- -

The Infovis Toolkit

Jean-Daniel Fekete

- -

The InfoVis Toolkit

Jean-Daniel Fekete

- -

- -

Table of Contents

- -

List of Figures

- -

List of Examples

- -

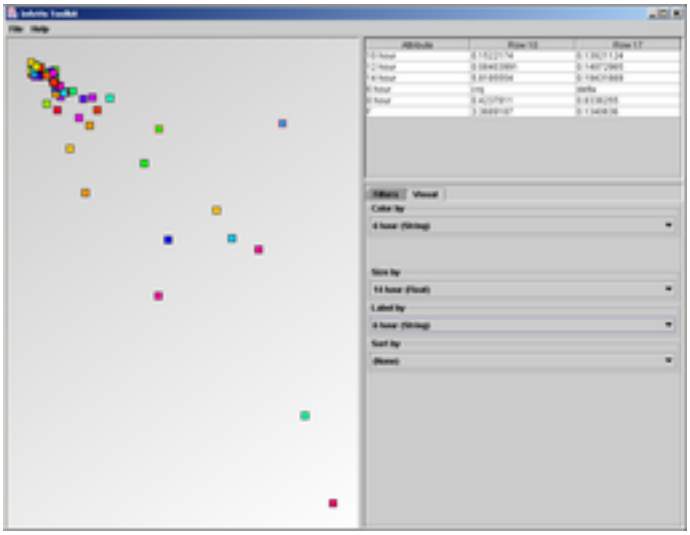
Chapter 1. Introduction

The InfoVis toolkit is a software package aimed at simplifying the development of Information Visualization Systems. It is written in Java, capitalizing on its rich interactive graphics environment and portability.

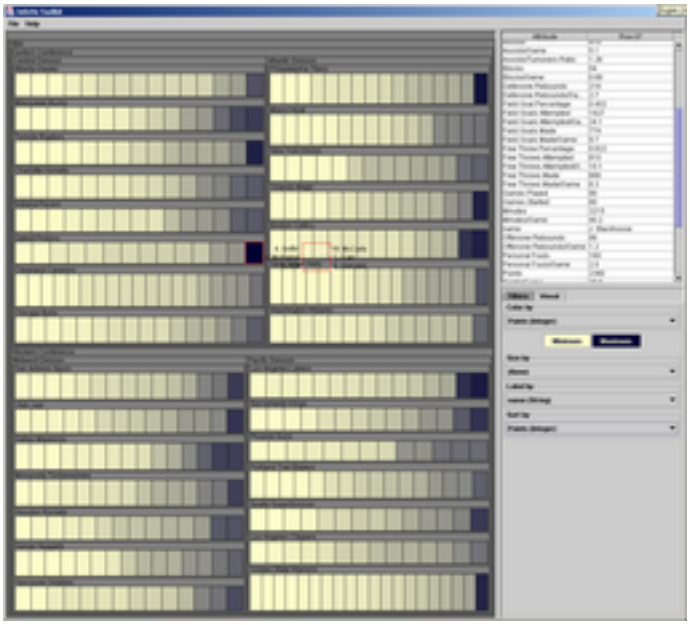
Information Visualization is a domain that emerged in the early 1990 and has expanded at a steady pace since then, showing great results, innovative concepts and techniques. So many concepts and techniques that it is challenging to keep pace with the implementation of the most useful techniques. The InfoVis toolkit is designed to be a repository of know-how for building highest quality information visualization systems.

