Advanced Object Explorer for the Lively Kernel

Alexander Lüders, Richard Metzler, Kai Schlichting

Hasso-Plattner-Institut
Universität Potsdam
http://www.hpi.uni-potsdam.de/

Abstract. Building new applications of any kind requires standard development tools, e.g. debuggers or object explorers. The Lively Kernel, an experimental web programming environment written in JavaScript and running in the web browser, lacks some important tools that are crucial to developers.
To fill this gap we implemented an object explorer for Lively that empowers developers browsing object hierarchies and inspecting objects. This paper primarily presents and discusses JavaScript metaprogramming features and techniques to dynamically introspect and intercede objects on runtime. A design for an object explorer is proposed that utilizes these language features.

1 Introduction

Lively was originally developed by Sun Microsystems Laboratories and is now available as MIT licensed open source software [3]. It is heavily influenced by Squeak, supporting both desktop-style applications with rich graphics and the ability to modify these applications on the fly by functioning as an integrated development environment (IDE). In Lively every graphical manipulable object is a morph, the most top-level morphs are called worlds. Morphs have properties that define their look, behavior and relations with other objects and are part of a morph hierarchy.

While Lively provides a quite acceptable number of development tools for working on Lively based applications, developers miss the ability to browse object hierarchies and the morph/ submorph relations in particular. For this reason, an object explorer should be able to display these relations and the resulting hierarchy. The immediate reflection of changes would be essential to the usefulness of the specified tool as a single snapshot would be outdated very fast. Also it should be possible that an object explorer instance can explore itself (self referenced browsing).

On the basis of these requirements, this paper discusses an object explorer design and implementation with the help of JavaScript's metaprogramming features: In section 2, we provide a short introduction of JavaScript objects and inheritance and show the differences in Lively. The following section describes the architecture of the object explorer while sections 4 and 5 show how introspection and intercession techniques can be used to enable object browsing and live

updates. Benchmarks concerning memory and performance are given in section 6. The last section summarizes and concludes our work.

2 Metaprogramming in Javascript

Despite its name JavaScript is fundamental different from most other object-oriented programming languages including Java. The major differences in JavaScript are prototypal inheritance and the dynamic nature of objects that can be modified on runtime despite the lack of a meta object protocol.

As seen in listing 1.1 JavaScript objects are associative arrays that consist of key/value pairs. Functions can be added to an object and called in the context of that object. Changing the behaviour of objects at runtime is a feature that would require an explicit meta object protocol in most of today's programming languages.

```
var book = { "title" : "Faust", "author" : "Goethe" };
book.wrote = function() {
        return this.author + " wrote " + this.title;}
keys(book); // ["title", "author", "wrote"]
```

Listing 1.1. JavaScript objects are associative arrays

In JavaScript functions are objects too. Functions may have attributes and can be modified on runtime. Functions can also be used as constructors creating new objects. Objects created through constructors inherit all properties from the constructor's *prototype*. [2] By adding new functions to the prototype it is possible to add functionality to existing objects as shown in listing 1.2.

```
function Book(author, title) {
  this.author = author; this.title = title; };
var hamlet = new Book("Shakespeare", "Hamlet");

Book.prototype.wrote = book.wrote; // reuse function
hamlet.wrote(); // "Shakespeare wrote Hamlet"
```

Listing 1.2. Constructors and prototypal inheritance

Lively additionally adds a more established style of inheritance - the class based inheritance. It is introduced by means of metaprogramming techniques. A small example demonstrates the feature in listing 1.3.

```
Object.subclass("Book", {
  initialize: function(author, title){
    this.author = author; this.title = title;
  },
  wrote: function(){...}
```

```
});
Book.subclass("DrawingBook", {});
var bookForChildren = new DrawingBook();
```

Listing 1.3. Lively's class based inheritance

3 Object Explorer Overview

The object explorer is implemented as a morph and thus easy to integrate into the Lively Kernel. By simple right-clicking on a morph and selecting *explore*, any morph can be browsed in the object explorer. It displays the properties of the morph object in a tree view. Depending on the type of the property the information displayed varies. Underneath the tree view there is a console view, which is used to display or manipulate a function's source. Besides the manipulation of objects, the object explorer provides a few more options accessible through the side menu in the upper right corner. Available options are expanding/ collapsing all items, enabling/ disabling the morph view and displaying/ hiding functions. Some of the features may be only accessible in morph view respectively the normal view.

