Designing a Live Development Experience for Web-Components

Jens Lincke
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
jens.lincke@hpi.uni-potsdam.de

Patrick Rein
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
patrick.rein@hpi.uni-potsdam.de

Stefan Ramson
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
stefan.ramson@hpi.uni-potsdam.de

Robert Hirschfeld
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

Marcel Taeumel
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
marcel.taeumel@hpi.uni-potsdam.de

Tim Felgentreff
Oracle Labs
Potsdam, Germany &
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
tim.felgentreff@oracle.com

Abstract

Explorative and live development environments flourish when they can impose restrictions. Forcing a specific programming language or framework, the environment can better enhance the experience of editing code with immediate feedback or direct manipulation. Lively Kernel's user interface (UI) framework Morphic provides such a development experience when working with graphical objects in direct way giving immediate feedback during development. Our new development environment Lively4 achieves a similar development experience, but targeting general HTML elements. Web Components as a new Web standard provide a very powerful abstraction mechanism. Plain HTML elements provide direct building blocks for tools and applications. Unfortunately, Web Components miss proper capabilities to support run-time development. To address this issue, we use object migration to provide immediate feedback when editing UI code. The approach is evaluated by discussing known problems, resulting best practices and future work.

PX/17.2, October 22, 2017, Vancouver, BC, Canada

ACM ISBN 978-1-4503-5522-3/17/10…$15.00
https://doi.org/10.1145/3167109

CCS Concepts • **Human-centered computing → User interface programming**; • **Software and its engineering → Development frameworks and environments**; *Software creation and management*;

Keywords Web Components, JavaScript, Live Programming, Web-based Programming Environment

ACM Reference Format:
Jens Lincke, Patrick Rein, Stefan Ramson, Robert Hirschfeld, Marcel Taeumel, and Tim Felgentreff. 2017. Designing a Live Development Experience for Web-Components. In *Proceedings of 3rd ACM SIGPLAN International Workshop on Programming Experience (PX/17.2).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3167109

1 Introduction

Lively Kernel [5] was one of the first Web-based development environments that explored new ways of interactively develop Web applications and active content directly in the browser [8]. In this Smalltalk-like tooling environment, content and applications developed are written in a special way so that they can be modified and evolved while being used. Components not developed in Lively Kernel, such as the visualization library D3[1], could be used, but had to be specially wrapped to be integrated. This wrapping also resulted in a hard border that could not be crossed by the tools and the user interface of Lively Kernel, e.g. a user could not select or directly manipulate an element in a D3 visualization, because it is not part of the Morphic framework.

[1] https://d3js.org/

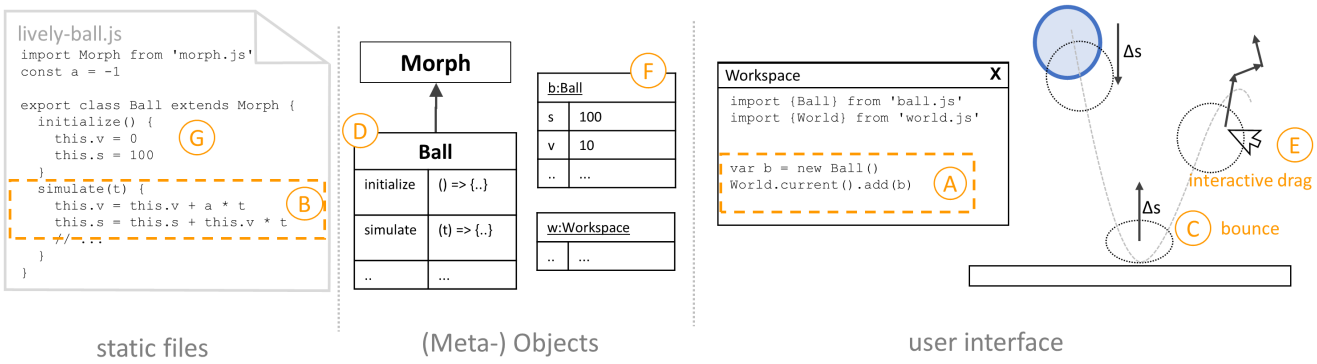

Figure 1. Bouncing ball: when the domain lends itself to it, a programmer can develop some parts of an application with continuous feedback, just by editing code that will patched into the system.

