# BabySRE,
# Squeak Reverse Engineering

**Trygve Reenskaug.**
**Dept. of Informatics, University of Oslo, Norway.**

## Abstract

SRE (Squeak Reverse Engineering) consists of three tools making existing Squeak objects visible and tangible:

1. The SRE Execution Tracer dumps a snapshot of the stack on the Transcript.
2. *The SRE Object Inspector* gives me a snapshot of a single object with its identity, state, and behavior. I can edit its state as in a Squeak Inspector and its behavior as in a class Browser. The class hierarchy is flattened so that I see the object as it appears at runtime.
3. The *SRE Context Browser* lets me diagram the instantaneous structure of selected objects. With this tool, I find that I can master larger and more complex systems than I could without it

Taken together, these tools put me in closer contact with the objects as they exist in the system, thereby giving me a better understanding of the system as it is.

## Document History

2004-12-19 (Trygve): First release: BabySRE-TRee.11
2005-01-13 (Trygve): Updated for release: BabySRE-TRee.34
2018-07-30: (Trygve): Updated and extended the document.
2018-10-27 (Trygve) - Updated copyright notice and app.1:Installation..
2018-10-31: (Trygve) - Added ref to draft Personal Programming article, Improved installation section.

## Copyright notice

# Introduction

"*Conceptual abstractions may be formed by filtering the information content of a concept or an observable phenomenon, selecting only the aspects which are relevant for a particular subjectively valued purpose*"[1]. In Squeak, everything of interest is represented by an object; there is nothing but objects in Squeak. Two abstractions support programmers' mental models:

1. *The Class abstraction.* An object is an instance of a class. This abstraction stresses the inner construction of an object; its instance variables and methods. It hides the object's identity and its place among objects in its environment.

2. *The Role abstraction*. An object is an entity with an immutable and globally unique identity that collaborates with other objects to achieve a goal. An object can play many roles and a role can be played by many objects at different times. An object exhibits its state and behavior through its provided message interface that hides its inner construction.

The second abstraction will be fully described in my article "Personal Programming." [draft Reenskaug2018]

Squeak's many browsers gives excellent support for thinking and programming with the class abstraction. *SRE, Squeak Reverse Engineering*, complements this by providing three tools for working with runtime structures of collaborating objects in the role abstraction:

- *SRE Execution Tracer*. `Object>>traceRM:levels:` is like `Transcript>>show:` with the addition of the oop of the writing object and a dump of the stack to a specified depth.

- *SRE Object Inspector.* A class is a partial description of its instances. Partial, because the description is fragmented between its superclasses and also because the class does not disclose the state or identity of its instances. The SRE Object Inspector shows the state and behavior of an object. The state is shown as the instantaneous value of all its instance variables. The behavior is shown as all its methods in its flattened class hierarchy.

- *SRE Context Browser.* The essence of object orientation is that objects collaborate to achieve a goal. The Context Browser is used to plot an instantaneous context; a substructure of collaborating objects.

[1] https://en.wikipedia.org/wiki/Abstraction

I first revisit Smalltalk/Squeak and see that it is a universe of objects with a VM that continuously executes its CompiledMethod objects. I next describe the three SRE tools illustrated by examples. I frequently find I use the tools when I wander into uncharted parts of a system. It is much easier to read the code when I understand the important objects and the relationships between them.

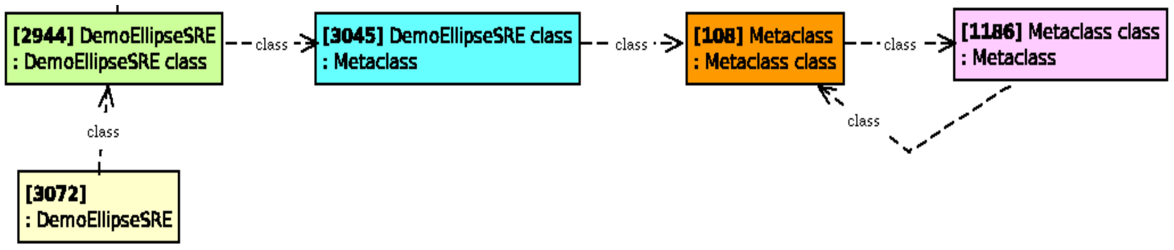
## Smalltalk

Smalltalk, of which Squeak is a derivative, is a universe of objects and nothing but objects. They are under the control of a Smalltalk virtual machine that is a combination of an interpreter and a runtime system. Smalltalk includes functionality that is normally found in IDEs, applications, programming languages with their compilers, and operating systems. The universe of objects includes objects that represent a variety of things such as applications, numbers, strings, dictionaries, text editors, classes, compilers, stack frames, methods, and graphical elements. A Smalltalk object has a unique and immutable identify; its oop. An object encapsulates state and behavior; state in its instance variables and behavior in its methods. Conceptually, the methods are encapsulated within the object. It would be very inefficient if the realization of every object should include a copy of its methods, so the methods are stored in a shared class object. Smalltalk methods are instances of class CompiledMethod. An instance contains a sequence of byte codes and primitive operations that are translated from textual form by a compiler method. Different classes can implement different compilers that define different languages. So when people say *the Smalltalk language*, they are imprecise and often mean the default language that is understood by most, but not all, class objects. Further, the language is used to specify methods only; it does not include the declaration of classes, packages, or anything else. (New classes are created programmatically).

The Squeak execution takes the form of messages flowing from object to object under the control of their methods. Conceptually, the execution started in the nineteen-seventies and has never stopped. The execution may be paused and a snapshot of the state of the object universe saved to a *Smalltalk image file*. The image may later be loaded into a computer and the execution continues with the first operation after the snapshot operation. The snapshot is achieved with a primitive operation:

```
SmalltalkImage>>snapshotPrimitive
"Primitive. Write the current state of the object memory on a file in the same format as
the Smalltalk-80 release. The file can later be resumed, returning you to this exact
state. Return normally after writing the file."
<primitive: 97>
^nil "indicates error writing image file"
```

Every object is an instance of a class. Since the class is represented by an object, this object must be an instance of a class, and so on until there is a class that directly or indirectly is an instance of itself. This is illustrated by the class (i.e., instance-of) links in this SRE context diagram[2]:


[2944] DemoEllipseSRE : DemoEllipseSRE class
class
[3045] DemoEllipseSRE class : Metaclass
class
[108] Metaclass : Metaclass class
class
[1186] Metaclass class : Metaclass
class
class
[3072] : DemoEllipseSRE

The *instance-of hierarchy* is shown as arrows marked class in the diagram. It is essential to the operation of Smalltalk since the behavior of an object is declared in its class. In contrast, the *class inheritance* hierarchy has no runtime significance since the VM works as if all methods were specified in the object itself[3]. The inheritance hierarchy is thus in the nature of a comment created for the convenience of the programmer.

