UNIVERSITY OF CALGARY

Usability Evaluation of an API for Visualizing and Interacting with Trees

by

Md Zabedul Akbar

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

January, 2013

Abstract

Tree visualization is a branch of information visualization dedicated to visualizing hierarchy within a dataset. It has widespread applications in visualizing Ancestry, the File System, an Organizational Chart, Internet Addressing, and many more. To adequately represent a particular hierarchical dataset, customization of existing tree layouts or even building a novel tree layout is necessary. A review of the existing information visualization toolkits shows their limitations in providing layout customization and task-specific interaction support for trees. Using the existing toolkits, developers often are required to write code from scratch to implement/customize a tree layout if it is not supported by the toolkit. This process requires significant effort from developers both in terms of coding and in understanding the domain.

This thesis presents findings from usability studies of AVIT – an API for Visualizing and Interacting with Trees. AVIT has been developed with the main focus of providing ease of use to software developers in implementing and customizing different tree layouts with task-specific interaction support. To gather developers' feedback, two usability evaluations of AVIT were conducted. The first evaluation identified usability problems in AVIT. Based on the feedback, changes were made to the API and the documentation to improve usability. A second evaluation was conducted on an updated version of AVIT to determine what impact the changes had on the usability experience of the API and identified additional usability issues for future iterations of development. Lessons learned from developing and evaluating AVIT are also discussed to aid future work in this area.

Publications

Materials, ideas and figures in this thesis have been accepted for the following publication:

Hans-Jörg Schulz, Zabedul Akbar, and Frank Maurer. **A Generative Layout Approach for Rooted Tree Drawings.** *In Proceedings of the IEEE Pacific Visualization, Sydney, Australia, February, 2013*

Acknowledgements

I am very fortunate to have received enormous help from many individuals while completing this research. I would like to take this opportunity to thank everyone that helped and supported me with this research.

To my supervisor Dr. Frank Maurer, thanks for your guidance, encouragement and support throughout the process. Thanks for giving me enough freedom to choose my area and you have always been a source of inspiration since I set foot in the ASE lab.

To Hans-Jörg Schulz, thank you very much for visiting all the way from Germany to work with me in this research. I enjoyed working with you a lot.

To Jenifer and Darren, thanks for helping me reviewing the thesis chapters. Without all the feedback and mental support from you guys, it would have been difficult for me to complete this thesis.

To my fellow researchers, Arlo, Seyed, Tulio, Teddy, Rojin, Ted, Uli and Abhi for letting me interrupt you as often as I did to get help from you.

To my family, thanks for helping me and supporting me throughout my entire education.

Table of Contents

List of Tables

List of Figures and Illustrations

List of Symbols, Abbreviations and Nomenclature

Symbol	Definition
API	Application Programming Interface
IDE	Integrated Development Environment
DOM	Document Object Model
DFS	Depth First Search
RQ	Requirement
AVIT	API for Visualizing and Interacting with Trees
SVG	Scalar Vector Graphics

Chapter One: Introduction

This thesis presents the findings from the evaluations of a newly developed API for visualizing and interacting with trees (AVIT). Tree visualization, also known as hierarchy visualization, is a branch of information visualization dedicated to the graphical representation of connected acyclic graphs. It is an area of research that has widespread practical use in visualizing hierarchy in datasets. Some examples include ancestry (family trees), file systems (directory trees), organizational charts, internet addressing, and library catalogues. There are many different tree layouts available, each having pros and cons when visualizing particular types of datasets [5].

Currently, some information visualization toolkits are available that provide support for drawing tree layouts. A review of the existing toolkits (for details see Chapter 2) by the author has shown that they support only a handful of common tree layouts and are not flexible enough in providing customization and interaction support like: using different shapes for the different levels of tree, generating hybrid tree layout (for details see Chapter 2). On one hand, to provide developers with the most common tree layouts, a lot of implementation work is required by a toolkit implementer. On the other hand, it is very hard to fulfill a user's individual needs as each new hierarchical dataset may require a tailored tree layout for its adequate representation.

Also most of the current toolkits have been evaluated to show their performance and expressiveness in generating different types and sizes of tree layout, but limited user studies have been done to evaluate the usability of those toolkits [e.g. 7, 8, 10, 11]. While performance evaluation, code comparison and expressiveness evaluation is important,

those evaluation techniques do not address the learnability and ease of use of those toolkits by target users. This thesis discusses the usability evaluation of AVIT. The main findings from the evaluation are that an interactive demo helps learn the API, and while less code to do more is good but there are difficulties in understanding the underlying tree layout generation concept.

This chapter aims at providing necessary background information for this thesis. Motivations and background will be discussed in Section 1.1, research problems, research questions and research goals are stated in Section 1.2, Section 1.3 and in Section 1.4 respectively. Finally the chapter concludes with an overview of the organization of the remainder of this document.

1.1 Motivation and Background

This section provides details about the background and the motivation for the research. First, a short overview of different tree layouts is provided. Next an overview of existing information visualization toolkits and their limitations in providing tree drawing and interactions support is discussed. Finally, to address the limitation of existing toolkits, a short discussion on the proposed operator-based tree layout generation process [6] is provided.

1.1.1 Tree Layouts

Unlike other plots and charts, tree drawings subsume an entire family of diagrams. There is not THE tree layout but more than 200 variants of them [5]. Despite their diversity, tree visualization techniques can be categorized into three major types: Node-link diagram, Space-filling diagram and Layered diagram.

In a node-link diagram, nodes are distributed in space and are connected by a graphical edge from parents to their children. In a space-filling diagram each node occupies an area and child nodes are “contained” within their parent. A layered diagram signifies the tree structure using layering, adjacency and alignment. A hybrid layout combining node-link and space-filling tree diagram is also possible [27].

Within each type of tree diagrams there are many different variants available [5]. The algorithm by Reingold and Tilford [28], later modified by Walker [29] provides examples of various node-link tree layout techniques. The algorithm produces a classical tree drawing (shown in Figure 1.1 (1)) where the inherent hierarchy of the data is clearly noticeable. Eades [30] proposed a radial algorithm for tree drawing where nodes are placed on concentric circles according to their depth in the tree. The root of the tree lies on the center. The children of the root lie on the smallest inner ring, and their children lie on the second smallest ring, and so on (shown in Figure 1.1 (2)). A radial view is good for representing wide trees with a lot of children nodes as the available drawing space (circumference of the circular layout) increases with the tree’s depth.

The main drawback of node-link diagrams is their inefficient use of screen space, wasting space in the root side of the tree and cluttering the opposite side. On the other hand, space filling techniques such as tree-maps [31] ((showed in Figure 1.1 (4)) make full use of screen space. Tree-maps encode structure using spatial enclosure. Using a tree-map, it is

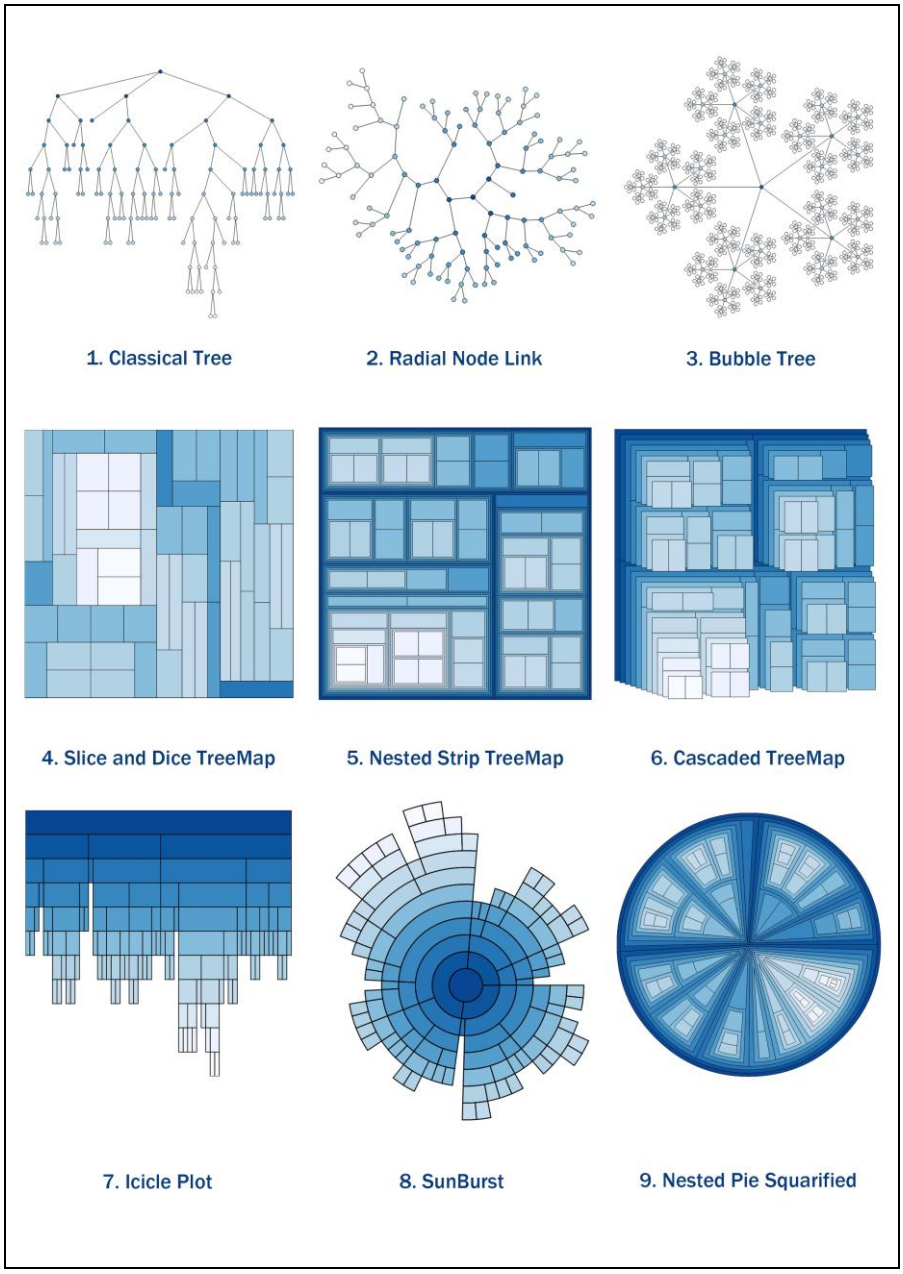


Figure 1.1: Different variants of Tree Layout, (1)-(3) are variants of node-link tree, (4)-(6) and (9) are examples of space-filling tree and (7)-(8) are examples of layered tree drawing.[Generated using AVIT]

easy to get a single view of an entire tree and it is easier to spot large/small nodes. However, one of the important limitations of tree-maps is the difficulty in discerning the hierarchical structure. Variations of tree-maps like nested tree-map and cascaded tree-map layout [32] has been developed to provide insight into the hierarchical structure by adding nesting or cascaded effects ((shown in Figure 1.1 (5, 6)).

Layered tree diagrams preserve the hierarchy structure and are slightly more space efficient than node-link diagrams as there is no explicit edge between parent and child nodes. Also, child levels are layered constrained to parent's extent to restrict the growth in width. Example of layered tree diagram are the Sunburst tree ((shown in Figure 1.1 (8)) and Icicle Plot ((shown in Figure 1.1 (7))

1.1.2 Information Visualization Toolkits

Tree drawings have become a standard type of diagram, which information visualization tools are expected to support. Some examples of existing information visualization toolkits that provide tree drawing supports are The InfoVis Toolkit [11], Protovis [7], prefuse [9], Programmable Tree Drawing Engine [8], D3.js [13], JavaScript InfoVis Toolkit [12] and HiDE [10]. However, existing toolkits are limited in the sense that they only provide support for limited number of tree layouts (for details see Table 1.1: Tree Layout). Also customizing a tree layout e.g., changing the color or shape of a node in a particular level or having a hybrid tree layout, is not well supported in the existing toolkits.

If developers, who want to incorporate a tree visualization component in their application, need a different tree layout that is not supported by the toolkits, they either

have to entirely implement the new visualization component or they require sub classing a pre-existing visualization widget in the toolkit to fulfill their requirements [9]. This process has a steep learning curve and might reduce the developer's productivity.

Furthermore, many of the mentioned toolkits do not provide adequate task-specific interaction support for tree visualizations (for details on task-specific interaction see Section 2.2) which hinders the efficient exploration of the tree visualization. For example, the Programmable Tree Drawing Engine [10] has no interaction support and only generates static tree layout. Protovis [7] provides interaction support like zoom, pan, and drag but is limited in providing task-specific interaction support, e.g. no support for highlighting a sub-tree rooted on a selected node of a tree or showing a path between nodes.

In Table 1.1 a feature-based comparison between The InfoVis Toolkit [11], Prefuse [9], D3.js [13] and JavaScript InfoVis Toolkit [12] has been shown.

Table 1.1: Feature comparison of existing information visualization toolkits

Features	**The InfoVis Toolkit (Released: 2004)**	**Prefuse (Released: 2007)**	**D3.js (Released: 2011)**	**JavaScript InfoVis Toolkit (Released: 2011)**	**HiDE**
Imported data types support	CSV, XML, TQD, TreeML,GraphML Newick, TM3, DOT	CSV, XML TreeML, GraphML	CSV, TSV, XML, JSON	JSON	TSV, XML
Implementation Language	Java	Java	JavaScript, SVG,CSS	JavaScript, SVG	Java, Processing
Tree Layout	Node Link, Tree-map, Icicle	Node Link, Tree-map, Sunbursts	Node Link, Tree-map, Icicle,	Node Link, Tree-map, Icicle,	Tree-map

			Sunbursts, Dendrograms, Indented trees, Circle Packing.	Sunbursts	
Toolkit Design	Polylithic[1]	Polylithic	Polylithic	Polylithic	Polylithic
Interaction	Interactive filter, distortion.	Pan, zoom, drag, rotate, interactive filter, distortion, animation, display node information, highlighting neighbor	Pan, zoom, drag, rotate, animation, display node information	Pan, zoom, drag, animation, display node information	Filter, Animation, display node information
Extensibility for new visualization	Need to write entirely new component or requires sub classing a pre-existing visualization widgets.	Some hybrid or new visualization can be constructed by using existing operators or by introducing new operator.	New visualization can be constructed by using a set of helpers provided by D3.	Need to write entirely new component or requires sub classing a pre-existing visualization widgets.	Visualization can be customized using different values to operator parameters. Currently extensibility to new tree layouts are not supported
Hybrid Tree	Not Supported	Currently not Supported but can be constructed by using existing operators or by introducing a new operator.	Not Supported but can be constructed by using set of helpers provided by D3.	Not Supported	Not Supported
Generating Novel Tree layout	Not Supported	Not Supported	Possible.	Not Supported	Not Supported
Documentation and Supporting materials	JavaDoc No User community	JavaDoc, Incomplete User Manual	User manual with details description and lots of complete	User manual with details description and lots of complete	User manual with details descriptions of the operator,

[1] http://www.infovis-wiki.net/index.php?title=Polylithic_design

			example, Tutorial materials, Active Google group	example, Tutorial materials, Active Google group	video tutorial.
Layout Drawing Approach (See Chapter 2)	Global	Global	Global	Global	Local

It is evident from the toolkit comparison in Table 1.1 that they provide support neither for generating hybrid trees nor for novel tree layouts and one needs to follow a complex procedure to incorporate those layouts into those toolkits. They also are equipped with only a limited number of tree layouts. Also, most of the interaction features on those toolkits are generic like zoom, pan, and drag; task specific interaction for trees is not implemented on the mentioned toolkits.

Considering the diversity of tree diagrams [5], it makes sense to have an API only focusing on drawing different types of tree layouts. If a generic tree drawing approach can be developed, it will be possible to render different tree layouts by following a simple process. It will require less coding effort for the developers to draw and customize different tree layouts through an API that has been developed using the generic tree drawing approach. Also if the APIs are evaluated with users, usability problems of the API and the documentation can be discovered. Findings from the usability studies can help make important design decisions to refine the API and associated documentation and thus will help improving the usability experience of the user of the API.

1.2 Research Problems and Collaboration

From the review of the existing information visualization toolkits, the author of this thesis found that most of the existing toolkits are not only limited in providing flexible customization support for drawing different tree layouts but also lack in providing task-specific interaction support (for details see Chapter 2). To address these limitations, the author of this thesis feels the needs for a new generic tree layout generation approach, which will provide flexibility in layout customization and interaction support with minimal coding effort. Generating hybrid layout and novel layout should also be possible using the new approach.

Developing a new generic approach for tree drawing is itself a very challenging problem to solve that requires deep knowledge about the inner workings of different tree layout algorithms. To address this challenge, the author of this thesis collaborated with Hans-Jörg Schulz, a visiting information visualization researcher from the University of Rostock, Germany in developing *"A generative Layout Approach for Rooted Tree Drawings"* based on the operators [6].

To clarify the author's contribution in the process, the following few paragraphs summarizes the different phases of the research work for this thesis.

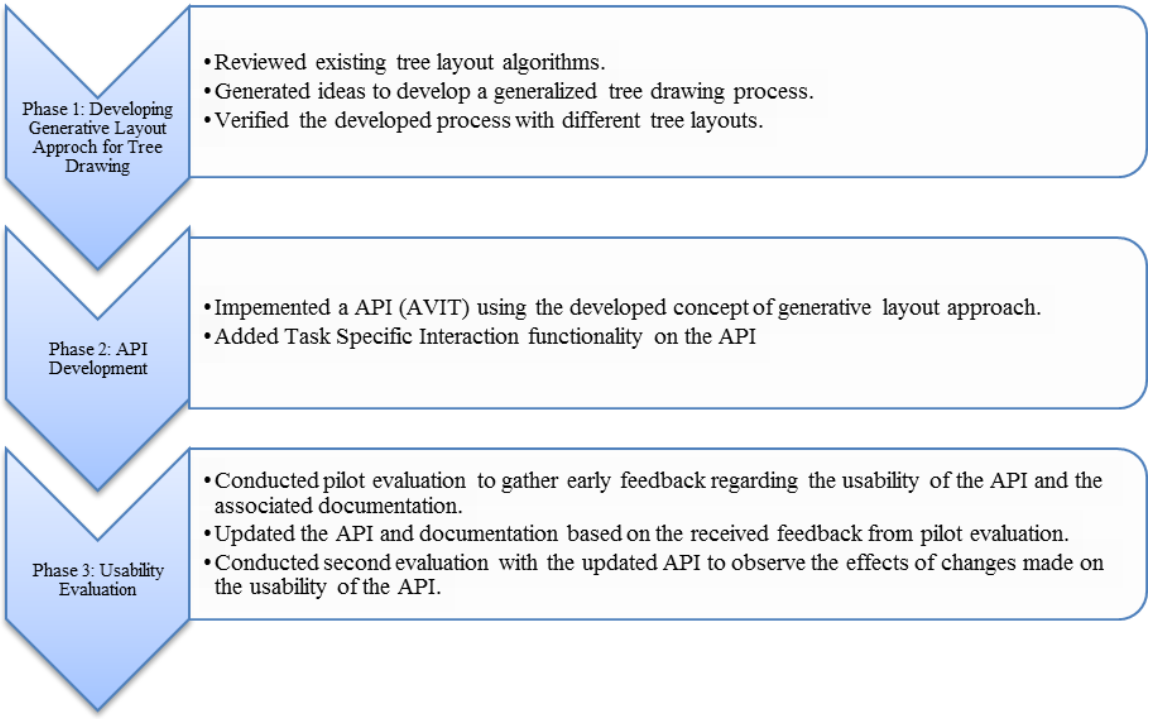


Figure 1.2: Overview of the different phases of the research.

The author collaborated with Schulz, in completing *Phase 1* and *Phase 2* of the research. *Phase 3*: Usability evaluation has been conducted solely by the author of this thesis and also forms the main contribution of this thesis. The following sub-sections provide some details regarding the contribution of the author during different phases of the research.

1.2.1 Phase 1: Developing Generative Layout Approach for Tree Drawing

During the collaboration, Schulz and the author of this thesis examined the tree layout literature to identify the patterns and commonalities in different tree layouts, participated in regular brainstorming sessions and worked closely together in developing a new generic tree layout drawing approach. While the author closely participated in the process, Schulz came up with the main concept of generative tree layout drawing approach using operators [6].

The operator-centric design pattern [49] is well known to information visualization community and has been applied in building different information visualization tools such as the HiDE toolkit [10]. Using an operator-centric design visual data processing is decomposed into a series of composable operators that enables flexible and reconfigurable visual mappings [61]. In the approach proposed by Schulz et al. [6] tree drawing pipeline is decomposed into a series of simple operators and by combining different operators in a particular order different tree layouts can be generated.

During the collaboration, the author acted as a developer who was looking for a web-based API for visualizing and interacting with trees. The author presented the requirements he had for such tree visualization API to Schulz. The author's inexperience with tree visualization helped bring the developer perspective in designing the tree drawing process, hiding the details of the complex tree drawing algorithm.

The author suggested that if the mathematical complexity of the tree drawing algorithm can be hidden behind abstracted methods, it will be easier to understand for a general developer who does not have expertise in tree visualization. Based on frequent discussions with Schulz and feedback provided by the author, several refinements of the operator-based tree generation concept along with its implementation in the API have been made by Schulz.

1.2.2 Phase 2: API Development

In Phase 2, a prototype API (AVIT) has been implemented using the developed tree drawing concept in Phase 1. For AVIT, the author of this thesis has implemented the data loader, renderer and interaction module while the layout pipeline module has been

written by Schulz (for implementation details of the each module of AVIT see Chapter 3).

1.2.3 Phase 3: Usability Evaluation

The target users of the API were the developers who do not have expertise in information visualization but need to use tree visualization components in their web-based application. To understand the developer's reaction using AVIT, it is essential to conduct usability studies with the target developers. Findings from the usability evaluation will help in identifying the usability problems they face while working with the API. Steps can be taken based on the identified issues to refine the API and the documentation materials to better support developer needs.

To address this issue, the author has designed and conducted two separate usability evaluations on different version of the API.

1.3 Research Questions

The usability evaluation of AVIT forms the foundation of this thesis. The main focus of the usability evaluation was to identify the usability problems of AVIT and to gather suggestions to refine the API and its associated documentation materials – so that the usability experience of the API can be improved. While conducting the usability evaluations, the following research questions were of interest:

1. Does the operator-based approach support drawing different tree layouts using a concise specification?
2. Can a tree layout be customized within a minimal amount of time using existing components of the API?

3. Is it possible to generate novel tree layouts using AVIT?
4. How much effort is needed from the developers to learn the operator-based approach of generating tree layouts?
5. How helpful are the documentation and other learning materials for completing a task?
6. How can the interaction features of AVIT be improved?
7. How can the usability experience of AVIT be improved?

The first two research questions can be answered by implementing different example tree layout using the API. The third research question can be answered by drawing usable tree layouts using AVIT which are not available in existing tree layout literatures. The fourth research question is hard to answer considering the time limitation of the usability study in a controlled environment but usability studies will help to identify some common learnability problems with the API. This will be a good starting point to get initial insight about the learnability issues of the API. To address research questions five to seven, iterative evaluations need to be conducted and each evaluation will add some new improvement on the API and its associated documentation to fulfill the identified needs of the developers.

1.4 Research Goals

There are two main goals of this thesis. The first goal of the thesis was to develop an API based on the operator-based approach for tree drawing. This will help to answer the research questions 1 to 3. This research goal has been discussed in Chapter 3, which describes the AVIT, the API created for fulfilling the mentioned first goal.

The second goal of the thesis was to identify the usability problems of the API and refine the API to provide better usability experience. This will help answer research questions four to seven. Two separate usability studies have been conducted by the author to determine the usability of the API and the associated learning resources.

The first study was a preliminary one and was concentrated on gathering early feedback from developers regarding the usability of the developed API. The API and the supported learning materials were updated based on the findings from the first user study. A second usability study has been conducted on the updated version of the API. The main goal of conducting the second usability study was to determine the impact that these changes had on the usability experience and also to determine further usability issues with the API. Details of the first usability study and the second usability study have been discussed in Chapter 4 and Chapter 5 of the thesis, respectively.

1.5 Thesis Structure

This thesis is divided into seven chapters:

Chapter 2 describes existing information visualization toolkits and their support in visualizing trees along with the description of their evaluation approaches. In this chapter, some existing visualization toolkits are described and task-specific interaction classification for tree visualization is provided. The advantages and shortcomings of using these types of toolkits are analyzed from a developers' perspective.

In Chapter 3, a detailed description of the developed API is provided. In this chapter, fundamental design, tree layout generation and interactions features of the API are

explained. Some walkthrough examples for generating and interacting with different tree layouts using the API are given.

Chapter 4 describes the first usability evaluation of the API. This evaluation was conducted to determine a developer's behavior and satisfaction level while using the newly designed API and its associated support materials. Chapter 4 provides the details surrounding the setup of the study, as well as the data analysis process. Findings from the data analysis are used to identify and categorize usability problems with the developed API and the documentation materials. Suggestions to improve the usability of the API are made.

Chapter 5 describes the second evaluation which was conducted on the revised version of the API. Detailed description of the changes made in the API, study setting, data analysis and findings are provided.

Chapter 6 describes the evaluation of AVIT based on the Cognitive Dimension of Notation (CDN) framework. Detailed evaluation of the operator-based notation of AVIT across different dimensions of CDN framework is described.

Chapter 7 closes this thesis by answering the research questions and outlining the contributions to the area of developing usable tree visualization APIs and suggests possible future work for this research area.

1.6 Chapter Summary

In this chapter, motivation and background for this thesis has been discussed. A brief overview explaining various tree layouts, limitations of existing information visualization

toolkits in providing tree drawing and customization support is provided. Research problems, different phases of the research and collaboration have been explained. Finally, research questions to investigate and goals of this thesis were stated.

Chapter Two: Related Work

Information visualization is an important area of research that focuses on making sense of complex abstract data via visual representations. It has numerous applications in Data Mining, Biology, Sociology and many other areas. Despite potential uses in many areas, information visualization applications are difficult to implement as they require a great deal of background knowledge regarding complex layout algorithms, mathematics, and design dynamic graphics before development can begin [9]. Both industry and academy have attempted to address the problem of building complex information visualization applications, and they focus on a tool support approach. The main purpose of building visualization toolkits and frameworks has been making it easier for the user who wants to use visualization component for their application.

Tree visualization is an important branch of information visualization that focuses on visualizing hierarchy in the data. There are more than 250 variants of tree layout [5] that can be generated through a variety of different algorithmic approaches [22]. To provide different tree layout support in existing visualization toolkits, the implementation of a substantial number of different layouts is necessary. As there is no "one size fits all" tree layout that will fulfill the individual needs of the users to adequately represent their hierarchal dataset, it is necessary to provide a customization option in each API to accommodate other layouts using a set of reusable components [9].

This chapter discusses the tree drawing approach, interaction support and toolkit evaluation approach of the existing information visualization toolkits. How the developed AVIT based on the operator-based approach fits in this scenario will also be discussed.

Section 2.1 discusses tree drawing approaches in existing information visualization toolkits, Section 2.2 provides an overview of interaction support in visualization toolkits and Section 2.3 provides an overview of the evaluation approach of current toolkits.

2.1 Tree Drawing in Existing Toolkits

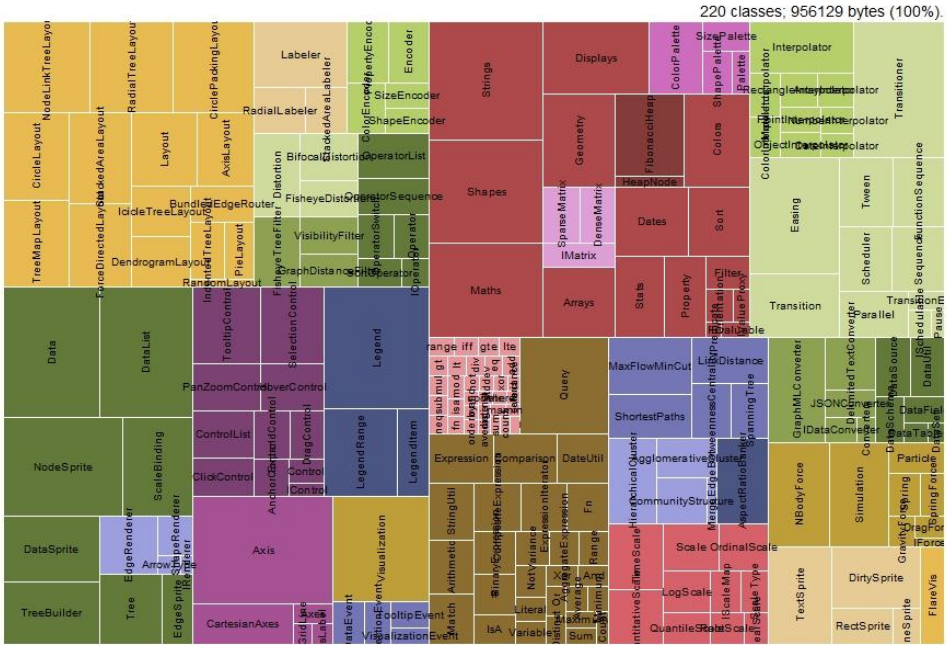
An overview of the available literature on information visualization toolkits done by the author shows that current approaches to generate tree drawings differ mainly in whether they allow a user to specify a tree layout *globally* or *locally*.

2.1.1 Global Tree Layout Approaches

Global tree layout approaches generally apply one layout specification to the entire tree. The use of these approaches is often quite compact, with only a single line of code for the actual generation of the drawing and a few extra lines to adapt drawing styles and colors. These approaches hide most of the complexity of the layout, which make them easy to use. On the other hand, customizing a tree layout according to user needs, such as assigning different shape for nodes in different levels of the tree or arranging sub-trees in a different order based on the topology or attributes of the tree or drawing a hybrid tree layout, is difficult to do using a toolkit that follows a global tree drawing approach.

Also in a global tree layout approach if an unsupported tree layout algorithm is needed, it has to be implemented as an entirely new layout – it cannot be put together as a combination of the existing tree layout functions. Notable examples for global tree layout approaches are the *layout.tree* function in *IBM SPSS Graphics Production Language*, which is based on [33], or the handful of tree layout classes built into *prefuse* [9] and *Protovis* [7].

For example, the Protovis [7] API uses a global layout specification for tree drawing. Figure 2.1 illustrates the output "Squarified tree-map" layout from the Protovis API. A code sample to generate the tree using Protovis [7] is also provided in Figure 2.1(b).



(a) Squarified tree-map layout using Protovis

```
var vis = new pv.Panel()
    .width(860)
    .height(568);

var treemap = vis.add(pv.Layout.Treemap)
    .nodes(nodes)
    .round(true);

treemap.leaf.add(pv.Panel)
    .fillStyle(function(d) color(d).alpha(title(d).match(re) ? 1 : .2))
    .strokeStyle("#fff")
    .lineWidth(1)
    .antialias(false);

treemap.label.add(pv.Label)
    .textStyle(function(d) pv.rgb(0, 0, 0, title(d).match(re) ? 1 : .2));

vis.render();
```

(b) Code to generate the Squarified tree-map layout using Protovis

Figure 2.1: Global tree layout specification. (Tree-map layout using Protovis [9]).

This example visualizes class hierarchy data from the Flare visualization toolkit [41]. Different colors refer to different packages of the Flare visualization toolkit [41] while each area encodes file size. From the code example in Figure 2.1, it can be seen that using Protovis [7] to draw a tree-map layout, first the tree-map layout specification has to be loaded, then styling information like coloring for leaf node and label orientation can be added via a few extra lines of code.

If developers need to customize the above layout in Protovis [7], e.g. making the size of the visualized shape of the packages of the Flare visualization toolkit [41] proportional to their actual file size as described in dataset, or if they want to use a top-down layout rather than traditional tree-map, they either need to implement a new layout algorithm to support their need or they need to extend the existing tree-map layout in Protovis [7]. Both of these approaches to customize the layout require significant effort from developers in terms of coding and understanding the domain.

2.1.2 Local Tree Layout Approaches

Local tree layout approaches on the other hand, provide more flexibility by allowing a user to use different layouts, node ordering and drawing styles for different levels, for leaves, or for different parts of the tree. Naturally, these approaches require specification to govern all the individual aspects of different local tree layouts, as well as to select the sub-trees on which to apply them. Yet this allows the user to influence the resulting layout on a more fine-grained level than do the global approaches do, which permits much more opportunity to customize a layout by mixing the ones that are provided.

Examples of local tree layout approaches are the Tree Visualization Language[2] with its XPath-based expressions to configure the underlying TreVis Framework [34], the Hierarchical Visualization Expression notation [10], and the Programmable Tree Drawing Engine [8].

