TreeMap visualization algorithm in Thermostat

Antonio Cesarano

Summary

1. Introduction. 4

2. TreeMap complexity. 4

3. TreeMap structure in Thermostat 5

3.1 The chosen algorithm. 5

3.2 Hierarchical data structure model. 5

3.3 Squarified algorithm implementation. 6

3.3.1 SquarifiedTreeMap. 6

3.3.2 TreeMapBuilder 9

4. Processing and optimizations. 11

4.1 Frames. 12

4.2 Tree processing. 12

5. TreeMap implementation 13

5.1 Performance optimizations 14

5.2 Resizing. 15

6. Usability. 16

6.1 Zooming 16

6.1.1 Zooming implementation 16

6.2 Coloring 17

6.2.1 Coloring implementation. 17

6.2.2 TreeMap coloring. 18

6.3 Communication with other objects 18

7. Integration in Thermostat 19

7.1 Heap dump usage in Tree Maps. 20

8 Conclusions 21

1. Introduction

One of the human-machine interaction field main goal is to improve the communication of information , through data visualization techniques, as much as possible.
In particular, in the recursive data structures field as tree structures, there could be many aspects to underline and each of them has to be supported by a specific visualization method. This bring us to the **TreeMap** visualization technique, which displays hierarchical data structures as nested rectangles: Each branch of the tree is represented as rectangle, which is filled by smaller rectangles representing sub-branches, and each of them is drawn proportionally to its **weight** value.
This kind of visualization is very appreciated because it uses space more efficiently than other layouts, giving a general and compact view of the whole structure.

2. TreeMap complexity

The TreeMap visualization method is as powerful as complex: in order to build the optimal TreeMap for a set of elements *E*, every element has to be represented as a square or, at least, a rectangle and rectangles generated for *E* must have a fixed area (proportional to the weight) and fill an arbitrary canvas.
Thus, squares need to be resized, along *X* or *Y* axis, and moved trying every possible location in the canvas, using every possible sides dimension to find the best configuration. This process makes the algorithm falling into NP problems' category.
However, using some heuristics, is possible to obtain a good approximation for the optimal solution in a reasonable computational time.

3. TreeMap structure in Thermostat

3.1 The chosen algorithm

For Thermostat TreeMap implementation it has been chosen one of the most known algorithms that solves the problem in a polynomial time: the *Squarified* algorithm. In particular, we will refer to the academic article *Squarified Treemaps*, available at the following link: https://graphics.ethz.ch/teaching/scivis_common/Literature/squarifiedTreeMaps.pdf.
The *Squarified* algorithm generates a layout for hierarchical structures, where elements are represented as rectangles that try to approximate squares as much as possible, allowing to the viewer a better visual assessment of items.
Furthermore, **an improvements has been added** in the Thermostat version of the concerned algorithm implementation, using a *Greedy* technique that allows to have an even better layout. The enhancement will be described farther.

3.2 Hierarchical data structure model

The first object to expose is the hierarchical data structure model used as base for this work: the ***TreeMapNode*** object is a recursive structure representing a Tree ADT and it is used to store tree's information and also additional useful information for the TreeMap building process.
This object provides an unique auto-assigned id, a *parent* object, a list of children objects, a weight field and an instance of ***Rectangle2D*** class. It also implements the *Comparable* interface and provides a *quicksort* algorithm implementation to sort in descending way a list of nodes. Last, but not least, this ADT implements a custom interface, called *Decorator*, which supply an implementation for the **Decorator Pattern**.

3.3 Squarified algorithm implementation

Here is presented the proposed implementation for Thermostat's TreeMap algorithm. Following the academic reference article, the *Squarified* algorithm has been developed using a row oriented approach where the main factor is the rectangle's **aspect ratio**.

Before going on, let's define a fundamental object used in *Squarified* algorithm: a **row** is a subset of nodes that will be drawn adjacently as rectangles, along *X* or *Y* axis. Partitioning a set of nodes in rows allows to build a TreeMap view. So, what the *Squarefied* algorithm does it create rows, using as criteria the aspect ratio of rectangles, which area are calculated in proportion to the nodes' weight.

In more details, when a node has to be process, its aspect ratio is evaluated to check if adding it to the current row, it improves the overall aspect ratio. If yes, the node is added to the current row. If not, the row can't be improved and a new row is created. The assumption that no other node can improve the current row is possible thanks to the items order: before starting, they are ordered in descending way.

3.3.1 SquarifiedTreeMap

The Java implementation for *Squarified* algorithm is represented by the ***SquarifiedTreeMap*** class and its main task is to of process rows. Their goodness allows to lay out rectangles with aspect ratio close to 1, so very similar to squares. Below is presented the pseudo code for the *squarefy* function:

```
procedure squarify (list of real children,list of real row,real w)
begin
    real c = head(children);
    if (worst(row, w) ≤ worst(row++[c], w)) then
        squarify(tail(children), row++[c], w)
    else
        layoutrow(row);
        squarify(children,[], width());
    fi
end
```

Table 1: squarefy pseudocode

Each one of the recursive iteration is made assessing the remained items to process, the current row, and the smaller container's side on which draw at the current time (obtainable by *width* function).