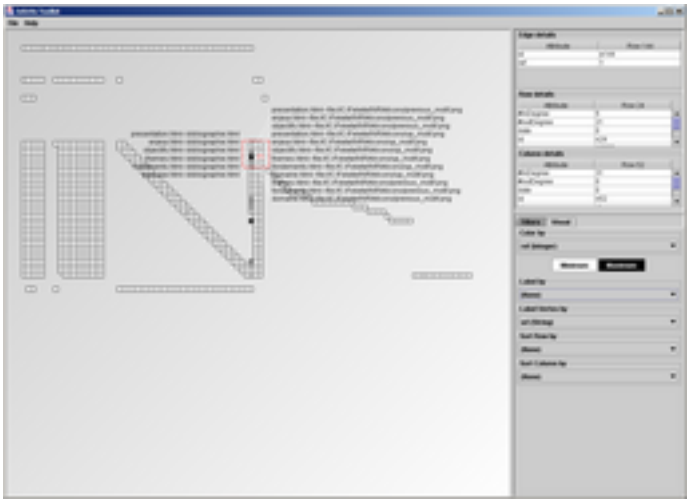
Practically, visualizations translate data items into visual marks. Data items are made of typed data attributes organized in a data structure, such as employee records in a table, or files in a hierarchical file system. Visual marks are eventually made of colored pixels on a screen, but can more accurately be described as geometric entities displayed with graphical attributes such as color or transparency. A visualization technique consists in translating each data item into a geometry and related graphical attributes to draw them on screen in a specified order. The visualization toolkit provides several visualization techniques for three main data structures: tables, trees and graphs. It also provides mechanisms to add visualization techniques and data structures.

Figure 1.1. Examples of Visualizations Provided by the Infovis Toolkit



- -





Visualizations also involves *interacting* with the visualized items and the data structures. Interactions include navigation into the data structure — including zooming, dynamically filtering visualized items — including dynamic queries, and various dynamic techniques such as space deformations using fisheye lenses or dynamic labeling. The visualization toolkit provides several techniques for interacting with the visualizations and mechanisms to add new interaction techniques.

Interacting with visualizations require stringent constraints on time. Dynamic updates of visualization should be performed in less than 100ms to appear smooth and continuous. This constraint requires the data structures to be organized in memory in a way specially crafted for fast access. The visualization toolkit provides homogeneous data structures to strore visualization data into memory. All data structures are represented as tables of columns. Each column contain data of homogeneous data type such as integers, floating point values or strings. This mechanism allows new columns to be added to existing data structures, allowing new attributes to be computed and used to enrich visualizations in a uniform way.

This document is both a tutorial and a user's manual. The reference information can be found on the html descriptions produced by Javadoc and will keep synchronized with the code more closely than this manual.

--

Chapter 2. Tutorial

Starting with the Infovis Toolkit

Several sample applications of the Infovis Toolkit are provided in the example folder in the distribution. Visualizing a table as a time-series data can be done as follows:

Example 2.1. Simple visualization program for time series data

```
public class Example1 {
   public static void main(String args[]) {
      String fileName =
         (args.length == 0) ? "data/salivary.tqd" : args[0];
      DefaultTable t = new DefaultTable();
      t.setName(fileName);
      AbstractReader reader =
         TableReaderFactory.createReader(fileName, t);
      if (reader == null || !reader.load()) {
         System.err.println("cannot load " + fileName);
         return;
      }
      TimeSeriesVisualization visualization =
         new TimeSeriesVisualization(t);
      VisualizationPanel panel =
         new VisualizationPanel(visualization);

      JFrame frame = new JFrame(fileName);
      frame.getContentPane().add(panel);
      frame.setVisible(true);
      frame.pack();
   }
}
```

The package declarations have been omited for clarity. The main program creates and displays a simple time series visualization from the file name specified in the first argument of the program. First, a table is created and named with the file name. Loading the file is done in two steps: finding a reader and actually loading the file from this reader. The reader is created through a *factory object*. Factories are used to create objects indirectly according to some specified parameters (we will describe them later in depth). Here, a reader object is created from a file name and a table object. The factory analyses the file name and maybe the file contents to create the most suitable reader for it. If no reader is returned or the reader cannot load the file, the program is exited with an error message. Otherwise, a visualization object is created and inserted in a standard Java/Swing JFrame insided a visualization panel.

In this example, the visualization is not interactive because we haven't created any control panel associated with the visualization. We can have the control panel if we replace the creation of the visualization panel by the following code:

```
- -
      ControlPanel panel =
         ControlPanelFactory.sharedInstance()
          .createControlPanel(visualization);
```

Factories are quite simple and very useful for extending the infovis toolkit. They are simple objects with one creator method that looks at it arguments and create an object according to them. The TableReaderFactory looks at the file name and, if it recognizes that it ends with the ".csv" extension returns a new CVSTableReader. If it receives a file name ending with ".csv.gz", it recognizes that the file is compressed by GZIP and decompress is on the fly. New file readers can then be defined by programmers and added to newer releases of the infovis toolkit and the same program will then be able to read these new file formats without modification. Other kinds of customizations are allowed by the "factory" pattern but we will describe them on a separate section.

