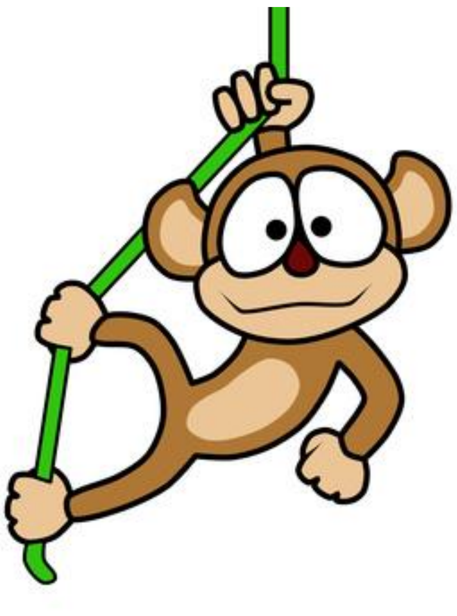
BrowserMonkey



Low Level Design Document

v 1.0

Software House One

Version History

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

Document version: 1.0

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: Sohani Amiruzzaman 2009/04/30

Last modification date: 2009/04/30

Contents

Notes

Since the last deliverable, we have moved Subversion server to be hosted on Google Code. To request admin rights on this project, please email any of the team leaders.

We made this move because of a few problems in the old system which make updating and committing a time consuming process. The old svn directory has been frozen and the new one has all the changes we've made to the project from the point we moved over.

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

We have also been forced to make a change in project leader since our previous one was no longer able to stay with us for personal reasons. We lost a member of the QA team on similar grounds as well. We are now left with only 9 team members.

Architectural Design

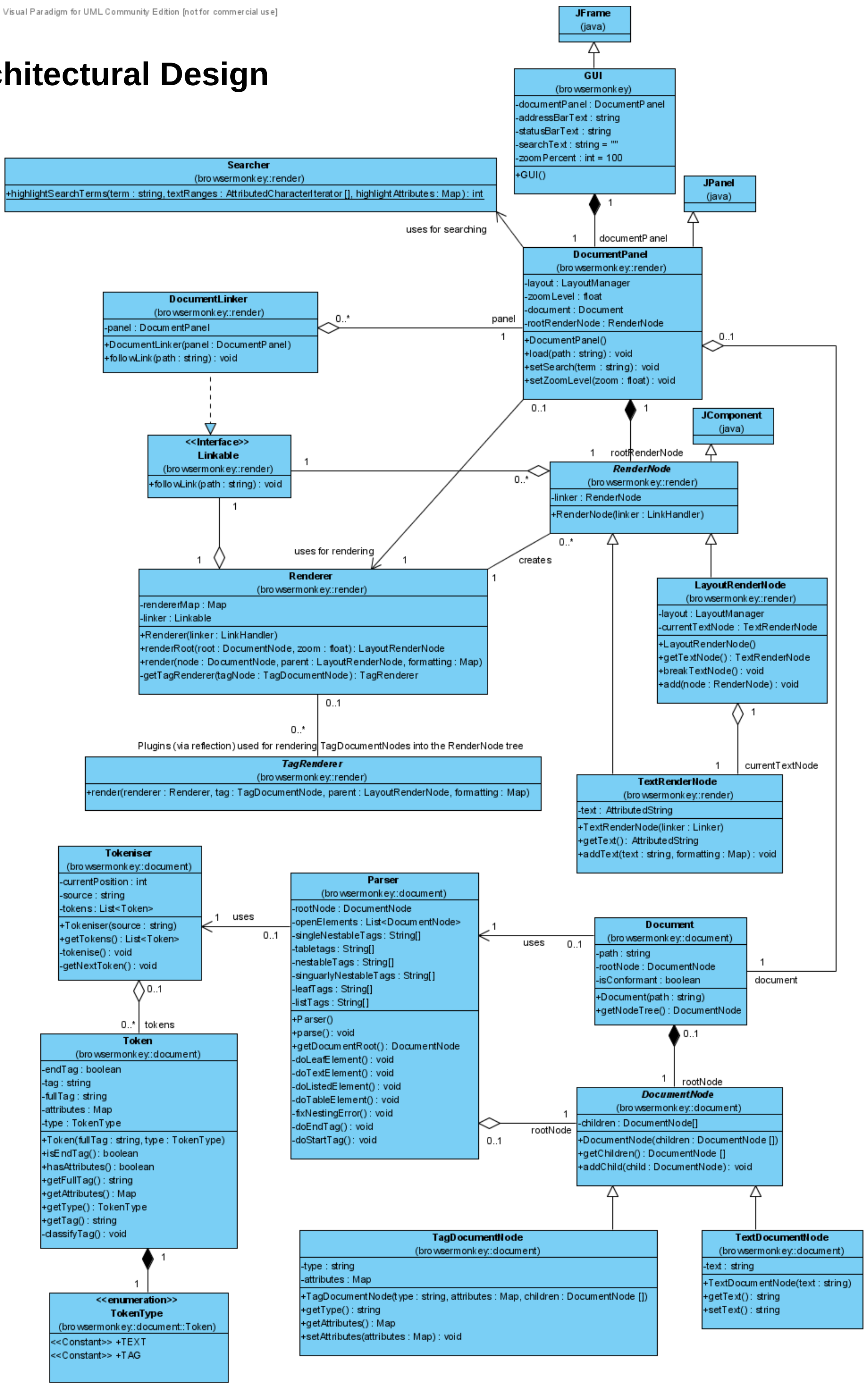

JFrame
(java)
GUI
(browsermonkey)
-documentPanel : DocumentPanel
-addressBarText : string
-statusBarText : string
-searchText : string = ""
-zoomPercent : int = 100
+GUI()
JPanel
(java)
Searcher
(browsermonkey::render)
+highlightSearchTerms(term : string, textRanges : AttributedCharacterIterator [], highlightAttributes : Map) : int
uses for searching
1
1
documentPanel
DocumentPanel
(browsermonkey::render)
-layout : LayoutManager
-zoomLevel : float
-document : Document
-rootRenderNode : RenderNode
+DocumentPanel()
+load(path : string) : void
+setSearch(term : string) : void
+setZoomLevel(zoom : float) : void
DocumentLinker
(browsermonkey::render)
-panel : DocumentPanel
+DocumentLinker(panel : DocumentPanel)
+followLink(path : string) : void
0..*
panel
1
0..1
0..1
1
JComponent
(java)
1
rootRenderNode
<<Interface>>
Linkable
(browsermonkey::render)
+followLink(path : string) : void
1
0..*
RenderNode
(browsermonkey::render)
-linker : RenderNode
+RenderNode(linker : LinkHandler)
1
1
uses for rendering
1
0..*
creates
1
Renderer
(browsermonkey::render)
-rendererMap : Map
-linker : Linkable
+Renderer(linker : LinkHandler)
+renderRoot(root : DocumentNode, zoom : float) : LayoutRenderNode
+render(node : DocumentNode, parent : LayoutRenderNode, formatting : Map)
-getTagRenderer(tagNode : TagDocumentNode) : TagRenderer
LayoutRenderNode
(browsermonkey::render)
-layout : LayoutManager
-currentTextNode : TextRenderNode
+LayoutRenderNode()
+getTextNode() : TextRenderNode
+breakTextNode() : void
+add(node : RenderNode) : void
0..1
0..*
Plugins (via reflection) used for rendering TagDocumentNodes into the RenderNode tree
1
1
currentTextNode
TagRenderer
(browsermonkey::render)
+render(renderer : Renderer, tag : TagDocumentNode, parent : LayoutRenderNode, formatting : Map)
TextRenderNode
(browsermonkey::render)
-text : AttributedString
+TextRenderNode(linker : Linker)
+getText() : AttributedString
+addText(text : string, formatting : Map) : void
Tokeniser
(browsermonkey::document)
-currentPosition : int
-source : string
-tokens : List<Token>
+Tokeniser(source : string)
+getTokens() : List<Token>
-tokenise() : void
-getNextToken() : void
1
uses
0..1
Parser
(browsermonkey::document)
-rootNode : DocumentNode
-openElements : List<DocumentNode>
-singleNestableTags : String[]
-tabletags : String[]
-nestableTags : String[]
-singuarlyNestableTags : String[]
-leafTags : String[]
-listTags : String[]
+Parser()
+parse() : void
+getDocumentRoot() : DocumentNode
-doLeafElement() : void
-doTextElement() : void
-doListedElement() : void
-doTableElement() : void
-fixNestingError() : void
-doEndTag() : void
-doStartTag() : void
1
uses
0..1
Document
(browsermonkey::document)
-path : string
-rootNode : DocumentNode
-isConformant : boolean
+Document(path : string)
+getNodeTree() : DocumentNode
1
document
0..1
0..*
tokens
Token
(browsermonkey::document)
-endTag : boolean
-tag : string
-fullTag : string
-attributes : Map
-type : TokenType
+Token(fullTag : string, type : TokenType)
+isEndTag() : boolean
+hasAttributes() : boolean
+getFullTag() : string
+getAttributes() : Map
+getType() : TokenType
+getTag() : string
-classifyTag() : void
0..1
1
rootNode
DocumentNode
(browsermonkey::document)
-children : DocumentNode[]
+DocumentNode(children : DocumentNode [])
+getChildren() : DocumentNode []
+addChild(child : DocumentNode) : void
1
rootNode
0..1
1
1
<<enumeration>>
TokenType
(browsermonkey::document::Token)
<<Constant>> +TEXT
<<Constant>> +TAG
TagDocumentNode
(browsermonkey::document)
-type : string
-attributes : Map
+TagDocumentNode(type : string, attributes : Map, children : DocumentNode [])
+getType() : string
+getAttributes() : Map
+setAttributes(attributes : Map) : void
TextDocumentNode
(browsermonkey::document)
-text : string
+TextDocumentNode(text : string)
+getText() : string
+setText() : string

