Visualization of Software Hierarchy and Dependencies

Nico de Poel

Initiator and supervisor
Prof. Dr. Alexandru C. Telea

Contents

List of Figures

1 Introduction

Compound graphs are often used as an abstract data structure for representing the architecture and design of a software project. Compound graphs can capture both the hierarchy (i.e. the containment relations) amongst the software's components, as well as dependencies between these components (i.e. the association relations). Such a graph can then be visualized using one of many graph layouts, to obtain an overview of the software's structure. The problem is that as the size of a software project grows, it becomes increasingly difficult to create an uncluttered view of its hierarchy and dependencies within a single graph visualization. There are simply no known graph layout algorithms that can display a graph containing hundreds of nodes with thousands of arbitrary relations in a single image, while still clearly conveying both a grand overview of the system and its subtle details.

Therefore, what we are looking for is a method to visualize very large graphs (in the order of thousands of nodes with tens of thousands of edges), that is both easy to comprehend and to navigate by a user. This solution should not be limited to just software diagrams, but has to be generic enough to work for any large graph that consists of both containment and association relations.

2 Proposed solution

The solution we propose is to split up the full graph into a tree hierarchy with all the containment relations, and an association graph. These are then rendered using two separate visualization methods. These visualizations are laid out onto the screen image in a way that is reminiscent of a Monopoly game board (Fig. 1). The border area of the image (also called the *context*) contains a visualization of the tree hierarchy using a known method. The central canvas area (also called the *focus* area) displays a small selection of nodes from the graph and their associations using a simple graph visualization algorithm. Associations between the focus nodes and other parts of the graph are drawn through *links* between the focus and context areas.

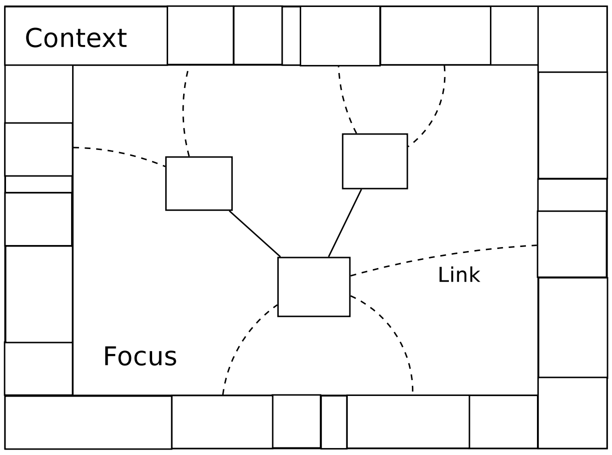

Figure 1: Concept drawing of the Monopoly board layout

The general idea of this setup is that the context area is used to display a broad overview of the graph, while the focus area is used for showing details. A user interacting with this visualization method will be able to click on a graph node within the context area, which will extract the selected node's contents and display it in the focus area. This allows the user to get a closer view of that particular area of the graph and its association with other parts of the graph. This method should provide a highly scalable view of any type of compound graph.

3 Method

Visualization of large graphs according to the Monopoly method consists of three distinct steps: visualization of the tree hierarchy within the context area, visualization of the highlighted subgraph in the focus area, and finally drawing association links between the two. The methods used for these steps are discussed separately.

3.1 Hierarchy visualization

For the visualization of containment relations within the context area, a treemap algorithm is used [4]. Treemaps are very good at visualizing large hierarchies within a small image area, and at giving a clear view of proportions between subtrees. They also have many options for customizing their looks and behavior. To further improve the clarity of the visualization, cushion treemaps are employed. This gives each node a three dimensional appearance with a white highlight, which makes it easier to distinguish the node from its surrounding nodes. A white 1-pixel border is added to each node to increase this distinction even more.

Nodes can be colored individually to convey additional information about the contents of the node. To determine the proportions between subtrees, the size of each leaf node is obtained using a user-customizable *node measure*, and these sizes are added up recursively for each of their ancestor nodes.

Using the entire border area for a single all-encompassing treemap gives rise to a considerable problem. Because most treemap algorithms are designed to fill a rectangular area, the non-rectangular border area would have to be subdivided into several sections, with one treemap per section. The size of these sections would have to be proportional to the size of the subtrees that they contain. This requires a method for partitioning the containment tree into appropriately sized subtrees that optimally fill the available border area. Because the size of the border area may vary at run-time (e.g. because the user resizes the window), this method should be able to dynamically recompute an optimal partitioning.

