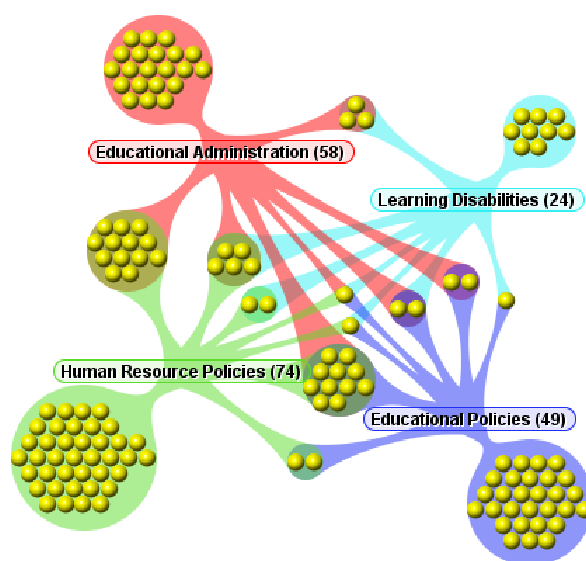


# Aduna Cluster Map Library version 2006.1

## Integration Guide



Aduna  
Prinses Julianaplein 14-b  
3817 CS Amersfoort  
The Netherlands

+31 33 465 9987 office

support@aduna-software.com  
<http://www.aduna-software.com>

- 1997-2006 Aduna BV. All rights reserved.

#### Copyright Information

A copyright is a property right in an original work of authorship. Copyright is recognized in most countries of the world by statutory copyright laws. Copyright exists in the expression of an idea, but not the idea itself. Copyrightable expressions can take many forms but are usually categorized as literary, musical, dramatic, pantomime and choreography, pictorial, graphic, sculptural, computer programs, motion pictures, and sound recordings. To obtain permission to use Aduna copyrighted materials, please refer to Rights and Permissions below.

#### Rights and Permissions

If you have any questions concerning the usage or licensing of Aduna copyrighted materials, for example, photographs, video footage, Aduna advertisements or other Aduna materials, please submit your detailed request in writing. Please be sure to include any surrounding copy or text to the Aduna material. Requests may be emailed to [copyright@aduna.biz](mailto:copyright@aduna.biz) or mailed to the appropriate addresses listed on Aduna's contact page and below.

Aduna  
Prinses Julianaplein 14-b  
3817 CS Amersfoort  
The Netherlands  
+31 33 465 9987  
[aduna.biz](mailto:aduna.biz)

Aduna, AutoFocus, Guided Exploration, Spectacle, Cluster Map are trademarks of Aduna BV registered in the Netherlands.

Adobe and Acrobat are trademarks or registered trademarks of Adobe Systems Incorporated in the U.S. and/or other countries.

Microsoft, MS Windows, MS Word, MS PowerPoint, MS Excel are trademarks or registered trademarks of Microsoft Corporation in the US and/or other countries.

Other company and product names mentioned herein are trademarks of their respective companies. Mention of third-party products is for informational purposes only and constitutes neither an endorsement nor a recommendation. Aduna assumes no responsibility with regard to the performance or use of these products.

# Table of Contents

1.	Introduction .....	4
2.	The Cluster Map Viewer .....	5
3.	Data File Format.....	6
	Global Structure .....	6
	Defining the Information Objects.....	6
	Defining the Taxonomy .....	7
4.	Public API's .....	8
	Package Overview .....	8
	Creating Data Models.....	9
	Using a ClusterMap Instance .....	10
	Creating a Graphical User Interface.....	11
	Thread Safety .....	13
	Jar Files.....	13
5.	Configuration.....	14
	Configuring a ClusterMap .....	14
	Configuring a ClusterMapMediator .....	18
	Configuring HTML output .....	20
	Obtaining Actions.....	21

# 1. Introduction

Welcome to the Aduna Cluster Map Library Integration Guide.

The Cluster Map library contains functionality for creating visualizations of collections of hierarchically classified objects, sometimes referred to as instantiated taxonomies or concept hierarchies.








The core of this library is formed by the set of the classes used for creating the visualization. Additionally, classes are provided for the construction of a Java Swing-based user interface, which makes it possible to create a tailor-made, interactive visualization.

Furthermore, the library contains the Cluster Map Viewer, a simple desktop application made from these components that demonstrates the primary functionality of the visualization and its GUI and that can be used for quick evaluation of the suitability of the Cluster Map visualization for a specific purpose and task through its use of a straightforward, XML-based data file format.

This integration guide consists of the following three parts:

1. A short introduction of the basics of the Cluster Map Viewer.
2. A definition of the format of the data files used by the Cluster Map.
3. An overview of and introduction to the usage of the Cluster Map's public APIs.

The software package you have received should have the following contents:

 clustermmap-2006.1	
 bin	(Launch scripts for various platforms)
 doc	(Documentation)
 api	(Javadoc documentation)
 examples	(Example data files)
 lib	(Program libraries)
 license	(Licenses for third-party code libraries)

