Stephen Oney | Research Statement

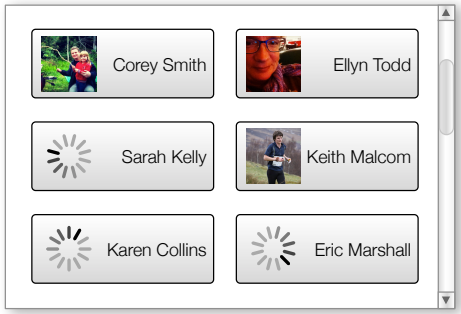
Computing is an indispensible tool in the 21st century, yet too few people know how to wield its power to augment their everyday lives. My research enables and encourages novice, everyday users to write and customize computer programs. To achieve this end, I build novel systems, programming paradigms and interaction techniques, drawing on a variety of disciplines. This includes *computer science*, *design*, and *software engineering* research, which guide solutions that are expressive, natural, and sound. *Psychology*-inspired studies help investigate how developers and non-developers naturally think of programming concepts. *Human-computer interaction* (HCI) techniques help evaluate and refine this solution. For my dissertation research, I focused on creating user interface (UI) development tools for programmers and on enabling end-user programmers to specify UI behavior.

Enabling and Improving User Interface Development

Designing a good UI requires more than carefully arranging the graphical elements that define its appearance; it also requires specifying the *interactive behaviors* that determine how the UI reacts to user and system events. Although there are many effective tools that allow designers to specify a UI's appearance, defining its behavior is costly, error-prone, and typically limited to expert developers. This is because the only way to define a custom UI behavior is by implementing it in code, with programming languages and frameworks whose features do not address many of the fundamental challenges of UI development.

In many ways, UI development is different from general-purpose programming. Most widely deployed UI frameworks use an event-callback programming model, where developers write imperative *callbacks* that determine how the user interface should react to every *event*. Event-callback code has an execution order that is often difficult to predict or understand because a typical behavior involves many potential user events, which splits the implementation of every behavior across many callbacks. Further, these callbacks typically reference and modify the same variables, which makes reasoning about their interactions even more difficult. Conceptually, much of the UI code developers write is dedicated to maintaining relationships between UI elements and underlying data models. However, this has been shown to be difficult in event-callback code [4].

For my dissertation, I designed, created, and evaluated a programming framework where the design was inspired and informed by non-programmers' natural descriptions of UI behavior [5]. This framework facilitates UI development by making *constraints*—relationships that are declared once and automatically maintained—more expressive by combining them with *state machines*—a mechanism to track and control a UI's status. These constraints help maintain the complex relationships between UI that are difficult to track in event-callback code. I have implemented this framework in two tools: *ConstraintJS* [3], a library for Web programmers, and *InterState* [4], an interactive graphical editor for end-user programmers.




```
1  friends = cjs.async(fb_request("/me/friends"));
2  pics    = friends.map(function(friend) {
3               return cjs.async(fb_request("/" + friend.id
4                                           + "/picture"));
5            });
6  //...
7
8  {{#diagram friends.state}}
9     {{#state pending }} Loading friends...
10    {{#state rejected}} Error
11    {{#state resolved}}
12       {{#each friends friend i}}
13          {{#diagram pics[i].state}}
14             {{#state pending }} <img src = "loading.gif" />
15             {{#state resolved}} <img src = "{{pics[i]}}" />
16             {{#state rejected}} <img src = "error.gif"   />
17          {{/diagram}}
18          {{friend.name}}
19       {{/each}}
20 {{/diagram}}
```

Figure 1: The ConstraintJS code on the right produces the interface on the left. Here, the Facebook API is called (asynchronously using `fb_request`) to fetch a list of friends (line 1) and a profile picture for each friend (lines 2—5). These values are placed into the `friends` and `pics` constraint variables respectively. Lines 8—20 declare a template that depends on these variables. As the list of friends is loading, `friends.state` will be `pending`, so the message "Loading friends…" is displayed (line 9). After the list of friends has loaded (lines 11—19) the pictures for all friends are displayed alongside their names. While the application is waiting for the Facebook API to return a picture URL for a friend, a loading image (`loading.gif`) is displayed (line 14). The code also correctly notifies the user of any errors (lines 10, 16).

Assisting Developers in Defining User Interface Behaviors for the Web

ConstraintJS (see Figure 1) is a library that provides constraints that can be used to control the content and appearance of Web pages, and integrates these constraints with HTML, CSS, and JavaScript. ConstraintJS is designed to take advantage of the declarative syntaxes of Web languages: it allows the majority of an interactive behavior to be expressed concisely in HTML and CSS, rather than requiring large amounts of imperative code. ConstraintJS improves UI development and contributes to the constraint literature by integrating the notion of state into its constraint system, which allows developers to write constraints that only hold at specific times. This model allows developers to create many interactive behaviors using only state machines and constraints, without JavaScript.

