

Lecture 11: Declarative UI

UI Hall of Fame or Shame?



Source: Daniel P.B. Smith, *Risks Digest*, v24 n44

Today's hall of fame or shame candidate is this DVD player. The arrow keys on a DVD player are supposed to move a cursor highlight around the screen. If you look carefully at the picture, however, you'll see that the arrow diamond has been rotated by 45 degrees.

Think about:

- natural mapping
- consistency, both external and internal

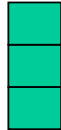
Today's Topics

- Declarative user interface
- HTML
- CSS
- Model-based UI

Declarative vs. Procedural

- Declarative programming
 - Saying *what* you want
- Procedural programming
 - Saying *how* to achieve it

Declarative
A tower of 3 blocks.



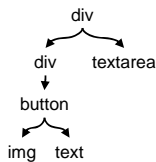
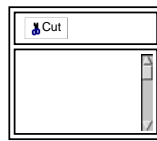
Procedural
1. Put down block A.
2. Put block B on block A.
3. Put block C on block B.

Today we'll be talking about ways to implement user interfaces using higher-level, more abstract specifications – particularly, **declarative specifications**. The key advantage of declarative programming is that you just say **what** you want, and leave it to an automatic tool to figure out how to produce it. That contrasts with conventional procedural programming, where the programmer has to say, step-by-step, how to reach the desired state.

HTML is a Declarative UI Language

- HTML **declaratively** specifies a view hierarchy

```
<div id="main">  
  <div id="toolbar">  
    <button>  
      </img>  
      Cut  
    </button>  
  </div>  
  <textarea id="editor"></textarea>  
</div>
```



Our first example of declarative UI programming is **HTML**, which is a declarative specification of a view hierarchy. An HTML **element** is a component in the view hierarchy. The type of an element is its **tag**, such as `div`, `button`, and `img`. The properties of an element are its **attributes**. In the example here, you can see the `id` attribute (which gives a unique name to an element) and the `src` attribute (which gives the URL of an image to load in an `img` element); there are of course many others.

There's an automatic algorithm, built into every web browser, that constructs the view hierarchy from an HTML specification – it's simply an HTML parser, which matches up start tags with end tags, determines which elements are children of other elements, and constructs a tree of element objects as a result. So, in this case, the automatic algorithm for this declarative specification is pretty simple. We'll see more complex examples later in the lecture.

Declarative HTML vs. Procedural Java

HTML

```
<div id="main">
  <div id="toolbar">
    <button>
      </img>
      Cut
    </button>
  </div>
  <textarea id="editor"></textarea>
</div>
```



Java Swing

```
JPanel main = new JPanel();

JPanel toolbar = new JPanel();
JButton button = new JButton();
button.setIcon(...);
button.setLabel("Cut");
toolbar.add(button);
main.add(toolbar);

JTextArea editor = new JTextArea();
main.add(editor);
```

Spring 2008

6.831 User Interface Design and Implementation

6

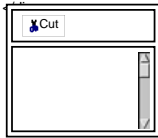
To give an analogy that you should be familiar with, here's some Swing code that produces the same interface procedurally. By comparison, the HTML is more concise, more compact – a common advantage of declarative specification.

Note that neither the HTML nor the Swing code actually produces the **layout** shown in the picture, at least not yet. We'd have to add more information to both of them to get the components to appear with the right positions and sizes. We'll talk about layout later.