Figure 1: Browser Monkey Class diagram

Some of the relationships in the above diagram are compositions as opposed to aggregation. The difference is subtle, and is mostly a conceptual one, where the implementation could often be identical. We have made the distinction mostly based on lifetime control. The composition relationships in our system are the following:

- GUI is composed of a DocumentPanel
    - The DocumentPanel as used in our application has its lifetime controlled by the GUI. The GUI instantiates it, and when the GUI ceases to exist (is closed), the DocumentPanel should also be destructed.
- DocumentPanel is composed of RenderNodes
    - The RenderNodes exist expressly for rendering parts of the document in the DocumentPanel. When the DocumentPanel is destructed, so should the RenderNodes be.
- Document is composed of DocumentNodes
    - Analogous to RenderNodes for the DocumentPanel, the DocumentNodes make up the Document.
- Token contains (as composition) TokenType
    - The TokenType enumeration is only instantiated and used by the Token class, it is only inspected as part of a Token by others (the Tokeniser). When a Token is destructed, its TokenType instance should be destructed also.

Tokeniser & Parser

The tokeniser and the parser are responsible for turning an HTML document into a tree of DocumentNodes which can later be rendered into a RenderNode tree.

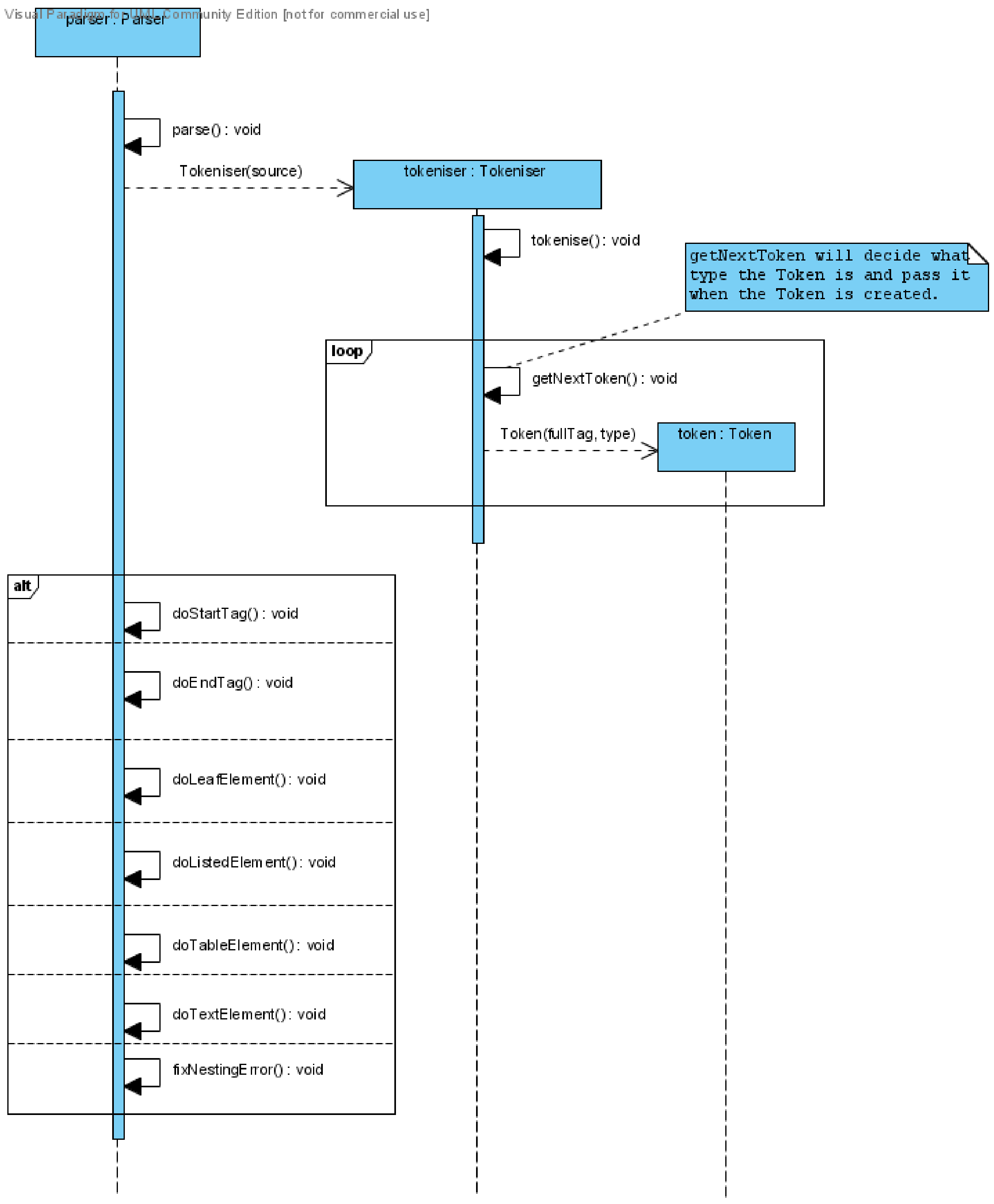


Figure 2: Parser and Tokeniser Sequence Diagram

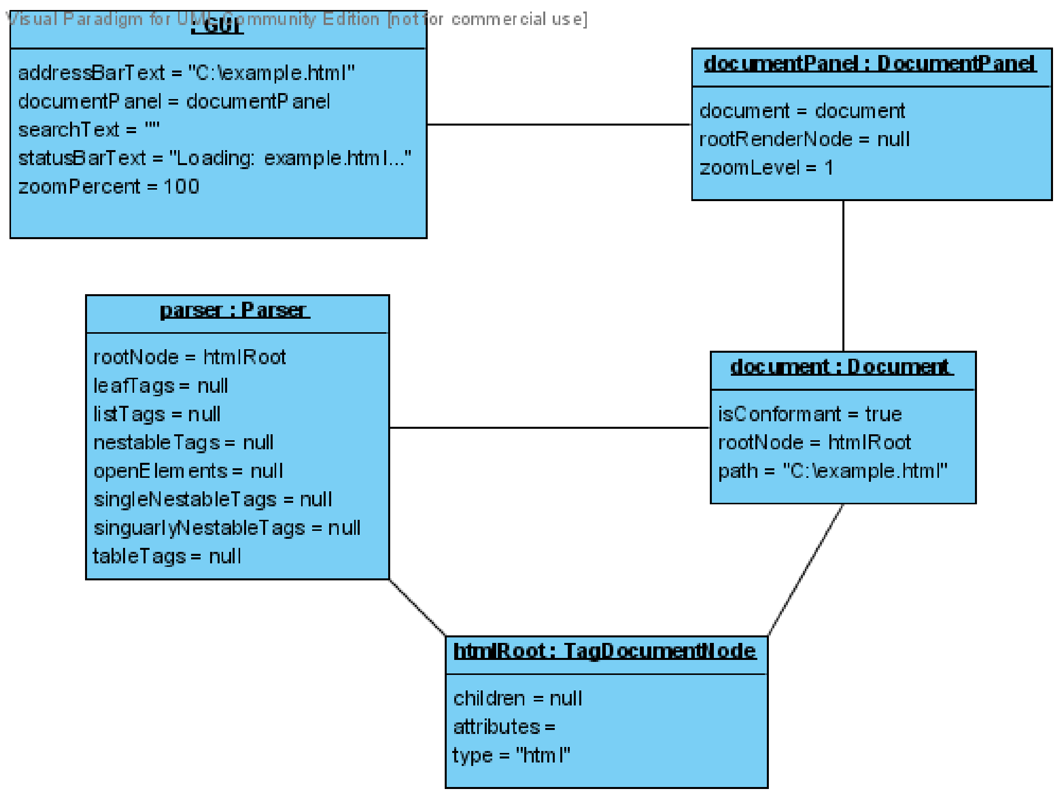


Figure 3: Parser Initial Object Diagram

Before the parsing process, the GUI has the Document loaded into the DocumentPanel, but there are no render nodes yet, and the Document and Parser contain only the root DocumentNode.

To demonstrate this process, the following example HTML file has been constructed:

```
<html>
<body>
Test page<br/>
Test <b>bold text</b>
</body>
</html>
```

The Object Diagram below shows the result of the parser as executed on this HTML file.