## The Demo Program

For the purposes of this article, I have created a class for a colored ellipse that cycles through a sequence of different colors:

```
EllipseMorph subclass: #DemoEllipseSRE
    instanceVariableNames: 'colorIndex'
    .....
EllipseMorph >>step
    | colors |
    colors := {Color red. Color green. Color blue. Color magenta. Color yellow}.
    colorIndex ifNil: [colorIndex := 1].
    colorIndex := colorIndex + 1 \\ colors size + 1.
    self color: (colors at: colorIndex).
```

I start the demo by executing

```
DemoEllipseSRE new openInWorld.
```

and observe a small ellipse in top-left corner of the screen that regularly changes its color.

[2] A box represents an object. The annotation is [oop], possibe instance name, : , class name.
[3] The hierarchy can always be refactored into a subclass of nil by judiciously copying and renaming its features.

## SRE Execution Tracer.

The SRE trace operation, Object>>traceRM: anObject, is a sophisticated Transcript operation. It works as a regular Transcript show: with the addition of information about the sending object and method and the current stack. There are 3 variants:

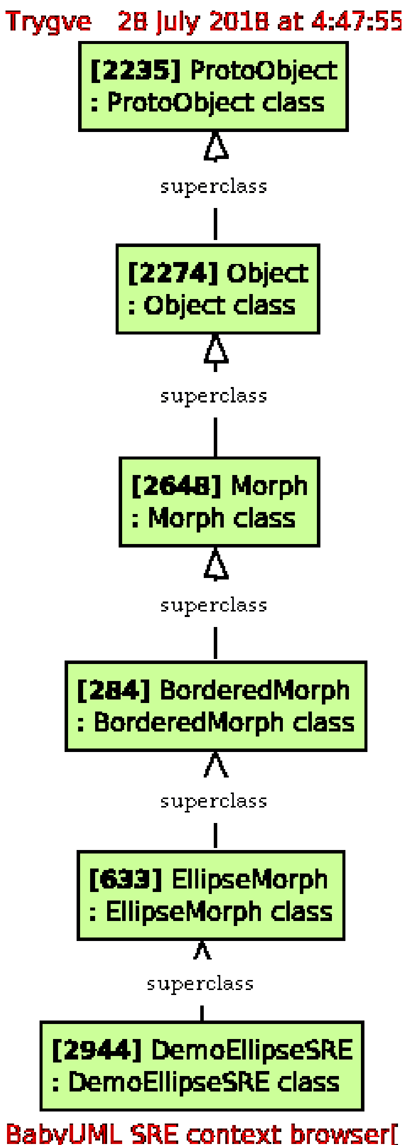
> **Object>>**traceRM: anObject,
> Object>>traceRM: anObject levels: levCount,
> Object>>traceRM: anObject levels: levCount  withContext: aContext

*anObject* is the object to be printed. The default is anObject printString.
*aContext* is a stack frame. The default is thisContext sender.
*levCount* is the number of stack levels to show. The default value is 1.

Examples of its use are shown in section *Projection 3: The Stepping Collaboration*.

## SRE Object Inspector for investigating a Single Object

The Squeak inspector is a powerful tool for investigating the current state of an object. It is easy to open new inspectors on its instance variables and thereby build a picture of the system. The downside is that the screen (and my brain) quickly gets cluttered with inspectors. Also, I might get confused if I have open inspectors for different instances of the same class because the instances all look the same: An ElllipseMorphSRE. This last problem has been remedied by adding identity information in Object>>printOn:, for example [3072]: DemoEllipseSRE. (Note: The problem of many inspectors is partially solved by the Squeak explorer).

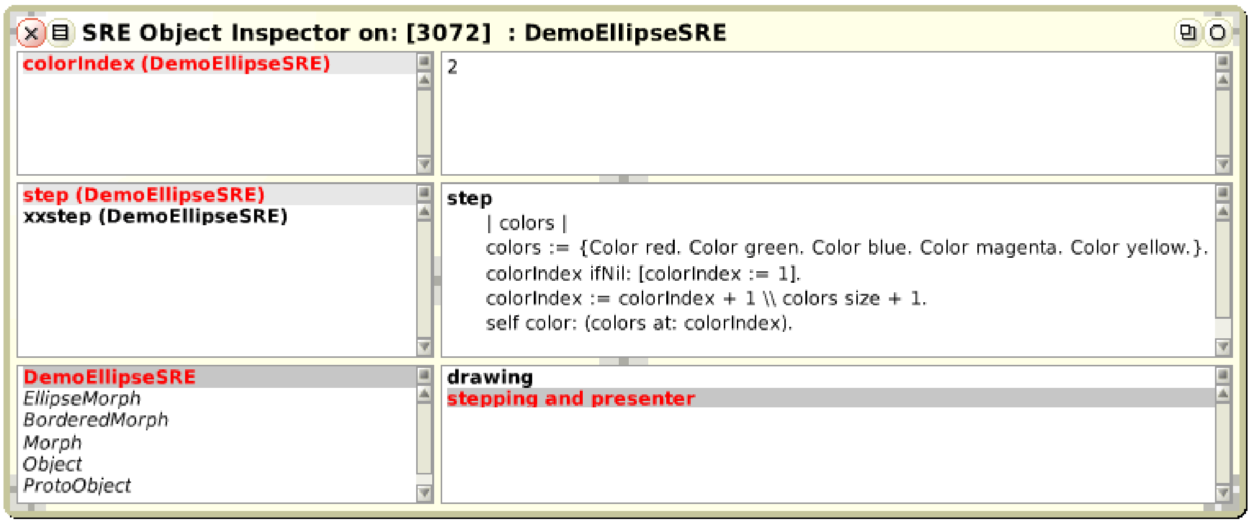


Trygve 28 July 2018 at 4:47:55
[2235] ProtoObject : ProtoObject class
superclass
[2274] Object : Object class
superclass
[2648] Morph : Morph class
superclass
[284] BorderedMorph : BorderedMorph class
superclass
[633] EllipseMorph : EllipseMorph class
superclass
[2944] DemoEllipseSRE : DemoEllipseSRE class
BabyUML SRE context browser[

The properties (state and behavior)of an object are given by its class and the superclasses of that class. The relevant objects are shown in the diagram on the left. [3819]DemoEllipseSRE is a subclass of [633]EllipseMorph etc. It is interesting to note that the objects shown in this diagram taken as a whole exactly describe the DemoEllipseSRE that is flashing different colors somewhere on the screen. It may also be interesting to note that the defining classes in all probability belong to several packages.

The *SRE Object Inspector* inspects a runtime object as a whole with its identity, state and behavior as declared in a flattened class hierarchy. The Object Inspector shifts the focus from the class to the object, very illuminating.

Developer note: Sometimes, it is useful to extend the name of the object with the name of the class. At other times, this can be confusing as illustrated in this diagram. May be the default should be that the class name should only be shown for anonymous objects without a name on their own.


SRE Object Inspector on: [3072] : DemoEllipseSRE

colorIndex (DemoEllipseSRE)

2

step (DemoEllipseSRE)
xxstep (DemoEllipseSRE)

```
step
    | colors |
    colors := {Color red. Color green. Color blue. Color magenta. Color yellow.}.
    colorIndex ifNil: [colorIndex := 1].
    colorIndex := colorIndex + 1 \\ colors size + 1.
    self color: (colors at: colorIndex).
```

DemoEllipseSRE
*EllipseMorph*
*BorderedMorph*
*Morph*
*Object*
*ProtoObject*

drawing
stepping and presenter

The top-left of this browser is a filtered list of the object's instance variables. The top-right is the value of the selected one (as in the Squeak Inspector, but filtered). The second row is a filtered list of the object's methods together with the code of the selected one. The third row shows two filters as multiple select lists; a superclass list and a list of method categories. Both filter what is shown in the instance variable and method lists. More examples of its use is in section Projection 4: The Instantiation Hierarchy.

