Integrating JavaScript Language Features into Smalltalk

Anton Gulenko

Hasso Plattner Institute, University of Potsdam, Germany
anton.gulenko@student.hpi.uni-potsdam.de

Abstract. The preferred way to develop web applications is to use a framework and a single programming language. Client-side functionality needs to be expressed in JavaScript, which is the widest supported scripting language of web browsers. In order to develop the client-side of the application in the same language as the server, it is necessary to expose the language features and libraries of JavaScript in this *host* language. This has to be done on a syntactical, as well as semantical level to use the full extent of JavaScript without writing or generating wrappers for libraries.
We chose Smalltalk as host language and map JavaScript concepts and features to equivalent or similar Smalltalk-counterparts. This way we expose the full client-side functionality in Smalltalk code. The presented ideas are implemented in Orca, a framework for web application development. We evaluate our solution and find it to produce expressive and comprehensive code while offering the full extent of JavaScript's client-side functionality.

Keywords: Smalltalk, JavaScript, programming languages, mapping language features

1 Introduction

Developing a web application includes development of the client-side, as well as the server-side. Both sides have to provide full functionality and include important parts of an application. The client-side tries to deliver a rich and visually appealing experience for the user, while the server-side of the application can handle many other aspects such as persistence of application data, business logic or management tasks between several clients.

Both sides need to be programmed in an appropriate programming language with full expresiveness for all tasks respectively. While the choice for the client-side functionality is mostly tied to JavaScript[8], the server-side language can be chosen freely as long as appropriate server functionality is provided.

Without a solution to express the client-side functionality in the server-side language, developers have to program web applications using two languages. Using two languages can be less productive for several reasons. First of all, developers have to learn both languages. This includes the syntax, the standard

library and the developer tools. Maintaining this knowledge can be a difficult task. Using multiple technologies also tends to produce expert knowledge and split the team into groups. As the code base contains multiple languages, it becomes harder to maintain than a homogenous code base. From the perspective of programmers, it can be difficult to exchange data between the two languages; the programming models don't always map cleanly onto each other.

We solve the two-language problem by choosing the server-side language and writing both parts of the application in that language. We chose Squeak Smalltalk[11] for this part. In this context, Smalltalk is called the *host language*. This results in a single code base for the whole application, better maintanability and a consistent tool chain for the whole development team. Also, each team member can take part in developing the whole system, instead of just a part of it.

Implementing this concept appropriately requires at least the following functionality. The client-side code has to be generated from the code written in the host language. Then a special runtime environment has to be supplied on the client; it has to provide all required means to execute the generated code. The host language has to support full client-side functionality.

This paper gives a solution for the last requirement by defining concepts to access JavaScript functionality in the Smalltalk programming language.

The mapping of language features is not straight forward. JavaScript and Smalltalk have some major differences; for exmaple, they are based on entirely different programming models and standard libraries. The main problem to overcome is the fact, that JavaScript has a prototype-oriented inheritance model and objects with dynamic properties called *slots*, while Smalltalk uses a strict class based inheritance model. These and more differences are discussed in Section 2.

We have found the host language to be sufficient to express all necessary client-side aspects.

We did not extend the host language by introducing a new compiler, as this results in a *new* language mixing the two grammars, syntaxes and semantics. The resulting language is hard to understand and does not eliminate the need to learn both intermixed languages. Our approach is to limit required languages to one well-known language with non-verbose syntax and an expressive programming model.

For the solution presented by this paper, we map important JavaScript concepts to the Smalltalk language, including it's grammar, syntax and semantics. The basic way to do this is to find counterparts in Smalltalk, that can be used to express certain aspects of JavaScript.

Not every aspect has a feasible counterpart in Smalltalk. If there is no such counterpart, the unmatched aspect cannot be implemented on the server. Instead, a complete API for that functionality must be added in form of an *empty* library (also called a *library stub*). The methods of such a library document the API, but don't really implement the functionality (as it *can* only be implemented on the client). At runtime, these methods will be overridden and the functionality will be available for the translated code for the browser.

The paper is organized as follows. Section 2 discusses differences between JavaScript and Smalltalk. Section 3 describes mappings of basic JavaScript concepts, Section 4 describes mappings of more complicated and important concepts. Section 5 gives details of the implementation of the presented mappings in Orca. Section 6 describes and interprets an evaluation to show the quality of our ideas. Section 7 lists several projects we share common ideas with and that inspirated our solution. Section 8 gives a summary and a lookout for future work.

2 Differences between JavaScript and Smalltalk

To understand the problems of integrating JavaScript language features into Smalltalk, it is important to know which aspects differentiate the two languages.

As pointed out in the introduction, the two languages are based on two very distinct programming models[7] [9]. In JavaScript, each object is a complete entity for itself, while in Smalltalk each object is an instance of a class and all of it's properties are defined by the class. In JavaScript, objects inherit from other objects, while in Smalltalk classes inherit from other classes.

When comparing the appearance of the code, Smtalltalk is a much cleaner language. Syntactical elements are limited to the most basic elements needed for object orientation; JavaScript's syntax includes a big amount of keywords, operators and artificial constructs, which can complicate the code comprehensibility. Often, there can be surprising results, for example when the semantics of the `this` keyword are not clear. This hinders less-than-expert programmers to write complicated code, as unexpected behaviour arises quickly[1].

In comparison to JavaScript, Squeak Smalltalk includes a rich standard library. JavaScript's standard library is rather small; developers have to use the operators for most common tasks. To use a comfortable API for a problem like iterating over objects, one has to find, include and learn a third party library.

Due to the it's extensive standard library, Smalltalk produces cleaner and more understandable code for most tasks, even without the help of additional libraries; the example of handling and iterating collections is maybe the most prominent one (listings 1.1 and 1.2).

```
for (var index = 0; index < anArray.length; index++) {
  var element = anArray[index];
  /* do something with element */
}
```

Listing 1.1. Iterating over an array in JavaScript

```
anArray do: [ :item | " do something with element " ]
```

Listing 1.2. Iterating over an array in Smalltalk

A final aspect for comparison are the development tools. In a Smalltalk image, all tools are integrated and uniform. Many JavaScript related tools as well, but they are not well integrated and a programmer has to learn the efficient

use of each tool seperately. Among other, a Smalltalk image provides a code-browser and -editor, an object explorer, a debugger and a test runner with the same design and perfectly integrated. To get similar functionality, a JavaScript programmer has to use several programs, all with different user interfaces and user experience.

2.1 Why Smalltalk?

The thoughts of Lukas Renggli and Tudor Girba [16] reflect greatly our thoughts of why to choose Smalltalk as host language to integrate JavaScript into.

2.2 Why not JavaScript?

The Node.js framework[1] allows using JavaScript as server-side language. Thus, developers could use Javascipt to program both the client- and the server-side. The server runs a JavaScript virtual machine, which executes just the same code as the client does.