3.1 Architecture

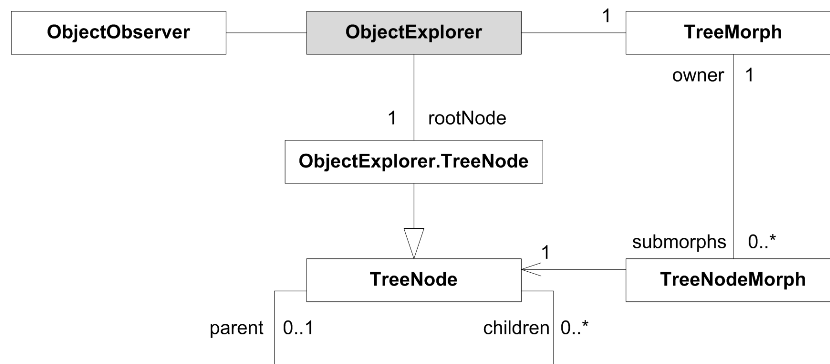

Fig. 1. UML class diagram of relevant object explorer classes

For separation of concerns we decided to apply the Model-View-Controller design pattern. The `ObjectExplorer` class can be used to create instances of the explorer and behaves in the MVC context as the *controller*: It builds up the tree of objects (*model*) and create the interface components to display this tree (*view*).

The view mainly consists of a tree view which is represented by the `TreeMorph` component. It contains several morphs for displaying the single nodes. Therefore the `TreeNodeMorph` is used which itself is composed of several other morphs again (not displayed in the diagram). The model part is represented by `ObjectExplorer.TreeNode` derived by `TreeNode`. The latter describes standard node functions like `expand`, `collapse`, `setText`, `getText` etc. The former however fulfills the model functionality, as it sets the text of the node according to the observed property.

Last but not least there remains the `ObjectObserver` component, which is responsible for observing an property by providing operations for registering/deregistering callbacks for a property.

4 Exploring Objects (Introspection)

In the following sections the aspects of displaying and manipulating properties will be covered. For each aspect the idea of the underlying JavaScript implementation will be discussed briefly.

4.1 Displaying properties

In order to display an object's properties, it is essential to get a list of all the properties the explored object holds. Therefor introspection is used. As already mentioned in section 1.1 JavaScript objects are associative arrays. Capitalizing this characteristic it is possible to receive the list of properties.

```
var printObjectProperties = function(obj){
  for(var property in obj){
    //show function source
    if (typeof(obj[property]) === 'function')
      alert(obj[property]);
  }
}
```

Listing 1.4. Iterating over JavaScript object's properties

In listing 1.4 the variable `property` holds the name of the current property. This can be used to access the property itself. In the object explorer the property name along with a literal description of the property's value is displayed. The description depends on the actual type of the property.

Using the keyword **typeof** we can determine the type which works as a discriminator for building the description string. It returns one of the following types: string, function, object, number, boolean, undefined. In the case of an object the description should look similar to Java's toString method which typically includes the name of the object and the properties with their respective values. In order to achieve that we use the `Object.inspect` method. Internally it calls the inspect method of the passed object if one exists. Anyway, for most Lively Objects this should be the case. Otherwise it calls the toString of the

object. It turns out that the Object.inspect fits our need to provide a useful literal description.

Displaying function source In more advanced usage scenarios it could be useful to examine the source code of some functions of an object. Therefor we introduced a console widget which is displayed underneath the tree view. By clicking on a function in the tree view, the source is displayed in the console. The underlying JavaScript implementation is fairly easy - `(new Function(){alert('foo');}).toString();`.

Browsing morph hierarchy Assume you want to explore the graphical Lively world. It consists of several morphs which themselve are composed of several other morphs. Internally a Lively morph has an array `submorphs` which will contain the morphs it is composed of. Of course, these submorphs may consist of several submorphs again. In order to get a quick overview of a Lively world, it is annoying to click through the entire `submorphs` array over and over. Hence we introduced a domain specific view - the morph hierarchy.

In this view all properties except Lively morphs are hidden. The implementation basically filters all non-morph objects using the Lively introspection.

4.2 Manipulating properties

Besides the pure functionality of displaying properties, there may be an interest in manipulating them too. Manipulation of properties is done using the object explorer's console. Here the `this` keyword refers to the explored object, e.g. `this.documentation = "A useless documentation."` will change the `documentation` property. To execute the statement, right-click it, select *text-functions* and afterwards *do-it*.