## SRE Context Browser for Displaying an Object Substructure.

The essence of object orientation is that networks of collaborating objects work together to achieve a common goal. The Squeak universe of objects is large and has a structure that is so complex that it can't be grasped by the human mind. The *SRE Context Browser* lets me build a model of an interesting substructure and project it onto an *SRE Context diagram*. I find that this tool helps me master larger systems than I could master without it. A diagram shows a substructure, or *projection*, of the universe of objects. A box identifies an object and represents the *role* that the object plays in the *context* of its peers. Its yellow menu command are:

- *SRE object inspector*. Open an SRE object inspector on the selected object.
- *Squeak inspect object*. Open a normal Squeak Inspector on the selected object.

- *Squeak inspect symbol*. A debug operation that opens a Squeak inspector on the symbol object.
- *Squeak browse class*. Open a normal Squeak Browser on the class of the selected object.
- *add link for variable...* Select an attribute object, create link to the this object, create new role if needed.
- *add link for DCI collaborator...* DCI experiment, not working.
- *add context object...* DCI experiment, not working.
- *add role playing object...* DCI experiment, not working.
- *add role for all pointers to this object...* Create a collection of all objects that point to this object, create a new role for this collection.
- *add role from expression...* Type an expression. Evaluate it in the context of the selected object. Add a role on the resulting object.
- *rename role...* Rename the selected role. The selected object is unchanged.
- *delete role.* Remove the selected role from the diagram. The selected object is unchanged.

Four projections illustrate its use. Each highlights some aspects of the same universe of objects while hiding all the rest.[4] You may, as I do, find it boring reading. But they illustrate what you can expect if you install the SRE tools in your own image. I find I frequently use them when I wander into uncharted parts of a system. It is so much easier to read the code when I understand the nature of the objects and the relationships between them.

- *Projection 1: Domain Collaboration*. The Morphic objects that collaborate to run the DemoEllipseSRE program.
- *Projection 2: Instantiation and Inheritance hierarchies*. The instantiation and inheritance hierarchies are often confused, yet they bear no relationship to each other.
- *Projection 3: The Stepping Collaboration.* An investigation into the Squeak stepping mechanism. How does it work?
- *Projection 4: The Instantiation Hierarchy.* Every class can file out an .st-file. Where is the method that does it?

## *Projection 1: Domain Collaboration*

The focus of attention is the DemoEllipseSRE I created when I executed

```
DemoEllipseSRE new openInWorld.
```

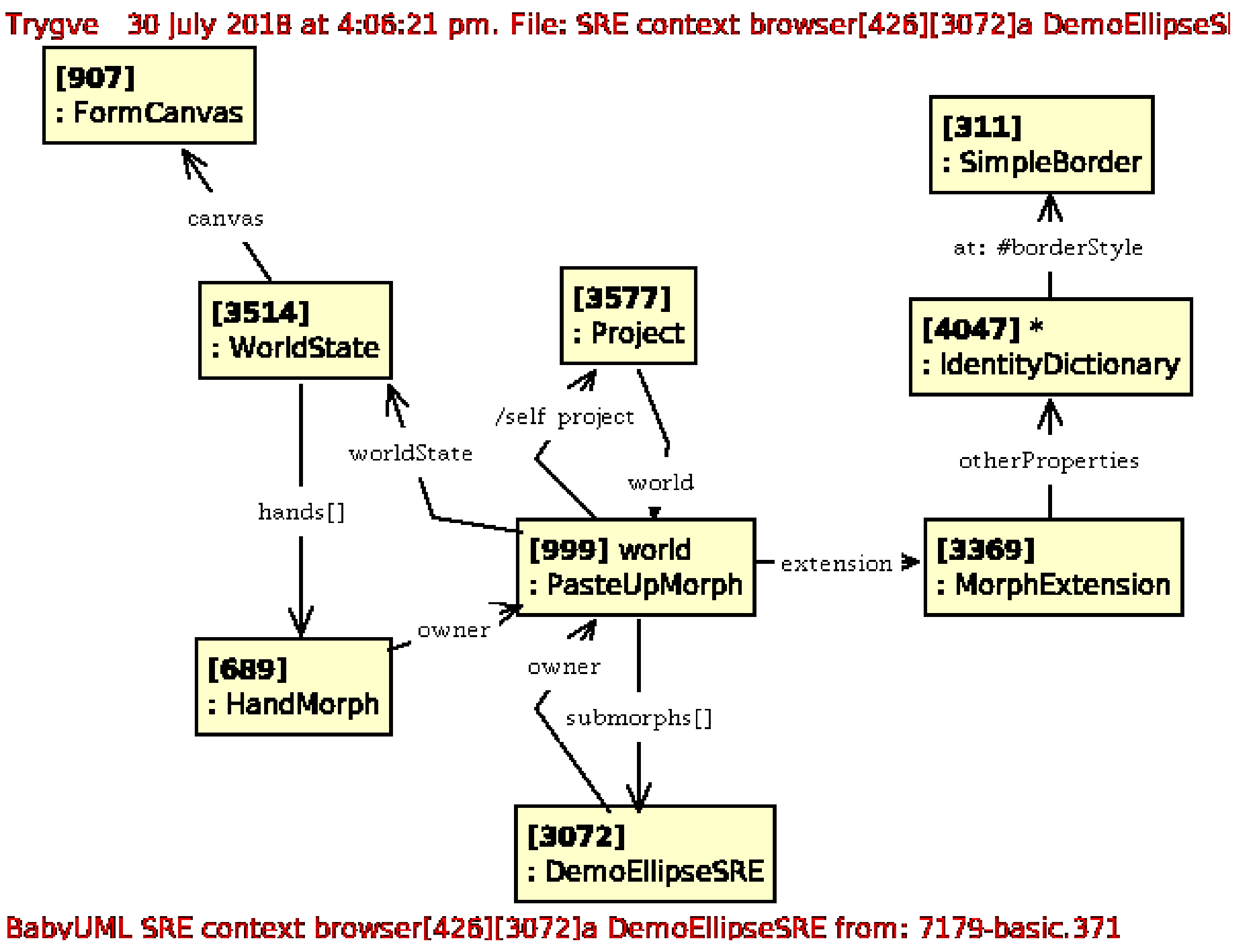
[4] The snapshot of the universe of objects while it executes the Demo Program contains 490,055 objects.

To create the diagram, I pointed to the ellipse, opened the halo, selected the debug command SRE Context Browser, opened a new diagram, and placed the new [3072]: DemoEllispeSRE box.[5]

This is an example of the simplest kind of morph. It is owned by [999] world, a morph that controls the whole Squeak screen and has all the visible elements as submorphs. (world is an instance of PasteUpMorph. Its Balloon help says that it is a morph whose submorphs comprise a paste-up of rectangular subparts which "show through").