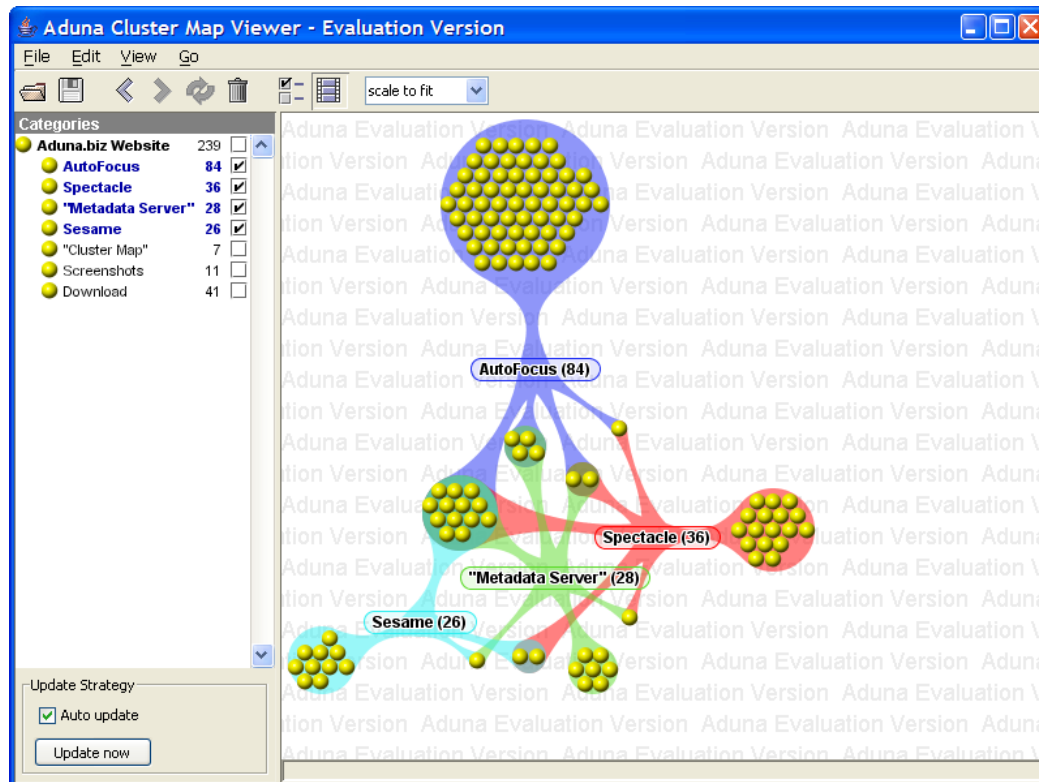
In order to start the Cluster Map Viewer, run (on Windows) the ClusterMapView.bat batch file or (on Unix/Linux) the ClusterMapView.sh shell script, which can be found inside the bin directory.

An example data file can be loaded and viewed by clicking on the *Open* button in the toolbar or by selecting "Open..." from the File menu.

The Cluster Map library requires Java 1.4.x to be installed.

## 2. The Cluster Map Viewer

To Do: an end user-oriented description of the interactive viewer.



## 3. Data File Format

The Cluster Map Viewer operates on XML-structured data files, describing the visualized objects, the class tree and the sets of objects contained by the classes. This section describes the format of these files. Note that the Cluster Map software package contains several example XML files.

The set of the attributes used to describe the objects and the classes is only dictated by the file format used by the Cluster Map Viewer. When integrated using the API, integrators are completely free to define which attributes an object or class needs to have.

### Global Structure

An XML data file is rooted by a Classification Tree element containing the following two information types:

1. The ObjectSet: a part describing all information objects (e.g. documents) displayed in the visualization.
2. The ClassificationSet: a part describing the class hierarchy or taxonomy.

The following partial XML document shows the outline of the XML structure:

```
<?xml version="1.0"?>
<!DOCTYPE ClassificationTree []>
<ClassificationTree version="1.0">
  <ObjectSet>
    . . .
  </ObjectSet>
  <ClassificationSet>
    . . .
  </ClassificationSet>
</ClassificationTree>
```

### Defining the Information Objects

The Object contains a list of objects, like this:

```
<ObjectSet>
  <Object ID="m1">
    <Name>All you ever wanted to know about Cluster Maps</Name>
    <Location>http://www.somehost.com/ClusterMaps.html</Location>
  </Object>
  <Object ID="m2">
    <Name>where to get support</Name>
    <Location>http://www.somehost.com/support.html</Location>
  </Object>
  . . .
</ObjectSet>
```

Every Object has an ID attribute that will be needed later on for the definition of the Classifications. Any legal XML ID can be used.

Furthermore, every Object has a Name and a Location. Both can be arbitrary XML PCDATA strings, since they are only used for display purposes, such as mouse-over effects or when a Cluster Map is exported as a clickable image map. The values are not required to be unique.

All the elements inside the Object element are optional. Omitting an element has the same effect as including the element with no content (i.e. an empty string) between its start and end tags.

## Defining the Taxonomy

The classes in the taxonomy are defined inside the ClassificationSet element, using a list of Classification elements. The ClassificationSet element has the following structure:

```
<ClassificationSet>
  <Classification ID="c1">
    <Name>All Pages</Name>
    <Objects objectIDs="m1 m2"/>
  </Classification>

  <Classification ID="c2">
    <Name>Some Subclass</Name>
    <SuperClass refs="c1"/>
    <Objects objectIDs="m1"/>
  </Classification>

  . . .

</ClassificationSet>
```

Every Classification has an ID that is used to register parent-child relationships between classes. Any legal XML ID value will do.

Classifications can be listed in any order, provided that a parent class is listed before all of its subclasses, i.e. forward references are not allowed.

Classifications have a name that is only used for display purposes. This can be any legal PCDATA String. The name of a classification is not required to be unique.

The super class of a classification is encoded by the SuperClass element. The value of its refs attribute should be the ID value of the super class' parent Classification. The name of this attribute has been chosen to be in plural form to anticipate on possible multiple inheritance features in the future. For the root of the class hierarchy the SuperClass element is omitted. When the file contains several Classifications without a super class reference, only the last Classification and its subclasses will be listed in the viewer.

The Objects element encodes all objects that are contained in this Classification. Its refs attribute is a space-separated list of all ID's of the respective Object elements, defined before in the ObjectSet element.

