**BrowserMonkey**

****

**Low Level Design Document**

v 0.5

**Software House One**

**Version History**

Document name: Design Team/scratch/Low Level/Low Level Design.docx

Document version: 0.5

Document author: Daniel Cooper 2009/04/21

Document author: Merrigan Baylis 2009/04/22

Document author: Paul Calcraft 2009/04/22  
Document author: Adam McGinness 2009/04/23

Document author: Lawrence Dine 2009/04/27

Document auditor: Ioanna Kyprianou 2009/04/30

Document auditor: Daniel Cooper 2009/04/30

Document auditor: Paul Calcraft 2009/04/30

Document auditor: QA Rep 2009/04/30

Last modification date: 2009/04/30

Contents

[Notes 2](#_Toc228761780)

[Architectural Design 3](#_Toc228761781)

[Tokeniser & Parser 4](#_Toc228761782)

[Tokeniser & Tokens 4](#_Toc228761783)

[Prototype 4](#_Toc228761784)

[Parser 7](#_Toc228761785)

[Renderer 8](#_Toc228761786)

[Logging 8](#_Toc228761787)

[Search 8](#_Toc228761788)

[Zoom 9](#_Toc228761789)

[Tag Interpretation Using Reflection 10](#_Toc228761790)

[In Theory 10](#_Toc228761791)

[In Java 10](#_Toc228761792)

[External Files 10](#_Toc228761793)

[Reflection 11](#_Toc228761794)

[In the BrowserMonkey Browser 11](#_Toc228761795)

[Pseudo Code 11](#_Toc228761796)

# Notes

Object diagrams need to be completed

Reflection process needs a bit of re-wording

Follow link sequence diagram need to be rescued from netbeans and inserted

Class Diagram needs some explaining.

