**BrowserMonkey**

****

**High Level Design Document**

v 1.0

**Software House One**

**Version History**

Document name: se-proj1/SVNRepository/Design Team/scratch/High Level Design.docx

Document version: 1.0

Document author: Daniel Cooper 2009/03/10

Document author: Merrigan Baylis 2009/03/10

Document author: Adam McGinness 2009/03/10

Document author: Paul Calcraft 2009/03/13

Document auditor: Daniel Cooper 2009/03/13

Document auditor: Paul Calcraft 2009/03/13

Document auditor: Sara Aspery 2009/03/13

Document auditor: Ioanna Kyprianou 2009/03/13

Last modification date: 2009/03/13

# High Level Design

Contents

[High Level Design 3](#_Toc224716861)

[Architectural design 4](#_Toc224716862)

[Requirement 1 - Basic Render 5](#_Toc224716863)

[Requirements 2 & 3 - Logging & Conformance 5](#_Toc224716864)

[Requirements 4 & 5 – Tags and Tables 6](#_Toc224716865)

[Requirement 6 - Links 7](#_Toc224716866)

[Requirement 7 - Search 7](#_Toc224716867)

[Requirement 8 - Zoom 8](#_Toc224716868)

[Common tactical policies 9](#_Toc224716869)

# Architectural design

## C:\Users\Daniel\Desktop\SVNRepository\Design Team\scratch\High Level\ClassModel.png

Figure 1: Basic Class Diagram for BrowserMonkey

Figure 1 shows the UML class diagram for the high level class diagram for the browser monkey browser.

* **GUI** – Displays GUI widgets and windows and handles the listeners on those events.
* **DocumentPanel** – An extension of the java2d paint pane (JPanel) that will act as the rendering surface for the browser.
* **DocumentPresenter**- A class that will create RenderNodes from the DocumentNode tree and subsequently traverse these to render the Document.
* **Searcher** – A class that will perform searches over nodes in the RenderNodes tree.
* **HotLinker** – Does the lookup between the pixel point at which a mouse button was clicked and which element was rendered in that space, allowing hyperlinks.
* **RenderNode** – A render node is a collapsed view of the tree, from the root to each leaf, representing a finite space and what is rendered in it at an atomic level – e.g. a piece of text with a certain formatting.
* **DocumentNode** – A representation, possibly corrected, of the HTML the browser has been supplied with.
* **Document** – One document represents one HTML file. Contains the Document Node tree as well as some other run time information.
* **Tokeniser** – Takes a single HTML stream and splits it into Tokens which can then be interpreted.
* **Parser** – Takes a set of sequential tokens and produces a DocumentNode tree.

## Requirement 1 - Basic Render

As outlined in the requirements, the first deliverable will be the basic render. This will have a complete GUI interface but will lack the ability to be able to render anything beyond simple text. HTML rendering, links, search and zoom will then be added onto this foundation to create a functional – and to specification – web browser.

In order to do any rendering two trees must be built.

1. **The Document Tree –** The document tree is very much like the Document Object Model found in modern browsers. It will store the browsers internal representation of the HTML, with any rendering mode fixes we choose to implement.
2. **The Render Tree –** The render tree is the resolved version of the document tree. Each leaf on the Document Tree tree ultimately leads to some content. The Render tree realises this by collapsing each path to each node in the document and creating a self contained object with information to allow this object to ‘paint’ itself to a JPanel surface.

The render works by first building a Document tree of the HTML file it is given, then creating a set of RenderNodes from that Document Tree and painting each one, in order, onto the paint surface.