Besides selecting Smalltalk as host language, we have also chosen to *not* use JavaScript for all programming tasks of a web application. The main reason are the results of the comparison between JavaScript and Smalltalk given in this section and the thoughts of Renggli and Girba; but we also wanted to avoid JavaScript. Generated JavaScript code can escape serious potential security problems, that are injected, when developers write JavaScript by hand. A study conducted by Yue and Wang shows, that this is the regular case[18]. Martin Johns lists many more JavaScript related threats and security problems[12].

We want to show that the client-side of web applications can be programmed in a programming language other than JavaScript.

3 Mapping Basic Concepts

To enhance Smalltalk code with JavaScript concepts, we have to cover the basic parts of the language.

The concepts discussed in the subsections below are inherent in most high level object oriented programming languages, including JavaScript and Smalltalk. Still, there are many details to notice and discuss. The main difference to overcome is, that JavaScript syntax relies heavily on keywords and operators, while Smalltalk syntax features (almost) nothing more than message sends. So we have to assure, that for each basic operator in JavaScript, there is an equivalent Smalltalk method on an appropriate class.

[1] `http://www.nodejs.org/`

3.1 Control Flow

JavaScript includes many keywords for control flow, which map directly to the control flow concepts in Smalltalk. Whereas Smalltalk does not have operators for that task, but message sends, the possibilities are the same and even extended. Here is an example mapping of basic control flow keywords to the according message sends in Smalltalk (Table 1).

JavaScript keyword(s)	Smalltalk message(s)
`if`, `else`	`#ifTrue:`, `#ifFalse:`, `#ifTrue:ifFalse`, `#ifFalse:ifTrue:`
`switch`, `case`, `default`	`#switch:default:`
`for`	`#to:do:`, `#do:`, `#timesRepeat:`
`while`	`#whileTrue:`, `#whileFalse:`
`return`	↑

Table 1. Mapping of control flow keywords

The `for` operator in JavaScript can express more than provided by just the listed messages, but browsing the message protocols of for example `BlockClosure` and `Number` will give all the functionality needed for typical programming.

However, there are keywords, that are not directly matchable to message sends from the Smalltalk standard library; the most important one is `continue`. Inside any JavaScript loop, the `continue` keyword quits the current inner-most loop cycle and executes the next one. Smalltalk does not support that; continuing to the next loop-cycle is only possible by letting the loop-block execute to it's end in a controlled fashion.

There is a very important difference between the `return` operator and the ↑ of Smalltalk. The `return` operator returns from the current function, while ↑ returns from the current method, even if the current execution is inside several block contexts. The only way to quit from the current block and not from the method is to let it execute until it's end. A JavaScript programmer has to get accustomed to this difference.

3.2 Variables

Developers can declare, use and assign JavaScript variables anywhere in the program code. They can also assign them before or without declaring them and reference them before assigning anything to them. An unassigned variable has the default value `undefined`.Smalltalk is much more explicit about variables — each Smalltalk temporary variable has to be declared at the beginning of a block or method, as well as each instance variable and the other types of Smalltalk variables. Smalltalk variables have the default value `nil`.

JavaScript also has global variables. In code, they can be accessed directly, when they are not shadowed by any temporary variable in any intermediate

scope. The variables are also available via the `window` keyword, which resolves to a common object holding all of the globals.

Smalltalk supports a concept of globals, for example to access the common `Transcript` object. However to access them and have the code compile properly, they must be defined.

Although the client-side code might never be executed in the Smalltalk image, it still has to compile to allow convenient development. Therefore it's not feasible to use this mechanism for accessing JavaScript globals. This is discussed in detail in Section 4.3.

3.3 Primitive Types

JavaScript features three real primitive types: numbers, strings and booleans. At some points, these types show behaviour differing from normal objects. Part of that behaviour is not specified and depends on the browser. For example, setting a slot of a string or number object is possible, but the value of the slot will get lost when accessing the same slot again. Despite these inconsistencies, primitive values can still be enhanced with additional functionality. This is done by adding new functions to their `prototype` objects (accessed, for example, by `Number.prototype` and `String.prototype`).

Contrary to all this, Smalltalk primitive types are normal objects like any other. They can receive messages and nothing more. The primitive types of Smalltalk are sufficient to map those of JavaScript; due to the extended standard library they are even more powerful. But all the special behaviour inherent in JavaScript has to be exposed explicitely, as it is not present in Smalltalk. A programmer *might* want to explicitely access this behaviour; however, to support browser independence and reduce error potential we consider it not necessary to expose this functaionlity.

The JavaScript values `null` and `undefined` are not real primitive types, rather just primitive values. Still, they are important and need to be mapped. The Smalltalk value `nil` covers the JavaScript value `null`. But `undefined` has no such equivalent. To solve this we propose creating an empty class named, for example, `Undefined`, which evaluates to `undefined` on the client-side.

3.4 Operators

JavaScript has several groups of operators. The operators are divided in the following categories:

- Arithmetical operators
- Assignment operators
- String concatenation
- Comparison operators
- Logical operators
- Operators for bitwise manipulation

Similar to control flow (see Section 3.1), most of these operators have matching counterparts in Smalltalk. As mentioned before, the Smalltalk versions are all normal messages sent to objects. The common operators are all covered in the method protocols of `Magnitude`, `String`, `Boolean` and their subclasses, some are defined on `Object`. However, an important difference is the precedence of arithmetic operators: while JavaScript executes arithmetical operations in their correct mathematical order, Smalltalk's binary messages are left-associative and executed in the order from left to right. Still, the code `1 + 2` is valid for both Smalltalk and JavaScript.

Again, there are JavaScript operators, that cannot be mapped directly. The first one is the string concatenation. It's behaviour is hard to understand and inconsistent. It has no direct Smalltalk equivalent. Like with the primitive types, we consider an explicit mapping unnecessary, as Smalltalk has a rich library of string manipulation itself.

The other class of unmatchable operators are the assignment operators. They perform an operation of some kind on a value and write the result back into the same variable. For example `a += 2` would store the result of `a + 2` back into `a`. The Smalltalk operator for this task is `:=`, the operation must be coded explicitely (`a := a + 2`).

3.5 Exception Handling

Exception handling mechanisms can be found in both JavaScript and Smalltalk. In JavaScript, any object can be treated as an exception and thrown. When using a `try-catch` statement, the type of the exception to catch cannot be specified. When only a certain kind of exception needs to be caught, the type (and/or other properties) of the caught object must be checked explicitely; the object may be thrown again, if it is not handled by the code-block that caught it.

Squeak Smalltalk defines many messages that can be sent to instances of `BlockClosure` to constrain their execution. Only subclasses of `Exception` can be thrown and caught. This is less flexible, but just as expressive as the JavaScript approach. Custom subclasses of `Exception` or `Error` can wrap arbitrary objects.

So despite the differences, we don't need to extend the Smalltalk exception handling mechanism to map the one of JavaScript. However, we propose adding convenience methods like `#signalYourself` in listing 1.3).

```
('Critical error, current state:', self state) signalYourself
```

Listing 1.3. Convenient way to throw an exception message

4 Mapping Complicated Concepts

The concepts discussed so far where basic language concepts of JavaScript. For most of them, a matching Smalltalk equivalent can be found. The language features presented in this section include fundamental differences between JavaScript and Smalltalk and are not easily handled.

