Taggle Technical Manual

Last updated: 17-Feb-11

Contents

Introduction 3

About Taggle 3

Purpose of this document 3

Installation and Usage 4

Source Code 4

Usage 4

Implementation 5

Source data 5

Cloud core 6

Data 6

Settings 7

Mapping 7

Layout 8

Dealing with large clouds 8

Typewriter Layout 9

Spiral Layout B 10

Spiral Aspect Ratio Layout 11

Force Directed Layout 11

Rendering 12

Actions & Events 13

Action Handlers 13

Events 13

Logging 14

Known Issues 15

Introduction

About Taggle

Taggle is a tag cloud generator written by Chris Deaker (cjd113@uclive.ac.nz), under the supervision of Dr. Neville Churcher, for the Software Engineering Visualisation Group (SEVG), from Department of Computer Science at the University of Canterbury. Its purpose is to provide a means of visualising complex multivariate data using tag clouds, with the specific aim of visualising software metric data.

Purpose of this document

This document is intended to provide a starting point for anyone wishing to further extend Taggle. This is also intended to provide a (mostly) up-to-date statement of the capabilities of the application, and assist anyone wishing to use this application, or develop their own similar tool.

At the time of writing the application remains in development. While the intent is for this document to remain as up-to-date as possible, there may potentially be significant differences between the latest version of the application and the descriptions here. It is advised to consider this document as a starting point only.

Installation and Usage

Source Code

The tag cloud project is located in the SEVG project repository on the Canterbury University COSC department network, at file:///home/cosc/research/svn/sevg/tagclouds

The project includes source code, documentation and UML diagrams, default configuration settings, and a number of example data sets.

Usage

The application has been developed to run under both Linux and Windows.

To run and debug using the Eclipse IDE, check the project out, and run as a Java Application, with main class `app.CloudApp`.

An Ant build file is also provided. After checking out the tag cloud project, in a Linux terminal, navigate to the base project directory and type:

```
ant clean compile jar run
```

To create Javadoc for the project use:

```
ant javadoc
```

Application logging is set to *INFO* by default, meaning that only errors and application level messages will be logged. See Logging for more details.

Implementation

A key aim during the course of development was to ensure that a generalized API would be produced, which could be utilized by future developers and researchers. A number of elements of the current implementation - while fully functional - serve as a 'proof of concept' rather than a part of a finalized product. The intention is that major facets of the software can be altered, removed or replaced, with minimal work.

The software itself can be separated out into a number of core logical areas.

Source data

Source data is provided in an XML format, and must be generated externally. The application parses this XML file to create a data set in memory, which is used during the rest of the tag cloud creation process.

Each XML file is divided into two key sections: metadata and value data. The first (‘metadata’) section contains definitions of known measurement and relationship types, while the second section contains actual tag data, as well as defining the concrete instances of relationships between tags.

Within a `variables` element, a user may define as many `variable` elements as needed. Each `variable` must specify a measurement scale type (nominal, ordinal, or ratio), a textual description of the variable (e.g. ‘Lines of code’), and a unique identifier (e.g. “LOC”). Variables with an ordinal measurement scale may also specify `ranks` for valid values, along with weighting values for each rank, used during processing and mapping. An example describing a ‘lines of code’ metric, which has a ratio measurement scale, is given:

```
<variable type="ratio" description="Lines of code" code="LOC" min="1" />
```

Tags themselves are defined by a `tag` element, within a single *tags* element, which represents all known tags. Each tag must contain at least one `measurement`. Each `measurement` references a single (previously defined) variable, using its unique identification code, and provides a value for the measurement. An example is given:

```
<tag id = "1">
      <measurement code = "TEXT">Foo</measurement>
      <measurement code = "LOC">355</measurement>
</tag>
```

In the above case, tag 1 has a text value of ‘Foo’ and a ‘lines of code’ value of 355. It is the responsibility of the user to ensure that values are appropriate for the variable type. While 355 is an integer value, all values for a ratio scale are parsed as double precision floating point numbers.