The latter is chosen because, processing nodes from biggest to smallest, it is possible to calculate a

squared rectangle on the smallest side. Vice versa, using the biggest one, rectangles will be thin and elongated.

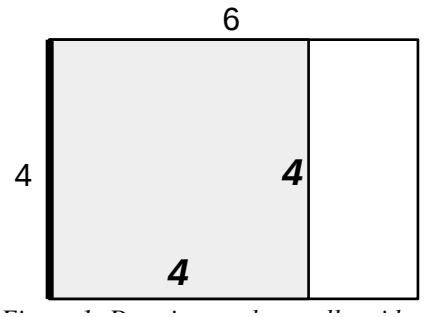
To better understand this, here is presented a graphic example:
let's assume to have a container having size 6 x 4 and a node to represent in it having weight equal to 16. There are two chance to draw the node in the proper container:

1. node's rectangle is drawn on the smaller side:

$$height = side = 4$$

$$width = \frac{weight}{height} = \frac{16}{4} = 4$$

$$aspect\ ratio = \frac{width}{height} = \frac{4}{4} = 1$$



2. node's rectangle is drawn on the bigger side:

$$width = side = 6$$

$$height = \frac{weight}{width} = \frac{16}{6} = 2.7$$

$$aspect\ ratio = \frac{width}{height} = \frac{2.7}{6} = 0.45$$

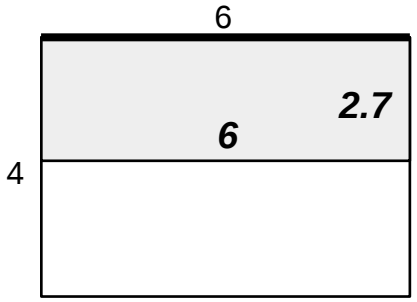


Figure 2: Drawing on the bigger side

As you can see, aspect ratio is improved when the small[illegible]

Others functions are mentioned in the above pseudo code, which are *layoutrow, worst* and *width*, that are respectively used to draw rows, to get the best rectangle's aspect ratio in the row and, again, to get the smallest container's side. For them, another class has been developed, which well be presented later. Let's present now the implementation for *squarefy* function:

Recursion base:

- if no more elements have to be added to the TreeMap just draw the actual row.
- if row is empty and the stack still has elements then the stack's top is popped and it is added to the row; then, *Squarefied* is recursively invoked passing as parameter the updated stack, the updated row and smaller container's side.

Recursion step:

The row is copied and expanded with the element on top of the stack. Now the best aspect ratio for current row is compared with the expanded row's one.

- If the first aspect ratio is closer to 1 than the second one, it means that the row can not be improved, so adding a new rectangle will only decrease the quality of rectangles' aspect ratio. Thus, the row is laid out.
- Vice versa, the algorithm is called recursively using the expanded row, the elements list (without the top element) and the same side of the container's rectangle.

These steps don't deviate from the algorithm's description in the reference paper.

