Submitted toIWST 2016

Mocks, Proxies, and Transpilation as Development Strategies for Web Development

Noury Bouraqadi
Mines Douai, France
noury.bouraqadi@mines-douai.fr

Dave Mason
Ryerson University, Canada
dmason@ryerson.ca

Abstract

With the advent of HTML 5, we can now develop rich web apps that rival classical standalone apps. This richness together with the portability of web technologies, turned HTML 5 into a viable (and in the case of mobile - essential) solution to develop cross-platform apps. This possibility is heavily dependent on Javascript having acceptable performance, good testability, and a modern development environment. Despite its extensive use in creating highly interactive environments, most Javascript development environments currently use a compile/run paradigm. Similarlily, testing is frequently tacked on, rather than being an integrated part of the development cycle. We propose PharoJS which leverages the Smalltalk IDE with a seamless transition from native Smalltalk tests, through proxied browser tests, to full browser-resident tests. We support the standard event-driven browser model and transpile Smalltalk code into efficient Javascript for execution in the browser. We further support testing - both manually and automatically - in a range of browsers to provide assured consistency upon deployment. In addition to transpiling the Smalltalk code to Javascript to perform tests in the browser, we can also run non-interactive tests within the Smalltalk environment. The unique feature we provide is the ability to run interactive tests largely within the Smalltalk IDE, so as to fully exploit the debugging and development environment, while the actual interaction occurs on the browser. We exhibit this new mode of development via a simple application.

1. Introduction

Javascript is one of the most ubiquitous languages in today's computing world. It is used for many applications ranging from web client side, through server side, to standalone apps on the web, on handheld devices based on infrastructures such as PhoneGap, or on the desktop based on infrastructures such as appjs, Electron, NW.js(Milani and Benvie; GitHub; Community). Although there have been many systems designed and sold to generate Javascript, only WebStorm(JetBrains) comes close to providing the kind of powerful development and execution environment afforded by a modern Smalltalk envionment such as Pharo.

We have produced an environment that runs within Pharo Smalltalk and allows the programmer to build an application using the tools available in that environment through a structured development process. Firstly, developing algorithms and tests using dummy objects for user interface. Then, running and debugging the application while using any web browser as the user interface via transparent proxies. Finally, when the application is tested and running, a standalone Javascript application can be extracted from the tested Smalltalk classes and deployed on the web or in a mobile device.

This paper is about language/run-time interoperability for web (Javascript) development using a host-based development environment. We describe this using Pharo as the host environment and Smalltalk as the development language, but the principles are not language or IDE specific.

The contributions of the paper include:

1. A development methodology for web applications that supports testing of true, browser-based semantics within a powerful IDE - for all the browsers targeted by the developer.
2. The use of proxy objects to access browser-based objects and semantics from the host language during development.
3. The use of reverse-proxy objects to access host code from browser-based events and libraries during development.
4. A seamless transition from the development environment to fully browser-based code for final testing and deployment.

In §2 we describe the problem that motivated this research, by evaluating the state of the art of Javascript-

based web application development. We show that the community is missing a powerful IDE for web development. §3 give an overview of the contributions of this paper. It presents our test-driven development process, and how to evolve a Smalltalk app running entirely within Pharo, into a Javascript one running entirely within a web browser. A detailled description of the proposed solution is then provided by the three following sections. We show how we rely on mocks for native Pharo testing in §4. Then, §5 introduces the different kind of proxies to support communications between the web browser and Pharo. §6 describes the final stage of development where the complete app is transpiled from Smalltalk to Javascript and run in the browser. Last, §7 draws conclusions and proposes future work arising out of the proposal described here.

2. State of Art

There are development tools for Javascript; there are debugging tools for Javascript; and there are testing tools for Javascript; but there is no integrated environment that includes all three.

This is particularly a problem when using Test-Driven-Development (TDD). When developing in a rich IDE, such as Pharo, it is natural to create and run a test, and when the test fails (either because of a missing method, an error, or an incorrect result) one can edit directly from the debugger, hot-swap code (including defining new methods), and then continue the test execution.