In the new Web-based development environment Lively4, we want to transfer Lively Kernel's explorative and live programming experience to a wider and more used model: plain HTML elements. By building tools that work directly on HTML elements, we are not limited to work on applications and content created in our environment, but can explore and work with all kinds of external resources such as visualization libraries and text editors. Lively4, being a purely client-side development environment, needs just some server or service that serves files[2].

Compared with a clean object composition hierarchy in the Morphic framework, however, a tree of HTML elements can be cluttered with content and presentation elements. To be able to use some kind of abstraction mechanism in the development of our own tools, we decided to use Web Components. Web Components is a new Web standard[3] that enables developers to create custom HTML elements and better separate content from presentation could help express needed abstractions, that previously the Morphs provided. This abstraction mechanism makes Web Components an ideal building material for the new environment, but Web Components have a downside: They are not designed to be updated after they are used, rendering the basic implementation strategy of Smalltalk-like programming, which is modifying meta-objects at run-time, useless in this case.

In this paper, we describe how we applied Web Components in Lively4 and enable modifying them at run-time, preserving the context [16] and giving immediate feedback by migrating all instances of the component under development.

The remainder of the paper is structured as follows: in next section we present a Lively Kernel like development experience, which lacks the capability of directly on HTML elements. We present Web Components as a powerful abstraction mechanism, but which not suited to be worked on at run-time. In section 3, we present our approach of migrating instances of HTML elements instead of updating just single meta-objects. In section 4, based on our experience of developing all tools in Lively4 with Web Components, we discuss open problems, best practices and future work. Section 5 discusses related work and section 6 concludes our thoughts on this topic.

[2]Lively4 uses a custom server for authentication and GitHub access, but it can also be run in principle from any HTML server or service

[3]https://www.w3.org/standards/techs/components

2 Development at Run-Time with Web Components

The Lively Kernel development environment provides an explorative Smalltalk-like development experience, but excels only when working on content created with the Morphic framework. Web Components can provide similar (or better) abstractions as Morphic, but does not lend itself to be developed at run-time.

2.1 Lively Kernel Development Experience

Lively Kernel is best described as a Smalltalk-like development experience in the browser. It provides its own Morphic UI framework [13, Maloney2001IMS] with a rendering abstraction over HTML, SVG, and CSS. Every graphical object (morph) can be directly modified through drag and drop, copied, resized, customized with a style editor and scripted with an object editor. The morph and its connected objects can then be stored in a shared repository, where they can be reused and modified by others [11].

The development experience of graphical tools and applications feels direct and immediate. It allows both, to evolve tools and applications while using them and share adaptations in a direct and informal way [9].

Since in Lively Kernel morphs are JavaScript objects and only rendered as HTML, they maintain their own data structures. Therefore, the underlying HTML can always be discarded and rendered again. That means a world of morphs can be serialized as an object graph. This is a very powerful