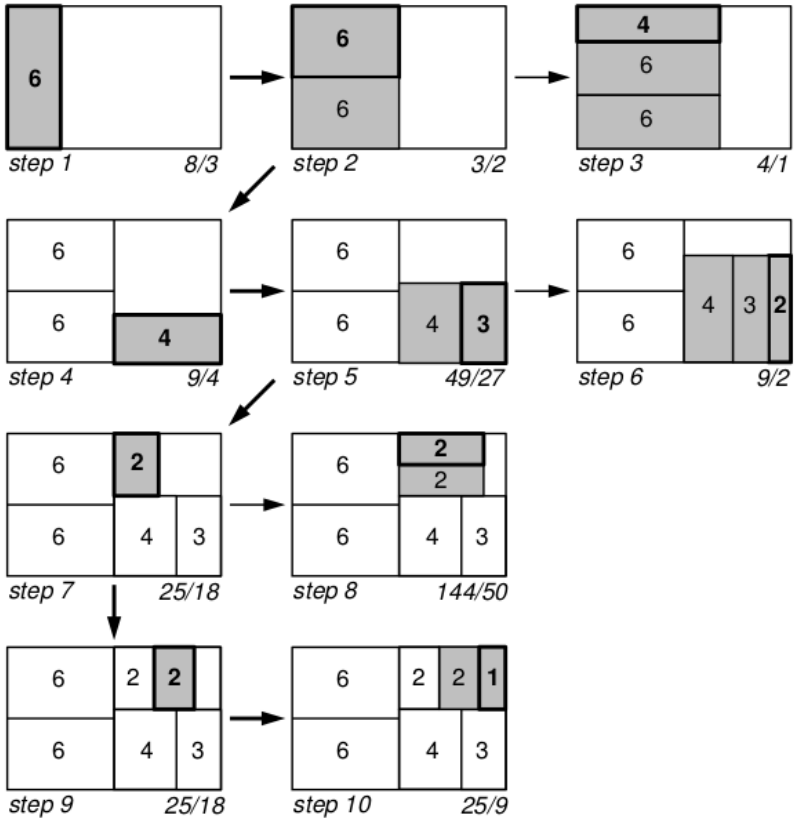


Figure 3: Example of Squarified algorithm execution

3.3.2 TreeMapBuilder

In the previous paragraph some functions have been used by the *squarefy* function but they have not been explicated. This is because they are implemented in the ***TreeMapBuilder*** object, which main task is to support the Squarefied algorithm, managing rows and drawing rectangles inside the container's area. Let's present now an high level overview for what *TreeMapBuilder* provides:

- **availableArea** represents the container in which rectangles are drawn. It is used as reference to know how much space is still available and where rectangles can be placed;
- the **squarifiedNodes** list stores nodes which rectangles have already been calculated;
- **currentRow** keeps the row that is being processed by S*quarifiedTreeMap* object.
- **direction** provides two drawing modes: *top to bottom* and *left to right*. The drawing direction is evaluated according to the *Greedy* approach, which will be explained later. Note that, using this variable involves an **improvement of the final TreeMap view in comparison to the original *Squarified* algorithm.**

The functions *layoutrow, worst* and *width*, previously seen in the *Table 1*, correspond respectively to ***finalizeRow***, ***bestAspectRatio*** and ***getPrincipalSide*** methods of SquarifiedTreeMap. As said above, the main task of TreeMapBuilder is to process rows, so let's analyze how this work is made.

First of all, TreeMapBuilder is **initialized** using the container size to be aware about how much space is available. After that, **prepareData** function needs to be called to order nodes in descending way (using a quicksort algorithm) and to ensure that items' weight are proportioned to the container size.

The weights calculation is the following: $node_i = \frac{container\ area}{\sum node_i} \times node_i$

At this point, the *SquarifiedTreeMap object* starts to process a row, adding elements to TreeMapBuilder's **currentRow** until ***finalizeRow*** is called:

1. The first step allows to this version of Squarified implementation to have an improved layout in

comparison to the original algorithm. Once TreeMapBuilder is aware about the optimum row to draw at a given time, it checks the best **direction** to draw it, comparing the row's best aspect ratio using the smaller and the bigger container's side.

2. Once the drawing direction has been established, it proceeds calculating width and height of rectangle for each node. The calculating process varies according to **direction** field:

   - if the direction is *top to bottom* then the first side to calculate is the rectangle's height;
   - vice versa, if the direction it *left to right* then the first side will be the rectangle's width.

   When the main side has been calculated, it is possible to get the other one, simply dividing the desired area (which corresponds to the node's weight) and the main side.

3. After a rectangle is processed, the location of next rectangle is updated.

4. When the row has been completely processed, the **availableArea** is updated subtracting the sum of last row's rectangles from it, *direction* and *coordinates* are also updated and the row is stored into the **squarifiedNodes** list. Here is shown the difference between the TreeMap generated by the original version of Squarefy (on the left) and the one coming from the above implementation (on the right), indicating each rectangle's aspect ratio.

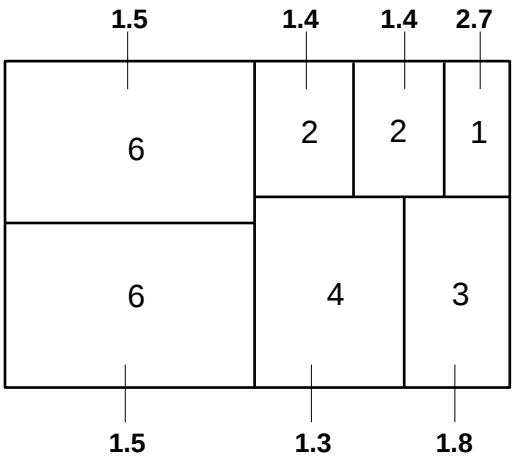


Figure 4: Original squarified TreeMap

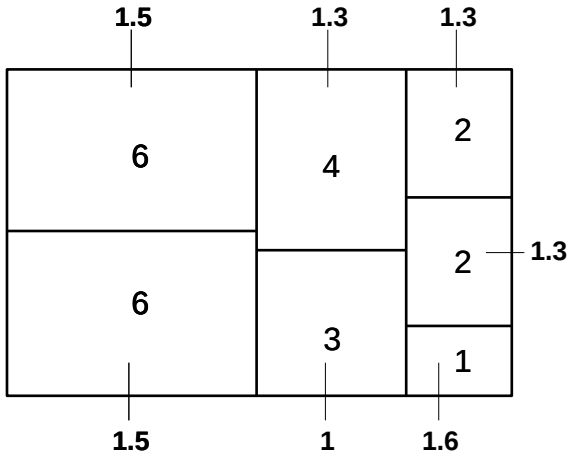


Figure 3: Thermostat TreeMap implementation

As you can see, aspect ratios in the second TreeMap are closer to 1 in comparison to the second TreeMap's aspect ratios. This means that the **Treemap view has been improved**.