There are many development environments for Javascript such as CoffeeScript(Ashkenas), most of which are primarily cleaned up languages or integrated editors that provide access to Javascript, HTML, and CSS[1] These usually add an additional step to the development process wherein the source code is translated to Javascript. Only WebStorm(JetBrains) appears to have a Javascript IDE even close to an environment like Pharo.

Debugging tools such as Firebug(Mozilla) are built into most browsers. They allow breakpointing, single-stepping, and examining of code and data. But they don't include editors that feed changes back into the codebase.

Testing tools such as Selenium(HQ) provide test environments for Javascript, but they have a couple of problems. Firstly, they are largely used for regression testing or for after-the-fact validation, rather than as part of the development process. Secondly, they are not actualy running the browser Javascript implementation and are not interacting with a browser DOM; therefore they do not capture the exact semantics of the production environment.

Amber(Petton et al.) implements a Smalltalk IDE in the browser. Compared to Pharo it is an impoverished environment IDE, with no hot-swap of code, limited debugger, and no refactoring.

[1] See blog post "What are the best javascript IDEs" http://www.slant.co/topics/1686/~what-are-the-best-javascript-ides.

DOPPIO(Vilk and Berger 2014) assumes that the execution model is sequential/batch execution where a program alternates running and waiting for I/O. This is at odds with web programming - and modern interactive applications - where users can typically enter data in their preferred order and then it is validated. The mechanisms they developed to support their execution model also supports a debugging capability, although at appreciable performance penalty.

3. Overview of the Proposed Solution

In this section we give an overview of our development process as well as the underlying platform. Our goal is to allow developers to benefit from the rich Pharo development environment, while targetting a web browser that executes Javascript.

The process is inspired by eXtreme Programming where tests serve as a guide to developers. It involves a progression in 3 steps: from running entirely within Pharo to running entirely within the browser. The test suite is transferred from one step to the other. Tests are developed once and run under different conditions. Only setup and resources are changed.

The 3 steps of our development process are the following:

1. *Native Pharo Testing*. This stage is perfect for testing processing and algorithms. Developers write tests related to the application's logic only, such as a state-machine and its transitions.

   Running the tests and ensuring they pass leads to the code that represents the core of the app. Since this is done within the Pharo environment, it benefits from the full power of the language and its IDE (e.g refactoring, versionning, hot-code swapping during debug).

   However, it is not always possible to fully disconnect the app's logic from other parts of the app provided by the web browser. In such situations, developers can rely on mock objects (Kim et al. 2006). We provide the simplest possible mocks, that silently accept any messages. Developers can however still develop their own specific mocks if needed.

2. *App with a remote browser*. At this stage, we introduce features that require the web browser. That is typically the UI. Both tests and application code still run on Pharo, so we still benefit from the language and its IDE. Entities such as DOM elements or third-party Javascript code run on the browser.

   Our infrastructure, PharoJS, allows controlling the web browser (e.g. open, close, reload), and establishing connections between entities on the web browser and ones on Pharo. This is achieved thanks to proxies. On the Pharo side, we have proxies of entities on the browser (e.g. window, document). Symetrically, on the browser side, we have proxies of Pharo code blocks and other objects so the browser can trigger application features by means of events and call-backs. This allows the code running in

Pharo to interact with the DOM and field browser events, all the while having full access to the Pharo debugging, development, and refactoring tools.

3. *Transpilaton to Javascript* In this final step, tests ensure the full app functions with multiple browsers. This allows the export (in Javascript) of the Smalltalk code that has been developed, and full execution of all of the code within the browser. The key to this is that the Smalltalk execution model is faithfully replicated in the transpiled Javascript code.

4. Mocks for Native Pharo Testing

Our proposed development methodology starts with pure code written in Pharo Smalltalk, focusing on the model classes for the application. Although all the tests developed in this phase are running completely within Pharo, they can all also be run in the browser(s) in phase 3 to verify that there have been no errors introduced when calling Javascript libraries and DOM objects.

