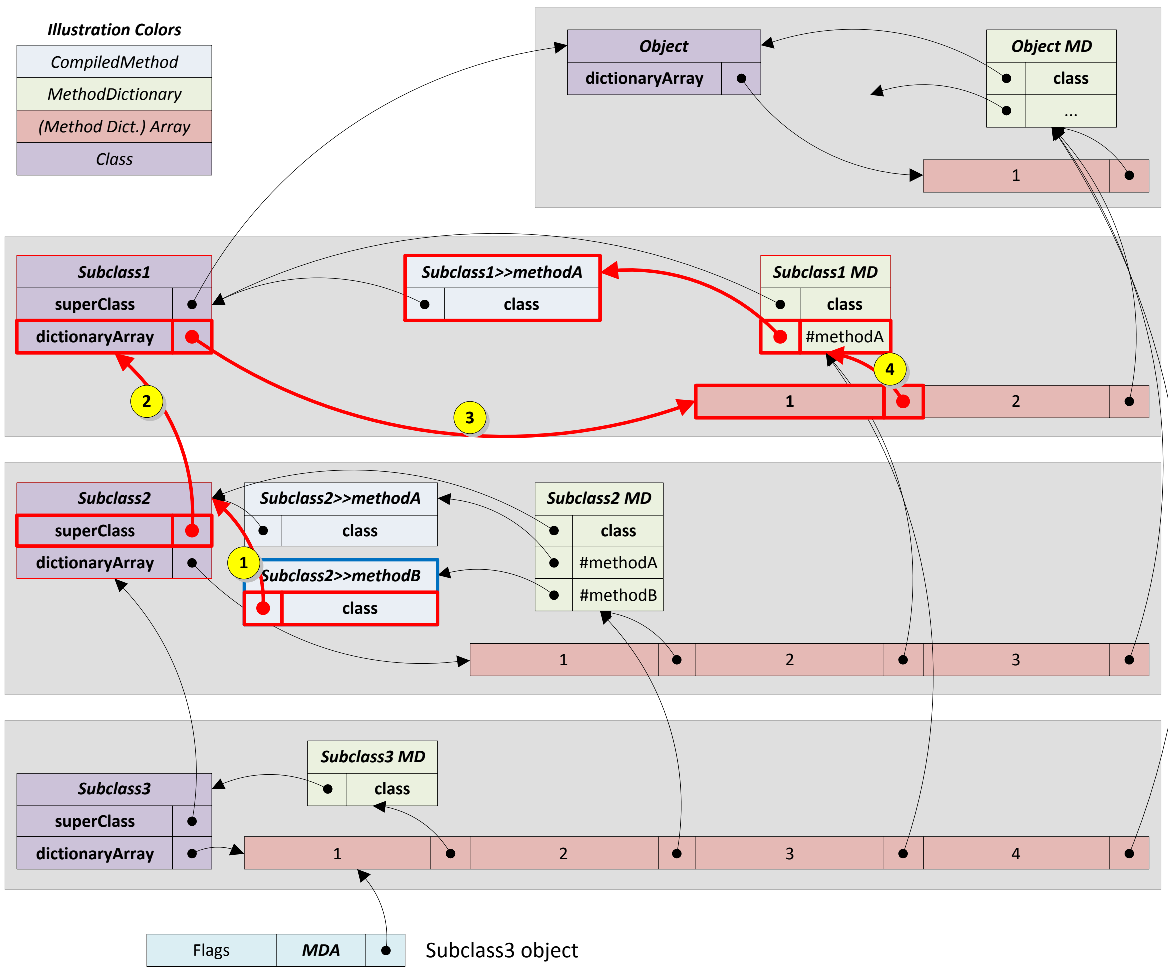
Lookup logic for send of `#methodB`:

1. Find the method dictionary array.
2. Look for `#methodB` in the method dictionary for **Subclass3**. No match.
3. Look for `#methodB` in the method dictionary for **Subclass2**. There's a match.
4. Use the compiled method implementation of `#methodB` to execute the logic in it.

The next step is to send `super methodA`. Obviously, since `Subclass2>>methodB` (coloured in blue) is doing the send, we must find the `Subclass1>>methodA` implementation (coloured in red in the illustration).