4. Processing and optimizations

Build an effective graphic user interface using TreeMaps, at this point, is very easy: to obtain a squarified version of your whole data structure you need to run recursively the TreeMap algorithm on it. This technique is known as *nesting* and its the base task to get a complete TreeMap view of your hierarchical data structure.

The simplicity of nesting operation will not give you a very comprehensible visualization in some situations: when the nodes' size varies strongly it is almost possible to interpret correctly the view, it means you will be able to identify structure's relationships; but, when it doesn't happen, you will get an unclear view, unable to help the viewer in its work. This problem is better explained by the following figure.

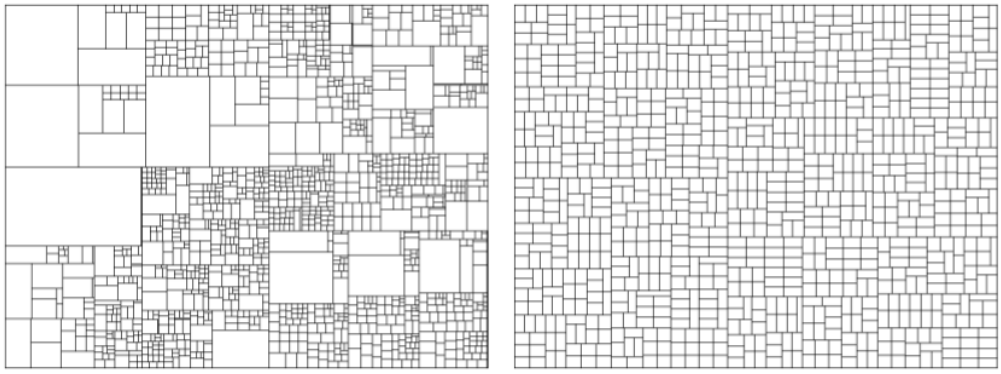


Figure 5: View of a file system (on the left) and a binary tree (on the right).

To solve the presented problem, has been developed a custom component which is able to emphasize relationships between a parent and children, or between sibling nodes, using *frames*.

4.1 Frames

In order to have a clear and an easy readable view, it is necessary to provide a way to distinguish parent nodes from children nodes. The main idea is to frame each item in the hierarchical structure and give to it a subarea where children can be drawn as TreeMap.
In this way, become easy to identify every relationship between nodes: the external frame represents the parent node and the inner rectangles represent its children. If a rectangle doesn't have nested elements then it is a leaf and rectangles placed in the same container are sibling.

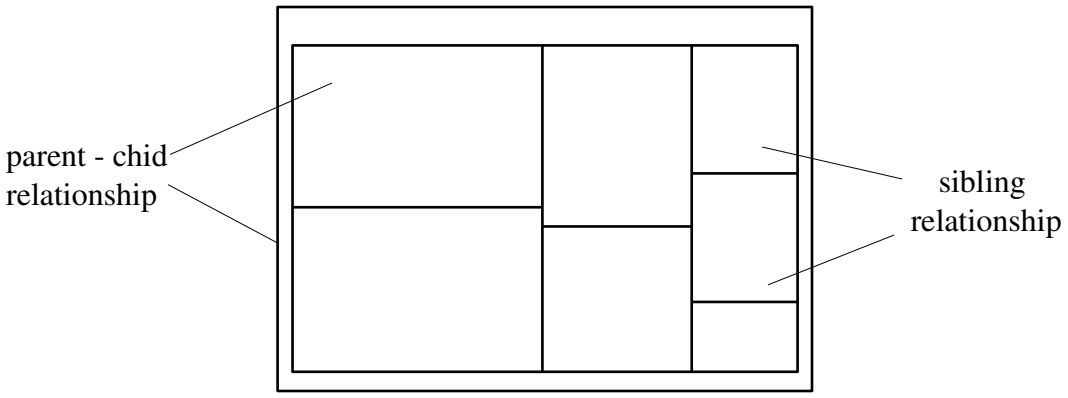


*Figure 5: Representation of a node and its children's TreeMap using **framing**.*

4.2 Tree processing

The first goal for a TreeMap implementation, after the correctness of the TreeMap itself, is the processing time, since it is used to be applied in complex and heavy environments. One of the best practice to achieve this goal is to decouple graphic operations from math calculations, which is the most CPU consuming phase in the TreeMap building process. For this reason an *ad hoc* object has been developed to perform math calculations, called ***TreeProcessor***.
It is responsible for the Squarefy algorithm execution on the whole structure also performing framing operations. This means that when a node is processed, a subarea is calculated using a fixed padding and it will be used as dimension reference to apply Squarified algorithm on its children.
Note that, when a TreeMap's rectangle has been calculated during the process, its reference is also given to the corresponding node.