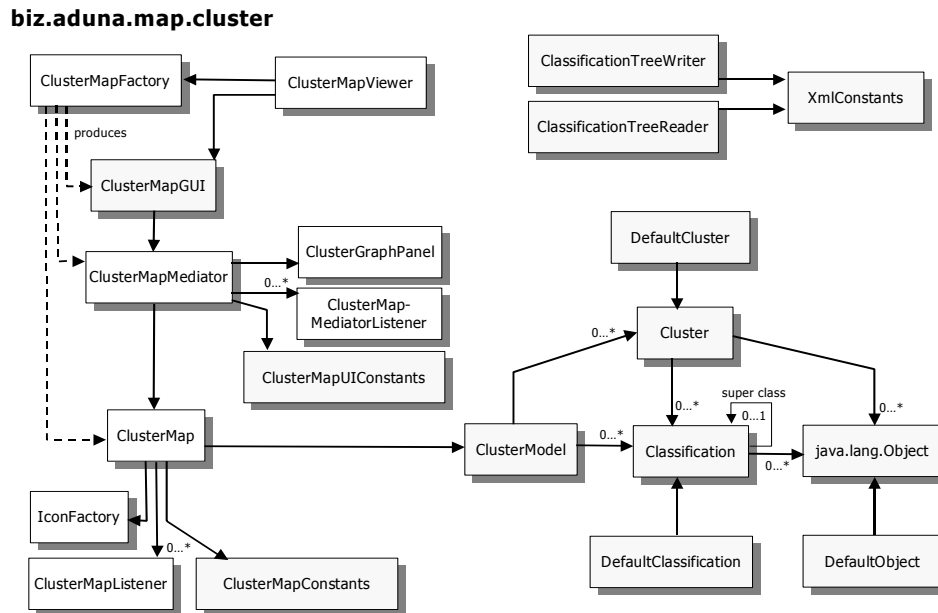
Strictly speaking, an Object can be a member of a Classification without being a member of its parent Classification(s). It depends on the nature of the taxonomy whether this makes sense from a conceptual point.

## 4. Public API's

In this section we will explore the contents and the use of the Cluster Map's API. We will give an overview of the publicly accessible classes and demonstrate their use through code examples. We assume that you have read the previous section and already know about Classifications, Objects, etc.

### Package Overview

The Cluster Map jar file contains only one package with a publicly accessible API. It contains classes and interfaces that embody the data structures, visualization algorithms and GUI front-end of the Cluster Map visualization. The image below contains a diagram showing the classes and their most important relationships.



The **ClusterMap** class is the most central class of this package, producing the Cluster Map visualization. A ClusterMap operates on a **ClusterModel** data structure, containing a set of **Clusters**. Each Cluster represents a unique combination of **Classifications** and contains all **Objects** (representing e.g. documents) that are contained in all of the classes represented by the Cluster but not of any of the other classes present in the ClusterModel.

For example, consider that a ClusterModel has been made representing the classes A, B and C. Suppose that it contains a Cluster representing the combination of the classes A and B. This Cluster then contains all Objects that are members of the classes A and B but not of class C.

Classification and ClusterModel are instances and java.lang.Object is used to represent the Objects in a Classification. This provides great flexibility for the integrator, letting him choose what classes are actually used to provide this functionality. The Cluster Map provides default implementations for all these cases that will suffice for simple



implementations. Most notably, **DefaultClusterModel** contains an efficient algorithm for calculating the set of Clusters based on a set of populated Classifications, so that integrators do not have to reinvent the wheel.

The ClusterMap displays the contents of a ClusterModel as a graph. It can display the graph using the GUI components discussed below or save it to a file, e.g. as an image or clickable image map.

A ClusterMap uses an **IconFactory** to retrieve the icons that are used to display Objects and Classifications. You can set your own implementation of this Interface on the ClusterMap, so that you have full control over the appearance of these objects.

One can register a **ClusterMapListener** on a ClusterMap to get notified about ClusterMap state changes, such as when the ClusterModel or the displayed graphs changed or when settings are altered.

A **ClassificationTreeWriter** can serialize a tree of Classifications in an XML-based format. The resulting document can be read by a **ClassificationTreeReader**, which reconstructs the Classification tree. The structure of this document has been described in the previous section. The names of the XML elements and attributes used in the document are defined in the **XmlConstants** interface. The ClassificationTreeWriter both handles arbitrary Classification and Object implementations but also knows about the specifics of DefaultClassification and DefaultObject. The deserialized Classification tree will only consist of DefaultClassifications and DefaultObjects.

The **ClusterMapMediator** is essential to the construction of a GUI around a ClusterMap. A ClusterMapMediator wraps a ClusterMap and produces various GUI components such as a **ClusterGraphPanel** showing the graph, a JTree derivative allowing for class selection, various Actions, etc. A ClusterMapMediator ensures consistency between all the components it creates. For example, when a class is selected for visualization in the classification tree, it is automatically added to the panel showing the Cluster Map graph. The ClusterMapMediator dynamically creates the ClusterModels visualized in the ClusterMap.

The ClusterMapMediator *itself* is not a GUI component; it merely creates and manages the components with which a sophisticated GUI can be constructed. In order to quickly build a GUI, one can also use a **ClusterMapGUI** instead. This is a JPanel subclass that wraps a ClusterMapMediator, lets it create a number of UI components and arranges them in itself, organized in a natural way. The Cluster Map Viewer has been built using a ClusterMapGUI.

One can register as a **ClusterMapMediatorListener** on a ClusterMapMediator and get notified when its state changes.

The instances of the ClusterMap class and some of the classes that use it are generated by a **ClusterMapFactory**.

In the remainder of this section we will show with example Java code how to use this API. Note that all Exceptions that may be thrown in any of the code fragments are ignored for the sake of simplicity.