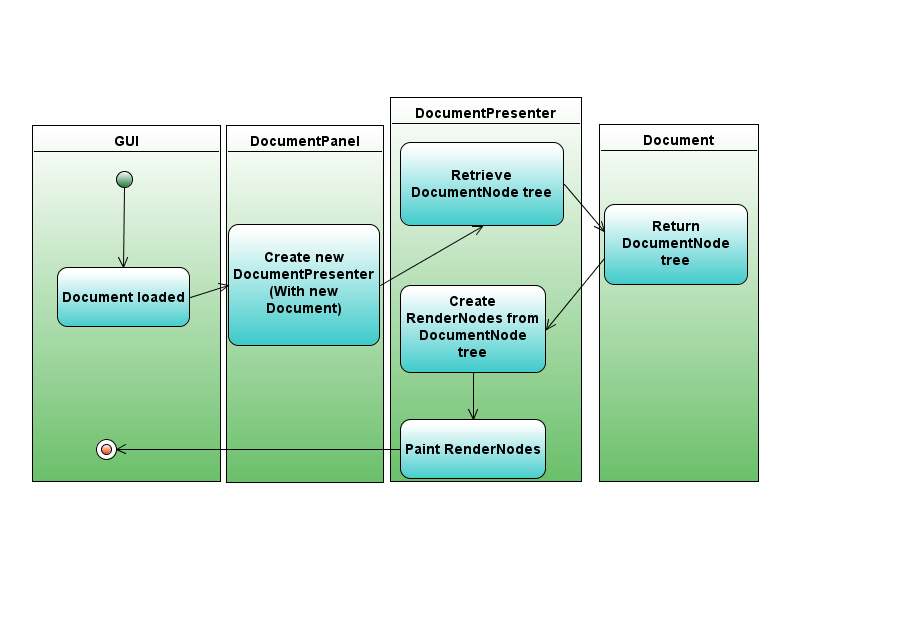


Figure 2: Activity Diagram Of Basic Rendering

## Requirements 2 & 3 - Logging & Conformance

After the basic render has reached a stable and working state, the next task is to add in the ability to render HTML. To do this there must be ‘conformance’ checks which test if the HTML supplied is valid, and if it’s not attempt some basic correction while producing the document tree.

