BrowserMonkey

Programming Documentation

v 1.0

Software House One

Version History

Document name: Programming Team/baselined/Documentation.pdf

Document version: 1.0

Document author: Paul Calcraft 2009/06/08

Document auditor: Ioanna Kyprianou 2009/06/08
Document auditor: Daniel Cooper 2009/06/08
Document auditor: Paul Calcraft 2009/06/08
Document auditor: Sohani Amiruzzaman 2009/06/08

Table of Contents

1 Navigating the Code

As stated in our previous Deliverable, we are using Google Code for our source control. The link to our project is:

http://code.google.com/p/browsermonkey/

On the SVN, our code is located in the Programming Team/baselined/ directory, and consists of 3 (NetBeans) projects, each with their own directory. Each project directory has a /src/ subfolder with all the packages and their classes as subdirectories/files – the java files can be accessed directly this way if not using NetBeans to open the projects. The 3 projects that comprise our software are:

- BrowserMonkey – the browser itself
- BrowserMonkeySDK – the set of shared libraries needed to develop rendering plugins
- BrowserMonkeyTagPlugins – the set of TagRenderers implemented for our project

Both of the other projects directly depend on the SDK as a linked library project, while the browser project knows nothing of the plugins and vice versa. The browser attempts to load any jar files within its /plugins/ directory dynamically on start up, and uses only the interface specified in the SDK project to interact with them.

When the TagPlugins project is built, a build script (specified in BrowserMonkeyTagPlugins/build.xml) is used to copy the newly compiled jar into BrowserMonkey/plugins/ and BrowserMonkey/dist/plugins. This means when running the browser after compiling the plugins, the browser will use the latest plugin code. If you wish to clean and build the main browser project, this will delete the dist directory and then rebuild BrowserMonkey into BrowserMonkey/dist. Because it deletes the dist directory in the cleaning process, it loses the tag plugins jar file, so you should clean and build the plugins project afterwards to automatically copy the jar back across.

Note that the browser will not choke if the plugins aren't there, it will make a note in the log file that the renderers could not be loaded, and all pages will render with no specialisation of tags – so just each tag's inner text will show.

The **JavaDocs** for the 3 projects are available here:

Programming Team/baselined/BrowserMonkey/dist/javadoc/index.html
Programming Team/baselined/BrowserMonkeySDK/dist/javadoc/index.html
Programming Team/baselined/BrowserMonkeyTagPlugins/dist/javadoc/index.html

2 Running the Software

To run the software:

- Obtain the distribution.zip from the root of the SVN and extract it.
- Run BrowserMonkey.jar.

3 Changes from the Low Level Design

3.1 RenderNode

In order to implement zooming functionality correctly, we added a setZoomLevel(float level) method to the RenderNode class (and then implemented it in all subclasses). In the design we had envisioned the DocumentPanel traversing the RenderNode tree, finding any TextRenderNodes and applying zoom to those. This way would have been highly restrictive, and indeed it was – we implemented zooming on images and for this we needed a general way to zoom any RenderNode. This setZoomLevel method is responsible for zooming a given RenderNode and all the components it contains. This allows the DocumentPanel to simply call setZoomLevel on the root RenderNode and have everything filter through and work.

We also added a method called extractTextInto(ArrayList<AttributedString>) which, for any given RenderNode, should extract a reference to every text node's AttributedString (using depth first search on child components) into the array list. This allowed us to implement searching functionality. The searching works how we designed it, but in order to effectively get the AttributedStrings from the text nodes without using really specialised code, and without the DocumentPanel having to traverse the RenderNode tree itself, this was the logical conclusion. It solves the problem in a similar way to the setZoomLevel method, above.

3.2 LayoutRenderNode

There were more subtle issues of component layout that needed to be addressed, than we had considered in our low level design. Things like indentation, padding, resize behaviour, and distance between sections of text (for correct paragraphing). For this we added methods to the LayoutRenderNode that the TagRenderers could use to manipulate the layout of the components. The methods added include ensureNewLine (replacing breakTextNode from the design), ensureLinespaceDistance, setNodePadding, and setPadding, but there are a few more subtle tweaks (like allowing a node added to a LayoutRenderNode to specify how its width should be resized).

3.3 Renderer

The renderer previously was imagined to have very little state; it was more of a class to perform some hard work for us using reflection-loaded TagRenderers. It became apparent that it should hold state of the document it is rendering, in order to provide needed functionality to the TagRenderers for certain types of tags. For example, the only way we could implement headings correctly (as per the user requirements, where <hn> (n=1-6) tags show the actual heading numbers) was to store the state of the heading numbering in the Renderer and provide a public method for a TagRenderer to retrieve and update this state. We also used state for determining the title of the document. We couldn't allow the TagRenderers to communicate with the DocumentPanel or GUI itself, so we stored a title property that the TitleTagRenderer could set on the Renderer, which is retrieved by the DocumentPanel and then GUI after loading.