# Architectural Design

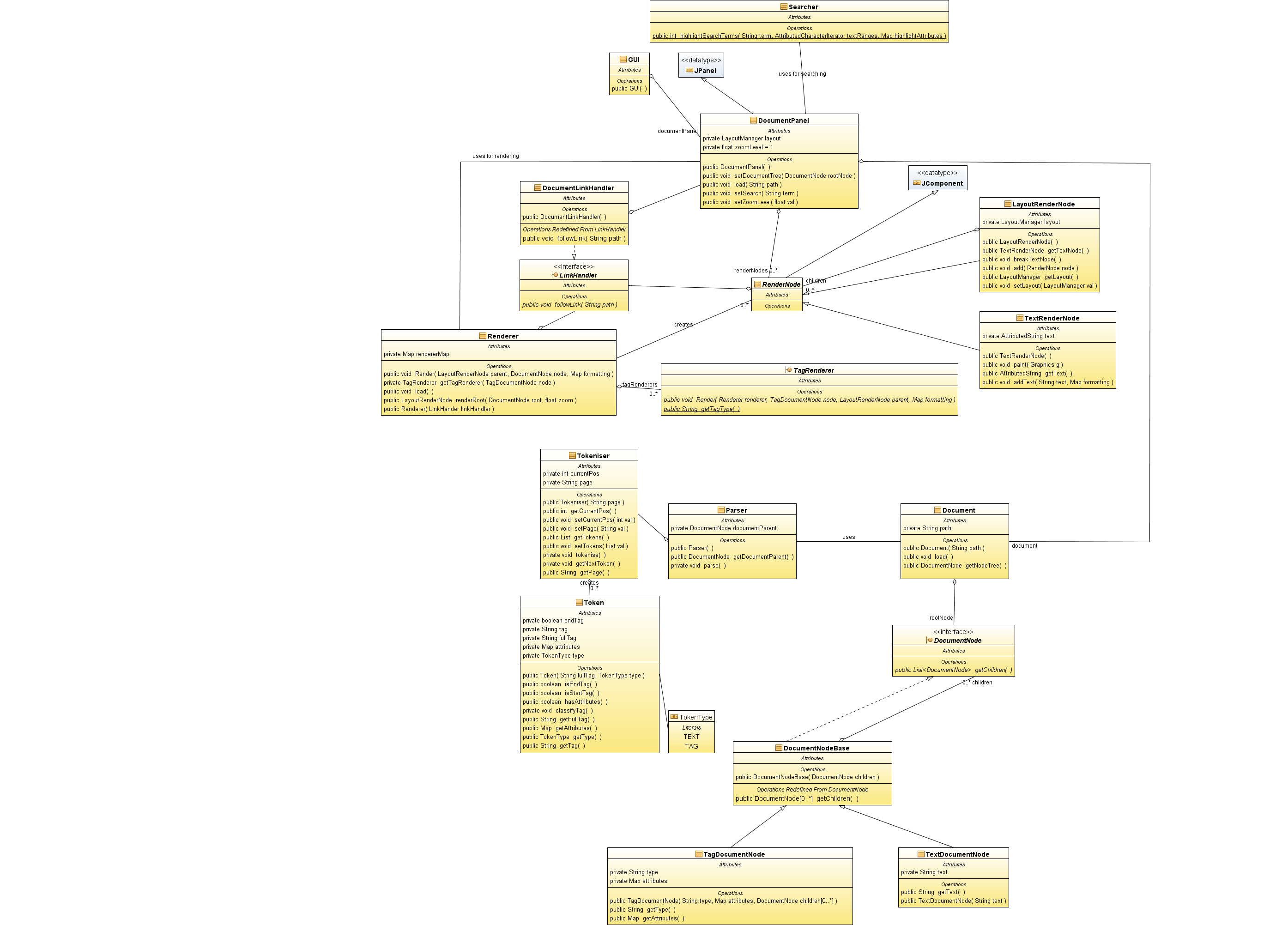


Figure 1: Browser Monkey Class Diagram

# Tokeniser & Parser

The tokeniser and the parser are responsible for turning a html document into a set of document nodes which can later be rendered to our surface.

## Tokeniser & Tokens

The tokeniser splits up the raw text of the html document into token object. A token object represents an atomic segment of the html. Consider the string “<b>bold</b>”: this would produce three tokens –“<b>”, “bold” and “</b>”.

### Prototype

A prototype tokeniser has been developed, using a scripting language called Ruby, to act as a proof of concept and as near- pseudocode like documentation.

class MonkeyTokeniser

#attr\_reader is ruby short hand for a getter method

# so in java we'd write getPage() and getToken

attr\_reader :page, :tokens

#in java this would be the public MonkeyTokeniser(String page){...}

def initialize(page)

#puts page

@current\_pos = 0

@page = page.to\_s

@tokens = Array.new #array of MonkeyTokens

tokenise

end

def get\_next\_token

if ?< == @page[@current\_pos] #is this a tag

if @page[(@current\_pos + 1), 3] == "!--" #is this a comment

#calculate length of token and move token

tag\_token\_end = @page.index('-->', @current\_pos + 1)

raise "malformed html" if tag\_token\_end == nil

@current\_pos += tag.length

else

#calculate length of token and move pointer - then add to token list

tag\_token\_end = @page.index('>', @current\_pos + 1)

raise "malformed html" if tag\_token\_end == nil

tag = @page[@current\_pos...tag\_token\_end + 1]

@tokens << MonkeyToken.new(tag, :tag)

@current\_pos += tag.length

end

else

#calculate length of text and move pointer - then add to token list

text\_token\_end = @page.index('<', @current\_pos)

text = @page[@current\_pos...text\_token\_end]

@tokens << MonkeyToken.new(text, :text)

@current\_pos += text.length

end

end

def tokenise

while @current\_pos < @page.length

get\_next\_token

end

end

end

class MonkeyToken

attr\_reader :full\_tag, :tag, :attributes, :type

def initialize(full\_tag, type)

#@variables are class variables

#<b>

@full\_tag = full\_tag

#:text or :tag

@type = type

#:foo => "bar"

@attributes = Hash.new

@end\_tag = false

# grab the tag name of the tag, if it's text then just keep the whole thing

@tag = full\_tag if type == :text

if type == :tag

#regex to get the a in <a href="b">

@tag = full\_tag.scan(/[\w:-]+/)[0]

if @tag.nil?

raise "Error, tag is nil: #{@full\_tag}"

end

classify\_tag

end

end

def has\_attributes?

return !@attributes.empty?

end

def is\_end\_tag?

return @end\_tag

end

def is\_start\_tag?

return !@end\_tag

end

def classify\_tag

#

# <a foo="test" bar="test" >

# produces three results: 1)a 2.1)foo 2.2)test 3.1)bar 3.2)test

#

# This function creates a hash of the attributes of a tag and determines if this tag is an end tag

#

# 'scan' breaks up a string based on a regex, with each group being a new element in an array. Sounds like hassle in java.