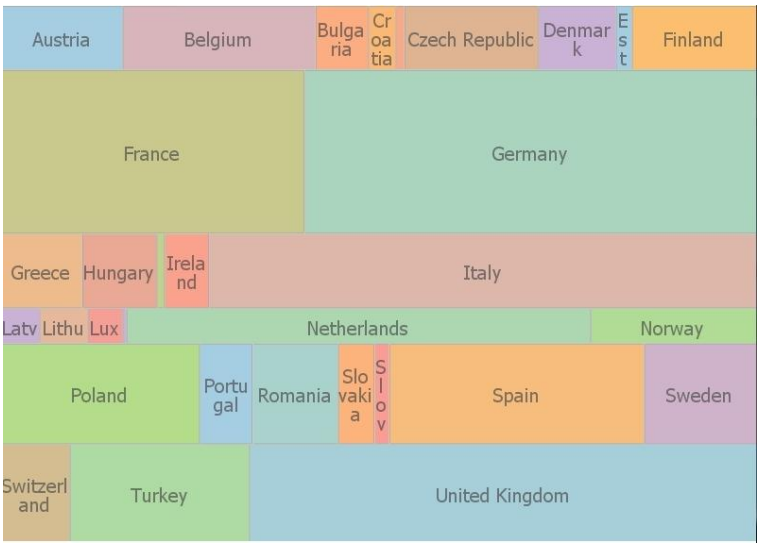
An example of the Cartesian Space Filling layout (tree-map) using the Hierarchical Visualization Expression notation [10] can be seen in Figure 2.2. HiVE [10] describes hierarchical visualizations in which variable values are used to condition the data above them in the hierarchy.

Expression: *sHier(/,$country); sOrder(/,HIER); sSize(/,FX); sColor(/,HIER); sLayout(/,SF);*

Figure 2.2: Cartesian Space Filling layout using HiDE toolkit and the corresponding HiVE expression.

[2] http://sape.inf.usi.ch/tools/trevil

The example layout in Figure 2.2 – visualizes data from Eurostats, "Consumption of energy by country over time" using the HiDE toolkit [50]. From Figure 2.2, it can be seen that a HiVE expression follows an expression-based approach for drawing and manipulating the tree layout where each expression contains a number of parameters. By assigning different variable values in the parameter of the expression "*sName(path, var1, var 2)",* different customization can be made in the layout (for details of the expression see [10]).



Expression: ***sHier****(/,$country);* ***sOrder****(/,HIER);* ***sSize****(/,$consump);*

sColor*(/,HIER);* ***sLayout****(/,SF);*

Figure 2.3: Cartesian Space Filling layout drawn using HiDE toolkit and the corresponding HiVE expression.

In Figure 2.2 the size of every node is fixed: *sSize(/,**FX**);*, if we want to assign a proportional size to each node, based on their consumption rate then we just change the parameter in ***sSize(/,FX);*** *to* ***sSize(/,$consump);*** where *$consum* is a variable containing

values of consumption rate for each country. After making the changes it will generate the layout shown in Figure 2.3.

The above example shows that using *local tree layout approaches*, layout customization can be made by minimal changes in the code using already available functions of the toolkit, but one needs to spend a significant amount of time understanding the functionality of each operator. Also the above-mentioned HiVE expression [10] and its associated HiDE toolkit [50] can only draw space-filling layouts like tree-map. Other types of tree layout are not supported yet.

It is notable that both global and local tree drawing approaches break down the tree layout along the data – applying the layout either to all of the data or to parts of it. The layout process itself is, if at all, only subdivided in the two steps of the actual node placement (e.g., radial, Slice and Dice, indentation) and the definition of a drawing style (e.g., color-coding, labels, node shapes). A further breakdown of the layout process into more fine-grained steps will be necessary to gain more flexibility in specifying tree layouts, so that an even wider range of tree drawings can be generated by mere specification.

To counter this problem, an operator-centric generative tree drawing approach has been recently proposed by Schulz at el. [6]. Schulz observed that despite the large variety of tree drawings, many of the most common ones follow a similar overall process [6]. Using this operator-based model of tree layout, it becomes possible for a user to generate

countless different tree drawings[3] by plugging different operators or operator sequences in the layout pipeline. This concept was later used by the author as a cornerstone of building the API for Visualizing and Interacting with trees (AVIT).

2.2 Interaction

Interaction is an essential part of information visualization through which user's understanding of the dataset is changed or enhanced [37]. With suitable interaction support some limitations of the static representation can be overcome and the "cognition of a user can be further amplified" (e.g., [38, 39]). Implemented interaction features that are common on current information visualization toolkits are zoom, pan, drag, search, select, distortion and filter [7, 9, 10, 11, 41]. While these interaction techniques are useful for revealing some property of the dataset, task-specific interaction for a specific visualization type, like tree, will help reveal important characteristics of the dataset.

Current information visualization toolkits have limited task-specific interaction support for tree visualizations and one needs to follow a complex procedure to incorporate new interactions in those toolkits. For a comparison of the list of interactions supported by the current toolkits see Table 1.1. Information visualization toolkits, in general, cover a broader area of visualizations techniques that are not limited to only tree visualization but also support graphs, scatter plots and other types of visualizations. Interaction techniques implemented in these toolkits mainly focus on covering a broad range of visualizations in general and focus less on interaction specific to a particular domain of visualization. For example, the prefuse toolkit [9], provides zoom, pan, drag, filter, distortion, and rotate

[3] http://tinyurl.com/operatordemo

interaction techniques (for details see Table 1.1), which can be used equally over any type of visualization, such as tree, graph or scatter plots. However, task-specific interactions for a tree like comparing sub-trees or showing a path between the root node and a selected node, are not currently supported in prefuse [9].

Even toolkits specific to tree visualization, either produce a static layout with no interaction support or provide specialized, domain-specific interaction techniques to visualize the dataset of a specific domain [40]. For example, the Programmable Tree Drawing Engine [10] which is a python based toolkit for visualizing trees, only produces static tree layouts. PhyloWidget [35] is a program for viewing, editing, and publishing phylogenetic trees. It supports zoom, pan, search, node edit, labeling, sorting, removing elbow (nodes with one parent and one child) and random mutation interactions that are useful in exploring phylogenetic trees. However, PhyloWidGet [35] lacks interaction support for other task-specific interactions such as tree comparison or sub-tree highlighting.

It has been observed from the taxonomy of tasks for tree visualization [40], graph visualization [36] and network evolution [56] that tasks can be categorized into four major groups: topology-based tasks, attribute-based tasks, browsing tasks, and the overview tasks. In the following subsections each task group focusing on tree visualization will be discussed.

2.2.1 Topology Based Tasks

Topology based tasks are tasks in which the user needs to identify global structures or patterns of interest in their data or among specific entities. Topology based tasks for trees as described in [40] are listed below:

- *Overall characteristics:* Identifying the size and depth of the tree. Which branch is the deepest? Is there any variation in depth between sub-trees?
- *Path:* Showing the path of a node from the root or path between selected nodes.
- *Local relatives:* Identifying the children, sibling or cousins of a node.
- *Distant relatives:* Finding a node's ancestor or descendent, finding the common ancestor of any two nodes of the tree.
- *Filtering by level*: Displaying only the first level of the tree or only 4 levels down or show the tree removing all the leaf nodes.
- *Counting nodes:* Which branch contains the largest number of nodes? Or which branch/node has the maximum number of leaves?
- *Comparison:* comparing similarity between sub-trees, identifying the difference between multiple trees in a structure or number of nodes.

2.2.2 Attribute Based Tasks

Attribute based tasks are tasks in which the user need to discover information based on the node or edge values.

- View detailed information about a node or edge.
- Search for a particular node/edge having a specific attribute value.
- Find all the nodes or edges within a range of values.

- Sort nodes based on attribute value.

2.2.3 Browsing Tasks

Browsing tasks help in exploring the tree layout.

- Locating a node, knowing its path.
- Going back to a previously visited node.
- Explore the tree by performing a series of up and down movements within the tree.

2.2.4 Overview Tasks

Overview tasks provide summary information about a tree layout.

- Size of the tree e.g. depth, number of total nodes, total leaves.
- Node with maximum or minimum number of children.

The list of tasks mentioned-above is not a complete list but covers a broad area of the most common tasks with trees [40].

It was observed throughout the course of this research that some task-specific interactions for tree visualization are not suitable for all types of tree layout. For example, topology-based tasks like showing the path between a node and the root of the tree or finding a common ancestor do not make sense for interacting with a space-filling tree layout like tree-maps where only the leaf nodes can be seen.

Different interaction techniques that are suitable for exploring a particular type of tree layout might not be useful for a different type of tree layout. If interaction techniques using the above task-taxonomy can be implemented in an API, it will provide much

freedom to the developer to select those interaction features that will better suit their requirements for a particular type of tree layout.

In the developed API for Visualizing and Interacting with Trees (AVIT), a specialized interaction layer was constructed allowing task specific interactions for trees (for details of the implemented interactions see Chapter 3).

2.3 Evaluating Visualization Toolkits

This section provides an overview of the toolkit evaluation approaches followed by their designer for existing information visualization toolkits.

After conducting a review of the existing information visualization toolkits, the author identified the following evaluation approaches which were commonly used in evaluating the existing information visualization toolkits by their designer:

- Performance evaluation
- Application Coverage
- Heuristic-based evaluation.
- Usability evaluation
- User adoption
- Longitudinal user studies.

Performance evaluation mainly focuses on evaluating the load time and memory uses for different types of visualization layouts with small to large datasets using the toolkit. For example Protovis [7] and D3.js [13] have evaluated the load time and memory uses of different visualization layouts with the dataset ranging from 10 to 100,000 points using

profiling tools. Performance evaluation provides valuable information about how efficiently the system processes different datasets and provides some insight about the scalability of the system.

Application coverage mainly tests the expressiveness of the toolkit. It is done by implementing existing visualizations or crafting a novel design using the toolkits by their toolkit designer. It provides valuable information regarding generating different layouts using the toolkits. All the information visualization toolkits reviewed by the author used this evaluation approach by building various example visualization applications using the toolkit. For example, using the prefuse toolkit [9] the toolkit designer built a novel hierarchy browser called degree-of-interest trees. The designer of the Flare Toolkit [41], D3.js [13], Polaris [47] and Protovis [7] also built various example applications using the toolkit to evaluate the expressiveness of those toolkits.

While expressiveness evaluation is helpful, it does not evaluate the difficulty a user might face in learning and using the toolkit. Visualization toolkits are typically built by the visualization researchers who have years of experience and a deep understanding of the area. When visualizations start to be used by a broader set of developers, it is better to conduct evaluations with developers, who do not have much background in visualization, in order to gather their feedback on the understandability and ease of use of those toolkits. Having expertise in visualization, some of these usability issues might not be noticeable to the visualization researcher if they only conduct a self-evaluation of the toolkit.

Heuristic-based evaluation is an evaluation technique that follows an inspection method, using measurements to identify conceptual barriers and pointing out side effects of design decisions [42]. This approach is not task-specific and does not need the involvement of API end users [42]. The main benefit of heuristic based evaluation is that, it can be applicable early in the design cycle of the API when usability testing is not possible or as a cost-effective method when resources are scarce.

The Cognitive Dimensions Framework [43] takes a similar approach as a heuristic-based evaluation for evaluating the effectiveness of notational systems such as programming languages and visual interfaces [43]. It has been used to evaluate the accessibility of the Protovis [7] API. The InfoVis Toolkit [11] followed a heuristic-based approach described by Shneiderman and Fekete [45] to evaluate the quality of their tool.

The main drawbacks of heuristic-based evaluation are that it requires significant expertise from the evaluator, multiple expert evaluations to ensure the reliability of the evaluation and the fact that often a large number of identified problems using heuristic evaluation are minor problems which might not have much impact on the actual user of the system but will be very costly to fix [44].

Usability evaluations are conducted with the actual users of the system in a lab environment, with limited time and specially designed tasks. A usability evaluation helps to identify the problem that might plague the actual user of the system and provide useful recommendations for the designer to address those problems. A study conducted by Jeffries et al. [46] comparing four different evaluation techniques showed that usability

evaluation exposed more severe problems, recurring problems and global problems than heuristic evaluation [46]. Also, some usability problems are highly unlikely to be discovered without conducting usability testing [44].

Some examples of information visualization toolkits that have been evaluated with usability studies in a lab environment are the prefuse toolkit [9] and the Papier-Mache toolkit [21].

Considering the limited time and simple nature of the task used in the usability studies, it is often hard to predict the learnability of the toolkit compared to the real task in an actual work environment where a developer might have more time. Sometimes the domain knowledge required for using an API is also quite high and conducting usability studies in a limited time frame might not be the most suitable method for evaluation. Usability studies are conducted in a controlled environment and, therefore, are limited in terms of how participants realistically react under time constraints and observation. It can also happen that a complex real task is much more difficult to do with the toolkit than the simpler tasks designed for the usability studies, which might give a false result about the usability of the API. To counter this problem, many researchers proposed longitudinal studies [57, 58, 59] where studies are performed over several weeks, even months, with the same participant group. It provides a better understanding about both the learnability and the usability of the toolkit to perform complex real tasks.

Examples of information visualization toolkits that have been evaluated using longitudinal studies are the XML toolkit [25] and the InfoVis Toolkit [11]. The XML

toolkit [25] and the InfoVis Toolkit [11] have been evaluated by projects developed by students using those toolkits over a period of a week to three months.

For a better understanding of the usability of an API, it is good to use a combination of the evaluation approaches stated above. Different methods have various strengths and the best evaluation of a new toolkit comes from applying multiple evaluation techniques [44]. For example, for the prefuse toolkit [9], a combination of application coverage, usability studies and the user adoption method has been used for evaluation; the InfoVis Toolkit [11] used application coverage, heuristic evaluation and longitudinal studies for evaluation.

In this thesis, the author has followed multiple evaluation techniques to evaluate the developed API: application coverage, heuristic evaluation and iterative usability studies. The main reason behind conducting iterative usability evaluation is that it helps determine the major usability issues with the API at an early stage of development, and provisions can be made to address those issues. Conducting follow-up usability studies will evaluate if there is any improvement in usability after making those changes and will also help identify further usability issues.

2.4 Chapter Summary

In this chapter, available literature on the topic of information/tree visualization toolkits has been presented covering three different aspects: tree drawing approaches in existing toolkits; task specific interaction support for trees; and different evaluation approaches followed by the researcher to evaluate those toolkits. While work has been done to support different tree layout and interaction features in existing toolkits, they are still

limited in providing flexibility in customization and task-specific interaction support for tree visualization. It will be helpful for the developer to have a new API for visualizing and interacting with trees that will address those limitations. Conducting multiple evaluations of the newly developed API will help identify usability issues. Steps can be taken to improve the usability of the API based on the discovered usability issues from the studies.

Chapter Three: AVIT – an API for Visualizing and Interacting with Trees

In this chapter requirement, the fundamental design rationale and the implementation details of the AVIT will be discussed. AVIT uses the operator-based generative tree drawing approach proposed by Schulz *at el*. [6] as a fundamental design consideration for tree drawing. For interaction support, the author has implemented an interaction layer in the API based on the task taxonomy for tree visualization as discussed in Chapter 2.

Section 3.1 describes the requirements of the API, in Section 3.2, API fundamentals, the tree layout generation approach based on the operator-based concept is discussed and Section 3.3 provides implementation details of the API with examples.

3.1 Requirements for the API

The API has been developed to address the limitations of the existing toolkits in providing customization and interaction support for different tree layouts as described in Chapter 1 and Chapter 2. The following requirements as discussed in Chapter 1 and Chapter 2 were the main focus for developing AVIT:

(1) Using AVIT it will be possible to draw a wide range of existing tree layouts.

(2) The layout specification should be concise.

(3) AVIT will provide flexibility in customizing the tree structure.

(4) Generating hybrid tree layouts will be possible.

(5) Drawing a novel tree layout will be possible using existing components of the AVIT.

(6) The API will provide task-specific interaction support for tree visualization.

3.2 Fundamentals of the API[4]

This section provides details regarding the operator-based tree drawing approach developed by Schulz *et al.* [8] which was used as a cornerstone for developing AVIT.

Schulz *et al.* [6] observed that despite the large variety of tree drawings, many of the most common ones follow a similar overall process of six *stages* [6]. Schulz named this process the *layout pipeline*. The different results of this process ranging from implicit, space-filling tree-maps to explicit, node-link radial layouts are merely due to different layout actions performed at each of these *stages*. These actions were named *layout operators*. Using this operator based model of a tree layout, it becomes possible for a user to generate countless different tree drawings by plugging different operators or operator sequences into the layout pipeline. The following subsections provide the details of the operator-based tree drawing [6].

3.2.1 Operator – based Tree Layout Generation Approach

The six *stages* of the layout process as described below permit a high-level differentiation between the intent with which different operations are carried out during layout generation:

0. **INITIALIZATION** for supplying the initial drawing space;
1. **TRAVERSAL** for moving up or down in the tree;
2. **PREPROCESS** for preparing the nodes to be laid out;
3. **PRELAYOUT** for preparing the drawing area for layout;
4. **ALLOCATION** for assigning drawing space to the nodes;

---

[4] This section uses contents from the co-authored paper [8]. Co-author's permission has been attached in

5. **POSTLAYOUT** for final beautification of the layout result.

The proposed operator-based approach uses this workflow of six stages as a fixed layout pipeline, in which *Stage 0* is only invoked once, whereas *Stages 1* through *5* are repeatedly passed through for each level of the tree. This helps the second requirement of developing the API, hiding most of the housekeeping functionality, such as data management and tracking the layout dimensions, and leaving only these stages exposed for customization with a few layout operators. The layout pipeline and its six stages are described in the following section.

3.2.1.1 The Tree Layout Pipeline

Tree layout procedures differ in whether they traverse the tree top-down or bottom-up. It has been observed by Schulz *et al.* [48] that the main distinction between the two is a subdivision layout for top-down traversals vs. a packing approach for bottom-up traversals.

In the operator-based approach, Schulz et al. partition the tree into its individual levels L_d, where d denotes the depth of a level. Each level consists of a set of tuples of the general form $(\{s_i\},\{n_i\})$.The first element of these tuples contains geometric shapes s_i which is a subset of R^{dim} with dim ϵ {2,3}, e.g., rectangles or circles in 2D, or cuboids or spheres in 3D. The second element contains the nodes of the tree that are associated with the geometric shapes. Shapes and nodes can be thought of as objects that internally hold a number of properties. Shape properties include their position, their extent, and their

orientation. Nodes contain information about their parent, as well as a number of numerical attributes, such as the number of children and siblings and the depth and value. The tuples can occur in three different variants:

- **1 shape, *n* nodes:** Such a tuple holds the initial state of a top-down, partitioning layout. The multiple nodes are siblings. The singular shape encloses the drawing space assigned to the parent of the multiple nodes. For such tuples, the layout algorithm should distribute that space among the nodes.
- ***m* shapes, 1 node:** Such a tuple holds the initial state of a bottom-up, packing layout. The multiple shapes belong to the children of the singular node. For these tuples, the layout algorithm should tightly pack the shapes and assign the bounding shape of the packing result to the parent node.
- **1 shape, 1 node:** Such a tuple holds the end result of a successful layout. Whether it was generated top-down by partitioning a single shape into multiples shapes or bottom-up by packing multiple into a single shape, in the end each node is assigned its individual shape.

This transition from (1, n) / (m, 1)-tuples into (1, 1) tuples is performed along the different stages of the layout pipeline. Each layout stage can be viewed as an iterator over all tuples t in L_d, which applies a set of layout operations to t. The layout pipeline is iteratively passed through until all nodes have been assigned their individual shape. For top-down traversal, a full pass through the layout pipeline detailing the changes that each stage makes to L_d have been shown in the following. A schematic overview of the entire process is given in Figure 3.1.

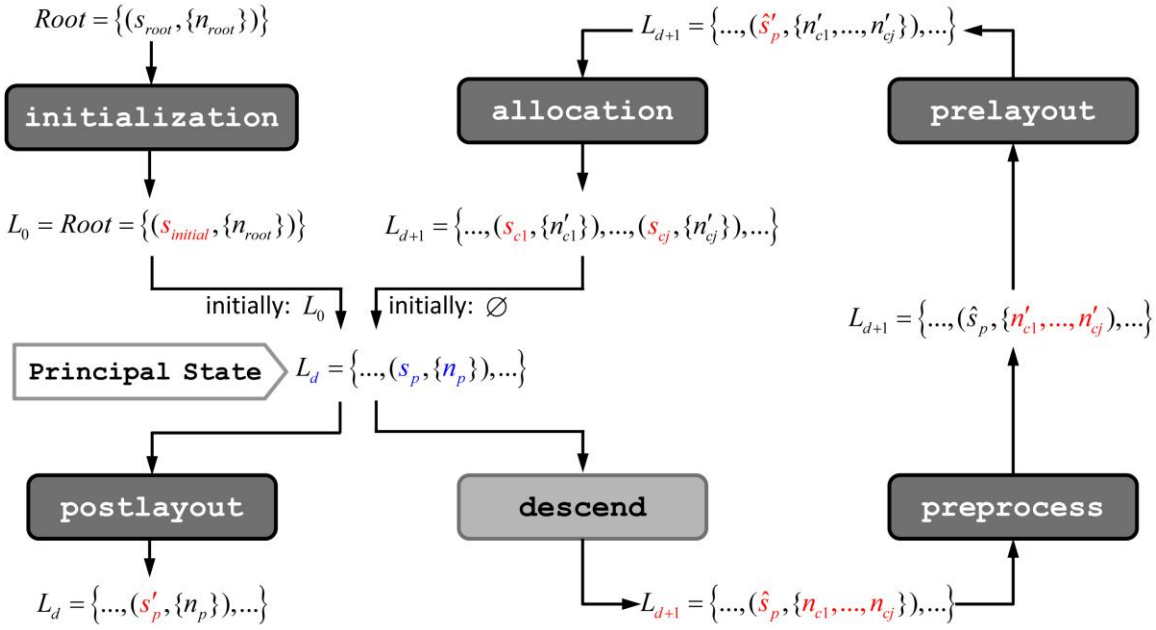


Figure 3.1: Schema for top-down tree layout pipeline. The stages colored dark gray are those that can be configured through operators. The light gray stages are constant as the direction of traversal is fixed depending on whether the layout is top-down or bottom-up. The variables *s* denote geometric shapes, the variables *n* denote nodes of the tree. The index *p* marks parent shapes/nodes; the index *c* marks child shapes/nodes. Changes made at the individual stages to the current level L_d are highlighted in red. Modifications are denoted with a prime symbol, copies are denoted with a hat symbol. Blue indicates a mere renaming of the variables without any change to them, which is done so that each iteration through the layout process starts with a level L_d. [6]

Stage 0: initialization is a preparatory stage that defines the shape of the root node for its subsequent subdivision in the layout process. It can be customized to transform the usually rectangular initial drawing space into an initial shape as it is expected by the

following layout. Common uses are radial layouts with angular subdivision that expect a circular space, or layouts that grow outwards and thus require a down-scaled initial space, so that they do not exceed the available overall space during layout.

The *Principal State* forms the defined starting point for the layout of each level L_d. It consists of a set of (1,1)-tuples. This is by definition true for the root level L_0 after initialization and it must be true for the result of *Stage 4* that assigns each node its own shape. The just laid out child nodes are now considered parents themselves and passed as an input to the following traversal to retrieve their children for laying out the next level.

Stage 1: traversal is fixed to a top-down DESCEND. It takes the current level L_d and advances it to L_{d+1} by composing a new tuple for each existing one. First, the new tuples contain a copy $\hat{s_p}$ of the parent shape s_p. This makes sure that all subsequent steps no longer manipulate the parent shape s_p itself, but the one in which the children are to be laid out. Second, the newly created tuples contain the set of children $\{n_{c1}, n_{c2}, \ldots.\}$ of the respective parent node $\{n_p\}$. If $L_{d+1} = \acute{O}$, the layout process terminates.

Stage 2: preprocess adapts the set of nodes of each tuple for its subsequent layout. This can be, for example, a sorting operator or a weighting operator. The latter multiplies a numerical attribute of a given node with a weight. Depending on this weight, the size of the later assigned space will be either smaller or larger than it would otherwise have been. It can thus be seen as a scaling on data level. This is particularly important for space-filling layouts in which scaling up a node in the view space after the space allocation would result in overlap and thus over-plotting.

Stage 3: prelayout adapts each tuple's drawing space. This is done, if not all of the given space shall be distributed among the children, e.g., shrinking the space as necessary to maintain a border, or reconfiguring the space entirely. The latter is used, for example, to realize parent-centric radial layouts, instead of further subdividing a circle section resulting from a previous subdivision and thus making just another subdivision with respect to the same circle center, one can simply embed a new full circle into the circle segment. This circle will then be subdivided with respect to its own center and thus produce a parent-centric layout.

Stage 4: allocation assigns each node of a tuple's node set a portion of the tuple's space. These portions are not required to be overlap-free, even though most allocation strategies adhere to a strictly exclusive subdivision. After the assignment of individual drawing space to each node, additional steps can be undertaken to further optimize a possibly crude first space allocation. The end result is again a set of tuples that can be mapped onto the *Principal State* and thus be used as a starting point for the next level's layout.

Stage 5: postlayout is performed after *Stage 1* has made its copy of the resulting space and starts off with the next level's layout on an independent drawing space. Then, this stage can perform any final adjustments regarding the appearance, such as reshaping it into a dot and selecting a connector style to produce an explicit node-link rendering. If none is made, the shapes will be drawn as they are – simply as rectangles, circle segments, etc.

3.2.1.2 The Tree Layout Operators

Operators are of imperative nature, they capture what to do in which order, which is close to the procedural thinking about layout generation. As the inputs, as well as the outputs of all operators are the aforementioned tuples, they can be called in arbitrary order, left out completely (identity operator), or even be called multiple times in a row with no conceptual restriction. Because of this consistent behavior, each pipeline stage will not only admit a single such operator, but also sequences of operators. At each pipeline stage, the operators of such a sequence are applied in order to all tuples of the level L_d, which is currently laid out:

```
foreach t ϵ L_d {
    foreach op ϵ op_sequence {
        op (t, P, c)
        }
}
```

The operators are thereby called with three parameters: *t* is the tuple it shall be applied to, *P* is a set of operator-specific parameters that govern the details of its function, and *c* is a conditional that can be used to select a range of nodes for which this operator is to be applied. The conditional is used to express local tree layouts that apply different operators to different parts of the tree. If the conditional does not hold true for the current tuple *t*,

then t is passed back unchanged. Otherwise, the operator transforms the tuple with respect to the given parameters:

$$\textbf{func}\ \ \mathrm{op}(t,P,c)\ \ \{\ \ \textbf{if}\ \ c(t)\ \ \textbf{then}\ \ t \xrightarrow{\mathrm{op}_P} t'\ \ \}$$

Depending on its purpose, each stage changes t only in one aspect – its geometry, the shape(s), or its data, the node(s). In line with [49], Schulz *et al.* further discern between two types of operators: creation operators and modification operators [49]. In combination, the scope of an operator (a tuple's shape or data element) and the type of an operator (creation or modification) yield four different kinds of operators: data creation, shape creation, data modification, and shape modification. These four kinds of operators give additional justification to the observed six *stages* of the pipeline, as there are exactly four stages (*Stage 1* through *4*) – one to apply each kind of operator, plus one stage each for preparing (*Stage 0*) and finalizing (*Stage 5*) the layout through additional shape modifications. Table 3.1 lists which types of operators are applicable at each stage and gives some examples for them.

Table 3.1: Applicable operators at each stage of the layout process.

Stage	**Type**	**Scope**	**Examples**
Stage 0: initialization	modification	Shape	RESHAPE, SCALE, ROTATE
Stage 1: traversal	creation	Data	DESCEND, ASCEND
Stage 2: preprocess	modification	Data	ORDER, WEIGHT

Stage 3: prelayout	modification	shape	SCALE, ROTATE, TRANSLATE,RESHAPE
Stage 4: allocation	creation	shape	SQUARIFY, SLICE, STRIP, PACK.
Stage 5: postlayout	modification	shape	RESHAPE, TRANSLATE, SCALE, ROTATE, FILL.

With this mapping in the background, the pipeline can check automatically whether a given operator is used correctly at a certain stage and thus aid debugging of the layout. In the following, all four kinds of operators are shortly discussed and some instances of such operators are given.

Data Creation Operators construct a tuple's node (set) from existing tuples. In the top-down case, this is done through the DESCEND operator, which takes a node and retrieves its children as a new node set. In the bottom-up case, this is done through the ASCEND operator, which takes a set of sibling nodes and retrieves their parent as a new node. Both operators can be used as an interface to a variety of data sources, e.g., not only given trees that are stored on disk, but also to tree generating algorithms that merely produce a new level when called. Data creation operators are used exclusively in *Stage 1* of the layout process.

Shape Creation Operators have to visually reproduce the effects of the used data creation operator. If, in Stage 1, a DESCEND was used to "split" a parent node into its children, the same has to be done to its geometry – the shape has to be subdivided into a number of shapes for the children. This can be done by using operators, such as SLICE for a slice/dice subdivision, STRIP for a Strip tree-map like subdivision, or SQUARIFY for a subdivision as it is used in Squarified tree-map. Yet, if an ASCEND operator was used in *Stage 1* to "merge" a number of child nodes into their parent node, this has to be reflected here as well, and the child shapes have to be packed with a PACK operator into a parent shape. Shape creation operators are only used in *Stage 4* of the layout process.

Data Modification Operators prepare the nodes for subdivision or packing. An example is the ORDER operator to sort a set of siblings, as it is required by some subdivision operators, such as SQUARIFY. Another possibility is to scale a node's attribute value through the WEIGHT operator to influence the shape creation. If a node is assigned a weight of 0, this is equivalent to a pruning of the tree at this node. Data modification operators can only be used in *Stage 2* of the layout process.

Shape Modification Operators adapt the visual appearance of shapes. This includes three different aspects: shape transformation, shape alteration, and shape representation. Operators that transform the shape are the common geometric transformations SCALE, ROTATE, and TRANSLATE. Yet, these operators cannot, for example, alter a rectangular shape into a circular one. This is what the RESHAPE operator does. Shape alteration is commonly used in *Stage 0* to yield a circular drawing space for radial layouts, but also in *Stage 5* to alter the shape into a dot to create a node-link diagram.

Furthermore, the RESHAPE operator is used to switch from a root-centric to a parent-centric layout approach simply by reshaping, for example, a circle segment from a previous subdivision step into a new full circle. While transforming or altering a shape modifies its geometry, shape representation operators, such as FILL, SET STROKE WIDTH, etc., customize its appearance. This also includes operators to configure a connector line in case the displayed shapes require an edge to make the parent-child-relationship explicit. Operators of this kind are used in *Stages* 0, 3, and 5.

3.3 Implementation of the AVIT

Using the operator-based tree layout generation process [6] as described in Section 3.2, AVIT − a web based API for Visualizing and Interacting with Trees − has been implemented. The API has been written using JavaScript, SVG and HTML5, so that it is independent of specific platforms and readily available as an interactive demo over the web. The following subsections provide the detailed description of the architecture of the API with examples.

3.3.1 API Architecture

This section describes the high level architectural details of the API. The API consists of four independent modules as shown in Figure 3.2. The details of the modules are described below:

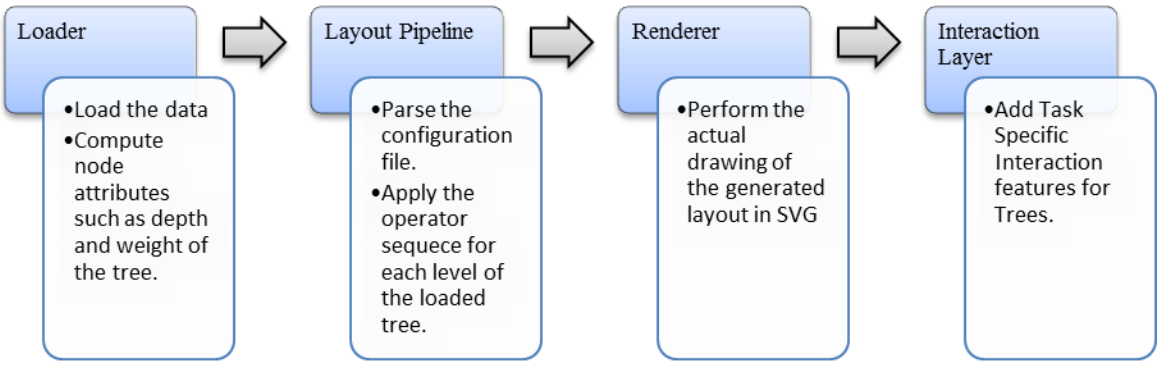


Figure 3.2: High level architecture of AVIT

3.3.1.1 Loader

Loader module loads and parses the data to be visualized. Currently, it supports the TreeML[5] data format but the module has been written in such a way so that it can be easily extended to support other types of data format in future.

