Interface Programming Model

A Design Model for Constructing Graphical User Interfaces

Rawld Gill

ALTOVISO, LLC.

First Published January 2016

Abstract

This paper describes a precise model for constructing non-trivial programs that implement graphical user interfaces. It is intended to describe a particular, though not the only, model that may be used to construct the user interface for a larger software system that follows the interface-compute design model.

The design model is described in the context of a browser-hosted user interface. However, the model may be applied to any environment that has the ability to dynamically display output and generate input.

The paper presents the model without justification or explanation; such argument and discussion will be provided in follow-on papers.

Interface Model

Prescription 1. User-Interface Component

A user-interface component (UIC) is a JavaScript object that interacts (consumes input, produces output) with the user, such interaction intending to model a particular organizational or data type paradigm[1]. Examples of an organizational paradigm include a set of manila file folders or a list of items. Examples of a data type paradigm include a counting number or a date.

Prescription 2. User-Interface Framework

UICs are implemented in terms of user-defined types, also known as “classes”. This paper shall use the term class from here on. The machinery used to define such classes must include factories for creating instances of particular classes. Addition machinery is provided which operates on those instances created. The UIC classes together with the machinery that operates on instances of those UIC classes is termed the “UIC Framework”, which is shortened to “framework” from here on.

[1] For the general model, other languages, e.g., C++, may be used.

Prescription 3. Output

Output is produced by inserting, deleting, and/or modifying DOM trees within a browser document object[2,3]. All DOM trees are rendered programmatically by a UIC. A single UIC may generate several DOM trees.

Prescription 4. Input

All input that is accessible via DOM components (e.g., keyboard, mouse, touch, and pen) is consumed by providing and attaching event handlers to DOM elements to consume such input[4].

Prescription 5. Unique Identifiers

All UICs shall contain the property **`uid`** (unique identifier), which shall contain a unique value among all UICs[5]. Typically, though not always, the values created for this purpose should be created with a description of the form "*class_index*" indicating the value represents the *index*th instance of the UIC of type *class*. For UICs that are intended to be singletons within a single running instance of an application, the **`uid`** may be describe that singleton, e.g., "topFrame".

A global, searchable, iterable registry shall be provided by the framework which allows retrieving a reference to a particular UIC instance and listing all instances.

Prescription 6. Containers and Terminals

A UIC may contain other UICs; such UICs are termed "containers". If a UIC is not a container UIC, it termed a "terminal" UIC. A container UIC is termed the "parent" of the UICs it contains, and those UICs are termed the "children" of the "owning" parent. Every UIC must have exactly one parent and cycles within the parent-child chains are prohibited: the parent-child relationship of a set of UICs must form a tree. The top-most UIC of a UIC tree has no parent. It is allowed to have several sets of disconnected trees within an application.

All UICs contain the property **`parent`**, which references their parent.

All container UICs contain the property **`children`**, which is an array of references to their contained children (if any).

Although typical, it is not required that parent and children UICs organize the DOM trees they control in parent-child relationships[6]. In particular, the one or several DOM trees a particular child UIC generates need not be children of any of the DOM trees its owning parent generates.

All container UICs define the method **`insChild()`** and **`delChild()`** which add/remove a child from the container in the default manner (whatever the semantics of default, given the particular parent UIC). Container UICs may define

---

[2] A document may reside in the top-level window and/or in a frame and/or in an iframe.

[3] For the general model, other input-output machinery (e.g., the Windows API) may be substituted for the DOM. The point is that whatever input-output machinery is selected, the UIC abstracts a user interface on top of that machinery and the underlying machinery is not visible to the consumer of UICs.

[4] ibid

[5] In JavaScript, typically implemented in terms of a JS6 symbol.

[6] This requirement reinforces the point that the UIC tree is independent of the DOM tree.

additional methods to add/remove children and/or optional, but not required, parameters to **`insChild()`**/**`delChild()`**.

Prescription 7. The Active Tree

Every document is comprised of exactly—nothing more, nothing less—the DOM controlled by a single UIC tree. This tree is termed the "active" UIC tree for that document.

Subtrees of UICs may be removed from the active tree. Similarly, trees of UICs may be created separate from the active tree for later insertion into the active tree or destruction. Trees not part of the active tree are termed "inactive" trees.

Prescription 8. Rendering

UICs must implement the methods **`render(hint)`** and **`unrender().`**

`render(hint)` causes the current visual representation of the UIC to be output to the user. **`hint`**, optional, may contain a height and width, typically, though not required, in device units. A parent UIC must apply **`render()`** to all of its children when it is rendered. **`render()`** may be intelligent and update only the parts of the visual representation that have changed since the last application of **`render()`**, perhaps resulting in a no-op. A parent must never apply **`render()`** to a child when said parent is in the unrendered state (see below).

`unrender()` causes the current output to be removed from the DOM and all resources associated with rendering should be conserved. Container UICs shall apply **`unrender()`** to their children during their own unrender process.

UICs must maintain the read-only state property **`rendered`** to indicate whether or not the UIC is rendered or unrendered. Children UICs may query the rendered property of their parent to determine whether or not they need to take action to conserves resources. For example, during the application of **`unrender()`** on a container with a parent that is in the rendered state, that container's DOM tree must be removed from the document; however, during the recursive process of applying **`unrender()`** to said container's children, those children may notice that the their parent container is unrendered, and, therefore need not remove their DOM tree from the document so long as their tree is a subtree of the parent container's DOM tree as determined by the semantics and design of the particular container and child.

