HotDrink

A Library for Web User Interfaces

John Freeman
Texas A&M University
jfreeman@cse.tamu.edu

Jaakko Järvi
Texas A&M University
jarvi@cse.tamu.edu

Gabriel Foust
Texas A&M University
gfoust@cse.tamu.edu

ABSTRACT

HotDrink is a JavaScript library for constructing forms, dialogs, and other common user interfaces for Web applications. With HotDrink, instead of writing event handlers, developers declare a "view-model" in JavaScript and a set of "bindings" between the view-model and the HTML elements comprising the view. These specifications tend to be small, but they are enough for HotDrink to provide a fully operational GUI with multi-way dataflows, enabling/disabling of values, activation/deactivation of commands, and data validation. HotDrink implements these rich behaviors, expected of high-quality user interfaces, as generic reusable algorithms. This paper/tool demonstration introduces developers to the HotDrink library by stepping through the construction of an example web application GUI.

The library is a concrete realization of our prior work on the "property models" approach to declarative GUI programming. To encourage adoption among developers, we have packaged the technology following established web programming conventions.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*User interfaces*; D.2.13 [**Software Engineering**]: Reusable Software—*Reusable libraries*

General Terms

Design, Algorithms, Languages

Keywords

Web, user interface, declarative programming, MVVM pattern

1. INTRODUCTION

Programming the user interface (UI) of an application tends to be a significant effort. One study observed that in an average application the UI implementation comprises 48% of all code [14]; another sampling found 30% as the average [10]. These numbers are high but perhaps not surprising; UI implementations often require large amounts of application specific code that is difficult to reuse.

The main service offered by a typical *graphical UI* (GUI) library is to translate user actions to *events* and deliver the events to the correct *event handler* functions. A UI programmer writes the event handlers and registers them to listen to particular events. This is the dominant Web UI programming model as well; the behavior of a UI is defined by the program logic in event handlers.

While the programming community has grown to accept the event handling programming model, alternatives have been and are being investigated. We identify two trends: (1) replacing imperative event handlers with declarative *data-flow constraints* and (2) design patterns that help programmers separate presentation from data from logic.

Regarding the first trend, data-flow constraint systems for UIs were studied actively a few decades ago [5, 13, 16], and have found uses, e.g., in the layout/placement of UI elements. Several recent Web programming frameworks [2, 1, 3] also utilize constraints, typically with the Observer pattern [8, §5], to manage dependencies among values displayed in a UI's widgets.

Regarding the second trend, the progression of popular design patterns—the *Model View Presenter* [15], *Presentation Model* [6], and *Model View ViewModel* [9] (MVVM)—reflects the quest for better separation of concerns in UI code. In the most recent of these patterns, MVVM, a goal is to make the *view* (i.e., a collection of widgets) devoid of logic, retaining only the tasks of receiving events from user actions and displaying pieces of data from the view-model. The *view-model* maintains the state of the UI, that is, the data closely related to what is displayed in the widgets, which may reflect a portion of the data held in the model. The *model* is the single authority on the "business" data and logic for the application, as in the traditional *Model View Controller* pattern [12].

HotDrink employs our *property models* approach [10, 11, 7] for declarative user interface programming. It combines the above two trends: views in HotDrink contain no program logic (typically they are just declarations of HTML widgets) and all of the UI state is contained in a view-model realized as a *multi-way data-flow constraint system*. The result is concise and clear UI code—HotDrink hides all event handling code from the application programmer and implements many complex UI behaviors (value propagation, wid-

GPCE'12, September 26–27, 2012, Dresden, Germany.

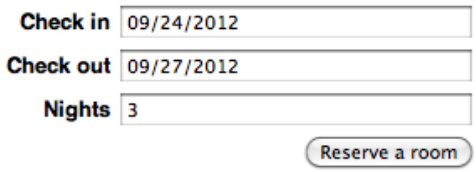

Figure 1: Form for selecting a duration of a hotel stay.

get enablement/disablement, command activation/deactivation, and more) as reusable algorithms.

This paper/tool demonstration describes the main features of HotDrink. We illustrate the programmer's view—the kinds of specifications a programmer needs to write in order to create a GUI—by walking the reader through the implementation of an example Web UI.

Finally, HotDrink is designed to allow incremental adoption. If so desired, parts of a user interface can be controlled by HotDrink while other parts are controlled directly through JavaScript event handling functions or other GUI frameworks. HotDrink adds a few methods to the native prototypes for Object and Array, but we do not anticipate these additions to conflict with other libraries.

