Telescope: A High-Level Model to Build Dynamic Visualizations

Guillaume Larchevêque Usman Bhatti
Synectique
guillaume.larcheveque@synectique.eu
usman.bhatti@synectique.eu

Nicolas Anquetil Stéphane Ducasse
RMod, Inria Lille Nord Europe
nicolas.anquetil@inria.fr
stephane.ducasse@inria.fr

Abstract

In this paper, we introduce Telescope, a high-level abstract model to create powerful, dynamic visualizations.

Keywords visualization, graphics, stylesheet

1. Introduction

In the domain of software analysis, visualization is a powerful tool to understand software systems and algorithms. Software visualization can be used as tool and technique to explore and analyze software systems to uncover hidden dependencies and discover anomalies.

Several visualizations engines such Roassal [1] exist which can be used to express powerful visualizations. However, it would be interesting to provide an abstraction to the user to express software visualizations and a set of useful recurrent visualization used in software analysis.

Telescope is an engine for efficiently creating meaningful visualizations. The purpose is to help the user concentrate on the problem at hand rather than understanding the nitty-gritty of a drawing library. It does so by exposing to the user few concepts of the domain of visualization: nodes and their contents, layouts, interactions, and update mechanism. For example, the user does not have to worry about the creation of composite or simple nodes, the engine handles it.

Another purpose of Telescope is to express a set of useful recurrent visualizations. Telescope comes with a built-in set of visualizations: Distribution map [2] and Butterfly are two examples. However, if one does not find an existing visualization which corresponds to the needs, adding new visualization or customizing existing ones is easy. The ease comes from the fact that the user does not manipulate the technical concept of drawing but the abstract concepts of a visualization exposed by Telescope. Indeed, Telescope provides a rich API to easily create a visualization to a specific purpose.

Telescope proceeds as follows: The user provides a description of the visualization and Telescope will take charge of all the graphics rendering and updating regardless of the graphics framework connected.

In this paper, we will review two of the existing visualization engines (Roassal for Smalltalk and D3 for Javascript) and expose their lacks to easily build interactive visualizations for software analysis. From this, we deduct requirements for Telescope. We then review the model of Telescope and explain the rationales behind this model. Before concluding, we propose an example of visualization built with Telescope.

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

2. State of the art

Visualization is a powerful tool to reason about data. Several tools provide services to visualize data, such as Graphviz [3] , D3[1], Mondrian [4], and Roassal[2]. We review here Roassal, popular in the Smalltalk ecosystem, and the Javascript library D3.

2.1 D3

D3 is javascript rendering library for producing interactive data visualizations in web browsers. It allows drawing charts and visualization on web pages. For example, one can use it to create an exploration tree for software dependencies or interactive SVG bar chart. It uses JavaScript functions to select Document Object Model (DOM) nodes, create SVG objects, style them, or add interactions to these objects.

Consider the following code in the visualization regarding the initialization SVG canvas taken from the collapsable tree visualization[3] in D3.

```
1 var vis = d3.select("#body").append("svg:svg")
2   .attr("width", w + m[1] + m[3])
3   .attr("height", h + m[0] + m[2])
4   .append("svg:g")
5   .attr("transform", "translate(" + m[3] + "," + m[0] + ")");
```

The script above appends svg tags to the page, sets the visualization height and width, and translated svg canvas to a particular position.

```
1 // Transition nodes to their new position.
2 var nodeUpdate = node.transition()
3   .duration(duration)
4   .attr("transform", function(d) {
5   return "translate(" + d.y + "," + d.x + ")"; });
```

```
1 // Transition exiting nodes to the parent's new position.
2 var nodeExit = node.exit().transition()
```

[1] http://d3js.org/

[2] http://objectprofile.com/Roassal.html

[3] http://mbostock.github.io/d3/talk/20111018/tree.html

```
3   .duration(duration)
4   .attr("transform", function(d) {
5   return "translate(" + source.y + "," + source.x + ")"; })
6   .remove();
```

The two code snippets above pertain to expansion and collapse of the visualization nodes. There are some limitation with the approach adopted by D3:

- D3 does not attempt to hide the fact that one is manipulating SVG and DOM elements. Hence, the user needs to have an understanding of DOM, SVG, and queries to select the nodes.
- Minor adaptation to the visualization will require the user to know the complex details. For example, vertical layout of tree would require the user to adjust transition algorithms manually.
- The notion of group of nodes is missing and each operation has to start with a query to select the nodes and decoration is performed. In the absence of an entity to encapsulate the selection and decoration logic, the code of the visualization becomes verbose and repetitive.
- When nodes are collapsed and expanded, a transition actions occurs and a node is moved to a new point. Ideally, one should define a transition action and reuse it rather than repeating the same logic.