Prescription 9. Resizing

UICs must implement the method **`resize(hint)`**, which signals to the UIC that the viewport size has changed and causes the UIC to re-render if and only if the UIC is in a rendered state. **`hint`**, an optional parameter, instructs the UIC of the available height and width. The re-rendering may or may not actually result in producing output: the particular UIC may contain intelligence so that output is only produced when required, and a resize may never require manipulating the DOM. If the UIC is a container UIC, then it must either (i) apply **`resize()`** to all of its children, optionally providing a hint, or (ii) explicitly resize its children, whatever that implies depending upon the particular parent and children UIC semantics.

Prescription 10. Values

A UIC may define a **`value`** property. Values take the form of scalars, tuples, or relations, where scalars, tuples, and relations are defined in The Third Manifesto[7] (TTM), loosely…

1. A scalar is one unit of data and has no user-visible internal parts.
2. A tuple is a set of ordered triples of (name, value, type), where value is a scalar, tuple, or relation. See below for type.
3. A relation is a set of tuples where each tuple has the same set of (name, type) with equivalent semantics. Though there should be no repeated set of values, this is not enforced (a break from TTM).

A type is either a scalar type, a tuple type, or a relation type, as defined in TTM, loosely…

1. A scalar type is a named set of scalar values.
2. Tuple and relation types is given by a particular set of (name, type) ordered pairs and associated semantics.

Prescription 11. Property enabled

UICs that consume input must define the readable and mutable property **`enabled`**. When **`enabled`** is true, the UIC consumes input; when **`enabled`** is false, the UIC to refuses focus, whatever that implies, depending upon the actual input device, and ignores all input.

Upon mutating **`enabled`** on a container UIC, that container shall mutate all **`enabled`** properties of all of its children.

Prescription 12. Property readOnly

UICs which define a value property must define the readable and mutable property **`readOnly`**. When **`readOnly`** is true, the value property may be changed by user input, and conversely. The value property may be changed programmatically irrespective of the **`readOnly`** property.

Upon mutating **`readOnly`** on a container UIC, that container shall mutate all **`readOnly`** properties of all of its children.

Notice the subtle difference between **`enabled`** and **`readOnly`**: **`enabled`** describes a behavior that applies to the entire UIC whereas **`readOnly`** describes a behavior that applies only to the value of a UIC. For example, consider a grid UIC: such a UIC could have **`enabled===true`** and **`readonly==true`**, thereby allowing the rows to be sorted, the columns to be rearragaged/resized, etc. yet mutating the values within the grid would be prohibited.

Prescription 13. Property visible

UICs must define the readable and mutable property **`visible`**. When **`visible`** is true, the UIC is visible on the display; when **`visible`** is false, the UIC is not visible on the display. It is neither required nor prohibited to conserve resources like **`unrender()`** when **`visible`** is false.

[7] http://www.dcs.warwick.ac.uk/~hugh/TTM/TTM-2013-02-07.pdf

Prescription 14. Signals

UICs must define the method **`watch(name, handler)`**, which causes **`handler(newValue, oldValue)`** to be applied when the property given by name is mutated. All public, mutable properties must be watchable.

Prescription 15. Keyboard Input

The requirements in this section apply to UICs which consume keyboard input.

Such UICs shall provide the methods **`focusIn()`** and **`focusOut()`**; processing accomplished by these methods (if any, these methods may be provided as no-ops) is determined by the semantics of the particular UIC.

Such UICs shall maintain the read-only property **`focused`**, which indicates whether or not the UIC is in the current focus chain.

Upon a change of focus, the framework shall apply **`focusOut()`** to all UICs in the old focus chain, starting at the leaf, to all UICs that are losing focus, and then shall apply **`focusIn()`** to all UICs in the new focus chain, starting at the first internal node to receive focus.

The framework shall provide a function to retrieve the current focus chain.

The framework shall publish topics as follows:

1. **`chainFocusOut`**: the chain of UICs losing focus
2. **`focusOut`**: Each UIC losing focus, from leaf to internal node.
3. **`focusIn`**: Each UIC gaining focus, from internal node to leaf.
4. **`chainFocusIn`**: The chain of UICs gaining focus

Prescription 16. Keyboard Preprocessing

UICs may implement the methods **`onKeyDown(e)`**, **`onKeyUp(e)`**, and **`onKeyPress(e); e`** is the event object provided by the browser. Processing accomplished by these methods (if any, these methods may be provided as no-ops) is determined by the semantics of the particular UIC. By returning true, these methods prevent further propagation down the UIC focus chain, see next. UICs need not directly control DOM elements that receive the focus in order to implement these methods.

The framework shall capture keydown, keyup, and keypress events at the root node of each document. The captured event shall be dispatched down the chain of UICs that have the focus, starting with the root node, to any UICs in that chain the contain the **`onKeyDown()`**, **`onKeyUp()`**, and **`onKeyPress()`** methods. Dispatch shall be stopped if a particular application returns true.

Prescription 17. Destructors

All UICs shall define the method **`destroy()`**, which shall:

1. Destroy all of its children, if it's a container.
2. **`delChild()`** itself from its owning parent.
3. Remove itself from the registry described in [Prescription 5].
4. Disconnect any connections to its members.
5. Delete all resources so they may be garbage collected.