Declarative HTML vs. Procedural DOM

HTML

```
<div id="main">
  <div id="toolbar">
    <button>
      </img>
      Cut
    </button>
  </div>
  <textarea id="editor"></textarea>
</div>
```



Document Object Model (DOM) in Javascript

```
var main = document.createElement("div");
main.setAttribute("id", "window");

var toolbar = document.createElement("div");
toolbar.setAttribute("id", "toolbar");

var button = document.createElement("button");
var img = document.createElement("img");
img.setAttribute("src", "cut.png");
button.appendChild(img);

var label = document.createTextNode("Cut");
button.appendChild(label);
toolbar.appendChild(button);
window.appendChild(toolbar);

var editor = document.createElement("textarea");
editor.setAttribute("id", "editor");
window.appendChild(editor);
```

Spring 2008

6.831 User Interface Design and Implementation

7

Here's procedural code that generates the same HTML component hierarchy, using the Javascript programming and the Document Object Model (DOM). DOM is a standard set of classes and methods for interacting with a tree of HTML or XML objects procedurally. (DOM interfaces exist not just in Javascript, which is the most common place to see it, but also in Java and other languages.)

There are a lot of similarities between the procedural code here and the procedural Swing code on the previous page – e.g. **createElement** is analogous to a constructor, **setAttribute** sets attributes on elements, and **appendChild** is analogous to add.

Incidentally, you don't always have to use the **setAttribute** method to change attributes on HTML elements. In Javascript, many attributes are reflected as properties of the element (analogous to fields in Java). For example, **obj.setAttribute("id", value)** could also be written as **obj.id = value**. Be warned, however, that only standard HTML attributes are reflected as object properties (if you call **setAttribute** with your own wacky attribute name, it won't appear as a Javascript property), and sometimes the name of the attribute is different from the name of the property. For example, the "class" attribute must be written as **obj.className** when used as a property.

Mixing Declarative and Procedural Code

HTML

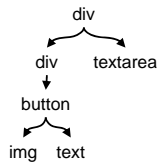
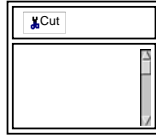
```
<div id="main">
  <textarea id="editor"></textarea>
</div>
```

<script>

```
var toolbar = document.createElement("div");
toolbar.setAttribute("id", "toolbar");

toolbar.innerHTML =
  "<button><img src='cut.png'></img>Cut</button>";

var editor = document.getElementById("editor");
var main = editor.parentNode;
main.insertBefore(toolbar, editor);
</script>
```



Spring 2008

6.831 User Interface Design and Implementation

8

To actually create a working interface, you frequently need to use a mix of declarative and procedural code. The declarative code is generally used to create the static parts of the interface, while the procedural code changes it dynamically in response to user input or model changes. Here's a (rather contrived) example that builds part of the interface declaratively, then fills it in with procedural code.

The `<script>` element allows you to introduce procedural code (which most web browsers assume is written Javascript) into the declarative specification. Code in the `<script>` element is executed immediately when the HTML page is first displayed, but of course you could also write functions or event handlers in the `<script>` element so that the procedural code runs later.

Even inside the procedural code, we can use declarative code. The `innerHTML` property of an HTML element represents the HTML between its start tag and end tag – in other words, the element's descendents in the view hierarchy. Setting this property removes all its current descendents and replaces them with elements created by the HTML you provide. Here, the button and img elements are created and added to the toolbar in this way.

The last part of the script shows a few other useful things. Putting `id` attributes on an element makes it easy to get a reference to it using `getElementById` in procedural code. (You can also refer to elements by id in declarative code.) You can also navigate around the element tree using `parentNode` and `childNodes[]` attributes. Also, you can insert new elements using `insertBefore`, not just append them; and you can remove and replace elements with `removeChild` and `replaceChild`. Documentation for these DOM operations can be found in many places on the Web; see the problem set for some useful references.

Advantages & Disadvantages of Declarative UI

- Usually more compact
- Programmer only has to know how to say *what*, not *how*
 - Automatic algorithms are responsible for figuring out how
- May be harder to debug
 - Can't set breakpoints, single-step, print in a declarative specification
 - Debugging may be more trial-and-error
- Authoring tools are possible
 - Declarative spec can be loaded and saved by a tool; procedural specs generally can't

Spring 2008

6.831 User Interface Design and Implementation

9

Now that we've worked through our first simple example of declarative UI – HTML – let's consider some of the advantages and disadvantages.