ConstraintJS also addresses other difficult aspects of Web programming. For instance, whenever a Web application references a third-party service, it makes an *asynchronous* call: a request that does not provide a return value right away, but instead uses a callback to provide the return value at some later time. For developers, dealing with asynchronous calls is challenging because it further complicates their application's control structure. ConstraintJS is well suited to handling asynchronous values because it automatically propagates dependencies when values become available [3], which was previously left to developers. I have demonstrated ConstraintJS's expressiveness through a series of example applications [3], including a custom cursor accessibility feature, a dynamic and customizable visualization, and a touchscreen-based photo viewing application. Other developers have also successfully incorporated ConstraintJS into their projects.

Lowering the Floor for User Interface Programming

InterState (see Figure 2) aims to enable end-user developers to write UI code by presenting code in a way that is easier to understand and modify. InterState extends the ideas behind ConstraintJS in four primary ways. First, it introduces a *computational model*

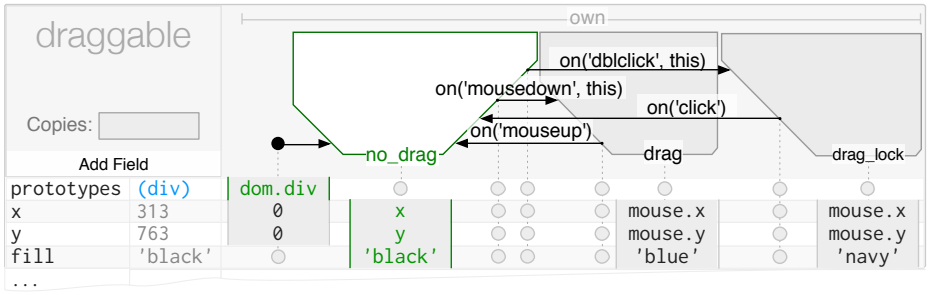


Figure 2: An InterState object, named `draggable`. Properties that control `draggable`'s display (here, `x`, `y`, and `fill`) are represented as rows. States and transitions are represented as columns (e.g. `no_drag`, `drag`, and `drag_lock`). An entry in a property's row for a particular state specifies a constraint that controls that property's value in that state. Here, while `draggable` is in the `drag` state, `x` and `y` will be constrained to `mouse.x` and `mouse.y` respectively, meaning `draggable` will follow the mouse.

that allows developers to express UI behaviors using simple expressions—like spreadsheet equations—that define constraints and transitions between states. This reduces the number of control structures that new programmers need to learn to write UI behaviors.

Second, it introduces a *visual notation* that allows programmers to better understand what user events affect a particular property or, conversely, what properties are affected by a particular user event. In traditional event-callback code, this can be difficult because code is typically distributed throughout multiple locations. In contrast, InterState's visual notation concisely represents interactive behaviors as a table where the rows are properties and columns are states. Combined with its computational model, the visual notation aids programmers by allowing them to see which events affect a property by scanning the property's row and which properties an event affects by looking at that event's column.

InterState also includes built-in mechanisms for *behavior re-use*. Programmers often want to reuse, combine, and inherit behaviors, but nearly every widely used programming language only allows properties and methods to be inherited. InterState's inheritance mechanism allows interactive behaviors to be re-used as easily as properties and fields.

Finally, InterState includes a *live editor* that enable quick experimentation and parameter tuning by removing the edit-compile-run evaluation cycle. In InterState, edits are immediately reflected in the running application and changes in runtime state and property values are highlighted in the editor. This helps bridge the "gulf of evaluation" in determining the effects of a change, which has been shown to be a significant barrier for experienced and new programmers alike. The live editor also allows the developer to always have a running application by "localizing" errors. This means that *only* the parts of the program that depend on problematic expressions are not executed, which avoids deterring new programmers with dozens of syntax and runtime errors.

InterState's design has benefited from several rounds of evaluations, iterative design, and collaboration with undergraduate researchers. A comparative laboratory study indicated that even developers with JavaScript experience were significantly faster at understanding and modifying UI code in InterState compared to using JavaScript [4].

Improving Traditional Programming

Beyond creating development frameworks, I have also explored ways to improve traditional development environments. For example, I developed *Codelets* [2] to improve the usefulness of example code snippets. Developers of all skill levels frequently use instructive code examples found on the Web to overcome cognitive barriers while programming. These examples couple the concrete functionality of code with rich contextual information about how the code works. To use these examples, developers must understand, configure, and integrate the example with their existing code base. However, all of this typically happens *after* the example enters the developer's code and has been removed from its original instructive context.