I see that world has a link named extension holding a SimpleBorder. The diagram doesn't indicate the meaning of this value, but it does warn us of its existence.

The world also has a link named worldState to [3514] :WorldState that has a link to [907] :FormCancas, a reference to the screen. [3514] :WorldState also has a collection hands, one of them is [689] :HandMorph, which is the global variable ActiveHand. The hand's submorphs hold anything being carried by dragging.


Trygve 30 July 2018 at 4:06:21 pm. File: SRE context browser[426][3072]a DemoEllipseSl

BabyUML SRE context browser[426][3072]a DemoEllipseSRE from: 7179-basic.371

In the diagram, an objects is identified with a String:

[<oop>]<object name, if any> : <class name>

[5] More details in Appendix 2: How I created the domain collaboration

The relationship between [999]world and [3577]project is not so easy to find. A Project has an instance variable, world, which indicates that there is at most one world in a project. But why doesn't world have an instance variable for its project? [999]world finds [3577]project programmatically with its project method; I use the *add role from expression* command with self project to create the link.

```
PasteUpMorph>>project
    "Find the project that owns me.  Not efficient to call this."
    ^ Project ofWorld: self
Project class>>ofWorld: aPasteUpMorph
    "Find the project of a world."
    "Usually it is the current project"
    CurrentProject world == aPasteUpMorph ifTrue: [^ CurrentProject].
    "Inefficient enumeration if it is not..."
    ^ self allProjects detect: [:pr |
        pr world isInMemory
            ifTrue: [pr world == aPasteUpMorph]
            ifFalse: [false]]
        ifNone: [nil]
```

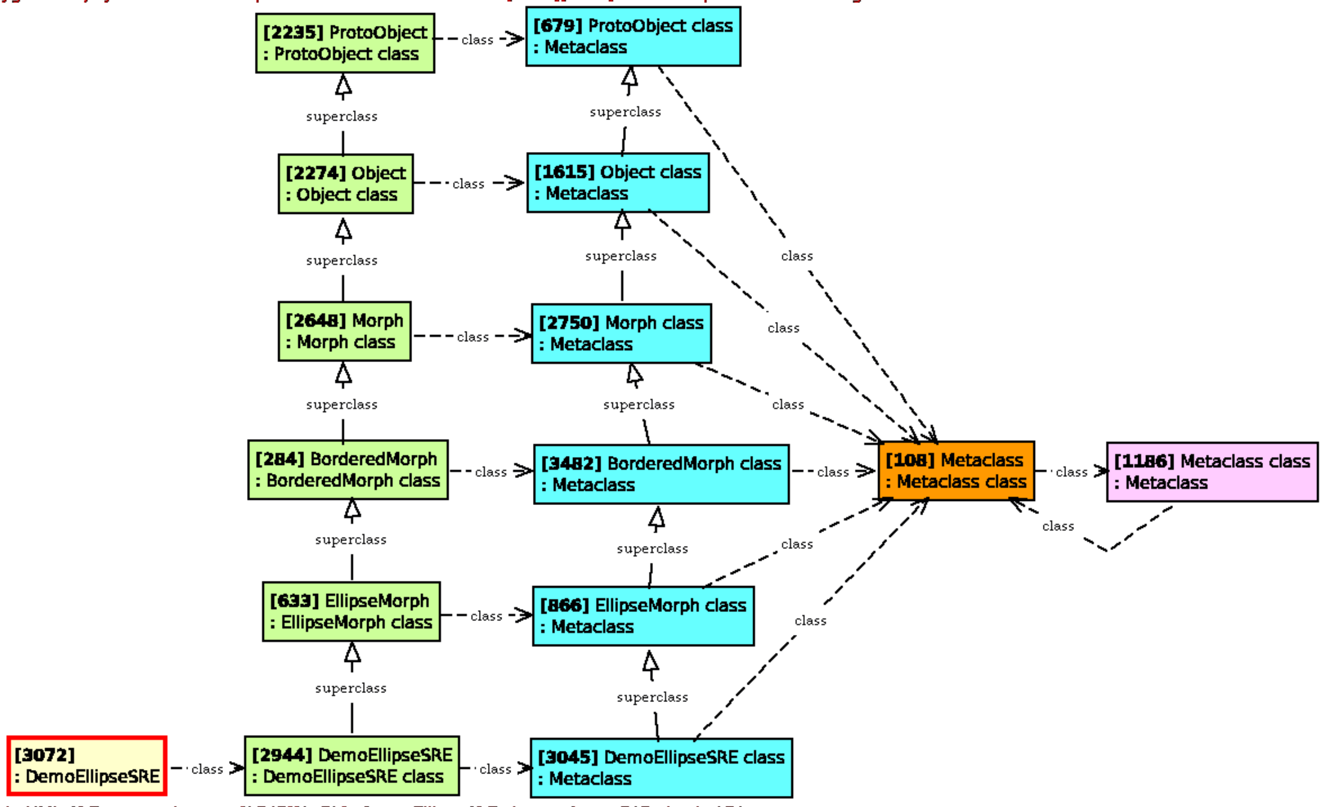
This smells like a scheme for swapping worlds in a virtual memory, but I will not investigate this further.

## *Projection 2: Instantiation and Inheritance hierarchies*

Every object is an instance of a class. A class object is likewise the instance of a class. (Classes whose instances are classes are often called *metaclasses*). The metaclass is also an instance of a class -- does it ever stop? [6] Subclassing and instantiation are independent mechanisms and are shown orthogonally in the diagram. The superclass links are shown vertically and the instance-of horizontally. (The latter links are marked class since an object is an instance of a class).

[6] The Smalltalk architecture is given in some complex diagrams in the [Blue book] (on pages 270-272 in my copy).

Trygve   28 July 2018 at 4:20:51 pm. File: SRE context browser[2747][3072]a DemoEllipseSRE-classes.2.gif


[2235] ProtoObject : ProtoObject class
[679] ProtoObject class : Metaclass
[2274] Object : Object class
[1615] Object class : Metaclass
[2648] Morph : Morph class
[2750] Morph class : Metaclass
[284] BorderedMorph : BorderedMorph class
[3482] BorderedMorph class : Metaclass
[108] Metaclass : Metaclass class
[1186] Metaclass class : Metaclass
[633] EllipseMorph : EllipseMorph class
[866] EllipseMorph class : Metaclass
[3072] : DemoEllipseSRE
[2944] DemoEllipseSRE : DemoEllipseSRE class
[3045] DemoEllipseSRE class : Metaclass
superclass
class

BabyUML SRE context browser[2747][3072]a DemoEllipseSRE-classes from: 7179-basic.371

The properties of the flashing [3072]:DemoEllipseSRE are specified by its class [2944] with superclasses, i.e., one right and 5 up in the diagram. Similarly, the properties of any other object are specified by its class object. E.g., [108]Metaclass is an instance of [1186]Metaclass class which in its turn is an instance of [108]Metaclass and the recursion is closed.

The squeak-dev mailing list sometimes shows questions arising from a confusion of a class with its metaclass. I am speculating that the metaclass object could be described as a regular object in the system design (sometimes called a *factory object*). May be we don't need one for every class; several classes could share a common factory object.