First, the declarative code is usually more compact than procedural code that does the same thing. That's mainly because it's written at a higher level of abstraction: it says *what* should happen, rather than *how*.

But the higher level of abstraction can also make declarative code harder to debug. There's generally no notion of time, so you can't use techniques like breakpoints and print statements to understand what's going wrong. The automatic algorithm that translates the declarative code into working user interface may be complex and hard to control – i.e., small changes in the declarative specification may cause large changes in the output. Declarative specs need debugging tools that are customized for the specification, and that give insight into how the spec is being translated; without those tools, debugging becomes trial and error.

On the other hand, an advantage of declarative code is that it's much easier to build authoring tools for the code, like HTML editors or GUI builders, that allow the user interface to be constructed by direct manipulation rather than coding. It's much easier to load and save a declarative specification than a procedural specification. Some GUI builders *do* use procedural code as their file format – e.g., generating Java code and automatically inserting it into a class. Either the code generation is purely one-way (i.e., the GUI builder spits it out but can't read it back in again), or the procedural code is so highly stylized that it amounts to a declarative specification that just happens to use Java syntax. If the programmer edits the code, however, they may deviate from the stylization and break the GUI builder's ability to read it back in.

Important HTML Elements for UI Design

- **Layout**
 - Box `<div>`
 - Grid `<table>`
- **Text**
 - Font & color ``
 - Paragraph `<p>`
 - List ``, ``
- **Widgets**
 - Hyperlink `<a>`
 - Textbox `<input type="text">`
 - Textarea `<textarea>`
 - Drop-down `<select>`
 - Listbox `<select multiple="true">`
 - Button `<input type="button">`, `<button>`
 - Checkbox `<input type="checkbox">`
 - Radiobutton `<input type="radio">`
- **Pixel output**
 - ``
- **Stroke output**
 - `<canvas>` (Firefox, Safari)
- **Procedural code**
 - `<script>`
- **Style sheets**
 - `<style>`

Spring 2008

6.831 User Interface Design and Implementation

10

To complete our survey of HTML as a language for generating component hierarchies, here is a cheat sheet of the most important elements that you might use in an HTML-based user interface.

The `<div>` and `` elements are particularly important, and may be less familiar to people who have only used HTML for writing textual web pages. By default, these elements have no presentation associated with them; you have to add it using style rules (which we'll explain next). The `<div>` element creates a box (not unlike JPanel in Swing), and the `` element changes textual properties like font and color while allowing its contents to flow and word-wrap.

HTML has a rather limited set of widgets. There are other declarative UI languages similar to HTML that have much richer sets of built-in components, such as XUL (used in Mozilla Firefox) and XAML (used in Microsoft Windows Vista).

HTML does support both pixel and stroke output, although the stroke output is nonstandard – some browsers support the `<canvas>` element, which has methods for making stroke output using procedural code, not much different from Swing's Graphics object.

Finally, we've already seen how to use the `<script>` element to embed procedural code (usually Javascript) into an HTML specification. The `<style>` element is used for embedding another declarative specification, style sheets, which is what we'll look at next.

Cascading Style Sheets (CSS)

- Key idea: separate the **structure** of the UI (view hierarchy) from details of **presentation**
 - HTML is structure, CSS is presentation
- Two ways to use CSS
 - As an attribute of a particular HTML element
`<button style="font-weight:bold;"> Cut </button>`
 - As a separate style sheet defining pattern/style rules, possibly for many HTML elements at once

```
<style>
  button { font-weight:bold; }
</style>
```

Our second example of declarative specification is Cascading Style Sheets, or CSS. Where HTML creates a view hierarchy, CSS adds style information to the hierarchy – fonts, colors, spacing, and layout.

There are two ways to use CSS. The first isn't very interesting, because we've seen it before in Swing: changing styles directly on individual components. The style attribute of any HTML element can contain a set of CSS settings (which are simply **name:value** pairs separated by semicolons).