For example, assume a developer needs to visualize a movie data set. Suppose that we have only two genres of movie, action and comedy. Within each genre we have some movie names. So, for our dataset we have three levels of hierarchy: Movies, Genres and Movie names.

[5] http://www.cs.umd.edu/hcil/iv03contest/datasets.html

Figure 3.3 shows the representation of the data in TreeML format. It starts the tree with a tag **<*tree*>**. Non–leaf nodes are represented with **<*branch*>** tag and leaf nodes are represented as **<*leaf*>** tag.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE tree SYSTEM "treeml.dtd">

<tree>
    <branch> <attribute name="name" value="Movies"/>
        <branch> <attribute name="action" value="Action"/>
            <leaf> <attribute name="1" value="Fight Club"/> </leaf>
            <leaf> <attribute name="2" value="The Matrix"/> </leaf>
            <leaf> <attribute name="3" value="Memento"/> </leaf>
        </branch>
        <branch> <attribute name="comedy" value="Comedy"/>
            <leaf> <attribute name="1" value="Toy Story"/> </leaf>
            <leaf> <attribute name="2" value="Shrek"/> </leaf>
        </branch>
    </branch>
</tree>
```

Figure 3.3: Movie Dataset in TreeML format

After the data has been loaded, a DFS traversal is run through the data to compute node attributes such as level, depth, number of children of the tree. Computed node attributes are stored in the DOM tree as a *stats* property.

3.3.1.2 Layout Pipeline

This module parses the operator sequence for each *stage*, as specified in the configuration file, and carries it out for each level of the loaded tree. The final result of this module is an assigned screen coordinate for each shape to be drawn based on the layout.

The implementation of the operators *op(t, P, c)* in the API hides the first argument, the tuple *t*, from the user as the pipeline takes care of looping through the tuples and carries out the operators on them. Furthermore, the last argument, the conditional *c*, is optional. If no conditional is given, it is assumed as TRUE and thus the operator is applied to all nodes. The result is an operator signature that looks very much like a procedure call and should thus be familiar to most programmers. The details of each operator can be found online.[6] The configuration file for drawing a classical tree layout using the API can be seen in Figure 3.4.

```
1  // Classical Tree Drawing config file
2  INITIALIZE:
3  {
4
5  }
6  PREPROCESS:
7  {
8      order(SHUFFLE);
9  }
10 PRELAYOUT:
11 {
12     scale(BY,TOP,"-root.dimY/root.height");
13 }
14 ALLOCATE:
15 {
16     slice(HORIZONTAL,"leaves");
17 }
18 POSTLAYOUT:
19 {
20     scale(TO,BOTTOM,"root.dimY/root.height")
21     reshape(DOT);
22     fill("#335500");
23 }
```

Figure 3.4: Configuration file for Classical Tree Layout

[6] http://tinyurl.com/operatordocs.

3.3.1.3 Renderer

After the layout task has been completed from the layout pipeline module, the renderer module produces the SVG code from the layout. Figure 3.5 shows the rendered classical tree layout generated using the dataset from Figure 3.3 and configuration file in Figure 3.4.

3.3.1.4 Interaction Layer

The interaction module has been implemented as a separate layer on top of the rendered tree layout. Based on the task taxonomy for tree visualization as described in Chapter 2, topology and attribute-based interaction has been implemented in the API.

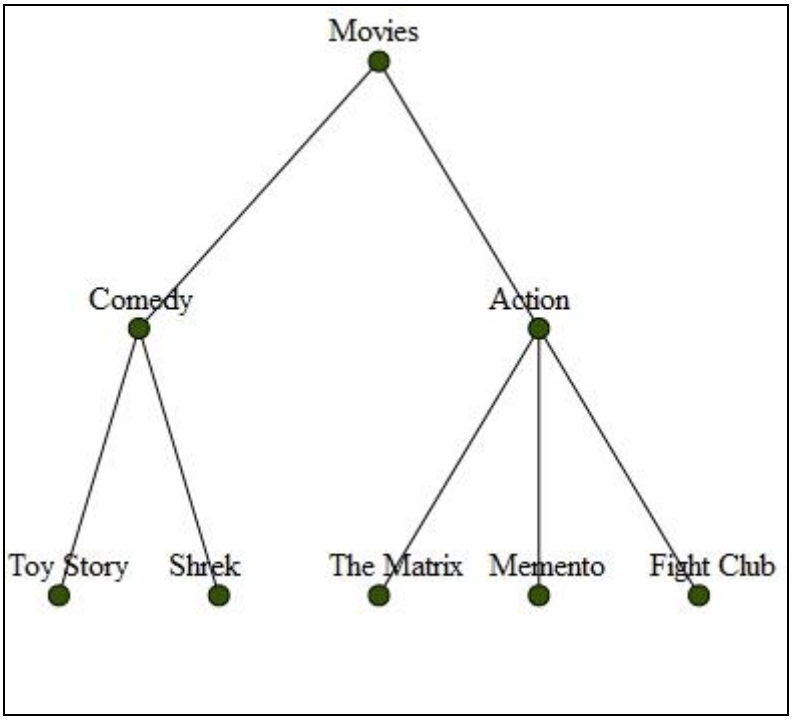


Figure 3.5: Rendered Classical Tree Layout using AVIT

In the API, interaction can be added via a simple call to the `addEvent(eventType, eventName)` method which takes the interaction type and interaction name as parameters.

Examples of *topology based interactions* as implemented in the API are, highlighting a sub-tree rooted on a node, finding the ancestor or descendent of a node, showing the path to the root, highlighting children and sibling, selecting an area of a tree for closer inspection.

Figure 3.6 shows an example of topology-based interaction: the sub-tree highlighting of a selected node. The interaction has been provided as an option to a menu in the API. To

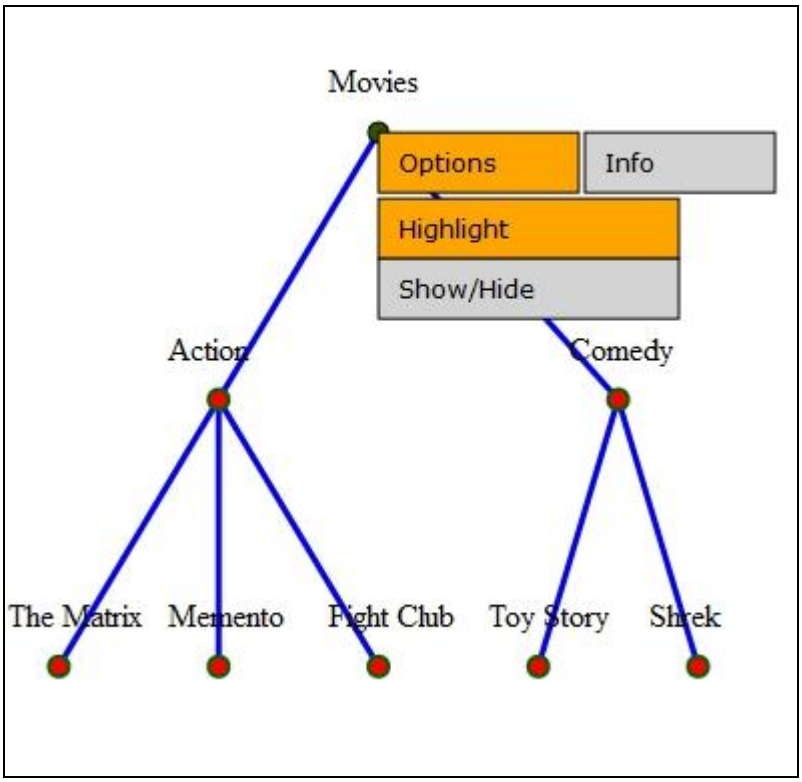


Figure 3.6: Topology based Interaction (Highlighting sub-tree)

add this menu interaction one needs to add the line `addEvent("click", "showMenue");` in the ***addInteraction.js*** file.

Attribute-based interactions implemented in the API were search interactions for searching by a particular attribute value of a node, showing details of a particular node as tooltip, sorting nodes based on these values, and filtering nodes based on a specified range .

Figure 3.7 shows an example of an attribute-based interaction where the mouse hovering on a node displays the detailed information about that node. To add this interaction `addEvent("mouseover", "showAttribute")` has to be added in the ***addInteraction.js*** file .

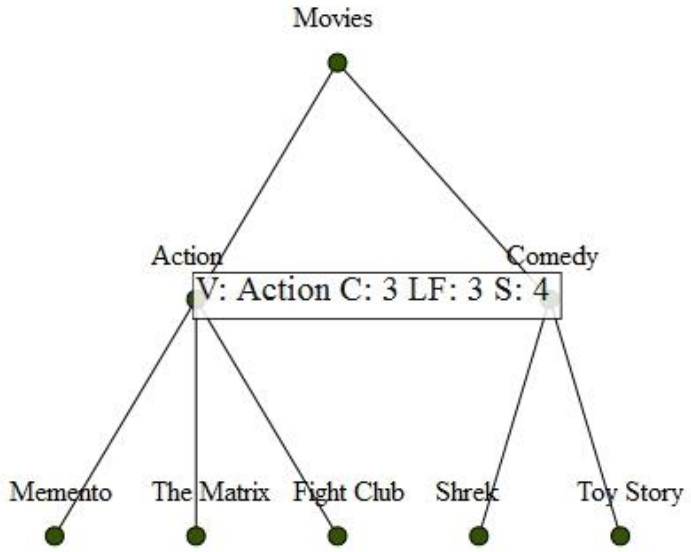


Figure 3.7: Attribute based Interaction (Showing details of the node *Action* where V: Value, C: Number of children, LF: number of leaves, S: Sub tree Size)

3.3.2 Examples

Examples in this section extend the previous example and also provide some new examples to show the features of the API in fulfilling the requirement (RQ) as mentioned in Section 3.1.

3.3.2.1 Generating Different Tree Layouts with Concise Specification (RQ 1 and RQ 2)

Examples presented in this section used the movie dataset described in section 3.3.1.1. Figure 3.8 shows the radial node-link tree layout along with the configuration file to generate the layout. It can be seen that with only 13 lines of code and seven operators, one can generate a radial node-link layout using AVIT.

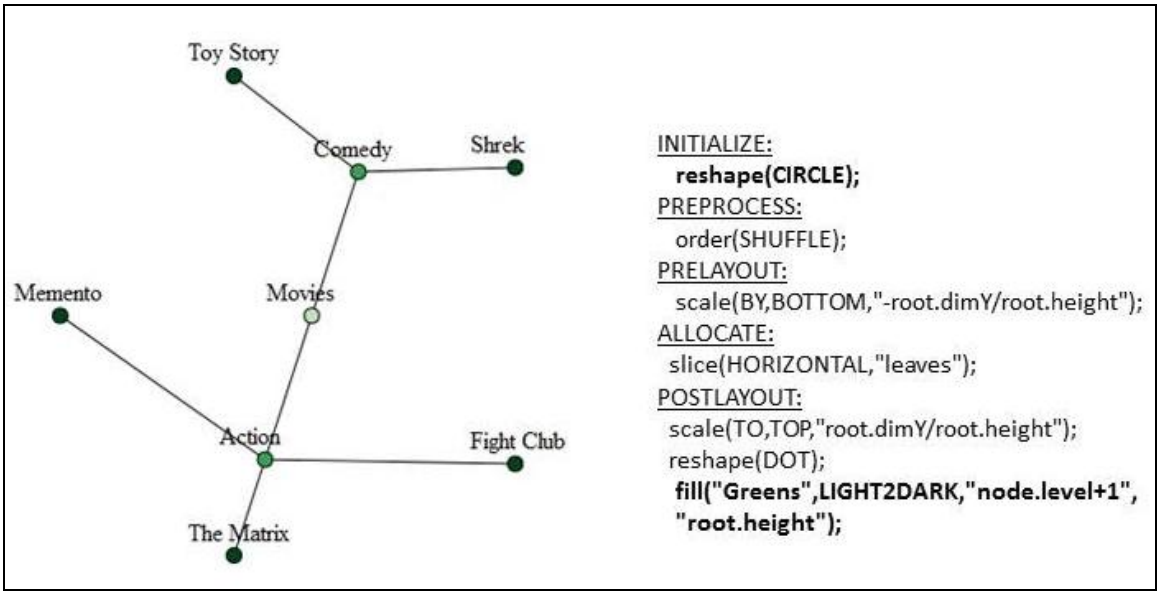


Figure 3.8: Radial node-link tree layout

To generate a sunburst layout from the above code, one just need to comment the line *reshape (DOT)* in the POSTLAYOUT *stage* of the radial node-link layout as by default tree layout is drawn using a space filling approach. The generated sunburst layout along with the configuration file is shown in Figure 3.9.

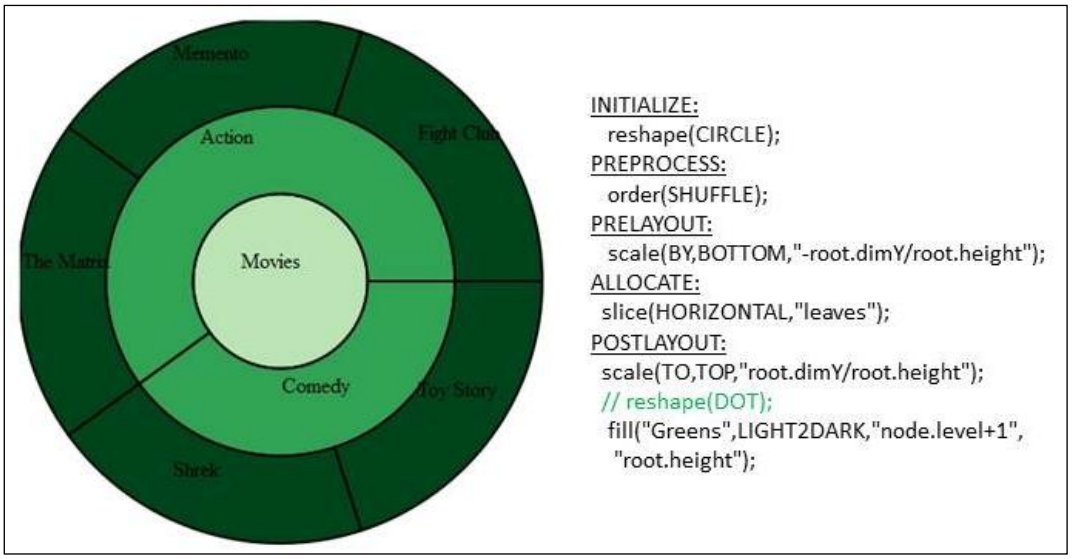


Figure 3.9: Sunburst tree layout

The following few examples shows, how by making minimal changes in the configuration file one tree layout can be generated from another. (Parts of the code changed from previous configuration file are highlighted in **Bold**). These examples also verify the flexibility provided by AVIT in customizing tree layout.

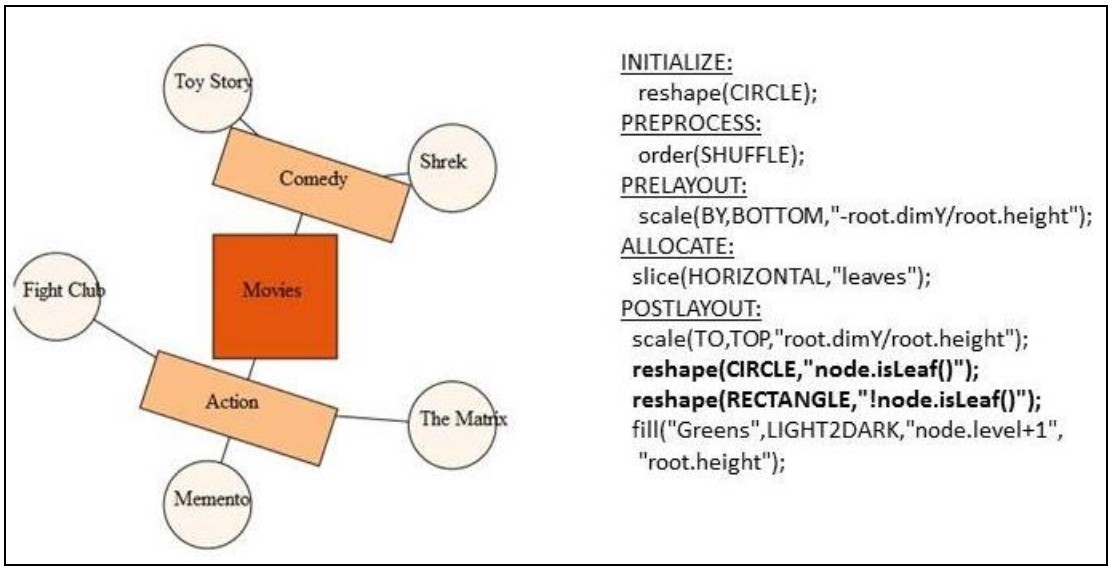


Figure 3.10: Custom node-link tree layout

For example in Figure 3.10, a customized radial node-link layout has been generated using circular shapes for the leaf nodes and rectangular shapes for non-leaf nodes as a conditional parameter.

In the layout specification it was required to put any conditional parameter within a quotation. As condition itself may contain commas, the design decision of putting condition parameter within a quote has been made to make clear separation of the condition parameter from the other parameters.

In Figure 3.11, an icicle plot tree layout has been generated by commenting a few lines from the previous configuration file and changing the color to blue in the *fill* operator. Commenting *reshape (CIRCLE)* operator in INITIALIZE phase in Figure 3.11 selects rectangular drawing area as a default drawing option as specified in the design of AVIT.

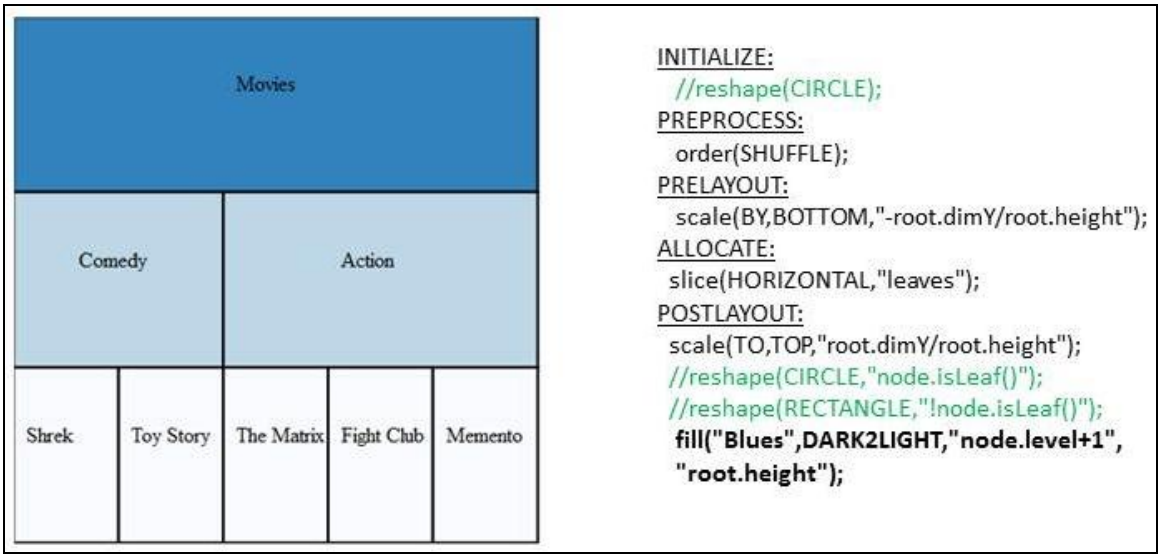


Figure 3.11: Icicle plot

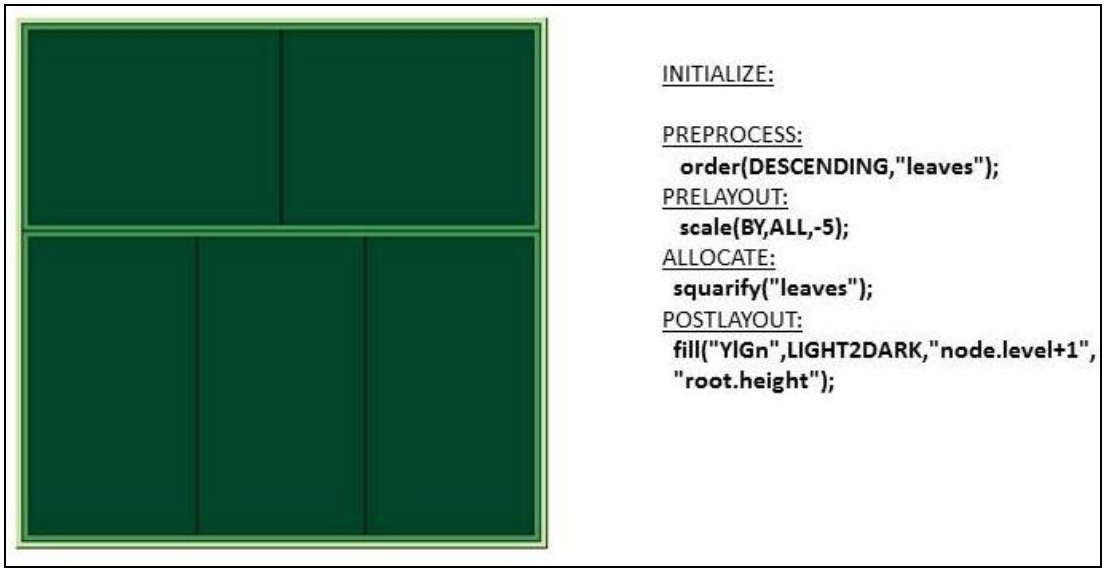
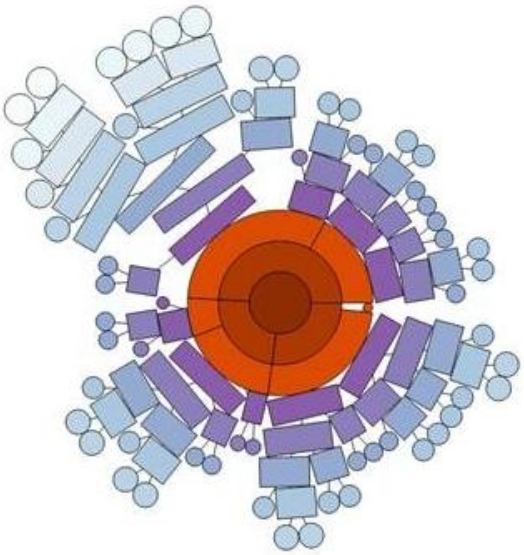


Figure 3.12: Nested Squarified tree-map layout

Figure 3.12 shows a nested Squarified tree-map and the configuration file used to generate the layout.

3.3.2.2 Customizing Tree Layout (RQ 3)

In Figure 3.13 a highly customized tree layout with different shapes and colors for nodes in different levels of tree has been drawn. It can be seen from t Figure 3.13 that the space-filling layout has been drawn up to level two of the tree and a node-link layout has been drawn for rest of the levels.

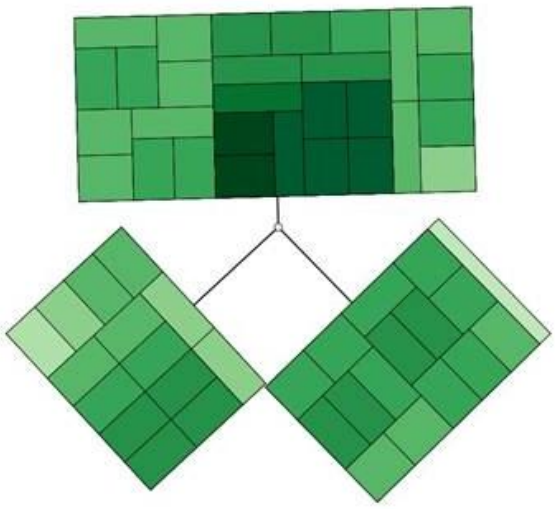


```
INITIALIZE:
  reshape(CIRCLE);
PREPROCESS:
  order(SHUFFLE);
PRELAYOUT:
  scale(BY,BOTTOM," root.dimY/root.height");
ALLOCATE:
  slice(HORIZONTAL,"leaves");
POSTLAYOUT:
  scale(TO,TOP,"root.dimY/root.height");
  reshape(CIRCLE,"node.isLeaf()");
  reshape(RECTANGLE,"node.level>2&&
  !node.isLeaf()");
  fill("Oranges",DARK2LIGHT,"node.level+1",
  "root.height", "node.level<3");
  fill("BuPu",DARK2LIGHT,"node.level+1",
  "root.height", "node.level>=3");
```

Figure 3.13: Customized tree layout

3.3.2.3 Generating Hybrid Layout (RQ 4)

The following example in Figure 3.14 shows a hybrid tree layout generated using AVIT. The hybrid layout in Figure 3.14 combines the radial node-link and a tree-map layout.