## Creating Data Models

In order to be able to create visualisations with the ClusterMap, we first have to prepare our data models. First we will show how to create a populated Classification tree.

A Classification tree can be dynamically created as follows:

```
// Create objects with a location and a name
Object object1 = new DefaultObject("object1", "http://www.site1.com");
Object object2 = new DefaultObject("object2", "http://www.site2.com");
...
Object object10 = new DefaultObject("object10", "http://www.site10.com");

// Create the Classifications
DefaultClassification rootClass = new DefaultClassification("Root");
rootClass.add(object1);
rootClass.add(object2);
...
rootClass.add(object10);

DefaultClassification subclass1 =
    new DefaultClassification("Subclass 1", rootClass);
subclass1.add(object1);
...
subclass1.add(object7);

DefaultClassification subclass2 =
    new DefaultClassification("Subclass 2", rootClass);
subclass2.add(object3);
...
subclass2.add(object10);
```

Another way to establish a Classification tree is to load it from an XML file whose format has been discussed in the previous section. ClassificationTreeReader implements the SAX 1.0 DocumentHandler interface and can thus be used in combination with any SAX 1.0 parser to parse an XML document and construct a Classification tree from its contents.

In order to visualize the contents of the two subclasses, a ClusterModel needs to be created. This ClusterModel contains the set of Clusters derived from the Classifications that will be visualized in the ClusterMap. Creating a ClusterModel is very trivial:

```
// Create a ClusterModel of subclass1 and subclass2
ArrayList classes = new ArrayList();
classes.add(subclass1);
classes.add(subclass2);
ClusterModel model = new DefaultClusterModel(classes);
```

DefaultClusterModel will take care of calculating the set of Clusters. Empty Clusters, i.e. those clusters that do not contain any Objects, are automatically removed. Alternatively, you can calculate them yourselves and also specify them as an argument to the DefaultClusterModel constructor.

## Using a ClusterMap Instance

Now that we have a ClusterModel to visualize, we create a ClusterMap instance that takes care of this:

```
// Create a ClusterMap instance
ClusterMapFactory factory = ClusterMapFactory.createFactory();
ClusterMap map = factory.createClusterMap();
```

We then set the ClusterModel on the ClusterMap and tell it to update its visualization:

```
// Let the ClusterMap visualize this ClusterModel
map.setClusterModel(clusterModel);
map.updateGraph();
```

The `updateGraph` method makes sure a graph is created reflecting the contents of the `ClusterModel` and that it is given a layout. Visualizing a `ClusterModel` requires at least two method invocations in order to allow you to change other settings as well before updating the visualization. Be aware that, when the `ClusterMap` is embedded in a GUI, all components displaying the graph will also automatically update when `updateGraph` is invoked.

The visualization can now be saved as a PNG image file and optionally an accompanying HTML file that defines the clickable regions:

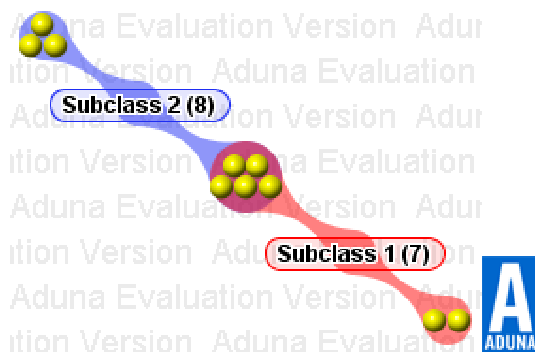
```
// Save the map as an image
FileOutputStream stream = new FileOutputStream("image.png");
map.exportPngImage(stream);
stream.close();

// Save the corresponding HTML image map
FileWriter writer = new FileWriter("map.html");
Properties props = new Properties();
props.setProperty("imageFileName", "image.png");
props.setProperty("fullDocument", "true");
props.setProperty("title", "Example of a Cluster Map");

map.exportImageMap(writer, props);
writer.close();
```

The `Properties` object passed to `exportImageMap` instructs the `ClusterMap` how to create the HTML code. In this case the name of the image file and the title of the resulting HTML document are passed and the `Cluster Map` is instructed to create a complete HTML document (i.e., something that can be directly viewed in a browser) rather than a HTML fragment that can be embedded in an HTML template.

The resulting image, saved in the current directory as “image.png”, will look more-or-less like this:



The resulting HTML document, saved as “map.html” in the same directory, can now be viewed in any regular web browser.

## Creating a Graphical User Interface

The previous example demonstrated the generation of image maps, which can be useful in server application environments. However, the visualization gains a lot in usefulness and usability when viewed inside a GUI. We will now show how to realize such a GUI using the `ClusterMapMediator` and the `ClusterMapGUI`.

Let's assume that we want to extend a certain application with a viewer frame that shows a Classification tree. We also assume that the tree data structure has already been constructed. First we repeat the steps to create a ClusterMap instance:

```
// Create a ClusterMap
ClusterMapFactory factory = ClusterMapFactory.createFactory();
ClusterMap map = factory.createClusterMap();
```

We wrap the ClusterMap in a ClusterMapMediator and set the Classification tree in it:

```
// Create the ClusterMapMediator
ClusterMapMediator mediator = factory.createMediator(map);
mediator.setClassificationTree(rootClass);
```

The ClusterMapMediator on its turn is wrapped in a ClusterMapGUI:

```
// Create the ClusterMapGUI
ClusterMapGUI gui = factory.createGUI(mediator);
```