The illustration below shows the lookup sequence for the `super methodA` message send:



1. Since `Subclass2>>methodB` is making the super send, lookup the class where it's defined. The class that defines the compiled method is stored in the **class** instance variable of the compiled method.
2. The class (instance of **Behavior**) that defines `#methodB` has an instance variable called **superClass**. The VM uses it to find the superclass object, in our case **Subclass1**.
3. In the super class (**Subclass1**), the VM uses the **dictionaryArray** instance variable to find the method dictionary array for **Subclass1**. This is where lookup for `#methodA` should start.
4. Perform a normal method lookup, as described previously. The example finds an implementation of `#methodA` in the first method dictionary.

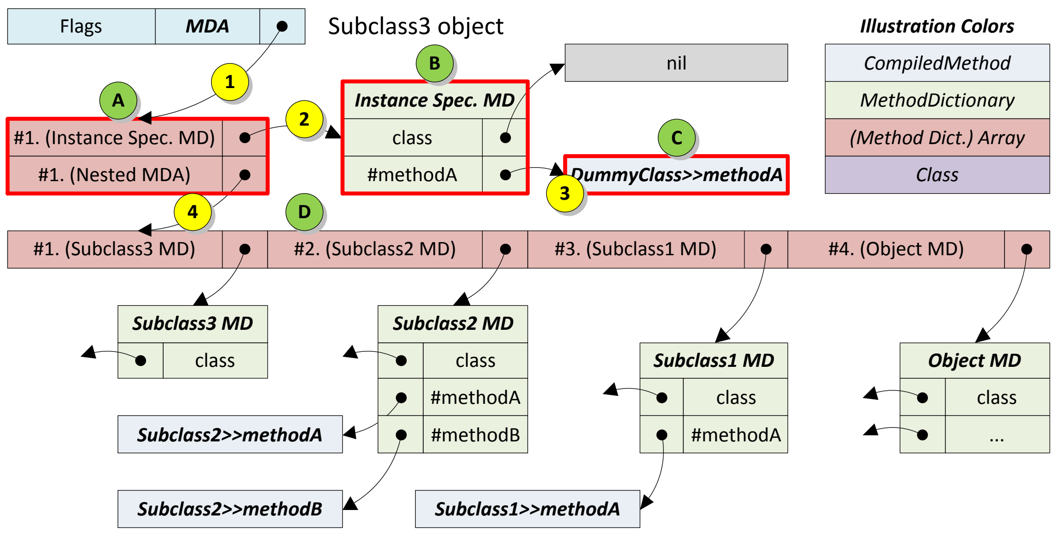
Instance Specific Behaviour

As the title suggests, it is possible to change the behaviour of a single object. This is similar to javascript, where the developer on-the-fly defines the behaviour of an object. To achieve an instance specific behaviour, we must be able ask the VM to perform a method lookup differently than it would usually do. This is done by manipulating the method dictionary array hold by an object.

The simplest solution would be to exchange the method dictionary array for an object with our version, containing the methods we want. This would sound reasonable, but there are some caveats.

- A complex structure of method dictionaries and method dictionary arrays must be copied, merely because we want to add one new method. This is expensive, and (in the old days) would require a lot of memory.
- VSE uses the fact that object has no direct reference to its class. This double dispatch/lookup comes to an advantage when the class needs to be recompiled. VSE creates a new class object, and instead of doing `#allInstances` and replacing the class on each instance, it just takes ownership of the method dictionary array of the old class object and replaces the **class** instance variable with the new class object. Voilà, all objects are now instance of the new class, without becomes or other tricks. If instance variables are added or removed, it's little trickier.
- The VM might internally do identity check / caching on the method dictionary array or in other ways expect correct reference.

Luckily for us, the VSE VM has an easy mechanism to add instance specific behaviour. The method dictionary arrays can be nested. This looks to me like a hack that was implemented to satisfy the needs required by PARTS, but it works. The illustration below shows the example from the previous section, simplified to remove unnecessary information, but this time with added instance specific method.



In this example, we have replaced the method for selector `#methodA` of the object with our own implementation. This is done for a single instance of Subclass3, and not for all instances. Objects added:

A. Method Dictionary Array containing the new instance specific method dictionary at position 1. and the original method dictionary array at the last position.
B. New Method Dictionary, that is specific to our instance. It contains one entry, our own implementation of `#methodA`. The **class** instance variable is set to `nil`. It is allowed to share it between instances.
C. Compiled method with our implementation of `#methodA`.
D. The original method dictionary array.