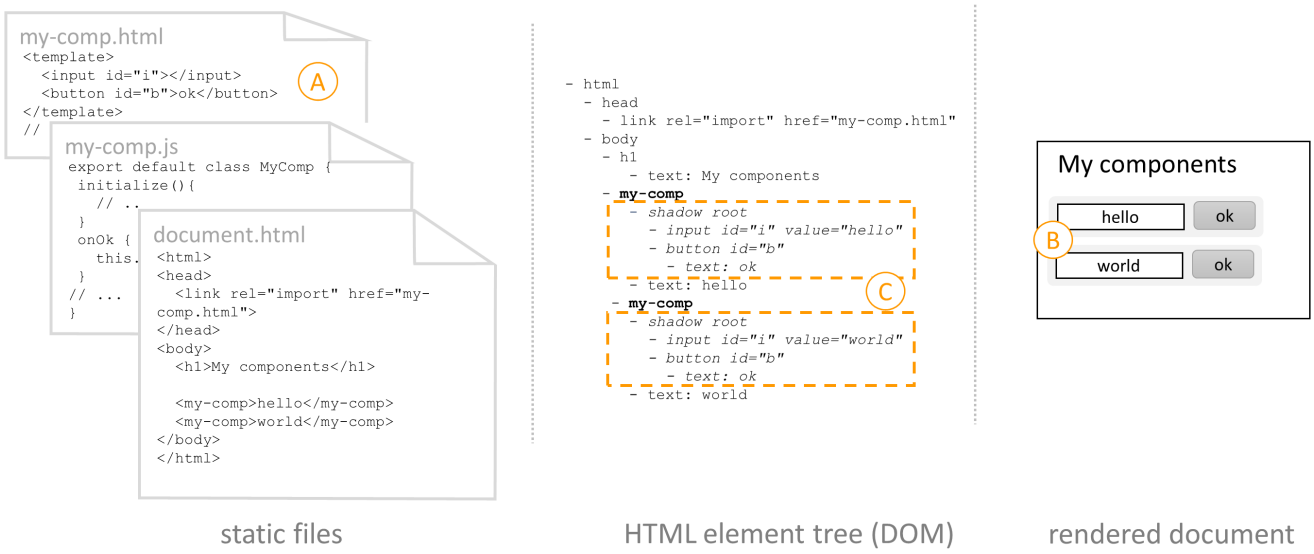

Figure 2. Web Components: the new elements in (B) are defined in (A), but at run-time the template is copied and the structure is redundant (C).

and general mechanism, but it does not play nicely with the way Web browser treat content. The browser usually does not wait for the full content to be delivered, but starts rendering as soon as the first HTML elements arrive. In contrast, in Lively Kernel, the browser has to wait until all serialized content is transmitted and all referenced JavaScript files are loaded. Then the objects are first deserialized and then rendered to HTML, after which the browser can start displaying them.

Because Lively Kernel's Morphic is implemented on the JavaScript object level, it maintains its own graphical composition hierarchy and therefore provides an abstraction over HTML. Through an indirection, each morph is rendered to several HTML or SVG elements, making the Morphic tree structure leaner than the structure of HTML elements in the browser, hiding unnecessary implementation details form the Morphic user.

Since Lively Kernel provides a Smalltalk-like development experience, the degree of liveliness a developer experiences can provide, depends on the program domain. The example in figure 1 show the simulation of ball (A). While working on the simulation source code (B) to modify or add new behavior, such as letting the ball bounce (C), the developer gets continuous feedback, because the new behavior is patched into the Meta-objects (D).

Such a feedback loop is present in most scenarios of live programming. As this example shows, it is not always necessary to have framework or language support that will provide or enforce such a loop. Often times such feedback can be achieved through developers best practices, especially when the domain inherently contains a feedback loop.

Even though Smalltalk-like approaches support the editing objects (E), they only updates the object state (F), but do not propagate the changes back the source code of classes (G).

Lively Kernel deals with this problem by adding the behavior to objects and persist those changes by storing and loading objects with depended objects. However, Lively Kernel limits this programming experience to their own specific objects in their own world (Morphic framework). Because of this, there is a gap between the development experience of Morphic code and objects and code underneath that abstraction. By building tools that work on the level of HTML elements, but allow for the same explorative direct manipulation and live feedback during development, we hope to close that gap.

2.2 Web Components in Modern Browsers

Even though HTML elements can provide a similar substrate for live development, they lack the abstraction mechanism when working with JavaScript objects provided. Even though CSS (cascading style sheets) allows for separating the style from content to some degree, it is not possible separate a bunch of elements and their code into a new widget.

Such an abstraction mechanism in plain HTML is missing: with ongoing development of new Web standards, Web Components bring the ability to compact several HTML elements into one. Web Components are a symbiosis of several new browser technologies, that are still in the process of standardization:[4]

- HTML Imports (Draft 2016-02-25)
- HTML Templates (Standard 2014-03-18)

[4]https://www.w3.org/standards/techs/components, as of 2017-08-09.

- Custom Elements (Draft 2016-10-13)
- Shadow DOM (Draft 2017-02-13)

Together, they allow users to define custom HTML tags. The definition uses an HTML template that has to be imported before being used in an HTML file. The new element can further use other HTML elements that will be hidden in its shadow DOM for the user of the new tag. Web Components are supported in most modern Web browsers and can be emulated using polyfills[5].

Different from the workflow of Lively Kernel, developing with Web Components is by default a typical edit and reload cycle. Web Components force this style of development further, since registering elements can not be undone or overridden.