When the children' TreeMap has been built the process reiterated itself on every child, but only in case they are viewable, i.e. their sides are greater than 1 pixel. Otherwise, the process for that branch is stopped. This trick drastically reduces the computational time, since is useless to process nodes which will be never displayed.

5. TreeMap Implementation

A custom component has been developed to build and manage a TreeMap and to perform some operations on it. The *TreeMapComponent*, that's how it is called, is an extension of Java *JComponent* object and it needs the tree to represent as TreeMap and an arbitrary size to be instantiated.
The *TreeMapComponent* incorporate the *TreeProcessor* object, previously presented, so at its instantiation time, the TreeMap is calculated from the given tree and the drawing process can start.

Let's proceed now describing the TreeMap graphic building process:

1. The first step consists of instantiate the *mainComp* object, which represents the TreeMap's root and its main task is to hold all graphic sub components that will be successively drawn. It is set up the given size and a label that identify which node it represents (located on the top left corner). Note that this component, and all the following components, does not have a layout manager, because the Squarified algorithm yield the exact bounds for each rectangle (location and sides' dimensions).
2. Now the sub tree nested into the root is drawn in a recursive way:
    - first, a rectangle is retrieved from each child, calculated before by *TreeProcessor* object;
    - here a check is made to optimize the process: if the retrieved rectangle (that represents the child size) is smaller than the container size then it is considered drawable, viceversa it will not and the sub tree drawing process will stop for that branch;
    - if drawable, a graphic component is instantiated for each child, using as dimension the rectangle's dimension stored inside the child itself; then, a label containing the node's id is created and located on the upper left corner;
    - the process reiterates recursively using as root each child. In this way, the drawing phase

runs until there is enough space to draw elements.

3. Then, the *TreeMapComponent* object is set up adding the *mainComp object*, which actually holds the whole graphic TreeMap.

Below is presented a simple scheme that illustrates the above steps.