Using the principles of Test-Driven-Design(Beck 2002), tests are created and run, code is written to allow the tests to pass, and the sequence repeats until a point is reached where the tests require interaction with the user via the browser. If the test does not require an actual result from the browser, then mock objects can be used. Some global names like `document`, `window`, etc. are bound to mock objects to allow code to reference browser objects - but they accept any message and return no useful results.

At any point in the subsequent development, if a test arises that doesn't require browser interaction it can be added to the set of tests to be run standalone in Pharo.

5. Proxies for Browser ↔ Pharo Communication

Sooner or later, a test will require true interaction with the browser. This may be because it needs to manipulate the DOM, to provide some event handling, or to interact with some third-party Javascript code.

Figure 1 shows a simple unit test. It demonstrates a very natural interaction with the browser Javascript interfaces, similar to that used in Amber(Petton et al.).

The programming model, shared with Amber represents Javascript function calls, field accesses, and field assignments as Smalltalk message sends. The translation is done a couple of different ways in different places in our system The code (from lines 3,4,7,11):

```
document createElement: 'button' ...
button id: 'who' ...
button addEventListener: #click
        block: [:xev| flag:=true]
button style backgroundColor: 'pink' ...
```

is effectively transpiled[2] into

```
document.createElement("button")
button.id="who"
button.addEventListener("click",
      function(xev){flag = true})
button.style.backgroundColor= 'pink'
```

and it is fairly easy to see how this would work in the browser.

In fact, this code would work perfectly well in Amber[3] which runs within the browser. We could transpile this test to Javascript and run it in the browser, and it would work. But if it didn't and we needed to debug it, we would have to switch out of our development environment and start debugging it as Javascript and using the breakpoints provided in the browser's debugger. While this might be reasonable for a Javascript developer who wanted to write his/her code in Smalltalk, for our target audience of Smalltalkers who want to write for the web this would be an unfamiliar world.

Instead, we want to run and debug the code in Pharo Smalltalk but use the browser as the User Interface. This introduces three main problems.

1. How can the code in Pharo get access to the DOM and other objects in the browser?
2. How can the browser get access to a `BlockClosure` in Pharo for event callbacks?
3. How can the browser access other callbacks?

5.1 Pharo access to the browser: Forward proxies

This is the mechanism for the Pharo code to access the objects in the browser. Some global names like `document`, `window`, etc. are bound to proxies to provide an initial entré into the browser.

5.2 Event callbacks to Pharo: Reverse proxies

This is the mechanism for the browser code to allow event callbacks to call code blocks in Pharo. When the application adds an event listener, the browser code is asked to create a callback proxie. A mapping is set up on the Pharo side to map that proxy to the corresponding BlockClosure. Then the listener is set to the proxy.

The complication here is that the Pharo object must be retained as long as there is a reference on the browser, but no longer. There is no weak map on the browser, so the proxies are garbage collected using a reference counting collector. When the count goes to zero, the browser sends a callback to remove the mapping on the Pharo side, which frees up the BlockClosure to enable it to be garbage collected. The reference count is maintained by adding or removing an event handler, so the reverse proxy remains until the listener is re-assigned.

[2] This is correct in principle, but the details are quite different, as will be explained in §6.

[3] with the exception of `slot_` and `native_` which will be discussed later

```
1  testBrowserButton
2      | button flag |
3      button := document createElement: 'button'.
4      button id: 'who';
5          slot_innerHTML: 'Click on this within 10 seconds or the test fails';
6          addEventListener: #click block: [: ev |
7              button addEventListener: #click block: [: xev | flag := true];
8                      innerHTML: 'Click again immediately or the test fails';
9                      style backgroundColor: 'yellow'
10         ].
11     button style backgroundColor: 'pink';
12                 height: '2cm'.
13     document body native_appendChild: button.
14     flag := nil.
15     self assert: (window confirm: 'If you can see the pink button,'
16                                   ' accept this and then click on the button twice')
17         description: 'no aknowledgement on browser'.
18     self delayFor: 10 seconds orUntil: [flag].
19     self assert: flag notNil description: 'button not clicked'.
20     self assert: (window confirm: 'Did the button turn yellow?')
```

Figure 1. Simple interactive test

It is thus the programmer's job to set listeners to `nil` before removing a DOM object. The ramifications of failing to do this are limited, since these reverse proxies are only used while testing and debugging, so no long-term memory leak is created.