2.2 Roassal

Roassal is a visualization engine made to visualize and interact with arbitrary data, defined in terms of objects and their relationships [Berg14]. Roassal provides services to render shapes (nodes and edges), layouts, and interactions. Moreover, the underlying data objects are available to reason about the data with ease.

Below is a simple visualization for Roassal to create a visualization for class hierarchy for RTShape [Berg14].

```
1  b := RTGraphBuilder new.
2  b nodes
3    color: Color gray;
4    whenOverHighlight: [:each | each withAllSubclasses];
5    @ (RTPopup text: [:each | each superclass name,':', each name]);
6    @ (RTMenuActivable item: 'browse' action: #browse ).
7  b edges
8    connectFrom: \#superclass;
9    useInLayout.
10 b layout tree.
11 b addAll: (RTShape withAllSubclasses).
12 b open
```

The script is concise however there are certain limitations:

- The reuse capability is minimal unless the same script is copied and modified accordingly.
- The notion of node groups is missing. For example, if one wanted to apply an interaction or a layout to a part of nodes, the nodes should be distinguished at the data level. To alleviate this problem, it proposed to use rules that apply to either all the nodes or a group of the nodes in the visualization [Berg14]]. However, due to the absence of groups as a first class entity in the code, the rules to identify groups of nodes tend to be repetitive.
- When adding new nodes to the visualization, all the nodes are need to be redrawn.
- When adding interactions (lines 2-6), one has to use blocks (anonymous functions in Smalltalk). Blocks composition and reuse is non-trivial. Ideally, these interactions should be captured as reusable objects that can be added to the nodes.

Having reviewed two visualization tools and discussed their limitation, we provide a list of requirements for a high-level model to build dynamic visualization to make visualization creation and analysis more effective.

3. Requirements

In the preceding section, we have provided some examples of the existing tools for creating dynamic visualizations. The list is not exhaustive because all these engines provide similar services. These tools, however, are too tied to the underlying drawing framework, therefore exposing complex technical details to the user. The concept of an abstract visualization exposing only necessary services for data visualization is missing from these engines. The foremost objective of the user of a visualization engine is to analyze the data and extract some meaningful patterns without getting bogged down by the nitty-gritty of the drawing engine.

The paper introduces the concept of a visualization specified using a high-level model. The model needs to expose the domain of visualization only e.g. nodes and edges with certain interactions necessary to understand the underlying data. The visualization is a reusable drawing that can be applied in different contexts. For example, a tree visualization should show outgoing branches. Such a visualization should come with a few default operations such as possibility of hiding/showing branches and leaves, inspecting associated data objects, and zooming into a particular branch (drill down/drill up).

Below, we list requirements that a high-level model should fulfill to rapidly create dynamic visualizations.

Independent of rendering framework The model should be independent of a rendering framework so that the technical details are hidden from the user. For example, it should provide a concept of zoom instead of camera which is a technical detail for the user. Different rendering framework could also be used to adapt the visualization to different interfaces (e.g., windowed application or on the web).

Declarative syntax A declarative syntax is often preferable to a procedural one because it allows one to express what the user wishes rather than how to draw it. This goes along with the preceding requirement by hiding the technical details to the user.

Dynamic, updatable visualizations Whereas scripts and builders provide a quick mechanism to construct a visualization to have a first insight into the data, their capability for reuse is limited. The model should differentiate between building the visualization and updating it. Building is concerned with rendering the drawing for the first time, while updating applies changes to a selection of nodes. The requirement not only improves rendering performance but preserves user modifications to the visualization.

Easy to use for novices ... The visualization model should be easy to use, reuse and extend for novices by providing default behavior for common tasks. So, the user only needs to describe the necessary parts pertaining to the desired visualization.

..., yet configurable and extendable for experts The ease of use of the model should still allow experts to rapidly extend and configure existing visualizations. For example, the model can provide a certain number of default actions on the nodes, and accept new actions described by the user.

Definition order does not matter In some declarative visualization DSL's, one has to respect the order in which input is specified. For example, layout is specified after specifying the edges between the nodes. Novice needs to refer to the documentation or, in the worst case, may discover such an order after diving

into the technical details. The visualization model should be uniform in evaluating the input regardless of the order.