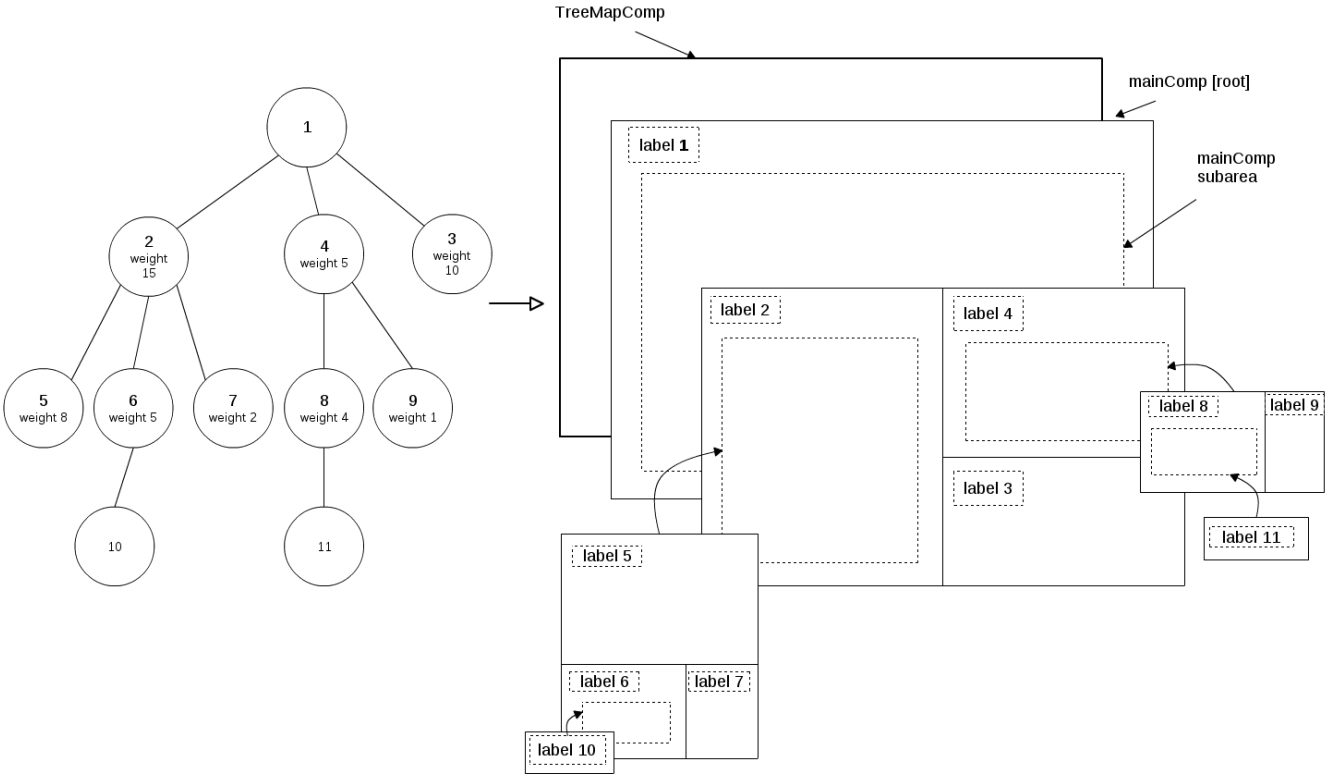


Figure 6: Example of TreeMap building

5.1 Performance Optimizations

The precious process is supported by two custom object developed in order to improve the overall performance. These two **inner classes** of TreeMapComponent, called *Comp* and *Label*, extend respectively JComponent and JLabel classes and their main feature is to implement the java.lang ***Cloneable*** interface, so they can be cloned from other objects. This little trick allow to increase the time of instantiation up to **25%.**

The **Comp** object is a simple graphic component used to represent a node and its look is defined by the *paintComponent* method; in addition to the JComponent class, this object holds two more fields: an instance of TreeMapNode and one for *java.awt.Color*.
The first one is a reference to its corresponding node in the tReeMap, the latter holds a color object and its usage will be explained later.
The **Label** object has nothing more than JLabel except the *clone* method.
These two classes also supply a good modularity to maintain the code, giving as result a very easy and clean way to change and extend TreeMap functions.

5.2 Resizing

To correctly integrate the TreeMapComponent in a UI it needs to adapt itself to its container or to be able to scale up and down. There are two way to develop the resizing behavior of a TreeMap. The first one is to stretch elements in according to the desired dimension, not caring about the view; this approach could be fast but compromises the TreeMap quality. The second one is to draw again the TreeMap, which is more expensive but it keeps the view's quality.
Considering as main goal the view performance, the first choice would be the right one, but it would need a great amount of additional code that, probably, will increase resizing performance not too much. For this reasons, and considering TreeMap building performance optimizations illustrated in the previous paragraph, TreeMap's resizing has been developed reusing existing code, so it is necessary just to use a new size to build the enlarged or reduced TreeMap. In this way, it is also possible to discover new node when enlarging the component and hiding smallest components when reducing.
The TreeMapComponent resizing implementation provides some tricks to avoid useless calculations:

- it holds a `MIN_DRAG_TIME` constraint, which contains the minimum time between two resize function calls;
- it provides the method `isChangedSize()`, which ensures that new size is effectively grater or lesser than the old one.

Combining these expedients, when the TreeMapComponent is manually resized, dragging its corners, the resize operation is not performed each time a ComponentEvent is triggered. This behavior allows to the user to see the the resizing operations just few time in a second, which is enough to make it feel the UI responsiveness, while saving computational operations.

6. Usability features

TreeMap is a very powerful visualization technique, but its usage on very wide structures could result no so effective: depending on the screen available area, it is possible to look at many depth level of your hierarchical structure but, very probably, you will never see leaves, because they are too small. Also, in case your structure results very dense, it becomes harder to keep in mind all relationships you are looking at. For these reasons, two more feature have been added to the Thermostat TreeMap implementation: zooming and coloring.

6.1 Zooming

Let's consider a tree structure having about one million of nodes. Using the TreeMap technique presented above, you will be able to see just few levels, because of the decreasing size that deeper nodes will have. To obviate this problem **zoom features** have been introduced.
In more detail, double clicking a TreeMap's element, using the left mouse button, it will fill the window, showing you deeper nodes. This can be done until a leaf is reached and if you wish to come back, you can do it just double clicking using the right mouse button. It is also possible to immediately go back to the root view double clicking the middle mouse button.

6.1.1 Zooming implementation

The *TreeMapComponent* object implements zooming operations just managing a **stack** of nodes, drawing every time the top element. So, let's see how the stack, called *zoomStack*, is handled:

- zoom in – when a TreeMap element (which is a Comp object) is clicked twice, using the left mouse button, its TreeMapNode field is retrieved and putted on top of the stack;
- zoom out – when a right double click is performed, the top element of the stack is removed;
- zoom full – when the middle mouse button is double clicked, the top of the stack is removed until it remains just one element, which is the root.

Note that, in the TreeMapComponent constructor, the tree's root element is added to the stack and all operations performed on it never remove the root. After the stack is updated, the TreeMapComponent object draw the sub tree nested in the top element of the stack, as explained in the Chapter 5.

This process, however, can not work without defining a click behavior somewhere, so a *MouseListener* object has been added to the Comp class, to make it able to recognize how many clicks have been performed and which mouse button has been clicked.

6.2 Coloring

The first time you approach to a TreeMap could be very muddler, especially with rich data structures: what you see is a huge number of lines that makes rectangles and looking at the most nested elements often you are not able, at least not immediately, to identify sibling or ancestors relationships.
The **coloring** technique has been introduces to workaround these difficulties, in fact is much simpler to distinguish vary colors from counting and searching how many lines a rectangle is apart from another one. In terms of usability, few and quite colors have been chosen to don't tire and disorient the user's eyesight.