Because no acceptable simple solution could be found for this problem, the decision was made to instead break up the border area into eight separate rectangular areas, as shown in Figure 2. Each independent area can then be assigned to a *graph tool*, displaying some kind of information about the graph. This can be a treemap as before, but other uses are also possible, such as a legend or a different tree visualization method. This decision effectively generalizes the concept of the context area; the border area can contain *any* kind of information about the graph, not necessarily a hierarchy visualization.

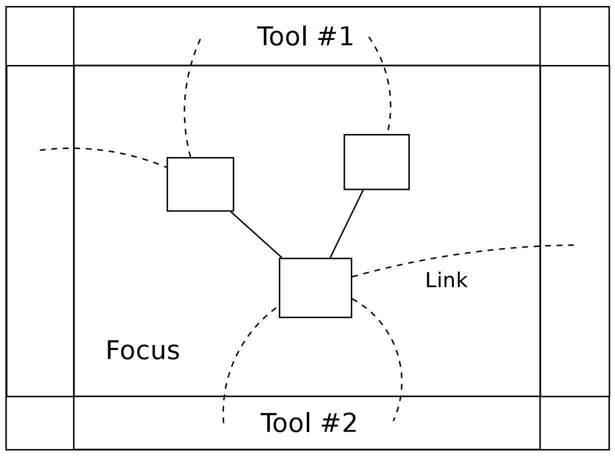

Figure 2: Revised concept of the Monopoly board layout

3.2 Focus graph visualization

For the visualization of the highlighted selection of nodes and edges within the canvas area, a standard boxes-and-lines graph visualization method is used. The focus graph can be laid out using tried and tested algorithms, such as a radial layout, a spring model layout or possibly a circular layout. Selecting the appropriate layout algorithm can be left as an option for the user.

In the context of software diagrams, the focus visualization can be extended to display graphs as UML class diagrams. These can include information about class members and methods, and software metric analysis results.

Graph layouts and visualizations of this kind are commonplace and are generally considered to be very intuitive. This makes them an obvious choice for the focus graph. Also, because the focus graph is generally only a small subset of the complete graph (several tens of nodes at most), most 'normal' graph layout method will be capable of creating an uncluttered view of the focus graph and so they will suffice.

3.3 Links

Besides the association edges connecting nodes within the focus area to each other, there are also edges connecting the focus graph to nodes in the context area. These edges are called *links* and require a special rendering method of their own. Simple straight lines are not good enough for this purpose; the amount of links that need to be drawn for an average graph would quickly cause too much visual clutter to convey any useful information.

This problem of drawing association edges has been solved very elegantly by Danny Holten [1]. Holten uses the hierarchical information available within an existing tree visualization to create control points, which are then used to draw a B-spline that connects the two nodes. This causes edges between nodes with a common ancestry to bend towards each other, effectively 'bundling' edges that move along a similar path. Not only does this decrease the visual clutter, it actually conveys additional information about how associations are concentrated within the graph.

Holten's method does not translate directly to this application however, since it requires that a path between the two nodes already exists. Holten's method is capable of connecting two nodes within a single hierarchy visualization, but it can not be used to connect nodes between two independent visualizations. This meant we had to take Holten's idea and extend it with a new method to create control points using the available hierarchical information.

The method we pioneered is explained visually in Figure 3. When a link needs to be drawn from a focus node to a node within the treemap, first the nodes are collected that make up the path between the treemap's root node and the destination node (red arrows). Of all these nodes, the center points are taken (green dots), and this path is 'unfolded' onto the empty space between the treemap and the focus graph (blue arrows). This unfolding means translating the node positions in the direction perpendicular to the border area's orientation. This results in a series of control points (purple dots), from which a B-spline can be drawn (black dotted curve).