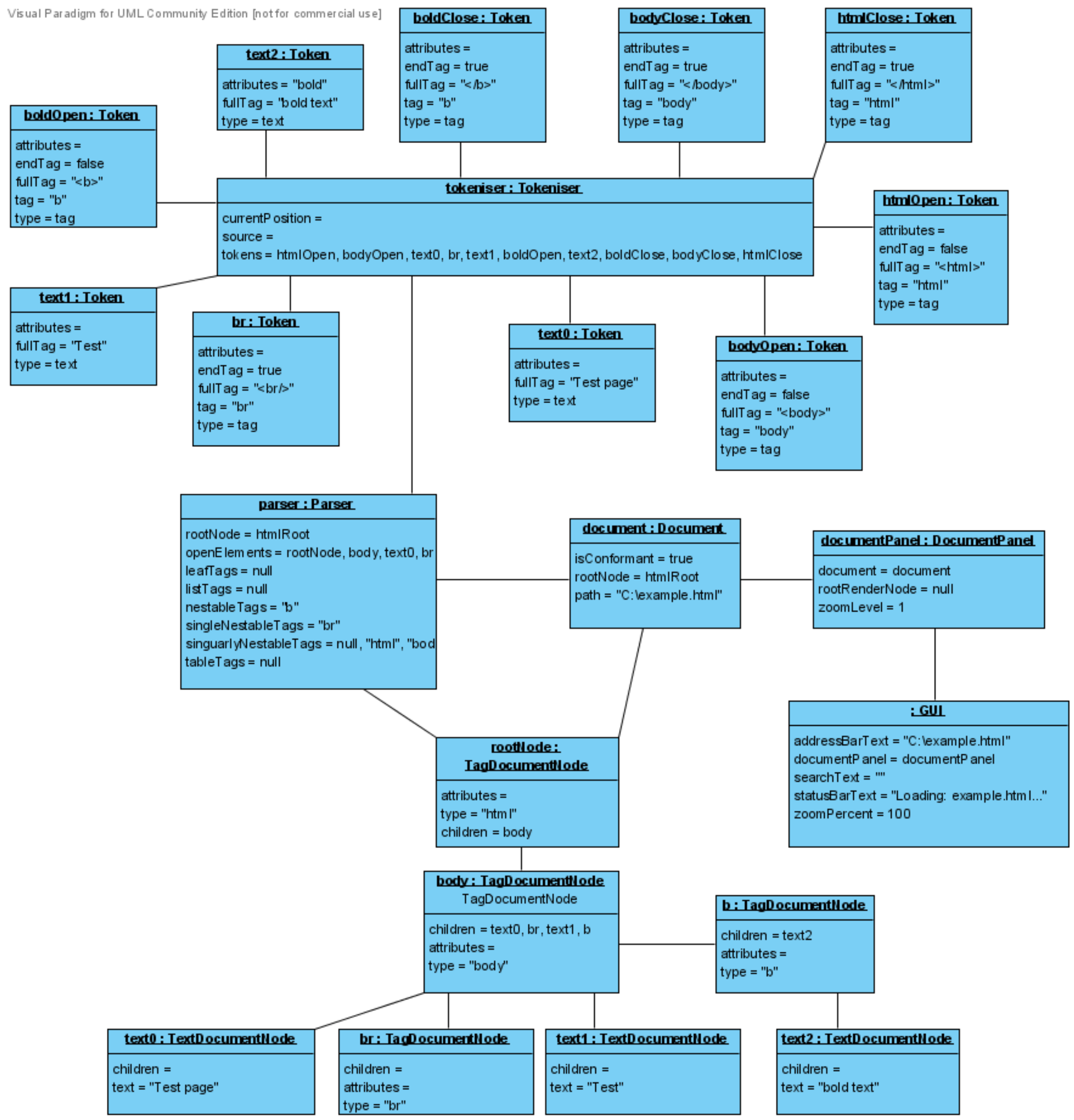


Figure 4: Parser Completed Object Diagram

Once the parser is completed, the Tokeniser holds all the tokens, and the parser has used these tokens to create the DocumentNode tree, which is held by the Document.

Tokeniser and Tokens Detail

The tokeniser splits up the raw text of the HTML document into token objects. 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- pseudo-code 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] if text_token_end != nil
      text = @page[@current_pos...@page.length] if  text_token_end == nil

      @tokens << MonkeyToken.new(text, :text)
      @current_pos = @page.length if 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
```

The tokeniser works by keeping a counter of its position in the string to be evaluated then stepping through. 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 passed 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 Detail

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

It maintains a series of lists which detail elements and how they are to be treated. In the implementation, these lists will be created from a properties file when the parser is created.

The parser is able to correct basic HTML errors, like the incorrect nesting of tags and the improper use of tags. It is however fairly limited and does not always produce the correct output when given extremely poor HTML.

Prototype

```
class MonkeyParser

  attr_reader :original_page, :document_node


  def initialize(page)
    @original_page = page
    @tokeniser = MonkeyTokeniser.new page
    @tokens = @tokeniser.tokens


    #define all the tags and their general attributes. In the actual implementation
    #this will be populated by a properties file.

    #these tags are to demo, and are by no means an exhaustive list.

    @single_nestable_tags = ['HTML','head','body'] #tags that can only be used once
    @table_tags = ['table','tr','td'] #table tags need a special case
    @nestable_tags = ['b','i','strong','em'] #normal, nestable, tags
    @singuarly_nestable_tags = ['p'] #these tags cannot be nested inside themselfs
    @leaf_tags =['br','img'] #can have no children
    @listed_tags =['li','ol','ul'] #list elements also need a special case
    @open_elements = Array.new


  end

  def parse

    @tokens.each_with_index do |token,i|

      #add HTML if needed
      if i == 0 && token.tag != “html”
        @document_node = MonkeyDocumentNode.new(“html”, :tag)
        @open_elements << @document_node
      end
      if i == 0 && token.tag == “html”
        @document_node = MonkeyDocumentNode.new(“html”, :tag)
        @open_elements << @document_node
      end


      if token.type == :tag
        if token.is_start_tag?

          #if it's a table tag or if a row has been opened but not a cell - add the appropriate
elements
          if @table_tags.member? token.tag
            do_table_element token
          elsif @open_elements.length >= 1
            if @open_elements.last.tag == "tr" || @open_elements.last.tag == "table"
              do_table_element MonkeyToken.new '<td>', :tag
            end
          end

          #perform listed tag functions
          if @listed_tags.member? token.tag
            do_listed_element token
          end

          #pre elements should not have any children, just text
          if @open_elements.length >= 1 && @open_elements.last.tag == "pre"
            @open_elements.last.text = @open_elements.last.text + token.full_tag
          end

          #For singularly nestable tags, check if the last tag is the same. If it is
          #fix the nesting, if not - carry on.
          if @singuarly_nestable_tags.member? token.tag
            if @open_elements.length > 1 && @open_elements.last.tag == token.tag
              do_end_token token
            end
            do_start_token token
          end

          #basic nestable tag
          if @nestable_tags.member? token.tag
            do_start_token token
          end

          #add the leaf tag
          if @leaf_tags.member? token.tag
            do_leaf_element token
          end

        else
          #close and remove the token
          do_end_token token
        end
      else
        #add a text element - but not without checking the state of the tables.
        if @open_elements.length >= 1
          if @open_elements.last.tag == "tr" || @open_elements.last.tag == "table"
            do_table_element MonkeyToken.new '<td>', :tag
          end
        end
        do_text_element token
      end

    end
  end


  #Does a standard list. If the user chooses to not close li tags then it assumes that the
next li
```

```
#signifies the start of the tag. Also, if no list type is given, it defaults to ul
def do_listed_element token
  if token.tag == 'li'
    if @open_elements.length >= 1
      if @open_elements[-1].tag == "ol" || @open_elements[-1].tag == "ul"
        do_start_token token
      elsif @open_elements[-1].tag == "li"
        do_end_token @open_elements[-1]
        do_start_token token
      else
        do_listed_element MonkeyToken.new '<ul>', :tag
        do_start_token token
      end
    else
      do_table_token MonkeyToken.new '<ul>', :tag
      do_start_token token
    end

  elsif token.tag == 'ol' || token.tag == 'ul'
    do_start_token token
  end

end


#ensures that tables are properly nested.
def do_table_element token
  if token.tag == 'td'
    if @open_elements.length >= 1
      if @open_elements[-1].tag == "tr"
        do_start_token token
      else
        do_table_element MonkeyToken.new '<tr>', :tag
        do_start_token token
      end
    else
      do_table_token MonkeyToken.new '<tr>', :tag
      do_start_token token
    end

  elsif token.tag == 'tr'
    if @open_elements.length >= 1
      if @open_elements[-1].tag == "table"
        do_start_token token
      else
        do_table_element MonkeyToken.new '<table>', :tag
        do_start_token token
      end
    else
      do_table_element MonkeyToken.new '<table>', :tag
      do_start_token token
    end

  elsif token.tag == "table"
    do_start_token token
    end
  end

  def do_leaf_element token
    @open_elements.last << MonkeyDocumentNode.new(token.tag, token.type,
token.attributes)
  end

  def do_text_element token
    do_leaf_element token
  end

  def fix_nesting_error token
    @open_elements.reverse_each do |element|
      if element.tag == token.tag
        @open_elements.delete element
        break
      end
    end
  end

  def do_end_token token
    if token.tag == @open_elements.last.tag
      @open_elements.pop
    else
      fix_nesting_error token
    end
  end



  #adds a new standard token onto the last open element and appends a new element
onto
  #that list
  def do_start_token token
    new_node = MonkeyDocumentNode.new(token.tag, token.type, token.attributes)

    @open_elements.last << new_node
    @open_elements << new_node
    @document_node << @open_elements.first if @open_elements.size == 1
  end

end

class MonkeyDocumentNode
  #A MonkeyDocumentNode is a node in a tree. It has children and a parent.
  #In the implementation, two types of node will exist - tag and text.