6.2.1 Coloring implementation

The main goal was to don't overload the existing structure, so a very simple object has been thought to supply an easy colors handling. The ***ColorManager*** is a utility class which provides a short list of colors and some methods to retrieve them. Before continuing, you should know that this object implements a **Singleton Patter**, just because there is no reason to have multiple instances and to share its status between vary objects and threads.

The ColorManager object has a list of *java.awt.Color* objects and an index, that remembers the last color used. To get a color you can proceed in two way:

- calling `getNextColor()` passing no arguments will give you the next color of the list in according to the index value;;
- calling `getNextColor()` passing a color as argument will give you the successive color of the given one in the list.

This is just what you need to introduce colors in TreeMap.

6.2.2 TreeMap coloring

A small amount of additional code is required to apply coloring. Since every TreeMap element have to be colored, a *Color* field has been introduced into **TreeMapNode** structure, which will be assigned during its traversal by the TreeProcessor object, described in the Paragraph 4.2.
In more detail, when the processing task starts, the first color is retrieved using `getNextColor()` and assigned to the root; from this point, until leaves are reached, the color to assign to each node is obtained by invoking `getNextColor(parent.getColor())`. These statements make parent-children relationships recognizable by color difference and sibling relationships recognizable by colors correspondence. A consequence of this is the chance to individuate nodes at the same level in the structure, because of they have the same color.
Once the tree has been traversed and colors set, they need to be applied to the UI, which is done by the `paintComponent()` method of Comp objects, obtaining the color from their node reference field.

Another feature to make the UI more effective is the Comp's color changes when it is clicked. This gives to the user a better feeling of what he is looking at and is developed by adding a coloring function, called `selectComp()`, to its MouseListener object, illustrated in the Paragraph 6.2. That function just stores a reference to the clicked object; then gives to it a darker color to emphasize the selection event and restores the previous clicked element's color, giving to it a color brighter than its current one, that will correspond to the original color.

6.3 Communication with other objects

In case of usage with other objects, TreeMapComponent provides a way to communicate some events occurring on its structure. This is because it implements the **Observer Pattern**, that allows to objects which implement the *Notifiable* interface to register and unregister themselves and to be notified about item selections and zoom events.
TreeMapComponent object holds a list of Notifiable objects, which can be increased filled by invoking `register(Notifiable observer)` on it. In a similar way it is possible to unregister objects. When an item is selected or a zoom operation is performed, then Notifiable interface's method is invoked on registered objects.

7. Integration in Thermostat

Thermostat provides a wide section for heap analysis which is able to show heap usage diagrams and histograms about its composition. The TreeMap view can greatly improve the reporting work quality, allowing to analyze the heap structure together its elements' dimension.

In order to have an high level of maintainability, the TreeMapComponent object has been wrapped by a structure that looks very similar to the Heap Histogram API.