2. PROGRAMMER'S VIEW BY EXAMPLE

Figure 1 shows one section of a typical hotel reservation form. The reservation service needs *check-in* and *check-out* dates to query room availability. By adding the (number of) *nights* field, the UI offers three ways to specify these dates—from any two fields, the third can be derived. Here we describe the steps to build this interface with HotDrink and the Model-View-ViewModel pattern. The implementation is available at hotdrinkjs.com/date-example.

2.1 Model

The model itself is outside the scope of HotDrink. For a hotel reservation application, it might consist of a database of rooms and bookings with algorithms for common queries, such as checking availability and adding or removing a booking. HotDrink presumes a model exists and helps only to layer a user interface on top of it. It is the responsibility of the programmer to handle bindings between the model and the view-model, i.e., to move data to one in response to changes in the other. HotDrink provides notifications of changes in the view-model to support this task.

2.2 View-Model

The HotDrink API provides what is essentially an embedded domain-specific language (in JavaScript) for defining view-models. In this language, programmers declare variables and relationships among them which together comprise the view-model's constraint system.

All aspects of the API live in the hd namespace. Figure 2 shows the specification of the view-model for the hotel reservation form.

Variables hold values. A variable can be read by calling it as a function with no arguments or written by calling it with the new value as the only argument. The example in Figure 2 has three variables: one each for nights stayed, check-in date, and check-out date. Each variable holds an integer. For the two date variables, it denotes milliseconds

```
1  var nights = hd.variable(3);
2  var checkIn = hd.variable(new Date().setHours(12, 0, 0, 0));
3  var checkOut = hd.variable();
4  var day = 1000 * 60 * 60 * 24; // 1 day in ms
5  hd.constraint()
6    .method(checkIn, function () {
7      return checkOut() - day * nights(); })
8    .method(checkOut, function () {
9      return checkIn() + day * nights(); })
10   .method(nights, function () {
11     return Math.round( (checkOut()-checkIn()) / day ); });
```

Figure 2: The view-model for the form in Figure 1.

since the Unix epoch.[1] The first two variables are initialized with the values passed to hd.variable.[2] The last will be initialized from the other two during the first update (described below) of the view-model.

Constraints express a relationship among variables. Each constraint consists of a set of *methods*, and each method is a procedure that satisfies the constraint—i.e., establishes the relationship—by computing new values for some of the constraint's variables using the others as input. In our example, the definition of the sole constraint starts on line 5 and specifies three methods. Each method definition lists the set of variables to which it outputs and defines the method's procedure with an unnamed function.

HotDrink requires programmers to explicitly specify each method of a constraint. For some relationships, e.g., equations with simple arithmetic, it would be possible to derive every method—but not for all relationships. Every partition of a constraint's variables into sets of inputs and outputs defines a *direction* for solving, and not all directions may be possible for a relationship, e.g., a one-way hash.

The variables and constraints together form a multi-way data-flow constraint system. To *solve* a constraint system is to execute exactly one method of each constraint, in an order that does not invalidate previously satisfied constraints. We provide a more detailed account elsewhere [10, 11].[3]

Updating the view-model solves the constraint system and sends out notifications of every variable change. An update can be triggered by adding a variable or constraint, by writing a variable, or by an explicit call to hd.update. Event handlers subscribed to the notifications handle the bindings from the view-model to the model, as discussed above, and from the view-model to the view, as discussed below.

HotDrink supports incremental construction of the underlying constraint system—new variables and constraints can be added to a system already bound to a view—which allows the view-model to grow along with a dynamic user interface.

2.3 View

The view, like the model, is outside the scope of HotDrink. Views may be constructed from standard HTML elements

[1]This representation allowed for concise code examples.

[2]The check-in date's initializer computes today's date and sets the time to noon. The time must be dealt with because the standard date manipulation facilities only provide a combined date-time type.

[3]The constraint system is completely separable from the view bindings, and could be reused for other applications.

```
<input type="text" data-bind="number: { value: nights }"/>
<input type="text" data-bind="text: { value: checkIn,
        toModel: stringToDate, toView: dateToString }"/>
<input type="text" data-bind="text: { value: checkOut,
        toModel: stringToDate, toView: dateToString }"/>
```

Figure 3: The view for Figure 1's hotel reservation form.

or third-party widget toolkits. The view for our hotel reservation form is written in HTML and listed in Figure 3.