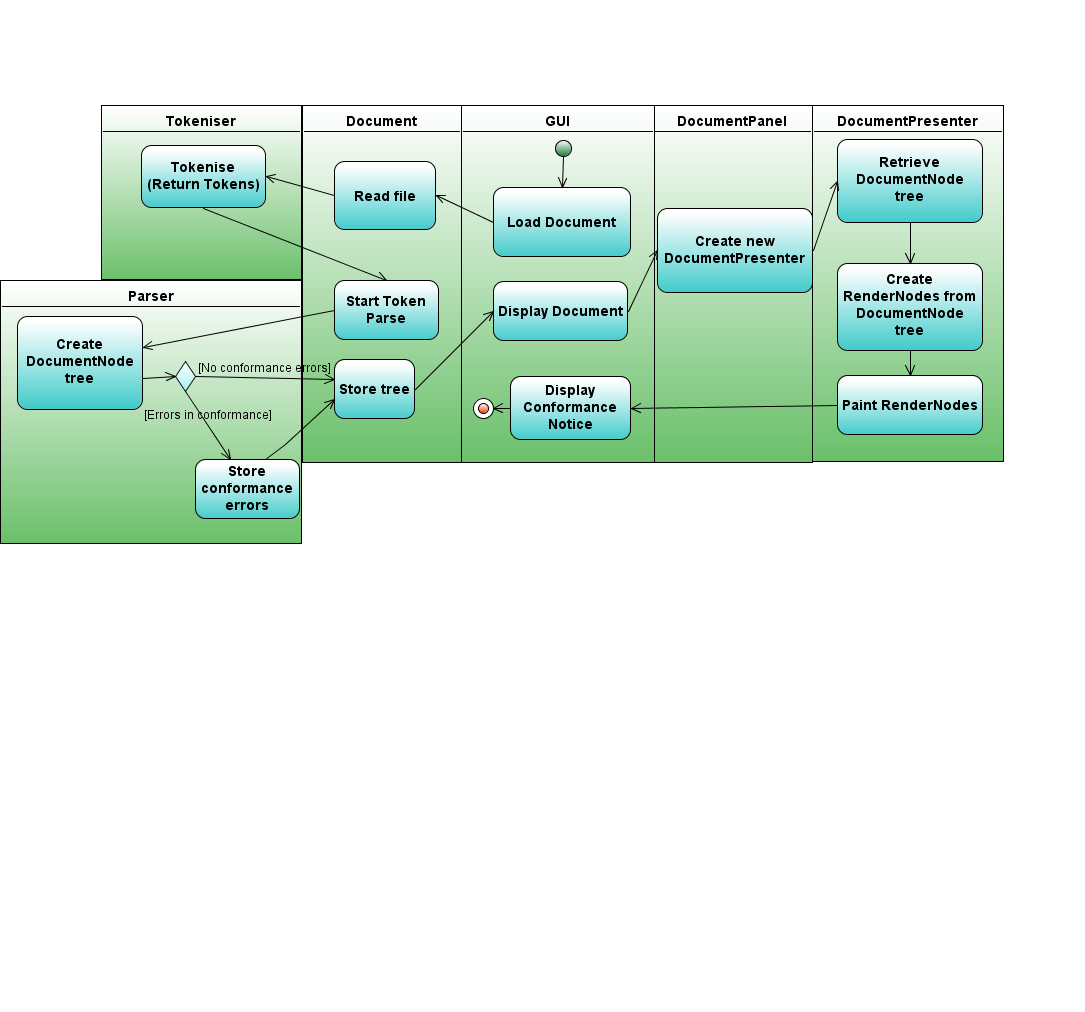


Figure 3: Activity Diagram of Conformance.

Figure 2 demonstrates in higher detail the full steps which will be taken in order to render a file, including the tokenising stage.

A very similar set of activities will be undertaken for logging (if the user starts the program in logging mode.)