atts = @full\_tag.scan(/<[\w:-]+\s+(.\*)>/)

attr\_arr = atts[0].to\_s.scan(/\s\*([\w:-]+)(?:\s\*=\s\*("[^"]\*"|'[^']\*'|([^"'>][^\s>]\*)))?/m)

#this is a bit like a for each loop

attr\_arr.each do |n|

@attributes[n[0].downcase] = n[1]

end

#determine if the tag is an end tag by looking for a / before the tag name ( </b> )

end\_tag = @full\_tag.index('/', 0)

if end\_tag != nil

tag\_pos = @full\_tag.index(@tag, 0)

@end\_tag = true if end\_tag < tag\_pos

end

end

end

#little test and usage demo, feel free to butcher.

page = "<b>test</b><strong class='fakeBold' test='foo'>bold</strong>"

tokeniser = MonkeyTokeniser.new(page)

tokeniser.tokens.each do |n|

if n.type == :tag

string = "start\_tag: " if n.is\_start\_tag?

string = "end\_tag: " if n.is\_end\_tag?

string << n.tag

if n.has\_attributes?

n.attributes.each\_pair {|key, value| string << " with attrubute: #{key} is #{value}" }

end

puts string

end

puts "text:" + n.tag if n.type == :text

end

This code is able to correctly split up tokens and identify attributes. If provided with the text string "<b>test</b><strong class='fakeBold' test='foo'>bold</strong>"

then it produces the output:

start\_tag: b

text:test

end\_tag: b

start\_tag: strong with attrubute: class is 'fakeBold' with attrubute: test is 'foo'

text:bold

end\_tag: strong

The tokeniser works by keeping a counter of it’s position in the string to be evaluated then stepping though. If it sees a ‘<’ then it checks forward to the next “>” and assumes that to be a tag type token (unless it’s a comment). It then produces a new token object. A token object is able to be passes a string like “<foo att=’bar’>” and correctly identify that:

1. It’s a start tag
2. It’s a foo tag
3. It has an attribute “att” with the value “bar”

Each token object is kept by the tokeniser in a list. Although not demonstrated in this prototype, the tokeniser must also perform basic syntactic error correction – that is to say that the tokeniser must be able to correct “bad” html where tags are not closed or invalid symbols are present.

## Parser

The parser takes a stream of tokens from the tokeniser and applies a set of rules on those tokens. It then produces a list of DocumentNodes of various types. The parser checks that:

1. Each element is an element type we can render
2. Each element is correctly nested in appropriate elements
3. Each element has valid attributes
4. Each element correctly terminates