  #both a getter and a setter
  attr_accessor :type, :tag, :attributes, :children

  def initialize(value, type, attributes = Hash.new)

    @tag = value
```

```ruby
    @type = type
    @children = Array.new
    @attributes = attributes
  end


  #aliases for the tag method.
  def text
    return @tag
  end

  def text=(text)
    @tag = text
  end


  #This allow to add documentnode to the children of a parent node  documentnode1 <<
documentnode2
  def <<(value)
    @children << value
    return value
  end

  #each provides the standard ruby way of iterating over a collection
  #We're traversing the tree recursively and each time we find an element we yield back
to the calling block.
  def each
    yield value
    @children.each do |child_node|
      child_node.each { |e| yield e }
    end
  end

  #prints as if it was still HTML
  def friendly_print
    if @type == :tag
      return "<"+@tag+"/>" if @children.empty?
      string = "<"+@tag+">"
      @children.each do |n|
        string <<  n.friendly_print
      end
      string << "</"+@tag+">"
    else
      return @tag
    end

  end
end
```

Prototype Testing

In order to informally test the workings of the tokeniser and the parser, a small test suite has been written. The code for this is listed here:

```
def run_test page, name

  puts ""
  puts "-------------------------------------------"
  puts "Test name: "+name
  puts "testing with string: "+page
  puts "testing tokeniser:"
  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?
        string << " with attributes:"
        n.attributes.each_pair {|key, value| string << " #{key}=#{value}"}
      end
      puts string
    end
    puts "text:" + n.tag if n.type == :text
  end

  puts "testing parser:"
  parser = MonkeyParser.new(page)
  parser.parse
  puts parser.document_node.friendly_print
  puts "-------------------------------------------"

end
```

The test will focus on the edge cases of the parser – correcting mistakes in the HTML.

The tests are defined as so:

```
run_test "<b>hello<i>world</i></b>", "Simple"
run_test "<b>hello<i>world</b></i>", "Incorrect Nesting"
run_test "<tr>hello</tr>", "Missing table cell"
run_test "<html><table><td>hello</table></td></html>", "Badly Formed Table"
run_test "<table><tr>hello<table>world</table></tr></table>","badly formed and
nested tables"
run_test "<li>hello<li>world", "li shortcut"
run_test "hello world", "just text"
run_test "<p>hello<p>world</p></p>", "p nesting"
run_test "<ol><b>hello</b><li>world</li></ol>", "element inside list"
run_test "<br>hello</br>", "nest attempt on br"
run_test "<html><body>Test page<br/>Test <b>bold text</b></body></html>",
"Standard Example"
```

And the output of this script is as follows:

```
-------------------------------------------
Test name: Simple
testing with string: <b>hello<i>world</i></b>
testing tokeniser:
start_tag: b
text:hello
start_tag: i
text:world
end_tag: i
end_tag: b
testing parser:
<html><b>hello<i>world</i></b></html>
-------------------------------------------

-------------------------------------------
Test name: Incorrect Nesting
testing with string: <b>hello<i>world</b></i>
testing tokeniser:
start_tag: b
text:hello
start_tag: i
text:world
end_tag: b
end_tag: i
testing parser:
<html><b>hello<i>world</i></b></html>
-------------------------------------------

-------------------------------------------
Test name: Missing table cell
testing with string: <tr>hello</tr>
testing tokeniser:
start_tag: tr
text:hello
end_tag: tr
testing parser:
<html><table><tr><td>hello</td></tr></table></html>
-------------------------------------------

-------------------------------------------
Test name: Badly Formed Table
testing with string: <html><table><td>hello</table></td></html>
testing tokeniser:
start_tag: HTML
start_tag: table
start_tag: td
text:hello
end_tag: table
end_tag: td
end_tag: HTML
testing parser:
```

```
<html><table><tr><td>hello</td></tr></table></html>
-------------------------------------------

-------------------------------------------
Test name: badly formed and nested tables
testing with string: <table><tr>hello<table>world</table></tr></table>
testing tokeniser:
start_tag: table
start_tag: tr
text:hello
start_tag: table
text:world
end_tag: table
end_tag: tr
end_tag: table
testing parser:
<html><table><tr><td>hello<table><tr><td>world</td></tr></table></td></tr><
/table></html>
-------------------------------------------

-------------------------------------------
Test name: li shortcut
testing with string: <li>hello<li>world
testing tokeniser:
start_tag: li
text:hello
start_tag: li
text:world
testing parser:
<html><ul><li>hello</li><li>world</li></ul></html>
-------------------------------------------

-------------------------------------------
Test name: just text
testing with string: hello world
testing tokeniser:
text:hello world
testing parser:
<html>hello world</html>
-------------------------------------------

-------------------------------------------
Test name: p nesting
testing with string: <p>hello<p>world</p></p>
testing tokeniser:
start_tag: p
text:hello
start_tag: p
text:world
end_tag: p
end_tag: p
testing parser:
<html><p>hello</p><p>world</p></html>
-------------------------------------------

-------------------------------------------
Test name: element inside list
testing with string: <ol><b>hello</b><li>world</li></ol>
testing tokeniser:
```

```
start_tag: ol
start_tag: b
text:hello
end_tag: b
start_tag: li
text:world
end_tag: li
end_tag: ol
testing parser:
<html><ol><b>hello</b><li>world</li></ol></html>
-------------------------------------------

-------------------------------------------
Test name: nest attempt on br
testing with string: <br>hello</br>
testing tokeniser:
start_tag: br
text:hello
end_tag: br
testing parser:
<html><br/>hello</html>
-------------------------------------------

-------------------------------------------
Test name: Standard Example
testing with string: <html><body>Test page<br/>Test <b>bold
text</b></body></html>
testing tokeniser:
start_tag: html
start_tag: body
text:Test page
start_tag: br
text:Test
start_tag: b
text:bold text
end_tag: b
end_tag: body
end_tag: html
testing parser:
<html><body>Test page<br/>Test <b>bold text</b></body></html>
-------------------------------------------
```

All but the last of these tests test bad data, and as such there is not standard way to know if it the corrections are correct. However, we believe that the parser prototype is able to make a good attempt at invalid code while being compliant on standard code (as proved by the standard example).

Renderer

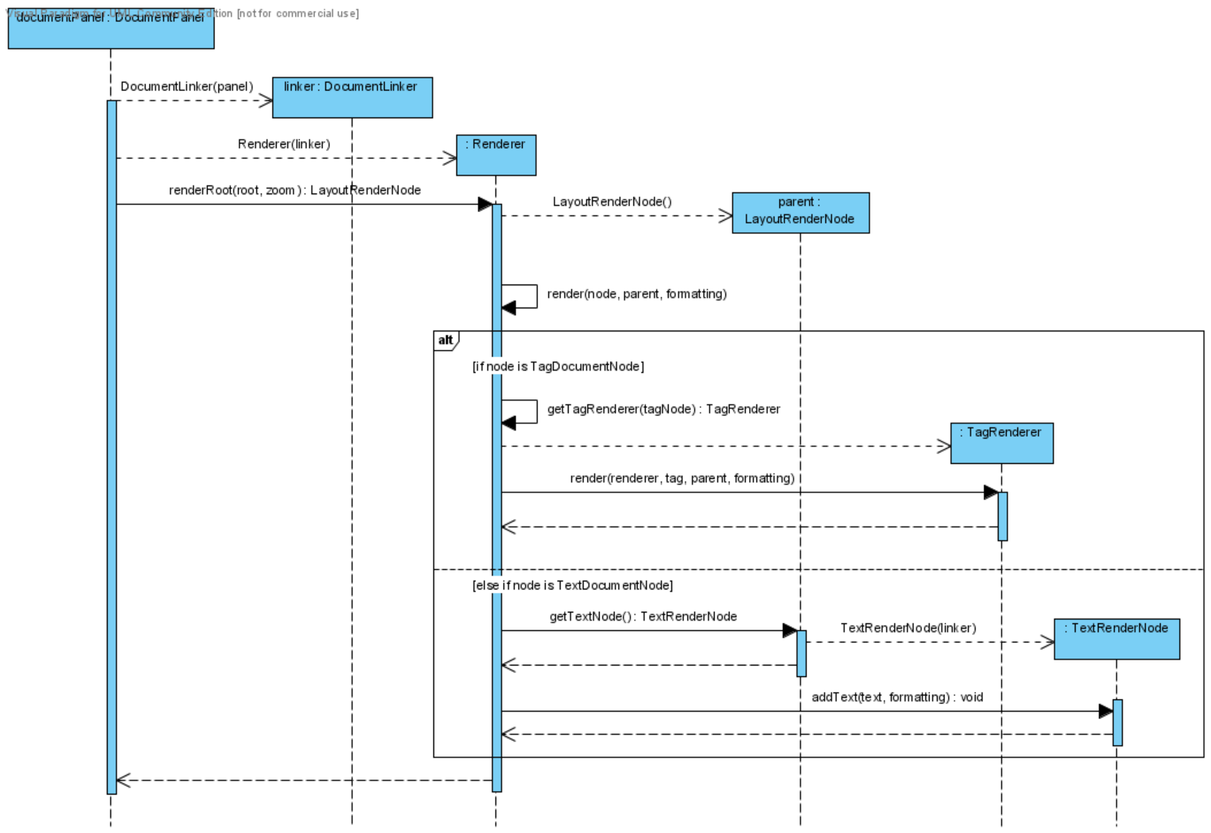

documentPanel : DocumentPanel
DocumentLinker(panel)
linker : DocumentLinker
Renderer(linker)
: Renderer
renderRoot(root, zoom) : LayoutRenderNode
LayoutRenderNode()
parent : LayoutRenderNode
render(node, parent, formatting)
alt
[if node is TagDocumentNode]
getTagRenderer(tagNode) : TagRenderer
: TagRenderer
render(renderer, tag, parent, formatting)
[else if node is TextDocumentNode]
getTextNode() : TextRenderNode
TextRenderNode(linker)
: TextRenderNode
addText(text, formatting) : void

Figure 5: Renderer Sequence Diagram: Rendering the DocumentNode tree into RenderNodes in the DocumentPanel

The renderer takes the DocumentNode tree as outputted by the parser, and creates a RenderNode tree.

The same example HTML file as used for the parser will be used for demonstration:

```
<html>
<body>
Test page<br/>
Test <b>bold text</b>
</body>
</html>
```

Before the renderer process begins, the state is as follows:

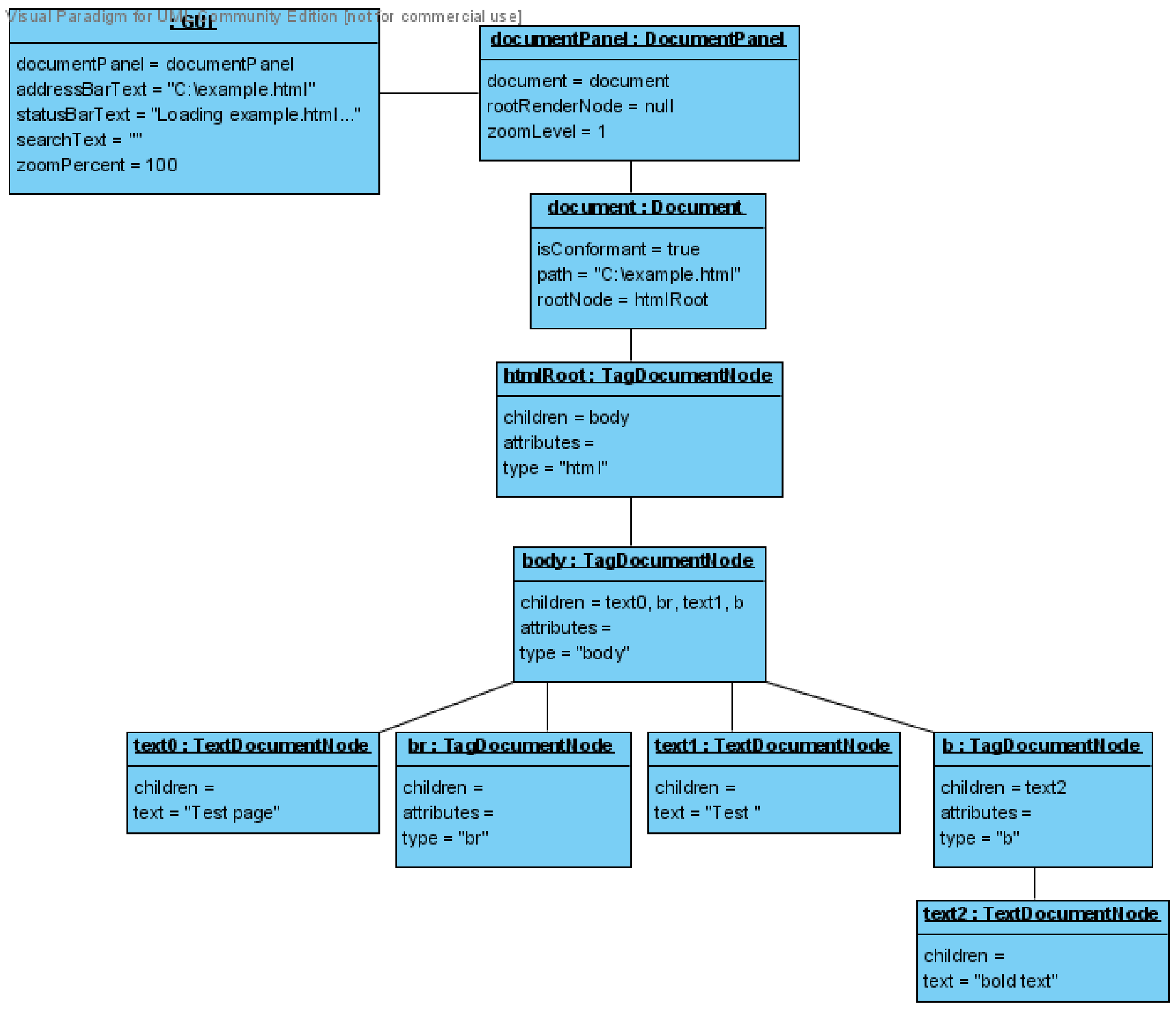


Figure 6: Renderer Initial Object Diagram

The DocumentPanel contains no rootRenderNode, so at this point no components are visible, and the render tree hasn't been created. The parser has provided the Document with a DocumentNode tree, the root of which is the HTML tag, a TagDocumentNode object called htmlRoot.

After the process, the following is the state of the system:

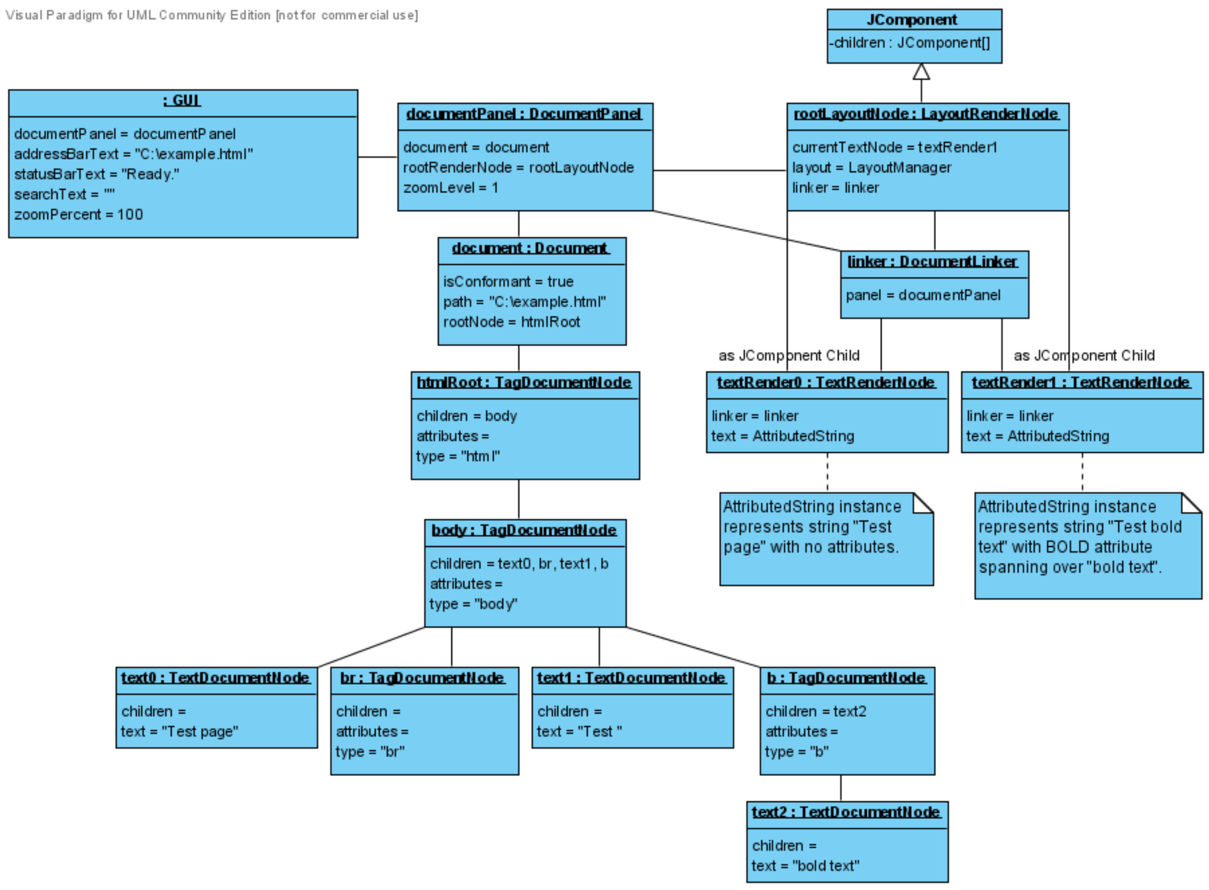


Figure 7: Renderer Completed Object Diagram

The DocumentPanel now has a root RenderNode which is a LayoutRenderNode containing the two text nodes that comprise the document. The first text node (representing the first line) has the text “Test page” and the second contains “Test bold text” with the BOLD attribute assigned to the range of characters “bold text”.

Rendering Process Detail

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 an 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

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 a class has and invoke the one you need by name and signature.

In Java

Java has a properties class that can be used to aid a system like this. 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, to map some relevant data (in this case the tag type string) to the name of a class to instantiate with reflection.

External Files

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