The abstraction mechanism provided by Web Components is only present on a source code level. As the name *HTML Templates* already suggests, the HTML elements in the template are copied on each usage of that new element. Even though the shadow DOM with its shadow root marks the separation between the custom HTML element and its implementation, its implementation is just a copy of the templates elements in the shadow root. Making the elements in the shadow root forming normal child nodes in the bigger tree allows the browser's event processing and rendering to work as usual.

This is different to the property lookup in JavaScript objects, where the abstraction of having some properties or methods defined in a prototype is still kept at run-time. Changing those properties or methods at run-time will change the behavior of all objects inheriting from that object, without having to modify each object again.

3 Live Web Component Migration

In our Web-based development environment Lively4, we aim to bring the development experience of Lively Kernel to plain development with HTML elements. Developing tools and applications with Web Components introduces abstractions on the HTML level but imposes challenges to make the experience of developing Web Component live and explorative.

3.1 Object Mutation and HTML Element Migration

To achieve short feedback loops and preserve the context during development, we need to avoid full page reloads. We therefore have to update the system behavior and state depending on the code changes at run-time.

The mutation of shared Meta-structures (as shown in figure 1) is not suitable for developing Web Components, because the state and behavior of Web Components depends on the HTML element structure in the shadow DOM. Different to editing a method's source code in a class, the editing of the element structure of a template affects not only one object that can be mutated, but it affects several object structures as shown in figure 2.

Editing a template itself represents changing many HTML elements at once. Even further, as shown in figure 2, this template might have been used in multiple places (all usages of the new custom HTML element) and every element has copies of that template's element structure in its own shadow root. This means even if we could mutate the template's HTML elements in a perfect way, we would have only affected the appearance of new instances, which is not satisfying in live programming.

Our solution is to migrate all existing instances. It is not enough to mutate the linked template element (A), but we have to deal with each instance individually (B). To achieve this, we replace each element with a new instance and migrate all persisted state from the previous instance to it. By default we keep the custom style, properties and child nodes, but elements can specify additional migration behavior by implementing a `livelyMigrate` method.

The approach has the downside of breaking existing references, which will be a problem when we replace internal components that are used and referenced by other components. We plan to address this issue in future work, by either automatically forcing a migration of components that use the component under development, or combining mutation and migration in away that we keep the old instance, but replace the private properties and element in the shadow DOM of the component.

3.2 Different Kinds of Feedback in the System

Even though a Lively Kernel-like development experience can give feedback of the behavior in a running system, not all behavior has immediate feedback. Figure 3 exemplifies such kinds of feedback in a simulation of a soap bubble. The bubble extends the balls behavior in several ways. It will burst after a time has passed (A), the user touched it (B), or it collides with the ground (D). When it collides with another bubble, the bubbles will be combined into a new and bigger one (C). All this behavior is explorable at run-time, but may need some interaction, waiting or manual setup. The environment can only support the developer in not having to perform those interaction-, waiting- and setup-tasks over and over again. Record and replay techniques [15] can automate those repetitive task and make the feedback immediate, but it is the developer's task to invoke the desired behavior. In Lively4, we strive to support programming at run-time as much as possible, not to immediately give feedback to every line a programmer typed, but to provide programmer with the freedom to explore and adapt the running tools and applications as needed.

4 Discussion

Lively4 uses Web Components for all its UI. All tools such as the file editor and internal content browser are custom