5.3 Other browser callbacks to Pharo: Reverse proxies

This is also the mechanism for the browser code to access other callbacks in Pharo. In fact, any time a `BlockClosure` is passed to a function on the browser, a reverse proxy is created. In this way, browser-side objects can call blocks that reside on the Pharo side.

Unfortunately, because of the Javascript execution model, these cannot return useful values, so this only works properly for a stylized use of callbacks - callbacks that work by side-effect or that call back their own parameters, rather than return any useful value. In other words, these callbacks must use continuation-passing-style (CPS)(Appel and Jim 1989). Fortunately (if surprisingly) this is a very common style of coding for Javascript code, mandated by the Javascript execution model.

The bad news is that there is no way to garbage collect these proxies, so they create a memory leak until the bridge between the Pharo and browser is reset, at which point the values will be collected on the Pharo side. Again, this is not a long-term problem as reverse proxies are only used for development/debugging/testing. The unary message `beOneShot` can be sent to a `BlockClosure` so that the reverse proxy will be removed after the first time it is invoked, which will commonly be the case in the CPS style of coding.

5.4 Example of Browser ↔ Pharo Communications

Figure 2 shows the interactions across a web socket between the browser and Pharo– from the perspective of the browser (so "Received" means a message from Pharo to the browser and "Sent" means a message from the browser to Pharo).

The browser receives a string of Javascript, which it executes. It then responds with the result. The responses are of two forms - both objects with different keys:

- the key `basic` has a value which is a basic Javascript value type: number, boolean, string;
- the key `proxy` has a value with is a string of the form `$_n`, which represents an anonymous Javascript object, or a string with the name of a named Javascript object such as `document`, `window`, `undefined`, etc.

The log also shows another form, an object with the key `cb` which is an asynchronous callback with a value of an array. The first element of the array is the callback proxy, which is used to access the corresponding BlockClosure in Pharo. The next element is the event that triggered the callback. The third element is the object which received the event (`this` in the Javascript function), which is rarely needed.

The received messages are of 4 different forms:

1. Direct assignment of a field. This is seen in line 5 of the log and also line 5 of figure 1. The `slot_` prefix on the name identifies it as a slot reference (for value or for

```
Received: document._createElement_("button")
Sent: {"proxy":"$_1"}
Received: $_1._id_("who")
Sent: {"basic":"who"}
Received: $_1.innerHTML="Click on this within 10 seconds or the test fails"
Sent: {"basic":"Click on this within 10 seconds or the test fails"}
Received: JbLoggingEvaluatorWebSocketDelegate._default()._makeBlockClosureProxy_(false)
Sent: {"proxy":"$_2"}
Received: JbLoggingEvaluatorWebSocketDelegate._set_callback_to_($_1,"click",$_2);
Sent: {"basic":true}
Received: $_1._style()
Sent: {"proxy":"$_3"}
Received: $_3._backgroundColor_("pink")
Sent: {"basic":"pink"}
Received: $_3._height_("2cm")
Sent: {"basic":"2cm"}
Received: document._body()
Sent: {"proxy":"$_4"}
Received: $_4.appendChild($_1)
Sent: {"proxy":"$_1"}
Received: window._confirm_("If you can see the pink button, accept this and then click on the
Sent: {"basic":true}
Received: :-$_4
Received: :-$_3
Sent: {"cb":["$_2",{"proxy":"$_5"},{"proxy":"undefined"}]}
Received: $_1._innerHTML_("Click again immediately or the test fails")
Sent: {"basic":"Click again immediately or the test fails"}
Received: JbLoggingEvaluatorWebSocketDelegate._default()._makeBlockClosureProxy_(false)
Sent: {"proxy":"$_6"}
Received: JbLoggingEvaluatorWebSocketDelegate._set_callback_to_($_1,"click",$_6);
Sent: {"remove":true,"proxy":"$_2"}
Received: :-$_2
Sent: {"basic":true}
Received: $_1._style()
Sent: {"proxy":"$_7"}
Received: $_7._backgroundColor_("yellow")
Sent: {"basic":"yellow"}
Sent: {"cb":["$_6",{"proxy":"$_8"},{"proxy":"undefined"}]}
Received: window._confirm_("Did the button turn yellow?")
Sent: {"basic":true}
```

Figure 2. Log of messages between Pharo and the browser

assignment). This is rarely needed except when setting a field that doesn't already exist, or setting a field that may have a functional value.