4.1 Functions

JavaScript's functions are first class objects which can be created anonymously and invoked. If a function is placed into a named slot of an object, it takes the role of a method. Functions in slots can be inherited prototypically, deleted, and copied into other slots, objects and variables.

The closest equivalent in Smalltalk to that are blocks (instances of `BlockClosure`). They are first class objects as well and can be created, stored, referenced and evaluated. However, contrary to JavaScript functions, Smalltalk blocks are still different from methods. Methods are placed inside method dictionaries of classes and changing a method of a class reflects on each instance of that class. Developers cannot use blocks to achieve similar effects. Blocks are mostly short living objects and cannot be used to replace the method of another given object.

Another difference between JavaScript functions and Smalltalk blocks is that JavaScript functions can be called with any number of parameters. Missing parameters are bound to the value `undefined`, additional parameters are ignored and stay available over the `arguments` keyword. Consequently, in JavaScript there is no way to check, how many parameters a function is actually expecting. Invoking a block with the wrong number of parameters in Smalltalk leads to an exception and blocks respond their number of parameters on the message `#numArgs`. An example using that possibility is the method `#ifNotNil:`. It evaluates the block given as argument, if the receiver object is other than `nil`. If the block takes one parameter, it is evaluated with the receiver as argument; if it does not take arguments, it is evaluated without.

Furthermore, in JavaScript the value returned after evaluating the function is the value given to the `return` operator. A Smalltalk block returns the value of the last statement contained in it. This means, leaving the block spontaneously without leaving the surrounding method is not possible.

These differences would not really affect using Smalltalk blocks to map JavaScript functions. Table 2 gives examples of statements creating Smalltalk blocks and their equivalent JavaScript counterparts. The JavaScript versions of the statements are not the result of our Smalltalk-to-JavaScript translator. For example the `#+` message is mapped to JavaScript's `+` operator for simplicity.

Smalltalk statement	JavaScript statement
`[ :a | ]`	`function(a) {}`
`[ :a :b | a ]`	`function(a, b) { return a; }`
`[ :a :b | |c| c := a + b. c ]`	`function(a, b) { var c; c = a + b; return c; }`
`[ :a | a ] value: 1`	`(function(a) { return a; })(1)`
`anObject functionAbc value: 1 value: 2`	`anObject.functionAbc(1, 2)`

Table 2. Mapping of JavaScript functions

The last two examples in table 2 show how developers can invoke JavaScript functions in Smalltalk code: the same way blocks are invoked in Smalltalk. The Smalltalk way of invoking a block "`value: 1 value: 2`" is more verbose than the JavaScript notation "`(1, 2)`".

Given the described mappings, the basic concepts of JavaScript functions can be used in Smalltalk code directly. A good Smalltalk-to-JavaScript translator and a well set up client-side environment.

Using concepts which have no counterpart in existing Smalltalk funtionality requires extending the API of Smalltalk blocks.

The `new` Keyword Functions in JavaScript can play the role of constructors. Invoking a function using the `new` operator creates a new, empty object and executes the function once, with the newly created object bound to the `this` keyword. Additionally, the new object is set up to inherit prototypically from the `prototype`-slot of the constructor function.

There is no equivalent procedure in Smalltalk, since there is no prototypical inheritance. Objects cannot be created from blocks — just from classes, by creating an instance of them. Making this functionality available for client-side code written in Smalltalk requires enhancement of the message-protocol of Smalltalk blocks (the class `BlockClosure`).

Table 3 gives an example of how to use the `new` operator from within Smalltalk code.

Smalltalk code	JavaScript equivalent
`[] jsNew`	`new (function() {})()`
`anyLibraryFunction jsNew: 123`	`new anyLibraryFunction(123)`
`[ :a :b | self aSlot: a ] jsNew: 123 with: 'abc'`	`new (function(a, b){ this.aSlot = a; })(123, "abc")`

Table 3. Mapping of the `new` operator

In the Smalltalk image, there can be no real implementation of messages like `#jsNew`. Adding methods for them can be fully omitted, as the code would still be perfectly readable and compilable. However, we implemented signaling an error state to provide more convenience for the programmer. Receiving an error message like shown in listing 1.4 improves the ability to debug the application in comparison to calling `#doesNotUnderstand:` with the erroneous message send.

This "empty" implementation of the method has to be replaced with the actual functionality, when the translated code is executed on the client. Section 5.1 describes how this is implemented in Orca.

```
'The message #jsNew can be used only on the client-side,
but has been called on the server!'
```

Listing 1.4. Example error message

The `prototype` Slot Convenient mapping of the `new` keyword enables to use libraries, that provide their own constructor functions. But if the programmer wants to use the prototypical inheritance of JavaScript, an important feature is the `prototype` slot of function objects.

Settings this slot must be enabled through additional messages on the `BlockClosure` class. Similar to the `#jsNew` implementation (Section 4.1) we propose a version in the Smalltalk image throwing understandable error messages; the actual implementation is then hooked into the system at runtime on the client-side.

These messages are named `#prototype:` and `#prototype` in the Orca implementation.

When developers use the `#jsNew` message to create an instance from a JavaScript block, it is important, that the `prototype` slot has the same semantics like in JavaScript.

4.2 Objects and Slots

As explained in the introduction, slots are named properties contained in JavaScript objects. An important feature of JavaScript and it's prototypical inheritance is the creation of empty objects, filling any slot with any other object or value and accessing any slot without an immediate error. Most JavaScript libraries use these features, for example requiring a configuration object from the user. That object can be filled with values in certain expected slots or just with any number of values, that the library iterates over.

The Smalltalk syntax does not provide this possibility — the only thing that can be done with an object is sending it a message. There is no convenient way to access fields directly, like the dot-syntax in JavaScript (accessing slot `s` of object `obj` with `obj.s`). Depending on the expected or needed functionality, we propose several ways to handle this problem.

We have chosen the alternative implemented in Orca trading off between flexibility and convenience.

One possibility is to allow *any* Smalltalk object to carry arbitrary JavaScript slots (only on the client-side, or even inside the image itself). The other option is to force the creation of explicit "slot objects", that behave like JavaScript native objects.

Targeting Arbitrary Slots for *any* Object It might be useful to allow *any* object in the system (including instances of `Number`, `Dictionary` and `BlockClosure`) to be able to contain additional slots. This would map the complete possibilities of JavaScript.

The way to access these slots should in no way collide with the methods provided by the classes of the objects. That means, simple getter-setter methods (like `#a` and `#a:` for the slot named `a`) cannot be used, as they could shadow actual methods `#a` and `#a:`, that an object would respond to otherwise. This

would for example disallow putting an object into the slot named `first` of any instance of `Collection`.

To stay as close as possible to the JavaScript syntax used for slot-access, as this *might* be more intuitive for JavaScript programmers, the mapping shown in table 4 can be used.