[5] https://www.webcomponents.org/

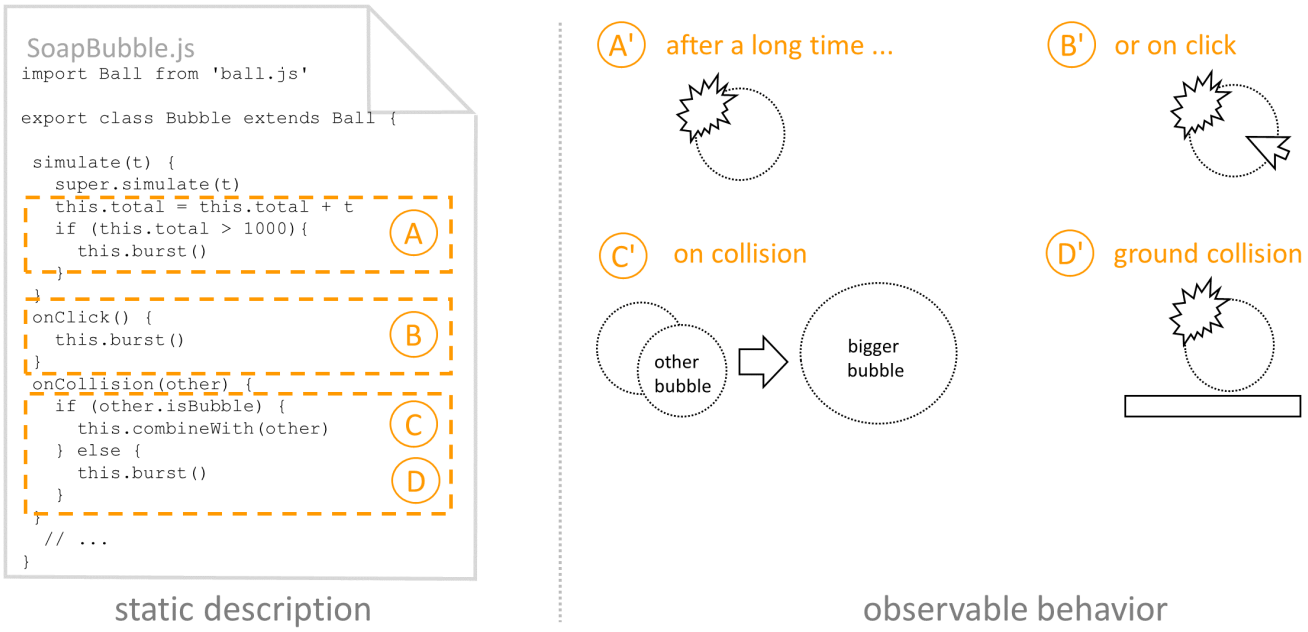

Figure 3. The static description of a soap bubble and its resulting behavior at run-time.

HTML elements. During this development we enjoyed the experience of being able to work with HTML elements and have a Lively Kernel like feedback at run-time. But we also encountered problems and developed best practices and insights for future work.

4.1 Lively4 Development Experience

The Lively4 development environment[6] is build using Web Component at its core. The figure 4 shows three windows: a browser/editor, an inspector, and the GitHub sync tool. The browser shows the source code of the sync tool's module. Editing and accepting code there will automatically update the open instance of the sync tool. Editing this `initialize` method to change to window title will also update the window title of the sync tool. This is possible because our migration approach takes care of also executing the initialization code, so that shared state that is not persisted will also update.

This inclusion of initialization code into the feedback loop is a major contribution compared with Smalltalk-like development approaches.

4.2 Stale Code and Dangling Event Listeners

Changing code in a live system will often produce new behavior and new objects, but it also results in some objects, methods or reference to become obsolete. A good example for such *stale code* [2] are dangling event listeners. A framework has to take care of not letting the old code and behavior get in the way of the new one. A developer team that used Lively4 to build an *Exposé* window management feature[7] run into such a problem. Their component registered itself for global key or mouse events, which resulted in problems when its new version registered handlers again. Their approach to deal with the issue was to fall back into a full page reload development workflow, taking no advantage of the faster feedback loop of run-time development. After this, we established a best practice to use an event registering mechanism that allows us to clean up and unregister old listeners automatically.

4.3 JavaScript Objects and HTML Elements

The new abstraction mechanisms of Web Components allows us to maintain a cleaner, more domain-specific data structure, allowing us to push such structures and not JavaScript objects in the center of the programming of applications and tools in Lively4. We still use JavaScript objects, classes and modules to describe the behavior of tools and applications in Lively4, but, when developing the UI, they are second class citizens and they will not be persisted. We decided to use just HTML as the format to preserve the state of our tools and applications. Compared with Lively Kernel, we switched the double-sided coin of JavaScript objects and HTML Elements in favor of the HTML Elements. In Lively4, the JavaScript objects are second class citizens and we throw them away

[6] The current Lively4 system is hosted under https://lively-kernel.org/lively4/lively4-core/start.html and used from there to directly work on its own GitHub project https://github.com/LivelyKernel/lively4-core/ in a self-supporting way.

[7] *Exposé* shows an overview of all windows side by side

when we need to, relying on the fact that the HTML Element will persist the application state and preserve the context for development.