It turned out that manipulating objects is quite a useful feature, even during the development time of the object explorer itself. In order to provide such a feature we reused existing Lively functionality. The console is a Lively TextMorph which supports the evaluation of entered text. It is often used to execute selected text, which is refered to as *Do-It functionality*. Internally the provided text is executed using JavaScript's `eval` function. Thrown exceptions are displayed in an TextMorph overlay. However, to manipulate the explored object, a binding of the `this` keyword to the object itself was needed. A TextMorph defines a DoItContext which is set to our explored object (see listing 1.5).

```
panel.textPane.innerMorph().connectModel({
  model: {contextForEval: function(){return this.
      objectToExplore}.bind(this)},
  getDoitContext: 'contextForEval'
});
```

Listing 1.5. Setting Do-It context

As functions are object properties, it is reasonable to make them changeable too. Therefor select a function and change the source accordingly. Afterwards right click the source and select *save function source*. The internal mechanism is similar to the manipulation of properties as described above. Solely the evaluated text has to be prefixed with the object path relative to the explored object, e.g.`this.fullbounds.intersects = <console_source>`. The prefixing is invisible to the user and is applied after the user selects *save function source*.

5 Live Updates (Intercession)

The previous section described how to display and browse a snapshot of object hierarchies. To display the actual, not out-dated state of an object, the object explorer has to look for changes on objects. Since setting properties of an object is not usually done via a setter function, the challenge here is to hook into an object so that changes can be recorded. For this purpose, this section discusses JavaScript intercession techniques that could be used to achieve this goal.

5.1 Approaches

Basically, we want to achieve implementing an observer that can be used to watch a property of an object and that runs a custom handler when this occurs. Listing 1.6 shows an example how this object observer could be used while in the following different implementation approaches will be discussed.

```
var object = { myProp: null };
var o = new ObjectObserver(object, 'myProp', function(newV,
     oldV){
  alert('changed from ' + oldV + ' to ' + newV);
});

object.foo = 'bar'; //=> 'changed from null to bar'
o.unregister(); // no further changes should be recorded
```

Listing 1.6. Exemplary usage of an object observer

Object.watch Mozilla proposed a JavaScript extension `object.watch(prop, handler)`which enables observing a property of an object [4]. Listing 1.7 shows an example how the ObjectObserver could be implemented using this extension. Unfortunately, other browsers than Mozilla Firefox doesn't support this function for which reason this is not suitable in our case.

```
Object.subclass('ObjectObserver', {
  initialize: function(obj, prop, fn){
    obj.watch(prop, fn);
```

```
  }
});
```

Listing 1.7. Implementing ObjectObserver with Object.watch

Polling for Changes Additionally, the object observer could poll frequently for updates on the property of an object (see listing 1.8). Therefore, a function checks regularly if the current value has changed and, if there are changes, calls the handler. For implementation we used Prototype's `Function.prototype.delay` method that enables calling a function with a defined delay (so it is more convenient way than calling JavaScript's `window.setTimeout`). An obvious drawback of this approach is the unavoidable delay between changing a property and being notified about it.

```
Object.subclass('ObjectObserver', {
  initialize: function(obj, prop, fn){
    var oldValue = obj[prop];
    var check = function(){
      if(oldValue !== obj[prop]){
        fn(obj[prop], oldValue);
        oldValue = obj[prop];
      }
      check.delay(1);
    };
    check();
  }
});
```

Listing 1.8. Implementing ObjectObserver with Polling

Getter & Setter A quite unknown JavaScript feature are setter and getter functions [5]: Those can be defined for any object and are dealing with getting and setting the value of an object's property. Listing 1.9 demonstrates how to profit from this in the context of observing properties of an object. After having stored the current value of the property in a local variable, a getter and setter function are defined to simulate the old behavior of the property while calling additionally our callback handler. The problem here is that existing getter and setter could be overwritten.

```
Object.subclass('ObjectObserver', {
  initialize: function(obj, prop, fn){
    var value = obj[prop];
    obj.__defineSetter__(prop, function(v) {
      fn(v, value);
```

```
      value = v;
    });
    obj.__defineGetter__(prop, function() {
      return value;
    });
  }
});
```