Organization of the Infovis Toolkit

Although the Infovis Toolkit may seem large by the number of classes it defines, it follows a very simple structure. At the top-level, it defines the most important interfaces and classes: `Column`, `Table`, `Tree`, `Graph`, `Visualization` and `Metadata`. The first-level defines the several column classes in the `column` package, basic input/output classes in the `io` package, several metadata categories in the `metadata` package, control panels and interaction components in the `panel` package, a `visualization` package for the visualization internals and subpackages for the data-structures: `table`, `tree` and `graph`. Finally, a `utils` package contains utility classes that don't fit in other packages.

Each data-structure related package also contain sub-packages for their input/output components and their visualizations. This is where the default readers and writers are defined, as well as the visualizations provided by default.

Carrying on: Visulizing a Tree as a Treemap

To visualize a tree data-structure using the "Treemap" [] visualization technique, the code would be:

Example 2.2. Simple visualization program for hierarchical data

```
--
import infovis.tree.DefaultTree;
import infovis.tree.io.TreeReaderFactory;
import infovis.tree.visualization.TreemapVisualization;
import infovis.io.AbstractReader;
import infovis.panel.ControlPanel;
import infovis.panel.ControlPanelFactory;

import javax.swing.JFrame;

public class Example2 {
   public static void main(String[] args) {
      String fileName =
         (args.length == 0) ? "data/salivary.tqd" : args[0];
      DefaultTree t = new DefaultTree();
      AbstractReader reader =
         TreeReaderFactory.createReader(fileName, t);
      if (reader == null || !reader.load()) {
         System.err.println("cannot load " + fileName);
      }

      TreemapVisualization visualization =
         new TreemapVisualization(t, null, Squarified.SQUARIFIED);
      ControlPanel control =
         ControlPanelFactory.sharedInstance().createControlPanel(
            visualization);

      JFrame frame = new JFrame(fileName);
      frame.getContentPane().add(control);
      frame.setVisible(true);
      frame.pack();
   }
}
```

The package declaration for the program Example 2.1, “Simple visualization program for time series data” is:

```
import infovis.io.AbstractReader;
import infovis.table.DefaultTable;
import infovis.table.io.TableReaderFactory;
import infovis.table.visualization.TimeSeriesVisualization;
import infovis.panel.ControlPanel;
import infovis.panel.ControlPanelFactory;

import javax.swing.JFrame;
```

Specifying Visual Attributes

Each visualization technique use a layout algorithm and several visual attributes to set the color or geometric properties of the displayed items. For example, scatter plots compute the position of items based on two visual attributes: one for the X coordinate and one for the Y coordinate. Each item can have a size and a color. These visual attributes can be associated with data columns or set to default values.

Some visual attributes are generic to all visualizations and others are specific. The generic attributes are color, label and size. For example, if you know that the tree in example Example 2.2, “Simple visualization program for hierarchical data” has an column called "name", you can specify it as the label column like this:

```
visualization.setVisualColumn(
   Visualization.VISUAL_LABEL,
   t.getColumn("name"));
```

The label will then appear with a form depending on the visualization: inside items for scatter plots or trees. If dynamic labels are enabled, they will use the content of this column for their labels.

Visualization technique may use additional visual attributes such as shape for scatter plots. Some visual attributes, such as color and stroke, are defaulted when they not specified. The default value can be set through the methods of the form: `setDefault<ATTRIB>(ATTRIB default)`.

Using Fisheye Lenses or Dynamic Labeling

Allowing Fisheye Lenses is just a visualization option:

```
visualization.setFisheyes(new Fisheyes());
```

Enabling Dynamic Labeling involves creating a new visualization layer above the main visualization. Here is the required code:

```
TimeSeriesVisualization visualization =
   new TimeSeriesVisualization(t);
ControlPanel control =
   ControlPanelFactory.sharedInstance().
      createControlPanel(excentric);
```

The toolkit provides higher-level mechanisms to create visualizations and display them using option panels allowing fisheye views and dynamic labeling to be enabled or disabled at will.