4. Model

We have designed a high-level visualization model and an accompanying processor of that model to allow expressing at a high level of abstraction what a visualization must show and how it should interact with the user. We present the main classes of the Telescope model in Figure 1.

The basic assumption of Telescope is that a visualization is composed of elements connected to each other through different possible relationships, i.e. a graph of elements.

TLObject is the superclass of every Telescope class, it has three major parts:

- subclasses of TLDrawable group everything that will be displayed in the visualization. Physically, the drawn objects are instances of TLNode that are rendered through a Shape (not shown in Figure 1) that can be mainly a rectangle, an ellipse, or a label ;
- subclasses of TLAction allow one to modify the visualization or the model, for example by adding/removing nodes or changing their properties ;
- subclasses of TLInteraction relate an action to a drawable and allow one to trigger it with an event (mouse click, mouse over, menu, ...). The interaction is created from the action itself through a set of predefined selectors. A TLInteraction will also have the responsibility to update the visualization after executing the action.

TLAbstractConnector allows one to link a concrete rendering framework to Telescope model. It currently has one concrete subclass to build visualizations with Roassal.

The current model includes other classes that are primarily subclasses of TLVisualization and TLAction (e.g. an action to expand or collapse a node, or to highlight it).

TLConnectableGroups allows to draw all its inner nodes (or sub-groups) according to its given layout. It is also a powerful entity that defines properties (graphical properties such as a color, as well as TLInteractions) for its children. All the nodes of a group "inherit" the properties of their group but can also override some properties. In Telescope, a visualization is a tree whose root is an instance of TLVisualization. Other nodes are either TLConnectableGroups or TLNodes. A TLVisualization is simply a group capable of performing its own rendering; this is why it is associated with the connector.

Another powerful entity is TLStylesheet that allows one to gather in a style object a set of graphical properties that one wished to reuse over different groups and/or nodes. Again, TLStyleSheets are inherited downward the Telescope model tree and the properties can be overridden by any element as needed.

Telescope has a model focused exclusively on the visualization aspect; it voluntarily ignores all the drawing aspects that are deduced from the model but not directly accessible. For example one does not draw directly a rectangle in a visualization but declare a node represented by a rectangle that will be drawn and placed automatically in its context.

The Convention over Configuration is a principle where some default values are defined for everything following domain conventions. It avoids having to define everything when usually you will use always the same parameters. Telescope follows this principle so the definition of a visualization doesn't require any color, shape or layout definition if one uses the default parameters.

4.1 Requirement: Independent from rendering framework

Telescope uses connectors for the rendering. This choice allows one to decouple the model from its rendering in addition to providing easy adaptation with different possible design frameworks. To create a new connector, one needs to subclass TLAbstractConnector and implement its abstract methods (for example, methods to draw the known shapes, or methods to register the interactions and the events that trigger them).

The counterpart is that one cannot customize visualizations with graphic elements that were not generated by Telescope. For example drawing a line in the middle of a visualization is not allowed. Workarounds are available by tapping directly into the underlying graphic framework through the TLConnector which is accessible from the visualization. This practice is, however, discouraged because it couples the visualization to a specific graphics rendering tool.

4.2 Requirement: No scripts

Telescope tries to reduce at a maximum the need for scripts by putting semantics on top of code. Instead of providing a script to paste and then fill with custom code, Telescope offers a structure to fill with this same code but in a subclass of a template. This way the custom code does not require to duplicate anything and will be placed in a named class.

Moreover Telescope offers many pre-defined interactions and visualizations to minimize the user effort.

We chose to avoid builders that are a way to encapsulate scripts because builders are static; once executed, the visualization cannot be adapted to changes. A possible solution would be to re-apply the building script, but it would be hard to take into account all changes made. The specific code put into classes can be seen as script but in that case, the restriction is that this code should just act on the model, not on anything else.

4.3 Requirement: Easy to use, yet configurable

The model of Telescope is designed to be easily extended in order to capitalize on the knowledge of visualization experts. On the other hand, pre-defined visualizations and default behaviour are meant to allow novice users to get good result with a smooth learning curve.

In Telescope only visualization definitions and actions are supposed to be defined by the user. We choose the subclassing principle for that because it is a way to store code with semantic on top of it and then injecting it in the framework using polymorphism. The expert user also knows what he need to implement to make his code work.