```
INITIALIZE:
  reshape(CIRCLE);
PREPROCESS:
PRELAYOUT:
  scale(BY,BOTTOM,-40,"node.level==1");
  reshape(RECTANGLE,"node.level==1");
ALLOCATE:
  slice(HORIZONTAL,"leaves","node.level==0");
  squarify("leaves","node.level>0");
POSTLAYOUT:
  reshape(DOT,"node.level==0");
  scale(BY,BOTTOM,-40,"node.level==1");
  reshape(RECTANGLE,"node.level==1");
  setStrokeWidth(EDGES,2);
  fill("Greens",LIGHT2DARK,"node.level+1",
  "root.height");
```

Figure 3.14: Hybrid tree layout (radial node link + Squarified tree-map)

3.3.2.4 Generating Novel Layout (RQ 5)

Just by changing the drawing area to a circular one by adding *reshape (CIRCLE)* in the INITIALIZE stage (with comparison to the configuration file in Figure 3.12), a novel tree layout named “Nested Squarified Pietree” (a modified version of nested Squarified tree-map), has been generated using AVIT (see Figure 3.15). This layout is presented in our paper [6] and has been accepted by the information visualization researcher community as a novel tree layout.

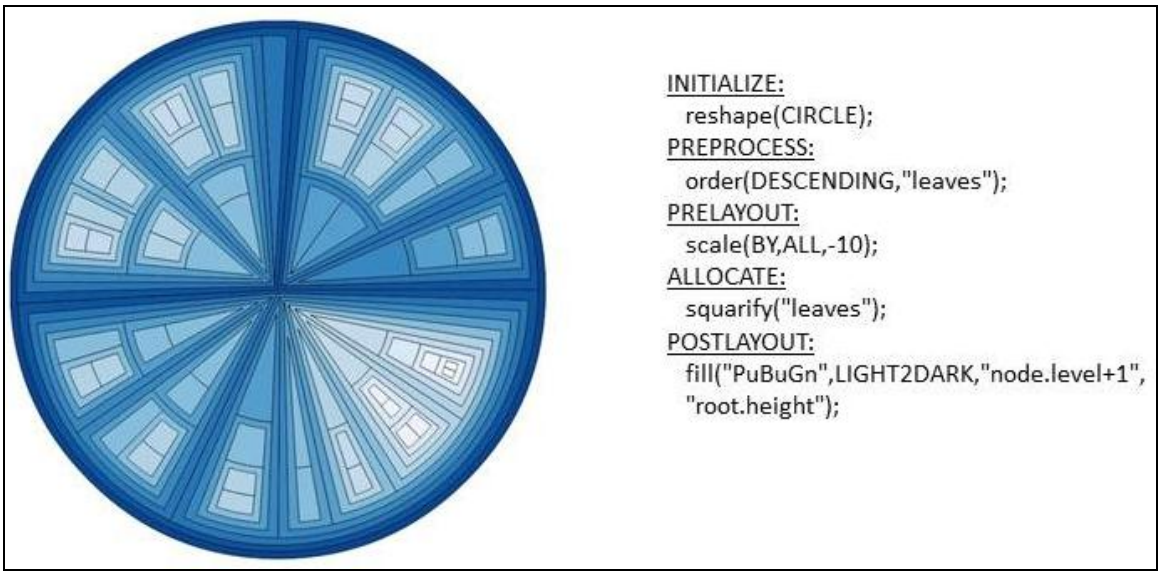


Figure 3.15: A novel tree layout generated using AVIT.

3.3.2.5 Interaction (RQ 6)

Examples related to fulfilling RQ 6 regarding interaction features of AVIT have already been provided in Section 3.3.1.4.

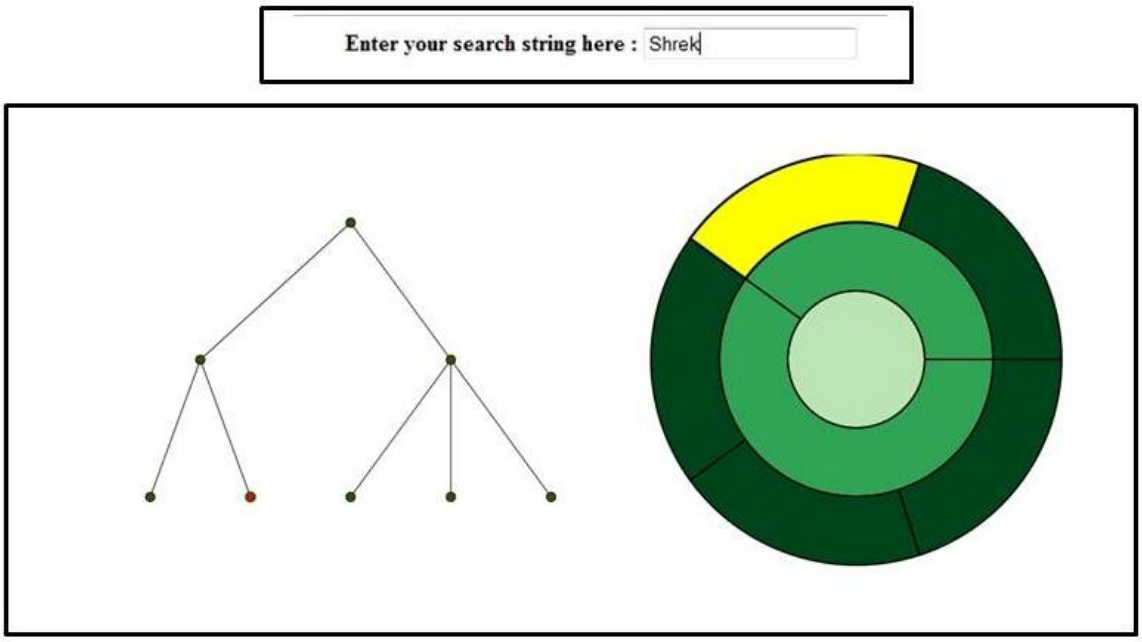


Figure 3.16: Search and Brushing-and-Linking interaction techniques using AVIT.

Figure 3.16 shows an additional example of implemented interaction features in AVIT: search interaction. Matching movie nodes ("Shrek") is highlighted in both layouts. The example in Figure 3.16 shows that, using AVIT, it is possible to add brushing and linking interaction [51] where interactive changes made in any visualization are automatically reflected in the other visualizations. To add brushing and linking using AVIT, one just needs to use two or more svg drawing areas on the screen so that different tree layouts can be drawn. Any interaction performed in any of the layouts will reflect the outcome on both layouts.

3.4 API Documentation

To support the API, online wiki-based documentation[7] has been written by the author. In the online documentation, details of interaction features, operator definitions, their uses and example code are provided. An interactive tutorial with complete examples of different tree layouts was also provided with the documentation. Details of the documentation features will be discussed in Chapter 5, Section 5.2.

3.5 Limitation of the API

Currently the API supports TreeML format datasets to visualize a tree. The data loader module has been written in a way so that support for other data formats can be added easily. To add support for a new dataset format, one needs only to write a specific parser to convert the data format into TreeML format.

The operator-based tree drawing approach used in the API, supports only 2D trees with rectangular and circular shapes. Therefore, layouts relying on 3D or polygonal subdivision cannot be reproduced with it. Yet, most common layouts can be generated with this set of geometric shapes, while the involved computational geometry, e.g., for the RESHAPE or the shape-agnostic SQUARIFY operators, is still manageable and has satisfactory runtimes even for larger trees.

Current version of the API only support circular, rectangular and point shape for nodes for a node-link tree layout. Also, only straight edge between nodes using a line shape can be drawn. Other type of shape for nodes and edges can be added in the API by making

---

[7] http://tinyurl.com/operatordocs

changes in the renderer module but it will require coding effort and knowledge of SVG drawing from the user.

Interactions specific to comparing different trees, or sub-trees of the same tree, are not yet added in the API. It is possible, though, to add these features using the existing components of the interaction layer of the API. For example, if someone wants to compare whether two trees are of equal size, he just needs to compare the size property for each tree which is automatically computed when the tree data is loaded using the loader module of AVIT.

3.6 Chapter Summary

In this chapter, detailed descriptions of the requirements, design decisions and implementation details of AVIT have been provided. Step-by-step examples are shown to evaluate the expressiveness of the API in fulfilling the stated requirements. Limitations of the API have also been discussed.

Chapter Four: Usability Study 1: Developers reaction to AVIT

This chapter describes a preliminary evaluation of AVIT. The preliminary evaluation was performed to receive early feedback regarding the usability and appropriateness of AVIT. Another goal of conducting the preliminary study was to debug the experiment and come up with a better design for the second usability study. This evaluation was conducted after major features of AVIT, such as operator-based tree drawing, basic interaction support and reference-based documentation had been implemented.

The study was guided by four primary research questions, identified in Section 1.3, namely:

(1) Which of AVIT's features do developers like?

(2) What difficulties do developers face while using the operator-based approach to drawing tree layouts?

(3) How useful are the given documentation and tutorial materials for finding task related information?

(4) How can AVIT be improved to better support developers' expectations?

The following sections provide details of the study.

4.1 Study Setting

The preliminary evaluation was mainly concentrated on collecting developers' behavior and exposing their satisfaction with the API. The following sub-sections provide details regarding the method followed for the study.

4.1.1 Participants

Participants were recruited from the student population of the Computer Science and Environmental Design department at the University of Calgary using mailing lists. A monetary compensation of $20 was offered for participation. Respondents were pre-screened about their programming experience using a questionnaire. Participants must have had at least 1 year of experience with programming to be included in the participant pool.

The preliminary study consisted of eight participants (referred to as P1…P8). Six out of the eight participants reported a minimum of 2 years' experience with Object Oriented programming languages, while others had experience with the Processing programming language[8]. Participants included three PhD students, three M.Sc. students, and two senior undergraduate students. Two of the participants had industry experience before coming back to academia (P5, P8). Although all the participants were from academia, their expertise level is anecdotally comparable to that of recent graduates in software development positions. Figure 4.1 and Figure 4.2 shows the summary of participants' background experience.

[8] http://processing.org

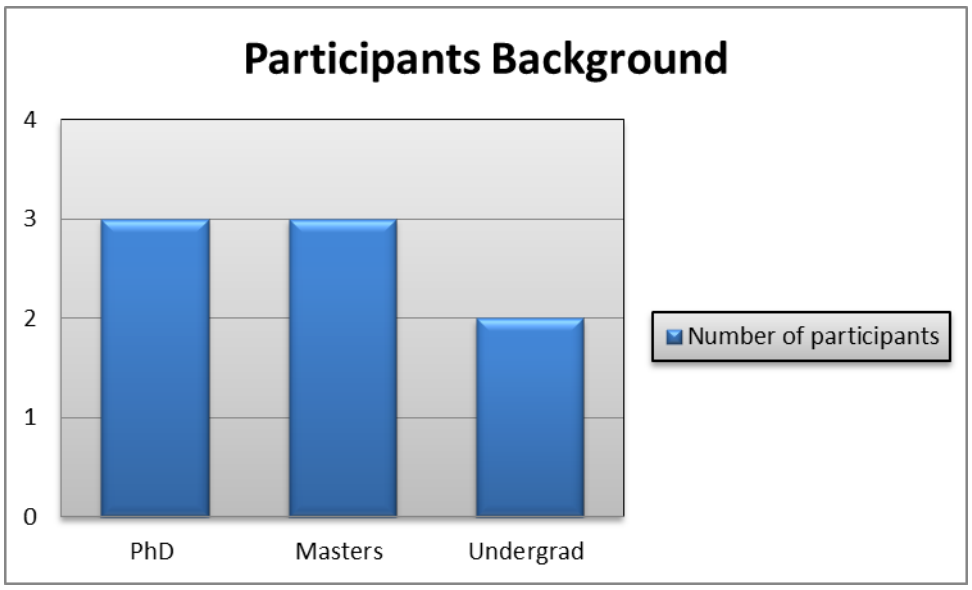


Figure 4.1: Participants Background

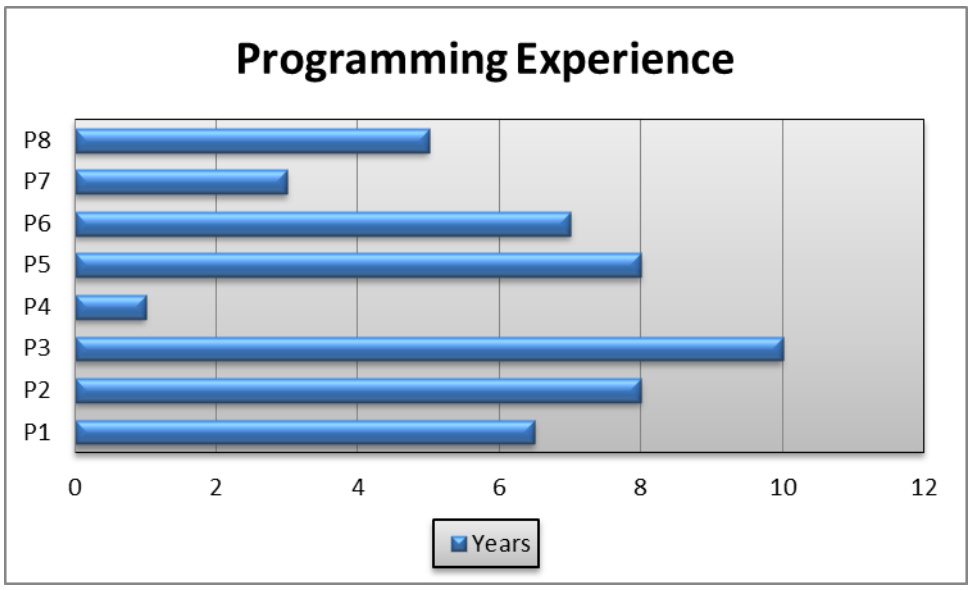


Figure 4.2: Participants Programming Experience

From the background questionnaire asked during the evaluation, I found that participants had different levels of experience with visualization tools. A participant was considered to possess an intermediate level of experience with visualization tool if s/he had taken

courses in visualization or had worked on at least one project focusing on visualization or had familiarity working with visualization APIs. If a participant's experience was limited to the use of common visualization tools like the Excel graph or the Google chart API, their experience level was considered novice.

4.1.2 Tasks

Participants were asked to complete two programming tasks – each with several subtasks of increasing difficulty. Task 1 was a training task and meant to familiarize participants with the API. For the Task 1, participants were given a predefined operator-based tree layout in which they had to make small changes to modify the layout. For example, some of the subtasks of Task 1 were re-ordering the nodes or rotating the layout. Participants were also required to add and test some interactions to the visualization as a subtask of Task 1. As this first task was designed for training the participants for the actual coding tasks, they were given hints in the task specification to help them with completing the task. For example, in some subtasks of Task 1 the concrete operator to be used was

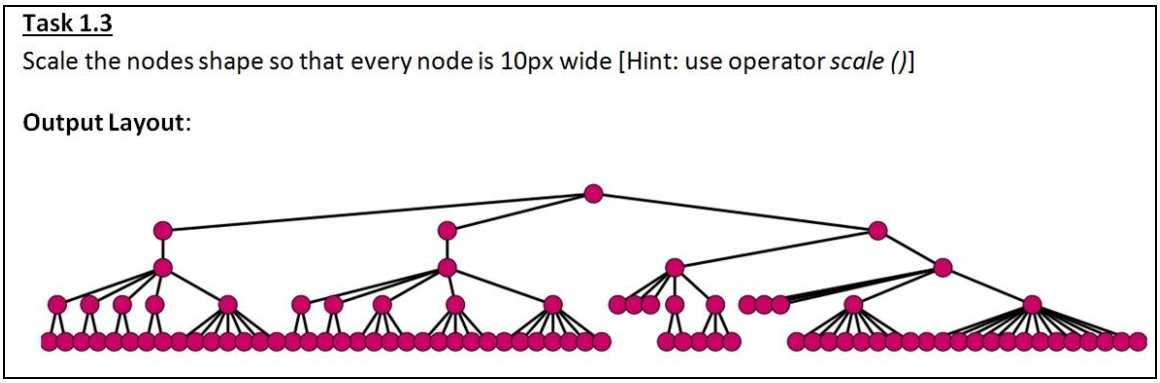


Figure 4.3: Example of a sub task of Task 1

specified and they had to find the right stage in the layout pipeline to place it in and in some other subtasks the correct stage was pointed out to them and they had to find the right operator to use. An example sub-task of Task 1 can be seen in Figure 4.3. In Task 2, the participants were handed a print-out of a desired layout and they were asked to build it from scratch.

It has been reasoned by the author that dividing the tasks as a training task and an actual task helps to bring participants quickly up to speed with the API. Use of training tasks has also been observed in the usability study performed in [21]. For all the given tasks and subtasks, participants were provided with the printout of the expected output so that they could verify whether the task was complete. A complete task description used for the study can be found in Appendix D.1: Task Description.

4.1.3 Study Setting

The study was conducted individually with each participant in a laboratory setting. The participants had no prior knowledge of the API and they were given a 15 minute introductory tutorial, after which they had to complete two the programming tasks. The participants were given a time limit of 50 minutes (**Task 1** 30 min, **Task 2** 20 min) to complete the entire programming task. Participants completed the study using a text editor (Notepad++) to write code and a web browser (Firefox) to test the outcome. Two main information sources were used in the study: the API documentation and the tutorial material. After the programming phase a semi-structured interview was conducted in which the participants were asked to comment on the challenges they experienced during the programming study. The interviews lasted 10-15 minutes.

4.2 Data Collection and Analysis

This section describes the details of the data collection and analysis method followed for the study.

4.2.1 Data Collection

For the study, four data collection techniques have been used: the think-aloud protocol; author's notes from observations, screen-capture videos, and semi-structured interviews. Code that was created by the participants during the programming phase of the study was also collected. In the think-aloud protocol [1], participants were asked to verbalize their thought process while solving a particular programming task.

After the programming phase, a semi-structured interview was conducted in which the participants were asked to comment on the challenges they experienced during the programming study. The primary reason behind keeping these interviews semi-structured was to let participants freely express their opinions and experiences about using the API. A sample questionnaire of the post study interview can be seen in Appendix F. The questionnaire works as a starting point of the conversation and many follow up questions were asked based on the participant's responses.

The screen contents, the verbalizations of the participants, and the interview sessions were captured using Camtasia[9] and a standard audio recorder. The study produced a total of eight different programming sessions and about eight hours of screen-captured videos and verbalizations of participants working with the API.

---

[9] http://www.techsmith.com/camtasia.html

4.2.2 Data Analysis

For every participant, a measurement of how successful they were in completing a programming task and the amount of time they spent completing each task were recorded. Almost all the participants used the entire allotted time for each task; therefore completion time was not a good differentiating factor for judging developer success. As a result, the decision was made to base the analysis on the rate of completion of the tasks. Accordingly, a system of three-valued success levels for the task similar to Tullis and Albert [2] was devised. Tasks could be completed, partially completed, or incomplete. Each task and sub-task were further divided into a set of granular steps that need to be completed for the successful completion of the task and determined the completion rate of those steps.

Table 4.1: Breakdown of Task 2 to determine complete, partially complete and incomplete task.

Task	**Steps for completion**	**Completed**	**Partially Completed**
2	(1) Select the *scale* operator with appropriate parameters for nesting effect. (2) Placing the *scale* operator in PRELAYOUT Stage. (3) Select *squarify* operator with appropriate parameters for allocation. (4) Placing the *squarify* operator in ALLOCATE stage. (5) Select *fill* operator with appropriate parameters for coloring effects. (6)Placing the *fill* operator in POSTLAYOUT stage.	Five successful sub-tasks including steps 1, 2, 3, 4, 6 and a partially successful one.	Four successful sub-tasks including step 1, 2, 3, 4 and a partially successful one.

For example, Table 4.1 shows the breakdown of Task 2 and how the completion rate of Task 2 is determined based on the completion rate of the individual steps. All the steps mentioned in Table 4.1 for Task 2 can be completed in any particular order by the participants and will still produce a valid output as long as all the steps are completed.

For complete details regarding the breakdown of each task and sub-tasks into the individual steps, and how their success levels were measured see Appendix E.

A task was considered to be complete if the participants completed all the major steps as explained in Appendix E. The remaining steps should also be partially completed, so that there are no major effects on the output. For example, a task has been considered complete if the participant completes all the major steps but make a typographical error

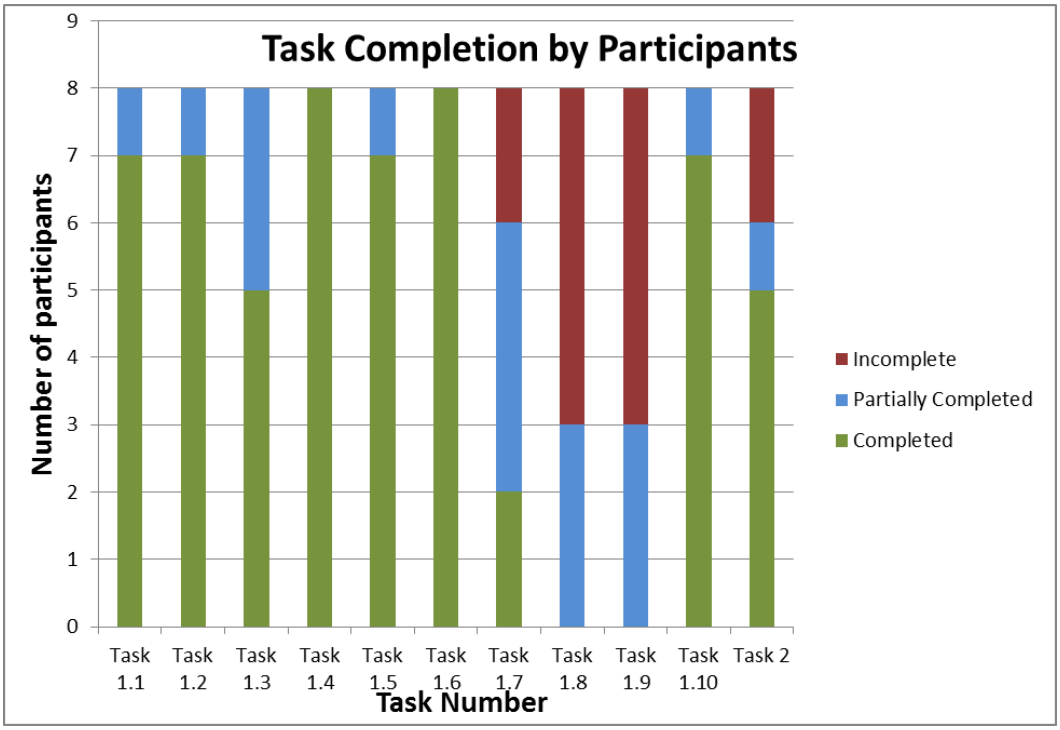


Figure 4.4: Task Completions by Participants

(e.g. missing comma or quotation) in the parameter value. As there was not much error message support for typographical errors in the preliminary version of the API, code with minor typo has been accepted as complete. The summary of task completions by participants can be seen in Figure 4.4.

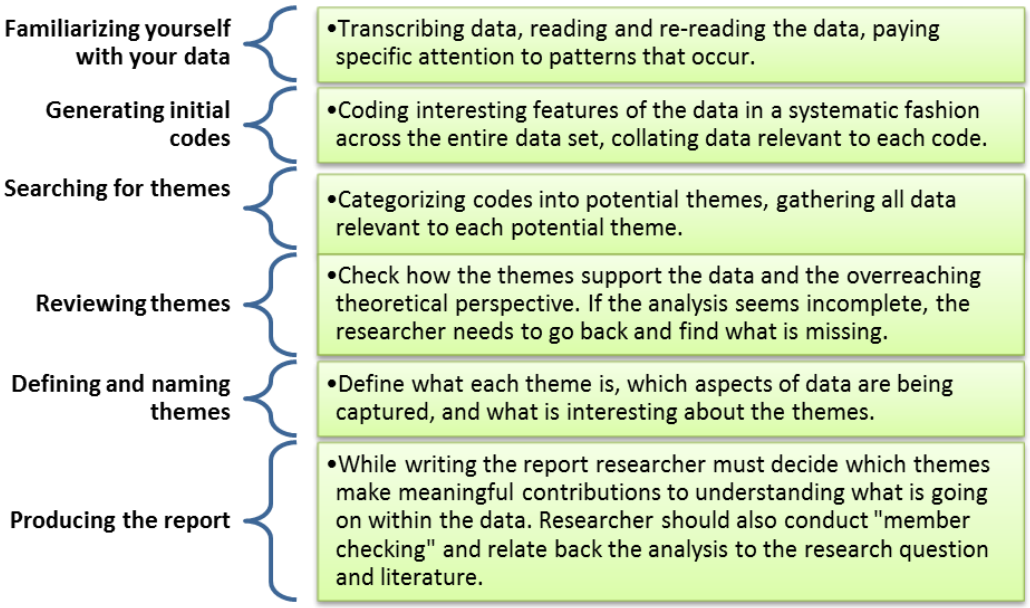


Figure 4.5: Phases of thematic analysis [15] [17]

The next step of the data analysis was to determine the challenges participants faced while using the API and identifying the main reasons behind the failure in completing a task. Ways to minimize the difficulties faced by the participants were also explored.

For the analysis a thematic analysis approach, a type of qualitative data analysis process has been followed. Thematic analysis mainly focuses on finding, examining and

recording patterns within data [17]. A theme "captures something important about the data in relation to the research question" [17]. According to Braun et al. [19] thematic analysis can be performed through the six phase analysis process as presented in Figure 4.6.

Braun et al.'s six phases of thematic analysis has been used as an inspiration for the data analysis. The author slightly adapted the analysis process to make the analysis task faster. For instance, the author did not do a thorough transcription of the think-aloud data but instead applied codes directly to think aloud and screen capture data based on the identified initial themes from the observation notes. The following subsection describes the details of the data analysis process followed by the author.

4.2.2.1 *Stage 1*: Identifying Participant's Actions from Observation Notes.

In this stage, observation notes taken by the author while conducting the usability study were reviewed. Notes were taken in a structured format for each task to record the action performed by a participant (see Figure 4.6). From the analysis of the note data, participants' actions that were relevant to answering the research question were listed. Actions like reading the task specification and viewing output were not listed as these were not relevant to the research questions.

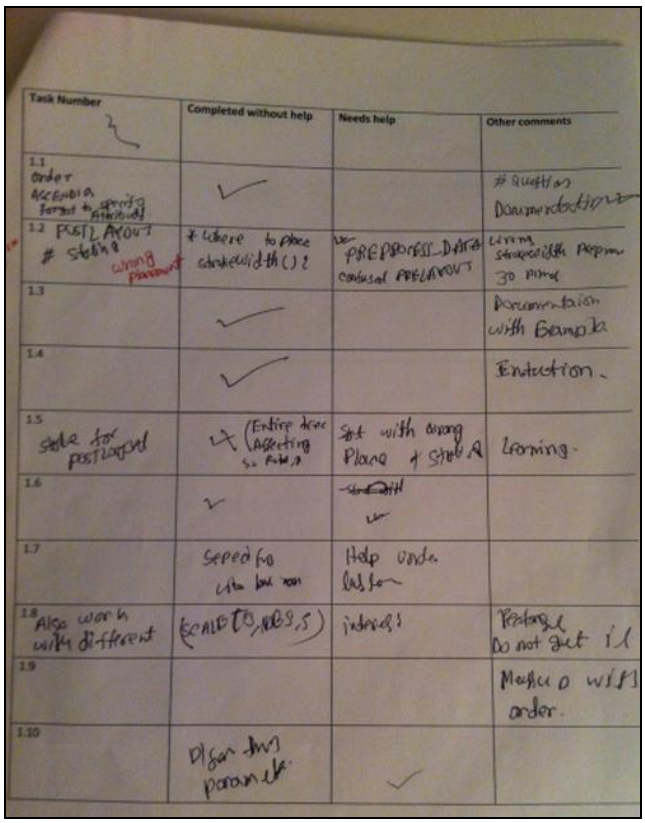


Figure 4.6: Observation notes taken by the author.

After that, the listed actions of interest were categorized into documentation-related, error- related, understandability of the API related and other relevant issues categories. Categorized actions listed by the author can be seen in Figure 4.7.

Documentation	
Consult Documentation for Definition	CDD
Consult Documentation for Relevant Examples	CDRE
Consult Documentation for API Architecture	CDAA
Consult Documentation for Stage	CDS
Consult Documentation for error handling	CDEH
Error	
Operator Misplacement Error	OME
Parameter Mistype Error	PME
Wrong operator selection	WOS
Frustration for No Feedback	FNF
Incorrect Operator Ordering	IOO
Wrong Parameter Selection	WPS
Understandability of the API	
Right Guess	RG
Wrong guess	WG
Code Reuse	CR
Lack of Domain Knowledge	LDK
Confusion about operator	CO
Confusion about naming of Stage	CNS
Trial and Error	TE
Others	
IDE Support	IS

Figure 4.7: List of actions and their abbreviation as recorded by the author.

4.2.2.2 *Stage 2*: Coding of Think-Aloud and Screen-Captured Data

After completing the *Stage 1* of the data analysis, the author went through the recorded screen capture and think-aloud data for each participant and applied codes based on the listed actions in the *Stage 1*. The analysis of the screen capture and think-aloud data helped the author identify additional issues with the API, and also gave him a better understanding of the intention behind an action performed by a participant. For each participant, a summary of their actions and issues faced was recorded.

P1	
Action	**Comments**
TE+WG+OME	Unsuccesssful trial and error
CDRE	No hints regarding stage in example.
CNS	Preprocess data or layout?
CDS	It is confusing because normally in CSS you cinfigure all style first before you draw anything but here you are applying styling later at post layout.
CNS	Pre layout mean styling and other thing that he has to be done before layout.
CDRE	Scale nodes.
CDD	Fill operator
PME	Forgot the quote for fill

Figure 4.8: Parts of the coded actions with comments for P1 from screen-capture and think-aloud data.

For example, Figure 4.8 shows for P1, the partial recorded summary of actions performed with comments.

The actions performed by the participant can result in success or failure. For example, if a participant found an element via browsing the documentation that was relevant to the task and it ultimately helped him complete the task, the action was considered a success. Otherwise, it is considered a failure. From the example in Figure 4.8, it can be seen that P1 first started with a trial and error approach based on his learning from the tutorial. After making some wrong guesses with operator selection and an operator misplacement error, P1 looked into the documentation for a relevant example to find the appropriate stage for placing the operator. Unfortunately, there was no stage information provided in

the documentation, so the action was unsuccessful. P1 then expressed his confusion about the naming of the stages of the tree layout pipeline. He said that the names of the *stages* were not intuitive for him to learn and he didn't understand where to place an operator. P1 also mentioned, how his knowledge of CSS mislead him to put the styling related operator in the PRELAYOUT stage, as he thought all the styling code, like in CSS would have to be applied before the layout is drawn and to him the PRELAYOUT stage is the stage to do that. After that, P1 looked into the documentation for relevant examples to perform a scale operation. Next he looked into the wiki for the method definition of the *fill* operator and later faced a parameter mistype error for the *fill* operator.

After the coding, frequencies of each type of action performed by a participant were also counted by the author. Frequencies of different actions performed along with the recorded comments made helped identify the usability issues faced by a participant and the reason behind those issues.

4.2.2.3 Stage 3: Transcribing and Summarizing Interview Data

After transcribing the interview data, the author went through the data to know more about the participant's overall impression of the API and the suggestions they have for improving the API and its associated documentation. Transcribed interview data has been coded using the actions listed from *Stage 1* as starting codes. Additional codes from the transcribed data, beside the mentioned codes in *Stage 1,* are listed in Table 4.2.

Table 4.2: Additional codes from the Interview Data

Codes
Provide Complete Example.
Provide details of the operator functionality.
Confusion about operator ordering
Requested new Features
Provide explicit error message.
Provide example covering more usage scenario.
Provide details of different tree layout algorithm.
Domain knowledge.
Where to place an operator?
Simplicity of the operator-based approach.

4.2.2.4 Stage 4: Generating Central Themes

Finally, the author went through the codes from *Stage 2* and *Stage 3* of the analysis to identify interesting patterns in participants' behaviors and the challenges they faced while using the API. Related instances are organized and presented as a central theme in the findings sections.

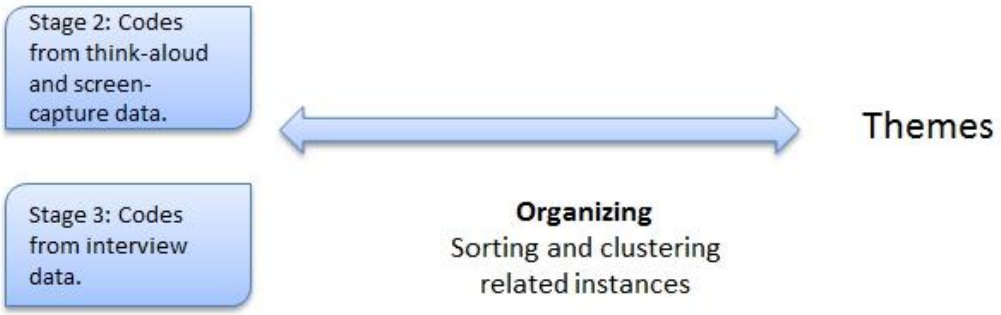


Figure 4.9: Generating main themes from coded data.

For example, codes that were related to difficulty in understanding the tree layout generation process were sorted and later clustered to generate Theme 2: "*Participants had difficulties in understanding the recursive nature of the tree layout generation process and the dependencies between different layout stages."*

High frequencies of codes, like operator misplacement error, wrong guessing about selecting an operator and confusion about an operator, as observed from the coded think-aloud and screen-capture data, suggests that participants had difficulties understanding the tree layout generation process used in AVIT. This observation was also later verified by the participants during their interview while answering to a question regarding the understandability of the operator-based tree generation process. Relevant codes from this category helped generate Theme 2.

4.3 Findings

This section provides the detailed description of the identified themes from Stage 4 of the data analysis.

Theme 1: *All participants liked manipulating the tree visualization using the operator-based approach for its simplicity and immediateness.*

In the API, the complex algorithm for generating tree layouts was hidden behind simple operators. Developers were only exposed to the operators; following the steps of the developed tree layout generation algorithm and using the operator in a certain order, they could generate a tree layout. This simplicity was highlighted by many participants, as they enjoyed not having to deal with the complexity hidden behind the operators. As P8 mentioned, "*with few instruction I can generate complex visualization which I really like very much. So I think it is very powerful, write less and do more."*
Another participant (P2) said *"I just need to understand what effects it [an operator] has on the visualization but I do not need to understand the inner workings."*

The observed simplicity is directly tied to the immediateness, as a code change means to move an operator from one place to another one, without having to care about any surrounding code and the effect can be immediately seen in the output browser window. It was remarked by P6, that *"when you have very simple commands like this it makes it easier for you to explore and try out different things."* These statements mirror the two properties of operator-based design: the declarative nature of the operators, which only specifies what, shall be done, but leaves how to the software, and their consistency in performing a particular action, which permits to experiment by shuffling them around freely [60].

Theme 2: *Participants had difficulties in understanding the recursive nature of the tree layout generation process and the dependencies between different layout stages.*

It has been observed from *Stage 2* of the analysis process, and later verified from the interview data, that participants had difficulties understanding the underlying concept of the tree layout generation process. During *Stage 2* of the analysis, it has been observed that all the participants faced operator misplacement errors and made some wrong guesses while working on Task 2. For instance P2's main problem was " *the understanding of the process of the underlying stages that the data visualization goes through and how they affect the visual representation*."

P6 said: *"... I think the most difficult things for me were to know where to put everything."*