Bindings manage the data exchange between widgets in the view and variables in the view-model; they can be unidirectional (from view-model to view) or bidirectional. HotDrink provides two mechanisms for binding views to view-models: either in JavaScript or with HTML attributes. Our example opts for declarative bindings as seen in Figure 3.

Binders are functions that construct and register the event handlers that implement a particular binding. For example, the HotDrink text binder binds an HTML text input widget to a variable in the view-model using two event handlers:

- the first listens for changes to the widget and writes the variable;
- the second subscribes to changes in the variable and updates the widget.

Event handlers become such trivial boilerplate because the view-model constraint system handles all dependencies among the data in the UI. Reacting to a change in the view, the first handler assigns a new value for a variable in the constraint system, triggering an update. After solving the constraint system, the view-model notifies the second handler that the bound widget it manages may need to be updated.

All details for observing changes in the view, updating the view, and converting data from the type(s) used in the view to the type(s) used in the view-model (and vice-versa) are encapsulated in binders. HotDrink can be extended with custom binders to support any view; in simple cases, binders require no more than a few lines of code. HotDrink provides built-in binders for the standard HTML widgets.

Bindings are constructed by passing a widget and a set of options to a binder. As a convenience, bindings can be declared in HTML per widget with the data-bind attribute. This attribute is a JSON object where keys are binder names and values are binder options. When directed, HotDrink calls each named binder with the widget and the options. To construct complex bindings, e.g., for third-party widgets built in JavaScript, the programmer must explicitly call a binder in JavaScript.

In the hotel example, seen in Figure 3, each binding uses the built-in text binder described above. The value option names the corresponding variable in the view-model. In many instances, such as for the nights field here, no other options are necessary. The date fields, however, require more configuration. They manifest a commonly arising complication in web forms: the representation of data in the view is different from that in the view-model. In this case, the view maintains dates as strings whereas the view-model uses integers. The binding thus indicates, with the toModel and toView options, functions to convert values between the two representations:

```
function dateToString(m) {
  var d = new Date(m);
  return { value: (d.getMonth() + 1) + "/" +
           d.getDate() + "/" + d.getFullYear() };
}
function stringToDate(s) {
  var m = Date.parse(s);
  if (isNaN(m)) return { error: "Invalid date format" };
  else return { value: m };
}
```

Since conversions may fail, each conversion function labels its result as either the converted value or an error. In our example, the conversion from the view-model to view is guaranteed to succeed and thus never returns an error.

If a conversion function fails, the view-model's variable will not be modified. In this way, conversion functions serve as "validators" guarding the view-model from faulty values.

Finally, to instantiate the bindings, the programmer must pass the view-model to HotDrink:

```
var viewModel = {
  nights: nights,
  checkIn: checkIn,
  checkOut: checkOut
}
hd.bind(viewModel);
```

2.4 Behaviors

An advantage to using a constraint system to model the relationships among the values in a user interface is the ability to implement rich UI functionality as reusable algorithms. We have described a few such algorithms in the past [10, 7], including one for disabling command widgets when the preconditions of the command are violated [11]. HotDrink implements an updated version of that algorithm; we describe here how to use it.

The programmer first declares a *command*: a variable that holds a deferred function call constructed from a function and its arguments (cf. Command pattern [8, §5]). Whenever the command is invoked, the deferred function is called with those arguments. Typically, a command packages a call to some method of the model, passing as arguments the values of some variables in the view-model. For the hotel reservation form, an example command that reserves a room appears below:

```
viewModel.reserve = hd.command(function () {
  return hd.fn(model.reserve)(checkIn(), checkOut());
})
```

The hd.fn function defers a call to the reserve method of the model. That deferred call is constructed by an anonymous function. That function forms the sole method in a one-way constraint in the view-model. Whenever one of the arguments to the deferred call changes, the command will be reconstructed. By separating construction and invocation, HotDrink can identify the dependencies of a command before it is executed and implement rich UI behaviors like command activation and widget enablement.