4.4 Losing Object Identity

In our current approach, we replace HTML elements with new versions of themselves, creating a new JavaScript reference in the process. We take care of updating some known JavaScript references, but we do not have full control over all aliases that could for example be bound in closures. As a best practice, we do not rely on object identity in our own programs and use names or IDs to look up elements at runtime. In future work, we plan to combine object mutation and migration, so that the JavaScript references stay intact.

4.5 Dynamic Elements in Static Templates

A Web Component is not only defined through pure HTML template, but can contain arbitrary script tags or use an external JavaScript module (as shown in figure 2). This allows the developers to use the very static element structure in the template to describe the static UI, and use JavaScript to generate more dynamic UI elements as needed. This is clearly not a unified way to describe UIs, but it seems to be a preferred way of many Web developers. This is not an issue of Lively4 since it does not set out to invent a new programming- or UI description language. Further, we do not often run into this issue since the major parts of the UI, that we developed for our tools and applications, were static.

4.6 Level of Preserving Context

Our approach of using our custom HTML persistence for migrating Web Component instances to preserve the context while developing has a catch: the Web Component might not persist all its relevant state that the developers expects, yet. This is especially apparent because Lively4's white listing object persistence approach highly differs from the *serialize everything* and black list later approach in Lively Kernel and Smalltalk images. Our approach to deal with this issue is to develop the persistence (loading and saving behavior) of Web Components in parallel with other features.

4.7 Changing of a JavaScript Module

Web Components may depend on JavaScript modules. Since those modules can affect custom HTML elements, it is not clear what happens when those modules change. We decided to treat such changes similar to changing the source of Web Components directly. In particular, tools check for dependencies and update all dependent Web Component instances. In rare cases, some core modules will affect the whole system and will trigger a complete migration of nearly all Web Components, causing the system to nearly fully reload. We experimented with maintaining a blacklist for modules that should not update all dependents, but opted for manually disable dynamic feedback as needed.

4.8 Problems of Combined Run-Time- and Development Environments

Evolving a running system from inside can lead to some challenging situations, e.g. when trying to debug something that is used by the editing tools. We experimented with using Context-oriented Programming [10] to scope the changes in development layers [9], when we were working on JavaScript modules.

While working on Web Components, this approach was not enough. But we kept our environment working through making use of the per instance migration. This allowed us further to exclude special objects as needed. In the bootstrapping phase of developing the `lively-editor` and `lively-container` components, we excluded specially marked instances so making an error while working on the editor code would not break the whole system since there was still a working editor around that used an stable template. Later we disabled this because we seldom ran into such problem. If we did, we had a second system to fall back to, so we could fix the first one.

5 Related Work

Squeak/Smalltalk [3][6] with its explorable world of objects and its Morphic framework [12] already features a programming at run-time experience, that many systems are still not capable of today. Free from the security and design limitations of current Web browser technology, the developers have full control over all objects. All objects are persisted in an image, can be mutated and even replaced. Objects can `become` other objects, making object migration trivial. Since code can be changed and debugged at run-time, a live development experience can be achieved in case an application continuously uses the code under development. Different to our approach, the (UI) initialization code in Smalltalk is not continuously executed, giving no feedback when changing it.

Lively Kernel [7][8][5] was used to continuously evolve itself in a self-supporting manner, experimenting with various approaches of providing a direct manipulation experience and live feedback. Even though it runs in the browser and can use all kinds of JavaScript libraries, its tools and workflow only excel when working on content created in and for Lively Kernel. Creating a similar experience but with a broader reach by targeting plain HTML is part of the motivation of this paper.

Live programming [17][4][14] is a development experience that brings editing static source code and its dynamic behavior closer together. The implementation strategies to achieve immediate feedback while programming depends a lot on the application domain and restrictions of the used programming language. Implementing a feedback mechanism for editing code that expresses just a functional view is