- -

Chapter 3. Data Structures

Data structures define two things: a topology and data attributes. Classical topologies include tables, trees and graphcs. data attributes are usually defined as tuples associated with the topology. For example, an employee data type can be described by a set of attributes such as "birth date", "first name", "last name", and "social security number". A table of empolyees is an indexed container that contains one employee per defined index.

Tables, trees and the graphs data structures are well documented on text books such as [Knuth97] [Cormen01], albeit with a different focus than ours. The associated data attributes associated with the topology are well described in databases -- where they are sometimes called tuples and managed as tables -- and in programming languages also provide simple means to define data types and aggregate them in compound structures such as tables, trees or graphs. Advanced computer languages allow the definition of tables of employees, trees of employees or graph of employees related by some relationship.

These systems usually represent tables as an array of tuples where each tuple is a row and each attribute is a column. They allow for easy insertion or deletion of new tuples, but don't allow the easy modification of the data structure.

The Infovis Toolkit takes a different approach than databases tuples and language aggregated data structures: it uses "columns" internally so that new attributes can easilly be added to existing data structures. Inserting or removing rows is as easy as inserting or removing attributes.

This approach is important in Information Visualization as it is in spreadsheet calculators since several computations or visualizations require new values to be computed and manipulated.

In the next sections, we explain the main properties of a column and how it can be used. We then describe how traditional data structures are built on top of the columns.

Columns

A column is an index container that contains a homogeneous data structure, such as integers, floating point values or string. Columns also manage *Metadata*, *notification* and *undefined values*.

Column is a Java interface with several concrete implementations. There are two derivation paths for columns, one is for concrete data types and the other one is for dense versus sparse columns.

Dense columns are based on primitive array, providing a constant access time and a memory footprint linear with the number of entries. Sparse columns are based on sorted trees, providing a log(n) access time and a memory footprint linear with the number of defined entries. They come specialized for the most used data types such as int, long, float, double, String, Object, bool and some others used interally.

Figure 3.1, "The Column interface" describes the interface of `Column` using a slightly extended Java syntax parametrizing the class by a concrete type called TYPE. Each time you read this TYPE in the interface, it means that each concrete class implementing the interface provides the specified method with a concrete type. For example, the IntColumn class implements all the methods with "TYPE" replaced by int so the `get` method returns an int.

The Java language doesn't allow this kind of type parametrization yet — although a newer experimental version does — so the declaration doesn't express this requirement and the java compiler doesn't enforce these methods, but we have worked hard to implement them and check them by hand. If you want to implement a new kind of `Column`, you will also have to implement them.

- -

Figure 3.1. The Column interface

```
public Column<TYPE> implements Metadata, RowComparator {

  String name = null;

  Format format = null;

 boolean isInternal();

 boolean isEmpty();

 int getRowCount();

 void clear();

 int capacity();

 boolean isValueUndefined(int index);

 void setValueUndefined(int index,
                        boolean undef);

 String getValueAt(int index);

 void setValueAt(int index,
                 String value)
   throws ParseException;

 void setValueOrNullAt(int index,
                       String value);

 void addValue(String value)
   throws ParseException;

 void addValueOrNull(String value);

 String minValue();

 String maxValue();

 Class getValueClass();
```

- -

```
 void disableNotify();

 void enableNotify();

 void addChangeListener(ChangeListener listener);

 void removeChangeListener(ChangeListener listener);

 int firstValidRow();

 int lastValidRow();

 RowIterator iterator();

 TYPE get(int index);

 void set(int index,
          TYPE element);

 void setExtend(int index,
                TYPE element);

 void add(TYPE element);

 TYPE getMin();

 TYPE getMax();

 TYPE parse(String value)
   throws ParseException;

 String format(TYPE value);

 static Column<TYPE> getColumn(Table t,
                               int index);

 static Column<TYPE> getColumn(Table t,
                               String name);

 static Column<TYPE> findColumn(Table t,
                                String name);
```

```
--
}
```

The `RowComparator` interface allows elements to be compared and sorted without having to know their type. It declares the `compare(int,int)` method for comparing the values of two rows in the column.