Participants were also not clear about how placing the operator in a different order will affect the output layout. According to P8*"the clarity of putting certain instructions in certain order was not very intuitive for me. ..sometimes I did not know the right order."*

Theme 3: *Training tasks and trial and error approach were helpful to learn the API.*

Although the participants faced difficulty understanding the underlying concept as described in Theme 2, five out of eight participants were able to complete Task 2 from what they learned in the training task and just by using a trial and error approach while experimenting with different orders of the operators. As participants can see the immediate effect of the changes they made in the output browser, they can make

necessary changes in their code to generate the desired output based on their learning from their previous actions.

Since this playful approach in turn also gives them insight into the layout process, each trial was valuable as a hands-on learning experience about tree layouts.

Theme 4: *Failure due to the lack of domain knowledge about different tree layouts.*

From Figure 4.4 it can be seen that no participant was able to complete Task 1.8 and Task 1.9. They also struggled with Task 2

In Task 1.8 and Task 1.9, participants were asked to manipulate the code so that it generates a Sunburst tree layout and a node link variant of it. To successfully complete these tasks, participants needed to make the necessary changes in the code of the radial node-link tree layout (from Task 1.7), so that the layout transforms to a layered tree layout (see Section 1.1.1 for different types of tree layout). In a layered tree layout there are no explicit edges between parent and child nodes and child levels are layered constrained to parent's extent to restrict the growth in width. To complete Task 1.8, it was necessary to hide the edges between nodes. By not having this knowledge of layered tree, most participants struggled with starting with the correct step.

Also for completing Task 2, "generate a nested Squarified tree-map layout", one needs to use the *squarify("leaves")* operator in the ALLOCATION stage of the tree layout generation process. It was observed from the *Stage 2* of the data analysis that six out of the eight participants selected the wrong allocation operator in their first try of Task 2. Four out of the eight participants selected the wrong parameter for *order()* and *squarify()*

operator while working on Task 2. As P1 expressed his confusion and frustration while working on Task 2 "*which function I should use to allocate the space?" ... how could I get nesting effect by default it is [a] rectangle."*

From the background questionnaire, and post study interviews, it has been confirmed that most participants were not familiar with different tree layouts: tree-map, sunburst. It is, obviously, helpful to have some background knowledge regarding the tree layout if you want to generate that layout. For example, if someone wants to generate a slice and dice tree-map layout, it is necessary to know that allocation of space among child nodes has to be done via horizontal and vertical subdivision. Having this knowledge about different tree layouts is necessary to select the appropriate operator for the task.

Theme 5: *The effect of operator ordering on the output layout was difficult to understand.*

Participants also faced difficulty making the connection regarding how the output tree is affected depending on placing operators in a particular order in the configuration file. As P7 said while working on Task 1.9 for resizing the node shape, *"the obvious problem right now is I don't know. I don't have natural feelings for the stages yet, so I actually have no clue where and what to do. I kind of have a feeling of what I thought it should be but right now what is missing in my head is a link between at what stage does this part get resized and where I should change it."*

It has been observed from *Stage 2* of the data analysis that all participants have faced this difficulty of ordering operators.

Theme 6: *Not displaying explicit error messages was frustrating.*

For the preliminary evaluation, displaying explicit error messages was not implemented in AVIT. For example, if a user places an operator incorrectly, the system does not notify the user in any way. Not having this feature confused the participant as they had no way of knowing of the reason behind the error. Most participants were expecting a consolidated error message that will explain the root cause of the error and will provide guidance to correct the error.

As P8 expressed frustration while he misses a parameter in the function and the system does not warn him with any explicit error messages. *"I forgot to put the leaves as a second parameter and it did not give me any error messages, that is a problem but there is no distinct visual output. So I know something going wrong but don't know what is going wrong."*

In another task, P4 mentioned *"I got confused, … little syntax highlight or error message showing the system did not recognize this value will be very helpful"*. P3 and P7 also mentioned the need for syntax highlighting and auto complete features.

Theme 7: *Confusion about operators.*

There are two different operators in the API to perform a scaling (resizing) operation for nodes, shapes and edges − *scale_to* and *scale_by*. Operator *scale_to* does an absolute scaling while *scale_by* does a relative scaling. The *scale_by* operator is mostly used for creating borders/nesting effect where one wants to shave off a few pixels along the

border, whereas the *scale_to* operator is used in most cases for encoding parameter values (such as depth in the tree, number of siblings etc.) directly in a node's shape.

It has been observed, in *Stage 2* of the analysis, that participants struggled to understand when to use which scale operator to have a desired effect on the output.

As P6 mentioned: *"it was kind of confusing to a naïve user like me understanding what a scale_to and scale_by does? If you can make it more explicit in the documentation via examples showing when I say scale to top, what it actually does so give example of each combination with figure, so that user can understand what it actually does"*
P2 said *"If I scale it in ALL direction and only top, I don't see any difference in the output which is weird."*

Theme 8: *Limitation in customizing the implemented interaction features.*

In the preliminary version of the API, the option of adding interaction was provided via a single method named *addEvent()*. Different interaction options can be added by selecting different parameters to the *addEvent()* method. For example, if someone wants to show menu interaction on mouse click, s/he just needs to call a function *addEvent("click", addMenue).* It has been observed from the data analysis that participants found it easy to add different interactions option using this approach and the completion rate of interaction related task (Task 1.7 and 1.10) was also very high.

However, the then-current implementation of the interaction feature only provided some default interaction options like highlighting sub-trees, and displaying node attributes. No options were provided to add customized or new interaction features.

Three out of eight participants stated that it would be really nice to have flexibility in adding new or customized interactions implemented by the developer using the API. This was difficult using the current approach used in the AVIT as it hides all the details of the implementation code from the developer.

4.4 Suggested Improvements

This section describes the suggested improvements for the API and its associated documentation materials to address the identified usability issues of AVIT. The improvements have been suggested by the author based on the findings from the preliminary evaluation.

4.4.1 Interactive Demo Tutorial

To provide better understanding of the tree layout generation process, an interactive demo tutorial with complete example code for different tree layouts can be provided as learning material in addition to the documentation. The demo tutorial will be a good starting point for the developer who wants to use AVIT for drawing trees. The interactive tutorial will allow making changes in the demo code and will show the effect on the output layout right away. It will help developers understand the effect of different operators and their various parameters on the output tree layout. Also, in the documentation, adding the details of the API architecture and functionality of individual modules with a step-by-step example will provide better understandability of the API [4]. The author expects that interactive demo tutorial will help improving the usability issues as mentioned in Theme 2 and Theme 5 and will provide better learnability of the API.

4.4.2 Displaying Explicit Error Message

To address the operator misplacement error mentioned in Theme 6, explicit error messages can be shown describing the cause of the error with suggestions regarding what needs to be done to fix it. Also, the documentation can be updated with recommended stage information for an operator. This will help minimize the confusion due to operator misplacement errors. As P7 commented during the post-session interview, *"when I was trying to make changes and I can't see anything as output, that's kind of frustrating, it should show at least something in there, if not it should show at least an explicit error message that says you need to finish these."*

4.4.3 IDE Support

To address errors due to the missing parameter or typographical mistakes as described in Theme 6, IDE support for the API code can be provided. If features provided by modern IDEs – like auto complete and syntax highlighting – can be added, it will help minimize the missing parameter and typographical error.

4.4.4 Detailed Operator Documentation

As mentioned in Theme 7, all participants found the scale operator difficult to understand. A detailed example showing the effect of placing *scale_to* and *scale_by* operator in different stages of the layout generation process can be added in the documentation, to increase understandability. Similar explanations can also be added for other operators in increase the understandability.

To address the issue regarding operator ordering as mentioned in Theme 5, additional examples showing the effect of placing an operator in different stages can be added in the

operator documentation. As P2 suggested during the post-session interview regarding operator ordering, *"show the effect of the positioning of different operators, like give a visual example of how the API is used, for example if you make scale here it makes this smaller but if you make scale there it makes it bigger. So you have to go to a point where the user understands how the different stages affect the visualization. That was my main problem."*

4.4.5 Documentation on Tree-Layout

Short descriptions of different types of tree layout, like the sunburst and tree-map should be added to the documentation. This will help developers who do not have domain knowledge about tree visualization to quickly learn what operators they need, to draw a particular type of tree layout.

4.5 Limitations of the Evaluation

Participants had no previous experience with the API and most of them were not familiar working with data visualization tools. As the target audience of the API is the general developer who does not have a background in visualization, it makes sense for the usability study to gather feedback from such developers. However, it also means that our results provide limited insights into the behavior of the developer who has years of experience working with visualization tools.

The time limitations of the study for each task also limit the validity of our results. It has been seen from the studies done by other visualization toolkit researchers that it takes days or sometimes even weeks to get a good understanding of their visualization toolkit and building useful visualizations with them [11]. Also, in a real development scenario, a

developer might have more time to spend with the API and, over time, might have a better understanding of the API to complete the task. Although, given the lab setting and pre-defined tasks, the findings cannot be considered complete, but it was a good starting point to identify some major usability issues with the API and the documentation.

The study consisted of only eight participants. This suggests that the findings from the study can't be generalized. However, given the preliminary nature of the study and the goal of discovering major usability issues with the API and its documentation, the findings informed the next steps of the API development and evaluation.

4.6 Discussion

In exploring the answers to the research questions listed at the beginning of this chapter from the analysis results, the first research question about *"Which of AVIT's features do developers like? "* has been answered in Theme 1 in Section 4.3. It has been seen in Theme 1 that participants liked the simplicity and the conciseness of the operator-based tree drawing approach very much.

In answering the second research question regarding *"What difficulties do developers face while using the operator-based approach to drawing tree layouts?"*, it has been found from the analysis results that participants had difficulties understanding the tree layout generation process, correct operator ordering and recovering from an error. Details of the difficulties faced are explained by Theme 2, Theme 5, and Theme 6 in Section 4.3.

The next research question was about *"How useful are the given documentation and tutorial materials for finding task related information?"* In general, the documentation

and training task was found helpful to learn about operator definition but lacks in providing any detailed description about the functionality of different operators and their effect on the output layout.

Finally, to answer the research question *"How can AVIT be improved to better support developers' expectations?"* improvements have been suggested as mentioned in Section 4.4.

4.7 Chapter Summary

The preliminary study was conducted to get early feedback from developers regarding the usability of the API. From the results, it can be seen that although participants liked the simple, playful nature of the API, they had difficulties understanding the underlying concept of the tree layout generation process. Based on these findings, several improvements have been suggested to update the API and the documentation to improve the usability. To determine the usability of the updated API and its documentation a second evaluation was undertaken. The next chapter discusses the results of the second evaluation conducted on the updated API.

Chapter Five: Usability Study 2: Improving Usability Experience of AVIT

Based on the findings from *Study 1* (see Chapter 4: Findings), several ideas for improving AVIT and its supporting documentation materials have been suggested. AVIT and the documentation materials have been updated based on those ideas (see section 5.1 for details of the changes made). This chapter describes a second evaluation of AVIT, conducted to gain further insight into developers' reactions to the updated API.

The second evaluation has been conducted to address the following research question "*How do the changes made in the API affect the usability experience?"* The broad research question can be broken down further into the following questions.

- Does updated supporting material improve the learnability of AVIT?
- How effective were the examples provided in the documentation?
- How can the interaction features implemented for AVIT be improved?
- What other possible interactions can be added in AVIT?
- Can this evaluation identify additional usability issues?

5.1 Changes Made in the API

Findings from *Study 1* identified usability problems with AVIT and the supported documentation materials (see Section 4.3: Findings). Suggestions derived from the analysis of *Study 1* (see Section 4.4: Suggested Improvements) were used to update the API and supporting documentation. This section provides details regarding the changes made by the author in the API code and the documentation materials for the second

evaluation. Section 5.2.1 and 5.2.2 discusses the changes made in the documentation materials while Section 5.2.3 and 5.2.4 discusses the changes made in the API code.

5.1.1 Interactive Demo Website for Tree Layout

An interactive demo website with some full tree layout examples has been added as support material for the API.[10] In the interactive demo, a user can make changes in the code section (Figure 5.1(3)) and then re-run the layout to see the effect of the change in the output section (Figure 5.1(4)). A list of example tree layouts (Figure 5.1(1)) along with different datasets has been provided (Figure 5.1(2)) in the demo. A user can browse through different example tree layouts along with their associated code just by clicking the link on the left.

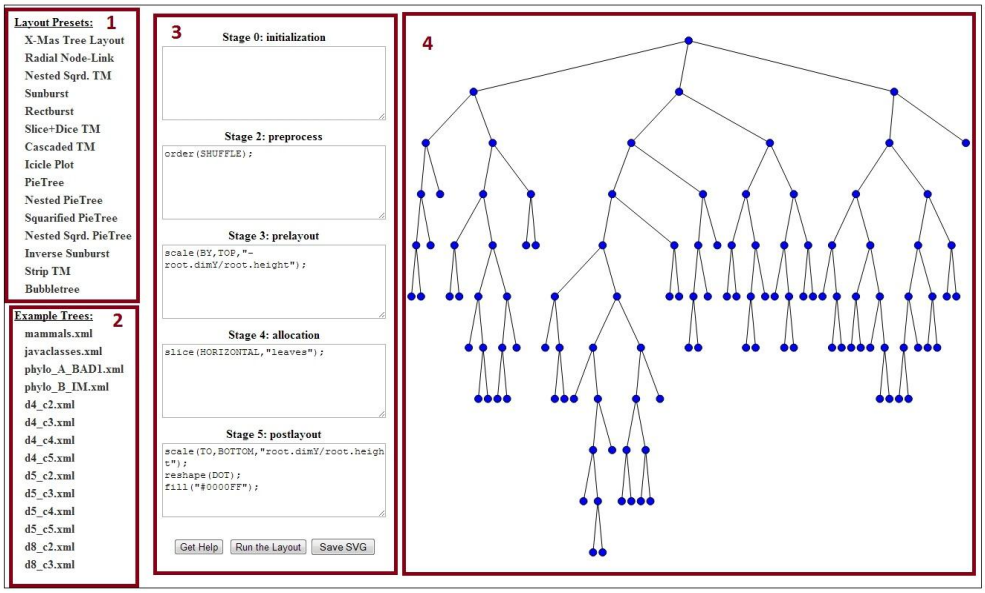


Figure 5.1: Screenshot of the interactive demo website.

[10] http://tinyurl.com/operatordemo

5.1.2 Updated Wiki Documentation

The Wiki for the API was updated with a more detailed description of the functionality of the operators, the recommended stage to put the operator and with detailed example code covering different usage scenarios. [11] A screenshot of the wiki can be seen in Figure 5.2. At the top, navigation links to switch between different operator definitions are provided. In Figure 5.2, the *order* operator has been selected and details of the *order* function definition, parameter details along with example code can be seen.

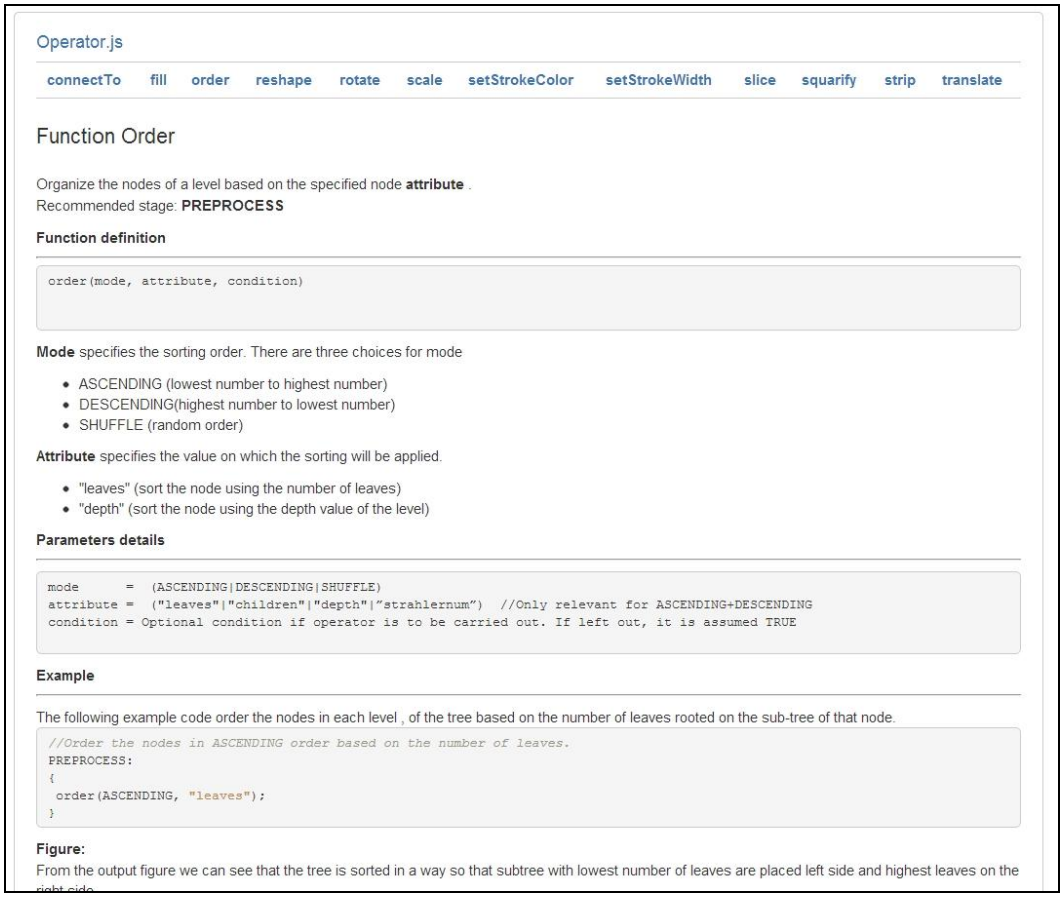


Figure 5.2: Screenshot of the API documentation.

[11] http://tinyurl.com/operatordocs

5.1.3 Error Messages

Error handling code for operator misplacement has been added in the API. For example, from Figure 5.3 it can be seen that putting *order (SHUFFLE)* in the initialization stage and then running the layout displays an error message. The error message shows the operator placement error along with a recommendation how to fix the error. Some of the other error messages added in AVIT were for leaving all the stages in the layout specification blank and for not selecting any allocation operator.

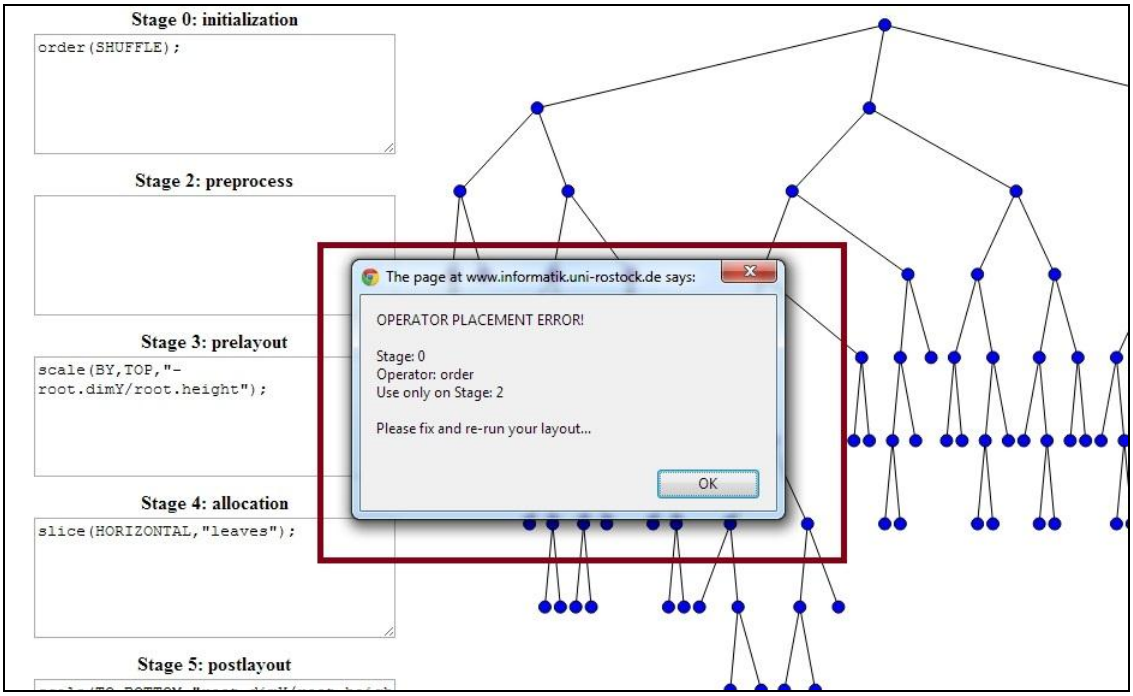


Figure 5.3: Operator misplacement error message.

5.1.4 Renaming of Layout Stage

The ROOT_LAYOUT stage in the configuration file from the preliminary study has been renamed to INITIALIZE. The reason for the change is that, before, it was implied that the overall pipeline needed to start with the root. While this is true for the implementation, the operator-based concept is much broader and would also accommodate for a bottom-to-top layout, which starts with the leaves instead of the root. In order to avoid confusion, this stage has been renamed to INITIALIZE.

Some of the suggested improvements from the preliminary evaluation have not been incorporated in the updated API due to time limitations. For example, IDE support for the configuration file is still not provided by the updated API. Furthermore, documentation describing different tree layout algorithms was not provided considering the limited time of the usability study and having the option to find the relevant information by searching on the web.

5.2 Study Setting

The following subsections provide the details of the methodology followed for the second usability study.

5.2.1 Participants

A new group of participants were recruited from the Computer Science student population at the University of Calgary and from industry by using mailing lists. Monetary compensation of $20 was offered for participation. Respondents were pre-screened using a questionnaire and selected based on relevant programming experience. To be included in the participant pool participant must have had at least 1 year of

programming experience. To avoid the learnability effect, participants who participated in the preliminary evaluation were not eligible to participate in the second evaluation.

The study consisted of twelve participants (referred to as P1…P12). All the participants have at least 3 years' experience with programming. Participants included two people from industry, two PhD students, seven M.Sc. students, and one senior undergraduate student. Although most of the participants were from academia, their expertise level is anecdotally comparable to that of recent graduates in software development positions. A summary of the participants' backgrounds is presented in Figure 5.4 and Figure 5.5. Figure 5.4 shows a chart with the proportion of participant from industry and academia while Figure 5.5 displays a bar chart of participants programming experience in terms of years.

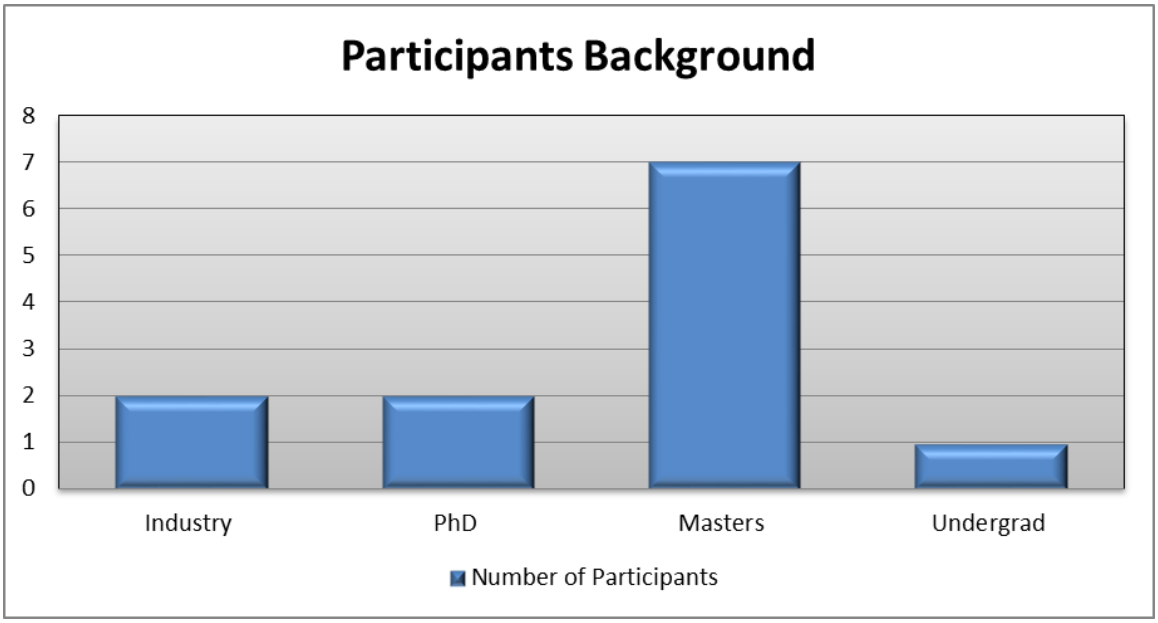


Figure 5.4: Participants Backgrounds

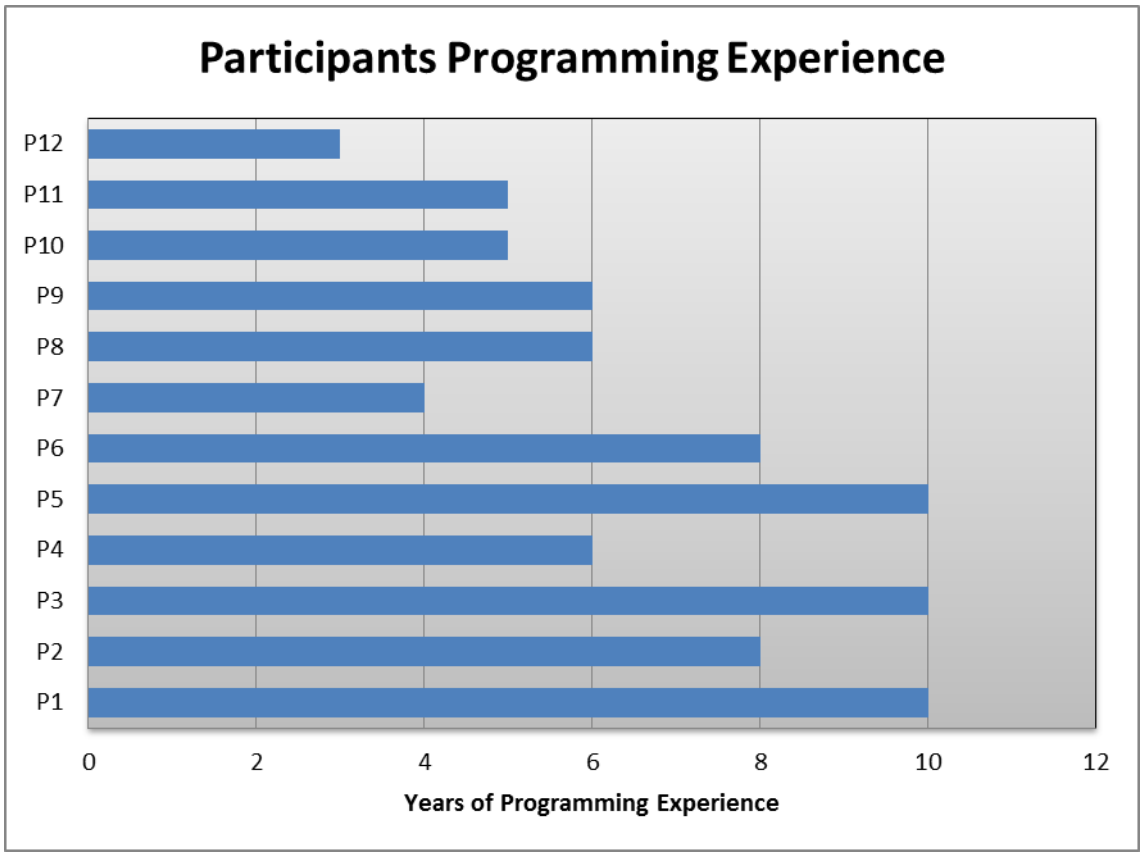


Figure 5.5: Participant Programming Experience

As described in Chapter 4, from the background questionnaire asked during the evaluation, I found that participants had different levels of experience with visualization tools. A proportion of participant's levels of experience are presented in Figure 5.6.

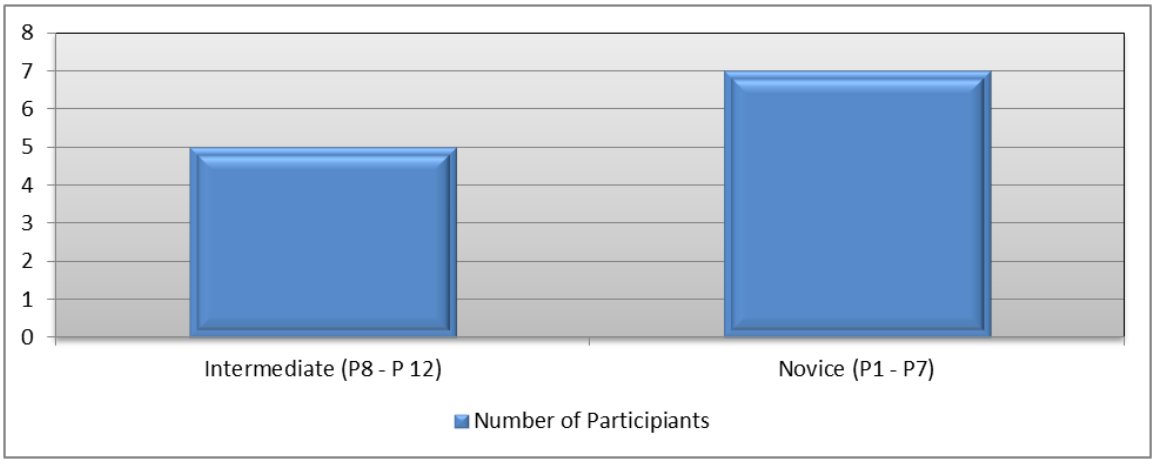


Figure 5.6: Participants experience with visualization tools

5.2.2 Tasks

Participants were asked to complete three programming tasks with several subtasks of increasing difficulty. Task 1 was a training task, Task 2 was a tree layout generation task and Task 3 was an interaction related task.

Task 1 was designed to familiarize participants with the API. For Task 1, participants were given a predefined, operator-based tree layout in which they had to make small changes to adapt it − for example some of the subtasks of Task 1 were re-ordering the nodes or rotating the layout. Participants were also required to add and test some interactions to the visualization as a subtask of Task 1. As this first task was designed for training the participants with the actual coding, they were given hints in the task specification to help them with completing the task: In some subtasks of Task 1 the concrete operator to be used was provided to them and they had to find the right stage to place it in, and in some other subtasks the correct stage was pointed out to them and they

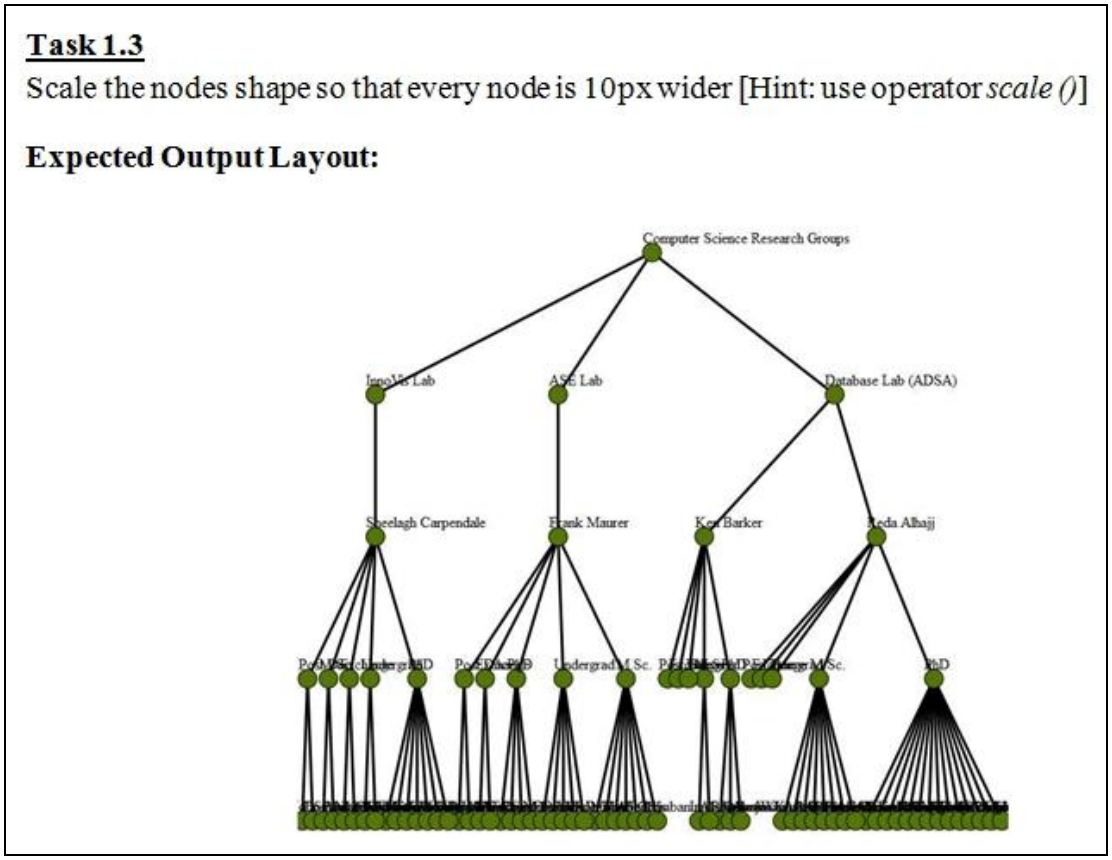


Figure 5.7: Example of a training sub-task

had to find the right operator to put there. An example of a subtask of Task 1 can be seen in Figure 5.7. In comparison to study 1 some of the sub-tasks of Task 1 were enhanced with a detailed explanation of the outcome of the change made to provide better understandability of the operator function.

In Task 2, the participants were handed a printout of a desired layout similar to study 1 and they were asked to build it from scratch. Details of Task 2 can be seen from Figure 5.8.

Task 2

Nested **Squarified** Treemap is type of treemap layout with offsetting/gap between successive levels that produce the **nesting effect** [Child nodes are embedded within parent nodes]. It **allocates** the space between nodes in a way so that it resembles a **square**.

Modify the config file for Task 2 to generate the following layout.

Expected Output Layout

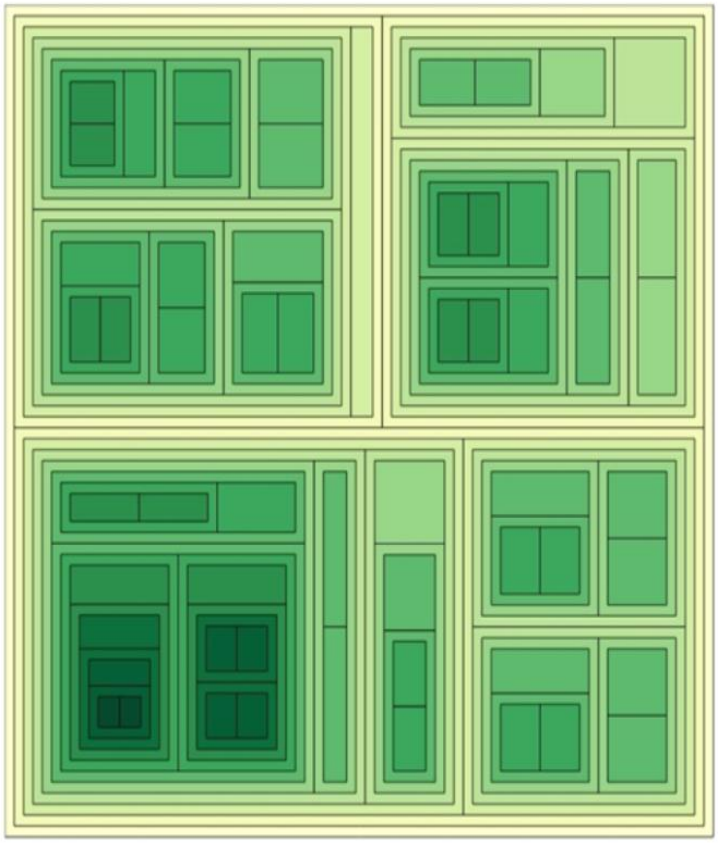


Figure 5.8: Generating Tree layout (Task 2)

In Task 3, participants were directed to add some interactions to the visualization. An example of a subtask of Task 3 can be seen in Figure 5.9.

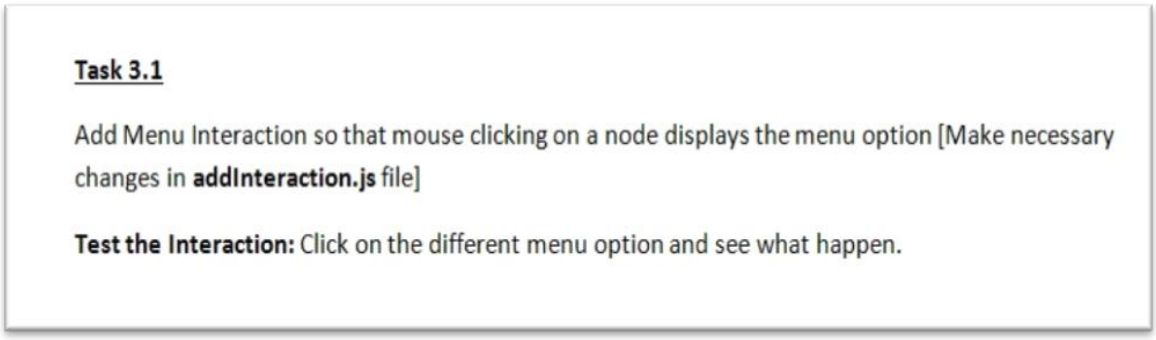


Figure 5.9: Interaction sub task.

The author observed that dividing the tasks into a training task and an actual task helps to bring participants quickly up to speed with the API. For all the given tasks and subtasks, participants were provided with a printout of the expected output so that they could verify whether their task is complete.

The complete task description used for the study can be seen in Appendix D.2.

5.2.3 Study Setting

The study was conducted individually with each participant in a laboratory setting. The participants had no prior knowledge of the API and they were given a 15-minute introductory tutorial, after which they had to complete some tree layout tasks using the operator-based approach. The participants were then given a time limit of 50 minutes (**Task 1** 15 min, **Task 2** 20 min, **Task 3** 15 min) to complete the entire programming task. Participants completed the study using a text editor (Notepad++) to write code and a web browser (Firefox) to test the outcome. Two main aids were used in the study: the API documentation and the interactive demo materials. After the programming phase, a semi-structured interview was conducted in which the participants were asked to

comment on the challenges they experienced during the programming study. The interviews lasted 10-15 minutes.

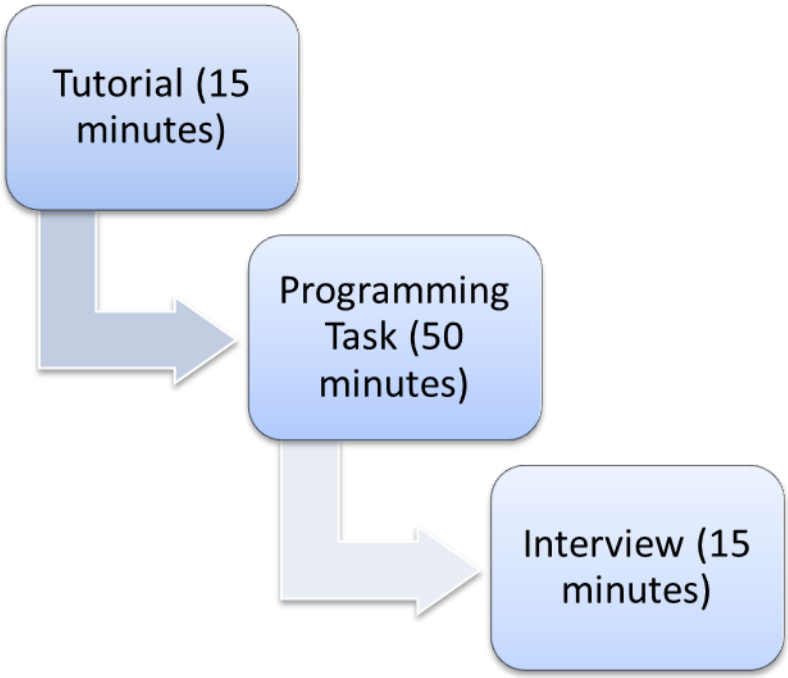


Figure 5.10: Overview of study setup.

5.3 Data Collection and Analysis

This section describes the details of the data collection and analysis method followed for the study.

5.3.1 Data Collection