## *Projection 3: The Stepping Collaboration*

The demo example gave me the inspiration to find out how and from where the *step* method was called. I augmented the step method with traceRM: self levels: 10 and made sure to execute it only once to avoid an overflowing the Transcript :

```
DemoEllipseSRE>>step
    " Version for stepping experiements. "
    | colors |
    self doOnlyOnce: [self traceRM: self levels: 10].
    colors := {Color red. Color green. Color blue. Color magenta. Color yellow.}.
    colorIndex ifNil: [colorIndex := 1].
    colorIndex := colorIndex + 1 \\ colors size + 1.
    self color: (colors at: colorIndex).
```

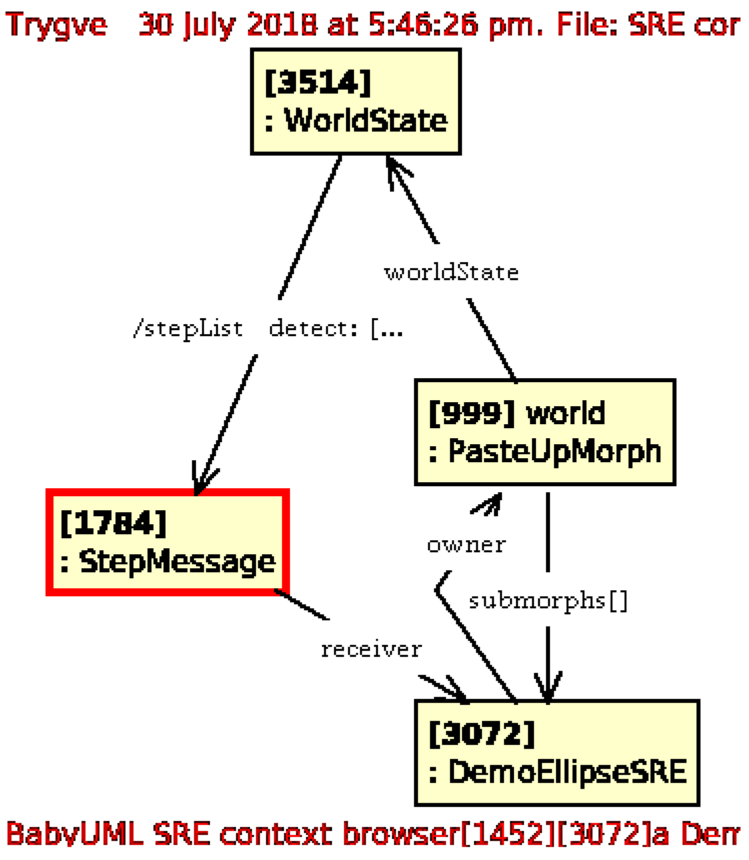
The result was the following trace in the Transcript:

```
1  [3072] : DemoEllipseSRE >> step {[3072]a DemoEllipseSRE}
2  [3072] : DemoEllipseSRE >> doOnlyOnce:
3  [3072] : DemoEllipseSRE >> step
4  [3072] : DemoEllipseSRE >> stepAt:
5  [1927] : StepMessage >> value:
6  [3514] : WorldState >> runLocalStepMethodsIn:
7  [3514] : WorldState >> runStepMethodsIn:
8  [999] : PasteUpMorph >> runStepMethods
9  [3514] : WorldState >> doOneCycleNowFor:
10  [3514] : WorldState >> doOneCycleFor:
```

I see that the invocation of the step method originates in [3514]WorldState via [999]world and back to [3514]WorldState>>runLocalStepMethodsIn: to [1927]StepMessage. It transpires that this StepMessage is one of many in [1042:Heap]. This Heap changes very rapidly and the addLinkForVariable-command didn't work properly. I removed the Heap from the diagram, selected the WorldState, and used addRoleFromExpredssion to find the appropriate StepMessage:

```
stepList  detect: [:step | step receiver asOop = 3072]
```

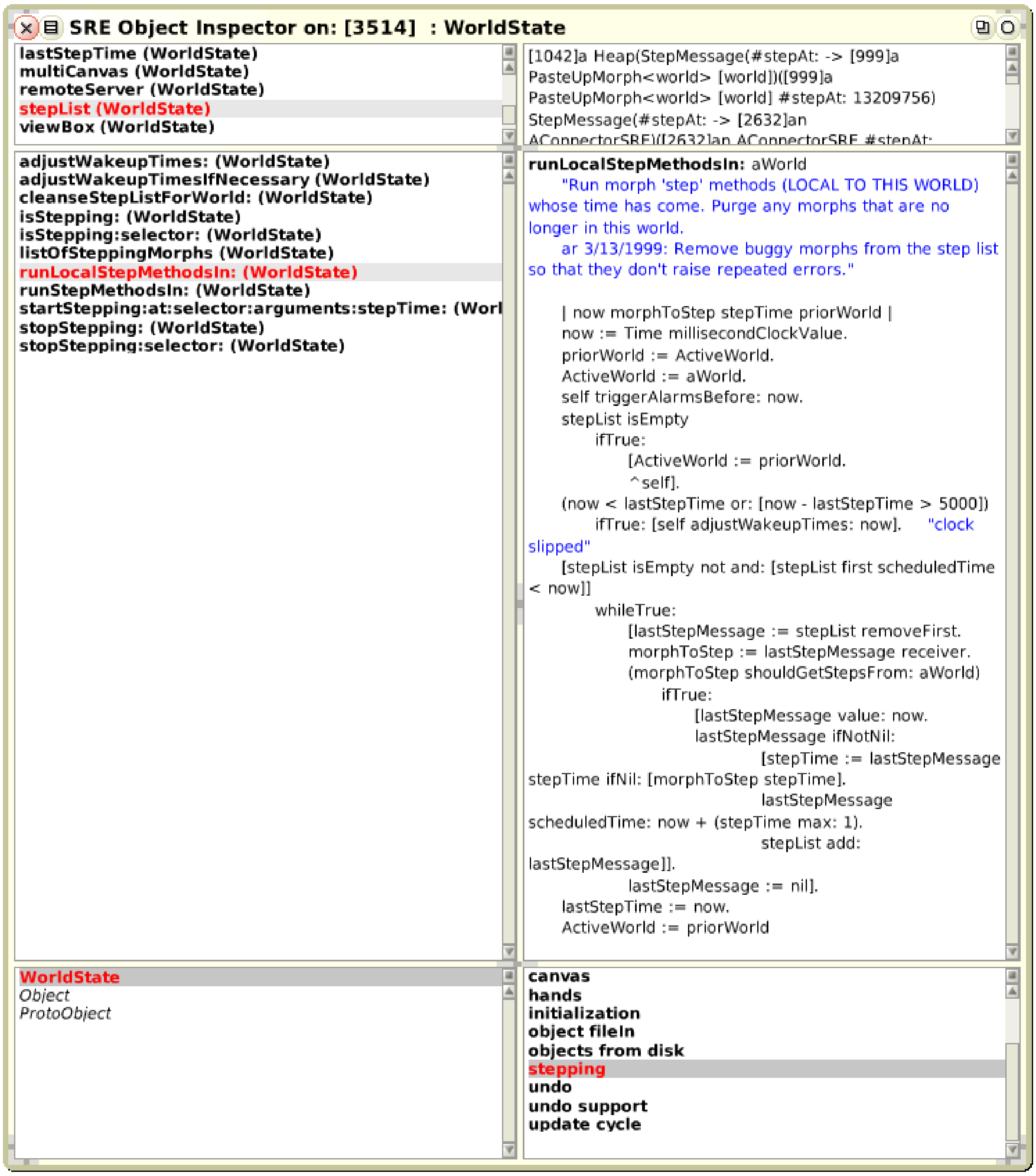
which added [441]:stepMessage to the diagam and its link to receiver completed it.