The basic interface specifies that column elements can always be manipulated in their textual representations. This is convenient for loading column contents from a textual file or saving them. It also provides a representation for using the columns for labeling visualizations. However, the real value stored in a concrete column can be of any type.

When the concrete type of the column is known, the concrete method, shown as parametrized in the interface, such as `get` or`set`, can be used to access the concrete values of the column elements. These methods are fast since they don't involve any allocation or transformation.

Presentation Format

Transforming from String values to internal values is performed by `java.text.Format` objects. These objects can transform a String into an Object through the `parse` method. They can also transform an Object into a String through the `format` method. This mechanism is used internally by columns and can be extended by users if needed.

For example, date and time information associated to computer files is usually represented as a 32bits or 64bits integer value. Storing file dates in a table or tree structure can be done either as a column of String or as a column of `long`. The later offer two advantages: it is more compact and dates can then be compared using standard integer comparison. Translating from a standard date format such as the Unix date for files can be specified as follows:

Example 3.1. Creating a `LongColumn` Reading Dates in the Unix Format

```
LongColumn dateColumn = new LongColumn("date");
dateColumn.setFormat(new UTCDateFormat());
dateColumn.addValueOrNull("29 Oct 2003 21:47:32");
```

If the values are read as integers, maybe because they have been read directly from a function that return them as a long, the format can still be specified but the value can now be added as a long using the `add` method.

NumberColumns

Very often, Columns contain number of different types such as integers, floats, long or doubles. It is however very convenient to abstract the differences between these representation of numbers and manipulate them all in a similar way. `NumberColumn` is an interface that allows columns holding numeric types to be manipulated in a similar way. For example, when a visualization needs to map a number column to the X screen position, it usually applies a simple linear transformation to the values of the column, regardless of their concrete type. Without the `NumberColumn` abstraction, the mapping would have to consider each concrete number type, which would be painful and error prone.

`NumberColumn` should define methods to get or set a row with all the Java number types. These methods are `get<TYPE>at(int)` and `set<TYPE>At(int,TYPE)`. The implementation should take care not to lose precision when possible. The safest bet when getting values from a NumberColumn is to ask for a double. Only some long values may loose precision when represented as a double. Ideally, this should be checked but isn't currently.

--
When setting a value, it could be specified using its "natural" type to insure that the conversion will be as lossless as possible.

Methods are also defined to get the minimum and maximum values contained in the columns: `get<TYPE>Min()` and `get<TYPE>Max()`. Finally, a `round(double)` method returns the closest representation of a number for the column, as a double. For example, on a column of integers, round(3.5) will return 3.

Dense Columns

Dense columns are based on primitive Java arrays. For now, they use one large array but this may change for several array chunks to avoid copying large blocks when resizing the array. Access time and modification time are constant, not counting the time required to perform notification when the changing the values.

Undefined values are still allowed. For columns containing object-values (non-literal values), undefined rows contain the `null` value. For Column containing literal types, undefined rows are stored as integers in a balanced binary tree.

Literal Columns: `IntColumn`, `LongColumn`, `FloatColumn` and `DoubleColumn`

BooleanColumn

FilterColumn

StringColumn and ObjectColumn

Sparse Columns

Column Depandancies

The values of one column can be the result of a computation made from values taken from other columns. The depandancies can be maintained so that each time an column value is modified, all the depandant columns are updated.

MORE TO COME

Tables

Tables are the simplest data structure of the Infovis toolkit. A simple way to create and populate a table is:

```
Table table = new DefaultTable();
table.setName("Cities");
StringColumn name = StringColumn.findColumn(table, "name");
IntColumn population = IntColumn.findColumn(table, "population");
name.add("New York");
population.add(2000000);
name.add("Los Angeles");
population.add(1000000);
```

--

Figure 3.2. The Table interface

```
 public Tableimplements Metadata, TableModel {
}
```

The Table interface is described in Figure 3.2, “The Table interface”. A Table also manages Metadata and implements the TableModel interface, enabling its use in a standard Java Swing JTable. Table defines methods to add/remove/access columns by name or index. The `getRowCount` method returns the maximum number of rows of each of the columns in the table; Tables don’t maintain their own row counts. The columns contained in a Table may have different row counts. This is not a problem at all considering that columns can contain undefined rows; the row count of a table is indeed the largest row count and columns with less elements are considered as having undefined values at their ends.