For the study, four data collection techniques have been used: the think-aloud protocol [1], structured notes from the observations, screen-capture videos, and semi-structured interviews. Code that was created by the participants during the programming phase of the study was also collected. In the think-aloud protocol [1], participants were asked to

verbalize their thought process while solving a particular programming task. The think aloud protocol gave insight about understanding why a participant experienced difficulty completing a certain task. Comparisons between participants' verbalized thought processes and the design decisions made for the tree layout generations were also made possible.

After the programming phase, a semi-structured interview similar to the preliminary study was conducted. The screen contents, the verbalizations of the participants, and the interview sessions were captured using Camtasia[12] and a standard audio recorder. The study produced a total of 12 different programming sessions and about 12 hours of screen-captured videos and verbalizations of participants working with the API.

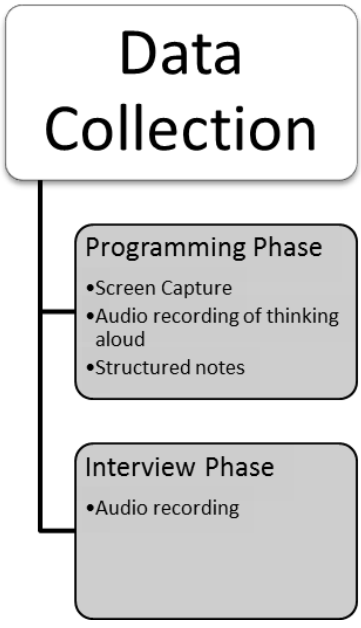


Figure 5.11: Overview of data collection process.

5.3.2 Data Analysis

For every participant a measurement of how successful they were in completing the programming tasks was recorded. For the measurement, Task 2 and Task 3 have been considered. Task 1 was a training task and was not considered for the evaluation. Eight of the twelve participants successfully completed Task 2. Task 3 was successfully completed by all the participants.

Figure 5.12 shows the Task 2 completion time by the participants with a cutoff time of 20 minutes. P1, P2, P6 and P9 failed to complete the Task 2 within the time limit as highlighted red on the chart in Figure 5.12.

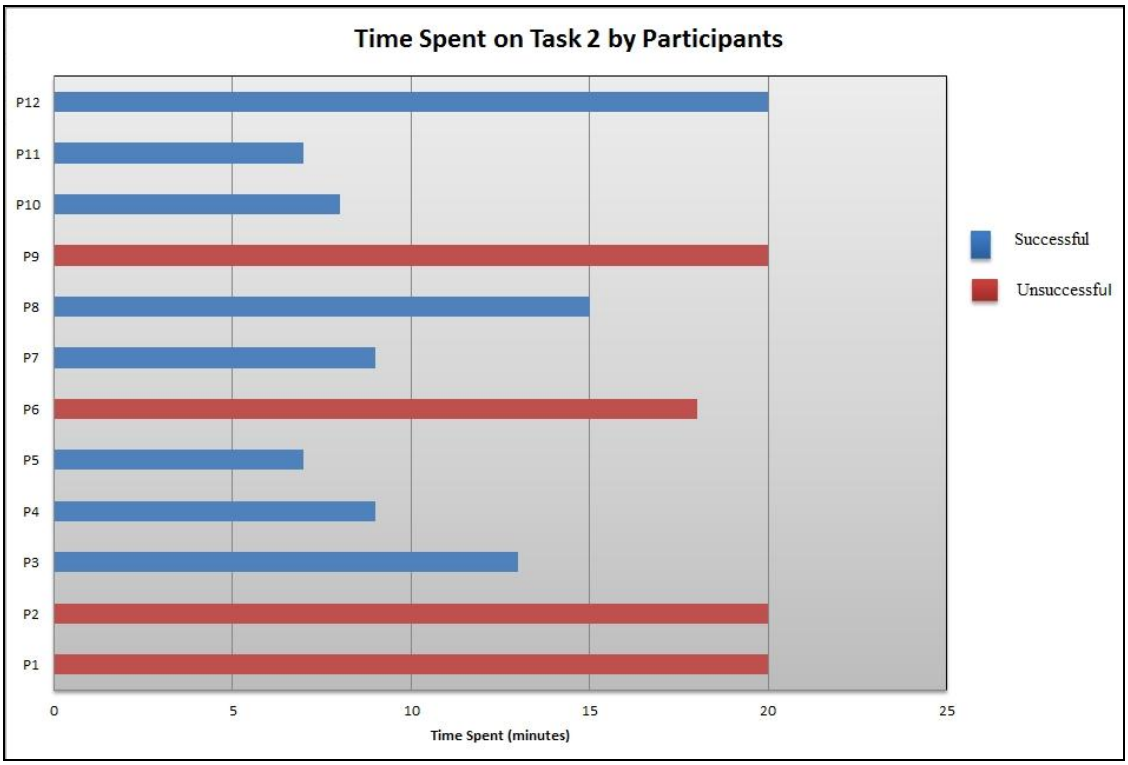


Figure 5.12: Task 2 completion time by participants.

---

[12] http://www.techsmith.com/camtasia.html

A descriptive statistics analysis using Microsoft Excel with task completion time for the participants produces the following result.

Task Time	
Mean	11
Standard Error	1.636634177
Median	9
Mode	9
Standard Deviation	4.629100499
Sample Variance	21.42857143
Kurtosis	0.66864
Skewness	1.209738264
Range	13
Minimum	7
Maximum	20
Sum	88
Count	8
Confidence Level(95.	3.870024865

Figure 5.13: Descriptive statistics analysis for Task 2 completion time.

It can be seen from Figure 5.13 that on average a participant took 11 minutes to complete the Task 2. From the confidence interval calculation in Figure 5.13, it can be said with 95 percent confidence that the population mean is 11 minutes plus or minus 3.8 minutes.

Findings from the screen capture data also suggest that a participant who use the help from the interactive demo tutorial while working on the Task 2 has a higher success rate in completing Task 2. They also took less time than other participants who did not use the interactive demo but were able to complete the task as shown in Table 5.1.

Table 5.1: Participants using the live demo (red means did not complete the task).

	Participants did not use Live Demo	**Participants used Live Demo**
Participant Number	P1,P2, P3, P6, P8, P9, P11, P12	P4, P5, P7, P10
Average completion time in minutes	13.75	8.25

While asking the other participants for the reason behind not using the interactive demo, some of them replied they forgot about it, and one participant said it seemed time consuming for him to copy code from the demo to the actual code. However, the implication from the screen capture data analysis by the author is that developers may not yet be used to interactive demo based documentation. When they face any problem, many just consulted the main wiki documentation and did not look into the additional interactive tutorial provided. It suggests that embedding the interactive example in an appropriate place in main wiki documentation might be more helpful than giving it as a separate source.

The next step of the data analysis was to determine how participants reacted to the API, the difficulties they were facing while completing the tasks and measures taken by them to overcome those difficulties.

As described in 4.2.2.1, the author noted a list of actions performed by each of the participants during the programming session of the study. While conducting the analysis

of the screen capture and think aloud data, in addition to the actions described in the preliminary study, the following actions by the participants were points of interest for answering the research questions.

- Playing with the interactive demo to understand the API.
- Comments regarding uncertainty of the outcome of a piece of code.
- Comments regarding addition of different interaction features to the layout.

For analysis of the data, a thematic analysis approach as described in Figure 4.5 in Chapter 4 has been followed to generate themes from the data [14]. The following subsection discusses the phases of the thematic analysis as applied in the data analysis for the study.

5.3.2.1 Phase 1: Data Familiarization

The author went through the collected think aloud and interview data and completed the transcription. All the transcriptions have been put into the Saturate application [16] for further analysis. Figure 5.14 shows a screenshot of a transcribed data from P6 in the Saturate application.

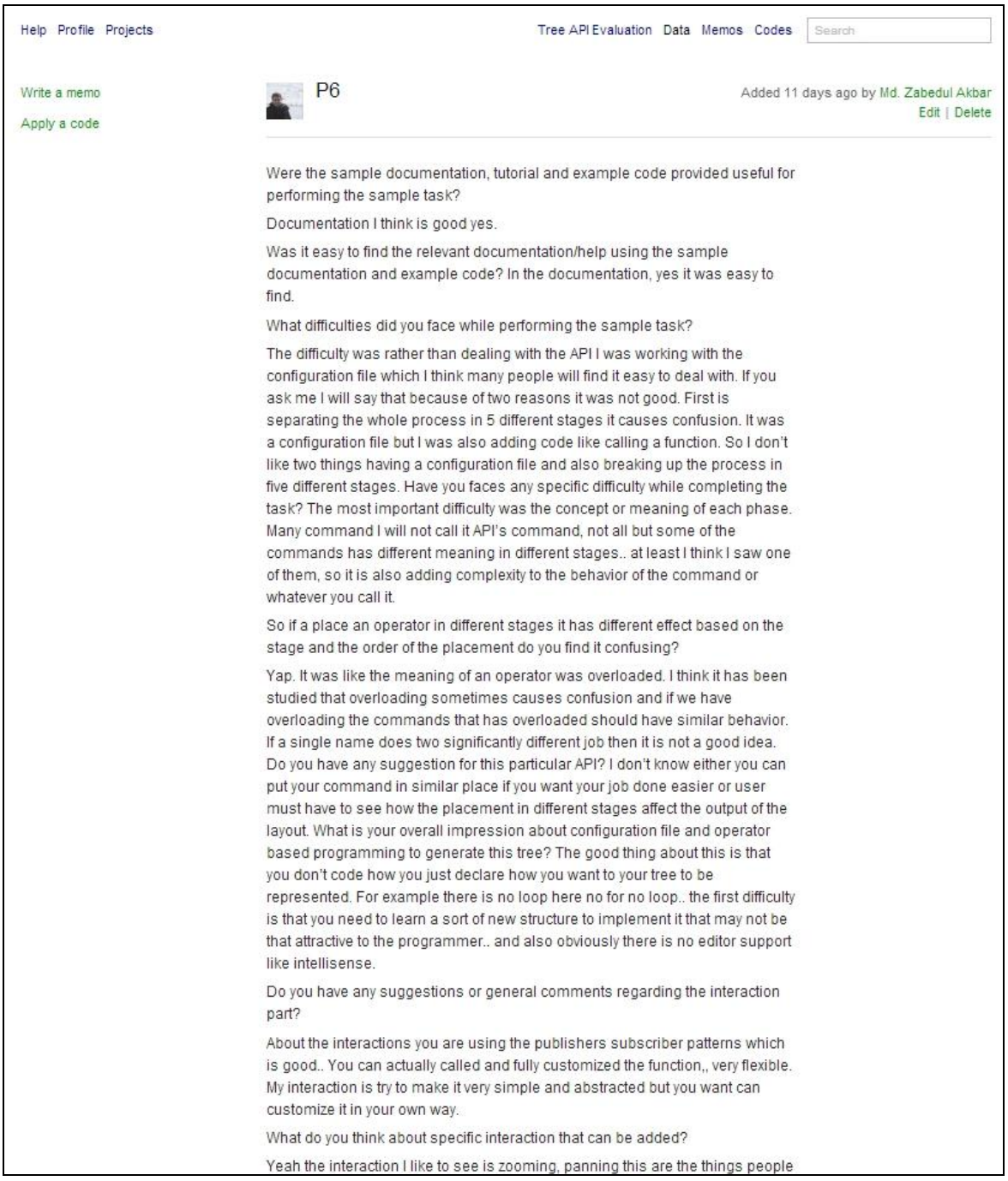


Figure 5.14: Transcribed data from P6 as stored in the Saturate application.

5.3.2.2 Phase 2: Generating Initial Codes

After the transcription was completed, the author went through the transcribed data to find interesting parts that are related to the research questions. Relevant codes were applied to interesting parts of the data. For example, while going through the transcribed data any quotes from participants that mentioned the usefulness of the interactive demo were tagged with a code *"interactive demo was very useful"*. Figure 5.15 listed all the quotes from different participants that were coded with *"interactive demo was very useful"*.

After completion of the initial coding phase a total of 61 initial codes had been recorded. Some of the applied codes along with their data source can be seen in Figure 5.16.

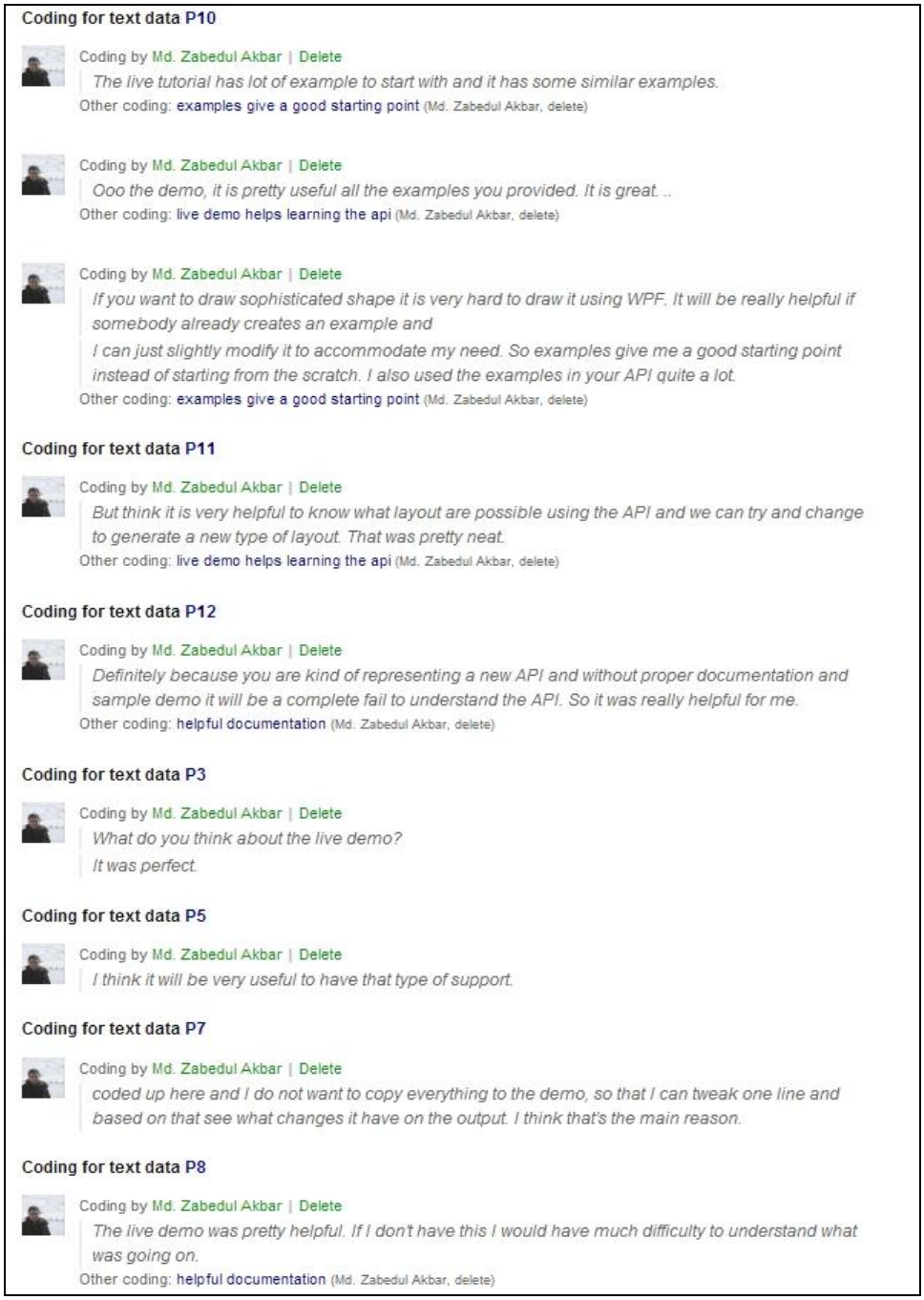


Figure 5.15: Data for the initial code *"interactive demo was very useful"* from the Saturate application.

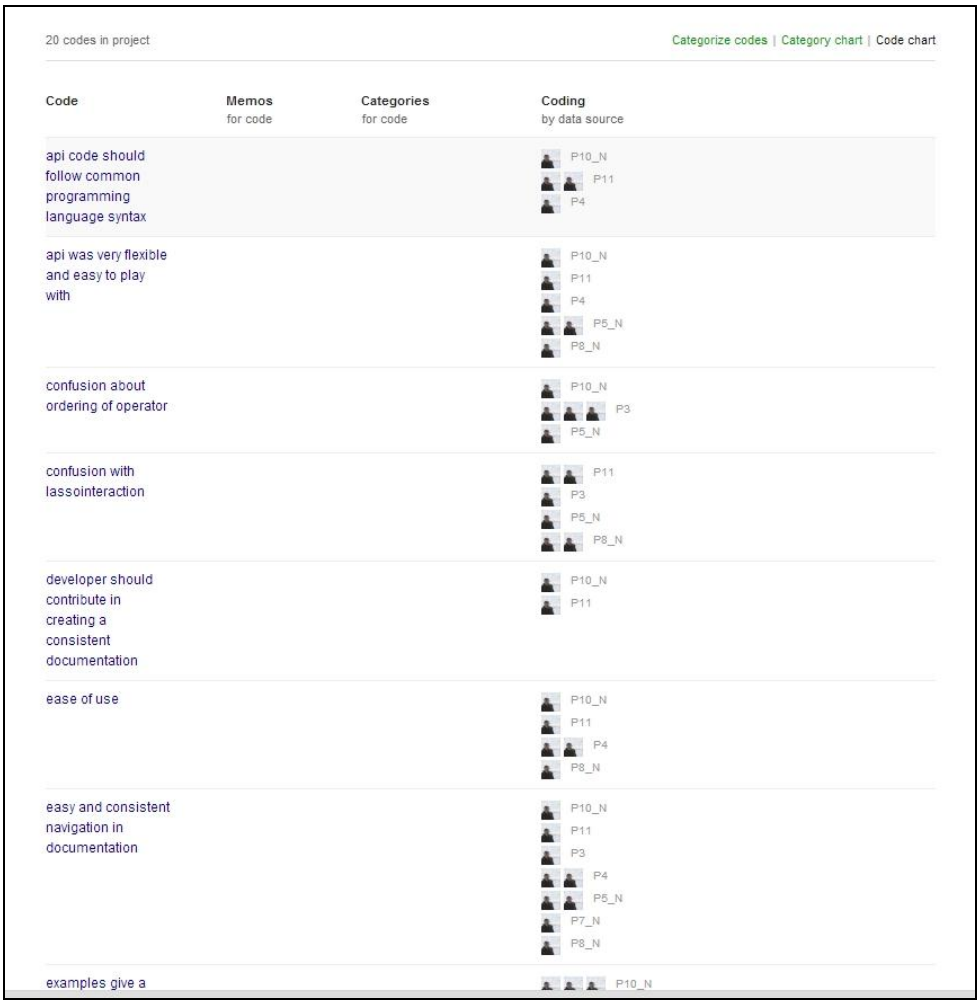


Figure 5.16: Some applied initial codes.

5.3.2.3 Phase 3: Searching for Themes

After finishing the initial phase, codes were categorized into potential themes. For example, one of the potential themes identified was *learnability of the API*. Any code that was relevant to learning the API was listed under the learnability category. Figure 5.17 shows the codes listed under the learnability category.

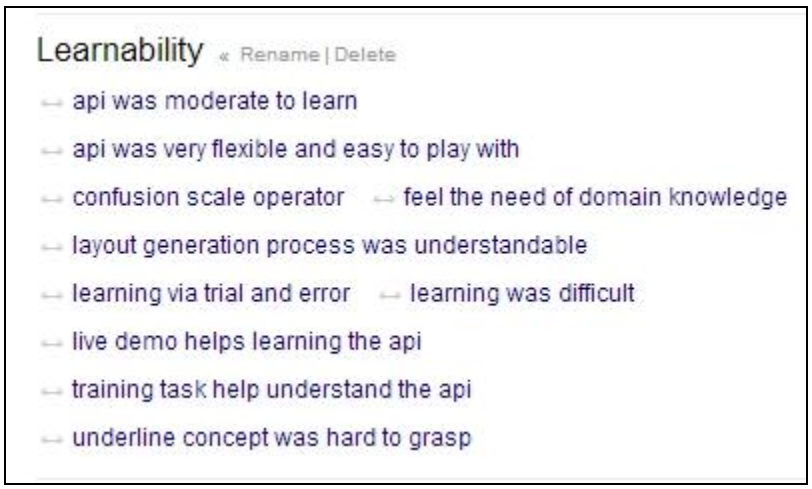


Figure 5.17: Theme Learnability.

All the potential themes identified after this phase can be seen in

Table **5.2**. For example, codes that provide suggestions regarding how the documentation can be improved are

Table 5.2: Potential Themes

Potential Themes
Learnability
Usefulness of the documentation
Issues with Documentation
How can documentation be improved?
Good feature of the API
Issues with API Code
How can Interaction be improved?
Additional issues (IDE, auto-complete)

listed under the theme *how documentation could be improved?* Codes that are not related to the API or the learning materials but which affect the usability of the API were categorized under the theme *additional issues*.

From each potential theme and its relevant codes, several main themes have been generated. For example, from the potential theme *issues with documentation,* Theme 4 and Theme 5 have been generated to discuss different issues with the examples provided in the documentation as described in Section 5.5.

5.3.2.4 Phase 4: Reviewing Themes

To reduce biasing effect, all the themes were reviewed and refined in consultation with another researcher, who went through the applied codes and raw data and agreed with all the codes applied by the author. Some of the codes were combined to generate a new code while others were discarded as not being relevant to the research question. Conflicting codes were resolved by going back to raw the data and identifying the main reason behind the conflict.

For example, the conflict between the codes *"underlying concept was hard to grasp"* and *"layout generation process was understandable"* was resolved by careful analysis of the raw data. From the data, it was noticed that seven participants have been coded with the code *"underlying concept was hard to grasp"* while only one participant (P11) was coded with *"layout generation process was understandable"*. To find out why P11 has a

different opinion about the API, a detailed analysis of the think aloud, interview and background data of P11was performed.

From the data, it was clear that P11 had 2 years of research experience working with tree visualization and was quite familiar working with some graphics framework like OpenGL. P11 also mentioned that his knowledge with tree visualization and graphics framework helped him to understand the API as he said *"I think also my domain knowledge with OpenGL helps me to understand the functionality of different operators and where to place them."*

While it was an interesting finding that suggests that evaluating the API with a domain expert might have a different result in terms of usability of the API, I did not presented it as a main theme in the findings as I have only one participant having research experience in tree visualization. Also, the target audience for the API is developers who do not have domain knowledge in this area. This issue needs further exploration, probably by conducting a usability study with domain experts.

Phase 5 and 6 (as explained in Figure 4.5) of the thematic analysis process have been combined and represented in the findings section.

5.4 Findings

In this section, themes identified in Phase 4 of the thematic analysis are defined and presented in detail. How the presented themes help answer the research questions is also discussed. Themes 1 to Theme 8 are related to the usability of the API while Themes 9 and 10 describe the user experience with the API.

Theme 1: *Interactive demo and training task was helpful for learning the API.*

Relevant codes supporting this theme are listed in

Table 5.3.

Table 5.3: Relevant codes for Theme 1

Relevant Codes	**Participant Frequency**
Live demo helps learning the API	7
Training task helps understand the API	3
Learning via trial and error	7

The interactive demo was a good starting point to learn about the power of the API. In the interactive demo, many example tree layouts along with their associated code were provided. Participants can also make changes in the code and see the effect right away in the output panel. Most participants found it very useful in learning and exploring the API. As P11 expresses his satisfaction regarding the demo tutorial *"it was very helpful to know what layouts are possible using the API and we can try and change to generate a new type of layout. That was pretty neat."*

The purpose of adding a training task in the programming session was to facilitate the learning of the API. Many participants found the training task very helpful in learning the API. As P12 mentioned *"I also think the training task was helpful for me to understand the API better, by trying and making changes in the given configuration file."*

Observations very similar to the preliminary evaluation were also observed in the second evaluation: “because of the simplicity and immediateness of the operator-based programming trial and error played an important role in learning the API”.

From P8 *“This interactive demo really helps me to get insights via trial and error as I can see the output right away in the browser. It was trial and error and learning at the same time for me.”*

Theme 2: *Difficulty in understanding the underlying concept of the tree layout generation process.*

Relevant codes supporting this theme are listed in Table 5.4.

Table 5.4: Relevant codes for Theme 2

Relevant Codes	**Participant Frequency**
Domain Knowledge	6
Underlying concept was hard to grasp	7
Confusion with scale operator	8

Although eight out of twelve participants were able to complete the tree layout generation task, most of them faced difficulty understanding the underlying concept of the tree layout generation process. Most of the participants were unfamiliar with different tree layouts and had no prior background working with a tree visualization API. As a consequence, the main concept of the tree layout generation was hard for them to grasp.

Also, some operators of AVIT were very specific to certain tree layouts and without having the domain knowledge; it is difficult to understand which operator has to be

selected to implement them. For example, for drawing a Squarified nested tree-map layout as specified in Task 2, one needs to allocate the space between nodes so that they resemble a square. To do that, the *allocate (SQUARIFY)* operator has to be called in the ALLOCATION stage. It is also necessary to order the nodes in ascending/descending order for the Squarified tree-map algorithm to work properly. So, if a developer had never used a tree-map layout before, it would be very difficult for him to understand which parameter he should use to get the job done. As P11 mentioned *"First of all the squarify operator seems a bit magical to me. I was not sure what it really did, I just knew that I should use it for this task, so I used it."*

Participants suggested that providing detailed descriptions of different tree layout algorithms and architectural details of the API in the documentation and spending more time with the API might help them to understand the API better.

Theme 3: *Examples, without having the detailed description of the usage scenario, were hard to follow by the participants.*

Relevant codes supporting this theme are listed in Table 5.5.

Table 5.5: Relevant codes for Theme 3

Relevant Codes	**Participant Frequency**
Provide a story or connection with the examples	6
Demo video explaining different operators	2
Detailed description of different stages	3
Detailed visual example of scale operator	4
Provide complete example	3

Most of the examples provided in the API documentation were just concise code fragments with the corresponding output figure. Some participants found the code examples difficult to follow. They suggested following a storytelling approach describing the motivation behind the example code to make it more understandable. As P3 said, on a question regarding how to improve the API documentation, *"provide a better connection regarding what you trying to accomplish and how you will accomplish that. Try to have a user story with the examples."*

Participants also demanded addition of a section in the documentation describing the architectural details of the API. A complete, step-by-step example explaining the connection between different layers of the API will help them to understand the main design concept of the API.

As P12 mentioned *"You should provide me some documentation that describes the architecture of your API and how all these different layers are working together to generate a tree layout. Because if I understand why you put those things I can understand how to use this layer and where to put the relevant operator much better. If you can give more philosophical details before the technical details in your API that will help me understand the API better."*

Theme 4: *Examples should cover as many different usage scenarios as possible.* Relevant codes supporting this theme are listed in Table 5.6.

Table 5.6: Relevant codes for Theme 4

Relevant Codes	**Participant Frequency**
Provide more examples covering different parameter values	2
Provide link to the interactive demo in the operator documentation	5
Provide documentation describing different tree layouts	3

Although some examples were added in the revised documentation showing the effect of placing an operator in different stages of the tree layout pipeline − participants still asked for more examples showing the effect on the output layout for using different parameter values in an operator. Some participants suggested that providing a relevant link to the interactive demo for every operator definition would be very helpful as they can try out coding directly in the demo website and observe the effect on output right away.

Theme 5: *Confusion due to inconsistency in method definition and parameter ordering.*

Relevant codes supporting these themes are listed in Table 5.7.

Table 5.7: Relevant codes for Theme 5

Relevant Codes	**Participant Frequency**
Use consistent function definition	8
Confusion with lasso interaction	7
Inconsistency in parameter definition	4

It has been observed from the study that some inconsistency in method name and parameter ordering creates confusion among the participants. For example, method definition for *setStrokeWidth* and *setStrokeColor* has a scope parameter where the value of scope could be either NODES or EDGES, while the *fill* method has no scope parameter.

setStrokeWidth(scope, width, condition)

setStrokeColor(scope, color, condition)

fill (color, condition)

While working with the fill method, some participants were also expecting a scope parameter for the *fill* method and were confused when they later found in the documentation that there is no scope parameter for *fill*. Participants also suggested renaming the fill method to *setFillColor* to remain consistent with the other methods name. It is also considered a good API design practice to have consistent name and parameter ordering across methods [18].

Participants were also confused by the overloaded *addEvent* method. The purpose of *addEvent* was to add different interaction features to the visualization. For example, for displaying menu options, one needs to call *addEvent("click","showMenu")* and for displaying the node attribute on mouse over, one needs to call *addEvent("mouseover", "showAttribute").* Most participants suggested that it is better to give different method names to these different interaction technique like *addMenu("click")* or *showNodeAttribute("mouseover")* than using only the *addEvent* method to add them.

As P9 mentioned *"If you have a specific function for every separate interaction instead of just addEvent that will be more understandable for me. Like for lasso you can have a addLassoEvent() function and in the parameter you can provide how you want your lasso to be like."*

It is also a recommended practice by Bloch, an API usability researcher [18], to have different names to methods rather than overloading them if their behavior is significantly different.

Theme 6*:* *Examples provided in the live demo were a good starting point for the tree layout task.*

While working on Task 2: Generating a Squarified tree-map layout, some participants were looking for a similar example in the interactive demo website. While browsing the examples in the demo website, the example tree-map layout caught their attention, which they thought was relevant to Task 2. They started modifying the code in the demo website to see the effect on the output and to have an understanding of the process. P5, P7 and P10 were able to generate the Squarified tree-map layout by making changes in the example tree layout from the website. Examples provided in the documentation were also helpful for understanding the API, as P12 mentioned:

"The examples also give me some intuition regarding your API about how a developer should use the API. From initial tutorial lot of things were not clear to me but those examples give me more feedback to understand how it is actually working."

Theme 7: *Operator* m*isplacement error messages and stage recommendation in the documentation helps in recovering from the error.*

Relevant codes supporting this theme are listed in Table 5.8.

Table 5.8: Relevant codes for Theme 7

Relevant Codes	**Participant Frequency**
Error message was helpful	4
Stage recommendation	6

It has been observed that compared to the first usability study there was far less confusion about the operator misplacement in the second usability study. Participants who faced such errors were always able to fix them by following the suggestion in the error message or using the recommended stage information in the documentation. However, it was also noticed by the author from the observation made during study that four out of the twelve participants were just closing the error message box without reading the message and thus failed to use the suggestion provided in the message to fix the error. All of those four participants were able to fix that error later by looking into the recommended stage information in the documentation. This observation suggests that it is good to have multiple sources of help, so that, if a user misses one source of help, somehow s/he can use help from some alternate source.

Theme 8: *Lack of explicit error messages and not having IDE support was frustrating.*

Relevant codes supporting this theme are listed in Table 5.9.

Table 5.9: Relevant codes for Theme 8

Relevant Codes	Participant Frequency
Provide IDE support	5
Lack of an auto complete feature	4
No error message for mistype	5

Most developers are used to working in integrated development environments (IDEs) and expect to have code completion and IntelliSense [62] support. Lacking this feature requires more effort from the developer to memorize method names along with their available parameters. As all participants were new to the API, they needed to look back and forth between the documentation and their code for the appropriate methods and their parameters. This was very time consuming and a frustrating experience for them as they expected to have this kind of support provided by an IDE.

As P12 mentioned *"I also miss the IDE support very much. … While doing the coding a very simple mistype leads to error and I have no idea what is going wrong without carefully looking into the documentation. There is no feedback. Lacking this kind of IDE is a kill for the developer."*

P7 said "*Actually one time I ended up putting a parameter that I should not have. I got tripped on that."*

Also, in the API under study, there was no explicit error message shown for mistyping a method name or parameter value. Although all the participants were able to fix their error

by consulting the provided documentation, it was very time consuming and sometimes they got lost without being able to find the root cause of the error.

As P1 expressed his frustration while facing a parameter mistype error *"Don't see any difference.. don't know why.. why it does not work..OK,, quotes gosh. And I don't understand why I need to use quote here. It is getting strange."*

The next two themes describe the user experience of AVIT regarding interaction features.

Theme 9: *The cognitive effort in finding the resulting node after performing an interaction was high.*

When a participant performed a search interaction or a node selection operation the selected nodes were highlighted (see Figure 3.16). It was observed from the study that participants faced difficulties to find the highlighted node as a result of the interaction. Suggestions received from the participants for improvements in this included making the selected node bigger or providing animation support to make it clearly evident to reduce the cognitive burden on the user to identify the selected nodes.

Theme 10: *Expectations due to familiarity with similar interaction features.*

Relevant codes supporting this theme are listed in Table 5.10.

Table 5.10: Relevant codes for Theme 10

Relevant Codes	**Participant Frequency**
Use familiar interaction for multiple node selection	3

In the API, for selecting multiple nodes, the developer had the option to draw any circular shape in the visualization area and any node that will lie inside the circular area will be selected. It was observed from the study that participants preferred a drag option to draw an automatic rectangle for them and expected that any node that will lie inside the bounding box of the rectangle will get selected. When asked about the reason behind this choice, participants expressed their familiarity with similar rectangular-based interaction for node selection in other applications.

5.5 Limitations of the Second Evaluation

Findings presented from the second usability study are based on systematic observation of programmers working with AVIT in a laboratory environment. Given this setting, there are factors that limit the generalizability of the observations.

The ease of use and the challenges faced by the participants are related to a certain extent to the tasks and experience of the participants. Some of the findings from the study also have been observed in other API usability studies in different settings [3, 6, 9,21 and 24]. However, given the exploratory nature of this study for finding the effect on the usability experience of the API for the changes made, and also given the lab setting and pre-defined tasks, the findings cannot be considered complete, but only a starting point.