```
Code:
        Properties properties = new Properties();
        properties.load("<root>:\example.properties");
```

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 package of classes for handling reflection: java.lang.reflect.

Here is some code for a simple example of reflection. This serves to demonstrate how Java reflection code looks and could be used in our application.

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 the 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 properties file that will be loaded into a Map within the Renderer class during its constructor code – this will map the tag type strings (e.g. “b”, “center”, “html”) to TagRenderer plugin subclasses.

When rendering DocumentNodes into RenderNodes, the DocumentPanel calls renderRoot on the Renderer object, passing in the DocumentNode tree as the root node.

This in turn instantiates a root LayoutRenderNode to hold the document’s render nodes, and then calls render with default formatting.

The render method takes the given DocumentNode, if it is a TextDocumentNode, it handles the rendering itself. This involves adding the text string in the node to the current TextRenderNode for the parent render node it is in, with the current formatting. Alternatively, if the DocumentNode is a TagDocumentNode, the rendering responsibility is deferred to the appropriate implementation of TagRenderer, as specified by the properties file (via the Map).

TagRenderers are supplied with a reference to the Renderer itself, the TagDocumentNode, the parent RenderNode and the current formatting. In order to render a bold tag for example, the TagRenderer would defer the responsibility of rendering all of its children directly back to the Renderer, but with a modified formatting object which adds bold to the attributes. More complex tags like table will in fact instantiate their own subclasses of RenderNode (e.g. TableRenderNode) and add them to the parent render node as a component, and then defer the rendering back to Renderer, but with the new TableRenderNode as the parent node. In this way, tag rendering is recursive, and external plugin TagRenderers can still make use of the entire standard rendering code (and other plugins) for their child nodes. Our application need not know that TableRenderNode exists; it will handle its rendering itself as a Swing component, and will be instantiated by the potentially external TagRenderer as per the properties file.

Pseudo Code

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

During Renderer 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:

```
render() [called by renderRoot, and then recursively by TagRenderers]
      if documentNode is TagDocumentNode
            currentTagRenderer = rendererMap.get(documentNode tag type)
            currentTagRenderer.render(documentNode as TagDocumentNode, ...)
      else
            add text into parent’s current TextRenderNode with formatting
```

Rendering is a recursive process, TagRenderers should initiate (by calling back to the Renderer) the rendering of their children.

View Manual

To view the manual, the software makes use of the previous processes linked together. To illustrate how these are linked, here is a sequence diagram for this process:

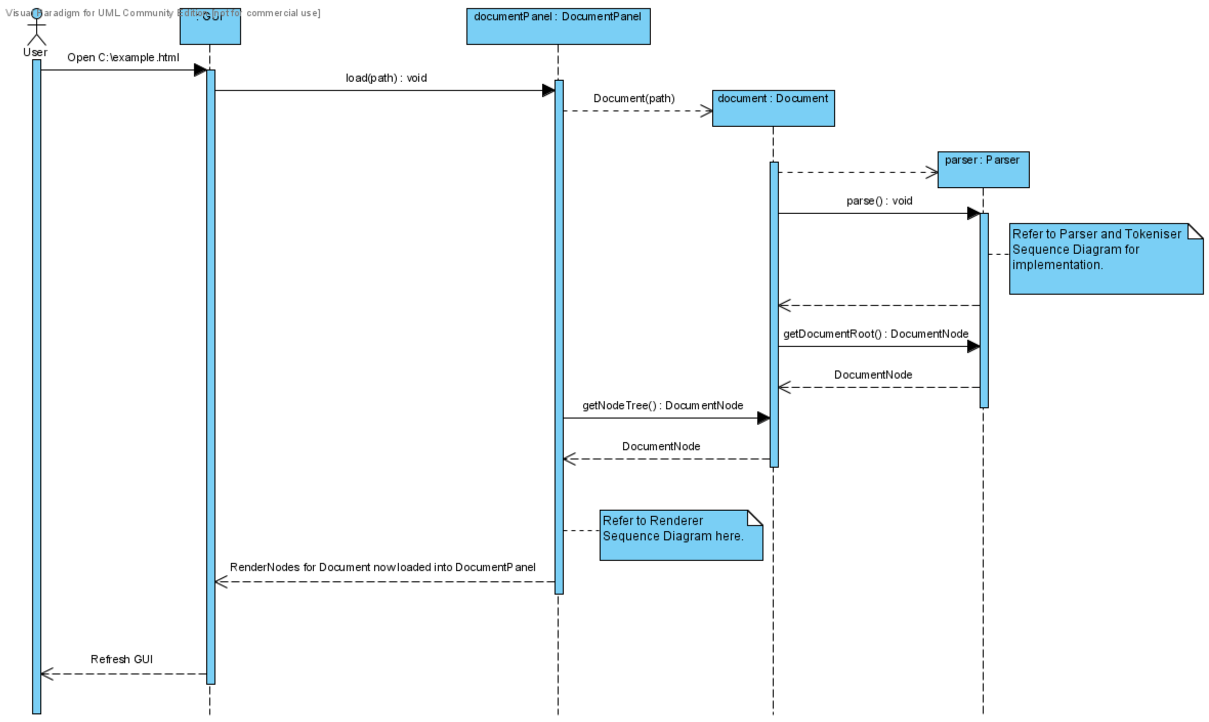

Visual Paradigm for UML Community Edition [not for commercial use]
User
: GUI
documentPanel : DocumentPanel
document : Document
parser : Parser
Open C:\example.html
load(path) : void
Document(path)
parse() : void
Refer to Parser and Tokeniser Sequence Diagram for implementation.
getDocumentRoot() : DocumentNode
DocumentNode
getNodeTree() : DocumentNode
DocumentNode
Refer to Renderer Sequence Diagram here.
RenderNodes for Document now loaded into DocumentPanel
Refresh GUI

Figure 8: View Manual Sequence Diagram

As can be seen from the notes, it makes use of the Parsing/Tokenising and Rendering processes.

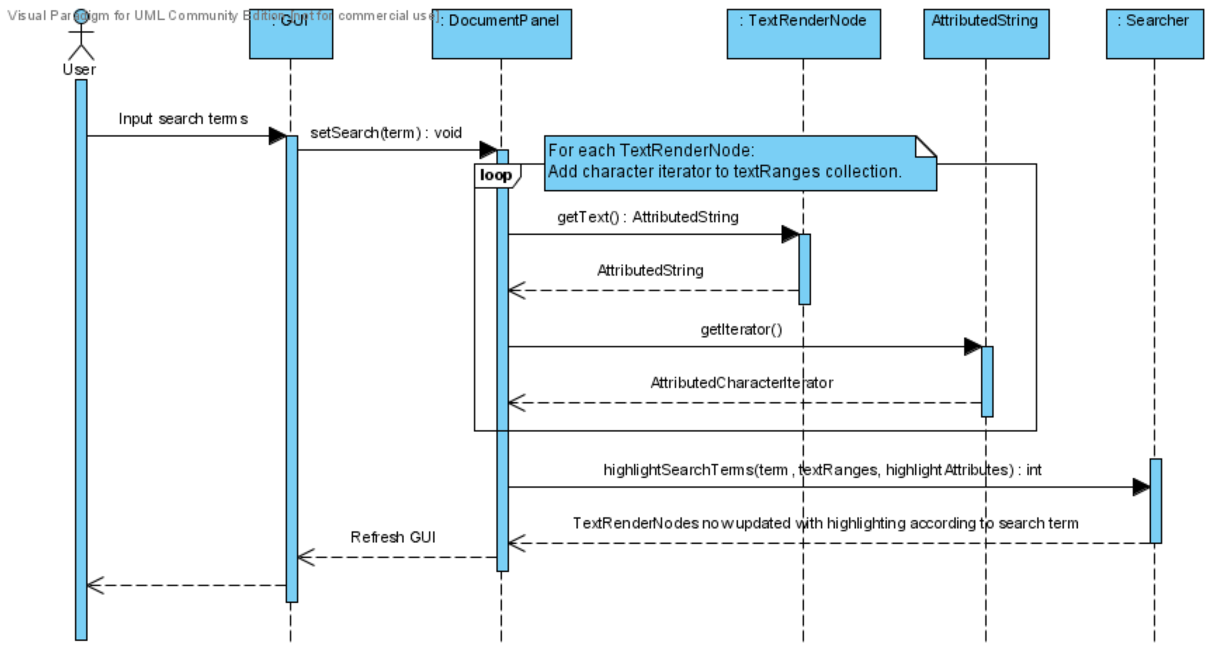
Logging

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.



Visual Paradigm for UML Community Edition [not for commercial use]
User
: GUI
: DocumentPanel
: TextRenderNode
AttributedString
: Searcher
Input search terms
setSearch(term) : void
loop
For each TextRenderNode:
Add character iterator to textRanges collection.
getText() : AttributedString
AttributedString
getIterator()
AttributedCharacterIterator
highlightSearchTerms(term , textRanges, highlightAttributes) : int
TextRenderNodes now updated with highlighting according to search term
Refresh GUI

Figure 9: Search Sequence Diagram

The process of searching loops through all the TextRenderNodes in the Document and adds their AttributedCharacterIterators to an array, textRanges. This array gets passed into the highlightSearchTerms static method, which tries to find instances of the search term (which it can find even if straddling multiple character iterators) and then uses the attributed iterator to add highlighting to the appropriate parts of the strings.

The Object Diagram representing the state of the system before a search can be seen by referring to the Renderer Completed Object Diagram above. It represents the state of all the components as a page is being viewed. The user would then opt to search via the GUI, and the following would be the resulting Object Diagram. This assumes the user has searched for “Test bold”:

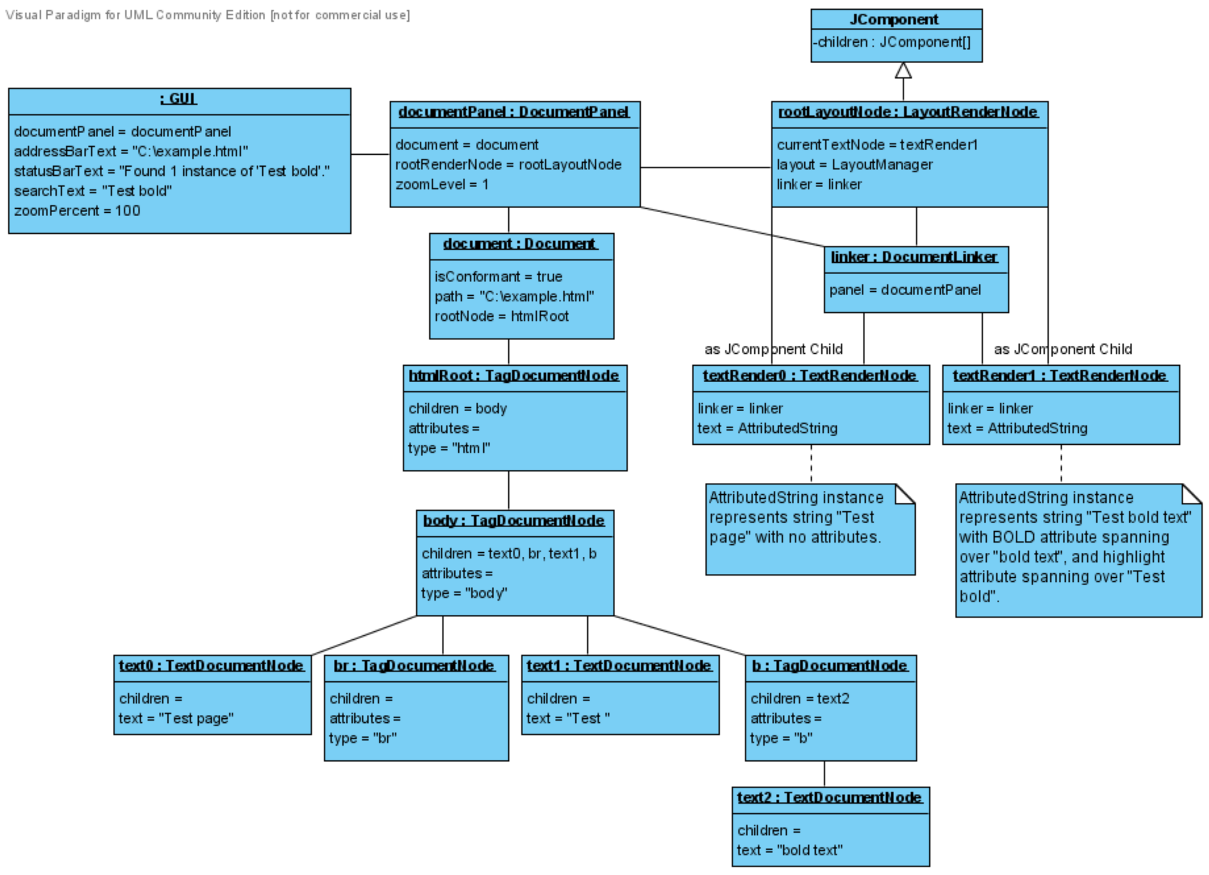
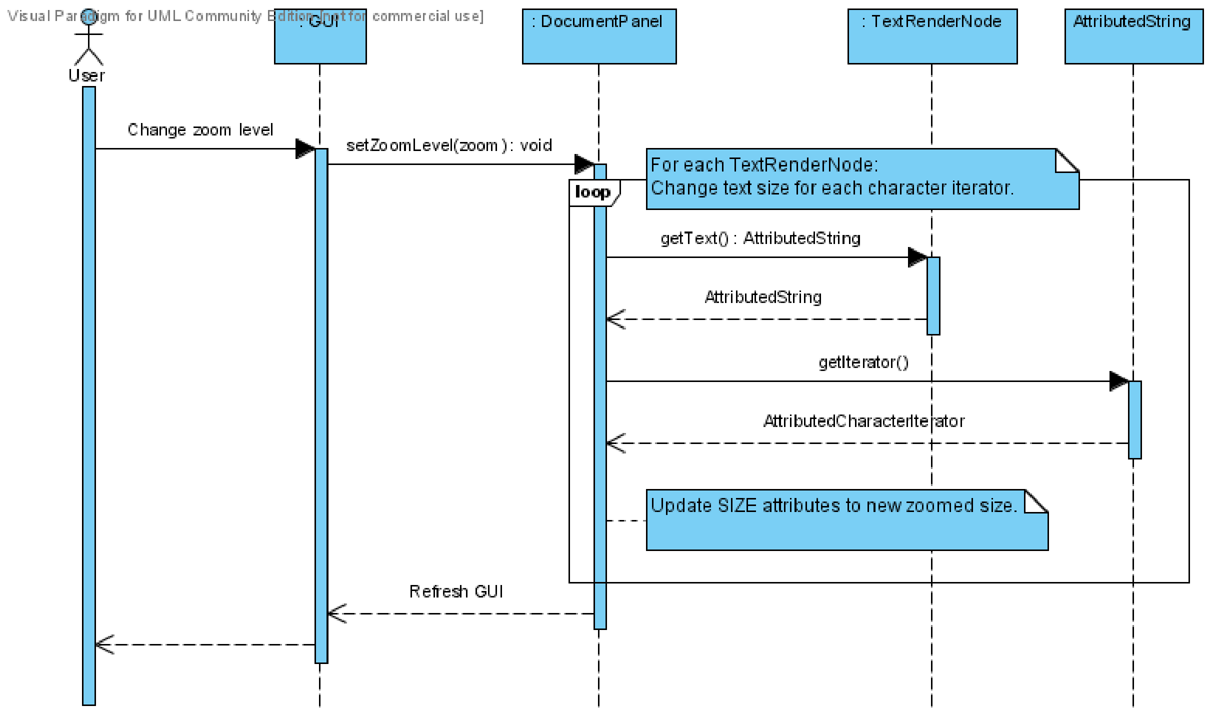


Figure 10: Search Completed Object Diagram

The second text render node, textRender1, has had its AttributedString updated to add a highlighted range over the characters “Test bold”, and the GUI has updated to reflect the search – the status bar and search text specifically.

Zoom



Visual Paradigm for UML Community Edition [not for commercial use]
User
: GUI
: DocumentPanel
: TextRenderNode
AttributedString
Change zoom level
setZoomLevel(zoom): void
loop
For each TextRenderNode:
Change text size for each character iterator.
getText() : AttributedString
AttributedString
getIterator()