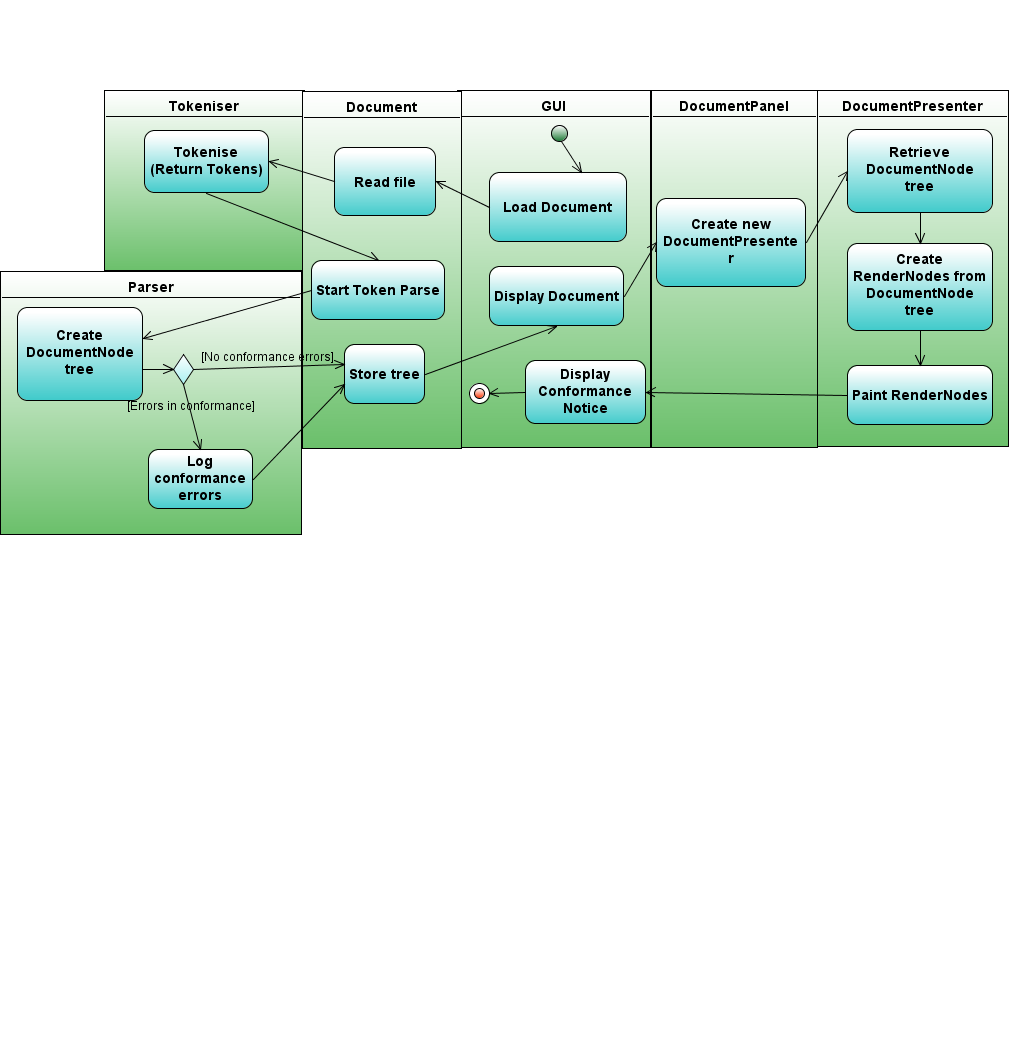


Figure 4: Activity Diagram for Logging

## Requirements 4 & 5 – Tags and Tables

The details of the implementation of these two requirements were deemed too low level to include in the high level design. Their implementations are encapsulated in the DocumentNode class and its associates as seen in this high level design. Explanations of their workings will be included in the detailed design specification.

## Requirement 6 - Links

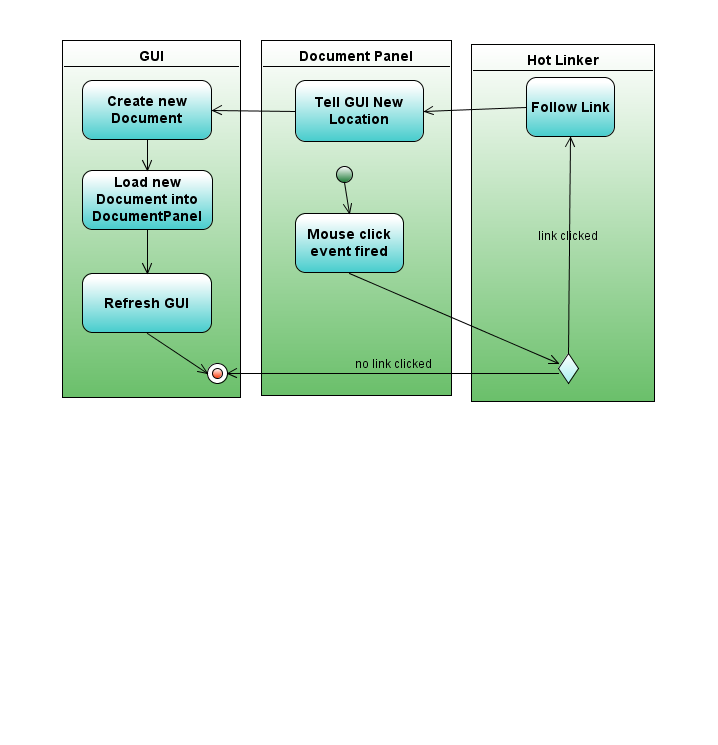


Figure 5: Activity Diagram for Links

One of the major disadvantages of using a 2d canvas to render on is that we have no easy way of telling what has been clicked. To get around this limitation, we employ a “hotlinker” class. This class takes the position that a mouse click event was registered at and looks up what element was rendered in that position. From there we can look up the element and then, if required, follow through on a hyperlink.

## Requirement 7 - Search

The Searcher Class provides much of the searching functionality. When a Search is entered; the DocumentPanel creates a new Searcher object. This Searcher object is then passed to the DocumentPresenter as an additional render node processor.

The Searcher keeps an internal buffer of matched search terms that it builds up – which gets emptied if it encounters a ‘breaking character’. A breaking character is a character that is not in the search term.

As the search builds this internal buffer, it also maintains a list of render nodes that contains segments of text that were inputted into the buffer. This list is also emptied when the buffer id emptied.

When the searcher has completed its iteration over the document nodes, it splits the nodes along the borders of the search term and makes the newly split nodes highlighted nodes. The document is then rendered.

Consider the two render nodes “foo” and “bar” adjacent to each other. If the user searches for “ob” then the following transformation would occur.