Relationships are represented similarly. Valid relationships are defined by a `relationshiptypes` element, such as the following:

```
<relationshiptype description="superclass" code="SUPER"/>
```

These types can then be used to link two previously defined tags together, using tag ID's. An example is given:

```
<relationship from="48" to="60" code="SUPER"></relationship>
```

In the above case tag 48 is related to tag 60. Since tag ID's are used for relationship definitions, they must be unique. It is also legal to define relationships across multiple data sets. For example, relationships can be defined from tag 17 to tag x, from two separate data sets. This can be useful for listeners which are capable of linking distinct clouds together, as there is no assumption that two clouds need share a data set (see Events).

External source files proved convenient for the purposes of development. While the current implementation reads source data from XML, the provided API enables future extensions to create data sets from any source.

Cloud core

The Cloud class is the central control-point of the application. It provides an interface into key areas, allowing for communication from the rendered cloud to the internal data and user-defined settings, and back, enabling rich interaction for the user.

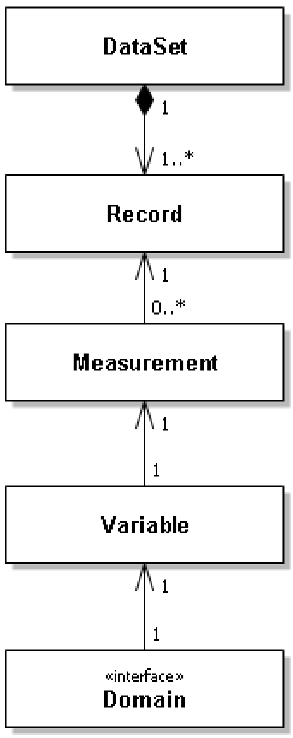


Figure 1

Data

The data package defines the internal representation of our data format. A `DataSet` contains a number of Records - the internal data representation of what is eventually displayed as a visualised tag. Each `Record` contains a number of `Measurements` for different `Variable` types. The legal values for each `Variable` are determined by its `Domain`, which can be nominal, ordinal, or ratio. `Relationships` can be used to bind two `Records` together, as well as defining a weighting for relationship strength.

A `RecordIterator` class is provided to support efficient sorting. A `RecordIterator` is instantiated with a list of records (usually all known records), a Variable used to determine order, and a boolean flag which can be set to true to indicate a reverse ordering. A vector of ordered indexes is created, such that for a record set r with n records, and an ordered index vector v, r(v(0)).m <= r(v(1)).m <= ... <= r(v(n - 1)).m, where m is the measurement value of that record for the variable type of the `RecordIterator`. After creation, the standard Java iterator methods `hasNext()` and `next()` can be used to traverse the list of records.

Record iterators also provide a means of calculating the 'rank value' of any record, as a decimal value between 0.0 and 1.0, inclusive, which is used later for mapping visual properties.

Settings

User-defined settings are saved directly into a properties file which is later read to determine the visual cloud characteristics for a particular data set. While settings are stored and read from a Java Properties XML file, actual access to settings is controlled by a wrapper class, `ConfigSettings`. Sensible default values are provided for properties requiring data. For example, font size minimum and maximum values are always required, whereas the variable mapped to tag alpha may be null. If a defaultsettings.xml file can be found during start-up, these properties are loaded and used as a starting point for user configuration. If no file can be found, this is created using default property values and saved.

Users may save and load settings configurations for use with other clouds later. While these settings can be applied to other data sets, the available variables in each data set must match.

Mapping

There are three implemented visual properties for each tag: font size, colour, and transparency, with possibilities for a number of additional properties to be added. During configuration, users may select which variable is to be mapped on to which property, as well as specify the lower and upper bounds for each property (for example, minimum and maximum font size).