Also, the same researcher designed the API documentation, supporting material and the usability study; there is a possibility of some bias. To address this issue the author has designed the tasks for the study in a way so that there is no single direct solution to the task available, in the provided documentation, and it needed some exploration by the participant to complete the task.

Participants were given a small and fixed amount of time. This makes the study a bit unrealistic considering the real work environment of a developer. Multiple studies show that program development time is different on the order of 10 to 1 for different developers [23]. As in a real world scenario, some developers might have more time to spend in the task and thus might have a different experience with the API. To compensate for this limitation, the data analysis has been done not focusing on the task completion rate but on the overall usability experience of the developer with the API.

To use the full power of the API in generating and customizing diverse tree layouts, one needs to have a clear understanding of the tree layout generation process used in the API. Participants had no previous experience with the API and most of them had no experience working with tree visualizations. This is a common scenario for programmers in industry and is convenient for conducting controlled studies. However, it also means that findings from the study provide limited insights in to the behavior of the developer who has years of experience working with the tree visualization tools.

With only a 15 minute tutorial, two real tasks and 12 participants, the questions and the challenges observed in the study are limited. Furthermore, given the observation that

“programmers often approach larger programming tasks by focusing on smaller subtasks” [26], it can be said that the challenges and usability problems observed should be seen as a good starting point for further explorations.

5.6 Discussion

The second usability study showed that the updated version of the API with the support of the interactive demo tutorial and updated documentation has slightly better learnability compared to the previous version. The updates in the API also helped recovering from operator placement errors by showing explicit error messages.

It was also evident from the study that developers still had some difficulties grasping the main concept of the tree layout generation process. Similar observations were made in the first study. Results of both usability studies suggest that without having domain knowledge about different tree layouts or without spending longer time with the API, understandability of the tree layout process will remain difficult.

Empirical studies have also shown that tree layouts, like tree-map and Sunburst, require training before users can use them effectively [11]. It is quite difficult to grasp the tree layout generation process in the limited time period of the study without having a background in this area.

While examples in the documentation were helpful and provided a good starting point for task completion, they still need to be improved with detailed use case descriptions and, preferably, with more usage scenarios. Documentation and the associated example code should be updated iteratively based on the developer feedback.

Although it was supposed to be easy to add different interaction features via a simple function call, participants suggested using different method names for different interaction features rather than overloading them with a single *addEvent* function. The author will make necessary changes in the interaction layer to incorporate those changes in a future version of the API.

The study also suggested to explicitly highlight the outcome of an interaction so that it requires less effort from participants to identify the results of an interaction. Participants also provided valuable suggestions regarding new interaction features such as showing path between nodes, animation − will be added in a future version of the API by the author.

In comparison with the first usability study, in the second study, some usability issues from the first study have been minimized, some issues remained the same and additional usability issues have been found. The main concern about the API is the difficulty in understanding the underlying tree generation concept. The author feels that the API will require some time from its user to understand different tree layouts and their generation process. Conducting longitudinal studies with developers who will be spending more time with the API for building real world application can add some light from that perspective.

Also, operator-based tree drawing introduces a new notational syntax for specifying tree layouts. Findings from the usability study showed that participants still had difficulties in understanding the recursive nature of the layout pipeline and the effect on the output

layout for placing operator in different order. To understand the cognitive load of AVIT notational syntax on the developers, the author has decided to conduct an evaluation of AVIT syntax using the Cognitive Dimension of Notation (CDN) framework proposed by Green et al. [43]. The Cognitive Dimension of Notation (CDN) framework are design principles for notations, user interfaces and programming language design and are used to evaluate the usability of an existing information artifact. The author expects that an evaluation based on the CDN framework will provide more insight regarding the learnability and the understandability of AVIT syntax and will help understand the findings from the usability studies. The evaluation will be described in Chapter Six.

Chapter Six: Evaluation of AVIT using the Cognitive Dimension of Notation Framework

The Cognitive Dimensions of Notation (CDN) framework proposed by Green et al. [43] is an inspection method for evaluating the effectiveness of notational systems such as programming languages and visual interfaces. CDN provides a collection of *cognitive dimensions*: useful heuristics for evaluating a notation system and the environment in which it is manipulated [43].

In AVIT, operator-based tree drawing introduces a new notational syntax for specifying tree layouts. To understand the cognitive load of this notation on the developers, the author has decided to conduct an evaluation of AVIT notational syntax using CDN framework [43]. Inspired by work done by Clarke et al. [52, 53, 54], the author expected that an evaluation based on the CDN framework would provide more insight regarding the learnability and the understandability of AVIT and would help describe the findings from the usability studies.

In the following subsections, the operator-based notation system of AVIT has been evaluated using the dimensions of CDN framework.

6.1 Evaluating AVIT using CDN Framework

The following sections provide succinct descriptions of different dimensions of CDN framework (see Green et al. [43] for more detailed description of each dimension). The operator-based notations of the tree layout specification used in AVIT are evaluated using

the dimensions described in [43]. Relevant findings from usability studies are described in terms of dimensions from CDN framework where appropriate.

6.1.1 Abstraction Gradient

"What are the minimum and maximum levels of abstraction exposed by the notation? Can details be encapsulated?"[43]

All operators implemented in AVIT for specifying tree layouts are highly abstracted and hide the details of the complex mathematics necessary to actually create these layouts from developers. These abstractions were made to make it easier for a developer to use those operators without worrying about their inner workings. However, as mentioned in [45], having to master several abstractions all at once in the limited time period of the usability study might be difficult for the developers and can affect the learnability of the API.

Findings from usability studies also confirm the benefits and drawbacks of abstraction provided by operators. From Theme 1 in Section 4.3, participants liked the abstraction provided by the operator-based approach because it allowed them to focus on being able to generate the tree layout without worrying about complex mathematical details. However, it was also evident from the findings that participants found it difficult to master the overall process of tree layout generation and had difficulties understanding some operators as described in Theme 2 and Theme 7 in Section 4.3.

6.1.2 Closeness of Mapping

"How closely does the notation represent the problem domain?"[43]

In AVIT, the notation and problem domain are loosely linked: the mapping requires conceptualizing the visualization in terms of operators, which in turn have a number of effects (and side-effects) on visual properties.

To increase the closeness of mapping in AVIT, operators have been named according to their function. For example, the *rotate* operator rotates a shape by a specified degree. The operator *reshape* updates the shape of node/s as specified in the parameter.

However, it has been observed from the usability studies that participants had difficulties selecting the right operator for a task' possibly due to the poor closeness of mapping from the task to the operator needed to complete it. For example, in the second usability study, six out of the twelve participants had difficulties creating the nesting effect for Task 2: "drawing a nested Squarified tree-map layout". The nesting step of Task 2 required a scale operator to create the nesting effect. Part of the problem faced by the participants might lie in the poor closeness of mapping from the nesting task to the operator (*scale*) to accomplish that.

6.1.3 Consistency

"When a part of the notation has been learned, how much of the rest can be inferred?" [43]

Most operators in AVIT are idiosyncratic but composed in a consistent fashion. The input as well as the output of all operators are the aforementioned tuples, they can be called in an arbitrary order, left out completely (identity operator), or even be called multiple times in a row with no conceptual restriction.

6.1.4 Diffuseness/Terseness

"How many symbols or how much space does the notation require to produce a certain result or express a meaning?" [43]

The number of operators needed to generate a tree layout using AVIT is small and each operator with its few parameters can be considered terse. Also the entire code to generate a tree layout using AVIT is concise.

As observed from Theme 1 of the first usability study, participants liked the concise specification for drawing trees using AVIT. This property also helped their learning of AVIT via trial and error, as making small changes in the operator usually shows visible changes in the output layout (for details see Theme 3 in Section 4.3).

6.1.5 Error-proneness

"Does the design of the notation influence the likelihood of the user making a mistake?" [43]

As observed from the usability studies there is one notation of the operator parameters in AVIT that encourages users to make mistakes based on their familiarity with object-oriented programming languages.

In AVIT, a condition parameter *c* of an operator was required to be put within quotes. For example, if developer wants to change the shape of the nodes in level 2 of the tree to rectangular shape, they need to use the *reshape* operator in the POSTLAYOUT stage as `reshape(RECTANGLE,"node.level==2");`.

It was observed in the studies, five out of twelve participants did not use quotes around the conditional parameter. They thought, as *node* is an object, it does not require putting quotes for accessing its property.

6.1.6 Hard Mental Operations (HMO)

"How much hard mental processing lies at the notational level? Are there places where the user needs to resort to fingers or penciled annotation to keep track of what's happening?" [43]

Each operator in AVIT is simple and is defined for a specific purpose. However, understanding proper operator sequence to draw a particular tree layout might increase the HMO. Also some operators like *squarify* , *slice, strip* require domain knowledge to understand their functionality.

This high level for HMO has been observed in both usability studies where participants had difficulties understanding the underlying tree layout generation process in AVIT (see Theme 2 in Section 4.3 and Theme 2 in Section 5.4).

6.1.7 Hidden Dependency

"Are dependencies between entities in the notation visible or hidden?" [43]

Operators in AVIT have different effects on the output tree layout based on their placement in different stages of the layout pipeline. For example, placing a reshape operator in the INITIALIZE stage will affect the entire drawing area of the tree, while

placing the reshape operator in the POSTLAYOUT stage only affects the nodes in a particular level of the tree.

This dependency is not evident from the layout specification, which might give a false impression that all operators perform the same function irrespective of their placement in the layout pipeline.

Results from both usability studies shows that participants had difficulties understanding the effect on output for shuffling operators around different stages of the tree layout pipeline. This observation is described in Theme 5 in Section 4.3 and Theme 2 in Section 5.4.

6.1.8 Premature Commitment

"Do programmers have to make decisions before they have the information they need?" [43]

In AVIT, there is less premature commitment in the sense that each operator is independent and does not restrict the user to which operator has to be called first. It will show an output layout based on the current placement of the operator, although this might not always make sense as a useful tree layout.

However, to generate a particular type of tree layout, AVIT operators have to be placed in a particular order in the layout specification file and demand some premature commitment from the developer to understand proper ordering [55].

It has been observed from the usability studies (see Theme 5, Section 4.3) that participants had difficulties understanding the proper ordering of operators to generate a particular tree layout.

6.1.9 Progressive Evaluation

"Can a partially-complete program be executed to obtain feedback on "How I am doing"?" [43]

AVIT has excellent support for progressive evaluation. Programs can be executed any time and the program environment supports viewing the output based on partially completed code. It allows evaluating the problem-solving progress at frequent intervals. The programmer has the option of simply changing the code based on their understanding from progressive evaluations and AVIT executes the program again to view the updated output. This progressive evaluation support was helpful to learn the API via trial and error as observed from the usability studies.

6.1.10 Role-expressiveness

"Can the reader see how each component of a program relates to the whole?" [43]

In AVIT, the operators are typically identified by their name and constructor parameters. In many cases, documentation or code-inspection is required to understand the effects of executing the operator. This in turns make it difficult to understand the relevance of an operator to generate the complete tree layout.

This difficulty of understanding how each individual operator fits into the overall tree layout generation process has been evident in the conducted usability studies (see Theme 2 in Section 4.3 and Theme 2 in Section 5.4 for details).

6.1.11 Secondary Notation

"Can programmers use layout, color, or other cues to convey extra meaning, above and beyond the official semantics of the language?" [43]

In AVIT secondary notation is available through the comment syntax. As no editor/IDE support was provided for AVIT's operator-based notation, no other type of secondary notation like syntax highlighting was available.

It was observed from the usability studies that not having syntax highlighting and IDE support for AVIT was a frustrating experience for all the participants (see Theme 8 in Section 5.4).

6.1.12 Viscosity

"How much effort is required to perform a single change?" [43]

AVIT has low viscosity. In AVIT, with minimal effort, significant change in the output can be made. For example a classical node-link layout can be converted to a radial node-link layout just by adding the `reshape (CIRCLE)` operator in the INITIALIZE stage.

Participants from usability studies found this feature of doing more by writing less code very useful, as mentioned in Theme 1, Section 4.3.

6.1.13 Visibility

Local visibility of AVIT is quite good. The entire specification for a tree layout can be seen in the computer screen. Also, the six stages of the layout pipeline provided recommendations in grouping related operators and thus increases the readability of the layout specification.

6.2 Limitation of the CDN Analysis

Findings from the CDN framework analysis provided a better understanding of the results from the usability studies and helped to point out some of the root causes of the usability issues in AVIT. However, without conducting usability studies, many of those issues would have been difficult to identify using only the CDN framework. For example, from the CDN framework analysis, it was not so evident that participants will face operator misplacement errors. However, the usability studies show a lot of misplacement errors

Also, findings from the CDN analysis cannot be viewed as a list of usability problems. Dimensions described in the CDN framework are interrelated and fixing a problem in one dimension in CDN usually affects some other dimensions [43]. For example, increasing *abstraction* can cause the *closeness of mapping* dimension to be reduced and it can also worsen the *hidden dependencies* and *hard mental operations* dimensions if it is not chosen properly. Usability studies, on the other hand, provide facts about the specific usability problems that actual users of the API are facing. API designers can thus prioritize those problems and can take appropriate measures to address those problems either by making changes to the API code or updating the documentation. Also as the

CDN framework only evaluates notational systems, the effects of documentation and tutorial materials on the usability of an API can't be conclusively evaluated using CDN.

The CDN analysis of AVIT has been conducted solely by the author from his understanding of the dimensions and usability study results. While self-evaluation of AVIT provided valuable insight it might be less reliable as it is not verified by any other researchers.

6.3 Discussion

Findings from usability studies showed that developers had difficulties in understanding the tree layout generation process used in the AVIT. This observation has been explained in terms of the weakness of AVIT in *closeness of mapping, hard mental operations, hidden dependencies, role-expressiveness* dimensions in the CDN analysis. Also, some of the errors observed in the usability studies have been explained by the weakness of AVIT notations in the *error-proneness* dimension of the CDN framework.

In line with Jeffries et al. [44], the author of this thesis believes that it is always better to conduct multiple evaluations using different evaluation techniques to assess the usability of a tool. Evaluations conducted using usability studies and CDN gave valuable insight about the usability of AVIT and helped in describing the root cause for many usability issues. However, the main purpose of developing AVIT was to being able to generate and customize different tree layouts via a concise specification without worrying about the complexity of the tree layout algorithms. So, abstracting those mathematical complexities behind operators was necessary to address that goal. This abstraction – although it increased the mental operations and hidden dependencies, and decreased the closeness of

mapping – is liked by many developers who enjoyed the ability to do more by writing less code. Also, a high task completion rate from the usability study suggests that this approach provide enough flexibility to learn the system by having high *progressive evaluation* support and low *viscosity* and thus supports learning of the API via playful trial and error approach as mentioned in findings of second usability study.

As making changes arbitrarily across a dimension might worsen usability in some other dimensions, further investigations need to be done to find standard remedies which will provide ways of improving performance on selected dimensions that matters most to the actual users of AVIT.

Chapter Seven: Conclusion

This thesis presents findings from usability evaluations of an API for visualizing and interacting with tree layouts. First, an overview of the challenges involved in drawing and interacting with trees using current toolkits was presented to provide the background necessary to understand the challenges of this field. An API, AVIT was created to make flexible customization and task-specific interaction with tree visualizations possible. The structure of AVIT was explained, and the design of the system and its concrete implementation were discussed. Two usability evaluations were conducted to point to potential answers for the research questions, described in Section 1.3, and to give an insight into the strengths and weaknesses of AVIT. Ways to improve the usability of AVIT were also explored.

7.1 Thesis Contributions

The first contribution of this thesis is exploring the usability of AVIT by conducting two separate usability evaluations. Usability evaluations were conducted to find answer to the research questions in Section 1.3.

The first three research questions, related to generating different tree layouts, flexible customization support, and concise layout specifications have been answered by building example tree layouts using AVIT, as explained in Section 3.3.2. It has been seen that, using AVIT different tree layouts can be constructed with concise specifications. Also, a customized tree layout can be generated using existing components of the API rather than

implementing from scratch in a short period of time. Generating hybrid and novel tree layouts is also possible.

The research question, about *"How much effort is needed from the developers to learn the operator-based approach of generating tree layouts?"* was difficult to answer considering the limited time period of the usability studies. But the findings from the usability studies suggest that AVIT has a steep learning curve. Conducting longitudinal studies with developers using AVIT to build real world applications will help finding a better answer of this research questions.

The next research question was *"How helpful are the documentation and other learning materials for completing a task?"* In general, documentation and learning materials were helpful to participants in completing tasks but were not sufficient. Suggestions have been received to enhance the documentation. Also, it was observed that the improved documentation for the second usability study provides better performance in terms of error reduction and improved the learning effort via an interactive demo tutorial and detailed description of each operator in the wiki documentation.

The research question , *"How to improve the interaction features of the API?"* has been answered by gathering suggestions regarding improving the way to add interaction features in the API and collecting new interaction features request as explained in Section 5.4 .

The next research question regarding *"How can the usability experience of the operator-based Tree API be improved?"* is explored by running multiple user studies and by evaluating the API using the CDN framework. Results from the usability studies and the

CDN framework analysis provide suggestions to improve the usability of the API and thus provide a starting point in answering this research question.

Answers to these research questions, fulfills the second research goal as listed in Chapter 1.

The second contribution is the AVIT − API for Visualizing and Interacting with Trees. AVIT was co-developed by the author based on a new concept for tree drawing to address the limitation of existing toolkits. AVIT provides flexibility in tree layout customization via concise, operator-based syntax and has task-specific interactions support for trees. AVIT fulfills the first research goal listed in Chapter 1: "developing an API based on the operator based approach for tree drawing".

The third contribution of this thesis was the literature review covering the challenges of tree layout customization and task-specific interaction support in existing information visualization toolkits, which is presented in Chapter 1 and Chapter 2. This review provides the current state of tree visualization support in existing toolkits and also discusses the evaluation approach used for those toolkits by their designer.

7.2 Future Work

AVIT has shown promise in its ability to improve support for tree visualization and interaction. However, there is always room for improving the API and augmenting the research.