Luckily, VSE has helper methods to help us with some of the work, especially for creating the nesting method dictionary array. We still have to construct the method dictionary containing our methods. The key methods here are `#addBehavior:` and `#removeBehavior:`. Example:

```
| md cm |
" Create an empty method dictionary "
md := MethodDictionary new.
" Find the compiled method prototype "
cm := DummyClass compiledMethodAt: #prototype_methodA.
cm isNil ifTrue: [ ^self ].
cm := cm copy.
" Add it to the dictionary "
cm selector: #methodA.
md at: #methodA put: cm.
" Change the instance behavior of <obj> "
obj addBehavior: md.
```

Once we have the new method dictionary, VSE does the rest of the work. It is possible to nest several instance specific behaviours (method dictionary arrays) inside each other; take a look at `#addBehavior:`. The magic happens inside `#methodDictionaryArray:`, which is a primitive call to the VM. This "fixes" the object's header with the new method dictionary array. It appears that the VM performs no checking, so the doing something like `obj methodDictionaryArray: true` will crash the image. To change class of the `true` object to the **False** class, execute: `true methodDictionaryArray: false methodDictionaryArray`, but the image obviously will crash.

For the example `obj methodA`, where `obj` in an instance of Subclass3, method lookup is done as follows (see the illustration on the previous page):

1. Find the method dictionary array for the object; in our case the instance specific method dictionary array created by the `#addBehavior:` method.
2. Look for `#methodA` in the first element in the method dictionary array, i.e. the instance specific method dictionary we created.
3. Execute the compiled method found in the instance specific method dictionary. In the example, that's the `DummyClass>>methodA`. **DummyClass** is a class we use to compile the methods. If the method has no direct references to instance variables, we can compile it on any class we want.
4. If the method was not found is step 2 and 3, perform a normal lookup in the original method dictionary array as described previously.

Note: There are no changes to the lookup for `super methodA` sent in `#methodB`.

Things instance specific behaviour allows us to do:

- Overwrite an existing method; just add a compile method to the method dictionary with the selector of an existing method in the hierarchy.
- Add a new method with its own selector, which is new to the hierarchy.
- "Remove a method". Just add a selector to the method dictionary and `nil` instead of a compiled method. E.g. `md at: #printOn: put: nil` will result in *does not understand* if you try to do `#printOn:`.

Generic Message Send Tracing

Why did I spend so much time writing about message sends and instance specific behaviour? Obviously, because this is a helpful tool to implement a generic tracing mechanism. Our requirements are as follows:

- Trace messages sent to an object.
- Enable and disable tracing without recompiling code.
- Optional: Trace parameters.
- Optional: Trace return value.
- Optional: Trace errors that may have occurred.
- Optional: Small impact on performance.
- Flexible

Message Send Tracing

Well, the general idea is simple. Let's say we want to trace message sends to `#methodA`. All we have to do is something like:

```
methodA

      | result |
      Transcript show: 'Calling #methodA'.
      result := self originalMthodA.
      Transcript show: 'Returned from #methodA with: ', result printString.
      ^result.
```

Obviously, naïve implementation that writes on the Transcript, but the general idea is still valid. The drawback however is that we need to recompile a new method and rename `#methodA` to `#originalMethodA`. Also, if done the classical way, the trace will be active for all instances of the class.

Instance Specific Tracing

As you might have guessed, instance specific behaviour will come to our rescue. All we need to do is create an instance specific method dictionary. Generalizing the method from above, we see a pattern:

```
methodA

      | result |
      MessageTracer traceSendBegin: #methodA.
      result := self originalMethodA.
      MessageTracer traceSendReturn: #methodA result: result.
      ^result.
```

We can create (pre-compile) a method prototype and simply copy the compiled method and patch the selector and few literal arrays items to create the instance specific trace method. Of course we need a version that takes 1 parameter, 2 parameters etc.

We still have an issue with how to call the original method. Two options exist:

1. Add the original compiled method to the instance specific method dictionary with a unique/special name. In our example, we prefixed the name with #original. This way the method has been "renamed" for the object instance being traced.

2. Find a way to do `super methodA`, so it ends calling the original method. This is what I ended doing, because I didn't think of the first solution when I implemented the tracing. Also, the first solution ends up having a messy method dictionary and needs to generate a lot of dummy method selectors.