Purpose	Message name	Smalltalk example	JavaScript version
Accessing a slot	`#,,`	`obj ,, #a`	`obj.a`
Setting a slot	`#,,` and `#,=`	`obj ,, #a ,= 1`	`obj.a = 1`
Invoking a slot function	`#,,` and `#,!`	`obj ,, #a ,! { 1 }`	`obj.a(1)`

Table 4. Mapping of slot-access operators

Slot names are identified using symbols.

This mapping has several advantages. As slot names are written down as Smalltalk symbols or strings, any JavaScript slot can bea accessed conveniently. Slots with names, that are invalid Smalltalk selectors, can still be accessed. For example the slot `$` can be accessed using `obj ,, #'$'`. Also, the code has a similar structure as JavaScript code, which supports JavaScript programmers in learning the new language. Less important, the code is rather short.

However, this mapping has several disadvantages. The biggest one is, that it is really unintuitive for Smalltalkers. Inside a Smalltalk code base, such code looks cryptic, symbolic and unclean.

Some of the disadvantages can be avoided by modifying the mapping to the one shown in table 5.

Purpose	Message name	Smalltalk example	JavaScript version
Accessing a slot	`#slotNamed:`	`obj slotNamed: #a`	`obj.a`
Setting a slot	`#slot:be:`	`obj slot: #a be: 1`	`obj.a = 1`
Invoking a slot function	`#invoke:with:`	`obj invoke: #a with: { 1 }`	`obj.a(1)`

Table 5. More verbose mapping of slot-access operators

Although a bit more verbose, this syntax is much more readable and less cryptic and symbolic than the first one.

However, both possibilities are not fully satisfying, as the code becomes either long or hard to read.

Using Dedicated "Slot Objects" If there is no need to enable every regular Smalltalk object to carry additional slots, the much simpler syntax in table 6 can be chosen.

Simple getter-setter pairs can be used to access slots. This is the default way to set properties in Smalltalk and also really close to the JavaScript slot-access notation.

Purpose	Smalltalk example	JavaScript version
Accessing a slot	`obj a`	`obj.a`
Setting a slot	`obj a: 1`	`obj.a = 1`
Invoking a slot function	`obj a value: 1`	`obj.a(1)`

Table 6. Smalltalk-like mapping of slot-access operators

The main disadvantage of this approach is that slots with names not matching legal Smalltalk selectors cannot be accessed directly. This is the case with slot-names containing the character `$` or `_`. They require using the `#perform:` message to "emulate" sending a message called that way. For example accessing the slot `$` is done with `obj perform: #'$'` and setting this same slot with `obj perform: #'$:' with: 1` (note the colon).

Also, as mentioned, this removes the possibility to put slots in instances of `Dictionary`, `BlockClosure` or other classes (meaning loss of flexibility). However, we consider this functionality as not necessary. Storing a slot in such a Smalltalk object is only necessary to pass that object into a native JavaScript library. This is *discouraged*, as such an object has no native JavaScript representation and JavaScript libraries are not able to cooperate with such a "strange" object; they usually expect regular JavaScript objects and values.

Object Literals An important JavaScript syntax element is the notation of object literals. Since objects are frequently used as configuration-dictionaries by libraries, this is important for developers. Listing 1.5 is an example for this notation.

```
{ a: 1, b: 'hello' }
```

Listing 1.5. Example for Javascropt object notation

This creates an object with the `a` and `b` slots set to the respective values.

An adequate replacement for this feature in Smalltalk syntax would be the common way to declare a dictionary. For this, the notation of a dynamic array filled with instances of `Association` is used (listing 1.6). The array has to be "marked" in some way to distunguish it from ordinary array creation.

```
{ #a -> 1. #b -> 'hello' } asObject
```

Listing 1.6. Example for mapped object notation

To actually implement this, either a compiler extension—treating this syntactical construct specially and printing the JavaScript object literal directly—or a correct implementation of the `#asObject` message is required. It can be added to the class `Array` in the way described for the `#jsNew` message in Section 4.1.

More Slot Functionality Further slot related means of JavaScript are the following:

1. Iterating over all slots of an object

2. Deleting a slot

For this we provide two further client-side-only methods called `#allSlotsDo:` and `#deleteSlot:`. Listing 1.7 shows how to use these methods.

```
obj allSlotsDo:
    [ :slotName | allSlots add: (obj perform: slotName) ].

obj deleteSlot: #a.
```

Listing 1.7. Examples for other mapped slot-operations

The first statement collects all slot-values of `obj` into the collection `allSlots`. The second statement simply deletes the slot named `a` in the object `obj`. This is almost, but not completely, equivalent to setting the named slot to `undefined`.

4.3 Global Variables

As already mentioned in Section 3.2, JavaScript and Smalltalk both provide global variables. Still, the two concepts cannot be mapped onto each other. Global variables in Smalltalk have to be present during compile time of the containing code. The client-side code however needs access to *any* global variable. Variables, that are present in the browser, for example provided by an external library, are not present in the Smalltalk image. Programmers would have to manually add a wrapper for each global variable they want to use.

We solve this problem by adding a special class called `Js`, that acts like the global namespace of JavaScript. Sending getter- and setter- messages to this class means getting and settings global variables on client-side. For example, to access and invoke the global function `alert`, the code in table 7 would be used.

Smalltalk	JavaScript
`Js alert value: 'hello'.`	`alert("hello");`

Table 7. Invoking global `alert` function

Shortcuts In some cases, code written in the syntax proposed in the previous sections is longer than the equivalent JavaScript code. This is especially the case when accessing slots that are not named like legal Smalltalk selectors. Doing so forces use of the `#perform:` method (see 4.2). Also, invoking functions is more verbose due to the need to use the protocol of `BlockClosure`. A good example for that is the frequently used jQuery[2] function `$`. Table 8 gives a comparison between the Smalltalk and JavaScript code using that function.

Programmers should be able to write shorter and more Smalltalk-like code. We allow this by providing shortcuts for global functions like `$`. These shortcuts

[2] `http://www.jquery.com/`

Smalltalk	JavaScript
`(Js perform: #'$') value: '#elementA'`	`$("#elementA")`

Table 8. Invoking the jQuery function

can be added to the already existing special class `Js`. They can be implemented in regulare client-side Smalltalk code and wrap the needed functionality. Note, that this is not a necessity; it's just for convenience in some rare circumstances.

For example the method `Js class >> #@` can wrap the jQuery `$` function (listing 1.8).

```
@ jqueryString
 "Provide a shortcut for the jQuery $ method."

 ↑ (Js perform: #'$') value: jqueryString
```

Listing 1.8. Implementation of a shortcut for jQuery

A new comparison between the Smalltalk and JavaScript versions of a jQuery call shows the difference (table 9).