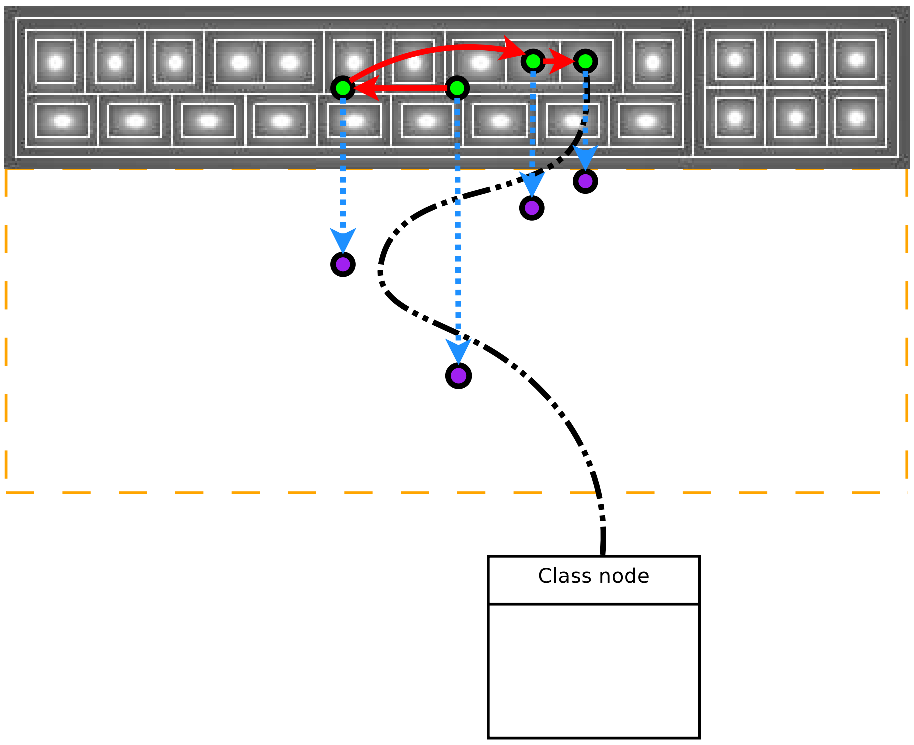

Figure 3: Example of control point unfolding

In practice, several modifications to this unfolding method are necessary to improve the quality of the generated splines. First of all, the root node is always discarded from a path, because every treemap node has the root as ancestor. This prevents the links from all clumping together at a single point.
Secondly, if a node along a path has only one child node, then that child node is omitted. This is similar to Holten's least common ancestor technique [1], and prevents links from curving too strongly towards a single point.
Furthermore, an additional control point is added near the focus node, in the direction of the targeted border area. Without this, all links would start out at different angles and intersect

each other in awkward ways. With this addition, every link starts out in the same direction and has to make a sharper initial bend, which improves the bundling effect.

Several other techniques from Holten's paper have been employed to further improve the quality of the link curves. The strength of the bundling can be adjusted through a customizable parameter. This allows users to switch the links from straight lines (bundling strength = 0) to completely bundled curves (bundling strength = 1), or anywhere in between. On a different conceptual level, this allows the user to switch between low-level details (point-to-point connections between individual nodes), or a high-level overview of the general structure of the graph.

Links can also be color coded, to distinguish between the source and the destination of an association. These colors are interpolated along each spline. This color coding avoids the need to render the associations with arrows or textual annotations, which would have added visual clutter.

Links are only drawn for focus nodes that are (partially) visible within the focus area. Not only does this prevent links to be drawn that originate from outside the screen, it also allows the user to zoom in on a specific part of the focus graph and view only associations for that part.

4 Implementation

The implementation of the Monopoly Graph Viewer application is based on the C++ graph viewer framework for wxWidgets written by Alex Telea. Not only does this framework contain a fast and fully featured OpenGL graph visualization component, it also offers a comprehensive API for building and manipulating graph objects. A separate demonstration application built on this framework also includes functionality for rendering graphs as UML diagrams. This functionality has been extracted from this application and has been included in the Monopoly Graph Viewer through a generic Graph Renderer interface. A high-level schematic overview of the application's design can be found in Appendix B.

Care has been taken throughout the implementation to ensure the application is compatible with both Windows and Linux. To facilitate cross-platform development, premake [5] has been employed to allow platform-specific build scripts to be generated from a single platform independent script file.