Once the constraints of the visual property are set, a relative weighting for each record is calculated, according to the record‟s rank when all records are ordered by the selected variable. The method of calculation is dependent upon variable type. As ordinal domains have no intrinsic concept of "distance" between values, a "bucket ranking" value is calculated as follows:

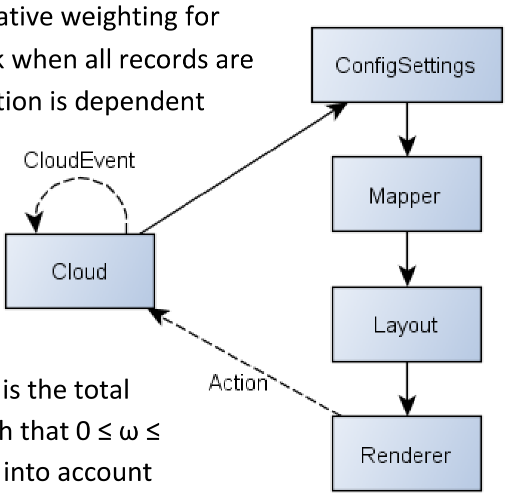


Figure 2 Cloud lifecycle

$\omega = \rho / \eta$

Where ρ is the records rank in the ordered record set, η is the total number of records, and ω is the resulting weighting, such that $0 \leq \omega \leq 1$. In ratio domains we can calculate a value which takes into account distance between values on a scale, using the following formula:

$\omega = (\alpha - \min\xi) / (\max\xi - \min\xi)$

Where α is the actual value of the records measurement for the specified variable, $\min\xi$ and $\max\xi$ are the lowest and highest values for all record measurements for the specified variable, and ω is the resulting weighting, such that $0 \leq \omega \leq 1$.

At this point, if specified by the user, ω is converted to its hyperbolic tangent. Discussion of the effect of the different mapping scaling options available can be found in Section 4.2. Using this ω, we can calculate the actual value of the visual property. An example using font size is shown:

$\beta = (\text{int}) (((\max\theta - \min\theta) \times \omega) + \min\theta + 0.5)$

Where ω is the previously calculated record weighting, minθ and maxθ are the specified minimum and maximum font sizes, and β is the tag‟s final font size. As shown, values are cast and rounded to the nearest integer value. In the case of font colour, values are calculated according to the above algorithm for each of the hue, saturation, and brightness values.

Layout

Once a Cloud has been created, its words are positioned according to the selected algorithm. `Layout` is an abstract class which provides utility methods for algorithms, such as for intersection and boundary testing, and defines the abstract method `placeWords`, which takes a list of non-positioned Words, ordered according to user configuration, and returns a list containing the same words, but with specified positions. All layout algorithms must subclass `Layout`. This is the extension point for any future alternate layout methods.

In addition to laying out an entire cloud, `Layout` provides a means for step-by-step layouts. `Layout.resetSteps()` will cause a layout algorithm to return a single positioned word with the next call to `placeWords()`. Each subsequent call to `placeWords()` will return previously placed words plus one additional word, until eventually a complete cloud is returned. `Layout.prevLayoutStep()` will return previously placed words minus the most recently placed. Effectively this allows for a cloud layout to be visualised step-by-step.

Three example algorithms have been provided. Each layout describes a tags position as an x, y co-ordinate on a 2-dimensional canvas, where x determines horizontal position, y determines vertical position, and an x,y co-ordinate of 0,0 describes the upper-left-most corner of the canvas.

Dealing with large clouds

Regardless of the layout algorithm used, the possibility always exists that it will be impractical or impossible to place all tags on a given canvas, with their specified visual properties. This may be caused by any combination of font and canvas size, font face, and selected layout algorithm. In these cases a strategy must be devised that will ensure the cloud continues to communicate individual tag information, as well as context.

We consider a 'problem tag' to be any tag which is preventing the layout algorithm from completing successfully. Realistically the most problematic tags will almost exclusively be those which are larger, as placing these may prevent several other tags from being able to fit within the available canvas constraints. Practically, however, 'problem tags' tend to be those which are positioned later, as we are often not aware that there will be difficulties positioning all tags until many have already been placed.