Smalltalk	JavaScript
`Js @ '#elementA'`	`$("#elementA")`

Table 9. Invoking the jQuery function with a shortcut

5 Implementation

The concepts discussed in the previous sections of this paper were implemented in the Orca web framework. An architectural description of Orca has been written by Stephan Eckardt[19]. Hauke Klement describes Orca from a programmer's point of view[20] and Sebastian Woinar evaluates the Orca framework comparing it with other web application frameworks[24]. Lauritz Thamsen writes on the concepts of object-collaboration in Orca[22]. Some further implementation details of Orca are closely related to the concepts discussed in this paper and are presented in the following sections.

5.1 Client-side-only Code in Orca

To understand the following sections it is useful to know how Orca initializes the JavaScript environment when a web application is started.

Figure 1 shows the data-flow sequence taking place when initializing the system.

The scripts sent to the client in the last step contain most of the implementation of the concepts described in this paper. They are sent at the very end

to ensure, that they *replace* the code produced by the Smalltalk-to-JavaScript compiler.

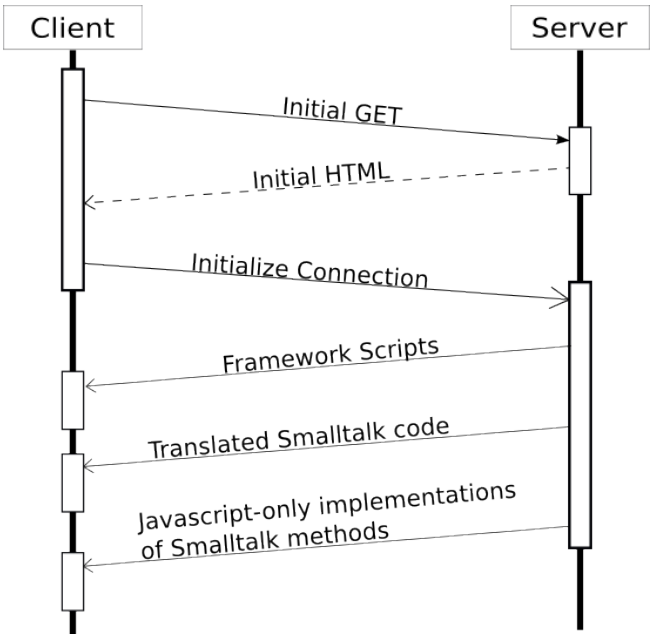

Fig. 1. Sequence of system initialization

5.2 Boxing

The concepts to work with JavaScript objects (described in 4.1 and 4.2) require a quite special implementation. As described, they rely on message-sends, which result in slot-accesses. A regular JavaScript object does not understand these messages (throwing an error like "TypeError: Object xyz has no method 'abc'"). Therefore, we need a wrapping object, that is able to understand them and perform the required slot-accesses on the actual object. These "boxes" are created automatically whenever a JavaScript object is created or enters the system from outside. Figure 2 visualizes this concept: the squares in the "Libraries" environment are regular JavaScript objects; upon entering the "Translated Environment" (the part of the JavaScript runtime containing the code compiled from Smalltalk), these objects are wrapped by "boxes" — normal Smalltalk objects like the ones on the server.

This implementation requires a working implementation of the `#doesNotUnderstand:` functionality of Smalltalk, as any possible message send must be processed. Robert Strobl writes on the impementation of this and other Smalltalk concepts in the Orca framework[21].

Not only JavaScript objects are held in boxes in Orca. Also all kinds of primitive objects (strings, numbers, booleans) and functions follow that principle.

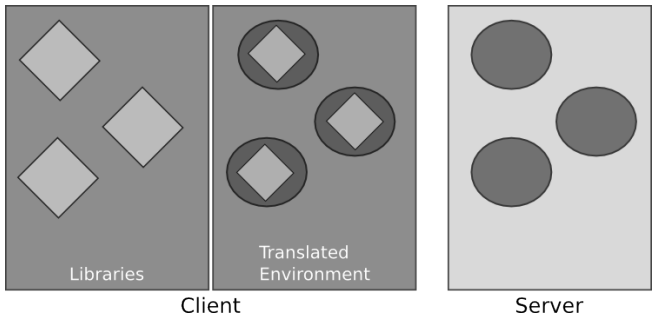

Fig. 2. Box-objects holding JavaScript native versions of themselves

5.3 Library Interoperability

A very important goal of the discussed concepts and the implementation in Orca is a good interoperability with existing JavaScript libraries.

Implementing just the concepts allows us to use the JavaScript language — but we could possibly put the JavaScript environment. Libraries cannot cooperate with the objects created in the special runtime environment for the translated code. Figure 3 illustrates that incompatibility: the "Translated Environment" contains the same kind of objects as the server; they are different from the objects in the rest of the JavaScript environment. For example, they contain several meta-properties used to emulate Smalltalk concepts in the JavaScript environment. For similar purposes, they build up complex inheritence structures. Strobl's work describes this matter in detail.

If for example a Smalltalk `String` tries to cooperate with the native JavaScript environment as-is, like attempting native string-concatenation, this could fail or produce unwanted results.

Another example is the incopatibility between Smalltalk number objects and JavaScript numbers. If a number has to be given as argument to a library, the library has to receive an actual JavaScript number and not the Smalltalk object.

We use the principle of "boxing" to solve this problem as well. This means each object in the runtime environment for translated code has a native representation of itself, which is capable to cooperate with the rest of the JavaScript environment.

Given these prerequisites, we only have to "box" and "unbox" the objects at the correct places to always keep the right version of each object in it's inherent part of the system. The boxed version stays in the Translated Environment and the unboxed version stays in the rest of the JavaScript environment. We hold a link between the two to support object identity.

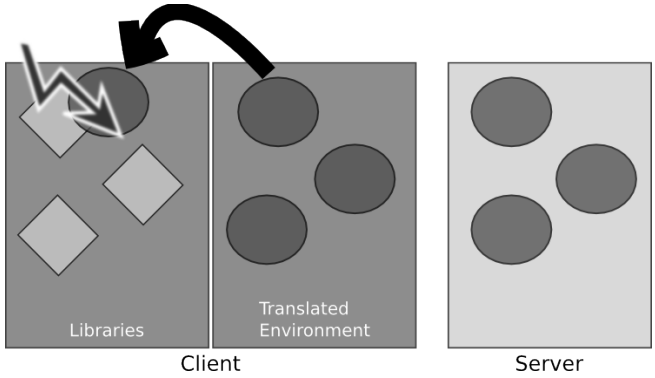

Fig. 3. Passing an incompatible object into a JavaScript library wihtout unboxing

Figure 4 shows how an object is transformed, when crossing the border between the two parts.

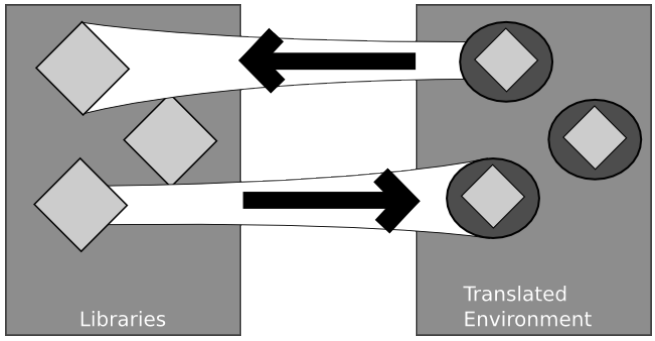

Fig. 4. Boxing and unboxing of objects in the JavaScript environment

6 Evaluation

This section evaluates the concepts and their implementation presented in this paper. As the main contribution is the quality of client-side code and the possibilities to interact with JavaScript libraries, the evaluation will focus on these two aspects.