Listing 1.9. Implementing ObjectObserver with getter and setter

Wrapper As mentioned before, setter and getter of object properties can be implemented through functions. Since JavaScript functions are objects and can be stored into a variable, the object observer can just overwrite existing setter by its own implementation that invokes the old setter. This way, the old behavior is conserved but the callback handler can be called. Listing 1.10 shows an accordant implementation and even provides an unregister function to restore the previous state (this essential method has already been used in listing 1.6).

```
Object.subclass('ObjectObserver', {
  initialize: function(obj, prop, fn){
    [...] // Define setter and getter, if not available
    var setter = obj.__lookupSetter__(prop);
    var wrappedSetter = function(val){
      fn(val, obj[prop]);
      setter.call(obj, val);
    });
    obj.__defineSetter__(prop, wrappedSetter);
    this.unregister = function(){
      obj.__defineSetter__(prop, setter);
    }
  }
});
```

Listing 1.10. Implementing ObjectObserver with wrapped getter and setter

5.2 Implementation Considerations

We now have a handful of techniques which allow to observe objects. Although the approach using wrapped setters seems to be the most clean one, it has some limits for arrays where setters are not called when accessing indexes (see listing 1.11). Additionally, in JavaScript arbitrary properties can be added (and deleted) to an object at any time just by giving it a value what makes it difficult to track those changes. To solve these issues, we implemented the object observer with the following combination of techniques:

- *Changed object properties*: For each observed property, a getter and setter is defined, if not available. Afterwards, the setter is wrapped to call the handler.
- *Added and deleted object properties*: For each observed object, a scheduled function polls for changes in the list of properties. Handlers are (un)registered if a property is identified as added or deleted.
- *Added, changed and deleted array indexes*: For each observed array, polling functions are used as described before, but they are additionally looking for changes on array indexes.

Because of this, our object observer interface supports registering rather on object than on property level. Additionally, the `ObjectObserver` class is a singleton to be able to manage registered callbacks and wrapped setters so that our final interface is `ObjectObserver.register(obj, fn)` instead of `new ObjectObserver(obj, prop, fn)`).

```
var a = [1];
a.__defineGetter__('0', function(){return 'foo'});
a[0]; // Getter does work #=> 'foo'
a.__defineSetter__('0', function(){alert('changed')});
a[0] = 1; // Setter does not work, no alert :(
```

Listing 1.11. Setters on array indexes have no effect

5.3 Self Referenced Browsing

One requirement of the object explorer is to be able to explore itself. With live updates, this could lead to infinite recursion when exploring itself: While the explorer is drawing its internal structure, this causes updates on it which requires another redraw (and so on). To handle this, we implemented a switch to detect when the explorer browses itself and just disables live updates for that specific part.

Another candidate for infinite recursion could be having two reciprocal browsing explorers. Against our expectations, this scenario couldn't be constructed by reason of Lively removes and re-adds morphs when bringing an inactive object explorer to front. This way, all the expanded object explorer components are collapsed when clicking on another object explorer which makes it impossible to produce infinite recursion. Once the implementation in Lively has been changed, this could be solved with a similar switch as for self referenced browsing.

6 Performance and Memory Analysis

We profiled our application to get an impression of how it affects the performance and memory consumption. In our first benchmark (table 1 and `lively.Tests.ObjectExplorerBenchmark` respectively) we analyzed the performance of our object observer implementation: With a variable number of iterations, properties

of objects are set to different values see the impact of explicitly defined setter (b) and of an installed object observer (c); (a) stands for the situation without any explicit manipulation. Apparently, the usage of setters have little impact. Since the object observer installs additional wrapper, invokes change handler and polls frequently, using the object observer slows down setting properties to a factor of three (what is still acceptable).

	number of iterations			
	1.000	10.000	100.000	500.000
(a) "usual way"	0 ms	6 ms	80 ms	500 ms
(b) getter & setter	0 ms	9 ms	100 ms	740 ms
(c) object observer	1 ms	15 ms	170 ms	1600 ms

Table 1. The impact of manipulating objects with setter and installing the object observer

The other benchmark (table 2) concentrates on the performance and memory impacts of the whole object explorer. In general, we compared the object explorer in the example Lively application, with disabled and enabled live mode. We took the time for one rotation of the example engine widget as performance measure, for memory consumption Google Chrome's task manager has been used. As it can be seen in the table, the object explorer has high performance and memory impacts when many properties/ objects are displayed (case(c) and (e)). Especially when observing many submorph arrays (e), the system is extremely slowed down because arrays have to be observed through polling. But the more general use cases (b) and (d) reveals that the application is still very responsive.