Some utility methods and static properties were also defined for convenience, like an attribute map representing default formatting, and a method for constructing a new text node representing standard indentation.

A big addition to the project was support of linked images. This did not make it into the customer requirements but was indicated as still desirable; we managed to implement it successfully. In order to do so, we needed TagRenderers to be able to load external resources, so we made a loadResource method in the Renderer which uses our IO methods. For this to work correctly, the Renderer needed to know the URL context of the document it was rendering, so that it could load these external resources from relative paths. To facilitate this, the DocumentPanel now passes the context URL in the renderRoot method on the Renderer and it is stored as a private property of the Renderer.

3.4 IOUtility

In our design, file input was imagined to be solely the responsibility of the Document itself. As we added image support and needed external resources to be loaded as well, we decided to take out the file and URL handling from the Document and into an external utility class in the SDK project that both the Document (in the browser main project) and Renderer (in the SDK project) could access. It does all the hard work of determining where to look for files and actually making connections and retrieving the data byte by byte. Because of this abstraction, adding internet support was quite painless. We just needed to ensure we opened a URL connection, and caught all the errors correctly.

3.5 Swing worker threads

As we added internet support and had files that were loading slowly it became apparent we could no longer block waiting for the full input file on the GUI thread – the program became

immediately unresponsive until either the page was fully loaded (including all resources), or it timed out. To solve this we used swing to create a worker thread for page loading functionality. We initiate this in the DocumentPanel, and it's setup to cancel the current loading operation (if not finished) when the user decides they wish to navigate elsewhere while another page is loading.

4 Extensibility

The renderer is extremely extensible. All tags that are rendered use the tagRenderers.properties file to dictate what class is used to render them, here is a snippet:

```
b = browsermonkey.render.BoldTagRenderer
strong = browsermonkey.render.BoldTagRenderer
em = browsermonkey.render.ItalicsTagRenderer
i = browsermonkey.render.ItalicsTagRenderer
table = browsermonkey.render.TableTagRenderer
a = browsermonkey.render.AnchorTagRenderer
br = browsermonkey.render.LineBreakTagRenderer
p = browsermonkey.render.ParagraphTagRenderer
font = browsermonkey.render.FontTagRenderer
```

The actual classes themselves can be added to with great ease, without having to even recompile the BrowserMonkey project. One could create a new project that links to the SDK library, and implement a set of TagRenderers in there. To hook them up, the compiled jar file for this new project would just need to be placed (alongside the existing TagPlugins.jar, or not) inside the /plugins/ directory, and the new TagRenderers would need to be named in the properties file for the appropriate tag types. Our Renderer searches the plugins directory for all jar files and uses them all to try to find classes of the given names, which is why multiple jar files can be used simultaneously, allowing very easy extension to the browser in terms of what tags it can render (and indeed redefining what it currently renders and how it does it).

5 Extra features

5.1 Images

We implemented images in our application, which was above and beyond our specification. Thanks to our systems of extensibility this took relatively little effort (though some functionality requirements cascaded from this as outlined above). The ImageTagRenderer has a private class ImageRenderNode which is the JComponent itself that renders an image on its surface, the TagRenderer adds a new ImageRenderNode into the render tree for each "img" tag. It supports zooming, and even links.

5.2 Table layout system

When it came to table layout, the initial implementation split each table into equispaced columns given the full width it had available to it. While this was fine, it was not as good as we wanted it to be. We decided to make tables shrink to only the size they need to be. This caused a lot of problems with text nodes resizing themselves, word wrap, and all sorts of concerns, but we finally got it implemented, and now tables are much more impressive. We also implemented the border attribute on tables (default is 1pixel) which was not required by the specification. These two additions to table layout make browsing web sites much closer to the real thing.

5.3 Debugging and testing

At the start of the project when the parser and tokeniser were not yet implemented, a debug feature was added to allow testing of rendering features. We could type for example “t b” in the address bar and it would build a document node tree with a b tag put in the middle somewhere to see how the renderer handled it. This did not need to be rewritten for simple tests – it just constructs a tag with the given type. We added basic attribute support (for attribute values without spaces) so we could do “t font face=Arial” and it would build a test document correctly. When it came to tables, we implemented a specialised test (as we couldn’t interpret a whole table definition from a “t...” input. When you typed “t table” a special table test was constructed. We later added nested tables to this for further testing. This testing functionality is still available in the final renderer by using “t “-type strings in the address bar.