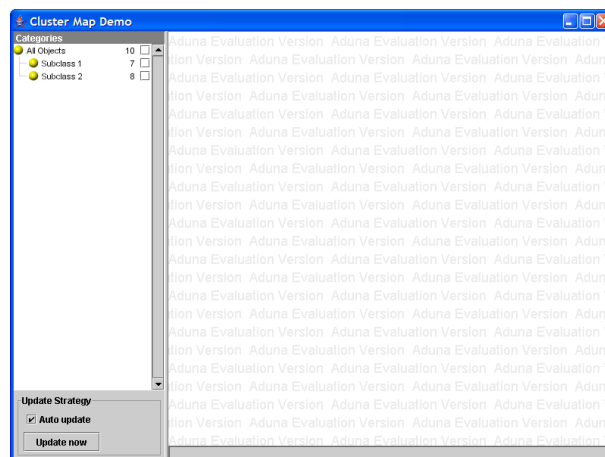
The ClusterMapGUI will retrieve Swing components and actions from the ClusterMapMediator and arrange them in itself. Replace this code with your own code if you want to organize the components differently.

Finally, we embed the ClusterMapGUI in a JFrame and make it visible:

```
// Open a JFrame containing the ClusterMapGUI
JFrame frame = new JFrame("Cluster Map Demo");
frame.getContentPane().add(gui);
frame.setSize(new Dimension(800, 600));

SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        frame.setVisible(true);
    }
});
```

A window will be opened showing the following content:



You can now toggle the checkboxes in the tree to select the classes to display.

## Thread Safety

Just like most Swing components, the ClusterMap class and the various GUI-related classes are not thread safe. This means that in a GUI application it is probably best to do all method calls on any Cluster Map-related object in the AWT event dispatch thread once it has been made visible in a GUI component.

Single-threaded applications such as command line-driven applications for batch processing have nothing to worry about.

## Jar Files

The Cluster Map software package contains a number of jar files. The Cluster Map Viewer needs all of them to run properly, but other applications integrating the library may only need a few of them. You can use the explanation of the files below to judge which files you should put in your classpath.

- The `aduna-clustermap-2006.1.jar` and `aduna-clustermap-2006.1-resources.jar` files contain the Cluster Map code base and are always necessary.
- The `jlfr-1.0.jar` file contains a set of icons provided by Sun Microsystems and is used in the GUI of the Cluster Map.
- The `sesame-1.1.3.jar` file contains classes that are required when using the provided functionality for displaying the contents of a Sesame repository. See <http://www.openrdf.org> for more information on the Sesame RDF database.
- The `looks-1.3.1.jar` and `winlaf-0.5.jar` files are only used by the Cluster Map Viewer to refine its look and feel. They are applied when the Viewer detects that it is running on a Linux/Unix desktop or on a Windows desktop, respectively.

## 5. Configuration

As an integrator you will probably want to fine-tune the appearance and behavior of parts of the visualization. Both the ClusterMap and the ClusterMapMediator have a set/get-mechanism based on simple key-subject-value triples that can be used to adapt their behavior. Furthermore, when exporting the ClusterMap's current visualization to an HTML image map or when obtaining Actions or JComponents from a ClusterMapMediator, keys or key-value pairs are also used. The possible keys, subjects and values are all described in this section.

### Configuring a ClusterMap

The parameters that can be set on a ClusterMap mainly influence the rendering of the graph, with only a few exceptions. Each key is a String that is compared with the known keys using String's equals(Object) method, after the white space at the start and the end have been removed using the trim() method. These keys, as well as the *symbolic* (non-numerical) values, are defined as constants in the ClusterMapConfigurationConstants interface. We will give the symbolic name as well as the actual string value of each key and symbolic value.

The subject is a parameter object that is involved in the property. This needs to be specified when the property is not global for the entire ClusterMap(Mediator), e.g. when it involves a particular Classification or Object. In case of global properties one can specify *null* as the subject.

The parameter value is either a symbolic value, encoded as a String and defined in the ClusterMapConfigurationConstants interface, or a reference type such as a Font, Color, Boolean, etc.

The following code demonstrates how the settings of a ClusterMap can be adjusted:

```
// Update some visual properties
map.setProperty(KEY_RENDERING_CLUSTER_MODE, null, VALUE_SINGLE_ENTITY);
map.setProperty(KEY_RENDERING_SCALE, null, new Double(0.5));
map.applyProperties();
```

This code assumes that the class in which it is contained “implements” the ClusterMapConfigurationConstants interface, so that its name does not have to be repeated every time when one of its symbols is referenced. This code is equivalent to:

```
// update some visual properties
map.setProperty("rendering.cluster.mode", null, "singleEntity");
map.setProperty("rendering.scale", null, new Double(0.5));
map.applyProperties();
```

The properties set on the ClusterMap do not become effective until applyProperties() is invoked. This design allows the ClusterMap to optimize the updating of the settings, which can sometimes be computationally expensive. The ClusterMap interface also allows the retrieval of the current value and the default value of a given property. The result is a String containing the return value, using the same value encoding as the setProperty method.

We will now give a systematic explanation of all keys, their subjects and their possible values currently supported by the ClusterMap. Default values of these parameters are not described, as these tend to change a lot over time as the Cluster Map is further developed.

Name: **layout.reuseLayout**

Symbol: KEY\_LAYOUT\_REUSE\_LAYOUT

Subject: n/a

Value: a Boolean

Description: indicates whether the layout algorithm should try to use the layout of the existing graph as a starting point when `updateGraph()` is invoked. This is mainly useful when the ClusterMap's user interface is used and several subsequent maps are generated. In that case you do not want to have a radically different graph layout between two subsequent graphs sharing some Classification objects, as this will disturb the mental map that the user has constructed of the information. The drawback is that the resulting graph layout can be less optimal, although this is very rare.

Name: **rendering.font**

Symbol: KEY\_RENDERING\_FONT

Subject: n/a

Value: a Font that is available on the platform used.