CODE OR PROTOTYPE HERE

# Renderer

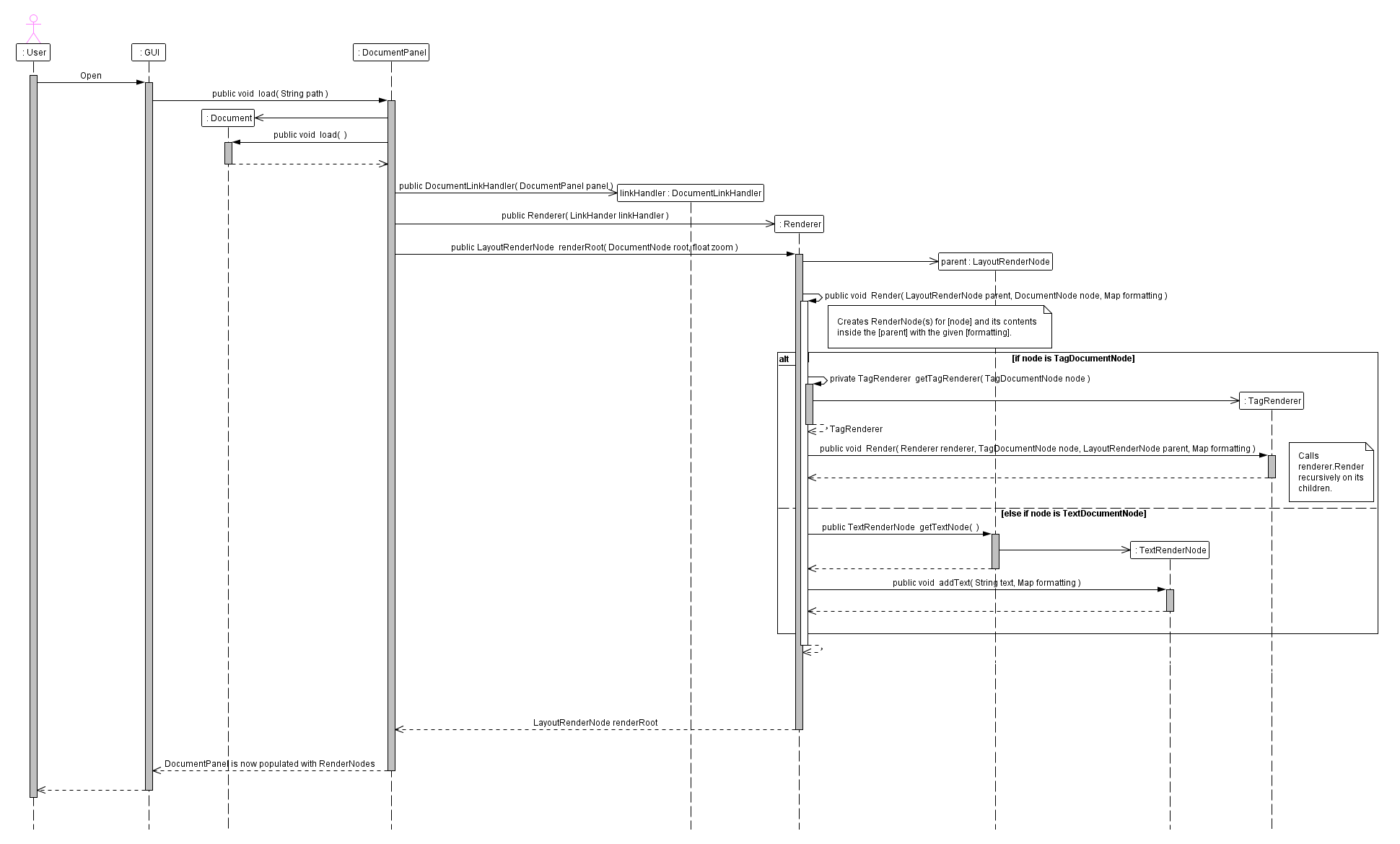


Figure 2: Viewing the Manual

Above is the sequence diagram for the rendering process – this diagram, for sake of brevity, omits the tokeizer and parser segments. The renderer remains largely unchanged from the high level design.

# Logging

Java provides a built in logging tool which is able to write well formatted and time stamped logs to a file of our choosing. A thin static wrapper called BrowserMonkeyLogger will be produced that will allow the programmer – from anywhere that the utilities classes have been imported – to write to the log at a given level of severity: ranging from info to fatal error.

# Search

Searching is one of the areas where there is some change from the high level design. Due to the built in java API we’re using in the DocumentPanel, which enables highlighting on the view level, there is no longer a need to split up render nodes as we previously thought.

The below sequence diagram illustrates the steps that are required.