The default implementation of the Table interface is the `DefaultTable` class. Concrete implementations of `Tree` and `Graph` derive from this class.

Trees

Figure 3.3. The Tree interface

```
 public Treeimplements Table {
}
```

The Infovis Toolkit implements rooted trees with the interface describe in Figure 3.3, “The Tree interface”. A Tree manages a topology and associated attributes, stored in columns. Creating a file-system like tree can be done using this example program:

```
   Tree tree = new DefaultTree();
   StringColumn name = StringColumn.findColumn(tree, "name");
   LongColumn date = LongColumn.findColumn(tree, "date");
   date.setFormat(new UTCDateFormat());

   name.setValueAt(Tree.ROOT, "/");
   date.setValueAt(Tree.ROOT, "29 Oct 2003 21:47:32");

   int n1 = tree.add(Tree.ROOT); // Child of tree root
   name.setValueAt(n1, ".classpath");
   date.setValueAt(n1, "23 Feb 2004 22:46:30");

   int n2 = tree.add(Tree.ROOT); // 2nd child of tree root
   name.setValueAt(n2, "CVS");
   date.setValueAt(n2, "6 May 2003 23:32:34");
```

```
--
   int n3 = tree.add(n2); // child of node n2
   name.setValueAt(n3, "Entries");
   date.setValueAt(n3, "24 Feb 2003 23:30:13");
```

A Tree is implemented as a Table. Nodes are simply indices. Currently, four internal columns are used to manage the topology. These columns can be read and even modified, but doing so would certainly break the structure so it should be avoided. The internal columns are all `IntColumn`s. Here is their description:
* 0.60+1em

* 0.60+1em "#parent"
  Contain the parent node for each node. The first line in the following examples is equivalent to the next two lines:

```
     int p = tree.getParent(node);

     IntColumn parentColumn = IntColumn.findColumn(tree,↩
Tree.PARENT_COLUMN);
     int p = parentColumn.get(node);
```

  The parent of the node `Tree.ROOT` is `Tree.NIL`. All other valid nodes have non-nil parents.

* 0.60+1em "#child"
  Contains the first child of a node or `Tree.NIL` for a leaf node.

* 0.60+1em "#next"
  Contains the next sibling of nodes, or `Tree.NIL` if the node is the last child of a node.

* 0.60+1em "#last"
  Contains the last child of a node or `Tree.NIL` for a leaf node. The last child of a node could be computed by following the siblings of the first child of a node but Maintaining the last node is faster.

Other internal columns can be maintained automatically by the `DefaultTree` implementation to speedup topological queries such as the depth of a node, its degree (number of children) or to access the children list faster. The depth column is created and maintained using the method `createDepthColumn`. The degree column is created and maintained using the method `createDegreeColumn`.

Graphs

Figure 3.4. The Graph interface

```
 public Graphimplements Table {
}
```

- -
Graphs are composed of two tables: a vertex table and a link table. The first holds topological informations about the vertices and their associated attributes as columns. The second holds the topological information about the links. The Graph interface is described in Figure 3.4, “The Graph interface”.

Metadata

Metadata is information about the data. Columns and tables can contain metadata implemented as an associative map between a key and a value, usually strings.

There are many types of metadata; well known uses include the description of the dataset in term of a title, an author, a creation date, etc. Another use could be describing the processing performed on a table or on a column to create it. The Infovis toolkit uses the metadata for several different purposes that we describe here with no particular order.

Standard Description

It is useful to describe a dataset using standard and well-understood metadata categories. We use the Dublin Core metadata vocabulary, meant to standardize simple and useful description attributes.

MORE TO COME

Aggregation

Aggregation information applies to columns in a Tree. Some columns only define values for the leaf nodes of the tree. For example, when loading the a file directory in a Tree, InfoVis doesn’t provide size information for directories, only for files. However, we know what the file size of a directory means, it is the sum of the file sizes. In that situation, the column containing the file sizes will have an aggregation metadata explaining just that: the file sizes add up with the hierarchy.