Description: the Font used to display the names and cardinalities of the Classifications.

Name: **rendering.antiAliasing**

Symbol: KEY\_RENDERING\_ANTI\_ALIASING

Subject: n/a

Value: a Boolean

Description: indicates whether the rendering should be anti-aliased. This considerably improves the quality of the resulting image, at the expense of a decrease in rendering speed. Under normal circumstances and with reasonable hardware, the extra costs will typically be justified.

Name: **rendering.antiAliasing.fractionalTextMetrics**

Symbol: KEY\_RENDERING\_ANTI\_ALIASING\_FRACTIONAL\_TEXT\_METRICS

Subject: n/a

Value: a Boolean

Description: indicates whether fractional metrics may be used during text rendering when anti-aliasing is turned on. This additionally improves the quality of the rendering of the text and is usually worth the extra computational costs.

Name: **rendering.insets**

Symbol: KEY\_RENDERING\_INSETS

Subject: n/a

Value: an Insets, with each of its four sizes  $\geq 0$ .

Description: indicates the amount of white space (in pixels) that should be kept around the border of the graph.

Name: **rendering.scale**

Symbol: KEY\_RENDERING\_SCALE

Subject: n/a

Value: a Double with a value  $> 0.0$

Description: the scale at which the graph should be rendered. The default scale is "1.0".

Name: **rendering.size.maximum**

Symbol: KEY\_RENDERING\_SIZE\_MAXIMUM

Subject: n/a

Value: either a Dimension with both width and height  $> 0$ , or null.

Description: indicates the maximum size (in pixels) the rendering may occupy. Note that this

relates to the actual size on screen or in the produced image, it is not scaled by the “rendering.scale” parameter. When the value is “null”, then the size is entirely determined by the size of the graph, its insets and the scale. When the value is non-null and the size of the rendering with the insets and scaling taken into account is bigger than this size, then the actual scaling used during rendering will be decreased in order to let the rendering fit the maximum size, while keeping a 1:1 aspect ratio and without changing the scale property.

Name: **rendering.size.maximum.expandToMaximum**

Symbol: KEY\_RENDERING\_SIZE\_MAXIMUM\_EXPAND\_TO\_MAXIMUM

Subject: n/a

Value: a Boolean

Description: when the “rendering.size.maximum” property is non-null, this flag can be used to instruct the ClusterMap to add whitespace to the right and/or the bottom of the rendering when necessary, in order to always have a size equal to the set maximum size. This is for example useful when generating images in a server environment which are to be automatically embedded in a website or some other publication mechanism which expects a fixed size. With this property you can enforce that the created images always have the same size, even though part of it may remain blank.

Name: **rendering.colorScheme**

Symbol: KEY\_RENDERING\_COLOR\_SCHEME

Subject: n/a

Value: VALUE\_UNIFORM (“uniform”), VALUE\_CLASSIFICATIONS (“classifications”) or VALUE\_RELEVANCE (“relevance”)

Description: indicates the color scheme that determines the colors used in the rendering of the graph.

Name: **rendering.object.defaultNormalIcon**

Symbol: KEY\_RENDERING\_OBJECT\_DEFAULT\_NORMAL\_ICON

Subject: n/a

Value: a String specifying the resource location of an image

Description: specifies which image should be used for non-selected Objects

Name: **rendering.object.defaultSelectedIcon**

Symbol: KEY\_RENDERING\_OBJECT\_DEFAULT\_SELECTED\_ICON

Subject: n/a

Value: a String specifying the resource location of an image

Description: specifies which image should be used for selected Objects

Name: **rendering.classification.defaultNormalIcon**

Symbol: KEY\_RENDERING\_CLASSIFICATION\_DEFAULT\_NORMAL\_ICON

Subject: n/a

Value: a String specifying the resource location of an image

Description: specifies which image should be used for non-selected Classifications.

Name: **rendering.classification.defaultSelectedIcon**

Symbol: KEY\_RENDERING\_CLASSIFICATION\_DEFAULT\_SELECTED\_ICON

Subject: n/a

Value: a String specifying the resource location of an image

Description: specifies which image should be used for selected Classifications.

Name: **rendering.iconFactory**

Symbol: KEY\_RENDERING\_ICON\_FACTORY



Subject: n/a

Value: an IconFactory instance

Description: specifies the IconFactory that will be used to retrieve icons for the display of Objects and Classifications in the graph.

Name: **rendering.iconPadding**

Symbol: KEY\_RENDERING\_ICON\_PADDING

Subject: n/a

Value: an Integer with a value  $\geq 0$

Description: specifies the space left between Object icons, when displayed in the hexagonal layout of a Cluster.

Name: **rendering.displayingClassIcons**

Symbol: KEY\_RENDERING\_DISPLAYING\_CLASS\_ICONS

Subject: n/a

Value: a Boolean

Description: indicates whether the Classifications should be rendered with their icons in the graph, or whether only their names should be displayed. Be aware that the visibility of the icons may be overruled by other settings such as the cluster edge rendering mode.

Name: **rendering.classification.maximumPixelLength**

Symbol: KEY\_RENDERING\_CLASSIFICATION\_MAXIMUM\_PIXEL\_LENGTH

Subject: n/a

Value: an Integer

Description: the maximum string width in pixels that can be used to display Classification names. When the preferred length is larger, the text is truncated in the typical JLabel style, by removing characters at the end and replacing them with “...” so that it all fits in the allowed space. A width less than or equal to zero indicates that there is no maximum length.