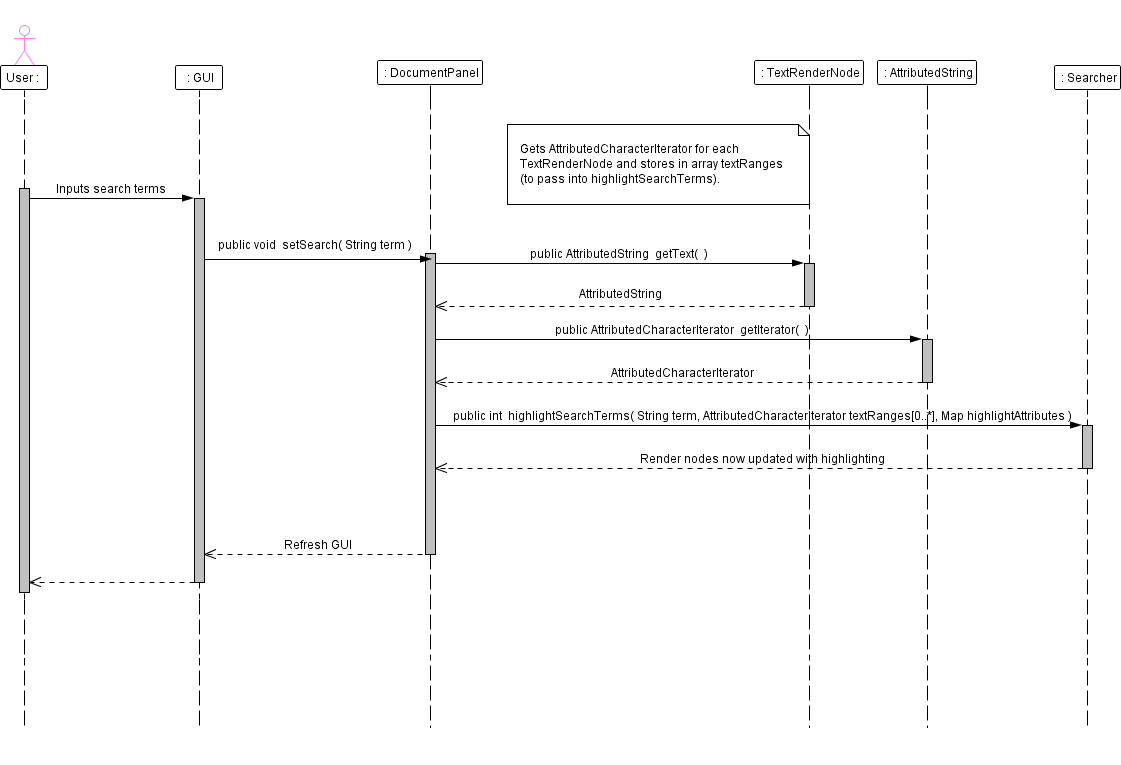


Figure 3 Sequence Diagram of Search

# Zoom

Zoom is unchanged from the high level design. The renderer simply re-renders with a new zoom level. When a fresh page is rendered without a zoom level – we actually set the internal zoom level to 100%.