	Engine Rotation			Memory Consumption		
	Disabled	Enabled	Factor	Disabled	Enabled	Factor
(a) without object explorer	1,65 s	1,65 s	1,00	101,0 Mb	101,0 Mb	1,00
(b) standard view	1,70 s	1,85 s	1,08	112,0 Mb	113,8 Mb	1,01
(c) all functions view	2,00 s	3,05 s	1,52	134,6 Mb	140,5 Mb	1,04
(d) morph view	1,65 s	1,70 s	1,03	102,6 Mb	107,7 Mb	1,04
(e) morph view (expanded)	4,00 s	17,55 s	4,38	137,7 Mb	258,5 Mb	1,87

Table 2. Benchmarks for exploring example world (with different views and enabled/ disabled live mode)

7 Summary & Outlook

Many functionalities that would require a certain metaobject protocol in other programming languages are quite common in JavaScript. Metaprogramming (like adding new functions to objects) is directly built into the language and is used by developers regularly without even knowing it. Still there are limitations on what is possible in JavaScript. One major flaw we found were restricted intercession possibilities for arrays. It is also difficult to recognize new properties of objects and client code is required to poll for changes. Additionally, the lack of consistent cross-browser support is quite challenging, a challenge that ”traditional” JavaScript web developers also have to face. While testing our implementations we recognized that the performance of the various JavaScript implementations vary greatly (even between Safari and Google Chrome).

Despite of the challenges mentioned above, we conclude that our object explorer implementation fulfills the requirements from the introduction. For future work, the following topics should be considered: First the object observer should be aware of the fact that wrapped properties may be wrapped by a different meta program again. The current implementation would not be able to uninstall the wrapper in that case. Fortunately there are existing solutions out there [1]. Another performance improvement would result by drawing the svg elements for the tree view nodes directly rather then using Lively morphs. A large number of morphs drops the performance significantly, especially in case of moving the object explorer around.

8 Appendix

8.1 HowTo Install/ Setup

When deploying to an already existing Lively project, the files mentioned in table 3 should be placed in the Lively source folder. In addition the files mentioned in table 4 should replace the original Lively files in the source folder. To start Lively with object explorer enabled simply load `objectExplorer.xhtml` in your browser (latest Google Chrome or Safari preferred).

File	Function
objectExplorer.xhtml	The xhtml (with test) links to all js files
ObjectExplorer/ObjectExplorer.js	The main application class
ObjectExplorer/ObjectExplorerTreeNode.js	A tree node representing an object to explore
ObjectExplorer/ObjectObserver.js	An Observer that registers callbacks
ObjectExplorer/Tree.js	Logic for addind nodes to a tree
ObjectExplorer/TreeMorph.js	A morph that displays a tree
ObjectExplorer/Utils.js	Some functions we needed but did not fit anywhere else
ObjectExplorer/collapsed.png	The icon for collapsed treenode
ObjectExplorer/expanded.png	The icon for expanded treenode
ObjectExplorer/Tests/ObjectExplorerTest.js	Tests for observer
ObjectExplorer/Tests/TreeTest.js	Tests for treenode implementation

Table 3. Files for Object Explorer

File	Line	Changes
Core.js	#2790	added 'explore' to context menu in order to start Object Explorer
Core.js	#4359	added 'explore' to morph menu

Table 4. Changed Files

References

1. John Brant, Brian Foote, Ralph E. Johnson, and Donald Roberts. Wrappers to the rescue. In *Proceedings ECOOP 98, volume 1445 of LNCS*, pages 396–417. Springer-Verlag, 1998.
2. Douglas Crockford. Prototypal inheritance in javascript. `http://javascript.crockford.com/prototypal.html`, 2006-2008.
3. Sun Labs and Hasso-Plattner-Institute. Lively kernel project page. `http://www.lively-kernel.org`, 2010.
4. Mozilla. Core javascript 1.5 reference - object.watch. `https://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Global_Objects/Object/watch`, 2008.
5. Allen Wirfs-Brock. *Proposed ECMAScript 3.1 Static Object Functions: Use Cases and Rationale*. Microsoft Corporation, 2008. `http://wiki.ecmascript.org/lib/exe/fetch.php?id=es3.1%3Aes3.1_proposal_working_draft&cache=cache&media=es3.1:rationale_for_es3_1_static_object_methodsaug26.pdf`.