Trygve 30 July 2018 at 5:46:26 pm. File: SRE cor
[3514] : WorldState
worldState
/stepList detect: [...
[999] world : PasteUpMorph
[1784] : StepMessage
owner
submorphs[]
receiver
[3072] : DemoEllipseSRE
BabyUML SRE context browser[1452][3072]a Dem

An interesting part of the trace was

6 [3514] : WorldState >> runLocalStepMethodsIn:

and I inspect the WordState 0bject to look at it. (I could of course have used the regular class browser if I hadn't been stubborn).


SRE Object Inspector on: [3514] : WorldState

```
lastStepTime (WorldState)
multiCanvas (WorldState)
remoteServer (WorldState)
stepList (WorldState)
viewBox (WorldState)
```

```
[1042]a Heap(StepMessage(#stepAt: -> [999]a
PasteUpMorph<world> [world])([999]a
PasteUpMorph<world> [world] #stepAt: 13209756)
StepMessage(#stepAt: -> [2632]an
AConnectorSRE)([2632]an AConnectorSRE #stepAt:
```

```
adjustWakeupTimes: (WorldState)
adjustWakeupTimesIfNecessary (WorldState)
cleanseStepListForWorld: (WorldState)
isStepping: (WorldState)
isStepping:selector: (WorldState)
listOfSteppingMorphs (WorldState)
runLocalStepMethodsIn: (WorldState)
runStepMethodsIn: (WorldState)
startStepping:at:selector:arguments:stepTime: (Worl
stopStepping: (WorldState)
stopStepping:selector: (WorldState)
```

```
runLocalStepMethodsIn: aWorld
	"Run morph 'step' methods (LOCAL TO THIS WORLD)
whose time has come. Purge any morphs that are no
longer in this world.
	ar 3/13/1999: Remove buggy morphs from the step list
so that they don't raise repeated errors."

	| now morphToStep stepTime priorWorld |
	now := Time millisecondClockValue.
	priorWorld := ActiveWorld.
	ActiveWorld := aWorld.
	self triggerAlarmsBefore: now.
	stepList isEmpty
		ifTrue:
			[ActiveWorld := priorWorld.
			^self].
	(now < lastStepTime or: [now - lastStepTime > 5000])
		ifTrue: [self adjustWakeupTimes: now].	"clock
slipped"
	[stepList isEmpty not and: [stepList first scheduledTime
< now]]
		whileTrue:
			[lastStepMessage := stepList removeFirst.
			morphToStep := lastStepMessage receiver.
			(morphToStep shouldGetStepsFrom: aWorld)
				ifTrue:
					[lastStepMessage value: now.
					lastStepMessage ifNotNil:
							[stepTime := lastStepMessage
stepTime ifNil: [morphToStep stepTime].
							lastStepMessage
scheduledTime: now + (stepTime max: 1).
							stepList add:
lastStepMessage]].
			lastStepMessage := nil].
	lastStepTime := now.
	ActiveWorld := priorWorld
```

```
WorldState
Object
ProtoObject
```

```
canvas
hands
initialization
object fileIn
objects from disk
stepping
undo
undo support
update cycle
```

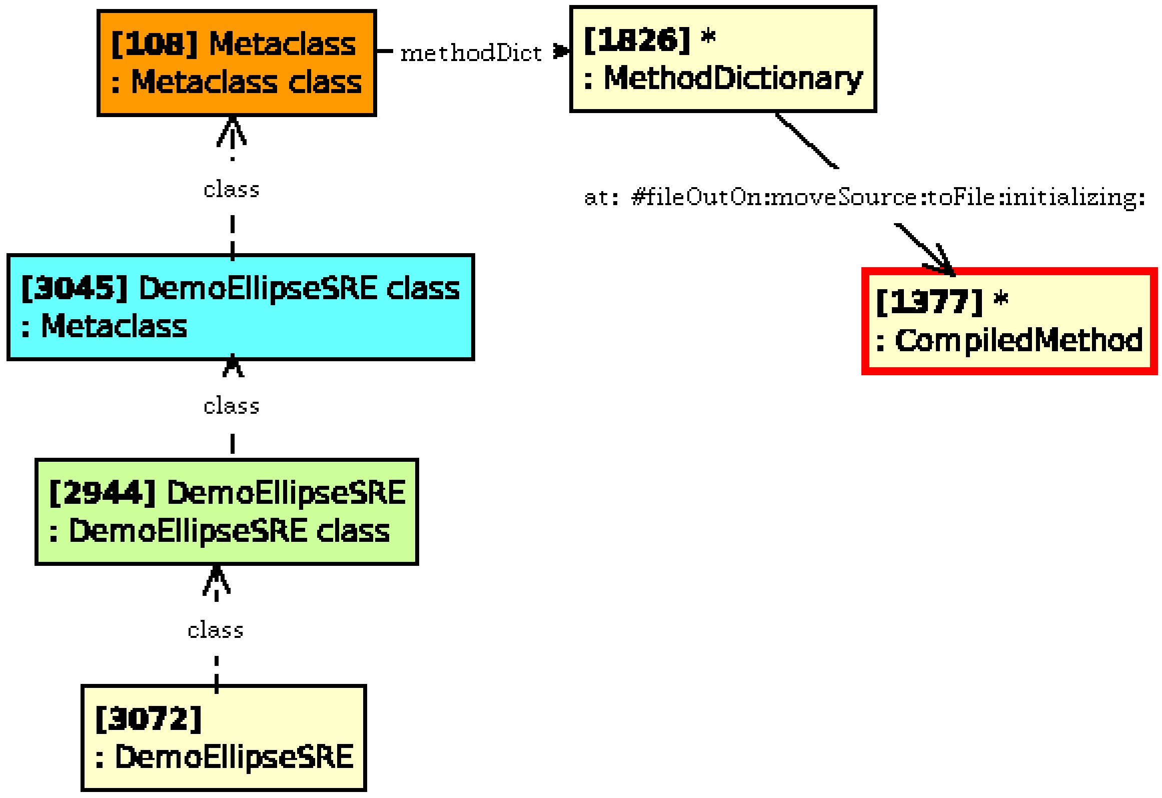
## *Projection 4: The Instantiation Hierarchy*

I spent some time locating the fileOut method that create *.st-files. I started with the regular class browser and selected the DemoEllipseSRE class. No fileout selector in the instance or the class. The hierarchy and protocol browsers were the same; no fileout. I look at various fileOut methods and find that the real work is done in