2. Directly calling a function. This is seen in line 19 of the log and line 15 of the code. The `native_` prefix on the name identifies it as a method call. This is very rarely required except in some of the infrastructure code when the DoesNotUnderstand code has not yet been put in place.
3. Most lines call a Smalltalk method on an object. This is seen in line 1 of the listing and line 3 of the code. From this one can see the translation of a Smalltalk method to a Javascript function name: an underscore is prefixed, and every colon is replaced with an underscore. If the object is a Smalltalk object, it will presumably have a field of that name bound to a function and that function will be executed with the object bound to `this` (in Javascript, `self` in Smalltalk). If the object is not a Smalltalk object (as in this case), then the search for the name – `_createElement_` in this case - will fall through to Object, and which has a binding for every such name! This fall-through code looks at the first part of the name (between the first 2 underscores) and tries to handle this as the name of a Javascript field accessor/setter or as a Javascript function. If there is a `slot_` or a `native_` prefix, then it is forced to be a field accessor/setter or function call, respectively. If not, then it checks if there is a field with that name in the object and whether that field is a function. If it is a function then this message send is interpreted as a method call and the Javascript function is invoked on the object with the original parameter list. The appropriate accessor/setter/function is created/aliased to the original name with the undescores so that a subsequent call will complete without all this work next time. Therefore a second call to `document._createElement_` will dispatch directly to the builtin function. If nothing was found, then the original call is bound up into a Message object and the original target object is called at the `_doesNotUnderstand_` method. If that falls through to the default code in Object, then an exception is raised.
4. Lastly, in listing lines 23, 24, and 32 are proxy delete requests indicating that the Pharo side no longer has references to the proxy object and it should be removed from the browser side.

Lines 7–10 and 28–31 in the log are just normal Smalltalk message sends, but line 7 asks the browser to create a reverse-proxy, which it does, returning proxy 2, and then line 9 sets the "click" callback on the button.

These are not the ES6(Van Cutsem and Miller 2010) proxies, although we may use them in the future for implementation.

6. Transpilation to Javascript

The final stage of development is to transpile the complete program from Smalltalk to Javascript and run it in the browser.

The complete stand-alone test suite from §4 should work. The complete browser test suite from §5 should also work, although obviously at higher speed as there is no longer any websocket between the program and the DOM.

The transpilation uses the standard Smalltalk parser to convert Smalltalk classes and methods to an Abstract Syntax Tree, which is then walked to generate the desired Javascript code. The vast majority of Smalltalk code is message sends. In §5.4 we described the translation from Smalltalk message names to Javascript function names. Every message that is sent is recorded so that the browser code can set up the requisite trampolines to handle `doesNotUnderstand` for every possible message. Messages can also be dynamically created and dispatched using `perform:withArguments:`, so before dispatching the message, that method must make sure that there is a trampoline to catch the possible missing method for the message.

The goal of the transpilation is to have the best possible fidelity to the original Smalltalk semantics, while interoperating smoothly with Javascript libraries and the DOM. Choosing optimal Javascript code sequences to implement Smalltalk semantics can be challenging, so we follow the results of (Mason 2015).

To get good performance the Pharo code generator shortcuts many messages and generates more optimal VM code for conditional and looping messages like: `ifTrue:ifFalse:`, `whileTrue:`, and `to:by:do:`. Taking our lead from the Pharo code generator, we also short-cut many of those operations. In Smalltalk, primitive messages such as comparison and arithmetic operations are handled as normal function calls, but the VM has extremely efficient handling for them. In our transpiled code these must be generated as function calls for semantic fidelity reasons unless we know either staticaly or dynamically that they are safe, in which case we optimize them to the standard Javascript operators.