Bar

Foo

oB

ar

Fo

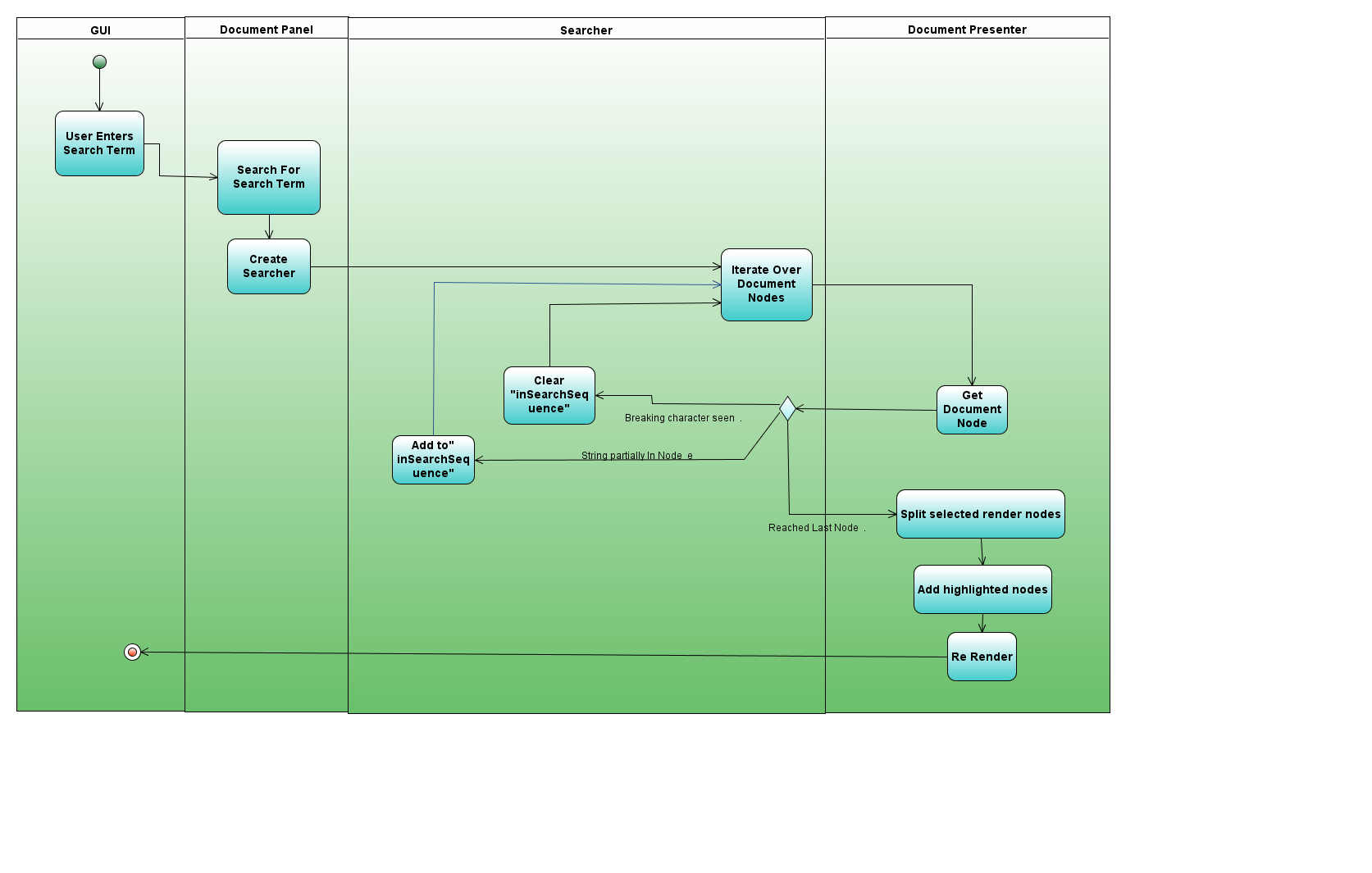
The Activity diagram for this scenario follows.

Figure 6: Activity Diagram of Search

When a the search field is cleared, the page is re-rendered as if there was no search term. That is to say – the RenderNodes are re created from the original DocumentNodes.

## Requirement 8 - Zoom

As previously discussed in the rendering section, the DocumentPresenter class renders to the document panel by iterating over a collection of render nodes and calling the paint method on each one.

When a page is first rendered it’s drawn with a zoom level of 1, and presented to the user. Via the GUI, the user will be presented with widgets which will allow the zooming to be changed. When the zoom value is changed the page is then re-rendered with the desired zoom level.

In essence, the zoom is always considered in the rendering of a page, but is set to 1 when a page is initially rendered.