Telescope is designed to be used at two levels. The first one is the on-expert level allowing to create visualizations by using the predefined shapes/interactions/styles created in Telescope and applying to the user's needs. The second level is the expert level where the user will define new TLObjects to suit his particular needs.

4.4 Requirement: Declarative syntax

To simplify visualization definition, interactions and style definitions are inherited down along the tree of groups and nodes. Consequently if one wants a set of nodes to have a specific interaction, one only needs to group the nodes together and attach an interaction to the group. For any given element and any given property, the priority is given to the lowest definition of the property from and above that element, so that a property definition at the TLNode level will supersede the same property definition of a parent TLConnectableGroups higher in the tree. Nodes can belong to only one group at a time to enforce, "single inheritance of properties", but contrary to inheritance in OO programming, in Telescope, a

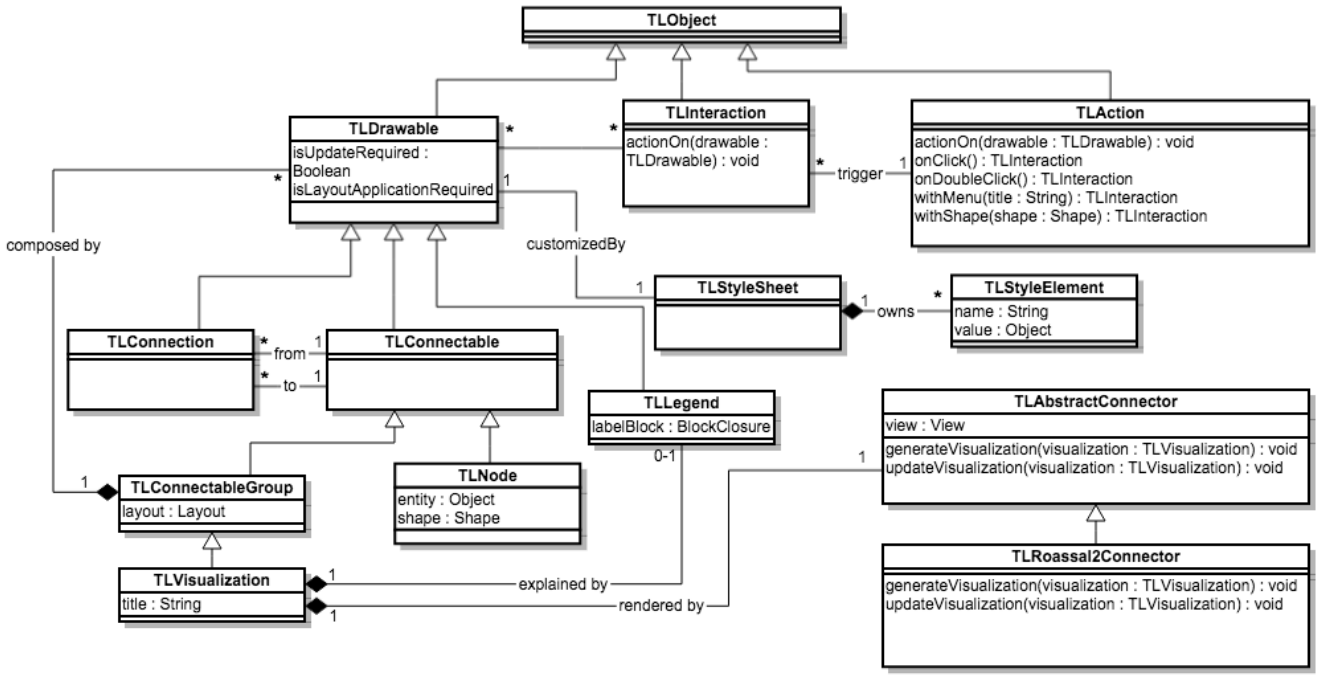

Figure 1. Core Model of Telescope

node can move from one group to another, and thus lose interactions and style properties of the first group to gain those of the second.

5. Example

Here will be presented an example of the full definition of a dynamic visualization.

This visualization will allow to select entities by placing them in two groups. The first group will be entities to keep and the second entities to drop.

In order to accomplish that, all the entities will be placed at the beginning in one group and interactions will allow moving entities from one group to the other.

The first step is to subclass TLVisualization to create a specialized visualization (TLSelectVisualization). Then some methods have to be implemented:

- #defineStructure : here will be created the structure of the visualization (subgroups) and layouts of those subgroups
- #buildVisualization : In this method will be defined interactions and add initial nodes in the model (the order does not matter)

An instance variable with getter/setter to store the collection of entities to sort will also be required.