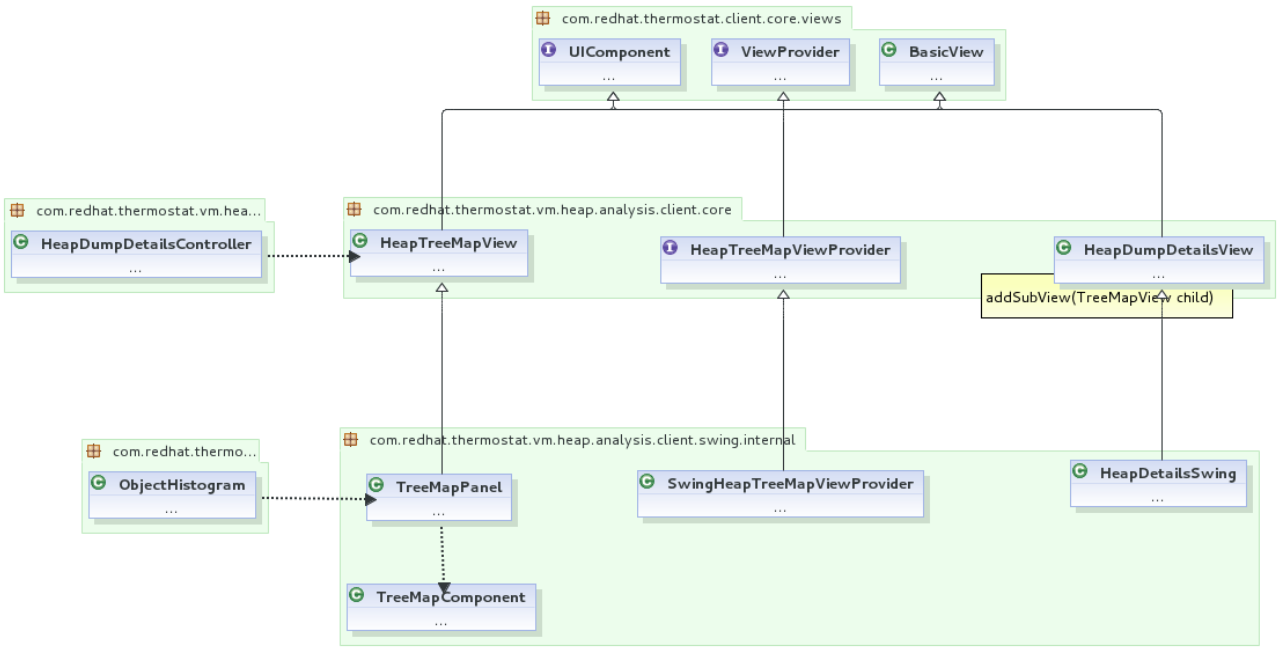


Figure 7: Class diagram of TreeMap API in Thermostat

Using the illustrated structure become very simple to add a tree map view next to the existing ones. In fact, when an heap dump is processed, all declared view are built, so information can easily be shown. The views building process has been modified to supply the option to add a tree map to the view (HeapDumpDetailView class shown in the above diagram), which is called bu the controller object.

7.1 Heap dump usage in Tree Maps

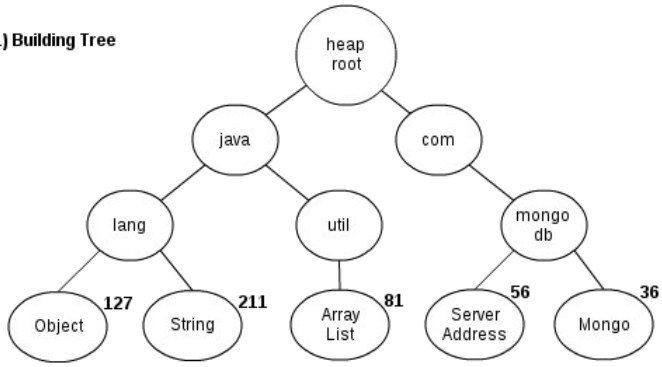
Since the TreeMapComponent class uses to work on TreeMapNode objects and the heap dump to process is a Histrogram class, a conversion between them is needed. This task is performed by the utility class HistrogramToTreeMapNode and it gives a full and compact representation of histograms.

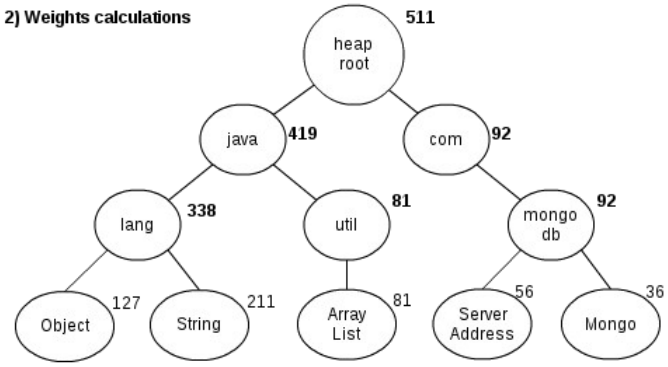
Histograms are list of records containing class name, number of instances and their overall size. Each record's class name is splitted (they have the package dotted format) and a node is created for each element, if it doesn't exist, then a label is given to the node, containing the current class name.

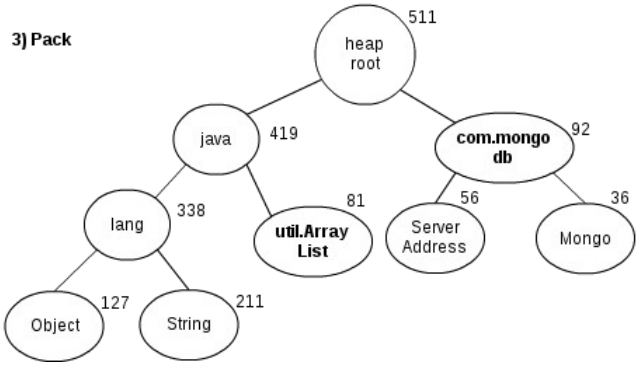
If a leaf class is reached then the overall size is assigned to its respective node object.

After the tree has been created, a post order traversal is executed to fill nodes' weight. In more detail, a node which is not a leaf has as weight value the sum of the weights of its children. Completed the weights calculation, the tree is packed: in order to increase the tree map effectiveness, nodes with only one child collapse into the parent node; the parent weight remains the same and its labels is concatenated to the child's one, preceded by a dot (“.”). The overall process is summarized in the following schema.

Class name	Size
com.mongodb.Mongo	36
com.mongodb.ServerAddress	56
java.lang.Object	127
java.lang.String	211
java.util.ArrayList	81







8. Conclusions

In this paper has been explicated how a TreeMap implementation for Thermostat Heap Dumper has been developed. As you guess, it can be used in a wide range of fields, allowing you a great overview on your whole recursive structure.