The implementation of the treemapping algorithms has been ported to C++ from a Java implementation written by Ben Bederson and Martin Wattenberg [4]. Although there are other implementations available in C/C++, these turned out to be less generic and are more complicated in their use because of the inclusion of custom visualization code. Porting the aforementioned Java implementation to C++ required less work than adapting the existing C/C++ implementations, and yielded a more flexible treemapping solution. This generic treemapping API has been extended to work with graph objects defined through Alex Telea's graph API.

Graph data is loaded into the application from a single large data set. These can be formatted in either Rigi Standard Format (RSF) or Graphviz's DOT format. The application offers several methods for extracting hierarchy and association information from a graph.

In order for the graph visualizations to be customizable by the user, some interactive elements had to be added to the application. Implementing an intricate graphical user interface, however, was deemed to be too risky. Not only does it take a lot of time to design and implement an intuitive GUI, such a GUI is also likely to be outdated as soon as it is finished; users will always find that certain functionality is missing. Also, since the application is merely intended as a proof of concept, building a GUI is outside the scope of the project and not worth the investment.

Consequently, the choice was made to implement customizability by including a scripting language. Lua [3] was chosen as the scripting language because of its easy integrability, its clean and flexible syntax, and because much of the code for the implementation was already available. With this scripting language in place, it becomes very easy to add new functionality to the application and quickly making it available to the user. This makes it well suited for usage in proof-of-concept situations. Documentation for the current scripting interface is available in Appendix A.

5 Evaluation and future work

The proposed solutions work as expected and desired. Separating the hierarchy and association portions of a large graph and laying them out using the Monopoly method succeeds in making both overview and fine details of the graph visible at the same time. The use of edge bundling succesfully adds extra visual information about correlations between graph nodes.

Rendering performance of the application is real-time. Performance was tested on a graph containing 1000 nodes and 4300 edges, with a single squarified treemap on the top border, a selection of 23 nodes and 29 edges on the focus area, and 300 links connecting the focus and context areas (Fig. 4). On a modest computer by today's standards (Athlon 64 3200+, 1 GB RAM, GeForce 7600GT), after initial caching, rendering a single frame takes roughly 70 milliseconds. This means framerates are more than fast enough to allow users to smoothly interact with the graph visualization.

Despite these good results, there is room for improvement. While the general principle of this solution is sound, the solutions for some of the subproblems might not be optimal. Additional experimentation might yield different and arguably better solutions.

For the hierarchy visualization, we have simply assumed that treemaps would provide a good solution. While this turned out to be true, treemaps do have their disadvantages and other visualization methods might give better results. The main disadvantage of treemaps for this application is that child nodes overlap part of their ancestor nodes. A link connecting the focus graph to such an ancestor node in a treemap will appear to be connected to one of the child nodes instead. This makes it difficult for a user to visually pinpoint which nodes are connected by a link.

It is worth exploring how different hierarchy visualization methods affect the behavior of link curves. The curvature of the splines used to visualize links is determined by the location of nodes within the hierarchy visualization. Different visualization methods will therefore also result in different link splines; this is already demonstrated by the variety of treemapping algorithms that have been implemented. With the current treemap algorithms, the link splines can exhibit undesired behavior, making bends that intuitively seem to be unnecessary.

An interesting alternative to the use of treemaps might be the icicle plot. Icicle plots are designed to give an intuitive view of the hierarchy between nodes within a graph, much like treemaps, but