The second way is more interesting to us here, because it's more declarative. Rather than finding each individual component and directly setting its style attribute, you specify a **style sheet** that defines rules for assigning styles to elements. Each rule consists of a pattern that matches a set of HTML elements, and a set of CSS definitions that specify the style for those elements. In this simple example, **button** matches all the button elements, and the body of the rule sets them to boldface font.

The style sheet is included in the HTML by a `<style>` element, which either embeds the style sheet as text between `<style>` and `</style>`, or refers to a URL that contains the actual style sheet.

CSS Selectors

- Each rule in a style sheet has a **selector** pattern that matches a set of HTML elements

Tag name	<code><div id="main"></code>
button	<code>{ font-weight:bold; }</code>
ID	<code><div id="toolbar"></code>
#main	<code>{ background-color:</code>
Class attribute	<code>rgb(100%,100%,100%);</div></code>
.toolbarButton	<code>{ font-size: 12pt; }</code>
Element paths	<code></code>
#toolbar button	<code>{ display: hidden; }</code>

The pattern in a CSS rule is called a **selector**. The language of selectors is simple but powerful. Here are a couple of the more common selectors.

CSS selectors aren't the only way to declaratively specify a set of HTML nodes (although it's the only way that's permitted in a CSS style sheet rule). Another declarative way to describe a set of elements is **XPath**, a pattern language that has some similarities to CSS selectors but is strictly more powerful.

Cascading and Inheritance

- If multiple rules apply to the same element, rules are automatically combined with **cascading** precedence
 - Source: browser defaults < web page < user overrides

```
Browser says:  a { text-decoration: underline; }
Web page says: a { text-decoration: none; }
User says:     a { text-decoration: underline; }
```
 - Rule specificity: general selectors < specific selectors

```
button { font-size: 12pt; }
.toolbarButton { font-size: 14pt; }
```
- Styles can also be **inherited** from element's parent
 - This is the default for simple styles like font, color, and text properties

```
body { font-size: 12pt; }
```

Spring 2008

6.831 User Interface Design and Implementation

13

There can be multiple style sheets affecting an HTML page, and multiple rules within a style sheet. Each rule affects a set of HTML elements, so what happens when an element is affected by more than one rule? If the rules specify independent style properties (e.g., one rule specifies font size, and another specifies color), then the answer is simple: both rules apply. But what if the rules *conflict* with each other – e.g., one says the element should be bold, and another says it shouldn't?

To handle these cases, declarative rule-based systems need a conflict resolution mechanism, and CSS is no different. CSS's resolution mechanism is called **cascading** (hence the name, Cascading Style Sheets). It has two main resolution strategies. The overall idea is that more specific rules should take precedence over more general rules. This is reflected first in where the style sheet rule came from: some rules are web browser defaults, for all users and all web pages; others are defaults set by a specific user for all web pages; others are provided by a specific web page in a <style> element. In general, the web page rule wins (although the user can override this by setting the priority of their own CSS rules to *important*). Second, rules with more specific selectors (like specific element IDs or class names) take precedence over rules with more general selectors (like element names).

This is an example of why declarative specification is powerful. A single rule – like a user override – can affect a large swath of the behavior of the system, without having to write a lot of procedural code, and without having to make sure that procedural code runs at just the right time.

But it also illustrates the difficulties of debugging declarative specifications. You may add a rule to the style sheet, maybe trying to change a button's font size, only to see *no change* in the result – because some other rule that you aren't aware of is taking precedence. CSS conflict resolution is a complex process that may require trial-and-error to debug.