Metaclass>>fileOutOn: aFileStream moveSource: moveSource toFile: fileIndex initializing: aBool

The methods of an object are listed in the methodDict of its class; here [108]Metaclass. I start with the demo object and follow its instance-of-links[7] until I find a class object that has my method in its methodDict:

Trygve   29 July 2018 at 11:49:04 am. File: SRE context browser[2455][3072]a Dem

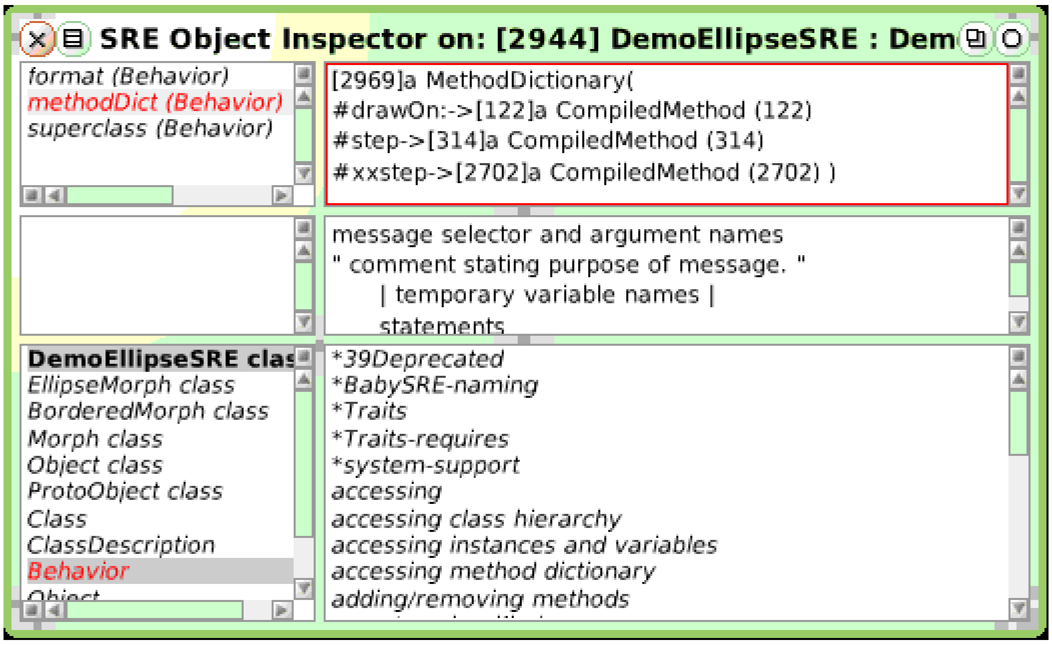


BabyUML SRE context browser[2455][3072]a DemoEllipseSRE-fileout from: 7179-ba

I check my thinking by inspecting the relevant objects. First the class, [2944]:DemoEllipseSRE. The methodDict is named in superclass Behavior, so I the

[7] The *instance-of* link is shown as a dashed line in the diagram. It is named *class* since an object is an *instance-of* a class.

filter for this superclass. The methodDict has the expected entries for the DemoElipseSRE; drawOn:, step, and the experimental xxstep.


SRE Object Inspector on: [2944] DemoEllipseSRE : Dem

```
format (Behavior)
methodDict (Behavior)
superclass (Behavior)
```

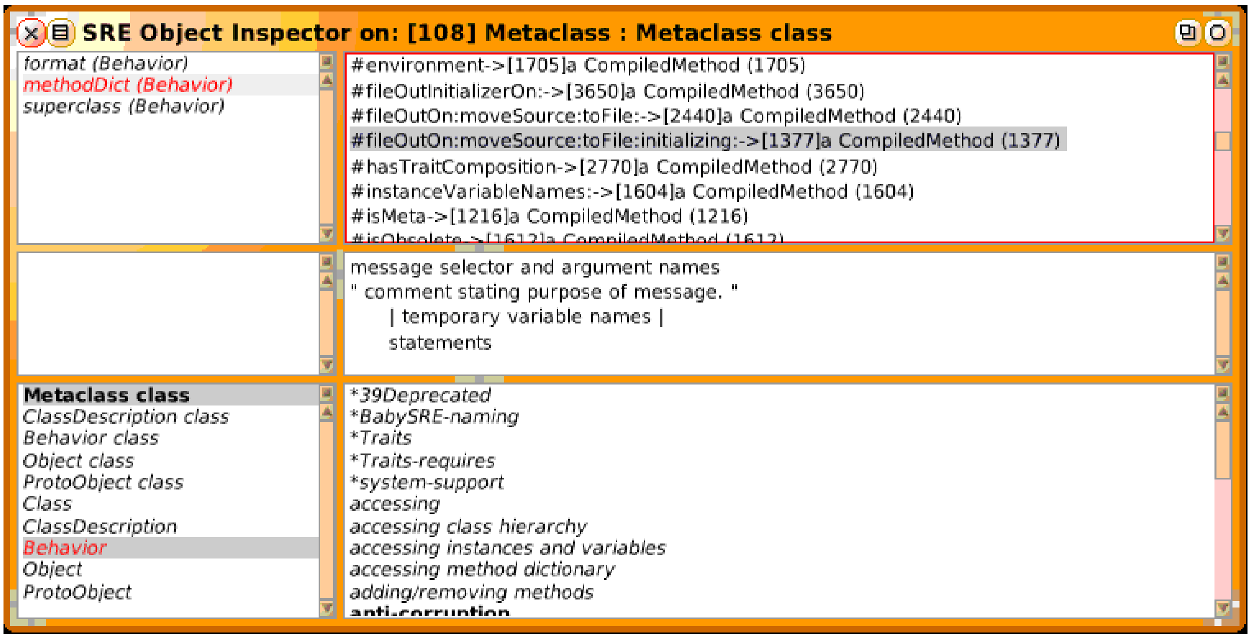
```
[2969]a MethodDictionary(
#drawOn:->[122]a CompiledMethod (122)
#step->[314]a CompiledMethod (314)
#xxstep->[2702]a CompiledMethod (2702) )
```

```
message selector and argument names
" comment stating purpose of message. "
      | temporary variable names |
      statements
```

```
DemoEllipseSRE clas
EllipseMorph class
BorderedMorph class
Morph class
Object class
ProtoObject class
Class
ClassDescription
Behavior
Object
```

```
*39Deprecated
*BabySRE-naming
*Traits
*Traits-requires
*system-support
accessing
accessing class hierarchy
accessing instances and variables
accessing method dictionary
adding/removing methods
```

Note that this methodDict is a variable in the [2944]DemoEllipseSRE object even if it is declared in class Behavior. Similarly, [3045]DemoEllipseSRE class does understand fileOutOn:moveSource:toFile:initializing: and its methodDict should be a variable in [108]Metaclass:Metaqclass class, which it is:


SRE Object Inspector on: [108] Metaclass : Metaclass class

```
format (Behavior)
methodDict (Behavior)
superclass (Behavior)
```

```
#environment->[1705]a CompiledMethod (1705)
#fileOutInitializerOn:->[3650]a CompiledMethod (3650)
#fileOutOn:moveSource:toFile:->[2440]a CompiledMethod (2440)
#fileOutOn:moveSource:toFile:initializing:->[1377]a CompiledMethod (1377)
#hasTraitComposition->[2770]a CompiledMethod (2770)
#instanceVariableNames:->[1604]a CompiledMethod (1604)
#isMeta->[1216]a CompiledMethod (1216)
#isObsolete->[1612]a CompiledMethod (1612)
```

```
message selector and argument names
" comment stating purpose of message. "
      | temporary variable names |
      statements
```

```
Metaclass class
ClassDescription class
Behavior class
Object class
ProtoObject class
Class
ClassDescription
Behavior
Object
ProtoObject
```

```
*39Deprecated
*BabySRE-naming
*Traits
*Traits-requires
*system-support
accessing
accessing class hierarchy
accessing instances and variables
accessing method dictionary
adding/removing methods
anti-corruption
```

I see what confused me. The class browser works with the object's methods in the methodDict of its class. When I browse the instance side of DemoEllipseSRE, I edit the methods of the methodDict in its class, DemoEllipseSRE class; [2944]DemoEllipseSRE :DemoEllipseSRE class.

When I browse the class side, I move one step up and edit the methodDict of [3045]DemoEllipseSRE class :Metaclass.

But the fileout messages are understood by this class object and its methodDict is in [108]Metaclass :Metaclass class. My mental model was one off in the instantiation hierarchy.

You may find this complex and it is. I don't know how I should have grasped these relationships without using the Context Browser. But this is where Smalltalk finds much of its power. It is up to the system programmers to build on Smalltalk's leverage to create powerful tools that protect the programmer from this complexity.

This example illustrates that the instantiation hierarchy is fixed and essential for the semantics of Squeak programs. The class inheritance hierarchy is different. We can rearrange it in any way we like without changing the program's semantics. It is called refactoring and the class hierarchy is in the nature of a comment.

Never confuse *instantiation* with *subclassing*.
The first is essential, the second is in the nature of a comment.
Both structures coexist in the universe of objects.

## Further Work

Hannes. Perhaps you would like to add a section with something like "Hannes Hirzel plan/want to …"?

## Conclusion

The object diagrams and stack dumps shown here are all created with the help of the BabySRE tools. They augment the traditional class browsers and inspectors with object browsers and inspectors. Where most Squeak programming tools apply to the compile-time classes, the SRI tools apply to the run-time objects.

The SRI tools are useful when we need to understand and document an existing system. The SRE focus on the run time is reflected in objects no longer being anonymous but named with their unique oop and no longer being stand-alone but being nodes in a structure of collaborating objects. The examples given here illustrate the value of understanding on what actually happens at runtime. I have on many occasions found that the BabySRE tools help me better understand programs written by myself and by others.

The SRE Object Inspector can also be used to modify the system; its user can change the values of instance variables and define methods in the object's immediate class. (Modifications of the superclasses have been blocked because a local change can have system-wide repercussions). The development of the SRE tools is part of the *DCI* project where I look for higher level constructs for object oriented programming (as opposed to class oriented programming) [DCI].

It is tempting to take SRE a step further. SRE Context diagrams resemble DCI Context diagrams and it is tempting to explore if they could be merged so that SRE becomes a tool for both reverse and forward engineering. This idea points to a merger of the SRI tools and the DCI programming tools. Comment to Hannes: This para should be changed to reflect your thoughts,

Dynabook, Kay77. Design of everyday things, Norman88

## Acknowledgements

Many thanks to Ned Konz for the Connectors package. Also for his permission to copy/rename the classes I use from this package so that the evolution of BabySRE becomes independent of the evolution of the Connectors package.

Sincere thanks to Milan Zimmermann and Chris Muller for excellent and very useful suggestions that have been realized in the second version of BabySRE.

## References

[Blue book] Goldberg and Robson: Smalltalk-80. The language and implementation. Addison-Wesley 1983. ISBN 0-201-11371-6

[DCI] https://en.wikipedia.org/wiki/Data,_context_and_interaction. and the DCI home page http://fulloo.info/.

[Kay77] Kay, Alan, *Microelectronics and the Personal Computer*, Scientific American, September 1977, p. 244.

[Norman88] Donald A. Norman: *The Design of Everyday Things*. Doubleday, 1988

[draft Reenskaug2018] Trygve Reenskaug: Personal Programming. Draft article
http://heim.ifi.uio.no/~trygver/themes/Personal/PP-005 - Draft (16).docx

## Appendix 1: Installation

The first version of SRE was programmed and used in
Windows 7>>Squeak3.10.2. The SRE tools and examples are in this image:

> http://folk.uio.no/trygver/assets/BabyIDE-2018.07.31.ZIP

The image also includes the Personal Programming demo; *Ellen's smart alarm clock*. Expand *Resources* and *BBa11PP*. You now see the DCI *Interaction* projection of the clock program. The personal programmer, Ellen, selects and moves objects from her Resources into her diagram and links them up to create her object substructure of the program (The rest is either predefined or created automatically). She then augments the objects with the required behavior (called role scripts here).

Hannes Hirzel has ported the SREObjectInspector to Squeak 5.2 beta:

> On 30.10.2018 15:02, H. Hirzel wrote:
>
> > A SqueakMap entry means that if you choose the menu 'Apps' -> 'SqueakMap catalog' -> 'Update button' then you get an 'ObjectInspector' entry which allows to install the most recent version into Squeak 5.2.
>
> To use it do
>
> SREObjectInspector inspect: myObject

## Appendix 2: How I created the domain collaboration

1. I pointed to the ellipse and pressed the left button with the ALT-key (Windows) down to open the halo. I opened the debug menu in the white button on the right hand edge. I selected SRE Context Browser to open a contextdiagram on the morph.
2. I placed the object [3072] :DemoEllipseSRE in the diagram.
3. I right-clicked on the diagram background (the 'playfield') and selected the menu item change title... . I typed Domain Collaboration.
4. I right-clicked on the role symbol marked [3072] :DemoEllipseSRE and selected the menu item add link for variable.... This gave a new menu. I selected the owner item. I placed the resulting [999] world.
5. Right-clicking [999] world, I added the link to submorphs and placed [3394]*:Array.
6. Right-clicking [3394]*:Array, I added the link to [3072].
7. Again right-clicking [999]world, I added link to extension, then otherProperties, #borderStyle,
8. Right-clicking [999]world, I selected add role from expression., typed 'self project' and placed [2604]:Project. I then linked this project back to [999]world.
9. I right-clicked [3523]: World State, selected add role from expression, typed

   ```
   stepList detect: [:sm | sm receiver asOop = 3072]
   ```

I then placed [518]: StepMessage. (I had to look around a bit to discover this statement. An alternative way for finding this object is given in the body of this note.)

10. I right-clicked [518]: StepMessage, selected add link for variable and chose receiver. This created

    the link from stepMessage to [3072]: DemoEllipseSRE.

## About the Author

**Trygve Reenskaug** , prof.em.,
Department of informatics, University of Oslo, Norway.
mailto: trygver <at> ifi.uio.no
http://heim.ifi.uio.no/~trygver
DCI home: http://fulloo.info/