The Orca framwork is compared to two similar systems. The two selected reference frameworks both include a Smalltalk-to-JavaScript compiler like Orca, and also feature programming client-side JavaScript code in Smalltalk.

For the evaluation, a small client-side application is programmed using all of the three compared frameworks. Important code fragments of the different

implementations are shown and compared. The application is constructed to focus on the functionality mentioned above and explicitely does *not* contain the following parts:

- Communication between client and server
- Executing code on the server
- Sending HTML-code to the client (the page is set up with JavaScript)

The following two sections give short descriptions of two related systems (6.1 and 6.2). The next section (6.3) describes the chosen example application. After that (6.4), selected code listings are presented and compared; aspects, that cannot be demonstrated by the code, are explained. An evaluation result and summary is given in 6.5.

6.1 Jtalk

Jtalk by Nicolas Petton[3] [4] is an implementation of Smalltalk, that runs on top of the JavaScript runtime. The Smalltalk code is completely compiled to JavaScript. However, there are differences to other Smalltalk implementations [14]. For example, many collection-classes are not supported; instead, `Array` is used as the size of an `Array` in Jtalk is dynamic. The differences originate in the differences between Smalltalk and JavaScript.

Writing client-side code in Jtalk is described in more detail, when the code of the example application is shown. However, when building an application with Jtalk (including not only JavaScript functionality), one can take advantage of the additional framework functionality provided by Jtalk, that is not used in this evaluation.

6.2 SWT

The Smalltalk Web Toolkit (SWT)[2] is a framework for web applications. It includes a Smalltalk-to-JavaScript compiler, but also the infrastructure needed for proper web development; for example, synchronization of data between multiple clients.

The compiler of SWT (named St2Js[3]) tries to use as many JavaScript native code elements as possible, thus producing relatively fast JavaScript code. However, this reduces the flexibility of Smalltalk code; for example the `#==` message is translated to JavaScript's `===` operator, instead of being polymorphic.

6.3 Example application

The example application is a simple, graphical client-side-only application. Small, moving balls are painted inside a filled rectangle representing a movement area.

[3] `http://github.com/NicolasPetton`
[4] `http://github.com/NicolasPetton/jtalk`

They bounce off the sides and a button can be clicked to add a new ball to the field.

An HTML5 canvas[5] and an arbitrary JavaScript library are used to paint the field and the balls. The JavaScript `setInterval` function generates the movement by invoking a stepping method in regular intervals. Figure 5 shows the classes used to implement this application.

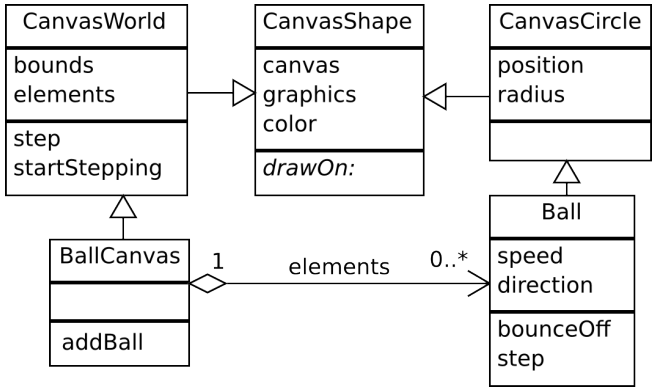

Fig. 5. Class diagram of the example application

The application starts off in an empty HTML-page and adds all DOM-elements it needs dynamically. The following elements are added directly to the body of the HTML-page:

- An HTML5 canvas element
- An input element (the button)

The library used to paint on the canvas is the Canvas Toolkit[6]. We used a part of the provided functionality, mainly to draw the rectangles and circles.

After the elements have been created and the library has been initialized, a function is registered with a call to `setInterval`. In a regular interval, the application paints a filled rectangle to delete everything on the canvas; the position of the balls changes based on the balls' current direction and speed, they bounce off the wall if necessary and are painted afterwards.

This application demonstrates all typical functionality needed when programming client-side JavaScript:

- Working with JavaScript native objects and functions
- Using HTML GUI-elements
- Using native JavaScript libraries

[5] `http://www.w3schools.com/html5/html5_canvas.asp`
[6] `https://canvastoolkit.codeplex.com/`

6.4 Comparison

The example application has the following important pieces of code:

- The initialization sequence
  - Create the canvas-element
  - Create a JavaScript-instance of `ctk.Graphics`
  - Set up the movement with a call to `setInterval`
  - Create the button
- The code to draw a circle on the canvas

The following sections show the according code lines written for the three different systems.

The listings don not contain the actual code for the three implementations, rather a condensation of the most interesting pieces. The lines are listed in the same order as shown in the structure above.

For all three implementations, the invokation of `self class canvasCode` returns the piece of HTML-code shown in listing 1.9.

```
'<canvas
id="ballCanvas" width="400" height="300"
style="background-color: #ffffff; position: relative;
border: solid 1px #000000;">
</canvas>'
```

Listing 1.9. HTML-code for an HTML5-canvas-element

This HTML-code is added to the HTML-tree to create the required canvas-element.

Jtalk The main way to write client-side code in Jtalk is the use of inline JavaScript code with the syntax shown in listing 1.10.

```
smalltalk statements...
abc := {' javascript statements '}.
smalltalk statements...
```

Listing 1.10. Inline JavaScript syntax of Jtalk

Jtalk provides wrapper-libraries for the following functionality:

- Setting up the DOM using a HTML-canvas object
- Drawing on an HTML5-canvas
- The jQuery library

These wrappers are implemented using the inline-syntax shown above.

Listing 1.11 shows the code from the example application for Jtalk.

```
BallCanvas >> #renderOn: html
  html input onClick: [ self addBall ].

BallCanvas >> #initialize
```

```
  | canvasCode canvas intervalAction |
  canvasCode := self class canvasCode.
  {' document.body.innerHTML += canvasCode '}.
  canvas := {' document.getElementById("ballCanvas") '}.
  self canvas: canvas.
  self graphics: {' new ctk.Graphics(canvas) '}.
  intervalAction := [ self step ].
  {' setInterval(function(e){intervalAction(e, self), 3) '}.

CanvasCircle >> #drawOn: graphics
  | color positionX positionY radius|
  color := self color
  {' aGraphics.setColor(color); '}.
  positionX := self position x.
  positionY := self position y.
  radius := self radius.
  {' graphics.fillCircle(positionX, positionY, radius); '}.
```

Listing 1.11. Example application for *Jtalk*

Jtalk provides the `#renderOn:` method to access an HTML-canvas object (similar to Seaside). The button is created using this functionality.