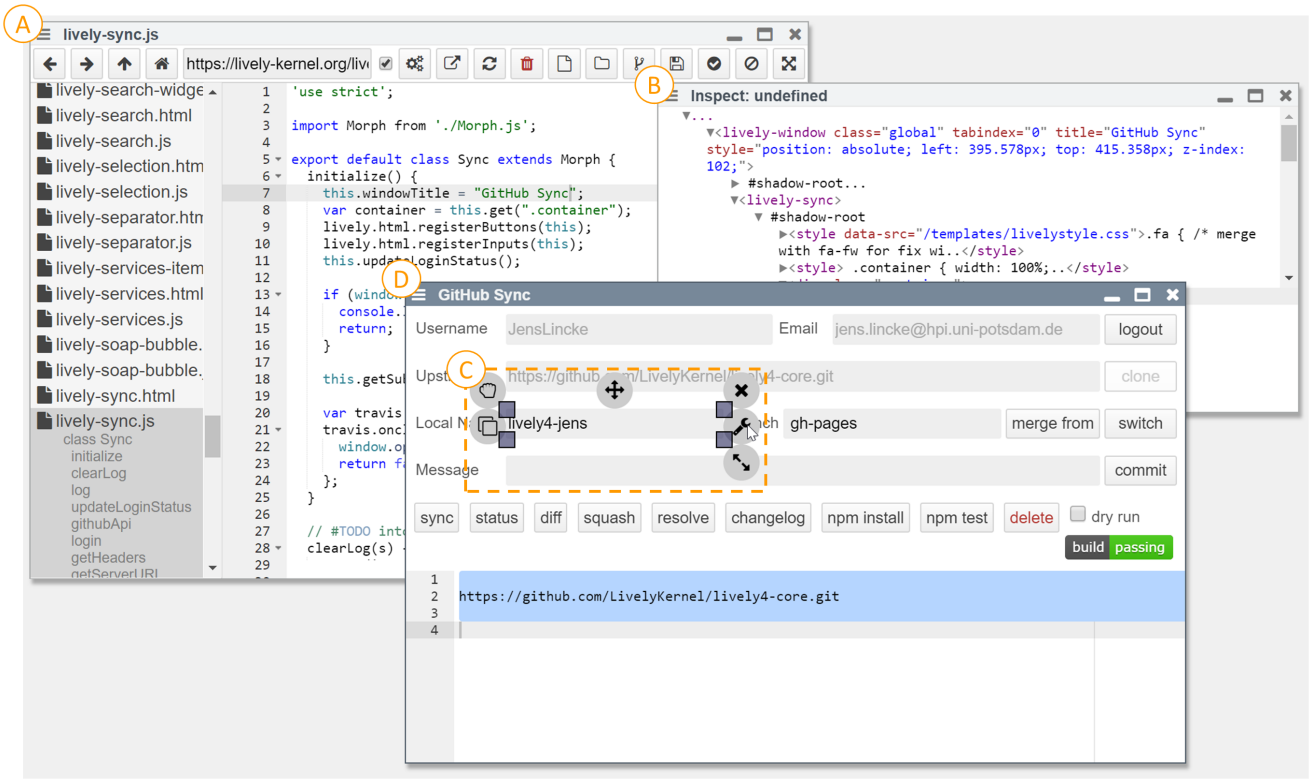

Figure 4. Tools in Lively4: (A) code browser and editor, (B) Element and object inspector (C) Halo for direct manipluation, (D) GitHub version control tool

trivial, providing the same experience in an object-oriented world were objects and meta-objects live together, is much harder. Giving live feedback while imperatively constructing UIs is easier to achieve and often studied. In "it's alive!", the approach is to separate "UI state from ordinary state, and the render code that builds UI state from ordinary code" [2]. In our approach, we also take advantage of having different kinds of state: we treat HTML elements and attributes differently from JavaScript object state. In Lively Kernel and other environments, objects hold the model and the view is often thrown away and rendered again. In our approach we keep the elements defining the view and throw away the objects

Cascading Tree Sheets [1] address the problem of separating content and presentation, where Cascading Style Sheets (CSS) do not go far enough. This approach allows programmers to add HTML elements that are not part of the actual content, but needed for the presentation and UI later by inserting HTML elements and scripts based on CSS selectors. This approach is not suited for run-time development, because at run-time, the separation is not there any more. Web Components share a similar motivation, but preserve the separation between content and presentation elements even at run-time by keeping private elements hidden in each element's shadow DOM.

6 Conclusion

Lively Kernel, as the Smalltalk-like Web-based development environment, provides an explorative and live development experience based on a JavaScript Morphic framework. In this paper, we discussed the problem of providing a similar experience when working with plain HTML elements and how we solved it by using Web Components in live programming experience.