A zoom value of 2 would imply a 200% zoom level and a zoom value of 0.5 would indicate a zoom level of 50%.

A change in the zoom level starts with the change in the GUI zoom level widget. When that widget is altered a zoom event will fire. The new zoom level will be compared with the current level and if they are different the browser will proceed to re render the document with the newly selected level.

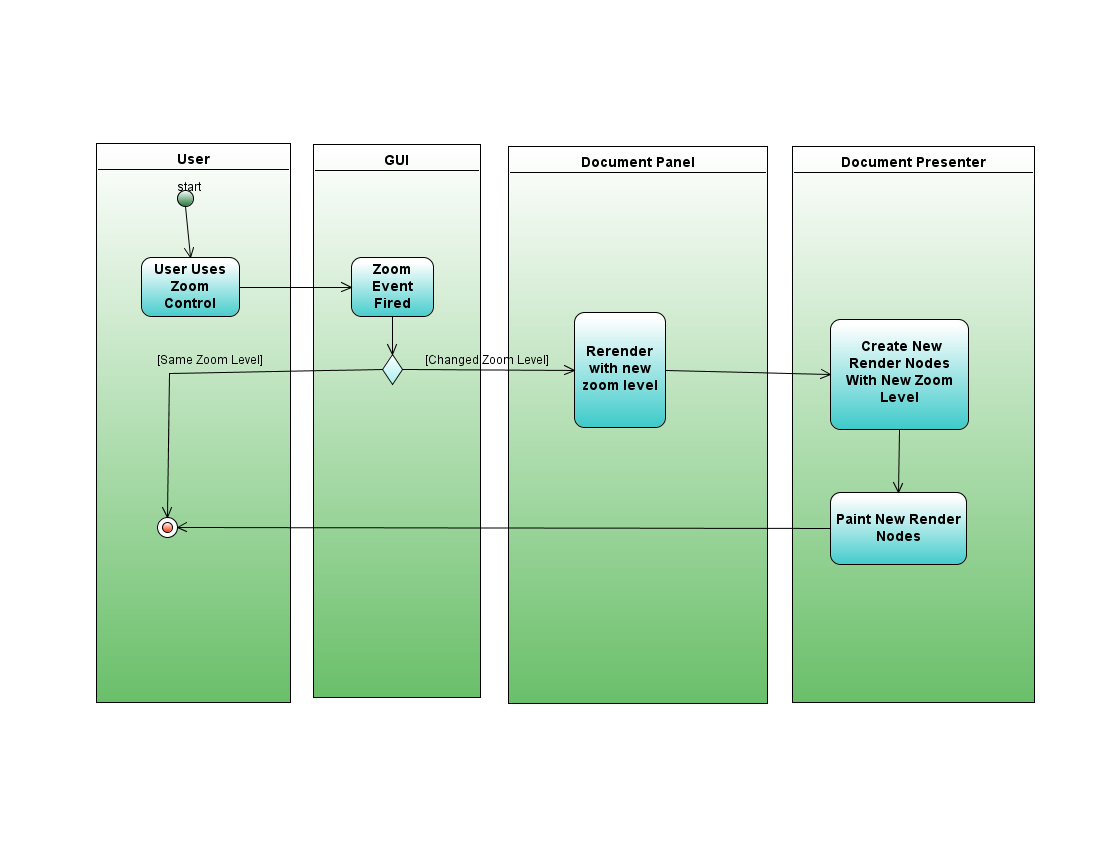


Figure 7: Activity Diagram of a Zoom Level Change

# Common tactical policies

The design tries to follow object orientation principles, attempting to minimise coupling and keep the responsibilities of each class distinct and manageable. The visitor pattern will be used to traverse trees and collections of data where possible to make the code more flexible. The strategy pattern will also be considered for the handling of the many different tag classes that there are to be rendered; avoiding less maintainable methods of handling this such as a growing if/else statement. In all places possible the project will try to remain extendable and updateable. The ability to perform maintenance on the code will hold great importance.

This browser will be implemented in java and will, where possible, follow established java conventions with regard to method, class and variable naming as well as code formatting.

SVN will be used for version control. This software house will follow a ‘commit early and often’ policy – thus ensuring each commit is a small evolutionary change, and not anything which would be troublesome to revert.