The rest of the initialization sequence is in the `BallCanvas >> #initialize` method. First, the HTML-snippet provided by `self class canvasCode` is added to `body.innerHTML`. Note, that the result of the `self class canvasCode` call is stored in a local variable first. This is done to avoid calling Smalltalk methods inside the inline JavaScript code. Doing so would mean using the mangled function-names and result in the line shown in listing 1.12 (note the underscores).

```
{' body.innerHTML += self._class()._canvasCode() '}.
```

Listing 1.12. Inline JavaScript code with mangled function names

As the rules of the mangling-mechanism are not always clear, this was avoided.

Next, the created canvas-element is acquired. Again, this is done using inline JavaScript code with a call to `getElementById`. The result is stored in the local variable `canvas` to use it in the constructor call to `ctk.Graphics`, which initializes the canvas library.

To implement a call to `setInterval`, the block to be called (`intervalAction`) is created first. `setInterval` is then invoked with an anonymous function, which again invokes the `intervalAction` block. This contstruct is needed to invoke a Smalltalk block from the inline JavaScript code, because the block-invokation requires the correct `self` value.

To draw a circle on the canvas, the functions of the `graphics` object are called using inline JavaScript. The parameter-values are stored into local variables again.

SWT Contrary to Jtalk, SWT does not rely solely on JavaScript inline code (it is provided nevertheless).

In SWT, the mapping from Smalltalk to JavaScript datatypes is very direct. This means, for example, Smalltalk strings are compiled to real JavaScript strings. Many message-invokations are compiled to the according JavaScript operators, like the messages `#==` and `#+`.

SWT provides several key messages, that are compiled directly to JavaScript object operations; these are shown in table 10).

Smalltalk code	JavaScript equivalent
`a jsSet: 'b' to: c`	`a.b = c`
`a jsGet: 'b'`	`a.b`
`a jsPerform: 'b' with: 123`	`a.b(123)`
`self jsLiteral: 'alert(1)'`	`alert(1)`
`a jsNew: 'b' with: c`	`new a.b(c)`

Table 10. Messages for JavaScript object manipulation in SWT

These messages provide all the functionality needed to write the example application (shown in listing 1.13). In particular, the `#jsLiteral:` method gives the only access to global variables.

```
BallCanvas >> #initialize
  | body |
  body := self jsLiteral: 'document.body'.
  body
    jsSet: 'innerHTML'
    to: (body jsGet: 'innerHTML'), self class htmlCode.
  self canvas: (
    (self jsLiteral: 'document')
      jsPerform: 'getElementById'
      with: 'ballCanvas');
  self graphics: (
    (self jsLiteral: 'ctk')
      jsNew: 'Graphics'
      with: self canvas);
  (self jsLiteral: 'window')
    jsPerform: 'setInterval'
    with: [ self step ] with: 3.
  self rootWidget add:
    SWTButton
      caption: 'Add Ball'
      onClick: [ self addBall ].

CanvasCircle >> #drawOn: graphics
  graphics jsPerform: 'setColor' with: self color.
  graphics jsPerform: 'fillCircle'
    with: self position x
    with: self position y
```

```
    with: self radius.
```

Listing 1.13. Example application for *SWT*

Using the messages shown in table 10, the implementation of the initialization sequence is pretty straght forward. First the `innerHtml` property of `body` is concatenated with `self class htmlCode` and written back. Then the created canvas-element is retrieved using the `getElementById` function of `document`; an instance of `ctk.Graphics` is created using the `#jsNew:` message and the movement is set up passing a simple block into the `setInterval` function.

The button is realized using the widget-library of SWT. It contains wide support for most HTML elements, wrapped in a Smalltalk API. The widgets are implemented using the means listed in table 10.

Drawing the circle is done using the `#jsPerform:` message; the resulting code is self explaining.

Orca Orca implements the client-side coding concepts described in this paper. These means are used to implement the example application for the Orca system (listing 1.14).

```
BallCanvas >> #initialize
  Js Document body innerHTML:
    Js Document body innerHTML, self class canvasCode.
  self canvas:
    (Js Document getElementById value: #ballCanvas).
  self graphics: (Js Global ctk Graphics jsNew: self canvas).
  self add: (
    OrcaButton new
      text: 'Add Ball';
      onClickDo: [ self addBall ];
      yourself).
  Js Global setInterval value: [ self step ] value: 3.

CanvasCircle >> #drawOn: graphics
  graphics setColor value: self color.
  graphics fillCircle
    value: self position x
    value: self position y
    value: self radius.
```

Listing 1.14. Example application for *Orca*

First, the code from `self class canvasCode` is appended to `Js Document body innerHTML`. Then the created element is fetched using the `getElementById` function and a new instance of `Js Global ctk Graphics` is created. The button is realized using an `OrcaButton`, which is a simple wrapper around an HTML-input element, providing a Smalltalk-like API. The same could have been realized without this class, using plain JavaScript.

The `#drawOn:` of `CanvasCircle` calls the two required JavaScript-functions of the `graphics` object. As parameters, values taken from `self` are passed directly.

6.5 Evaluation Results

The presented evaluation is cannot be conducted on pure objective basis. Choosing the "best" source code depends on personal style and real metrics are hard to define.

However, we tried to identify the aspects that are focused by this evaluation:

- The code should resemble regular Smalltalk code as closely as possible
- The code should be maintainable, short and understandable

The coding style that fulfills these requirements is best fit to be used in an actual developer team. It should not be very different from the rest of the application code to avoid the feeling of two different programming languages and frequent "context switches". A code syntax, that is readable and maintainable in general, greatly supports developers in producing good code. We think, that in the case of Smalltalk the second point is implied by the first.

Based on these values we think, that the code produced for Orca stands out against the two compared systems. The following are the two most important arguments:

- There is *no need* for inline JavaScript code
- Names of slots are not written as string literals; everything is expressed using messages in Smalltalk style

Negative aspects about the code written for Orca are the frequent occurence of the class `Js` and the lacking support of inline code whatsoever.

7 Related Work

7.1 HOP

HOP[17] is a programming language for web applications. The language is based on Scheme[6] but not fully compatible with it; however, HOP code *can* be written in clean Scheme. The HOP language has many similiarties with Orca and the concepts described in this paper. As related work for this paper, the most relevant part is a Scheme-to-JavaScript compiler called Scm2Js[13]. The compiler does not compile pure Scheme, but has several extensions simplifying the syntax to access JavaScript from Scheme. The result is clean (pseudo) Scheme code.

Contrary to our approach, objects and data structures of Scheme are mapped directly to JavaScript counterparts, as the two sides are considered close enough to each other, JavaScript being derived from Scheme.

Orca on the other hand uses wrapper objects everywhere to combine full Smalltalk and JavaScript functionality (see Section 4).

Another difference is, that the Scm2Js compiler does not care much about the cleanness of the produced code, while the Orca's compiler does. Lars Wassermann writes about Orca compiler[23].