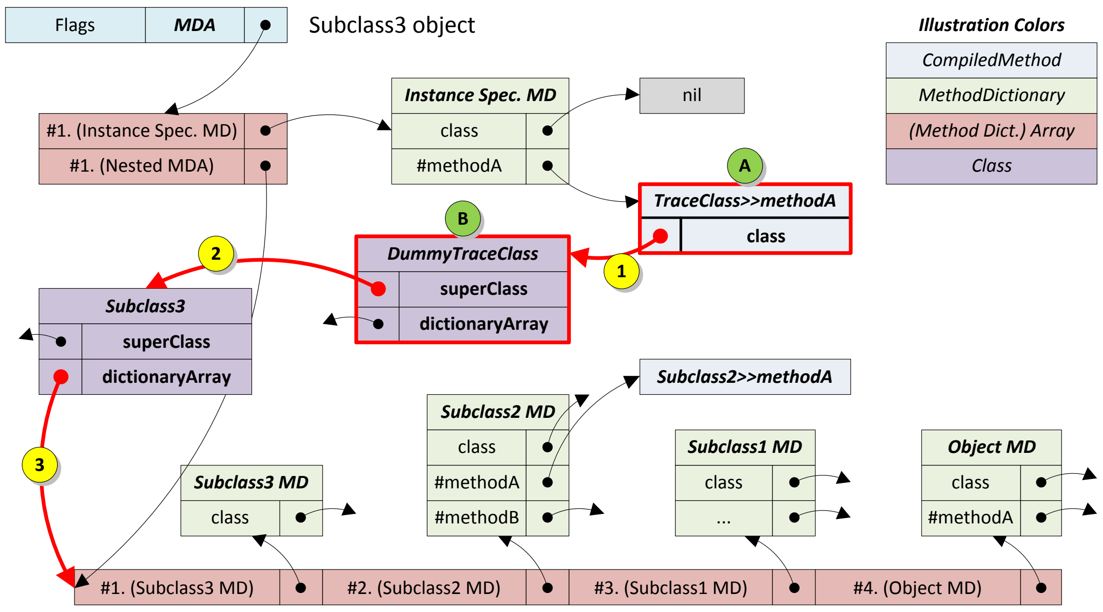
Hacking Instance Specific Super Sends

This is where the magic happens. I am not sure the VSE designers intended this to be done, but as we know, VSE can be tweaked and forced to do a lot of interesting stuff.

The goal is very simple:

- Have an instance of **Subclass3**.
- Implement a trace method named `#methodA` on that instance.
- Have the trace method send `super methodA`, and start the method lookup exactly in **Subclass3**, as if the tracing method did not exist.

It is tricky, but possible! All we need to do is trick VSE to do the *super* method lookup correctly.



The trick is to create a dummy class object (marked B on the illustration) and change the class instance variable of the trace compiled method (marked A on the illustration) to point to that class. The dummy class has the original class as its superclass. In that way, if `TraceClass>>methodA` sends `super methodA`, method lookup happens as follows:

1. Lookup the class of the tracing method. This is the dummy class object we created.
2. Lookup the superclass of that class. This is the real class the object being traced is an instance of.
3. Lookup the method dictionary array for that class and proceed with method lookup as usual.

Note: The dummy class (marked B on the illustration) does very little. The only purpose we have it is to have the VM lookup the superclass and get reference to the real class.

The dummy class can be created with the following method:

```
createTracingClass: aClass

    | cls |
    cls := aClass objectShallowCopy.
    self ASSERT: cls ~~ aClass.
    " Unfortunately, too difficult to change that info for meta classes "
    cls isMetaClass ifFalse: [
        cls newNameSymbol: ('Tracing', cls name asString) asSymbol ].
    " Now, fix the superclass so method lookup for <super> works ... "
    " NB: They still share the same method dictionary array,
        but we don't care - we'll not use it. "
    cls superclass: aClass.
    ^cls.
```

As you can see, the dummy class is pretty raw copy of the class where the trace methods are installed. Only thing changed is the superclass and the name (for now, only for instance classes).