Adding up with the hierarchy is a common aggregation method, but others exist as well. First, some column don’t aggregate at all. For example, the file names don’t aggregate, but it turns out they are defined for all the directories so we don’t need to invent a new name. Usually, nominal and categorical information don’t aggregate. If your file have types, such as image or text, the directory cannot simply compute a similar category. We will see later that we could still create an aggregation function in similar cases, but let’s continue with simpler cases.

The InfoVis Toolkit defines seven well understood aggregation types: *additive*, *max*, *min*, *mean*, *concat*, *atleaf* and *none*. We have already discussed the additive type. The "max", "min" and "mean" are similar. "Max" computes the maximum over all the children as the aggregation function. "Min" and "mean" compute the minimum and the mean respectively. As an example, consider file dates in a directory tree. If you are interested by finding the latests work performed, you want to aggregate dates on the maximum date value of each directory.

The "concat" type is for string values and simply specifies that the values will be concatenated into a string with a space between them. Finally, the "atleaf" means that the attributes are only defined at the leaves, not for interior nodes. In that case, any of the numeric aggregation function can be freely applied to the column if it is a numerical column, and the concatenation function can be applied in all cases.

There is one class defined to manage each aggregation function. These classes are useful to compute the aggregated values of a column or checking whether a column belongs to one aggregation category. Furthermore, new aggregation classes can be defined if you need it. In that case, you will need to define how to recognize a column that aggregates using your function and to compute the function. The `AggregationCategory` class is a factory for aggregation functions so you can add yours and it will be correctly applied.

Aggregation information is stored with the `AGGREGATION_TYPE` metadata key. The following constants are defined in `infovis.metadata.AggregationConstants.java`:

- -

- AGGREGATION_TYPE_NONE
- AGGREGATION_TYPE_NONE
- AGGREGATION_TYPE_ATLEAF
- AGGREGATION_TYPE_ADDITIVE
- AGGREGATION_TYPE_MAX
- AGGREGATION_TYPE_MIN
- AGGREGATION_TYPE_MEAN
- AGGREGATION_TYPE_CONCAT

The Aggregation Class

```
public Aggregation implements AggregationConstants {

  public static final short AGGREGATE_NO = 0;

  public static final short AGGREGATE_YES = 1;

  public static final short AGGREGATE_COMPATIBLE = -1;

 public short isAggregating(Column col);

 public Column aggregate(Column src,
                         Tree tree,
                         Column dst);
}
```

Converting to Number Columns

Color Scheme Categories

- -

Chapter 4. Visualizations

Visualizations transform a data structure into a visual representation, allowing exploration and interaction. To create the visual representation, a visualization relies on the topology of the data structure and a set of attributes implemented as columns in the data structure.

More formally, a visualization creates one graphical mark for each table row. This mark is defined by a shape, position, color and transparency. The shape itself can be decomposed into a basic shape, a size and a position. These attributes altogether are called visual attributes.

Table Visualizations

Tree Visualizations

Graph Visualizations

Dynamic Labeling

Fisheye Lenses

- -

Chapter 5. Interaction

Dynamic Filtering

- -

Chapter 6. Readers and Writers

Table Readers and Writers

Tree Readers and Writers

Graph Readers and Writers

- -

Chapter 7. Design Patterns

Abstract Factory

Observer

Visitor

- -

Appendix A. Installing the Infovis Toolkit

Dependencies

* 0.60+1em

* 0.60+1em A java compiler

  Obviously

* 0.60+1em Agile2D [http://www.cs.umd.edu/hcil/agile2d]

  Agile2D is an implementation of `java.awt.Graphics2D` based on the OpenGL accelerated graphics library. The Infovis Toolkit requires this library because is uses some mechanisms profit from hardware acceleration when it is available.

* 0.60+1em GraphViz [http://www.graphviz.org]

  GraphViz is a set of programs developed at AT&T Research for graph drawing. The Infovis Toolkit relies on these programs to layout general graphs.

Index