A number of strategies were considered for dealing with clouds which could not be practically displayed:

- Removal: Excluding any problem tag from a cloud.
- Truncation: Shortening the text of all tags by some uniform amount, thereby reducing the amount of required space for all long tags. For example, we might truncate all tags to a certain character length, determined by a pre-defined value, or some other calculation, such as the mean length of known tags.
- Resizing: Reducing the font size of all tags by some uniform amount, thereby reducing the amount of required space for all tags. If font sizes could be reduced no further without falling below the minimum defined value, tags text can be replaced with a small icon. In this way tags are still present in the cloud, albeit with drastically reduced visibility.
- Allowing for overflow: Adopting no tag resizing strategy, and instead allowing for tags to be placed beyond the absolute boundaries of the canvas. The user can utilize zooming and canvas dragging (see Action Handlers) to see view those tags which have been placed outside of the visible canvas boundaries.

Typewriter Layout

The "typewriter layout", as we have termed it, is a simple layout algorithm that lays out a cloud from left to right, top to bottom. The process can be described as follows:

1. Place the first tag at the top left corner of the canvas.
2. For each remaining tag, continue to place the tag on the canvas with an x value equal to the cumulative value of the widths of tags already placed on this line, plus a pre-defined spacing value. If a tag will extend past the width of the canvas:
3. Set the y value of all tags on the previous line to the cumulative y value of all prior lines + the height of the largest tag in the line. Go to Step 1.

Spiral Layout B

Spiral Layout B was the second 'spiral layout' algorithm developed. The intention is to cluster words as tightly as possible to the centre of the canvas. An example follows:

1. The first tag 'ArgListVisitor' is placed in the centre of the canvas. The second tag 'ArrayType' is placed below and to the right.

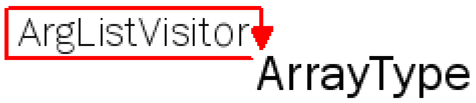


Figure 3

2. We attempt to place the third tag 'Block' to the bottom right of the anchoring first tag. As this would intersect 'ArrayType', 'Block' is gradually moved leftward until it can be placed without intersection.

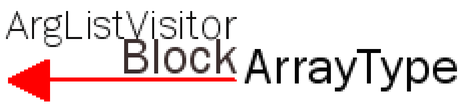


Figure 4

3. We continue the process by placing tags clockwise around 'ArgListVisitor'. We attempt to place 'Decl' to the right of 'ArgListVisitor', however it will intersect previously placed tags above and below. As we have completely exhausted available space around the first tag, we place 'Decl' below and to the right of the second tag, 'ArrayType'.

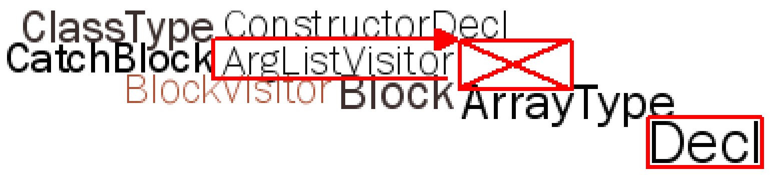


Figure 5

4. The next tag again attempts to place itself around 'ArgListVisitor', and is successful in finding a position to the right.

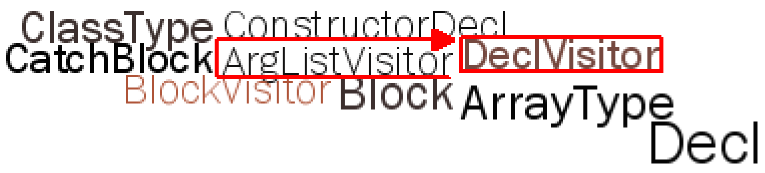


Figure 6

5. The process continues. As available locations around each tag are exhausted, the 'anchor' becomes the next tag placed in the cloud.