Declarative Styles vs. Procedural Styles

```
CSS
<div id="main">
  <div id="toolbar">
    <button style="font-size: 12pt">
      </img>
    </button>
  </div>
  <textarea id="editor"></textarea>
</div>

Javascript
var buttons =
  document.getElementsByTagName("button");
for (var i = 0; i < buttons.length; ++i) {
  var button = buttons[i];
  button.style.border = "2px";
  // not button.setAttribute("style", "border: 2px");
}
```

Spring 2008

6.831 User Interface Design and Implementation

14

Just as with HTML, we can change CSS styles procedurally as well. We saw earlier that HTML attributes can be get and set using Javascript object properties (like `obj.id`) rather than methods (like `obj.setAttribute("id",...)`). For CSS styles, this technique is actually *essential*, since calling `setAttribute()` will replace the current style attribute entirely. In this example, if we called `button.setAttribute("style", "border:2px")`, the original style attribute (which set the font size to 12pt) would be lost. So it's best to use the style property, not the style attribute, when you're changing styles procedurally. The style property points to an object with properties representing all the style characteristics in CSS.

Java Swing: Declarative vs. Procedural

Java Swing

```
JFrame frame = new JFrame();
JButton button = new JButton("Press me");
frame.getContentPane().add(button);
button.addActionListener(...)
frame.pack();
frame.setVisible(true);
```

JavaFX

```
Frame {
  content: Button {
    text: "Press Me"
    action: ...
  }
  visible: true
}
```

Spring 2008

6.831 User Interface Design and Implementation

15

The closest thing Java Swing has to declarative specification is probably the JavaFX scripting language (formerly called F3). Although JavaFX is essentially a procedural scripting language, it includes a declarative syntax for constructing Java objects, shown on the right.

Other Declarative UI Languages

- XUL
 - used by Firefox
- XAML
 - used by Windows Presentation Foundation and Silverlight

Spring 2008

6.831 User Interface Design and Implementation

16

This lecture has discussed the most widely-used declarative user interface language (HTML), but other up-and-coming languages include XUL and XAML.

Model-Based User Interfaces

- Programmer writes logical model of UI
 - State variables (bool, int, string, list)
 - Commands
- System generates actual presentation
 - Grouping into windows, tabs, panels
 - Widget selection
 - Layout
- Same motivation as other declarative UI
 - Higher-level programming
 - Adapting to change: particularly for devices and users
 - Screen size (watch, phone, PDA, laptop, desktop, wall)
 - Widgets available (phone vs. desktop)
 - Input style (mouse vs. arrow buttons; speech, finger, pen)
 - Output style (speech vs. display)
 - User behavior (uses some components more)

Spring 2008

6.831 User Interface Design and Implementation

17

Finally, let's discuss one more example of declarative UI, which is still farther-out. A **model-based user interface** is not only declarative but also *abstract*, so that the user interface can be adapted automatically to different situations.

The programmer provides a high-level description of the user interface, often (and confusingly) called a model, which consists of a set of data variables and commands. A model for a login dialog box, for example, might state that there are two string variables (a username and password) and one command (login).

This description is then used to generate a presentation. Aspects of the presentation that must be generated include its **grouping** (how variables and commands are organized); the particular **widgets** selected for each model element (e.g., a string variable might be represented by a textfield, a text area, or a combobox; a command might be a button, menu item, keyboard shortcut, or all three). **Layout** is part of presentation, of course; so are labels for the widgets, font and color choices.

The presentation can't be generated completely automatically, of course – UI design isn't that easy to automate (yet). Generally, the programmer has to provide some presentation specification as well – perhaps completely specified, or merely as hints or constraints on legal or desirable presentations.

Model-based user interface is driven by the same motivations as other declarative UI: simpler programming, and adapting to change. But the kinds of change that model-based UI can adapt to is broader. Ideally, a model-based UI should be able to generate presentations for a broad variety of **devices** and I/O styles: not just a desktop with mouse and keyboard, but a cellphone with a tiny screen, buttons, and voice I/O; or a large wall screen with touch-sensitive finger input. Another kind of adaptation envisioned by model-based UI is **user adaptation**; an interface might change depending on how the user interacts with it.

Summary

- Declarative says *what*, leaving *how* to automatic mechanisms
- Separation of structure (HTML) and presentation (CSS)
- Need to know how to change HTML and CSS procedurally too