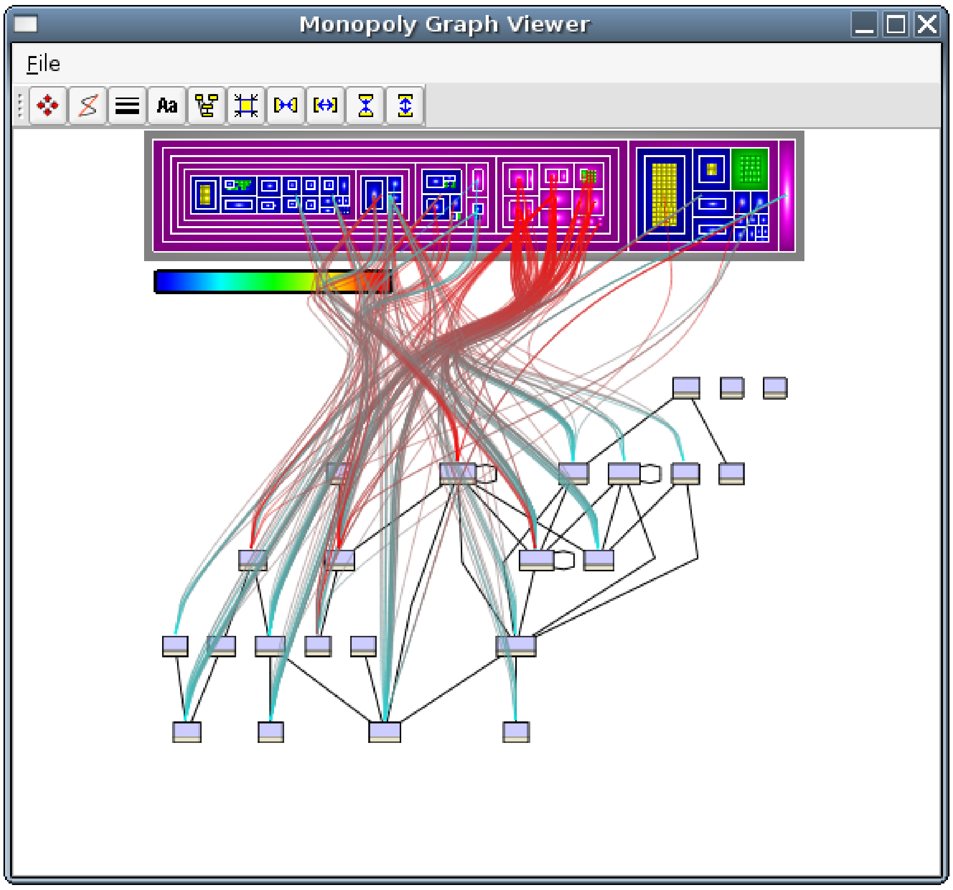

Figure 4: Screenshot of the graph visualization used for performance testing

do so without making nodes overlap each other. The downside to this approach is that icicle plots can not fit as much information within a certain amount of screen space as treemaps can, but the lack of overlapping nodes would make it easier to trace the destination of a link. (Note that the application has been designed to easily allow addition of different hierarchy visualizations, in the shape of a Graph Tool class.)

In his paper, Holten explains a more intricate use of alpha blending for the rendering of edges [1]. Longer curves are rendered using a lower opacity than short curves, which has the effect of emphasizing these short curves, where normally they would be obscured by the longer curves. Our application currently only uses straightforward alpha blending, and might benefit from this more subtle approach.

Additionally, Holten proposes several methods for interacting with bundled edges that our application could make use of as well. One of these ideas involves user selection of a group of edges, after which the non-selected edges are hidden and the selected edges are drawn separately, optionally in straightened out form. This would allow users to delve deeper into the details of a graph than is currently possible.

6 Conclusion

We have designed and implemented a novel way of visualizing large compound graphs. By separately visualizing a graph's containment hierarchies and associations, it is possible to capture both the graph's overall structure and its fine details in a single view.

Treemaps ensure that large hierarchies can be visualized compactly, while remaining clear to the viewer. A traditional boxes-and-lines representation is used for a detailed view on subgraphs and their associations. A modified version of Holten's hierarchical edge bundling technique is used to draw associations focus area to the context area. Using edge bundling causes edges between nodes with similar ancestry to curve towards each other.

These methods, when used together, provides users with a graph visualization that is both easy to comprehend and navigate. The performance of the implementation is well enough to allow for a smooth user experience.

There is still room for improvement in several areas. Implementing other kinds of hierarchy visualization methods, such as the icicle plot, might improve clarity of the overall visualization. Fine-tuning the rendering of edge bundles and user interaction with them, could also result in better usability.

A Lua script documentation

The following section contains an overview of the scripting interface exposed by the application. Lua is a weakly typed language and as such does not have a standard notation to strictly define an API with. For the notation of function signatures in this document, a pseudo-C++ notation has been used, including templates and default parameters. Function arguments are checked at runtime for their correct type whenever possible, but it is still possible to cause undefined behavior by incorrect usage of the script commands.