Firstly, the interaction layer in the API can be extended to take advantage of the operator sequences already implemented for drawing trees. For example, topology based

interactions like folding of a sub-tree rooted on a node can be accomplished by carrying out a "*reshape(NONE)*" operator for the selected node and all its descendants. A zooming interaction can be tied to the WEIGHT operator to enlarge sub-trees of interest, while at the same time automatically scaling down other parts of the layouts. This would make the operators useful even beyond the pure layout generation.

Secondly, longitudinal studies have to be conducted to gather a better understanding about the usability of the API. The API has recently been made publicly available for developers to use in their web-based applications. Developers can spend a longer time with the API while building a real world application. Results from such longitudinal usage based on the applications built by the developers can add additional insight into the evaluation findings. Also conducting a usability study with domain experts can add another dimension to the findings.

Thirdly, further investigation needs to be done to reduce the cognitive load of learning the API by finding an appropriate balance in the different dimensions of the CDN framework as discussed in Chapter 6.

Fourthly, to make the underlying concept of the tree layout generation process used in the API more understandable, documentation should be updated with video demos, scenario-based examples and interactive training tasks.

References

1. T. Boren and J. Ramey, Thinking aloud: reconciling theory and practice, *IEEE Transactions on Professional Communication, vol. 43, no. 3, pp. 261–278, 2000.*
2. T. Tullis and B. Albert, measuring the user experience: collecting, analyzing, and presenting usability metrics, *Morgan Kaufmann, 2008.*
3. Seyed Mehdi Nasehi and Frank Maurer: Unit Tests as API Usage Examples, *Proc. of 26th IEEE International Conference on Software Maintenance (ICSM 2010).*
4. Ko, A. J. and Riche, Y. (2011). The Role of Conceptual Knowledge in API Usability. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*
5. Schulz, H.-J.; "Treevis.net: A Tree Visualization Reference," *Computer Graphics and Applications, IEEE , vol.31, no.6, pp.11-15, Nov.-Dec. 2011* doi:10.1109/MCG.2011.103
6. Hans-Jörg Schulz, Zabedul Akbar, and Frank Maurer. A Generative Layout Approach for Rooted Tree Drawings. Accepted *In Proceedings of the IEEE Pacific Visualization, Sydney, Australia, February, 2013*
7. Michael Bostock, Jeffrey Heer, Protovis: A Graphical Toolkit for Visualization, *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis), 2009*
8. Huerta-Cepas, J., Dopazo, J., Gabaldon, T.: ETE: A Python environment for tree exploration. *BMC Bioinformatics11 (24) (2010)*

9. Heer, J., Card, S., Landay, J.: prefuse: A toolkit for interactive information visualization. *In: Proc. of CHI'05, ACM (2005) 421-430*
10. Slingsby, A., Dykes, J., Wood, J.: Configuring hierarchical layouts to address research questions. *IEEE TVCG 15(6) (2009) 977-984*
11. Jean-Daniel Fekete. 2004. The InfoVis Toolkit. *In Proceedings of the IEEE Symposium on Information Visualization (INFOVIS '04). IEEE Computer Society, Washington, DC, USA, 167-174.* DOI=10.1109/INFOVIS.2004.64
12. Nicolas Garcia Belmonte. JavaScript InfoVis Toolkit: Create Web Standards based interactive data visualizations. Web Link: http://philogb.github.com/infovis
13. Michael Bostock, Vadim Ogievetsky and Jeffrey Heer. D3: Data-Driven Documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis), 2011*.
14. Word tree. (2012, June). [Online] http://hint.fm/projects/wordtree/
15. Richard E. Boyatzis, "Transforming Qualitative Information – Thematic Analysis and Code Development"
16. Jonathan Sillito and Brian de Alwis, Saturate: A Collaborative Memoing Tool. *In Proceedings of UBC's First Annual Workshop on Qualitative Research in Software Engineering, 2009*. Web link: www.saturateapp.com
17. Braun and V. Clarke "Using thematic analysis in psychology". *Qualitative Research in Psychology 3 (2): 77-101, 2006*. doi: 10.1191/1478088706qp063oa.
18. Joshua Bloch. 2006. How to design a good API and why it matters. *In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications (OOPSLA '06). ACM, New York, NY, USA, 506-507.* DOI=10.1145/1176617.1176622

19. Jakob Nielsen and Rolf Molich. 1990. Heuristic evaluation of user interfaces. *In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '90), Jane Carrasco Chew and John Whiteside (Eds.). ACM, New York, NY, USA*, 249-256. DOI=10.1145/97243.97281

20. Mark Keil, Peggy M. Beranek, and Benn R. Konsynski. 1995. Usefulness and ease of use: field study evidence regarding task considerations. *Decis. Support Syst. 13, 1 (January 1995), 75-91*. DOI=10.1016/0167-9236(94)

21. Scott R. Klemmer, Jack Li, James Lin, and James A. Landay. 2004. Papier-Mache: toolkit support for tangible input. *In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04). ACM, New York, NY, USA, 399-406*. DOI=10.1145/985692.985743

22. A. Rusu. Tree drawing algorithms. In R. Tamassia, editor, *Handbook of Graph Drawing and Visualization*, chapter 5. CRC press, 2013.

23. S. McConnell, Code complete: A practical handbook of software construction, *2nd ed., Microsoft Press, June 2004.*

24. Seyed Mehdi Nasehi, Jonathan Sillito , Frank Maurer, and Chris Burns: What Makes a Good Code Example? A Study of Programming Q&A in StackOverflow. *Proceedings of 28th IEEE International Conference on Software Maintenance (ICSM 2012), Riva del Garda, Italy, 2012.*

25. Baumgartner, Jason, Börner, Katy, Deckard, Nathan J., Sheth, Nihar. (2003). An XML Toolkit for an Information Visualization Software Repository. *Poster Compendium, IEEE Information Visualization Conference,* pp. 72-73

26. Jeffrey Stylos and Brad A. Myers. 2008. The implications of method placement on API learnability. *In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering (SIGSOFT '08/FSE-16). ACM, New York, NY, USA*, 105-112. DOI=10.1145/1453101.1453117

27. Shengdong Zhao, Michael J. McGuffin, and Mark H. Chignell. 2005. Elastic Hierarchies: Combining Tree-maps and Node-Link Diagrams. *In Proceedings of the Proceedings of the 2005 IEEE Symposium on Information Visualization (INFOVIS '05). IEEE Computer Society, Washington, DC, USA*, 8-. DOI=10.1109/INFOVIS.2005.12

28. M. Reingold and J.S. Tilford, Tidier Drawing of Trees, *IEEE Transactions on Software Engineering, Vol. 7, No. 2, pp. 223-228, 1981*.

29. J. Q. Walker II, A Node-Positioning Algorithm for General Trees, *Software-Practice and Experience*, Vol. 20, No. 7, pp. 685-705, 1990.

30. P. Eades, Drawing free trees, *Bulletin of the Institute of Combinatorics and its Applications, Vol. 5, pp. 10-36, 1992*.

31. Brian Johnson and Ben Shneiderman. 1991. Tree-Maps: a space-filling approach to the visualization of hierarchical information structures. *In Proceedings of the 2nd conference on Visualization '91 (VIS '91), Gregory M. Nielson and Larry Rosenblum (Eds.). IEEE Computer Society Press, Los Alamitos, CA, USA*, 284-291.

32. Hao Lü and James Fogarty. 2008. Cascaded tree-maps: examining the visibility and stability of structure in tree-maps. *In Proceedings of graphics interface 2008*

(GI '08). Canadian Information Processing Society, Toronto, Ont., Canada, Canada, 259-266.

33. Wilkinson, L.: The Grammar of Graphics. 2nd edn. *Springer (2005)*
34. Andrea Adamoli and Matthias Hauswirth. 2010. Trevis: a context tree visualization and analysis framework and its use for classifying performance failure reports. *In Proceedings of the 5th international symposium on Software visualization (SOFTVIS '10). ACM, New York, NY, USA*, 73-82. DOI=10.1145/1879211.1879224
35. Gregory E. Jordan and William H. Piel. 2008. PhyloWidget. *Bioinformatics 24, 14 (July 2008), 1641-1642*. DOI=10.1093/bioinformatics/btn235
36. Bongshin Lee, Catherine Plaisant, Cynthia Sims Parr, Jean-Daniel Fekete, and Nathalie Henry. 2006. Task taxonomy for graph visualization. *In Proceedings of the 2006 AVI workshop on BEyond time and errors: novel evaluation methods for information visualization (BELIV '06). ACM, New York, NY, USA, 1-5*. DOI=10.1145/1168149.1168168
37. R. Spence, Information Visualization: Design for Interaction, 2nd ed: *Prentice Hall, 2007*.
38. Alan Dix and Geoffrey Ellis. 1998. Starting simple: adding value to static visualization through simple interaction. *In Proceedings of the working conference on Advanced visual interfaces (AVI '98)*,DOI=10.1145/948496.948514
39. G. W. Furnas. 1986. Generalized fisheye views. *In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '86*), DOI=10.1145/22627.22342

40. InfoVis 2003 Contest: Visualization and pair Wise Comparison of Trees, 2003. Web link: http://www.cs.umd.edu/hcil/iv03contest

41. Flare: Data Visualization for web. Flare is an ActionScript library for creating visualizations that run in the Adobe Flash Player, 2008. Web link: http://flare.prefuse.org/

42. Jack K. Beaton, Brad A. Myers, Jeffrey Stylos, Sae Young (Sophie) Jeong, and Yingyu (Clare) Xie. 2008. Usability evaluation for enterprise SOA APIs. *In Proceedings of the 2nd international workshop on Systems development in SOA environments (SDSOA '08). ACM, New York, NY, USA, 29-34*. DOI=10.1145/1370916.1370924

43. Green, T.R.G. & Pete, M. (1996) usability analysis of visual programming environments: a 'cognitive dimensions' framework. *J. Visual languages and Computing, 7, 131-174*.

44. Robin Jeffries and Heather Desurvire. 1992. Usability testing vs. heuristic evaluation: was there a contest? *SIGCHI Bull. 24, 4 (October 1992), 39-41*. DOI=10.1145/142167.142179

45. B. Shneiderman and C. Plaisant *Designing the user Interface*. Addision-Wesley Publisher, 2004.

46. Robin Jeffries, James R. Miller, Cathleen Wharton, and Kathy Uyeda. 1991. User interface evaluation in the real world: a comparison of four techniques. *In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '91), Scott P. Robertson, Gary M. Olson, and Judith S. Olson (Eds.). ACM, New York, NY, USA, 119-124*. DOI=10.1145/108844.108862

47. Chris Stolte, Diane Tang, and Pat Hanrahan. 2008. Polaris: a system for query, analysis, and visualization of multidimensional databases. Commun. *ACM 51, 11 (November 2008), 75-84*. DOI=10.1145/1400214.1400234

48. Hans-Jorg Schulz, Steffen Hadlak, and Heidrun Schumann. 2011. The Design Space of Implicit Hierarchy Visualization: A Survey. *IEEE Transactions on Visualization and Computer Graphics 17, 4 (April 2011), 393-411*. DOI=10.1109/TVCG.2010.79

49. Zaixian Xie, Zhenyu Guo, Matthew O. Ward, Elke A. Rundensteiner: Operator-centric design patterns for information visualization software. *VDA 2010: 75300*

50. "HiDE: Hierarchical Data Explorer", is software for visually exploring categorical data using hierarchical layouts. *Web link:http://gicentre.org/hide/*

51. Daniel A. Keim. 2002. Information Visualization and Visual Data Mining. *IEEE Transactions on Visualization and Computer Graphics 8, 1 (January 2002), 1-8*. DOI=10.1109/2945.981847

52. Clarke, S. & C.Becker (2003). Using the cognitive dimensions framework to measure the usability of a class library. *In Proceedings of the First Joint Conference of EASE & PPIG (PPIG 15)*

53. Clarke, S. (2001). Evaluating a new programming language. In G. Kadoda (Ed.) *Proceedings of the Thirteenth Annual Meeting of the Psychology of Programming Interest Group, 275-289.*

54. S. Clarke, Describing and measuring API usability with the cognitive dimensions. *Cognitive Dimensions of Notations 10th Anniversary Workshop.*

http://www.cl.cam.ac.uk/_afb21/CognitiveDimensions/workshop2005/Clarke_position_paper.pdf

55. Green, T.R.G. (2006). Aims, achievements, agenda - where CDs stand now. *Journal of Visual Languages and Computing, 17(4), 285-394.*
56. Ahn, J., Plaisant, C., Shneiderman, B. "A Task Taxonomy of Network Evolution Analysis."; under review *IEEE VGTC.*
57. Catherine Plaisant. 2004. The challenge of information visualization evaluation. *In Proceedings of the working conference on Advanced visual interfaces (AVI '04). ACM, New York, NY, USA, 109-116.* DOI=10.1145/989863.989880
58. Purvi Saraiya, Chris North, Vy Lam, and Karen A. Duca. 2006. An Insight-Based Longitudinal Study of Visual Analytics. *IEEE Transactions on Visualization and Computer Graphics 12, 6 (November 2006), 1511-1522.* DOI=10.1109/TVCG.2006.85
59. Ben Shneiderman and Catherine Plaisant. 2006. Strategies for evaluating information visualization tools: multi-dimensional in-depth long-term case studies. *In Proceedings of the 2006 AVI workshop on BEyond time and errors: novel evaluation methods for information visualization (BELIV '06). ACM, New York, NY, USA, 1-7.* DOI=10.1145/1168149.1168158
60. Z. Xie, M. O. Ward and E. A. Rundensteiner, "Operator-centric Design Patterns for Information Visualization Software", *Visualization and Data Analysis, Part of IS&T/SPIE Symposium on Electronic Imaging 2010.*

61. Software Design Patterns for Information Visualization, Jeffrey Heer, Maneesh Agrawala. *IEEE Transactions on Visualization and Computer Graphics (Proc. InfoVis'06), 12(5), pp. 853-860, Sep/Oct 2006*.

62. "Using IntelliSense", *Microsoft MSDN library.* Last access November, 2012. Web link: http://msdn.microsoft.com/en-us/library/hcw1s69b.aspx

Appendix A: Background Questionnaire

- How many years of experience do you have with any programming language?

- Do you have any experience/familiarity working with data visualization? If yes please mention the type of data visualization you have used so far.

- If applicable, provide the names of the data visualization tools you have used so far?

- Are you currently in academia or industry?

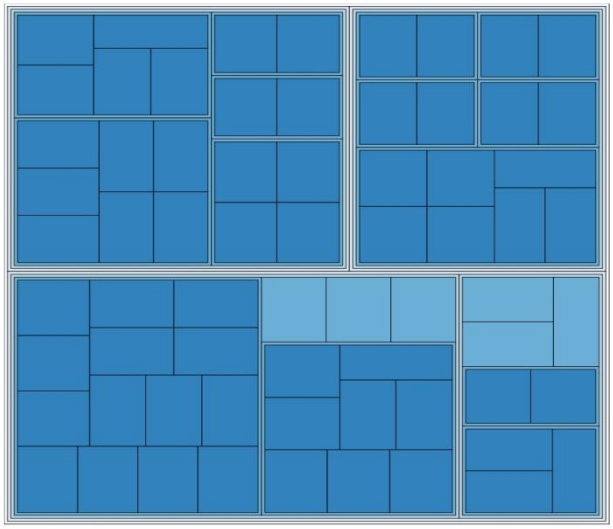
- What is your role in your current organization?

Appendix B: Task Description

B.1 Task Description for Study 1

Task		**Descriptions**
1	1.1	Order the nodes in **ascending order** of number of **leaves**.
	1.2	Make the **edges** between nodes 3px wide [Hint: use operator ***setStrokeWidth()*****]**
	1.3	Scale the nodes shape so that every node is 10px wide [Hint: use operator *scale ()*]
	1.4	Rotate the layout so that it looks like the given figure (Hint: Entire layout is affected).
	1.5	Rescale the node shape to 5px, then fills the node color Dark to light, flowing from root to subsequent levels [You can choose any color from palette [see documentation].] [Hint: Use node.level as a value and root.height as max value for color filling].
	1.6	Change the layout so that nodes are arranged in circular topology [Hint: Change the drawing space shape in ROOT_LAYOUT]
	1.7	Add Lasso selection Interaction with rectangle as a bounding area [Make necessary changes in addInteraction.js file]

	1.8	Make necessary changes in POSTLAYOUT stage, so that it shows sunburst layout [Given Figure].
	1.9	Transform the node shape to circular shape. [Given Figure].
	1.10	Add Menu based Interaction so that mouse clicking on a node display the menu option [Make necessary changes in addInteraction.js file]
2		Nested **Squarified** Tree-map is type of Tree-map where children nodes are embedded within parent nodes layout with offsetting between successive levels to produce the **nesting effect**. It allocates the drawing space between leaves node in a way so that the node resembles a square. Modify the config file for Task 2 to generate the following Nested Squarified Tree-map layout. [Given Figure]

B.2 Task Description for Study 2

Training Task

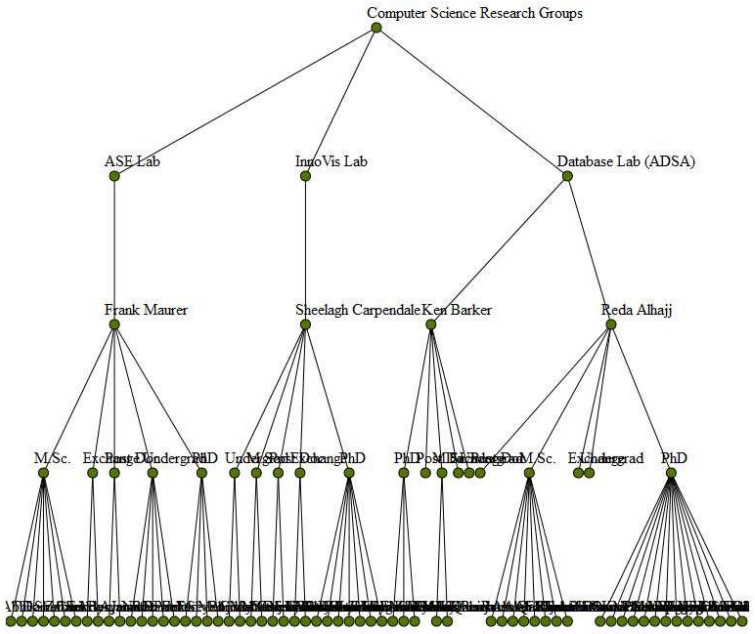
Task 1

In this task you have to make small changes in the **configuration file** for a given tree layout. The purpose of the task is to understand the **functionality** of different **layout operators** and how the tree layout changes based on their placement in different stages of the layout generation process.

See the example Tree layout in the browser for Task 1 (your starting point):

You have been provided with a sample configuration file that produces the following output.

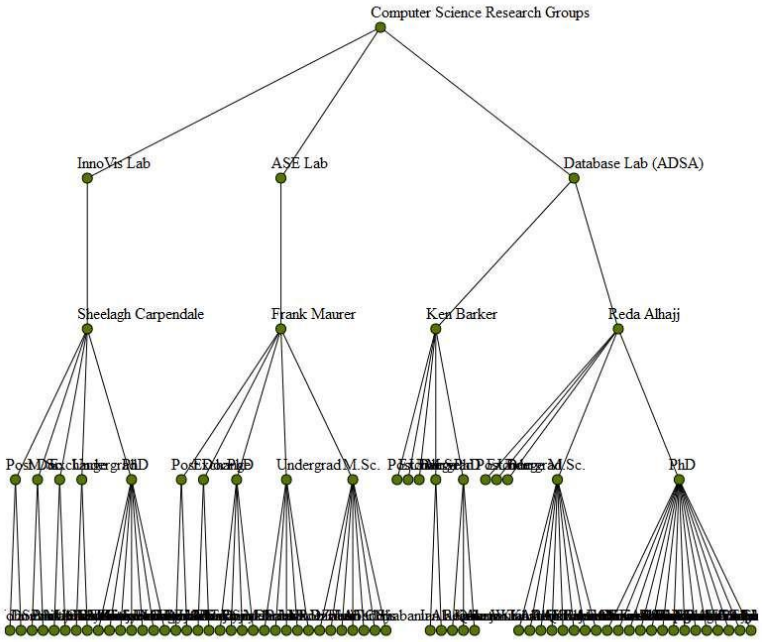
Dataset: Computer Science research group at University of Calgary.



Task 1.1

Order the nodes in **ascending order** of number of **leaves**.

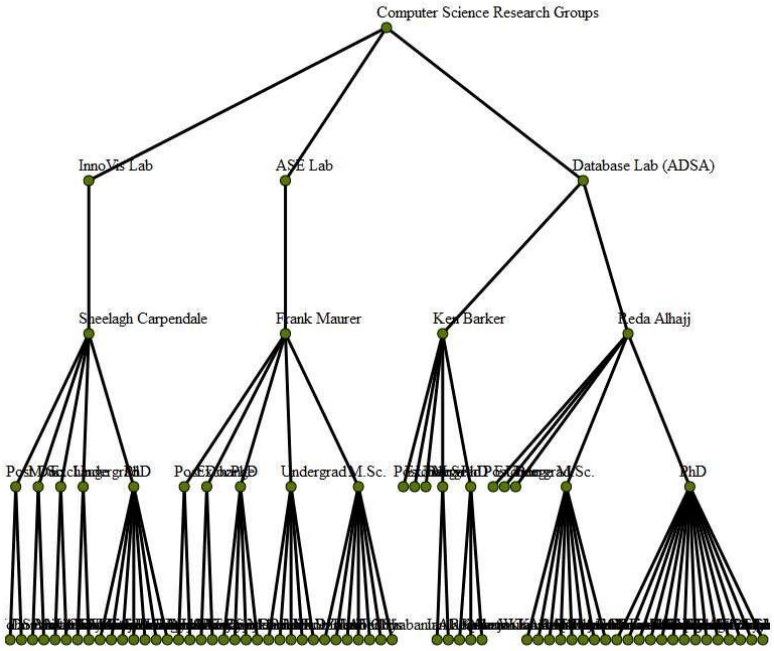
Expected Output Layout:



Task 1.2

Make the **edges** between nodes 3px wide [Hint: use operator ***setStrokeWidth()*****]**

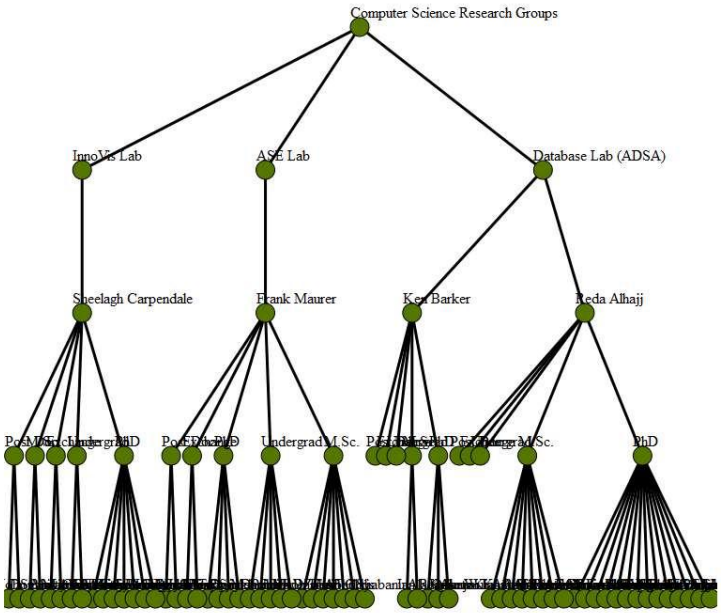
Expected Output Layout:



Task 1.3

Scale the nodes shape so that every node is 10px wider [Hint: use operator *scale ()*]

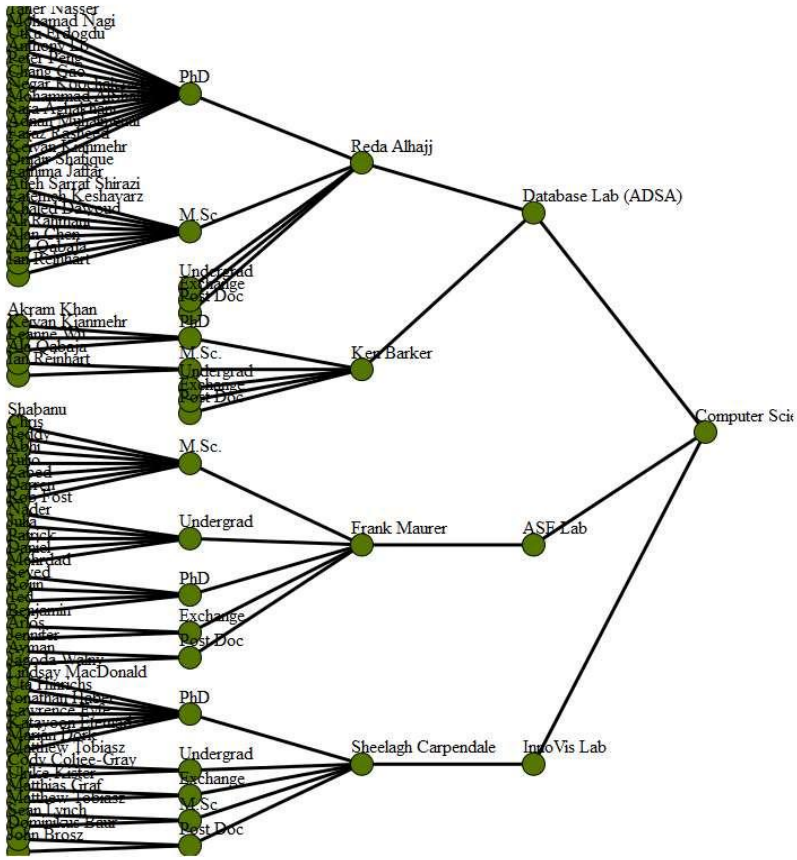
Expected Output Layout:



Task 1.4

Rotate the layout so that it looks like the following figure (Hint: Entire drawing area is affected).

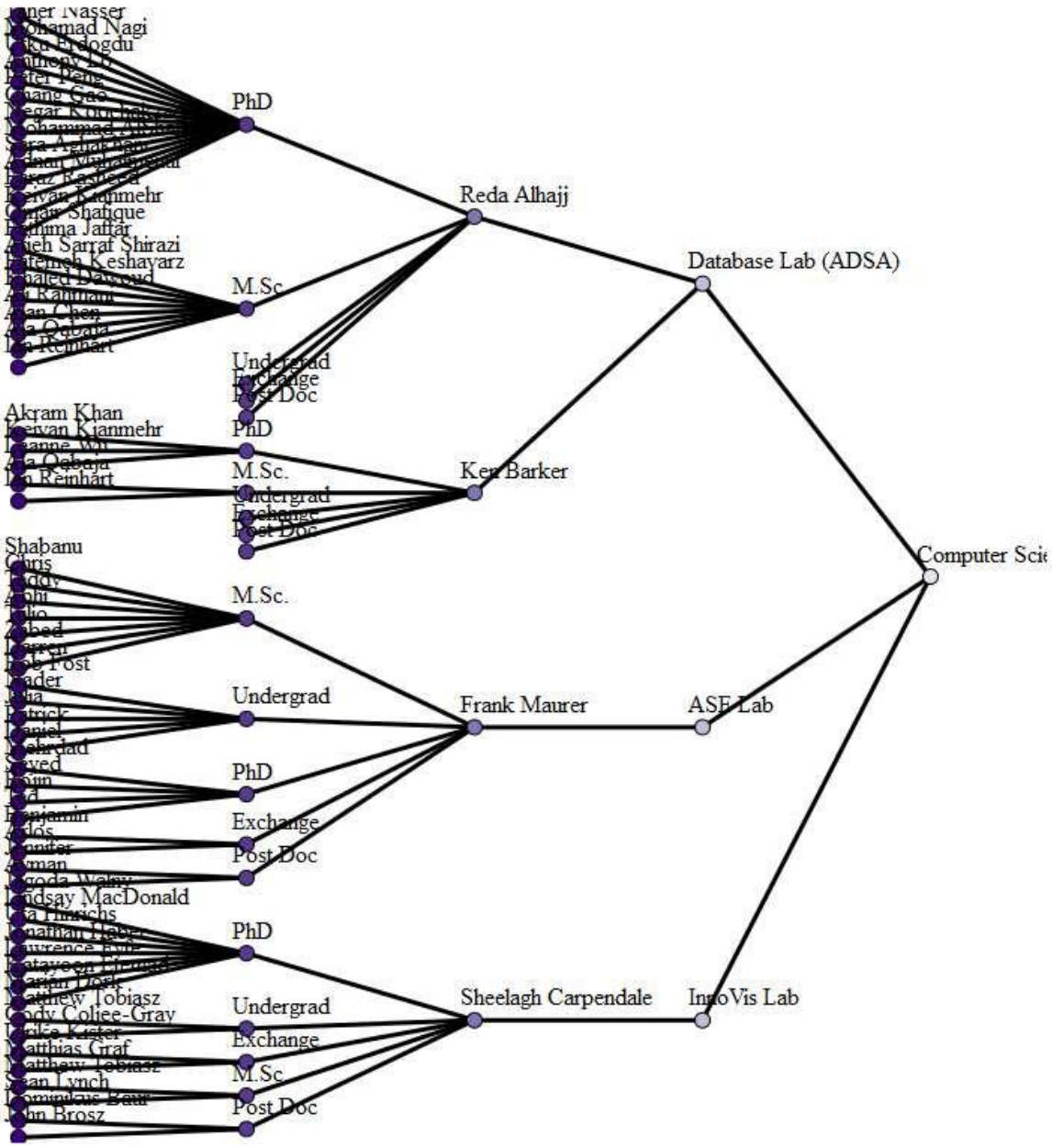
Expected Output Layout:



Task 1.5

Rescale the node shape back to 5px, then **fills** the node color Light to Dark, flowing from root to subsequent levels [You can choose any color from palette [see documentation]. [Hint: Use node.level as a value and root.height as max value for filling].

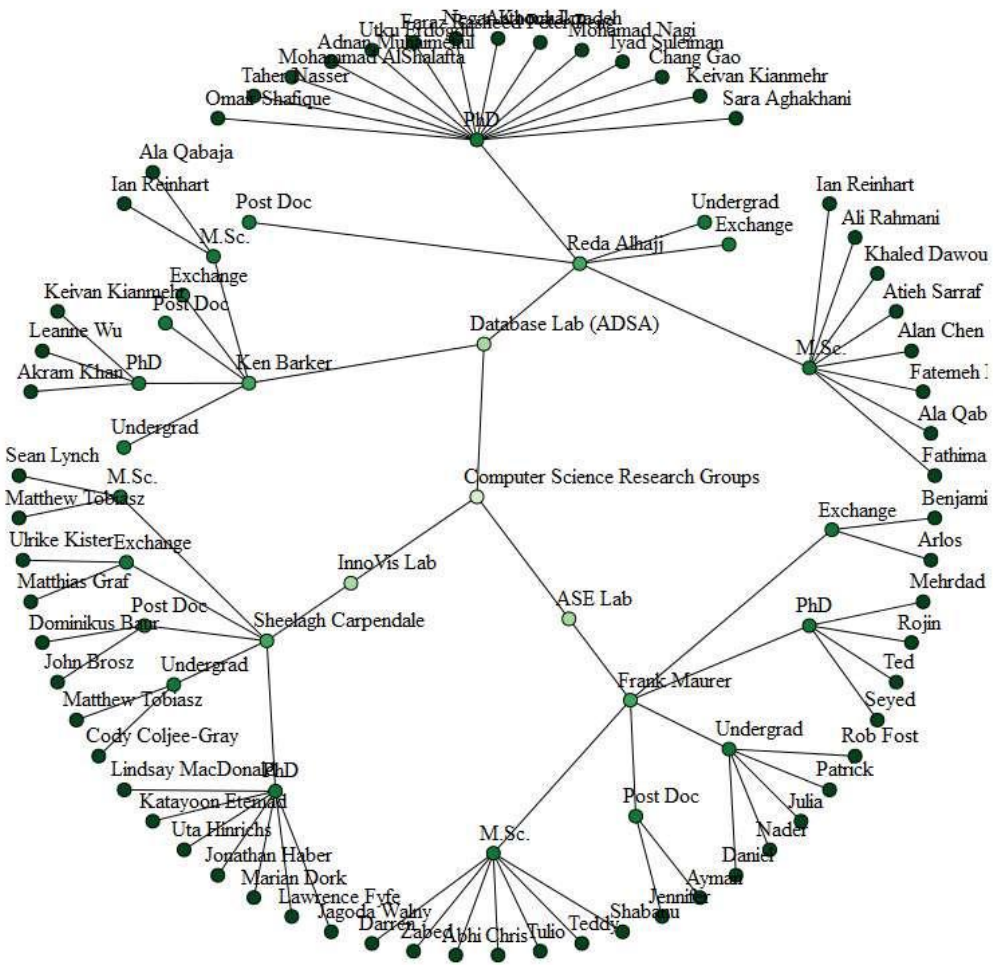
Expected Output Layout:



Task 1.6

Open the radial.cfg file and also click the Radial Node-Link in the browser. It will display radial tree layout (a different representation) of the same dataset.

Expected Output Layout:



Try the following change in the config file:

- Comment out the **reshape (CIRCLE)** in **INITIALIZE** stage and see what happens (click the Radial Node-Link to reload the layout).
  Expected Output:

  - It will generate the bottom up version of the layout similar to the layout you have seen in task 1.1.

Reasoning behind the Output:

  - Why bottom up? [If you remember by default drawing area is selected as rectangular region. As scaling in the **prelayout** has been done from bottom direction it generates the bottom up layout].
- Uncomment the **reshape(CIRCLE).**
- Comment out the **reshape (DOT)** in **POSTLAYOUT** stage and see what happen.
  - By default every node is assigned a drawing space. When **reshape(DOT)** is called in **post layout** it actually draw a dot on the middle of the assigned space of that node, hide the view of the space and thus generate the explicit layout. By commenting out the **reshape(DOT),** it will generate the default implicit layout known as sunburst layout.
- Comment out the **reshape (CIRCLE)** in **INITIALIZE** stage and see what happen (click the Radial Node-Link to reload the layout).
  - It will generate the bottom up tree (implicit layout) commonly known as icicle plot.

Task 1.7

Currently node values are being displayed as labels which are quite messy and hard to read for some of the nodes. Please make necessary changes in the **addInteraction.js** file so that node labels are not displayed. Then add the interaction code to show the node value only when there is a mouse over event is fired on that node.

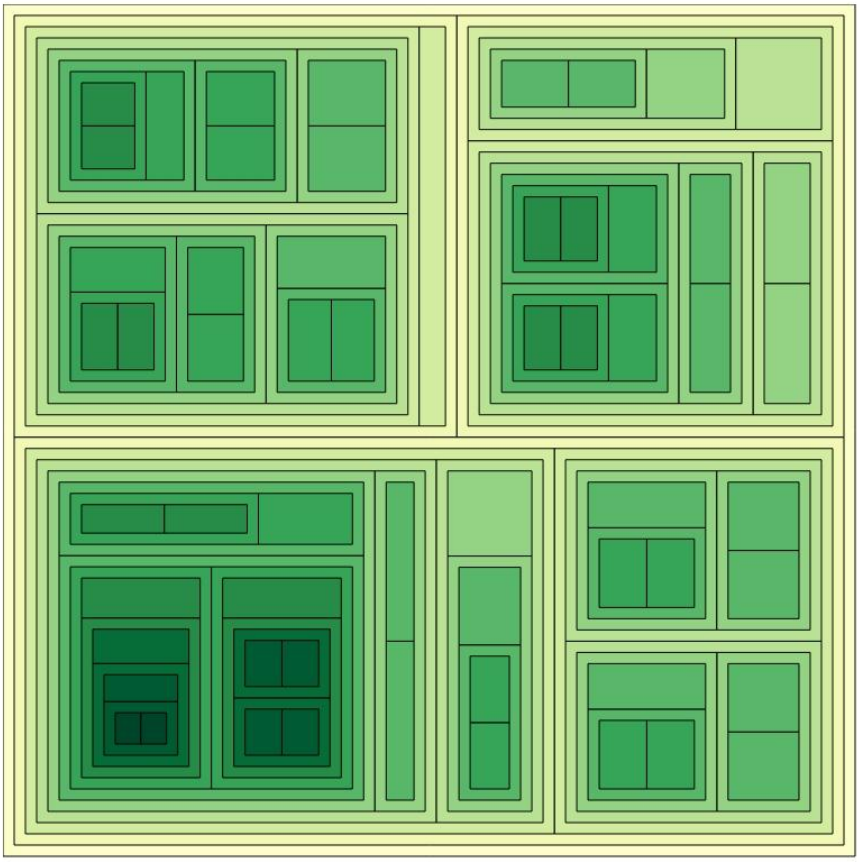
Layout Task

Task 2

Nested **Squarified** Tree-map is type of tree-map layout with offsetting/gap between successive levels that produce the **nesting effect** [Child nodes are embedded within parent nodes]. It **allocates** the space between nodes in a way so that it resembles a **square**.

Modify the config file for Task 2 to generate the following layout.

Expected Output Layout



Interaction Task

Task 3.1

Add Menu Interaction so that mouse clicking on a node displays the menu option [Make necessary changes in **addInteraction.js** file]

Test the Interaction: Click on the different menu option and see what happen.

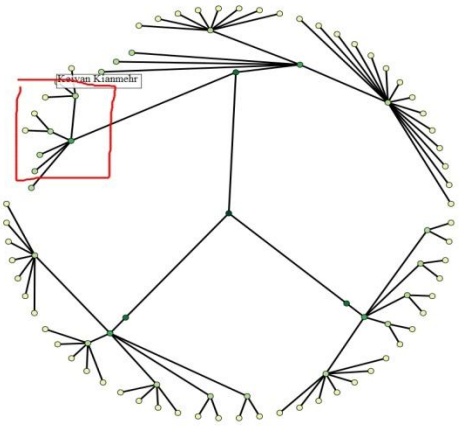
Comments:

Task 3.2

Add **Lasso selection** Interaction by selecting **rectangle** as a bounding area [Make necessary changes in **addInteraction.js** file]

Test the Interaction:

Drawing a rectangular region on the layout will highlight the node that lies inside the region.



Comments:

Task 3.3

[Search Interaction] Please type the name of a Node in the search box above the layout to search for that particular node (Case sensitive) (Example: PhD, ASE Lab, Frank Maurer).

Test: Matched node should be highlighted.

Comments:

Appendix C: Task Breakdown

Table: Breakdown of programming task and how their success level is measured

Task		Individual Steps	Completed	Partially Completed
1	1.1	1. Select the order operator. 2. Select appropriate parameter (ASCENDING and leaves) for the operator. 3. Placing the operator in appropriate stage (Stage: PREPROCESS).	Two successful sub-tasks including sub tasks 1 and 3 and a partially successful one (typo).	One successful sub-tasks including sub tasks 1 and a partially successful one.
	1.2	1. Select the **setStrokeWidth()** operator. 2. Choose appropriate parameter for the operator (3px). 3. Placing the operator in appropriate stage (POSTLAYOUT).	Two successful sub-tasks including step 1 and 3 and a partially successful one.	One successful sub-tasks including step 1 and a partially successful one.
	1.3	1. Select the scale operator. 2. Choose appropriate parameters value for the operator (SCALE_TO, 10px). 3. Placing the operator in appropriate stage (POSTLAYOUT).	Two successful sub-tasks including step 1 and 3 and a partially successful one.	One successful sub-tasks including step 1 and a partially successful one.
	1.4	1. Select the rotate operator. 2. Choose appropriate parameter value for the operator (90). 3. Placing the operator in appropriate stage. (ROOT_LAYOUT).	Two successful sub-tasks including step 1 and 3 and a partially successful one.	One successful sub-tasks including step 1 and a partially successful one.
	1.5	1. Select the scale operator. 2. Choose appropriate parameter value for the operator (5px). 3. Placing the operator in appropriate stage (POSTLAYOUT). 4. Select the fill operator. 5. Choose appropriate parameter value for the operator (“Color from platte”, DARK2LIGHT. “node.level”,	Successful sub-tasks including step 1, 3, 4, 6 and a partially successful one.	One successful sub-tasks including step 1, 4, 5 and a partially successful one.

		"root.height"). 6. Placing the operator in appropriate stage (POSTLAYOUT).		
	1.6	1. Select the reshape operator. 2. Choose appropriate parameter value for the operator (CIRCLE). 3. Placing the operator in appropriate stage. (ROOT_LAYOUT).	Two successful sub-tasks including step 1 and 3 and a partially successful one.	One successful sub-tasks including step 1 and a partially successful one.
	1.7	1. Call addLasso function for adding interaction. 2. Choose appropriate parameter value for the interaction function (rectangle).	Two successful sub-tasks.	One successful sub-tasks including step 1 and a partially successful one
	1.8	1.Commenting the reshape operator in POSTLAYOUT 2. Make changes in scale operator in POSTLAYOUT to produce the space filling effect.	Two successful sub-tasks.	One successful sub-tasks including step 1 and a partially successful one
	1.9	1. Select the reshape operator. 2. Choose appropriate parameter value for the operator. 3. Placing the operator in appropriate stage. (POSTLAYOUT). 4. Placing the operator in right order in the stage (right after the scale operator to produce the desired effect).	Three successful sub-tasks including step 1 and 3 and a partially successful one.	Two successful sub-tasks including step 1 and 3 and a partially successful one.
	1.10	1. Call appropriate function for adding interaction. 2. Choose appropriate parameter value for the interaction function.	Two successful sub-tasks.	One successful sub-tasks including step 1 and a partially successful one
2		(1) Select the *scale* operator with appropriate parameters for nesting effect. (2) Placing the *scale* operator in PRELAYOUT Stage. (3) Select *squarify* operator with appropriate parameters for allocation. (4) Placing the *squarify* operator in ALLOCATE stage. (5) Select *fill* operator with appropriate parameters for coloring effects. (6)Placing the *fill* operator in POSTLAYOUT stage.	Five successful sub-tasks including step 1, 2, 3, 4, 6 and a partially successful one.	Four successful sub-tasks including step 1, 2, 3, 4 and a partially successful one.

Appendix D: Post Study Questionnaire

- Were the sample documentation, tutorial and example code along with the live demo provided useful for performing the sample task?
- Was it easy to find the relevant documentation/help using the sample documentation and example code?
- What difficulties did you face while performing the sample task? And how were those difficulties overcome (how was the participant able to move on with the task)?
- What do you think about the operator based programming for manipulating the visualization?
- Do you find the naming and placement of the operator understandable?
- Did the operator name matched with your expectation about the functionality of the operator?
- Suggestions/comments regarding the interaction part?
- Was the documentation for the interaction part helpful?
- Suggestion to improve the interaction parts (parameter, more flexibility)
- What are the interactions you suggest for exploring tree visualization?
- Was the API easy or difficult to use? What was easy, what was difficult?
- Would you use the API in future for any visualization task? If not, why?
- Do you have any final suggestions for improving the API?