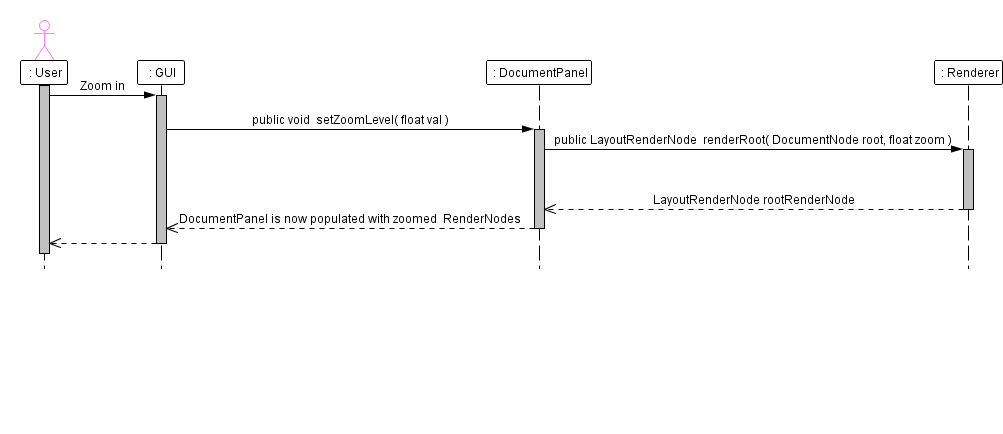


Figure 4 Zoom Sequence Diagram

# Follow Link

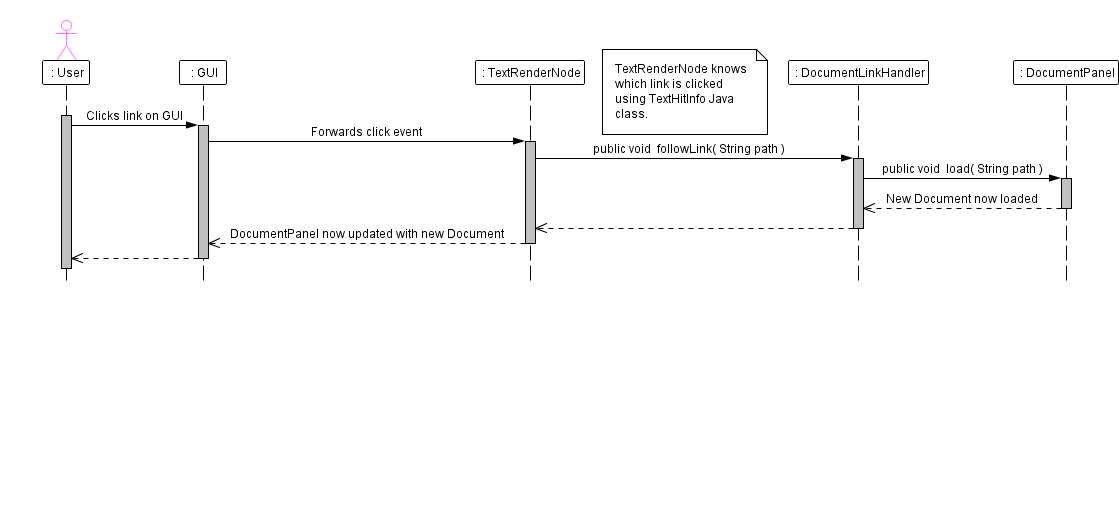


Figure 5 Follow Link Sequence Diagram

# Tag Interpretation Using Reflection

The BrowserMonkey program needs to interpret a large number of tags and do this process in a modular way to allow for future usage of the tag handling system. Most importantly this must be done very efficiently. It would be possible create a html tag handler by using an enormous if-else statement but this would be hard to debug and horrible to reuse or update (for example to new HTML standards). This is why we have decided to use reflection.

## In Theory

The program can use an external file (such as a config file) that is easily editable to lookup required information for use while the program is running. An alternative way of doing this is having the information in the file loaded into the program when is it initially run. This allows for the addition of useful modular features to a program.

Reflection is the process by which a computer program can observe and modify its own structure and behaviour.

Reflection allows the programmer to use multiple different classes and methods in a generic way, perfect for something like html rendering where you have many different processes used to draw the code. Reflection is also very handy for creating a modular piece of software such as a tag interpreter.

Essentially when you instantiate a class using a reflection method you don’t need to know what the class is or what methods it has, the reflection implementation will have methods that will allow you to grab any methods that class has and invoke the one you need.

## In Java

Now I will give a brief introduction of how these techniques could be implemented in Java. Java has a properties class that can be used to implement this kind of idea. There’s a specific convention to use to create a properties file which can be read by included methods. This will then generate a kind of hash table that can be accessed easily.

### External Files

To access the external file to be read into the properties object:

Code:

Properties properties = new Properties();

properties.load(“<root>:\properties.txt”);

You can then access the properties in this file using a Hash table-like key function:

Code:

String thisPropertyIsSetTo = properties.get(thisProperty);

This Code will find the property key ‘thisProperty’ and set the String thisPropertyIsSetTo to the value stored in the properties file as the value for ‘thisProperty’.

### Reflection

Java has a built in class for handling reflection: java.lang.reflect

Here is some code for a simple example of reflection. Code:

import java.lang.reflect.\*;

public class DumpMethods {

public static void main(String args[])

{

try {

Class c = Class.forName(args[0]);

Method m[] = c.getDeclaredMethods();

for (int i = 0; i < m.length; i++)

System.out.println(m[i].toString());

}

catch (Throwable e) {

System.err.println(e);

}

}

}

This code reads the command line arguments and compares the first argument to all available classes in java if it finds a matching class it will output a toString of each method that class contains. If there is no matching class it will throw an error.

## In BrowserMonkey Browser

To apply the above techniques in the BrowserMonkey Browser we will use a Class for each HTML tag that extends an abstract class called TagRenderer that has a render method which for each different class tag will have code used for rendering the item related to the current tag.

There will be an external file that will be loaded into a Map within the Renderer class during its constructor code. We will then use the map to relate the tag the renderer is looking at to the name of the equivalent mini Tag Class. Next the renderer will use reflection to get the necessary rendering method and pass it the current attributes and variables so that it can build the required component based on the tag.

### Pseudo Code

A simple pseudo code representation of how the above ideas would work within the program:

Each mini Tag Class will have the required render method for the tag they relate to. This will be so because it will implement the TagRenderer abstract class. The render method is called repeatedly and relies on information from the former rendering methods that have been run above it in the document node tree.

The signature for the render method is as follows: render(Renderer renderer, TagDocumentNode tag, LayoutRenderNode parent, Map formatting).

The render method will decide what needs to be done with the tag based on the current formatting provided in the formatting variable.

In the Renderer:

During Constructor:

Map rendererMap = new Map();

properties.load(<root>:\properties.txt)

tags = properties.propertyNames()

for each tag in tags

String tagClass = properties.get(tag)

TagRenderer thisTagRenderer = new Class.forName(tagClass)

rendererMap.add(tag,thisTagRenderer)

The above code builds the renderMap from the properties file when the renderer is first initialised. This allows the renderer to find the TagRenderers later during the rendering process:

During Rendering Process:

Recursively going over each documentNode:

if documentNode is not plain text

currentTagRenderer = rendererMap.get(Tag)

currentTagRenderer.render(documentNode as TagDocumentNode)

Rendering is a recursive process, document nodes will continue render their children.