A widget, e.g., a button, can be bound to the command so that interacting with it will invoke the command. HotDrink provides the built-in command binder for this purpose:

```
<button data−bind="command: reserve">
  Reserve a room</button>
```

For command activation, the programmer declares *preconditions*—Boolean predicates—that must be satisfied before a command is invoked. The reserve command might, for example, require a minimum stay of one night:

```
hd.precondition(viewModel.reserve, function () {
  return nights() > 0;
})
```

Whenever a precondition evaluates to false, the command activation algorithm will fire an event from the command variable indicating that it can be deactivated. Command widgets, like the button above, can subscribe to that event and disable themselves as appropriate. The built-in HotDrink command binder implements such behavior by default.

HotDrink can be extended with new behaviors. Though it generally takes significant effort to add them, great value lies in their reusability.

3. CONCLUSIONS AND FUTURE WORK

The example UI discussed in this paper is necessarily simple. Its dependencies can be expressed with a single (multi-way) constraint in the view-model. Larger UIs will likely contain more complex dependencies that become difficult to manage with ad-hoc event handling code. HotDrink helps to manage such complexity by encapsulating dependencies within constraints in a constraint system. Programmers can reason about each constraint in isolation—HotDrink's constraint system manages their aggregate behavior.

To demonstrate the use of HotDrink for a slightly more complex UI, we have implemented a canonical "todo list" application (http://hotdrinkjs.com/todomvc) according to the specifications of the TodoMVC [4] project. The TodoMVC application is a useful benchmark for objectively comparing different JavaScript MVC frameworks. HotDrink is very competitive in terms of conciseness. Note that TodoMVC does not offer an opportunity to demonstrate some of the more powerful features of HotDrink that are not offered by other frameworks, like multi-way or multi-output constraints or generic, reusable UI behaviors.

HotDrink is available at https://github.com/HotDrink and under active development. We are extending HotDrink with several reusable behaviors presented previously [7], such as undo/redo functionality, controls to manage the direction of the flow of data in a UI, and visualization of that flow.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0845861.

References

[1] Ember.js, an open source JavaScript GUI framework, Accessed May 2012. URL http://emberjs.com.

[2] Knockout, an open source JavaScript GUI framework, Accessed May 2012. URL http://knockoutjs.com.

[3] OpenLaszlo: a rich internet application development framework, Accessed May 2012. URL http://www.openlaszlo.org.

[4] TodoMVC: A common learning application for popular JavaScript MV* frameworks, Accessed May 2012. URL http://addyosmani.github.com/todomvc/.

[5] A. Borning and R. Duisberg. Constraint-based tools for building user interfaces. *ACM Trans. Graph.*, 5(4): 345–374, Oct. 1986. ISSN 0730-0301. URL http://doi.acm.org/10.1145/27623.29354.

[6] M. Fowler. Presentation Model Pattern, 1994. URL http://martinfowler.com/eaaDev/PresentationModel.html.

[7] J. Freeman, J. Järvi, W. Kim, M. Marcus, and S. Parent. Helping programmers help users. In *Proceedings of GPCE '11*, pages 177–184, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0689-8. URL http://doi.acm.org/10.1145/2047862.2047892.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[9] J. Gossman. Introduction to Model/View/ViewModel pattern for building WPF apps, October 2005. URL http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx.

[10] J. Järvi, M. Marcus, S. Parent, J. Freeman, and J. N. Smith. Property models: from incidental algorithms to reusable components. In *Proceedings of GPCE '08*, pages 89–98, New York, NY, USA, 2008. ISBN 978-1-60558-267-2. URL http://doi.acm.org/10.1145/1449913.1449927.

[11] J. Järvi, M. Marcus, S. Parent, J. Freeman, and J. N. Smith. Algorithms for user interfaces. In *Proceedings of GPCE '09*, pages 147–156, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-267-2. URL http://doi.acm.org/10.1145/1621607.1621630.

[12] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988.

[13] B. Myers, D. Giuse, R. Dannenberg, B. Zanden, D. Kosbie, E. Pervin, A. Mickish, and P. Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *Computer*, 23(11):71–85, Nov. 1990. doi: 10.1109/2.60882.

[14] B. A. Myers and M. B. Rosson. Survey on user interface programming. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 195–202, New York, NY, USA, 1992. ACM.

[15] M. Potel. MVP: Model-view-presenter: The Taligent programming model for C++ and Java, 1996. URL http://www.wildcrest.com/Potel/Portfolio/mvp.pdf.

[16] M. Sannella. Skyblue: A multi-way local propagation constraint solver for user interface construction. In *UIST '94: Proceedings of the 7th annual ACM symposium on User interface software and technology*, pages 137–146, New York, NY, USA, 1994. ACM.