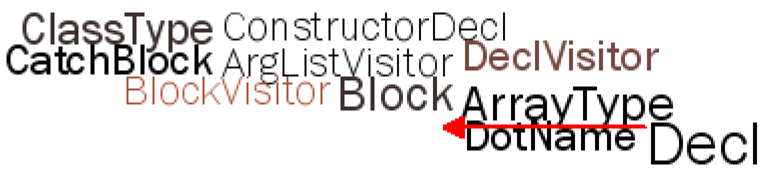


Figure 7

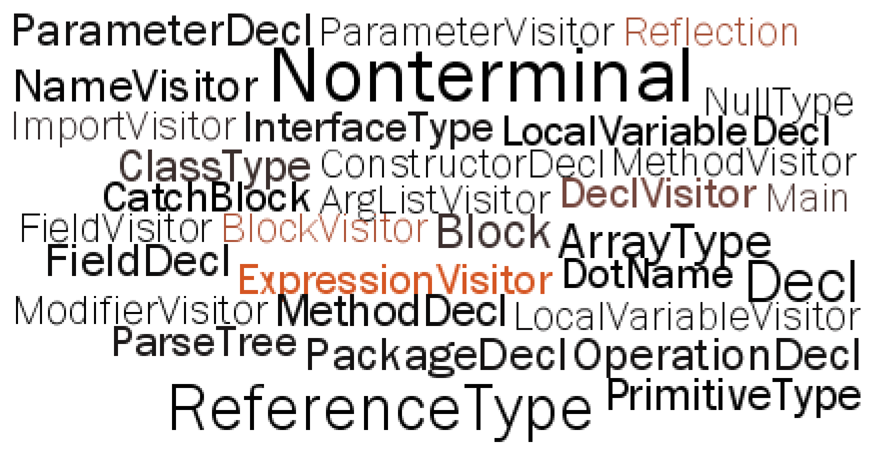

ParameterDecl ParameterVisitor Reflection
NameVisitor Nonterminal NullType
ImportVisitor InterfaceType LocalVariableDecl
ClassType ConstructorDecl MethodVisitor
CatchBlock ArgListVisitor DeclVisitor Main
FieldVisitor BlockVisitor Block ArrayType
FieldDecl ExpressionVisitor DotName Decl
ModifierVisitor MethodDecl LocalVariableVisitor
ParseTree PackageDecl OperationDecl
ReferenceType PrimitiveType

Figure 8 The final cloud.

Spiral Aspect Ratio Layout

The Spiral Aspect Ratio Layout behaves similarly to Spiral Layout B with a single significant alteration. The default response to overly large clouds (where not all tags could fit on the specified canvas) is to reduce the size of every tag in the cloud, and begin the layout process again (see Dealing with large clouds). Spiral Layout B triggers this resizing process whenever a single word cannot be placed within the canvas. This can be determined in one of three ways:

a) The total combined area of all tags exceeds the total available area of the canvas, in which case the tags are resized before any placement is attempted.
b) The width or height of a single tag exceeds the width or height of the canvas, in which case the tags are, again, resized before any placement is attempted.
c) During the layout process it becomes obvious that a tag cannot be placed without either intersecting a previously placed tag, or being placed outside of the bounds of the canvas.

The Spiral Aspect Ratio Layout responds to this issue differently. Rather than attempting to fit all tags within a specified canvas, it fits all tags onto the smallest canvas capable of containing all tags, with the same aspect ratio as the original canvas. Checks (a) and (b) above are ignored, as the actual dimensions of the canvas are irrelevant. The third check is only used to determine that the canvas size should be increased. Canvas size begins at 1% of the original provided dimensions, and increases by 10% increments, retaining the same aspect ratio.

The resulting cloud will often exceed the boundaries of the canvas display window. However, all tags will be displayed at the originally 'intended' (according to mappings) size. Zoom can be used to scale the cloud on the final rendered canvas, to a point at which the cloud fits within the bounds of the display window with minimal wasted space.

Force Directed Layout

The force directed layout is a layout implementation which utilizes relationship data. The purpose is to position tags on the canvas in such a way that related tags are in close physical proximity to one another, while unrelated tags maintain a regular distance.

This is essentially an implementation of a spring embedded layout. The method applied uses a number of distinct forces against each tag, added together to create a net force, which ultimately determines the amount of movement of a tag in one layout step. The factors in play are:

- Repulsion: A repulsive force pushes every tag on the canvas away from every other tag on the canvas. Using Coulomb's Law (http://hyperphysics.phy-astr.gsu.edu/hbase/electric/elefor.html#c1), the repulsive force is stronger when tags are closer together, and weakens as tags draw apart.