For more information about the Lua programming language and its usage, please refer to the documentation on the official Lua site [3].

Note that the data types `Graph`, `Node`, `Edge` and `Tool` mentioned below are in fact all Lua's `userdata` type, but have been given a more specific name in the context of this document, for clarity's sake.

Callback functions

The following callback function types are used by various script commands. It is up to the script writer to define a function implementing the correct callback interface and to supply it to the appropriate script command.

Type name	**Arguments**	**Return type**	**Description**
`EdgeFilter`	`(Edge edge)`	`bool`	Given an edge, return whether or not the edge conforms to a filtering condition.
`NodeMeasure`	`(Node node)`	`number`	Given a graph node, returns the size of the node according to an arbitrary measure.
`NodeColorer`	`(Node node)`	`number, number, number`	Given a graph node, return a color for the node in RGB format. The three return values each represent a single color channel value within the range $[0,1]$.
`EdgeColorer`	`(Edge edge)`	`number, number, number, number, number, number`	Given an edge, return colors for each end of the edge in RGB format. The first three return values are the color channels for the 'from' end, the last three are for the 'to' end.
`ClickHandler`	`(Node node)`		Perform an action in response to the user clicking on the given graph node.

Script commands

The following is a list of all available script commands that can be used to configure and manipulate the application. Script commands can be entered in the application's script console, or by loading them from a script file.

Function	Return type	Description
General functions		
`script_load(string filename)`	`boolean`	Load and execute a Lua script from the given filename. Returns `true` on success.
Graph functions		
`graph_new()`	`Graph`	Create a new graph object. Returns a reference to the new graph, or `nil` on failure.
`graph_loadDot(string filename)`	`Graph`	Load a graph from the given filename in Graphviz DOT format. Returns a reference to the loaded graph, or `nil` on failure.
`graph_loadRsf(string filename, table<string> attrNames = {})`	`Graph`	Load a graph from the given filename in Rigi Standard Format. `attrNames` is a list of keywords that should be interpreted as node attributes. Returns a reference to the loaded graph, or `nil` on failure.
`graph_hasNode(Graph graph, Node node)`	`boolean`	Returns whether or not the given graph contains the given node.
`graph_hasEdge(Graph graph, Edge edge)`	`boolean`	Returns whether or not the given graph contains the given edge.
`graph_addNode(Graph graph, Node node)`		Adds the given node to the given graph.
`graph_addEdges(Graph graph, Graph edges)`		Copies all the edges contained in the second graph to the first graph.
`graph_getAttribute( [Graph|Node|Edge] item, string key)`	`string`	Returns the attribute value of the given key for the given graph element. Returns an empty string if the key does not exist.
`graph_getNode(Graph graph, string name)`	`Node`	Returns a reference to the node with the given name from the given graph. If the graph does not contain such a node, returns `nil`.
`graph_getNodesByAttr(Graph graph, string key, string value)`	`Graph`	Returns a new graph with all the nodes from the given graph whose key attribute has the given value.
`graph_getEdges(Graph graph, EdgeFilter filter)`	`Graph`	Returns a new graph with all the edges from the given graph that pass the given edge filter. See the definition of `EdgeFilter` callback.
`graph_getChildren(Graph graph, Node node, EdgeFilter filter, string direction = "out")`	`Graph`	Returns all the children of the given graph node that are found by moving along its edges in the given direction that pass the given edge filter. Accepted values for `direction` are `"in"`, `"out"` and `"both"`.