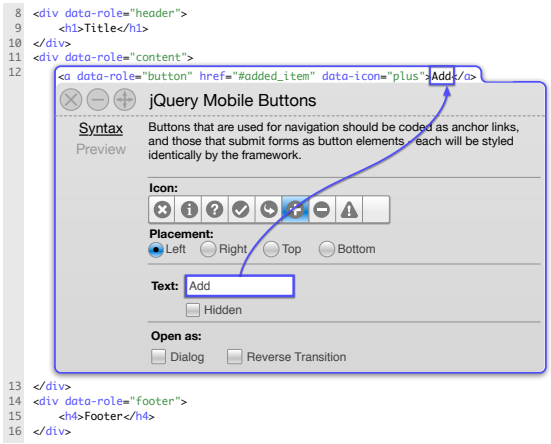


Figure 3: Codelets attach helpers to regions of example code. Helpers are displayed inline with the rest of the code without obscuring it. This Codelet creates a jQuery Mobile button. A developer can customize the button with an interactive form and the Codelet draws a line to illustrate which region of code each form element is controlling. The developer may also directly edit the example code and the form will automatically update itself.

Codelets allow developers to keep the rich contextual descriptions of example code after that code has been pasted into their editor by tightly integrating documentation into the development environment. A Codelet is presented inline with the developer's code, and consists of a block of example code and an interactive helper widget that assists the user in understanding and integrating the example (Figure 3). The Codelet persists throughout the example's lifecycle, even after configuration and integration is done. Codelets are effective in helping developers integrate examples into their code. In a comparative laboratory study with 20 participants, programmers were able to complete tasks involving examples an average of 43% faster when using Codelets compared to using a standard Web browser. Programmers also rated their understanding of the example code as higher when using Codelets.

Future Research Agenda

Many of the principles and findings of my previous research are applicable in new domains, which I look forward to investigating as my future research.

Enabling Casual and Exploratory Programming

We conventionally see "programming" as a solitary activity, done in a text editor on a desktop computer. However, it is sometimes useful to create programs to explain or explore concepts. Just as we sometimes use drawings to communicate an idea, we may use programming to explain more dynamic ideas. Supporting this kind of "on-the-fly" programming will require rethinking the environments and primitives we use for development. With its focus on live development and simplifying programming models, InterState is a promising step in this direction.

Improving Programming Education

Many of the principles behind InterState have the potential to help programmers learn more traditional languages. For example, by immediately updating the running application in response to edits and vice-versa, InterState allows developers to better understand the effect of their changes. This kind of feedback would also be beneficial to students who are learning programming.

Supporting Emerging Computing Platforms and Modalities

The ideas and techniques behind InterState can also be extended to new UI platforms, including touch and gestural interfaces. Touch is an interesting space because it requires wholly new and often custom techniques to maximize the display space for applications. For this reason, touch applications often use non-standard and custom widgets and gestures that need to be manually created or tuned. As new computing platforms and modalities emerge, I look forward to applying my experience to address the difficulties of allowing newcomers to program and customize them.

Integrating Programming with Creative Tools

I also plan on exploring ways to bridge the gap between designers and developers by integrating development into creative tools, such as sketching or mockup creators that make it easy to define an interface's *look*. I have conducted preliminary investigations into this idea, with an early-stage mockup tool that integrates with Photoshop [1].

Impact

I place an emphasis on addressing research questions that are relevant to real-world problems. I have publicly released my development tools, which has created a "living laboratory" that allows me to gather valuable feedback from outside the research space. Other developers are starting to successfully integrate ConstraintJS into their JavaScript projects. InterState is also now available as an open source project and users have started sending feedback and feature suggestions. Additionally, some of the concepts behind Codelets have been integrated into Adobe's Brackets code editor. I look forward to continuing to work on high-impact problems in my future research.

References

1. Oney, S., Barton, J., Myers, B., Lau, T., and Nichols, J. Playbook: revision control and comparison for interactive mockups. In *End-User Development*. Springer, 2011, 295–300.
2. Oney, S. and Brandt, J. Codelets: linking interactive documentation and example code in the editor. *ACM CHI* 2012, 2697–2706.
3. Oney, S., Myers, B., and Brandt, J. ConstraintJS: Programming Interactive Behaviors for the Web by Integrating Constraints and States. *ACM UIST* 2012, 229–238.
4. Oney, S., Myers, B., and Brandt, J. InterState: A Language and Environment for Expressing Interface Behavior. *ACM UIST* 2014, 263–272.
5. Ozenc, F.K., Kim, M., Zimmerman, J., Oney, S., and Myers, B. How to support designers in getting hold of the immaterial material of software. *ACM CHI* 2010, 2513–2522.