Trace Methods

Next thing to do is create trace methods for every method that needs to be traced. We can compile the methods every time, but this leaves traces in the change.log; we don't want that. I prefer using method prototypes. This is done by having precompiled methods on a class somewhere. Those compiled methods are used to create the trace methods. Example of a prototype:

```
prototype: a1 w: a2 w: a3 w: a4

    "Auto generated method for tracing - method source may be mismatching"

    SSDefaultMessageTracer isNil ifTrue: [
        ^super original: a1 w: a2 w: a3 w: a4.
    ] ifFalse: [
        | args result |
        args := Array new: 4.
        args at: 1 put: a1.
        args at: 2 put: a2.
        args at: 3 put: a3.
        args at: 4 put: a4.

        SSDefaultMessageTracer traceSendBegin: self
            selector: #original:w:w:w:
            class: #OriginalMethodClass
            arguments: args
            sender: self sender.

        [
            result := super original: a1 w: a2 w: a3 w: a4.
        ] on: SSDefaultMessageTracer errorClass do: [ :err |
            SSDefaultMessageTracer traceSendError: self
                selector: #original:w:w:w:
                class: #OriginalMethodClass
                arguments: args
                sender: self sender
                error: err.
            err pass.
        ].

        SSDefaultMessageTracer traceSendReturn: self
            selector: #original:w:w:w:
            class: #OriginalMethodClass
            arguments: args
            sender: self sender
            result: result.

        ^result.
    ].
```

This is a compiled method precompiled on a Smalltalk class somewhere; in our case the class is called **SSErrorHandlingTraceFactory** (see attached source). This method prototype is for methods taking four parameters. Obviously, we have a prototype for methods accepting different number of parameters – currently up to 20. The prototypes are generated by code, so we can easily create more. Also, we have optimized prototypes for methods accepting 0, 1, 2 or 3 parameters. For performance reasons, those do not create the `args` array, but call directly into optimized versions of the trace methods.

The method shown above is a prototype. To create a trace method out of it, we must:

1. Find the prototype method accepting the given number of parameters.
2. Create a copy of the prototype.
3. Change the selector (name) of the copy to the selector of the method we want to trace.
4. Clear the primitive! This is the JIT compiled cached native code. Bad things will happen if this is not nil'ed.
5. Replace literals in the literal array of the method with the correct values. Those are shown with bold (hard to see) and underlined in the above example.
    a. **SSDDefaultMessageTracer** is a global with the object responsible for tracing. In reality, once compiled as a global, the **Association** holding the global can be replaced with a private association. This way, we have a "private" global for each set of trace methods.
    b. `#original:w:w:w:` is the constant symbol used as argument to the *traceSendBegin:* and *traceSendReturn:* methods. It simply indicates which method we are tracing.
    c. `#OriginalMethodClass` is the name of the class where the method being traced is originally located. This together with b. can be used to create a trace string like: *Subclass2>>methodB*.
    d. Finally, `original:w:w:w:` is the literal symbol that tells what method is send as part of the *super send*. We replace that with the selector of the method being traced.

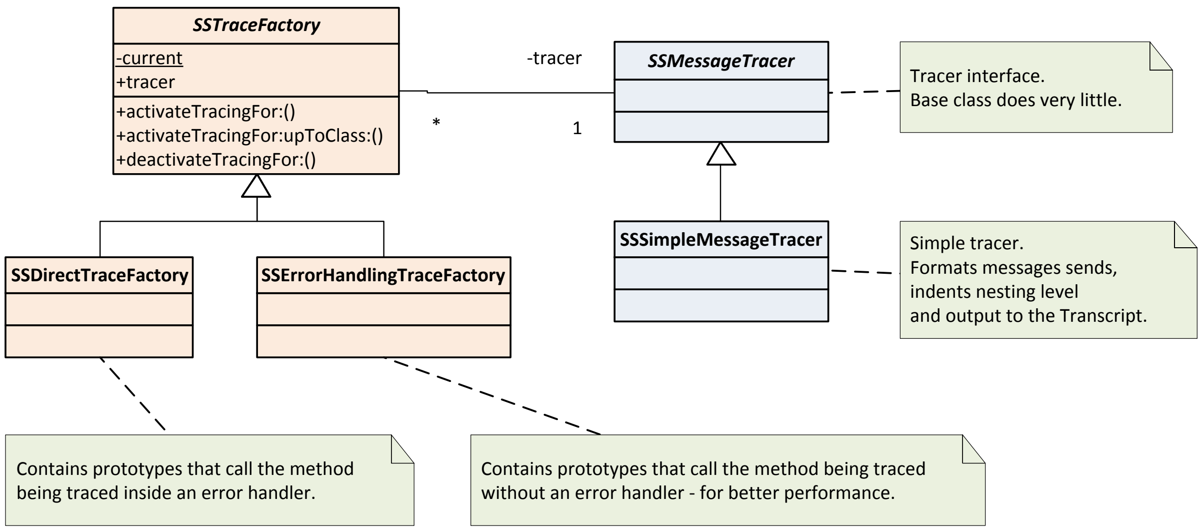
NB: The functionality described here is mainly implemented in `SSTraceFactory>> installTracingMethod:in:`.