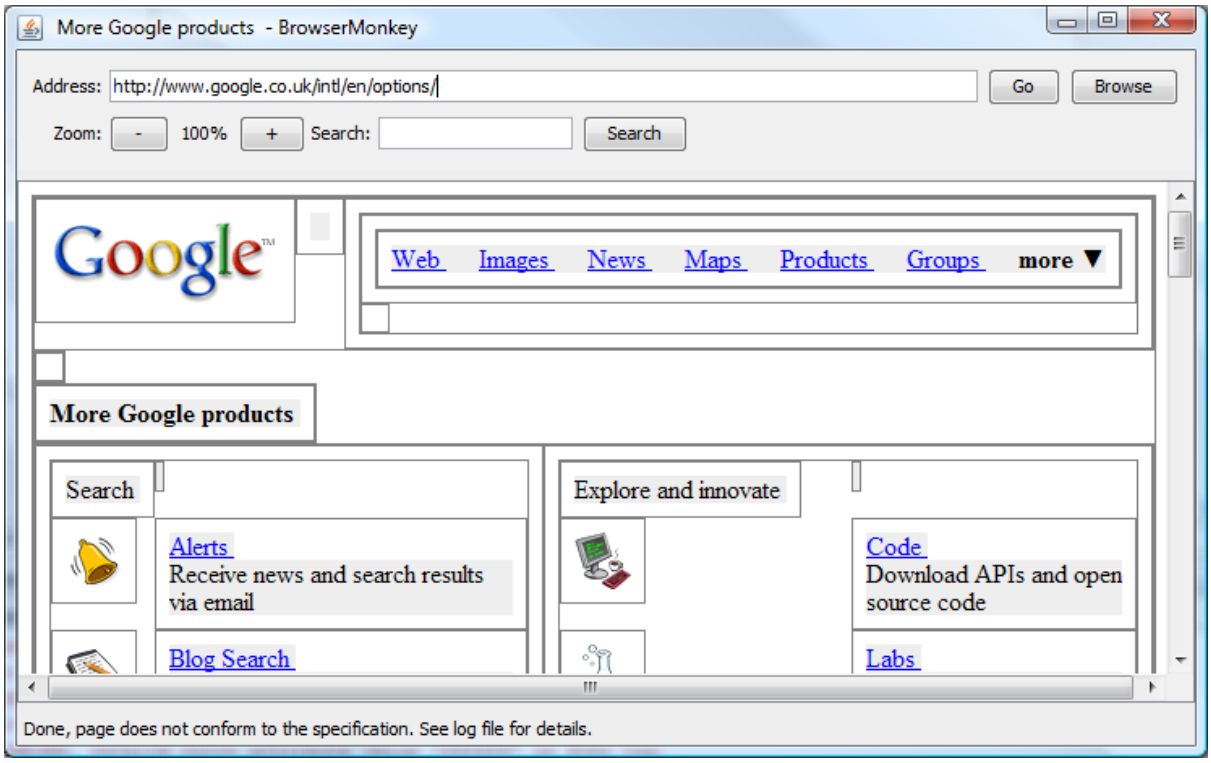
Another debug feature added was to output the DocumentNode tree as constructed by the tokeniser and parser. This allowed us to see exactly where errors were occurring in the file loading process, and whether certain problems were because of the parser or because of the renderer. The feature copied a full debug string output of the DocumentNode into the system clipboard to be pasted wherever, and inspected. The following code was used (now commented out in the build):

```
try {
      Clipboard systemClipboard =
      Toolkit.getDefaultToolkit().getSystemClipboard();

      Transferable transferableText = new
      StringSelection(document.getNodeTree().toDebugString());

      systemClipboard.setContents(transferableText, null);

} catch (IllegalStateException ex) {
      BrowserMonkeyLogger.warning("Couldn't write debug parse information to
      clipboard.");
}
```

This was particularly useful in implementing pre tags, and the HTML auto-correction for non-conforming documents.

When it came to implementing internet functionality with pages that have extremely complicated (and often corrupt) structure, it was very difficult to get everything working correctly, especially within swing's component layout system. With something as involved as GUI rendering and swing, it is very difficult to use traditional debugging methods, and sometimes simply unfeasible. To solve this, we occasionally would modify the rendering code to show extra borders and background colours for different types of components to allow us to see exactly where the boundaries of all our RenderNodes are, and how our text nodes are resizing themselves.

The following screenshot demonstrates this on a Google labs page. A light gray is used for the background of any text nodes, and a dark gray border shows the boundaries of all LayoutRenderNodes.



This debugging technique was absolutely vital in getting text node zooming and the new table layout system working correctly.