As our key contribution, we showed how we allow developers to modify the source code (both template and class of the HTML element) of Web Components to be changed at run-time: in our approach, we control the instantiation of each Web Component and use the latest template and class for new usages of the Web Component. Existing instances are migrated by replacing the old instance with a new instance. This includes going through the whole initialization code and applying the persisted state, such as attributes, external styles and external child nodes.

Striking the balance in the ongoing self-supportive development of Lively4 between practical usability and a live and explorative programming experience can cause friction, but also provides interesting challenges for future work.

References

[1] Edward O. Benson and David R. Karger. 2013. Cascading Tree Sheets and Recombinant HTML: Better Encapsulation and Retargeting of Web Content. In *Proceedings of the 22Nd International Conference on World Wide Web (WWW '13)*. ACM, New York, NY, USA, 107–118. https://doi.org/10.1145/2488388.2488399

[2] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. 2013. It's Alive! Continuous Feedback in UI Programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 95–104. https://doi.org/10.1145/2491956.2462170

[3] Adele Goldberg. 1984. *SMALLTALK-80: The Interactive Programming Environment*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[4] Chris Granger. 2012. Light Table. Sofware. (2012). http://www.lighttable.com/

[5] Daniel Ingalls, Tim Felgentreff, Robert Hirschfeld, Robert Krahn, Jens Lincke, Marko Röder, Antero Taivalsaari, and Tommi Mikkonen. 2016. A World of Active Objects for Work and Play: The First Ten Years of Lively. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2016)*. ACM, New York, NY, USA, 238–249. https://doi.org/10.1145/2986012.2986029

[6] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. *ACM SIGPLAN Notices* 32, 10 (1997), 318–326. https://doi.org/10.1145/263700.263754

[7] Daniel Ingalls, Krzysztof Palacz, Stephen Uhler, Antero Taivalsaari, and Tommi Mikkonen. 2008. The Lively Kernel A Self-Supporting System on a Web Page. In *S3 2008 (LNCS 5146)*. Springer-Verlag Berlin Heidelberg.

[8] Robert Krahn, Dan Ingalls, Robert Hirschfeld, Jens Lincke, and Krzysztof Palacz. 2009. Lively Wiki A Development Environment for Creating and Sharing Active Web Content. In *WikiSym '09*. ACM.

[9] Jens Lincke. 2014. *Evolving Tools in a Collaborative Self-supporting Development Environment*. phdthesis. Universität Potsdam. https://lively-kernel.org/publications/media/Lincke_2014_EvolvingToolsInCollaborativeSelfSupportingDevelopment\Environment_PRINT.pdf

[10] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. 2011. An Open Implementation for Context-oriented Layer Composition in ContextJS. *Science of Computer Programming* (2011). https://doi.org/DOI:10.1016/j.scico.2010.11.013

[11] Jens Lincke, Robert Krahn, Dan Ingalls, Marko Röder, and Robert Hirschfeld. 2012. The Lively PartsBin–A Cloud-Based Repository for Collaborative Development of Active Web Content. In *Hawaii International Conference on System Sciences*. IEEE Computer Society, Los Alamitos, CA, USA, 693–701. https://doi.org/10.1109/HICSS.2012.42

[12] John Maloney. 2001. An Introduction to Morphic: The Squeak User Interface Framework. *Squeak: OpenPersonal Computing and Multimedia* (2001).

[13] John H. Maloney and Randall B. Smith. 1995. Directness and Liveness in the Morphic User Interface Construction Environment. In *UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology*. ACM, 21–28. https://doi.org/10.1145/215585.215636

[14] Sean McDirmid. 2007. Living It Up with a Live Programming Language. *SIGPLAN Not.* 42, 10 (Oct. 2007), 623–638. https://doi.org/10.1145/1297105.1297073

[15] Sean McDirmid. 2013. Usable Live Programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming (Onward! '13)*. ACM, New York, NY, USA, 53–62. https://doi.org/10.1145/2509578.2509585

[16] Patrick Rein, Robert Hirschfeld, and Marcel Taeumel. 2016. Gramada: Immediacy in Programming Language Development. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2016)*. ACM, New York, NY, USA, 165–179. https://doi.org/10.1145/2986012.2986022

[17] Bret Victor. 2012. Learnable programming. (2012), 2012.