Activating Tracing

Now, when we are able to create trace methods, the final step is to activate tracing on an object. The steps needed here are:

1. Create an empty method dictionary.
2. Decide which methods to trace and create a trace method for each method to be traced, as described above.
3. Create the dummy trace class, so super sends work correctly (as described above).
4. Replace the **class** instance variable of each trace method (not the originals, but the newly created trace methods) with the dummy trace class. This will make super sends work.
5. Store the method dictionary somewhere, so we later can deactivate tracing. I've chosen to use a property on the object.
6. Activate instance specific behaviour by calling `addBehavior:`.

Of course, there are helper classes for this. The UML diagram below shows the classes.


SSTraceFactory
-current
+tracer
+activateTracingFor:()
+activateTracingFor:upToClass:()
+deactivateTracingFor:()
-tracer
*
1
SSMessageTracer
Tracer interface.
Base class does very little.
SSSimpleMessageTracer
Simple tracer.
Formats messages sends, indents nesting level and output to the Transcript.
SSDirectTraceFactory
SSErrorHandlingTraceFactory
Contains prototypes that call the method being traced inside an error handler.
Contains prototypes that call the method being traced without an error handler - for better performance.

Main class here is the base trace factory, with the two concrete classes: one that traces errors as well as sends, and one that ignores errors.

Tracing for an object is activated by sending:

```
SSDirectTraceFactory current activateTracingFor: obj upToClass: Subclass2.
```

If `obj` is instance of **Subclass3** (see previous example), this will create trace methods for all methods in **Subclass2** and **Subclass3**, but not for methods defined on **Object** and **Subclass1**. In this example, we use the tracer that does not catch and trace errors.

Deactivating tracing is similar:

```
SSDirectTraceFactory current deactivateTracingFor: obj.
```

Stopping and starting tracing is simple:

```
SSDirectTraceFactory current tracer: nil.
SSDirectTraceFactory current tracer: SSSimpleMessageTracer new.
```

Internals

This section mentions some of the internals of the trace mechanism. The whole source code is attached.

As mentioned before, two concrete trace classes are implemented: SSDirectTraceFactory and **SSErrorHandlingTraceFactory**. The source code for the prototype methods was created with the `#createMethodPrototypeSource:` helper method and manually added to the classes in the browser. The source code for the optimized trace methods was created by hand.

Currently no caching is implemented for trace methods. Since tracing on two objects of the same class will need a method dictionary with the same trace methods, it would be beneficiary to cache the method dictionary – but currently this is not implemented.

Only a simple tracer is implemented (class **SSSimpleMessageTracer**) that traces output to the Transcript. It is not super optimized and a little naïve some times. It has an instance variable named **tracing** that is sets to `true` while tracing, so `#printString` and other diagnostic messages sent to the object will not end in a never-ending recursion. It also has an instance variable named **level** that indicates the nesting hierarchy, so we can visualize what method calls what method.

Finally, **Object** implements `#ssActivateTracing`, `#ssActivateTracingUpToClass: aClass` and `#ssDeactivateTracing` helper methods to activate and deactivate tracing with the **SSDirectTraceFactory** class.

Known Limitations

Most of the known limitations are described above. The tracer can trace methods with up to 20 parameters – I was too lazy to implement more. Tracing output can also be optimized.

Perhaps the most important limitation is the **inability to trace super sends**. It is simply not possible with instance specific behaviour to hook the method lookup for a super send. This would require (if I understand correctly) altering the method dictionary / method dictionary array for a whole class and its super classes. This is too complex, radical and dangerous. See *Super Sends* section in the *Messages and Message Sends* chapter for more on super sends. In other words, super sends will not appear in the trace!

Currently, no mechanism is implemented to prevent tracing of trace methods, if the #activateTracing: methods are called repeatedly.