Name: **rendering.cluster.mode**

Symbol: KEY\_RENDERING\_CLUSTER\_MODE

Subject: n/a

Value: VALUE\_SINGLE\_ENTITY (“singleEntity”) or VALUE\_MULTIPLE\_ENTITIES (“multipleEntities”)

Description: indicates whether clusters should be displayed as a single visual entity (normally a colored cylinder) or as a set of individual Objects. The latter mode is more intuitive but clearly has a scalability problem.

Name: **rendering.cluster.forceSingleEntityThreshold**

Symbol: KEY\_RENDERING\_CLUSTER\_FORCE\_SINGLE\_ENTITY\_THRESHOLD

Subject: n/a

Value: an Integer with value  $\geq 0$ .

Description: indicates the amount of objects a Cluster needs to contain in order for it to become automatically displayed in single entity mode, regardless of the global rendering.cluster.mode setting.

Name: **rendering.cluster.singleEntity.sizeFunction**

Symbol: KEY\_RENDERING\_CLUSTER\_SINGLE\_ENTITY\_SIZE\_FUNCTION

Subject: n/a

Value: VALUE\_SQUARE\_ROOT (“squareRoot”) or VALUE\_LOGARITHMIC (“logarithmic”)

Description: when the current cluster mode is “singleEntity”, the diameter of a Cluster is determined by an abstract size function. This parameter determines whether this function is square root-based or logarithmic-based. The former appears to be more realistic (the *area* of

a Cluster increases linearly with the number of objects), but the latter allows for even better scalability of the visualization.

Name: **rendering.cluster.singleEntity.squareRootSize.scale**

Symbol: KEY\_RENDERING\_CLUSTER\_SINGLE\_ENTITY\_SQUARE\_ROOT\_SIZE\_SCALE

Subject: n/a

Value: a Double with a value > 0.0

Description: when the current cluster mode is “singleEntity” and the size function is square root-based, this parameter can be used to scale the cluster diameter.

Name: **rendering.cluster.singleEntity.logarithmicSize.scale**

Symbol: KEY\_RENDERING\_CLUSTER\_SINGLE\_ENTITY\_LOGARITHMIC\_SIZE\_SCALE

Subject: n/a

Value: a Double with a value > 0.0

Description: when the current cluster mode is “singleEntity” and the size function is logarithmic-based, this parameter can be used to scale the cluster diameter.

Name: **rendering.cluster.singleEntity.radius.minimum**

Symbol: KEY\_RENDERING\_CLUSTER\_SINGLE\_ENTITY\_RADIUS\_MINIMUM

Subject: n/a

Value: an Integer with a value > 0

Description: indicates the minimum radius in pixels a cluster can have. This is only used when the current cluster mode is “singleEntity”.

Name: **rendering.cluster.singleEntity.radius.maximum**

Symbol: KEY\_RENDERING\_CLUSTER\_SINGLE\_ENTITY\_RADIUS\_MAXIMUM

Subject: n/a

Value: an Integer with a value > 0

Description: indicates the maximum radius in pixels a cluster can have. This is only used when the current cluster mode is “singleEntity”.

Name: **rendering.clusterEdge.shapeMode**

Symbol: KEY\_RENDERING\_CLUSTER\_EDGE\_SHAPE\_MODE

Subject: n/a

Value: VALUE\_TRIANGULAR\_SHAPE (“triangular”), VALUE\_BALLOON\_SHAPE (“balloon”), VALUE\_AMOEBA\_SHAPE (“amoeba”) or VALUE\_LINES\_SHAPE (“lines”)

Description: indicates the shape the edges between clusters and classes should have.

## Configuring a ClusterMapMediator

Currently only a few properties of the ClusterMapMediator are adaptable, influencing its interactive behavior. The settings are applied similarly to the ClusterMap, including the use of an applyProperties() method that applies the specified properties batch-wise. The keys supported by the ClusterMapMediator are defined in the ClusterMapUIConstants interface. The possible values are either symbolic values defined in the same interface or primitive or reference types using the same String encoding as the ClusterMap.

We will now list the adaptable ClusterMapMediator properties.

Name: **animation.enabled**

Symbol: KEY\_ANIMATION\_ENABLED

Subject: n/a

Value: a Boolean

Description: indicates whether the transition from one ClusterModel visualization to the next should be shown as an animation. When set to false, the transition is displayed instantaneously.

Name: **animation.edgesEnabled**

Symbol: KEY\_ANIMATION\_EDGES\_ENABLED

Subject: n/a

Value: a Boolean

Description: indicates whether the animation should render the edges. Setting this to “false” will increase the maximum frame rate that can be achieved.

Name: **animation.textEnabled**

Symbol: KEY\_ANIMATION\_TEXT\_ENABLED

Subject: n/a

Value: a Boolean

Description: indicates whether the animation should render the text in the graph. Setting this to “false” will increase the maximum frame rate that can be achieved, although the effect will be marginal. Consider turning off edge animation (animation.edgesEnabled) when frame rate is an issue.

Name: **animation.allowedDuration**

Symbol: KEY\_ANIMATION\_ALLOWED\_DURATION

Subject: n/a

Value: a Long indicating milliseconds

Description: sets an upper bound on the amount of time an animation is allowed to consume.

Name: **autoApplyCheckedClasses**

Symbol: KEY\_AUTO\_UPDATING\_GRAPH

Subject: n/a

Value: a Boolean

Description: indicates whether the graph display should be automatically updated when the user (un)checks classes in the Classification tree. When switched off, the Action associated with the “suspendedUpdate” key is automatically enabled. Firing this Action will then update the graph to reflect the current set of checked classes.

Name: **scaleToFitViewport**

Symbol: KEY\_SCALE\_TO\_FIT\_VIEWPORT

Subject: n/a

Value: a Boolean

Description: indicates whether the graph should be automatically scaled down when viewed in a scroll pane, so that the scroll bars never need to appear. Since the ClusterGraphPanel uses a scroll pane, this makes sure that the graph always fits in the ClusterGraphPanel.