In the #defineStructure method will be created the two groups (#retained and #rejected):

```
1 TLSelectVisualization>>defineStructrure
2     self nodeLabelPosition: #inside.
3     self > #retained backgroundNodeColor: Color lightgreen.
4     self > #rejected backgroundNodeColor: Color red.
```

First line changes the stylesheet of the visualization to express that all nodes will have a label positioned inside the shape representing each. The #> method is a quick way to define subgroup in Telescope or to access one if it was existing. In this method "retained" and "rejected" groups are subgroups of the visualization (self in this method is the visualization) and we change style of nodes by defining different background colors in the stylesheet of each subgroup (green for the nodes to keep, red for the ones to reject).

In the #buildVisualization method will be added entities at the start of the visualization, here we suppose the visualization already has a collection of entities accessible by the getter #entities .

```
1 TLSelectVisualization>>buildVisualization
2     self > #retained addNodesFromEntities: self entities.
3     self > #retained addInteraction:
4       (TLMoveNodeAction destinationGroup:
5           self > #rejected) onClick.
6     self > #rejected addInteraction:
7       (TLMoveNodeAction destinationGroup:
8           self > #retained) onClick.
```

Then will be defined on each group an interaction with a predefined action that move the node to a destination group. This way the model will be modified by removing the node of its current group and add it to the specified group. Telescope will then apply to the rendering tool each model modification and re-apply any required layout.

The TLSelectVisualization is now fully operational and can be used on any Collection of Objects.

TLSelectVisualization new entities: (1 to: 20); open

Before any interaction, all nodes are in the #retained subgroup and displayed following default layout (Figure 2).

By left clicking on any node, in this initial visualization, the nodes are moved to the #rejected group (see Figure 3) and automatically gain all the properties associated to this new group: they turn red, they gain the group interaction (which would move them

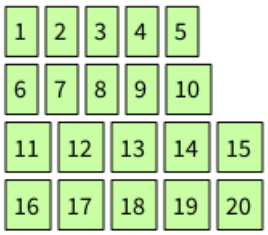

Figure 2. Example visualization, initial state

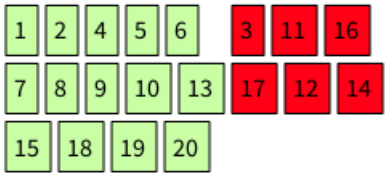

Figure 3. Example visualization, after some interactions

back to the #retain group if one would click again on them), and lose their initial group style and interaction.

6. Conclusion

In the domain of software analysis, visualization is a powerful tool to understand software systems and algorithms. Software visualization can be used as tool and technique to explore and analyse software systems to uncover hidden dependencies and discover anomalies.

Several visualizations engines such Roassal [1] exist which can be used to express powerful visualizations. However, it would be interesting to provide an abstraction to the user to express software visualizations and a set of useful recurrent visualization used in software analysis.

Telescope is an engine for efficiently creating meaningful visualizations. The purpose is to help the user concentrate on the problem at hand rather than understanding the nitty-gritty of a drawing library. It does so by exposing to the user few concepts of the domain of visualization: nodes and their contents, layouts, interactions, and update mechanism. For example, the user does not have to worry about the creation of composite or simple nodes, the engine handles it.

References

[1] A. Bergel, S. Maass, S. Ducasse, and T. Girba. A domain-specific language for visualizing software dependencies as a graph. In *Vissoft'14, NIER Track*, 2014. URL `http://rmod.lille.inria.fr/archives/papers/Berg14a-Vissoft-DomainSpecific.pdf`.

[2] S. Ducasse, T. Gîrba, and R. Wuyts. Object-oriented legacy system trace-based logic testing. In *Proceedings of 10th European Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 35–44. IEEE Computer Society Press, 2006. . URL `http://scg.unibe.ch/archive/papers/Duca06aTestLogtestingCSMR.pdf`.

[3] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng.*, 19(3):214–230, Mar. 1993. ISSN 0098-5589. . URL `http://dx.doi.org/10.1109/32.221135`.

[4] M. Meyer, T. Gîrba, and M. Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization*, SoftVis'06, pages 135–144, New York, NY, USA, 2006. ACM Press. . URL `http://scg.unibe.ch/archive/papers/Meye06aMondrian.pdf`.