`graph_extractDAG(Graph graph, Node root, EdgeFilter filter, string direction = "out")`	`Graph`	Extract a Directed Acyclic Graph from the given graph, starting at the given root node, traversing only along edges in the given direction that pass the given edge filter. See `graph_getChildren` for values of `direction`.
`graph_divideEdges(Graph edges, Graph a, Graph b)`	`Graph, Graph`	Divides the given edges into two groups: 1. edges that connect nodes which are both present in graph 'a' 2. edges that connect nodes in graph 'a' with nodes in graph 'b' These edges are returned in a duo of graphs, in the order listed above.
Graph view functions		
`gv_setFocus(Graph graph)`		Shows the given graph in the focus area.
`gv_clearFocus()`		Removes any graph currently shown in the focus area.
`gv_showTool(Tool tool, string border)`		Shows the given graph tool on the given screen border. Accepted values for `border` are `"topleft"`, `"top"`, `"topright"`, `"right"`, `"bottomright"`, `"bottom"`, `"bottomleft"` and `"left"`.
`gv_removeTool(string border)`		Removes any graph tool currently shown on the given screen border. See `gv_showTool` for values of `border`.
`gv_showLinks(Graph edges, string border, EdgeColorer colorer = nil)`		Show the given edges as links between the focus graph and the graph tool positioned on the given border. The edges will be colored according to the given `EdgeColorer` callback function. See `gv_showTool` for values of `border`.
`gv_setBundlingStrength(number strength)`		Set the bundling strength of link curves. `strength` is a value within the range $[0, 1]$ (default value: 0.85). A bundling strength of 0 results in straight-lined links, while a value of 1 results in completely bundled curves.

Treemap Tool functions		
`treemap_new(Graph tree, Node root)`	`Tool`	Creates a new treemap tool for the given graph, with its hierarchy starting at the given root node. Returns a reference to the new tool on success, or `nil` on failure.
`treemap_setLayout(Tool treemap, string layout)`		Sets the treemap layout algorithm for the given treemap tool. Accepted values for `layout` are `"binarytree"`, `"squarified"` and `"strip"`.
`treemap_setMeasure(Tool treemap, NodeMeasure measure)`		Sets the node measuring callback function for the given treemap tool. The treemap will automatically get layed out anew, using the values supplied by the `NodeMeasure` callback to determine the size of each treemap element.
`treemap_setColorer(Tool treemap, NodeColorer colorer)`		Sets the node coloring callback function for the given treemap tool. See the definition of `NodeColorer` callback.
`treemap_setClickHandler(Tool treemap, ClickHandler handler)`		Sets the click handler callback function for the given treemap tool. This callback will be called whenever the user clicks on a node in the treemap visualization. See the definition of `ClickHandler` callback.

B Design diagrams

Figure 5 shows an overview of the principal software modules that make up the Monopoly Graph Viewer application, and the dependencies between them.

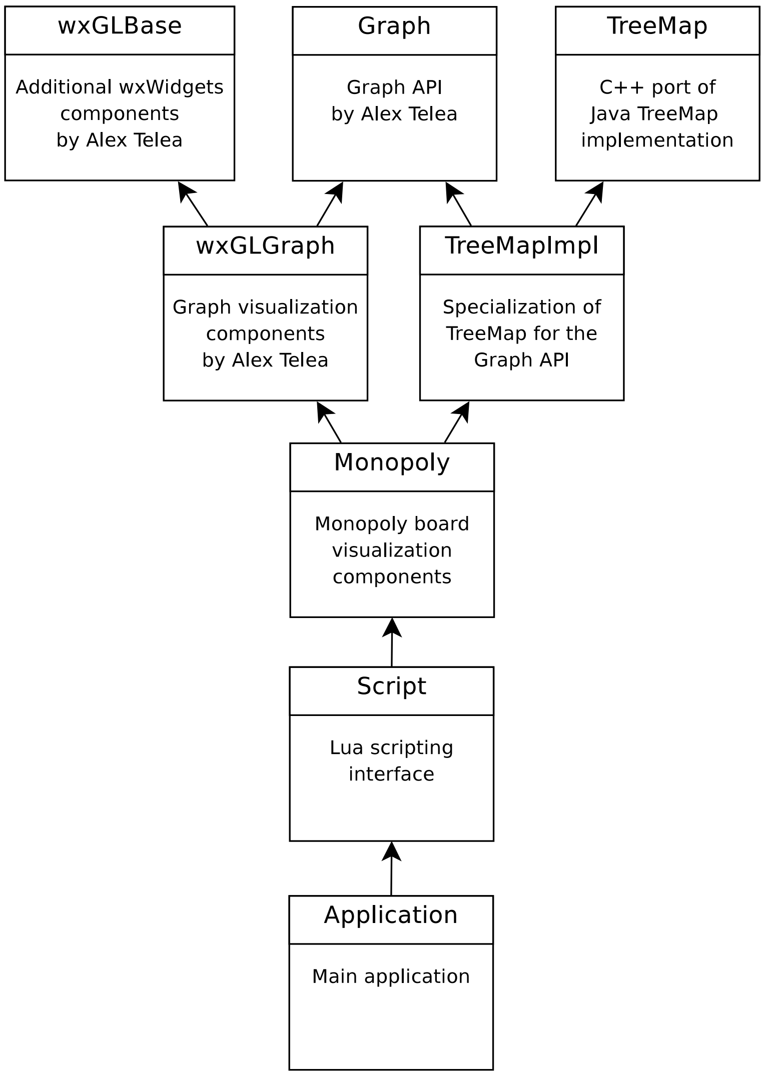
wxGLBase
Additional wxWidgets components by Alex Telea
Graph
Graph API by Alex Telea
TreeMap
C++ port of Java TreeMap implementation
wxGLGraph
Graph visualization components by Alex Telea
TreeMapImpl
Specialization of TreeMap for the Graph API
Monopoly
Monopoly board visualization components
Script
Lua scripting interface
Application
Main application