- Attraction: An attractive force pulls each tag towards every other tag which is *directly related*. A modified version of Hooke's Law (http://ocw.mit.edu/courses/physics/8-01-physics-i-classical-mechanics-fall-1999/video-lectures/lecture-10/) is used to enact a stronger attraction on tags which are further separated, and weaker on those which are closer together. There is no attractive force between unrelated tags.

- Centering: A centering force is used to cause all tags in the cloud – regardless of relationships – to gravitate towards the centre of the canvas.

- Velocity: This is essentially a multiplier which determines the final velocity of the compound force on any tag. Higher velocity causes tags to move further more rapidly.

Rendering

The `CloudCanvas` class provides an end point for the actual visualisation of a provided Cloud. `CloudCanvas` subclasses the popular Java Abstract Window Toolkit's Canvas, meaning that this can be added to an existing AWT or Java Swing component in a custom implementation of this application. Alternate renderers can be implemented and selected through settings configuration.

`CloudCanvas` renders each tag as a GlyphVector using Graphics2D. Word positions are pre-determined at render time - no checking is done to verify that tags are not overlapping, or will be drawn outside of the bounds of the canvas. `CloudCanvas` also provides a means for a number of in-canvas interactions such as zooming/scaling, and dragging. While these interactions are actually invoked by separate action handlers (see Action Handlers), `CloudCanvas` listens for mouse events and notifies the bound handler, providing it with data such as the current location of the mouse on screen, and the currently selected tag.

The current implementation uses a Java Swing JFrame (`CloudWindow`) to display the canvas, as well as providing configuration options through dropdown menus. `CloudWindow` also provides legends for variable-to-property mappings, including a colour palette legend displaying the available colour range and the positioning of measurement values at increments on that range. This frame is not strictly necessary, though any alternate implementation not using `CloudWindow` would most likely need to provide an alternate means of accessing some of the provided interactive functionality, such as the ability to draw new or duplicated clouds, attach listeners, or alter cloud settings.

Actions & Events

Action Handlers

The action package provides handlers for canvas interactions. This allows for simple actions such as clicking, scrolling or dragging the canvas with the mouse to invoke feedback from the cloud. In order to be used, action handlers are bound to user interactions, such as mouse clicks or key presses. These bindings are defined in the properties file.

When the canvas is created, it queries `ConfigSettings` for the action handler bound to each user action (eg. Left mouse button click). `ConfigSettings` checks for an entry in the properties file, and if a match is found, uses `ActionHandlerFactory` to create and return an instance of the bound handler. When the user interacts with the canvas, the handler for that interaction is notified, and reacts accordingly.

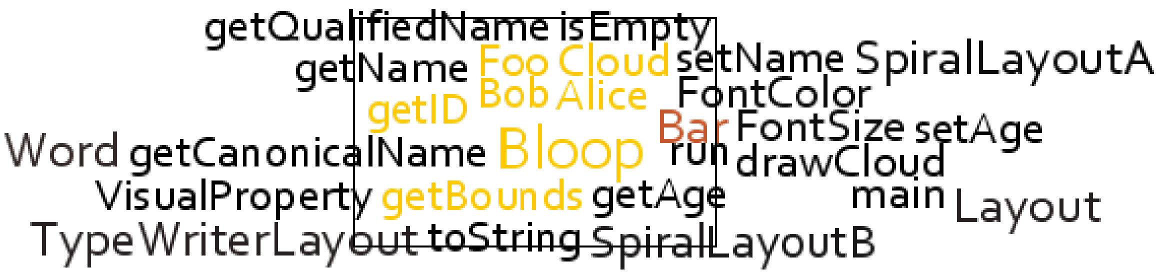

getQualifiedName isEmpty
getName Foo Cloud setName SpiralLayoutA
getID Bob Alice FontColor
Word getCanonicalName Bloop Bar run FontSize setAge drawCloud
VisualProperty getBounds getAge main Layout
TypeWriterLayout toString SpiralLayoutB

Figure 9 Selecting tags within a cloud. When the selection is complete, the appropriate handler is notified.