7.2 GWT

The Google Web Toolkit[7] (documented in [5]) is a Java framework for writing web applications. As an important part, it includes a sophisticated Java-to-JavaScript compiler, that can produce optimized code for each browser. GWT also includes many other functionality needed for a web application, like a communication layer.

The way client-side code is written for GWT is quite different from Orca. The root of the difference probably lies in the difference between Java and Smalltalk — Java is statically typed, while Smalltalk is not. To use existing JavaScript libraries in GWT, a wrapper or even a full port of the library is needed as a Java API. For example, the full jQuery library has been reimplemented in GWT, called GwtQuery[8]. The JavaScript Native Interface (JSNI) of GWT allows implementing whole methods in real JavaScript code; accessing JavaScript from Java code is not supported. GwtQuery has been implemented using JSNI.

Writing such wrappers is not necessary for accessing arbitrary libraries from Orca, where just any object in the JavaScript environment can be accessed from the Smalltalk code. When using GWT, a client-side programmer has to use the widgets and APIs provided by GWT, while in Orca developers can access any third-party library.

7.3 SWT

The Smalltalk Web Toolkit[2] has been presented in the evaluation in Section 6.2. Being a system, that uses Smalltalk to program the client-side, it has many similarities with Orca. Regarding the client-side code, SWT uses special messages to work with JavaScript objects (like the concept described in Section 4.2), resulting in a more powerful, but also more verbose syntax; we chose a different approach and allow slot-accesses only on explicit JavaScript objects.

7.4 Jtalk

Jtalk[14] has been presented in Section 6.1. It is a client-side only Smalltalk system, also providing a Smalltalk-to-JavaScript compiler. JavaScript is programmed solely using inline JavaScript code, resulting in a totally different programming experience compared to Orca, wich allows using pure Smalltalk code to handle all tasks.

7.5 Language-oriented Programming

Language-oriented programming is a paradigm stating, that each problem should be solved and each solution expressed in the best-fitting language[15]. Such a language is called a Domain Specific Language (DSL) [4]. Humm and Engelschall

[7] `http://code.google.com/webtoolkit/`
[8] `http://code.google.com/p/gwtquery/`

have described the concept of *DSL Stacking*[10]. This concept integrates DSLs directly into a basic host language. To describe a specific DSL, it is based upon an already existing DSL or the host language itself.

This approach benefits the goals pursued in this paper. Although we did not implement the DSL stacking principle, we still integrated the *internal DSL* of JavaScript inside the Smalltalk base language. Following the ideas of Humm and Engelschall would have probably made our implementation simpler.

8 Conclusion

Writing the client-side code of a web application is a frequent and important task nowadays. The ideas presented in this paper show, that this task can be handled using an arbitrary, object-oriented programming language. In particular we demonstrated how to use the full extent of the JavaScript programming language in the host language Smalltalk.

We have shown, that the presented concepts are feasible and the resulting client-side code is maintainable and looks just like normal Smalltalk code.

As the Web and related web technologies evolve, it is important to apply the experience collected in server-side programming to the client-side also. To produce structured and maintainable not only on the server- but also on the client-side, any programming language of choice must be usable as host language for client-side code.

Although we concentrated on using Smalltalk as host language, many of the described ideas are applicable for any object-oriented programming language. In the future, more and more of these languages should be equipped with the possibility to express client-side code.

References

1. Crockford, D.: JavaScript: The Good Parts. O'Reilly Media, Inc. (2008)
2. Deck, D.: Introduccin a smalltalk web toolkit (swt) (spanish only) (Jun 2011), `http://ceibo.wordpress.com/2008/01/06/introduccion-a-smalltalk-web-toolkit-swt/`
3. Deck, D.: St2js - traductor de smalltalk a javascript (spanish only) (Jun 2011), `http://diegogomezdeck.blogspot.com/2006/07/st2js-traductor-de-smalltalk.html`
4. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. SIGPLAN Not. 35, 26–36 (June 2000)
5. Dewsbury, R.: Google Web Toolkit Applications. Addison-Wesley Professional, 1 edn. (Dec 2007)
6. Dybvig, R.K.: The Scheme Programming Language, 4th Edition. The MIT Press, 4th edn. (2009)
7. Ecma-262: Ecmascript language specification (dec 1999), `http://www.ecma-international.org/publications/standards/Ecma-262.htm`
8. Flanagan, D.: JavaScript: The Definitive Guide. O'Reilly Media, Inc. (2006)

9. Goldberg, A.: SMALLTALK-80: the interactive programming environment. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1984)
10. Humm, B.G., Engelschall, R.S.: Language-oriented programming via dsl stacking. In: Jos Cordeiro, Maria Virvou, B.S. (ed.) Proceedings of the 5th International Conference on Software and Data Technologies. ICSOFT 2010, vol. 2, pp. 279–287 (July 2010)
11. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future: the story of squeak, a practical smalltalk written in itself. In: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. pp. 318–326. OOPSLA '97, ACM, New York, NY, USA (1997)
12. Johns, M.: On javascript malware and related threats. Journal in Computer Virology 4, 161–178 (2008), 10.1007/s11416-007-0076-7
13. Loitsch, F., Serrano, M.: Hop client-side compilation (Jun 2011), `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.107.5025&rep=rep1&type=pdf`
14. Petton, N.: Jtalk homepage and documentation (Jun 2011), `http://jtalk-project.org/`
15. Pickering, R.: Language-oriented programming. In: Foundations of F#, pp. 271–297. Apress (2007)
16. Renggli, L., Gîrba, T.: Why smalltalk wins the host languages shootout. In: Proceedings of the International Workshop on Smalltalk Technologies. pp. 107–113. IWST '09, ACM, New York, NY, USA (2009)
17. Serrano, M.: Hop: an environment for developing web 2.0 applications. In: Proceedings of the 2007 International Lisp Conference. pp. 6:1–6:1. ILC '07, ACM, New York, NY, USA (2009)
18. Yue, C., Wang, H.: Characterizing insecure javascript practices on the web. In: Proceedings of the 18th international conference on World wide web. pp. 961–970. WWW '09, ACM, New York, NY, USA (2009)

Bachelor Theses

19. Eckardt, S.: An Architecture Overview of the Orca Web Application Framework. Bachelors thesis, Hasso Plattner Institute for Software Systems Engineering (2011)
20. Klement, H.: The Development Process with Orca. Bachelors thesis, Hasso Plattner Institute for Software Systems Engineering (2011)
21. Strobl, R.: Implementation of Smalltalk Language Features in JavaScript. Bachelors thesis, Hasso Plattner Institute for Software Systems Engineering (2011)
22. Thamsen, L.: Object Collaboration in the Orca Web Framework. Bachelors thesis, Hasso Plattner Institute for Software Systems Engineering (2011)
23. Wassermann, L.: Translating Smalltalk to JavaScript. Bachelors thesis, Hasso Plattner Institute for Software Systems Engineering (2011)
24. Woinar, S.: Comparing Frameworks for Web Application Development. Bachelors thesis, Hasso Plattner Institute for Software Systems Engineering (2011)