The code that is generated is very readable in most cases. The thing that detracts the most from that readability is the necessity to handle `null` and `undefined` values. In Smalltalk `nil` is an object, like any other and it can be sent messages. In Javascript, `null` and `undefined` are the only things that cannot be sent messages, so every message send has to include a test to convert those values into `nil` as necessary so that messages can be sent directly.

7. Conclusion

In this paper, we looked at the problem of program development, testing and debugging for Javascript web development, including the limitations of Javascript development support on browsers.

We proposed addressing this primarily by developing and testing the code in a more mature IDE while accessing exact DOM semantics on the browser via the use of proxies. The resulting code can then be deployed within the browser because of the high-fidelity translation from our development language (Smalltalk) to production-quality Javascrpt. This provides a rich development environment allowing access to debugging, refactoring, and code analysis tools in the IDE. The only major drawback is the performance of the proxy communication, making certain kinds of interaction (such as tracking mouse movement) difficult to test. There are also a few corner-cases where somewhat stylized coding is currently required.

Looking forward, we would like to improve the performance of the proxy communication and reduce the special corner-cases. We are also working to better integrate with Javascript libraries and package managers. As browsers acquire weak arrays, we will be able to improve the handling of reverse-proxies. As browsers expose APIs for debugging, we would like to integrate those so that support for debugging in the native IDE could extend even further into the development workflow. If we had better information about the types of values we could optimize Javascript operators rather than using method calls, so we are considering use of the Roell typer(Pluquet et al. 2009) to do dynamic type inference.

References

A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 293–302, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: 10.1145/75277.75303. URL http://doi.acm.org/10.1145/75277.75303.

J. Ashkenas. CoffeeScript is a little language that compiles into JavaScript. URL http://coffeescript.org.

K. Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, 2002.

Community. Node-Webkit for desktop applications. URL http://nwjs.io.

GitHub. Electron: Build cross platform desktop apps with web technologies. URL http://electron.atom.io.

S. HQ. Selenium: Browser Automation. URL http://www.seleniumhq.org.

JetBrains. WebStorm Javascript IDE. URL https://www.jetbrains.com/webstorm/.

T. Kim, C. Park, and C. Wu. Mock object models for test driven development. In *Fourth International Conference on Software Engineering, Research, Management and Applications (SERA 2006), 9-11 August 2006, Seattle, Washington, USA*, pages 221–228. IEEE Computer Society, 2006. ISBN 0-7695-2656-X. URL http://dblp.uni-trier.de/db/conf/sera/sera2006.html#KimPW06.

D. Mason. Performance from aligning smalltalk & javascript classes. In *Proceedings of the International Workshop on Smalltalk Technologies*, IWST '15, pages 4:1–4:8, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3857-8. doi: 10.1145/2811237.2811301. URL http://doi.acm.org/10.1145/2811237.2811301.

M. Milani and B. Benvie. appjs: Build desktop applications. URL http://appjs.com.

Mozilla. Firebug: Web Development Evolved. URL http://getfirebug.com.

N. Petton, H. Vojčík, et al. Amber smalltalk. URL http://amber-lang.net.

F. Pluquet, A. Marot, and R. Wuyts. Fast type reconstruction for dynamically typed programming languages. In *Proceedings of the 5th Symposium on Dynamic Languages*, DLS '09, pages 69–78, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-769-1. doi: 10.1145/1640134.1640145. URL http://doi.acm.org/10.1145/1640134.1640145.

T. Van Cutsem and M. S. Miller. Proxies: Design principles for robust object-oriented intercession APIs. In *Proceedings of the 6th symposium on Dynamic languages*, number 12 in DLS '10, pages 59–72, New York, NY, USA, October 2010. ACM. doi: 10.1145/1899661.1869638.

J. Vilk and E. D. Berger. DOPPIO: Breaking the browser language barrier. In M. F. P. O'Boyle and K. Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 52. ACM, 2014. ISBN 978-1-4503-2784-8. URL http://dblp.uni-trier.de/db/conf/pldi/pldi2014.html#VilkB14.