Figure 5: Software modules and their dependencies

Figure 6 shows a high-level view of the class structure within the Monopoly module.

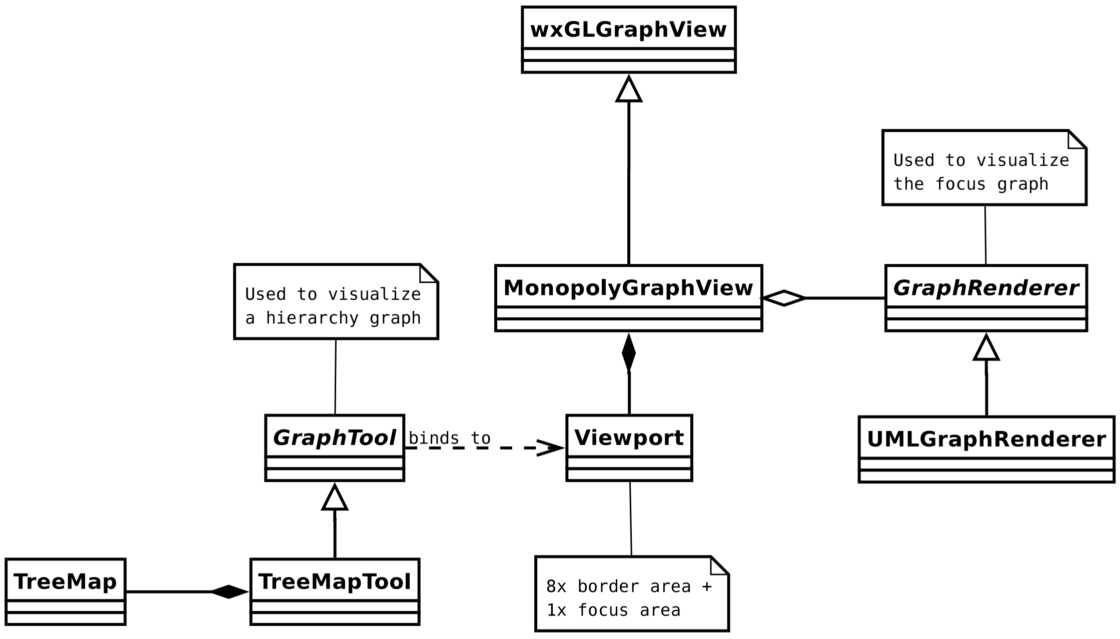

Figure 6: High-level overview of classes making up the Monopoly module

References

[1] Danny Holten *Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data*. IEEE Transactions on Visualization and Computer Graphics (TVCG; Proceedings of Vis/InfoVis 2006), Vol. 12, No. 5, Pages 741 - 748, 2006.

[2] wxWidgets (http://www.wxwidgets.org)

[3] The Programming Language Lua (http://www.lua.org)

[4] Treemaps for space-constrained visualization of hierarchies (http://www.cs.umd.edu/hcil/treemap-history)

[5] premake build script generation (http://premake.sourceforge.net)

[6] Rigi Group (http://www.rigi.cs.uvic.ca)

[7] Graphviz (http://www.graphviz.org)