Name: **selection.vertexSelectionMode**

Symbol: KEY\_SELECTION\_VERTEX\_SELECTION\_MODE

Subject: n/a

Value: VALUE\_CLUSTER\_SELECTION\_MODE (“clusterSelectionMode”) or VALUE\_OBJECT\_SELECTION\_MODE (“objectSelectionMode”)

Description: used to set whether the user selects Clusters or individual Objects when clicking in the graph.

Name: **export.image.enabled**

Symbol: KEY\_EXPORT\_IMAGE\_ENABLED

Subject: n/a

Value: a Boolean

Description: indicates whether export of a PNG image is allowed. When set to false, the UI will no longer offer this possibility.

Name: **export.imageMap.enabled**

Symbol: KEY\_EXPORT\_IMAGE\_MAP\_ENABLED

Subject: n/a

Value: a Boolean

Description: indicates whether export of a clickable image map is allowed. When set to false, the UI will no longer offer this possibility.

Name: **export.animation.enabled**

Symbol: KEY\_EXPORT\_ANIMATION\_ENABLED

Subject: n/a

Value: a Boolean

Description: indicates whether export of a selected list of graphs as an animation is allowed. When set to false, the UI will no longer offer this possibility.

## Configuring HTML output

When exporting a ClusterMap's contents as an HTML clickable image map, one can specify a Properties instance containing key-value pairs that encode the desired characteristics of the generated HTML. What follows is a description of the possible keys and values, which are all Strings. Boolean values are encoded as either "true" or "false", ignoring case and white space. When an unparseable value is used, the setting is ignored (i.e. the default setting is used).

Key: **fullHtmlDocument**

Value: a Boolean

Description: when set to "true", a complete HTML document is generated that can immediately be viewed in a web browser. This is useful for debugging purposes. On the other hand, in production environments one typically wants to embed the clickable image in an HTML document. When "false", only the HTML MAP element and its contents, defining the clickable regions, are generated.

Key: **title**

Value: any String that is legal as content of the HTML TITLE element.

Description: the string used as the title of the generated HTML document. This is only used when a complete HTML document is generated.

Key: **imageFileName**

Value: a valid image URL that can be used as a value of the SRC attribute of the HTML IMG element.

Description: the file name that will be used to refer to the corresponding image, relative to the path of the generated HTML document. This is only used when a complete HTML document is generated.

# Obtaining Actions

In order to enable the construction of UI components like toolbars, menu's, etc., the ClusterMapMediator facilitates the retrieval of Action instances (see the javax.swing.Action interface) through the use of the `getAction` method and the appropriate keys.

The following Actions are currently supported:

Key: **open**

Action: opens a JFileChooser that allows the user to open a new Classification tree, to be shown by the ClusterMapMediator.

Key: **openSesame**

Action: opens a dialog that allows the user to select and load an RDF repository from a Sesame server (see <http://www.openrdf.org>).

Key: **saveAs**

Action: opens a JFileChooser that lets the user enter a file name and saves the current Classification tree to that file.

Key: **export**

Action: opens a wizard that allows the user to export the current visualization to several formats. The wizard will only offer export possibilities that are currently enabled.

Key: **back**

Action: restores the previous graph in the navigation history, similar to a back button in a web browser.

Key: **forward**

Action: restores the next graph in the navigation history, similar to a forward button in a web browser.

Key: **reset**

Action: resets the current set of visualized Classifications in the ClusterMap to the visualization showing only the root Classification.

Key: **clear**

Action: clears the current display, so that no Classifications are displayed.

Key: **preferences**

Action: creates and makes a preferences editor visible. The preferences editor allows the setting of everything that is configurable in the ClusterMap and ClusterMapMediator (and currently even more).

Key: **animation**

Action: toggles the use of animation in the graph display.

Key: **scaleToFitViewport**

Action: toggles the value of the “scale to fit viewport” setting.

Key: **zoomSpecified**

Action: opens a dialog asking the user to enter a percentage, and scales the graph to that level.

Key: **suspendedUpdate**

Action: updates the graph display so that it displays the current set of classifications checked in the classification tree. This action is tightly connected to the `autoUpdatingGraph` property of the `ClusterMapMediator`. The enabled state of the Action is adapted automatically when necessary (i.e., when `autoUpdatingGraph` is turned off and the user (un)checks a classification in the tree).

Key: **updateLayout**

Action: reruns the graph layout producer, possibly resulting in a better graph layout.

Key: **treeExpandOneLevel**

Action: expands the tree node in the classification JTree that is currently beneath the mouse pointer, if it is not a leaf node. This Action is meant to populate a popup menu on the JTree. This Action contains all the functionality to determine the node on which the user has clicked and whether it should be enabled.

Key: **treeExpandTwoLevels**

Action: expands the tree node beneath the mouse pointer two levels deep, if it is not a leaf node. Similar in usage to `treeExpandOneLevel`.

Key: **treeExpandNLevels**

Action: expands the tree node beneath the mouse pointer to a depth specified by the user through a dialog. Similar in usage to `treeExpandOneLevel`.

Key: **treeCollapseAll**

Action: collapses all tree nodes except the root node, i.e. brings the expansion state of the tree back into its original state.

Key: **treeArrangeName**

Action: arranges the children of the clicked tree node alphabetically. This Action contains all the functionality to determine the node on which the user clicked and whether it should be enabled.

Key: **treeArrangeSize**

Action: arranges the children of the clicked tree node in decreasing number of Objects. Similar in usage to `treeArrangeName`.

Key: **treeArrangeOriginalOrder**

Action: restores the original order of the children of the clicked tree node. Similar in usage to `treeArrangeName`.