AttributedCharacterIterator
Update SIZE attributes to new zoomed size.
Refresh GUI

Figure 11: Zoom Sequence Diagram

Similar to searching, the zoom loops through all TextRenderNodes and scales up the size attributes of the text according to the new zoom level.

The initial Object Diagram for this feature is also the Renderer Completed Object Diagram.

After zooming to a level of 125%, the Object Diagram is the following:

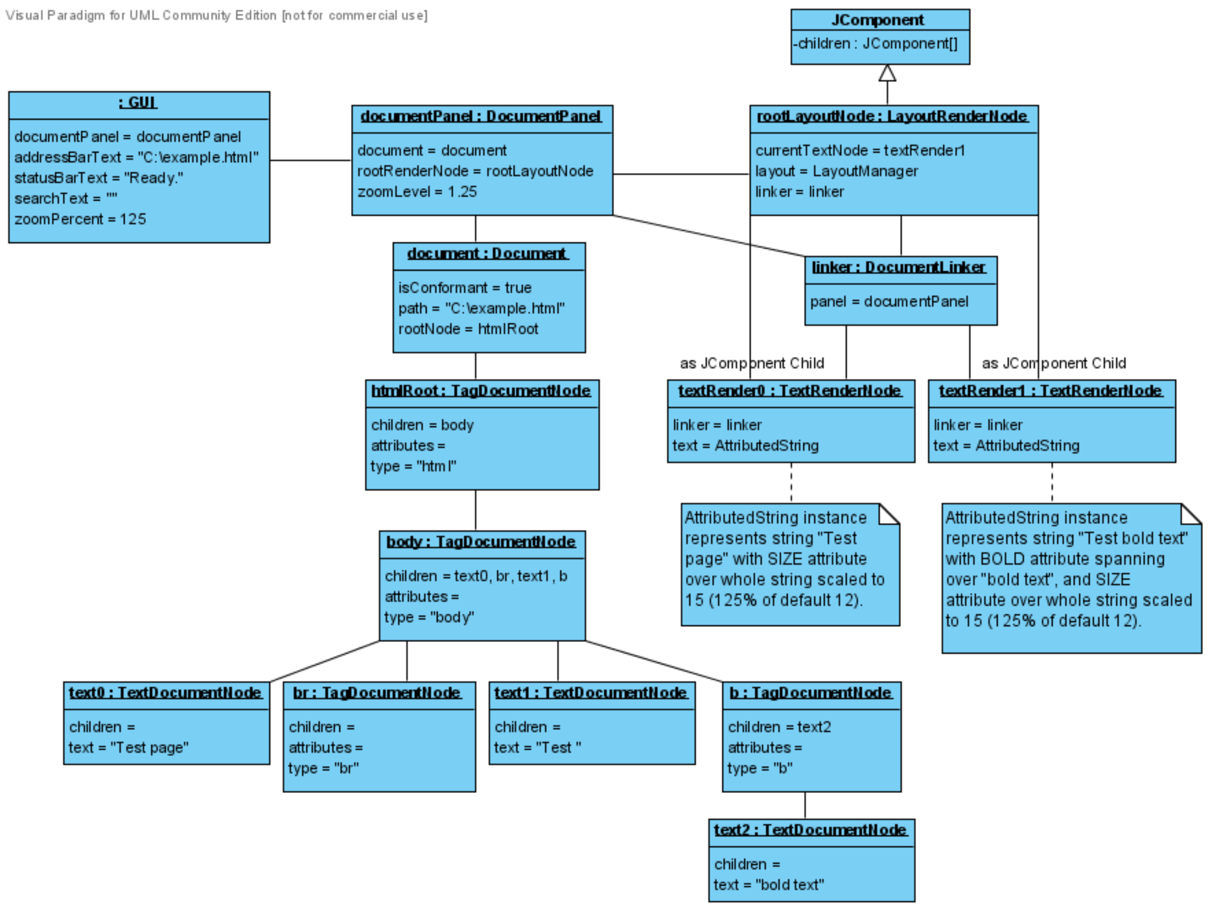


Figure 12: Zoom Completed Object Diagram

The AttributedStrings in the TextRenderNodes have been updated to reflect the new zoom level. The GUI holds this new zoom value as a percentage, and the DocumentPanel holds it as a floating point scale factor. The way our renderer works uses Swing resizable components, so resizing the text will cause the “preferred size” of that component to increase, and with that any components containing that (e.g. a table cell) will enlarge accordingly. In this way, it allows us to centre the zooming functionality on the text and allow the rest of the page to react correctly without having to write lots of explicit zooming code.

Follow Link

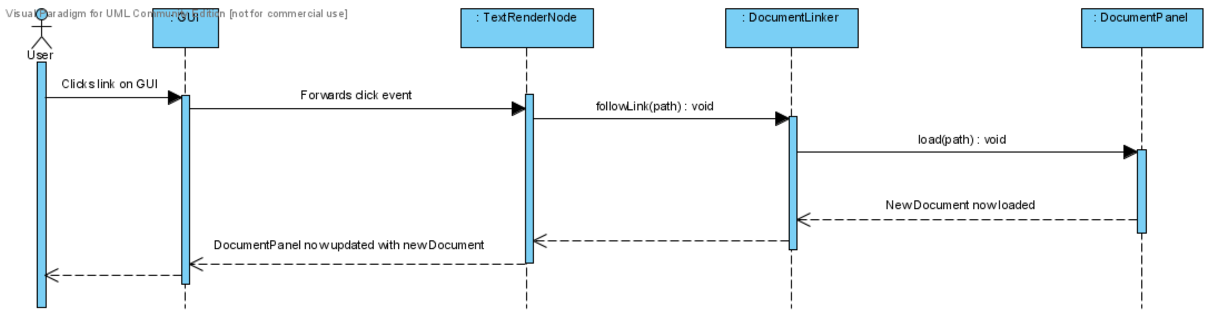


Figure 13: Follow Link Sequence Diagram

The details of the DocumentPanel.load(path) method can be seen in the View Manual Sequence Diagram (which in turn refers to the Renderer and Parser/Tokeniser diagrams).

The following is a cut down Object Diagram of a document called linkerDocument.html which contains a link to a document named destination.html. For brevity, the detailed document and render trees have been omitted, they would be very similar to those featured in the Renderer Completed Object Diagram.

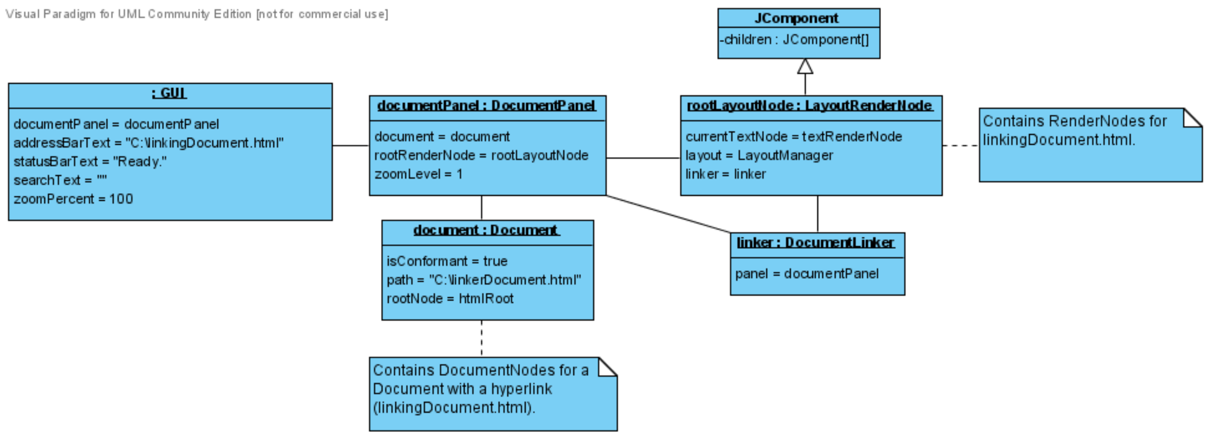

Visual Paradigm for UML Community Edition [not for commercial use]
JComponent
-children : JComponent[]
: GUI
documentPanel = documentPanel
addressBarText = "C:\linkingDocument.html"
statusBarText = "Ready."
searchText = ""
zoomPercent = 100
documentPanel : DocumentPanel
document = document
rootRenderNode = rootLayoutNode
zoomLevel = 1
rootLayoutNode : LayoutRenderNode
currentTextNode = textRenderNode
layout = LayoutManager
linker = linker
Contains RenderNodes for linkingDocument.html.
document : Document
isConformant = true
path = "C:\linkerDocument.html"
rootNode = htmlRoot
linker : DocumentLinker
panel = documentPanel
Contains DocumentNodes for a Document with a hyperlink (linkingDocument.html).

Figure 14: Follow Link Initial Object Diagram

After the link is clicked, the state of the system would be the following:

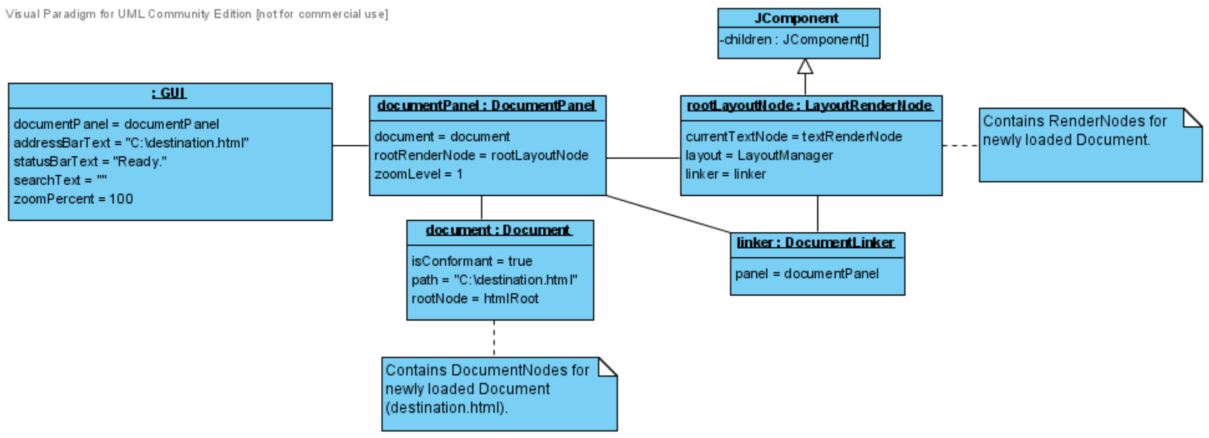

Visual Paradigm for UML Community Edition [not for commercial use]
JComponent
-children : JComponent[]
: GUI
documentPanel = documentPanel
addressBarText = "C:\destination.html"
statusBarText = "Ready."
searchText = ""
zoomPercent = 100
documentPanel : DocumentPanel
document = document
rootRenderNode = rootLayoutNode
zoomLevel = 1
rootLayoutNode : LayoutRenderNode
currentTextNode = textRenderNode
layout = LayoutManager
linker = linker
Contains RenderNodes for newly loaded Document.
document : Document
isConformant = true
path = "C:\destination.html"
rootNode = htmlRoot
linker : DocumentLinker
panel = documentPanel
Contains DocumentNodes for newly loaded Document (destination.html).

Figure 15: Follow Link Completed Object Diagram

The new document is now loaded as shown by the GUI with the updated address bar and Document with new path.