The methods `#vmInterrupt:`, `#doesNotUnderstand:` and `#sender` will not be traced, because we use `#sender` internally, and the two other I find too dangerous to trace at the moment.

I haven't tested if the **SSErrorHandlingTraceFactory** class handles curtailing (return from blocks) correctly.

Conclusion

Well, this is a document about the message send and method lookup mechanism in VSE, and how it can be used to trace message sends. As we can see, VSE is relatively flexible and the VM is relatively generic, not exclusively bound to the Smalltalk semantics.

More experiments with VM are needed. Most of the knowledge in this document comes from hard work, common sense and experimenting! Good way to find out when the VM uses what is to put garbage in an instance variable that you expect the VM to access. If it crashes, usually with a GPF, then the VM needs it. Example is the super sends; put nil in the **class** instance variable of a compiled method, and it works fine, until it does a super send. Put a bogus class in the same place and you will confuse the VM. Put a class from another class hierarchy, and the super send will look up a method in completely different class hierarchy and end executing completely different method than one would expect.

Some unexplored options are left:

Mixins

Mixins (http://en.wikipedia.org/wiki/Mixin) are something I've really wanted in Smalltalk. They are like interfaces plus implementation. Some naïve examples:

```
do: aBlock
	self implementedByConcreteClass.

everySecondDo: aBlock
	| even |
	even := false.
	self do: [ :each |
		even ifTtrue: [ aBlock value: each ].
		even := even not.
	].

withWaitAndStatusDo: aBlock
	CursorManager execute changeFor: [
		self do: [ :each |
			StatusIndicator current text:
				'Processing: ', each printString, ' ...'.
			aBlock value: each.
		].
	].
```

It would be nice to add this methods to all classes (collections) that implement `#do:`.

From what I understand, the VM creates objects the following way:

1. Looks up the size of the object in the class (**structure** instance variable) and reserves memory space on the heap.
2. Sets the objects flags (not interesting for us).
3. Copies to value of the **dictionaryArray** instance variable of the class object to the header of the new object. This is like setting the class of the new object. From what I can guess, the VM blindly copies this object reference and doesn't use it

internally. This opens the possibility to add "instance specific behaviour" to all new objects of a class! Cool!

With this knowledge, it should be possible to create a mixin. Having every instance have the mixed behaviour described above would require us to patch the **dictionaryArray** instance variable of classes. We need to have the development tools access the *"correct"* information, and the runtime use the compiled/combined method dictionary array for method lookup. One must experiment what order the VM looks up methods in the nested method dictionary arrays. What if we add the "instance specific methods" **after** the standard class methods? How to handle super sends in mixins? How to handle methods that need to be implemented by the concrete classes? Why the class instance variable of an instance specific method dictionary does needs to be `nil`? Is this needed for the `#class` message to work correctly, or due to instance mutation during class shape changes and similar?

Anyway, combining the method dictionary of a mixin class with the method dictionary of a concrete class may be the way to implement mixins in VSE. More research is needed!

Break Points

Another pain in the *"behind"* in VSE is the need to put `self halt` so often. May be using similar techniques like the tracing and mixins described above can help is implement break points. Those of course cannot be put arbitrary inside a method, but only on entry or exit. This should be enough in many cases.

Proxy Objects / Generic Tracing

Often tracing all messages to an object is needed, or some more tricky proxy operations are needed. One way people have done that is create a class outside the **Object** hierarchy and have it implement `#doesNotUnderstand:` and have some logic in there.

The downside with this approach is that we need a proxy object to encapsulate the real object. What if the real object already exists? It's not possible (or easily possible) to replace the real object with the proxy object.

An idea here is; what if we change the class of the real object to behave like a proxy object, without need to change reference to it or losing its state. To do this:

1. Create a method dictionary array and a single method dictionary.
2. Add the required `#doesNotUnderstand:` and `#vmInterrupt:` methods to it and needed methods to access instance state and variables, like `#basicAt:` and `#basicAt:put:`.
3. Set the **class** instance variable correctly of the method dictionary.
4. Replace the whole method dictionary array of the object with the new method dictionary array using the `#methodDictionaryArray:` method.

The object now behaves as if it has the behaviour of a class defined outside the object hierarchy.

As you can see, possibilities are almost endless ... enjoy!

Todor ;-)