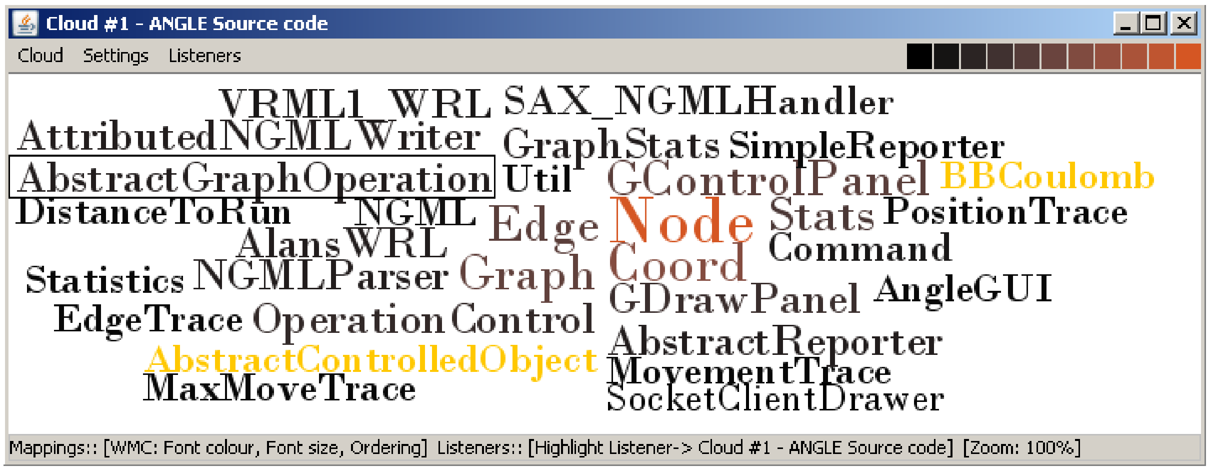
A number of handlers have been provided, which enable interactions including: canvas magnification and zooming, popup source file previewing, listing of record details, tag removal, and controlling layout steps. Alternate handlers can be added, and bound to actions by swapping handlers in the properties file, or adding new listeners to `CloudCanvas`, and binding those in the properties file.

Events

A simple event framework is provided to support interaction between multiple distinct clouds. Listeners can be added using the `CloudWindow` menu, linking one cloud to another. An arbitrary number of links can be made between any combinations of clouds. When an action mapped to a particular event type is performed, all listeners are notified. Each listener responds accordingly by altering its target cloud, if appropriate.

Currently a single listener, `HighlightListener`, is implemented, which uses colour to highlight related tags in any known cloud.

This framework was designed with extensibility in mind - alternate listener implementations can easily be added for a wider array of interactions.



Mappings:: [WMC: Font colour, Font size, Ordering] Listeners:: [Highlight Listener-> Cloud #1 - ANGLE Source code] [Zoom: 100%]

Figure 10 Using a highlight listener to highlight related classes

Logging

A simple logging framework has been implemented for informational and debugging purposes. `CloudLog` provides a number of static methods supporting logging at appropriate levels. The logging output level is set at application launch, depending on provided program arguments of a `java.util.logging.Level` description. Following the standard Java logging model, messages can be sent at any level, but those below the logger's level will be discarded.

The default logging level (set at application launch if no arguments are provided) is INFO – which will log application level messages and above (including errors). To enable more specific logging, program arguments can be provided as follows:

1. INFO – Application level messages, as well as 'severe' error messages. This is the default logging level.
2. FINE – Detailed application process messages, and above.
3. FINEST – Highly detailed information, such as co-ordinate and measurement data, and above.

Messages are logged to .log files in the /taglogs folder, and are named by time stamps.

Known Issues

- In some environments tags with a font size greater than 100 fail to be rendered. At this point this has only been seen while using Fedora 12 with Kernel version 2.6.31.12-174.2.22.fc12.x86_64.
- The current renderer implementation (`CloudWindow`) is hardcoded into `ConfigSettings`.
- Performance of spiral layouts suffers greatly for large sets. Performance for each layout attempt is approximately $O(n^2)$. Most intensive processing is involved in tag intersection checking. A layout implementation utilizing R-tree's for intersection testing might be beneficial for large sets.